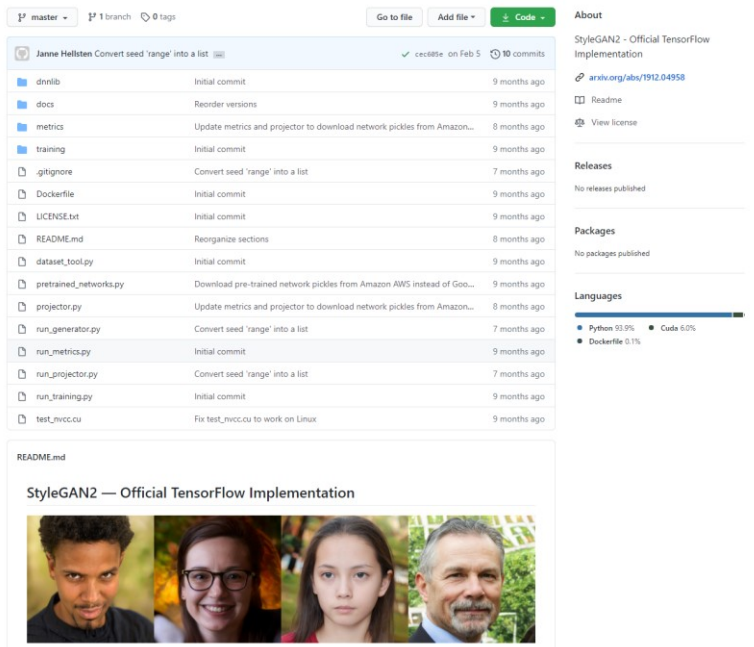
# MULTI-NODE TRAINING FOR STYLEGAN2

Niki Loppi (NVIDIA AI Tech Center)

Tuomas Kynkäänniemi (Aalto University)

# CONTENTS

- StyleGAN2 introduction
- Multi-node implementation
- Validation
- Performance benchmarks



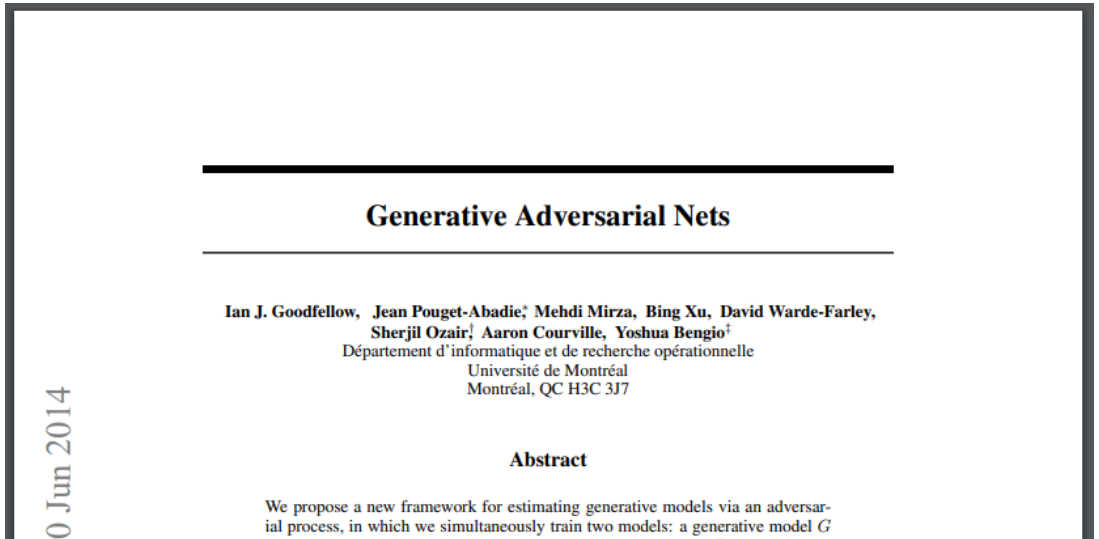


## Acknowledgement

NV Research Helsinki: Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, Timo Aila
https://github.com/NVlabs/stylegan2

# GENERATIVE ADVERSARIAL NETWORKS

- Generative modelling:
  - To learn the characteristics of a dataset and encode into a probabilistic model
  - Sampling the probabilistic model allows generation of similar data
- GAN:
  - Use an auxiliary Discriminator network as a "loss function"
  - Generator wants to maximize score and Discriminator wants to minimize score



Generative Adversarial Nets

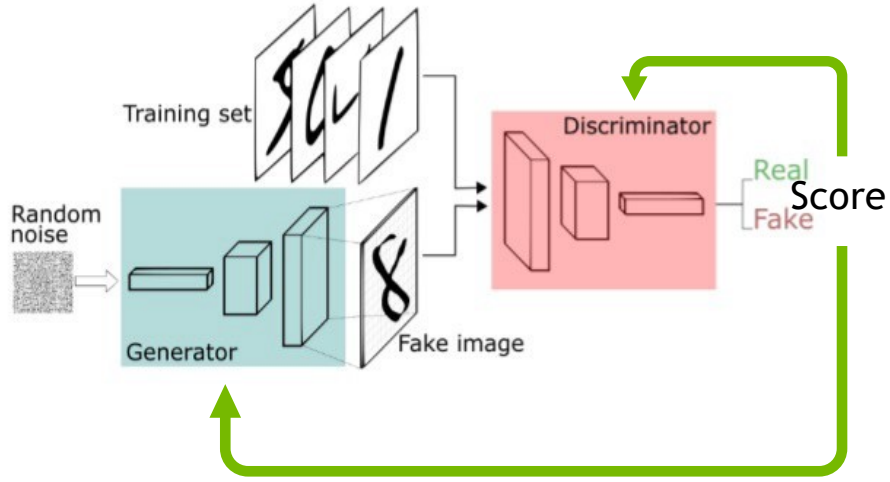
Ian J. Goodfellow, Jean Pouget-Abadie*, Mehdi Mirza, Bing Xu, David Warde-Farley,
Sherjil Ozair†, Aaron Courville, Yoshua Bengio‡
Département d'informatique et de recherche opérationnelle
Université de Montréal
Montréal, QC H3C 3J7

**Abstract**

We propose a new framework for estimating generative models via an adversarial process, in which we simultaneously train two models: a generative model $G$

10 Jun 2014

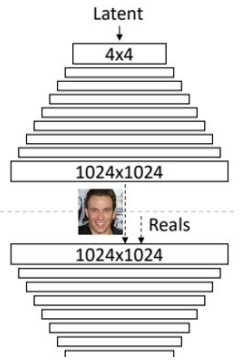
GAN Architecture

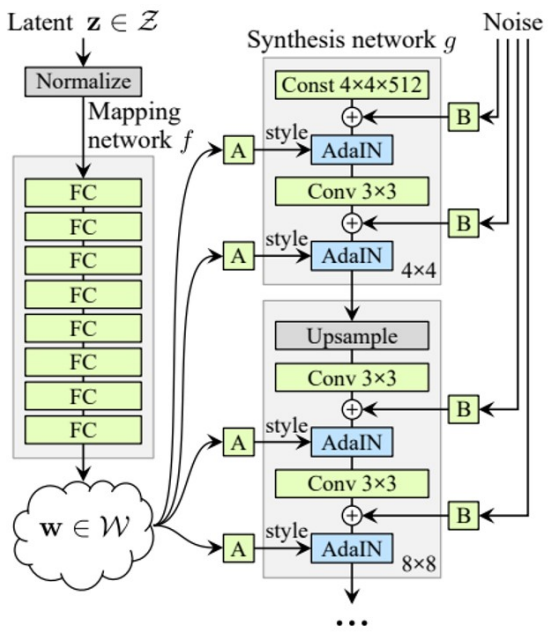



https://mc.ai/learning-generative-adversarial-networks-gans/

# STYLEGAN2

- Tensorflow - approx. 6000 lines of code
- Progressive growing of GANs (Karras et al. 2018)
  - Progressive growing allows to tackle low frequency errors first
  - Linear cross-fading during resolution transition
- An effective mode collapse avoidance mechanism (Karras et al. 2018)
  - Compute std dev over images in a minibatch
  - Average over features and pixels
  - Append scalars a feature
- A style-based Generator architecture (Karras et al. 2019)
  - Mapping network (MLP) + Synthesis network
  - The mapping network and affine transformations to draw styles from a learned distribution
  - Synthesis network to construct the image



# MULTI-NODE STYLEGAN2

## Motivation

| Configuration | Resolution | Total kimg | 1 GPU | 2 GPUs | 4 GPUs | 8 GPUs | GPU mem |
|---|---|---|---|---|---|---|---|
| config-f | 1024×1024 | 25000 | 69d 23h | 36d 4h | 18d 14h | 9d 18h | 13.3 GB |

- Standard implementation
  - Single process, NCCL allreduce GPU Direct P2P
  - python run_training.py --num-gpus=8 --config=config-e --dataset=ffhq --mirror-augment=true --total-kimg=10000
  - ```
    for gpu in range(num_gpus):
        with tf.device('/gpu:%d' % gpu):
            do things
    ```
- Multi-node implementation via Horovod
  - One MPI process per GPU
  - horovodrun –n 8 python run_training.py --num-gpus=1 --config=config-e --dataset=ffhq --mirror-augment=true --total-kimg=10000
  - ```
    hvd.init()
    os.environ['CUDA_VISIBLE_DEVICES'] = str(hvd.local_rank())

    for gpu in range(num_gpus):  # num_gpus is now 1!
        with tf.device('/gpu:%d' % gpu):
            do things

    if hvd.rank() == 0:
        do something
    ```

# MULTI-NODE STYLEGAN2

## Data Sharding

```
class TFRecordDataset:
    def __init__(self,...)

    with tf.name_scope('Dataset'), tf.device('/cpu:0'):
        self._tf_minibatch_in = tf.placeholder(tf.int64, name='minibatch_in', shape=[])
        self._tf_labels_var = tflib.create_var_with_large_initial_value(self._np_labels, name='labels_var')
        self._tf_labels_dataset = tf.data.Dataset.from_tensor_slices(self._tf_labels_var)
        for tfr_file, tfr_shape, tfr_lod in zip(tfr_files, tfr_shapes, tfr_lods):
            if tfr_lod < 0:
                continue
            dset = tf.data.TFRecordDataset(tfr_file, compression_type='', buffer_size=buffer_mb<<20)
            if max_images is not None:
                dset = dset.take(max_images)
            if self._sharding:
                self._tf_labels_dataset = self._tf_labels_dataset.shard(num_shards=hvd.size(), index=hvd.rank())
                dset = dset.shard(num_shards=hvd.size(), index=hvd.rank())

            dset = dset.map(self.parse_tfrecord_tf, num_parallel_calls=num_threads)
            dset = tf.data.Dataset.zip((dset, self._tf_labels_dataset))
            bytes_per_item = np.prod(tfr_shape) * np.dtype(self.dtype).itemsize
            if shuffle_mb > 0:
                dset = dset.shuffle(((shuffle_mb << 20) - 1) // bytes_per_item + 1)
            if repeat:
                dset = dset.repeat()
            if prefetch_mb > 0:
                dset = dset.prefetch(((prefetch_mb << 20) - 1) // bytes_per_item + 1)
            dset = dset.batch(self._tf_minibatch_in)
            self._tf_datasets[tfr_lod] = dset
        self._tf_iterator = tf.data.Iterator.from_structure(self._tf_datasets[0].output_types,
                                                            self._tf_datasets[0].output_shapes)
        self._tf_init_ops = {lod: self._tf_iterator.make_initializer(dset) for lod, dset in self._tf_datasets.items()}
```

- Training dataset is split into partitions
- Progressive growing uses multiple TFRecords files

```
nloppi sa   1078000 Sep  9 04:06 fewthumbs-r02.tfrecords
nloppi sa   2717000 Sep  9 04:06 fewthumbs-r03.tfrecords
nloppi sa   9053000 Sep  9 04:05 fewthumbs-r04.tfrecords
nloppi sa  34397000 Sep  9 04:06 fewthumbs-r05.tfrecords
nloppi sa 135773000 Sep  9 04:06 fewthumbs-r06.tfrecords
nloppi sa 541354000 Sep  9 04:06 fewthumbs-r07.tfrecords
```

# MULTI-NODE STYLEGAN2

## Seeding

```python
def G_logistic(G, D, opt, training_set, minibatch_size):
    _ = opt
    latents = tf.random_normal([minibatch_size] + G.input_shapes[0][1:])
    labels = training_set.get_random_labels_tf(minibatch_size)
    fake_images_out = G.get_output_for(latents, labels, is_training=True)
    fake_scores_out = D.get_output_for(fake_images_out, labels, is_training=True)
    loss = -tf.nn.softplus(fake_scores_out) # log(1-sigmoid(fake_scores_out)) # pylint: disable=invalid-unary-operand-type
    return loss, None


def D_logistic(G, D, opt, training_set, minibatch_size, reals, labels):
    _ = opt, training_set
    latents = tf.random_normal([minibatch_size] + G.input_shapes[0][1:])
    fake_images_out = G.get_output_for(latents, labels, is_training=True)
    real_scores_out = D.get_output_for(reals, labels, is_training=True)
    fake_scores_out = D.get_output_for(fake_images_out, labels, is_training=True)
    real_scores_out = autosummary('Loss/scores/real', real_scores_out)
    fake_scores_out = autosummary('Loss/scores/fake', fake_scores_out)
    loss = tf.nn.softplus(fake_scores_out) # -log(1-sigmoid(fake_scores_out))
    loss += tf.nn.softplus(-real_scores_out) # -log(sigmoid(real_scores_out)) # pylint: disable=invalid-unary-operand-type
    return loss, None
```

- A lot of GAN functionalities are based on random number generators that require a seed, e.g. latent vectors

```python
def run(dataset, data_dir, result_dir, config_id, num_gpus, total_kimg, gamma, mirror_augment, metrics):
    train     = EasyDict(run_func_name='training.training_loop.training_loop') # Options for training loop.
    G         = EasyDict(func_name='training.networks_stylegan2.G_main')       # Options for generator network.
    D         = EasyDict(func_name='training.networks_stylegan2.D_stylegan2')  # Options for discriminator network.
    grid      = EasyDict(size='8k', layout='random')                           # Options for setup_snapshot_image_grid().
    ...
    sc        = dnnlib.SubmitConfig()                                          # Options for dnnlib.submit_run().
    tf_config = {'rnd.np_random_seed': 1000 + hvd.rank()}
```

- Multi-node StyleGAN2 is process parallel, perturbing the manual seed is necessary.

```python
def training_loop():
    ...
    bcast_op = hvd.broadcast_global_variables(0)
    tflib.run(bcast_op)

    while cur_nimg < total_kimg * 1000:
        if dnnlib.RunContext.get().should_stop(): break
        # training loop
```

- Because of this, initial states differ between ranks
- Need to broadcast from root before trainining

# MULTI-NODE STYLEGAN2

## Communication

```
class Optimizer(object):
    ...
    def apply_updates(self, allow_no_op: bool = False) -> tf.Operation:
        """Construct training op to update the registered variables based on their gradients."""

        # Clean up gradients.
          ...

        # Sum gradients across devices.
        if len(self._devices) >= 1:
            with tfutil.absolute_name_scope(self.scope + "/Broadcast"), tf.device(None):
                for all_vars in zip(*[device.grad_clean.keys() for device in self._devices.values()]):
                    if len(all_vars) > 0 and all(dim > 0 for dim in all_vars[0].shape.as_list()):
                        all_grads = [device.grad_clean[var] for device, var in zip(self._devices.values(), all_vars)]
-                       all_grads = nccl_ops.all_sum(all_grads)
+                       all_grads = [hvd.allreduce(grad, average=False) for grad in all_grads]
                        for device, var, grad in zip(self._devices.values(), all_vars, all_grads):
                            device.grad_clean[var] = grad

        # Apply updates separately on each device.
        for device_idx, device in enumerate(self._devices.values()):
            with tfutil.absolute_name_scope(self.scope + "/Apply%d" % device_idx), tf.device(device.name):
                ...

                # No overflow => apply gradients.
                all_ok = tf.reduce_all(tf.stack([acc_ok] + [tf.reduce_all(tf.is_finite(g))
                                                            for g in device.grad_acc.values()]))
                apply_op = lambda: device.optimizer.apply_gradients([(tf.cast(grad, var.dtype), var)
                                                                     for var, grad in device.grad_acc.items()])
                all_ops.append(tf.cond(all_ok, apply_op, tf.no_op))
```

- In distributed training gradients are averaged across ranks
- Previously

```
all_grads = nccl_ops.all_sum([grad_0, grad_1 …])
```

- Now

```
len(self._devices) = 1
all_grads = hvd.all_reduce(grad)
```

# MULTI-NODE STYLEGAN2

## Metrics

```python
class FID(metric_base.MetricBase):
    def __init__(self, num_images, minibatch_per_gpu, **kwargs):
        super().__init__(**kwargs)
        self.num_images = num_images
        self.minibatch_per_gpu = minibatch_per_gpu

    def _evaluate(self, Gs, Gs_kwargs, num_gpus):
        minibatch_size = hvd.size() * self.minibatch_per_gpu
        inception = misc.load_pkl('http://d36zk2xti64re0.cloudfront.net/stylegan1/networks/metrics/inception_v3_features.pkl'
        activations = np.empty([self.num_images, inception.output_shape[1]], dtype=np.float32)
        # Calculate statistics for reals.
        cache_file = self._get_cache_file_for_reals(num_images=self.num_images)
        os.makedirs(os.path.dirname(cache_file), exist_ok=True)
        if os.path.isfile(cache_file):
            mu_real, sigma_real = misc.load_pkl(cache_file)
        else:
            for idx, images in enumerate(self._iterate_reals(minibatch_size=minibatch_size)):
                begin = idx * minibatch_size
                end = min(begin + minibatch_size, self.num_images)
                activations[begin:end] = inception.run(images[:end-begin], num_gpus=num_gpus, assume_frozen=True)
                if end == self.num_images:
                    break
            mu_real = np.mean(activations, axis=0)
            sigma_real = np.cov(activations, rowvar=False)
            misc.save_pkl((mu_real, sigma_real), cache_file)

        # Construct TensorFlow graph.
        for gpu_idx in range(num_gpus):
            with tf.device('/gpu:%d' % gpu_idx):
                Gs_clone = Gs.clone()
                inception_clone = inception.clone()
                latents = tf.random_normal([self.minibatch_per_gpu] + Gs_clone.input_shape[1:])
                labels = self._get_random_labels_tf(self.minibatch_per_gpu)
                images = Gs_clone.get_output_for(latents, labels, **Gs_kwargs)
                images = tflib.convert_images_to_uint8(images)
                result_expr = inception_clone.get_output_for(images)

        with tf.device(None):
            result_expr = hvd.allgather(result_expr)

        # Calculate statistics for fakes.
        for begin in range(0, self.num_images, minibatch_size):
            self._report_progress(begin, self.num_images)
            end = min(begin + minibatch_size, self.num_images)
            activations[begin:end] = tflib.run(result_expr)[:end-begin]
        mu_fake = np.mean(activations, axis=0)
        sigma_fake = np.cov(activations, rowvar=False)

        # Calculate FID.
        m = np.square(mu_fake - mu_real).sum()
        s, _ = scipy.linalg.sqrtm(np.dot(sigma_fake, sigma_real), disp=False) # pylint: disable=no-member
        dist = m + np.trace(sigma_fake + sigma_real - 2*s)
        self._report_result(np.real(dist))
```

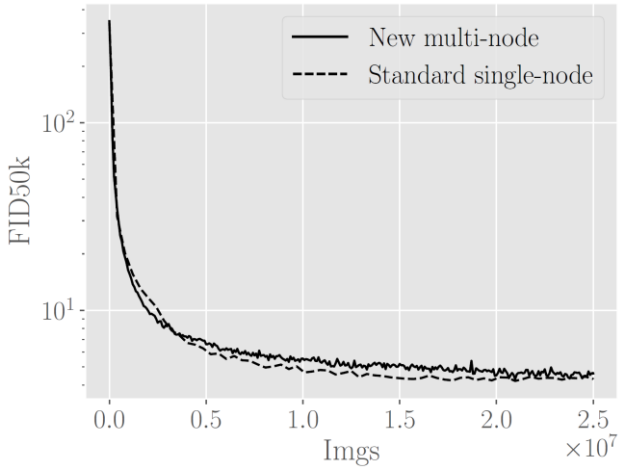
- Frechet inception (50k) score :
  - Feed 50k fakes and reals through pretrained convolutional neural network
  - Get the activations from last layer
  - Compute mean and covariance across the 50k samples of these activations
  - Compute the Frechet distance between mean and fake statistics
- 50k reals taken from the entire dataset, not from a shard.
- All processes can generate fakes and feed them through the inception network
- All_gather before computation of mean and covariance

# VALIDATION

## On Puhti-AI

- 4 x NVIDIA Tesla V100 nodes
- StyleGAN2 config-f
- FFHQ set 256x256
- Effective batch size = 32
- nvcr.io/nvidia/tensorflow:20.08-tf1-py3

| Case | $N_{GPU}$ | Performance (s/kimgs) |
|---|---|---|
| Standard single-node | 4 | 30.8 |
| New single-node | 4 | 28.5 |
| New multi-node | 16 | 9.3 |



# PERFORMANCE BENCHMARKS

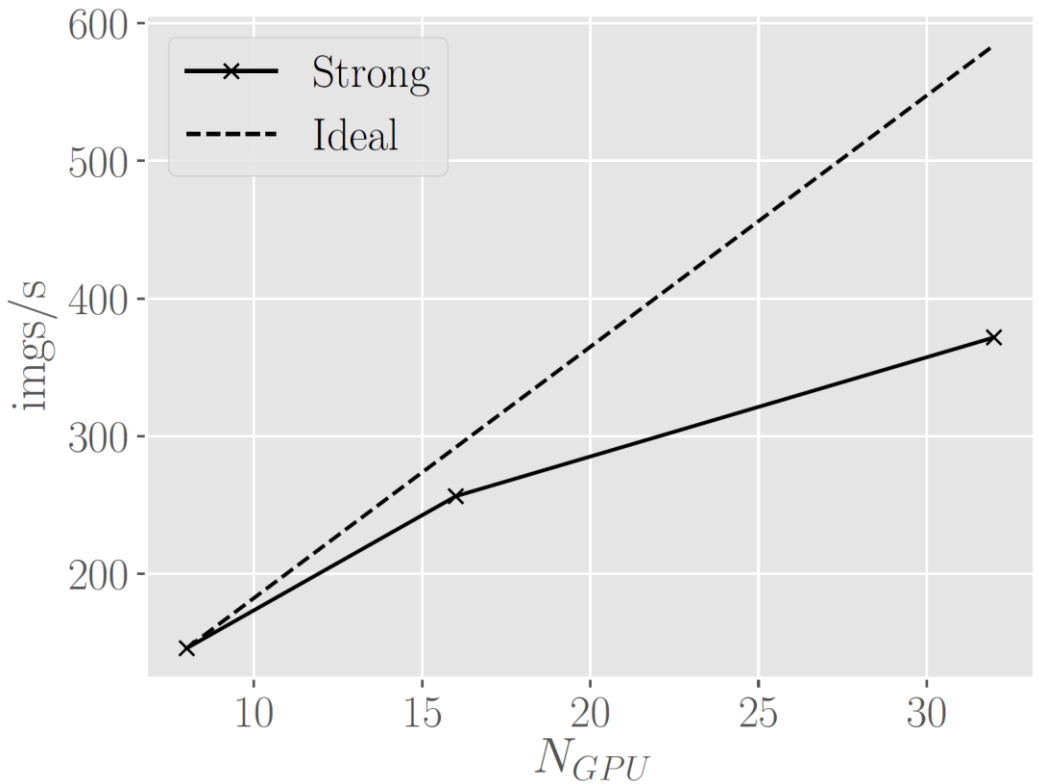
## On NVIDIA Selene

- DGX-A100 nodes
- StyleGAN2 config-f
- FFHQ set 256x256
- Default batch_size = 64
- nvcr.io/nvidia/tensorflow:20.08-tf1-py3
- export OMPI_MCA_pml=ucx
- export OMPI_MCA_btl=^openib
- export HOROVOD_CACHE_CAPACITY=0

# MULTI-NODE SCALING TESTS

## Strong scaling

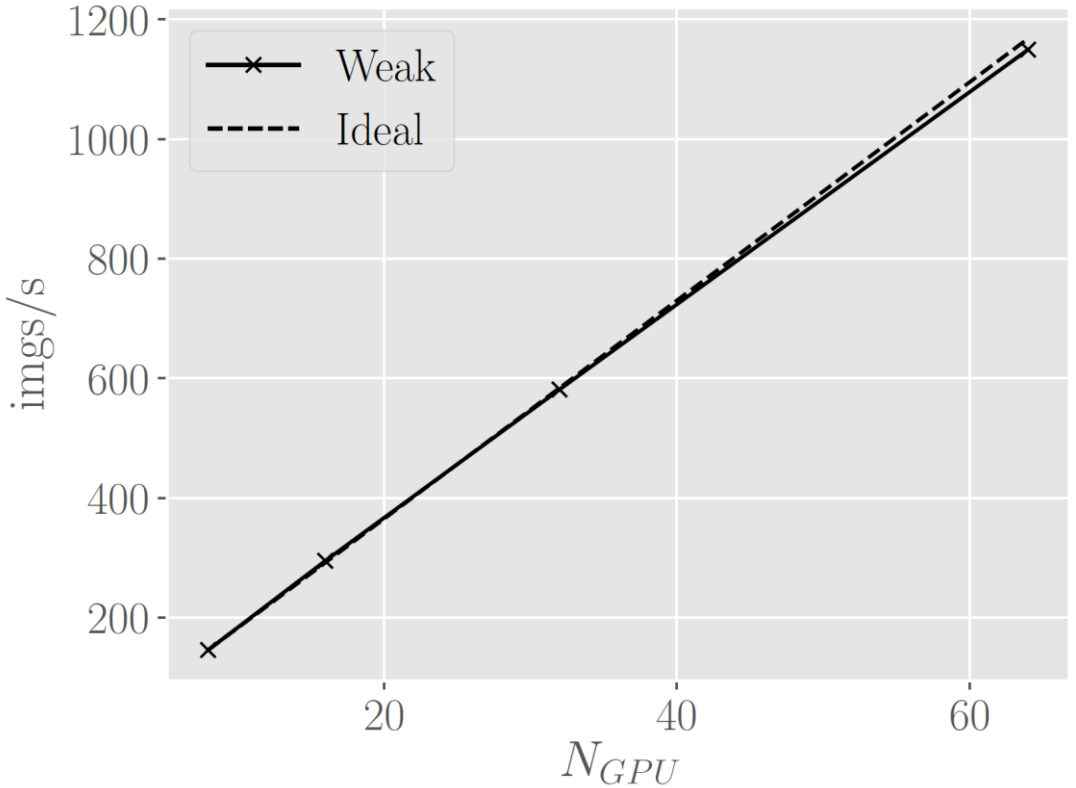
- Effective batch size constant 64
  (per gpu: 8, 4, 2 )
- 63 % parallel efficiency (2.5x speed-up by quadrupling GPUs)



# MULTI-NODE SCALING TESTS

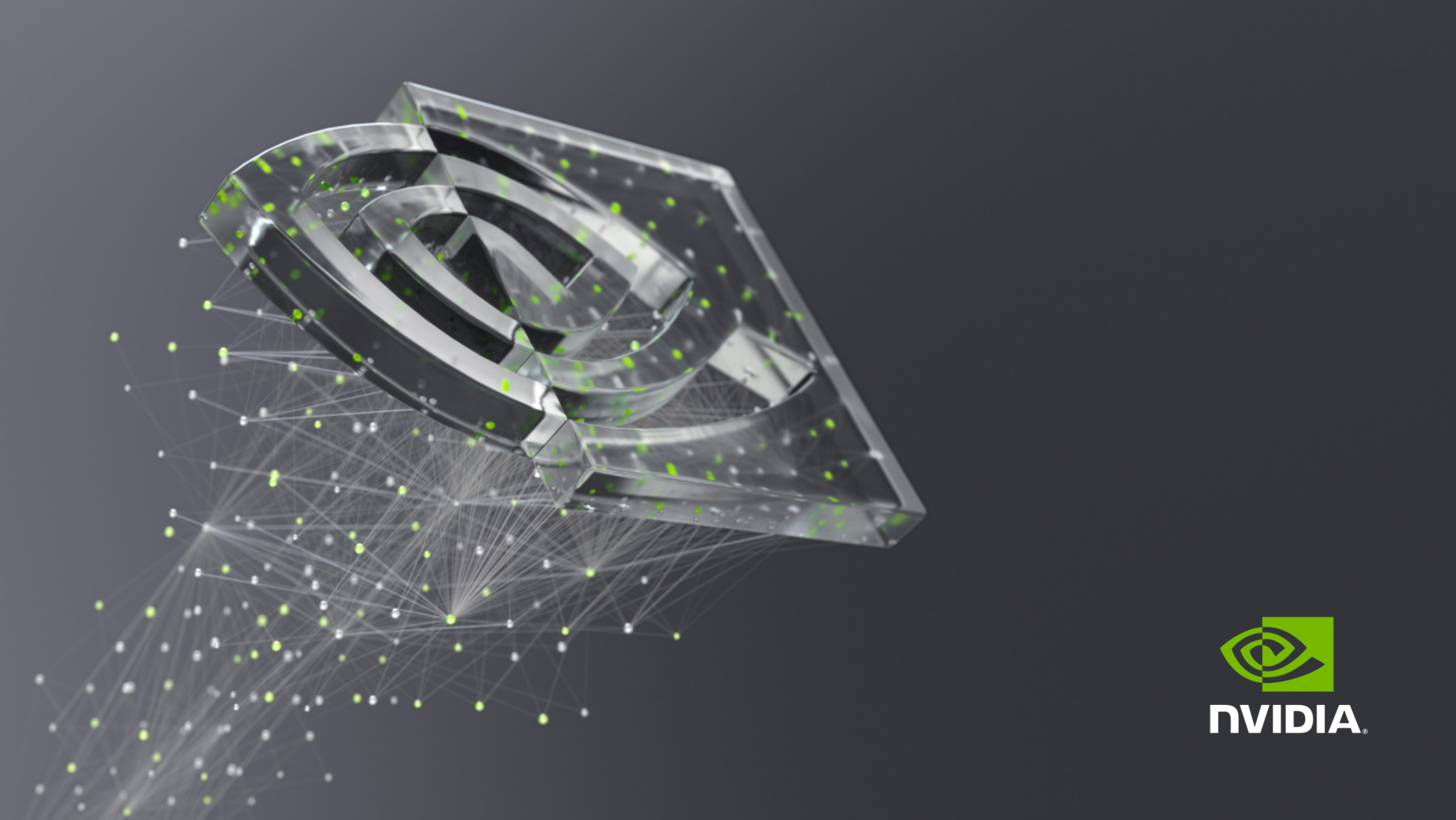
## Weak scaling

- Effective batch size increases: 64, 128, 256, 512 (per gpu: 8)
- Approx 98% parallel efficiency



# SUMMARY

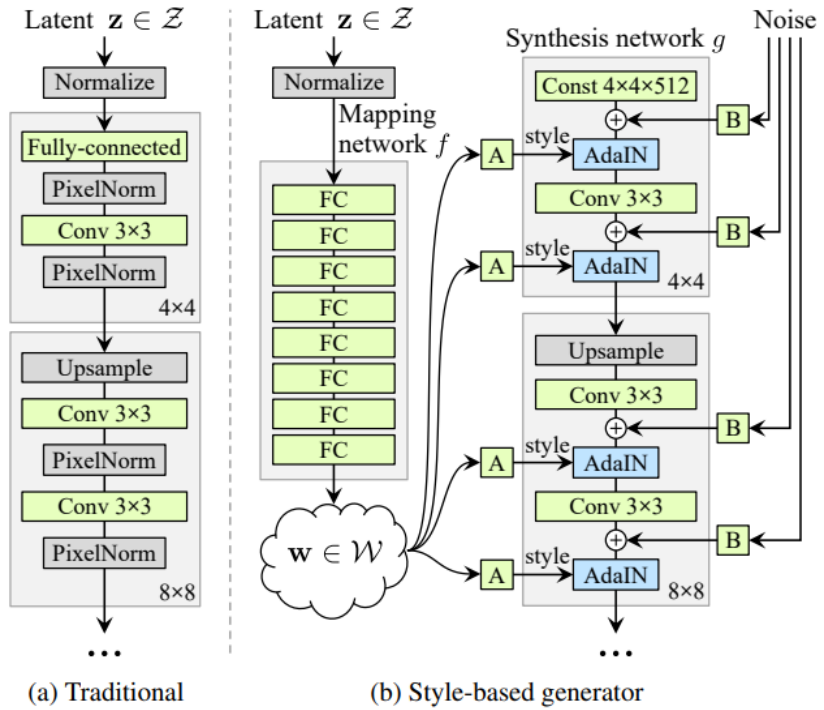
- We implemented multi-node training capability into StyleGAN2 via Horovod
- We demonstrated that the new implementation does not compromise the performance on a single node.
- We demonstrated that multi-node training does not compromise the accuracy of StyleGAN2 for a constant effective batch size.
- In a weak scaling sense (effective batch size constant), the implementation achieves 63% parallel efficiency up to 32 GPUs.
- In a strong scaling sense (batch size * the number of nodes), the implementation achieves 98% parallel efficiency up to 64 GPUs.


NVIDIA

# STYLEGAN2 FEATURES

## Style-based Generator Architecture (Karras et al. 2019)

- Map the input to an intermediate latent space W using an MLP
- This controls the generator through adaptive instance normalization
- AdaIN normalizes each channel to zero mean and unit variance, and then applies scales and biases based on the style
- Input only through styles! No latent input to the first conv layer
- The mapping network and affine transformations are a way to draw samples for each style from a learned distribution
- The synthesis network is a way to generate a novel image based on a collection of styles.




(a) Traditional (b) Style-based generator

$$\mathrm{AdaIN}(\mathbf{x}_i, \mathbf{y}) = \mathbf{y}_{s,i} \frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)} + \mathbf{y}_{b,i}$$