Agenda

A100 Tensor Cores and Tensor Float 32 (TF32)

Mixed Precision Tensor Cores: Recap and New Advances

Accuracy and Performance Considerations
MOTIVATION - COST OF DL TRAINING

Vision tasks: ImageNet classification
- 2012: AlexNet trained on 2 GPUs for 5-6 days
- 2017: ResNeXt-101 trained on 8 GPUs for over 10 days
- 2019: NoisyStudent trained with ~1k TPUs for 7 days

Language tasks: LM modeling
- 2018: BERT trained on 64 GPUs for 4 days
- Early-2020: T5 trained on 256 GPUs
- Mid-2020: GPT-3

What’s being done to reduce costs
- Hardware accelerators like GPU Tensor Cores
- Lower computational complexity w/ reduced precision or network compression (aka sparsity)
Standard way to represent real numbers on a computer

- Double precision (FP64), single precision (FP32), half precision (FP16/BF16)

**Cannot store numbers with infinite precision, trade-off between range and precision**

- Represent values at widely different magnitudes (range)
  - Different tensors (weights, activation, and gradients) when training a network
- Provide same relative accuracy at all magnitudes (precision)
  - Network weight magnitudes are typically $O(1)$
  - Activations can have orders of magnitude larger values

**How floating-point numbers work**

- **exponent**: determines the range of values
  - Scientific notation in binary (base of 2)
- **fraction (or mantissa)**: determines the relative precision between values
  - $(2^{\text{mantissa}})$ samples between powers of two (exponent)
A100 TENSOR CORES AND TENSOR FLOAT 32 (TF32)
TENSOR CORES - WHAT ARE THEY

Specialized hardware execution units

- Perform matrix and convolution operations, which represent most fundamental and time-consuming operations for most DL workloads

Scalar vs matrix instructions

- FP32 cores perform scalar instructions: multiplication of an element of A with an element of B
- Tensor Cores perform matrix instructions: multiplication between vectors/matrix of elements at a time

Compared to scalar FP32 operations, Tensor Cores are:

- 8-16x faster (up to 32x faster with sparsity) and more energy efficient

\[
D = AB + C
\]
FLAVORS OF TENSOR CORES

Floating point types (for DL and HPC applications):

- 16-bit inputs: fp16, bfloat16
- 32-bit inputs: TF32 mode
- 64-bit inputs: fp64

Integer types (for quantized DL inference):

- int8, int4, int1

In italic are options that are newly introduced in A100

Integer Quantization for DNN Inference Acceleration

Sparsity (not exactly a type, but also for DL inference):

- 2:4 structure → two elements in a 4-element vector are zero

Accelerating Sparsity in the NVIDIA Ampere Architecture
TENSOR CORE OPTIONS FOR DL TRAINING

TensorFloat (TF32) mode for single-precision training (A100):

- Accelerates only math-limited operations
- Compared to FP32 training
  - 8x higher math throughput
  - Same memory bandwidth pressure
- Does not require any changes to training scripts
  - Default math mode for single-precision training on NVIDIA Ampere GPU Architecture

16-bit formats for mixed-precision training (V100 or A100):

- Fastest option: accelerate math- and memory-limited operations
- Compared to FP32 training:
  - 16x higher math throughput
  - 0.5x memory bandwidth pressure
- Requires some changes to training scripts: fp32 master weights, layer selection, loss-scaling
  - Automatic Mixed Precision (AMP) reduces these changes to just a few lines (TF, PyT, MxNet)
TF32 MODE FOR SINGLE PRECISION TRAINING

TF32 is a Tensor Core mode, not a type

- Only convolutions and matrix multiplies convert inputs to TF32
  - All other operations remain completely FP32
- All storage in memory remains FP32
- Consequently, it’s only exposed as a Tensor Core operation mode
  - Contrast with fp16/bfloat16 types that provide: storage, various math operators, etc

Operation:

- Read FP32 inputs from memory
- Round inputs to TF32 prior to Tensor Core operation
- Multiply inputs without loss of precision
- Accumulate products in FP32
- Write FP32 product to memory
TF32 PRECISION DETAILS

Range (Exponent) 8-bit:
- Matches FP32, covers the same range of values

Precision (Mantissa) 10-bit:
- 1024 samples between powers of 2
- Higher precision than BF16
  - 8x more samples between powers of 2 than BF16
- Only difference from FP32
- Sufficient margin for DL training and results in loss in accuracy as seen across 80+ networks when compared to FP32 and mixed precision modes
TF32 VERIFICATION

Verification on unmodified model scripts for 80+ networks

- **Model architectures:**
  - Convnets, MLPs, RNNs, Transformers, BERT, GANs, etc.
- **Tasks including:**
  - Image tasks (classification, detection, segmentation, generation, gaze)
  - Language tasks (translation, modeling, question answering)
  - Recommenders
  - Meta learning
  - More niche tasks (logic reasoning, combinatorial problems)
- **First and second order methods**

All experiments match FP32 accuracy and loss values
A NOTE ON RUN-TO-RUN VARIATION

DL networks have run-to-run variance during training

- Different seeds affect weight initialization, dropout, etc
- Operations that use atomic adds (e.g. floating-point addition)
- cuDNN heuristics/algorithms
- SW (e.g. container, framework, external libraries)
- Reproducibility in frameworks (e.g. pytorch)

DenseNet201 example

- FP32/TF32 with 60 different seeds
- Visualize data with scatter, sorted from smallest-to-largest, etc
- Accuracy varies up to 0.5% (more for other workloads)
- But FP32/TF32 are statistically equivalent
  Have the same mean and median

<table>
<thead>
<tr>
<th>Precision</th>
<th>Mean</th>
<th>Median</th>
<th>Max</th>
<th>Min</th>
<th>Stdev</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP32</td>
<td>77.53</td>
<td>77.57</td>
<td>77.67</td>
<td>77.29</td>
<td>0.09</td>
</tr>
<tr>
<td>TF32</td>
<td>77.54</td>
<td>77.55</td>
<td>77.79</td>
<td>77.29</td>
<td>0.09</td>
</tr>
</tbody>
</table>
### SAMPLING OF NETWORKS

#### Classification Tasks

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Network</th>
<th>Top-1 Accuracy</th>
<th>Dataset</th>
<th>Metric</th>
<th>Model Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet</td>
<td>RN18</td>
<td>70.43</td>
<td></td>
<td></td>
<td>37.81</td>
</tr>
<tr>
<td></td>
<td>RN32</td>
<td>74.03</td>
<td></td>
<td></td>
<td>37.95</td>
</tr>
<tr>
<td></td>
<td>RN50</td>
<td>76.78</td>
<td></td>
<td></td>
<td>42.05</td>
</tr>
<tr>
<td></td>
<td>RN101</td>
<td>77.57</td>
<td></td>
<td></td>
<td>42.14</td>
</tr>
<tr>
<td>ResNext</td>
<td>RX50</td>
<td>77.51</td>
<td></td>
<td></td>
<td>37.89</td>
</tr>
<tr>
<td></td>
<td>RXN101</td>
<td>79.10</td>
<td></td>
<td></td>
<td>37.89</td>
</tr>
<tr>
<td>WideResNet</td>
<td>WRN50</td>
<td>77.99</td>
<td></td>
<td></td>
<td>34.65</td>
</tr>
<tr>
<td></td>
<td>WRN101</td>
<td>78.61</td>
<td></td>
<td></td>
<td>34.69</td>
</tr>
<tr>
<td>DenseNet</td>
<td>DN121</td>
<td>75.57</td>
<td></td>
<td></td>
<td>35.16</td>
</tr>
<tr>
<td></td>
<td>DN169</td>
<td>76.75</td>
<td></td>
<td></td>
<td>35.25</td>
</tr>
<tr>
<td>VGG</td>
<td>V11-BN</td>
<td>68.47</td>
<td></td>
<td></td>
<td>34.65</td>
</tr>
<tr>
<td></td>
<td>V16-BN</td>
<td>71.54</td>
<td></td>
<td></td>
<td>34.69</td>
</tr>
<tr>
<td></td>
<td>V19</td>
<td>72.54</td>
<td></td>
<td></td>
<td>35.16</td>
</tr>
<tr>
<td></td>
<td>V19</td>
<td>71.75</td>
<td></td>
<td></td>
<td>35.25</td>
</tr>
<tr>
<td>GoogleNet</td>
<td>InceptionV3</td>
<td>77.20</td>
<td></td>
<td></td>
<td>34.65</td>
</tr>
<tr>
<td></td>
<td>Xception</td>
<td>79.09</td>
<td></td>
<td></td>
<td>34.69</td>
</tr>
<tr>
<td>Dilated RN</td>
<td>DRN A 50</td>
<td>78.24</td>
<td></td>
<td></td>
<td>34.65</td>
</tr>
<tr>
<td>ShuffleNet</td>
<td>V2-X1</td>
<td>68.62</td>
<td></td>
<td></td>
<td>34.65</td>
</tr>
<tr>
<td></td>
<td>V2-X2</td>
<td>73.02</td>
<td></td>
<td></td>
<td>34.69</td>
</tr>
<tr>
<td>MNASNet</td>
<td>V1_0</td>
<td>71.62</td>
<td></td>
<td></td>
<td>35.16</td>
</tr>
<tr>
<td>SqueezeNet</td>
<td>V1_1</td>
<td>60.90</td>
<td></td>
<td></td>
<td>35.25</td>
</tr>
<tr>
<td>MobileNet</td>
<td>MM-V2</td>
<td>71.64</td>
<td></td>
<td></td>
<td>34.65</td>
</tr>
<tr>
<td>Stacked UNet</td>
<td>SUN64</td>
<td>69.53</td>
<td></td>
<td></td>
<td>34.69</td>
</tr>
<tr>
<td>EfficientNet</td>
<td>B0</td>
<td>76.79</td>
<td></td>
<td></td>
<td>34.69</td>
</tr>
</tbody>
</table>

#### Detection & Segmentation Tasks

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Network</th>
<th>Metric</th>
<th>Model Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faster RCNN</td>
<td>RN50 FPN 1X</td>
<td>mAP</td>
<td>37.81</td>
</tr>
<tr>
<td></td>
<td>RN101 FPN 3X</td>
<td>mAP</td>
<td>40.04</td>
</tr>
<tr>
<td></td>
<td>RN50 FPN 3X</td>
<td>mAP</td>
<td>42.05</td>
</tr>
<tr>
<td>Mask RCNN</td>
<td>TorchVision</td>
<td>mAP</td>
<td>37.89</td>
</tr>
<tr>
<td></td>
<td>RN50 FPN 1X</td>
<td>miOU</td>
<td>34.65</td>
</tr>
<tr>
<td></td>
<td>RN50 FPN 3X</td>
<td>miOU</td>
<td>34.69</td>
</tr>
<tr>
<td>Retina Net</td>
<td>RN50 FPN 1X</td>
<td>mAP</td>
<td>38.45</td>
</tr>
<tr>
<td></td>
<td>RN50 FPN 3X</td>
<td>mAP</td>
<td>41.04</td>
</tr>
<tr>
<td></td>
<td>RN101 FPN 3X</td>
<td>mAP</td>
<td>42.99</td>
</tr>
<tr>
<td>RPN</td>
<td>RN50 FPN 1X</td>
<td>mAP</td>
<td>36.46</td>
</tr>
<tr>
<td></td>
<td>RN50 FPN 1X</td>
<td>mAP</td>
<td>38.04</td>
</tr>
<tr>
<td></td>
<td>RN101 FPN 3X</td>
<td>mAP</td>
<td>39.75</td>
</tr>
<tr>
<td>Single-Shot Detector (SSD)</td>
<td>RN18</td>
<td>mAP</td>
<td>19.13</td>
</tr>
<tr>
<td></td>
<td>RN50</td>
<td>mAP</td>
<td>24.91</td>
</tr>
</tbody>
</table>

**Dataset is MS COCO 2017**

#### Language Tasks

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Network</th>
<th>Dataset</th>
<th>Metric</th>
<th>Model Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer</td>
<td>Vaswani Base</td>
<td>WMT</td>
<td>BLEU</td>
<td>27.18</td>
</tr>
<tr>
<td></td>
<td>Vaswani Large</td>
<td>WMT</td>
<td>BLEU</td>
<td>28.63</td>
</tr>
<tr>
<td></td>
<td>Levenshtein</td>
<td>WMT</td>
<td>BLEU</td>
<td>6.16</td>
</tr>
<tr>
<td>Convolutional</td>
<td>Light Conv Base</td>
<td>WMT</td>
<td>BLEU</td>
<td>28.55</td>
</tr>
<tr>
<td></td>
<td>Light Conv Large</td>
<td>WMT</td>
<td>BLEU</td>
<td>30.10</td>
</tr>
<tr>
<td></td>
<td>Dynamic Conv Base</td>
<td>WMT</td>
<td>BLEU</td>
<td>28.34</td>
</tr>
<tr>
<td></td>
<td>Dynamic Conv Large</td>
<td>WMT</td>
<td>BLEU</td>
<td>30.31</td>
</tr>
<tr>
<td>Recurrent</td>
<td>GNMT</td>
<td>WMT</td>
<td>BLEU</td>
<td>24.53</td>
</tr>
<tr>
<td></td>
<td>FairSeq Conv</td>
<td>Wikipedia</td>
<td>PPL</td>
<td>35.80</td>
</tr>
<tr>
<td>Transformer</td>
<td>XL Standard</td>
<td>WikiText</td>
<td>PPL</td>
<td>22.89</td>
</tr>
<tr>
<td>BERT</td>
<td>Base Pre-train</td>
<td>Wikipedia</td>
<td>LA Loss</td>
<td>1.34</td>
</tr>
<tr>
<td></td>
<td>Base Downstream</td>
<td>SQUAD v1</td>
<td>F1</td>
<td>87.95</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQUAD v2</td>
<td>F1</td>
<td>76.68</td>
</tr>
</tbody>
</table>

**Dataset is ISLVRC 2012**

---

**No hyperparameter changes**

Differences in accuracy are within typical bounds of run-to-run variation (different random seeds, etc.)
LOSS AND ACCURACY CURVES FOR RESNEXT-101
LOSS AND ACCURACY CURVES FOR MASKRCNN WITH RN101 BACKBONE
LOSS AND ACCURACY CURVES FOR TRANSFORMER XL
SAME ACCURACIES FOR FP32, TF32 AND FP16

Results are easily reproducible using NGC containers and Deep Learning Examples.
SAMPLE OF TRAINING SPEEDUPS

- 4-6x faster for transformer-based architectures
- >3x for recurrent networks
- About 2x for convolutional models

All models can be found at:
Source: https://github.com/NVIDIA/DeepLearningExamples/
All performance collected on DGX A100 (8xA100)
Results can be reproduced with PyTorch 1.6 and TensorFlow 1.15 in NGC containers pytorch:20.06-py3, tensorflow:20.06-tf1-py3
TF32 - ON BY DEFAULT

No changes needed to use TF32 and get up to 6X speedup

Supported for TensorFlow, PyTorch and MXNet:
- Default mode for A100 from 20.06 Nvidia container release
- Upstream support in progress
- IEEE FP32 paths remain selectable for non-DL operations
  (i.e. HPC applications, some use of GEMM in frameworks for solvers such as LU decomposition etc)

TF32 is enabled for:
- Single-precision convolution and matrix-multiply layers including linear/fully-connected layers, recurrent cells, attention blocks

TF32 is not enabled for:
- Convolution or matrix-multiply layers that operate on non-FP32 tensors
- Any layers that are not convolutions or matrix-multiplies
- Optimization/solver operations
GLOBAL PLATFORM CONTROL FOR TF32

Global variable **NVIDIA_TF32_OVERRIDE** to toggle TF32 mode at system level (and override libraries/frameworks)

<table>
<thead>
<tr>
<th>NVIDIA_TF32_OVERRIDE=0</th>
<th>Not Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disables TF32 so that FP32 is used</td>
<td>Defaults to library and framework settings</td>
</tr>
</tbody>
</table>

Debugging tool

- quick way to rule out any concern regarding TF32 libraries and look for other issues
BEHAVIOR OF TF32 IN LIBRARIES FOR A100

For developers using NVIDIA libraries

<table>
<thead>
<tr>
<th>cuDNN &gt;= 8.0</th>
<th>cuBLAS &gt;= 11.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Convolutions</strong></td>
<td><strong>Linear algebra operations</strong></td>
</tr>
<tr>
<td>TF32 is the default math</td>
<td>Default math mode is FP32 because of HPC</td>
</tr>
<tr>
<td>TF32 kernels selected when</td>
<td>TF32 enabled when math mode set to</td>
</tr>
<tr>
<td>operating on 32-bit data</td>
<td>CUBLAS_TF32_TENSOR_OP_MATH *</td>
</tr>
</tbody>
</table>

* Places guards around solver operations in DL frameworks to keep math in FP32

1. Cache current cuBLAS state
2. Set cuBLAS math mode to FP32
3. Execute solver operation
4. Restore original cuBLAS state
CHOOSING SINGLE-PRECISION TRAINING ON A100

Great starting point if you used FP32 training on Volta and other processors

A100 hardware provides up to 10X speedup over Volta default

TF32 is on by default, does not require changes in training scripts

Same accuracy as FP32
MIXED PRECISION TENSOR CORES RECAP AND NEW ADVANCES
TENSOR CORES FOR 16-BIT FORMATS

Fastest way to train networks

Operation:

- Multiply and add FP16 or BF16 tensors
- Products are computed without loss of precision, accumulated in FP32
- Final FP32 output is rounded to FP16/BF16 before writing to memory

NVIDIA Ampere Architecture enhancements:

- New tensor core design: 2.5x throughput for dense operations (A100 vs V100)
- Sparsity support: additional 2x throughput for sparse operations
- BFloat16 (BF16): Same rate as FP16
MIXED PRECISION TRAINING

Combines single-precision (FP32) with lower precision (e.g. FP16) when training a network

• Use lower precision where applicable (e.g. convolutions, matrix multiplies)
• Keep certain operations in FP32

Achieves the same accuracy as FP32 training using all the same hyper-parameters
BENEFITS OF MIXED PRECISION TRAINING

Accelerates math-intensive operations with specialized hardware (GPU Tensor Cores)
- FP16/BF16 have 16x higher throughput than FP32

Accelerates memory-intensive operations by reducing memory traffic
- 16-bits require half number of bytes to be read/written to memory

Reduces memory requirements
- 16-bits reduce storage of activation and gradient tensors
- Enables training of larger models, larger mini-batches, larger inputs
BENEFITS OF MIXED PRECISION TRAINING

Accelerates math-intensive operations with specialized hardware (GPU Tensor Cores)

- FP16/BF16 have 16x higher throughput than FP32

Accelerates memory-intensive operations by reducing memory traffic

- 16-bits require half number of bytes to be read/written to memory

Reduces memory requirements

- 16-bits reduce storage of activation and gradient tensors
- Enables training of larger models, larger mini-batches, larger inputs

Benefits unique to 16-bit mixed-precision, not offered by TF32
SAMPLING OF NETWORKS TRAINED IN MIXED PRECISION

3 years of networks trained with 16-bit formats

Proven to match FP32 results across a wide range of tasks, problem domains, deep neural network architectures

<table>
<thead>
<tr>
<th>Image Classification</th>
<th>Detection / Segmentation</th>
<th>Generative Models (Images)</th>
<th>Language Modeling</th>
<th>Speech</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>DeepLab</td>
<td>DLSS</td>
<td>BERT</td>
<td>Deep Speech 2</td>
<td>Convolutional Seq2Seq</td>
</tr>
<tr>
<td>DenseNet</td>
<td>Faster R-CNN</td>
<td>Vid2vid</td>
<td>GPT</td>
<td>Jasper</td>
<td>Dynamic Convolutions</td>
</tr>
<tr>
<td>Inception</td>
<td>Mask R-CNN</td>
<td>GauGAN</td>
<td>TrellisNet</td>
<td>Tacotron</td>
<td>GNMT (RNN)</td>
</tr>
<tr>
<td>MobileNet</td>
<td>SSD</td>
<td>Partial Image Inpainting</td>
<td>Gated Convolutions</td>
<td>Wave2vec</td>
<td>Levenshtein Transformer</td>
</tr>
<tr>
<td>EfficientNet</td>
<td>NVidia Automotive</td>
<td>Progress GAN</td>
<td>BigLSTM/mlLSTM</td>
<td>WaveNet</td>
<td>Transformer (Self-Attention)</td>
</tr>
<tr>
<td>ResNet</td>
<td>RetinaNet</td>
<td>Pix2Pix</td>
<td>RoBERTa</td>
<td>WaveGlow</td>
<td></td>
</tr>
<tr>
<td>ResNeXt</td>
<td>UNET</td>
<td></td>
<td>Transformer XL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShuffleNet</td>
<td>DETR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SqueezeNet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VGG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xception</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dilated ResNet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stacked U-Net</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SAMPLE OF ACHIEVED TRAINING SPEEDUPS

V100 mixed precision is between 2x to 6x faster than V100 single precision training

A100 mixed precision gives an additional 2-3x

All models can be found at:
Source: https://github.com/NVIDIA/DeepLearningExamples/
All performance collected on DGX V100/A100
Results can be reproduced with PyTorch 1.6 and TensorFlow 1.15 in NGC containers pytorch:20.06-py3, tensorflow:20.06-tf1-py3
MANY SUCCESS STORIES USING MIXED PRECISION

Generative Models

Mixed Precision improves NVIDIA GauGAN, the viral AI tool that uses GANs to convert segmentation maps into lifelike images
- Reduces training from 21 days to less than 10 days
- Larger generative models improve visual quality
- High-res images using larger inputs

Machine Translation

Mixed precision helps Facebook speedup training for machine translation tasks (Fairseq) by 5x due to faster math and large batch training

Computer Vision

Mixed Precision being used as the default training option for DL workloads for a number of customers
- 2-3X faster training of AI models

Language Modeling

Mixed Precision fuels research on the largest Transformer models for state-of-the-art NLP
- Megatron → Turing-NLG → GPT-3 (8B → 17B → 175B)
- Reduce training time and memory storage
MIXED PRECISION CONSIDERATIONS

Considerations for training with 16-bit formats:

- **LAYER SELECTION**: Decide which operations to compute in FP32/16-bits
- **WEIGHT STORAGE**: Keep model weights and updates in FP32
- **LOSS SCALING**: Retain small gradient magnitudes for FP16
KINDS OF OPERATIONS

8-16x acceleration from FP16/BF16 Tensor Cores

Matrix Multiplications
linear, matmul, bmm, conv

Loss Functions
cross entropy, l2 loss, weight decay

Reductions
batch norm, layer norm, sum, softmax

Pointwise
relu, sigmoid, tanh, exp, log

2x acceleration with 16-bit formats (but should not sacrifice accuracy)
RECOMMENDATIONS THAT ARE INTEGRATED INTO AMP

Operations that can use 16-bit storage (FP16/BF16)
- Matrix multiplications
- Most pointwise operations (e.g. relu, tanh, add, sub, mul)

Operations that need more precision (FP32/FP16)
- Adding small values to large sums can lead to rounding errors
- Reduction operations (e.g. sum, softmax, normalization)

Operations that need more range (FP32/BF16)
- Pointwise operations where $|f(x)| \gg |x|$ (e.g. exp, log, pow)
- Loss functions
Weight updates can become too small for addition in FP16/BF16 during late stages of training.

Update gets clipped to zero when weights (w) >> weight update ($\alpha \nabla$).

**Conservative default**: keep weights in FP32 so that small updates accumulate across iterations.

$$w_{t+1} = w_t - \alpha \nabla_t$$
FP32 WEIGHT STORAGE AND UPDATES IN FRAMEWORKS

Weights are *always* stored in FP32

Make an FP16 copy of weight during the forward pass (for linear and conv layers)

Optimizer performs weight gradient updates in FP32
LOSS SCALING KEEPS TENSORS WITHIN REPRESENTABLE RANGE

Weights, activations, and gradients have wide range of values

Range representable in FP16

Gradients are small

some *lost to zero*

can affect network accuracy

but most of range *remains unused*

implies its not a dynamic range problem

Move small gradient values to FP16 range

multiply loss by a constant factor

all gradients are scaled (shifted) by chain rule

![Graphs showing distribution of weights, activations, and gradients.](image-url)
1. Forward pass of the model

2. **Scale the loss** and backpropagate the scaled gradients

3. **Un-scale the gradients** and optimizer performs the weight update
AUTOMATIC LOSS SCALING

1. Start with a very large scale factor (e.g. FP16 max)

2. If gradients overflow (with inf or nan)
   - Decrease the scale by two and skip the update

3. If no overflows have occurred for some time (e.g. 2k iterations)
   - Increase the scale by two

https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html#scalefactor
**AUTOMATIC MIXED PRECISION FOR 16-BITS**

**Automatic Mixed Precision (AMP)** makes mixed precision training with FP16 easy in frameworks

- AMP automates process of training in mixed precision
- Example: Converts matrix multiplies/convolutions to 16-bits for Tensor Core acceleration

Works with multiple models, optimizers, and losses
## AMP SUPPORT IN FRAMEWORKS AND CONTAINERS

<table>
<thead>
<tr>
<th>Framework</th>
<th>Availability</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow</td>
<td>Available in TF 1.14+, TF 2+, and NVIDIA Container 19.07+. Documentation can be found here:</td>
<td><a href="https://tensorflow.org/guide/mixed_precision">https://tensorflow.org/guide/mixed_precision</a></td>
</tr>
</tbody>
</table>
| PyTorch        | Native support available in PT 1.6+ and NVIDIA Container 20.06+. Documentation can be found here: | [https://pytorch.org/docs/stable/amp.html](https://pytorch.org/docs/stable/amp.html)  
| MXNet          | Available in MXNet 1.5+ Contrib, NVIDIA Container 19.04+. Documentation can be found here: | [https://mxnet.apache.org/api/python/docs/tutorials/performance/backend/amp.html](https://mxnet.apache.org/api/python/docs/tutorials/performance/backend/amp.html) |
AMP FOR TENSORFLOW USING GRAPH OPTIMIZATION API

Recommended option for TensorFlow 1.x

Optimization wrapper

- Graph optimization pass that converts (the type of) certain fp32 operations to fp16 in the TF backend
- Loss-scale optimizer

Example

```python
model = tf.keras.models.Sequential([...])

opt = tf.keras.optimizers.SGD()
opt = tf.train.experimental.enable_mixed_precision_graph_rewrite(opt)

model.compile(loss="cross_entropy", optimizer=opt, metrics=["accuracy"])

model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs)
```
AMP FOR TENSORFLOW USING KERAS MIXED PRECISION API

Recommended option for TensorFlow 2.x

Ability to control precision as the model is constructed for eager and graph execution

For training the model with Model.fit

- Policy determines the type of layer computations and layer variables
  
  ```python
  policy = tf.keras.mixed_precision.experimental.Policy('mixed_float16', loss_scale='dynamic')
  tf.keras.mixed_precision.experimental.set_policy(policy)
  ```

- E.g. `mixed_float16` uses fp16 computations and fp32 variables for numerical stability
- Override the policy/type of layers that are not numerically stable in fp16
  
  ```python
  outputs = layers.Activation('softmax', dtype='float32', name='predictions')(x)
  ```

For training the model with a custom training loop

- need to explicitly use loss scaling w/ `mixed_float16`
AMP FOR APACHE MXNET

Initialize AMP by changing behavior/types of operations

```
amp.init()
```

Wrap the Gluon trainer

```
amp.init_trainer(trainer)
```

Apply automatic loss scaling

- Scale the loss to preserve the gradients

```
with amp.scale_loss(loss, trainer) as scaled_loss:
    autograd.backward(scaled_loss)
```
APEX AMP FOR PYTORCH

AMP is supported in our APEX extension for PyTorch

But recommend using the native PyTorch automatic mixed precision

Patch operations so that they are casted to the correct type

```python
model, optimizer = amp.initialize(model, optimizer, opt_level="O1")
```

Apply automatic loss scaling

- Scale the loss to preserve the gradients

```python
with amp.scale_loss(loss, trainer) as scaled_loss:
    scaled_loss.backward()
```
import torch

# Creates once at the beginning of training
scaler = torch.cuda.amp.GradScaler()

for data, label in data_iter:
    optimizer.zero_grad()

    # Casts operations to mixed precision
    with torch.cuda.amp.autocast():
        loss = model(data)

        # Scales the loss, and calls backward()
        # to create scaled gradients
        scaler.scale(loss).backward()

        # Unscales gradients and calls
        # or skips optimizer.step()
        scaler.step(optimizer)

        # Updates the scale for next iteration
        scaler.update()
BF16 IN LIBRARIES FOR A100

Bfloat16 is accessible in the following ways in CUDA 11

- ptxas (ex: mma.sync)
- Native CUDA C++ datatype called __nv_bfloat16
- CUDA C++ support for WMMA
- CUDA Math Libraries

Conversion options for 16-bits

- Avoid custom conversions as they are prone to bugs
- Recommend using type casts or intrinsic functions
- Must include the appropriate headers (see code example)

```c
#include <cuda_fp16.h>
half a = (half)(1.5f);
half b = (half)(1.0f);
half c = a + b;

#include <cuda_bf16.h>
nv_bfloat16 a = (nv_bfloat16)(1.5f);
nv_bfloat16 b = (nv_bfloat16)(1.0f);
nv_bfloat16 c = a + b;
```


Why should I use it?
Tensor Core acceleration for matrix multiplies
Reduce memory traffic for custom layers
CHOOSING MIXED-PRECISION TRAINING ON A100

Option to use if you:

- Use mixed-precision training (FP16 or BF16) on Volta and other processors
- Are using single-precision on A100 training and want further speedup
- Need memory savings to train larger models

Fastest options for training: up to 2x faster than single-precision with TF32

Requires minimal additions to training scripts with AMP

No impact on accuracy when compared to FP32
ACCURACY AND PERFORMANCE CONSIDERATIONS
MISTAKES TO AVOID WHEN TRAINING WITH MIXED PRECISION

Casting tensors to 16-bits
- Some manually cast to half/float16 for more perf or fix type mismatch
- Avoid manual casts - AMP keeps fp32 weight storage and ensures operations that are safe are computed in fp16

Gradient computations using scaled gradients
- Gradients after backward pass are scaled, and can affect subsequent gradient computations
- Unsacle gradients for any operation that uses gradients (e.g. gradient clipping)
  - `scaler.unscale_(optimizer)`

Not checkpointing and resuming the loss scale
- Automatic loss scaling algorithm starts from a very high loss scale
  - Likely won’t be the same loss scale obtained after sufficient training
- Store AMP loss scale factor to continue training from the same loss scale
  - `checkpoint = {'amp': scaler.state_dict()}
  - `checkpoint = torch.load('checkpoint')`
  - `scaler.load_state_dict(checkpoint['amp'])`
END-TO-END PERF DEPENDS ON TRAINING COMPOSITION

Amdahl’s law: if you speed up part of your training session (GPU work), then the remaining parts (CPU work) limit your overall performance.

<table>
<thead>
<tr>
<th>Single Precision</th>
<th>Mixed Precision (TF32/FP16/BF16)</th>
<th>GPU</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATH (linear, conv, matmul)</td>
<td>MEMORY (pointwise, reductions)</td>
<td>~2x overall faster</td>
<td>~2x overall faster</td>
</tr>
<tr>
<td>8-16x</td>
<td>1-2x</td>
<td>1x</td>
<td>1x</td>
</tr>
</tbody>
</table>
IMPROVING DL TRAINING PERFORMANCE

No single recommendation as perf implications vary across DL workloads

Top-down approach

- three levels of profiling to understand & improve training perf

1. Profile the training session
   - Find time spent on the GPU
   - Reason: Mixed precision only accelerates GPU work
   - Measure time spent on different high-level components of the network (e.g. forward, backward, loss, optimizer)

2. Profile the network layers
   - Measure time spent on different layer types (e.g. that perform matrix math)
   - Reason: Tensor Cores have largest benefits in training perf

3. Profile Tensor Cores
   - Make sure TCs are being used & achieve good efficiency
NVIDIA DEEP LEARNING PROFILER

Designed for analyzing performance of neural networks on DL frameworks

- Provide layer-resolved breakdown of network time
- Determine issues that limit performance, e.g. "Am I using Tensor Cores"

TensorFlow 1 and PyTorch from 20.07+ Nvidia container release

- **TensorFlow 1.x:** `nvcr.io/nvidia/tensorflow:<xx.yy>-tf1-py3`
- **PyTorch:** `nvcr.io/nvidia/pytorch:<xx.yy>-py3`

For PyTorch also add following lines to the model script

```
dlprof python train.py  # Wrap training command line
tensorboard --logdir ./eventsfile  # Visualize on TensorBoard
```

```
import torch.cuda.profiler as profiler
import pyprof
pyprof.init()
```
SIMPLE MODE FOR NVIDIA DEEP LEARNING PROFILER

An easy-to-use profiler from 20.06+ Nvidia container release

Can profile any program or python script and is agnostic to the framework
  - useful for DL/ML researchers using other DL frameworks

Provides basic metrics for understanding mixed precision performance

# Wrap training command line with DLPROF
dlprof --mode=simple python train.py

> Total Wall Clock Time (ns): 25812693595  # Time spent on the entire session
> Total GPU Time (ns): 19092416468   # Time spent on GPU work
> Total Tensor Core Kernel Time (ns): 10001024991 # Time spent on Tensor Cores

https://docs.nvidia.com/deeplearning/frameworks/dlprof-user-guide/
FRACTION OF TRAINING SESSION SPENT ON THE GPU

GPU time can be obtained w/ DLProf simple mode

How to profile different portions of model code

```python
start = time.time()  # start timer
loss = model.forward()  # code to be profiled
loss.backward()  #
torch.cuda.synchronize()  # wait for GPU work to complete
bwd_time = start - time.time()  # compute elapsed time
```

A few things to keep in mind

- Skip measurements of the first few iterations
- Average time over tens of iterations to account for variance
- Compute speedups over the same mini-batches for FP32 & AMP

Common pitfalls

- Small batches or models that don’t saturate GPU resources
- Unoptimized bits of model code (e.g. data pre-processing or loss computation)
SPEEDUP DEPENDS ON NETWORK COMPOSITION

Network computations can be broken down into

1. Memory-bound layers
   - Accelerated for FP16/BF16 16-bit formats
   - Can get up to 2x from reduced memory traffic
   - e.g. losses, activations, normalizations, pointwise

2. Math-bound layers
   - Accelerated for TF32/FP16/BF16 Tensor Cores
   - Can get up to 8-16x from faster matrix math
   - e.g. linear, matmul, batched gemms, convolutions

DLProf to find time breakdown of the network (see right)
   - Correlates GPU kernels/functions with network ops or layers

---

Layer breakdown for Pix2PixHD

<table>
<thead>
<tr>
<th>Network</th>
<th>FP32</th>
<th>AMP</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (ns)</td>
<td>% of Total</td>
<td>Time (ns)</td>
</tr>
<tr>
<td>conv2d</td>
<td>1903349214</td>
<td>82.66</td>
<td>652519136</td>
</tr>
<tr>
<td>instancenorm</td>
<td>99453870</td>
<td>4.32</td>
<td>71592641</td>
</tr>
<tr>
<td>pad</td>
<td>79588125</td>
<td>3.46</td>
<td>79102365</td>
</tr>
<tr>
<td>relu</td>
<td>44597183</td>
<td>1.94</td>
<td>28446370</td>
</tr>
<tr>
<td>l1_loss</td>
<td>27966155</td>
<td>1.21</td>
<td>26933653</td>
</tr>
<tr>
<td><strong>truediv</strong></td>
<td>19459864</td>
<td>0.85</td>
<td>19177567</td>
</tr>
<tr>
<td>max_pool2d</td>
<td>16249430</td>
<td>0.71</td>
<td>12963561</td>
</tr>
<tr>
<td>Interpolate</td>
<td>16062200</td>
<td>0.70</td>
<td>12117584</td>
</tr>
<tr>
<td>mv</td>
<td>12737865</td>
<td>0.55</td>
<td>7773164</td>
</tr>
<tr>
<td>add</td>
<td>11288118</td>
<td>0.49</td>
<td>8080203</td>
</tr>
<tr>
<td>add_</td>
<td>8816455</td>
<td>0.38</td>
<td>5519740</td>
</tr>
<tr>
<td>leaky_relu</td>
<td>8578987</td>
<td>0.37</td>
<td>5308481</td>
</tr>
<tr>
<td>sum</td>
<td>8150963</td>
<td>0.35</td>
<td>7294031</td>
</tr>
<tr>
<td>cat</td>
<td>7054962</td>
<td>0.31</td>
<td>6878672</td>
</tr>
<tr>
<td>mul_</td>
<td>6539092</td>
<td>0.28</td>
<td>6513756</td>
</tr>
<tr>
<td><strong>add</strong></td>
<td>5630074</td>
<td>0.24</td>
<td>4482021</td>
</tr>
<tr>
<td>Interpolate</td>
<td>5457247</td>
<td>0.24</td>
<td>5328349</td>
</tr>
</tbody>
</table>
TIME BREAKDOWN BETWEEN NETWORK LAYERS

- **0. Most time spent on math bound layers**
  - Linear: 4.8x speedup
  - Matmul: 3.2x speedup
  - Softmax: 1.9x speedup
  - Dropout: 1.7x speedup
  - Layer norm: 1.6x speedup

- **1. Good end-to-end perf because of Tensor Core speedup**

- **2. Up to 2x speedup from memory bound layers**

BERT Large, Single V100
TIME BREAKDOWN BETWEEN NETWORK LAYERS

- **TOTAL**
  - >3x

- **LINEAR**
  - 4.8x

- **MATMUL**
  - 3.2x

- **SOFTMAX**
  - 1.9x

- **DROPOUT**
  - 1.7x

- **LAYER NORM**
  - 1.6x

**Recommendation:** Have the network spend more time on math-bound layers

- **2. Up to 2x speedup from memory bound layers**
- **3. Good end-to-end perf because of Tensor Core speedup**

**Network Breakdown**

- **FP32**
- **AMP**

**BERT Large, Single V100**
MAKE SURE TENSOR CORES ARE BEING USED

NVIDIA Deep Learning Profiler TensorBoard Plugin

Nodes using TC are ops that use Tensor Cores

Nodes Eligible For TC are ops that did not use Tensor Cores but could have (e.g. conv/linear)

For individual layers can check whether input shapes satisfy TC constraints
DOUBLE CHECK ON TENSOR CORE EFFICIENCY

If a few layers dominate training time, then make toy example for those layers

```python
n, k = (1024, 1024)  # layer dimensions
x = torch.randn(k, n).cuda().half()
linear = torch.nn.Linear(k, n).cuda().half()
y = linear(x) + x
```

**NVIDIA Nsight Compute** (next gen profiler for CUDA applications)

```
nv-nvsight-cu-cli --metrics sm__pipe_tensor_cycles_active.avg.pct_of_peak_sustained_active python train.py
```

<table>
<thead>
<tr>
<th>Kernel Name</th>
<th>Metric Name</th>
<th>Metric Unit</th>
<th>Metric Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>volta_fp16_s884cudnn</td>
<td>sm__pipe_tensor_cycles_active.avg</td>
<td>%</td>
<td>86.35</td>
</tr>
<tr>
<td>elementwise_kernel</td>
<td>sm__pipe_tensor_cycles_active.avg</td>
<td>%</td>
<td>0</td>
</tr>
</tbody>
</table>
1. Satisfy shape constraints to enable tensor cores
   - For linear layers: input size, output size, batch size should be multiples of 8
   - For convolutions: input and output channel counts should be multiples of 8
   - Not requirement for cuBLAS >= 11.0 and cuDNN >= 8.0, but can help better perf

2. Ensure Tensor Cores are doing enough math
   - If any GEMM dimension is 128 or smaller, operation is memory bound rather than math bound
   - Speedup will be in 1-2x range rather than 8-16x

---

**Linear layer with M=N=8192**

_Benchmarked on V100 with cuBLAS 10.1_

(numbers indicate Tensor Core speedups)

<table>
<thead>
<tr>
<th>K</th>
<th>FP32</th>
<th>AMP</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>8190</td>
<td></td>
<td>6.5x</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td>2.5x</td>
<td></td>
</tr>
<tr>
<td>8192</td>
<td></td>
<td></td>
<td>6.5x</td>
</tr>
</tbody>
</table>

K%8 != 0 → No tensor core
K%8 == 0 → Using tensor core
GENERAL PERFORMANCE GUIDELINES

Follow a few simple guidelines to maximize performance from mixed-precision

1. Ensure most of training time is spent doing GPU work
   • Ensure GPU is being utilized (e.g. larger model/batch size)
   • Eliminate CPU inefficiencies such as data preprocessing

2. Ensure math-bound layers (gemms and convs) dominate training time
   • Leverage fusions to reduce time spent on memory-bound layers
   • Adapt network architecture to be more hardware-friendly

3. Improve Tensor Core utilization with good parameter choices
   • Favor multiples of 8 for linear/conv layer dimensions
   • Ensure linear/conv layers are large enough to fully utilize TCs
CONCLUSIONS
A100 introduces the next generation of Tensor Cores for DL acceleration

- TF32 is the default math mode on A100
- Accelerates single-precision training
- 10x more math throughput than Volta single-precision
- Network speedups up to 6x

FP16 and BF16 formats for maximum speed

- FP16 and BF16 Tensor Cores provide 16x more math throughput than FP32 (2x faster than TF32)
- AMP makes FP16 training easy in all major frameworks
- Training results match those of single-precision, require no changes to hyper-parameters
- Also reduce memory consumption, enabling larger batches, larger models, etc

Sparsity support for a further 2x math throughput

- Accelerates DL inference