DataStates-LLM: Lazy Asynchronous Checkpointing for Large Language Models

Avinash Maurya, Robert Underwood, M Mustafa Rafique, Franck Cappello, Bogdan Nicolae

DataStates-LLM: Lazy Asynchronous Checkpointing for Large Language Models

Avinash Maurya
Rochester Institute of Technology
Rochester, NY, USA
am6429@cs.rit.edu

Robert Underwood
Argonne National Laboratory
Lemont, IL, USA
runderwood@anl.gov

M. Mustafa Rafique
Rochester Institute of Technology
Rochester, NY, USA
mrafique@cs.rit.edu

Franck Cappello
Argonne National Laboratory
Lemont, IL, USA
cappello@anl.gov

Bogdan Nicolae
Argonne National Laboratory
Lemont, IL, USA
bnicolae@anl.gov

ABSTRACT

LLMs have seen rapid adoption in all domains. They need to be trained on high-end high-performance computing (HPC) infrastructures and ingest massive amounts of input data. Unsurprisingly, at such a large scale, unexpected events (e.g., failures of components, instability of the software, undesirable learning patterns, etc.), are frequent and typically impact the training in a negative fashion. Thus, LLMs need to be checkpointed frequently so that they can be rolled back to a stable state and subsequently fine-tuned. However, given the large sizes of LLMs, a straightforward checkpointing solution that directly writes the model parameters and optimizer state to persistent storage (e.g., a parallel file system), incurs significant I/O overheads. To address this challenge, in this paper we study how to reduce the I/O overheads for enabling fast and scalable checkpointing for LLMs that can be applied at high frequency (up to the granularity of individual iterations) without significant impact on the training process. Specifically, we introduce a lazy asynchronous multi-level approach that takes advantage of the fact that the tensors making up the model and optimizer state shards remain immutable for extended periods of time, which makes it possible to copy their content in the background with minimal interference during the training process. We evaluate our approach at scales of up to 180 GPUs using different model sizes, parallelism settings, and checkpointing frequencies. The results show up to 4× faster checkpointing and 2.2× faster end-to-end training runtime compared with the state-of-art checkpointing approaches.

CCS CONCEPTS

• **Software and its engineering** → *Checkpoint/restart*; • **Computing methodologies** → *Machine learning*; • **Computer systems organization** → **Reliability**; • **Information systems** → **Parallel and distributed DBMSs**.

KEYWORDS

LLMs and transformers; scalable checkpointing; asynchronous multi-level checkpointing

HPDC '24, June 3–7, 2024, Pisa, Italy
2024. ACM ISBN 979-8-4007-0413-0/24/06…$15.00
https://doi.org/10.1145/3625549.3658685

ACM Reference Format:
Avinash Maurya, Robert Underwood, M. Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. 2024. *DataStates-LLM*: Lazy Asynchronous Checkpointing for Large Language Models. In *The 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24), June 3–7, 2024, Pisa, Italy.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3625549.3658685

1 INTRODUCTION

Context. Large-language models (LLMs) have seen an increasing adoption in various domains ranging from academic and scientific research to industrial applications. They have been traditionally used for creative text generation, prompt completion, comprehension, and summarization, etc. Additionally, recent initiatives such as LLMs for science (e.g., DeepSpeed4Science [39]) are beginning to explore use cases that involve specialized domain-specific languages for tasks such as genome sequencing, protein structure prediction, equilibrium distribution prediction, etc. The versatility and democratization of LLMs have led to an unprecedented scale of development and discovery across multiple fields.

In a quest to improve the quality of large language models (LLMs), the size of the training data and the size of the LLMs are rapidly increasing. LLMs are routinely made of billions of parameters and there are predictions that they will reach the trillion scale in the near future, e.g., Google Switch-C (1.6T) [7], WuDao 2.0 (1.75T) [46], and M6-10T [18]. Under such circumstances, LLMs need to be trained in a scalable fashion on high-performance computing (HPC) machines comprising a large number of compute nodes and GPUs. Despite advances in technologies that enable LLM training to scale (hybrid data-, pipeline- and tensor parallelism, sharding of model parameters and optimizer state, layout and communication optimizations, etc.), training remains a resource and time-intensive task: LLMs often require weeks if not months to either be trained from scratch (also referred to as pre-training) or be fine-tuned for specialized tasks.

Motivation: Checkpointing as a Fundamental Primitive. During such a long runtime involving a large number of components, unexpected events are frequent and can have devastating consequences. For example, due to the tightly coupled nature of distributed training of LLMs, hardware failures, software bugs, or communication timeouts, can occur, which may lead to globally corrupted states even if they affect a small number of components.

Unicron [12], a recent effort from Alibaba, highlights a 43.4% failure rate of resource-intensive LLM training, out of which 37% were hardware failures, while the remainder 73% could be fixed by system restarts. In both cases, a checkpoint is needed to effectively resume the LLM training.

Even in the absence of failures, the training can take an undesirable trajectory that leads to dead-ends, e.g., slow or no convergence, undesirable learning patterns that need to be "unlearned", instability, etc. For example, loss spikes are one type of an undesirable trajectory. They were reported by PaLM [6] and GLM-130B [46] and were observed during the training of popular models such as BLOOM-175B and OPT-175B. Since they are hard to predict and defend against, the only viable strategy is to roll back to a past checkpoint and try an alternative strategy, such as skipping over problematic mini-batches or reorganizing the model, e.g., by switching some parameters to higher precision or different floating point representation.

Additionally, checkpointing of intermediate states during the training is a fundamental pattern used in several other scenarios: understanding the evolution of the learning patterns captured by the model, continuous testing of alternatives without disturbing production deployments, switching between divergent model states based on Reinforcement Learning from Human Feedback (RLHF).

Challenges and Limitations of State of the Art. Widely used deep-learning models (ResNet [11], VGG [38], etc.) of moderate sizes, i.e., hundreds of MBs, and their associated optimizer state typically fit in the memory of a single GPU. In this case, data parallelism is often enough to scale the training, which means that it is enough to checkpoint a single model replica by gathering the relevant state from a single GPU. On the other hand, LLMs are sharded across a large number of GPUs, which means that a checkpoint needs to gather distributed data structures. Such an operation involves much larger sizes, i.e., in the order of hundreds of GBs. Therefore, synchronous checkpointing solutions, e.g. default checkpointing implemented in DeepSpeed [34], that block the training until the model state is captured to stable storage incur high runtime overheads. Alternatively, one may use a multi-level asynchronous checkpointing solution that copies the model state to a fast memory tier and then flushes it from there to slower tiers in the background without blocking the training. In general, this is a widely used solution in the HPC community that successfully reduces the runtime overheads compared with synchronous checkpointing. However, it is not straightforward to implement this approach in the context of LLM training for two reasons. First, there is simply not enough free memory on the GPUs to hold a full copy of the checkpoint shards, due to which it is not possible to benefit from the high GPU memory bandwidth to reduce the overhead of creating a blocking copy. Second, while it is possible to create the copy directly on the host memory (e.g. TorchSnapshot [30], TorchLightning [17], CheckFreq [24]), this involves data transfers that are an order of magnitude slower and subject to competition due to shared PCIe links between multiple GPUs and the host memory. Ultimately, this results in significant overheads that reduce the benefit of asynchronous checkpointing to the point where it is not significantly faster as compared to synchronous checkpointing approach. To put this in perspective, despite the availability of high speed links (50+ GB/s network and 25+ GB/s PCIe), the LLM checkpointing throughput is far from saturating the link capacity (e.g., REFT [42] reports 38% saturation), and often drops as low as a few GB/s (e.g., Nebula [23], Microsoft's DeepSpeed closed-source implementation of asynchronous checkpointing reports 1-4 GB/s).

Key Insights and Contributions. In this paper, we propose *DataStates-LLM*, a novel asynchronous checkpointing technique that overcomes the limitations of the aforementioned state-of-the-art approaches. Our key idea is to leverage the observation that the model parameters and optimizer state remain immutable for extended periods of time during an iteration (i.e., during the forward pass and backward pass) and are updated in bulk at specific points. Specifically, we can copy the model state (parameters, optimizer state) during the forward and backward pass from the GPU to the host memory without blocking the training iteration. At the same time, we can hide the overhead of contention for the memory and storage tiers and guarantee the consistency of checkpoints asynchronously once the checkpointing data is available on the host memory. We summarize our contributions as follows:

(1) We perform a gap analysis that highlights the checkpoint sizes, load-balancing among the checkpoint shards corresponding to 3D parallelism, and when the LLM model parameters and optimizer state remain immutable during each training iteration. This analysis is essential in shaping our contribution (§ 4).
(2) We introduce a series of key design principles, i.e., hybrid flushing of GPU model/optimizer shards to host memory, lazy copy that overlaps with the intervals during which the LLM remains immutable, streamlined multi-level flushing to persistent storage, and asynchronous consolidation of model/optimizer shards (§ 5.1).
(3) We discuss an architecture that integrates these design principles into widely used LLM training runtimes, namely DeepSpeed and Megatron (§ 5.2).
(4) We design and implement the components of the architecture, insisting on details related to high-performance aspects, such as, efficient data movements and serialization of LLM shards, orchestration of background parallelism, bridging between high-level abstractions in Python and low-level C++ implementation, coordination and consistency (§ 5.3).
(5) We evaluate our implementation in a series of extensive experiments in which we train large LLMs (up to 70B parameters) on modern HPC systems (512 nodes, each consisting of four A100 40GB GPUs). We show significant speed-up of end-to-end runtime and up to 4× higher checkpointing throughput in a variety of configurations (§ 6).

Limitations of the Proposed Approach. By leveraging the fact that the LLM remains immutable during a significant part of each training iteration, we can perform lazy device-to-host copies of the tensors that make up the LLM model state, which reduces the time each iteration is blocked while waiting for device-to-host I/O related to checkpointing to finish. This accelerates the training iterations during which a checkpoint is taken, but at the cost of accumulating checkpointing data on the host memory faster,

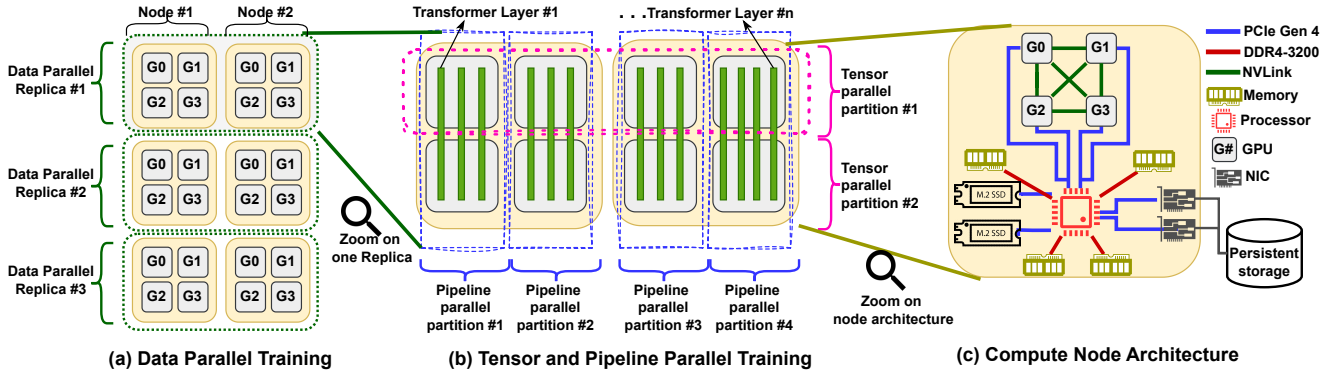


Figure 1: Data, pipeline, and tensor parallel runtime training. Compute node configuration consisting of four A100-40GB GPUs.

especially for high checkpoint frequencies. Thus, if the asynchronous flushes of the checkpointing data from the host memory to the lower-level storage tiers, e.g., node-local NVMe storage and parallel file systems (PFS), are not fast enough to keep up with the device-to-host lazy copies, this will eventually become a bottleneck. In this case, our approach needs to be complemented with other techniques, e.g., compression, for reducing the bottleneck caused by the flushes. Nonetheless, even under such circumstances, our approach will still exhibit less overhead than other state-of-the-art LLM checkpointing approaches, albeit the difference will be smaller.

2 BACKGROUND

2.1 Data Parallelism

Data parallelism is the most widely used technique to accelerate the training of deep learning models [16]. It creates replicas of the learning model on multiple workers, each of which is placed on a different device and/or compute node, as illustrated in Figure 1(a). The input data is randomly shuffled and partitioned among the workers at each epoch. During the forward pass, the workers simply process their mini-batches from the partition of their dataset in an embarrassingly parallel fashion. Then, during the backward pass, the model parameters are updated based on the average gradients of all replicas (instead of the local gradients), which effectively synchronizes all replicas to learn the same patterns from all partitions. Data parallelism leads to accelerated training because the partitioning of the input data results in fewer iterations per epoch.

2.2 Pipeline and Tensor Parallelism

Pipeline and tensor parallelism are two complementary techniques that enable the training of large learning models that do not fit in the memory of a single GPU. Pipeline parallelism leverages the idea that learning models can be split into stages, each of which can placed on a separate GPU. Then, the forward and backward pass corresponding to different mini-batches can be overlapped by activating all stages in parallel: as soon as the forward pass of one mini-batch has been moved to the next stage, another mini-batch can be processed in the current stage. This idea applies similarly to the backward pass but in reverse order: as soon as the backward pass of one mini-batch has been moved to the previous stage, another mini-batch can be processed in the current stage [14]. Tensor parallelism leverages the idea that even individual layers and tensors can be sharded and distributed horizontally across multiple GPUs. Figure 1(b) illustrates these ideas for an example decomposition of an LLM consisting of n layers into multiple pipeline parallel (highlighted by the vertical dotted blue box) and tensor parallel (denoted by the horizontal dotted magenta box) shards. Nvidia Megatron-LM is a prominent example of the LLM framework that is widely adopted in practice and offers configurable mechanisms to partition the model in pipeline and tensor parallel modes. The trade-off in this case is that the computations on the stages and shards are tightly coupled and distributed at fine granularity among the GPUs, which introduces the need for frequent communication that is subject to overheads. Amongst data, pipeline, and tensor parallel approaches model training, the tensor parallel approach is the most communication-intensive approach since it requires intra-layer interaction. Therefore, if tensor-parallelism cannot be completely avoided for a given model configuration, it is typically configured to use node-local GPU resources, thereby exploiting fast node-local fabrics such as NVLinks [5]. On a typical A100 GPU compute node, illustrated in Figure 1(c), the degree of tensor parallelism should not exceed the number of node-local GPUs in order to take advantage of fast 600 GB/s NVLinks to mitigate communication overheads. The combination of data parallelism, pipeline parallelism, and tensor parallelism is often called *3D parallelism*.

2.3 State Sharding to Eliminate Redundancy of Data-Parallel Replicas

Data parallelism introduces high redundancy in maintaining independent model replicas. This can be exploited to maintain a single replica across all workers, where each replica is responsible for the management of a distinct shard. Then, when a worker needs to access a full model, it needs to obtain all missing shards from the rest of the workers. Just like in the case of model parallelism, such an approach sacrifices performance, due to extra communication overheads, for improving memory efficiency. A prominent example that implements this idea is DeepSpeed [32] which is widely used for training LLMs in combination with Megatron [37]. DeepSpeed offers a set of incremental optimization stages: stage-1, stage-2,

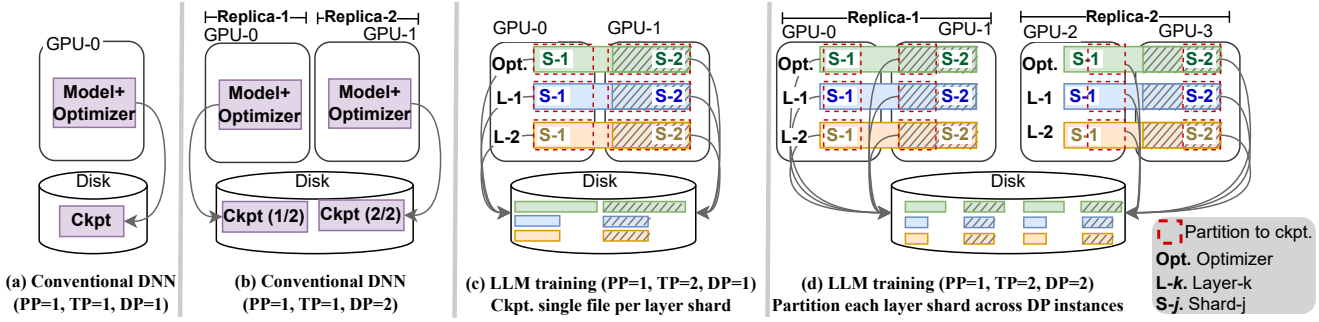


Figure 2: Sharding of checkpoints during training of conventional DNNs and LLMs for different degrees of pipeline (PP), tensor (TP), and data (DP) parallelism.

and stage-3, which correspond to sharding the optimizer state, gradients, and model parameters across all data parallel ranks, respectively. DeepSpeed also offers additional tunable optimizations such as out-of-core management of shards using the host memory for swapping.

2.4 Implications of State Sharding on Checkpointing

For conventional DNN models, the state captured in the checkpoint (typically model parameters and optimizer state) is usually serialized as a single file, as depicted in Figure 2(a). When using data parallelism, since there are many identical DNN model replicas available, it is possible to split the model into shards and parallelize the I/O by ensuring each worker captures and flushes a different shard as a separate file, as shown in Figure 2(b). This approach is adopted by DeepFreeze [25], TorchSnapshot [30], and LightCheck [4]. In the case of LLMs, sharding can be exploited even without data parallelism to enable parallel writes of different layers into different files, as shown in Figure 2(c). Finally, this can be complemented by another level of sharding when data parallelism is added, as shown in Figure 2(d). By default, the DeepSpeed runtime implements the latter case, which results in a large number of shards being stored in separate files. On many HPC systems, this provides the best I/O performance especially for parallel file systems. However, it also raises the problem of managing a large number of shards and potential metadata bottlenecks [9]. In this work, we assume that the default DeepSpeed strategy is to serialize the LLM checkpoint shards into separate files while leaving the question of how to find better file aggregation layouts as future work.

2.5 Problem Formulation

For the scope of this paper, we only focus on scenarios considering 3D parallelism combined with stage-1 (optimizer partitioning across data-parallel ranks), which corresponds to a configuration in which DeepSpeed and Megatron were successfully used to train the largest LLM models, such as BLOOM [44] (up to 175 billion parameters). Our goal is to design scalable multi-level asynchronous checkpointing solutions that: (1) capture a globally consistent checkpoint of LLMs that includes all shards of all GPUs corresponding to both the model parameters and the optimizer state (which is needed to successfully restart the training); (2) maximize the checkpointing throughput in order to reduce the amount of time during which the training is blocked by checkpointing; and (3) minimize the contention for resources and interference between the training and the overlapped background data transfer tasks for reducing the end-to-end training duration.

3 RELATED WORK

3.1 Checkpointing in Deep Learning

Checkpointing techniques have been extensively explored in the specific context of deep learning for minimizing the I/O overheads on training. Systems such as CheckFreq [24] aim at performing fine-grained iteration-level checkpoints and overlap checkpoint flushes with the training phases, but do not support checkpointing in pipeline parallel training setups and are inefficient in utilizing the available network and PCIe interconnect and memory subsystems, showing only up to 40% peak efficient checkpointing throughput across data-parallel replicas. Approaches such as DeepFreeze [25], TorchSnapshot [30], and LightCheck [4] attempt to mitigate the checkpointing overheads by both overlapping transfers with training and partitioning checkpoints across data-parallel replicas, but do not support hybrid pipeline, tensor, data-parallel training setups.

3.2 Checkpointing for LLMs

Several recent efforts specifically target checkpointing for LLMs and focus on efficient asynchronous 2-phase CPU-based snapshotting and lazy persistence. However, the reported checkpointing throughputs are far from saturating the network (50+ GB/s and PCIe (25+ GB/s) links. For example, Gemini [43] reports 3.13 GB/s checkpointing throughput (9.4 GB shard of GPT-100B takes about 3 seconds for checkpointing). REFT [42] reports 38% PCIe bandwidth utilization at 6 GB/s, while TRANSOM's checkpointing engine (TCE) [45] reports achieving a throughput of ~1.2 GB/s. Nebula [23], which is Microsoft's DeepSpeed closed-source implementation of asynchronous checkpointing and is only available on the Azure cloud, reports achieving 1-4 GB/s (GPT2-XL checkpoint of 20.6 GB takes 5 seconds to checkpoint). These results hint at significant gaps in existing checkpointing techniques for LLMs.

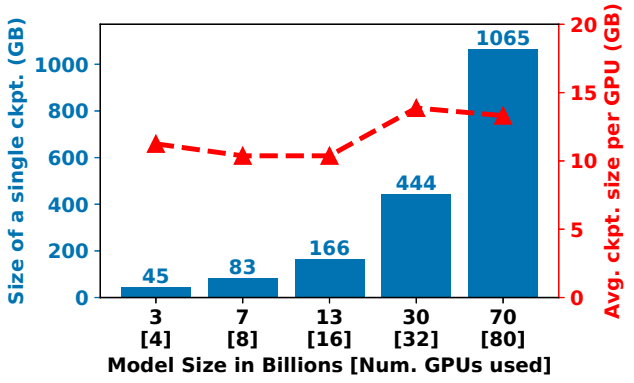


Figure 3: Aggregate checkpoint sizes of different model sizes and average checkpoint size per GPU.

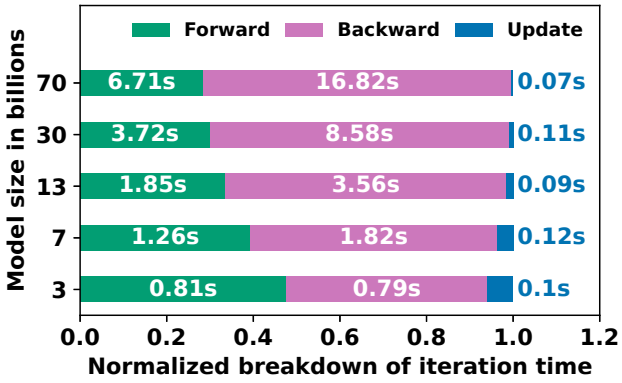


Figure 4: Different iteration phases. Model and optimizer states are immutable during forward and backward passes.

3.3 High-Performance Checkpointing Runtimes

HPC workloads have widely adopted checkpointing runtimes for resilience. User-transparent runtimes, e.g., BLCR [10] and DMTCP [1], capture the entire state of all processes distributed across multiple nodes, which is exclusively used for restarting from failures. GPU-based transparent checkpointing runtimes such as CheCUDA [40] and NVCR [27] provide similar functionality for capturing GPU-based working state of the application. While these approaches are transparent, they incur higher checkpointing overhead because the entire state of the application (including non-critical data structures) is captured and flushed to disk. Application-level checkpoint-restart runtimes such as VELOC [20–22, 26] and FTI [3, 28] require the application to mark critical data structures necessary to restart application from failures for both CPU-only and hybrid CPU-GPU applications. Canary [2] supports containerized checkpointing. However, none of these runtimes exploit the immutable phases of LLM training to optimize checkpointing by overlapping the checkpointing phase with the training phase.

3.4 I/O Optimizations in Data Movement and Checkpoint Runtimes

Data-movement and checkpoint engines in HPC such as ADIOS [8], VELOC [19, 26], and FTI [3] support efficient asynchronous data movement through multi-level cache hierarchy. VELOC [20], for instance, reserves a pinned cache on both the device (GPU) and host memory for buffering checkpoints in an overlapping fashion with the application execution. However, given the large device memory required for LLM training, the GPU does not have enough spare capacity to even hold a few tensors that need to be checkpointed; thereby compelling runtimes to use host memory as the fastest memory tier to cache/buffer checkpoints from. Furthermore, unlike conventional DNNs where the size of the input dataset is typically larger than the model states (and therefore checkpoints), in the case of LLMs, the model is usually larger than the micro-batches consisting of a few thousand integer-based tokens. Therefore, as highlighted in Gemini [43], the available pool of host memory is generally large enough to accommodate both the next subset of prefetched input micro-batches and LLM checkpoints.

4 ANALYSIS OF LLM CHECKPOINTING BEHAVIOR

4.1 LLM Checkpoint Sizes and Load Balancing

Unlike the case of lightweight optimizers such as stochastic-gradient descent (SGD) [35], which are widely used in conventional DNN models, LLMs adopt advanced adaptive learning rate optimizers such as Adam (Adaptive momentum estimation) [15]. Such optimizers need to store additional state information (momentum, variance, gradients), which leads to an explosion of the optimizer state size. Unfortunately, this state information cannot be simply left out of the checkpoint as it is essential for a successful restart of the training process. Coupled with the already large number of LLM parameters (billions), the overall checkpoint size becomes massive. Even worse, while the size of checkpoints grows proportionally to the number of transformer layers, it grows quadratically with respect to the number of hidden dimensions [33]. To study this effect, we ran a series of experiments (the setup is explained in detail in § 6.1) that use DeepSpeed to train the models listed in Table 1. The results are depicted in Figure 3. As expected, the checkpoint sizes quickly grow to large sizes and exhibit similar checkpoint size per GPU for different model sizes, hinting at the fact that DeepSpeed achieves good load-balancing among the shards as highlighted by the minor y-axis.

4.2 Immutability of Model Parameters and Optimizer States During Each Iteration

We also study the behavior of each training iteration at fine granularity by breaking down the runtime into forward pass, backward pass, and update duration. The results are shown in Figure 4. We observe that regardless of the model size, the forward and backward passes take up the majority of the training iteration duration. In addition to the increasing computational complexity involved in training larger models, the long iteration duration can be attributed to operations, such as, send/recv of activations and gradients (pipeline and tensor parallelism) and gradient all-reduce (data parallelism), are expensive and become a bottleneck. With increasing the LLM model size, they get amplified and lead to a negligible update phase. Fortunately, this situation presents an opportunity

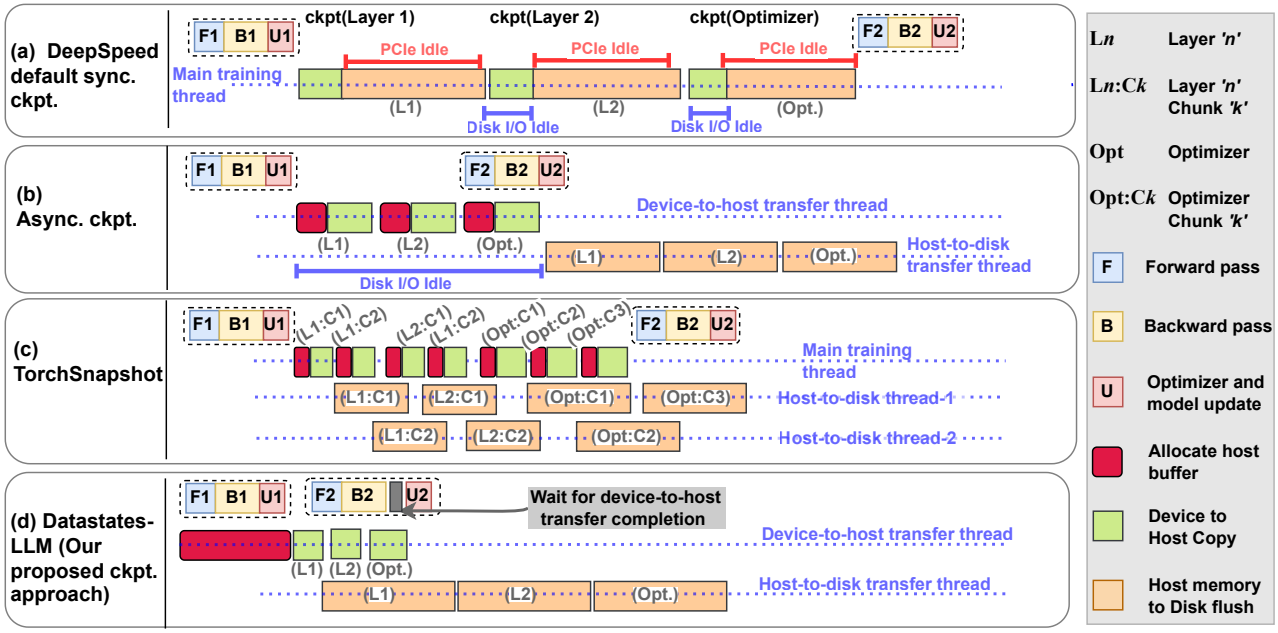


Figure 5: Overlapping LLM training with checkpointing using different approaches.

that can be leveraged to our advantage. First, both the model parameters and the optimizer state remain immutable during both the forward and backward passes. Thus, any copies from the GPU memory to the host memory can be issued asynchronously during the forward pass and the backward pass without causing coherency issues. Second, such copies utilize the PCIe link between the GPU and the host, which is different from the communication links (i.e., NVLink [5] and GPUDirect RDMA [29]) between GPUs and between the compute nodes that are used for communication during training. Thus, asynchronous copies do not compete for bandwidth with the forward and the backward passes and therefore they do not cause interference.

5 DATASTATES-LLM: SYSTEM DESIGN

5.1 Design Principles

Based on the observations outlined in § 4, we introduce a series of high-level design principles that we adopt in *DataStates-LLM* to mitigate the limitations of state-of-art LLM checkpointing runtimes.

Coalescing of GPU Model/Optimizer Shards to Host Memory: Conventional asynchronous multi-level checkpointing techniques (as implemented in the related works mentioned in § 3) move the checkpoints one-at-a-time through the storage levels: first they allocate host memory to hold the checkpoint, then they capture the checkpoint on the host memory by performing a GPU-to-host copy, then they asynchronously flush the checkpoint from the host memory to persistent storage. If another checkpoint request arrives before the previous checkpoint is finished flushing, it will be blocked waiting for the flushes to complete. For small learning models that fit in the memory of a single GPU, such an approach works reasonably well because all model parameters and the optimizer state can be captured at once in a single file. However, the combination of 3D parallelism and optimizer state sharding targeted by our checkpointing scenario results in many independent shards per GPU that correspond to both the model parameters and the optimizer state. Eventually, each of these shards needs to be flushed to persistent storage, typically as a separate file, as illustrated in Figure 2(c).

In this case, conventional asynchronous multi-level approaches would serialize the checkpointing of the shards. For example, if we consider three shards in a checkpoint, two of which correspond to layers $L1$ and $L2$ and the third corresponds to the optimizer state shard, then only the flushing of the optimizer state shard will overlap with the next iteration (forward pass, backward pass, and updates), while the rest of the operations (allocate, copy, flush $L1$; allocate, copy, flush $L2$; allocate, copy optimizer state) are serialized. This severely degrades the performance of asynchronous checkpointing to the point where it may become slower than synchronous checkpointing. To optimize and extend the conventional asynchronous multi-level checkpointing approach for multi-layered LLMs, the following approach, illustrated in Figure 5(b), can be used — all the three shards in the checkpoint ($L1$, $L2$, and optimizer) can be first *snapshotted* quickly using device-to-host copies, which will block the training for all layers except the snapshot of last layer, which can be overlapped with the next training iteration. Once the snapshot of all layers involved in the checkpoint is complete, they can be persisted through asynchronous flushes from host to disk. However, even such an advanced asynchronous approach slows down training due to slow host memory allocation and transfers (as evaluated in Figure 12c). To mitigate this issue, we propose three optimizations. First, we pre-allocate enough host memory to hold all shards on the host memory. This pre-allocated memory

will be reused for all checkpoint requests, effectively eliminating the allocation overheads for all shards, both belonging to the same and different checkpoints. Second, we pre-pin the allocated host memory, which accelerates GPU-to-host data transfers, again for all shards of both the same and different checkpoints. Third, we coalesce the copies of the shards to host memory, which eliminates the need to wait for the flushes of the shards belonging to the same checkpoint to finish before initiating more GPU-to-host copies.

Lazy Non-Blocking Copies Overlapping with Forward and Backward Pass: We leverage a key observation that the model and optimizer shards on each GPU remain immutable during the forward pass and the backward pass, and are updated later in bulk (typically through `optimizer.step()` for optimizers such as Adam). Therefore, unlike conventional asynchronous multi-level checkpointing techniques, there is no need to block the next training iteration until a full copy of the checkpoint is available on the host memory. Instead, we allow the next training iteration to start immediately after the checkpoint request, and proceed to copy the shards to the host memory while the forward pass and the backward pass are progressing in parallel. Only when the update phase is about to begin, if the shard copies on host memory are not finished, then we delay the update phase until they are finished. Furthermore, the flushes from the host memory to persistent storage are also allowed to overlap with the update phase. It is for this reason that we refer to our technique as "lazy" non-blocking copies: in effect, we reduce the duration of blocking the training by postponing the wait for as much as possible until there is a risk for consistency issues. An example is illustrated in Figure 5(d): the forward and backward pass of the second iteration $F2$ and $B2$ proceed immediately after the first iteration has finished, at which point a checkpoint request was issued. They overlap with the GPU-to-host copies. The update phase $U2$ is delayed until the GPU-to-host copies have finished, thereby blocking the application. Meanwhile, the previously captured checkpoints on the host are asynchronously flushed to persistent storage. Finally, if the host memory that is reserved for checkpointing is full, then the next checkpoint request needs to wait for previous tensors to get evicted from the host memory after they are flushed to the persistent storage, e.g., node-local NVMe storage or parallel file system. We enforce this wait in order to avoid running out of the host memory since GPU-to-host copies are faster than host copies to persistent storage.

Streamlined Multi-level Flushing to Persistent Storage: Although we coalesce the shards into a single pre-allocated memory region on the host memory, it is important to note that it is not necessary to wait until all shards are successfully copied to the host memory before starting the flushes to persistent storage. Instead, we can imagine a streaming pattern: as soon as partial checkpointing data is copied from the GPU to the host memory, we can immediately flush it to the persistent storage. Using this approach, two separate physical links (GPU-to-host and host-to-persistent storage) can be used in parallel to transfer the checkpointing data, which reduces the I/O overheads associated with checkpointing. Furthermore, it is important to note that GPUs have a separate GPU-to-host hardware copy engine. Therefore, the memory accesses on a GPU issued during the forward pass and the backward pass, regardless of whether to run computational kernels or to communicate with other

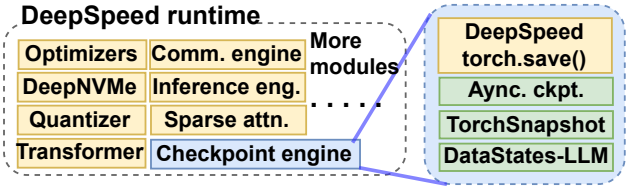


Figure 6: Three checkpointing engines added in DeepSpeed runtime (highlighted in green) for comparative evaluation.

remote GPUs (through NVLinks and/or GPUDirect RDMA [29]), do not compete with the copies of the shards. Likewise, flushing from host memory to persistent storage uses an entirely different I/O path that does not interfere with the GPUs. As a consequence, our approach maximizes the use of the I/O paths needed for checkpointing, while it maximizes the overlapping with the training iterations without slowing them down due to interference or contention for shared resources. Thanks to this approach, except for unavoidable waits not sufficiently postponed by lazy non-blocking copies, training iterations can effectively progress almost undisturbed by checkpointing.

Asynchronous Distributed Consolidation of Model and Optimizer Shards: While often overlooked, a significant source of overhead in the case of synchronous checkpointing is the consensus needed among the GPUs to validate all shards as being successfully saved to the persistent storage. Only then can a global checkpoint be declared to hold a valid model parameter and optimizer state that can be later reused to restart the training or study its evolution. Thanks to our asynchronous streamlined multi-level flushing, there is an opportunity to hide the consensus overhead: once each GPU finished flushing the shards to persistent storage, it can enter into a consensus protocol asynchronously, which can perfectly overlap with the training iterations. Furthermore, it is possible to reduce the number of participants in the consensus by introducing a hierarchic consolidation protocol that first validates the shards belonging to the same GPU, then the partition of shards belonging to the GPUs sharing the same compute node, and finally, all partitions belonging to all compute nodes. In this work, we have considered a simple two-phase commit protocol, but we note that our approach is generic and can accommodate more advanced consensus protocols that are tolerant to byzantine failures (e.g. Paxos, Raft [13]).

5.2 *DataStates-LLM* Architecture

We implement our multi-level asynchronous checkpointing approach as a modular extension to the DeepSpeed runtime in the form of a checkpointing engine, as an alternative to the default synchronous engine (based on `torch.save()`) and the asynchronous Nebula engine (which is closed-sources and exclusively available on Microsoft Azure cloud). This is illustrated in Figure 6. Our engine can be enabled in the configuration file which is supplied to the DeepSpeed engine at runtime and consists of a single attribute object specifying the size of the host buffer which can be reserved per process for caching checkpoints. Note that this extension does not utilize anything specific to DeepSpeed and can be easily adopted by other training runtimes as well.

We note that all checkpointing primitives and APIs of *DataStates-LLM* are the same as those used by DeepSpeed's default checkpointing engine, except for one additional method which blocks as long as any previous snapshot capture operations are pending. At the application level, checkpointing is transparent to the user, and no code modifications are needed to select any of the available checkpointing approaches, including the one that is proposed in *DataStates-LLM*. The integration of *DataStates-LLM* was performed through DeepSpeed's fork of Megatron-LM, which contains ZeRO-based optimizations for the Megatron framework and does not need any modifications to use our checkpointing approach.

5.3 Implementation

Our checkpointing engine[1], is written in C++/CUDA and is exposed to DeepSpeed through Python and C++ APIs. The pinned host buffer is managed through a simple lightweight circular buffer manager, considering the producer-consumer pattern described in the design principles (§ 5.1). Dedicated CUDA streams and threads are used for device-to-host and host-to-file transfers. Such offloading of transfers and flushes in C++ enables our approach to overcome the limitations of the state-of-the-art asynchronous approaches (e.g., CheckFreq [24], LightCheck [4], and Lightning's AsyncCheckpointIO [17]) which perform background checkpointing and flushes through Python threads. These Python thread-based implementations are prone to inefficiencies arising from Python Global Interpreter Lock (GIL), lack of stream-based copies through GPU-copy engines supporting DMA, and host buffer re-allocation overheads.

Given a Python object (composed of tensors on both GPU and host memory, arrays, objects, and other data structures) that needs to be checkpointed, our checkpointing engine decomposes this operation into three phases as follows: (1) recursively parse the Python object, and create a list of large arrays and tensors (across both GPU and host memory) by storing their memory pointers and sizes; (2) create a header by computing the file offsets for each tensor/object marked for asynchronous transfer in step (1); and (3) enqueue asynchronous device-to-host transfer (if required) and host-to-disk writes of headers, tensors and large objects (obtained in step-1).

6 PERFORMANCE EVALUATION

6.1 Experimental Setup

Platform: We conduct our experiments on ALCF's Polaris [2] HPC testbed. It consists of 560 nodes, each equipped with 512 GB of DDR4 memory (aggregated from four NUMA domains), a 32-core AMD Zen 3 (Milan) (64 threads), two 1.6 TB SSDs (2 GB/s) and four Nvidia A100 GPUs aggregating to a total of 160 GB HBM memory. On each node, the four A100 GPUs are connected with each other using four NVLinks and with the host memory through a PCIe Gen 4 interface. The peak unidirectional Device-to-Device (D2D), and pinned Device-to-Host (D2H) (and vice versa) bandwidths on each GPU are 85 GB/s and 25 GB/s, respectively. There is a one-to-one mapping between the GPU and the NUMA domains, therefore concurrent device-to-host access by multiple GPUs does not create contention on the PCIe interface. The checkpoints are flushed to persistent storage, which is a Lustre [36] parallel file system, composed of 160 Object Storage Targets (OSTs) and 40 Metadata targets, with an aggregated bandwidth of 650 GB/s.

Software: All the nodes run Nvidia CUDA driver 470.103, NVCC v11.8.89, Python v3.10, PyTorch v2.1, and DeepSpeed v0.11.2 on top of the Cray SUSE Linux Enterprise Server 15 operating system. In our experiments, we use up to 128 nodes (512 GPUs) to study the impact of large model sizes through data, tensor and pipeline parallelism, and contention of checkpoint flushes for the parallel file system.

6.2 Compared Approaches

DeepSpeed: This is the default checkpointing approach used in the DeepSpeed [34] LLM training runtime using PyTorch's default `torch.save()` approach. This approach blocks the LLM training and performs synchronous writes of the checkpoint to the persistence storage, thereby providing consistency guarantees for the checkpoint (illustrated as *(a) DeepSpeed default synchronous checkpointing* in Figure 5).

Asynchronous Checkpointing: This approach is representative of the in-memory snapshotting techniques adopted by CheckFreq [24], LightCheck[4], and PyTorch Lightning's AsyncCheckpointIO [17] (illustrated as *(b) Asynchronous checkpointing* in Figure 5), and is replicated to mimic AsyncCheckpointIO [31] (we had to adapt such techniques for LLMs since the original implementations do not support pipeline and tensor parallelism). Specifically, in the first phase, it allocates a buffer for each shard on the host memory (red block), then copies the shard from the device to the host buffer (green blocks). Once the first phase has finished, it proceeds to asynchronously flush the shards from the host memory to persistent storage (Lustre PFS in our case). This is depicted in Figure 5(b). The allocation overhead can be significant due to the need to pin the host memory [20], especially when considering a large number of shards. It highlights an important limitation of many state-of-the-art approaches that are not optimized for LLM checkpointing.

TorchSnapshot: This is a state-of-the-art checkpointing runtime developed by the PyTorch team (illustrated as *(c) TorchSnapshot* in Figure 5). It optimizes checkpointing by (1) parallelizing state capture across data-parallel replicas (which is moot for DeepSpeed/Megatron since the latter shards the checkpoints by default); (2) splitting tensors in chunks for overlapping transfers in streaming fashion from the device-to-host and host-to-disk; and (3) multi-threaded write of chunked tensors in different files on the disk, thereby utilizing higher disk write bandwidth, but incurring additional metadata and flushing overheads because of larger number of files [9]. We limit the number of parallel flush threads per GPU to 4, which shows peak write throughput to persistent storage in our experimental testbed.

***DataStates-LLM* (Our Approach):** This is the implementation of *DataStates-LLM* based on the design proposed in § 5 and illustrated as *(d)* DataStates-LLM in Figure 5.

[1]The source code of *DataStates-LLM* is available at https://github.com/DataStates/datastates-llm.
[2]https://www.alcf.anl.gov/polaris

Table 1: Configuration of models and runtime used for evaluations derived from BLOOM [44] (highlighted by gray column) and LLaMA [41].

Model size in billions	3	7	13	30	70
Layers	30	32	40	60	80
Hidden dim.	2560	4096	5120	6656	8192
Atten. heads	32	32	40	52	64
Num. of nodes	1	2	4	8	20
Tensor parallelism	4 (=Number of GPUs per node)				
Pipeline parallelism	=Number of nodes				
ZeRO optimization	Stage 1 (Partition optimizer state)				

6.3 Evaluation Methodology

Models, Sharding, and Dataset: We use five different LLM model sizes in our evaluations based on the real-world setups: BLOOM (3B) [44], LLaMA (30B), and LLaMA2 (7B, 13B, 70B) [41] model architectures. The models and their runtime configurations are summarized in Table 1.

To minimize the intra-layer communication overheads, the tensor-parallel degree is set to 4, which is the number of GPUs in a single node and all are interconnected through fast NVLinks. To fit the model across distributed GPU memories, the pipelines are split evenly across the number of nodes described in Table 1 using the default partitioning scheme of uniformly balancing the number of trainable parameters on each pipeline stage. Unless otherwise noted, the data-parallelism degree is set to 1, representing a single LLM replica being used for training. For the experiments that involve the data parallelism approach, the optimizer state is sharded across the replicas. This corresponds to the configuration Figure 2(d).

Throughout our experiments, we use a subset of the OSCAR-en dataset included in the repository of the BLOOM model. It consists of 79K records, [44], and use the default LLaMA2 [41] tokenizer for pre-processing the dataset into tokens. Similar to BLOOM training, the default sequence length is set to 2048, and the micro-batch size is 16 to avoid out-of-memory (OOM) errors in any configuration.

Memory and Storage Tiers: Each of the compared approaches is allowed to use up to a maximum of 64 GB of host memory, the rest of which is reserved for caching the training data. Since the average checkpoint size per GPU is 10-15 GB (shown in Figure 3) and there are four GPUs per node, this is enough to hold a full checkpoint across all compute nodes. From the host memory, the checkpoint shards are flushed directly to Lustre, which acts as the shared persistent storage.

Key Performance Metrics: Throughout our evaluations, we measure the following metrics for comparing the aforementioned approaches: (1) checkpointing throughput of different model sizes to evaluates the blocking checkpointing overhead on the application for a broad range of increasing complex LLMs; (2) impact on iteration duration during checkpointing to evaluate the slowdown and interference caused by checkpointing on training iterations; and (3) end-to-end training runtime to study the broader impact on overall job completion times. We evaluate the above metrics under different settings: (a) varying degrees of data parallelism since DeepSpeed runtime partitions the checkpoints across data-parallel ranks for faster checkpointing, this setting studies the impact of strong scaling (more flushing bandwidth available to capture the checkpoint of the same size), and (b) varying checkpointing frequency to study how the training performs for different degrees of I/O pressure arising from frequent or sparse checkpointing scenarios.

6.4 Performance Results

Increasing LLM Model Size Without Data Parallelism: In our first set of experiments, we evaluate the following two metrics for increasing model sizes: (1) the average checkpointing throughput perceived by the training process, which is defined as the total checkpoint size divided by the time for which the training was blocked for each checkpointing operation; and (2) the average iteration duration when checkpointing, which shows the overheads of checkpointing on the training process in both direct form — the amount of time for which training is blocked to capture checkpoint, and indirect form — slowdown in training process caused by interference from checkpointing I/O. The training is run for five iterations with a checkpoint being taken at every iteration. Such high-frequency checkpointing at every iteration allows us to study the performance overheads of different approaches under high I/O pressure. We note two interesting observations for evaluating this metric. First, Since the asynchronous checkpoint operations from device-to-host and host-to-file overlap with the computations of the next iterations, from an application perspective, this metric is important to study the checkpointing stalls experienced by the application by different checkpointing approaches. Second, the checkpoint operation is a blocking collective with respect to the model and optimizer update stage during training, i.e., none of the processes can start updating the model or optimizer states until all parts of the previous checkpoint are consistently captured either on the host memory or on the persistent file. Therefore, the checkpointing throughput observed by the application is dictated by the slowest process across all processes.

As observed in Figure 7, the checkpointing throughput increases with increasing model size. This is because of two reasons: (1) The training duration per iteration increases with larger models due to the higher complexity of transformer layers and higher communication overheads (for sharing activations, gradients, optimizer partitions, and model updates) across multiple nodes (as depicted in Figure 4). The increasing iteration duration allows for more time to asynchronously flush the previous checkpoints, thereby not blocking future checkpoint requests due to pending flushes. (2) Larger models are run on more number of nodes (as outlined in Table 1), leading to more device-to-host interconnects which can be exploited for parallel flushing of checkpoints between node-local memory tiers, and higher write bandwidth available for flushing checkpoints to the persistent file system. As a consequence of the above two factors, we observe a linear scalability trend of checkpointing throughput in Figure 7 for all approaches. However, compared to DeepSpeed, Asynchronous checkpointing, or TorchSnapshot, *DataStates-LLM* demonstrates at least 4× and up to 34× higher checkpointing throughput across various model sizes.

Next, we study the impact on the overall iteration duration. Figure 8 shows the breakdown of per-process iteration duration as

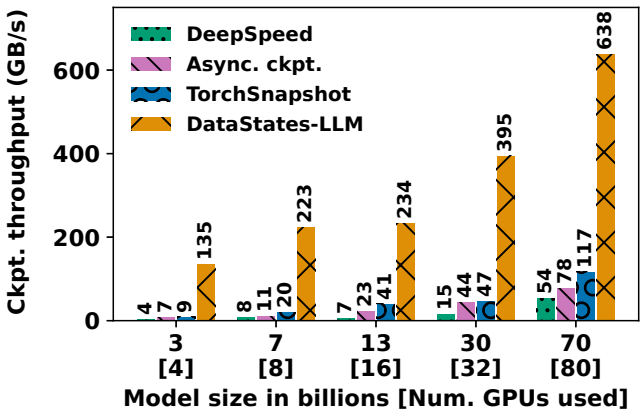


Figure 7: Aggregate checkpointing throughput for different model sizes. Higher is better.

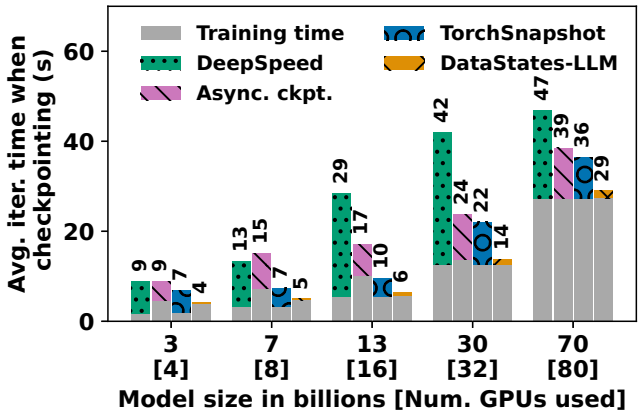


Figure 8: Average training iteration time for different model sizes when checkpointing. Lower is better.

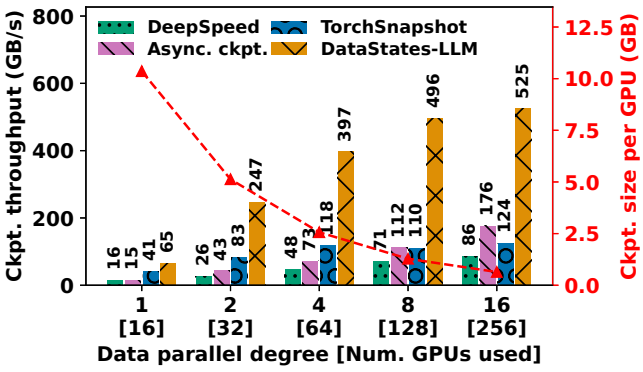


Figure 9: Aggregate checkpointing throughput for a 13B model for different data-parallel degrees. Higher is better.

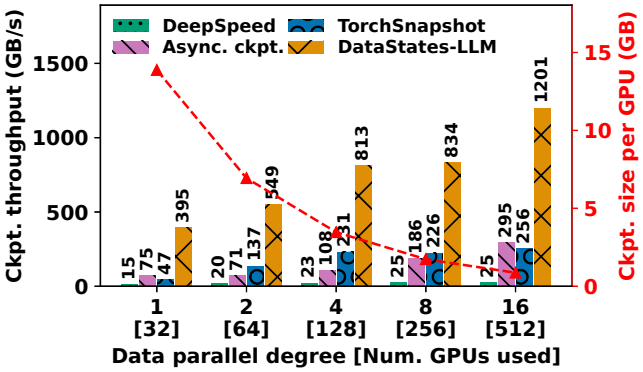
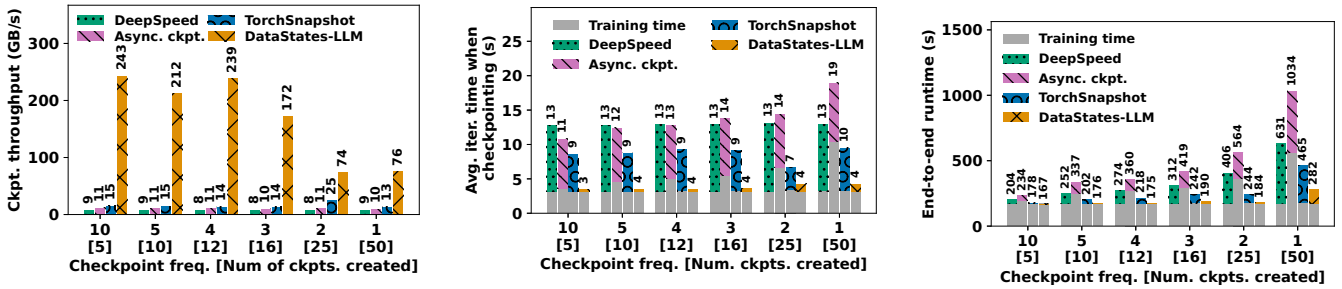


Figure 10: Aggregate checkpointing throughput for a 30B model for different data-parallel degrees. Higher is better.

training time vs. checkpointing time. We observe that the training time (consisting of forward pass, backward pass, and update phases) of smaller models (3B, 7B, 13B, and 30B) are similar for all approaches except for the Asynchronous checkpointing approach. This is because of the interference caused by slow host-memory allocation, slow transfers to unpinned host-memory, and PCIe contention with loading the next micro-batch on the GPU from the data pipeline. This effect is not observed in the larger 70B model because, for large models with the same amount of checkpoint data per GPU (shown in Figure 3), the long forward and backward passes amortize the slow allocation and transfer overheads. With increasing model size, the training time increases (Figure 4), while the checkpoint size per GPU remains consistent (Figure 3). Therefore, the ratio of the training duration to blocking duration while waiting for checkpoints to finish increases with the model size. However, irrespective of the fact that the training phase dictates the major proportion of the iteration time, *DataStates-LLM* speeds up the iteration by at least 23%, and up to 4.5× compared to other approaches we studied in evaluating *DataStates-LLM*.

Fixed LLM Model Size with Increasing Data Parallelism: In our next set of experiments, we evaluate the checkpointing throughput as a function of increasing degrees of data parallelism. Similar to the previous set of experiments, we conducted this experiment by checkpointing during each of five consecutive iterations. This evaluation is important to study the efficiency of concurrent flushing of the partitioned optimizer state across the data parallel replicas. We evaluate the checkpointing throughput by scaling the data parallelism degrees from 1 to 16 for two model sizes: 13B and 30B. We do not consider the smaller 3B and 7B models because at high degrees of data parallelism, such models are partitioned at excessive levels, which results in tiny shards that lead to the underutilization of GPUs. On the other hand, large models such as 70B show similar trends as the 30B model, but run for much longer. We only scale up to a data-parallel degree of 16 with 512 GPUs because it is not trivial to train a large number of data-parallel replicas in practice due to the high costs of GPU resources — for instance, BLOOM 175B was trained with 8 data-parallel replicas on a total of 384 GPUs.

Figure 9 and Figure 10 show the checkpointing throughput with increasing scale of data parallelism for the 13B and 30B models. We observe that the checkpoint size per GPU, referenced by dashed-red

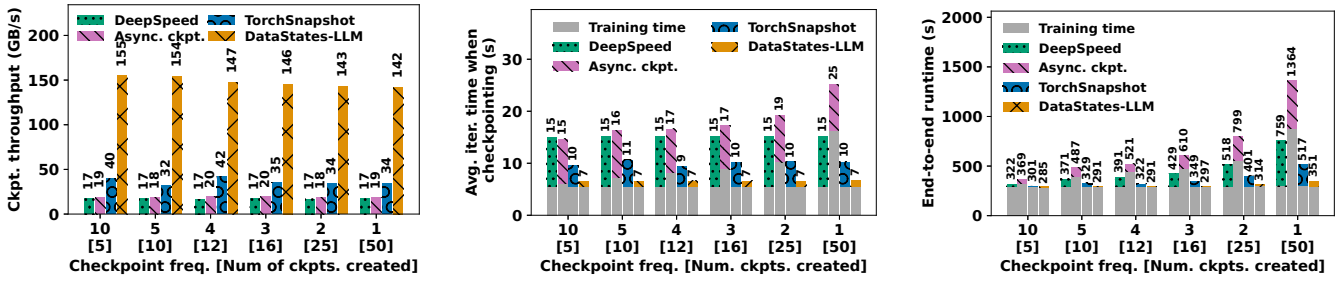


(a) Aggregate checkpointing throughput. Higher is better.

(b) Per process iteration time when checkpointing. Lower is better.

(c) End-to-end training time. Lower is better.

Figure 11: Running training for 50 iterations for a 7B model with different checkpointing frequencies.



(a) Aggregate checkpointing throughput. Higher is better.

(b) Per process iteration time when checkpointing. Lower is better.

(c) End-to-end training time. Lower is better.

Figure 12: Running training for 50 iterations for a 13B model with different checkpointing frequencies.

lines on the minor y-axis, shows a linear decrease of checkpoint size per GPU with increasing degrees of data parallel replicas. Therefore, this study captures the strong scalability of checkpoint performance, i.e., how well can various checkpointing approaches perform when the same checkpoint is distributed across multiple ranks, such that they can be flushed in parallel. More specifically, the checkpoint size per GPU drops from ~10.4 GB to ~650 MB per GPU for the 13B model, and from ~13.8 GB to ~870 MB per GPU for the 30B model, when scaling the data parallel degree from 1 to 16. When comparing the 13B and 30B models for the same number of GPUs (e.g., the 13B model with DP=4 and 30B model with DP=2 for 64 GPUs), we see that the checkpointing throughput of the 13B model is lower than the 30B model even though both approaches have the same number of parallel channels for flushing the checkpoint. This is because the training iteration of the 13B model is significantly faster than the 30B model and therefore needs to stall training for checkpointing more frequently as compared to the long-running iteration of the 30B model. While all approaches scale well to the increasing data parallel replicas due to concurrent flushes, our approach outperforms the DeepSpeed synchronous, Asynchronous checkpointing approach, and TorchSnapshot by 2.8×, 1.75×, and 1.78×, respectively for the 13B model; and for the 30B model by 48×, 4.12×, and 4.7×, respectively. In terms of end-to-end training runtime of the 30B model, we observe that *DataStates-LLM* shows up to 2.5× to 1.86× faster training completion time when scaling from DP=1 to DP=16 as compared to other approaches. Similar trends are observed for the 13B model. Therefore, our approach excels at strong scalability experiments of checkpointing and demonstrates significant speedup in end-to-end training runtimes.

Increasing Checkpointing Frequency: Next, we study the impact of scaling the checkpoint frequency, i.e., the number of iterations elapsed between consecutive checkpoint operations. This allows us to understand the efficiency of overlapping between the training and asynchronous checkpoint flushes such that the large intervals between subsequent checkpoint operations would allow for more time to complete the flushes to persistent storage and free up the host-memory buffer for the next checkpoints.

In particular, we evaluate the checkpointing throughput, iteration slowdown caused due to checkpointing, and the end-to-end runtime for a variable number of checkpoints captured during a 50-iteration run of the 7B and 13B models. Thanks to fast forward and backward passes, the 7B model presents less opportunities to overlap asynchronous I/O with the training iterations. Therefore, we chose it to highlight the difference between the approaches when the I/O pressure dominates. Conversely, the 13B model captures the opposite trend observed in larger model, where slower forward and backward passes enable more opportunities for overlap.

For the 7B model, we observe in Figure 11a that the checkpointing throughput of *DataStates-LLM* decreases with an increasing checkpointing frequency due to higher I/O pressure, which arises

due to the bottleneck of slow checkpoint flushes to the disk. On the other hand, the 13B model, depicted in Figure 12a, does not exhibit this effect. Instead, the checkpointing throughput remains high regardless of the checkpointing frequency. In any case, the other approaches suffer from I/O bottlenecks regardless of model size. As a consequence, *DataStates-LLM* achieves at least 3× higher checkpointing throughput for the 7B model and 4.2× higher checkpointing throughput for the 13B model.

Furthermore, we observe in Figure 11b and Figure 12b, respectively, that with increasing checkpointing frequency, the Asynchronous checkpointing approach slows down the training phase significantly, due to slow host memory allocation and transfers, similar to the effect illustrated in Figure 8. On the other hand, the other compared approaches do not increase the duration of the training iteration. However, thasnks to better overlapping with the forward and backward pass, *DataStates-LLM* achieves at least 1.3× and up to 3.8× faster iteration duration during checkpointing as compared with the other approaches.

Lastly, we study the end-to-end time taken to complete the entire training process, including the pending flushes towards the end of training. Figure 11c and Figure 12c depict the end-to-end runtime of the 7B model and the 13B model, respectively. The end-to-end training runtime shows performance trends similar to those observed in iteration-scale analysis (Figure 11b and Figure 12b). Specifically, our approach remains up to 3.86× faster in end-to-end training as compared to the other approaches even for an increasing checkpointing frequency.

7 CONCLUSIONS

In this work, we address the problem of high overheads incurred due to checkpointing in large-scale distributed LLM training running with advanced hybrid parallelism strategies using widely adopted runtimes such as DeepSpeed. State-of-the-art checkpoint engines, specifically designed for LLMs slow down the training while checkpointing because (1) they do not exploit the characteristics of various training phases to overlap checkpoint I/O efficiently; and (2) they underutilize the available interconnects and memory resources, leading to significant stalls during training. The checkpointing overheads are exacerbated when model and/or optimizer states need to be frequently checkpointed for defensive and productive use cases. To address these limitations, we design and develop *DataStates-LLM*, which efficiently and transparently overlaps the checkpoint I/O with the *immutable* phases of forward and backward passes during training. *DataStates-LLM* proposes key design ideas to mitigate checkpoint overheads in LLMs, such as preallocating and reusing pinned host buffer for fast DMA transfers, coalescing of model/optimizer shards while transferring checkpoints from GPU to host-memory, lazy non-blocking checkpoint snapshotting overlapping with forward and backward training phases, streaming multi-level flushing to persistent storage, and asynchronous distributed consensus of checkpoint persistence. We ran extensive evaluations with varying model sizes derived from production-grade runs of BLOOM and LLaMA2, different data parallelism configurations, and checkpointing frequency intervals. Results show that *DataStates-LLM* checkpoints 3× to 4.2× faster than existing state-of-the-art checkpointing runtimes, which achieves a speedup of the end-to-end training by 1.3× to 2.2×.

Encouraged by these promising results, in future we plan to explore data reduction techniques such as differential checkpointing and compression to further minimize the network and storage costs when checkpointing at high frequencies. Furthermore, we will explore efficient checkpointing strategies when model and/or optimizer states are offloaded across multiple memory tiers. Finally, we did not study the metadata overheads resulting from storing each shard as a separate file. This may lead to interesting trade-offs that justify investigating novel aggregation and consolidation strategies.

ACKNOWLEDGEMENTS

This work is supported in part by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research under contract DEAC02-06CH11357/0F-60169 and the National Science Foundation (NSF) under award no. 2106634/2106635. Results presented in this paper are obtained using Argonne's ALCF HPC systems, and NSF Cloudlab and Chameleon testbeds.

REFERENCES

[1] Jason Ansel, Kapil Arya, and Gene Cooperman. 2009. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *IPDPS'09: International Symposium on Parallel & Distributed Processing*. IEEE, Rome, Italy, 1–12.

[2] Moiz Arif, Kevin Assogba, and M. Mustafa Rafique. 2022. Canary: Fault-Tolerant FaaS for Stateful Time-Sensitive Applications. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Dallas, TX, USA, 1–16.

[3] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. 2011. FTI: High performance Fault Tolerance Interface for hybrid systems. In *SC'11: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Seattle, WA, USA, 1–12.

[4] Menglei Chen, Yu Hua, Rong Bai, and Jianming Huang. 2023. A Cost-Efficient Failure-Tolerant Scheme for Distributed DNN Training. In *ICCD'23: Proceedings of the International Conference on Computer Design*. IEEE, Milan, Italy, 150–157.

[5] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. NVIDIA A100 Tensor Core GPU: Performance and Innovation. *IEEE Micro* 41, 2 (2021), 29–35.

[6] 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. *JMLR'23: Journal of Machine Learning Research* 24, 240 (2023), 1–113.

[7] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: scaling to trillion parameter models with simple and efficient sparsity. *JMLR'22: Journal of Machine Learning Research* 23, 1, Article 120 (jan 2022), 39 pages.

[8] William F Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, et al. 2020. Adios 2: The adaptable input output system. a framework for high-performance data management. *SoftwareX* 12 (2020), 100561.

[9] Mikaila Gossman, Bogdan Nicolae, and Jon Calhoun. 2023. Modeling Multi-Threaded Aggregated I/O for Asynchronous Checkpointing on HPC Systems. In *ISPDC'23: Proceedings of the International Conference on Parallel and Distributed Computing*. IEEE, Bucharest, Romania, 101–105. https://hal.inria.fr/hal-04343661

[10] Paul H Hargrove and Jason C Duell. 2006. Berkeley lab checkpoint/restart (blcr) for linux clusters. *IOP Publishing* 46, 1 (2006), 494.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Las Vegas, USA, 770–778.

[12] Tao He, Xue Li, Zhibin Wang, Kun Qian, Jingbo Xu, Wenyuan Yu, and Jingren Zhou. 2023. Unicron: Economizing Self-Healing LLM Training at Scale. arXiv:2401.00134 [cs.DC]

[13] Heidi Howard and Richard Mortier. 2020. Paxos vs Raft: have we reached consensus on distributed consensus?. In *PaPoC'20: The 7th Workshop on Principles and Practice of Consistency for Distributed Data*. ACM, Heraklion, Greece, Article 8, 9 pages.

[14] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng

Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *NeurIPS'19: Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc., Vancouver, Canada.

[15] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG]

[16] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.* 13, 12 (2020), 3005–3018.

[17] PyTorch Lightning. 2023. Welcome to PyTorch Lightning — PyTorch Lightning 2.1.0 Documentation. https://lightning.ai/docs/pytorch/stable/.

[18] Junyang Lin, An Yang, Jinze Bai, Chang Zhou, Le Jiang, Xianyan Jia, Ang Wang, Jie Zhang, Yong Li, Wei Lin, Jingren Zhou, and Hongxia Yang. 2022. M6-10T: A Sharing-Delinking Paradigm for Efficient Multi-Trillion Parameter Pretraining. https://openreview.net/forum?id=TXqemS7XEH

[19] Avinash Maurya, Bogdan Nicolae, Mustafa Rafique, Thierry Tonellot, and Franck Cappello. 2021. Towards Efficient I/O Scheduling for Collaborative Multi-Level Checkpointing. In *MASCOTS'21: The 29th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, Virtual, Portugal, 1–8. https://hal.inria.fr/hal-03344362

[20] 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 *HiPC'22: The 29th IEEE International Conference on High Performance Computing, Data, and Analytics*. IEEE, Bangalore, India, 262–271.

[21] Avinash Maurya, Mustafa Rafique, Thierry Tonellot, Hussain AlSalem, Franck Cappello, and Bogdan Nicolae. 2023. GPU-Enabled Asynchronous Multi-level Checkpoint Caching and Prefetching. In *HPDC'23: The 32nd International Symposium on High-Performance Parallel and Distributed Computing*. ACM, Orlando, USA, 73–85. https://hal.inria.fr/hal-04119928

[22] Avinash Maurya, M. Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. 2023. Towards Efficient I/O Pipelines using Accumulated Compression. In *HIPC'23: 30th IEEE International Conference on High Performance Computing, Data, and Analytics*. IEEE, Goa, India, 256–265.

[23] Microsoft. 2023. Optimize Checkpoint Performance for Large Models - Azure Machine Learning. https://learn.microsoft.com/en-us/azure/machine-learning/reference-checkpoint-performance-for-large-models.

[24] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *FAST'21: The 19th USENIX Conference on File and Storage Technologies*. USENIX Association, Boston, USA, 203–216.

[25] 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 *CCGrid'20: The 20th International Symposium on Cluster, Cloud and Internet Computing*. IEEE/ACM, Melbourne, Australia, 172–181.

[26] Bogdan Nicolae, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Franck Cappello. 2019. VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale. In *IPDPS'19: IEEE International Parallel and Distributed Processing Symposium*. IEEE, Rio de Janeiro, Brazil, 911–920.

[27] Akira Nukada, Hiroyuki Takizawa, and Satoshi Matsuoka. 2011. NVCR: A transparent checkpoint-restart library for NVIDIA CUDA. In *IPDPS'11: Proceedings of the International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, Anchorage, AK, USA, 104–113.

[28] Konstantinos Parasyris, Kai Keller, Leonardo Bautista-Gomez, and Osman Unsal. 2020. Checkpoint restart support for heterogeneous hpc applications. In *CCGRID'20: The International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE/ACM, Melbourne, Australia, 242–251.

[29] Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K Panda. 2013. Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs. In *ICPP'13: The International Conference on Parallel Processing*. IEEE, Lyon, France, 80–89.

[30] PyTorch. 2024. Welcome to the TorchSnapshot documentation. https://pytorch.org/torchsnapshot/stable/.

[31] PyTorch-Lightning. 2024. AsyncCheckpointIO– PyTorch Lightning. https://lightning.ai/docs/pytorch/stable/api/lightning.pytorch.plugins.io.AsyncCheckpointIO.html.

[32] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. arXiv:1910.02054 [cs, stat]

[33] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-infinity: breaking the GPU memory wall for extreme scale deep learning. In *SC'21: The International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, St. Louis, Missouri, Article 59, 14 pages.

[34] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *KDD'20: The 26th SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, Virtual Event CA USA, 3505–3506.

[35] Sebastian Ruder. 2017. An overview of gradient descent optimization algorithms. arXiv:1609.04747 [cs.LG]

[36] Philip Schwan et al. 2003. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, Vol. 2003. Linux symposium, Ontario, Canada, 380–386.

[37] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs]

[38] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556 [cs.CV]

[39] Shuaiwen Leon Song, Bonnie Kruft, Minjia Zhang, Conglong Li, Shiyang Chen, et al. 2023. DeepSpeed4Science Initiative: Enabling Large-Scale Scientific Discovery through Sophisticated AI System Technologies. arXiv:2310.04610 [cs]

[40] Hiroyuki Takizawa, Katsuto Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. 2009. CheCUDA: A Checkpoint/Restart Tool for CUDA Applications. In *PDCAT'09: The International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE, Higashi-Hiroshima, Japan, 408–413.

[41] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs]

[42] Yuxin Wang, Shaohuai Shi, Xin He, Zhenheng Tang, Xinglin Pan, Yang Zheng, Xiaoyu Wu, Amelie Chi Zhou, Bingsheng He, and Xiaowen Chu. 2023. Reliable and Efficient In-Memory Fault Tolerance of Large Language Model Pretraining. arXiv:2310.12670 [cs.DC]

[43] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. 2023. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *SOSP'23: The Proceedings of the 29th Symposium on Operating Systems Principles*. ACM, Koblenz, Germany, 364–381.

[44] BigScience Workshop, Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, et al. 2023. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model. arXiv:2211.05100 [cs]

[45] Baodong Wu, Lei Xia, Qingping Li, Kangyu Li, Xu Chen, Yongqiang Guo, Tieyao Xiang, Yuheng Chen, and Shigang Li. 2023. TRANSOM: An Efficient Fault-Tolerant System for Training LLMs. arXiv:2310.10046 [cs.DC]

[46] Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, Weng Lam Tam, Zixuan Ma, Yufei Xue, Jidong Zhai, Wenguang Chen, Peng Zhang, Yuxiao Dong, and Jie Tang. 2023. GLM-130B: An Open Bilingual Pre-trained Model. arXiv:2210.02414 [cs.CL]