

GEM: GPU Erasure-Coded Memory for Fast Training Recovery

Ronglong Wu[1,2], *Huayu Zeng*[3], *Xinbin Hu*[1,2], *Shuyue Zhou*[1,2], *Zhirong Shen*[1,2]*, *Kunlin Yang*, *Jiwu Shu*[1], *and Yiming Zhang*[4,1]

[1]*Xiamen Key Laboratory of Intelligent Storage and Computing, Xiamen University*

[2]*Fujian Engineering Research Center of High-Performance Intelligent Computing Systems*

[3] *Institute of Artificial Intelligence, Xiamen University* [4]*Shanghai Jiao Tong University*

Abstract

Training large deep learning models typically requires a massive number of GPUs operating continuously for weeks or even months. Unfortunately, GPU memory failures are not uncommon in such long-running jobs. When these failures occur, existing solutions reload the latest checkpoint from CPU memory or remote persistent storage to resume training, incurring substantial storage overhead and expensive network transfers.

This paper presents GEM, a lightweight fault-tolerant mechanism that introduces GPU Erasure-coded Memory for fast recovery from GPU memory failures. GEM first employs *selective* erasure coding (EC) to protect critical model states instead of relying on periodic checkpointing. Our insight is that these critical states (including parameters and optimizer states) are updated infrequently, allowing them to be protected with EC in a computation-/storage-efficient manner. GEM adopts a wide-stripe EC design to minimize recovery redundancy, enabling redundancy to reside entirely in GPU memory and avoiding additional data transfers. Furthermore, GEM incorporates a mixed-granularity retry mechanism to recover the remaining frequently updated data. We conduct extensive experiments on NVIDIA GPUs with real-world deep learning models and show that GEM reduces wasted time caused by GPU memory failures by an average of 96.6%, while incurring only 0.3% additional GPU memory overhead.

1 Introduction

Background: Deep learning models (including GPT-4 [40], LLaMA3 [27], and PaLM [5]) have gained widespread adoption due to their remarkable performance across diverse tasks, such as computer vision [3, 61, 65] and natural language processing [4, 10]. Training these large models typically requires thousands of GPUs [5, 27] running over a long period of time—often spanning several weeks or even months [64, 70]. However, large-scale computing clusters are susceptible to failures that can disrupt the training process. For example, the training of the OPT-175B model was reported to encounter two failures per day [70], while the LLaMA3-405B model experiences seven failures daily [27]. Such failures stall the training process and significantly reduce training efficiency.

Among various failure types, GPU memory failures emerge as a critical issue for large-scale training [27, 66]. This is primarily because modern GPUs typically use *high-bandwidth memory* (HBM), which features a stacked architecture and weaker *error correction codes* (ECC), making it more prone to failures [66]. For example, GPU memory accounts for 64.4% of infrastructure incidents in Azure deployments [67]. Similar symptoms are observed at Meta [27], where HBM failures are reported as the second-largest cause of training interruptions. Currently, most training systems adopt checkpointing [9,28,30,59,62] for failure recovery. However, checkpointing is not tailored to GPU memory failures and thus suffers from the following performance inefficiencies (see §2.4). First, checkpointing to persistent storage is limited by the low storage bandwidth, which slows down the training process (up to 43% [25]). Second, checkpointing to CPU memory interferes with CPU–GPU communication [63], thereby degrading training efficiency. For example, overlapping checkpointing with training reduces throughput (e.g., by about 11% in [44]) because the two operations share the same communication channel.

Unsuccessful attempts and findings: Motivated by this observation, we evaluate two preliminary attempts that store redundant data directly in GPU memory for failure recovery (§3). Both attempts reconstruct lost data using this in-GPU redundancy, thereby avoiding costly storage I/O and CPU–GPU data transfers. Our first attempt adopts retry-based recovery, which discards the affected intermediate computation and re-executes the current training step from preserved critical model states. To support retry, we maintain replicas of the model parameters and optimizer states (i.e., critical states) required to re-execute the step. However, maintaining these replicas incurs 73.4% additional GPU memory overhead and can trigger out-of-memory errors. Our second attempt re-

*Corresponding author: Zhirong Shen (shenzr@xmu.edu.cn)

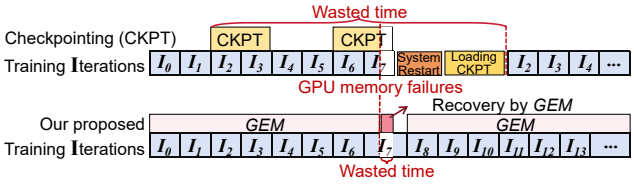


Figure 1: **GEM (EC + retry) vs. checkpointing.** *GEM reduces wasted time from multiple iterations to less than one.*

places full replication with *erasure coding* (EC) to reduce redundancy overhead. Although EC incurs only 1.6% additional memory overhead and encodes 12 GB of data in 0.04 s, it remains inefficient for frequently updated states. Each update requires the associated parity chunks to be recomputed, reducing training efficiency to 76.7% of the baseline without EC. Hence, *achieving both low memory overhead and high training efficiency during recovery from unexpected GPU memory failures remains challenging.*

Our solution: This paper presents GEM, which combines *selective* EC with a flexible retry mechanism to enable fast recovery from GPU memory failures. GEM specifically targets GPU memory failures, a major source of training interruptions. Compared with conventional checkpointing, GEM recovers more efficiently from such failures while remaining complementary to existing checkpointing mechanisms. Figure 1 compares the recovery processes of GEM and checkpointing after a GPU memory failure during iteration I_7. Because the second checkpoint is incomplete, the checkpointing system must reload the first checkpoint and resume from I_2, wasting approximately 10 iterations (six for retraining and four for system restart and checkpoint reload). In contrast, GEM protects critical model states with EC and preserves the inputs required to retry each iteration, allowing recovery within the failed iteration I_7. As a result, GEM reduces *wasted time*—the time that does not contribute to training progress—from approximately 10 iterations to less than one iteration. This improvement stems from storing EC redundancy directly in GPU memory, which avoids costly storage I/O and CPU–GPU transfers during recovery. Because GEM targets GPU memory failures rather than all GPU failures, it complements checkpointing (Exp#11 in §7.2), which remains necessary for failures outside its scope (e.g., faulty GPU).

Specifically, GEM employs *wide-stripe* EC [15], which uses a large number of data chunks per stripe to encode data that is updated less frequently. To identify these data segments, we perform an analysis and find that critical model states [1] are updated less frequently than non-critical model states, making them suitable for EC. Moreover, since non-critical states can be recovered via retry using the critical model states as input, GEM introduces a mixed-granularity retry mechanism. It supports retrying at either the training-phase level (e.g., forward pass) or the training-step level, thereby balancing the memory overhead of storing replicated training inputs with the computational overhead of retraining. In addition, GEM incorporates a hash-based directory to efficiently identify the appropriate recovery strategy for each data segment upon failure.

Performance: We implement and evaluate GEM on an NVIDIA GPU platform. The results show that GEM preserves training consistency during GPU memory failure recovery while incurring only 0.3% memory overhead and 0.5% runtime overhead. We further demonstrate that GEM reduces GPU memory failure recovery time by 99.4–99.7% compared with representative checkpoint-based methods, including CheckFreq [30] and Gemini [62].

To summarize, we make the following contributions.

- This work demonstrates that EC is a storage-/computation-efficient mechanism for recovering from GPU memory failures using accessible data within the GPU (§3).
- We propose a fault-tolerant mechanism that employs selective EC (GemEC) to protect critical model states, which are updated less frequently, thereby enabling fast training recovery from GPU memory failures (§4.2).
- We introduce a mixed-granularity retry mechanism (GemRetry) to ensure reliability of frequently updated data (i.e., non-critical model states) (§4.3).
- We implement GEM and evaluate its effectiveness on various large foundation models and under different parallelism configurations (§7).

We have released the source code of GEM at https://github.com/wrl297/gem.

2 Background and Limitations

2.1 Distributed Training Systems

As deep learning requires massive amounts of data for long-term training, existing studies commonly distribute training workloads across large-scale GPU clusters (e.g., 992 GPUs for training OPT [70]). However, data transmission across nodes is significantly slower than communication within a single node, making efficient coordination across GPUs a critical challenge. To address this, various parallelization strategies have been proposed, each with different trade-offs in memory usage, communication overhead, and computational efficiency. Three representative strategies are: (i) data parallelism, which distributes training data across workers [47, 71]; (ii) tensor parallelism, which partitions the parameter tensors of individual layers across multiple workers [33, 53]; and (iii) pipeline parallelism, which divides model layers into stages, with each stage executed on a different worker [32, 45].

2.2 Training Failures

Failures in training. Frequent failures are observed in large-scale distributed training, primarily due to the use of a large

[1] Critical model states refer to those stored by existing checkpointing systems [9, 30, 44, 55] for failure recovery.

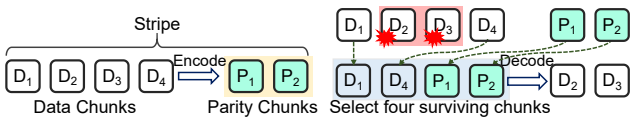


Figure 2: **Example of a (6,4)-code EC (§2.3).**

number of GPUs, which substantially increases the likelihood of failures. For example, in a 384-GPU training setup, 1–2 GPU failures are reported per week [64]. The failure rate scales with system size: training on 992 GPUs incurs about two failures per day [70], while a 16,384-GPU platform may experience up to seven failures per day [27].

GPU memory failures. GPU memory failures are particularly concerning because HBM is widely used in modern GPUs [36, 37, 66]. For example, GPU memory accounts for 64.4% of infrastructure incidents in Azure [67]. Similar trends have been observed in other HBM-equipped accelerators (e.g., HBM faults account for 44% of NPU failures [74]). The reason is that HBM's complex 3D-stacked architecture and limited ECC capability introduce distinct fault modes and increase the failure rate [31, 39, 66]. In particular, HBM faults often exhibit row- or column-level locality [66], affecting only a small portion of the model state. However, existing training systems rely on generic recovery mechanisms, such as checkpointing [30, 44, 62], without exploiting this fault locality. They therefore incur substantial recovery overhead and waste considerable training time. To reduce fault-induced disruption, this paper develops a recovery strategy tailored to GPU memory failures. Our evaluation shows that it complements existing general-purpose fault-tolerance mechanisms to further reduce recovery time (Exp#11 in §7.2). Moreover, it can also be extended to handle silent data corruption and failures during inference (§5).

2.3 Failure Recovery

Failure recovery in model training. Checkpointing [9, 30] is a widely adopted approach to cope with training failures. It periodically saves model states, including parameters and optimizer states, to remote persistent storage [9, 30, 55] or CPU memory [44, 62]. Upon a failure, it restores the most recently saved states to resume training. This introduces a trade-off between checkpointing frequency and wasted time: more frequent checkpointing reduces recovery time in model training, but at the cost of higher bandwidth consumption. Given the large and rapidly growing model sizes, saving a full checkpoint to persistent storage or reloading it is often constrained by storage bandwidth (e.g., PaLM requires tens of minutes [62]). Extensive efforts have been made to improve checkpointing efficiency, including differential checkpointing [9] (i.e., checkpointing only the modified parts), storing checkpoints in CPU memory [62] (as opposed to persistent storage), and enabling efficient checkpoint resharding [59].

Since workers in the same data-parallel rank hold identical

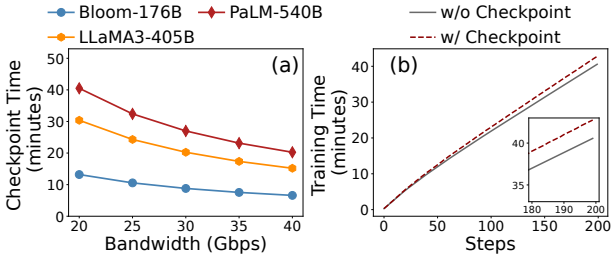


Figure 3: **Checkpointing time and its impact on training (§2.4).** *(a) Checkpoint time across various disk bandwidths and (b) training time without (w/o) and with (w/) checkpointing stored in CPU memory.*

replicas of model states, Swift [73] and JITCKPT [12] leverage these replicas for failure recovery. Alternatively, some works introduce GPU operating systems [63] or device-proxy clients [54] to achieve OS-level checkpointing and restoration.

Erasure coding. Erasure coding (EC) is a storage-efficient means for failure recovery in distributed storage systems. It is defined by two parameters, k and m, where a $(k+m,k)$-code encodes k fixed-size data chunks to generate $k+m$ coded chunks using finite field arithmetic [52]. These $k+m$ chunks form a *stripe* and enable recovery of the k original chunks from up to m chunk failures. While providing fault tolerance, EC stores m additional parity chunks per stripe, introducing storage redundancy (i.e., $\frac{m}{k}$). To reduce overhead while maintaining fault tolerance, *wide-stripe* EC [15] uses large k values to achieve ultra-low redundancy, attaining the same fault tolerance degree (i.e., m). Figure 2 shows a $(6,4)$-code EC, where four data chunks $\{D_1, D_2, D_3, D_4\}$ generate two parity chunks $\{P_1, P_2\}$. The resulting six chunks form a stripe that tolerates any two chunk failures (e.g., D_2 and D_3).

2.4 Limitations of Existing Efforts

Efficiency depends on storage bandwidth. Existing checkpointing mechanisms [9, 28, 30, 59] typically store model states in persistent storage, incurring significant communication overhead. Thus, their efficiency depends largely on the available storage bandwidth. We analyze the time required to store checkpoints for three models of different sizes: 176B Bloom [64], 405B LLaMA3 [27], and 540B PaLM [5]. We vary storage bandwidth from 20 Gbps to 40 Gbps, which is consistent with datacenter SSDs [19, 62]. Figure 3(a) shows that the time required for checkpointing roughly doubles as bandwidth decreases from 40 Gbps to 20 Gbps, reaching up to 44 minutes. This restricts checkpoint frequency, leading to high recovery costs. Given that large-scale training often uses thousands of GPUs and can encounter hundreds of failures, slowdowns can reach up to 43% [25, 27].

Interference with CPU–GPU communication. To improve checkpointing efficiency, prior studies [44, 55, 62] store checkpoints in CPU memory. However, this design introduces con-

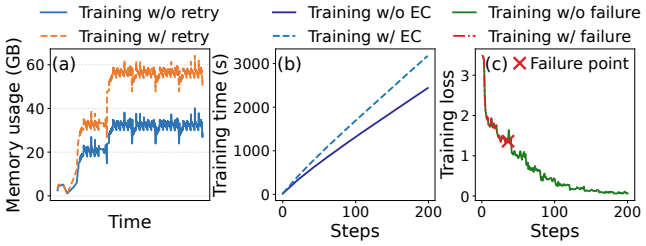


Figure 4: **Unsuccessful attempts to maintain redundancy in GPU memory for failure recovery (§3).** *(a) Memory usage without (w/o) and with (w/) retry; (b) training time w/o and w/ EC; (c) training loss w/o failures and w/ failures when enabling EC.*

tention on the CPU–GPU communication channel, degrading overall training performance. Figure 3(b) reports the training time of a 3B LLaMA model on an A100 platform under mixed-precision training (BF16 parameters, activations, and gradients; FP32 optimizer states). We compare runs with and without CPU-memory checkpointing. Checkpointing to CPU memory increases training time by 5.4%. Although this overhead appears modest, it translates to more than one million wasted GPU hours in a 54-day LLaMA training run on a 16K-GPU cluster [27]. Although Gemini [62] interleaves model-saving traffic with training, it still incurs high serialization overhead during recovery. Moreover, recent works [42,49,50] offload data to CPU memory to reduce GPU memory pressure, which exacerbates communication interference and further degrades training efficiency.

3 Unsuccessful Attempts and Findings

To address these limitations, a straightforward approach is to store recovery redundancy in GPU memory, thereby avoiding additional data transfers between GPU memory and CPU memory or persistent storage. Driven by this insight, we carry out two attempts to store redundancy in GPU memory for failure recovery. First, we employ a retry-based mechanism to recover training state, which uses the preserved model states for retraining (Attempt#1). Second, we employ EC to protect all data residing in GPU memory with high memory efficiency (Attempt#2). From these explorations, we derive two key findings (Findings #1 and #2) that motivate our design.

3.1 Attempt#1: Employ Retry to Recover Data

Methodology: Existing checkpoint-based methods recover training by restarting from the latest model states, which is conceptually similar to a retry [16, 38, 56]. Retry discards intermediate results and recomputes them from preserved model states. Thus, as long as the critical model states remain intact in GPU memory, the lost data caused by failures can be treated as discarded data and recovered via retry.

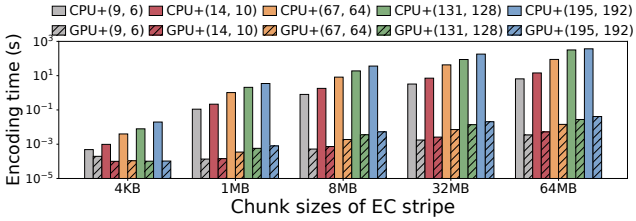


Figure 5: **Encoding time on CPU and GPU platforms (§3).** *The y-axis is represented on an exponential scale.*

Motivated by this observation, we replicate critical model states during training to enable efficient recovery. Figure 4 (a) shows the memory consumption when employing retry for failure recovery during 3B LLaMA training. The results indicate that retry incurs additional GPU memory overhead of 73.4% compared to training without retry. This overhead arises from storing replicas of critical model states, such as parameters and optimizer states. Given that GPU memory is usually limited (e.g., 80GB in NVIDIA H100 [37]) and highly utilized (70–97% [11]), such redundancy introduced by retry can easily lead to out-of-memory failures during training.

Finding#1: *Although retry is capable of recovering from GPU memory failures, it incurs significant memory overhead during model training.*

3.2 Attempt#2: Enable EC in Training

Methodology: As retry incurs substantial memory overhead, we then resort to erasure coding (EC) for failure recovery with low GPU memory overhead (§2.3). A (k,m) EC incurs only $\frac{m}{k}$ redundancy (e.g., 1.6% for a (195,192)-code [15]) to tolerate any m chunk failures. However, EC introduces additional computation overhead to generate parity chunks (§2.3), which may become a bottleneck in CPU-based systems [6]. We first study the encoding performance when employing EC on GPUs. Figure 5 shows that an NVIDIA A100 GPU [36] reduces the encoding time by 59.5% for a (9,6)-code and up to 99.5% for a (195,192)-code, compared to an Intel Xeon Gold 5320 CPU [52] under a 4KB chunk size; when the chunk size increases to 64 MB, a GPU can generate a (195,192)-code stripe (with the data size of 12 GB) in only 0.04 seconds, implying that GPUs support efficient data encoding for EC.

Driven by this observation, we enable EC during training to protect data in GPUs. Figure 4(b) and (c) report the training time and training consistency under injected HBM failures using a $(128,3)$-code, following the experimental setup described in §7.1. We make two key observations. First, although EC improves reliability, it increases training time by 30.3% (Figure 4(b)). This overhead arises because the parity chunks in EC must be recomputed whenever data chunks are updated, and many GPU-resident states are frequently modified (e.g., activations and gradients). As a result, EC incurs substantial computation overhead. Second, directly applying EC degrades model accuracy (Figure 4(c)). This is

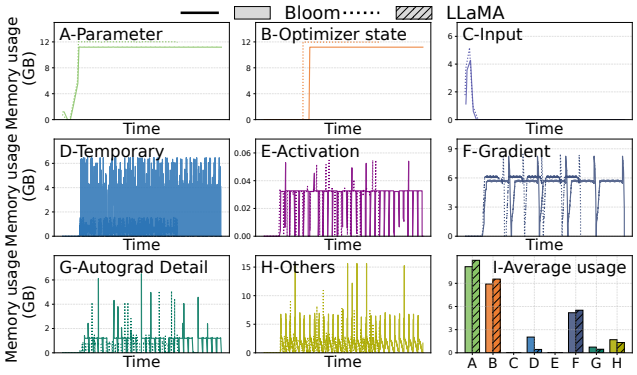


Figure 6: **Memory usage during training of Bloom (solid lines and bars) and LLaMA (dotted lines and hatched bars).** A-H show memory usage changes over time for different data segments, while I shows the average usage.

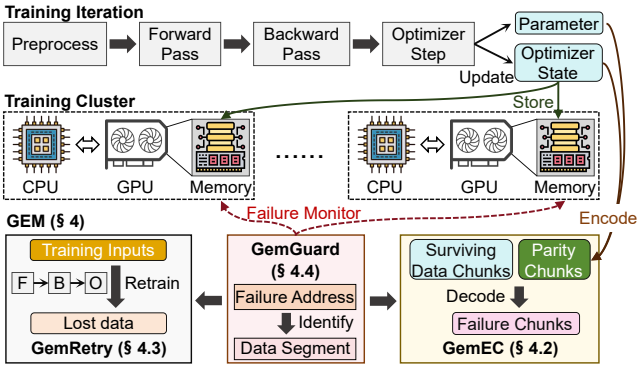


Figure 7: **Overview of GEM (§4.1).** *GEM ensures GPU memory reliability by using an EC-based mechanism (GemEC) to protect parameters and optimizer states, while employing retry (GemRetry) to recover the remaining data.*

because EC operates over finite fields (e.g., Galois fields [48]), whereas model states use floating-point representations that do not form a finite field. Consequently, EC introduces rounding errors, leading to training inconsistency (see Figure 4(c)).

Finding#2: *GPUs support efficient data encoding in EC. However, directly applying EC in training introduces high computation overhead due to frequent parity regeneration and degrades consistency because of finite-field incompatibility.*

4 Design of GEM

4.1 Design Overview

Key ideas. To efficiently recover from GPU memory failures while preserving training consistency, we propose GEM, which combines selective EC with a retry mechanism. It enables recovery directly from the accessible data already in GPU memory, rather than loading redundancy (e.g., replicated model states) from CPU memory or persistent storage. As a result, GEM reduces redundancy overhead and improves recovery efficiency.

Design insights. Neither EC nor retry is a one-size-fits-all solution (Attempts #1 and #2), as applying them to all training data incurs either high computation overhead (EC) or high memory overhead (retry). To identify which data segments are appropriate for EC versus retry, we profile GPU memory usage during the training of two widely used foundation LLMs (LLaMA3 [27] and Bloom [64]) in Figure 6. We profile both models over a single epoch, which is representative because their training behavior follows a highly regular pattern [11]. We find that parameters, optimizer states, and input data remain stable after initialization, whereas the remaining model states (D–H) exhibit 38.5× more frequent memory changes. Since changes in memory usage indicate data updates, these results suggest that data segments D–H are unsuitable for EC protection, as they are frequently updated. Input data is fully freed after initialization, making EC protection unnecessary. In contrast, parameters and optimizer states consistently occupy most of the GPU memory (Figure 6-I) and are therefore suitable candidates for EC protection. Moreover, parameters and optimizer states are the critical model states. Existing checkpointing systems [9, 30, 62] have shown that training can be restarted from these critical states. Therefore, once these states are protected by EC, the remaining data segments can be recovered via retry without the need to load additional replicas from CPU memory or persistent storage.

System overview. Based on these insights, GEM introduces GemEC and GemRetry to handle GPU memory failures occurring in different data segments. Figure 7 provides an overview of GEM. Specifically, GEM employs EC (GemEC, §4.2) to protect critical model states stored in GPU memory, as EC is storage-/computation-efficient for these data with low update frequency. To ensure the reliability of the remaining unprotected data (i.e., non-critical states) in GPU memory, we further introduce GemRetry (§4.3), which leverages the critical model states protected by GemEC together with essential training states to perform retry. By doing so, GEM efficiently recovers from GPU memory failures across different data segments while reducing the incurred overhead. Since different data segments use different protection methods, we further propose GemGuard (§4.4), which employs a hash-based directory to identify data segments and quickly assign the appropriate recovery mechanism.

4.2 GemEC: Erasure-Coded GPU Memory

While EC is a promising solution for protecting critical model states, two key challenges remain unaddressed. First, EC encodes data chunks using finite-field arithmetic (e.g., Galois fields [48]) to avoid rounding errors [51,52], whereas floating-point values do not form a finite field. Consequently, directly protecting training data in floating-point format can degrade model accuracy (see Figure 4(b)). Second, GPU memory re-

sources are limited during training, with utilization reaching up to 97% [11]. Therefore, reducing the memory overhead introduced by EC without compromising reliability is essential. To address these challenges, we design GemEC, which provides reliability with negligible memory overhead.

Parity chunk computation. Although EC is suitable for protecting critical model states (i.e., parameters and optimizer states), these data are stored in floating-point format. Hence, GemEC first obtains the memory addresses of updated parameters and optimizer states, then reads the raw bytes at these addresses and interprets each byte as an 8-bit unsigned integer (i.e., `uint8`), ensuring that the data reside within the finite field $GF(2^8)$. Next, GemEC groups multiple `uint8` values into data chunks and constructs an EC stripe with k data chunks, after which it computes and stores m parity chunks. By doing so, GemEC efficiently generates parity chunks to protect data against GPU memory failures without sacrificing model accuracy. To ensure that redundancy can be stored entirely in GPU memory, GemEC adopts a wide-stripe EC configuration [15]. It lowers memory overhead by using a large k while keeping m (the number of parity chunks) fixed, thereby reducing redundancy (i.e., $\frac{m}{k}$) while still tolerating any m chunk failures. For example, increasing k to 128 with $m = 3$ reduces redundancy to only 2.3% while still allowing recovery from any three chunk failures. This design enables the redundancy to fit in GPU memory under high utilization.

Adaptive computation across parallelisms. GemEC computes parity chunks for model states stored in local GPU memory. Because parallel strategies distribute model states differently, parity computation should be adapted to their placement. Under strategies that fully shard parameters and optimizer states, such as tensor parallelism [33, 53] and pipeline parallelism [32, 45], each worker stores distinct model states, allowing GemEC to compute parity without redundant work. In contrast, some systems, such as ZeRO stages 1 and 2 [47], replicate model states to reduce the communication overhead of sharding. Consequently, multiple workers may store identical states, causing GemEC to redundantly compute their parity chunks. To eliminate this redundancy, GemEC distributes EC computation across workers. Within a data-parallel group that holds replicated model states, each worker computes parity only for a designated partition and gathers the parity chunks for the remaining partitions from other workers. As a result, each partition is encoded only once, while every worker still obtains the complete set of parity chunks. This design allows GemEC to efficiently support both sharded and replicated parallel strategies (Exp#10). Moreover, this optimization can be disabled in bandwidth-constrained environments, in which case GEM still incurs less than 1% overhead (Exp#4).

Example of GemEC. Figure 8 illustrates how GEM operates under data parallelism with two GPUs. We use ZeRO-1 [47], which partitions optimizer states while keeping parameters fully replicated, as an example. Specifically, each GPU holds

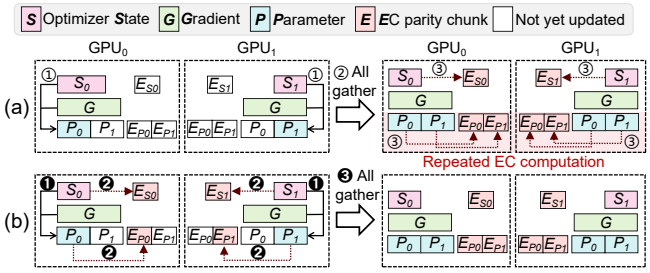


Figure 8: **Example of GemEC under ZeRO-1 parallelism (§4.2).** *(a) GemEC without any optimization for data parallelism, which leads to repeated EC computations; and (b) GemEC leveraging the characteristics of parallelism to reduce unnecessary computational overhead under ZeRO-1.*

one partition of the optimizer states (e.g., S_0 on GPU_0 and S_1 on GPU_1) and updates only the parameters corresponding to its partition (e.g., GPU_1 updates P_1 in Step ①). Then, the full set of parameters is updated via an all-gather operation, where both GPU_0 and GPU_1 obtain P_0 and P_1 (Step ②). Figure 8(a) shows GemEC without any parallelism-aware optimization, where EC computation occurs after all parameters are updated, resulting in redundant parity computations for the same parameters (Step ③). In contrast, Figure 8(b) demonstrates how GemEC eliminates redundant computation under ZeRO-1. Each GPU computes parity chunks only for the parameters it updates locally (e.g., GPU_0 computes E_{P0} for P_0, and GPU_1 computes E_{P1} for P_1 in Step ❷) and the parity chunks are then exchanged via all-gather (Step ❸), ensuring that each parity chunk is generated only once.

Recovery by GemEC. To recover from GPU memory failures using parity chunks, GemEC records metadata related to its EC encoding. It first maintains a hash-based chunk directory (implemented by Python's built-in `dict`) that uses the address of each chunk as the key and the index of its corresponding stripe as the value. Because the directory is organized as a hash table, GemEC can locate the encoding stripe of a failed data chunk in $O(1)$ time. Next, GemEC reads the surviving chunks from the corresponding stripe. To facilitate this, it maintains a stripe list that records the addresses of both data and parity chunks for each stripe. From this list, GemEC selects k surviving chunks sequentially, as adjacent chunks often occupy consecutive physical addresses, which benefits GPU parallelism. Finally, these k chunks are used by GemEC to reconstruct the failed data. Multiple simultaneous HBM failures are typically concentrated within a contiguous physical address space (e.g., a 1 KB row or a 16 MB bank [66, 74]), whereas the chunks encoded by GemEC are usually tens of megabytes in size (similar to prior wide-stripe EC [15]). Hence, HBM failures generally affect only a single chunk. This ensures that even when employing wide-stripe EC encoding (e.g., k=128 and m=3), GemEC still provides protection against multiple simultaneous HBM failures.

4.3 GemRetry: Training Again

As the critical model states are already protected by EC, we further introduce GemRetry to recover the remaining non-critical model states via retry. However, in addition to the critical model states, retry still requires certain training states and occasionally intermediate results to ensure that retry preserves training consistency. In this subsection, we first describe the training inputs required to retry at both the step and phase levels. We then present a mixed-granularity retry strategy that balances memory overhead (for storing training inputs) with the time wasted due to retraining.

Training inputs for GemRetry. To avoid compromising model accuracy, GemRetry must use the same training inputs. When retrying a training step, it requires both the critical model states and the training states, where the former are already protected by GemEC. Therefore, GemRetry must ensure that training states can be restored when GPU memory failures occur. Fortunately, training states represent runtime metadata such as the current epoch number and global step, which are stored as scalars or small arrays and occupy negligible memory (e.g., <1 MB; see Exp#3 in §7.2).

Each training step comprises multiple phases, which raises the question of whether GemRetry can retry individual phases to reduce wasted time. We first examine the inputs for the forward pass, which consist of parameters and a subset of the training states. This is similar to the inputs required for a full training step. In contrast, the backward pass and optimizer step require not only the same (or a subset) of the step-level inputs but also the intermediate outputs generated by the preceding phase. For example, the backward pass requires the loss from the forward pass to compute gradients, and the optimizer step relies on these gradients to update parameters. Such intermediate results occupy a non-negligible portion of GPU memory (e.g., gradients account for 18.0% of memory usage; see Figure 6) and are frequently updated.

Mixed-granularity retry. To balance memory usage and wasted time, GemRetry adopts a mixed-granularity retry mechanism that combines phase-level and step-level retries. First, GemRetry performs a phase-level retry when a failure occurs during the forward pass, since retraining the forward pass incurs only low memory overhead to protect the training input but can significantly reduce time wasted during training. Second, supporting retries for other phases with intermediate outputs requires substantial memory, which can lead to out-of-memory failures. Therefore, GemRetry performs a step-level retry for these phases, providing a memory-efficient failure recovery method.

Example of GemRetry. Figure 9 illustrates an example of how GemRetry recovers the training process. In this example, training is performed with gradient accumulation [22], where each training step consists of multiple forward and backward passes, followed by a single optimizer update. First, if a failure occurs during a forward pass (e.g., F_2, Step ❶), GemRetry

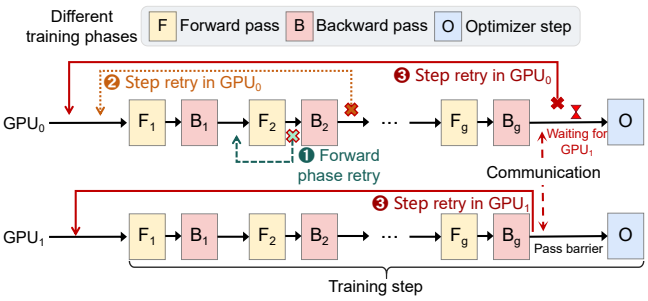


Figure 9: **Example of using GemRetry to recompute the failed training data (§4.3).** *GemRetry retries a training phase when a failure occurs during the forward pass, while it retries an entire training step for failures in other phases.*

performs a phase retry (i.e., retry only the forward pass F_2), which reduces computation overhead compared to retrying the entire training step from F_1. Second, if a failure occurs during backward pass B_2, the gradients may be only partially updated. Hence, GemRetry discards these intermediate results to maintain consistency and retries the training step on GPU_0 (Step ❷), as the gradients have not yet been propagated to other workers. Third, failures may occur during gradient synchronization (i.e., between B_g and the optimizer step) on GPU_0, while GPU_1 may have already passed the communication barrier and proceeded to the subsequent training phase (i.e., the optimizer step). If we only retry the training step on GPU_0, communication stalls occur because GPU_0 is waiting to synchronize at the barrier with GPU_1, while GPU_1 has already moved on. To address this, GemRetry performs a step retry for all GPUs within the same parallelism rank (Step ❸).

4.4 GemGuard: Failure Handling

Although both GemEC and GemRetry can recover from GPU memory failures, they target different data segments. Therefore, we introduce GemGuard, which monitors GPU memory failures and dispatches the appropriate recovery mechanism.

GemGuard monitors failure signals from GPU memory and intercepts them to extract the failure address. Specifically, GemGuard installs a signal handler that intercepts GPU memory failure signals (`SIGBUS`). This prevents the training process from terminating, which is the default behavior when a user process receives `SIGBUS` [43]. The handler then parses the failed GPU memory address and identifies the corresponding data segment to determine the appropriate recovery action. However, GPU memory contains a large number of allocated data segments, which can make segment lookup inefficient. To address this, GemGuard reuses the directory maintained by GemEC, a hash-based directory that locates failure addresses in $O(1)$ time (see §4.2). If the failure address appears in the EC directory, indicating that it is protected by GemEC, GemGuard invokes GemEC to restore the lost data. Otherwise, GemGuard calls GemRetry to recover the training process.

5 Extensions of GEM

GEM for model inference. During model inference, GPU memory stores the model parameters and the key–value (KV) cache. Rather than reconstructing approximate predictions using parity models (e.g., ParM [17]), GEM tolerates GPU memory failures during inference by restoring lost model states through EC while preserving exact inference correctness. Specifically, GEM protects both the parameters and KV blocks. First, we encode all parameters stored in GPU memory using GemEC. Since parameters remain unchanged during inference, the computation overhead introduced by GemEC is negligible. For the KV cache, new KV blocks are generated continuously, and encoding every KV block immediately would cause frequent EC updates, significantly reducing efficiency. To address this, GemEC employs a batch-encoding strategy in which previously generated KV blocks are encoded into a stripe in batches, rather than updating the EC stripe each time a KV block is produced. This approach reduces the computation overhead of protecting KV blocks, but leaves newly generated KV blocks temporarily unprotected. For these unprotected KV blocks, we additionally enable GemRetry, which recomputes the missing KV blocks using the EC-encoded KV blocks and parameters.

Extending GEM to SDC. Silent data corruption (SDC) leads to incorrect computation results without triggering any failure signals [24]. Since matrix multiplication is the most frequent operation in deep learning workloads, GEM can integrate GemEC into matrix computation by multiplying each matrix with an EC encoding matrix to generate parity data. GEM then verifies each output by checking whether the results and their corresponding parity remain within a predefined error bound. If any discrepancy exceeds this bound, GemRetry selectively recomputes the potentially corrupted portion.

Effectiveness when enabling data offload. To accommodate large models within limited GPU memory, existing studies offload certain model states (e.g., optimizer states and gradients) to CPU memory [49]. With data offloading, the set of data segments that remain in GPU memory changes. Therefore, we must reconsider which data segments need protection. When data offloading is enabled, the principle of GEM is to protect only the data segments that remain in GPU memory, while the offloaded segments rely on CPU-side ECC for correctness. As a result, the effectiveness of GEM in protecting GPU memory remains unaffected, and the overhead is even lower because fewer data segments need to be encoded.

6 Implementation

We implement GEM primarily in Python and C/C++, with support for deployment on NVIDIA GPUs. We integrate its two core components—GemEC and GemRetry—into `DeepSpeed` [29], a deep learning optimization library supporting modern LLM training. While the implementation of GemRetry may vary across training frameworks, most frameworks separate different training phases into modular code blocks [7], allowing GemRetry to invoke the corresponding module for retraining. For GemEC, we use CUDA C/C++ to perform the matrix operations required for EC. In addition, Python is used to manage the metadata for GemEC, including the mapping between chunk addresses and their corresponding stripe indices (implemented using Python's built-in `dict`) as well as the list of EC stripes. GEM provides transparent reliability support for GPU memory, as all components are integrated into the training system without requiring any user code modification. To handle GPU memory failures, we implement GemGuard with a dedicated signal-handling thread.

7 Evaluation

We evaluate GEM across various models to answer the following questions.

- How do GEM and its recovery techniques perform during failure recovery (Exp#1-2)?
- What overhead does GEM introduce during training (Exp#3–#4)?
- How does GEM scale with more GPUs, frequent failures, and larger model sizes (Exp#5–#8)?
- How does GEM perform under different EC configurations and parallelism strategies (Exp#9–#10)?
- How does GEM complement existing checkpointing methods to further reduce wasted time (Exp#11)?
- Can GEM be extended to other accelerators (Exp#12)?

7.1 Experimental Setup

Testbed and configurations. We evaluate GEM on a GPU platform with two nodes, each equipped with two NVIDIA A100 (80GB) GPUs, two Intel Xeon Gold 5320 CPUs, and 512 GB DRAM. Intra-node GPU communication uses peer-to-peer transport, while inter-node communication is handled via an InfiniBand RNIC. We adopt the ZeRO optimization strategy [47] for distributed training with data parallelism, as it is a widely used memory-optimization technique [2, 64]. We also evaluate GEM under different parallelism strategies in Exp#10. Since GEM leverages local GPU memory for failure recovery, we use a single node as the default setup and also demonstrate multi-node performance (Exp#6). We evaluate three representative LLMs: Bloom [64], LLaMA 3 [27], and Qwen [2]. Given that a full training iteration requires substantial time [2, 27, 64], we train each model for only ten epochs, with each epoch consisting of twenty training steps. Gradient accumulation [22], widely used in models such as LLaMA [27] and Bloom [64], is configured with six accumulation steps per training step. The batch size and sequence

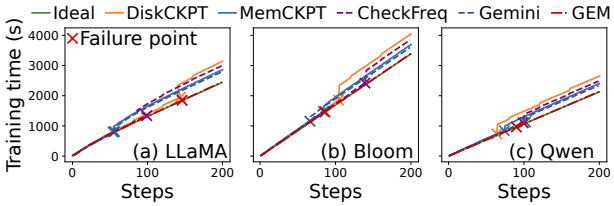


Figure 10: **Exp#1 (End-to-end training time).** *Sudden spikes indicate substantial time spent on retraining.*

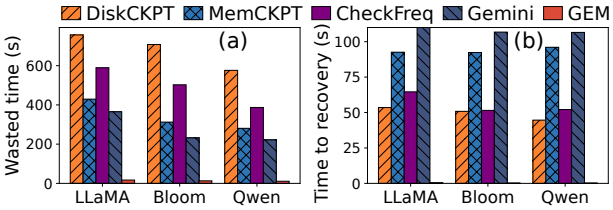


Figure 11: **Wasted time analysis.** *(a) Wasted time due to failure recovery and (b) time to recovery for each failure.*

length are set to 4 and 2,048, respectively, to utilize more of the HBM capacity [14]. All models are configured with 3B parameters, with additional sizes tested for scalability analysis (Exp#7). The software stack includes CUDA v12.4.99 [34], PyTorch v2.6.0 [41], DeepSpeed v0.15.4 [29], and NCCL v2.21.5 [35].

Comparison approaches. We compare GEM against four checkpointing-based approaches. (i) DiskCKPT, which checkpoints model states to storage. We perform DiskCKPT every two epochs and implement it using `torch.save()` [41]. (ii) MemCKPT, which checkpoints model states to CPU memory. We perform MemCKPT every epoch, and MemCKPT serializes the latest complete checkpoint using `torch.save()` upon failure [62]. (iii) CheckFreq [30], a representative approach for reducing checkpoint overhead when saving model states to disk. It pipelines memory writes with computation and asynchronously persists the checkpoint to disk. We perform it every two epochs. (iv) Gemini [62], a CPU-memory checkpointing method that overlaps checkpoint communication with computation. We perform Gemini every five iterations to balance checkpointing time and recovery cost. For GEM, we adopt a (131, 128)-code by default [15]. Since GEM is orthogonal to existing checkpointing methods, we also evaluate their complementarity in Exp#11. In addition, we consider training without any runtime or recovery overhead as an idealized baseline (denoted as *Ideal*).

Failure simulation. We evaluate recovery efficiency by injecting simulated GPU memory failures during training. In real systems, when on-die ECC cannot correct memory errors, the GPU memory subsystem sends a `SIGBUS` signal to the training process, causing it to crash [43]. Therefore, we simulate GPU memory failures by sending a `SIGBUS` signal associated with a randomly selected GPU memory address [13, 72], indicating that the corresponding memory region has experienced an uncorrectable error. This signal interrupts training unless properly intercepted and handled. By default, we inject one failure per training run, and we further evaluate scalability under more frequent failures (Exp#5).

7.2 Experimental Results

Exp#1 (End-to-end training time). We first evaluate the end-to-end training time of GEM and other comparison methods on three representative models. All evaluated methods preserve training correctness and consistency after failures, as evidenced by their training loss curves closely matching those of the Ideal execution. Therefore, any performance difference stems solely from their recovery mechanisms. Figure 10 shows that DiskCKPT and MemCKPT increase the training time by 26.4% and 13.4% on average compared to the Ideal, respectively. Although CheckFreq and Gemini adopt pipelining and computation–checkpoint interleaving to reduce checkpointing overhead and enable shorter checkpoint intervals, they still incur 19.1% and 10.8% additional training time. In contrast, GEM preserves training efficiency close to the Ideal, introducing only 0.5% overhead.

We further analyze the wasted time incurred by different approaches across three models. As shown in Figure 11(a), DiskCKPT, MemCKPT, CheckFreq, and Gemini incur 49.8×, 24.1×, 35.4×, and 19.0× more wasted time than GEM, respectively. These improvements arise from two primary factors. First, GEM significantly shortens the recovery time after failures. As shown in Figure 11(b), it reduces recovery time by 99.4–99.7% compared with four representative checkpoint-based methods. This improvement comes from performing EC decoding (via GemEC) or restoring training inputs (via GemRetry) directly in GPU memory, whereas the other methods must load checkpoints from CPU memory or disk. Second, GEM minimizes retraining overhead by retrying a single training step or even just the forward pass. In contrast, existing systems must restart from the most recent checkpoint, which wastes the computation of multiple steps depending on the gap between the failure point and the latest saved state. Moreover, MemCKPT and Gemini need to serialize the checkpoints during recovery, which is time-consuming.

Exp#2 (Breakdown analysis). To investigate the efficiency of the design techniques in GEM, we train models and perform recovery using GemEC, GemRetry, and their combination (GEM). Figure 12 shows that the three techniques incur only 0.4%–0.9% overhead in training time. While GemRetry incurs wasted time from retraining, it achieves the lowest recovery time (9.1 seconds on average). In contrast, GemEC requires extra time (17.2 seconds on average) to compute parity chunks during training, adding overhead even without failures. Although GemRetry incurs a lower overhead, it depends on the parameters and optimizer states protected by GemEC for retry (§4.3). When combining both techniques (i.e., GEM), the wasted time (22.2 seconds on average) is

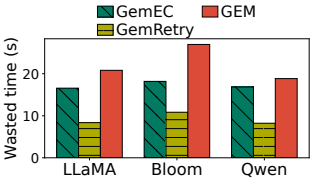


Figure 12: **Exp#2 (Breakdown analysis).**

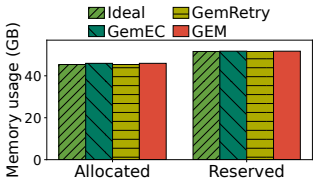


Figure 13: **Exp#3 (Memory overhead).**

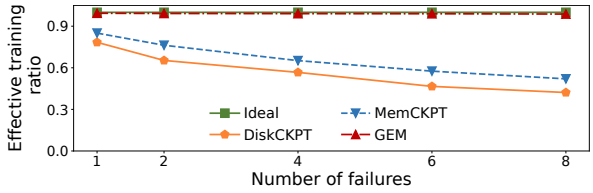


Figure 15: **Exp#5 (Scaling to frequent failures).** *GEM remains effective as the failure frequency increases.*

higher than either technique alone, as GEM inherits the runtime overhead of GemEC and uses GemRetry to retrain data that are not protected by GemEC.

Exp#3 (Memory overhead). Recall that GEM maintains additional parity chunks (for GemEC) and replicated training inputs (for GemRetry). We measure both allocated memory (i.e., actually used) and reserved memory (i.e., total requested) of GEM during LLaMA training. We focus on peak memory usage, as it directly affects CUDA out-of-memory errors, and omit average usage due to its rapid fluctuations (see Figure 6). Figure 13 reports the resulting memory usage.

We make two observations. First, GEM introduces 1.2% additional allocated memory (576MB) but increases reserved memory by only 0.3% (142MB). Although existing checkpoint-based methods do not require additional GPU memory for storing redundancy, their reserved memory (e.g., 128MB in Gemini [62]) is still comparable to that of GEM, as they typically maintain a buffer for transferring model states. Second, the memory overhead of GemRetry is negligible (1MB), as it requires only a small buffer to keep replicated training inputs. In contrast, GemEC introduces 575MB of additional memory to store parity chunks. However, only 24.7% of these parity chunks (i.e., $\frac{142\text{MB}}{576\text{MB}}$) require new allocations, since most chunks can be stored by exploiting fragmentation within the already reserved memory space.

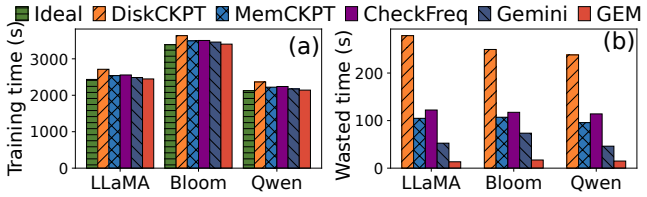


Figure 14: **Exp#4 (Runtime overhead).** *GEM introduces negligible runtime overhead (0.6%) for EC encoding.*

Exp#4 (Runtime overhead). To evaluate the efficiency of these fault-tolerance systems, we measure their runtime overhead, defined as the extra time incurred when enabling these methods during normal training (i.e., without failures). Figure 14(a) shows that DiskCKPT, MemCKPT, CheckFreq, and Gemini incur 2.2–10.0% additional training time. We find that in-memory checkpointing methods (e.g., MemCKPT and Gemini) incur lower runtime overhead even at high checkpointing frequencies, primarily due to the high bandwidth of CPU–GPU data transfers. Figure 14(b) further presents the wasted time incurred by these methods. Although Gemini efficiently reduces runtime overhead by interleaving model-saving traffic with training, its wasted time is still 3.7× that of GEM. The reason is that GEM stores the redundancy required for recovery directly in GPU memory, thereby avoiding frequent checkpoint transfers to remote storage and significantly reducing communication overhead. Notably, although GEM provides an adaptive-computation optimization across parallelisms to eliminate redundant EC computations on replicated model states (§4.3), this optimization can be disabled in bandwidth-constrained environments. Even without this optimization, GEM incurs only 0.9% runtime overhead (a 60.5% reduction compared with Gemini).

Exp#5 (Scaling to frequent failures). We evaluate the effectiveness of GEM when training LLaMA under frequent failures by varying the number of failures from 1 to 8. We consider the *effective training time ratio* [62], which indicates the percentage of productive training progress achieved within a given period. Figure 15 shows that DiskCKPT and MemCKPT drop sharply to 42.2% and 52.0%, respectively, under eight failures due to their higher recovery overheads. In contrast, GEM remains close to the Ideal, with only a 0.67%–1.2% reduction, as it recovers data without interrupting the training process and avoids checkpointing overhead.

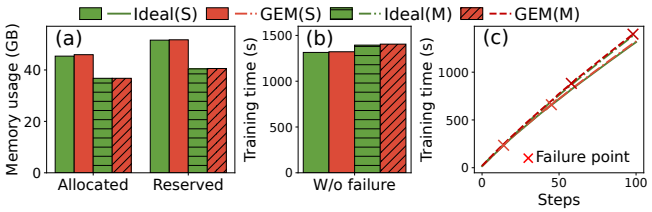


Figure 16: **Exp#6 (Scaling to more GPUs).** *(a) Memory usage; and training time under (b) training without (w/o) failures and (c) training with failures. "S" denotes the single-node platform, while "M" denotes the multi-node platform.*

Exp#6 (Scaling to more GPUs). We assess GEM when scaling to more GPUs by training LLaMA on single-node (two GPUs) and multi-node (four GPUs) platforms. First, we assess memory and runtime overheads when training without failures. Figure 16 (a) shows that GEM adds only 1.2% memory overhead on the single-node platform, decreasing to 0.97% on the multi-node platform. For reserved memory, it adds only 142 MB on average for both platforms. Figure 16 (b) fur-

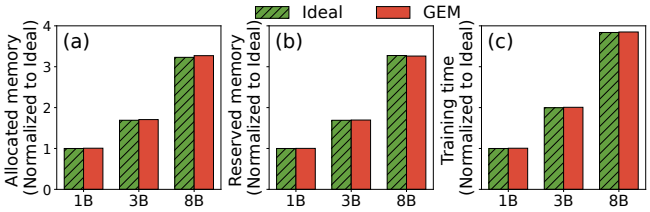


Figure 17: **Exp#7 (Effectiveness for various model sizes).** *(a) allocated and (b) reserved memory; and (c) training time.*

ther demonstrates that training time increases by just 0.64% on average. These results indicate that GEM imposes minimal overhead when scaling to more GPUs. We then evaluate the scalability of GEM during failure recovery. Figure 16 (c) shows that GEM increases training time by only 1.0% on average for failure recovery, demonstrating that it can ensure GPU memory reliability in larger systems with negligible overhead.

Exp#7 (Effectiveness for different model sizes). We further evaluate memory overhead and training time by varying the LLaMA model size from 1B to 8B on a multi-node platform. Figure 17 shows that GEM achieves memory usage and training time comparable to the Ideal across various model sizes. Specifically, it incurs at most 1.3% allocated memory overhead (for the 8B model), while reserved memory increases by only 0.34% (for the 3B model). The reason is that the memory overhead of GEM is proportional to the partition size of the critical model states residing in GPU memory. Moreover, its runtime overhead is also minimal, ranging from 0.31% for the 8B model to 0.61% for the 1B model.

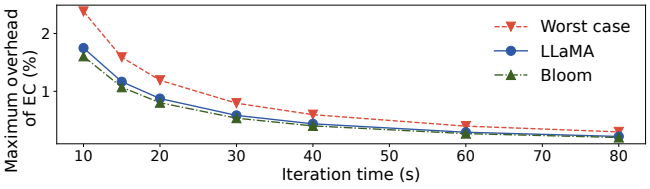


Figure 18: **Exp#8 (Overhead in large-scale training).**

Exp#8 (Overhead in large-scale training). To evaluate the effectiveness of GEM at large scale (i.e., with more GPUs and larger models), we conduct a formal analysis of its overhead. Since GEM applies EC only to data stored in each worker's local GPU memory, its overhead depends on the partition size of the critical model states on each GPU (denoted as S) rather than the full model size. We focus on computation overhead, as memory overhead is fixed by the EC configuration (i.e., $\frac{m}{k} \cdot S$) and is negligible under wide-stripe settings.

Specifically, GEM generates parity chunks for the protected model states, which may extend the training iteration time. The computation overhead is $O = \frac{P/S}{T_{\text{iter}}}$, where P denotes the parity computation throughput and T_{iter} is the iteration time. Based on our measurements on NVIDIA A100 GPUs, the EC throughput P and the maximum supported partition size S (worst case) are 335.8 GB/s and 80 GB (i.e., the GPU memory capacity), respectively. We further use the ratio of the critical model states' partition sizes obtained from profiling (Figure 6) to evaluate the effectiveness of GEM when training LLaMA and Bloom. Figure 18 reports the resulting computation overhead as we vary the iteration time from 10 s (similar to training LLaMA on 3072 GPUs [68]) to 80 s (similar to training a 100B BERT model [62]). Across these settings, GEM incurs only 0.3–2.4% additional computation overhead in the worst case, which further reduces to 0.7% and 0.8% on average for LLaMA and Bloom, since not all GPU memory is allocated for storing critical model states. These results demonstrate that GEM maintains low overhead even when scaling to large clusters and large models.

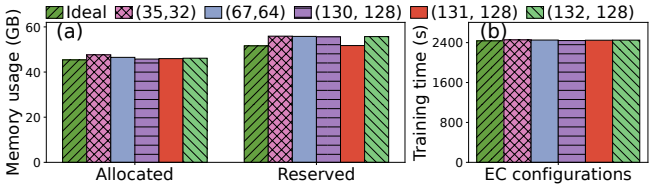


Figure 19: **Exp#9 (Impact of EC configurations).** *Higher parity ratios increase (a) memory usage and (b) training time.*

Exp#9 (Impact of EC configurations). We evaluate the impact of different EC configurations on GEM by varying the number of data chunks (i.e., k, from 32 to 128) and the number of parity chunks (i.e., m, from 2 to 4). These configurations provide reliability for scenarios where 1.6% (for a (130,128)-code) to 10.7% (for a (35,32)-code) of data chunks fail. Figure 19 shows that higher parity ratios (i.e., $\frac{m}{k}$ in a $(k+m,k)$-code) lead to increased memory overhead and training time. This is because higher parity ratios increase the size of coded chunks, which requires more GPU resources for parallel computation and for storing parity chunks. Although GEM increases wasted time for a wider stripe, the maximum wasted time remains only 0.88%.

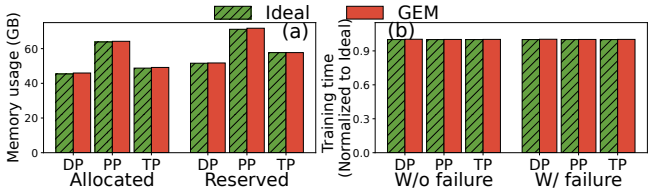


Figure 20: **Exp#10 (Scaling across different parallelism strategies).** *(a) Memory overhead and (b) training time.*

Exp#10 (Scaling across different parallelism strategies). To evaluate the generality of GEM, we measure computation and memory overheads under different parallelism strategies. Figure 20(a) shows that GEM incurs negligible memory overhead (0.6% on average) across all strategies, thanks to its lightweight encoding mechanism (§4.2). Figure 20(b) reports the training time with and without failures. Similar to DP, GEM encodes parameters and optimizer states during training

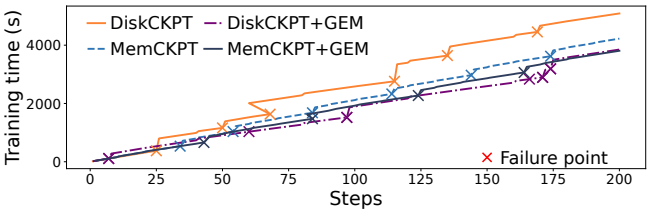


Figure 21: **Exp#11 (Combination with checkpointing).**

and leverages parity chunks to recover lost data upon failures. This increases training time by only 1.7% on average for PP and TP. These results demonstrate that GEM effectively recovers from failures across different parallelism strategies.

Exp#11 (Combining with checkpointing). GEM is orthogonal to existing checkpointing methods and can enhance recovery efficiency for GPU memory failures. We simulate six GPU failures during training, two of which are GPU memory failures that can be recovered by GEM, reflecting failure ratios observed in real-world deployments [27]. Figure 21 shows that GEM reduces the overall training time of DiskCKPT and MemCKPT by 24.3% and 9.9%, respectively. This improvement comes from GEM's ability to substantially reduce the wasted time caused by GPU memory failures (Exp#1).

Exp#12 (Extension to other accelerators). To assess the generality of GEM, we further evaluate it on an Ascend 910B NPU platform consisting of eight NPUs, each equipped with 32 GB of HBM. We run the 3B LLaMA3 and 3B Qwen2 models and measure recovery efficiency and memory overhead. Our evaluation shows that GEM requires only 1.4 s and 1.6 s, on average, to recover from memory failures (including recovery using both GemEC and GemRetry) during the training of LLaMA3 and Qwen2, respectively. In addition, GEM introduces additional allocated- and reserved-memory overheads of only 0.24% and 0.26%, respectively, across the two models. These results demonstrate that GEM can effectively protect HBM in accelerators beyond GPUs.

8 Related Work

Checkpointing in deep learning. Checkpointing is widely used in deep learning frameworks for failure recovery [1, 41]. To reduce overhead, CheckFreq [30] pipelines checkpointing and automatically determines the optimal checkpointing frequency, while Nebula [28] saves checkpoints asynchronously. Since the limited bandwidth of disks restricts checkpointing frequency, Gemini [62] transfers model states to CPU memory to enable more frequent checkpointing. ECCheck [44] further introduces erasure coding to protect checkpoints stored in CPU memory, whereas GEM targets online recovery of model states in GPU memory. On the other hand, Check-N-Run [9] and AdaCheck [23] increase checkpointing frequency by using differential techniques and exploiting state redundancy to reduce checkpoint size. ByteCheckpoint [59] and UCP [20] further enable efficient and adaptable checkpointing across various parallelism strategies. Unlike these approaches, Singularity [54] and Cricket [8] leverage GPU virtualization for live migration, while PhOS [63] overlaps checkpoint/restore (C/R) with application execution and GCR [69] adopts a hybrid C/R mechanism with control–data separation.

Fault tolerance for distributed training. Several work develop fault-tolerant mechanisms for distributed training. SCAR [46] quantifies the effects of computation errors caused by inconsistency, while CPR [25] relaxes consistency requirements to balance accuracy and performance. Similarly, ATTNChecker [21] ensures reliability by enforcing consistency only for critical components through an algorithm-based fault-tolerance technique. As data parallelism maintains replicas across nodes, SWIFT [73] and Just-in-Time Checkpoint [12] leverage these replicas for training recovery. Moreover, Bamboo [57] replicates model partitions within the same pipeline to enable recovery. For runtime recovery, Elastic Horovod [18] supports dynamic scaling of workers to eliminate faulty nodes during training, while KungFu [26] and SpotNik [58] provide online recovery by partially retrying failed steps instead of re-executing the entire training step. ByteRobust [60] further improves the robustness of large-scale training by introducing an infrastructure-level management system.

However, existing failure recovery mechanisms typically load model states from CPU memory or disk. In contrast, GEM computes and stores erasure-coded parity chunks in GPU memory, reducing additional data transfers and lowering runtime and recovery overhead. Moreover, GEM can complement existing methods to reduce wasted time.

9 Conclusion

We design GEM, a fault-tolerant system that combines selective erasure coding with a retry mechanism for GPU memory failure recovery. GEM employs EC to protect parameters and optimizer states and uses a mixed-granularity retry mechanism to protect the remaining frequently updated data. It further stores redundancy directly in GPU memory to shorten the recovery process. Extensive experiments demonstrate the effectiveness, efficiency, scalability, and generality of GEM.

Acknowledgements

We thank our shepherd, Danny Harnik, and the anonymous reviewers for their thorough and insightful comments. This work is supported by the Major Research Plan of the National Natural Science Foundation of China (Grant No. 92373114), the National Natural Science Foundation of China (Grant No. 624B2120), the CCF-Huawei Populus Grove Fund (Grant No. CCF-HuaweiSY202519), and the State Key Lab of Processors, Institute of Computing Technology, CAS (Grant No. CLQ202610).

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: a System for Large-Scale Machine Learning. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[2] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen Technical Report. *arXiv preprint arXiv:2309.16609*, 2023.

[3] Zhe Chen, Jiannan Wu, Wenhai Wang, Weijie Su, Guo Chen, Sen Xing, Muyan Zhong, Qinglong Zhang, Xizhou Zhu, Lewei Lu, Bin Li, Ping Luo, Tong Lu, Yu Qiao, and Jifeng Dai. InternVL: Scaling up Vision Foundation Models and Aligning for Generic Visual-Linguistic Tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.

[4] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality, March 2023.

[5] 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, Sashank 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. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[6] Xiaowen Chu, Chengjian Liu, Kai Ouyang, Ling Sing Yung, Hai Liu, and Yiu-Wing Leung. PErasure: a Parallel Cauchy Reed-Solomon Coding Library for GPUs. In *Proceeding of the 2015 IEEE International Conference on Communications (ICC)*, 2015.

[7] Jae W. Chung, Yifan Gu, Inho Jang, Lingda Meng, Niket Bansal, and Mosharaf Chowdhury. Reducing Energy Bloat in Large Model Training. In *Proceedings of ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP)*, 2024.

[8] Niklas Eiling, Jonas Baude, Stefan Lankes, and Antonello Monti. Cricket: A Virtualization Layer for Distributed Execution of CUDA Applications with Checkpoint/Restart Support. *Concurrency and Computation: Practice and Experience*, 34(14):e6474, 2022.

[9] 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 *Proceedings of USENIX 19th Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[10] Yingqiang Ge, Wenyue Hua, Kai Mei, Jianchao Ji, Juntao Tan, Shuyuan Xu, Zelong Li, and Yongfeng Zhang. OpenAGI: When LLM Meets Domain Experts. *Advances in Neural Information Processing Systems*, 36:5539–5568, 2023.

[11] Cong Guo, Rui Zhang, Jiale Xu, Jingwen Leng, Zihan Liu, Ziyu Huang, Minyi Guo, Hao Wu, Shouren Zhao, Junping Zhao, et al. GMLAKE: Efficient and Transparent GPU Memory Defragmentation for Large-Scale DNN Training with Virtual Memory Stitching. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[12] Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev, Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, and Muthian Sivathanu. Just-In-Time Checkpointing: Low Cost Error Recovery from Deep Learning Training Failures. In *Proceedings of the*

Nineteenth European Conference on Computer Systems (EuroSys), 2024.

[13] Hari, Siva Kumar Sastry and Tsai, Timothy and Stephenson, Mark and Keckler, Stephen W and Emer, Joel. SASSIFI: An Architecture-Level Fault Injection Tool for GPU Application Resilience Evaluation. In *Proceedings of 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017.

[14] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, et al. Characterization of Large Language Model Development in the Datacenter. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 709–729, 2024.

[15] Yuchong Hu, Liangfeng Cheng, Qiaori Yao, Patrick PC Lee, Weichun Wang, and Wei Chen. Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage. In *Proceeding of 19th USENIX Conference on File and Storage Technologies (FAST)*, 2021.

[16] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing Activation Recomputation in Large Transformer Models. In *Proceedings of the 6th Annual Conference on Machine Learning and Systems (MLSys)*, 2022.

[17] Kosaian, Jack and Rashmi, KV and Venkataraman, Shivaram. Parity Models: Erasure-Coded Resilience for Prediction Serving Systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[18] LF AI & Data Foundation. Elastic Horovod, 2023.

[19] Xinyu Lian, Sam Ade Jacobs, Lev Kurilenko, Masahiro Tanaka, Stas Bekman, Olatunji Ruwase, and Minjia Zhang. Universal Checkpointing: A Flexible and Efficient Distributed Checkpointing System for Large-Scale DNN Training with Reconfigurable Parallelism. In *Proceedings of the 2025 USENIX Annual Technical Conference (USENIX ATC)*, 2025.

[20] Lian, Xinyu and Jacobs, Sam Ade and Kurilenko, Lev and Tanaka, Masahiro and Bekman, Stas and Ruwase, Olatunji and Zhang, Minjia. Universal Checkpointing: A Flexible and Efficient Distributed Checkpointing System for Large-Scale DNN Training with Reconfigurable Parallelism. In *Proceedings of the 2025 USENIX Annual Technical Conference (USENIX ATC)*, 2025.

[21] Liang, Yuhang and Li, Xinyi and Ren, Jie and Li, Ang and Fang, Bo and Chen, Jieyang. ATTNChecker: Highly-Optimized Fault Tolerant Attention for Large Language Model Training. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2025.

[22] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. *arXiv Preprint ArXiv:1712.01887*, 2017.

[23] Liu, Weijie and Li, Shengwei and Lai, Zhiquan and Ge, Keshi and Chen, Qiaoling and Sun, Peng and Li, Dongsheng and Lu, Kai. AdaCheck: An Adaptive Checkpointing System for Efficient LLM Training with Redundancy Utilization. In *24th USENIX Conference on File and Storage Technologies (FAST)*, 2026.

[24] Jeffrey Ma, Hengzhi Pei, Leonard Lausen, and George Karypis. Understanding Silent Data Corruption in LLM Training. *arXiv Preprint ArXiv:2502.12340*, 2025.

[25] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, and Carole-Jean Wu. Understanding and Improving Failure Tolerant Training for Deep Learning Recommendation with Partial Recovery. *Proceedings of Machine Learning and Systems*, 3:637–651, 2021.

[26] Mai, Luo and Li, Guo and Wagenländer, Marcel and Fertakis, Konstantinos and Brabete, Andrei-Octavian and Pietzuch, Peter. {KungFu}: Making Training in Distributed Machine Learning Adaptive. In *Proceedings of 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[27] Llama Team @ Meta. The Llama 3 Herd of Models. *arXiv Preprint ArXiv:2407.21783*, 2024.

[28] Microsoft. Boost Checkpoint Speed and Reduce Cost with Nebula, 2024.

[29] Microsoft Research. DeepSpeed, 2024.

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

[31] Prashant J Nair, David A Roberts, and Moinuddin K Qureshi. Citadel: Efficiently Protecting Stacked Memory from TSV and Large Granularity Failures. *ACM Transactions on Architecture and Code Optimization*, 12(4):1–24, 2016.

[32] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger,

Phillip B Gibbons, and Matei Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[33] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[34] NVIDIA. CUDA Toolkit, 2007.

[35] NVIDIA. NVIDIA Collective Communications Library (NCCL), 2015.

[36] NVIDIA. NVIDIA A100 Tensor Core GPU, 2020.

[37] NVIDIA. NVIDIA H100 Tensor Core GPU, 2022.

[38] NVIDIA. NVIDIA NeMo Framework User Guide, 2024.

[39] Ataberk Olgun, Majd Osseiran, A Giray Yağlıkçı, Yahya Can Tuğrul, Haocong Luo, Steve Rhyner, Behzad Salami, Juan Gomez Luna, and Onur Mutlu. An Experimental Analysis of RowHammer in HBM2 DRAM Chips. In *Proceeding of the 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, 2023.

[40] OpenAI. GPT-4 Technical Report. *arXiv Preprint ArXiv:2303.08774*, 2023.

[41] A Paszke. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *arXiv Preprint ArXiv:1912.01703*, 2019.

[42] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based GPU Memory Management for Deep Learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[43] Poulos, Alexandra and Wallace, Dylan and Robey, Robert and Monroe, Laura and Job, Vanessa and Blanchard, Sean and Jones, William and DeBardeleben, Nathan. Improving Application Resilience by Extending Error Correction with Contextual Information. In *Proceedings of IEEE/ACM 8th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*, 2018.

[44] Guicheng Qi, Zongpeng Li, Chuan Wu, Zhuwei Peng, and Yi Zheng. ECCHECK: Enhancing In-Memory Checkpoint with Erasure Coding in Distributed DNN Training. In *Proceedings of the 45th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2025.

[45] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero Bubble (Almost) Pipeline Parallelism. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024.

[46] Aurick Qiao, Bryon Aragam, Bingjing Zhang, and Eric Xing. Fault Tolerance in Iterative-Convergent Machine Learning. In *Proceedings of International Conference on Machine Learning (ICML)*, 2019.

[47] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[48] Irving S Reed and Gustave Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[49] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC)*, 2021.

[50] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[51] Rizzo, Luigi. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM Computer Communication Review*, 27(2):24–36, 1997.

[52] Zhirong Shen, Yuhui Cai, Keyun Cheng, Patrick PC Lee, Xiaolu Li, Yuchong Hu, and Jiwu Shu. A Survey of the Past, Present, and Future of Erasure Coding for Storage Systems. *ACM Transactions on Storage*, 21(1):1–39, 2025.

[53] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv Preprint ArXiv:1909.08053*, 2019.

[54] 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. *arXiv Preprint ArXiv:2202.07848*, 2022.

[55] Foteini Strati, Michal Friedman, and Ana Klimovic. PCcheck: Persistent Concurrent Checkpointing for ML. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.

[56] Yu Tang, Qiao Li, Lujia Yin, Dongsheng Li, Yiming Zhang, Chenyu Wang, Xingcheng Zhang, Linbo Qiao, Zhaoning Zhang, and Kai Lu. DELTA: Memory-Efficient Training via Dynamic Fine-Grained Recomputation and Swapping. *ACM Transactions on Architecture and Code Optimization*, 21(4):1–25, 2024.

[57] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. In *Proceedings of USENIX 20th Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[58] Wagenländer, Marcel and Mai, Luo and Li, Guo and Pietzuch, Peter. Spotnik: Designing Distributed Machine Learning for Transient Cloud Resources. In *Proceedings of the 12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2020.

[59] Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, Liu Xin, and Wu Chuan. ByteCheckpoint: A Unified Checkpointing System for Large Foundation Model Development. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2025.

[60] Wan, Borui and Liu, Gaohong and Song, Zuquan and Wang, Jun and Zhang, Yun and Sheng, Guangming and Wang, Shuguang and Wei, Houmin and Wang, Chenyuan and Lou, Weiqiang and others. Robust LLM Training Infrastructure at ByteDance. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP)*, 2025.

[61] Wenhai Wang, Zhe Chen, Xiaokang Chen, Jiannan Wu, Xizhou Zhu, Gang Zeng, Ping Luo, Tong Lu, Jie Zhou, Yu Qiao, and Jifeng Dai. VisionLLM: Large Language Model is also an Open-Ended Decoder for Vision-Centric Tasks. *Advances in Neural Information Processing Systems*, 36:61501–61513, 2023.

[62] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, TS Eugene Ng, and Yida Wang. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

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

[64] BigScience Workshop. BLOOM: A 176b-Parameter Open-Access Multilingual Language Model. 2023.

[65] Jiannan Wu, Muyan Zhong, Sen Xing, Zeqiang Lai, Zhaoyang Liu, Zhe Chen, Wenhai Wang, Xizhou Zhu, Lewei Lu, Tong Lu, Ping Luo, Yu Qiao, and Jifeng Dai. VisionLLM V2: An End-to-End Generalist Multimodal Large Language Model for Hundreds of Vision-Language Tasks. *Advances in Neural Information Processing Systems*, 37:69925–69975, 2024.

[66] Ronglong Wu, Shuyue Zhou, Jiahao Lu, Zhirong Shen, Zikang Xu, Jiwu Shu, Kunlin Yang, Feilong Lin, and Yiming Zhang. Removing Obstacles Before Breaking Through the Memory Wall: A Close Look at HBM Errors in the Field. In *Proceedings of 2024 USENIX Annual Technical Conference (USENIX ATC)*, pages 851–867, 2024.

[67] Xiong, Yifan and Jiang, Yuting and Yang, Ziyue and Qu, Lei and Zhao, Guoshuai and Liu, Shuguang and Zhong, Dong and Pinzur, Boris and Zhang, Jie and Wang, Yang and others. SuperBench: Improving Cloud AI Infrastructure Reliability with Proactive Validation. In *Proceedings of 2024 USENIX Annual Technical Conference (USENIX ATC)*, 2024.

[68] Yao, Zhiyi and Hu, Pengbo and Miao, Congcong and Jia, Xuya and Liang, Zuning and Xu, Yuedong and He, Chunzhi and Lu, Hao and Chen, Mingzhuo and Li, Xiang and others. Holmes: Localizing Irregularities in LLM Training with Mega-Scale GPU clusters. In *Proceedings of 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2025.

[69] Zeng, Shaoxun and Ren, Tingxu and Shu, Jiwu and Lu, Youyou. GPU Checkpoint/Restore Made Fast and Lightweight. In *Proceedings of 24th USENIX Conference on File and Storage Technologies (FAST)*, 2026.

[70] 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. *arXiv Preprint ArXiv:2205.01068*, 2022.

[71] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *arXiv preprint arXiv:2304.11277*, 2023.

[72] Zheng, Wenxin and Xu, Bin and Gu, Jinyu and Chen, Haibo. SAVE:Software-Implemented Fault Tolerance for Model Inference against GPU Memory Bit Flips. In *Proceedings of 2025 USENIX Annual Technical Conference (USENIX ATC)*, 2025.

[73] 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)*, 2023.

[74] Shuyue Zhou, Xinbin Hu, Ronglong Wu, Jiahao Lu, Zhirong Shen, Zikang Xu, Yue Yu, Yuze Jiang, Jiwu Shu, Kunlin Yang, Feilong Lin, and Yiming Zhang. Looking back to move forward: Unveiling the mysteries of hbm errors to predict future failures. *ACM Transactions on Storage*, 2025.

A Artifact Appendix

Abstract

This artifact provides the implementation and evaluation scripts for GEM. GEM is a GPU memory fault-tolerance system that combines selective erasure coding with a flexible retry mechanism to enable fast recovery from GPU memory failures. First, GEM employs wide-stripe erasure coding to protect model states that are updated relatively infrequently, thereby balancing memory overhead and fault-tolerance capability. Second, GEM introduces a mixed-granularity retry mechanism that supports recomputation at either the training-phase level (e.g., the forward pass) or the training-step level. This design balances the memory overhead of preserving replicated training inputs against the computational overhead of recomputation.

Scope

This artifact is intended to validate the main experimental claims of GEM under the model-training workloads and hardware configurations evaluated in the paper. In particular, the artifact can be used to evaluate the following aspects:

- the training overhead introduced by GEM during failure-free execution;
- the effectiveness of GEM in recovering from GPU memory failures;
- the memory overhead introduced by GEM's selective erasure-coding and retry mechanisms; and
- the impact of different models on training and recovery performance.

Contents

The artifact contains the implementation of GEM, including its modifications to DeepSpeed, the CUDA-based erasure-coding shared library, and the scripts and configurations used for artifact evaluation. The main directory structure is as follows:

```
gem/
|-- data/
|-- deepspeed_codes/
|-- EC_so/
|-- examples/
|-- requirements.txt
`-- README.md
```

The directories and files are described below:

- `data/`: contains the training and evaluation datasets used by the model-training workloads included in the artifact.
- `deepspeed_codes/`: contains GEM's modifications to DeepSpeed. These modifications integrate selective erasure coding, preserve the model states required for recovery, and perform retry-based recomputation after a simulated GPU memory failure.
- `EC_so/`: contains the CUDA-based erasure-coding shared library used by GEM to generate parity chunks and recover protected model states on GPUs.
- `examples/`: contains the scripts and configurations used for artifact evaluation, including the model-training, failure-recovery, and result-processing scripts.
- `requirements.txt`: lists the Python packages required to configure the artifact environment.
- `README.md`: provides detailed instructions for environment setup, experiment execution, and result reproduction.

Hosting

The GEM artifact is publicly available on GitHub at https://github.com/wrl297/gem. Artifact evaluation should be performed using the latest version of the `main` branch. Reviewers can obtain the artifact using the following commands:

```
git clone https://github.com/wrl297/gem.git
cd gem
git checkout main
git pull origin main
```

Requirements

Hardware requirements. GEM requires NVIDIA GPUs with CUDA support. The complete experimental evaluation reported in the paper was conducted on a two-node GPU platform. Each node is equipped with two NVIDIA A100 GPUs with 80 GB of HBM per GPU, two Intel Xeon Gold 5320 CPUs, and 512 GB of system memory. Intra-node GPU communication uses peer-to-peer transport, whereas inter-node communication is provided through InfiniBand RNICs. Because GEM primarily performs recovery using GPU memory within the same node, most artifact experiments require only one node with two NVIDIA A100 80 GB GPUs. A second similarly configured node is required only for experiments that evaluate multi-node training or recovery.

Software requirements. The artifact was developed and tested on Ubuntu 22.04.5 LTS with Python 3.9.21, CUDA 12.4, NCCL 2.21.5, PyTorch 2.6.0, and DeepSpeed 0.15.4. The complete set of Python dependencies is provided in `requirements.txt`. We recommend using a Conda environment to avoid conflicts with existing Python packages. The main Python dependencies include Accelerate 1.2.0, Transformers 4.45.0, PEFT 0.14.0, Importlib Metadata 8.6.1, and Matplotlib 3.9.4.