IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 29 April 2023, accepted 4 May 2023, date of publication 8 May 2023, date of current version 15 May 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3274201

RESEARCH ARTICLE

Computationally Efficient Neural Rendering for Generator Adversarial Networks Using a Multi-GPU Cluster in a Cloud Environment

ASWATHY RAVIKUMAR AND HARINI SRIRAMAN, (Member, IEEE)
School of Computer Science and Engineering, Vellore Institute of Technology, Chennai 600127, India

Corresponding author: Harini Sriraman (harini.s@vit.ac.in)

ABSTRACT Due to its fantastic performance in the quality of the images created, Generator Adversarial Networks have recently become a viable option for image reconstruction. The main problem with employing GAN is how expensive the computations are. Researchers have developed techniques for distributing GANs across multiple nodes. However, these techniques typically do not scale because they frequently separate the components (Discriminator and Generator), leading to high communication overhead or encountering distribution-related problems unique to GAN training. In this study, the training procedure for the GAN is parallelized and carried out over many Graphical Processing Units (GPUs). TensorFlow's built-in logic and a custom loop were tweaked for more control over the resources allotted to each GPU worker. In this study, GPU image processing improvements and multi-GPU learning are used. The GAN model is accelerated using Distributed TensorFlow with synchronous data-parallel training on a single system and several GPUs. Acceleration was accomplished using the Genesis Cloud Platform and the NVIDIA®GeForceTM GTX 108 GPU accelerator. The speed-up of 1.322 for two GPUs, 1.688 for three GPUs, and 1.7792 for four GPUs using multi-GPU acceleration. The parameter server model's data initialization and image production bottlenecks are removed, but the results' speed-up is not linear. Increasing the number of GPUs and removing the connectivity constraint will accelerate things even more. The bottlenecks are detected using new network lines and resources, and solutions are suggested. Recomputation and quantization are the two techniques to reduce the amount of GPU acceleration in memory. Deployment and versioning are essential for successfully operating multi-node GAN models in MLflow. Properly deploying and versioning these models can improve scalability, reproducibility, and collaboration across teams working on the same model. MLflow provides built-in tools for versioning and tracking model performance, making it easier to manage multiple versions of the model and reproduce it in different environments.

INDEX TERMS All reduce, bottleneck, data parallel, fault tolerance, generative adversarial network, GPU, parallel learning, cloud computing.

I. INTRODUCTION

GAN permits the learning of the distribution curve of a given dataset and the generation of new samples on request from that dataset. A GAN network mainly has a Generator network and the Discriminator network. They play a game in which the Generator strives to produce authentic data (i.e., generated from the accurate data distribution) to feed the discriminator. Training is discontinued when the latter can no longer distinguish between actual and generated data. After recording the data distribution, the Generator has progressed to the point where new samples may be generated [1]. GAN training is a resource and time-consuming procedure. Given the number of potential applicants, increasing this training time is crucial. Scalability is achieved by distributing the computing burden among several machines. The most recent development in large-scale GAN training uses massive models and distributed training approaches performed on centralized deep

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

learning frameworks. Each worker in a centralized network design must interact with all worker nodes throughout each iteration. In cases where network capacity or latency is constrained, performance suffers significantly. Despite recent advancements in decentralized algorithms for neural network training, it is still being determined if decentralized training of GANs is feasible. GANs are built on the same concept of creating, testing, gaining, and exploiting data and knowledge via artificial systems, computational experiments, and simultaneous execution of real and virtual situations, as the parallel transportation theory defines. GANs [1] are very effective at representing high-dimensional data, such as pictures, but are notoriously difficult to train.

Recent research on large-scale GAN training by [2] suggests that distributed large-scale training procedures may be helpful in enormous models. Their method relies on a central network design [3], whereby each worker calculates a local stochastic gradient using input and then communicates the result to a central node. Throughout each iteration, every worker in the centralized design should either explicitly or implicitly interact with the single point. However, performance would be severely hampered by limited network capacity or excessive network latency. Furthermore, network capacity or delay may result in a communication traffic jam. When centralized communication is too expensive, decentralized algorithms are often suggested. With decentralized algorithms, each worker communicates only with its immediate neighbors, eliminating the need for a central node. Recent research [4] has established a decentralized method for training deep neural networks.

Moreover, decentralized algorithms encourage workers to engage only with their trusted neighbours, which is a typically successful method for ensuring privacy [5]. Although decentralized algorithms are beneficial, they have optimization limitations. AlphaGo [6], a seminal achievement in artificial intelligence, achieves considerable success by playing against itself and learning from its errors. The adversarial notion has inspired numerous theoretical research and applications in various disciplines.

Due to their strong modelling capability, generative adversarial networks (GANs) can learn the intrinsic distribution of the original data without sacrificing its variety [7], [8]. Thus, despite its first proposal in 2014, GAN has developed into a research area for generating necessary data in various domains [9], [10], [11]. To address the challenge of detecting malicious software, a transferred GAN with an autoencoder structure was developed to provide a steady process and create the necessary malware data [12]. Data augmentation GANs were developed to produce needed picture data to enhance neural network performance in low-data environments. The Generator consisted of a UNet and ResNet, and a DenseNet was the discriminator [13]. Unprecedented growth has occurred in deep learning research. The design of a Neural network is based on the problem, and the structure of the neural network affects its accuracy and performance [14], [15]. The associated models get increasingly sophisticated and intricate, and the number of layers resulting from hierarchical structure rises consistently. The training speed has increasingly become the main obstacle to the growth of deep learning, and the need for lowering the training time of the model is rising. Google began offering distributed and parallel APIs for TensorFlow, beginning with version 0.8.0.

The parameter server [16] design was a watershed moment for distributed machine learning [3], [17] since it enabled much faster computations than a single centralized server. The parameter server assigns data to workers and organizes the learning process in this model. Local workers do calculations and transmit gradients to the server, which combines them into a global model. Workers then get the most recent model from the server and repeat it until the global model converges. A single model is reproduced across several devices or computers in data parallelism. Each of them analyses distinct batches of data and then combines their findings. There are other variations on this system, which vary in how the various model replicas integrate results, whether they remain in sync between batches or are more loosely connected, and so forth.

Using the Tensorflow [3], [18], [19] data parallel technique, this study investigates several cloud service types for GAN training in a similar context. While maintaining most of the performance based on physics outcomes, the training is linearly accelerated. To compare the efficiency and cost-effectiveness of the proposed techniques, we also test them at scale over many GPU nodes. Data science, cloud-based deployment options, and related economics enable a diverse range of applications to emerge, enabling the maximum capabilities of cloud-based solutions.

The main contributions of this study include the following:

- Data Augmentation using GAN Network is computationally intensive and time-consuming, so accelerating GAN using Multi GPU in Cloud
- Data Parallel Synchronous All Reduce Model for parallelization and to overcome the centralized parameter server model
- Evaluation using different All Reduce Logic: Hierarchical Copy, NCCL, Copy to One Device Logic
- Checkpointing mechanism to ensure model fault tolerance
- Speed up for different GPU configurations (Multi GPU Model)
- Identifying the communication bottleneck issues and proposing a solution for the same.
- Deployment of the proposed model in a model serving platform from which it can be easily moved to the production environment for real-time and industry applications using GAN.

II. SCOPE

The information loading capacity in deep learning applications must be as big as possible. Nevertheless, the capacity for actual model learning is similarly restricted because of the restricted on-device storage of GPUs and other accelerators. From a data flow standpoint, it is not true that bigger input

data sizes result in lengthier training times for single-node training, contrary to popular belief. From a system viewpoint, the fundamental problem is the imbalance between data loading capacity and model train bandwidth. Information loading capacity and model construction bandwidth can be matched during single-node development. In that case, it is unnecessary to undertake an in-parallel training model since distributed data processing always introduces control overhead costs. Because of the mismatch between the data load capacity and the network training capacity, training a model on a node requires significant time. By adding data parallel, model-building capacity may be increased according to the number of accelerators engaged in the same learning operation. There are several uses for GANs, data augmentation, and applications for translating pictures from one category to another. The training of this type of system is challenging due to its computational complexity and a high degree of supervision, which is time-consuming. So, multi-node GPU parallelization makes it faster and more efficacious [19]. This work aims to reduce the training time using multi-node GPU. The parallel models are built on the practical All Reduce logic to overcome the bottleneck faced in the Parameter server model. This, in turn, increases the stability of the model. The bottleneck issues in Speed up and Performance vs. Scaling were identified in the multi-GAN All Reduce logic case, and solutions are proposed in this work.

III. RELATED WORKS

The original GAN model was introduced by Goodfellow et al. in 2014 [1], which consists of a generator network and a discriminator network that play a two-player minimax game.

Since then, there have been many improvements to the GAN architecture, such as Conditional GANs (CGANs), Progressive GANs (PGANs), and CycleGANs, among others. Multi-GPU training refers to using multiple GPUs to train a deep neural network, which can significantly speed up the training process. One popular approach to multi-GPU training is data parallelism, where each GPU processes a subset of the training data and updates the model parameters based on its local gradients. Another approach is model parallelism, where different GPUs process different parts of the model and communicate their results to update the model parameters.

StyleGAN and BigGAN are the most recent GAN variants. StyleGAN is a GAN architecture introduced by [21], which generates high-quality synthetic images with high resolution and varied styles. The StyleGAN model was trained on large-scale datasets such as FFHQ and LSUN, which required significant computing resources. To train the StyleGAN model, Karras et al. used a distributed training approach to train the model on multiple GPUs in parallel. Specifically, they used a data-parallel approach, where each GPU processed a different batch of images and backpropagated the gradients to update the model parameters. BigGAN is another GAN architecture introduced by [2], which generates high-quality synthetic images with high resolution and diverse classes. The BigGAN model was trained on the ImageNet dataset, which contains over a million images and requires more significant computing resources than StyleGAN.

To train the BigGAN model, Brock et al. used a cloud-based approach that involved training the model on Google Cloud TPUs, specialized hardware accelerators for deep learning. In addition, they used a model-parallel approach, where different parts of the model were processed by different TPUs and communicated their results to update the model parameters. StyleGAN and BigGAN required significant computing resources to train and were trained using distributed approaches on multiple GPUs or in the cloud. These approaches allowed the models to be trained faster and more efficiently than possible with a single GPU or machine.

Wang et al. proposed a federated learning approach for training GANs using multiple nodes, where different nodes process subsets of the training data and update the model parameters in a decentralized manner. They demonstrated that this approach could lead to better privacy and communication efficiency than centralized training methods [22].

Karras et al. used a distributed training approach to train StyleGAN on multiple GPUs. Each GPU processed a different batch of images and backpropagated the gradients to update the model parameters. This enabled the generation of high-resolution images with varied styles [21].

Micikevicius et al. proposed a mixed-precision training approach for GANs that uses half-precision floating point arithmetic to reduce the memory needed to store the model parameters. They demonstrated that this approach can significantly speed up the training process without sacrificing accuracy [23].

Overall, multi-node and multi-GPU training have been explored for GANs and can significantly improve training efficiency and enable the processing of larger datasets or models. These approaches are particularly relevant for large-scale applications of GANs, such as generating high-quality images or videos, in [24], accelerating a document clustering algorithm that utilizes flocking using GPU clusters. Our experiments demonstrate that GPU clusters resulted in a significant performance improvement compared to CPU clusters, achieving a speed-up of 30X to 50X. As a result, the execution time for clustering large amounts of documents was reduced from approximately half a day to merely ten minutes.

In [25], a distributed deep learning framework for a heterogeneous multi-GPU cluster combines the advantages of All-reduce and parameter-server methods. In addition, the proposed design performs significant mini-batch training asynchronously to increase the overall utilization of available computing power in the cluster.

IV. MOTIVATION AND CONTRIBUTIONS

The primary motivation behind the study is to ensure:

Scalability: Cloud environments offer scalable computing resources, making them an ideal platform for multi-GPU GAN models. By leveraging the cloud's ability to provide additional resources as needed. In addition, multi-GPU GAN

models can also be deployed to run on multiple servers, allowing for even greater scalability.

Cost savings: multi-GPU GAN models can be expensive to train due to the high computing resources they require. Deploying them in the cloud takes advantage of pay-as-you-go pricing models, only paying for the resources utilized, resulting in cost savings.

Flexibility: Cloud environments provide flexibility in deploying multi-GPU GAN models. The required option can be selected from various hardware configurations, operating systems, and software stacks to meet specific needs. This flexibility allows for experimentation with different configurations to optimize performance.

Deployment and versioning: Proper deployment and versioning are critical for multi-GPU GAN models to function correctly in the cloud. A well-deployed and versioned model should consider the target hardware, software dependencies, and data input/output specifications to ensure optimal performance. In addition, versioning allows the reproduction of specific model versions at any time and tracking their performance over time. The need for multi-GPU GAN in the cloud arises from their ability to handle large datasets, generate more accurate and diverse data, and provide researchers with flexibility and speed to experiment with different models and architectures. Therefore, proper deployment and versioning in the cloud environment are essential to ensure optimal performance and reproducibility of the model.

In the proposed work to accelerate GAN using multi-GPU in the cloud, a data-parallel synchronous all-reduce model can be used to overcome the limitations of the centralized parameter server model. Different all-reduce logic, such as hierarchical copy, NCCL, and copy-to-one device logic, can be evaluated to optimize the process. In addition, a checkpointing mechanism can be implemented to ensure model fault tolerance. As a result, the proposed model can be speeded up for various GPU configurations, and communication bottleneck issues can be identified and addressed. The model is deployed in a model-serving platform; it can be easily moved to the production environment for real-time and industry applications using GAN.

V. METHODOLOGY

The MNIST dataset [26] was used to implement the GAN Discriminator and Generator, generating new data. The data generation and augmentation process is accelerated using the GPU in the Genesis Cloud Platform [27]. The single GPU setup the 1 NVIDIA® GeForce™ GTX 1080 Ti with four vCPUs Intel Xeon Scalable Skylake,12GB memory DDR4-2666,80GIB SSD and multi-GPU setup three nodes in which the first node with 2 NVIDIA® GeForce™ GTX 108 with eight vCPUs Intel Xeon Scalable Skylake,24GB memory DDR4-2666,80GIB SSD, a second node with 3 NVIDIA® GeForce™ GTX 108 with 12 vCPUs Intel Xeon Scalable Skylake,36 GB memory DDR4-2666,80GIB SSD and the third node with 4 NVIDIA® GeForce™ GTX 108 with 16 vCPUs Intel Xeon Scalable Skylake,48 GB memory DDR4-2666, 80GIB SSD. The cloud specifications used Public IPv4 address 147.189.195.218, Private IPv4 address 192.168.8.46 connected using SSH keys to the Ubuntu 3 Operating system. The cloud location Iceland-HAF1, hostname gc-prickly-goldstine, with TensorFlow 2.2 image with inbound rules for TCP Protocol ports 80, 443, and SSH port 22.

In the model, x denotes actual data, z denotes the latent vector, G(z) denotes the fake image generated, D(x) denotes the Discriminator function to identify the actual image, D(G(z)) denotes the Discriminator function to find the fake image, error (a,b) denotes the error generated between a,b. GAN works on the principle that Generator and discriminator are opposite and are competing with each other. The discriminator network the generated images and the actual images. The Generator network generates images using a noise z, which is used to fool the discriminator network. The discriminator D and generator G compete with each other and are playing a min-max game using (1). Where G represents Generator and D Discriminator, respectively, Pdata(x) represents the actual data set and P(z) generator distribution, and x is the data sample for the discriminator, z noise vector.

$$minG\ maxDV(D,G) = Ex \sim pdata(x)[logD(x)] + Ez \sim p(z)[log(1 - D(G(z))] \quad (1)$$

The Discriminator network uses the loss function in (2).

$$LD = E(D(x), 1) + E(D(G(z)), 0) \quad (2)$$

Generator - The loss function is given in (3), which is used to minimize the error generated between 1, the discriminator identification of fake and accurate data.

$$LGE\ (D\ (G\ (z))\ ,\ 1) \quad (3)$$

Algorithm 1

Begin
For N epochs
  For k steps
    Sample m noise vector z from noise prior $p_g(z)$
    Sample data x from generating data distribution $_{data}(x)$
    Discriminator Module:
      Update the D function using ascending SGD
      $\nabla_{\ominus d}\ 1/m \sum_{i=1\ to\ m} \log(x)] + \log(1\text{-}D(G(z))$
  Sample m noise vector z from noise prior pg(z)
  Generator Module:
    Update G using the descending SGD function
    $\nabla_{\ominus d}\ 1/m \sum_{i=1\ to\ m} \log(1\text{-}D(G(z))$
  Repeat till N
End

In the GAN, only one network is trained at a time, so it works like a min-max game, as shown in Fig. 1. The Steps are given in algorithm 1.

GAN Training consists mainly of the discriminator and the generator part. In the discriminator part, the main two

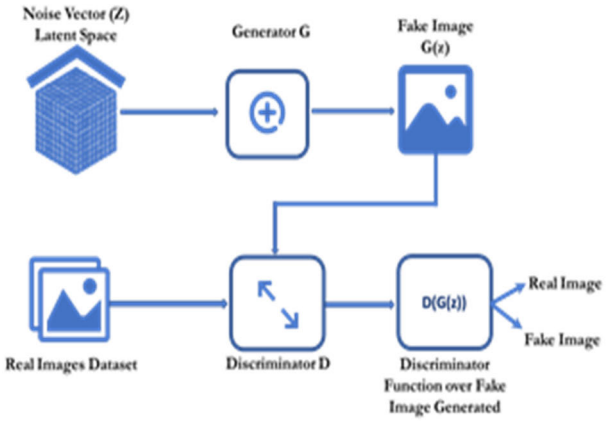

FIGURE 1. GAN model.

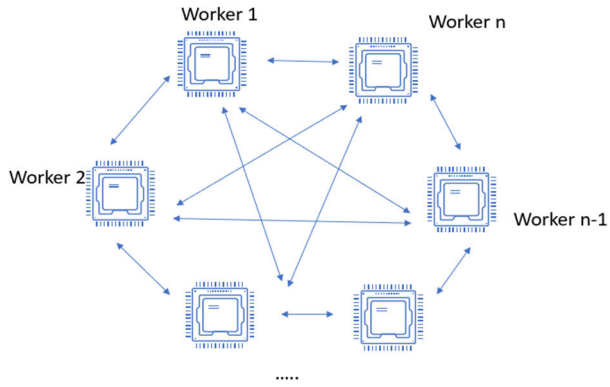

FIGURE 2. All reduce.

steps are training the actual data and the fake generated data. In the real data training, the label'1' is used for real, and '0' is used for the fake generated images. In the generator training in GAN, only the randomly generated image is used for training and later passed to the discriminator. The models combined form the GAN, generate fake images, and keep updating the weights of the generator network based on the new images. This paper improves the training time for the proposed GAN structure using the distributed TensorFlow for multi-GPU and the Fault Tolerance mechanism. Furthermore, the model introduces checkpoints to ensure the worker nodes are active and the fault tolerance is implemented.

The proposed GAN – the Generator and Discriminator structure is shown in Fig 2. In the Generator Network and Discriminator Network, the neural network model sigmoid activation with the binary cross-entropy loss with Adam optimizer. The CUDA version 10.1 and Diver version 430.50 in Nvidia SMI 430.50 is used for the GPU configurations. The Model parameters are given in Table 1 and Table 2.

The parallelism technique speeds up and improves the data-generating process. There are several methods for parallelizing training using several GPUs currently available. The model is copied to each GPU in the data-parallel scenario, and the batch size is equally divided among the number of devices and distributed to the GPUs allotted. Each GPU processes each batch of data; during the backpropagation of such information, each GPU's data is reduced so that only

TABLE 1. Model parameters.

Parameter	Data
Input size	28x28
Batch size of single GPU	64
Learning rate	0.0001
Number of epochs	20
Adam optimization	β1=0.9, β2=0.999
Number of train samples	60,000
Step Size	20
Global_batchsize	batch size of single GPU * number of GPU
Total parameters	2289769
Trainable parameters	2051384
Non-Trainable parameters	238385
Input size	28x28
Batch size of single GPU	64
Learning rate	0.0001
Number of epochs	20

TABLE 2. GAN parameters.

Generator		Discriminator	
Layers	Details	Layers	Details
Input Layer	(None, 50)	Input Layer	(None,28,28,1)
Dense Layer	(None, 1200)	Flatten Layer	(None, 784)
Batch Normalization	(None, 1200)	Dense Layer	(None, 256)
Dense Layer	(None, 1000)	Dropout	(None, 256)
Batch Normalization	(None, 1000)	Dense	(None, 128)
Dense Layer	(None,784)	Dropout	(None, 128)
Reshape	(None,28,28,1)	Dense	(None,1)
Activation	(None,28,28,1)	Activation	(None,1)

one model is replicated across all GPUs. Data parallelism is the best approach because the neural network model is small. Loss functions [28] work well when a single GPU is used to train the model since there is no data distribution. The discriminator must be able to identify which images do not correspond to the ground truth and label the images made by the Generator as false.

In contrast, the ground truth images are marked as accurate. Both networks are trained using the output of the discriminator network. Binary cross-entropy is the loss function used. The function returns the error calculated among the discriminator net-work output value and its desired value. As a result, the binary cross-entropy function in TensorFlow calculates the dimensions of the input and batch tensors and averages the likelihood of the discriminator network's output.

A. INPUT PIPELINE IN GPU

The GPU's massive number of cores enables image analysis in parallel across the different cores, resulting in a rapid processing speed. It takes far less time to process photos on the GPU than to retain the images and transfer the findings between the CPU processor and the GPU. Therefore,

improving the loading of pictures and the data exchange between the CPU processor and the GPU is essential. In the original implementation, the CPU collects all training pictures and completes their changes (standardization, scaling) before loading them onto the GPU. Every epoch, these processes are repeated, depleting Processor resources. Therefore, the cache is implemented such that after the photos have been analyzed, they are kept and accessible throughout the subsequent epoch. Furthermore, once the computer is ready to run a batch, the CPU must analyze the pictures and transfer them to a GPU, so during this period, the GPU is idle and squandering its capabilities. Therefore, the preloading of the pictures to the GPU has indeed been implemented into the program, so the GPU does not need to wait for the CPU to provide the photos before processing can begin.

B. DATA PARALLEL MODEL

The dataset is divided into ''N'' portions in the parallel data model, where ''N'' is the count of GPUs. Finally, these components are allocated to parallel computing systems. Gradients are calculated for each model copy, and the gradients are exchanged amongst all the models. The aggregate of these gradients' values is calculated in the end. The forward propagation uses the same variables for each GPU or node. Every node receives a tiny batch of data, which is then used to compute its gradient and send it back to the central node. Distributed training uses synchronous and asynchronous *[29], [30]*.

When training synchronously, the model provides various data components to each accelerator. Each model contains an identical replica of the model and has only been trained using a portion of the data. Every component begins the forward pass simultaneously and calculates a unique output and gradients. An all-reduce technique is used in synchronous learning to gather all the learnable parameters from different workers and processors. Synchronous learning has many benefits but is challenging to scale and occasionally leads to idle workers. Asynchronous eliminates the need for workers to wait on one another during repair outages, capacity constraints, or competing objectives. Asynchronous training might be a superior option, mainly if machines are smaller, less dependable, and more constrained. Synchronous training is better suited when the gadgets are more powerful and have a stable connection. The Synchronous Data Parallel Model algorithm is given in Algorithm 2.

C. DISTRIBUTED TENSOR FLOW

The deep learning capabilities of the TensorFlow framework have been extensively used in several disciplines. Its native distributed solution has trouble extending for big models due to difficulties with poor GPU usage and sluggish distribution compared to operating on a single system. TensorFlow is an open-source project with exceptional deep-learning capabilities. TensorFlow currently allows distributed iterative training, which has the issue of lengthy training time has been

Algorithm 2

Begin
For N epochs
  For N batches
    Replicate the model and scatter the dataset in the GPU
    Execute the data model copies of the training script in each
      GPU device:
      Read the data set
      Execute the forward pass of the model
      Calculate the loss and execute the backward pass to calculate the model parameters' gradients.
    Average gradients among those multiple GPU devices using
    inter-GPU communication and Gradient All Reduce logic.
    Update the model parameters
  Repeat till N
End

resolved, which might lessen the training duration; there is still room for improvement. TensorFlow is frequently utilized in many fields because of its versatility. TensorFlow is still in its infancy, and the distributed implementation needs help to extend for big models, given such concerns as poor GPU usage and slowness distribution as opposed to single-machine operation.

GPU contains many cores, allowing image processing across multiple cores in parallel, resulting in a rapid processing speed. The time required to process images on the GPU is less than required to store photos and communicate findings between the CPU and GPU. The distributed tensor flow *[19]* implements the distributed framework on GPU. The Mirrored strategy is used in GPU, where the synchronous training with NVIDIA NCCL uses the all-reduce logic. Here the scope is used for model initialization. TensorFlow is a graph-based machine learning system in which a user constructs a graph of tensors representing multi-dimensional data arrays and operations that consume and generate tensors. A user may request the value of a tensor, which requires executing all ancestor actions. tf.distribute.Strategy is a TensorFlow application programming interface for distributing training over several GPUs, computers, or TPUs. MirroredStrategy facilitates synchronous distributed training over many GPUs on a single machine. It generates a single copy per GPU device. Each model variable is replicated across all copies. Together, these variables compose the MirroredVariable concept variable. These variables are maintained in sync by performing identical modifications. Utilizing efficient all-reduce techniques, variable updates are sent across devices. All-reduce aggregates tensors across all devices by adding them together, making them accessible on any device. It is a fusion approach that is very efficient and can considerably minimize synchronization overhead. Several all-reduce algorithms and implementations are available depending on the communication between devices.

D. ALL REDUCE

The Mirrored strategy is used in GPU, where the synchronous training with NVIDIA NCCL *[31]* uses the all-reduce logic. The all-reduce function of parallelization is often used to efficiently transmit gradient information and summation amongst computing. Each device gets the total of all devices' gradients and updates local variables with the outcome of gradient summation, as shown in Fig 2. All reduction algorithms need workers to share the storage and maintenance of global parameters. In this approach, each worker distributes their gradients and performs a reduction operation. All reduction lowers the target array in each worker to a single set and returns that array to each worker. In this method, all workers communicate their gradients to a single worker, the driving worker responsible for gradient reduction and providing updated gradients to every employee. However, the issue with this strategy is that the driver has become a bottleneck as the number of tasks rises, as its communications and implementation of the reducing operation increase. A less naïve option is the ring-all reduce method, wherein the workers are arranged in a ring. Each worker is responsible for a subset of parameters that are only shared with the subsequent work in the ring. This approach is a powerful instrument that significantly decreases synchronization overhead. The total number of processes is P with an array (Ap) of size N, and the element is denoted Ap, i. A binary operator Op, the aggregate operation, is applied to obtain the result B, as shown in (4).

$$Bi = A1, i\ Op\ A2, i\ Op\ A3, i\ Op\ A4, i\ Op\ Ap, i \quad (4)$$

This work uses the three reduce algorithms, the Reduction to One Device, Nccl All Reduce, and the Hierarchical Copy All Reduce Algorithm. In tf.distribute.ReductionToOneDevice() approach always transfers values to a single device for reduction, then broadcasts the reduced values to their destinations. It does not support batching efficiently. In tf.distribute.NcclAllReduce() approach uses Nvidia NCCL is used. Tensors will be repacked or aggregated for more efficient cross-device transfer via the batch API. In tf.distribute.HierarchicalCopyAllReduce() function works along the boundaries of some hierarchy, and it reduces to a single GPU and broadcasts back to each GPU along the same route. Tensors will be repacked or aggregated for more efficient cross-device transfer via the batch API.

E. FAULT TOLERANCE

The Failure recovery mechanism in this model works on Checkpoints. In failure detection, the variable node has a save node and periodically writes variable contents to persistent storage. Each variable also has a Restore node associated with it. In the first iteration, after a restart, restore nodes are enabled. The Checkpoint Files have a Binary file that includes a mapping between variable names to tensor values. Checkpoints record the precise value of all parameters (objects of type tf. variable) used by a model. Checkpoints do not include any description of the calculation required by the model. They are often only relevant when the source code that will utilize

```
checkpoint                      ckpt_4.data-00000-of-00001
ckpt_10.data-00000-of-00001     ckpt_4.index
ckpt_10.index                   ckpt_5.data-00000-of-00001
ckpt_11.data-00000-of-00001     ckpt_5.index
ckpt_11.index                   ckpt_6.data-00000-of-00001
ckpt_12.data-00000-of-00001     ckpt_6.index
ckpt_12.index                   ckpt_7.data-00000-of-00001
ckpt_1.data-00000-of-00001      ckpt_7.index
ckpt_1.index                    ckpt_8.data-00000-of-00001
ckpt_2.data-00000-of-00001      ckpt_8.index
ckpt_2.index                    ckpt_9.data-00000-of-00001
ckpt_3.data-00000-of-00001      ckpt_9.index
ckpt_3.index
```

FIGURE 3. Checkpoints in the proposed model.

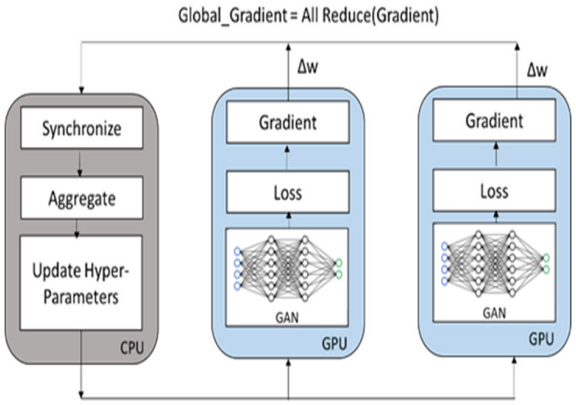

FIGURE 4. Proposed model.

the stored parameter values is available. The checkpoints in the model are shown in Fig 3.

The single GPU and the distributed tensor flow Mirrored Strategy for the parallel implementation on the multi-GPU node are done in the GPU implementation. In the multi-GPU, the data-parallel model was tried using the distributed TensorFlow with mirrored strategy using a single host with multiple devices using the synchronous logic shown in Fig.4. The function tf.distribute.MirroredStrategy is used for in-graph replication and concurrently training multiple GPUs. This API maintains a training distribution abstraction across several processing units. This API replicates model parameters on each GPU. Then, it combines the gradients from all GPUs, obtains a combined value, and applies it to every model copy. The model described inside the scope of the strategy is replicated across all threads, and each change affects all copies. The loss function and optimizer are specified in the strategy's scope, and the fit function is required for training.

F. DEPLOYMENT AND VERSIONING

Multi-node GAN in Mlflow requires careful deployment and versioning to ensure the successful operation of the model. Here are some considerations for deployment and versioning multi-node GAN in MLflow:

Deployment: Multi-node GAN models require specialized hardware and software environments to run effectively. A well-deployed model should consider the target hardware, software dependencies, and data input/output specifications to ensure optimal performance. MLflow provides a platform-agnostic approach to model deployment, allowing deployment across different environments.

Scalability: Multi-node GAN models are designed to scale, and versioning helps manage the various versions of the models deployed in the system. It also allows deploying multiple model versions and tracking their performance over time. MLflow provides built-in tools for versioning and tracking model performance, making it easier to manage multiple versions of the model.

Reproducibility: Versioning allows the reproduction of specific model versions at any time. Keeping track of the changes made to the model over time can identify the specific version that produced a particular result, which can help debug and troubleshoot. In addition, MLflow allows for tracking of the input data, code, and environment variables used to train the model, making it easier to reproduce the model in different environments.

Collaboration: Multiple teams may be working on different versions of the same model. Versioning ensures that everyone is working on the model's latest version, making it easier to merge different versions of the model. MLflow provides a collaborative environment where teams can work together on the same model and track changes made to the model over time.

VI. BOTTLENECK ISSUES

The main bottleneck issue in parallel data processing in GPU is the communication cost and limited device memory.

A. COMMUNICATION COST

Communication costs within the All-Reduce system. All nodes are employees within the All-Reduce design. Therefore, all the employees experience gradient synchronization. For model synchronization to be completed, All-Reduce architecture needs the following different communications steps:

1. The reduced operation is performed from the root node and later aggregates the received gradients from the other nodes.

2. Sent the aggregated updated gradient to all nodes.

The time consumed during the All Reduce design has two parts: t1(the time to perform aggregation of the gradient) and t2 (the time to update the model). So, the total time for All reduces model synchronization is t was given using (5), (6), and (7). N represents the number of GPU devices, g is the gradient, and Bandwidth_GPU is the network bandwidth per GPU.

$$t1 = (N * g)/(Bandwith\_GPU) \tag{5}$$

$$t2 = ((N - 1) * g)/(Bandwith\_GPU) \tag{6}$$

$$t = t1 + t2 \tag{7}$$

Within the All-Reduce approach, model synchronization typically occurs after each training phase. Model synchronization also involves a substantial communication overhead. More than fifty percent of the final neural network training time is accounted for model synchronization, according to the most recent research. This occurs due to the inefficient current communication schemes. Due to network congestion, this cost might be exacerbated if numerous GPUs/nodes use the same physical connection (for instance, multiple GPUs use the very same PCI-e connection for model synchronization inside a computer). One possible solution is to maximize the use of all communication lines inside a data parallel training model. Next, expand it to use idle connections on the CPU Processor side of the client. Only homogeneous connections have been explored so far for model synchronization. Homogeneous connections in this context refer to connections with the same network capacity. There are several connection types with varying network connection speeds. This is what is known as a heterogeneous network. NVIDIA DGX-1 machine is an example of this. There are two distinct types of connections among GPUs: PCIe bus (Common connection with 10 GB/s of capacity) and NVLink (GPU-exclusive). Using a heterogeneous network solves the communication bottleneck to a great extent. However, the communication bottleneck leads to stragglers and stale gradient problems in the models *[32]*.

B. ON-DEVICE MEMORY

Another major bottleneck is the on-device memory issues. CPU capacity is frequently measured in tens or even thousands of gigabytes. GPU memory sizes are often relatively modest relative to this enormous amount of memory. Even the popular GPU options like NVIDIA 1080 have only 8GB of memory, NVIDIA K80 - 12 GB, NVIDIA V100- 16 GB, and NVIDIA A100- 40 GB have limited memory. Furthermore, during DNN learning, the intermediary outputs created (such as extracted features) are often several orders of magnitude larger than the initial data input. Therefore, it exacerbates the GPU storage restriction. There are primarily two approaches to decreasing accelerator memory usage: Recomputation and quantization.

Recomputation is deleting unused tensors and recalculating the result when they are required again. Quantization implies that decreased physical bits are used to express a specific value. For instance, if a specific integer number requires 4 bytes, quantization allows the expression of the same value using just two bytes. For example, consider a neural network with four layers. We calculate the activation function in the forward pass of each layer and store it in intermediate results to calculate the loss during the backward pass. The backward propagation begins using the training data supplied from the last layer (layer 4). The gradient for layer four is calculated using the activation function in that layer calculated during the forward pass and the loss value. Once the gradient is calculated for layer 4, the activation of

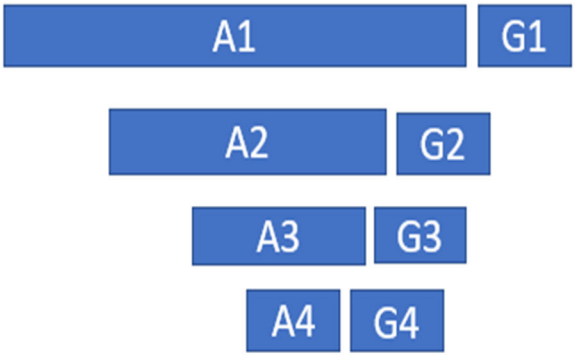

FIGURE 5. Proposed memory usage using both activation and gradient.

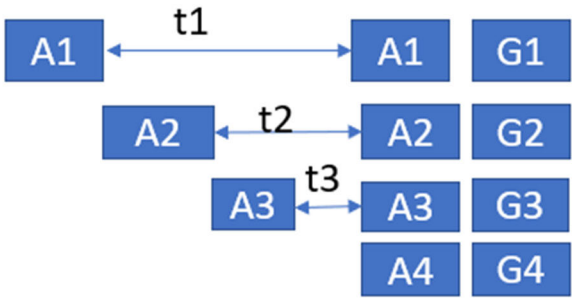

FIGURE 6. Memory usage after recomputation.

layer four can be deleted, and memory can be freed. For the gradient calculation of layer 3, the inputs required are the gradient in layer four and activations in layer 3.

Similarly, after calculating the gradients in layer 3, the activation of layer three can be deleted. From the steps, it is evident activation in layer one is only required once the calculation of gradient 1. So, in the recomputation method, the activations can be recomputed when required and deleted. Memory can be free after calculating in the forward pass, as shown in Fig 5 and 6.

After Recomputation, the memory (storing activation 1) can be freed for layer 1 for time t1, layer 2 for time t2, and layer 3 for time t3. For large-size deep neural networks, the memory can be freed for a significant amount of time. The method reduces memory usage but leads to computation overhead since activation is recalculated twice for each layer.

Quantization is a lossy optimization, which could lose important data to express the gradients with fewer bits. When necessary, Recomputation is undertaken to duplicate the earlier findings. Thus, the calculation process is lossless. However, as discussed above, most quantization approaches could be better. Therefore, Recomputation and quantization may both minimize the memory footprint of a device. In contrast, the computational cost of quantization is often substantially lower than Recomputation. This is mostly because quantization is optional for computationally intensive network layers.

VII. RESULT ANALYSIS

The single GPU and the distributed tensor flow Mirrored Strategy for the parallel implementation on 2,3,4 GPU nodes are done in the GPU implementation. The single GPU

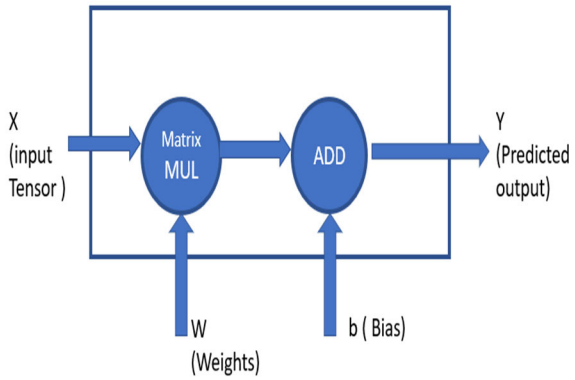

FIGURE 7. Single node execution.

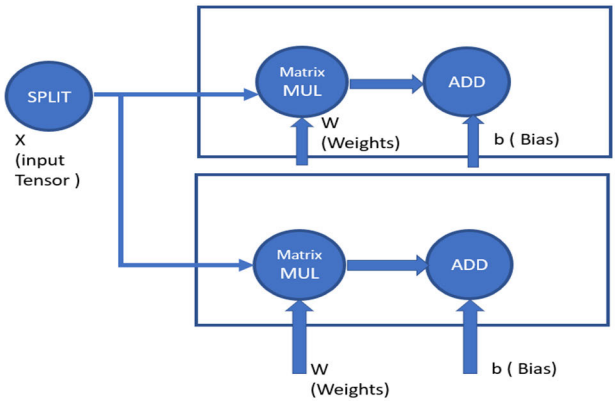

FIGURE 8. Multi-node execution.

training code uses the distributed Tensor Flow in multi-GPU using tf. distribute and strategy API. The generated code will be cloud platform-independent, allowing execution in any environment without modification.

In TensorFlow, this execution is explained in terms of its computational graph. The matrix multiplication operation can be taken as an example to explain how it accepts the X and W tensors, which are the training batch input and weights similar to the single neuron structure. The resultant tensor is then supplied to the add operation using the bias terms tensor b (8). This operation yields Ypred, representing the model's predictions as shown in Fig 7.

$$\boldsymbol{Ypred} = \sum \boldsymbol{XW} + \boldsymbol{b} \tag{8}$$

X represents the input data, W represents t, the weight matrix, and b represents the bias.

In the multi-node execution, half of the input dataset X is given to GPU 0, while the other half is transmitted to GPU 1. In this instance, each GPU performs identical operations on different data slices, as shown in Fig 8. However, here the data-parallel processing is done with each GPU calculating the gradient separately and is aggregated synchronously.

MirroredStrategy is an approach for data parallelism. MirroredStrategy will thus create a clone of the model on both GPUs when the model.fit is called. The CPU (host) prepares the tf.data file, dataset batching, and GPU data transmission. The following gradient modifications will co-occur.

This implies that each worker device computes the forward and backward runs through the model on a unique slice of the input data. In a procedure called AllReduce, the gradients calculated for each of these slices are then aggregated across all of the devices and reduced. The optimizer then conducts parameter updates with these decreased gradients, maintaining synchronization across devices. Since each worker can only move to the next training step once all other workers have completed the current step, gradient computation becomes the most time-consuming aspect of distributed training for synchronous techniques. When using the tf.distribute.Strategy API and tf.data for distributed training, the batch size now corresponds to the global batch size. On providing a batch size of 10 and two GPUs, each machine will process five instances in every step. In this instance, the global batch size is ten, and the per-replica batch size is 5. Scale the batch size by the number of copies to get the most out of the GPUs. In the single-node GPU, the global batch size is 64; for two GPU nodes, the global batch size is 128; for three GPU nodes batch size is 192; for four GPU nodes, the batch size is 256. The global batch size is calculated using (9).

$$Global\_batchsize = batch\ size\ of\ single\ GPU * number\ of\ GPU \tag{9}$$

The code runs sequentially and becomes a bottleneck as the number of mirrored images increases. On the other hand, the generator initialization time grows proportionally with the GPU count because it is run consecutively. The custom training loop, which uses tf.function to modify the forward step in training mode directly, was designed to alleviate this bottleneck. The tf.function also includes all previously completed stages. The data pre-processing and distribution procedures represent the remaining bottleneck at this point. The speed-up will be smaller than ideal due to the communication delay between the CPU and GPU. The performance metrics of the single and multi-node GPU are given in Table 3.

For the Multi-node GPU, the global batch size is batch size X no of devices. Each GPU uses a batch-wise of 64, so the multi-node has a global batch size of $64 \times 2 = 128$. So, the 2 GPU node process doubles the data processed by the single node GPU. For example, the GAN model was trained for 20 epochs with a step size of 20. Single node GPU took 94.56 seconds to process 64 batch size while the 2 GPU node took 143.01 seconds. So, in a single GPU, the single node takes $94.56 \times 2 = 189.12$ seconds to process the same data. Similarly, the time for processing the same data in 3 and 4 GPU is 283.68 and 378.24 seconds, respectively.

$$\text{Speed Up} = \text{Ts/Tp} \tag{10}$$

The speed-up is calculated using (10), where Ts represents the time taken for sequential execution, and Tp is the time taken for parallel execution. In two-node GPU, a speed-up of 1.322. For the three-node GPU, the speed-up obtained is 1.688, and for four GPUs, the speed-up is 1.7792.

Figures 9 and 10 show the loss curve of single-node and multi-node GPU. From the figure, the loss curve of single-node and multi-node GPU in GAN training can provide valuable insights into the training process and the

TABLE 3. Performance Metrics.

GPU	Metric	All Reduce	Hierarchical Copy	Reduction To One Device
1 GPU	Discriminator Loss	0.11	0.08	0.016
	Generator Loss	12.38	11.08	11.311.
	Training Time	94.56s	98.60s	97.20s
2 GPU	Discriminator Loss	0.16	0.11	0.14
	Generator Loss	14.814	12.980	11.66.
	Training Time	143.01s	145.86s	145.40s
3 GPU	Discriminator Loss	0.118	0.132	0.233
	Generator Loss	12.227	12.37	14.453
	Training Time	167.96s	174.62s	174.60s
4 GPU	Discriminator Loss	0.095	0.138	0.136
	Generator Loss	14.488	14.7	14.716
	Training Time	212.58	214.4	213.831

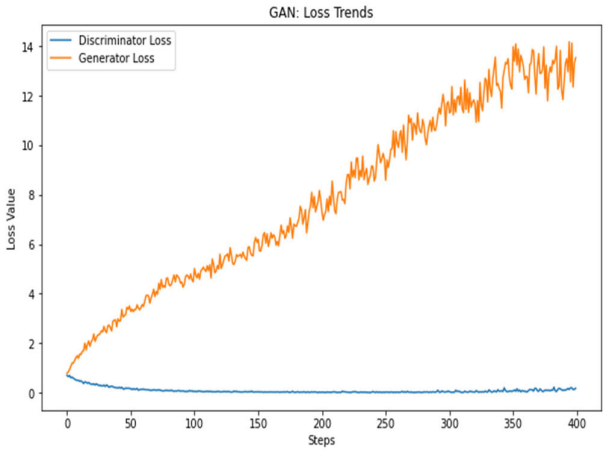

FIGURE 9. Loss curve of single-mode GPU GAN.

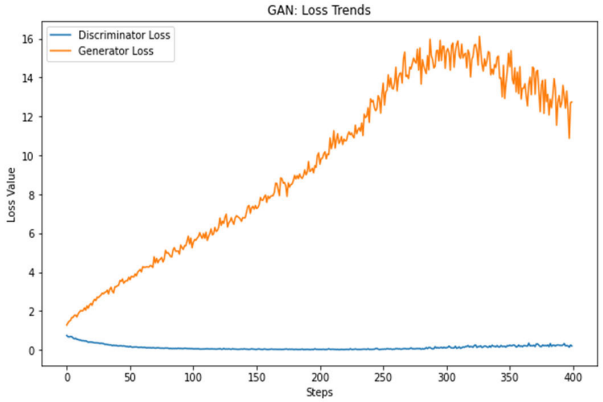

FIGURE 10. Loss curve of multi-node GPU (2-node GPU).

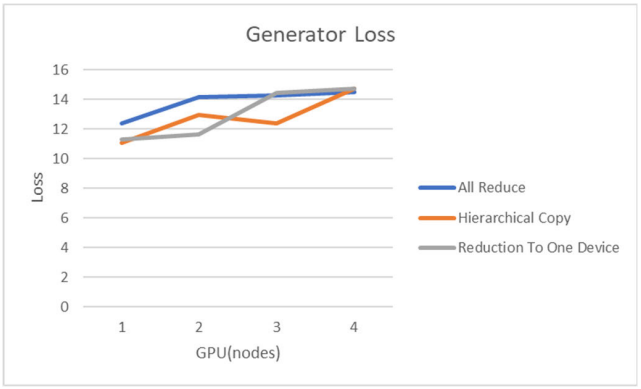

FIGURE 11. Generator loss curve.

model's performance. Here are some key differences between the loss curves of a single node and multi-node GPU GAN training:

Training speed: Multi-node GAN models typically train faster than single-node models due to the increased computational resources available. As a result, the loss curve for multi-node GAN models may show a steeper decline than that of single-node GAN models.

Stability: Multi-node GAN models can be more stable than single-node GAN models due to the improved computational efficiency. This can result in a smoother loss curve and fewer fluctuations in the loss over time.

Scalability: Multi-node GAN models are designed to scale, allowing for larger datasets and more complex models. As a result, the loss curve for multi-node GAN models may show a more gradual decline than that of single-node GAN models due to the increased complexity of the model and the larger dataset.

Convergence: Multi-node GAN models may converge faster and more reliably than single-node GAN models due to the improved computational resources available. As a result, the loss curve for multi-node GAN models may show a more consistent decline over time, indicating that the model is converging toward an optimal solution. The generator loss increases as the number of nodes (GPU) increases.

The comparison of the loss curve for multi-node GPU in the generator model is shown in Fig 11. The increased inter-process communication overhead in a multi-node setup can lead to increased latency and overhead, which can negatively impact the performance of the GAN generator. This can result in a higher generator loss. On the other hand, the discriminator's performance can be improved with a multi-node GPU setup due to the increased computational power available. The discriminator plays a critical role in distinguishing between real and fake data. A more powerful discriminator can provide better feedback to the Generator, resulting in a lower discriminator loss.

A. REAL-TIME APPLICATION

While considering the GAN model for medical data augmentation, the main emphasis is using high-performance

TABLE 4. Comparison with state-of-the-art models.

HPC	Features	Speed Up	Reference
2GPU	Pix2Pix GAN – Data Parallelism All Reduce	1.05	*[33]*
8 core TPU	Multi-core TPU for implementing data parallel in GAN	1.5	*[34]*
1 GPU	Data parallel parameter server model	1.2 - 1.5	*[35]*
4 GPU scalable up to n	Data Parallel Synchronous All Reduce Model	1.7792	Proposed Method

systems with accelerators - GPUs. Federated learning provides a unique approach by attempting to train models on the network edge, which often have far less computational capacity than GPUs. For conventional distributed training, every user can access the entire training sample. With federated learning, each user cannot access the entire training data. Mainly, federated learning provides dispersed and interactive learning without data exchange. The primary characteristic of federated learning is maintaining each user's personal information secret and never sharing it with other users. With federated learning, every worker retains its data locally, called Local data 1 and Local data 2, and all these workers seldom exchange their data input.

Consequently, each machine could only train its local models with localized information. However, more than this, training on local data is required to create a robust GAN model because local data is highly biased. Moreover, local data must be more significant to build a GAN model. Therefore, the proposed model is adequate for federated learning in medical data.

The proposed work is compared with existing state-of-the-art models in GAN in multi-GPU in Table 4.

B. COMMUNICATION AND COMPUTATION COST ANALYSIS

In a multi-GPU system, communication and computation costs play crucial roles in determining the performance of the system. Communication cost refers to the time and resources required to transfer data between different GPUs, while computation cost refers to the time and resources required to perform calculations on the data. The step time, the average communication time and computation time are measured for message transfer among the GPU nodes in ms. The fraction of the compute can be calculated from step time and computation time measured for the nodes. Figure 11 depicts the performance of both single-node and multi-node GPU setups, and it is evident that using multi-node setups results in a considerable penalty due to inter-process communication. Specifically, the ratio of time spent on computation to the total

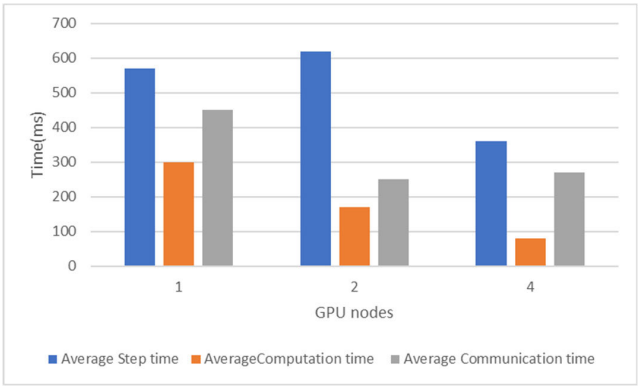

FIGURE 12. Computational and communication efficiency.

time for each step is notably lower for multi-node setups. This is particularly noticeable when multiple GPUs are present on each node, as they have a lower ratio between compute throughput and network bandwidth, leading to a lower ratio between time spent on computation and the total simulation time. In summary, the data in Figure 12 highlights that using single-node versus multi-node GPU setups depends on several factors, including the type of computation, available hardware, and performance requirements.

C. KEY TAKEAWAYS

Parallel distributed GAN in multi-GPU for speed up Automated deployment of the models in production and enabling proper versioning to ensure the continuous deployment and continuous monitor pipeline. In addition, multi-node GPU GAN provides several advantages over single-node GPU GAN, including faster training, scalability, improved accuracy, resource efficiency, and robustness.

The loss curve for single-node and multi-node GPU GAN training can provide valuable insights into the training process and the model's performance. Due to the increased computational resources, multi-node GAN models typically train faster and more reliably than single-node models. As a result, the loss curve for multi-node GAN models may show a steeper decline, be smoother, more gradual, and more consistent than that of single-node GAN models.

The single-node and multi-node acceleration on CNN and LSTM was deeply analyzed for different use cases but was not expanded to GAN *[36], [37]*. The impact of the multi-node on spark and GPU was examined for medical use cases *[35], [38]*. The impact of multi-node TPU on GAN for double precision was developed but lacked deployment and did not address the current bottleneck issues in TPU *[34]*. The proposed model is superior to the existing works since it uses multi-GPU GAN implementation addressing the bottleneck issues, ensures the deployment and continuous retraining of the model, and makes it usable for real-time applications.

VIII. CONCLUSION

In the proposed method, the advancement is made in GAN in computation, loss, and training time using TensorFlow distributed training methodologies. We explored the practical constraints and overhead of the training methodologies from the empirical findings. If the dataset is small enough to fit in a single system and the model has few trainable parameters, then the mirrored strategy technique would be enough to scale training and yield optimum results. The bottleneck issues are identified, and solutions are proposed. The in-memory issues can be solved using Recomputation and quantization. Deployment and versioning are essential for successfully operating multi-node GAN models in MLflow. Properly deploying and versioning these models can improve scalability, reproducibility, and collaboration across teams working on the same model. MLflow provides built-in tools for versioning and tracking model performance, making it easier to manage multiple versions of the model and reproduce it in different environments. Multi-node GPU GANs have immense potential for generating high-quality synthetic data.

In the future, the work needs to be extended to multi-worker training in the distributed synchronous pattern. In the multi-worker cluster pattern, multiple GPUs with the same model and different datasets for synchronous data-parallel training. While MirroredStrategy is a synchronous data parallelism approach, asynchronous data parallelism solutions are also possible. In an asynchronous data parallelism technique, each worker computes gradients from a slice of the incoming data and performs parameter adjustments asynchronously. Asynchronous training provides the advantage of fault tolerance since workers are not reliant on one another, but it might result in stale gradients. When performing distributed learning, the speed with which data is loaded is often crucial, and the data prefetching and caching must be done in the future to obtain better speed-up.

REFERENCES

[1] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014, *arXiv:1406.2661*.

[2] A. Brock, J. Donahue, and K. Simonyan, "Large scale GAN training for high fidelity natural image synthesis," 2018, *arXiv:1809.11096*.

[3] J. Dean, "Large scale distributed deep networks," Tech. Rep., 2012, p. 11.

[4] X. Lian, C. Zhang, H. Zhang, C.-J. Hsieh, W. Zhang, and J. Liu, "Can decentralized algorithms outperform centralized algorithms? A case study for decentralized parallel stochastic gradient descent," in *Proc. Adv. Neural Inf. Process. Syst.*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Red Hook, NY, USA: Curran Associates, 2017, pp. 5336–5346. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/f75526659f31040afeb61cb7133e4e6d-Paper.pdf

[5] A. Koppel, F. Y. Jakubiec, and A. Ribeiro, "A saddle point algorithm for networked online convex optimization," *IEEE Trans. Signal Process.*, vol. 63, no. 19, pp. 5149–5164, Oct. 2015, doi: 10.1109/TSP.2015.2449255.

[6] *Mastering the Game of Go With Deep Neural Networks and Tree Search | Nature*. Accessed: Jun. 16, 2022. [Online]. Available: https://www.nature.com/articles/nature16961

[7] X. Mao, Q. Li, H. Xie, R. Y. K. Lau, Z. Wang, and S. P. Smolley, "Least squares generative adversarial networks," 2016, *arXiv:1611.04076*.

[8] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 2015, *arXiv:1511.06434*.

[9] M. Zhang, N. Wang, Y. Li, and X. Gao, "Bionic face sketch generator," *IEEE Trans. Cybern.*, vol. 50, no. 6, pp. 2701–2714, Jun. 2020, doi: 10.1109/TCYB.2019.2924589.

[10] J. Zhang, Y. Peng, and M. Yuan, "SCH-GAN: Semi-supervised cross-modal hashing by generative adversarial network," *IEEE Trans. Cybern.*, vol. 50, no. 2, pp. 489–502, Feb. 2020, doi: 10.1109/TCYB.2018.2868826.

[11] M. Robin, J. John, and A. Ravikumar, "Breast tumor segmentation using U-NET," in *Proc. 5th Int. Conf. Comput. Methodologies Commun. (ICCMC)*, Apr. 2021, pp. 1164–1167, doi: 10.1109/ICCMC51019.2021.9418447.

[12] J.-Y. Kim and S.-B. Cho, "Obfuscated malware detection using deep generative model based on global/local features," *Comput. Secur.*, vol. 112, Jan. 2022, Art. no. 102501, doi: 10.1016/j.cose.2021.102501.

[13] A. Antoniou, A. Storkey, and H. Edwards, "Data augmentation generative adversarial networks," 2017, *arXiv:1711.04340*.

[14] S. Harini and A. Ravikumar, "Effect of parallel workload on dynamic voltage frequency scaling for dark silicon ameliorating," in *Proc. Int. Conf. Smart Electron. Commun. (ICOSEC)*, Sep. 2020, pp. 1012–1017, doi: 10.1109/ICOSEC49089.2020.9215262.

[15] A. Ravikumar and H. Sriraman. (2023). *Acceleration of Image Processing and Computer Vision Algorithms*. Handbook of Research on Computer Vision and Image Processing in the Deep Learning Era. Accessed: Nov. 21, 2022. [Online]. Available: https://www.igi-global.com/chapter/acceleration-of-image-processing-and-computer-vision-algorithms/www.igi-global.com/chapter/acceleration-of-image-processing-and-computer-vision-algorithms/313986

[16] M. Li, "Scaling distributed machine learning with the parameter server," in *Proc. Int. Conf. Big Data Sci. Comput.*, Aug. 2014, pp. 583–598, doi: 10.1145/2640087.2644155.

[17] J. Dean, "Large scale distributed deep networks," in *Proc. Adv. Neural Inf. Process. Syst.*, F. Pereira, C. J. Burges, L. Bottou, and K. Q. Weinberger, Eds. Red Hook, NY, USA: Curran Associates, 2012, pp. 1–9. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf

[18] M. Abadi, "TensorFlow: A system for large-scale machine learning," in *Proc. Osdi*, vol. 16, 2016, pp. 265–283.

[19] M. Abadi, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," 2016, *arXiv:1603.04467*.

[20] T. Jia, H. Chen, and J. Tang, "A research on generative adversarial network algorithm based on GPU parallel acceleration," in *Proc. Int. Conf. Image Video Process., Artif. Intell.*, Nov. 2019, pp. 397–404, doi: 10.1117/12.2539238.

[21] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks," 2018, *arXiv:1812.04948*.

[22] W. Wang, "Detection of SARS-CoV-2 in different types of clinical specimens," *J. Amer. Med. Assoc.*, vol. 323, no. 18, pp. 1843–1844, 2020, doi: 10.1001/jama.2020.3786.

[23] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," 2017, *arXiv:1710.03740*.

[24] Y. Zhang, F. Mueller, X. Cui, and T. Potok, "Large-scale multi-dimensional document clustering on GPU clusters," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. (IPDPS)*, Apr. 2010, pp. 1–10, doi: 10.1109/IPDPS.2010.5470429.

[25] Y. Kim, H. Choi, J. Lee, J.-S. Kim, H. Jei, and H. Roh, "Efficient large-scale deep learning framework for heterogeneous multi-GPU cluster," in *Proc. IEEE 4th Int. Workshops Found. Appl. Self* Syst. (FAS*W)*, Jun. 2019, pp. 176–181, doi: 10.1109/FAS-W.2019.00050.

[26] *MNIST Handwritten Digit Database, Yann LeCun, Corinna Cortes and Chris Burges*. Accessed: Jun. 16, 2022. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[27] *The Acceleration Cloud | Genesis Cloud*. Acceleration Cloud | Genesis Cloud. Accessed: Jun. 16, 2022. [Online]. Available: https://www.genesiscloud.com/

[28] D. Das, S. Avancha, D. Mudigere, K. Vaidynathan, S. Sridharan, D. Kalamkar, B. Kaul, and P. Dubey, "Distributed deep learning using synchronous stochastic gradient descent," 2016, *arXiv:1602.06709*.

[29] *Performance Analysis of Data Parallelism Technique in Machine Learning for Human Activity Recognition Using LSTM*. Kyung Hee University. Accessed: Jun. 24, 2022. [Online]. Available: https://khu.elsevierpure.com/en/publications/performance-analysis-of-data-parallelism-technique-in-machine-lea-2/fingerprints/

[30] H. K. Omar and A. K. Jumaa, "Distributed big data analysis using spark parallel data processing," *Bull. Electr. Eng. Informat.*, vol. 11, no. 3, pp. 1505–1515, Jun. 2022, doi: 10.11591/eei.v11i3.3187.

[31] *NVIDIA Collective Communications Library (NCCL) | NVIDIA Developer*. Accessed: Jun. 16, 2022. [Online]. Available: https://developer.nvidia.com/nccl

[32] A. Ravikumar, "Non-relational multi-level caching for mitigation of staleness & stragglers in distributed deep learning," in *Proc. 22nd Int. Middleware Conf., Doctoral Symp.*, Dec. 2021, pp. 15–16, doi: 10.1145/3491087.3493678.

[33] M. Lupión, J. F. Sanjuan, and P. M. Ortigosa, "Using a multi-GPU node to accelerate the training of Pix2Pix neural networks," *J. Supercomput.*, vol. 78, no. 10, pp. 12224–12241, Jul. 2022, doi: 10.1007/s11227-022-04354-1.

[34] A. Ravikumar and H. Sriraman, "A novel mixed precision distributed TPU GAN for accelerated learning curve," *Comput. Syst. Sci. Eng.*, vol. 46, no. 1, pp. 563–578, 2023, doi: 10.32604/csse.2023.034710.

[35] A. Ravikumar and H. Sriraman, "Real-time pneumonia prediction using pipelined spark and high-performance computing," *PeerJ Comput. Sci.*, vol. 9, Mar. 2023, Art. no. e1258, doi: 10.7717/peerj-cs.1258.

[36] A. Ravikumar, H. Sriraman, P. M. S. Saketh, S. Lokesh, and A. Karanam, "Effect of neural network structure in accelerating performance and accuracy of a convolutional neural network with GPU/TPU for image analytics," *PeerJ Comput. Sci.*, vol. 8, p. e909, Mar. 2022, doi: 10.7717/peerj-cs.909.

[37] A. Ravikumar, H. Sriraman, S. Lokesh, and P. M. S. Saketh, "Identifying pitfalls and solutions in parallelizing long short-term memory network on graphical processing unit by comparing with tensor processing unit parallelism," in *Inventive Computation and Information Technologies*, S. Smys, K. A. Kamel, R. Palanisamy, Eds. Singapore: Springer, 2023, pp. 111–125.

[38] A. Ravikumar and H. Sriraman, "Attenuate class imbalance problem for pneumonia diagnosis using ensemble parallel stacked pre-trained models," *Comput., Mater. Continua*, vol. 75, no. 1, pp. 891–909, 2023, doi: 10.32604/cmc.2023.035848.

ASWATHY RAVIKUMAR received the B.Tech. and M.Tech. (Hons.) degrees from the University of Kerala, in 2013. She is currently pursuing the Ph.D. degree with the Vellore Institute of Technology, Chennai, India. From 2013 to 2020, she was an Assistant Professor with the Mar Baselios College of Engineering and Technology, Kerala. Since 2020, she has been a Research Associate with the Vellore Institute of Technology. Her research interests include machine learning, deep learning, cloud computing, and high-performance computing. She is a member of ISTE, CSI, ACM, and SCRS.

HARINI SRIRAMAN (Member, IEEE) was born in Chennai, India. She received the B.E. degree in computer science from the University of Madras, in 2003, the M.E. degree in computer science from the College of Engineering, Guindy, in 2009, and the Ph.D. degree in computer science engineering from the Vellore Institute of Technology, Chennai, in 2018. She is currently an Associate Professor with the Vellore Institute of Technology. She has 12 years of teaching experience and one year of industrial experience. She has published more than 20 articles in reputed international journals, conferences, and book series. Her research interests include hardware architectures for accelerated computing, distributed deep learning, and parallel and distributed systems.