Performance characterization of GANs and multi-GANs

Submitted by – Balavignesh Vemparala Narayana Murthy

Contact : vemparalanarayanamurthy.1@osu.edu, +1 6142861705

I. Introduction/Motivation

GANs, short form for Generative Adversarial Networks, is an approach to generative modeling/design using Deep Learning techniques such as CNNs [1]. GANs is a supervised leaning task, which involves automatically learning features/patterns from a given input training data, such that the model is then able to generate new samples which are similar to the input training samples. GANs have two sub-models, known as generator and discriminator. While the generator learns the patterns/features of the input training data in order to generate new examples, discriminator tries to classify examples as either real or fake. Thus, the two models are trained together such that the generator learns to generate new examples, which the Discriminator is no longer able to judge whether or not is real or fake.

However, the issue with GANs is often that it can take a really long time to generate meaningful nice-looking outputs. Here comes the need for distributed training, which can cut down the time drastically. In distributed training, as the name suggests, the workload to train a model is split up and shared among multiple workers, which then work in parallel to speed up model training.

There are two main types of distributed training [2]: data parallelism and model parallelism. In Data parallelism, the data is divided into partitions, where the number of partitions is the total number of available nodes/GPUs. The model is then copied into each of these worker nodes, with each worker operating on its own part of the data. Each node then also computes the errors between its predictions and the labels, which are then used in the weight updates. Following this, the worker nodes need to synchronize the model parameters, at the end of a batch computation to ensure that they are training a consistent model.

On the other hand, in model parallelism, the model is segmented into different parts that can run concurrently in different nodes, and each one will run on the same data. In this type of parallelism, the worker nodes only need to synchronize the shared parameters, and larger models aren't a concern as each node operates on a sub-section of the model.

II. Problem Statement & methods

The goal in the current project is the performance characterization of GANs based architectures – namely, DCGAN (Deep Convolutional GAN), SNGAN (Self-Normalization GAN), WGAN-GP (Wasserstein GAN with Gradient Penalty), specifically on distributed training. That is, we'd like to use different distributed training strategies to see how the training time can be improved in GANs training. Tensorflow & Horovod compiled with MVAPICH-GDR have been used for distributed training tasks. Note that, with WGAN-GP, the distributed version of the model wasn't working with some undefined behavior and hence, experiments using this model were restricted to single-node experiments.

Also, the Celeb-A Celebrity Faces dataset was used for all experiments.

III. GAN Models

Deep Convolutional GAN

Deep Convolutional Generative Adversarial Networks (DCGANs) architecture [3,4] was introduced by Radford et. al. in 2015 [3], in which the discriminator (D) is made up of strided convolution layers, batch normalization layers and leakyReLU activations, whereas the generator (G) is made up of convolutional-transpose layers, batch normalization layers and ReLU activations.

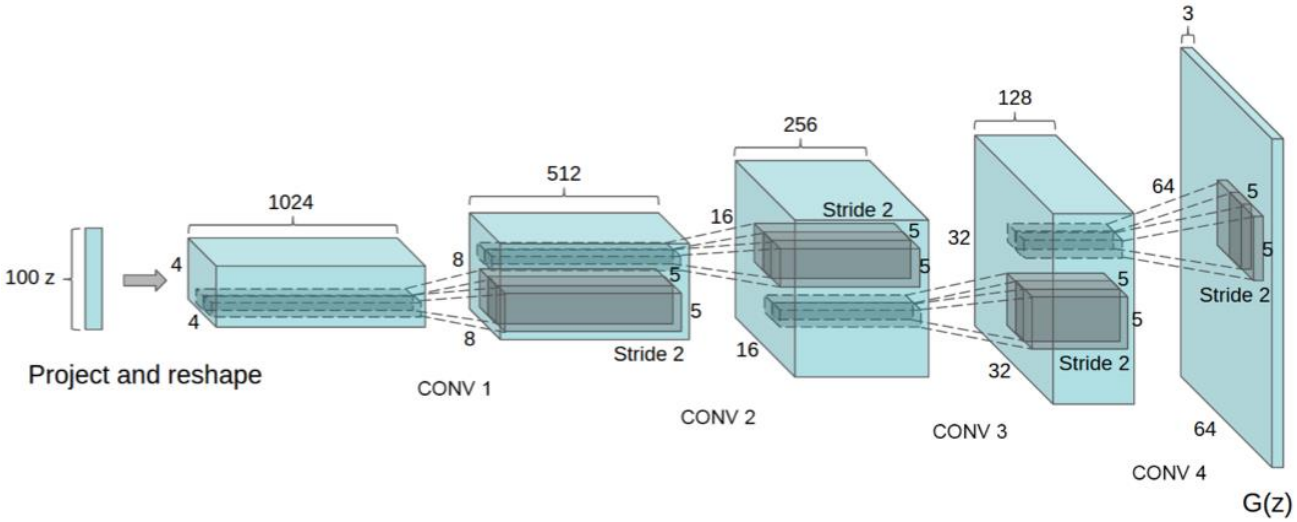


Fig 1 : DCGAN Generator used in this study [3]

The goal is to train the discriminator to maximize the probability of correctly classifying real and fake images, whereas we train the generator to classify realistic fakes, i.e., minimize $\log(1 - D(G(z)))$. In other words, D & G play a two-player minimax game where each one is trying to outsmart each other –

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

In other words, the GAN training aims to achieve a Nash equilibrium when the discriminator is no longer able to distinguish between real and fake samples, that is, given an image I, $P(real) = P(fake) = 0.5$. However, in reality, training to this point is very difficult to achieve and takes a very long time to produce realistic fake samples.

Spectral Normalization GAN

Miyato et. al. (2018) [5] proposed a novel weight normalization technique called spectral normalization to stabilize the training of the discriminator D. In this study, a standard DCGAN architecture was used with Spectral normalization applied in both Generator (G) & Discriminator (D), instead of batch normalization. This implementation of SNGAN used more or less the same architecture as DCGAN to get an apples to apples comparison.

Wasserstein GAN with Gradient Penalty

Arjousky et. al. (2017) [6] proposed an alternative way of training a generator in which, rather than using a discriminator to predict the probability of generated images being real or fake, they introduced a critic which tries to minimize the "earth mover" distance between generated images and the real dataset.

$$\min_G \max_{D\in\mathcal{D}} \mathop{\mathbb{E}}_{\boldsymbol{x}\sim\mathbb{P}_r}\left[D(\boldsymbol{x})\right] - \mathop{\mathbb{E}}_{\tilde{\boldsymbol{x}}\sim\mathbb{P}_g}\left[D(\tilde{\boldsymbol{x}})\right]$$

The issue with the original WGAN model is that it uses a weight clipping method to enforce Lipschitz constraint. It has been shown that such a method affects the learning process, and hence, Gulrajani et. al. (2017) [7] came up with a different way to enforce this constraint called the gradient penalty approach. In this, instead of "hard" clipping, a regularization term is added to the above Wasserstein Loss function, which is a form of "soft" clipping and the loss function takes the following form –

$$L = \underbrace{\mathop{\mathbb{E}}_{\tilde{\boldsymbol{x}}\sim\mathbb{P}_g}[D(\tilde{\boldsymbol{x}})] - \mathop{\mathbb{E}}_{\boldsymbol{x}\sim\mathbb{P}_r}[D(\boldsymbol{x})]}_{\text{Original critic loss}} + \underbrace{\lambda \mathop{\mathbb{E}}_{\hat{\boldsymbol{x}}\sim\mathbb{P}_{\hat{\boldsymbol{x}}}}\left[(\|\nabla_{\hat{\boldsymbol{x}}} D(\hat{\boldsymbol{x}})\|_2 - 1)^2\right]}_{\text{Our gradient penalty}}.$$

Another feature of WGAN & WGAN-GP is that, the critic (or the discriminator) network is trained 5 times for one training of a generator. This is why this model was an interesting choice for the performance characterization.

In this study, again the DCGAN generator and discriminator network was used in WGAN-GP, however, the batch normalization was replaced with layer normalization (as suggested in the WGAN-GP paper).

Dataset

The Celeb-A Faces dataset containing 202,599 face images with 10,177 unique identities was used for the experiments in the study. The cropped and aligned version of this dataset was used [8].

Single-node training

Experiments were run on single CPU and single GPU using the Celeb-A dataset. The dataset was stored as a "pickle" file which was then read back in the code. Also, note that all experiments on throughput analysis were run only for 5 epochs. And as we know, this number of epochs is definitely not a good indicator of which model is better with respect to accuracy as GAN training requires a lot of time to start giving good results.

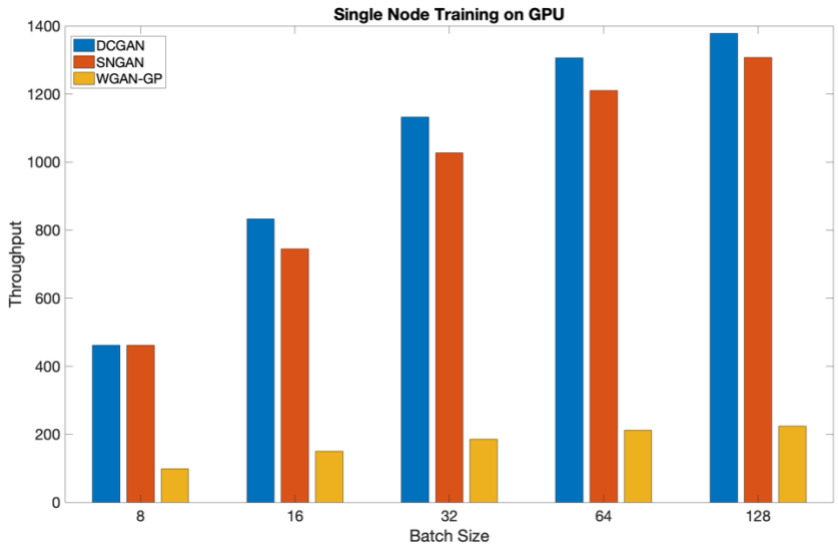
DCGAN: Effect of batch size on throughput in single-node training				
CPU/GPU	Batch size	Avg. per epoch time	Throughput	Total Fit time
CPU	8	1449.40	141.03	7247.00
CPU	16	1176.28	172.24	5881.39
CPU	32	1247.90	162.35	6239.51
CPU	64	996.38	203.33	4981.92
CPU	128	962.12	210.58	4810.59
GPU	8	416.86	486.02	2084.28
GPU	16	243.35	832.54	1216.75
GPU	32	178.89	1132.55	894.44
GPU	64	155.10	1306.22	775.52
GPU	128	147.00	1378.27	735.12

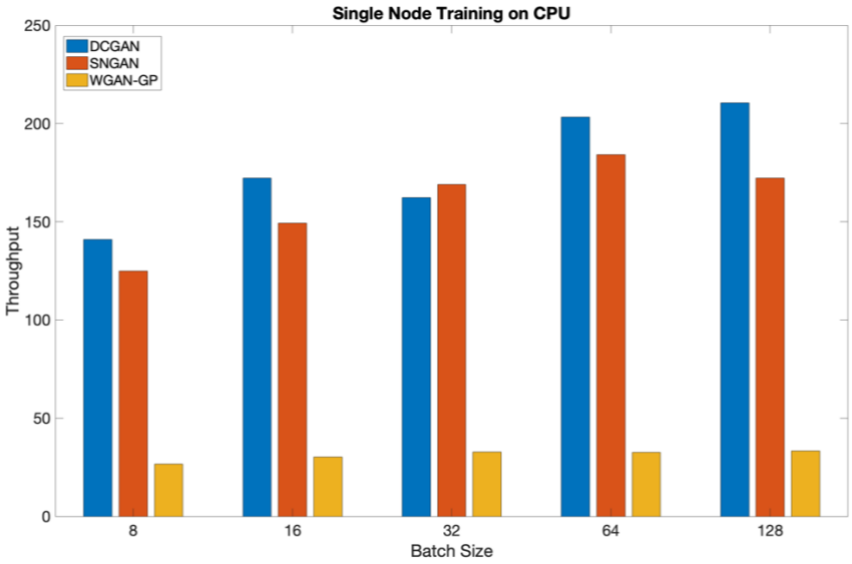
SNGAN: Effect of batch size on throughput in single-node training				
CPU/GPU	Batch size	Avg. per epoch time	Throughput	Total Fit time
CPU	8	1622.82	124.84	8114.14
CPU	16	1357.12	149.29	6785.60
CPU	32	1199.39	168.92	5996.95
CPU	64	1099.72	184.23	5498.61
CPU	128	1176.49	172.21	5882.43

GPU	8	438.45	462.08	2192.26
GPU	16	271.96	744.96	1359.79
GPU	32	197.31	1026.80	986.55
GPU	64	167.34	1210.69	836.71
GPU	128	154.91	1307.84	774.73

WGAN-GP: Effect of batch size on throughput in single-node training				
CPU/GPU	Batch size	Avg. per epoch time	throughput	Total Fit time
CPU	8	7607.11	26.63	38035.84
CPU	16	6678.54	30.33	33392.94
CPU	32	6178.90	32.79	30894.78
CPU	64	6205.05	32.65	31025.52
CPU	128	6094.41	33.24	30472.32
GPU	8	2057.42	98.47	10287.47
GPU	16	1352.68	149.78	6763.78
GPU	32	1089.19	186.01	5446.29
GPU	64	957.13	211.67	4786.16
GPU	128	904.87	223.90	4525.09

As it can be seen from the tables above, DCGAN has the highest throughput for a given batch size on both CPU and GPU, and also, the batch size of 128 seems to be the best in terms of throughput for all the models





As it can be observed from above plots, DCGAN is best in terms of throughput for most cases, closely followed by SNGAN. As expected, WGAN-GP has the lowest throughput owing to 5 iterations of discriminator (critic) training per each iteration of generator training, which slows down the model significantly

Issues with Distributed implementation of WGAN-GP on Horovod & Tensorflow Distributed

However, as mentioned previously, I was unable successfully come up a distributed implementation of WGAN-GP with Horovod and here's few issues I faced listed below –

1. **Missing ranks:**
   0:
   [DistributedGradientTape_Allreduce_1/cond_9/then/_88/DistributedGradientTape_Allreduce_1/cond_9/HorovodAllreduce_AddN_19_0]
   1:
   [DistributedGradientTape_Allreduce/cond/then/_0/DistributedGradientTape_Allreduce/cond/HorovodAllreduce_gradient_tape_discriminator_zero_padding2d_Slice_1_0,
   DistributedGradientTape_Allreduce_1/cond_1/then/_24/DistributedGradientTape_Allreduce_1/cond_1/HorovodAllreduce_AddN_11_0,
   DistributedGradientTape_Allreduce_1/cond_3/then/_40/DistributedGradientTape_Allreduce_1/cond_3/HorovodAllreduce_AddN_13_0,
   DistributedGradientTape_Allreduce_1/cond_5/then/_56/DistributedGradientTape_Allreduce_1/cond_5/HorovodAllreduce_AddN_15_0,
   DistributedGradientTape_Allreduce_1/cond_7/then/_72/DistributedGradientTape_Allreduce_1/cond_7/HorovodAllreduce_AddN_17_0]
   [2021-12-14 18:30:54.364797: W /tmp/pip-install-6fa67cu9/horovod_4df5b363cf0949c3bf6bfe0b5ce4d489/horovod/common/stall_inspector.cc:105] One or more tensors were submitted to be reduced, gathered or broadcasted by subset of ranks and are waiting for remainder of ranks for more than 60 seconds. This may indicate that different ranks are trying to submit different tensors or that only subset of ranks is submitting tensors, which will cause deadlock.
2. "Nan" loss in the distributed training version inspite of the serial implementation working just fine

Due to this, I also tried implementing a Tensorflow Distributed Implementation. Even though I was able to get a working implementation, this version gave me meaningless results. For instance, the distributed DCGAN implementation of Tensorflow Distributed took more and more fit time, as the number of GPU nodes increased, which doesn't make sense. I suspect that, there is some issue in my code with splitting the initial training data into different shards. Also, since there is no way to inspect the "tf.distribute.DistributedDataset" class in tensorflow, I was unable to find the reason for these results. I also tried to see if there is a way to get the rank of my current GPU to use my pre-sharded ".pickle" files, similar to my Horovod code, however, I couldn't find a way to get this to work either.

Hence, my distributed training experiments were limited to DCGAN & SNGAN in Tensorflow & Horovod.

I have however attached my implementation of WGAN-GP on Tensorflow Distributed – I'd really appreciate if you can provide any feedback regarding this. Moreover, I'd also appreciate any help with the Horovod errors with WGAN-GP.

Distributed Training – Weak vs Strong scaling

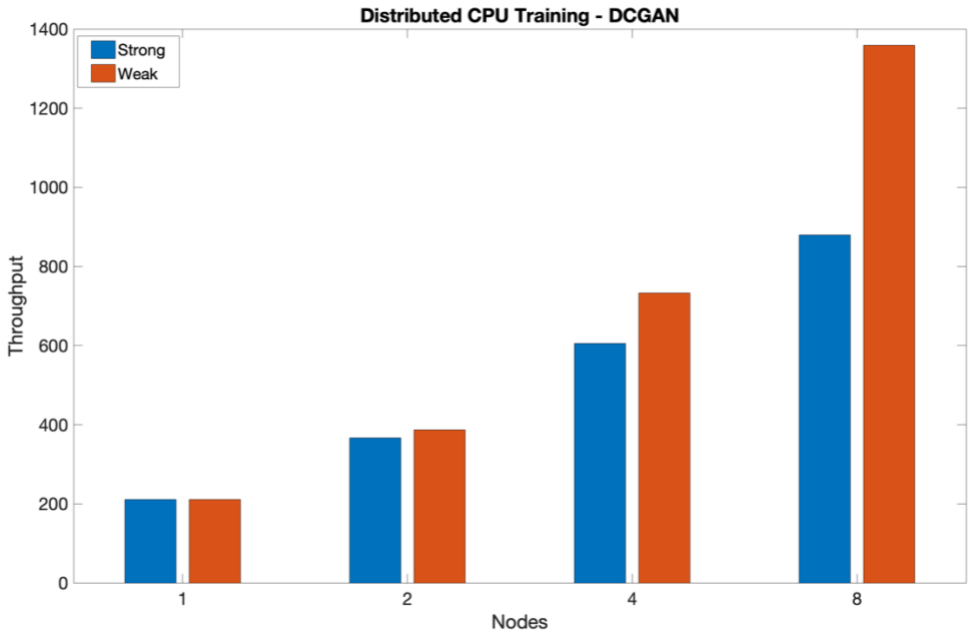
DCGAN

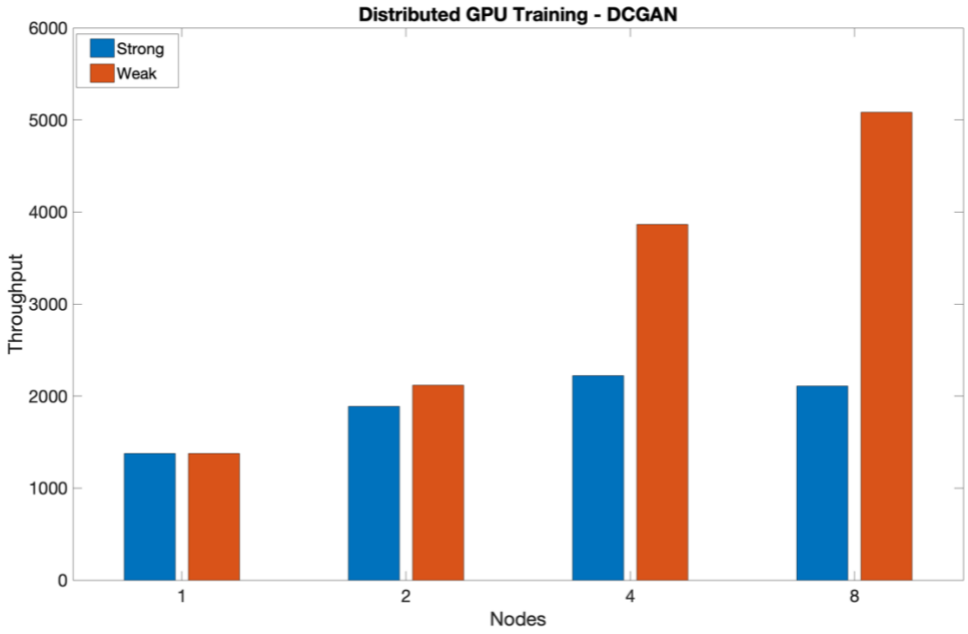
CPU (WITH BINDING) and GPU runs

Weak/ Strong scaling	Nodes	Global Batch Size	Batch Size/ Node	CPU/ GPU	One Epoch run time (secs)	**Images/ sec per epoch**	**Fit time**
Best Batch Size = ThrPut=202599/one-epoch time				CPU	MPI-MVAPICH2-GDR		
S	1	128	128	CPU	962.12	210.58	4810.59
S	2	128	64	CPU	552.71	366.55	2772.97
S	4	128	32	CPU	334.54	605.61	1687.65
S	8	128	16	CPU	230.23	879.97	1182.34
W	1	128	128	CPU	962.12	210.58	4810.59
W	2	128	128	CPU	523.09	387.31	2622.07
W	4	128	128	CPU	276.58	732.50	1388.14
W	8	128	128	CPU	149.04	1359.39	770.97

Weak/ Strong scaling	Nodes	Global Batch Size	Batch Size/ Node	CPU/ GPU	One Epoch run time (secs)	**Images/ sec per epoch**	**Fit time**
Best Batch Size = ThrPut=202599/one-epoch time				GPU	MPI-MVAPICH2-GDR		
S	1	128	128	GPU	147.00	1378.27	735.12
S	2	128	64	GPU	107.20	1889.89	568.05

S	4	128	32	GPU	91.10	2223.86	480.29
S	8	128	16	GPU	95.98	2110.83	521.15
W	1	128	128	GPU	147.00	1378.27	735.12
W	2	128	128	GPU	95.70	2117.06	515.81
W	4	128	128	GPU	52.40	3866.57	274.11
W	8	128	128	GPU	39.87	5081.34	210.71





As it can be seen from the above plots, the weak/strong scaling results on DCGAN show the expected trend with weak scaling generating more throughput with distributed training on the same number of nodes, compared to strong scaling. However, in the case of strong scaling on GPU, throughput seems to decrease

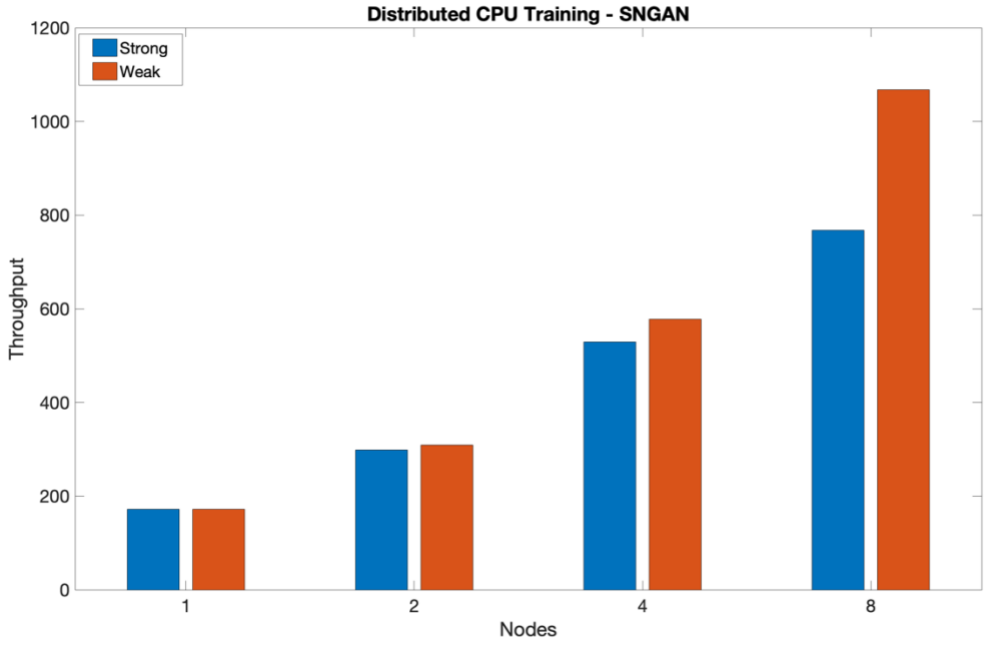
with increase in the number of GPU nodes from 4 to 8. This could be because, the overhead due to communication is more compared to the resulting speed-up, owing to DCGAN being a simple enough model

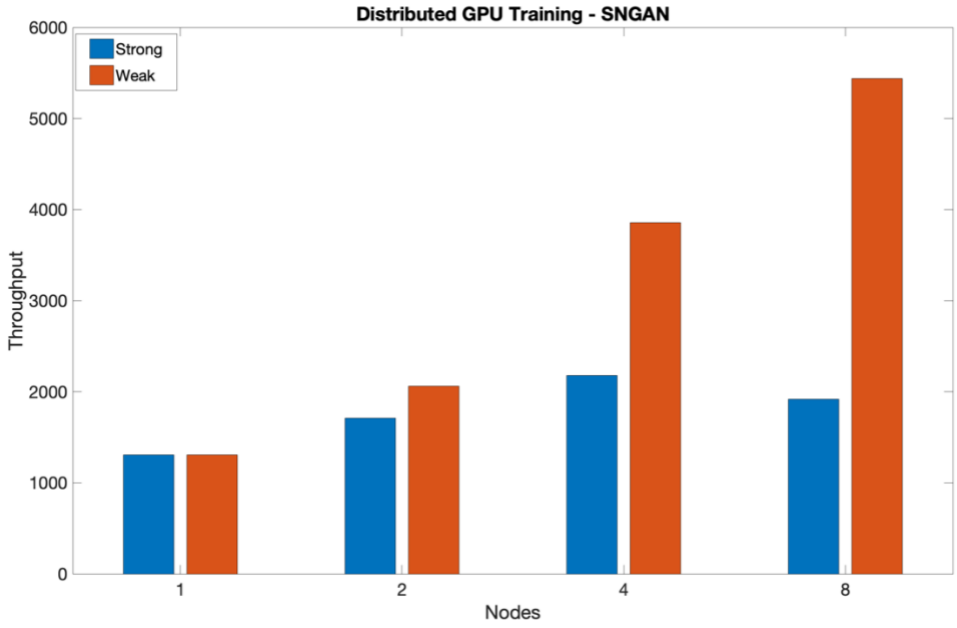
SNGAN

CPU (WITH BINDING) and GPU runs

Weak/ Strong scaling	Nodes	Global Batch Size	Batch Size/ Node	CPU/ GPU	One Epoch run time (secs)	**Images/ sec per epoch**	**Fit time**
Best Batch Size = ThrPut=40000/one-epoch time				CPU	MPI-MVAPICH2-GDR		
S	1	128	128	CPU	1176.49	172.21	5882.43
S	2	128	64	CPU	679.19	298.30	3400.69
S	4	128	32	CPU	382.65	529.46	1923.79
S	8	128	16	CPU	263.89	767.73	1336.80
W	1	128	128	CPU	1176.49	172.21	5882.43
W	2	128	128	CPU	656.01	308.83	3282.26
W	4	128	128	CPU	350.33	578.30	1767.36
W	8	128	128	CPU	189.73	1067.85	957.00

Weak/ Strong scaling	Nodes	Global Batch Size	Batch Size/ Node	CPU/ GPU	One Epoch run time (secs)	**Images/ sec per epoch**	**Fit time**
Best Batch Size = ThrPut=40000/one-epoch time				GPU	MPI-MVAPICH2-GDR		
S	1	128	128	GPU	154.91	1307.84	774.73
S	2	128	64	GPU	118.34	1712.05	628.02
S	4	128	32	GPU	92.92	2180.27	469.12
S	8	128	16	GPU	105.50	1920.40	588.80
W	1	128	128	GPU	154.91	1307.84	774.73
W	2	128	128	GPU	98.30	2061.09	518.93
W	4	128	128	GPU	52.53	3856.57	273.20
W	8	128	128	GPU	37.24	5439.93	235.84





As it can be seen from the above plots, the weak/strong scaling results on SNGAN show the expected trend with weak scaling generating more throughput with distributed training on the same number of nodes, compared to strong scaling. However, in the case of strong scaling on GPU, similar to DCGAN, throughput seems to decrease with increase in the number of GPU nodes from 4 to 8. Again, this could be because, the overhead due to communication is more compared to the resulting speed-up, as these models don't have a large number of parameters

Model Parameters

DCGAN: Here's the model parameters of DCGAN shown below –

Model: "discriminator"

_____

Total params: 2,768,321
Trainable params: 2,766,529
Non-trainable params: 1,792

_____

Model: "generator"

_____

Total params: 3,825,475
Trainable params: 3,823,555
Non-trainable params: 1,920

SNGAN: Here's the model parameters of SNGAN shown below –

Model: "discriminator"

_____

Total params: 4,855,617
Trainable params: 4,854,977
Non-trainable params: 640

_____

Model: "generator"

_____

Total params: 3,823,235
Trainable params: 3,821,891
Non-trainable params: 1,344

As it can be seen above, SNGAN has a heavier discriminator (4.85E6 vs 2.77E6 parameters) compared to DCGAN. This could be attributed to the lower relative throughput in SNGAN experiments compared to DCGAN. Moreover, these models are much smaller compared to VGG which has 138E6 parameters, and hence, the distributed training using data parallelism (strong scaling) doesn't seem to scale up beyond 4 GPUs.

Computation/Communication Overlap in Weak/Strong scaling experiments on DCGAN & SNGAN

Following the Weak/Strong scaling experiments, the computation/communication overlap was studied on the weak/strong scaling experiments performed above

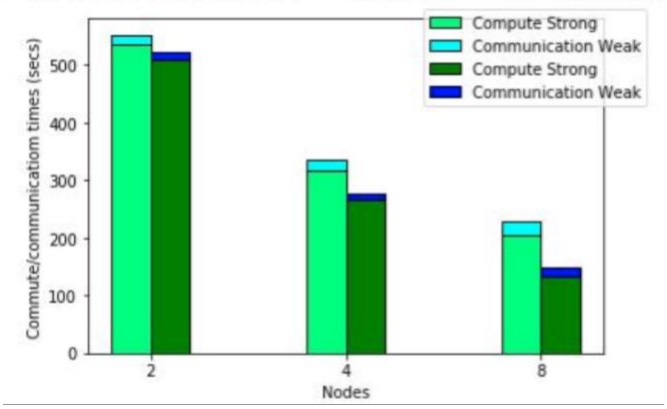
CPU

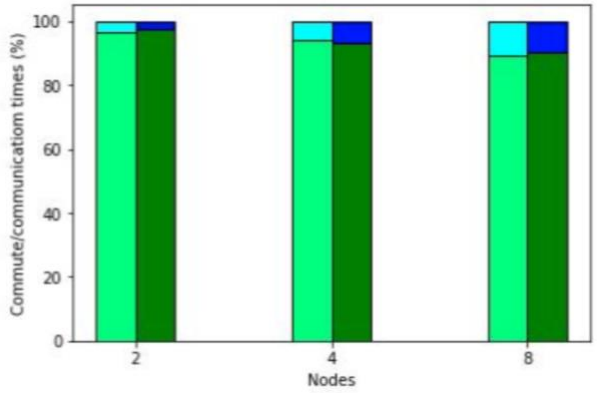
Nodes	Scaling	Model	Batch	Avg Epoch Time (secs)	Compute (secs)	Communication (secs)	Only Computation%	Only Communication%
2	S	DCGAN	64	552.71	536.92	15.79	97.14	2.86
2	W	DCGAN	128	523.09	510.31	12.78	97.56	2.44
4	S	DCGAN	32	334.54	316.74	17.80	94.68	5.32
4	W	DCGAN	128	276.58	265.23	11.35	95.90	4.10
8	S	DCGAN	16	230.23	206.25	23.98	89.58	10.42
8	W	DCGAN	128	149.04	134.21	14.83	90.05	9.95
2	S	SNGAN	64	552.71	533.88	18.83	96.59	3.41
2	W	SNGAN	128	523.09	508.58	14.51	97.23	2.77
4	S	SNGAN	32	334.54	314.56	19.98	94.03	5.97
4	W	SNGAN	128	276.58	258.10	18.48	93.32	6.68
8	S	SNGAN	16	230.23	205.29	24.94	89.17	10.83
8	W	SNGAN	128	149.04	134.73	14.31	90.40	9.60

GPU

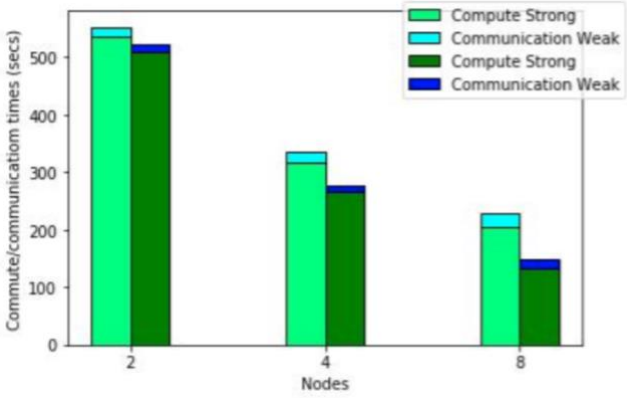
						OVERLAP						
Nodes	Scaling	Model	Batch size	Avg Epoch Time (secs)	Computation (secs)	WAIT FOR DATA	WAIT FOR OTHER TENSOR DATA	Overall Communication (secs)	Only Communication	Only Computation%	OVERLAP%	Only Communication%
2	S	DCGAN	64	107.2	90.56	0.958	0.086	16.64	15.596	84.4776119	0.973880 6	14.54850 75
2	W	DCGAN	128	95.7	76.03	1.17	0.043	19.67	18.457	79.446186	1.267502 61	19.28631 14
4	S	DCGAN	32	91.1	76.57	0.36	0.09	14.53	14.08	84.050494	0.493962 68	15.45554 34
4	W	DCGAN	128	52.4	42.42	0.46	0.022	9.98	9.498	80.954 1985	0.919847 33	18.12595 42
8	S	DCGAN	16	95.98	79.22	0.077	0.088	16.76	16.595	82.538 0288	0.171910 81	17.29006 04

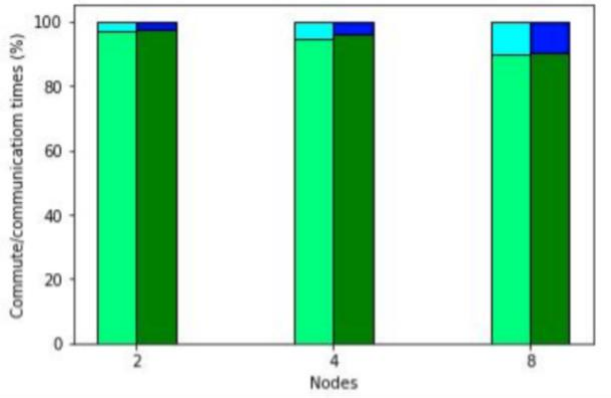
8	W	DCGAN	128	39.87	32.56	0.276	0.011	7.31	7.023	81.6654126	0.71983948	17.6147479
2	S	SNGAN	64	118.34	99.39	1.22	0.065	18.95	17.665	83.9868176	1.08585432	14.927328
2	W	SNGAN	128	98.3	82.15	1.77	0.04	16.15	14.34	83.5707019	1.84130214	14.5879959
4	S	SNGAN	32	92.92	87.05	0.57	0.066	5.87	5.234	93.6827378	0.68445975	5.63280241
4	W	SNGAN	128	52.53	44.76	0.62	0.02	7.77	7.13	85.2084523	1.21835142	13.5731963
8	S	SNGAN	16	105.5	84.13	0.23	0.065	21.37	21.075	79.7440758	0.27962085	19.9763033
8	W	SNGAN	128	37.24	17.11	0.35	0.01	20.13	19.77	45.9452202	0.96670247	53.0880773



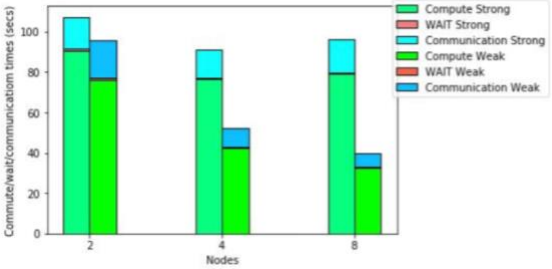


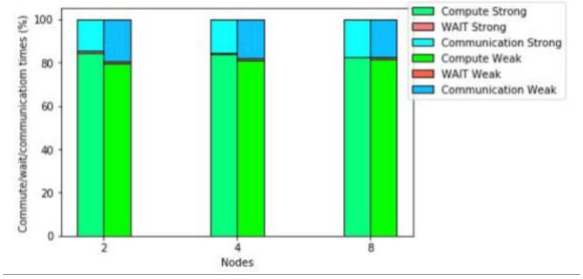
Communication/Computation Time in SNGAN in CPU





Communication/Computation Time in DCGAN in CPU





Communication/Computation Time in DCGAN in GPU

As it can be seen from the above graphs, as the number of nodes increases in distributed training, the communication time (%) increases in both DCGAN as well as SNGAN, however, computation time seems to take a large chunk. This could be because the models are relatively small, and probably the compute requirement is much higher compared to communication requirement.

However, I also wanted to test other performance enhancement techniques. Following Weak/strong scaling experiments, several other optimizations were tried upon, including **Tensor Fusion, FP16 compression, Mixed-Precision Training, and Hierarchical All-reduce**

I discuss the results using each of these optimizations below one by one –

Tensor Fusion Experiments using Horovod

According to Horovod website [9]**, "**Horovod is its ability to interleave communication and computation coupled with the ability to batch small **allreduce** operations, which results in improved performance. We call this batching feature Tensor Fusion.

Tensor Fusion works by attempting to combine all the tensors that are ready to be reduced at given moment of time into one reduction operation. "

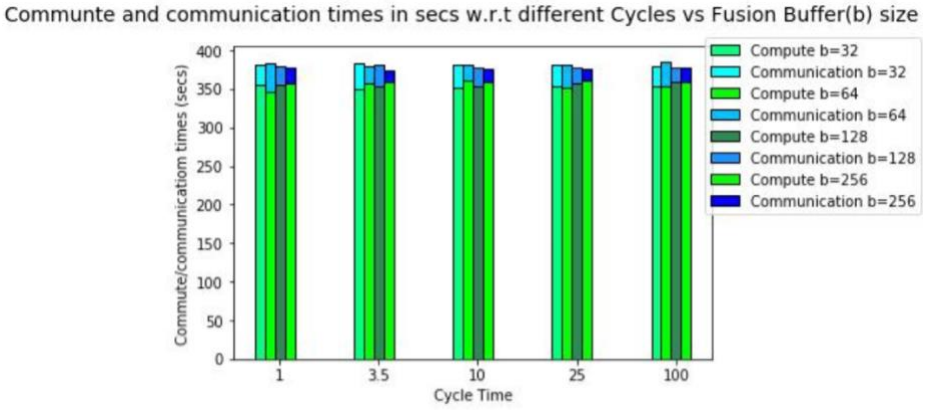
So, increasing the Fusion buffer leads to more tensors being reduced, eventually leading to speed-up theoretically.

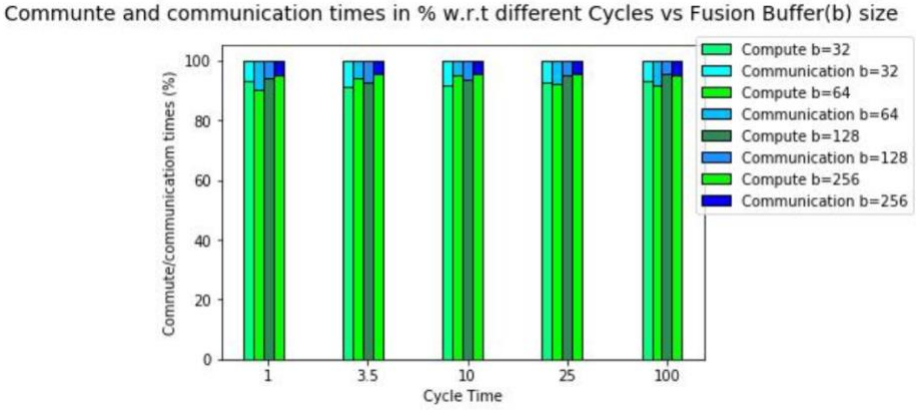
Tensor Fusion experiments were conducted both on CPU and GPU

CPU

Cycle Time	TF Buffer size	Avg Epoch Time (secs)	Compute (secs)	Communication (secs)	Only Computa-tion%	Only Communica-tion%
1	32	380.68	354.6	26.08	93.14910161	6.850898392
3.5	32	383.93	350.48	33.45	91.28747428	8.712525721
10	32	382.1	351.01	31.09	91.86338655	8.136613452
25	32	381.76	353.95	27.81	92.71531852	7.284681475
100	32	380	353.41	26.59	93.00263158	6.997368421
1	64	382.99	345.25	37.74	90.14595681	9.854043187
3.5	64	379.35	357.49	21.86	94.23751153	5.762488467
10	64	380.74	361.28	19.46	94.88890056	5.111099438
25	64	381.23	351.73	29.5	92.26188915	7.738110852

100	64	385.54	353.5	32.04	91.68957825	8.310421746
1	128	379.21	356.11	23.1	93.90838849	6.091611508
3.5	128	381.83	353.55	28.28	92.59356258	7.40643742
10	128	376.79	353.11	23.68	93.71533215	6.284667852
25	128	377.02	357.99	19.03	94.95252241	5.047477587
100	128	376.87	359.23	17.64	95.31934089	4.680659113
1	256	377.06	357.64	19.42	94.84962605	5.150373946
3.5	256	374.15	358.19	15.96	95.73433115	4.265668849
10	256	375.08	358.26	16.82	95.51562333	4.484376666
25	256	376.44	360.01	16.43	95.63542663	4.364573372
100	256	377.44	359.15	18.29	95.15419669	4.845803306

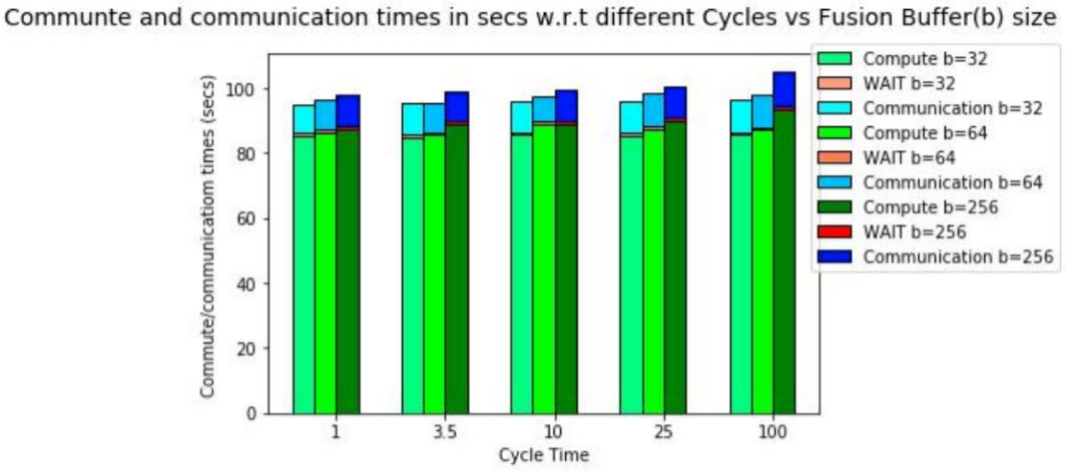


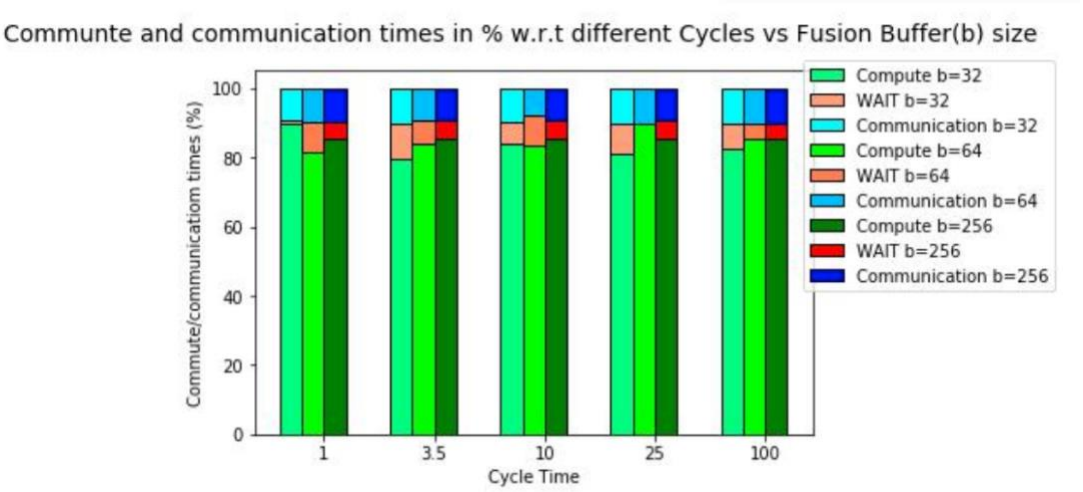


GPU

				OVERLAP						
Cycle time	TF buffer size	Avg Epoch Time (secs)	Computa-tion (secs)	WAIT FOR DATA	WAIT FOR OTHER TENSOR DATA	Overall Communica-tion (secs)	Only Communica-tion	Only Computa-tion%	OVERLAP%	Only Communica-tion%

1	32	95.07	85.35	0.726	0.087	9.72	8.907	89.7759546	0.8551593 6	9.3688860 8
3.5	32	95.64	85.01	0.729	0.087	10.63	9.814	79.446186	10.292417 1	10.261396 9
10	32	95.77	85.7	0.726	0.087	10.07	9.257	84.050494	6.2836398 6	9.6658661 4
25	32	96.19	85.48	0.725	0.0867	10.71	9.8983	80.9541985	8.7554386 8	10.290362 8
100	32	96.3	85.61	0.72	0.0865	10.69	9.8835	82.5380288	7.1987313 2	10.263239 9
1	64	96.39	86.31	0.728	0.0868	10.08	9.2652	81.6654126	8.7223869 6	9.6122004 4
3.5	64	95.35	85.687	0.749	0.087	9.663	8.827	83.9868176	6.7557099 3	9.2574724 7
10	64	97.33	88.8	0.73	0.087	8.53	7.713	83.5707019	8.5047116 4	7.9245864 6
25	64	98.45	87.361	0.731	0.087	11.089	10.271	93.6827378	-4.1154447 6	10.432707
100	64	98.04	87.26	0.732	0.086	10.78	9.962	85.2084523	4.6303889 9	10.161158 7
1	256	97.87	87.4	0.727	0.087	10.47	9.656	85.2084523	4.9253987 3	9.8661489 7
3.5	256	98.9	88.877	0.729	0.087	10.023	9.207	85.2084523	5.4821442 6	9.3094034 4
10	256	99.28	89.05	0.728	0.0875	10.23	9.4145	85.2084523	5.3087717 1	9.4827759 9
25	256	100.42	90.06	0.724	0.0867	10.36	9.5493	85.2084523	5.2821870 1	9.5093606 9
100	256	105.25	93.54	0.719	0.087	11.71	10.904	85.2084523	4.4314526 9	10.360095





<u>Tensor Fusion Summary</u>: As it can be observed from the Tensor Fusion experiments, there is no performance improvement from increasing the cycle time or fusion buffer in this case. Actually, the increase in fusion buffer to 256 and cycle time to 100 ms seem to be the worst of all. This could be probably because the models are relatively small, and probably the compute requirement is much higher compared to communication requirement. To summarize, the default fusion buffer of 64 and cycle time of 3.5 ms seem to perform well for these models (DCGAN & SNGAN)

FP 16 Compression / Mixed-Precision-Training

All jobs were run with 64 fusion buffer size and cycle time of 3.5 ms, and a batch size of 32, and esxperiments were only carried out on GPU as the NVIDIA webpage says the Mixed-Precision training is only optimized in Tensorflow for GPUs with compute capability of 7.0.

			OVERLAP						
Type	Avg Epoch Time (secs)	Computa-tion (secs)	WAIT FOR DATA	WAIT FOR OTHER TENSOR DATA	Overall Communica-tion (secs)	Only Communica-tion	Only Computa-tion%	OVERLAP%	Only Communica-tion%
No FP16/ MP	112.61	85.71	0.718	0.087	26.9	26.095	76.1122458	0.714856585	23.17289761
FP16	177.16	150.47	0.729	0.087	26.69	25.874	84.9345222	0.460600587	14.60487695
MP	191.17	165.19	0.459	0.093	25.98	25.428	86.4100157	0.288748235	13.3012502

<u>FP16/Mixed-Precision Training summary</u>: As it can be seen from the results, there is no improvement resulting from use of either FP16 compression or Mixed-Precision training in our models

Hierarchical All-reduce

All jobs were run on 4 nodes with a batch size of 32 per node and a cycle time of 3.5 ms

				OVERLAP						
TF buffer size	Hierarchical ?	Avg Epoch Time (secs)	Computa-tion (secs)	WAIT FOR DATA	WAIT FOR OTHER TENSOR DATA	Overall Communica-tion (secs)	Only Communica-tion	Only Computa-tion%	OVERLAP%	Only Communica-tion%
32	Yes	112.94	87.69	0.719	0.0867	25.25	24.4443	77.6429963	0.71338764	21.6436161
32	No	100.53	69.54	0.733	0.086	30.99	30.171	69.1733811	0.81468218	30.0119367
64	Yes	116.33	89.21	0.713	0.0865	27.12	26.3205	76.6870111	0.68726898	22.6257199
64	No	103.56	95.776	0.699	0.0868	7.784	6.9982	92.4835844	0.75878718	6.75762843

<u>Summary</u>: Again, using Hierarchical All-reduce, there is no performance improvement in our models. Actually, the use of this feature worsens the model performance. Again, this could be because the models are simple and have lesser number of parameters, and hence the compute requirements are much higher rather than communication requirements.

Finally, let us look at some fake images generated by DCGAN, SNGAN & WGAN-GP models on single-node training. These are thee fakes generated at the end of 25 epochs. As it can be seen below, the images generated by SNGAN seem to be best among the ones presented below. However, such as statement cannot be made without a thorough investigation on convergence. Moreover, WGAN-GP model could take a lot more time to converge as the loss function computes a difference between real and fake samples, and uses this to update the gradients.

DCGAN

SNGAN

WGAN-GP

REFERENCES

1. https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/
2. https://docs.microsoft.com/en-us/azure/machine-learning/concept-distributed-training
3. https://arxiv.org/abs/1511.06434
4. https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html
5. https://arxiv.org/abs/1802.05957
6. https://arxiv.org/abs/1701.07875
7. https://arxiv.org/abs/1704.00028
8. http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html
9. https://horovod.readthedocs.io/en/stable/tensor-fusion_include.html