# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2.1</td>
<td>Half Precision Format</td>
<td>2</td>
</tr>
<tr>
<td>2.2</td>
<td>Volta Tensor Core Math</td>
<td>3</td>
</tr>
<tr>
<td>2.3</td>
<td>Considering When Training With Mixed Precision</td>
<td>3</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Loss Scaling To Preserve Small Gradient Magnitudes</td>
<td>5</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Choosing A Scaling Factor</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Multi-GPU Training</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>Prerequisites</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>Frameworks</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>Deploying DNNs</td>
<td>21</td>
</tr>
</tbody>
</table>
Volta is NVIDIA’s latest architecture for deep learning frameworks. Volta retains and extends the same programming models provided by previous NVIDIA architectures such as Pascal. Applications that follow the best practices for those architectures should typically see speedups on the Volta architecture without any code changes. This guide summarizes the ways that a framework can be fine-tuned to gain additional speedups by leveraging the Volta architectural features.

For more information about Volta and its architecture, see Volta.
Chapter 2.
MIXED PRECISION TRAINING

The ability to train deep learning networks with lower precision was introduced in the Pascal architecture and first supported in CUDA® 8 in the NVIDIA Deep Learning SDK.

*Mixed precision* is the combined use of different numerical precisions in a computational method.

*Half precision* (also known as FP16) data compared to higher precision FP32 vs FP64 reduces memory usage of the neural network, allowing training and deployment of larger networks, and FP16 data transfers take less time than FP32 or FP64 transfers.

*Single precision* (also known as 32-bit) is a common floating point format (*float* in C-derived programming languages), and 64-bit, known as double precision (*double*).

### 2.1. Half Precision Format

IEEE 754 standard defines the following 16-bit half-precision floating point format: 1 sign bit, 5 exponent bits, and 10 fractional bits.

Exponent is encoded with 15 as the bias, resulting [-14, 15] exponent range (two exponent values, 0 and 31, are reserved for special values). An implicit lead bit 1 is assumed for normalized values, just like in other IEEE floating point formats.

Half precision format leads to the following dynamic range and precision:

**Normalized values**
- \(2^{-14}\) to \(2^{15}\), 11 bits of significand

**Denormal values**
- \(2^{-24}\) to \(2^{-15}\), significand bits decrease as the exponent gets smaller. Exponent \(k\) in [-24, -15] range results in \((25 - k)\) bits of significand precision.

Some example magnitudes:

**Maximum normalized**
- \(65,504\)

**Minimum normalized**
- \(2^{-14} \approx -6.10e-5\)
Minimum denormal
\[ 2^{-24} = \sim 5.96e^{-8} \]

Half precision dynamic range, including denormals, is 40 powers of 2. For comparison, single precision dynamic range including denormals is 264 powers of 2.

### 2.2. Volta Tensor Core Math

The Volta generation of GPUs introduces Tensor Cores, which provide 8x more throughput than single precision math pipelines. Each Tensor Core performs \( D = A \times B + C \), where \( A, B, C \) and \( D \) are matrices. \( A \) and \( B \) are half-precision 4x4 matrices, whereas \( D \) and \( C \) can be either half or single precision 4x4 matrices. In other words, Tensor Core math can accumulate half precision products into either single or half-precision outputs.

In practice, higher performance is achieved when \( A \) and \( B \) dimensions are multiples of 8. CUDA® Deep Neural Network library™ (cuDNN) v7 and CUDA® Basic Linear Algebra Subroutines library™ (cuBLAS) 9 include some functions that invoke Tensor Core operations, for performance reasons these require that input and output feature map sizes are multiples of 8. For more information, see cuDNN Developer Guide.

The reason half precision is so attractive is that the V100 GPU has 640 Tensor Cores, so they can all be performing 4x4 multiplications all at the same time. This is roughly 16 times faster than double precision (FP64) and about 8 times faster than single precision (FP32).

Matrix multiplies are at the core of Convolutional Neural Networks (CNN). CNN’s are very common in deep learning in many networks. Beginning in CUDA 9 and cuDNN 7, the convolution operations are done using Tensor Cores whenever possible. This can greatly improve the training speed as well as the inference speed of CNN’s or models that contain convolutions.

### 2.3. Considering When Training With Mixed Precision

Given a framework that supports Volta Tensor Core math, many networks can be trained faster by simply enabling the Tensor Core path in the framework (choosing FP16 format for tensors and/or convolution/fully-connected layers; for more details see Frameworks) and keeping all the hyperparameters of FP32 training session.

However, some networks require their gradient values to be shifted into FP16 representable range to match the accuracy of FP32 training sessions. The figure below illustrates one such case.
Figure 1  Histogram of activation gradient magnitudes throughout FP32 training of Multibox SSD network. The x-axis is logarithmic, except for the zero entry. For example, 66.8% of values were 0, 4% had magnitude in the $(2^{-32}, 2^{-30})$ range.

However, this isn’t always the case. You may have to do some scaling and normalization to use FP16 during training.

Figure 2  Histogram of activation gradient magnitudes throughout FP32 training of Multibox SSD network. Both x- and y-axes are logarithmic.
Consider the histogram of activation gradient values (shown with linear and log y-scales above), collected across all layers during FP32 training of Multibox SSD detector network (VGG-D backbone). When converted to FP16, 31% of these values become zeros, leaving only 5.3% as non-zeros which for this network lead to divergence during training.

Much of the FP16 representable range was left unused by the gradient values. Therefore, if we shift the gradient values to occupy more of that range, we can preserve many values that are otherwise lost to 0s.

For this particular network, shifting by 3 exponent values (multiply by 8) was sufficient to match the accuracy achieved with FP32 training by recovering the relevant values lost to 0. Shifting by 15 exponent values (multiplying by 32K) would recover all but 0.1% of values lost to 0 when converting to FP16 and still avoid overflow. In other words, FP16 dynamic range is sufficient for training, but gradients may have to be scaled to move them into the range to keep them from becoming zeros in FP16.

2.3.1. Loss Scaling To Preserve Small Gradient Magnitudes

As was shown in the previous section, successfully training some networks requires gradient value scaling to keep them from becoming zeros in FP16. This can be efficiently achieved with a single multiplication by scaling the loss value computed in the forward pass, prior to starting backpropagation. By the chain rule, backpropagation ensures that all the gradient values are scaled by the same amount. This requires no extra operations during backpropagation and keeps the relevant gradient values from becoming zeros and losing that gradient information.

Weight gradients must be unscaled before weight update, to maintain the magnitude of updates the same as in FP32 training. It is simplest to perform this descaling right after the backward pass but before gradient clipping or any other gradient-related computations. This ensures that no hyperparameters (such as gradient clipping threshold, weight decay, etc.) have to be adjusted.

While many networks match FP32 training results when all tensors are stored in FP16, some require updating an FP32 copy of weights. Furthermore, values computed by large reductions should be left in FP32. Examples of this include statistics (mean and variance) computed by batch-normalization, SoftMax.

Batch-normalization can still take FP16 inputs and outputs, saving half the bandwidth compared to FP32, it’s just that the statistics and value adjustment should be done in FP32. This leads to the following high-level procedure for training:

1. Maintain a master copy of weights in FP32
2. For each iteration:
   a. Make an FP16 copy of the weights
   b. Forward propagation (FP16 weights and activations)
   c. Multiply the resulting loss with the scaling factor $S$
   d. Backward propagation (FP16 weights, activations, and their gradients)
2.3.2. Choosing A Scaling Factor

The procedure described in the previous section requires you to pick a loss scaling factor to adjust the gradient magnitudes. There is no downside to choosing a large scaling factor as long as it doesn’t cause overflow during backpropagation, which would lead to weight gradients containing infinities or NaNs, that in turn would irreversibly damage the weights during the update. These overflows can be easily and efficiently detected by inspecting the computed weight gradients, for example, multiply the weight gradient with 1/S step in the previous section.

One option is to skip the weight update when an overflow is detected and simply move on to the next iteration.

There are several options to choose the loss scaling factor. The simplest one is to pick a constant scaling factor. We trained a number of feed-forward and recurrent networks with Tensor Core math for various tasks with scaling factors ranging from 8 to 32K (many networks did not require a scaling factor), matching the network accuracy achieved by training in FP32. However, since the minimum required scaling factor can depend on the network, framework, minibatch size, etc., some trial and error may be required when picking a scaling value. A constant scaling factor can be chosen more directly if gradient statistics are available. Choose a value so that its product with the maximum absolute gradient value is below 65,504 (the maximum value representable in FP16).

A more robust approach is to choose the loss scaling factor dynamically. The basic idea is to start with a large scaling factor and then reconsider it in each training iteration. If no overflow occurs for a chosen number of iterations N then increase the scaling factor. If an overflow occurs, skip the weight update and decrease the scaling factor. We found that as long as one skips updates infrequently the training schedule does not have to be adjusted to reach the same accuracy as FP32 training. Note that N effectively limits how frequently we may overflow and skip updates. The rate for scaling factor update can be adjusted by picking the increase/decrease multipliers as well as N, the number of non-overflow iterations before the increase. We successfully trained networks with N = 2000, increasing scaling factor by 2, decreasing scaling factor by 0.5, many other settings are valid as well. Dynamic loss-scaling approach leads to the following high-level training procedure:

1. Maintain a master copy of weights in FP32.
2. Initialize $S$ to a large value.
3. For each iteration:
   a. Make an FP16 copy of the weights.
   b. Forward propagation (FP16 weights and activations).
   c. Multiply the resulting loss with the scaling factor $S$.
   d. Backward propagation (FP16 weights, activations, and their gradients).
   e. If there is an Inf or NaN in weight gradients:
      e. Multiply the weight gradient with $1/S$
      f. Complete the weight update (including gradient clipping, etc.)
a. Reduce $S$.
b. Skip the weight update and move to the next iteration.
f. Multiply the weight gradient with $1/S$.
g. Complete the weight update (including gradient clipping, etc.).
h. If there hasn't been an Inf or NaN in the last $N$ iterations, increase $S$. 
Chapter 3.
MULTI-GPU TRAINING

For multi-GPU training, the same strategy applies for loss scaling. NVIDIA® Collective Communications Library ™ (NCCL) supports both half precision floats and normal floats, therefore, a developer can choose which precision they want to use to aggregate gradients. Batch size considerations depend on your training framework.
Chapter 4. PREREQUISITES

To take advantage of the Volta architecture and mixed precision training, ensure you meet the following minimum requirements:

1. Run on the Volta architecture.
2. Install NVIDIA drivers. It is recommended to install the latest 384 series of NVIDIA drivers for use with the Tesla V100 GPUs. The latest recommended version of the Linux driver for Tesla V100 is 384.66.
3. Install the CUDA® Toolkit™ 9.
4. Install cuDNN v7.

If using an NVIDIA optimized framework container, that was pulled from either the DGX or NGC container registry, you will still need to install an NVIDIA driver on your base operating system. However, CUDA and cuDNN will come included in the container. For more information, see the Frameworks Support Matrix.
Most major deep learning frameworks have begun to merge support for half precision training techniques that exploit Tensor Core calculations in Volta. Additional optimization pull requests are at various stages and listed in their respective section. For NVCaffe™, Caffe2™, MXNet™, Microsoft® Cognitive Toolkit™, PyTorch™, TensorFlow™ and Theano™, Tensor Core acceleration is automatically enabled if FP16 storage is enabled.

While frameworks like Torch™ will tolerate the Volta architecture, it currently does not exploit Tensor Core functionality.

### 5.1. NVCaffe

NVCaffe includes support for FP16 storage and Tensor Core math. To achieve optimum performance, you can train a model using Tensor Core math and FP16 mode on NVCaffe.

#### 5.1.1. Running FP16 Training On NVCaffe

1. Pull the latest NVCaffe container from the NVIDIA GPU Cloud (NGC) container registry. The container is already built, tested, tuned, and ready to run. The NVCaffe container includes the latest CUDA version, FP16 support, and is optimized for the Volta architecture. For step-by-step pull instructions, if you have a DGX-1, see the Containers for Deep Learning Frameworks User Guide, otherwise refer to the Using NGC with Your NVIDIA TITAN or Quadro PC Setup Guide.

2. Experiment with the following training parameters:
   a) Before running the training script below, adjust the batch size for better performance. To do so, open the training settings with your choice of editor, for example, vim:

```bash
caffe$ vim models/resnet50/train_val_fp16.prototxt
```
And change the **batch_size**: 32 setting value to \([64...128]\) \* <Number of GPUs installed>.

b) Experiment with pure FP16 mode by setting:

```plaintext
default_forward_math: FLOAT16
default_backward_math: FLOAT16
```

And by adding **solver_data_type**: FLOAT16 to the file `models/resnet50/solver_fp16.prototxt`.

3. Train ResNet-50. Open:

```bash
caffe$ ./models/resnet50/train_resnet50_fp16.sh
```

When the training is finished, it should look similar to the following:

```
```

The performance number of 5268 img/sec was trained on an 8-GPU system. For a single GPU system, you could expect around 660 img/sec training with NVcaffe.

4. View the output. Issue the following command:

```bash
caffe$ python plot_top5.py -s models/resnet50/logs/resnet50_fp16.log
```

Your output should look similar to the following:
5.1.2. NVCaffe Example

For examples on optimization, see the `models/resnet50/train_val_fp16.prototxt` file.

5.2. Caffe2

Caffe2 includes support for FP16 storage and Tensor Core math. To achieve optimum performance, you can train a model using Tensor Core math and FP16 mode on Caffe2.

When training a model on Caffe2 using Tensor Core math and FP16, the following actions need to take place:

- Prepare your data. You can generate data in FP32 and then cast it down to FP16. The GPU transforms path of the `ImageInput` operation can do this casting in a fused manner.
- Forward pass. Since data is given to the network in FP16, all of the subsequent operations will run in FP16 mode, therefore:
Select which operators need to have both FP16 and FP32 parameters by setting the type of Initializer used. Typically, the Conv and FC operators need to have both parameters.

- Cast the output of forward pass, before SoftMax, back to FP32.
- To enable Tensor Core, pass `enable_tensor_core=True` to ModelHelper when representing a new model.
- Update the master FP32 copy of the weights using the FP16 gradients you just computed. For example:
  - Cast up gradients to FP32.
  - Update the FP32 copy of parameters.
  - Cast down the FP32 copy of parameters to FP16 for the next iteration.

Gradient scaling.

- To scale, multiply the loss by the scaling factor.
- To descale, divide LR and `weight_decay` by the scaling factor.

### 5.2.1. Running FP16 Training On Caffe2

1. Pull the latest Caffe2 container from the NVIDIA GPU Cloud (NGC) container registry. The container is already built, tested, tuned, and ready to run. The Caffe2 container includes the latest CUDA version, FP16 support, and is optimized for the Volta architecture. For step-by-step pull instructions, see the Containers for Deep Learning Frameworks User Guide.

2. Run the following Python script with the appropriate command line arguments. You can test using the ResNet-50 image classification training script included in Caffe2.

```python
python caffe2/python/examples/resnet50_trainer.py --train_data <path> --test_data <path> --num-gpus <int> --batch-size <int> --dtype float16 --enable-tensor-core --cudnn_workspace_limit_mb 1024 --image_size 224
```

For more information about the additional command-line arguments, issue the following command:

```bash
caffe2/python/examples/resnet50_trainer.py --help
```

To enhance performance, the following changes must be made:

- The network definition in `caffe2/python/models/resnet.py` must be changed to reflect version 1 of the network by changing the residual block striding from the 3x3 convolution to the first 1x1 convolution operator.
- Enable optimized communication operators and disable some communication ops by adding the `use_nccl=True` and `broadcast_computed_params=False` flags to the `data_parallel_model.Parallelize` call in `caffe2/python/examples/resnet50_trainer.py`.
- Add `decode_threads=3` and `use_gpu_transform=True` to the `brew.image_input` call. This tweaks the amount of CPU threads used for data decode and augmentation (value is per-GPU) and enables the use of the GPU for some data augmentation work.
Increase the number of host threads used to schedule operators on the GPUs by adding `train_model.net.Proto().num_workers = 4 * len(gpus)` after the call to `data_parallel_model.Parallelize`.

### 5.2.2. Caffe2 Example

For more information, you can find examples at: Caffe2 Python Examples.

### 5.3. MXNet

MXNet includes support for FP16 storage and Tensor Core math. To achieve optimum performance, you need to train a model using Tensor Core math and FP16 mode on MXNet.

The following procedure is typical for when you want to have your entire network in FP16. Alternatively, you can take output from any layer and cast it to FP16. Subsequent layers will be in FP16 and will use Tensor Core math if applicable.

#### 5.3.1. Running FP16 Training On MXNet

1. Pull the latest MXNet container from the NVIDIA GPU Cloud (NGC) container registry. The container is already built, tested, tuned, and ready to run. The MXNet container includes the latest CUDA version, FP16 support, and is optimized for the Volta architecture. For step-by-step pull instructions, see the Containers for Deep Learning Frameworks User Guide.

2. To use the IO pipeline, use the `IndexedRecordIO` format of input. It differs from the legacy `RecordIO` format, by including an additional index file with an `.idx` extension. The `.idx` file is automatically generated when using the `im2rec.py` tool, to generate new RecordIO files. If you already have the `.rec` file without the corresponding `.idx` file, you can generate the index file with `tools/rec2idx.py` tool:

   ```bash
   python tools/rec2idx.py <path to .rec file> <path to newly created .idx file>
   ```

3. To use FP16 training with MXNet, cast the data (input to the network) to FP16.

   ```python
   mxnet.sym.Cast(data=input_data, dtype=numpy.float16)
   ```

4. Cast back to FP32 before SoftMax layer.

5. If you encounter precision problems, it is beneficial to scale the loss up by 128, and scale the application of the gradients down by 128. This ensures higher gradients during the backward pass calculation, but will still correctly update the weights. For example, if out last layer is `mx.sym.SoftmaxOutput` (cross-entropy loss), and the initial learning rate is 0.1, add a grad_scale parameter:

   ```python
   mxnet.sym.SoftmaxOutput(other_args, grad_scale=128.0)
   ```

   When initializing the optimizer, rescale the gradient down prior to the application:
Frameworks

www.nvidia.com

Training with Mixed Precision

mxnet.optimizer.SGD(other_args, rescale_grad=1.0/128)

Tip When training in FP16, it is best to use multi-precision optimizers that keep the weights in FP32 and perform the backward pass in FP16. For example, for SGD with momentum, you would issue the following:

mxnet.optimizer.SGD(other_args, momentum=0.9, multi_precision=True)

Alternatively, you can pass ‘multi_precision’: True to the optimizer_params option in the model.fit method.

5.3.2. MXNet Example

In the following example, ResNet-50 v1 is trained on 8 Volta GPUs with ImageNet data. This example assumes that data is in the /data/imagenet/ directory and is pre-resized to 480px shorter side.

```python
import mxnet as mx
import numpy as np

input_dim = 784
num_output_classes = 10
num_hidden_layers = 1
hidden_layers_dim = 200

# Input variables denoting the features and label data
feature = mx.sym.Variable('feature', shape=(None, input_dim))
label = mx.sym.Variable('label', shape=(None, num_output_classes))
```

For example networks, refer to the example/image-classification/symbols/ directory. You can enable FP16 in these networks by passing the --dtype float16 option to the train_imagenet.py script.

5.4. Microsoft Cognitive Toolkit

Microsoft Cognitive Toolkit includes support for FP16 storage and Tensor Core math. To achieve optimum performance, you need to train a model using Tensor Core math and FP16 mode on Microsoft Cognitive Toolkit.

5.4.1. Running FP16 Training On Microsoft Cognitive Toolkit

Tensor Core math is turned on by default in FP16. The following procedure is typical of Microsoft Cognitive Toolkit using FP16 in a multi-layer perceptron MNIST example.

```python
import cntk as C
import numpy as np

input_dim = 784
num_output_classes = 10
num_hidden_layers = 1
hidden_layers_dim = 200

# Input variables denoting the features and label data
feature = C.input_variable(input_dim, np.float32)
label = C.input_variable(num_output_classes, np.float32)
```
feature16 = C.cast(feature, np.float16)
label16 = C.cast(label, np.float16)

with C.default_options(dtype=np.float16):
    # Instantiate the feedforward classification model
    scaled_input16 = C.element_times(C.constant(0.00390625, dtype=np.float16),
                                      feature16)

    z16 = C.layers.Sequential([C.layers.For(range(num_hidden_layers),
                                lambda i: C.layers.Dense(hidden_layers_dim,
                                                         activation=C.relu)),
                                C.layers.Dense(num_output_classes)])(scaled_input16)

    ce16 = C.cross_entropy_with_softmax(z16, label16)
    pe16 = C.classification_error(z16, label16)

    z = C.cast(z16, np.float32)
    ce = C.cast(ce16, np.float32)
    pe = C.cast(pe16, np.float32)

    # fake data with batch_size = 5
    batch_size = 5
    feature_data = np.random.randint(0, 256, (batch_size, 784)).astype(np.float32)
    label_data = np.eye(num_output_classes)[np.random.randint(0, num_output_classes,
                                                          batch_size)]
    ce.eval({feature:feature_data, label:label_data})

5.4.2. Microsoft Cognitive Toolkit Example

For a more complete example of ResNet-50 with distributed training, see the TrainResNet_ImageNet_Distributed.py example.

5.5. PyTorch

PyTorch includes support for FP16 storage and Tensor Core math. To achieve optimum performance, you can train a model using Tensor Core math and FP16 mode on PyTorch.

5.5.1. Running FP16 Training On PyTorch

To run FP16 training jobs, you need to make modifications to the PyTorch framework on the user script level. The following steps are implemented in the ImageNet and world language model examples in the PyTorch examples repository.

To run these examples with FP16, follow the instructions for the corresponding examples and add `--fp16` to command line arguments. For example, assuming your ImageNet training and validation folders are in the examples/imagenet folder, you would issue the following commands:

```
python imagenet/main.py -a resnet50 imagenet/ --workers 10 --batch-size 256 --fp16
python/word_language_model/main.py --cuda --emsize 1536 --nhid 1536 --dropout 0.65 --epochs 40 --fp16
```
1. Pull the latest PyTorch container from the NVIDIA GPU Cloud (NGC) container registry. The container is already built, tested, tuned, and ready to run. The PyTorch container includes the latest CUDA version, FP16 support, and is optimized for the Volta architecture. For step-by-step pull instructions, see the Containers for Deep Learning Frameworks User Guide.

2. Cast the model and inputs to FP16.

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

   a) Optional: For parallel training, using `torch.nn.DataParallel`, instead of directly casting inputs to FP16 on the GPU, add a layer to the model that would convert inputs on the GPU from FP32 to FP16.

3. Create 32-bit master copy of the parameters. Create the optimizer using the master copy of the parameters.

```python
param_copy = [param.clone().type(torch.cuda.FloatTensor).detach() for param in model.parameters()]
for param in param_copy:
    param.requires_grad = True
optimizer = torch.optim.SGD(param_copy, lr,momentum=momentum, weight_decay=weight_decay)
```

4. Optional: If the model uses batch normalization, replace the batch normalization layers in the model definition with a special batch normalization layer that uses cuDNN and stores its parameters and buffers in FP32.

```python
nn.BatchNorm2d = torch.nn.contrib.BatchNorm2dFP16
```

5. Optional: Scale the loss.

```python
loss = loss * scale_factor
```

6. At each optimization step in the training loop, perform the following:

   a) Cast gradients to FP32. If a loss was scaled, descale the gradients.

   b) Apply updates in FP32 precision and copy the updated parameters to the model, casting them to FP16.

```python
model.zero_grad()
loss.backward()
set_grad(param_copy, list(model.parameters()))
if scale_factor != 1:
    for param in param_copy:
        param.grad.data = param.grad.data/args.loss_scale
optimizer.step()
params = list(model.parameters())
for i in range(len(params)):
    params[i].data.copy_(param_copy[i].data)
```

5.5.2. PyTorch Example

For more information, you can find examples at: PyTorch Examples.

5.6. TensorFlow
TensorFlow supports FP16 storage and Tensor Core math. Models that contain convolutions or matrix multiplications using the `tf.float16` data type will automatically take advantage of Tensor Core hardware whenever possible.

5.6.1. Running FP16 Training On TensorFlow

1. Pull the latest TensorFlow container from the NVIDIA GPU Cloud (NGC) container registry. The container is already built, tested, tuned, and ready to run. The TensorFlow container includes the latest CUDA version, FP16 support, and is optimized for the Volta architecture. For step-by-step pull instructions, see the Containers for Deep Learning Frameworks User Guide.

2. Use the `tf.float16` data type on models that contain convolutions or matrix multiplications. This data type automatically takes advantage of the Tensor Core hardware whenever possible. For example:

   ```python
dtype = tf.float16
   data = tf.placeholder(dtype, shape=(nbatch, nin))
   weights = tf.get_variable('weights', (nin, nout), dtype)
   biases  = tf.get_variable('biases', (nout, dtype),
                               initializer=tf.zeros_initializer())
   logits = tf.matmul(data, weights) + biases
   ```

3. Ensure the trainable variables are in float32 precision and cast them to float16 before using them in the model. For example:

   ```python
tf.cast(tf.get_variable(..., dtype=tf.float32), tf.float16)
   ```

   This can also be achieved by using the `float32_variable_storage_getter` shown in the following example.

4. Ensure the SoftMax calculation is in float32 precision. For example:

   ```python
tf.losses.softmax_cross_entropy(target, tf.cast(logits, tf.float32))
   ```

5. Apply loss-scaling as outlined in the previous sections. Loss scaling involves multiplying the loss by a scale factor before computing gradients, and then dividing the resulting gradients by the same scale again to re-normalize them. For example, to apply a constant loss scaling factor of 128:

   ```python
   loss, params = ...
   scale = 128
   grads = [grad / scale for grad in tf.gradients(loss * scale, params)]
   ```

5.6.2. TensorFlow Example

The following script demonstrates construction and training of a simple multinomial logistic regression model. The script uses the FP16-training guidelines described in the previous section.

```python
import tensorflow as tf
import numpy as np

def float32_variable_storage_getter(getter, name, shape=None, dtype=None,
                                     initializer=None, regularizer=None,
                                     trainable=True, *args, **kwargs):
```
"""Custom variable getter that forces trainable variables to be stored in float32 precision and then casts them to the training precision.
"""
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

def gradients_with_loss_scaling(loss, variables, loss_scale):
    """Gradient calculation with loss scaling to improve numerical stability when training with float16.
    """
    return [grad / loss_scale
            for grad in tf.gradients(loss * loss_scale, variables)]

def create_simple_model(nbatch, nin, nout, dtype):
    """A simple softmax model."""
    data = tf.placeholder(dtype, shape=(nbatch, nin))
    weights = tf.get_variable('weights', (nin, nout), dtype)
    biases = tf.get_variable('biases', nout, dtype,
                              initializer=tf.zeros_initializer())
    logits = tf.matmul(data, weights) + biases
    target = tf.placeholder(tf.float32, shape=(nbatch, nout))
    # Note: The softmax should be computed in float32 precision
    loss = tf.losses.softmax_cross_entropy(target, tf.cast(logits, tf.float32))
    return data, target, loss

if __name__ == '__main__':
    nbatch = 64
    nin = 100
    nout = 10
    learning_rate = 0.1
    momentum = 0.9
    loss_scale = 128
    dtype = tf.float16
    tf.set_random_seed(1234)
    np.random.seed(4321)
    # 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(nbatch, nin, nout, dtype)
        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()
    # Run training
    sess = tf.Session()
    sess.run(init_op)
    np_data   = np.random.normal(size=(nbatch, nin)).astype(np.float16)
    np_target = np.zeros((nbatch, nout), dtype=np.float32)
    np_target[:,0] = 1
    print 'Step Loss'
    for step in xrange(30):
        np_loss, _ = sess.run([loss, training_step_op],
5.7. Theano

Theano includes support for FP16 storage and Tensor Core math. To make use of Tensor Core math, set the `dnn.conv.algo_XXX` configuration parameter to `time_once` or `time_on_shape_change`, for example:

```
[dnn]
conv.algo_fwd=time_once
conv.algo_bwd_filter=time_once
conv.algo_bwd_data=time_once
```

5.7.1. Running FP16 Training On Theano

Theano is fully parameterized on `floatX` type, therefore, to run most Theano scripts in FP16, you can issue:

```
THEANO_FLAGS="floatX=float16"
```
After you have trained a neural network, you can optimize and deploy the model for GPU inferencing with TensorRT™. For more information about optimizing and deploying using TensorRT, see Deep Learning SDK Documentation.
Notice

THE INFORMATION IN THIS GUIDE AND ALL OTHER INFORMATION CONTAINED IN NVIDIA DOCUMENTATION REFERENCED IN THIS GUIDE IS PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE INFORMATION FOR THE PRODUCT, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the product described in this guide shall be limited in accordance with the NVIDIA terms and conditions of sale for the product.

THE NVIDIA PRODUCT DESCRIBED IN THIS GUIDE IS NOT FAULT TOLERANT AND IS NOT DESIGNED, MANUFACTURED OR INTENDED FOR USE IN CONNECTION WITH THE DESIGN, CONSTRUCTION, MAINTENANCE, AND/OR OPERATION OF ANY SYSTEM WHERE THE USE OR A FAILURE OF SUCH SYSTEM COULD RESULT IN A SITUATION THAT THREATENS THE SAFETY OF HUMAN LIFE OR SEVERE PHYSICAL HARM OR PROPERTY DAMAGE (INCLUDING, FOR EXAMPLE, USE IN CONNECTION WITH ANY NUCLEAR, AVIONICS, LIFE SUPPORT OR OTHER LIFE CRITICAL APPLICATION). NVIDIA EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR SUCH HIGH RISK USES. NVIDIA SHALL NOT BE LIABLE TO CUSTOMER OR ANY THIRD PARTY, IN WHOLE OR IN PART, FOR ANY CLAIMS OR DAMAGES ARISING FROM SUCH HIGH RISK USES.

NVIDIA makes no representation or warranty that the product described in this guide will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this guide. NVIDIA does not accept any liability related to any default, damage, costs or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this guide, or (ii) customer product designs.

Other than the right for customer to use the information in this guide with the product, no other license, either expressed or implied, is hereby granted by NVIDIA under this guide. Reproduction of information in this guide is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

Trademarks

NVIDIA, the NVIDIA logo, and cuBLAS, CUDA, cuDNN, cuFFT, cuSPARSE, DALI, DIGITS, DGX, DGX-1, Jetson, Kepler, NVIDIA Maxwell, NCCL, NVLink, Pascal, Tegra, TensorRT, and Tesla are trademarks and/or registered trademarks of NVIDIA Corporation in the Unites States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2018 NVIDIA Corporation. All rights reserved.