AsymCheck: Asymmetric Partitioned Checkpointing for Efficient Large Language Model Training

Zhangqiang Ming[1,3], Yuchong Hu[1,*], Zhiyuan Luo[1], Patrick P. C. Lee[2], Yuanhao Shu[3], Xinjue Zheng[1], Wenxiang Zhou[1], Dan Feng[1]
[1]*Huazhong University of Science and Technology, Wuhan, China*
[2]*The Chinese University of Hong Kong, Hong Kong, China*
[3]*Innovation Research Institute of Cethik Group Co., Ltd., Hangzhou, China*
{zqming, yuchonghu, zhiyuanluo, wxzhoucs, zhengxinjue, dfeng}@hust.edu.cn,
pclee@cse.cuhk.edu.hk, yuanhaoshu@cethik.com

ABSTRACT

Distributed large language model (LLM) training faces prevalent failures and requires efficient checkpointing. State-of-the-art approaches employ partition-based pipelined checkpointing, splitting checkpoints into partitions for concurrent processing. However, existing solutions rely on a *fixed* partition size, which our analysis reveals is suboptimal for LLM training: large partitions cause bandwidth stalls during forward passes, while small partitions incur substantial startup overhead during backward passes. We propose AsymCheck, which employs asymmetric partitioning: small partitions for forward passes and large partitions for backward passes, plus selective partition compression and batched flushing optimizations. Evaluation on 64 GPUs shows AsymCheck reduces training time by 20.1%-48.2% over state-of-the-art methods, approaching no-checkpointing efficiency.

1 INTRODUCTION

Distributed large language models (LLM) training with massive parameters (e.g., PaLM's 540B parameters [9]) requires robust fault tolerance due to prevalent failures; for example, Llama3 encounters 466 interruptions over 54 days [11], and OPT-175B suffers 112 failures over 90 days [1]. *Checkpointing* is a common fault tolerance approach by regularly saving model states to persistent storage, yet traditional checkpointing approaches incur significant overhead under limited storage bandwidth. Recent partitioned pipelined checkpointing solutions [20, 32, 34] divide checkpoints into *partitions* for efficient processing, but rely on *fixed-size* partitioning.

Our analysis (§3.2) reveals that fixed-size partitioning is suboptimal in modern distributed LLM training frameworks with state sharding [28, 30, 40]: large partitions cause bandwidth contention during forward passes, while small partitions incur excessive copying startup overhead during backward passes. We observe that backward passes benefit from larger partitions, whereas forward passes favor smaller ones. This motivates our main idea (§3.3) of *asymmetric* checkpointing, which employs small partitions during forward passes and large partitions during backward passes.

We propose AsymCheck (§4), which performs **asym**metric partitioned **check**pointing between forward and backward passes. AsymCheck uses smaller partitions for forward passes and larger partitions for backward passes, reducing both bandwidth contention in forward passes and frequent checkpoint copying and startup overhead in backward passes, thereby mitigating training stalls.

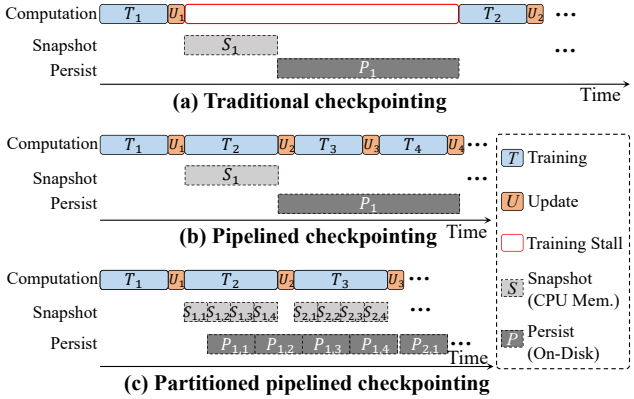


Figure 1: Comparison of different checkpointing solutions.

Further, we propose a selective partition compression scheme and an optimal batched flushing scheme to enhance performance.

We implement a distributed LLM training framework, with a checkpointing library containing AsymCheck and state-of-the-art methods, atop PyTorch [2] and DeepSpeed ZeRO-3 [4]. Evaluations (§5) on 64 GPUs show AsymCheck reduces training time by 20.1%–48.2% and checkpointing time by 62.9%–80.1% compared to state-of-the-art checkpointing solutions. AsymCheck is open-sourced at https://github.com/DAC26-AsymCheck.

2 BACKGROUND AND RELATED WORK

LLMs require distributed training across multiple GPUs, operating on datasets across multiple *epochs*, each with multiple *iterations*. Each iteration has three phases: *forward pass* (prediction computation), *backward pass* (gradient calculation and synchronization), and *parameter updates* (adjusting model weights using gradients).

Modern distributed training frameworks like DeepSpeed ZeRO-3 [30] employ *state sharding* [28] to partition model states across GPUs, optimizing memory usage. This introduces cross-GPU communication: forward passes require All-Gather operations to reconstruct parameters, while backward passes involve reverse-order All-Gather and Scatter-Reduce for gradient synchronization.

Checkpointing enables fault-tolerant distributed LLM training by persisting model states for recovery. There are three main checkpointing approaches (Figure 1):

- **Basic checkpointing** saves states at epoch boundaries, but epoch-level granularity wastes hours of computation during re-

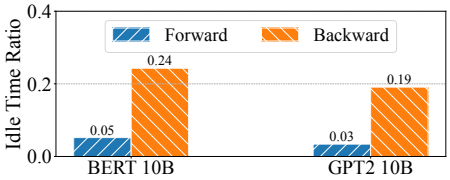
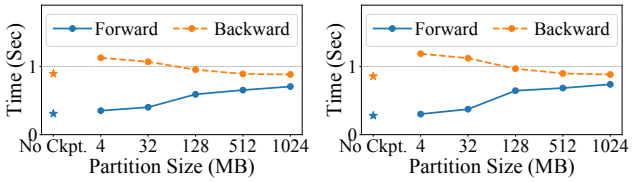


Figure 2: Forward and backward communication idle periods.



(a) BERT 10B on SQuAD **(b) GPT2 10B on WikiText-103**

Figure 3: Forward and backward training time per iteration.

covery. *Iteration-level* checkpointing improves granularity but causes training stalls due to bandwidth limits (e.g., iteration T_2 is blocked by its preceding iteration T_1 in Figure 1(a)).

- **Pipelined checkpointing** enables parallelization [20, 25, 26] by decoupling checkpointing into: (i) *snapshot* phase that pipelines GPU-to-CPU state transfer with tensor computation, and (ii) asynchronous *persistence* phase that flushes snapshots to storage (e.g., snapshot S_1 overlaps with iteration T_2 in Figure 1(b)).
- **Partitioned pipelined checkpointing** splits checkpoints into *partitions* for finer-grained pipelining [20, 32, 34]. This allows each partition to persist independently upon snapshot completion, reducing latency (e.g., snapshot S_1 is divided into partitions $S_{1,1}$-$S_{1,4}$ where each partition triggers its persistence as soon as its snapshot finishes in Figure 1(c)).

We consider three state-of-the-art partitioned pipelined checkpointing solutions: (i) *Gemini* [34] partitions checkpoints into 32 MB chunks pipelined through GPU sub-buffers during network idle periods; (ii) *DataStates-LLM* [20] uses three fixed partitions (parameters and optimizer states) copied to pre-allocated CPU buffers; (iii) *PCcheck* [32] uses 100–500 MB partitions with pipelined host transfers and parallel I/O persistence. Despite various optimizations, all the above partitioned pipelined checkpointing approaches employ static partitioning that *fixes* the partition size (§3.2), ignoring the distinct features of forward and backward passes (§3.1).

3 ANALYSIS AND MOTIVATION

3.1 Analysis of Communication Idleness

We characterize forward/backward pass features via *communication idle periods*, which arise from the overlap between shard transfers and GPU computations in state-sharding frameworks (e.g., DeepSpeed ZeRO-3 [28, 30]). These frameworks organize shards into *buckets* [4] and must wait for all shards within each bucket to be ready before transfers, thus creating idle intervals [34].

We examine the communication idle periods using two LLM training tasks: GPT2 10B [27] on WikiText-103 [23] and BERT 10B [10] on SQuAD [29], both implemented with DeepSpeed ZeRO-3. Figure 2 shows that the *idle time ratio* (i.e., communication idle time relative to total training time) is much higher in backward passes (up to 24%) than in forward passes (up to 5%). This reveals

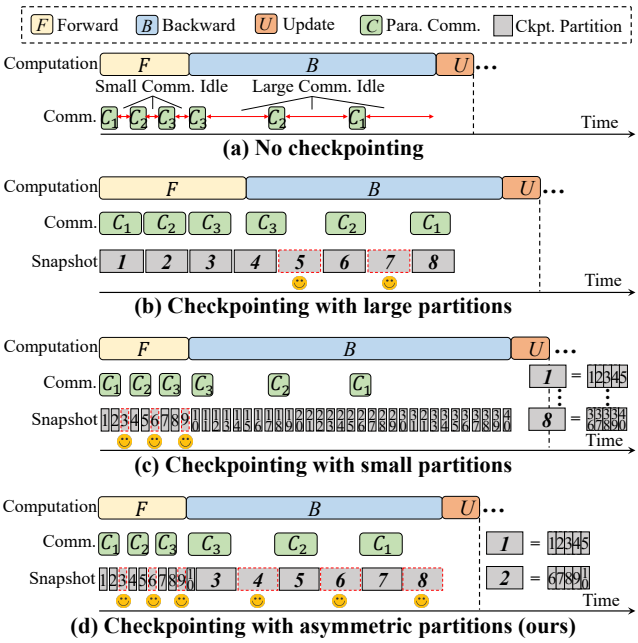


Figure 4: Motivating example: comparison of different partitioned checkpointing solutions.

asymmetric communication idleness between forward and backward passes. As shown in Figure 4(a), a forward pass involves multiple All-Gather operations (C_1 to C_3) for parameter retrieval with short idle spans, while a backward pass executes them in reverse (C_3 to C_1) with much longer idle periods. Thus, the backward pass experiences significantly longer idle periods than the forward pass.

Asymmetric communication idleness stems from two key factors. First, a backward pass involves more intensive computation of gradient derivatives than a forward pass, which only computes model outputs [10, 14, 18, 27]. This leads to long communication idle periods. Second, while both passes require All-Gather operations, a backward pass also performs Reduce-Scatter operations for gradient synchronization (§2) and must wait for all gradient computations to complete, further prolonging its idle periods.

3.2 Limitations of Fixed-Size Partitioning

We further analyze how the partition size affects forward and backward pass performance (under the same setting as §3.1). As shown in Figure 3, for both LLM training tasks, (i) the backward training time increases as the partition size decreases and peaks at the smallest partition (4 MB); and (ii) the forward training time increases as the partition size increases and peaks at the largest partition (1024 MB). This reveals that fixed-size partitioning leads to two limitations for large and small partitions:

- **Limitations for large partitions:** In Figure 4(b), the checkpoint is divided into eight large partitions for snapshotting. Such large partitions significantly increase the forward pass duration compared to no-checkpoint training (Figure 4(a)), since state sharding causes the forward pass to introduce additional parameter communication via All-Gather operations. When large partitions are copied from GPU to CPU memory during a forward pass,

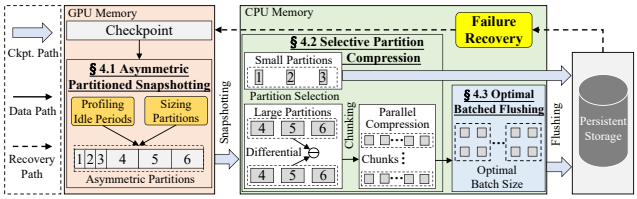
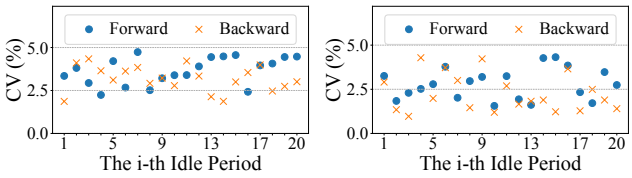


Figure 5: Architecture of AsymCheck.



(a) BERT 10B on SQuAD **(b) GPT2 10B on WikiText-103**

Figure 6: Stable idle periods within forward/backward passes.

they *continuously* occupy PCIe bandwidth [15] and compete with parameter communications and data prefetching for the next mini-batch [20–22]. Such contention creates stalls and degrades training performance.

- **Limitations for small partitions:** In Figure 4(c), the checkpoint is divided into 40 small partitions for snapshotting. Small partitions substantially increase backward pass time compared to no-checkpoint training (Figure 4(a)), since each partition triggers an independent `copy_` from GPU to CPU memory. This leads to frequent small transfers with high startup overhead [19–22, 33]. Also, small partitions trigger excessive memory allocations, leading to CPU memory fragmentation and substantial overhead from frequent `malloc` and `free` operations [5].

3.3 Our Idea

Asymmetric communication idleness (§3.1) and the limitations of fixed-size partitioning (§3.2) reveal opportunities for improving LLM training performance: the longer communication idle periods from backward passes can accommodate large partitions (e.g., partitions 5 and 7 in Figure 4(b)), while the smaller communication idle periods from forward passes can accommodate small partitions (e.g., partitions 3, 6, and 9 in Figure 4(c)). As shown in Figure 3, both the forward pass time at the smallest partition size (4 MB) and the backward pass time at the largest partition size (1024 MB) are close to the no-checkpoint training time.

Leveraging this insight, we propose AsymCheck, an **asym**metric partitioned **check**pointing mechanism that decouples forward and backward partition sizes, in contrast to existing schemes with fixed-size partitions across both passes, so as to mitigate training stalls by adapting partition sizes (i.e., small partitions for forward passes; large partitions for backward passes). As shown in Figure 4(d), small partitions 3, 6, and 9 fit short idle spans in a forward pass, while large partitions 4, 6, and 8 fit long idle spans in a backward pass.

4 DESIGN AND IMPLEMENTATION

AsymCheck realizes the main idea (§3.3) with the following goals:

- **Facilitating asymmetric snapshotting.** AsymCheck explores stable idleness to optimize partition sizes via profiling (§4.1).
- **Reducing compression overhead:** AsymCheck uses partition asymmetry to selectively compress checkpoints at low cost (§4.2).
- **Minimizing flushing time:** AsymCheck minimizes the flushing time to persistent storage via optimal batching (§4.3).

Figure 5 shows AsymCheck's system architecture, comprising the above three modules, as well as a common failure recovery module that decompresses checkpoints and resumes training [8, 17, 37].

4.1 Asymmetric Partitioned Snapshotting

A key challenge of asymmetric partitioning is to carefully partition snapshots into sizes that match asymmetric idle periods. To address this challenge, we first systematically analyze bucketized communication patterns.

Analysis of communication idle periods. As described in §3.1, distributed LLM frameworks typically group multiple parameters into buckets during forward and backward passes, performing All-Gather to enhance communication efficiency [16, 30, 31]. Notably, the bucket sizes used for All-Gather remain fixed throughout training (e.g., ZeRO-3 uses a default `allgather_bucket_size`=0.5 B [4], corresponding to 0.5 billion `float16` elements per bucket).

We observe that fixed-size bucketing produces a *stable distribution* of idle periods across iterations; that is, each iteration's i-th idle period maintains nearly identical length. This stability stems from two facts: (i) each iteration's i-th bucket consistently processes the same parameters under a given parameter partitioning scheme [30], and consequently (ii) the idle waiting time (§3.1) required to fill the fixed-size i-th bucket remains consistent across iterations.

Our measurements (under the same setting as §3) confirm this stability using the *coefficient of variation* (CV), a statistical metric defined as the ratio of the standard deviation to the mean, which is widely used to quantify stability. As shown in Figure 6, the i-th communication idle period during forward/backward passes exhibits a small CV (less than 5%) across 200 iterations, where $1 \le i \le 20$, with 20 derived from a 10B model and 0.5B buckets.

The stability of the communication idle periods in both forward and backward passes motivates our profiling-based partitioning scheme, which consists of three steps:

Step 1: Profiling idle periods. We establish profiling by skipping checkpoint operations during the initial 10 iterations to measure baseline patterns. Using DeepSpeed ZeRO-3's hooks `pre_forward_module_hook` and `pre_backward_module_hook` [3], we measure idle intervals by timestamping the completion of the forward pass and initiation of the backward pass across consecutive buckets. For the forward pass with M idle periods, the i-th idle period T_f^i is the time between the i-th bucket's communication end and the $(i+1)$-th bucket's start ($1 \le i \le M$). Similarly, T_b^j ($1 \le j \le N$) is computed for the backward pass with N idle periods. Total forward and backward idle times $T_f = \sum_{i=1}^{M} T_f^i$ and $T_b = \sum_{j=1}^{N} T_b^j$ provide stable baselines for partitioning.

Step 2: Sizing snapshot partitions. We use PyTorch's `torch.narrow()` to dynamically size partitions to fit profiled idle periods:

- *Forward partition size*: $R_f^i = R \times \frac{T_f^i}{T_f + T_b}$ for $i \in [1, M]$,
- *Backward partition size*: $R_b^j = R \times \frac{T_b^j}{T_f + T_b}$ for $j \in [1, N]$,

where R is the total checkpoint size, with R_f^i and R_b^j denoting the

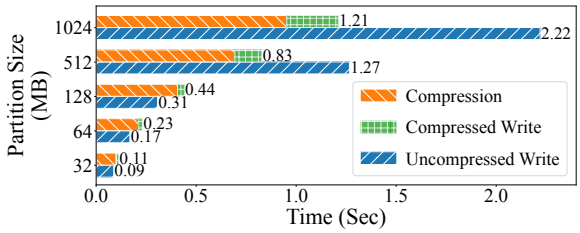


Figure 7: Comparison of compressed and uncompressed write times for different partition sizes.

i-th forward and j-th backward partition sizes, respectively.

Step 3: Asynchronous copying partitions. We pre-allocate CPU buffers sized to partition dimensions to speed up checkpointing. Each partition is copied to CPU memory via CUDA streams in a pipeline. We use PyTorch's `copy_()` with `non_blocking` enabled, allowing asynchronous transfers alongside training during communication idle periods.

4.2 Selective Partition Compression

Issues of LLM checkpoint compression. Recall from §2 that frequent iteration-level checkpointing is crucial for fast failure recovery [25, 34]. However, terabyte-scale checkpoints in LLM training [39] cause severe bandwidth contention during writes, significantly increasing flushing time. Ultimately, this prolongs the total time to complete model state checkpoints (referred to as *checkpointing time* [34]), and reduces the achievable checkpointing frequency [12, 32, 34]. Recent studies on reducing checkpointing time have developed *differential compression* [8, 12, 17, 37] to reduce checkpoint size by compressing inter-version differences. However, its high computational cost can significantly stall training [12, 17]. Thus, we aim to reduce AsymCheck's compression overhead.

Selective compression for asymmetric partitions. Asymmetric partitioning (§4.1) shows a size-dependent trade-off (Figure 7): large partitions (512–1024 MB) benefit from compression, while small ones (e.g., 64 MB) incur 35% longer write time (0.23s vs. 0.17s), since the computational overhead of compressing small partitions may negate the benefits of reduced write time from compression. This trade-off motivates AsymCheck's *selective compression* strategy: (i) large partitions are compressed before persistence (§2) to maximize flush throughput; and (ii) small partitions bypass compression to avoid computational overhead, written directly to storage. The threshold between them follows a time-saving condition:

Compression Time + Compressed Write Time < Uncompressed Write Time,

which ensures compression benefits outweigh its costs.

Specifically, we first treat each snapshot partition as a set $S = [S_1, S_2, \ldots, S_{M+N}]$. For each partition S_i, we compute three metrics:

- Compression time: $T_c^i = \alpha_c + \beta_c \cdot |S_i|$, $1 \le i \le M+N$,
- Compressed write time: $T_{cw}^i = \alpha_w + \beta_w \cdot |\widetilde{S_i}|$,
- Uncompressed write time: $T_w^i = \alpha_w + \beta_w \cdot |S_i|$,

where α_c and β_c denote the startup and per-byte costs of compression, α_w and β_w represent the startup and per-byte costs of write operations, $|S_i|$ is the partition size, and $\widetilde{S_i}$ is the compressed version. The parameters α_c, β_c, α_w, and β_w can be profiled offline following established analysis frameworks [35, 36].

Finally, a partition S_i is selected as large if it satisfies $T_c^i + T_{cw}^i < T_w^i$ (i.e., the time-saving condition); otherwise, it is treated as small.

Parallel compression for large partitions. While selective compression reduces overhead, large partitions still incur non-negligible compression time. Inspired by existing methods [7, 24], which accelerate compression by chunking gradients and parallelizing across chunks, we chunk large partitions for parallel compression.

Specifically, for the selected (large) partitions, we first compute their differential $\triangle S_t^i = S_t^i - S_{t-1}^i$, representing the i-th large partition difference between the t-th and the $(t$-1)-th checkpoints. Then, we use PyTorch's `torch.chunk()` function to split $\triangle S_t^i$ into P chunks, where P is typically set to the number of CPU cores to maximize parallel efficiency. Finally, we employ PyTorch's `torch.topk()` function to compress each of P chunks with a compression ratio of 0.01 ([17, 38]) and use Python's `ThreadPoolExecutor` to perform parallel compression on the P chunks.

Convergence discussion. Prior work on differential checkpoint compression [17] proves end-to-end convergence. In contrast, AsymCheck only selectively compresses part of the differential checkpoints (i.e., large partitions), so its selective compression is more likely to preserve convergence under compression. We will verify AsymCheck's convergence accuracy in Experiment 4 (§5.2).

4.3 Optimal Batched Flushing

Small file handling challenges in AsymCheck. As described in §4.2, all large partitions are split into multiple compressed chunks (32 chunks by default, matching the 32 cores per CPU) as small files, so AsymCheck may incur significant I/O overhead from flushing these numerous small files to persistent storage. Such small writes generate excessive I/O at iteration-level checkpointing frequencies (§2). AsymCheck addresses this via *batched flushing*, which aggregates multiple writes into a single operation; the key challenge is determining the *optimal* batch size to minimize flush time.

Characterizing the flushing time. We first define the parameters: storage write bandwidth W, size of the compressed chunk c, total number of chunks N_c, average compression time per chunk T_c, and fixed startup write overhead per batch T_s. The following expressions are based on the variable B (i.e., the batch size) and the constant system parameters above:

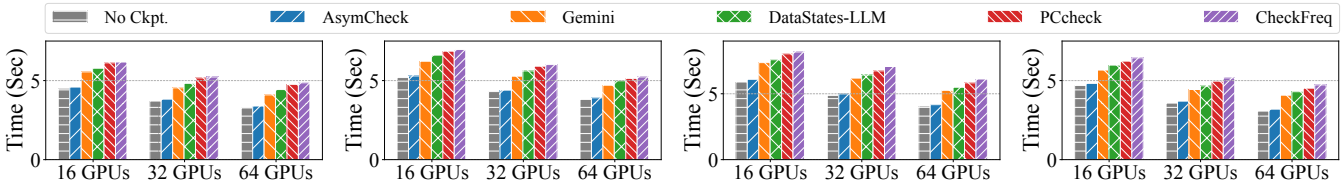
- Total compressed checkpoint size: $C = N_c \cdot c$;
- Disk transfer time of the compressed checkpoint: $\frac{C}{W}$;
- Number of batches: $\frac{C}{B}$;
- Total startup time across all batches: $\frac{C}{B} \cdot T_s$;
- Total write time of all batches: $T_{write} = \frac{C}{W} + \frac{C}{B} \cdot T_s$;
- Number of compressed chunks merged per batch: $\frac{B}{c}$;
- Compression time for the first batch: $T_{first} = \frac{B}{c} \cdot T_c$.

When compression is parallelized with persistence (§4.2), the persistence phase must *wait* for the first batch's compression to complete before starting, while the flushing of subsequent batches overlaps with compression. Thus, the total flushing time T_{flush} can be calculated as:

$$T_{flush} = T_{write} + T_{first} = \frac{C}{W} + \frac{C}{B} \cdot T_s + \frac{B}{c} \cdot T_c. \quad (1)$$

To optimize B that minimizes T_{flush}, we take the derivative of T_{flush} with respect to B:

$$\frac{\partial T_{flush}}{\partial B} = -\frac{C}{B^2} \cdot T_s + \frac{T_c}{c} = 0, \quad (2)$$



(a) GPT2 40B (b) BERT 40B (c) RoBERTa 40B (d) BLOOM 40B

Figure 8: Average training time per iteration on four LLMs (Exp. 1).

Table 1: Configurations of LLM variants. #AH is the number of attention heads. #Layers is the number of layers.

Models Size	Hidden Size	Intermediate	#AH	#Layers
GPT2 10/20/40B	2560/5120/5120	10k/20k/20k	40	46/64/128
BERT 10/20/40B	2560/5120/5120	10k/20k/20k	40	46/64/128
BLOOM 40B	5120	20k	40	128
RoBERTa 40B	5120	20k	40	128

which results in the optimal batch size B^* as follows

$$B^* = \sqrt{\frac{C \cdot T_s \cdot c}{T_c}}. \quad (3)$$

Batching compressed chunks for large partitions. We merge multiple compressed checkpoint chunks from large partitions into buffers sized by the optimal batch size B^* derived from Equation (3). We implement batched I/O operations by concatenating multiple compressed chunks into unified buffers via `torch.cat()`, which are then managed through a `torch.multiprocessing.Queue()` allocated in CPU memory to ensure sequential persistence. Additionally, we maintain model fidelity through periodic full checkpoints (typically at epoch boundaries) alongside frequent compressed snapshots, similar to the approach in [12].

Asynchronous flushing for all partitions. We create an independent flushing process to a shared CPU memory queue for flushing operations asynchronously [20, 33]: (i) both (uncompressed) small partitions and batched compressed chunks of large partitions are first staged in the queue; and (ii) the flushing process then commits the queue to persistent storage asynchronously.

5 EVALUATION

5.1 Evaluation Setup

Testbed. We evaluate on two GPU clusters: (i) a cloud cluster with 64 GPUs across 16 nodes interconnected via 200 Gbps InfiniBand, each node equipped with 1 TB CPU memory, 32-core AMD EPYC 7543 CPU, dual 1.6 TB SSDs, and 4 NVIDIA A100 80 GB GPUs; (ii) a local cluster with 4 nodes via 100 Gbps InfiniBand, each node containing 512 GB CPU memory, dual 32-core Intel Xeon Platinum 8352V CPUs, 1.8 TB SSD, and 2 NVIDIA A100 80 GB GPUs. All nodes employ PCIe Gen4 for GPU interconnects and HDFS v3.3.6 [6] for persistent storage, running Ubuntu 20.04 with CUDA v12.6, NCCL v2.20.5, PyTorch v2.4.0, and DeepSpeed v0.15.3.

Workloads. We use popular LLMs [34], creating variants by adjusting hidden/intermediate sizes, attention heads, and layers (Table 1). All models use Adam [13], with BERT trained on SQuAD [29] and others on WikiText-103 [23].

Compared approaches. We compare AsymCheck against partitioned pipelined checkpointing solutions: Gemini [34], DataStates-LLM [20], PCcheck [32] and the pipelined method CheckFreq [25], using DeepSpeed ZeRO-3 [28, 30] as the no-checkpoint *baseline*.

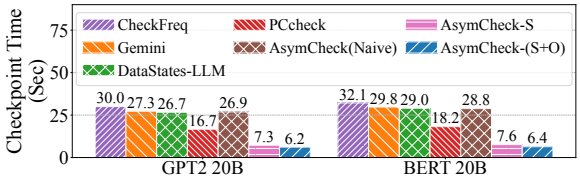


Figure 9: Checkpointing time (Exp. 2).

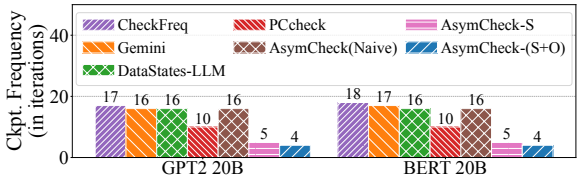


Figure 10: Checkpoint frequency (Exp. 3).

5.2 Experimental Results

Experiment 1 (Training time): We run 1000 training iterations on four 40B LLMs, with the per-iteration snapshot frequency as in [34]. Figure 8 shows AsymCheck reduces training time by 20.1%–48.2% versus state-of-the-art methods. Its training time increases by only 1.7%–3.7% over the baseline, compared to 22.3%–54.1% for others.

Analysis: We elaborate on AsymCheck's benefits over the state-of-the-art solutions and analyze the reasons behind.

(1) AsymCheck reduces training time by 25.9%-48.2% over CheckFreq and PCcheck. CheckFreq performs the worst as it lacks partitioning for pipelining (§2). Although PCcheck uses partitions, it fixes a large partition size (≥ 100 MB) (§2), stalling forward passes.

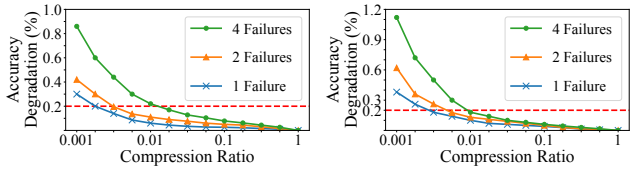
(2) DataStates-LLM outperforms CheckFreq and PCcheck by buffering partitions. However, it splits each checkpoint into only three large partitions (§2), which still stalls forward passes.

(3) Gemini outperforms the above by splitting checkpoints into small partitions to fit idle timespans, but it reserves a 32 MB GPU buffer for each partition (§2), resulting in small partitions (≤ 32 MB) that stalls backward passes (§3.2).

(4) AsymCheck performs the best, reducing the training time by 20.1%–27.6% compared to Gemini across all models via asymmetric partition checkpointing (§4.1), which uses small and large partitions for forward and backward passes, respectively, to optimize both.

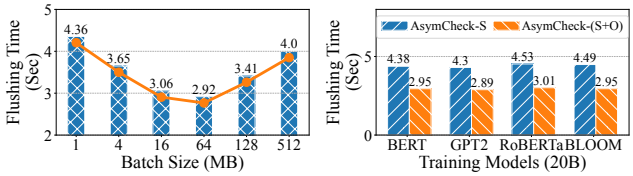
Experiment 2 (Checkpointing time): We evaluate checkpointing time (defined in §4.2) across 3 AsymCheck variants: AsymCheck(Naive) (asymmetric partitioning in §4.1), AsymCheck-S (plus selective compression in §4.2), and AsymCheck-(S+O) (further incorporating optimal flushing in §4.3).

Figure 9 shows AsymCheck-(S+O) reduces checkpointing time by 62.9%-80.1% versus the state-of-the-arts, and outperforms AsymCheck(Naive) and AsymCheck-S. We analyze the reasons as follows: (1) CheckFreq performs the worst because it lacks partitioned checkpoints to pipeline snapshot and persistence, requiring the full snap-



(a) BERT 10B on SQuAD (b) GPT2 10B on WikiText-103

Figure 11: Convergence accuracy degradation (Exp. 4).



(a) Flushing time on GPT2 20B (b) Optimal batched flushing time

Figure 12: Flushing time (Exp. 5).

shot copy to complete before persistence (§2).

(2) Gemini, DataStates-LLM, and AsymCheck(Naive) have close checkpointing time, meaning that their partitioned checkpointing fairly improves snapshot efficiency. However, they have longer checkpointing time than PCcheck, because PCcheck employs concurrent persistence to improve write efficiency (§2).

(3) AsymCheck-S reduces checkpointing time by up to 73.61% over AsymCheck(Naive) via selective compression (§4.2). AsymCheck-(S+O) further reduces checkpointing time by up to 15.79% over AsymCheck-S due to further optimizing batched flushing (§4.3).

Experiment 3 (Checkpoint frequency): We evaluate maximum checkpointing frequency achievable by each method under a 3.5% training speed degradation bound, following Microsoft's study [25].

Figure 10 shows AsymCheck-S and AsymCheck-(S+O) checkpoint every 5 and 4 iterations, respectively, compared to 10–18 iterations for other schemes, confirming selective compression's effectiveness (§4.2) in reducing persistence time and thus improving frequency. AsymCheck-(S+O)'s higher frequency than AsymCheck-S stems from batch optimization (§4.3).

Experiment 4 (Convergence accuracy): To evaluate AsymCheck's selective partition compression impact (§4.2) on convergence accuracy, we measure convergence accuracy degradation after failure recovery for BERT 10B/SQuAD and GPT2 10B/WikiText-103 under varying compression ratios (0.001–1) and failure counts (1, 2, 4).

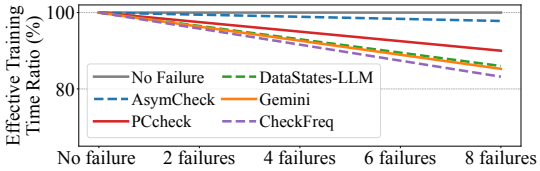
Figure 11 shows that with one failure, AsymCheck's accuracy degradation ranges from 0.012%–0.27% and 0.006%–0.38% across tasks as compression ratio decreases from 0.1 to 0.001; with four failures, degradation increases to 0.026%–0.86% and 0.016%–1.09%. At the 0.01 compression ratio used in AsymCheck (§4.2), degradation remains near 0.2% even under multiple failures, very close to the no-compression baseline (i.e., compression ratio = 1).

Experiment 5 (Flushing time): To evaluate AsymCheck's optimal batched flushing impact (§4.3) on flushing time, we conduct microbenchmark experiments across: (i) varying batch sizes, and (ii) with/without optimal batching.

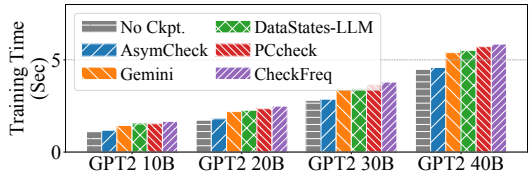
Figure 12(a) shows that flushing time first decreases, then increases with the batch size: from 4.36s at 1MB to 2.92s at 64MB, then to 4.0s at 512MB. This U-shaped trend occurs because small batches amplify flushing startup overhead, while large batches increase first-batch compression latency (§4.3), creating an optimal batch size that minimizes the total flushing time.

Table 2: Breakdown of recovery time (Exp. 6).

Checkpoint Solutions	GPT2 20B			BERT 20B		
	Recovery Time (Sec)			Recovery Time (Sec)		
	T_{rb}	T_{rr}	Total	T_{rb}	T_{rr}	Total
AsymCheck	11.6	3.4	15.0	10.6	3.7	14.3
PCcheck	9.2	9.3	18.5	8.5	9.3	17.8
DataStates-LLM	9.1	12.9	22.0	8.6	14.1	22.7
Gemini	8.9	13.8	22.7	8.5	15.7	24.2
CheckFreq	8.7	15.5	24.2	8.2	16.7	24.9



(a) Scaling to more failures



(b) Scaling to different model sizes

Figure 13: System scalability (Exp. 7).

Figure 12(b) shows that, compared to AsymCheck without batching (AsymCheck-S), AsymCheck with optimal batching (AsymCheck-(S+O)) significantly reduces flushing time for writing batched chunks, achieving a maximum reduction of 34.3% across four LLMs.

Experiment 6 (Recovery time breakdown): We evaluate the recovery time of AsymCheck and existing methods under training failures, which is defined as T_{rb} (rollback time: restoring the last checkpoint) + T_{rr} (rerun time: reprocessing lost work since the last checkpoint), following [25]. Specifically, for the rollback process, AsymCheck requires decompressing the last checkpoint via its failure recovery module (Figure 5), while other methods only read it from disk. For the rerun process, we consider best-case ($T_{rr} = 0$, failure right after checkpoint) and worst-case ($T_{rr} = f \cdot T_{iter}$, failure right before checkpoint) scenarios, using the frequency f (Exp. 3) and the iteration training time T_{iter} (Exp. 1). Thus, the average rerun time is calculated as $T_{rr} = f \cdot T_{iter}/2$.

Table 2 shows that AsymCheck reduces recovery time by 18.9%-42.6% over other methods. This improvement holds despite a modest increase in rollback time (e.g., an increase of 2.4s over PCcheck for GPT2 20B), stemming from the need to decompress checkpoints and reconstruct the latest model state. The key driver of the overall reduction is AsymCheck 's frequent checkpointing, which drastically shortens rerun times (e.g., a reduction of 5.9s compared with PCcheck for GPT2 20B), outweighing the added rollback cost.

Experiment 7 (System scalability): We evaluate the scalability of AsymCheck under scenarios involving different numbers of failures and model sizes in a cloud cluster.

(1) *Scaling to more failures.* We use the effective training time ratio metric [34] (i.e., the percentage of productive training progress achieved within a given period). Figure 13(a) shows AsymCheck maintains 97.8% efficiency with 8 failures, approaching the no-failure baseline and outperforming other methods, demonstrating AsymCheck's high training efficiency even under frequent failures.

(2) *Scaling to different model sizes.* We evaluate the average training

time per iteration with GPT2 model sizes ranging from 10B to 40B on a cloud cluster. Figure 13(b) shows that AsymCheck reduces training time by 20.6%–35.2% compared to others across different model sizes while remaining close to the no-checkpoint baseline. This indicates that AsymCheck exhibits consistent performance advantages across different model scales and has the potential to scale to even larger models.

6 CONCLUSION

In this paper, we propose AsymCheck, an asymmetric partitioned checkpointing mechanism, which uses small partitions for forward passes and large partitions for backward passes, tailored for efficient distributed large language model training. We also propose a selective partition compression scheme and an optimal batched flushing scheme to enhance AsymCheck. Experimental evaluations across cloud and local clusters demonstrate that AsymCheck outperforms state-of-the-art checkpointing solutions by reducing training time by 20.1%-48.2%, while maintaining training efficiency and convergence accuracy close to the no-checkpoint baseline.

ACKNOWLEDGMENTS

We thank all the anonymous reviewers for their constructive comments. This work was supported by the National Key R&D Program of China (No. 2022YFB4501300), the National Natural Science Foundation of China (No. 62272185), and the Key Laboratory of Information Storage System, Ministry of Education of China. The corresponding author is Yuchong Hu.

REFERENCES

[1] 2022. OPT-175B Logbook. https://github.com/facebookresearch/metaseq/tree/main/projects/OPT/chronicles.

[2] 2024. PyToch. https://pytorch.org/.

[3] 2024. Zero Documentation. https://deepspeed.readthedocs.io/en/latest/zero3.html.

[4] 2025. DeepSpeed. https://github.com/deepspeedai/DeepSpeed.

[5] 2025. Distributed Checkpoint. https://pytorch.org/blog/distributed-checkpoint-efficient-checkpointing-in-large-scale-jobs.

[6] 2025. Hadoop 3.3.6. https://hadoop.apache.org/docs/r3.3.6/hadoop-project-dist/hadoop-hdfs.

[7] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. 2021. Gradient compression supercharged high-performance data parallel dnn training. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 359–375.

[8] Yu Chen, Zhenming Liu, Bin Ren, and Xin Jin. 2020. On efficient constructions of checkpoints. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*. Article 152, 10 pages.

[9] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[11] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

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

[13] Diederik P Kingma. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

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

[15] Kyushick Lee, Michael B Sullivan, Siva Kumar Sastry Hari, Timothy Tsai, Stephen W Keckler, and Mattan Erez. 2019. Gpu snapshot: checkpoint offloading for gpu-dense systems. In *Proceedings of the ACM International Conference on Supercomputing*. 171–183.

[16] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[17] Wenshuo Li, Xinghao Chen, Han Shu, Yehui Tang, and Yunhe Wang. 2024. ExCP: Extreme LLM Checkpoint Compression via Weight-Momentum Joint Shrinking. In *International Conference on Machine Learning*. PMLR, 27575–27588.

[18] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv preprint arXiv:1907.11692* (2019).

[19] Avinash Maurya, Bogdan Nicolae, M Mustafa Rafique, Amr M Elsayed, Thierry Tonellot, and Franck Cappello. 2022. Towards Efficient Cache Allocation for High-Frequency Checkpointing. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 262–271.

[20] Avinash Maurya, Robert Underwood, M Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. 2024. Datastates-llm: Lazy asynchronous checkpointing for large language models. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. 227–239.

[21] Avinash Maurya, Jie Ye, Mustafa Rafique, M, Franck Cappello, and Bogdan Nicolae. 2024. Breaking the memory wall: A study of i/o patterns and gpu memory utilization for hybrid cpu-gpu offloaded optimizers. In *Proceedings of the 14th Workshop on AI and Scientific Computing at Scale using Flexible Computing Infrastructures*. 9–16.

[22] Avinash Maurya, Jie Ye, M Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. 2024. Deep optimizer states: Towards scalable training of transformer models using interleaved offloading. In *Proceedings of the 25th International Middleware Conference*. 404–416.

[23] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843* (2016).

[24] Zhangqiang Ming, Yuchong Hu, Wenxiang Zhou, Xinjue Zheng, Chenxuan Yao, and Dan Feng. 2024. ADTopk: All-Dimension Top-k Compression for High-Performance Data-Parallel DNN Training. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. 135–147.

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

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

[27] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[28] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[29] Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know what you don't know: Unanswerable questions for SQuAD. *arXiv preprint arXiv:1806.03822* (2018).

[30] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[31] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[32] Foteini Strati, Michal Friedman, and Ana Klimovic. 2025. PCcheck: Persistent Concurrent Checkpointing for ML. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 811–827.

[33] Guanhua Wang, Olatunji Ruwase, Bing Xie, and Yuxiong He. 2024. FastPersist: Accelerating Model Checkpointing in Deep Learning. *arXiv preprint arXiv:2406.13768* (2024).

[34] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, TS Eugene Ng, and Yida Wang. 2023. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 364–381.

[35] Zhuang Wang, Haibin Lin, Yibo Zhu, and T. S. Eugene Ng. 2023. Hi-Speed DNN Training with Espresso: Unleashing the Full Potential of Gradient Compression with Near-Optimal Usage Strategies. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*. 867–882.

[36] Zhuang Wang, Xinyu Crystal Wu, Zhaozhuo Xu, and TS Eugene Ng. 2023. Cupcake: A Compression Scheduler for Scalable Communication-Efficient Distributed Training. In *Proceedings of the Sixth Conference on Machine Learning and Systems (MLSys' 23)*. Proceedings of the Sixth Conference on Machine Learning and Systems (MLSys' 23).

[37] Lingrui Xiang, Xiaofen Lu, Rui Zhang, and Zheng Hu. 2024. SSDC: A Scalable Sparse Differential Checkpoint for Large-scale Deep Recommendation Models. In *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–5.

[38] Chenxuan Yao, Feifan Liu, Yuchong Hu, Zhengyu Liu, Xinjue Zheng, and Wenxiang Zhou. 2025. LowDiff: Efficient Frequent Checkpointing via Low-Cost Differential for High-Performance Distributed Training Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1113–1126.

[39] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).

[40] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. 2023. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277* (2023).