IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 27 June 2024, accepted 17 July 2024, date of publication 20 August 2024, date of current version 2 September 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3446770

RESEARCH ARTICLE

Optimizing Multi-Level Checkpointing for Distributed Deep Learning Workloads on Cloud Spot VM Clusters

YONGHYEON CHO[1,2], YOOCHAN KIM[2], KIHYUN KIM[2], JINWOO KIM[2], HONG-YEON KIM[3], AND YOUNGJAE KIM[2], (Member, IEEE)

[1]webOS SW Development Group, LG Electronics, Seoul 07336, South Korea
[2]Department of Computer Science and Engineering, Sogang University, Seoul 04107, South Korea
[3]Electronics and Telecommunications Research Institute (ETRI), Daejeon 34129, South Korea

Corresponding author: Youngjae Kim (youkim@sogang.ac.kr)

This work was supported in part by the Institute of Information Communications Technology Planning Evaluation (IITP) grants funded by Korean Government, Ministry of Science and ICT (MSIT), under Grant 2022-0-00498; and in part by the National Research Foundation of Korea (NRF) Grant funded by Korean Government, MSIT, under Grant RS-2024-00416666.

ABSTRACT Spot Virtual Machines (Spot VMs) offer access to underutilized computing resources at significant discounts, sometimes up to 90% off regular on-demand pricing. For budget-conscious organizations, using clusters of Spot VMs is an effective strategy for training large-scale distributed deep learning (DDL) models. However, the risk of preemption by cloud providers poses a challenge, as it can result in the loss of unsaved data in memory and local storage. To mitigate this risk, one solution involves using networked storage systems for checkpoints, though their low write throughput can slow down training. An alternative approach is to use the memory of a remote, on-demand computing node for temporary checkpoint storage, balancing data protection with training efficiency. In this paper, we propose a novel approach, ACUTE, to optimize temporary checkpointing in the memory of on-demand nodes during DDL training. ACUTE includes three key optimizations: 1) *Check-Mem*, which reduces memory copying overhead on the training node; 2) *Check-Trans*, which accelerates checkpoint data transfer through parallel processing; and 3) *Check-Pack*, which eliminates unnecessary data unpacking and repacking. Implemented using PyTorch's distributed data-parallel library, ACUTE was evaluated against two other checkpointing schemes on AWS VM instances. Results show that ACUTE reduces makespan delay to nearly zero and achieves, on average, 43.30% faster checkpointing compared to a baseline multi-level checkpointing scheme, without compromising the precision of Deep Neural Network (DNN) models.

INDEX TERMS Distributed deep learning, cloud computing, fault-tolerant systems, checkpoint and restart.

I. INTRODUCTION

Distributed Deep Learning (DDL) is an effective method for training extensive Deep Neural Networks (DNNs) using large datasets. This approach necessitates the deployment of multiple computational resources, often GPUs, or a network of interconnected machines. Through the application of parallel processing methods such as data and model parallelisms [1], [2], [3], [4], [5], [6], DDL efficiently tackles the substantial computational requirements of DNN training, considerably shortening the time needed for training relative to the use of a singular computational resource.

Constructing a GPU-based cluster for comprehensive training presents significant challenges and expenses, making such endeavors less feasible for smaller organizations or governmental agencies interested in conducting large-scale DDL projects. In response to these constraints, public cloud services emerge as a practical alternative, with leading providers including Amazon AWS, Microsoft Azure, and

The associate editor coordinating the review of this manuscript and approving it for publication was Mehrdad Saif.

Google Cloud Platform offering the necessary infrastructure to support extensive DDL operations.

Leveraging public cloud services allows users to execute DDL operations at reduced costs, eliminating the necessity for on-site GPU clusters [7]. Cloud Service Providers (CSPs) proffer a variety of options designed to meet diverse fiscal requirements. While on-demand Virtual Machine (VM) instances are available at standard rates, Spot VM instances offer a more cost-effective solution, with discounts up to 90% off, notably with Amazon Web Services (AWS). This pricing model attracts considerable interest from budget-conscious entities, providing access to VMs of equivalent specifications at significantly reduced costs. Spot VMs represent surplus idle computing capacities that CSPs leverage to enhance their profit margins by optimizing resource utilization.

Nonetheless, it is crucial to recognize the inherent limitations of Spot VM instances, especially in terms of their reliability and availability. The allocation of Spot VMs is highly dependent on fluctuating market prices and demand [8], which may lead to their abrupt termination to repurpose resources for other clients. Consequently, this allocation introduces a risk of unforeseen disruptions or terminations of Spot VM instances by the CSP. Therefore, deploying DDL tasks within this unstable Spot VM framework frequently results in job failures, underscoring the importance of weighing the benefits of cost savings against the potential for service interruptions.

The Check and Restart mechanism plays a pivotal role in mitigating disruptions during DDL training [2], [9]. It focuses on the safeguarding of DNN model parameters and vital states of the training process by storing them in persistent storage. This strategy is essential for preserving the training progress, ensuring that in the face of unforeseen interruptions or failures, the process can resume precisely where it left off, using the saved checkpoint data. This approach eliminates the need to start over from the beginning of the training cycle, offering a significant advantage in maintaining the continuity of model training.

The frameworks PyTorch and TensorFlow, which are extensively employed in DNN development, natively support the checkpointing process. Despite this, the fundamental implementation of checkpointing in these platforms can introduce significant Input/Output (I/O) overhead, potentially detracting from performance. For instance, consider a scenario where a PyTorch user engages the `torch.save()` function to execute checkpointing at the end of each epoch in a DNN model's training. Although this method effectively secures the training progress against data loss, it also increases the overall duration required to complete the training–referred to as the makespan, primarily because of the delays introduced by the I/O activities involved in saving checkpoint data, thereby causing interruptions in the training flow. More importantly, within the framework of Spot VM clusters, both node-local memory and storage are vulnerable to loss in the event of a Spot VM interruption.

Users can use network storage for checkpointing. However, we observed that, as outlined in Section III-B, the network storage's performance within the AWS Cloud environment is slower than expected. This reduced speed fails to meet the efficiency requirements for checkpointing, a crucial element in the learning process. Consequently, this leads to noticeable delays in saving data to remote storage, resulting in extended checkpointing durations that could negatively affect the entire training process.

As an alternative, the memory space of a reliable on-demand VM can be leveraged supporting faster write speed and reliability. We call this alternative a remote memory-based, multi-level checkpointing scheme. In this scheme, a neighboring on-demand VM serves as an intermediate repository with persistent storage connected via a network. This approach allows the stable on-demand VM's memory space to be used as a staging area for checkpointing, offering the chance to utilize faster media for writing data. However, merely implementing this multi-level checkpointing approach reveals further optimization opportunities.

In this paper, we propose ACUTE: Advanced Checkpointing in Unreliable DDL Training Environments. ACUTE is a remote memory-based, multi-level checkpointing scheme enhanced by three key optimization techniques (Check-Mem, Check-Trans, and Check-Pack) as follows:

- **Check-Mem:** Checkpointing in the middle of the DDL process involves dumping data from GPU memory to host memory and then from host memory to the persistent storage area. ACUTE seeks to minimize the delays caused by memory copy during the training process by overlapping the foreground deep learning processes with I/O operations.
- **Check-Trans:** Transferring checkpoint data to the memory of a neighboring VM may require a high amount of data communication over the network. To optimize this checkpoint data communication, ACUTE applies checkpoint data sharding and parallel transfer to minimize communication overheads. This parallelized communication accelerates the process of transferring checkpoint data to the memory space of neighboring on-demand VM.
- **Check-Pack:** ACUTE packs the checkpoint data only once which avoids unnecessary additional unpacking and repacking process. In a naive manner when using multi-level checkpointing, every data communication accompanies packing data right before sending. At the same time, it also leads to unpacking of the data right after receiving. However, on an intermediate level, unpacking of received data and repacking of that data are never necessary, since it just serves as temporary staging area. Therefore ACUTE removes packing and unpacking overheads that can occur during the data transfer via network communications. This optimization decreases the stall time of user's DDL process by performing faster writing.

We developed ACUTE in PyTorch (v2.0.1+cu117) using our own MPI package, specifically tuned from OpenMPI version 5.0.0, to facilitate the transfer of checkpoint data

to neighboring VMs. To best of our knowledge, we are the first to propose the remote-memory based, multi-level checkpointing strategy in a Spot VM cluster environment. We conducted experiments on AWS using up to 16 Cloud Spot VMs and 1 additional on-demand VM. The evaluation results of 10 DDL training workloads demonstrate that ACUTE showed near zero overhead due to the checkpointing operation. With respect to the time to guarantee the persistent save, ACUTE acheives 43.30% improvement in speed on average compared to the baseline muliti-level checkpointing scheme. Furthermore, there was no difference in accuracy of DNN model between training using ACUTE and training without checkpointing at all.

II. BACKGROUND

A. DISTRIBUTED DEEP LEARNING

Deep Learning (DL) is a powerful machine learning approach that trains a Deep Neural Network (DNN) by optimizing its learnable parameters [10]. The DL training process involves multiple **iterations**, typically comprised of forward pass, backward pass, and parameter update phase.

- **Forward Pass:** Input data is fed into the DNN through the input layer, and it propagates through the network. At the output layer, the network produces a computed output based on the most recently updated parameters.
- **Backward Pass:** The computed output is compared with ground truth labels, and a loss value is calculated. This value is then propagated backward through the network, computing gradients of the parameters with respect to the loss value. It determines how each parameter should be adjusted to minimize the universal loss value.
- **Parameter Update:** The computed gradients from the backward pass are used to update the parameters of the DNN. The parameters are adjusted in a direction that reduces the loss value, typically using optimization algorithms such as Stochastic Gradient Descent (SGD) or its variants and so on.

This iterative cycle, so-called **iteration**, is repeated multiple times during the training to improve performance of the DNN. Training the entire dataset at once, also known as batch training, offers advantages in terms of convergence speed and generalization of the DNN. However, it is often constrained by memory capacity and may result in overfitting issues. To overcome these limitations, it is common practice to divide the dataset into smaller units called *mini-batches*. The training process is then performed on these mini-batches. After all the mini-batches complete an iterative cycle, that constitutes an **epoch**.

In traditional DNN training, a single device, such as a GPU, is typically used. However, when dealing with large-scale datasets or larger DNN models, computational resources of a single device may not be sufficient. To address this problem, **Distributed Deep Learning (DDL)** techniques are often employed, utilizing multiple GPUs or machines to distribute the training workload. This approach enables parallel processing, reducing training time and allowing for larger-scale training.

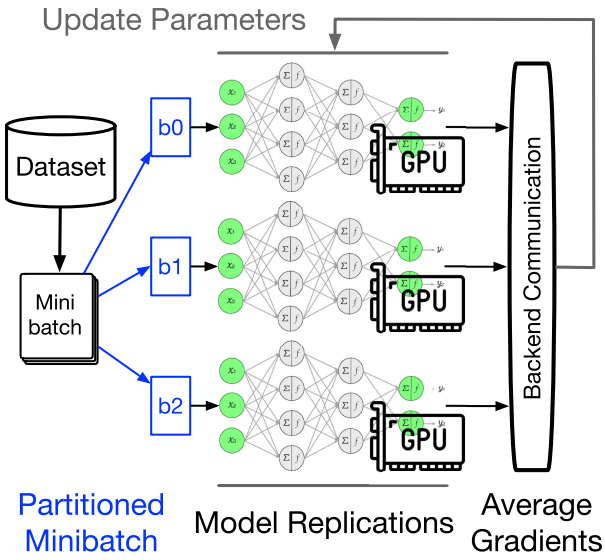


FIGURE 1. Data parallelism, one of the DDL methodologies.

When the size of a mini-batch is extremely large, it may lead to out-of-memory errors on a GPU device, making it impossible to process an iteration for a single mini-batch. In such cases, a mini-batch can be split and allocated across multiple devices to train the DNN model. This technique of partitioning a mini-batch into multiple GPUs and training them in parallel is known as *data parallelism*. Some DNN models such as emerging Large Language Model (LLM) have such extraordinarily large sizes that they cannot fit on a single GPU. In this case, a DNN model can be partitioned across multiple GPU devices for distributed training, which is known as *model parallelism*.

1) DATA PARALLELISM

In synchronous data parallelism, each GPU holds a replica of the DNN model with the same content, but processes different mini-batches [11]. In this work, we refer to synchronous data parallelism as data parallelism. After distributing the training of one mini-batch across multiple GPUs and performing the forward pass, the gradients for each loss value need to be synchronized during the backward pass. This process is known as gradient averaging, and it usually involves communications using a backend framework such as Message Passing Interface (MPI) in PyTorch's Distributed Data-Parallel (DDP) and Horovod [3], [4]. As the dataset size increases, using data parallelism provides an advantage in scaling up the training data.

B. CLOUD COMPUTING AND SPOT VM INSTANCE

Cloud Service Providers (CSP) offer computing resources to users in the form of various Virtual Machine (VM) instance types. To maximize profits by utilizing idle resources

more effectively, CSPs provide unclaimed or low-demand resources as **Spot VM** instances. Users can purchase those in a Spot market through a bidding process. Prices dynamically fluctuates by supply and demand. The prominent CSP include AWS, MS Azure, and GCP. To start Spot instances on AWS as an example, users can either directly create Spot instance requests or automatically create those through Amazon EC2.

General **on-demand VM**s are run in a stable manner with higher fixed prices. However, depending upon policies of the CSP, Spot VM instances can be preempted at any time when there is high demand or supply pressure for those resources. For example, in the case of Google Cloud Platform (GCP), preemption of Spot VMs is reported to occur at least once every 24 hours [7]. While there is an advantage of accessing these resources at a lower cost, users need to be aware that Spot VM instances are temporarily allocated, and the CSP can reclaim the resources unexpectedly. Therefore, when running repetitive tasks, it is crucial to prepare for an unexpected preemption. In this paper, we call this hazard the *preemption hazard*. We further explain this hazard and the strategy to avoid the data loss in the following section III.

C. MULTI-LEVEL CHECKPOINTING SCHEME

The **checkpoint/restart strategy** is one of the techniques used to provide fault tolerance in preparation for sudden system failure when performing a simulation program with a long run time typically in High Performance Computing (HPC). This strategy may be accompanied by a delay in the original task makespan, because I/O traffic that writes checkpoint data to storage occurs frequently.

To address this problem, a multi-level checkpointing scheme that writes checkpoint data with multiple staging hierarchies is often used [12], [13]. By utilizing this scheme, makespan delay can be decreased by first writing to media with a faster write speed. The system can flush checkpoint data later to persistent media with a slower write speed. This technique can be used not only for simulation workloads in HPC but also for DL workloads [1], [14].

This study also uses a multi-level checkpointing scheme to provide fault tolerance against *preemption hazards* that may occur in DDL workloads. However, popular open source DDL frameworks such as PyTorch DDP or Horovod provide only single-level checkpointing in an API. Additionally, multi-level checkpointing can be implemented in various ways depending on what media is used as a staging heirarchy. For example, when configuring a cluster to perform distributed tasks, **the memory space of neighboring nodes can be used as the primary first staging area.** This study directly implements and uses it as a baseline by using the open source MPI framework.

III. MOTIVATION

This work is motivated by three factors: (i) the need for a checkpointing method to provide fault tolerance to DDL jobs against the *preemption hazard* when using cloud Spot VMs to form a cluster, (ii) particularly low write performance of network storage in cloud environments, and (iii) several optimization opportunities when using multi-level checkpointing scheme on top of DDL workloads.

A. UNSTABLENESS OF CLOUD SPOT VM CLUSTER

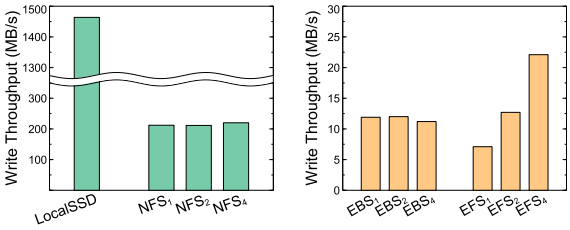
A Spot VM is a type of cloud virtual server that offers users a cost-effective option due to its low pricing or even free usage. However, Spot VMs are subject to preemption based on fluctuations in cloud service prices, which can pose certain challenges. Preemption refers to the termination of a Spot VM instance by the CSP when the demand for resources exceeds the available supply. When a Spot VM instance is preempted, any data stored in the local memory and local storage is lost. This situation is analogous to a system failure that happens on a typical server. The most generally used method to avoid this hazard is to apply a **checkpoint/restart strategy**.

AWS can terminate Spot instances when there is an increase in demand or a decrease in supply [15]. If a user sets a configuration when using a portion of instance types, AWS sends a termination notice to the instances 2 minutes before the actual termination [16]. Therefore, users can establish protective measures if they want to prepare for potential terminations. For example, they can write a script to save the DNN parameters and DDL training context, which is handled as **checkpoint data**, at regular intervals during the termination notice period [17].

However, a grace period of 2 minutes is insufficient to save DDL checkpoint data effectively. If the saving process is guaranteed to complete within the grace period, it is obvious that there is no problem to simply trigger the saving script right after the notice from AWS. However, if the training process is in the middle of a parameter update phase, DNN parameters are just going to be overwritten. That makes it meaningless to interrupt that process (if possible) and save DNN parameters which are supposed to be updated to other values. Even if a checkpointing script can wait for the training process to finish the parameter update phase, 2 minutes would be gone before it could initiate a checkpointing process. Therefore, 2 minutes can hardly provide any appropriate opportunity to save checkpoint data.

B. LOW WRITE PERFORMANCE OF CLOUD STORAGE

Checkpointing is a commonly employed technique in distributed computing to facilitate failure recovery and ensure software availability. However, as previously mentioned, storing data associated with the states of a DNN model on local storage of a Spot VM can be risky with possible data loss occurring. On the other hand, cloud-based VM instances have the advantage of being able to utilize network storage connected to the cloud network. This advantage enables users to perform checkpointing to network storage, which offers improved persistence compared to Spot VM local storage. Nonetheless, a significant challenge arises from the relatively low write performance of cloud storage services.



(a) Testbed Storage (b) Cloud Storage

FIGURE 2. Write throughput measurements of in-house testbed and AWS storage. We used 16 writer threads.

1) AWS CLOUD STORAGE PERFORMANCE

To empirically support the claim that cloud performance is a magnitude lower compared to on-premises local testbed, we performed write performance evaluation using the FIO [18] benchmark across various storage configurations in our local testbed and AWS cloud. Specifically, we used `g4dn.xlarge` GPU VM instance of AWS. The experimental setup involved executing a total of 16 writing threads with each thread performing 4KB block writing. The execution time for each experiment was set to 60 seconds.

For our local in-house testbed, we used two storage setups: a local SSD device with an ext4 file system mounted, and an SSD mounted with NFS utilizing the NFSv4 protocol on a 1G network. The hardware specifications are listed in Table 1. In the case of the NFS configuration, we measured the aggregate write throughput by simultaneously writing to the storage from a single node, two nodes, and four nodes. This approach allowed us to gauge the scalability and performance of the NFS-mounted SSD across multiple nodes, providing a comprehensive understanding of its capabilities. Similarly, for evaluating the write performance of AWS storage, specifically Elastic Block Store (EBS) and Elastic File System (EFS), we employed a comparable methodology as we did earlier on the NFS in-house testbed.

Figure 2 shows the results. Notably, the AWS cloud storage exhibited an average 90% lower write performance compared to the storage in local testbed for all cases. In particular, when utilizing a single thread, the default EBS storage provided by AWS on the `g4dn.xlarge` instance showed a 98% slower write performance compared to the local SSD. These findings demonstrate the need to utilize **another media space with a fast write speed**. As a solution, we suggest **memory space of a remote node** in this paper.

C. TOWARDS EFFICIENT MULTI-LEVEL CHECKPOINTING

The previous discussion highlighted the surprisingly low write performance of AWS cloud storage, posing an I/O bottleneck for DDL applications relying on synchronous checkpointing.

To address these kinds of challenges, existing DL checkpointing methods have proposed two techniques [1], [14]. The first technique involves asynchronous I/O, where the performance degradation resulting from I/O is mitigated by parallel processing of threads responsible for storing in-memory data of model parameters needed for recovery. This technique aims to hide the I/O-related performance impact. The second technique aims to reduce I/O time by leveraging the high bandwidth and ultra-low-latency local SSD. However, both of these methods primarily rely on resources of the node's local SSD for checkpointing.

To address these limitations, we adopt **remote memory-based multi-level checkpointing scheme**. In our approach, checkpointing is performed with remote memory by utilizing a stable on-demand VM as one level of a heterogeneous store hierarchy, thereby eliminating dependency on Spot VM local resources. Our work especially focuses on **how to further optimize this multi-level checkpointing scheme**. To optimize checkpointing performance and minimize the overhead due to the checkpointing, we address the following questions.

- Can we resume the training process before the checkpointing process ends completely?
- How can we utilize multi-node parallelism in performing checkpointing of data-parallel DDL?
- Is there any unnecessary packing and unpacking procedures while the checkpoint data passes through heterogeneous multi-levels of store hierarchy?

The optimization details about our proposed questions come in the subsection IV-D.

TABLE 1. In-house testbed storage node specifications.

Processor	AMD Ryzen 9 3900XT 12-Core Processor
Main Memory	DDR4, 16 GB x 4 (= 64 GB)
OS Kernel	Linux version 5.15.0-72-generic #79-Ubuntu SMP
Network File System	ext4, mount point: /mnt/nfs - 464.7 GB
Storage Device	Samsung SSD 970 EVO 500GB

IV. DESIGN

In this section, we first explain the target architecture of the Spot VM cluster that performs DDL training. Each *level* is defined to further elaborate the process of the checkpoint data transfer. Next, we explain the remote memory-based multi-level checkpointing scheme that we consider as a baseline. Last, we describe design principles and optimization of the checkpointing in detail.

A. TARGET ARCHITECTURE

Figure 3 shows the target architecture for the proposed approach in this study. In this architecture, a Spot VM cluster offers multiple computing resources to the DDL workload. During the training, the cluster periodically sends the in-memory checkpoint data of the DNN model to the memory space of a *peer VM* utilizing network transfer. The *peer VM* is an on-demand VM which ensures stability, so it will not be suddenly preempted like a Spot VM. And then, it asynchronously flushes the checkpoint data from its own memory space to a cloud storage service to ensure persistence.

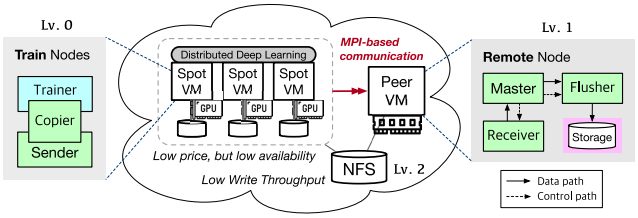


FIGURE 3. Target architecture of the multi-level cloud DDL system supporting fault-tolerance.

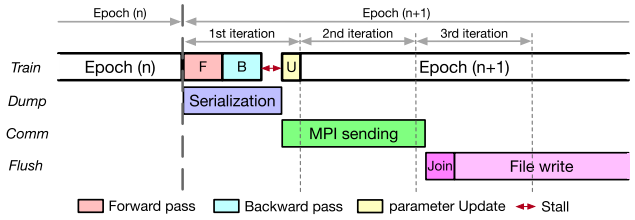


FIGURE 4. Illustration of checkpointing process in ACUTE. Serialization means the process of dumping the checkpoint data from GPU memory to Host memory on Lv.0.

We define *levels* in this target architecture.

- Level 0 **(Lv.0)** is a Spot VM cluster's memory space where a DDL application runs. *Train nodes* compose the cluster.
- Level 1 **(Lv.1)** is a peer VM's memory space that holds the checkpoint data of the DDL application temporarily but also reliably. Since this remote node is created as an on-demand VM, it is safe from preemption. A *remote node* is the peer VM.
- Level 2 **(Lv.2)** is a cloud storage service which stores the checkpoint data persistently. *NFS-based* file system is mounted on the storage. The service can be set by both local volume of Lv.1 node or external storage service.

B. BASELINE MULTI-LEVEL CHECKPOINTING SCHEME

The train nodes are composed of three modules: Trainer, Copier, and Sender, as depicted in Figure 3. These modules are responsible for transferring the checkpoint data from *Lv.0* to *Lv.1*. Trainer is an abstract of DDL training process. Copier dumps GPU-memory data to the host memory. Sender takes charge of sending checkpoint data to the remote node.

Meanwhile, the remote node consists of three modules: Receiver, Master, and Flusher. These modules are used to store the checkpoint data passed from *Lv.0* to *Lv.2*. Master orchestrates whole sequence of receiving checkpoint data and flushing it. Receiver communicates with Sender by a 1:1 mapping to receive data. Flusher takes charge of collating and flushing the checkpoint data as a file to *Lv.2* storage. *Lv.2* represents the persistent storage, where the checkpoint files are stored at last.

We have identified three potential bottlenecks in the multi-level checkpointing scheme above: (i) the process during which the Copier dumps the data, (ii) the process during which the Sender communicates with the Receiver over network, and (iii) the process during which the Flusher flushes checkpoint data in a buffer to the storage. In the following subsections, we will explain how to address these bottlenecks.

C. DESIGN PRINCIPLES

Now, we introduce the design approach of ACUTE. ACUTE's key design principles are as follows:

- **Quick persistence guarantee:** ACUTE aims to minimize the risk of losing trained contents of a DNN model due to instance preemption in an unreliable Spot VM environment. To achieve this minimization, checkpoint data should be saved in the persistent storage area as quickly as possible.
- **Minimum delay in foreground DDL training:** ACUTE does not want to interleave too much stall time to protect users from DNN model loss. It aims to minimize the stall by resuming the training process as soon as possible.
- **No change in training semantic:** ACUTE optimizes the execution of checkpointing operations through multi-threading. Our aim is to ensure that there are not any changes in the training semantics of the DDL workload that could negatively impact accuracy. ACUTE prevents the parameters of the DNN model from becoming stale by avoiding any overlaps between parameter update phase of the foreground training process and dumping in-memory checkpoint data of the background checkpointing process.
- **Correct and portable save file:** ACUTE aims to achieve accurate and seamless recovery of saved DNN models when resuming a DDL application after preemption. Checkpoint data is to be loaded on a resumed DDL application by using existing popular APIs. Furthermore, even in DDL applications that do not utilize ACUTE, any checkpoint data stored through ACUTE can be loaded in a compatible manner, i.e. preserving portability.

D. OPTIMIZATION OF MULTI-LEVEL CHECKPOINTING SCHEME

Figure 4 illustrates the checkpointing process which occurs between the end of an epoch and the start of the next epoch in a foreground DDL training process. Let's consider the scenario where the n^{th} checkpointing is in progress.

With ACUTE, the checkpoint data can be **asynchronously dumped from GPU memory to host memory** right after the n^{th} epoch ends. This overlapping allows for the forward pass and backward pass of the $(n+1)^{th}$ epoch's first iteration to proceed simultaneously.

Next, the data dumped into the host memory is transmitted to the remote node through MPI communication. During this process, ACUTE performs **data sharding and parallel transfer optimization**. It splits the checkpoint data on each train node of a cluster and sends one piece to a remote node in parallel. In this paper, this piece is called a checkpoint *shard*.

With a normal MPI protocol, once transmitted data arrives, it is unpacked on the memory space of the receiver side.

However, with ACUTE, it deliberately skips unpacking. Thus, it also avoids the accompanying re-packing procedure right before the flush to the *Lv.2* storage. By **avoiding packing and unpacking the checkpoint data on the remote node**, unnecessary overhead can be eliminated.

Finally, the checkpoint data is flushed to the *Lv.2* storage. **The process of flushing is completely separated from the foreground training process.** It periodically flushes the packed checkpoint data to the storage ensuring that it does not impact the training process and makespan at all.

We provide detailed explanations of each optimization technique below.

- **Check-Mem: memory copy overhead reduction**
  Train nodes in *Lv.0* dumps GPU-memory data to the host memory. It also transmits checkpoint data to *Lv.1* using MPI. If these processes are performed naively, a stall occurs in the foreground training process. To minimize the additional makespan increment with ACUTE, the training thread uses multithreading to trigger the GPU-memory dumping and the transfer of checkpoint data without stopping the training. This overlap can avoid a possibly lengthy checkpointing stall in a DDL training process.
- **Check-Trans: acceleration of checkpoint data communication through parallel transfer**
  ACUTE optimizes the transfer of checkpoint data from *Lv.0* to *Lv.1* by parallelizing it. Checkpoint data transfer time is reduced by sending small checkpoint shards in parallel rather than making one train node take full responsibility for the transfer of the entire checkpoint data. In the case that multiple GPUs are in a train node, ACUTE only transfers the checkpoint data from one GPU. The remote node of *Lv.1* launches the same number of communication threads as the number of shards. So each thread is in charge of receiving one shard. Fragmented checkpoint shards are to be merged through a join operation right before the flush thread performs flushing. Through this operation, it becomes one entire file. It allows checkpoint data to be created in a portable format file that can be used in an existing framework API, PyTorch `load()`.
- **Check-Pack: removal of unnecessary unpacking and repacking of checkpoint data**
  ACUTE uses the MPI when checkpointing from *Lv.0* to *Lv.1*. Data communication through universal MPI includes a process of packing during transmission and unpacking during reception. However, in ACUTE, the purpose of checkpointing is to save the in-memory data to persistent storage in the form of a file. Therefore, the purpose of checkpointing can be achieved more quickly by flushing already packed bytes in *Lv.0* to *Lv.2* as a file itself without going through the unpacking process. Therefore, the process of unpacking the received checkpoint data in *Lv.1* is unnecessary. Thanks to this optimization, the flush thread can run faster without unnecessary unpacking and packing prior to writing a checkpoint file.

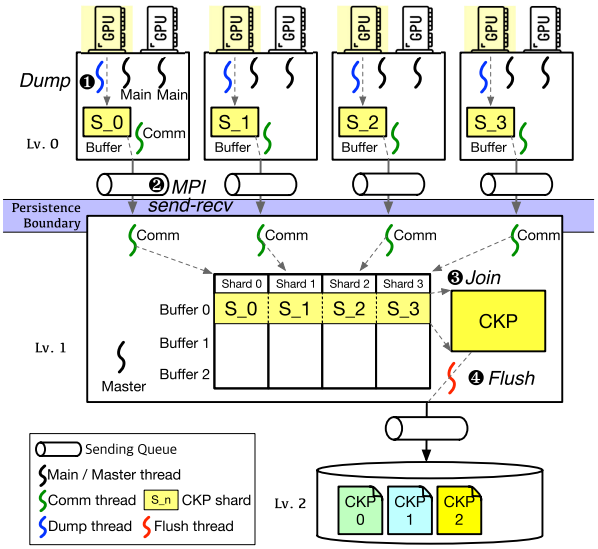


FIGURE 5. Operation overview which is performed by each module in ACUTE.

Finally, the checkpoint data stored in the memory of the remote node of *Lv.1* is flushed to the persistent storage *Lv.2* to complete the checkpoint saving. Please note that the flusher in the remote node of *Lv.1* is decoupled from the foreground training process and flushes the checkpoint data regardless of the flow of the process in *Lv.0*. The checkpoint data shards sent from *Lv.0* train nodes are stored in a buffer in the memory space of *Lv.1* remote node. Since this process is completely decoupled from foreground training performed in *Lv.0*, it does not affect the makespan of the DDL workload.

V. IMPLEMENTATION

In this section, first, we explain how each module of ACUTE can be implemented with Figure 5, based on the target architecture described earlier. Next, we track the flow on which checkpoint data travels from train nodes to persistent storage with Figure 6.

A. MODULES

When the Lv.0 train nodes need to checkpoint after an epoch ends, they dump the checkpoint data from the GPU device memory to host memory. At this time, the dump thread is executed separately and overlaps without interfering with main thread's training. However, if the main thread tries to update the parameters of the DNN model while the dump thread works, data consistency problems may occur. If this happens, the contents stored at the checkpoint data do not correctly include the trained contents.

To avoid this hazard, while the dump thread dumps checkpoint data immediately after the n^{th} epoch ends, the main thread keeps training up to a certain point. The training can be overlapped until the backward pass of the first iteration of the next epoch ends. If the GPU dump is not done by this time, a stall will occur.

The checkpoint data buffer at Lv.1 is managed as a circular buffer by the master thread. At the same time, the master thread controls the operation of the MPI communication (comm) threads and the flush thread. There are multiple receiving comm threads in one remote VM, and each thread receives only the checkpoint shard from the Lv.0 sending comm thread which is mapped to it. One Lv.0 sending comm thread can only communicate with one Lv.1 receiving comm thread. Each comm thread pair can transmit data in parallel with other pairs.

The master thread calls the comm thread to receive only when checkpoint data can be written to any buffer. The called comm thread stores the received data shard in a buffer and notifies the master thread. The master thread keeps monitoring this process and when all comm threads put their checkpoint shard into the buffer, the flush thread is called so that all shards in corresponding buffers are merged into one data piece and written to a file. The flush thread merges the checkpoint data shards stored in the circular queue and greedily flushes the complete checkpoint data to Lv.2 in a FIFO manner. Through this operation, the checkpoint data split in Lv.0 is joined as a packed checkpoint data in Lv.1, and immediately flushed as a single file, resulting in a complete and portable save file.

After merging, the flush thread stores the data in Lv.1 memory space to Lv.2 persistent storage. Unlike Lv.0 and Lv.1, Lv.2 is not a VM instance, but a storage service.

B. I/O FLOW

Figure 6 illustrates the process of performing checkpointing right after n^{th} epoch ends. Specifically Figure 6 represents the moment of storing the *j*-th shard of the *i*-th checkpoint data. First, the main thread triggers checkpointing. And then it enters into $(n+1)^{th}$ epoch and continues to train the DNN. Note that it can only proceed until the end of the backward pass of the first iteration of the $(n+1)^{th}$ epoch. For this reason, the main thread uses a lock mechanism. If dumping doesn't complete before the end of the backward pass, the main thread waits until the dumping ends. The gray box in Figure 6 represents the wait time.

At Lv.0, when the *i*-th epoch of the train nodes ends, checkpoint *data(i)* is generated. Since *data(i)* is still in the GPU memory, the address of the data is passed to the dump thread by the main thread. Then the dump thread dumps the data pointed by the address to create packed *data(i)*. In this process, *data(i)* is moved from the GPU memory to the host memory. Remember that each train node sends only one distinct shard. *data(i)* is split by each dump thread, becoming shard *data(i,j)*. It is then transmitted to the receiving comm thread in the remote node.

The remote node stores each *data(i,j)* into a buffer designated to hold *data(i)*. To determine whether there is an empty buffer in the remote node, a dirty bit is used for each buffer. The dirty bit becomes 1 right before the receiving comm thread stores data in a buffer, and becomes 0 right after the data in the buffer is flushed. Therefore, if the dirty bit of all buffers is 1, the receiving comm thread is stalled until the flush thread flushes a buffer. However, we have not observed any cases that buffers become full, since the frequency of triggering checkpointing in Lv.0 is not as high as that of flushing in Lv.1. The checkpointing interval includes one epoch of DDL training. In contrast, the flushing interval which is performed greedily includes the write time of the checkpoint data.

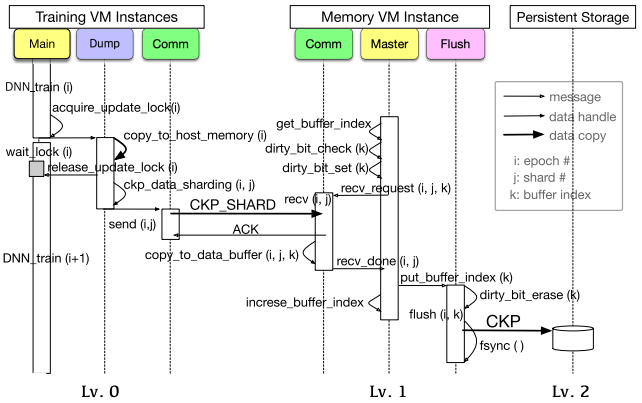


FIGURE 6. Data transfer and message passing flow in ACUTE.

Subsequently, when a checkpoint shard is received by the remote node's receiving comm thread, it is stored in an Lv.1 buffer managed by the master thread. Once all the shards of *data(i)* are stored in the *k*-th buffer, *k* (the index of that buffer) is passed to the flush thread by the master thread. *data(i,j)* is merged with other shards by the flush thread to become the complete *data(i)*. The *data(i)* just constructed is eventually written to the Lv.2 storage by the flush thread. In this way, the *i*-th checkpoint file is stored in the storage persistently. After writing the data in *k*-th buffer, the flush thread sets the dirty bit of the *k*-th buffer to 0 to indicate it can hold a new *data(i)*.

C. RECOVERY

If external cloud storage is used, when a DDL job running on a Spot VM cluster recovers after a failure, direct access to the checkpoint file can be obtained by mounting the corresponding external storage. Additionally, data access is still possible if the local volume of the reliable on-demand instance of *Lv.1* is used as *Lv.2*. In this case, recovering can be achieved by directly accessing the checkpoint data on that volume.

The starting point for recovery varies depending on where a failure due to preemption occurs during the execution of the DDL workload. If a failure occurs during the training process of the n^{th} epoch, the DDL workload can roll back to the $(n-1)^{th}$ checkpoint data. In addition, even if the training of the n^{th} epoch ends, if dumping and sending checkpoint data are not performed completely, users can use the $(n-1)^{th}$ checkpoint data to roll back.

Once transfer of the checkpoint data to the memory of the remote node is completed, the data is protected from

preemption hazard. Note that the remote node is a reliable on-demand instance. Therefore, after the n^{th} epoch ends, if a failure occurs after the checkpoint data is sent to remote memory, the DDL workload can roll back using the n^{th} checkpoint data.

VI. EVALUATION

A. EXPERIMENTAL SETUP

We implemented ACUTE using PyTorch (v2.0.1+cu117), a popular DL framework, along with the MPI library (OpenMPI-v5.0.0rc12). In the ACUTE framework, distributed training is facilitated using PyTorch's Distributed Data-Parallel (DDP) library [19]. ACUTE will be publicly open sourced. The source code can be downloaded from https://github.com/lass-lab/ACUTE. The training VM instances transmit checkpoint data to remote VM instances using the MPI library (OpenMPI-v5.0.0rc12). To ensure resilience in the face of interruptions, ACUTE leverages the User-Level Failure Mitigation (ULFM) technique [20]. This approach safeguards flush operations in progress on the remote VM instances, ensuring that they are unaffected when interruptions occur in the training Spot VM. Users can enable fault-tolerant checkpointing by specifying the `ulfm` option when initiating training through the `mpirun` command. The integration of PyTorch, MPI, and ULFM in the implementation of ACUTE provides a robust and reliable environment for DDL with enhanced fault tolerance.

To evaluate the performance of ACUTE across various workloads, we conducted experiments by setting up a DDL training cluster environment using 16 `g4dn.xlarge` instances on AWS. These instances were chosen for their cost-effectiveness and were configured as Spot VMs. And, an on-demand `c4.xlarge` instance is used as the remote VM instance.

To evaluate the improvement in checkpointing performance offered by ACUTE across various workloads, we employed several popular DNN models with checkpoint data of varying sizes. Table 2 shows the workloads used for evaluation. The models used were as follows: ResNet series: ResNet-18, ResNet-50, ResNet-152, EfficientNet-v2 series: EfficientNet-v2 (small), EfficientNet-v2 (medium), EfficientNet-v2 (large), DenseNet series: DenseNet-121, DenseNet-201, VGG series: VGG-16, VGG-19.

We conducted the experiments using the CIFAR-10 dataset. The batch size for a single GPU processing in one iteration was fixed at 128 data units. The training process consisted of a total of 50 epochs, utilizing the Adam optimizer with the ReduceLROnPlateau scheduler.

B. METRICS

We define the following two metrics to evaluate the performance of ACUTE.

- **Persistence Guarantee Time (PGT):**
  PGT refers to the time that takes for the checkpoint data to be persistently stored. This metric allows us to evaluate the speed at which checkpoint data reaches the persistent area, providing insights into the efficiency of ACUTE. When utilizing the remote VM instance of *Lv.1* as an on-demand instance, it is considered as persistent space starting from level *Lv.1*. However, if the instance is used as a Spot instance, it corresponds to a persistent space starting from storage level *Lv.2*. For all of our evaluation, the remote node was an on-demand VM instance.
- **Checkpointing Stall Time (CST):**
  CST refers to the duration of time that a training job is stalled due to checkpointing. It represents the overhead incurred during the process of saving the training progress. By measuring CST, we can evaluate the delay introduced by checkpointing and evaluate its impact on the overall training time.

Since PyTorch's checkpointing module is synchronous, PGT and CST are the same.

Using the upper two metrics, we compare the four systems below.

- **No Checkpointing** is a system that does not trigger any checkpointing during a DDL training.
- **PyTorch save** is a checkpointing system using the default save module provided by PyTorch during training.
- **MLC-base** is a system that stores data asynchronously using only the multi-level checkpointing scheme.
- **ACUTE** is our proposed system with all optimizations of ACUTE.

TABLE 2. Checkpoint file size statistics of 10 workloads. Checkpoint data size is dependent to DNN model, optimizer and scheduler. CKP means checkpoint file.

Workload	CKP size	Workload	CKP size
ResNet-18	128 MB	EfficientNet-v2-s	232 MB
ResNet-50	270 MB	EfficientNet-v2-m	607 MB
ResNet-152	667 MB	EfficientNet-v2-l	1.31 GB
DenseNet-121	92.2 MB	VGG16	1.50 GB
DenseNet-201	231 MB	VGG19	1.56 GB

C. PERFORMANCE EVALUATION

To demonstrate the effectiveness of the ACUTE design, we measured and compared the changes in makespan, PGT, and CST of the total training time based on variations in the checkpoint data size across different systems.

1) MAKESPAN

In this experiment, we investigated the impact of checkpoint data size on the makespan of the training time for each system. Statistics on the checkpoint file size for different workloads are presented in Table 2.

Figure 7 presents the comparison results of the makespan for each workload on each system. To account for the significant variability in makespan observed in the cloud VM environment, the results of each experiment were normalized to the makespan without checkpointing (*No Checkpointing*).

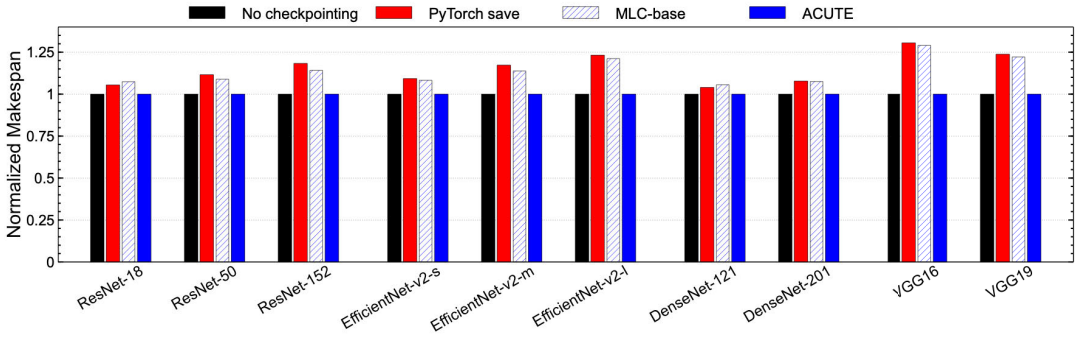


FIGURE 7. Comparison of the makespan of each comparison system for each model. Each run time was normalized to the training time.

The makespan includes both the DL training time and the checkpointing time.

MLC-base exhibited a shorter makespan compared to PyTorch save in 8 out of 10 workloads, with the exception of ResNet-18 and DenseNet-121. During the checkpointing process, MLC-base sends the data to a remote node, while PyTorch save writes the data to network storage. The longer makespan observed in MLC-base for ResNet-18 and DenseNet-121 is attributed to the fact that the time required to send a small amount of data to the memory of a neighboring remote node exceeds the time needed to write it to network storage. For both workloads, the size of the checkpoint data is less than 130 MB.

Next, ACUTE demonstrated significantly superior performance compared to both PyTorch save and MLC-base. The experimental results revealed that ACUTE incurred a delay of less than 0.01% compared to the case without checkpointing. This remarkable improvement means that ACUTE has achieved the significant makespan reduction through three optimizations, especially through Check-Mem.

The only potential scenario where an extraordinary stall may occur is during the update pass of the first iteration of an epoch while dumping data from the GPU memory. However, this situation did not arise in any of the tested workloads. In extreme cases where the batch size is very small and the time required for parameter update falls within one iteration, which is shorter than the dump time, there may be some cases of stalling.

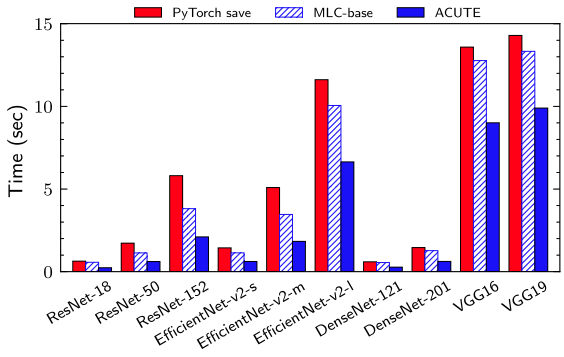


FIGURE 8. Comparison of Persistent Guarantee Time (PGT) for 10 workloads.

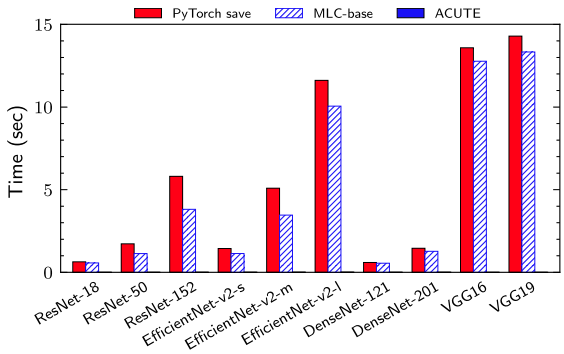


FIGURE 9. Comparison of Checkpoint Stall Time (CST) for 10 workloads. For ACUTE, CST is effectively zero.

2) PERSISTENCE GUARANTEE TIME (PGT)

To evaluate the speed at which each checkpointing system/technique ensures the persistence of checkpoint data, we measured the checkpoint time for each model workload and each system. Figure 8 presents the PGT for each checkpoint technique across 10 workloads. First and foremost, ACUTE exhibited the shortest PGT. PyTorch save took the longest time to ensure persistence among the 10 workloads. For all workloads, MLC-base guaranteed persistence of checkpoint data in an average of 43.3% less time than PyTorch save. This result demonstrates the benefits of an asynchronous multi-level checkpointing strategy. ACUTE, which applied three more optimization techniques in MLC-base, showed a PGT that took 53.1% less time than PyTorch save. Across all workloads, ACUTE consistently yielded smaller PGT compared to both the PyTorch save module and MLC-base. It means that ACUTE reduces the time required for checkpoint data to reach the remote VM instance through three optimizations: Check-Mem, Check-Trans, Check-Pack.

3) CHECKPOINT STALL TIME (CST)

Through CST, we can evaluate how effectively the checkpointing technique performs its tasks without causing delays in the makespan. Figure 9 presents the CST measurements

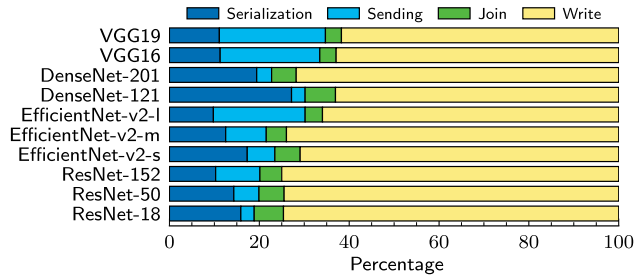


FIGURE 10. Checkpoint time breakdown of ACUTE. Serialization means the process of dumping the checkpoint data from GPU memory to Host memory on Lv.0.

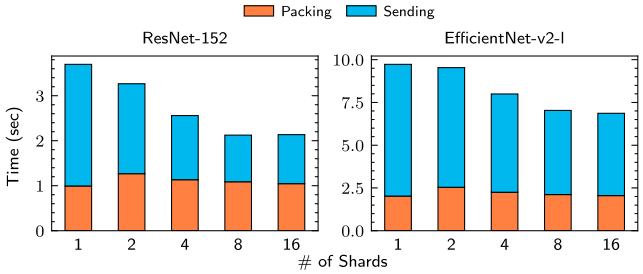


FIGURE 11. PGT variations when changing the number of shards for two workloads. Serialization means the process of dumping the checkpoint data from GPU memory to Host memory on Lv.0.

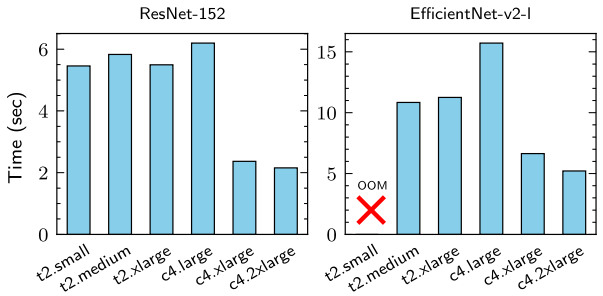


FIGURE 12. ACUTE's PGT comparison with variable peer/remote VM instances.

TABLE 3. Reomte VM instance specifications used in Figure 12 experiment.

Remote VM	vCPU	Memory	Network	On-demand Price (USD/hr)
t2.small	1	2 GB	Low to Moderate	0.0288
t2.medium	2	4 GB	Low to Moderate	0.0576
t2.xlarge	4	16 GB	Moderate	0.2304
c4.large	2	3.75 GB	Moderate	0.114
c4.xlarge	4	7.5 GB	High	0.227
c4.2xlarge	8	15 GB	High	0.454

for the 10 workloads. Surprisingly, the CST of ACUTE is near zero. This result is because almost all operations of ACUTE are overlapped with the training process: Check-Mem. Therefore, ACUTE does not increase the makespan of DDL. In the case of PyTorch save, since the checkpoint data is written synchronously, the writing time to network storage contributes to the CST. MLC-base showed a 17.9% shorter stall time than PyTorch save. This difference between PyTorch save and MLC-base refers to the advantage of the baseline asynchronous multi-level checkpointing scheme.

D. CHECKPOINT TIME BREAKDOWN

We analyzed the time distribution of each step (refer to Figure 4) during a checkpointing operation using ACUTE. The experiment was conducted with 8 Spot VMs, and a sharding degree of 8 was used for all workloads. Figure 10 illustrates that the time taken to write the file to storage is the largest among all workloads. ACUTE has significantly optimized the MPI sending process through Check-Trans.

Meanwhile, we observed that the portion of the MPI sending time increased proportionally with the checkpoint data size for each workload (refer to Table 2). Among the workloads, VGG19 had the highest proportion of MPI sending time, accounting for 23.6% of the total time, with a checkpoint data size of 1.56 GB. On the other hand, ResNet-18 had the smallest proportion of MPI sending time, with a ratio of 3.0% and a checkpoint data size of 128 MB.

E. SHARDING DEGREE SELECTION

ACUTE accelerates the MPI sending process by employing the checkpoint data sharding and parallel transfer technique. To evaluate this Check-Trans's effectiveness, we conducted experiments to observe how the PGT of ResNet-152 and EfficientNet-v2-l workloads decreases with increasing sharding degrees (number of shards). In this experiment, we used 16 `g4dn.xlarge` Spot instances instead of 8. Figure 11 shows that the MPI sending time decreases until a certain point. When the sharding degree was set to 16, there was an improvement of 42.2% for ResNet-152 and an improvement of 29.4% for EfficientNet-v2-l compared to when set to 1.

For ResNet-152, the PGT decreases as the number of shards increases from 1 to 8. However, when the number of shards reaches 16, the PGT slightly increases compared to when it is 8. This means that there is a saturation point in the performance improvement when the number of shards lies between 8 and 16. On the other hand, for EfficientNet-v2-l, the PGT continues to decrease until the number of shards reaches 16. However, in both cases, there is minimal difference in the PGT when changing from 8 to 16.

The saturation point is influenced by the size of the checkpoint data. For instance, ResNet-152 has a checkpoint data size of 667 MB, while EfficientNet-v2-l has a size of 1.31 GB. This implies that randomly splitting a certain amount of checkpoint data into a large number of shards and sending them does not always lead to a speed-up. Additionally, since there is only one remote VM instance, contention may arise if data is pouring into it from multiple Spot train instances simultaneously, which can actually result in performance degradation.

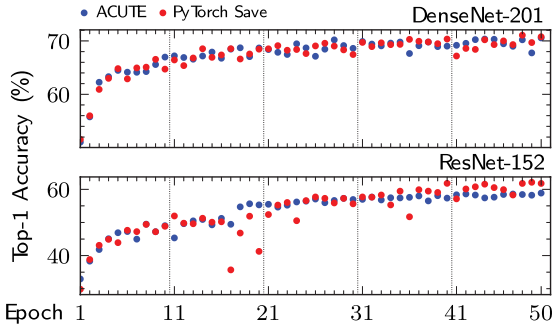


FIGURE 13. Top-1 validation accuracy of two workloads when running with PyTorch save module and ACUTE. Dotted vertical line means when checkpoint & restart happened.

F. REMOTE VM INSTANCE SELECTION

We evaluate how the checkpointing performance of ACUTE varies depending on the type of remote VM instance. Table 3 presents the different types of VM instances used as remote VMs, while Figure 12 illustrates the PGT of ACUTE when executing ResNet-152 and EfficientNet-v2-l workloads using 8 Spot VMs. In Figure 12, we observe that the `c4.2xlarge` instance, which is the most expensive and offers superior overall performance, exhibits the smallest PGT. This performance is attributed to the `c4.2xlarge` instance's high performance in receiving checkpoint data through MPI and writing files to storage efficiently. When the remote VM instance was selected as c4.2xlarge, there was an improvement of 65.2% for ResNet-152 and an improvement of 66.8% for EfficientNet-v2-l compared to when set to c4.large. Notably, during the execution of the EfficientNet-v2-l workload with a `t2.small` instance as the remote VM, an out-of-memory error occurred, preventing ACUTE from proceeding with the checkpointing process. Consequently, there is no corresponding bar for the `t2.small` instance in Figure 12. This outcome highlights the trade-off between price and performance when selecting a remote VM instance in ACUTE.

G. END-TO-END TRAINING

When a failure occurs due to the preemption of a Spot VM, ACUTE performs recovery using previously checkpointed data. To verify the accuracy of DNN model when recovering with checkpointed data and resuming training, we conducted the following experiment. We assumed that an interruption (preemption) occurred once every 10 epochs while performing checkpointing every epoch.

The checkpointed data was reloaded to resume model training. We measured and compared the validation accuracy at every epoch for both PyTorch save and ACUTE. For the experiment, we used 8 Spot VMs and evaluated the ResNet-152 and DenseNet-201 training workloads.

Figure 13 shows the validation accuracy obtained for each epoch. The slight differences observed in each epoch are due to the different mini-batch inputs provided to the model in each epoch. However, it is evident that the validation accuracy values of PyTorch save and ACUTE follow a similar pattern for each epoch as a whole. Therefore, it can be concluded that ACUTE does not impact DNN model's accuracy.

VII. RELATED WORK

A. CHECKPOINTING FOR DEEP LEARNING

There have been several studies related to checkpointing for availability and resiliency [2], [9], [13], [14], [21] for deep learning applications. CheckFreq [9] proposes the use of fine-grained and frequent checkpointing as a strategy to reduce recovery time in the event of potential deep learning job failures. By implementing fine-grained and frequent checkpointing asynchronously, while still maintaining accuracy, the runtime overhead is minimized. However, there is a limit that CheckFreq [9] focuses solely on local checkpointing of a single-node DL job, rather than DDL jobs. Furthermore, since it only supports checkpointing to node-local storage, it cannot be applied to unstable Spot VM clusters. DeepFreeze presented a multi-level checkpointing mechanism that leverages storage at multiple levels, including the local storage of neighboring nodes in a high-performance computing (HPC) environment. However it does not utilize memory space of neighboring nodes as a checkpoint data staging area.

B. MULTI-LEVEL CHECKPOINTING SCHEMES

In distributed environments, multi-level checkpointing [1], [14] – a strategy that utilizes multiple storage tiers to improve data recovery and system resilience – is employed. These studies take advantage of the storage hierarchy in High-Performance Computing (HPC) environments. They are to use as little system resources as possible during the checkpointing process, by leveraging node-local storage or the local storage of neighboring nodes between the compute node and the Parallel File System (PFS). By organizing several storage tiers effectively, they aimed to minimize the overhead of checkpointing itself. By using specific checkpointing modules such as VELOC [22] or SCR [12], they perform checkpointing towards the file system-mounted storage. However, they do not consider leveraging the memory space of neighboring nodes in a multi-level manner.

Fault Tolerant Service in Cloud Computing: In cloud computing, it is important to provide robust platforms in order to provide reliable, effective and seamless services for multiple users simultaneously. For this purpose, research on fault-tolerance, load balancing, and simulators has been conducted to improve the cloud computing environment itself [23], [24], [25]. Several recent studies have specifically used machine learning techniques to optimize fault-tolerance and load balancing of cloud systems [23], [24].

VIII. CONCLUSION

This paper introduces ACUTE, a design optimized for multi-level checkpointing schemes using the remote memory space of neighboring on-demand VMs to enhance reliability in distributed deep learning across cloud Spot VM clusters.

It improves the fault tolerance of distributed deep learning systems by safeguarding against the unexpected preemption of Spot VMs. ACUTE achieves near-zero overhead through three optimization techniques. Our comprehensive experiments, conducted with 8 and 16 VMs on AWS for 10 distributed deep learning workloads across various DNN models, demonstrate that ACUTE incurs minimal checkpointing overhead and only marginally affects the overall makespan.

REFERENCES

[1] Q. Anthony and D. Dai, "Evaluating multi-level checkpointing for distributed deep neural network training," in *SC Workshops Supplementary Proc. (SCWS)*, Nov. 2021, pp. 60–67.

[2] A. Eisenman, K. K. Matam, S. Ingram, D. Mudigere, R. Krishnamoorthi, K. Nair, M. Smelyanskiy, and M. Annavaram, "Check-n-run: A checkpointing system for training deep learning recommendation models," in *Proc. 19th USENIX Symp. Networked Syst. Design Implement.*, 2022, pp. 929–943.

[3] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "PyTorch distributed: Experiences on accelerating data parallel training," 2020, *arXiv:2006.15704*.

[4] A. Sergeev and M. Del Balso, "Horovod: Fast and easy distributed deep learning in TensorFlow," 2018, *arXiv:1802.05799*.

[5] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 1–26.

[6] C. Kim, H. Lee, M. Jeong, W. Baek, B. Yoon, I. Kim, S. Lim, and S. Kim, "Torchgpipe: On-the-fly pipeline parallelism for training giant models," 2020, *arXiv:2004.09910*.

[7] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Analysis and exploitation of dynamic pricing in the public cloud for ml training," in *Proc. Workshop Distrib. Infrastruct., Syst., Program., AI*, Aug. 2020, pp. 1–24.

[8] Z. Cai, X. Li, R. Ruiz, and Q. Li, "Price forecasting for spot instances in cloud computing," *Future Gener. Comput. Syst.*, vol. 79, pp. 38–53, Feb. 2018.

[9] J. Mohan, A. Phanishayee, and V. Chidambaram, "CheckFreq: Frequent, fine-grained DNN checkpointing," in *Proc. 19th USENIX Conf. File Storage Technol.*, Feb. 2021, pp. 203–216.

[10] A. Khan, A. K. Paul, C. Zimmer, S. Oral, S. Dash, S. Atchley, and F. Wang, "Hvac: Removing I/O bottleneck for large-scale deep learning applications," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2022, pp. 324–335.

[11] M. Langer, Z. He, W. Rahayu, and Y. Xue, "Distributed training of deep learning models: A taxonomic perspective," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 12, pp. 2802–2818, Dec. 2020.

[12] A. Moody, G. Bronevetsky, K. Mohror, and B. R. D. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2010, pp. 1–11.

[13] T. Dey, K. Sato, B. Nicolae, J. Guo, J. Domke, W. Yu, F. Cappello, and K. Mohror, "Optimizing asynchronous multi-level checkpoint/restart configurations with machine learning," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2020, pp. 1036–1043.

[14] B. Nicolae, J. Li, J. M. Wozniak, G. Bosilca, M. Dorier, and F. Cappello, "DeepFreeze: Towards scalable asynchronous checkpointing of deep learning models," in *Proc. 20th IEEE/ACM Int. Symp. Cluster, Cloud Internet Comput. (CCGRID)*, May 2020, pp. 172–181.

[15] Amazon. (2023). *Spot Instance Interruptions*. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-interruptions.html

[16] (2023). *Prepare for Interruptions*. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/prepare-for-interruptions.html

[17] (2023). *Spot Instance Interruption Notices*. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-instance-termination-notices.html

[18] J. Axboe. *Flexible I/O Tester*. Accessed: Apr. 2023. [Online]. Available: https://github.com/axboe/fio

[19] PyTorch. (2023). *Distributed Data Parallel*. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html

[20] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure recovery of MPI communication capability: Design and rationale," *Int. J. High Perform. Comput. Appl.*, vol. 27, no. 3, pp. 244–254, Aug. 2013.

[21] B. Nicolae and F. Cappello, "BlobCR: Efficient checkpoint-restart for HPC applications on IaaS clouds using virtual disk image snapshots," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2011, pp. 1–12.

[22] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "VeloC: Towards high performance adaptive asynchronous checkpointing at large scale," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2019, pp. 911–920.

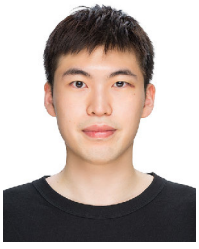
[23] M. A. Shahid, N. Islam, M. M. Alam, M. M. Su'ud, and S. Musa, "A comprehensive study of load balancing approaches in the cloud computing environment and a novel fault tolerance approach," *IEEE Access*, vol. 8, pp. 130500–130526, 2020.

[24] M. A. Shahid, N. Islam, M. M. Alam, M. S. Mazliham, and S. Musa, "Towards resilient method: An exhaustive survey of fault tolerance methods in the cloud computing environment," *Comput. Sci. Rev.*, vol. 40, May 2021, Art. no. 100398.

[25] M. A. Shahid, M. M. Alam, and M. M. Su'ud, "A systematic parameter analysis of cloud simulation tools in cloud computing environments," *Appl. Sci.*, vol. 13, no. 15, p. 8785, Jul. 2023.

YONGHYEON CHO received the B.S. and M.S. degrees in computer science and engineering from Sogang University, Seoul, Republic of Korea, in 2020 and 2023, respectively. He spent half a year at Carnegie Mellon University, Pittsburgh, PA, USA, as a Visiting Scholar, in 2022. He is currently a SW Engineer with the webOS SW Development Group, LG Electronics, Seoul, South Korea. His research interests mainly cover computer systems, including I/O optimization for distributed deep learning, cloud computing, high-performance computing, storage systems, file systems, and memory management.

YOOCHAN KIM received the B.S. degree in computer science and mathematics from Sogang University, Seoul, South Korea, in 2023, where he is currently pursuing the M.S. degree with the Department of Computer Science and Engineering. His research interests include I/O optimization for AI, cloud computing solutions, mathematical modeling, and distributed deep learning.



KIHYUN KIM received the B.S. degree in computer science and engineering from Sogang University, Seoul, Republic of Korea, in 2022, where he is currently pursuing the integrated M.S. and Ph.D. degree with the Department of Computer Science and Engineering. His research interests include service optimizations for AI/ML applications, large-scale data checkpointing, and key-value storage systems.

JINWOO KIM received the B.S. degree in computer science from Iowa State University, Ames, IA, USA, in 2019. He is currently pursuing the integrated M.S./Ph.D. degree in computer science and engineering with Sogang University, Seoul, South Korea. His research interests include distributed deep learning in cloud computing, high-performance computing storage systems, graph neural networks, and data processing unit offloading systems.

HONG-YEON KIM received the Ph.D. degree in computer engineering from Inha University, Incheon, South Korea, in 1999. In 1999, he joined the Electronics and Telecommunications Research Institute (ETRI), where he is currently a Principal Researcher. His primary research interests include operating systems, distributed computing, and artificial intelligence.

YOUNGJAE KIM (Member, IEEE) received the B.S. degree in computer science from Sogang University, Seoul, Republic of Korea, in 2001, the M.S. degree in computer science from KAIST, in 2003, and the Ph.D. degree in computer science and engineering from Pennsylvania State University, University Park, PA, USA, in 2009. Before joining Sogang University, he was a Research and Development Staff Member at the Oak Ridge National Laboratory under U.S. Department of Energy, from 2009 to 2015. From 2015 to 2016, he was an Assistant Professor with Ajou University, Suwon, South Korea. He is currently a Professor with the Department of Computer Science and Engineering, Sogang University. His research interests include distributed deep learning and its system optimization, file and storage systems, database systems, distributed systems, and cloud computing.