

ParaGAN: A Scalable Distributed Training Framework for Generative Adversarial Networks

Ziji Shi

National University of Singapore
Singapore
zijishi@comp.nus.edu.sg

Jialin Li

National University of Singapore
Singapore
lijl@comp.nus.edu.sg

Yang You

National University of Singapore
Singapore
youy@comp.nus.edu.sg

ABSTRACT

Recent advances in Generative Artificial Intelligence have fueled numerous applications, particularly those involving Generative Adversarial Networks (GANs), which are essential for synthesizing realistic photos and videos. However, efficiently training GANs remains a critical challenge due to their computationally intensive and numerically unstable nature. Existing methods often require days or even weeks for training, posing significant resource and time constraints.

In this work, we introduce ParaGAN, a scalable distributed GAN training framework that leverages asynchronous training and an asymmetric optimization policy to accelerate GAN training. ParaGAN employs a congestion-aware data pipeline and hardware-aware layout transformation to enhance accelerator utilization, resulting in over 30% improvements in throughput. With ParaGAN, we reduce the training time of BigGAN from 15 days to 14 hours while achieving 91% scaling efficiency. Additionally, ParaGAN enables unprecedented high-resolution image generation using BigGAN.

CCS CONCEPTS

• **Computing methodologies** → **Computer vision**; • **Networks** → **Cloud computing**; • **Computer systems organization**;

KEYWORDS

Generative Adversarial Network, Distributed Training

ACM Reference Format:

Ziji Shi, Jialin Li, and Yang You. 2024. ParaGAN: A Scalable Distributed Training Framework for Generative Adversarial Networks. In *ACM Symposium on Cloud Computing (SoCC '24)*, November 20–22, 2024, Redmond, WA, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3698038.3698563>

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SoCC '24, November 20–22, 2024, Redmond, WA, USA

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1286-9/24/11

<https://doi.org/10.1145/3698038.3698563>

Table 1: A summary of reported training time and model size for GANs trained on ImageNet dataset.

GANs	Hardware	Time	# Params.
SNGAN [27]	8 V100 GPUs	3d 13.6h	81.44M
ProgressiveGAN [16]	8 V100 GPUs	4d	43.2M
ContraGAN [15]	8 V100 GPUs	5d 3.5h	160.78M
SAGAN [38]	8 V100 GPUs	10d 18.7h	81.47M
BigGAN [3]	8 V100 GPUs	15 d	158.42M

1 INTRODUCTION

The last decade has witnessed the success of Generative Adversarial Networks [8], which has a wide range of applications including image super-resolution [19], image translation [12, 40], photo inpainting [7, 37]. However, training GAN at scale remains challenging because of the computational demands and optimization difficulties.

Unlike other conventional neural networks where optimization is straightforward by taking gradient descents, there are two sub-networks to optimize in GAN, namely generator and discriminator. The generator samples from the noise and produces a fake sample as close to the real sample as possible, and the discriminator evaluates the generated sample. The generator aims to fool the discriminator, and the discriminator will try to identify the fake images from the real ones. Since the two components are optimized for two contradicting goals, it has been observed that GANs are difficult to converge. Therefore, to speed up the GAN training at large scale, we need a framework optimized on both system and numerical perspectives.

Due to the difficulty of optimizing GAN, many state-of-the-art GAN models take days or even weeks to train. For instance, BigGAN [3] took 15 days on 8× V100 GPUs to converge. Table 1 summarizes the reported training time of some of the state-of-the-art GAN models. This makes it difficult to quickly reproduce, evaluate, and iterate GAN experiments. Also, current GAN frameworks usually only support training on a small number of nodes [5, 15, 20].

We argue that training speed is an important yet often ignored factor in the current GAN training landscape, and we propose to accelerate it with distributed training. However, distributed GAN training has several challenges. First of all,

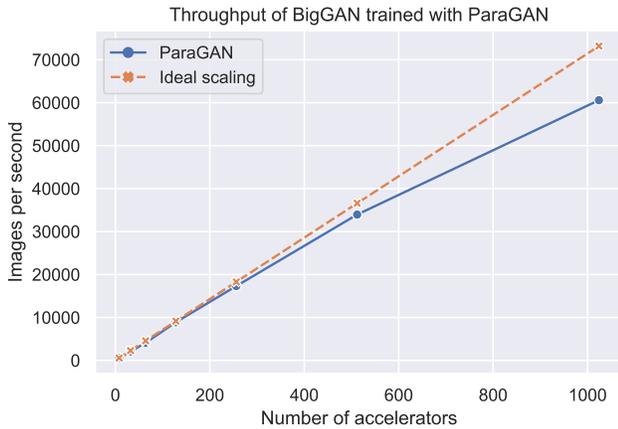


Figure 1: ParaGAN scales to 1024 TPU accelerators at 91% scaling efficiency.

most data centers have storage nodes and compute nodes separated for elasticity, but network congestion can happen from time to time, which prolongs the latency between nodes and affects training throughput. Secondly, there are usually different types of accelerators in the data center, but each of them has its architectural design and preferred data layout. If ignored, it could lead to under-utilization of accelerators. Last but not least, training GAN at scale may cause a convergence problem, in which the GAN loss does not converge to a stable equilibrium. Therefore, this framework has to consider both system and numerical perspectives.

In this work, we present **ParaGAN**, the first distributed training framework that supports large-scale distributed training for high-resolution GAN. We identify the performance bottlenecks when training at scale and optimize them for efficiency. ParaGAN has a simple interface for building new GAN architecture, and it supports CPU, GPU, and TPU.

The main contributions of ParaGAN include:

- We design and implement the first scalable distributed training framework for GAN with optimizations on both system and numerical perspectives. With ParaGAN, the training time of BigGAN can be shortened from 15 days to 14 hours with 1024 TPU accelerators at 91% scaling efficiency, as shown in Fig. 1. ParaGAN also enables direct photo-realistic image generation at unprecedented 1024×1024 resolution, which is 4× higher than the original BigGAN model.
- From the system optimization perspective, we use congestion-aware data pipeline and hardware-aware layout transformation to improve the accelerator utilization, and low-precision training to alleviate the memory stress. They contribute to 30-40% throughput improvements over baseline.

- From the numerical optimization perspective, we show that the generator and discriminator can be optimized independently, and present an asynchronous update scheme together with an asymmetric optimization policy.

The paper is organized in the following manner: we discuss the motivation and requirement for large-scale GAN training in Section 2; in Section 3, we will explain our design for ParaGAN, and how those architectural considerations can address the requirements; in Section 4 and Section 5, we will cover the system-level and numerical-level optimizations for scalable training in ParaGAN respectively; in Section 6, we present our scalability evaluation of ParaGAN and study the effect of different optimization techniques; a brief review of related work on GAN and large-scale distributed training is presented in Section 7; we conclude this study of GAN in Section 8.

2 MOTIVATION AND REQUIREMENTS

We begin by considering the basic components of GAN to discover the key requirements for GAN training. As shown in Fig. 2, a GAN consists of a generator and a discriminator. The generator generates fake data samples, while the discriminator distinguishes between the generated samples and real samples as accurately as possible. The learning problem of GANs is a minimax optimization problem. The goal of the optimization is to reach an equilibrium for a two-player problem:

$$\min_G \max_D \mathbb{E}_{x \sim q_{data}(x)} [\log D(x)] + \mathbb{E}_{z \sim p(z)} [\log (1 - D(G(z)))]$$

where $z \in \mathbb{R}^{d_z}$ is a latent variable drawn from distribution $p(z)$. The discriminator seeks to maximize the sum of the log probability of correctly predicting the real and fake samples, while the generator tries to minimize it instead. The convergence of GAN is defined in the form of Nash Equilibrium: one network does not change its loss regardless of what the other network does.

Since the two networks have contradicting goals, the training process of GAN is a zero-sum game and can be very unstable. Recent works show that: i) GAN may converge to points that are not local minimax using gradient descent, in particular for a non-convex game which is common [6, 13], and ii) gradient descent on GAN exhibits strong rotation around fixed points, which requires using very small learning rates [1, 25]. Also, GANs training is sensitive to the hyperparameters and initialization [23]. Therefore, it is observed that GANs are difficult to optimize, and this is also the reason why it takes a long time to train them.

There are some existing GAN libraries [5, 15, 20, 23] for training state-of-the-art GANs. They provide standardized

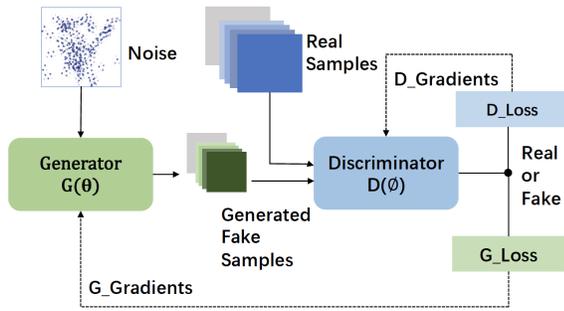


Figure 2: Typical GAN architecture.

building blocks like network backbone and evaluation metrics, making it easy to build new models. However, they focus less on the system performance, and training GAN can still take days if not weeks. If the training process can be massively paralleled, the iteration cycles can be greatly shortened. Motivated by the challenges with the status quo, we outline the following requirements for a desirable distributed GAN training framework:

Performance and Scalability. Performance is of critical importance to GAN training. The experiments from [23] in total took 60K GPU hours to train, making it hard to reproduce and improve over. Completing such a workload is almost infeasible without using distributed training. Also, since most GAN models are compute-intensive and can be fitted into the memory of one worker, data parallelism can be used as the distributed training strategy, by placing identical copies of the model on each accelerator worker. The framework should be scalable as the number of accelerators scales.

Numerical Stability. The convergence of GAN can be very volatile. The framework should have means to stabilize the training process. Also, the framework should support large batch training as the number of accelerator scales.

In ParaGAN, our approach is to co-design the solution on the system and optimization levels. At the system level, we pinpoint performance bottlenecks, primarily arising from network congestion and sub-optimal accelerator utilization. To alleviate these issues, we introduce a congestion-aware data pipeline and implement a hardware-aware layout transformation. On the optimization front, we argue that the training process of the discriminator and generator can be treated independently, and we find that decoupling the training of the generator and discriminator offers advantages. To capitalize on this insight, ParaGAN employs an asynchronous update scheme along with an asymmetric optimization policy.

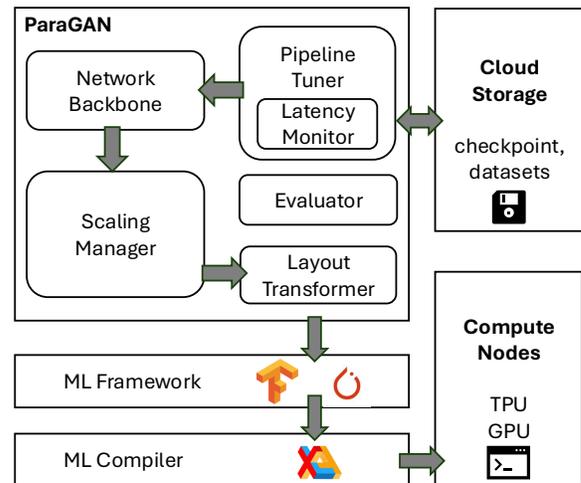


Figure 3: Overview of ParaGAN.

3 DESIGN AND ARCHITECTURE

In this section, we will give an overview and discuss the design decisions of ParaGAN. We recognize that the scalability is usually limited by the latency between nodes. Furthermore, when scaling up the GAN training, the numerical instability problem happens more often. We therefore divide the discussions into two folds and present our co-designed approach for system throughput and training stability as it scales.

3.1 Programming Model

The architecture of ParaGAN is presented in Fig. 3. ParaGAN is implemented on top of TensorFlow, but it can be ported to support other DL frameworks. ParaGAN provides high-level APIs for GAN which include scaling manager, evaluation metrics, and common network backbones. Users of ParaGAN can import from ParaGAN or define their components. ParaGAN then performs layout transformation on tensors and invokes TensorFlow, which converts the model definition to a computational graph. An optional XLA [31] pass can be performed followed by that. After that, the training loop starts on the compute nodes that host accelerators like GPUs and TPUs.

We introduce three concepts in ParaGAN:

3.1.1 Scaling Manager. The scaling manager is in charge of hyper-parameters that need to be tuned when scaling, including learning rate, optimizer, and local batch size. Users can use the best hyper-parameters from a single worker as a starting point, and ParaGAN will scale them based on the number of workers and learning rate schedules. Users can also define their scaling manager.

Listing 1: Parallel GAN Training and Evaluation

```

import paragan as pg

class Generator:
    def model_fn(x, y):
        # generator model
        return output

class Discriminator:
    def model_fn(x, y):
        # discriminator model
        return output, out_logits

scaling_mgr = pg.ScalingManager(config=cfg,
                                batch_size=2048, num_workers=128)

g = Generator()
d = Discriminator()
model = pg.Estimator(g,d)

# training
for step in cfg.max_train_steps:
    scaling_mgr.train(gan)

# evaluation
scaling_mgr.eval(metric="fid")

```

3.1.2 Network Backbones. Users usually start by building upon existing GAN architectures. We also provide some popular GAN architectures as backbone, including but not limited to:

- BigGAN [3];
- Deep Convolutional GAN (DCGAN) [30];
- Spectral Norm GAN (SNGAN) [27]

3.1.3 Evaluation Metrics. Evaluation metrics can be implemented differently across papers, and this can cause inconsistency. We provide commonly used evaluation metrics including Frechet Inception Distance (FID) and Inception Score (IS).

3.2 Computation Model

Training on the cloud usually involves host machines (mostly CPU nodes), compute nodes and storage nodes. As depicted in Fig. 3, the host fetches input data from the storage node, builds the model, chooses the tensor layout for the target accelerator, and feeds to the compute nodes. After that, the accelerators execute forward and backward passes and then synchronize gradients among other accelerators. At the same time, the host prefetches and transforms the data from the storage node. Model checkpoints are saved to the storage node which can be an object store or mounted file system.

A host machine can be a standalone node, while most of the time it co-locates with the compute nodes in the same

chassis, and is connected to the accelerators via high-speed PCI-e bus or NVLink. Meanwhile, the host-to-host and host-to-storage connections usually go through Ethernet.

3.3 System Optimizations

To satisfy the scalability requirement, we design ParaGAN with optimizations on data pipeline, computation, and memory.

We optimize the data pipeline performance by using a congestion-aware data pipeline. For data centers, the compute and storage nodes are usually physically distributed and interconnected via Ethernet instead of high-speed InfiniBand. The network traffic between them may not always be stable since the infrastructure is shared with other tenants. This issue is further pronounced when the number of participating data parallel workers scales up because data parallelism is a synchronous training method that is sensitive to the latency of the slowest participant. Therefore, ParaGAN continuously monitors the data pipeline latency and implements a congestion-aware data pipeline tuner.

To achieve a higher accelerator utilization, ParaGAN performs hardware-aware layout transformation. A data center usually has multiple types of accelerators, and different accelerators have different micro-architectures and instructions, thereby having different preferred data layouts. For example, Nvidia A100 GPUs prefer half-precision data in multiples of 64, and single-precision data in multiples of 32, while previous generations prefer multiples of 8. For TPU, the preferred data layout should have a multiple of 128 on the lane dimension and 8 on the sublane dimension. If misconfigured, this will result in unnecessary padding which reduces accelerator utilization and increases memory consumption. We come up with the hardware-aware layout transformation to transform the data into an accelerator-friendly format to maximize accelerator utilization.

Memory usage can be reduced by mixed-precision training with Brain Float 16 (bf16) format. Theoretically, using purely bf16 can save half of the memory for activation. However, we observe through experiments that bf16 is not suitable for the layers that are sensitive to overflow/underflow. To be precise, we found that the generator and discriminator’s last layer are more sensitive to precision. As such, we apply full precision (fp32) on these layers.

3.4 Numerical Optimizations

Another key contribution of ParaGAN is using asymmetric training to stabilize GAN. As the number of workers scales, a larger batch size can accelerate training. However, we observe that the performance of large batch training for GAN is not as stable, and mode collapse happens frequently. Since mode collapse is a type of GAN failure raised due to a highly

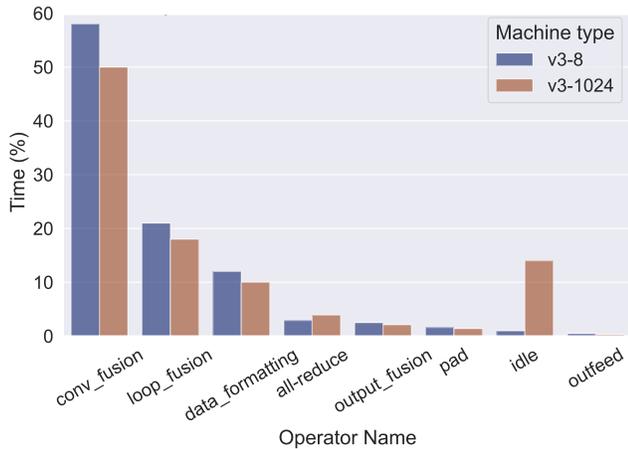


Figure 4: Operator usage profile when training at scale.

coupled optimization process, to solve this problem, ParaGAN comes with an asymmetric optimization policy and asynchronous update scheme to decouple the process.

4 IMPLEMENTATION

To begin with, we profile BigGAN training on native TensorFlow [23] and present the result in Fig. 4. As we increase the cluster size from 8 to 1024 TPU workers, idle time significantly increases due to increased communication, but convolution operation still makes up most of the time. It indicates that training GAN is a compute-bound workload. Therefore, we focus on improving the accelerator utilization in ParaGAN.

To achieve this goal, we use congestion-aware data pipelining to reduce data pipeline latency, hardware-aware layout transformation to increase accelerator utilization, and mixed-precision training with bfloat16 for reduced memory.

4.1 Congestion-Aware Data Pipelining

The communication between the compute node and storage node is much slower compared to the on-chip interconnect. Different from TPU-TPU communication using high-speed onboard interconnect or GPU-GPU communication that could go through NVLink/PCI-e bus, communication between the accelerator node and cloud storage goes through Ethernet, which is usually orders of magnitude slower than the former.

Furthermore, during GAN training, a huge amount of data will be transmitted via Ethernet. For instance, the ImageNet 2012 dataset is over 150 GB, and training BigGAN to convergence takes around 240 epochs, resulting in a total of 32.74 TB of data being transmitted during the training phase. When the number of workers increases, the amount of peak

data transmitted increases at the same rate, since data parallel training will send the same amount of data to each worker. This further exacerbates the congestion problem.

Last but not least, network jittering can significantly affect the training throughput. We observe that due to traffic congestion within the data center, the latency between the storage node and the accelerator node is not always stable during peak hours, and it can result in low accelerator utilization when the data pipeline cannot provide enough samples to saturate the accelerator’s compute capability.

Although both TensorFlow and PyTorch implement data pipelines to hide the data loading latency, when severe network jittering happens, data loading and pre-processing take much longer than usual, and it can be a bottleneck in large-scale distributed training. As shown in Fig. 4, when the number of workers scales from 8 to 1024, it spends 13.6% more time on idling, while data outfeeding time stays close. This indicates that the accelerators are busy waiting for data in-feed and gradient synchronization, which leads to reduced utilization.

ParaGAN dynamically adjusts the number of processes and size of the pre-processing buffer in response to the high-variance network. It is implemented by maintaining a sliding window for network latency during runtime. If the current latency over the window exceeds the threshold, ParaGAN will increase the number of threads and buffer for pre-fetching and pre-processing; once the latency falls below the threshold, it releases the resources for pre-processing. This may come at the expense of increased shared memory usage, but shared memory is usually abundant during model training.

Saving model checkpoints will also go through the host. We use an asynchronous checkpoint writer to save model checkpoints. The checkpoint will be streamed into the output buffer instead of having a blocking call to pass it to the CPU host.

4.2 Hardware-Aware Layout Transformation

Zero-padding is frequently used in GAN when the input cannot fit into the specified convolution dimension. For example, a matrix of shape [100, 100] will need 6384 zeros padded to run on a 128×128 matrix unit, which wastes 39% computing resources. As such, zero-padding hinders the accelerator performance because memory is wasted by padding, leading to lower accelerator and memory utilization rates.

Given that there exists an accelerator-dependent format requirement, ParaGAN performs batching opportunistically on the input data. In NCHW (batch size \times number of channels \times height \times width) format, ParaGAN tries to batch them such that N/H/W are multiple of 128 before running on TPU so that the accelerator memory can be efficiently utilized.

On top of the batch dimensions, ParaGAN also seeks opportunities to batch intermediate results to be a multiple of optimal layout dimensions. Such opportunities can be found at *reshape* and *matmul* operators. For instance, if two input matrices are to multiply the same weight, we can concatenate the two input matrices before the matrix multiplication operation to save kernel launch overhead.

4.3 Mixed-Precision Training

ParaGAN supports mixed-precision training with Bfloat16 [?] (bf16) format, which has a lower memory footprint compared to double precision format, allowing users to use a larger batch size or fit a bigger model into memory. However, porting bf16 while maintaining convergence is not straightforward because bf16 trades floating-point precision for range. As a result, hyperparameters with smaller values will also need to be adjusted to accommodate lower-precision bits. For example, ϵ in Adam optimizer is a small value added to the denominator to avoid zero-division. With low-precision bits, it is necessary to use a slightly larger ϵ value.

Throughout our evaluations, we observe that weights and gradients are more sensitive to the bf16 format, while activation can be represented using lower precision without significantly affecting the coverage. Also, the shallow layers are less sensitive compared to deeper layers. We provide bf16 as an option for users to explore.

5 NUMERICAL OPTIMIZATIONS

Synchronous large-batch training is a key technique for accelerating neural network training. However, a major issue with using large batches is that they can lead to numerical instability. In the context of GAN, numerical instability can cause mode collapse or divergence, failing to generate realistic samples with good variety.

A key reason for the numerical instability is that GAN is a two-player game. As such, when the training proceeds, the generator and discriminator can get tightly optimized towards each other. Meanwhile, the generator and discriminator are two networks with different learning dynamics. We argue that the optimization process for GAN should treat them differently.

Therefore, we propose an asynchronous update scheme to decouple the generator and discriminator, and an asymmetric optimization policy for optimizing the two differently.

5.1 Asynchronous Update Scheme

The optimization of GAN is traditionally a serial process where the generator (G) and discriminator (D) update one after another. We question the necessity of an iterative process and propose an asynchronous update scheme.

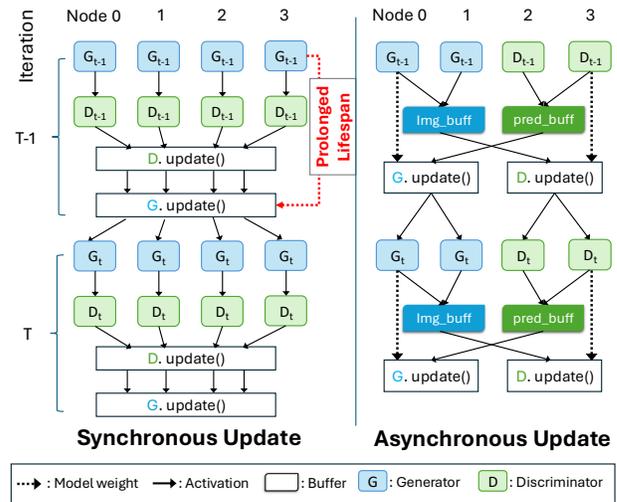


Figure 5: Synchronous update and asynchronous update scheme in ParaGAN. G_t and D_t are the model weights of the generator and discriminator at iteration t .

As shown in Fig. 5, the discriminator will not update its parameter until the generator finishes backpropagation, and vice versa. This is because of the data dependency: D reads both the real sample and the generated sample as input to calculate D loss, while G loss depends on the output and weight of D. The data dependency issue requires both sub-networks to only update one after another, which leads to prolonged tensor lifespan due to blocking, and frequent context switching when updating one or another.

Through our experiments, we find that the discriminator is less sensitive to changes in the generator output. In other words, the discriminator can still perform well even if its input comes from the generator of the previous iteration. This is because the primary goal of the discriminator is to distinguish the generated sample from real samples, and it is an easier task compared to that of the generator. This has motivated us to update the discriminator asynchronously.

As shown in Fig. 5 (right), ParaGAN proposes an asynchronous update scheme: instead of waiting on the other component, the generator/discriminator can write their intermediate output to the buffer and proceed to update using the *current* state of the network. For iteration t , discriminators D_t receive a batch of real and generated samples from the image buffer (img_buff). Similarly, the generators can use the snapshot of the current discriminator state and D predictions in pred_buff to calculate to gradient for backpropagation, breaking the data dependency. It is therefore possible to run both generator and discriminator in parallel

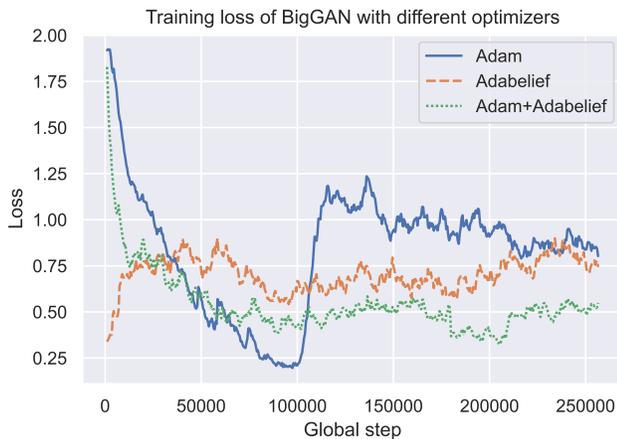


Figure 6: Effect of different optimizer policies. Adam+Adabelief: discriminator trained using Adam and generator trained using Adabelief.

on different nodes. Furthermore, the generator and discriminator ratio is now adjustable thanks to the decoupled design, making it possible to apply different batch sizes for both parties.

5.2 Asymmetric Optimization Policy

Generators and discriminators have different neural network architectures and play opposed roles during training. Thus, they are two entities with significantly distinct numerical properties. It is observed that the discriminator is more stable than the generator. This indicates that the generator and discriminator should be treated differently by using different sets of optimization techniques. However, previous researchers treat them as the same in terms of numerical optimization. We thus propose an Asymmetric Optimization strategy for generator and discriminator in GAN training.

To accomplish this goal, ParaGAN firstly implements some of the latest work on optimizers including Adabelief [41], rectified Adam (RAdam) [21], Lookahead [39], and LARS [36]. We then empirically explored different optimizers for training GAN, and we found that there may not be a single clear winner for all GAN architectures.

However, through experiments, we observe that it can be beneficial to use different optimizers for the generator and discriminator respectively. Fig. 6 shows that although Adam reaches the lowest loss within 100K steps, it collapses thereafter, which is not desirable as it indicates the training has not reached the stable equilibrium. Adabelief is a more adaptive variant of Adam optimizer, and it can adjust the size of the weight update based on a comparison with previous updates. Our experiments also validate that Adabelief outperforms Adam. However, when using an asymmetric pair

of optimizers (Adabelief for the generator and Adam for the discriminator), the training process can converge to a better equilibrium point (lower loss), and the training process is more stable (flatter loss curve), especially towards the end of the training.

We conjecture that the reason is that the two networks have different learning dynamics, thus they require different optimizers for gradient descent. The generator aims to learn from the real-world distribution, thus it must provide a good variety, and the optimizer for the generator should be more agile. However, the task for the discriminator is relatively easy: predicting the label (*real* or *fake*) for the supplied images. Therefore, it is required to make consistently good predictions and be robust to changes. As such, we believe using the asymmetric optimizer policy can combine the best of both worlds.

In ParaGAN, users can set the optimization policy for the generator and discriminator respectively, which currently includes optimizers, learning rate schedulers, warmup epochs, and gradient norms.

6 EVALUATION

In this section, we aim to answer the following questions: 1) how is the performance of ParaGAN compared to other frameworks? 2) how much does each part of the system contribute to the overall performance? And 3) how do the numerical optimizations improve convergence?

In this section, we first evaluate the end-to-end performance of ParaGAN using three metrics:

- **steps per second** measures the number of steps ParaGAN can train per second;
- **images per second** measures the throughput of ParaGAN trained with the ImageNet 2012 dataset;
- **time to solution** measures the time it takes to reach 150k steps on ImageNet at 128×128 resolution.

We first compare ParaGAN with other popular frameworks for end-to-end performance (Sec. 6.2) and evaluate the scaling efficiency for ParaGAN (Sec. 6.3). We ablate the optimizations performed on ParaGAN in Sec 6.5.

6.1 Experiment Setup

We choose the BigGAN model and train it on the ImageNet ILSVRC 2012 dataset as our evaluation method because of BigGAN’s profound impact on high-resolution image generation and its high computational requirements (Table 1), and ImageNet’s wide variety of classes (1000 classes) also presents a significant training challenge. The evaluations were conducted using V100 GPU and TPU v3.

While we use BigGAN to benchmark ParaGAN, our framework is generally applicable to other GAN architectures and

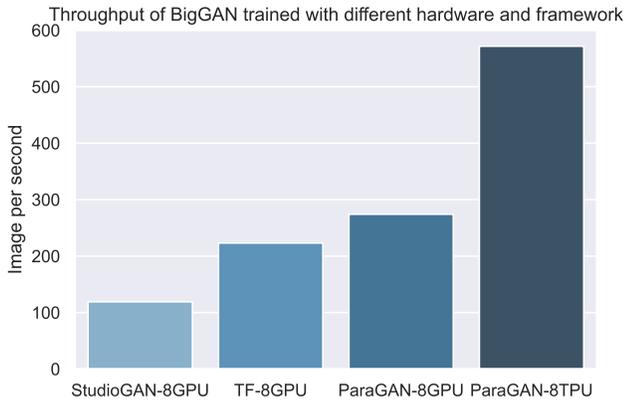


Figure 7: Throughput of different systems and hardware combinations.

datasets, and it is not tightly coupled with any specific accelerator backends.

6.2 Framework-level Experiments

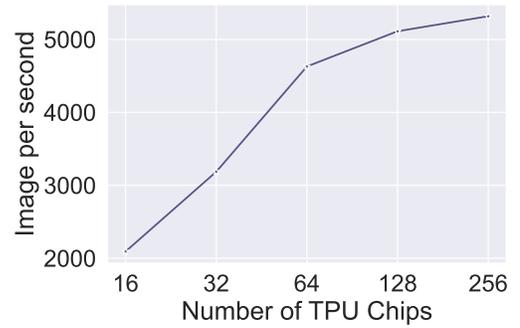
We compare ParaGAN with StudioGAN [15] and native TensorFlow [23] for GPU performance in Fig. 7. For each of the experiments, BigGAN is trained on ImageNet 128×128 resolution. We use $8 \times$ Tesla V100 GPUs except for ParaGAN-8TPU setting which employs 8 TPU.

We observe that ParaGAN outperforms both the native TensorFlow and StudioGAN with 8 GPUs. We conjecture that the performance gain on the GPU setting is mainly attributed to the use of congestion-aware data pipeline and hardware-aware layout transformations. We also observe that the performance gap is further pronounced when switching to the TPU as the accelerator. The following evaluations use the TPU as the accelerator unless otherwise specified.

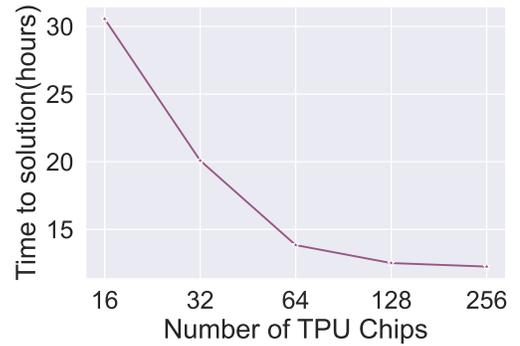
6.3 Scaling Experiments

We will discuss the strong and weak scaling results in this section. In the strong scaling experiments, we keep the total workload constant and vary the number of workers to examine the speedup on time-to-solution. Whereas in the weak scaling experiments, we keep the per worker workload (*batch size per worker*) constant and increase the number of workers.

6.3.1 Strong Scaling. For strong scaling experiments, we fix the total batch size to be 512 and train for 150k steps as target workload. Note that to be consistent with other experiments, we train on BigGAN at 128×128 resolution, which is smaller than the model trained in Fig. 1. We aim to study the effect of decreased per-worker workload when scaling.



(a) Image per second.



(b) Time-to-solution.

Figure 8: Strong scaling with ParaGAN. Each TPU chip has two accelerators.

As can be seen from Fig 8, with an increasing number of workers, the time to solution decreases from over 30 hours to 3 hours. We note that the scaling efficiency drops from 128 to 512 workers (64 to 256 TPU chips). This is because as we fix the global batch size to 512, the per-worker workload drops from 4 samples to 1 sample per batch, which underutilizes the TPU. Thus, the time spent on communication overweighs the computation when the batch size is too small. This is also verified by Fig 8, where the image per second barely improves with an increasing number of accelerator workers. However, when the workload can saturate the accelerator, the scaling efficiency can be near optimal as shown in Fig. 1.

6.3.2 Weak Scaling. In the weak scaling experiments, we fix the batch size per worker and evaluate the performance of our framework by increasing the number of workers. Firstly, we find the largest batch size for a single accelerator that does not lead to out-of-memory error. Then, we scale the total batch size proportionally concerning the number of workers. Therefore, the amount of workload per worker is kept identical. The weak scaling experiments examine how

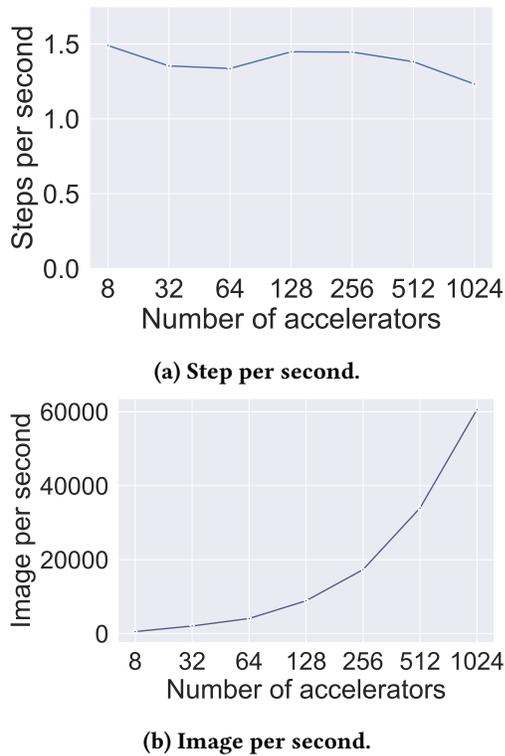


Figure 9: Weak scaling with ParaGAN.

well ParaGAN can handle communication with an increasing number of workers.

As can be seen in Fig. 9, the trend in step-per-second is relatively steady even when using 1024 workers. It shows that ParaGAN can scale out well to a large number of workers while keeping a high scaling efficiency. It is worth noting that, as the number of workers scales, the system will be more likely to suffer from network jittering and congestion. A relatively flat curve (Fig. 9a) indicates that the data pipeline optimization in ParaGAN is effective in case of congestion.

6.4 Accelerator Utilization

The basic computing unit of TPU is MXU (matrix-multiply unit). The utilization of MXU measures the time MXU is being occupied, and higher utilization is more desirable.

We compare the accelerator utilization of BigGAN 128x128 on baseline [23] and ParaGAN. Fig. 10 shows that ParaGAN outperforms native implementation with higher MXU utilization across different TPU configurations. We wish to highlight that even 2% improvement can be important when scaling to thousands of workers.

It is also worth noting that, with an increasing number of accelerators, the amount of communication increases, but ParaGAN can maintain a relatively higher utilization than

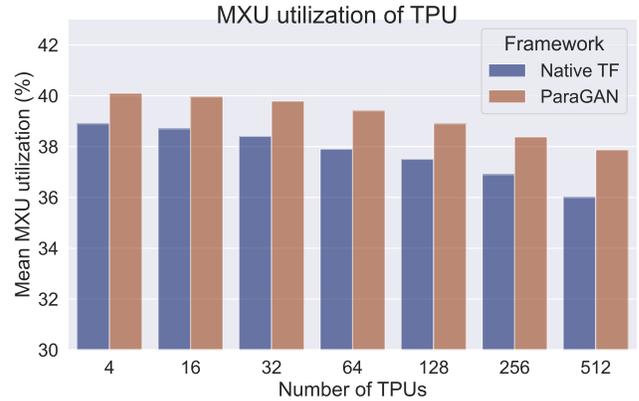


Figure 10: Accelerator utilization of BigGAN trained with native TensorFlow and ParaGAN.

native implementation, and the gap is increasing. It indicates that computation still dominates the training time as compared to native TensorFlow, and ParaGAN can keep up with scaling out.

6.5 Ablation Study

We present the ablation study for the system optimizations in Table 2. For the numerical optimizations, since the convergence also depends on the underlying GAN architecture and datasets, we leave them to be decided by the users. Results in Table 2 are collected on BigGAN trained on ImageNet at 128x128 resolution on 128 TPUv3 accelerators, under the same batch size (2048) from the original paper [3].

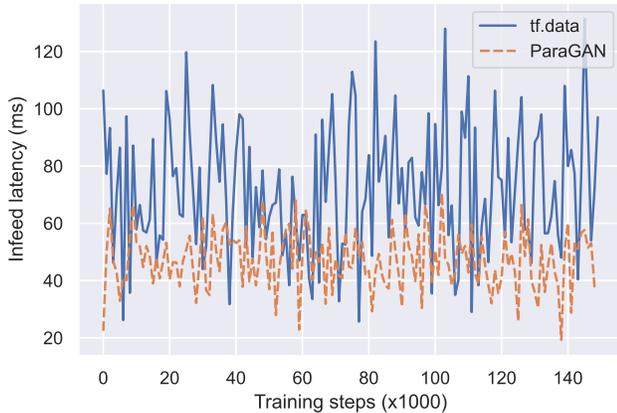
Data pipeline provides 8-15% performance improvement over the baseline. When the number of accelerators increases, network jitter caused by congestion is more likely to happen, making data loading the weakest point in the training process. In ParaGAN, we try to saturate the accelerators by dynamically adjusting the buffer budget for the data pipeline. This is generally applicable, and ParaGAN enables this feature by default.

We compare the performance of our congestion-aware pipeline with TensorFlow’s native data pipeline (tf.data). To ensure the results are comparable, they are run at the same time on the same type of machine pointing to the same dataset storage node, and latency is measured at the time taken to extract a batch of data. As shown in Fig. 11, our pipeline tuner has a lower variance in latency.

Low-precision (with bf16) training provides additional 14-17% performance gain and reduces TPU memory usage by 24%. Bfloat16 format saves activation values in lower bits, which makes it faster to load from memory and communicate with other workers. We have observed while converting activation to lower precision does not harm convergence,

Table 2: Ablation Study of System Optimizations.

Effective Batch Size	System Optimizations			Img/sec
	Data Pipelining	Layout Transformation	Mixed-Precision	
2048				6459
2048	✓			7158 (+10.8%)
2048	✓	✓		7412 (+3.9%)
2048	✓	✓	✓	8539 (+15.2%)

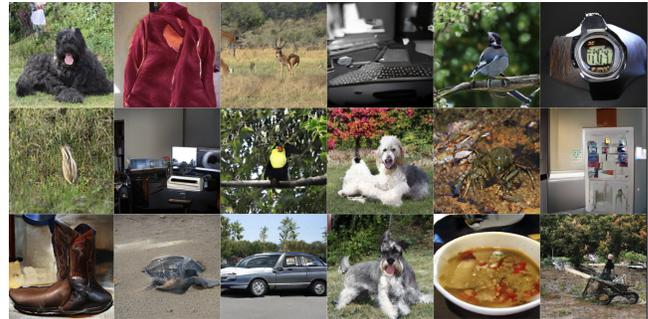
**Figure 11: Data pipeline latency.**

numerical instability may happen when converting gradient and weight into bfloat16. In ParaGAN, the users can control the precision for each layer.

Layout transformation provides 4% additional improvement by increasing the accelerator utilization and reducing kernel launch overhead. This is achieved using a profiling-guided approach, as we observe that many smaller tensors are fed into the same convolution kernel. Considering that it optimizes the data layout, it is possible to integrate it into the XLA as an HLO pass.

6.6 Generating High-Resolution Images

To our knowledge, we are the first to successfully train BigGAN at 1024×1024 resolution, which is 4 times higher than the result shown in the original BigGAN. Training at high resolution is particularly challenging because the generator will need to use more channels and deconvolutional layers to generate more details. It is therefore more sensitive to hyperparameters and initialization. Different from ProgressGAN [16] where they use progressive growing to train low-resolution images first before increasing the resolution, we directly train it on 1024×1024 resolution, which is more challenging, and it requires the numerical optimization techniques we discussed.

**Figure 12: Output of BigGAN at 1024x1024 resolution. Best viewed in color.**

The generated results (trained on ImageNet dataset) achieved an Inception Score (IS) [32] of 239.3 and Fréchet Inception Distance (FID) of 13.6. They are presented in Fig. 12 for visual evaluation.

6.7 Convergence of Async. Update Scheme

We study the convergence behavior of the asynchronous update scheme on SNGAN and present in Fig. 13. We observe that the asynchronous update scheme can accelerate convergence on smaller datasets, and the benefit is more obvious in the early stage of training. However, the asynchronous update scheme (Async G-512 D-256) struggles to converge on high-resolution image generation tasks despite reaching lower FID quicker than the synchronous versions before 16K steps. Throughout repeated experiments and other models, we observe similar results. We conjecture that the asynchronous update scheme may be used to accelerate the earlier stage of training before switching to the synchronous update scheme for better convergence.

We recognize that theoretical convergence analysis in the context of GAN training is an active area of research [2, 11, 18, 24, 28], and is particularly challenging. As GANs operate on a two-party optimization problem, their convergence is inherently difficult to guarantee, especially when using stochastic gradient descent. Furthermore, introducing asynchrony, while beneficial in terms of performance, adds another layer of complexity in analyzing convergence and stability. While we have relied on empirical results to

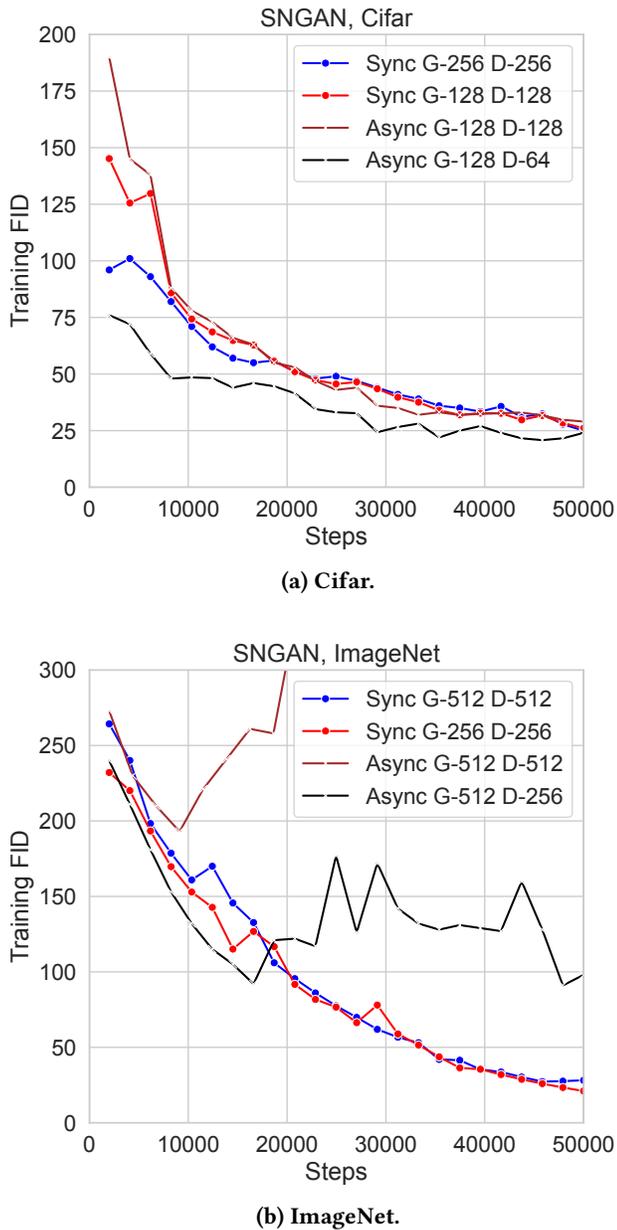


Figure 13: The FID score of images produced using the Async Update scheme (lower is better). The number in the legend represents batch size.

demonstrate the effectiveness of these techniques, we leave it as future work to formalize a theoretical framework for asynchronous GAN training.

7 RELATED WORK

7.1 Generative Adversarial Network

Several methods for GANs are proposed to improve the performance and training stability of GAN. SNGAN [27] uses spectral normalization to stabilize the training of the discriminator and achieves better inception scores relative to previous studies. BigGAN [3] scales up the batch size and the model size for training to generate high-resolution and high-quality images. StyleGAN [17] enhances the generator model to produce photorealistic faces with high resolution meanwhile being able to control the style of the generated image through varying the style vectors and noise. However, the above works were less focused on the training acceleration aspect.

7.1.1 Mode Collapse. Mode collapse is a common type of GAN failure where the generator fails to produce a good variety of samples that can fool the discriminator. When it happens, the generator sticks with a set of similar-looking output patterns without progressing, and the discriminator can almost always classify the generated samples. Essentially, the generator’s optimization process reaches a saddle point, and rotates through the small set of output. Several approaches have been proposed to try to solve the problem. VEEGAN [33] features a reconstruction network reversing the generator by mapping samples to random noise. Unrolled GAN [26] uses a loss function for the generator concerning an unrolled optimization of the discriminator. This way, the generator training can be adjusted by both current outputs and optimal outputs of the discriminator. Still, mode collapse remains challenging to identify and mitigate.

7.2 Distributed GAN Training

Existing works on distributed training for GAN are relatively limited with a focus on the privacy-preserving perspective. Since GAN consists of two sub-networks, a common approach is to let multiple discriminators serve one centralized generator using parameter server [4, 10, 29]. The central generator sends synthesized output images to the distributed discriminators, and the discriminators update the generator with their predictions. FeGAN [9] instead deploys a complete GAN on each device to address the data skewness and mode collapse issue. AI-GAN [14] tackles the image-deraining problem by using a two-branch network that learns a disentangled representation for the rain and background, and jointly optimizes the two branches via mutual adversarial optimization. [34] adopts the federated learning setting on the medical image generation problem on low-resolution images. [35] proposes a PDE-informed GAN architecture for subsurface flow characterization problem, achieving 93.5% scaling efficiency on 27500 GPUs with communication and

load balancing optimizations. [22] presents a decentralized training algorithm named DPOSG that caters to the non-convex non-concave min-max optimization process.

To the best of our knowledge, ParaGAN is the first system that efficiently scales GAN on high-resolution image generation tasks on a large-scale cluster.

8 CONCLUSION

ParaGAN is a large-scale distributed GAN training framework that supports high-resolution image generation with near-linear scalability. ParaGAN is optimized with a dynamic data pipeline, mixed-precision training, and layout transformation. We show that it is possible to train the generator and discriminator independently using an asynchronous update scheme and asymmetric optimization policy. ParaGAN scales almost optimally to 1024 accelerators, and it can greatly reduce the time to train a GAN model from weeks to hours. We believe ParaGAN can advance GAN research by accelerating the training process.

9 ACKNOWLEDGEMENT

This research is supported by the Ministry of Education AcRF Tier 1 grant (No. T1 251RES2104) and Tier 2 grant (No. MOE-T2EP20222-0016). This is also supported by the Cloud TPUs from Google’s TPU Research Cloud (TRC).

REFERENCES

- [1] David Balduzzi, Sebastien Racaniere, James Martens, Jakob Foerster, Karl Tuyls, and Thore Graepel. 2018. The mechanics of n-player differentiable games. In *International Conference on Machine Learning*. PMLR, 354–363.
- [2] Samuel A. Barnett. 2018. *Convergence Problems with Generative Adversarial Networks (GANs)*. Technical Report. <http://arxiv.org/abs/1806.11382>
- [3] Andrew Brock, Jeff Donahue, and Karen Simonyan. 2019. Large scale GAN training for high fidelity natural image synthesis. *7th International Conference on Learning Representations, ICLR 2019* (9 2019). <http://arxiv.org/abs/1809.11096>
- [4] Qi Chang, Hui Qu, Yikai Zhang, Mert Sabuncu, Chao Chen, Tong Zhang, and Dimitris N Metaxas. 2020. Synthetic learning: Learn from distributed asynchronous discriminator gan without sharing medical image data. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 13856–13866.
- [5] MMGeneration Contributors. 2021. MMGeneration: OpenMMLab Generative Model Toolbox and Benchmark. <https://github.com/open-mmlab/mmgeneration>.
- [6] Constantinos Daskalakis and Ioannis Panageas. 2018. The limit points of (optimistic) gradient descent in min-max optimization. *Advances in Neural Information Processing Systems* 31 (2018).
- [7] Ugur Demir and Gozde Unal. 2018. Patch-based image inpainting with generative adversarial networks. *arXiv preprint arXiv:1803.07422* (2018).
- [8] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial networks. *arXiv preprint arXiv:1406.2661* (2014).
- [9] Rachid Guerraoui, Arsany Guirguis, Anne-Marie Kermarrec, and Erwan Le Merrer. 2020. Fegan: Scaling distributed gans. In *Proceedings of the 21st International Middleware Conference*. 193–206.
- [10] Corentin Hardy, Erwan Le Merrer, and Bruno Sericola. 2019. *MD-GAN: Multi-discriminator generative adversarial networks for distributed datasets*. Technical Report. 866–877 pages. <https://doi.org/10.1109/IPDPS.2019.00095>
- [11] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. 2017. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems* 30 (2017).
- [12] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. 2017. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1125–1134.
- [13] Chi Jin, Praneeth Netrapalli, and Michael Jordan. 2020. What is local optimality in nonconvex-nonconcave minimax optimization?. In *International Conference on Machine Learning*. PMLR, 4880–4889.
- [14] Xin Jin, Zhibo Chen, and Weiping Li. 2020. AI-GAN: Asynchronous interactive generative adversarial network for single image rain removal. *Pattern Recognition* 100 (4 2020), 107143. <https://doi.org/10.1016/J.PATCOG.2019.107143>
- [15] Minguk Kang and Jaesik Park. 2020. ContraGAN: Contrastive Learning for Conditional Image Generation. *Conference on Neural Information Processing Systems (NeurIPS)*.
- [16] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. 2017. Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196* (2017).
- [17] Tero Karras, Samuli Laine, and Timo Aila. 2019. A Style-Based Generator Architecture for Generative Adversarial Networks. [arXiv:1812.04948](https://arxiv.org/abs/1812.04948)
- [18] Naveen Kodali, Jacob Abernethy, James Hays, and Zsolt Kira. 2017. On convergence and stability of gans. *arXiv preprint arXiv:1705.07215* (2017).
- [19] Christian Ledig, Lucas Theis, Ferenc Huszar, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. 2017. Photo-realistic single image super-resolution using a generative adversarial network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4681–4690.
- [20] Kwot Sin Lee and Christopher Town. 2020. Mimicry: Towards the Reproducibility of GAN Research. (2020).
- [21] Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. 2020. On the Variance of the Adaptive Learning Rate and Beyond. [arXiv:1908.03265](https://arxiv.org/abs/1908.03265)
- [22] Mingrui Liu, Wei Zhang, Youssef Mroueh, Xiaodong Cui, Jarret Ross, Tianbao Yang, and Payel Das. 2020. A decentralized parallel algorithm for training generative adversarial nets. *Advances in Neural Information Processing Systems* 33 (2020), 11056–11070.
- [23] Mario Lucic, Karol Kurach, Marcin Michalski, Olivier Bousquet, and Sylvain Gelly. 2018. *Are Gans created equal? A large-scale study*. Technical Report. 700–709 pages.
- [24] Lars Mescheder, Andreas Geiger, and Sebastian Nowozin. 2018. Which training methods for GANs do actually converge?. In *International conference on machine learning*. PMLR, 3481–3490.
- [25] Lars Mescheder, Sebastian Nowozin, and Andreas Geiger. 2017. *The numerics of GANs*. Technical Report. 1826–1836 pages. <https://github.com/LMescheder/>
- [26] Luke Metz, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. 2017. Unrolled Generative Adversarial Networks. [arXiv:1611.02163](https://arxiv.org/abs/1611.02163)
- [27] Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. 2018. Spectral Normalization for Generative Adversarial

- Networks. arXiv:1802.05957
- [28] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J. Wright. 2011. *HOGWILD!: A lock-free approach to parallelizing Stochastic Gradient Descent*. Technical Report.
- [29] Hui Qu, Yikai Zhang, Qi Chang, Zhennan Yan, Chao Chen, and Dimitris Metaxas. 2020. Learn Distributed GAN with Temporary Discriminators. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 12372 LNCS (8 2020), 175–192. https://doi.org/10.1007/978-3-030-58583-9_{ }11
- [30] Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).
- [31] Amit Sabne. 2020. XLA : Compiling Machine Learning for Peak Performance.
- [32] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. 2016. Improved techniques for training gans. *arXiv preprint arXiv:1606.03498* (2016).
- [33] Akash Srivastava, Lazar Valkov, Chris Russell, Michael U. Gutmann, and Charles Sutton. 2017. VEEGAN: Reducing Mode Collapse in GANs using Implicit Variational Learning. arXiv:1705.07761
- [34] Jamal Toutouh, Mathias Esteban, and Sergio Nesmachnow. 2020. Parallel/distributed generative adversarial neural networks for data augmentation of COVID-19 training images. In *Latin American High Performance Computing Conference*. Springer, 162–177.
- [35] Liu Yang, Sean Treichler, Thorsten Kurth, Keno Fischer, David Barajas-Solano, Josh Romero, Valentin Churavy, Alexandre Tartakovsky, Michael Houston, Mr Prabhat, et al. 2019. Highly-scalable, physics-informed GANs for learning solutions of stochastic PDEs. In *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE, 1–11.
- [36] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888* (2017).
- [37] Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, and Thomas S Huang. 2018. Generative image inpainting with contextual attention. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 5505–5514.
- [38] Han Zhang, Ian Goodfellow, Dimitris Metaxas, and Augustus Odena. 2019. Self-Attention Generative Adversarial Networks. arXiv:1805.08318 [stat.ML]
- [39] Michael R. Zhang, James Lucas, Geoffrey Hinton, and Jimmy Ba. 2019. *Lookahead optimizer: k Steps forward, 1 step back*. Technical Report. <https://github.com/michaelrzhang/lookahead>.
- [40] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. 2017. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*. 2223–2232.
- [41] Juntang Zhuang, Tommy Tang, Yifan Ding, Sekhar Tatikonda, Nicha Dvornek, Xenophon Papademetris, and James S. Duncan. 2020. *Adabelief optimizer: Adapting stepsizes by the belief in observed gradients*. Technical Report. <https://github.com/juntang-zhuang/Adabelief-Optimizer>