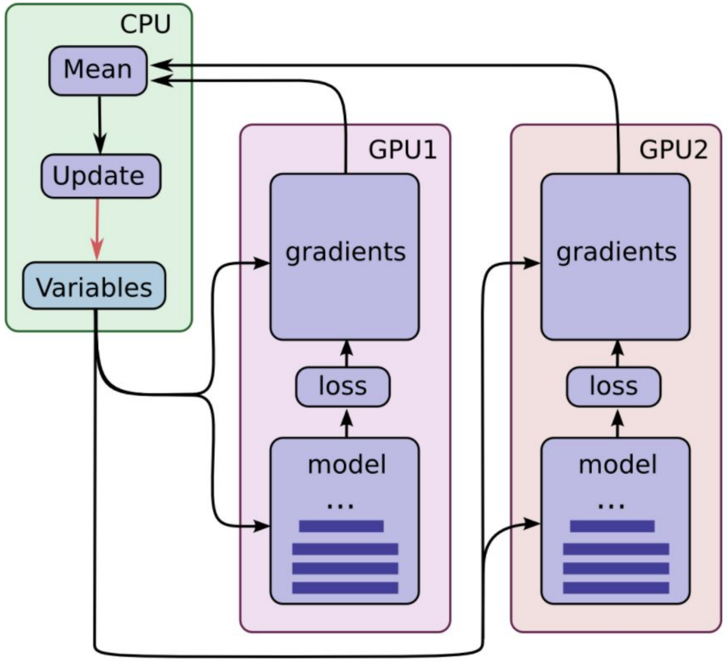
Multi-GPU Training on Single Node

Speaker, Date

Tensorflow

Single Node
Multi-GPU: Vanilla



Assign the same model to each device

- Reuse variables after building 1st graph
- Compute gradients on GPUs
- Compute average gradients and update variables on CPU (or one of the GPUs)

```
with tf.device('/cpu:0'):
    ...
    reuse_vars = False
    # Graphs for each GPU
    for i in range(n_gpu):
        with tf.device(assign_to_device('/gpu:{}'.format(i), ps_device='/cpu:0')):
            # split data between GPUs
            ...
            _pred, pred = CNN(reuse_vars, _x)
            ...
            # opt
            op = tf.train.AdamOptimizer(l_r)
            grads = op.compute_gradients(loss)
            tower_grads.append(grads)
            reuse_vars = True
    tower_grads = average_gradients(tower_grads)
    update_step = op.apply_gradients(tower_grads)
```

Image source: https://jhui.github.io/2017/03/07/TensorFlow-GPU/

Tensorflow

Single Node Multi-GPU: Horovod

https://github.com/uber/horovod

Functions

hvd.init(): Initialize the framework

hvd.DistributedOptimizer(): Opt.

hvd.size(): no. of GPUs

hvd.local_rank(): index on each GPU

Terminal command

```
$ mpirun --allow-run-as-root -np [no. of GPUs] -H
localhost:[no. of GPUs] python [python script]
```

Inside python scripts

Import the Horovod library for Tensorflow

```
import horovod.tensorflow as hvd
```

Horovod initialization

```
hvd.init()
```

Modify the learning rate (Optional)

```
learning_rate = learning_rate*hvd.size()
```

Define distributed optimizer

```
opt = tf.train.AdamOptimizer(learning_rate)
opt = hvd.DistributedOptimizer(opt)
```

Set visible device for each thread before training:

```
config = tf.ConfigProto().
config.gpu_options.allow_growth = True
config.gpu_options.visible_device_list = \
                    str(hvd.local_rank())
sess = tf.Session(config=config)
```

Keras with Multi-GPUs

Vanilla API - single line of code

Keras Multi GPUs

In [1]:
```
import tensorflow as tf
from keras.applications import Xception
from keras.utils import multi_gpu_model
import numpy as np

num_samples = 1000
height = 224
width = 224
num_classes = 1000
```

```
Using TensorFlow backend.
```

Instantiate the base model (or "template" model).

- We recommend doing this with under a CPU device scope, so that the model's weights are hosted on CPU memory. Otherwise they may end up hosted on a GPU, which would complicate weight sharing.

In [2]:
```
with tf.device('/cpu:0'):
    model = Xception(weights=None,
                     input_shape=(height, width, 3),
                     classes=num_classes)
```

Replicates the model on 8 GPUs.

- This assumes that your machine has 8 available GPUs.
- Training models with weights merge on GPU (recommended for NV-link)

In [3]:
```
try:
    parallel_model = multi_gpu_model(model, gpus=8, cpu_merge=False)
    print("Training using multiple GPUs..")
except:
    parallel_model = model
    print("Training using single GPU or CPU..")

parallel_model.compile(loss='categorical_crossentropy',
                       optimizer='rmsprop')
# Generate dummy data.
x = np.random.random((num_samples, height, width, 3))
y = np.random.random((num_samples, num_classes))

# This `fit` call will be distributed on 8 GPUs.
# Since the batch size is 256, each GPU will process 32 samples.
parallel_model.fit(x, y, epochs=20, batch_size=256)

# Save model via the template model (which shares the same weights):
model.save('my_model.h5')
```

Keras with Multi-GPUs

Horovod API - 10 lines of code

Keras Multi GPUs with Horovod

In [1]:
```
import keras
from keras.applications import Xception
from keras import backend as K
import numpy as np
import math
import tensorflow as tf
import horovod.keras as hvd

num_samples = 1000
height = 224
width = 224
num_classes = 100
```

```
Using TensorFlow backend.
```

Horovod

0. Initialize Horovod, and pin GPU to be used to process local rank (one GPU per process).
1. Adjust number of epochs based on number of GPUs.
2. Adjust learning rate based on number of GPUs.
3. Add Horovod Distributed Optimizer.
4. Callbacks API:
    - broadcast initial variable states from rank 0 to all other processes.
    - This is necessary to ensure consistent initialization of all workers when training is started with random weights or restored from a checkpoint.
5. Save checkpoints only on worker 0 to prevent other workers from corrupting them.

In [2]:
```
hvd.init()
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
config.gpu_options.visible_device_list = str(hvd.local_rank())
K.set_session(tf.Session(config=config))
epochs = int(math.ceil(20.0 / hvd.size()))
opt = keras.optimizers.Adadelta(1.0 * hvd.size())
opt = hvd.DistributedOptimizer(opt)
callbacks = [hvd.callbacks.BroadcastGlobalVariablesCallback(0)]
if hvd.rank() == 0:
    callbacks.append(keras.callbacks.ModelCheckpoint('./checkpoint-{epoch}.h5'))
```

As usual usage in Keras

In [3]:
```
# Generate dummy data.
x = np.random.random((num_samples, height, width, 3))
y = np.random.random((num_samples, num_classes))

with tf.device('/cpu:0'):
    model = Xception(weights=None,
                     input_shape=(height, width, 3),
                     classes=num_classes)

model.compile(loss='categorical_crossentropy',
              optimizer=opt)
model.fit(x, y,
          batch_size=256,
          callbacks=callbacks,
          epochs=epochs,
          verbose=1)
```

PyTorch

Multi-GPUs (single-node) - Vanilla

- PyTorch supports multi-GPUs configuration officially
- Warp your model with `torch.nn.DataParallel`

```
…
model = SOME_MODEL().cuda()
# Modification for multi-gpus
# * device_ids determine the GPUs used for training
# * if not given, default use all GPUs
model = torch.nn.DataParallel (model,
device_ids=[0,1])
# End modification
…
```

This single line of code will take care everything for:

- Copy model to different GPUs (Data-Parallel scheme)
- Sample different batch for multi-GPUs
- Gradient averaging

PyTorch

Multi-GPUs (single-node) - Horovod

In order to run the multi-GPUs in Horovod, The first step is to import the corresponding package

```
import horovod.torch as hvd
```

Notice, we only need to do testing/validation on a single GPU/CPU:

```
def metric_average(val, name):
      tensor = torch.tensor(val)
      avg_tensor = hvd.allreduce(tensor, name=name)
      return avg_tensor.item()
…
test_loss = metric_average(test_loss, 'avg_loss')
test_accuracy = metric_average(test_accuracy,
'avg_accuracy')
…

if hvd.rank() == 0:
    print(test_loss, test_accuracy)
```

1. Initial the configuration

```
hvd.init()
if use_cuda:
      # Horovod: pin GPU to local rank.
      torch.cuda.set_device(hvd.local_rank())
```

2. Create sampler for batch dispatch

```
train_sampler = torch.utils.data.distributed.DistributedSampler(
      train_dataset, num_replicas=hvd.size(), rank=hvd.rank())
# Do the same thing to test/validation dataset
```

3. Distributed the models

```
hvd.broadcast_parameters(model.state_dict(), root_rank=0)
# Horovod: wrap optimizer with DistributedOptimizer.
optimizer = hvd.DistributedOptimizer(optimizer,
            named_parameters=model.named_parameters())
```

4. Set the sampler at each epoch start

```
model.train()
train_sampler.set_epoch(epoch)
```

5. To invoke training script in terminal (example)

```
$ mpirun --allow-run-as-root -np 4 python main_multi_horovod.py
```

PyTorch

Multi-GPUs (single-node) - Apex

Make sure you've imported the apex distributed module

```
from apex.parallel import DistributedDataParallel as DDP
```

Apex used command-line argument for passing the GPU rank, expose `--local_rank` as API is needed

```
parser.add_argument("--local_rank", default=0, type=int)
```

1. Initial the configuration

```
torch.cuda.set_device(args.local_rank)
torch.distributed.init_process_group(backend='nccl',
                                     init_method='env://')
```

2. Create sampler for batch dispatch (same as in horovod section without passing num_replicas and rank)

```
train_sampler = torch.utils.data.distributed.DistributedSampler(
                    train_dataset)
```

3. Distributed the models

```
model = DDP(model)
```

4. Set the sampler at each epoch start

```
model.train()
train_sampler.set_epoch(epoch)
```

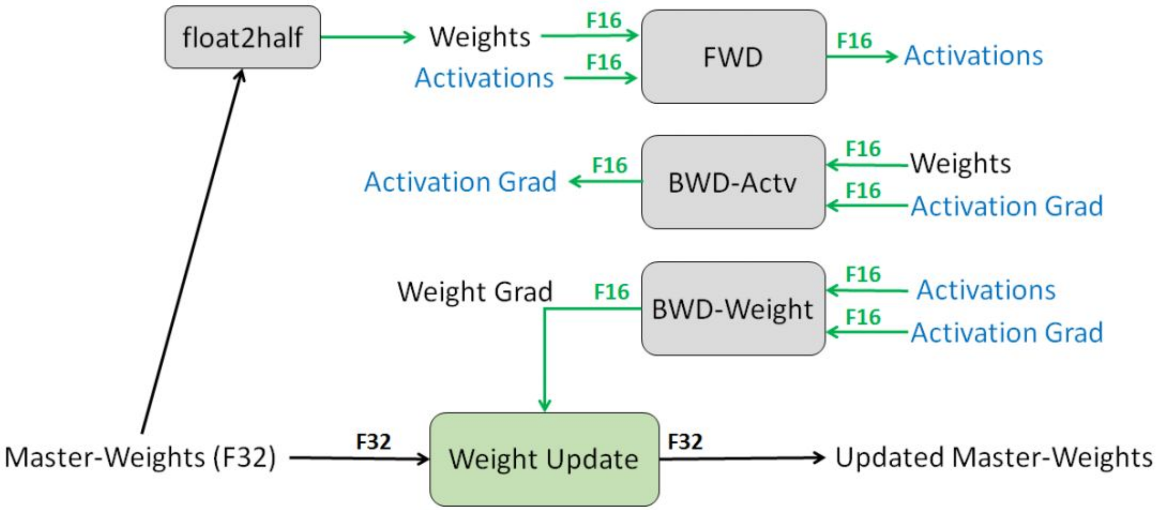
5. To invoke training script in terminal (example)

```
$ python -m torch.distributed.launch --nproc_per_node=4 main_multi_apex.py
```

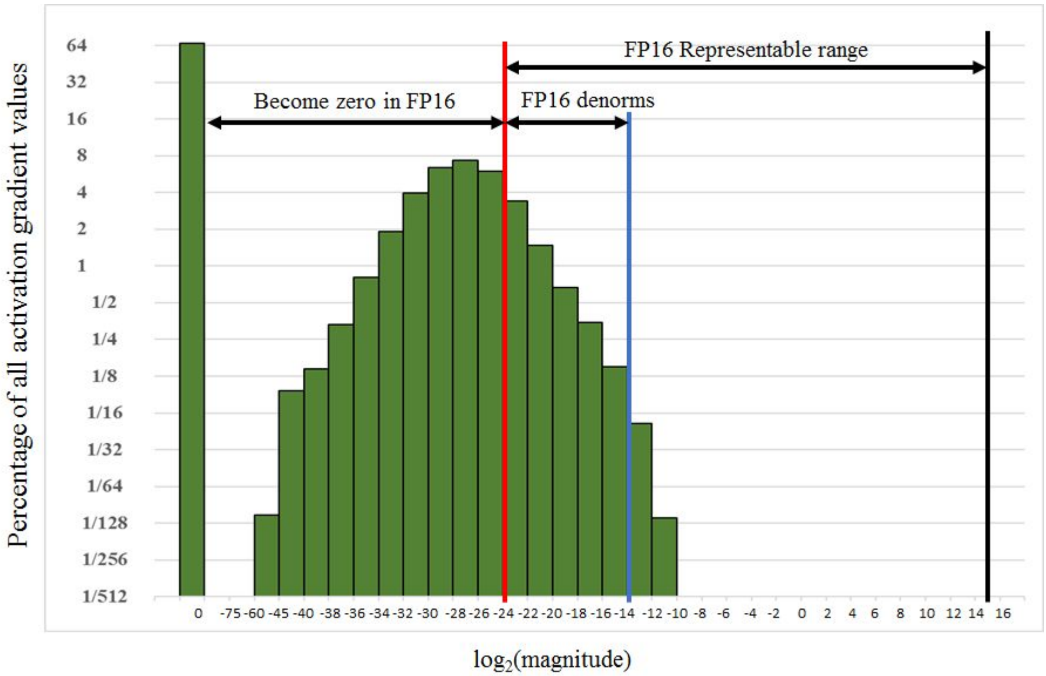
Mix-Precision Training

Tensorflow

Mixed Precision - ICLR 2018



1. **Make an FP16 copy of the weights.**
2. Forward propagate using FP16 weights and activations.
3. **Multiply the resulting loss by the scale factor S**
4. Backward propagate using FP16 weights, activations, and their gradients.
5. **Multiply the weight gradients by 1/S.**
6. Optionally process the weight gradients (gradient clipping, weight decay, etc.).
7. Update the master copy of weights in FP32.



Tensorflow

Mixed Precision - ICLR 2018

1. **Make an FP16 copy of the weights.**
2. Forward propagate using FP16 weights and activations.
3. **Multiply the resulting loss by the scale factor S**
4. Backward propagate using FP16 weights, activations, and their gradients.
5. **Multiply the weight gradients by 1/S.**
6. Optionally process the weight gradients (gradient clipping, weight decay, etc.).
7. Update the master copy of weights in FP32.

scale loss

- Gradient calculation with loss scaling to improve numerical stability when training with float16.

In [3]:
```
def gradients_with_loss_scaling(loss, variables, loss_scale):
    return [grad / loss_scale
            for grad in tf.gradients(loss * loss_scale, variables)]
```

Store as fp32, Train as fp16

- Custom variable getter that forces trainable variables to be stored in float32 precision and then casts them to the training precision.

In [2]:
```
def float32_variable_storage_getter(getter, name, shape=None, dtype=None,
                                    initializer=None, regularizer=None,
                                    trainable=True,
                                    *args, **kwargs):
    storage_dtype = tf.float32 if trainable else dtype
    variable = getter(name, shape, dtype=storage_dtype,
                      initializer=initializer, regularizer=regularizer,
                      trainable=trainable,
                      *args, **kwargs)
    if trainable and dtype != tf.float32:
        variable = tf.cast(variable, dtype)
    return variable
```

A simple Model

In [4]:
```
# Create training graph
with tf.device('/gpu:0'), \
     tf.variable_scope(
         # Note: This forces trainable variables to be stored as float32
         'fp32_storage', custom_getter=float32_variable_storage_getter):
    data, target, loss = create_simple_model()
    variables = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES)
    # Note: Loss scaling can improve numerical stability for fp16 training
    grads = gradients_with_loss_scaling(loss, variables, loss_scale)
    optimizer = tf.train.MomentumOptimizer(learning_rate, momentum)
    training_step_op = optimizer.apply_gradients(zip(grads, variables))
    init_op = tf.global_variables_initializer()
```

PyTorch

Mixed Precision

Make sure import the correct modules in Apex

```
from apex.fp16_utils import FP16_Optimizer
```

1. Wrap the optimizer with apex version

```
optimizer = FP16_Optimizer(optimizer, static_loss_scale=128.0)
```

2. Turn model and input to half-precision

```
model = SOME_MODERL().cuda().half()
input = input.cuda().half()
```

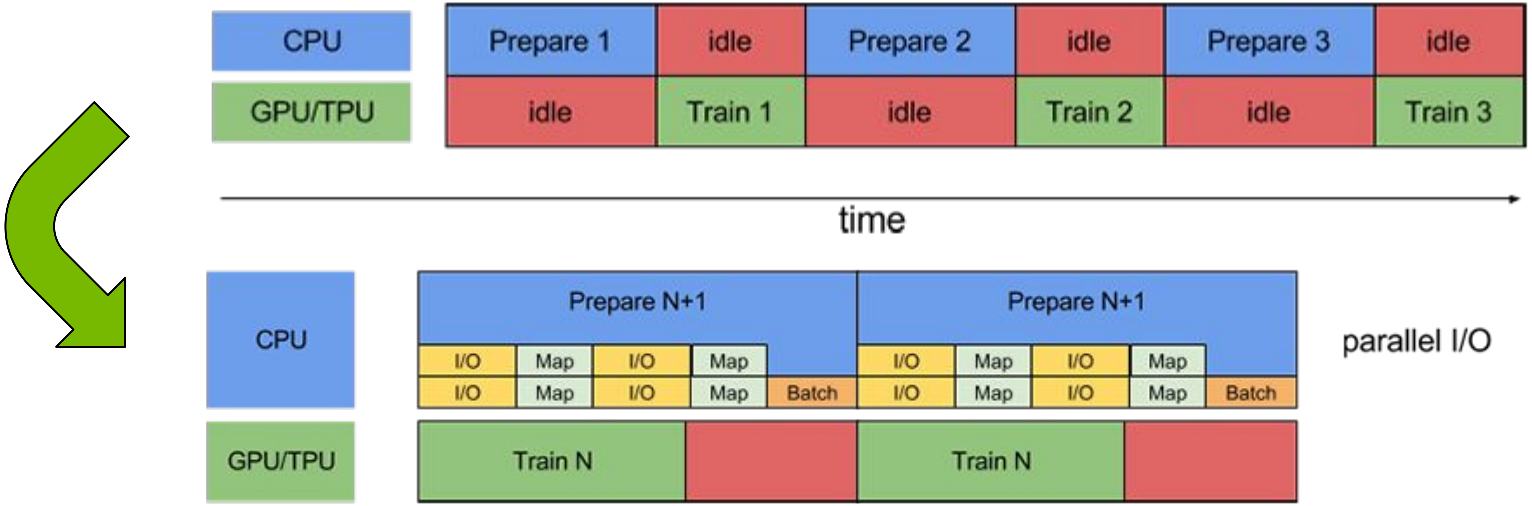
3. Use optimizer.backward instead of loss.backward

```
#loss.backward()              # Original
optimizer.backward(loss)      # Mixed-precision training
```

Tensorflow

High Performance Data Input Pipeline

TF Record




```python
def parse_fn(example):
    # Parse TFExample records and perform simple data augmentation.
    example_fmt = {
        "image": tf.FixedLengthFeature((), tf.string, ""),
        "label": tf.FixedLengthFeature((), tf.int64, -1)
    }
    parsed = tf.parse_single_example(example, example_fmt)
    image = tf.image.decode_image(parsed["image"])
    image = _augment_helper(image)  # augments image using slice, reshape, resize_bilinear
    return image, parsed["label"]

def input_fn():
    files = tf.data.Dataset.list_files("/path/to/dataset/train-*.tfrecord")
    dataset = files.apply(tf.contrib.data.parallel_interleave(
        tf.data.TFRecordDataset, cycle_length=FLAGS.num_parallel_readers))
    dataset = files.interleave(tf.data.TFRecordDataset)
    dataset = dataset.shuffle(buffer_size=FLAGS.shuffle_buffer_size)
    dataset = dataset.map(map_func=parse_fn, num_parallel_calls=FLAGS.num_parallel_calls)
    dataset = dataset.batch(batch_size=FLAGS.batch_size)
    dataset = dataset.prefetch(buffer_size=FLAGS.prefetch_buffer_size)
    return dataset
```


NVIDIA