NVIDIA

MIXED PRECISION TRAINING OF DEEP NEURAL NETWORKS

Carl Case, NVIDIA

OUTLINE

1. What is mixed precision training?
2. Considerations and methodology for mixed precision training
3. Automatic mixed precision
4. Performance guidelines and practical recommendations

OUTLINE

1. What is mixed precision training?
2. Considerations and methodology for mixed precision training
3. Automatic mixed precision
4. Performance guidelines and practical recommendations

MIXED PRECISION TRAINING

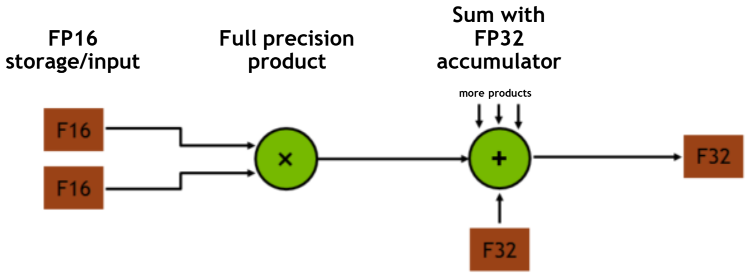
Motivation

- Reduced precision (16-bit floating point) for *speed* or *scale*
- Full precision (32-bit floating point) to *maintain task-specific accuracy*
- By using *multiple* precisions, we can avoid a pure tradeoff of speed and accuracy
- Goal: maximize use of reduced precision under the constraint of matching accuracy of full precision training with no changes to hyperparameters

TENSOR CORES

Hardware support for accelerated 16-bit FP math

- Peak throughput of **125 TFLOPS** (8x FP32) on V100
- Inherently mixed precision: internal accumulation occurs in FP32 for accuracy*
- Used by cuDNN and cuBLAS libraries to accelerate matrix multiply and convolution
- Exposed in CUDA as WMMA. See:
  https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/
  http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#wmma



*FP16 accumulator is also available for inference

MIXED PRECISION TRAINING

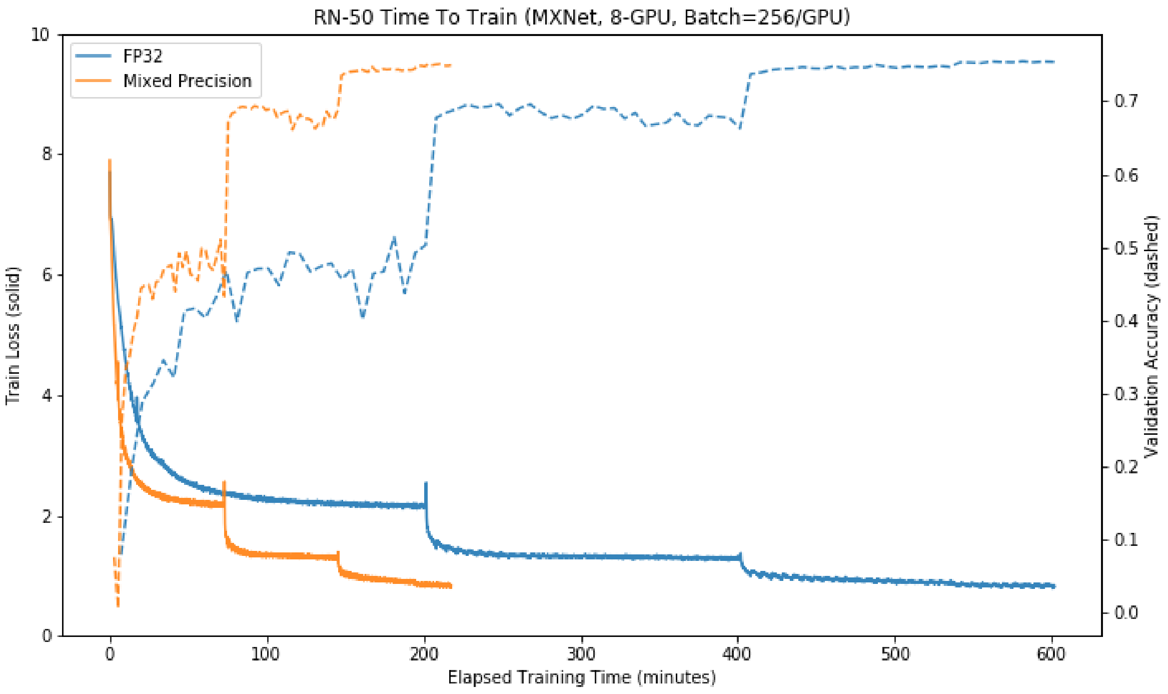
In a nutshell

- **Goal**
  - Keep stored values in half precision: weights and activations, along with their gradients
  - Use Tensor Cores to accelerate math and maintain accuracy
- **Benefits**
  - Up to 8x math speedup (depends on arithmetic intensity)
  - Half the memory *traffic*
  - Half the memory *storage*
    - Can enable larger model or batch sizes

MIXED PRECISION TRAINING

With Tensor Cores

- 8GPU training of ResNet-50 (ImageNet classification) on DGX-1
  - NVIDIA mxnet-18.08-py3 container
- Total time to run full training schedule in mixed precision is well under four hours
  - **2.9x speedup** over FP32 training
  - **Equal validation accuracies**
  - No hyperparameters changed
    - Minibatch = 256 per GPU



MIXED PRECISION IS GENERAL PURPOSE

Models trained to match FP32 results (same hyperparameters)

Image Classification
AlexNet
DenseNet
Inception
MobileNet
NASNet
ResNet
ResNeXt
VGG
XCeption

Detection / Segmentation
DeepLab
Faster R-CNN
Mask R-CNN
Multibox SSD
NVIDIA Automotive
RetinaNet
UNET

Recommendation
DeepRecommender
NCF

Generative Models (Images)
DLSS
Partial Image Inpainting
Progress GAN
Pix2Pix

Speech
Deep Speech 2
Tacotron
WaveNet
WaveGlow

Language Modeling
BERT
BigLSTM
8k mLSTM (NVIDIA)

Translation
FairSeq (convolution)
GNMT (RNN)
Transformer (self-attention)

MIXED PRECISION SPEEDUPS

Not limited to image classification

Model	FP32 -> M.P. Speedup	Comments
GNMT (Translation)	2.3x	Iso-batch size
FairSeq Transformer (Translation)	2.9x 4.9x	Iso-batch size 2x lr + larger batch
ConvSeq2Seq (Translation)	2.5x	2x batch size
Deep Speech 2 (Speech recognition)	4.5x	Larger batch
wav2letter (Speech recognition)	3.0x	2x batch size
Nvidia Sentiment (Language modeling)	4.0x	Larger batch

*In all cases trained to same accuracy as FP32 model

**No hyperparameter changes, except as noted

MIXED PRECISION IN DL RESEARCH

Both *accelerates* and *enables* novel research

- Large Scale Language Modeling: Converging on 40GB of Text in Four Hours [NVIDIA]
    - "We train our recurrent models with mixed precision FP16/FP32 arithmetic, which speeds up training on a single V100 by 4.2X over training in FP32."
- Scaling Neural Machine Translation [Facebook]
    - "This paper shows that reduced precision and large batch training can speedup training by nearly 5x on a single 8-GPU machine with careful tuning and implementation."
    - If you want to hear more:
        - "Taking Advantage of Mixed Precision to Accelerate Training Using PyTorch" [S9832]
        - Today (Mar. 18th) at 2pm in room 210D

OUTLINE

1. What is mixed precision training?
2. Considerations and methodology for mixed precision training
3. Automatic mixed precision
4. Performance guidelines and practical recommendations

MIXED PRECISION METHODOLOGY

For training

- Goal: training with FP16 is general purpose, not only for a limited class of applications
- In order to train with no architecture or hyperparameter changes, we need to give consideration to the reduced precision inherent in using only 16 bits
    - Note: true for any reduced precision format, though specifics may be different
- Three parts:

1. Model conversion, with careful handling of non-Tensor Core ops
2. Master weight copy
3. Loss scaling

1. MODEL CONVERSION

For Tensor Core ops

- For most of the model, we make simple type updates to each layer:
  - Use FP16 values for the weights (layer parameters)
  - Ensure the inputs are FP16, so the layer runs on Tensor Cores

```
# PyTorch
layer = torch.nn.Linear(in_dim, out_dim).half()

# TensorFlow
layer = tf.layers.dense(tf.cast(inputs, tf.float16),
                        out_dim)
```

1. MODEL CONVERSION

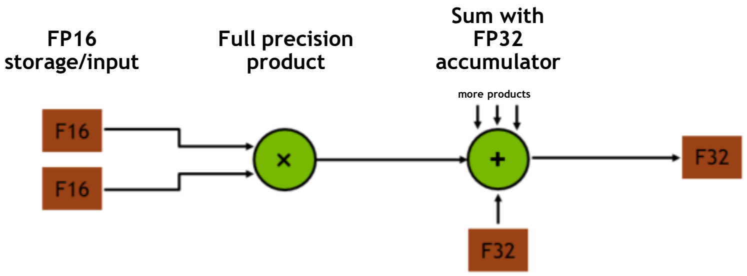
Pointwise and reduction ops

- Common operations that are not matrix multiply or convolution:
  - **Activation functions**: ReLU, sigmoid, tanh, softplus
  - **Normalization functions**: batchnorm, layernorm, sum, mean, softmax
  - **Loss functions**: cross entropy, L2 loss, weight decay
  - **Miscellaneous**: exp, log, pointwise-{add, subtract, multiply, divide}
- We want to maintain the accuracy of these operations, even though they will not run on Tensor Cores

POINTWISE AND REDUCTION OPS

Principles

- Tensor Cores increase precision in two ways:
  1. Each individual multiply is performed in high precision
  2. The sum of the products is accumulated in high precision



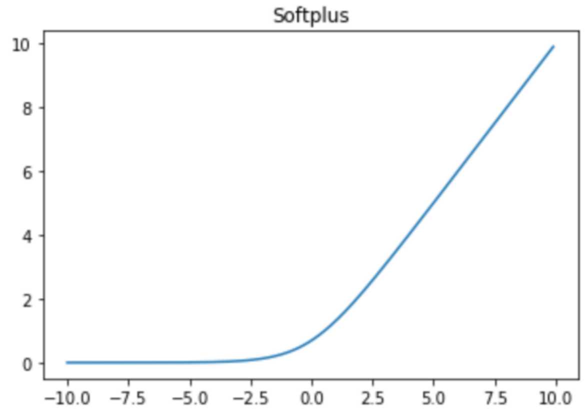
- For non-TC operations, we want to adhere to those same principles:
  1. Keep intermediate or temporary values in high precision
  2. Perform sums (reductions) in high precision

POINTWISE AND REDUCTION OPS

1. Intermediate and temporary values in high precision

- For **pointwise** operations, generally fine to operate directly on FP16 values.
- Exception: FP32 math *and storage* recommended for ops where $|f(x)| \gg |x|$ (or same for grads). Examples: Exp, Log, Pow.
- Most common to see these non-FP16-compatible ops as temporary values in loss or activation functions. Op *fusion* can reduce need for FP32 storage.

```
def softplus(x):
  return log(1 + exp(x))
```



POINTWISE AND REDUCTION OPS

2. Perform sums / reductions in high precision

- Common to normalize a large set of FP16 values in, e.g., a softmax layer
- Two choices :
  - Sum all the values directly into an FP16 accumulator, then perform division in FP16
  - Perform math in high precision (FP32 accumulator, division), then write the final result in FP16
- The first introduces the possibility of *compounding* precision error
- The second does what Tensor Cores do: limit reduced precision to final output
  - This is the desired behavior

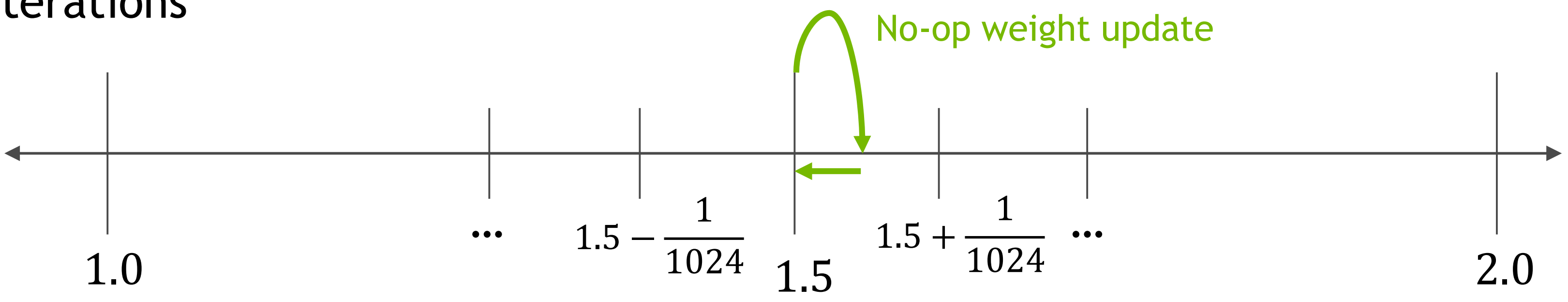
POINTWISE AND REDUCTION OPS

Practical recommendations

- **Nonlinearities:** fine for FP16
    - Except: watch out for exp, log, pow
- **Normalization:** input /output in FP16; intermediate results stored in FP32
    - Ideally: fused into single op. Example: cuDNN BatchNorm
- **Loss functions:** input / output in FP32
    - Also: attention modules (softmax)

2. MASTER WEIGHTS

- At each iteration of training, perform a *weight update* of the form $w_{t+1} = w_t - \alpha \nabla_t$
    - w_t's are weights; ∇_t's are gradients; α is the learning rate
- As a rule, gradients are smaller than weights, and learning rate is less than one
- Consequence: weight update *can be* a no-op, since you can't get to next representable value
- Conservative solution: keep a high-precision copy of weights so small updates accumulate across iterations



3. LOSS SCALING

Range representable in FP16: ~40 powers of 2

Gradients are small:

Some lost to zero

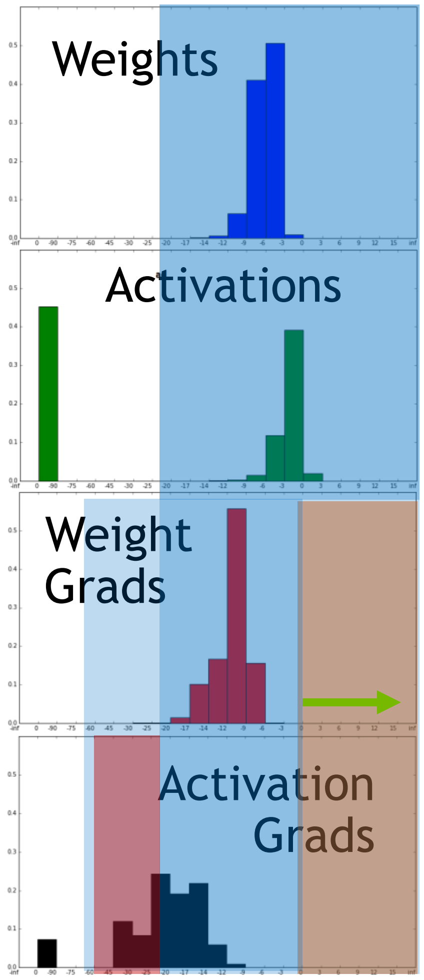
While ~15 powers of 2 remain unused

Loss scaling:

Multiply loss by a constant S

All gradients scaled up by S (chain rule)

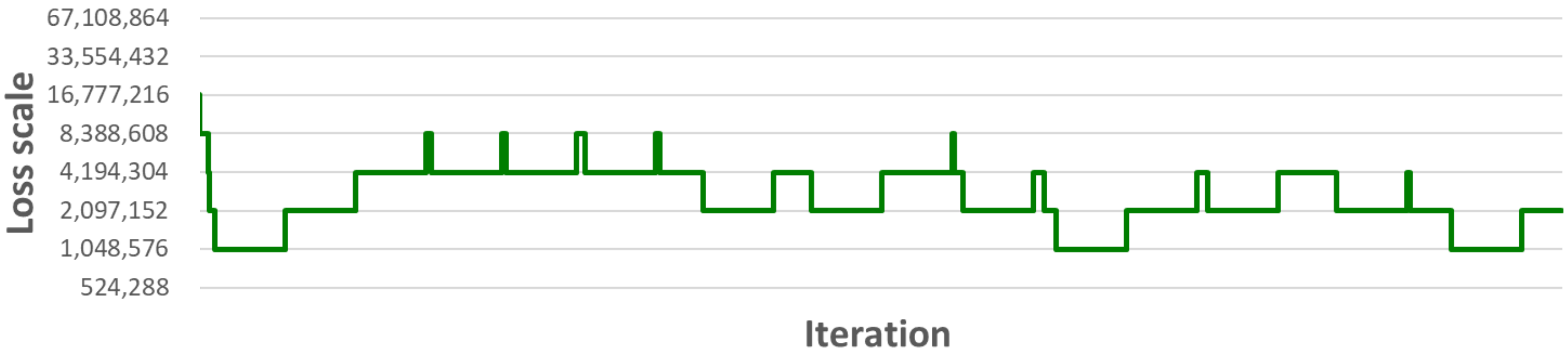
Unscale weight gradient (in FP32) before weight update



3. LOSS SCALING

Automatically choosing a scale factor S

- Intuition:
  - Start with a very large scale factor
  - If an Inf or a NaN is present in the gradient, decrease the scale
    And skip the update, including optimizer state
  - If no Inf or NaN has occurred for some time, increase the scale



3. LOSS SCALING

Automatic scaling: our recommendation

- Many possible settings of algorithm specifics – in our experience, a *wide* range of values below all work equally well
  - Contrast with: learning rate tuning
- Specific values we recommend:
  - Initialize loss scale to 2^24
  - On single overflow, multiply scale by 0.5
  - After 2000 iterations with no overflow, multiply scale by 2.0
    - Note: implies a skip rate of 1/2000 in steady-state
- Described in detail at https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html#scalefactor

OUTLINE

1. What is mixed precision training?
2. Considerations and methodology for mixed precision training
3. Automatic mixed precision
4. Performance guidelines and practical recommendations

ENABLING MIXED PRECISION

Review: recipe for FP16

- Model conversion:
  - Switch everything to run on FP16 values
  - Insert casts to FP32 for loss function and normalization / pointwise ops that need full precision
- Master weights:
  - Keep FP32 model parameters
  - Insert casts to use FP16 copies during forward / backward passes of the model
- Loss scaling:
  - Scale the loss value, unscale the gradients in FP32
  - Check gradients at each iteration to adjust loss scale and skip on overflow

AUTOMATING MIXED PRECISION

Automate *everything* from the previous slide

- Key observation: nothing in the recipe requires domain-specific knowledge
- Instead, *framework software itself* can transform existing model code to run with mixed precision fully automatically
- Details vary by framework, but the core ideas are simple:
  - **Automatic loss scaling** with optimizer wrapping
    - Straightforward: create a wrapper object that manipulates loss and gradients in such a way the base optimizer only ever sees the true FP32 gradient values on non-overflow iterations
  - **Automatic casting** with op classification
- Framework specifics in subsequent talks (see slide 30 for reference)

AUTOMATIC CASTING

Basic idea

- Details vary by framework, but all of them provide an interface of operations that transform or mutate tensor data
- What we want:
  - A static graph of all operations that occur during training
  - An oracle that identifies the optimal type for each operation
    - Maximize speed under the constraint of full accuracy
- We can do without a static graph by making *runtime* type decisions
- We can do without an oracle by pre-committing to a *conservative* set of rules
  - In practice, however, these rules almost always match "by-hand" mixed precision

AUTOMATIC CASTING

Operation classification

- We divide the universe of operations into three kinds:
  - **Whitelist**: ops for which using FP16 enables Tensor Core acceleration
    - Eg: MatMul, Conv2d
  - **Blacklist**: ops for which FP32 is required for accuracy
    - Eg: Exp, Sum, Softmax, Weight updates
  - **Everything else**: ops that *can* run in FP16, but only worthwhile if inputs already FP16
    - Eg: Relu, Add (pointwise), MaxPool

AUTOMATIC CASTING

Operation classification

- Given these lists, we can use simple rules to make types decisions, either in a static graph or at runtime:
    - Whitelist: always run in FP16, casting if necessary
    - Blacklist: always run in FP32, casting if necessary
    - Everything else: run in the existing input type
- In practice, these rules capture the same intuition as “by-hand” conversion:
    - Cast inputs and create weight copies to use FP16 and run on Tensor Cores
    - Keep activations in FP16 so long as pointwise ops do not require full precision
    - Cast to FP32 to compute the loss

MORE ON AUTOMATIC MIXED PRECISION

Talks later today

- PyTorch: “Automatic Mixed Precision in PyTorch” [S9998]
  - 1:00 – 1:50pm, Room 210A
- MXNet: “MXNet Computer Vision and Natural Language Processing Models Accelerated with NVIDIA Tensor Cores” [S91003]
  - 2:00 – 2:50pm, Room 210A
- TensorFlow: “Automated Mixed-Precision Tools for TensorFlow Training” [S91029]
  - 3:00 – 3:50pm, Room 210A

OUTLINE

1. What is mixed precision training?
2. Considerations and methodology for mixed precision training
3. Automatic mixed precision
4. Performance guidelines and practical recommendations

DEBUGGING MIXED PRECISION

Notes and what to watch for

- The “unreasonable effectiveness of gradient descent”
  - Bugs in code for mixed precision steps *often* manifest as slightly worse training accuracy
- Be sure to follow good software engineering practices, especially testing
- Common mistakes:
  - Gradients not unscaled correctly before weight update (AdaGrad / Adam will try to handle this!)
  - Gradient clipping or regularization improperly using scaled gradients
  - Incorrectly synchronizing master weight updates across multiple GPUs
  - Not running loss function in FP32
- *Highly* recommend using automatic mixed precision tools

GETTING THE MOST FROM TENSOR CORES

Performance guidelines

- Three levels of optimization to best use Tensor Cores:
    1. Satisfy Tensor Core shape constraints
    2. Increase arithmetic intensity
    3. Decrease fraction of work in non-Tensor Core ops

GETTING THE MOST FROM TENSOR CORES

Satisfy Tensor Core shape constraints

- Matrix multiplication:
    - All three dimensions (M, N, K) should be multiples of 8
- Convolution:
    - Number of *channels* for input and output should be multiples of 8
        - Note: this isn't always required. See https://devblogs.nvidia.com/tensor-ops-made-easier-in-cudnn/.

GETTING THE MOST FROM TENSOR CORES

Satisfy Tensor Core shape constraints

- In practice:
  - Choose minibatch a multiple of 8
  - Choose layer dimensions to be multiples of 8
  - For classification problems, pad vocabulary to a multiple of 8
  - For sequence problems, pad sequence length to a multiple of 8
- “Am I using Tensor Cores?”
  - cuBLAS and cuDNN are optimized for Tensor Cores, coverage is always increasing
  - Run with nvprof and look for “s[some digits]” in kernel name
    - Eg: `volta_fp16_`**`s884`**`gemm_fp16_128x128_ldg8_f2f_nn`

GETTING THE MOST FROM TENSOR CORES

Increase arithmetic intensity

- *Arithmetic intensity* is the amount of math per byte of input data
- Simple math for why we care about arithmetic intensity:
    - V100 GPU has 125TFLOPs math throughput, 900 GB/s memory bandwidth
    - If there are fewer than ~140 FLOPs per input byte, then memory bandwidth is limiting factor
        - As FLOPs/byte decreases below the threshold of ~140, Tensor Core acceleration decreases too

GETTING THE MOST FROM TENSOR CORES

Increase arithmetic intensity

- Increase arithmetic intensity in model *implementation*:
  - Concatenate weights and gate activations in recurrent cells
  - Concatenate activations across time in sequence models
- Increase arithmetic intensity in model *architecture*:
  - Prefer dense math (vanilla convolutions vs. depth separable convolutions)
  - Prefer wider layers – often little speed cost
  - Of course, always prefer accuracy first!

GETTING THE MOST FROM TENSOR CORES

Decrease non-Tensor Core work

- If 50% of the training routine runs on Tensor Cores, then the maximum speedup is 2x, even if Tensor Cores were infinitely fast
  - This is a simple consequence of Amdahl's Law
- Can speed up non-Tensor Core ops by hand
  - Custom CUDA op implementation + framework integration
- Cutting-edge work on speeding up non-Tensor Core ops automatically with compiler tools
  - TensorFlow: XLA
  - PyTorch JIT

GETTING THE MOST FROM TENSOR CORES

Learn more

- “Tensor Core Performance: The Ultimate Guide” [S9926]
    - Tomorrow (Mar. 19th), 3:00 – 3:50pm, Marriott Hotel Ballroom 4

RESOURCES

- Model implementations, including mixed precision: https://developer.nvidia.com/deep-learning-examples
- Automatic mixed precision: https://developer.nvidia.com/automatic-mixed-precision
- Reading:
    - “Mixed Precision Training” (ICLR 2018): https://arxiv.org/abs/1710.03740
    - Mixed precision guide: https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html

CONCLUSION

- Mixed precision training is a general-purpose technique with tremendous benefits:
  - Math and memory speedups
  - Memory savings, enabling larger models (or minibatches)
- Accuracy matches FP32 training across a *wide* range of models (all we have tried)
- Significant speedups are common and getting more common with each new library release
- Enabling mixed precision depends on a specific methodology:
  - Model conversion with special care for pointwise and reduction ops
  - Safe updates with master weights and loss scaling
- Frameworks have support to fully automate the methodology

NVIDIA®