Proactive Fault Tolerance System for Megatron-LM Training

Armaan Ashfaque
University of Illinois - Chicago
Email: aashf@uic.edu

Ragini Kalvade
Univeristy of Illinois - Chicago
Email: rkalv@uic.edu

Niharika Belavadi Shekar
Univeristy of Illinois - Chicago
Email: nbela@uic.edu

***Abstract*—Large Language Models (LLMs) demand extensive computational resources, typically distributed across multiple GPUs and nodes, making training vulnerable to hardware-induced interruptions. Existing checkpointing mechanisms in popular frameworks like NVIDIA Megatron-LM are typically infrequent, covering approximately only 2% of total iterations, leading to significant computational loss upon failures. In this paper, we propose a proactive, automated recovery framework integrated within Megatron-LM, consisting of three core components: (1) a GPU monitoring subsystem utilizing the NVIDIA Management Library (nvidia-smi) to detect anomalies in GPU performance metrics; (2) a strategic checkpointing scheduler that triggers checkpoints based on detected anomalies and elapsed intervals since the last checkpoint; and (3) an automated recovery module that seamlessly resumes training following GPU failures. We quantitatively assess the runtime overhead introduced by our approach to verify minimal disruption to existing workflows. Our methodology demonstrates effective automation and fault tolerance, offering a reusable solution adaptable to other large-scale LLM training frameworks.**

Keywords — Fault Tolerance, Deep Learning Training, Distributed Training, Megatron-LM

I. Introduction

Natural Language Processing (NLP) is advancing quickly in part due to an increase in available compute and dataset size. The abundance of compute and data enables training increasingly larger language models via unsupervised pretraining. By finetuning these pretrained language models on downstream natural language tasks, one can achieve state of the art results.

As these models become larger, they exceed the memory limit of modern processors, and require additional memory management techniques such as activation checkpointing. Several approaches to model parallelism overcome this limit by partitioning the model such that the weights and their associated optimizer state do not need to reside concurrently on the processor. Megatron-LM [1] emerged as NVIDIA's response to the computational challenges of training increasingly large lan- guage models. First introduced in 2019 and continuously updated since, the framework has become a foundational infrastructure for training state-of-the-art LLMs with billions or even trillions of parameters. It overcomes the limitations posed by traditional single-GPU-per-model training by implementing model parallelism with only a few modifications to existing PyTorch transformer implementations. It efficiently trains transformer-based models up to 8.3 billion parameter on 512 NVIDIA V100 GPUs with 8-way model parallelism. The framework's core innovation lies in its implementation of two primary parallelism strategies:

Tensor Parallelism (TP): Partitions individual layers across multiple GPUs, allowing each device to handle a portion of the neural network operations. This approach is particularly effective for transformer-based architectures, where attention heads and feed-forward networks can be naturally divided.

Pipeline Parallelism (PP): Divides the model vertically by assigning different layers to different GPUs. This creates a pipeline where activations flow forward and gradients flow backward across devices, with clever micro-batch scheduling to maximize device utilization.

These parallelism strategies can be combined with traditional Data Parallelism (DP) to efficiently scale training across massive GPU clusters while minimizing communication overhead and memory requirements. Megatron-Core v0.7 has introduced significant improvements to Megatron-LM's training resilience through enhanced checkpointing capabilities. The implementation of Fully Parallel Saving (FPS) allows data-parallel replicas to perform concurrent writes, maximizing file system bandwidth utilization. Additionally, asynchronous checkpointing minimizes training interruptions by first copying model parameters to CPU memory before persisting to stable storage in the background.

Megatron-Core supports the resumption of training with configurations of parallelism of the tensor and pipeline different from those used during the creation of checkpoints, providing valuable flexibility to change hardware resources [2]. Despite these advances, critical limitations remain. Checkpoints are

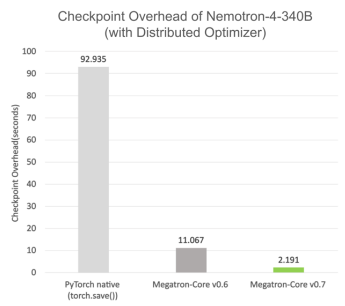

Fig. 1: Reduction in Checkpointing Overhead with Megatron-Core v0.7 as reported by Nvidia

still created only at predetermined intervals. The framework lacks proactive failure detection mechanisms that could trigger checkpoints when hardware issues are imminent. Furthermore, node or GPU failures still require complete training halt and manual intervention to restart the process. In this work, we implement a central system integrated with the current Megtron Core to preemptively identify GPU failures, create a more recent checkpoint, and restart the training after GPU failure. We leverage the new feature of Megatron-Core to allow different parallelism configurations to be executed for the previous saved checkpoints for the recovery mechanism.
For GPU monitoring, we have used commonly measured metrics and are consistently monitoring them. A risk score is evaluated based on outliers of standard GPU metrics during training, and outliers trigger a notification to the emergency checkpoint module. This module checks whether it would be conducive to do a checkpoint. This is mainly to avoid recurrent or unnecessary checkpoints. In case GPU failure occurs, the system has been automated to recalculate the parallelism parameters and resume training from previous checkpoint. Our work aims to decrease the re-training time induced due to GPU hardware failures and reduce downtime post GPU failure. We analyse the results by extrapolating the overhead time due to the extra emergency checkpoints triggered. We analyze worst case scenarios and conclude that benefits outweigh the cost of operation.

II. Literature Survey

We conducted an extensive review across three major topics to ensure comprehensive coverage for our analysis. Initially, we examined various systems similar to Megatron-LM, evaluating their checkpointing mechanisms to understand existing methods and their strengths or limitations. Subsequently, we conducted an in-depth review of Megatron-LM, closely analyzing its checkpointing process, identifying its key features, and clearly defining the scope of our experiment. Lastly, we explored GPU failure prediction methodologies to enhance accuracy in the development and implementation of an effective GPU monitoring module within Megatron-LM.

Analysis of other LLM training frameworks

Developing large language models (LLMs) requires substantial computational resources and advanced frameworks to optimize performance and scalability. Several prominent libraries have emerged, each providing distinctive features for specific training needs.[3]

- **Hugging Face's Transformers:** The Hugging Face Transformers library offers extensive pre-trained model repositories and user-friendly APIs for training, fine-tuning, and deploying transformer-based architectures like BERT, GPT, and RoBERTa. This library significantly reduces the entry barrier for leveraging state-of-the-art models. It focuses primarily on usability rather than large-scale parallelization optimization.
- **DeepSpeed:** Developed by Microsoft, DeepSpeed provides highly optimized training strategies designed for extremely large models. Its Zero Redundancy Optimizer (ZeRO) significantly reduces memory usage, enabling training of models that would otherwise require prohibitively large GPU resources. DeepSpeed effectively addresses scaling limitations and memory bottlenecks encountered with traditional frameworks.
- **FairScale:** Developed by Meta, emphasizes effective model parallelism strategies. It complements PyTorch with efficient techniques for scaling models across distributed systems, utilizing approaches such as pipeline and tensor parallelism. FairScale efficiently handles large model partitions but requires substantial manual tuning to optimize performance.

Megatron-LM stands out for its specific focus on ultra-large model scaling through multi-dimensional parallelism strategies. In comparison, frameworks like Hugging Face Transformers offer easier usability but limited parallel scalability. DeepSpeed provides competitive optimization techniques similar to Megatron, particularly with ZeRO; however, Megatron-LM's detailed parallelism implementation often achieves superior GPU utilization in highly parallel, multi-GPU setups. FairScale offers similar parallelism strategies but lacks the comprehensive optimization seen in Megatron-LM for the largest scales.

LLM Checkpointing

Checkpointing is a critical component of LLM training pipelines, especially as model sizes and training durations scale up. Traditional checkpointing involves periodically saving the full model state, optimizer states, and training metadata. For most LLMs, each token is seen only once by the training process - so training happens in roughly 1 epoch - so checkpoints are done after a certain set number of training steps [4]. This process ensures recovery from hardware or system failures but can become prohibitively slow and storage-intensive for multi-billion parameter models.
Checkpoint Frequency: NVIDIA has recommended checkpointing every 4 hours – this would constitute 0.3% overhead due to checkpointing. [4] PyTorch has introduced optimized checkpointing techniques that reduce I/O bottlenecks by parallelizing data writes and minimizing metadata sync delays. Checkpoints can take several minutes for large models; optimized backends using high-performance file systems (e.g., VAST) reduce this to seconds.
Types of Checkpoints:

- Full Checkpoints: Save model, optimizer, and scheduler states. Essential for precise recovery.
- Partial Checkpoints: Save only model weights, used during intermediate validation.
- Async/Background Checkpoints: Use threads or subpro-

cesses to save without blocking training.

- Sharded Checkpoints: Split state across nodes/devices to parallelize saves and reduce size per file.

Systems like VAST and DeepSpeed ZeRO-Offload employ techniques like incremental checkpointing and memory-aware data streaming. [4]. Research models now explore hybrid storage using both fast-access SSD/NVMe and distributed file systems to reduce latency.

Checkpointing in Megatron-LM

Megatron-LM employs a checkpointing mechanism that periodically saves the state of the model to allow seamless recovery from hardware failures or interruptions. Each checkpoint captures the model weights, optimizer states, learning rate schedulers, and essential metadata needed to resume training without loss of progress. The frequency of checkpointing is configurable via the –save-interval argument, allowing users to control how often snapshots of the training state are saved. Checkpoints are stored in user-specified directories, organized into subfolders labeled by training iteration number (e.g., iter_0000100/). This structured format enables users to easily locate and restore specific checkpoints corresponding to past training states. Significant enhancements to this mechanism were introduced in Megatron-Core v0.7, most notably through the support for Fully Parallel Saving (FPS) and Asynchronous Saving. FPS allows data-parallel replicas to write their portions of the model concurrently, maximizing I/O throughput and better utilizing filesystem bandwidth. Asynchronous saving further improves performance by first copying model parameters to CPU memory (or in future, local storage), and then writing them to disk in the background—thereby minimizing training stalls during checkpointing. Megatron-LM [2] normally uses distributed checkpointing, where each save iteration produces .distcp files (partitioned model states), a shared common.pt file (containing shared or global information), and a metadata.json file (describing the checkpoint layout). For workflows requiring portability or simplicity, Megatron also provides utilities to convert between distributed and non-distributed checkpoint formats. Megatron-LM also supports activation checkpointing with recomputation [5], a memory optimization technique widely used in large-scale deep learning training. Rather than storing all intermediate activations during the forward pass-as is done in standard backpropagation—only a subset of activations at designated checkpoints within the model are retained in memory. The remaining activations are discarded and later recomputed on-the-fly during the backward pass as needed.

GPU monitoring

There has been substantial research interest in developing proactive GPU monitoring and failure prediction systems that leverage telemetry data, machine learning techniques, and workload characteristics.

Early studies emphasized the classification of GPU failure types using hardware-level metrics. Bin Nie [6] systematically classified failures into soft errors (e.g. transient memory faults), hard errors (e.g. permanent hardware defects) and thermal degradation. Using CUDA profiling tools and low-level performance counters, the authors showed that thermally intensive or memory-bound kernels are more likely to induce failure over time. Their findings also revealed that temperature and power cycling, particularly under prolonged workloads, accelerates wear-out mechanisms, forming a critical foundation for later predictive modeling work.

In January 2022, Heting Liu [7] telemetry-driven machine learning pipeline to predict GPU failures at scale using internal fleet data. Their approach uses metrics such as thermal variance, memory errors, power draw, and utilization logs to train a multilayer perceptron capable of predicting failures within short lead times. The system achieved high precision (up to 94%) using class balancing techniques to address the data imbalance problem inherent in rare failure events. Their study demonstrates the feasibility of real-time, data-driven failure forecasting in production environments and underscores the value of granular telemetry in proactive infrastructure management.

GPU workload characteristics themselves have also been found to influence failure rates significantly. GPU failure prediction converges on three main pillars: telemetry analysis, interpretable machine learning, and workload-aware adaptation. Telemetry-based models provide high predictive accuracy, especially when incorporating error rates, thermal profiles, and power fluctuations. AI techniques enhance transparency and practical usability. Finally, deep learning workloads introduce unique stress patterns that must be integrated into scheduling and monitoring strategies.

Based on our research, we concluded GPU failure prediction models will require data well beyond the grasp of our resources and project duration. We decided to focus our efforts on implementing a module to showcase how the methodology can improve LLM training fault tolerance while leaving room for improvement in failure detection. We have described our technique for failure detection in the following sections.

III. System Design

A. Architecture Overview

The proposed fault tolerance system consists of three main components. First, the master node acts as the central controller, responsible for launching and managing all other processes. Second, a GPU monitoring script continuously tracks GPU health metrics and alerts the training process if a potential failure is detected. Finally, the recovery mechanism kicks in when training is interrupted due to hardware failure and automatically resumes the job using the remaining available resources.

B. Master Node

The master node is the only component the user needs to launch to bring the entire system online. It accepts paths to the monitoring script, the recovery module, and the training command—along with any arguments that would normally be used to run it manually. These script paths are not hardcoded,

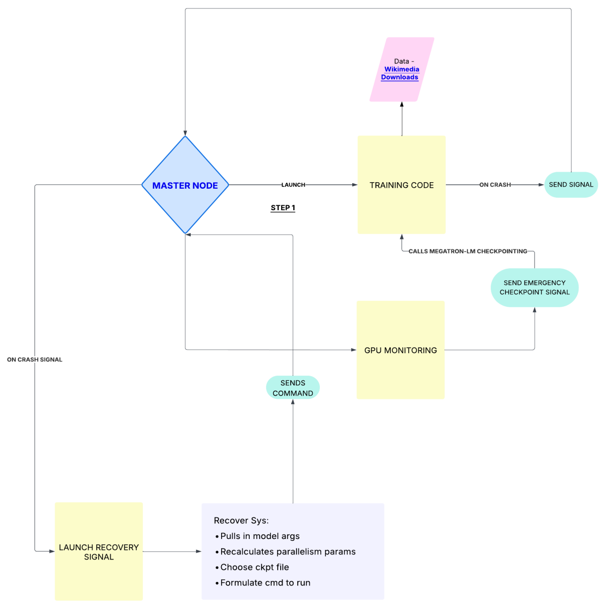

Fig. 2: Process Flow

Metric	Description
`timestamp`	Allows time-based correlation with training logs
`gpu_id`	Identifies which device reported the anomaly
`gpu_name`	Useful for multi-GPU, heterogeneous environments
`driver_version`	Tracks compatibility or known issues
`temperature`	High temperatures may precede thermal throttling or shutdown
`power_draw`	Sudden fluctuations may indicate instability or overdraw
`gpu_utilization`	Unexpected drops can signal idling or crash
`memory_utilization`	Spikes may indicate memory overload or leaks
`memory_used`, `memory_total`	Used to calculate headroom and memory pressure
`errors`	Reserved for future integration with ECC or critical error logs

TABLE I: Monitored GPU Metrics and Their Description

keeping the system modular and flexible. This allows users to plug in their own monitoring or recovery logic for Megatron-LM, if desired. The standard workflow begins with the master node launching the GPU monitoring and training scripts in parallel. The arguments provided for training are stored by the master node so they can later be reused by the recovery mechanism. If training is interrupted due to hardware failure, the recovery module generates a new training command, which the master node then uses to resume training from the most recent checkpoint.

C. GPU Monitoring

To proactively detect early signs of GPU degradation or potential failure, we designed a real-time monitoring subsystem that continuously samples and logs critical GPU performance metrics. This monitoring mechanism operates in the background during training, capturing data at fixed intervals and evaluating it to compute a risk score that reflects the GPU's health. When this risk score exceeds a specified threshold (set to **0.95** in our implementation), a signal is sent to the checkpointing module to initiate emergency checkpointing and recovery procedures.

- **Monitored Metrics and Rationale:** Our collector script uses NVIDIA's `nvidia-smi` utility to query low-level GPU statistics at 10-second intervals. The following metrics are logged to a CSV file (gpu_metrics.csv) in the gpu_logs directory. As detailed in Table I, we selected metrics that most directly correlate with early signs of GPU instability and failure.
- **Failure Risk Score Computation:** Rather than relying solely on static thresholds, our system computes a *composite risk score* dynamically for each GPU. This score is based on six risk factors: temperature, memory usage, presence of errors, power fluctuation, GPU utilization drop, and temperature rise rate.
  Each risk factor contributes a normalized value between 0 and 1, and the final risk score is computed using a weighted sum:

$$\text{Risk Score} = \sum_{i=1}^{n} w_i \cdot r_i$$

  where w_i is the weight assigned to the i^{th} risk component, and r_i is the corresponding normalized metric. The weights used are:

```
weights = {
    'temperature': 0.25,
    'memory': 0.15,
    'errors': 0.25,
    'power': 0.10,
    'utilization_drop': 0.15,
    'temp_rise': 0.10
}
```

  If the computed score exceeds **0.95**, an emergency flag file (trigger_checkpoint.flag) is generated, prompting the training system to trigger an emergency checkpoint.
- **Runtime Monitoring Loop:** The GPU monitoring script runs in a separate daemon thread and maintains a rolling window of the last 60 samples for each GPU. This window is critical for computing baselines (e.g., average utilization) and detecting short-term trends like rapid temperature rise or power fluctuations. If an exception occurs during metric collection (e.g., due to driver issues or temporary I/O failures), the loop logs the error and retries after a short delay.
  By integrating this monitoring system with Megatron-LM's training loop, we ensure that critical anomalies

are caught in near real-time and training progress is safeguarded through timely checkpoints.

D. *Emergency Checkpoint Strategy*

An **emergency checkpoint** is a standard model checkpoint saved outside regular intervals, specifically triggered in response to detected GPU anomalies. Unlike fixed-interval checkpoints, emergency checkpoints are proactively initiated when the GPU monitoring system detects a high risk of failure (risk score ≥ 0.95).

- **Trigger-Based Checkpointing Workflow:** Once the monitoring module (Section III-C) generates a trigger_checkpoint.flag file, a shell listener script continuously running in the background performs the following actions:
  - Checks every 30 seconds if the trigger flag exists.
  - If found, creates save_now.flag and deletes the trigger file.
  - Megatron-LM's training script is modified to respond to this emergency flag.
- **Smart Emergency Checkpoint Filtering:** To avoid redundant checkpointing during frequent GPU alerts, we introduce a filtering mechanism based on a configurable checkpoint gap (default 25% of save_interval). Emergency checkpoints are saved only if the current iteration is at least 25% ahead of the last checkpoint. For instance, if save_interval=100, and a checkpoint was saved at iteration 1000, the next emergency save will only happen at or after iteration 1025.
  This logic prevents excessive checkpointing in the presence of multiple consecutive alerts and ensures a balance between responsiveness and overhead.
- **Modified Checkpoint Condition:** The original checkpoint condition:

```
if args.save and (iteration != 0 and iteration %
      args.save_interval == 0):
    save_checkpoint(...)
```

  is updated to include both flag checking and 25% gap enforcement, keeping logic consolidated in a single block:

```
def should_emergency_save(iteration,
      last_checkpoint_iter, save_interval,
      threshold_frac=0.25):
    if not os.path.exists('/workspace/megatron2/
        save_now.flag'):
        return False
    gap = iteration - last_checkpoint_iter
    return gap >= int(threshold_frac * save_interval)

if args.save and (
    iteration % args.save_interval == 0 or
    should_emergency_save(iteration,
        last_checkpoint_iter, args.save_interval)):

    save_checkpoint(...)
    last_checkpoint_iter = iteration

    if os.path.exists('/workspace/megatron2/save_now.
        flag'):
        os.remove('/workspace/megatron2/save_now.flag'
            )
```

- **Shell Script Integration:** During training, a background process watches for GPU alerts:

```
while true; do
    if [ -f "/workspace/megatron2/trigger_checkpoint.
        flag" ]; then
        echo "GPU alert detected  triggering
            checkpoint!"
        touch /workspace/megatron2/save_now.flag
        rm /workspace/megatron2/trigger_checkpoint.
            flag
    fi
    sleep 30
done
```

- **Summary of Workflow:** Table II outlines each component's role in the emergency checkpoint pipeline.

Component	Role
GPUMetricsCollector	Computes risk scores, creates trigger flag
Shell Listener	Detects trigger, creates save flag
pretrain_gpt.py	Reads save flag, performs checkpoint if threshold gap is met
Checkpoint File	Stored using standard Megatron format

TABLE II: Emergency Checkpoint Workflow Components

This mechanism ensures seamless checkpointing when training is at risk of interruption, minimizing loss and improving recovery efficiency without manual intervention. The 25% threshold is user-configurable and can be tuned (e.g., 50%) depending on model size, training frequency, or system constraints.

E. *Recovery Mechanism*

The automated recovery mechanism was developed in response to Megatron-Core v0.7's new capability of loading checkpoints using different parallelism configurations. This allows the system to dynamically adjust tensor and pipeline parallelism settings after a crash, based on the set of GPUs that remain available. As a result, training can resume without manual intervention even in degraded resource scenarios.

The mechanism also accepts a user-defined argument that determines the recovery policy: whether to wait for all GPUs to come back online before restarting, or to recover immediately using the currently available resources.

When recovering with fewer GPUs, the system attempts to recompute the optimal parallelism configuration based on the following criteria:

1) Maximize parallelism such that:

$$\text{TP} \times \text{PP} = \text{Number of Available GPUs}$$

Data parallelism is then automatically calculated as:

$$\text{DP} = \frac{\text{Number of Available GPUs}}{\text{TP} \times \text{PP}}$$

Note: While the first condition is not strictly required, Megatron-LM mandates that the following constraint must be satisfied for training to function correctly:

$$\text{TP} \times \text{PP} \times \text{DP} = \text{Number of Available GPUs}$$

2) Preference is given to maximizing **Tensor Parallelism (TP)** while ensuring it satisfies:

$$\text{Number of Attention Heads} \mod \text{TP} = 0$$

3) **Pipeline Parallelism (PP)** is maximized next, under the constraint:

$$\text{Number of Available GPUs} \mod \text{PP} = 0$$

F. Hardware and Megatron-LM Setup

The development and testing of our system, was done on machines leased from Chameleon Cloud. Our setup included a single node with four NVIDIA V100 GPUs (32GB each), which we found to be a good balance between availability and performance.

Chameleon allows standard leases for up to a week, and getting access to GPU-equipped nodes—especially ones with V100s—can be quite competitive. Although NVIDIA A100s were the most powerful GPUs available on the platform, we weren't able to find an open slot to lease them during our timeline. P100s were easier to access, but we ran into versioning issues when trying to use Megatron-LM on them. Thus, we ultimately settled on the V100s.

All of our development and testing had to fit within the lease durations, which added a bit of time pressure to the project.

To get Megatron-LM running smoothly, we used NVIDIA's PyTorch container (NGC version 24.10). This container is optimized for GPU acceleration and comes with a validated set of libraries that help ensure compatibility and maximize performance. To use it, we first had to install Docker and the NVIDIA Container Toolkit on our machine

G. LLM Training Setup

1) **Dataset**: For training, we used a subset of the Wikipedia articles dump[1], with a file size of approximately 537MB, freely available for download. This represents only a small portion of the full Wikipedia dump, which totals around 22GB. We deliberately chose not to use the entire dataset, as our primary goal was not to evaluate the performance of the language model itself, but rather to develop and test our fault tolerance system around it. Additionally, the experimental data we aimed to collect was independent of the dataset size. Using a smaller dataset allowed us to save time and resources while still effectively validating our system. The Wikipedia dump is first extracted using the wikiextractor.py tool [2]. After extraction, the raw text is cleaned and preprocessed using the preprocess_data.py script provided by the Megatron-LM repository. This script handles tokenization and formats the data according to the requirements of the specific model being trained.
   The final processed data is stored as a pair of files: output_prefix.bin and output_prefix.idx. To use this dataset during training, you simply pass the /path/to/output_prefix (without the file extensions) to the Megatron-LM training scripts, which then load the data accordingly.

2) **Model Details**: Megatron-LM supports training several well-known public LLM architectures, including GPT-3 [8], BERT [9], , Mixtral [10] and so on. For our purposes, we focused exclusively on training different sizes of GPT-3 models. Since our primary interest was in evaluating checkpointing speed, overhead, and recovery performance under various model parallelism configurations, we treated model size as the key differentiating factor. Testing across different model architectures was not a priority, as Megatron-LM's underlying checkpointing and parallelism mechanisms remain consistent across supported models.
   Our experiments involved four GPT-3 model sizes: 345M, 857M, 1.7B, and 7.1B parameters. We configured both tensor parallelism and pipeline parallelism to 2, enabling maximum model parallelism on our 4×V100 GPU setup. The 7.1B model represents the upper limit of what can fit into memory on this hardware, fully utilizing the available GPU memory and achieving 100% GPU utilization across all devices. The 345M parameter model, was primarily used for quick testing and validation.

IV. Results and Evaluation

A. Failure Prediction Accuracy

Accurately quantifying the precision or recall of our GPU failure prediction logic was infeasible due to the unavailability of publicly accessible, real-world GPU failure datasets. While we designed our system around well-established GPU telemetry indicators (e.g., temperature, power draw, utilization drops), validating our approach with ground truth failure events remains an open challenge. Notably, the study by Liu et al. (2022) was the first to systematically explore GPU failure prediction under deep learning workloads at scale, using an internal production dataset of 350 million entries collected from ByteDance datacenters. However, this dataset is proprietary and has not been released publicly, preventing independent replication or benchmarking of such methods [11]. Consequently, although we implemented a real-time monitoring system and computed risk scores based on relevant metrics, we cannot report accuracy or precision and recall due to the lack of labeled failure data. Our architecture remains extensible, should such data become available in the future.

B. Checkpoint Overhead

The primary overhead introduced by our system arises from the additional checkpoints created in response to predicted GPU failures. This overhead manifests mainly as an increase in total training time. To quantify this, we perform a worst-case analysis: we assume the GPU monitoring system continuously predicts a high risk of failure, prompting frequent checkpointing, even though no actual failures occur. This scenario results in the maximum possible number of checkpoints being created while training proceeds uninterrupted.

We adopt this approach because genuine GPU failures—especially under sustained load—are relatively rare in

[1]https://dumps.wikimedia.org/enwiki/latest/
[2]https://github.com/attardi/wikiextractor

small-scale setups, such as our system with only four GPUs. Capturing a statistically meaningful number of organic failure events would require running long-term simulations over months or even years. Hence, a worst-case evaluation offers a practical upper bound on the system's overhead.

In our experiment, we evaluate the overhead introduced by our system using filtering gaps set to 50% (resulting in twice as many checkpoints) and 25% (resulting in four times as many checkpoints) of the regular save interval.

Model Size	Baseline Ckpt Time (s)	W/ 50% Gap (s)	W/ 25% Gap (s)
857M	476	952	1904
1.7B	1624	3248	6496
7.1B	7358	14716	29432

TABLE III: These values are estimated by multiplying the average checkpoint time for each model size by the total number of checkpoints created during training. For the 175B parameter GPT-3 model, training spans 500,000 iterations with a save interval of 10,000, resulting in 50 checkpoints in the baseline case. With emergency checkpointing enabled, this increases to 100 checkpoints for a 50% gap and 200 for a 25% gap.

As shown in the results, checkpointing overhead becomes increasingly significant with larger model sizes—reaching nearly 4 hours for the 7.1B parameter model. Extrapolating from this, training 100B+ models could result in worst-case checkpointing overheads spanning over a full day, which would be impractical and expensive. Such overheads could make the system difficult to integrate into production workflows. That said, this represents a worst-case scenario, and in practice, such a high number of full checkpoints would rarely be created during typical training runs.

While this level of overhead would normally be a cause for concern—especially with larger models—recent updates in Megatron-LM help mitigate the problem. The introduction of asynchronous checkpointing allows checkpoints to be saved in the background without pausing training, significantly reducing overhead. In addition, ongoing research in overlapping checkpointing with training steps aims to further optimize this process, making it increasingly feasible to use our system even with much larger models.

To evaluate the impact of Megatron-LM's asynchronous checkpointing, we conducted a small-scale experiment using the 857M parameter model. We ran two training jobs for 20 iterations—one with a single checkpoint and another with four checkpoints. The difference in total training time between the two runs was under 5 minutes, which is a very encouraging result and highlights the effectiveness of asynchronous saving in reducing checkpointing overhead. Therefore, even in the worst-case scenario, with asynchronous saves enabled, time overhead is no longer a major concern. Instead, storage requirements become the more prominent limiting factor that needs to be addressed.

C. Limitations

- The most significant limitation of this system is the lack of publicly available, high-quality GPU failure and health data, which restricts the development of accurate prediction models for proactive fault detection.
- Additionally, GPU monitoring models need to be tailored to specific hardware and system configurations, meaning they must be trained or calibrated individually to produce reliable results in different environments.
- Finally, limited access to GPUs during development prevented us from conducting larger-scale or more realistic experiments that would better reflect production scenarios.

D. Conclusion and Future Work

The main contribution of this work is the development of a modular, fault-tolerant framework that integrates with large-scale, multi-GPU training systems like Megatron-LM. Our system leverages recent advances in model training and checkpointing to automatically trigger checkpoints when a risk of hardware failure is detected, minimizing potential loss of training progress. In the event of failure, the built-in recovery mechanism resumes training using the available resources by taking advantage of Megatron-LM's support for flexible parallelism reconfiguration across checkpoints. Future work could involve integrating more sophisticated GPU failure prediction models, extending support to other training frameworks.

References

[1] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv*, 2020.

[2] NVIDIA, "Train generative ai models more efficiently with new nvidia megatron core functionalities," https://developer.nvidia.com/blog/train-generative-ai-models-more-efficiently-with-new-nvidia-megatron-core-functional 2024.

[3] ——, "Key libraries for developing large language models (llms)," https://medium.com/@sirjanabhatta6/key-libraries-for-developing-large-language-models-llms-60a740906bd6/, 2024.

[4] Subramanian Kartik, PhD, Global Systems Engineering Lead and Colleen Tartow, PhD, Field CTO and Head of Strategy, "A checkpoint on checkpoints in llms," https://www.vastdata.com/blog/a-checkpoint-on-checkpoints-in-llms/, 2024.

[5] V. A. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro, "Reducing activation recomputation in large transformer models," *Proceedings of Machine Learning and Systems*, vol. 5, pp. 341–353, 2023.

[6] S. G. T. P. C. E. E. S. D. T. Bin Nie, Ji Xue, "Machine learning models for gpu error prediction in a large scale hpc system," *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018.

[7] C. T. R. Y. G. C. Z. L. C. G. Heting Liu, Zhichao Li, "Prediction of gpu failures under deep learning workloads," *arXiv:2201.11853v1*, 2022.

[8] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 2019, pp. 4171–4186.

[10] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand *et al.*, "Mixtral of experts," *arXiv preprint arXiv:2401.04088*, 2024.

[11] H. Liu, Z. Li, C. Tan, R. Yang, G. Cao, Z. Liu, and C. Guo, "Prediction of gpu failures under deep learning workloads," *arXiv preprint arXiv:2201.11853*, 2022.

Appendix

A. Contributions

- **Armaan Ashfaque:** Contributed to the architecture design of the fault tolerance system and implemented the master node responsible for coordinating GPU monitoring, training, and automated recovery. He set up and validated Megatron-LM in a multi-GPU environment on Chameleon Cloud. Also, he developed the logging infrastructure and scripts for capturing runtime metrics and training timelines under various failure scenarios, and collected experimental data for system overhead analysis.
- **Ragini Kalvade:** Developed the recovery mechanism to dynamically reconfigure tensor and pipeline parallelism based on available GPUs during failure recovery. She contributed to the checkpointing strategy by implementing emergency flag detection and automating Megatron-compatible checkpoint triggers. She also researched distributed training recovery techniques and designed the logic to regenerate launch commands post-failure. Furthermore, she conducted extensive testing of recovery workflows under simulated GPU failure conditions to validate system resilience.
- **Niharika Belavadi Shekar:** Designed and implemented the GPU monitoring subsystem, which periodically collects low-level GPU metrics, computes dynamic risk scores, and signals checkpoint triggers during anomaly detection, after conducting targeted research on GPU failure prediction techniques. She led the integration of threshold-based emergency checkpoint filtering, including the 25% iteration-gap logic. Modified the Megatron-LM training loop to support conditional checkpointing based on real-time monitoring. Also ensured synchronization between the monitoring module and Megatron's checkpointing hooks.
- All documentation was divided equally amongst all three group members.

B. Codebase

Our complete system implementation is available here https://github.com/armaan10/Megatron-LM/tree/Gpu_metrics_and_recovery. The repository is a fork of the original Megatron-LM project, with our modifications located on the Gpu_metrics_and_recovery branch. For detailed instructions on navigating the codebase, please refer to the README section titled "Proactive Fault Tolerant System".