利用 data parallel 提供的 redundancy 来进行 recovery

Just-In-Time Checkpointing: Low Cost Error Recovery from Deep Learning Training Failures

Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev,
Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, Muthian Sivathanu
Microsoft Research

Abstract

Deep Learning training jobs process large amounts of training data using many GPU devices, often running for weeks or months. When hardware or software failures happen, these jobs need to restart, losing the memory state for the Deep Neural Network (DNN) model trained so far, unless checkpointing mechanisms are used to save training state periodically. However, for large models, periodic checkpointing incurs significant steady state overhead, and during recovery, a large number of GPUs need to redo work since the last checkpoint. This is especially problematic when failures are frequent for large DNN (such as Large Language Model) training jobs using many GPUs. In this paper, we present a novel approach of just-in-time checkpointing when failures happen, which enables recovery from failures with just a single minibatch iteration of work replayed by all GPUs. This reduces the cost of error recovery from several minutes to a few seconds per GPU, with nearly zero steady state overhead. This also avoids the guesswork of choosing a checkpointing frequency since failure rates usually have high variance. We discuss how just-in-time checkpointing can be enabled in training code, as well as design of key mechanisms for transparent just-in-time checkpointing without user code change. We analyze the wasted GPU work of just-in-time checkpointing and show that it is less than periodic checkpointing for large numbers of GPUs. We present results from our implementation in modern AI cluster infrastructure.

CCS Concepts: • **Computing methodologies** → **Neural networks**; **Distributed computing methodologies**; **Machine learning**; • **Computer systems organization** → **Reliability**.

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3650085

Keywords: Large Scale DNN Training Reliability, Systems for Machine Learning, Reliable Distributed Systems

ACM Reference Format:
Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev,, Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, Muthian Sivathanu, *Microsoft Research*. 2024. Just-In-Time Checkpointing: Low Cost Error Recovery from Deep Learning Training Failures. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3627703.3650085

1 Introduction

Large deep neural network models such as GPT-3 [13], PaLM [14] and GPT-4 [25] have recently become popular for their ability to generalize to many tasks in a zero-shot or few-shot manner. Training jobs for such large AI models process large amounts of training data using hundreds or thousands of GPU devices, often running for weeks or months. These GPU devices are organized in supercomputing clusters, and network interconnects such as NVLink and Infiniband are used to communicate between these GPU devices. Unfortunately, the GPU devices or network devices often encounter errors, including hardware errors which may be unrecoverable, and recoverable errors which may be of a transient nature (e.g. network congestion) or semi-permanent nature (e.g. corruption of driver state). Training jobs with thousands of GPUs may experience one or more errors daily [36].

The costs due to errors encountered by large multi-GPU training jobs are high because most training algorithms are synchronous, for which a single GPU error requires all GPUs to wait and redo work to recover from the error. Distributed training algorithms for large DNN models perform a large number of iterations over minibatches of the training data on multiple GPUs. Each minibatch consists of a few samples which are processed by the neural network using a forward pass, a backward pass, and an optimizer step such as Adam, which updates the model parameters, to be used in the next minibatch iteration. The time taken for processing each minibatch varies from a few hundred milliseconds to few seconds. Multi-GPU training jobs use data parallelism, in which the GPUs process different samples of data and compute the forward and backward passes in parallel, then find the global average of their locally computed gradients, using a synchronous collective operation such as all-reduce [10], before updating their parameters in the optimizer step at the

end of the minibatch iteration. Other parallelism approaches such as model/tensor parallelism or pipeline parallelism also require synchronization between GPUs at different stages of the processing of a minibatch. In case one GPU has hardware or transient errors, all other GPUs cannot proceed, and the entire training job must be restarted.

The state-of-the-art failure recovery solution today is for the user to implement periodic checkpointing support in their training scripts and design their jobs to be resumable from any prior checkpoint. Given that checkpointing of large models can be expensive, there have been a number of research proposals that seek to optimize the cost of checkpointing [2, 23, 35].

However, apart from the steady-state cost of periodic checkpointing, there is also a large amount of wasted GPU work when a failure happens since all GPUs restart from the last checkpoint and re-process many minibatches of data till the point of failure, hence on average, all GPUs repeat half the checkpoint interval of work for each failure. As we will see in § 5, these errors can cost up to few million dollars per month, depending on the size of the job. The overall job execution time also increases. Customer frustration due to failures or unavailability of required numbers of GPUs is often disproportionate to the actual cost or time impact, so recovering quickly from failures is important to retain customer trust in the infrastructure, especially for large training jobs which may have failures multiple times in a day.

Failure types and frequencies: We observe from failure data from large GPU clusters in our organization that most errors during training occur due to failures of a single GPU or network device (either hardware or transient errors), while host/CPU and simultaneous multi-node failures are extremely rare. For older GPU clusters [20] reports MTBF (mean time between failures) for training jobs to be 3-23 hours, and that for non-User errors, none are node failures, and one type (GPU ECC error, 0.05% of total errors) might be a hardware error. GPU errors in [33] are also not multi-node failures. [22] report MTBF for training jobs to be 14-30 hours, MTBF decreasing linearly with increasing number of nodes. The OPT 175 billion parameter model [36] training encountered over 100 failures over a 2 months-long job using 992 GPUs, most errors were single GPU or network errors, no multi-node catastrophic errors, and 70+ errors were resolved by automatic restarts. The BLOOM 175B model [1] also encountered many errors during training. Silent Data Corruption (SDC) of training memory state is rare, only 0.61-1.76% of SDCs cause training failures [37], and SDCs are often detectable by checks including underflow/overflow, to prevent proliferation of corrupted state across GPUs. Periodic checkpointing (which is the current standard for large DNN training jobs) works, implying that errors do not corrupt model parameters in the checkpoint. Hence it is very likely that there are healthy replicas from whom GPU state can be recovered.

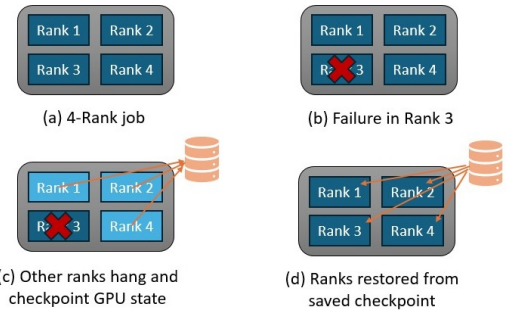

Figure 1. Just-in-time checkpointing

1.1 Just-In-Time Checkpointing

In this paper, we take a radically different approach to tackling this problem – instead of taking frequent periodic checkpoints, we propose to take *just-in-time checkpoints only after a failure has occurred!* While this may appear counter-intuitive at first, we provide justification by making several domain-specific observations.

First, deep learning training jobs process training data in an iterative manner, where each iteration processes a minibatch of training data using forward and backward passes. The key GPU state consisting of model parameters and optimizer state is updated only during a small interval as part of the optimizer step at the end of backward pass. To recover from a failure, we can recover this GPU state, either before or after the update, and any related CPU state. Second, large model training almost always employs data parallelism to improve training speed over large amounts of data. Data parallelism replicates the model across GPUs. Hence, when an error happens in a GPU in the middle of the minibatch processing, the exact same model parameters and optimizer state are available in a replica, for use during recovery.

By leveraging the fact that the GPU state is only updated during a short interval that we can track, along with the availability of multiple replicas holding the same state, we propose a method to checkpoint the crucial GPU and CPU state after a failure has been detected. Thus, the *just-in-time checkpointing solution is able to recover from common failures in few seconds by re-doing at-most a minibatch of work.* Figure 1 shows a summary of how JIT checkpointing works.

We demonstrate the feasibility and efficacy of the just-in-time approach using two design solutions, **user-level** and **transparent**. In the user-level approach (§ 3), we add library support to enable just-in-time checkpointing for user training jobs that can modify their code and may already have support for periodic checkpoints. In the transparent approach (§ 4), we make no assumptions about the user training script and enable system-level just-in-time checkpoints transparently for any deep learning training job.

Our design goal for both the user-level and transparent solution is to ensure that minimal developer work is needed to use just-in-time checkpointing. Note that our solution

does not change the job execution semantics, i.e., the job's behavior and results such as training loss / accuracy values after error recovery are the same as what they would have been without errors, within tolerances for non-deterministic operations. Further, we support a broad set of failure scenarios: since failures can happen in one or more hardware and software components of the GPU cluster, our mechanisms are not over-specialized to one type of error. If needed, periodic checkpointing could be enabled at a low frequency (in addition to JIT-checkpointing) to handle catastrophic multi-node failures.

In § 5, we use a simple analytical model to compare the various costs and trade-offs of periodic and just-in-time checkpointing approaches. We evaluate our just-in-time checkpointing mechanisms on various deep learning training jobs in § 6 and show that they can correctly recover from errors within a few seconds, thus, achieving significant speedup over prior periodic checkpointing approaches.

Contributions: The key contributions of this paper are:

- We propose a novel technique of just-in-time checkpointing when an error happens during deep learning training, which minimizes the wasted GPU work to just one minibatch, avoids overheads of frequent checkpointing, and avoids the work of re-initializing training job state, thus, greatly reducing the cost of failures.
- For user jobs that can modify their code and may already have periodic checkpointing support, we show how users can enable just-in-time checkpointing in their code with minimal change, leveraging a small interception library.
- For user-scripts that do not have checkpointing support, we design several mechanisms that enable transparent just-in-time checkpointing without changing application code.
- We derive the optimal checkpointing frequency for periodic checkpointing to minimize wasted GPU work, and show that JIT checkpointing has lower wasted work as compared to optimal periodic checkpointing.
- We evaluate these mechanisms on a variety of models and show that the error recovery time and steady-state overhead are both low.

2 Overview Of Key Mechanisms

In this section, we provide an overview of the key mechanisms that we utilize for our solutions.

Domain-aware Device API Interception: When an error happens, jobs either crash or hang today, and the job launcher usually kills all worker processes. We avoid this behavior, detect the error, and enable these jobs to checkpoint their state. To do this, we have created a library which does domain-aware transparent interception of CUDA and NCCL API calls made by the user or framework code.

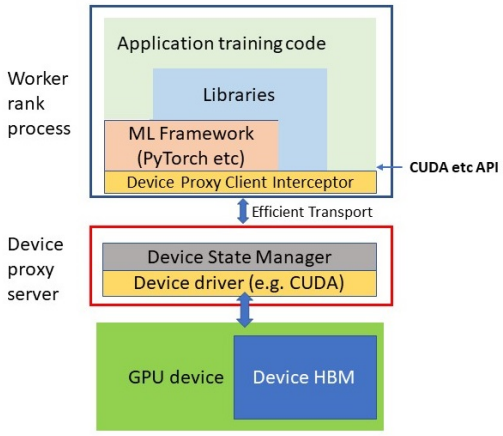

Figure 2. Device Proxy Server

In user-level checkpointing (§ 3), the interception allows us to detect errors and carefully execute a function in the user script that checkpoints both CPU and GPU state. Further, the interception also helps detect hangs and asynchronously copy the GPU state to CPU.

In the transparent solution (§ 4), the interception library does not even surface the error to the user code. It logs all GPU device APIs in steady state, detects errors without surfacing the error to application code, and then checkpoints and restores the GPU and CPU state, replays the GPU APIs, and resumes the user code. For this, we leverage the mechanism of a separate device proxy server (Figure 2), similar to the one used in [32]. The proxy server performs all device and network operations, and avoids holding any GPU or network driver state in the application's worker process. This enables clearing potentially corrupted network or GPU driver state by just restarting the device proxy server process, without impacting the application's worker process, and using mechanisms like CRIU [3] to take a system-level checkpoint of the CPU host process.

Recovery: For recovery, just-in-time checkpointing needs to redo at most a minibatch worth of work. In user-level checkpointing, the user script already has support for checkpointing CPU and GPU state. Our library helps determine the correct checkpoint state and minibatch iteration to resume the job, based on when and where the error occurs. In transparent checkpointing, our solution is broadly based on resetting the GPU state to start of minibatch and replaying device APIs from there (note that these APIs are already logged during steady state). Further, we customize the recovery solution to the different error types seen during failures.

We have created multiple error recovery solutions for different error types, using the insights and key mechanisms described above. A summary of these solutions is in Table 1, and details are provided in subsequent sections.

#	Solution	Errors Handled	User Code Change?
1	User-level	Single/multiple errors in node/GPU/network	Yes
2	Transparent; recoverable errors	Transient single / multiple errors in GPU/network	No
3	Transparent; hard errors	Single/multiple errors in node/GPU/network	No

Table 1. Summary of error recovery solutions

3 User-Level Just-In-Time Checkpointing

We now describe our user-level just-in-time checkpointing solution. Training jobs (which may have periodic checkpointing support) can enable just-in-time checkpointing with only a few modifications. The user mainly needs to initialize our library and provide a *save_checkpoint* function which the library can call.

The two main challenges in implementing just-in-time checkpointing are: 1) detecting failures, and 2) accessing consistent and uncorrupted GPU state even after failure. A naïve implementation of just-in-time checkpointing could involve modifying the Python training script to incorporate a try-except block around the training loop for detecting failures in a rank worker, such as during a CUDA call caused by GPU error. While this method can effectively detect these errors *just-in-time* on the failing rank, the other ranks may never see an exception and will not able to robustly detect such failures. Moreover, the failing rank which detects the error will not be able to access the GPU state in many cases as the failure reason could be GPU failure itself.

To address these challenges, we utilize domain knowledge of DNN training jobs and exploit the state redundancy present in data parallel replicas of large-scale jobs. In the event of a failure, we aim to leverage the healthy data parallel replicas to access and checkpoint the GPU state. However, for that, each rank worker of a distributed job must first be able to detect failures in any other data parallel replica. For this, we leverage the fact that all data parallel ranks regularly synchronize with each other via collective communication operations such as all-reduce. This means that if any rank fails, the collective operation for all other ranks will hang.

Leveraging these observations of DNN training jobs, the basic idea of our method is to use these hung data parallel rank workers to checkpoint the relevant state from each of their GPUs and then later recover using the collected state from all the ranks. At a high level, the steps involved in our solution are as follows:

1. In each rank, detect hangs during collective communication operations. A hang in one rank may indicate failure of some other rank (§3.1).
2. Upon hang detection, each healthy rank copies its GPU state to a checkpoint file (§3.2).
3. The scheduler is notified by the healthy ranks of failure detection and checkpointing. After at least one data parallel replica has completed checkpointing, the scheduler kills the job and reschedules it on a set of nodes which excludes any failing GPU(s).
4. Finally, on restart, the user code follows the existing restore path to load the GPU state from the checkpointed state and proceed with the next minibatch (§3.3).

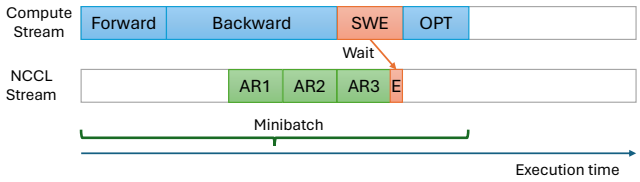

Figure 3. Computation-communication synchronization in deep learning frameworks. To overlap computation and communication, multiple all-reduces (AR) may be scheduled opportunistically as backward passes of few layers complete. A cudaStreamWaitEvent (SWE) waits on a cudaEvent (E) after the all-reduces to ensure that the optimizer step (OPT) runs only after the all-reduces complete.

3.1 Detecting communication hangs

Implementation of collective operations in DNN frameworks: We first summarize how current DNN frameworks implement calls to communication collectives such as all-reduce. Deep learning frameworks (e.g. PyTorch [11], Deepspeed [4], Huggingface [8]) schedule computation and communication kernels on separate GPU streams, so that they can execute in parallel. Moreover, for maximum GPU utilization and avoiding idle time due to latency of dispatching kernels from the CPU, these frameworks schedules kernels on the GPU streams asynchronously in advance, without waiting for their actual execution. To ensure that kernels on parallel streams execute in the correct order, synchronization primitives like *cudaStreamWaitEvent* are employed.

Consider a concrete example shown in Figure 3. As shown, the forward-pass, backward-pass and optimizer kernels are scheduled on the compute stream, while the all-reduce kernel is scheduled on the communication/NCCL stream. To make sure that the optimizer, which needs the all-reduced gradients, executes only after the all-reduce kernels finish, a *cudaStreamWaitEvent* is added on the compute stream which waits for the *cudaEvent* placed after the all-reduce kernel on the communication stream.

Using watchdog thread to detect hangs: Since an all-reduce kernel will hang if any of the participating ranks fails, we can use a watchdog thread to detect such hangs. However, we want to do this transparently without making any changes to user code or the framework (e.g. PyTorch). For this, we leverage the LD_PRELOAD [9] mechanism to intercept API calls to the synchronization primitives such as *cudaStreamWaitEvent*, *cudaEventRecord* and a few NCCL

APIs, and inject our library code. We first identify the communication / NCCL stream from the intercepted NCCL APIs. Next, we leverage the intercepted *cudaStreamWaitEvent* and *cudaEventRecord* APIs to identify all *cudaEvent*s on the NCCL stream, which have waiting *cudaStreamWaitEvent*s associated with them, and add them to a watch-list. Note that due to asynchronous execution, we may have multiple collectives and thus multiple active *cudaEvent*s at a given time. Such an event will trigger only if the corresponding all-reduce completes. To monitor these events, we start a watchdog thread at the first intercepted *cudaStreamWaitEvent*. The watchdog thread continuously monitors these *cudaEvent*s in the watch-list via calls to *cudaEventQuery*. Under usual circumstances, the events will trigger and *cudaEventQuery* will return a *cudaSuccess*. Our watchdog thread can then remove this event from the watch-list. However, in case of a failure in any rank, the all-reduce will never finish. In such cases, our watchdog thread will trigger a timeout and initiate checkpointing of the GPU state by calling the *save_checkpoint* function.

Working with large-scale training techniques: The proposed method is also compatible with large-scale training techniques such as 3D-parallelism (data-, model- and pipeline- parallelism) [5] and sharded-data-parallelism such as FSDP [7]. For FSDP jobs, JIT-checkpointing requires hybrid sharding where model and optimizer states are sharded within a node and replicated across the nodes. The replicated states are used for recovery during failures. The additional synchronization across GPUs used by these techniques act as additional target points for the hang detection mechanism. When one worker has an error, others will hang at the next synchronization point and perform a JIT-checkpoint. [1].

3.2 Saving GPU state

To keep our solution as general as possible, we allow the user to provide a *save_checkpoint* Python function to our library, which gets called by the watchdog thread upon detection of a failure. In this function, the user can save the relevant GPU state (model parameters, optimizer state, etc.) as well as any CPU state, e.g. iteration number, random number generator state. Typically, this will be based on the checkpointing function used by the user script for the periodic checkpointing implementation. An important modification is to remove any step which requires a collective communication across the GPUs. This is essential as this *save_checkpoint* function is called when an error is detected in which some GPUs might not be able to participate in the collective communication and the process might hang/crash. However, calling the user defined Python function poses a subtle challenge, due to the Python Global Interpreter Lock (GIL). Since our watchdog thread is in C++ space, calling the user defined Python function first requires the watchdog to acquire the GIL. However, the GIL may have been acquired by some other Python or C/C++ thread[2] which is hung in a synchronization API, e.g. *cudaStreamSynchronize*, which may never return since the GPU streams are stuck due to the hung collective calls, so the GIL would never be released. To get around this, we register a signal handler, which our watchdog thread triggers by raising SIGUSR1 for each thread via *pthread_sigqueue*. The signal handler releases the GIL, allowing the watchdog thread to acquire the GIL and call the user's Python checkpointing function. Since the Python thread which previously had the GIL is hung, and since the watchdog thread exits the process immediately after the checkpoint, race conditions cannot happen. The scheduler may then restart the job on replacement GPUs.

There is one more subtle issue we need to take care of. To checkpoint the GPU state, the user checkpointing function will issue *cudaMemcpy* APIs to copy the state from GPU to host memory. By default, these *cudaMemcpy* calls are issued on the default Cuda stream. The default stream, however, will be blocked due to *cudaStreamWaitEvent* on the hung communication collectives. This will lead to a deadlock and the *cudaMemcpy* will never complete. To avoid this, we simply intercept the *cudaMemcpy* call and in the case when it is called during checkpointing, we issue it on a new Cuda stream.

Saving checkpoints from multiple GPUs: When healthy ranks detect a failure, they simultaneously start to save their state and can possibly experience a failure during this process. Hence to prevent collision of checkpoint, each rank saves its state to a rank-dependent directory, which is later used to find relevant checkpoints. Additionally, a metadata file is stored at the end, which signals a complete and clean checkpoint.

3.3 Assembling the correct checkpoints

Once healthy rank workers have checkpointed their state, they can inform the scheduler, which will restore the job on replacement GPUs. In the case of model-parallel and/or pipeline-parallel jobs, the scheduler will wait for acknowledgement from at least one data parallel rank of each pipeline stage and model-parallel partition before it proceeds with the restore. During restore, the rank workers will look for checkpoint files from *any* data parallel replica and initiate restore in the usual manner (*e.g.*, using torch.load()). The *jit_get_checkpoint_path* API provided by the our library provides the corresponding path of the available replica checkpoint to be loaded. This involves discarding corrupted checkpoints, which is checked using the metadata file written while saving.

A point to note is that although the failing rank sees a failure in say minibatch iteration *i*, the data parallel replica

[1]For FSDP, minor modifications of the *sharded_optim_state_dict* function were made

[2]The recommended practice is to release GIL when entering C/C++ space, but we found violations of that in some cases.

may save the checkpoint for minibatch i or i+1 depending on the exact point of failure. If the failure happens before the all-reduce of the current minibatch, that is during the forward or the backward pass or the all-reduce itself, then the healthy ranks will get stuck at this all-reduce and save the parameter and optimizer state of minibatch i. If, however, the failure happens post the all-reduce and before the next minibatch starts, *e.g.*, during the optimizer step, then the healthy ranks will not know about the failure and proceed with the update of parameter and optimizer state and on to the next mini batch $i + 1$. They will then continue with the forward and backward pass of this $i + 1^{th}$ mini batch, and issue the all-reduce. This all-reduce, however, will get stuck as the failing rank is no longer participating. Our interception based hang detection will now trigger, and the healthy ranks will save a checkpoint of this $i + 1^{th}$ minibatch. Since in both cases the checkpoint is in a consistent state, the recovery in both cases works as usual.

4 Transparent Just-In-Time Checkpointing

While implementing the above error recovery mechanism is possible in user code, it assumes that the DNN training jobs already support user-level periodic checkpointing. However, many ML jobs today do not implement user-level checkpointing. Further, restarting the training job and re-running the application code from the beginning involves potentially large overheads of job initialization including training data preparation. Hence transparent error recovery at the system level in the ML framework or cluster infrastructure layer can be powerful, since it enables all jobs to leverage just-in-time checkpointing without users making any code changes, and without losing worker CPU state.

The design of transparent error recovery has two parts: work done during steady state and recovery work when an error happens. We first discuss error recovery for transient errors from which we can recover by resetting the GPU and network, and then recovery for hardware errors. These capabilities leverage device API interception using a device proxy server (Figure 2).

4.1 Steady State Work

In the steady state when no errors have happened, some information needs to be saved to enable recovery when errors happen. To enable replay of all device operations from the start of a minibatch iteration, the worker process needs to log all device (Cuda/Nccl) APIs, along with their input values, GPU object handles (events, streams etc), GPU buffer addresses, input training data buffer content, etc. This logging requires copying the input values etc. to a log buffer, and appending to a list of log buffers, called the replay log. At the start of every minibatch, the replay log is cleared. There is potentially a performance overhead for this logging, which we have measured to be negligible (§ 6.3). The device proxy client and server (Figure 2) used for interception and logging of of device APIs also enable these APIs to be executed asynchronously with respect to the CPU worker thread, which reduces the overhead of logging these device APIs.

Correctness Verification: We need to validate that the replay log is correct and provides the complete inputs for all device APIs (Cuda and NCCL). Our requirement is that all interaction between the host CPU process and GPU is through the device APIs. While it is theoretically possible for the host CPU process to send implicit input arguments in the form of host memory addresses to the device and accessed directly from the device without device APIs being invoked, we have not observed such implicit input in any of the models we tested (including current GPT and BERT implementations). Since we only capture device APIs called explicitly through the device proxy interception mechanism, in the unlikely case of such implicit communication, we need to disable the transparent just-in-time error recovery mechanism and fall back to the user-level mechanisms. We validate the replay log once at the 5th minibatch iteration and then once every N minibatches to detect any change of behavior as training progresses (N is configurable). During the validation minibatch, we turn off any non-deterministic device API behavior (by configuring CUDA to use only deterministic operations), since we need to compare results after two executions of the APIs for the same minibatch. Validation starts at the end of the backward pass just before the optimizer step. We first compute checksums for all GPU buffers at that point. We then reset the GPU state to minibatch start by discarding buffers which are not model parameters or optimizer state, and undoing the creation or destruction of GPU objects (such as events or streams) after start of the minibatch. We then invoke the APIs recorded in the replay log to re-execute the forward and backward passes. After this, we again compute checksums for all GPU buffers and compare with the original checksums. If the checksums match, we know that the replay log is accurate.

4.2 Transparent Recovery For Recoverable Errors

4.2.1 Recovery For Errors During Forward or Backward Pass. When an error occurs, it is necessary to catch the error and recover from it in the device API interception layer without surfacing the error to the ML framework (such as Pytorch) or application code, since they would not handle the error and probably crash the worker. In fact, the transparent error recovery mechanism keeps the ML framework and application unaware of the error: they merely see a short delay of a few seconds while recovery happens. After recovery, the original device API issued by the framework or application completes successfully, then control returns to them and they proceed with subsequent operations.

In addition to catching errors returned as error codes from device APIs, it is also necessary to detect hangs when device APIs never return. This is achieved using a separate watchdog thread (started by the device proxy interception layer). Once an error or hang is detected, the watchdog thread aborts all in-flight operations. All CPU threads (that Pytorch starts to perform GPU operations concurrently) which invoke device APIs are made to wait at the interception layer till normal execution resumes after recovery.

Once an error or hang has been detected, the first step to recovery is resetting the GPU state to the start of the current minibatch. This is done using multiple techniques depending on the type of error. In this section we consider errors which are recoverable such as transient network errors or driver errors. Subsequent sections will cover hardware errors. We first consider errors which happen during the forward or backward pass, during which the model parameters and optimizer state are not modified

- If the GPU is still accessible and no driver corruption is detected: the model parameters and optimizer state GPU buffers are retained in GPU memory, and other buffers for activations, gradients or other data are freed. This is the lowest-cost solution, since it is not necessary to incur the overhead of copying GPU buffers to host or disk and back to GPU. This case happens in GPUs which do not have any errors, but see a hang at a collective operation because of a transient network (e.g. Infiniband) issue or if another rank worker or GPU had an error.
- If the GPU is still accessible but network or CUDA driver corruption is suspected: the model parameters and optimizer state GPU buffers are copied to host. Then the GPU state is cleared or reset, by restarting the device proxy server process, which will clear any corrupted driver software state. After this, the GPU buffers saved to host (parameters and optimizer state as at start of minibatch) are copied back to the GPU.
- If the GPU state is not accessible, but there is no hardware error: e.g. this may happen in case of a CUDA "sticky" error which will cause all subsequent operations to also fail. The device proxy server process is restarted to reset and clear GPU state. The model parameters and optimizer state (as at start of current minibatch) are available in a data parallel replica which has not yet entered the optimizer step (since it is waiting at an all-reduce). So GPU buffers from the replica are copied to the GPU whose state was reset.

To ensure that at least one replica has unmodified parameter and optimizer state which can be used for error recovery, we leverage the property of neural network training jobs that parameter and optimizer state is updated only after the forward and backward passes have completed. A collective all-reduce operation is required to average gradients across all GPUs after the backward pass. All worker ranks enter this collective operation, before they modify their parameter and optimizer state. Further, each worker rank cannot exit from this collective operation till all others have reached it (so it is a barrier synchronization across all GPUs). In case of an error in any GPU, all other GPUs will be blocked at the collective operation, thus ensuring that they have not modified their parameter and optimizer state, and can supply it to the failed GPU rank worker.

Once all GPUs have reset their state to the start of a minibatch, the rank workers re-establish communication objects such as NCCL communicators, which requires a rendezvous synchronization across GPUs. They also recreate GPU objects such as events and streams which existed at the start of the minibatch. This is done by recording the device APIs used to create these GPU objects (usually at the start of training) and replaying them after state reset. Further, handles to these GPU objects are usually returned to the application or ML framework, and after re-creation these handles change, but we cannot change the handles already held in application variables. So it is necessary to return virtual handles to the application from the interception layer at the beginning of training, and map these virtual handles to new physical handles after re-creating the GPU objects. This virtual handles technique is also used in [32].

After resetting and re-initializing GPU state, all workers replay the device APIs recorded in the replay log. This brings them all to the last API where the error or hang occurred, and once that API successfully completes, the device interception layer can return the last API results to the ML framework or application which called the API. The application then proceeds with further work, unaware that an error happened.

4.2.2 Recovery For Errors During Optimizer Step.

In case an error happens during the optimizer step (and related steps such as learning rate scheduler) in a GPU, it may have partially modified model parameter and optimizer state, so its GPU state cannot be used to reset training to start of current minibatch. Also, since all other GPUs have probably entered the optimizer step too (since all GPUs have completed the collective all-reduce to average gradients before optimizer step), none of the replicas have the parameter and optimizer state as of the start of the current minibatch. So we cannot reset the GPU state to start of the current minibatch. However, since all GPUs independently perform the optimizer step, and compute the same model parameter and optimizer state (in parallel, using the averaged gradients which have been sent to all GPUs in the all-reduce operation), the remaining GPUs have the correct parameter and optimizer state as of the start of the *next* minibatch. So the GPU which got an error can copy the parameter and optimizer state from a replica, and set its GPU state to the start of the next minibatch instead of the current minibatch. In this case, replay of minibatch APIs is not needed, because the application will

execute the next minibatch APIs as usual. After completing the error recovery, the device interception layer in worker which got the error needs to ignore all device APIs called by the ML framework or application within the rest of the optimizer step of current minibatch (since they may modify the GPU state of parameters and optimizer), and start executing device APIs from the beginning of next minibatch (forward pass) onwards.

Since the device API interception layer only sees CUDA/NCCL API requests, we need additional hooks to enable it to differentiate between the APIs in different stages of a minibatch (optimizer step v/s forward & backward passes), so that the appropriate error recovery mechanism can be used in each stage. This is done by adding pre-optimizer-step and post-optimizer-step callback hooks in the ML framework such as PyTorch, which inform the device API interception layer about the start and end of the optimizer step.

4.3 Transparent Recovery for Hard GPU Errors

In case there is an unrecoverable hardware error in the GPU or network, it cannot be used further, and the worker rank for that GPU must be attached to another GPU, potentially on another host node. All other GPUs will hang at the next collective operation such as all-reduce, enter error recovery mode, do the just-in-time checkpoint of their GPU state at the start of the minibatch, and wait at the rendezvous synchronization point to initialize their communicators. At this point a consistent checkpoint of CPU state can be taken for the rank worker of the failed GPU and workers in that node, to enable migration to a new host node with a fresh set of GPUs.

Transparent migration of jobs without change in user code is supported by the existing distributed job scheduling and monitoring framework of the GPU cluster, which performs a checkpoint of all the worker CPU processes using the CRIU tool [3] on Linux hosts, similar to [32]. The scheduling framework then allocates a new set of hosts and GPUs for the job, and restores the worker processes' CPU states on the new set of hosts, again using the CRIU tool. So all worker ranks' CPU processes are resumed from the point where the CRIU checkpoint was done. Once resumed, all workers proceed to initialize their GPU state, including communicator objects and GPU objects such as streams, as described in the previous section. They can then restore GPU buffers for model parameter and optimizer state, from the checkpointed files.

Since the GPU which got the hardware error could not write any GPU state to checkpoint files, it must read those files written by a replica. As in the case of user-level checkpointing, all workers checkpoint their GPU buffers to a shared file system or object store, and a consistent file naming scheme across GPUs is used. We use a unique identifier for all parameter and optimizer state GPU tensors to be stored, which is same across GPUs. In case of transparent checkpointing at the device interception layer, we cannot use the key of the dictionary of tensors maintained by the ML framework such as PyTorch. So we use a hash value derived from the call-stack of the buffer memory allocation request (the call-stack is obtained by the device interception layer), a sequence count (in case of multiple allocations at same call-stack), and the buffer size. This enables the failed GPU rank to construct the same file name as written by replicas for the same GPU tensor buffers, and after restart, read all its GPU buffers written by replica ranks.

5 Failure Overhead Analysis

In this section, we first provide quantitative estimates of the failure recovery costs for large training jobs. We then analyze the costs due to wasted GPU work and overheads for failure recovery using traditional periodic checkpointing and just-in-time checkpointing. We derive analytically the optimal checkpointing frequency for periodic checkpointing, and the minimum wasted GPU work for this optimal frequency. We then derive the wasted GPU work for just-in-time checkpointing and show that it is lower than the wasted GPU work for periodic checkpointing with the optimal checkpointing frequency, for large numbers of GPUs.

5.1 Estimating Dollar Cost Of Errors

The frequency of errors encountered by deep learning training jobs varies to some extent depending on the specific GPU device and networking hardware and software. In large supercomputing clusters operating in the authors' organization, the normalized error rate for a 1000 GPU job averaged approximately 1 error per day. For training the OPT 175 billion parameter model [36], over 100 hardware failures were experienced over the course of a 2 months-long training job using 992 GPUs, averaging approximately 2 errors per day. The BLOOM 175B [1] model also encountered many errors during training. Since the failure rate increases linearly with number of components, larger training jobs with more GPUs are likely to encounter more frequent errors.

The cost due to errors during AI training jobs is the dollar cost of number of GPU-hours wasted when errors happen. It is proportional to the frequency of errors (the error frequency scales as O(N) for N GPUs, since the job experiences an error if any of the GPUs fails). The cost is also proportional to the number of GPU devices used by the job (O(N)), since each failure in one GPU causes all other GPUs to wait till the failure is detected, and then all GPUs must do some work to recover from the error and resume training from the point where the error happened. Hence, the cost scales as O(N^2), increasing quadratically with the number of GPUs. Large model training jobs, which use thousands of GPUs, are thus particularly vulnerable to errors.

With periodic checkpointing, the cost of errors can be large for AI training jobs using 1000s of GPUs. E.g. a 1000

GPU job which experiences 1 error per day would lose on average half the checkpoint interval (e.g., 15 minutes for periodic checkpoints every half hour), across all the 1000 GPUs. Assuming the cost of a current generation GPU such as the Nvidia A100 80GB GPU is $4 per hour, the monthly cost of errors would be 1000 GPUs * 30 errors/month * 0.25 hour/error * $4/hour = $30,000 per month. For a 10,000 GPU job, the cost increases quadratically to $3 million per month.

5.2 Optimal Checkpointing Frequency For Periodic Checkpointing

For each checkpoint, the training job has to copy the GPU state to a persistent store. This adds a steady-state overhead to the DNN job depending on the cost of one checkpoint, the checkpointing frequency and the number of GPUs. For large models, each checkpoint is huge (100s of Gigabytes to Terabytes for model parameters and optimizer state for billions of parameters). E.g. with a PCIe Gen 4 bus which has a bandwidth of up to 32 GB/sec, transferring memory state for 8x A100 80 GB GPUs (total 640GB) to host RAM would take at least 20 seconds, and more time to copy to SSD disk or cloud storage. If all GPUs are paused while their state is copied to RAM or disk in the host, that incurs significant overhead of wasted GPU time.

Ideally, we would like to have a lower checkpointing frequency to minimize this steady-state overhead. However, checkpointing infrequently would mean paying a high cost in case of failure since all GPUs of the DNN training job will have to redo a large amount of work from a long-ago checkpoint. Thus, we need to arrive at an optimal checkpointing frequency which does a careful trade-off between the two competing factors.

We can analytically derive the optimal checkpointing frequency based on a few parameters of the training job and GPU cluster infrastructure:

1. o: This is the overhead time for performing one checkpoint on one GPU. This will depend on the size of the DNN model, bandwidth, checkpointing mechanisms, etc. which will determine the time taken to copy the GPU state.
2. f: This is the failure frequency of one GPU. This will depend on the cluster provider, hardware used, etc.
3. r: This is the fixed cost recovery time from a failure per GPU. It includes costs due to downloading the checkpoint file to all GPUs, process and GPU initialization, training data preparation, etc. This fixed cost is independent of the amount of work redone since the last checkpoint.
4. N: The total number of GPUs used for training.

The total failure rate of the job will depend on both N and f — the higher the number of GPUs, the higher the probability of some GPU failing, causing the entire job to fail. The total failure rate of the job will thus be $N \times f$.

Let the periodic checkpointing frequency be c. Typically this is provided by the user, but here we want to compute the optimal frequency given the above parameters. For this, consider a DNN job which has been training for time t. Then, we can say that

- Expected number of failures in this duration = Nft
- Total number of checkpoints in this duration = ct
- Total periodic checkpointing steady-state overhead (GPU time) across all GPUs = $Ncto$
- Fixed cost recovery time r per failure across all GPUs = $Nft \times r \times N = N^2ftr$
- The wasted work which is redone upon restore from the last checkpoint across all GPUs = $Nft \times \frac{1}{2c} \times N = N^2ft\frac{1}{2c}$ (this is on average half the checkpoint interval $1/c$, in which a few hundreds or thousands of minibatches of data are processed again)
- Total GPU time spent in failure recovery (fixed + redone work) = $N^2ft(r + \frac{1}{2c})$

Thus the total (expected) GPU time wasted in checkpointing and failure recovery is:

$$\begin{aligned} W &= Ncto + N^2ft(r + \tfrac{1}{2c}) \\ &= Nt(co + Nfr + \tfrac{Nf}{2c}) \end{aligned} \tag{1}$$

To find the optimal c which minimizes W, we can take the derivative of W w.r.t. c and set it to 0:

$$\frac{dW}{dc} = Nt(o + 0 - \frac{Nf}{2c^2}) = Nt(o - \frac{Nf}{2c^2}) = 0, \tag{2}$$

which results in

$$c^* = \sqrt{\frac{Nf}{2o}}. \tag{3}$$

It can be seen that $d^2W/dc^2 = N^2tf/c^3$, which is always positive as t, N, f, c are positive. Hence, c^* is indeed the optimal checkpointing frequency which minimizes wasted work W.

It may be noted that the checkpoint frequency needs to be higher than the failure frequency else the job may not progress reliably. Further, since the optimal checkpoint frequency depends on the number of GPUs, for large training jobs with thousands of GPUs, the checkpoint frequency needs to be much higher than failure frequency of one GPU.

5.3 Wasted GPU Work with Periodic Checkpointing

We now derive the wasted GPU work of periodic checkpointing at this optimal checkpointing frequency. The total wasted GPU time can be obtained by substituting c^* (eq. 3) into equation 1:

$$W^* = Nt(\sqrt{\frac{Nfo}{2}} + Nfr + \sqrt{\frac{Nfo}{2}}). \tag{4}$$

The three terms correspond to the overheads due to 1) repeated checkpointing cost o, 2) the fixed recovery cost r, and 3) the expected wasted work redone upon restore from the checkpointed minibatch. Note that the first and last

terms come out to be the same for c^*, which is expected as the optimal to trade-off the two terms will be symmetric.

Wasted Time Fraction: Equation 4 gives the wasted time over N GPUs when the job is run for *useful* time t, i.e., job runtime excluding checkpointing and failure overheads. If we denote w^* to be the wasted time per GPU, per unit time, at optimal checkpointing frequency, we get:

$$w^* = \sqrt{\frac{Nfo}{2}} + Nfr + \sqrt{\frac{Nfo}{2}}. \quad (5)$$

The wasted time fraction per GPU, w_f is then the wasted time Wt divided by the total time $t + Wt$, i.e.:

$$w_f = \frac{W}{1+W} \quad (6)$$

We show the experimentally observed values of w_f in § 6.3.

5.4 Wasted GPU Work With User-Level Just-In-Time Checkpointing

In this paper we proposed methods to perform *just-in-time* checkpointing. This eliminates the first term (checkpointing overhead) in the wasted GPU work equation 1, as just-in-time checkpointing will initiate a checkpoint only upon a failure, and avoids the steady-state cost of periodic checkpointing. Instead there is a single checkpoint cost per failure: $Nfto$, where ft is the number of failures per GPU in time t.

Our implementation of just-in-time checkpointing does impose some added steady state costs, due to the interception of GPU APIs. This is nearly zero (see § 6) for the user level method (§ 3). For transparent just-in-time checkpointing, the overhead of logging all device APIs (used for replay during recovery) is also nearly zero since it is asynchronously overlapped with CPU work by the device proxy. Let the steady state overhead time of just-in-time checkpointing per GPU and per unit time be represented by o_{jit}.

Since the checkpoint will be at the start of the current minibatch iteration where the failure occurred, the wasted GPU work on restore is limited to half the minibatch time m on average (typically less than a second to few seconds). The total wasted GPU work with user-level just-in-time checkpointing is then:

$$W_{jit} = Nt(fo + o_{jit} + Nfr + Nfm/2) \quad (7)$$

Comparing this with the wasted GPU work for periodic checkpointing (eq. 1), we see that

1. the first term involving checkpointing overhead $Nfto$ is less for just-in-time-checkpointing than periodic checkpointing overhead $Ncto$, since the checkpoint frequency needs to be higher than the failure frequency else the job may not progress reliably. Further, since the optimal checkpoint frequency increases with the number of GPUs (eq. 3), for large training jobs with thousands of GPUs, the periodic checkpoint frequency and costs are likely to be much higher than failure frequency and on-failure checkpoint cost.
2. the second term o_{jit} is the nearly zero overhead of interception.
3. the third term (involving fixed cost r) is the same for periodic and just-in-time checkpointing.
4. the fourth term for recovery time is much less for just-in-time checkpointing, since redone GPU work is on average half a minibatch $m/2$ whereas for periodic checkpoint it is on average is half the checkpoint interval $\frac{1}{2c}$ consisting of hundreds or thousands of minibatches.

Thus, apart from the small overhead of interception, the total wasted GPU work is less for user-level just-in-time checkpointing.

5.5 Wasted GPU Work With Transparent Just-In-Time Checkpointing

Transparent just-in-time checkpointing eliminates the main fixed cost contributor to the recovery cost, r, since it restores the CPU process from the exact state at failure (the application code is unaware that failure and recovery have happened), and thus does not incur the training job initialization costs. This makes the fixed recovery cost r negligible.

Further, the transient-error recovery mechanism (§ 4.2) eliminates the checkpoint cost of copying GPU state to host and persistent store, by merely resetting GPU state to the start of minibatch iteration when a transient error happens. Hence the total wasted GPU work with transparent just-in-time checkpointing for transient errors becomes:

$$W_{jit} = Nt(o_{jit} + Nfm/2) \quad (8)$$

Comparing this with the wasted GPU work for periodic checkpointing in eq. 1, we see that the fixed cost term Nfr is eliminated, and the recovery cost third term is much less for just-in-time checkpointing, as explained above, since at most 1 minibatch of GPU work is redone. The first term involving steady state overhead of intercepting and logging GPU APIs is observed to be nearly zero (§ 6), less than typical periodic checkpointing overhead, depending on the actual implementation. Hence the total cost for this variant of just-in-time checkpointing is also very likely to be lower than periodic checkpointing.

6 Evaluation

6.1 Implementation

We have implemented User-level JIT-Checkpointing (§ 3) as a Python library and a C++ shared library. The library has built-in support for checkpointing with frameworks Megatron [31], DeepSpeed [4], and HuggingFace [8]. Transparent JIT-Checkpointing (§ 4) is implemented in the data plane of a modern AI job execution infrastructure which manages a large number of GPU clusters in our organization.

Model	#Params(B)	#GPUs	Parallelism	Framework
GPT2-S	0.124B	4xA100	4D	Megatron-DS
GPT2-S-3D	0.124B	8xV100	2D-2P-2T	Megatron-DS
GPT2-XL	1.5B	8xV100	2D-2P-2T	Megatron-DS
GPT2-8B	8.3B	2x(8xV100)	2D-4P-2T	Megatron-DS
GPT2-18B	18B	4x(8xV100)	2D-4P-4T	Megatron-DS
BERT-L-PT	0.334B	8xV100	8D	Megatron
BERT-B-FT	0.110B	8xV100	8D	Hugging Face
T5-3B	3B	2x(4xA100)	FSDP	PyTorch
ViT	0.632	8xV100	8D	PyTorch
PyramidNet	0.24B	4xA100	4D	PyTorch

Table 2. Experimental workloads used for evaluation. PT=Pre-training, FT=Fine-tuning, DS=DeepSpeed. For parallelism, 2D-4P-2T means 2-way Data-parallel, 4-way Pipeline-parallel, 2-way Tensor-parallel. For GPUs, 2x(8xV100) means 2 nodes of 8 V100s each.

6.2 Experiments

We evaluate the presented error-recovery mechanisms for multiple models of different sizes and multiple frameworks. The details of the experimental workloads are provided in Table 2. We evaluate our mechanisms on the wasted GPU hours as a fraction of the total time. Experiments were run on NVIDIA V100 32GB GPUs (each node having 8 V100 GPUs) and A100 80GB GPUs (each node having 4 A100 GPUs). Our mechanisms are semantics and accuracy preserving: we validate exact floating point match of training losses with and without JIT-checkpointing (under deterministic conditions).

6.3 Comparison with Periodic Checkpointing

We compare user-level JIT checkpointing to three variations of periodic checkpointing. *PC_disk* refers to saving checkpoints to a persistent disk in the critical path, which is commonly performed using *torch.save()*. We also consider a stronger baseline *PC_mem* where we use an optimized implementation akin to [2], and write the checkpoint to CPU memory, via a Linux *tmpfs* mount. The checkpoint file can then be copied asynchronously to a persistent store without impacting the job execution in the critical path. The third baseline is the optimized periodic checkpointing technique *CheckFreq* [23].

Table 3 shows the checkpointing overhead for these implementations, assuming checkpointing is performed at the optimal frequency, assuming a failure rate of 2 failures per day on 992 GPUs [36]. For these models, this optimal frequency corresponds to one checkpoint every 1 to 3 hours. As is clear from the table, JIT checkpointing has negligible steady state overhead while all three periodic checkpointing approaches suffer from increasing overheads as the model size increases. As we show in Section 6.5, even these small overheads can result in significant wasted GPU hours when models are trained using large number of GPUs.

Finally, JIT and periodic checkpointing may be used together. Both mechanisms use the same code and file formats for saving and loading model training state, so they are easy to enable together if desired, reducing the developer's work of supporting various failure scenarios. During recovery after failure, the most recent checkpoint will be used, which can be either a periodic checkpoint or a JIT checkpoint. Since multi-node failures are rare, and further, only catastrophic failures that eliminate all data-parallel replicas require periodic checkpointing (not every multi-node failure), the frequency of periodic checkpointing can be low (e.g. once a day). Table 3 shows the overhead for periodic checkpointing at this low frequency of once a day (*PC_1/day*), which may optionally be added to the overhead of JIT-checkpointing, if both are enabled.

Model	PC_disk	PC_mem	CheckFreq	PC_1/day	JIT-C
GPT2-S	0.042	0.042	0.024	0.004	0.0024
GPT2-XL	0.093	0.078	0.047	0.007	0
GPT2-8B	0.216	0.186	0.111	0.02	0
GPT2-18B	0.330	0.275	0.166	0.02	0
BERT-L-PT	0.07	0.068	0.031	0.005	0.0076
BERT-B-FT	0.039	0.036	0.026	0.0016	0

Table 3. Checkpointing overhead percentages for Periodic Checkpointing baselines, assuming optimal checkpointing frequency, compared to JIT-Checkpointing (JIT-C)

6.4 Just-In-Time Checkpointing Overhead Analysis

Tables 4, 5, 6 show the time taken for the just-in-time checkpointing based error recovery mechanism and steady state overhead, for user-level code change based error recovery, transient errors recovery (without clearing GPU state, as described in section 4.2), and hard errors recovery with CPU checkpointing, as described in section 4.3.

The recovery time is measured from the point of time the error was detected through all recovery mechanisms including checkpoint, training job initialization, restore, and execution of the current minibatch device APIs, till the time of start of the API or the minibatch where the error originally happened. We exclude the checkpoint file transfer time over network, the wait time for ranks to detect errors in other ranks, and scheduler delay in restarting the job, since this depends on the AI cluster monitoring and control plane mechanisms. As seen from Table 4, user-level JIT Checkpointing overheads are negligible and recovery time is only tens of seconds.

For transparent just-in-time checkpointing, there may be a steady state overhead due to logging of GPU APIs in the replay log. As shown in Table 5, this overhead is nearly zero for all our models. We measure the overhead as the increase in average minibatch execution time in steady state, with the baseline being job execution without the replay logging.

Table 6 shows the recovery times for hard errors. We exclude the time taken by the control plane to provision a new set of nodes and job initialization time such as downloading

Model	Checkpoint	Restore	JIT Recovery	Minibatch	Overhead
BERT-L-PT	5.0	9.9	14.8	0.418	0.00003
BERT-B-FT	1.4	8.8	10.1	0.416	0*
GPT2-S	3.8	7.2	10.35	0.629	0.00001
GPT2-XL	6.7	14.0	20.6	2.632	0*
GPT2-8B	18.8	28.6	46.9	2.953	0*
GPT2-18B	20.5	34.2	54.8	3.474	0*
T5-3B	7.6	35.25	42.65	0.498	0*
ViT	4.6	20.2	24.4	0.292	0*

Table 4. Baseline checkpoint and restore time, JIT Recovery time (checkpoint + restore), minibatch time and JIT steady-state overhead per minibatch (all in seconds) for user code change based recovery. *No measurable increase in minibatch times.

Model	Recovery Time	Minibatch Time	Overhead Time
	8x V100 32GB, 1 node		
BERT-B-FT	2.1	0.279	0*
GPT2-S	9.1	0.270	0.001
GPT2-S-3D	16.4	0.209	0*
Pyramidnet	1.9	0.315	0*
	4x A100 80GB, 1 node		
BERT-B-FT	2.6	0.079	0*
GPT2-S	11.8	0.343	0.001

Table 5. Recovery time, minibatch time and steady-state overhead per minibatch (all in seconds) for transparent transient error recovery. Times are not comparable across V100 and A100 GPU families. *No measurable increase, or decrease in minibatch times with device proxy.

Model	Recovery Time Healthy GPU	Recovery Time Failed GPU	Minibatch Time
	8x V100 32GB, 1 node		
BERT-B-FT	25.72	21.02	0.243
GPT2-S	23.97	20.85	0.210
GPT2-S-3D	23.07	18.11	0.156
Pyramidnet	38.42	30.34	0.270
	4x A100 80GB, 1 node		
BERT-B-FT	17.19	9.09	0.084
GPT2-S	14.68	8.55	0.350
Pyramidnet	28.79	17.56	0.451

Table 6. Recovery times (checkpoint and restore) for transparent hard error recovery (in seconds).

and starting Docker containers and downloading training data on the new nodes, since such work is independent of the error recovery mechanism. It may be seen that hard error recovery times are significantly higher than transient error (which does not copy or clear GPU state), this is primarily due to the cost of checkpointing GPU state to CPU and checkpointing worker process CPU state using the Criu tool. Further, healthy GPU ranks checkpoint all their GPU state whereas failed GPU ranks do not, so the recovery times are higher for healthy ranks.

We provide the breakup of recovery time for the transient error recovery case in Table 7. As may be seen, the majority of

Step	Bert-B-FT	GPT2-S	GPT2-S-3D	Pyramidnet
Delete communicators and GPU handles	1.013	0.779	0.831	0.850
Recreate NCCL communicators	1.054	8.340	15.54	1.038
Reset GPU buffers	0.001	0.001	0.001	0.002
Recreate GPU handles	0.006	0.004	0.004	0.027
Replay minibatch APIs	0.006	0.004	0.002	0.004

Table 7. Time (seconds) taken for each step of transparent transient error recovery, on one rank worker on 8x V100 GPUs

the time is taken for recreating NCCL communicators across all GPU workers, while the time taken to replay minibatch APIs is just a few milliseconds (this depends on how much each rank worker has progressed in the minibatch iteration, and is at most the minibatch processing time of a fraction of a second to few seconds).

6.5 Scaling Analysis of Wasted GPU Time

We now look at the GPT model and see concretely how the values derived analytically in Section 5 scale for this model, as the number of GPUs N increases. The cost overhead per checkpoint for BERT-L-PT is $o = 5s$ (Table 4). [36] reported a failure rate of approximately 2 failures per day on 992 GPUs. Assuming a similar failure rate, i.e., $f \approx 2 \times 10^{-3}$ per GPU per day, the optimal checkpointing frequency is given by equation 3:

$$c^*_{BERT-L-PT} = \sqrt{\frac{N \times 2/(1000*24*3600)s}{2 \times 5s}} \approx \sqrt{N}/6hr \quad (9)$$

For $N = 4$, we will need to checkpoint once every 3 hours to minimize wasted GPU time. This frequency keeps increasing as we increase the number of GPUs, *e.g.*, for $N = 1024$, the optimal checkpointing frequency is 5.54 times every hour or once every $\sim$ 11 minutes.

Now let us look at the wasted GPU time because of failures and checkpointing. BERT-L-PT has a restore and initialization time per failure, $r = 9.9s$. Substituting into equation 5, we get the wasted GPU time per unit time, w^*:

$$w^*_{BERT-L-PT} = 4.8 \times 10^{-4}\sqrt{N} + 2.3 \times 10^{-7}N \quad (10)$$

At lower N, this will be dominated by the first term (checkpointing overhead and wasted work), but at higher N, the second term (failure recovery cost) will dominate.

We now look at the wasted time fraction w^*_f (equation 6). At $N = 4$, $w^*_f = 0.1\%$. However, this rises with N, *e.g.*, at $N = 1024$, $w^*_f = 1.53\%$. Table 8 shows the optimal periodic checkpointing frequency and the corresponding wasted time fractions at different Ns.

Table 8 compares these wasted GPU times for periodic checkpointing with the two just-in-time checkpointing mechanisms (for which the recovery times and steady-state overheads are presented in 6.4). We find that the wasted GPU

Model	N=4		N=1024		N=8192	
	c^*	w_f	c^*	w_f	c^*	w_f
	Periodic Checkpointing					
BERT-L-PT	0.34 / hr	0.096%	5.54 / hr	1.53%	15.6 / hr	4.5%
BERT-B-FT	0.65 / hr	0.05%	10.47 / hr	0.83%	29.62 / hr	2.41%
GPT2-S	0.39 / hr	0.08%	6.35 / hr	1.34%	17.98 / hr	3.78%
GPT2-8B	0.17 / hr	0.18%	2.85 / hr	2.96%	8.08 / hr	8.24%
	User-level JIT Checkpointing					
BERT-L-PT	-	0.75%	-	0.77%	-	0.94%
BERT-B-FT	-	0.23%	-	0.26%	-	0.41%
GPT2-S	-	0.24%	-	0.26%	-	0.38%
GPT2-8B	-	0.0003%	-	0.07%	-	0.56%
	Transparent JIT Checkpointing (Transient error)					
BERT-B-FT	-	0.69%	-	0.69%	-	0.69%
GPT2-S	-	0.69%	-	0.69%	-	0.7%

Table 8. Scaling wasted GPU time for Just-in-time and Periodic Checkpointing. Legend: c^* = Optimal checkpointing frequency (eq 3), w_f = Wasted time percentage (at optimal frequency for periodic checkpointing. See eq 6). The Transparent JIT Checkpointing results are based on 8x V100 GPU experiments.

time fraction w_f of just-in-time checkpointing solutions are comparable to periodic checkpointing at small N, but *the wasted GPU time for JIT checkpointing grows much more slowly with large N.* This is due to the lower recovery time (eq 7). For transparent JIT checkpointing with transient errors, the wasted time stays flat as recovery time is very low (eq 8). Thus, for large N, just-in-time checkpointing solutions have much smaller wasted GPU time than periodic checkpointing.

6.6 Summary of evaluation results

Our evaluation of just-in-time checkpointing and comparison of overheads with periodic checkpointing shows that:

- JIT checkpointing works with small and large models across multiple GPUs with varying parallelism (data, model, pipeline parallelism) and model state sharding.
- The steady-state overheads for both user-level and transparent just-in-time checkpointing are nearly zero.
- JIT checkpointing recovery times are only a few seconds.
- Wasted GPU work due to JIT checkpointing increases much slower with number of GPUs than periodic checkpointing at optimal frequency. JIT checkpointing has significantly lower wasted GPU times for 1024 GPU jobs (Table 8).

7 Related Work

Error recovery for DNN training jobs is currently accomplished through periodic checkpointing of the model parameters and optimizer state from the GPU and CPU state such as the current minibatch iteration number, to a reliable file or object store. Errors are detected either when GPU operations (e.g. Cuda or NCCL APIs) return an error code, or when they hang. When such errors happen, the job's CPU processes (workers or ranks) simply exit with an error code. This is detected by the monitoring framework usually provided by the job execution infrastructure, which then restarts all workers of the job, usually on a new set of hosts with attached GPUs. The workers download the previously checkpointed state from the file or object store, and initialize their CPU and GPU state. They then proceed to download the next minibatch of training data and resume training from the iteration after the checkpoint.

Fault tolerance through replication and recovery by copying state checkpoints from a replica is well-known in distributed systems [12] [19]. The Delta-4 system [17] restarts a failed process using a checkpoint from a replica. [29] analyzes on-demand checkpointing of a replica after a failure happens, and shows that its job completion time is less than periodic checkpointing. We take inspiration from these works, and propose and implement a similar technique for DNN training jobs, which have domain-specific computation, communication and memory state characteristics including GPU usage, data-parallel replicas, synchronous minibatch bounded state update through all-reduce, and software frameworks such as Pytorch, Cuda and NCCL. We show that just-in-time checkpointing is a valid and low-overhead solution for fault tolerance for DNN training jobs.

Several works optimize the overhead of periodic checkpointing in steady-state. For HPC applications, many techniques have been developed for asynchronous checkpointing to overlap the I/O latency with computation [34], [30], optimize the checkpointing interval based on failure rates observed [15] at multiple levels of storage [16], using a failure prediction framework [21], and for applications using MPI [26]. For DNN training applications, Checkfreq [23] reduces overhead and enables more frequent checkpointing by overlapping computation with snapshotting of model GPU state to memory, and tuning the checkpointing frequency at run-time using profiling. Gemini [35] checkpoints GPU state to local and remote CPUs, and interleaves checkpointing communication traffic into gaps between training traffic, to reduce overheads and enable checkpointing on every iteration. However, it does not leverage the data parallelism in large model training jobs, which makes such copying unnecessary, since replica GPUs already have the model and optimizer state. Gemini also assumes long iteration times of 10's of seconds to hide the checkpoint copying time of 2-3 seconds per iteration, and does not support pipeline parallelism, which may increase the amount of training traffic in the middle of an iteration and leave less room for checkpointing traffic. Elastic Horovod [6] and Nebula [2] save checkpoints from GPU to CPU memory, and Nebula does the copy asynchronously thereby reducing the time during which the training job is paused to do checkpoints. DeepFreeze [24] performs asynchronous checkpointing of CPU state to local storage and then to external storage, shards the checkpointing work across rank workers, and uses a compact

binary format. Incremental checkpointing techniques such as Check-N-Run [18] copy to CPU only part of the model state which is modified, for certain types of models such as recommendation models which do not update all parameters in every minibatch.

Deepspeed ZeRO [28] and FSDP [7] shard model parameters and optimizer state across data-parallel GPUs, so that each GPU is only responsible for checkpointing a fraction of the entire state, reducing the overhead of copying GPU state to host memory. Hybrid sharding with small number of replicas (in FSDP) enables JIT-checkpointing, has performance benefits over ZeRO (full sharding), and reduces memory. ZeRO without replicas prevents JIT-checkpointing benefits, and periodic checkpointing could be used.

In contrast to such periodic checkpointing optimization work, our approach in this paper is to avoid checkpointing until an error occurs, and then do a just-in-time checkpoint leveraging existing data-parallel replicas. This avoids not only the overhead of periodic checkpointing in steady-state, but also minimizes the wasted work during recovery from checkpoint, since the checkpoint is taken at the beginning of the current minibatch iteration where the error happened, so at most one minibatch of training data is re-processed. Further, for transient errors such as networking issues or driver state corruption, no checkpointing is needed: re-initializing the state and retrying the operations is sufficient to proceed. Our approach is complementary to and can leverage optimizations for checkpointing such as asynchronous copying of state from GPU to CPU presented in the above related work. Our approach also avoids the need to guess the optimal checkpointing frequency, since the failure rate is often unknown, highly variable and unpredictable across job runs.

Swift [38] avoids steady state overhead of periodic checkpointing GPU state to CPU memory by recovering consistent model state in surviving workers using invertible operators to undo model update operations in case of partial model updates, and then sending the model to failed worker ranks. This is similar to user-level just-in-time checkpointing in avoiding steady state overheads, however, Swift requires optimizers to use only invertible operators, and may not work for all models. Also, Swift requires application code change to use their library with Pytorch, while in this work we provide a transparent just-in-time checkpointing capability which does not require code change.

There are ongoing efforts to train DNNs/LLMs in an asynchronous manner, such that failures of a single GPU worker can be ignored, without blocking all other GPUs from proceeding to subsequent minibatches [27]. CPR [22] for recommendation models uses partial recovery where only the failed worker recovers from checkpoint, while all other workers proceed without re-executing minibatches since the checkpoint, with techniques to reduce accuracy loss. Hoplite [39] proposes a fault-tolerant collective communication layer which is able to dynamically adapt data transfer schedules when errors happen. However, currently such asynchronous techniques are not used for LLM training due to multiple reasons. First, ignoring one GPU's contributions to the gradient does not preserve semantics of the original job (i.e. if the job ran without any failures), so ML practitioners have concerns on impact on model accuracy. Second, GPU communication frameworks (such as NCCL) require all pre-configured GPUs to participate in each operation. A single GPU failure will cause all GPUs to hang. Reconfiguring NCCL or pipeline/model-parallel algorithms on-the-fly to use one less GPU is not supported currently. Third, even if the job continues with training without one GPU, future training will happen with one less GPU, and parallelism reduces over time as more GPUs fail, resulting in longer job execution times. With fewer GPUs over time, the effective global batch size will reduce, so hyper-parameters such as learning-rate will need to be re-tuned. This is non-trivial and risks hurting accuracy if not done correctly. Finally, to avoid training with fewer GPUs, we need to recover the failed GPUs at some point. This will require techniques similar to JIT-checkpointing described in this paper: copy state from a replica and recover the failed GPU right after the failure in the same minibatch.

8 Conclusion

We have presented novel mechanisms for just-in-time checkpointing when an error happens during deep learning training jobs, which minimizes the wasted GPU work to just one minibatch. We have presented results that just-in-time checkpointing significantly reduces the time for error recovery to a few seconds, thus reducing the cost of wasted GPU time. We have shown experimentally and analytically that the wasted GPU work and overheads are less than what periodic checkpointing incurs at optimal frequency. In particular, the user-level recovery solution applies to jobs that can modify their code and may already have periodic checkpoint support, while the transparent error recovery mechanisms presented enable any deep learning training job to use these new capabilities without code change. We also highlight that in practice since failures are highly unpredictable and failure rates are variable from job run to run, it is difficult to calculate the optimal checkpoint frequency, so users often guess or estimate the frequency which may be too high or too low for a particular training job run, and incur the costs of much more wasted GPU work than optimal, even if there are no or very few failures. With just-in-time checkpointing, such guesswork is avoided. As deep learning models, training data and compute clusters scale to ever larger sizes, we believe such techniques will be essential to reduce the cost and time impact of errors.

Acknowledgments

We thank our shepherd Ken Birman and the anonymous reviewers for their valuable comments and suggestions.

References

[1] BLOOM Training. https://huggingface.co/blog/bloom-megatron-deepspeed#training-difficulties.

[2] Boost Checkpoint Speed and Reduce Cost with Nebula. https://learn.microsoft.com/en-us/azure/machine-learning/reference-checkpoint-performance-for-large-models.

[3] CRIU: Checkpoint Restore in Userspace. https://criu.org/Main_Page.

[4] Deepspeed. https://www.deepspeed.ai/.

[5] DeepSpeed: Extreme-scale model training for everyone. https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone/.

[6] Elastic Horovod. https://horovod.readthedocs.io/en/stable/elastic_include.html.

[7] Fully Sharded Data Parallel: faster AI training with fewer GPUs. https://engineering.fb.com/2021/07/15/open-source/fsdp/.

[8] Hugging Face Transformers. https://huggingface.co/docs/transformers.

[9] ld.so - Linux manual page. https://man7.org/linux/man-pages/man8/ld.so.8.html.

[10] NVIDIA Collective Communication Library (NCCL). https://developer.nvidia.com/nccl.

[11] Pytorch. https://pytorch.org/.

[12] Kenneth P. Birman. *Building Secure and Reliable Network Applications*. Manning Publications Co, 1996.

[13] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[14] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022.

[15] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, feb 2006.

[16] Sheng Di, Mohamed Slim Bouguerra, Leonardo Bautista-Gomez, and Franck Cappello. Optimization of multi-level checkpoint model for large scale hpc applications. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014.

[17] David Powell (Ed). *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer Berlin, 1991.

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

[19] 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), sep 2002.

[20] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, unjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant gpu clusters for dnn training workloads. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19. USENIX Association, 2019.

[21] Zhiling Lan and Yawei Li. Adaptive fault management of parallel applications for high-performance computing. *IEEE Transactions on Computers*, 57(12), 2008.

[22] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark C. Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, and Carole-Jean W. Cpr: Understanding and improving failure tolerant training for deep learning recommendation with partial recovery. In *Proceedings of the 4th MLSys Conference*, 2021.

[23] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. CheckFreq: Frequent, Fine-Grained DNN checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216. USENIX Association, February 2021.

[24] 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)*, 2020.

[25] OpenAI. Gpt-4 technical report, 2023.

[26] Konstantinos Parasyris, Giorgis Georgakoudis, Leonardo Bautista-Gomez, and Ignacio Laguna. Co-designing multi-level checkpoint restart for mpi applications. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2021.

[27] Aurick Qiao, Bryon Aragam, Bingjing Zhang, and Eric Xing. Fault tolerance in iterative-convergent machine learning. In *Proceedings of the 36th International Conference on Machine Learning*, Proceedings of Machine Learning Research, 2019.

[28] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimization towards training A trillion parameter models. *CoRR*, abs/1910.02054, 2019.

[29] S. Rangarajan, S. Garg, and Yennun Huang. Checkpoints-on-demand with active replication. In *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems*, 1998.

[30] Faisal Shahzad, Markus Wittmann, Thomas Zeiser, Georg Hager, and Gerhard Wellein. An evaluation of different i/o techniques for checkpoint/restart. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, 2013.

[31] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[32] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, Atul Katiyar, Vipul Modi, Vaibhav Sharma, Abhishek Singh, Shreshth Singhal, Kaustubh Welankar, Lu Xun, Ravi Anupindi, Karthik Elangovan, Hasibur Rahman, Zhou Lin, Rahul Seetharaman, Cheng Xu, Eddie Ailijiang, Suresh Krishnappa, and Mark Russinovich. Singularity: Planet-scale, preemptive and elastic scheduling of ai workloads, 2022. https://arxiv.org/abs/2202.07848.

[33] Devesh Tiwari, Saurabh Gupta, James H Rogers, Don E Maxwell, Paolo Rech, Sudharshan S Vazhkudai, Daniel Oliveira, Dave M Londo, Nathan DeBardeleben, Philippe Navaux, Luigi Carro, and Arthur S Buddy Bland. Understanding gpu errors on large-scale hpc systems and the implications for system design and operation. 1 2015.

[34] Devesh Tiwari, Saurabh Gupta, and Sudharshan S. Vazhkudai. Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[35] 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*, 2023.

[36] 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.

[37] Zhao Zhang, Lei Huang, Ruizhu Huang, Weijia Xu, and Daniel S. Katz. Quantifying the impact of memory errors in deep learning. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, 2019.

[38] Yuchen Zhong, Guangming Sheng, Juncheng Liu, Jinhui Yuan, and Chuan Wu. Swift: Expedited failure recovery for large-scale dnn training. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 447–449, New York, NY, USA, 2023. Association for Computing Machinery.

[39] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. Hoplite: Efficient and fault-tolerant collective communication for task-based distributed systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 641–656, New York, NY, USA, 2021. Association for Computing Machinery.