Computer Systems Science & Engineering
DOI: 10.32604/csse.2023.034710
Article

Tech Science Press

A Novel Mixed Precision Distributed TPU GAN for Accelerated Learning Curve

Aswathy Ravikumar and Harini Sriraman*

School of Computer Science and Engineering, Vellore Institute of Technology, Chennai, 600127, India
*Corresponding Author: Harini Sriraman. Email: harini.s@vit.ac.in
Received: 25 July 2022; Accepted: 28 October 2022

Abstract: Deep neural networks are gaining importance and popularity in applications and services. Due to the enormous number of learnable parameters and datasets, the training of neural networks is computationally costly. Parallel and distributed computation-based strategies are used to accelerate this training process. Generative Adversarial Networks (GAN) are a recent technological achievement in deep learning. These generative models are computationally expensive because a GAN consists of two neural networks and trains on enormous datasets. Typically, a GAN is trained on a single server. Conventional deep learning accelerator designs are challenged by the unique properties of GAN, like the enormous computation stages with non-traditional convolution layers. This work addresses the issue of distributing GANs so that they can train on datasets distributed over many TPUs (Tensor Processing Unit). Distributed learning training accelerates the learning process and decreases computation time. In this paper, the Generative Adversarial Network is accelerated using the distributed multi-core TPU in distributed data-parallel synchronous model. For adequate acceleration of the GAN network, the data parallel SGD (Stochastic Gradient Descent) model is implemented in multi-core TPU using distributed TensorFlow with mixed precision, bfloat16, and XLA (Accelerated Linear Algebra). The study was conducted on the MNIST dataset for varying batch sizes from 64 to 512 for 30 epochs in distributed SGD in TPU v3 with 128×128 systolic array. An extensive batch technique is implemented in bfloat16 to decrease the storage cost and speed up floating-point computations. The accelerated learning curve for the generator and discriminator network is obtained. The training time was reduced by 79% by varying the batch size from 64 to 512 in multi-core TPU.

Keywords: Data parallel; distributed model; generative model; learning curve; mixed precision

1 Introduction

Deep neural networks might answer unresolved problems by discovering connections in massive datasets. However, such training methods are time-consuming; various parallel and distributed methods are required to accelerate the training process. The recent success of computer vision, and deep learning, in particular, created a new ripple in hardware accelerators. Tensor Processing Unit is a promising solution to the communication and computation challenge posed by data's constant and incremental growth. Generative adversarial networks enable the learning of deep representations without the need for significant training data annotation. This is accomplished by obtaining backpropagation information via a competing procedure involving two nets. Generative Adversarial Networks have facilitated a range of applications in machine learning and natural language translation, among others, owing to its generative model's persuasive capacity to produce realistic examples convincingly chosen from an existing sample distribution. Furthermore, due to its game theoretic optimization technique, GAN (Generative Adversarial Network) delivers excellent performance on data generation-based problems and promotes privacy.

GANs need a huge training dataset to be appropriate for the required application. Data obtained at multiple data centers remain in situ since the data quantities examined would make it challenging to achieve scheduling requirements with data centralization. Some recent efforts [1,2] look at numerous generators and discriminators to increase GAN convergence; nevertheless, they are not designed to operate across datasets distributed on multiple servers. The Parameter Server model is the prevalent model of spreading the computation of conventional neural networks: workers calculate the neural net operations on the data share available and broadcast gradients to the central parameter server. Nevertheless, it leads to colossal communication traffic to the central parameter server. This paper proposes a novel method to train GAN in a distributed manner with the data distributed among TPU in TensorFlow using mixed precision and All Reduce logic.

The main advantages of deploying GAN in TPU are that the tensor operations maximize TPU's power efficiently for matrix multiplications. The second benefit is derived from the data breakdown and communication mechanisms that align with the interconnecting network architecture of TPU. All tensor functions are isolated on individual cores, so each iteration requires a minimum transfer of image-sized data. It is important to note that communication between TPU cores bypasses host CPUs (Central Processing Units) and networking resources. The third benefit is TPU's huge package memory capacity, which effectively handles large-scale issues. The fourth benefit is that TPU is readily customizable using software front ends like TensorFlow. In several sectors, distributed machine learning has been shown to reduce training time and latency. For instance, a TPU-based high-speed object tracking and prediction model has proven superior performance compared to alternative architectures and accelerators. Furthermore, when trained for face expression identification [3] and magnetic resonance image reconstruction [4], TPU-trained models were very efficient in computing time. In this research, we investigate the performance of TensorFlow 2.2 in distributed training utilizing Google Collaborative Notebooks.

The main applications of accelerated GAN using TPU are cybersecurity, animation, multimedia, and image translations in the health sector [5,6]. It could be challenging to achieve a high-quality image in specific evaluation protocols. Poor-quality scans have the consequence of hampering attempts to produce images of high quality. Super-resolution enhances collected pictures and can effectively reduce noise. However, acceptance of GANs in the medical field is difficult because several tests and trials need to address safety issues. GANs are widely used in a medical illness diagnosis. Nevertheless, generative adversarial networks have the potential to save human lives. Drug development is another area of healthcare where GAN may be of use. The networks may be exploited to generate molecular structures for drugs used for disease targeting and treatment. Using the current database, researchers may teach the generator to identify novel chemicals that might be utilized to treat new ailments. Researchers no longer need to manually go through the entire database in search of substances that may be used to combat new illnesses. The system discovers these molecules automatically and aids in reducing the time necessary for medication research and development. All medical applications require timely faster results, so accelerating GAN helps to solve the problem effectively. Hackers use adversarial attacks to enter the systems and destroy their security levels. Hackers inject harmful data into images. This impairs the algorithm's intended functionality by tricking the neural network. These result in the disclosure and compromising of sensitive information that was not intended to be shared. It is possible to train GAN to detect such cases of fraud. They could be used to strengthen the robustness of deep learning models. GAN provides immense benefits to the gaming industry. GAN could also be used to generate 2-dimensional animations. A specific database, including such anime character drawings, is used to train the GAN. Generating emojis from individual pictures is an additional fascinating use of the GAN. The neural network evaluates face characteristics to generate caricatured representations of persons. All these applications require timely results and accelerated learning of the networks. In this paper, we proposed a method to accelerate GAN in terms of performance and training time using multi-core TPU in the distributed platform. The main challenges in the GAN design are addressed when designing the proposed solution. The GAN structure is thoroughly analyzed before the training, and the GAN requires massive memory storage for storing the intermediate calculation, which is effectively addressed using the TPU. The non-convolutional operations and the varying computation structure are mitigated using the bfloat16. The main contribution is:

- Analysis of the effect of the different batch sizes on performance and training time.
- Analysis of the effect of the different batch sizes on the loss curve, learning curve, and optimization.
- Acceleration using XLA compiler and bfloat16.

2 Background

CPUs can successfully finish the learning phase of a short neural network in a reasonable amount of time due to the limited number of trainable parameters. Due to the frequent occurrence of multiple trainable parameters and training data, deep neural networks require more excellent concurrent processing than the CPU to sustain a shorter training period. As a result of the parallelization execution capabilities, (Graphic Processing Unit) GPU-based execution arose. The fundamental advantage of the graphics accelerator-based design is the abundance of execution units. Even though these units are less powerful than a typical CPU core, backpropagation training of neural networks can be accomplished using only basic operations like multiplication. The preparation of the neural network involves just simple matrix multiplication operations; therefore, the process can be successfully parallelized using graphics processors due to the large number of processing units in GPUs. To function well, deep neural networks need more training data than external networks. As a result, during the training phase, the memory capacity of GPU devices and the data transmission between CPU and GPU storage may be a barrier. One possible solution to the transfer capacity issue is to split the training set into smaller batches. However, the appropriate batch size is also unknown due to an imbalanced dataset leading to the overfitting of data or underfitting of the dataset. The low computational capacity of a single GPU card could be a challenge throughout training. GPUs may be vertically scaled to solve capacity issues. Deep neural networks can now be trained using more GPU devices or even more nodes that use GPU cards; the most common implementations are shown in the following sections. The operation of CNN (Convolutional Neural Network) on the GPU and CPU is explained in detail [7].

The efficiency of the hardware resources depends on several factors, but the memory interface is a restricting element. For the processing element to do computations, it must retrieve information from memory. There are two primary concerns. The first factor is memory access latency. Memory access requires much more clock cycles than executing a calculation on the acquired data since the processor must wait before the data is ready for processing. The second factor is energy use. Loading from (Dynamic Random Access Memory) DRAM consumes much more energy than reading from the chip's internal storage [8]. Therefore, the memory interface is the efficiency barrier. Experts in machine learning and computer vision are resolving relatively challenging artificial intelligence issues with more enormous datasets, resulting in a requirement for more complexity and processing capacity. As Moore's Law decelerates, more computation per price per unit and power can no longer maintain their previous pace. This disparity between the availability and requirement of processing capacity highlights the necessity for developing practical deep-learning algorithms on domain-specific hardware for big data.

A tensor processing element is a hardware accelerator that speeds up MAC (Multiply and Accumulate) matrix computations and is one of the most important deep learning operations, as shown in Fig. 1. For efficient resource usage and energy saving, we need to limit the number of memory access. A solution to this problem is the systolic array. The method continues to follow: information is read from memory, a PE (Processing Element) performs an arbitrary function, the data is passed towards the next PE, and the calculation output is posted back to memory after the last PE. Google's TPUv2 produces 180 Tera FLOPS (Floating-point operations per second) has eight cores, each with its 2-dimensional systolic array containing 128×128 PEs. Google's TPUv3 is capable of 420 Tera FLOPS. TPUv3 has twice the number of systolic arrays per core, twice the memory capacity per core, adds high bandwidth memory, and enhances connectivity, among many other enhancements. While training a sizeable neural net, TPU helps to reduce the training time while maintaining accuracy. TPU can quickly do massive quantities of addition and multiplication. The TPU loads variables into a matrix of adders and multipliers while doing calculations. Operations involving multiplication are carried out after data loading. Following each operation, the output is sent to the following multipliers while being added simultaneously. As a result, the output is the total of all parameter and data multiplication results.

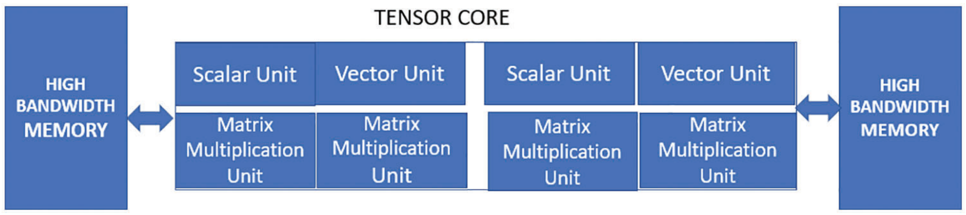
TENSOR CORE
HIGH BANDWIDTH MEMORY
Scalar Unit
Vector Unit
Scalar Unit
Vector Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
Matrix Multiplication Unit
HIGH BANDWIDTH MEMORY

Figure 1: Multicore TPU

For a dataset with a limited number of entries and imbalanced conditions, usually, data augmentation is used. Standard data augmentation approaches include rotation, resizing, zooming, and inverting pictures within the dataset to expand the number of synthesized images for training operations. These strategies improve the precision of machine learning methods and boost generalization capacities. Typically, image classification methods apply classic data augmentation techniques to increase the efficiency of a trained model. However, a further improvement in data can be obtained by adopting synthetic data samples produced by a GAN, which may achieve greater variety and increased training efficiency for machine learning categorization.

The existing GANs are mainly accelerated using the high processing system–GPU, TPU, and FPGA (Field Programmable Gate Array) in a single node execution [9]. Nevertheless, in the case of big data, the single node execution is not sufficient for data storage. Moreover, conventional deep learning accelerator designs are challenged by the unique properties of GAN, like the enormous computation stages with non-traditional convolution layers. LerGAN [10] offers several degrees of GAN acceleration for programmers. Experiments demonstrate that LerGAN outperforms FPGA-based GAN accelerator, GPU platform, and ReRAM-based neural network accelerator by 47.2X, 21.42X, and 7.46X, respectively. Nevertheless, the main drawback of this work was that the loss function was not optimized. In [11], a model for GPU parallel acceleration, which uses the potent computing power of GPU and the benefits of multi-parallel computing, drastically reduces the time required for model training, enhances the training efficiency of the GAN framework, and achieves superior model-based efficiency.

3 Distributed Deep Learning

Current Deep learning models are readily scalable and able to grow the model size and process enormous volumes of datasets, which operate well on parallel and distributed infrastructures. CNN is widely used in many forecasting applications which need timely results [12,13]. Distributed deep learning algorithms are well explained in [14], which discusses the infrastructure used for distributed deep learning systems and the problems and ways for data-parallel training for parallel DL (Deep Learning) training methods. Extra scheduler difficulties are created to map the distributed DL system component. The examination of data management issues and techniques in distributed DL systems is covered adequately in [14]. In the data parallelism of deep learning, a copy of the model is distributed over many devices. Training data is distributed across all available instruments to execute synchronously or asynchronously. Moving from only one learning to shared memory systems of data parallelism works well with MapReduce, which makes it simple to schedule parallel activities onto numerous processors. Model parallelism and data parallelism are the two most used parallelization strategies that allow many computers to learn together a single model. Model parallelism divides the collection parameter values and distributes them to all processors. However, the reliance on a variety of neurons and imbalanced parameter sizes in deep models make model parallelism challenging to expand operations. In contrast, data parallelism, on the other hand, spreads the computational work by delivering distinct data samples to separate processors that share the same model parameters.

Synchronous distributed training is a typical approach to spreading neural network models' training phase with data parallelism. In synchronous training, a root aggregate node fans out requests to numerous leaf nodes that operate in parallel across different data slices and submit their findings to the root node to combine. The delay of the leaf nodes considerably influences the effectiveness of this design, and when growing the number of variables and data sets, it may drastically increase the training period.

In Stochastic gradient descent, given a dataset D and denotes the θ model parameter, the objective is to minimize the loss function l(x, y), where x indicates the sample input and y the output using Eq. (1).

$$L(\theta) = \frac{1}{D} \cdot \sum_{(x,y)\in D} l((x,\ y)\theta)) \tag{1}$$

In SGD, the loss function is updated by updating the θ using the gradient calculation method. The learning rate (γ) is used to prevent both underfitting and overfitting and is calculated using Eq. (2).

$$\theta^{t+1} = \theta^{t} - \gamma_t \cdot G^{t} \tag{2}$$

In distributed synchronous Stochastic Gradient, a master aggregator node splits the entire into groups and sends requests to worker computer nodes to independently calculate the gradient for each batch. The master aggregator node combines the gradients and gives them back to the worker nodes to update the model's variables once all machines have returned their results. The master node iterates through this procedure for a specified number of iterations or following a conversion requirement, as shown in Fig. 2. The Sync SGD technique is susceptible to stragglers and excessive latency since it is meant to wait till all workers deliver the loss function. Mitigation of stragglers is crucial in maintaining the model performance [15].

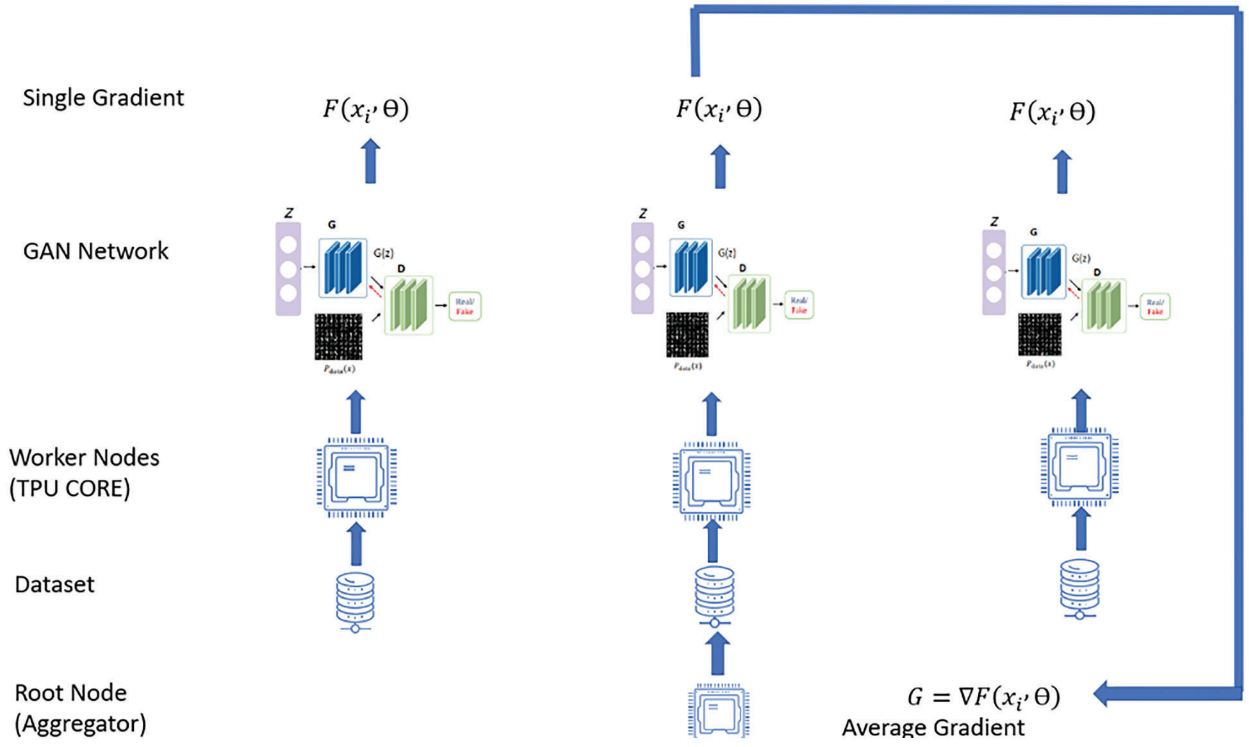

Figure 2: Synchronous SGD model for GAN

4 Method

Google Colab notebooks are used as coding platforms for the TPU since the TPU can currently only be utilized inside a Colab notebook. The MNIST dataset of handwritten digits, which contains a training dataset of sixty thousand instances and a testing dataset of ten thousand instances, is used in this study. TensorFlow is constructed using Python 3 to produce distributed Deep Learning models, timeout periods, maximum VM lifespan, GPU kinds accessible, and other aspects. In this model, an Intel Xeon processor (2.3 GHz) with 12.6 GB random access memory serves as the central processor, while a TPU v2 with 12.6 GB RAM (Random Access Memory) serves as the TPU. In Tensor Flow, the data is loaded using the prefetch function in which the next element is fetched when the current one is processed. The XLA compiler is used for acceleration.

4.1 GAN

The central concept of the study is to train a generator network in an adversarial environment, in which a discriminator network evaluates the produced samples. In recent years, generative adversarial networks have emerged as the new semi-supervised learning method [16]. The generator network creates an instance, while the discriminator network determines if it is authentic or not. Original samples are also delivered to the discriminator. The structure of the objective functions is given in Eq. (3).

$$minimize\, G\; maximize D,\;\; Ex \sim pdata[logD(x)] + Ez \sim p(z)[log(1 - D(G(z)))] \tag{3}$$

where: input data is represented by x∼pdata, z denotes random noise supplied as input, and G(z) is the generator-created images. The important fact to notice will be that the generator does not directly access essential information; instead, it merely communicates with the discriminator. As a result, through training, the generator gains knowledge from its mistakes and enhances its ability to provide real data.

The networks compete against one another in a zero-sum game, so their objectives are adversarial. When the generator is adequately trained, it can produce fake data that closely resembles actual data and trick the discriminator into accepting the created data as genuine.

In GAN [17] the discriminator is initially updated using the SGD in the first k steps during training. During this phase, the parameters and weights of the generator network remain fixed. Then when the noise is supplied, the generator starts producing the fake image. If n, noise elements are supplied, n fake images are generated {G(zi),…, G(zn)}. After that discriminator is fed with both real and fake images as input, and the max function is done using Eq. (4).

$$max_D V(D) = \frac{1}{n}\sum\nolimits_{i=1}^{n}\left[logD\left(x^i\right) + log\left(1 - D\left(G\left(z^i\right)\right)\right)\right] \tag{4}$$

Again, the generator is trained using random noise and generates fake images, and the generator network is updated to minimize the generator loss using Eq. (5).

$$min_G V(G) = \frac{1}{n}\sum\nolimits_{i=1}^{n}\left[log\left(1 - D\left(G\left(z^i\right)\right)\right)\right] \tag{5}$$

Deep convolutional generative adversarial nets [18] are a family of CNNs with particular architectural limitations and are a good option for unsupervised learning. Stride convolutions are used instead of spatial pooling functions in a convolutional network, allowing the network to train with its horizontal down-sampling. Fig. 3 illustrates how DCGAN deletes ultimately linked layers above the convolutional feature. A workable compromise was directly connecting the discriminator's input and output to the highest fully convolutional. Simple matrix multiplication makes up the first layer of the GAN, which uses a random noise distribution as input. The last convolution layer is reduced before being passed to the discriminator's single sigmoid production. By normalizing the input to each unit such that its mean and variance are both zero, batch normalization stabilizes learning. This helps resolve training problems brought on by poor initialization and enhances gradient mobility in deeper models. Since it stopped the generator from collapsing all samples into a single point, a typical failure situation was observed in GANs. However, model instability was seen when the batch norm was applied to all layers. To avoid this, the batch norm procedure was left out of the generator output and discriminator input layers. When creating dependable Deep Convolutional GANs, Replace In the discriminator, stride convolutions are used in place of the pooling layers. Batch normalization for the discriminator and generator reduces the number of ultimately connected hidden layers for structures that are deeper in depth. Use Leaky ReLU activation for all discriminator layers and ReLU activation in the generator. The GAN design and model parameters are given in Table 1 and are executed in Collab in distributed TensorFlow platform for 30 epochs using SGD with batch sizes 64 to 512.

4.2 Distributed TensorFlow

Dataflow graphs are used by TensorFlow to describe computation and functions that change that state. The vertices of a dataflow graph are translated across a cluster of many computers and various processing units inside a single system. The entire data set is often distributed among several nodes in distributed architectures; however, the learned parameters were ultimately merged at a central node. However, the central node for any centralized approach must have ample storage and processing power. Furthermore, central node failure may occur with centralized solutions. Therefore, only nodes close to one another are allowed to interact while all nodes' processing and data are dispersed. These are two standard methods for parallelizing training data using TensorFlow [19]. In sync training, gradients are accumulated at each stage as all workers train concurrently across various data input slices. Async training involves asynchronously updating variables while having all workers train on input data. Most frequently, all reduce and async architecture with parameter servers are used to offer sync training. The model is duplicated throughout all eight cores and may be maintained in sync with all reduce techniques. Eight TPU cores are used to create the Distributed GAN using synchronous distributed stochastic gradient descent, often referred to as distributed synchronous SGD. The gradients of each node are combined to enhance their models. The synchronous phase ensures that all the nodes' models are consistent throughout training. The reduce operation on the gradients between all nodes and the individual updating of the parameters on each node can be used to complete the aggregation phase.

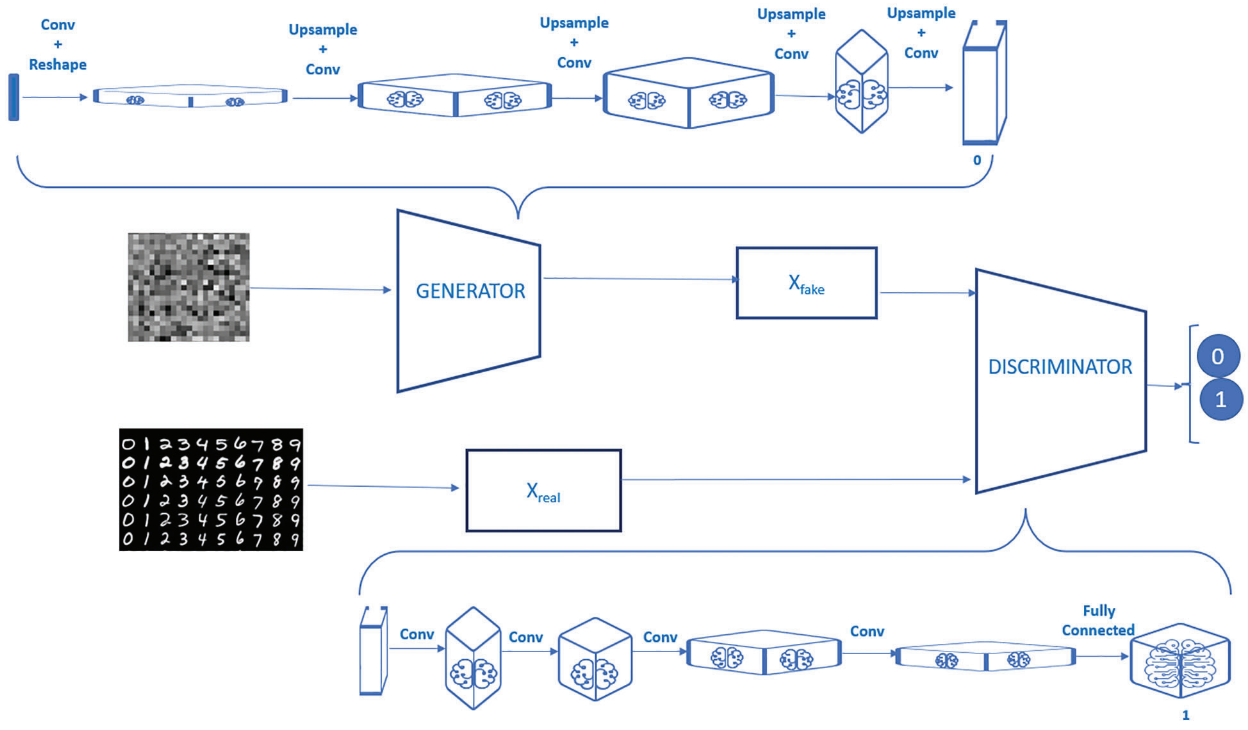

Figure 3: Proposed generative adversarial network

Table 1: Network topology

Generator network		Discriminator network	
Layers	Size	Layers	Size
InputLayer	(-, 100)	Input layer	(-, 28, 28, 1)
Dense	(-, 12544)	Conv2D	(-, 14, 14, 64)
Batch normalization	(-, 12544)	Leaky ReLU	(-, 14, 14, 64)
LeakyReLU	(-, 12544)	Dropout	(-, 14, 14, 64)
Reshape	(-, 7, 7, 256)	Conv2D	(-, 7, 7, 128)
conv2d_transpose	(-, 7, 7, 128)	Leaky ReLU	(-, 7, 7, 128)
conv2d_transpose	(-, 7, 7, 128)	Dropout	(-, 7, 7, 128)
LeakyReLU	(-, 7, 7, 128)	Flatten	(-, 6272)
conv2d_transpose	(-, 14, 14, 64)	Dense	(-, 6272)

(Continued)

Table 1 (continued)

Generator network		Discriminator network	
Layers	Size	Layers	Size
Batch normalization	(-, 14, 14, 64)	-	-
LeakyReLU	(-, 14, 14, 64)	-	-
conv2d_transpose	(-, 28, 28, 1)	-	-
Optimizer	Adam(1e-4)	Optimizer	Adam(1e-4)
Loss	Binary cross entropy	Loss	Binary cross entropy

4.3 Mixed Precision

Mixed precision refers to a strategy in which 16 bit and 32 bit precision floating point values are utilized to represent variables to decrease memory requirements and accelerate training. It is based on the notion that modern hardware accelerators, such as GPUs and TPUs, can do calculations more quickly in 16 bit, as shown in Fig. 4. Utilizing bfloat16 inside systolic arrays expedites multiplications computations on TPUs.

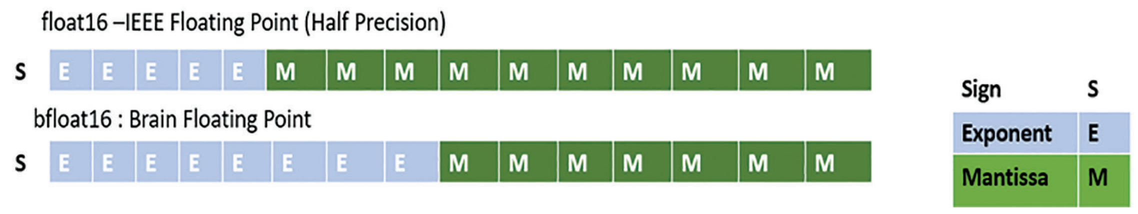
float16 –IEEE Floating Point (Half Precision)
S E E E E E M M M M M M M M M M
bfloat16 : Brain Floating Point
S E E E E E E E E M M M M M M M
Sign S
Exponent E
Mantissa M

Figure 4: Floating point formats-bfloat16 and float16

In the Multiplications Unit (MXU) of Cloud TPU v2 and Cloud TPU v3, bfloat16, a 128 × 128 systolic array, is the primary data type. Two MXUs are present in the TPUv3 chip, and numerous TPU chips are current per the Cloud TPU system. These MXUs together provide the bulk of the system's FLOPS. (A TPU may accomplish FP32 multiplications by iterating the MXU many times.) Multiplications are done in bfloat16 notation inside the MXU, whereas accumulations are handled with FP32 precision.

In backpropagation, in particular, gradients may get so tiny that an underflow event is initiated. When using loss scaling, the loss is scaled by a significant amount. This drastically reduces the possibility of an underflow occurs. After the gradients have been calculated, the result is reduced by the same scaling factor to yield the true, unscaled gradients. Cloud TPUs can train models with more profound, broader, or more significant inputs when bfloat16 is used more extensively. Moreover, since bigger models often result in more precision, this enhances the final quality of the items that rely on them.

5 Results

The training time and learning rate are used to determine the effectiveness and precision of both models. In addition, the impact of batch size on the training time and learning and optimization curve is analyzed.

5.1 Effect of the Different Batch Sizes on Performance and Training Time

The main advantage of extensive batch-size training is that it cuts overall training time. SGD can distribute all workloads for each iteration over several processors by using large packet sizes. Including a processor in a method can load additional data into each phase, increasing the batch size for synchronous SGD. The number of SGD iterations should theoretically decrease linearly by fixing the overall set of data access and increasing the batch size directly proportional to the number of processors. The total time is reduced linearly due to the constant time expense of each repetition. For every iteration, we must increase the number of processors in the algorithm and load more data, corresponding to increasing the batch size in synchronous SGD. Assume that the total number of data accesses is fixed and that the batch size increases directly proportional to the number of processors. In that instance, the overall time would drop linearly with the number of processors. At the same time, the frequency of SGD iterations would decrease, and the time expense of each iteration would remain constant.

Large batch sizes resulted in reduced training time and increased machine usage. However, no method allows us to employ unlimited-sized batches efficiently. For considerable batch training, it is necessary to maintain an accuracy rate with smaller batches while maintaining the same epochs. Furthermore, utilizing a big batch size may decrease communications volumes with fewer iterations, resulting in more incredible scaling performance than small batch size. Communication is sometimes the most significant impediment to the effective scalability of the program over several TPUs. The different batch sizes implemented using the novel GAN network in distributed TPU showed an increase in training time as the batch sizes increased, as shown in Figs. 5 and 6.

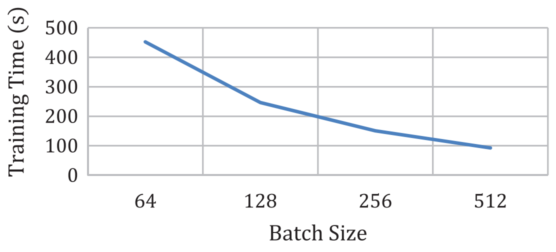

Figure 5: Training time *vs.* batch size

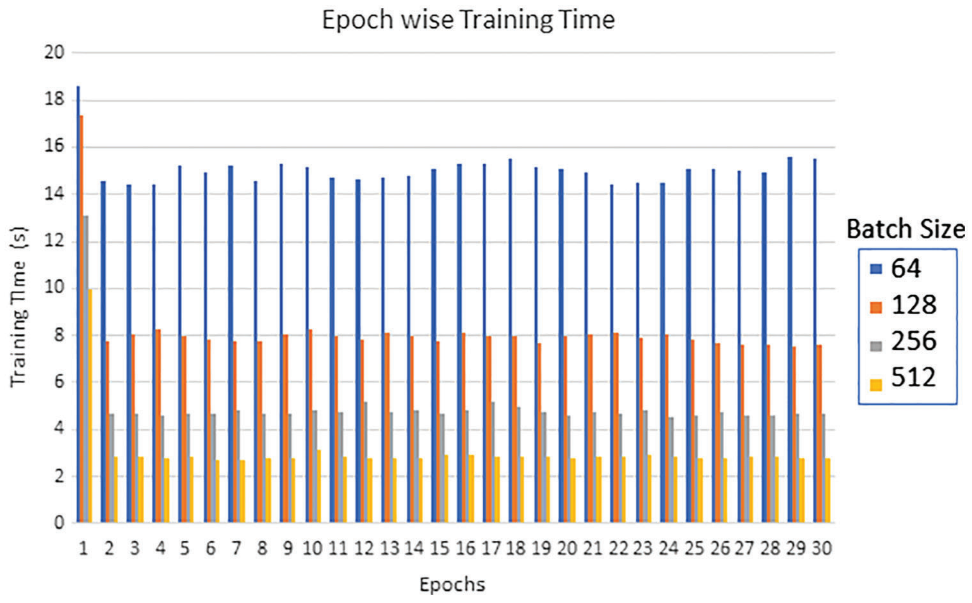

Figure 6: Epoch-wise training time

5.2 Effect of the Different Batch Sizes on Loss Curve, Learning Curve, and Optimization

Discriminator loss quantifies how successfully the discriminator can identify between actual and fraudulent pictures. It evaluates the discriminator's recommendations on genuine photos to an array of 1 s and false images to an array of 0. Generative Loss: The loss of the generator measures how well it could fool the discriminator. The discriminator will intuitively identify the bogus pictures as genuine if the generator operates correctly (or 1). The generator loss and discriminator loss for the varying batch sizes are shown in Fig. 7.

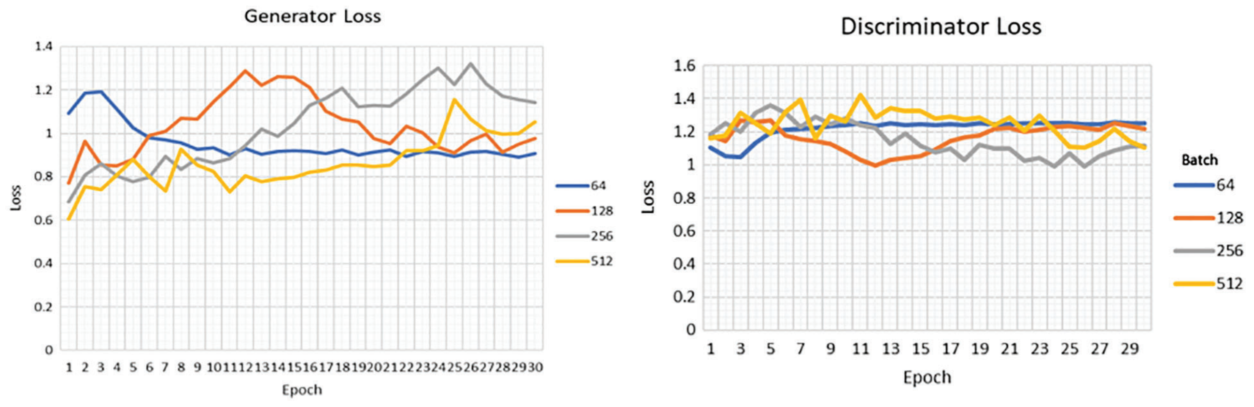

Figure 7: Generator and discriminator loss curve

Histogram loss curves of the generator and discriminator are shown in Fig. 8. The histogram loss curve's loss concertation on each epoch for all batch sizes is demonstrated. The batch size 64 and 128 represent the base histogram distribution for which the trend lines are shown using batches 128 and 256.

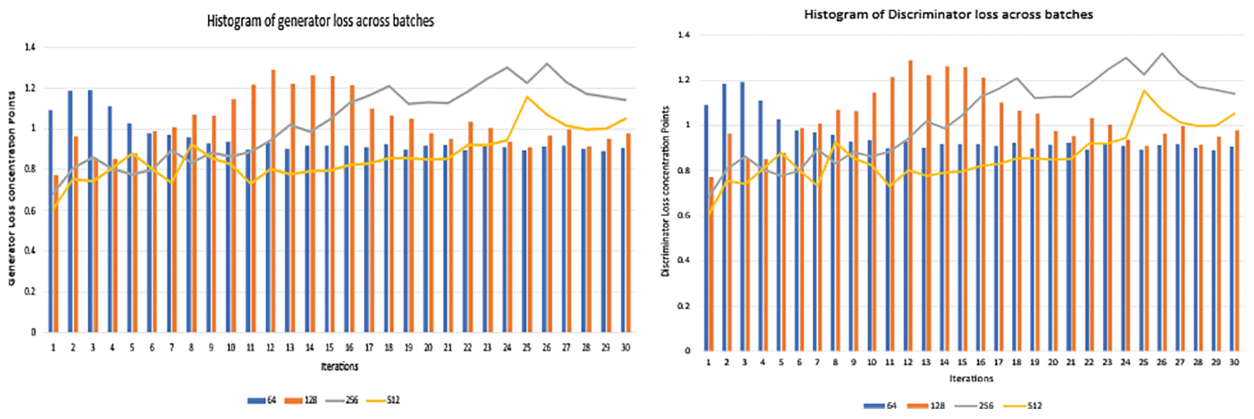

Figure 8: Histogram generator and discriminator loss curve

The discriminator offers essential data that can be used to adjust the generator and its parameters. The discriminator typically converges first before the generator has had a reasonable training period. This issue typically causes GANs to experience mode breakdown and failure in convergence [20]. This circumstance may result in poor learning. The learning rate typically regulates how quickly parameters update throughout the training phase. The update pace will be significantly slower whenever the learning rate is so low.

On the other hand, oscillations in the process will occur, leading the variables to stay close to the ideal value. The learning rates for each iteration is shown for all batch size in Fig. 9. It can infer that there is a dip in the learning rate at iteration 7 and iteration 24 for batch size 512, and finally, there is a sudden rise in the learning rate, and the peak is attained. The LR represents the learning rate for each batch size.

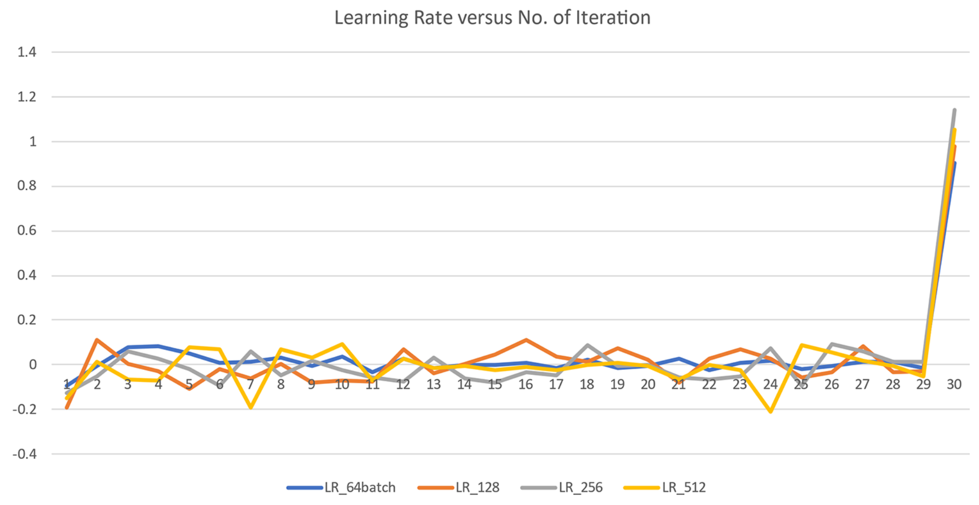

Figure 9: Learning rate *vs.* iteration curve

The loss Optimization curve for batch sizes 64, 128, and 256, 512 is shown in Fig. 10. The loss is optimized in batch sizes 256 and 512—so better optimization as batch size increases. The loss value does not direct the generator towards identifying the local minimum if the discriminator's loss value is near zero. This impacts both the accuracy and the range of the result obtained. The abrupt rise in the learning rate at the start of each cycle encourages the model to investigate more domains and broadens the range of outcomes.

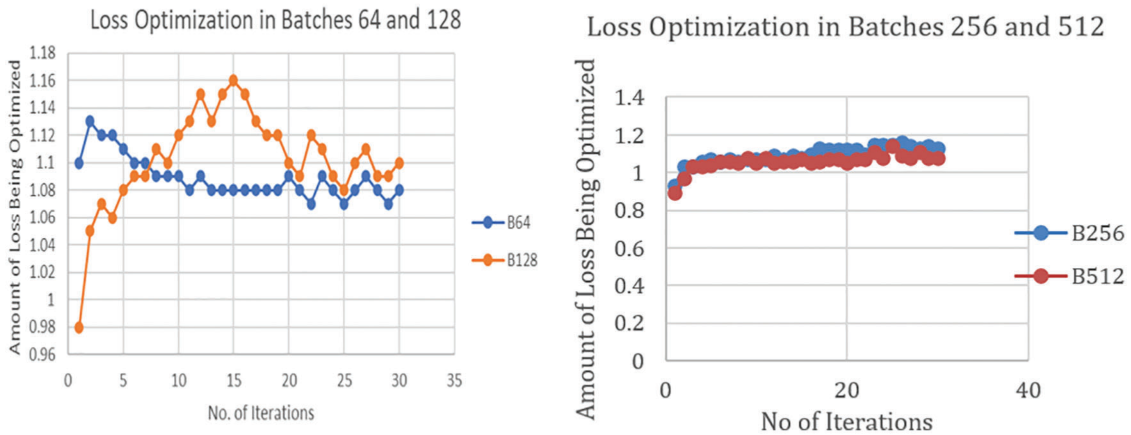

Figure 10: Loss optimization

The SGD algorithm is used for optimization, the generator network tries to minimize the loss curve, and the discriminator network maximizes the loss. From Fig. 11, the generator loss is minimized, and the radius is minimum for batch size 512, clearly showing it has the minimum loss. As the iteration increases, the loss is reduced; further, it is reduced with an increase in batch size. The loss is less for smaller batch sizes in the discriminator and increases with batch size. This show that the discriminator network is trained efficiently using TPU in large batch size.

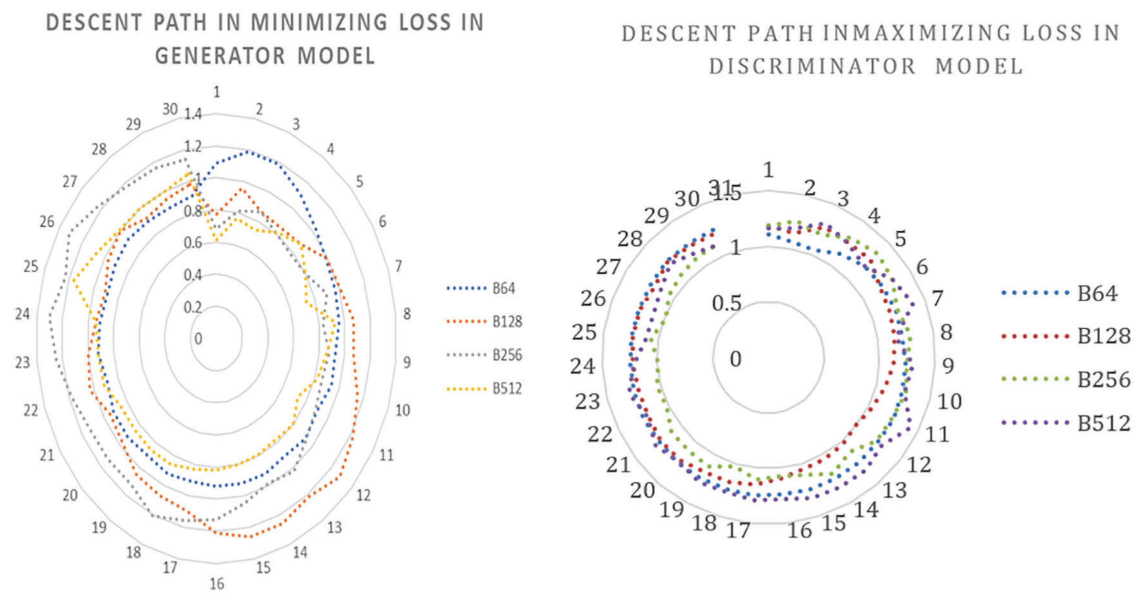

Figure 11: Descent path in minimizing loss in generator and maximizing loss in discriminator

5.3 With XLA Compiler and Bfloat16

TPU software stack supports automated format conversion: data is transformed effortlessly between floating point 32 and bfloat16 using the XLA compiler, which can enhance model performance by extending the usage of bfloat16. XLA converts the TensorFlow graph into a series of compute kernels tailored to the specified model. TensorFlow graph performs three kernels in the absence of XLA: including one addition, one multiplication, and another for reduction. XLA combines these three kernels into a single kernel, eliminating the need to store intermediate data during calculation. Both memory usage and training time are reduced using XLA. Mixed precision training provides a considerable increase in computing performance by carrying out these activities in the half-precision style. Using numerical forms with less accuracy than 32-bit floating-point has various advantages. First, they demand less memory, allowing more extensive neural networks to be trained and deployed. The second benefit is that they use less memory bandwidth, accelerating data transfer activities. Thirdly, arithmetic operations execute much quicker with lower precision, particularly on the tensor core for this precision. It does this by detecting the phases that need complete accuracy and employing a 32-bit floating point solely for those steps, while a 16-bit floating point is used everywhere else. Utilizes Tensor Cores to accelerate math-intensive processes such as linear and convolution layers. Memory-limited operations are accelerated by accessing half as many bytes as with single-precision. Reduces the memory needs for training models, allowing for bigger models or mini-batch sizes. After enabling XLA, the models run around 1.15 times quicker than before, as shown in Fig. 12.

5.4 Comparison with Other State-of-the-Art Models

The proposed model is compared with the current state-of-the-art models-execution of GAN in CPU, GPU, and TPU (without parallel data distribution). In the proposed model, the acceleration is achieved using the distributed data-parallel implementation of GAN using multi-core TPU. The comparison is shown in Fig. 13. The proposed model has the best training time acceleration compared to the existing

HPC-based acceleration platforms CPU, GPU, and TPU. The GPU is faster compared to the CPU due to the high storage and processing capacity of the GPU. On the other hand, the TPU (serial execution) is slower than GPU because of the network structure and the layers. Few layers in the novel GAN model are executed in CPU rather than in TPU, and this causes the swapping between TPU and CPU, which causes the additional delay. In the proposed model, the execution is split among multi-core TPU, which effectively utilizes the TPU cores and makes the model faster. The proposed model is 56.20% faster than a CPU, 19.7% faster than GPU, and 24.6% faster than serial TPU execution for batch size 32.

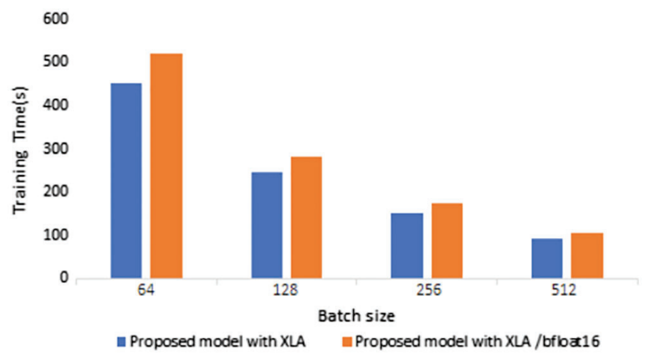

Figure 12: Performance analysis of XLA compiler

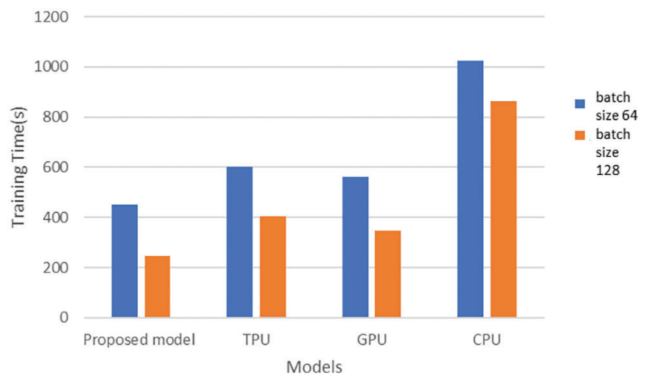

Figure 13: Performance analysis of proposed model

The significant limitations of the existing model are the costly TPUs, lack of multi-TPU nodes, and the switching between the TPU and CPU for the different layer's execution in the generator and discriminator networks. The dense layer, convolutional layers in the generator, and discriminator are executed in TPU. The other layers reshape, pooling, dropout, up sampling, and flatten performed in CPU in both discriminator and generator. This causes switching between CPU and TPU when the GAN model is trained in TPU. This weakness can be removed using muti TPU in Cloud and designing the GAN by reducing the number of switching operations between CPU and GPU.

6 Conclusion

Data is big enough to be centrally stored in a single location for processing. Moreover, in applications like the medical sector, the data itself is distributed. In such cases, the GAN must be employed in the distributed domain to obtain the results. The distributed GAN is further accelerated using TPU in this paper. While distributing a model input is restricted, TPUs conduct computations very quickly. To prevent the TPU from becoming idle, it is essential to guarantee that a continual stream of data is put into it. This depends on how your dataset is loaded and preprocessed. Our research shows that bfloat16 is preferable to float16 since most deep-learning algorithms do not need a high degree of precision to achieve the desired operational efficiency. The hyperparameter is mainly the weight factor and is stored in a 32-bit single-precision format to prevent loss of precision or even divergence. The SGD algorithm is used for optimization, the generator network tries to minimize the loss curve, and the discriminator network maximizes the loss. The paper shows the accelerated learning curve and efficient maximization and minimization of the loss function in SGD. The training time was reduced by 79% by varying the batch size from 64 to 512 in multi-core TPU. The proposed model is 56.20% faster than a CPU, 19.7% faster than GPU, and 24.6% faster than serial TPU execution for batch size 32. After enabling XLA, the models run around 1.15 times quicker than before. Theoretically, splitting the calculation among T worker computers should enhance performance by xT. However, in practice, the performance enhancement is hardly ever xT. Recent studies have identified Stragglers and High Latency as the primary underlying cause of productivity loss. The Sync SGD technique is susceptible to stragglers and excessive latency since it is meant to wait till all workers deliver the loss function. In future work, the straggler impact on parallel data distribution must be mitigated to obtain better accuracy and efficient communication.

Funding Statement: The authors received no specific funding for this study.

Conflicts of Interest: The authors declare they have no conflicts of interest to report regarding the present study.

References

[1] Q. Hoang, T. D. Nguyen, T. Le and D. Phung, “Multi-generator generative adversarial nets,” arXiv, Oct. 27, 2017. [Online]. Available: https://arxiv.org/abs/1708.02556.

[2] I. Durugkar, I. Gemp and S. Mahadevan, “Generative multi-adversarial networks,” arXiv, Mar. 02, 2017. [Online]. Available: http://arxiv.org/abs/1611.01673.

[3] R. D’Agostino and T. Schmidt, “EMOTIONET: A multi-convolutional neural network hierarchical approach to facial and emotional classification using TPUs,” in *Proc. of IEEE Int. Conf. on Engineering, Technology and Innovation (ICE/ITMC)*, Cardiff, UK, pp. 1–4, 2020.

[4] T. Lu, T. Marin, Y. Zhuo, Y. -F. Chen and C. Ma, “Accelerating MRI reconstruction on TPUs,” arXiv, Jun. 24, 2020. [Online]. Available: http://arxiv.org/abs/2006.14080.

[5] M. Robin, J. John and A. Ravikumar, “Breast tumor segmentation using u-net,” in *Proc. of 5th Int. Conf. on Computing Methodologies and Communication (ICCMC)*, Erode, India, pp. 1164–1167, 2021.

[6] J. John, A. Ravikumar and B. Abraham, “Prostate cancer prediction from multiple pretrained computer vision model,” *Health and Technology*, vol. 11, no. 5, pp. 1003–1011, 2021.

[7] 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, pp. e909, 2022.

[8] M. Horowitz, “1.1 computing’s energy problem (and what we can do about it),” in *Proc. of IEEE Int. Solid-State Circuits Conf. Digest of Technical Papers (ISSCC)*, San Francisco, CA, USA, pp. 10–14, 2014.

[9] M. Song, J. Zhang, H. Chen and T. Li, “Towards efficient microarchitectural design for accelerating unsupervised gan-based deep learning,” in *Proc. of IEEE Int. Symp. on High Performance Computer Architecture (HPCA)*, Vienna, Austria, pp. 66–77, 2018.

[10] H. Mao, M. Song, T. Li, Y. Dai and J. Shu, “Lergan: A zero-free, low data movement and PIM-based GAN architecture,” in *Proc. of 51st Annual IEEE/ACM Int. Symp. on Microarchitecture (MICRO)*, Fukuoka, Japan, pp. 669–681, 2018.

[11] H. Chen, T. Jia and J. Tang, “A research on generative adversarial network algorithm based on GPU parallel acceleration,” in *Proc. of Int. Conf. on Image and Video Processing, and Artificial Intelligence*, Shanghai, China, vol. 11321, pp. 397–404, 2019.

[12] M. Abdel-Basset, H. Hawash, K. Sallam, S. S. Askar and M. Abouhawwash, “STLF-Net: Two-stream deep network for short-term load forecasting in residential buildings,” *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 7, pp. 4296–4311, 2022.

[13] H. Balan, A. F. Alrasheedi, S. S. Askar and M. Abouhawwash, “An intelligent human age and gender forecasting framework using deep learning algorithms,” *Applied Artificial Intelligence*, vol. 36, no. 1, pp. 2073724, 2022.

[14] R. Mayer and H. -A. Jacobsen, “Scalable deep learning on distributed infrastructures: Challenges, techniques and tools,” arXiv, Sep. 25, 2019. [Online]. Available: http://arxiv.org/abs/1903.11314.

[15] A. Ravikumar, “Non-relational multi-level caching for mitigation of staleness & stragglers in distributed deep learning,” in *Proc. of the 22nd Int. Middleware Conf.: Doctoral Symp.*, Québec city Canada, pp. 15–16, 2021.

[16] A. R. Sajun and I. Zualkernan, “Survey on implementations of generative adversarial networks for semi-supervised learning,” *Applied Sciences*, vol. 12, no. 3, Art. no. 3, pp. 1–21, 2022.

[17] M. Abedi, L. Hempel, S. Sadeghi and T. Kirsten, “GAN-Based approaches for generating structured data in the medical domain,” *Applied Sciences*, vol. 12, no. 14, Art. no. 14, pp. 1–16, 2022.

[18] F. Gao, Y. Yang, J. Wang, J. Sun, E. Yang *et al.,* “A deep convolutional generative adversarial networks (DCGANs)-based semi-supervised method for object recognition in synthetic aperture radar (SAR) images,” *Remote Sensing*, vol. 10, no. 6, Art. no. 6, pp. 1–21, 2018.

[19] J. K. Ranbirsingh, H. Kimm and H. Kimm, “Distributed neural networks using tensorflow over multicore and many-core systems,” in *Proc. of IEEE 13th Int. Symp. on Embedded Multicore/Many-Core Systems-on-Chip (MCSoC)*, Singapore, pp. 101–107, 2019.

[20] K. Li and D. -K. Kang, “Enhanced generative adversarial networks with restart learning rate in discriminator,” *Applied Sciences*, vol. 12, no. 3, Art. no. 3, pp. 1–16, 2022.