GETTING MORE DL TRAINING WITH TENSOR CORES AND AMP

한재근 과장 | Solutions Architect | jahan@nvidia.com

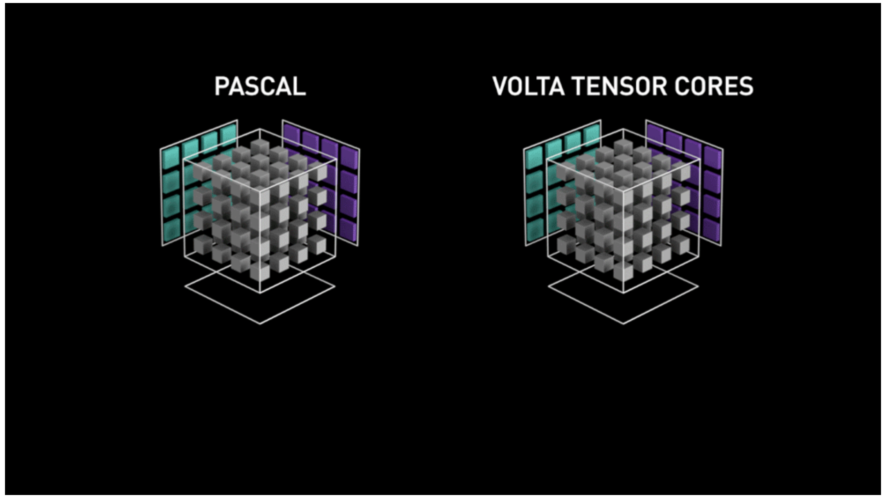
AGENDA

- What is Automatic Mixed Precision
- AMP Technical Details
- Getting started on AMP
- Performance Guide
- Additional Resources

BACKGROUD: TENSOR CORES

Hardware support for accelerated 16-bit FP math

- **125 TFlops in FP16** vs 15.7 TFlops in FP32 (8x speed-up)
- Inherently mixed precision: 32bit accumulation
- Available in **Volta** and **Turing** architecture GPUs
- Optimized 4x4 dot operation (GEMM)



$$\underbrace{D}_{\text{FP16 or FP32}} = \left(\underbrace{\begin{matrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{matrix}}_{\text{FP16}} \right) \left(\underbrace{\begin{matrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{matrix}}_{\text{FP16}} \right) + \left(\underbrace{\begin{matrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{matrix}}_{\text{FP16 or FP32}} \right)$$

Memory Savings

- Half Storage Requirements (larger batch size)
- Half the memory traffic by reducing size of gradient/activation tensors

WHAT IS AUTOMATIC MIXED PRECISION

MAXIMIZING MODEL PERFORMANCE

FP16 is fast and memory-efficient.

FP32	FP16 with Tensor Cores
1x compute throughput	8X compute throughput
1x memory throughput	2X memory throughput
1x memory storage	1/2X memory storage

MIXED PRECISION TRAINING

Motivation

- **Balance a pure tradeoff of speed and accuracy:**
    - Reduced precision (16-bit floating point) for *speed* or *scale*
    - Full precision (32-bit floating point) to *maintain task-specific accuracy*
- **Under the constraints:**
    - Maximize use of reduced precision
      while matching accuracy of full precision training
    - No changes to hyperparameters

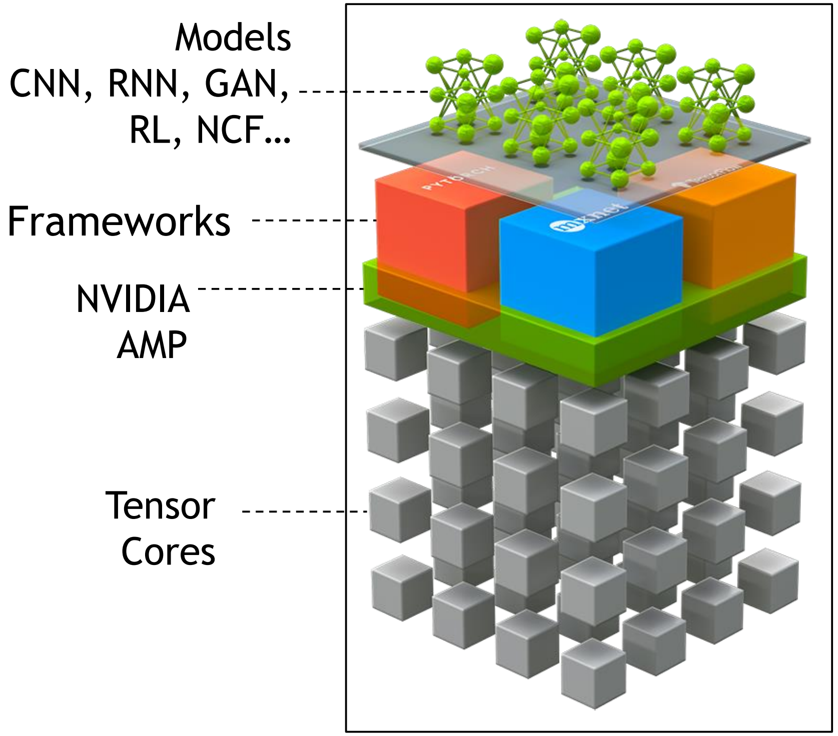
https://devblogs.nvidia.com/mixed-precision-training-deep-neural-networks/

AUTOMATIC MIXED PRECISION

Speedup Your Network Across Frameworks With Just Two Lines of Code

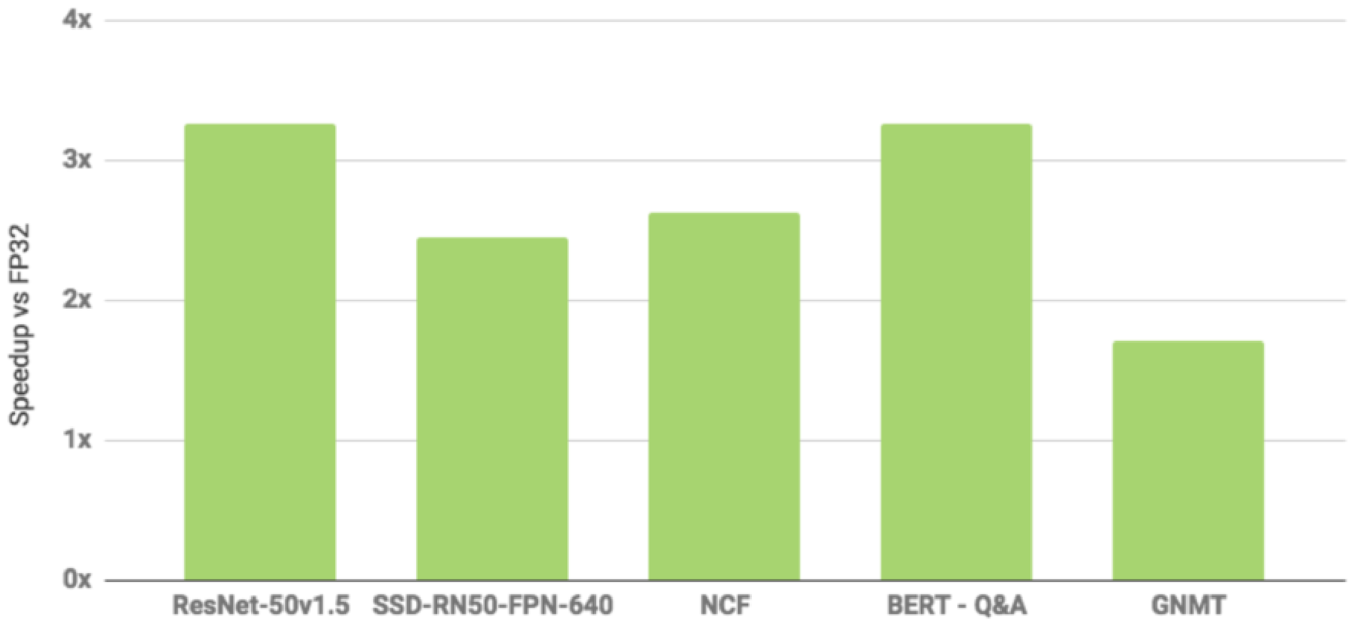
"This easy integration enables TensorFlow developers to literally flip a switch in their AI model and get up to 3X speedup with mixed precision training while maintaining model accuracy."

Rajat Monga, Engineering Director, TensorFlow



AUTOMATIC MIXED PRECISION IN TENSORFLOW

Upto 3X Speedup



TensorFlow Medium Post: [Automatic Mixed Precision in TensorFlow for Faster AI Training on NVIDIA GPUs]

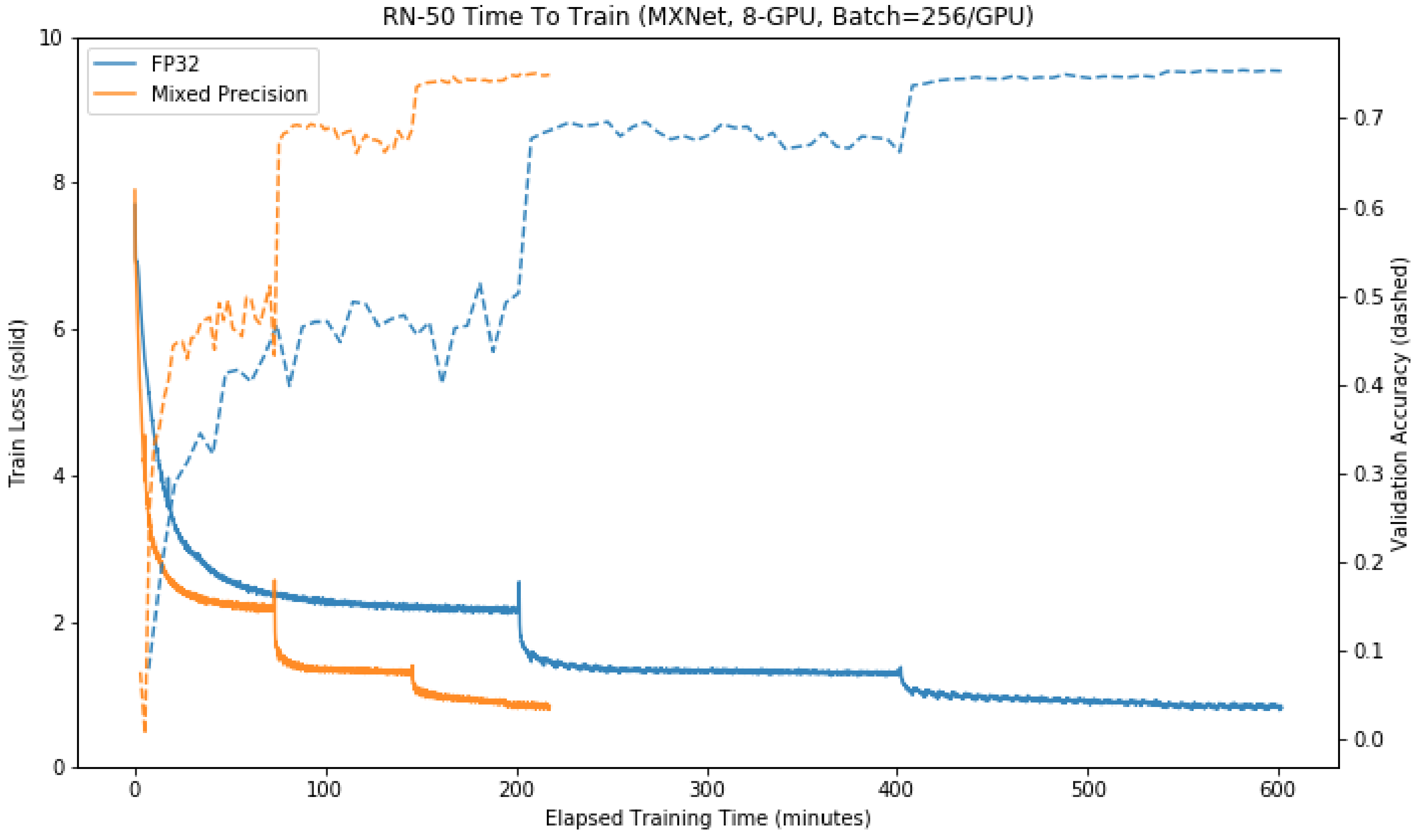
All models can be found at:
https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow, except for ssd-rn50-fpn-640, which is here: https://github.com/tensorflow/models/tree/master/research/object_detection
All performance collected on 1xV100-16GB, except bert-squadqa on 1xV100-32GB.
Speedup is the ratio of time to train for a fixed number of epochs in single-precision and Automatic Mixed Precision. Number of epochs for each model was matching the literature or common practice (it was also confirmed that both training sessions achieved the same model accuracy).
Batch sizes:. rn50 (v1.5): 128 for FP32, 256 for AMP+XLA; ssd-rn50-fpn-640: 8 for FP32, 16 for AMP+XLA; NCF: 1M for FP32 and AMP+XLA; bert-squadqa: 4 for FP32, 10 for AMP+XLA; GNMT: 128 for FP32, 192 for AMP.

NVIDIA

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
ResNet-50 v1.5	Image Recognition	3.5x	Iso-batch size
FairSeq Transformer	Translation	2.9x 4.9x	Iso-batch size 2x lr + larger batch
BERT	SQuAD (fine-tuning)	1.9x	Iso-batch size
Deep Speech 2	Speech recognition	4.5x	Larger batch
Tacotron 2 + WaveGlow	Speech synthesis	1.2x 1.9x	Iso-batch size
Nvidia Sentiment	Language modeling	4.0x	Larger batch
NCF	Recommender	1.8x	Iso-batch size

trained to **SAME ACCURACY** as FP32 model

No hyperparameter changes, except as noted

TECHNICAL DETAILS OF MIXED PRECISION

MIXED PRECISION TRAINING

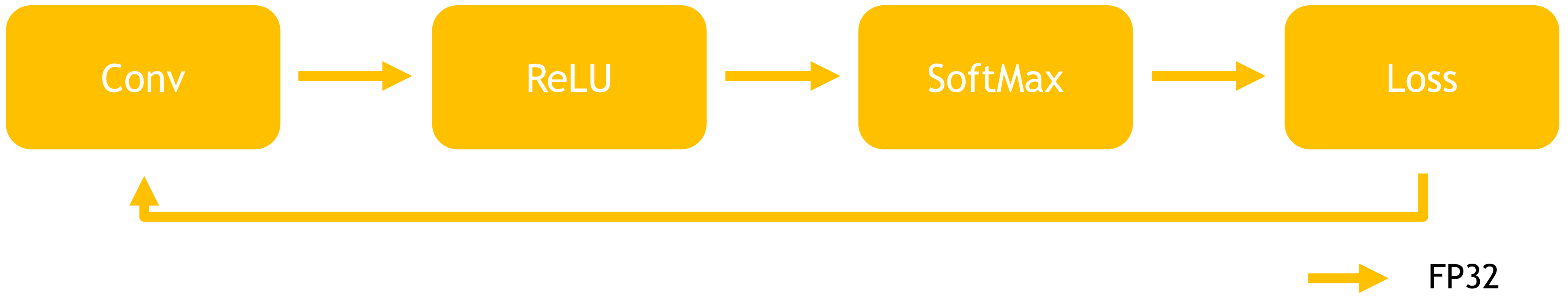
Three-part methodology

- **Model conversion:**
  - Switch everything to run on FP16 values
  - Insert casts to FP32 for loss functions and normalization/pointwise ops that need full precision
- **Master weights:**
  - Keep FP32 model parameters, update at each iteration
  - Use an FP16-casted copy for both forward pass and backpropagation
- **Loss scaling:**
  - Scale the loss value, un-scale the gradients (in FP32!)
  - Check gradients at each iteration for overflow – adjust loss scale and skip update, if needed

MIXED PRECISION TRAINING

Assign each operation its optimal precision & performance

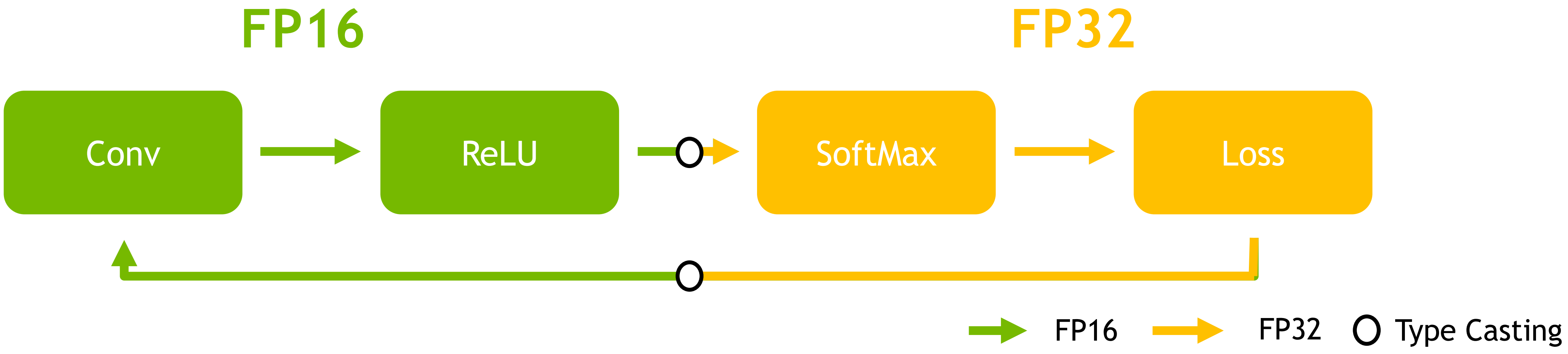
Before Mixed Precision



- **GEMMs**, **Conv**olutions can use Tensor Cores
- Most **pointwise ops** (e.g. add, multiply): 1/2X memory storage for intermediates, 2X memory throughput
- **Weight** updates benefit from precision
- **Loss** functions (often reductions) benefit from precision and range
- **Softmax**, norms, some other ops benefit from precision and range

MIXED PRECISION TRAINING

Assign each operation its optimal precision & performance



- GEMMs, Convolutions can use Tensor Cores
- Most pointwise ops (e.g. add, multiply): 1/2X memory storage for intermediates, 2X memory throughput
- Weight updates benefit from precision
- Loss functions (often reductions) benefit from precision and range
- Softmax, norms, some other ops benefit from precision and range

AUTOMATIC MIXED PRECISION

Concepts

- Allows to implement the three part methodology **automatically**:
    - The framework software can transform existing model code to run with mixed precision fully automatically
    - No new code required should result in no new bugs
- Two components:
    - **Automated casting**: operation-level logic to insert casts between FP32 and FP16, transparent to the user
    - **Automatic loss scaling**: wrapper class for the optimizer object the can scale the loss, keep track of the loss scale, and skip updates as necessary

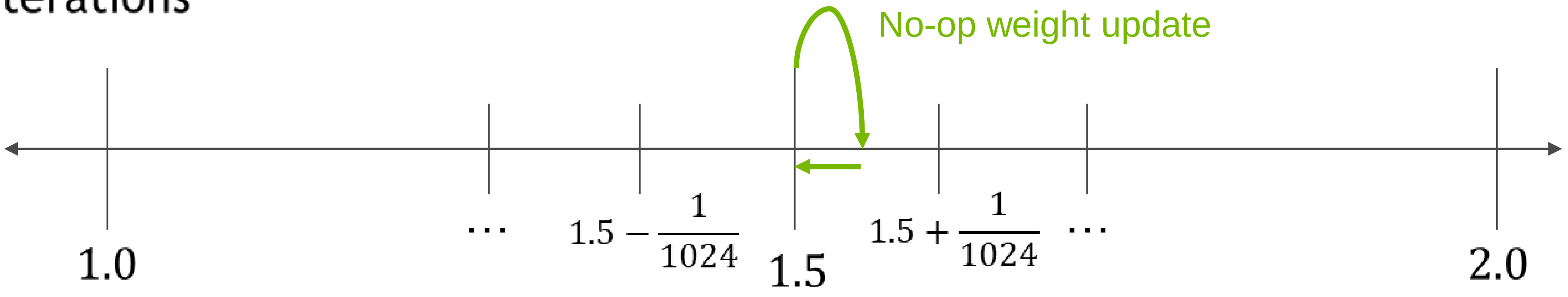
AUTOMATIC CASTING

Operation Classification

- Divide the universe of operations into three kinds
  - Whitelist: ops for which FP16 can use Tensor Cores (MatMul, Conv2d)
  - Blacklist: ops for which FP32 is required for accuracy (Exp, Sum, Softmax)
  - Everything else: ops that can run in FP16, but only worthwhile if input is already FP16 (ReLU, pointwise Add, MaxPool)
- Lists are framework-specific, since each framework has its own abstractions
  - TensorFlow: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/grappler/optimizers/auto_mixed_precision_lists.h
  - PyTorch: https://github.com/NVIDIA/apex/tree/master/apex/amp/lists
  - MXNET: https://github.com/apache/incubator-mxnet/blob/master/python/mxnet/contrib/amp/lists/symbol.py

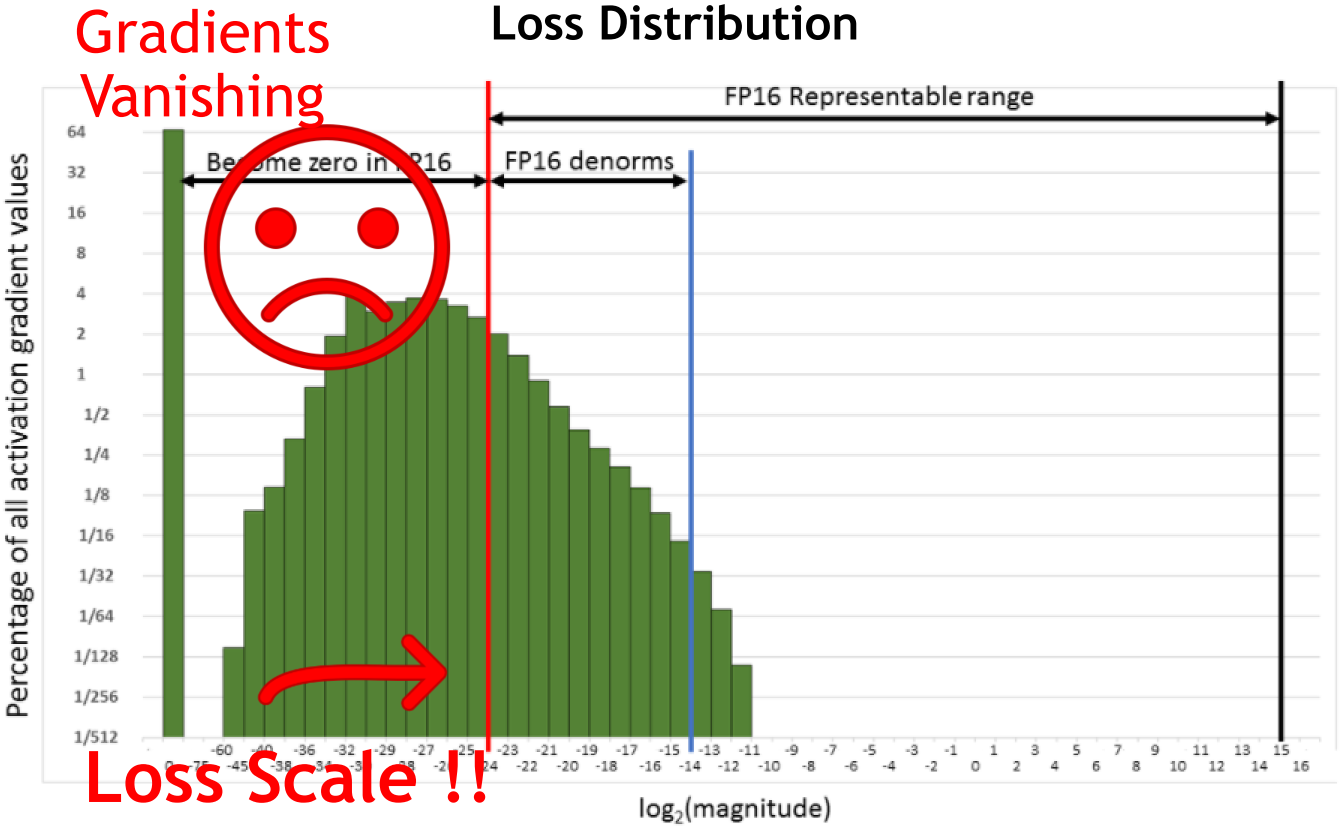
KEEP FP32 MASTER WEIGHTS

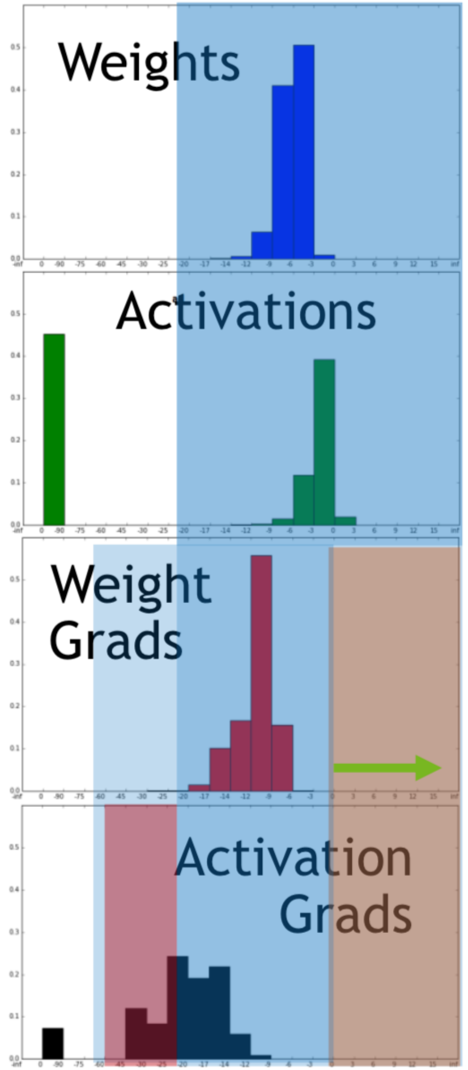
- At each iteration of training, perform a *weight update* of the form $w_{t+1} = w_t - \alpha\nabla_t$
  - w_t's are weights; ∇_t's are gradients; α is the learning rate
- As a rule, gradients are smaller than weights, and learning rate is less than one
- Consequence: weight update *can be* a no-op,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



LOSS SCALING

Precision of Weights & Gradients



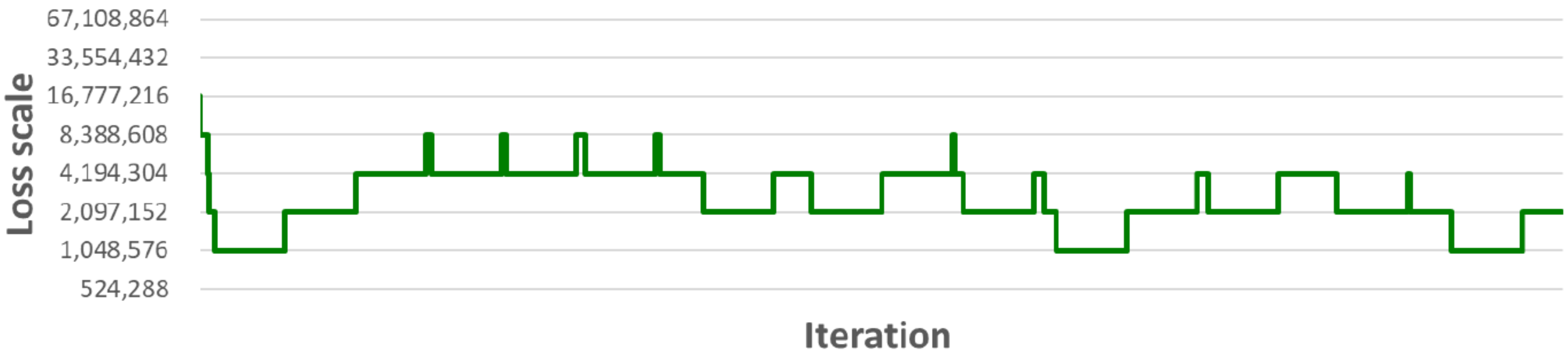


AUTOMATIC LOSS SCALING

- All frameworks implement some kind of optimizer wrapper
    - Internally, tracks the current loss scale and history of overflows
    - Provides the loss scale when needed for backpropagation
    - Overrides optimizer step to instead checking overflow
        - No overflow, then increase grads and pass to wrapped optimizer step
        - overflow, then decrease loss scale and don’t called wrapped optimizer step

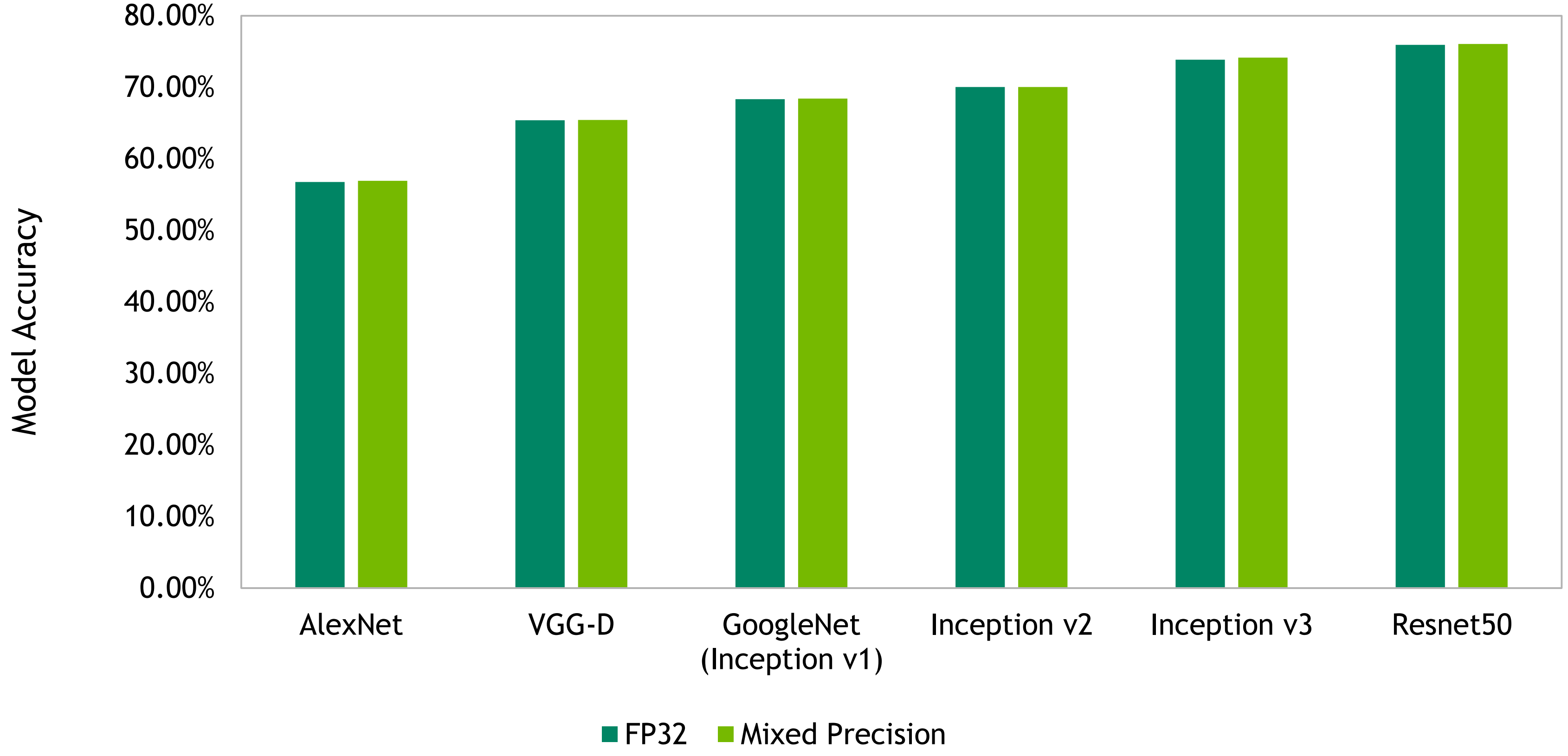
AUTOMATIC LOSS SCALE

- Internally, tracks the current loss scale and history of overflows
- Provides the loss scale when needed for backpropagation
  - 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



MIXED PRECISION MAINTAINS ACCURACY

Benefit From Higher Throughput Without Compromise



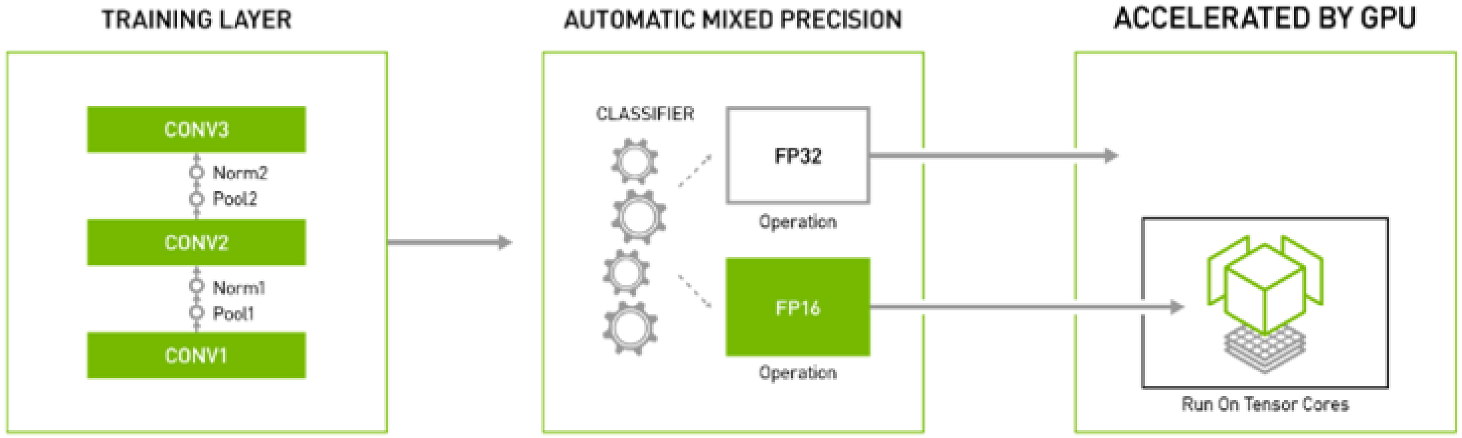
Mixed Precision - Same hyperparameters and learning rate schedule as FP32
ILSVRC12 classification top-1 accuracy.
(Sharan Narang, Paulius Micikevicius *et al.*, "Mixed Precision Training“, ICLR 2018)

GETTING STARTED AUTO MIXED PRECISION

AUTOMATIC MIXED PRECISION

NEW

Easy to Use, Greater Performance and Boost in Productivity



Insert ~ two lines of code

to introduce Automatic Mixed-Precision and get upto 3X speedup

AMP uses a graph optimization technique to determine FP16 and FP32 operations

Support for TensorFlow, PyTorch and MXNet

Unleash the next generation AI performance and get faster to the market!

More details: https://developer.nvidia.com/automatic-mixed-precision

TENSORFLOW AMP

A simple method

- Designed to work with existing float32 models, with minimal changes
- Support since NGC TensorFlow Container 19.03
- If your training script uses a `tf.train.Optimizer` to compute and apply gradients
  Both Loss Scaling and mixed precision graph conversion can be enabled with a single env var.

```
export TF_ENABLE_AUTO_MIXED_PRECISION=1

python training_script.py
```

- If your model does not use a `tf.train.Optimizer`, then
  You must add loss scaling manually to your model, then enable the grappler pass as follows

```
export TF_ENABLE_AUTO_MIXED_PRECISION_GRAPH_REWRITE=1

python training_script.py
```

TENSORFLOW AMP

A more explicit

- Since NGC TensorFlow Container 19.07 (TF 1.14+)
- Supports an explicit optimizer wrapper to perform loss scaling
- Enables auto casting / loss scaling and mixed precision graph optimizer

```
import tensorflow as tf

opt = tf.train.GradientDescentOptimizer(0.5)

opt = tf.train.experimental.enable_mixed_precision_graph_rewrite(opt)
```

PYTORCH AMP

```
N, D_in, D_out = 64, 1024, 512
x = torch.randn(N, D_in, device="cuda")
y = torch.randn(N, D_out, device="cuda")

model = torch.nn.Linear(D_in, D_out).cuda()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)


for to in range(500):
    y_pred = model(x)
    loss = torch.nn.functinoal.mse_loss(y_pred, y)

    optimizer.zero_grad()

    loss.backward()
    optimizer.step()
```

PYTORCH AMP

```
from apex import amp
N, D_in, D_out = 64, 1024, 512
x = torch.randn(N, D_in, device="cuda")
y = torch.randn(N, D_out, device="cuda")

model = torch.nn.Linear(D_in, D_out).cuda()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
model, optimizer = amp.initialize(model, optimizer, opt_level="O1")

for to in range(500):
    y_pred = model(x)
    loss = torch.nn.functinoal.mse_loss(y_pred, y)

    optimizer.zero_grad()
    with amp.scale_loss(loss, optimizer) as scaled_loss:
        scaled_loss.backward()
    optimizer.step()
```

* Use apex.parallel.DistributedDataParallel for multi-GPU training

PYTORCH OPTIMIZATION LEVEL

OPT_LEVEL="O0"

FP32 training.
Your incoming model should be FP32 already, so this is likely a no-op. O0 can be useful to establish an accuracy baseline.

O1

Mixed Precision
Patches Torch functions to internally carry out Tensor Core-friendly ops in FP16, and ops that benefit from additional precision in FP32. Also uses dynamic loss scaling. **Because cats occur in functions, model weights remain FP32**

O2

"Almost FP16" Mixed Precision.
FP16 model and data with FP32 batchnorm, FP32 master weights, and dynamic loss scaling. **Model weights, except batchnorm weights, are cast to FP16.**

O3

FP16 training.
O3 can be useful to establish the "speed of light" for your model. If your model uses batch normalization, and `keep_batchnorm_fp32=True`, which enables cudnn bachnorm.

PYTORCH OPTIMIZATION LEVEL

OPT_LEVEL="O0"

FP32 training.
Your incoming model should be FP32 already, so this is likely a no-op. O0 can be useful to establish an accuracy baseline.

O1

Mixed Precision
Patches Torch functions to internally carry out Tensor Core-friendly ops in FP16, and ops that benefit from additional precision in FP32. Also uses dynamic loss scaling. **Because cats occur in functions, model weights remain FP32**

O2

"Almost FP16" Mixed Precision.
FP16 model and data with FP32 batchnorm, FP32 master weights, and dynamic loss scaling. **Model weights, except batchnorm weights, are cast to FP16.**

O3

FP16 training.
O3 can be useful to establish the "speed of light" for your model. If your model uses batch normalization, and **`keep_batchnorm_fp32=True`**, which enables cudnn bachnorm.

MXNET AMP

```
net = get_network()
trainer = mx.gluon.Trainer(...)

for data in dataloader:
  with autograd.record(True):
    out = net(data)
    l = loss(out, label)

    autograd.backward(scaled_loss)
    trainer.step()
```

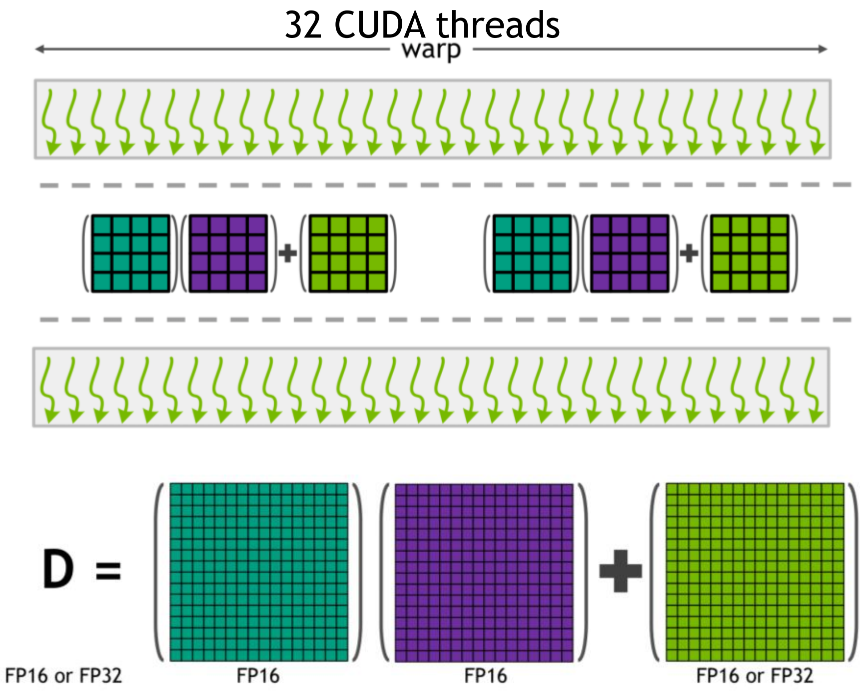
MXNET AMP

```
From mxnet.contrib import amp
amp.init()
net = get_network()
trainer = mx.gluon.Trainer(...)
amp.init_trainer(trainer)
for data in dataloader:
  with autograd.record(True):
    out = net(data)
    l = loss(out, label)
    with amp.loss_scale(loss, trainer) as scaled_loss:
      autograd.backward(scaled_loss)
    trainer.step()
```

TENSOR CORES PERFORMANCE GUIDE

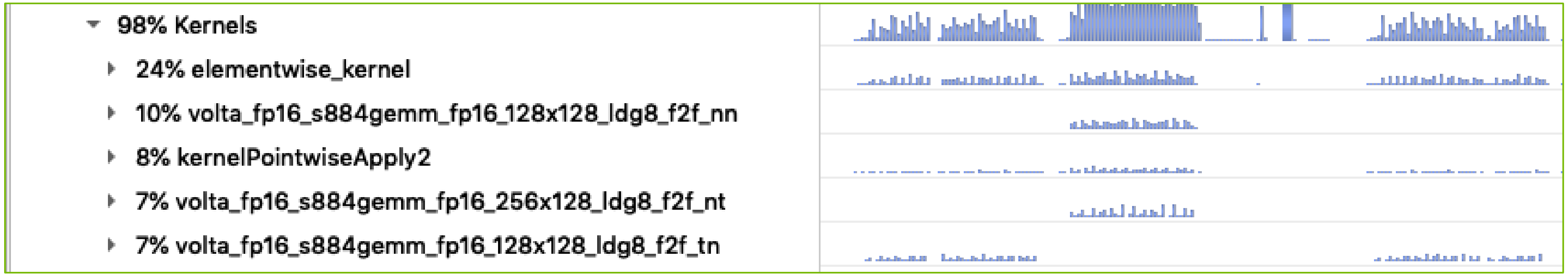
GETTING MORE FROM TENSOR CORES

- Matrix Multiplication
  - All the dimensions (M, N, K) should be multiples of 8
- Recommended to be a multiple of 8
  - Input size, output size, batch size
  - Linear layer dimensions
  - Convolution layer channel counts (NCHW format)
  - Pad the sequence length For sequence problems
- Ensure good Tensor Cores GEMM efficiency
  - Choose the above dimensions as multiples of 64/128/256
- Finally, Double the batch size



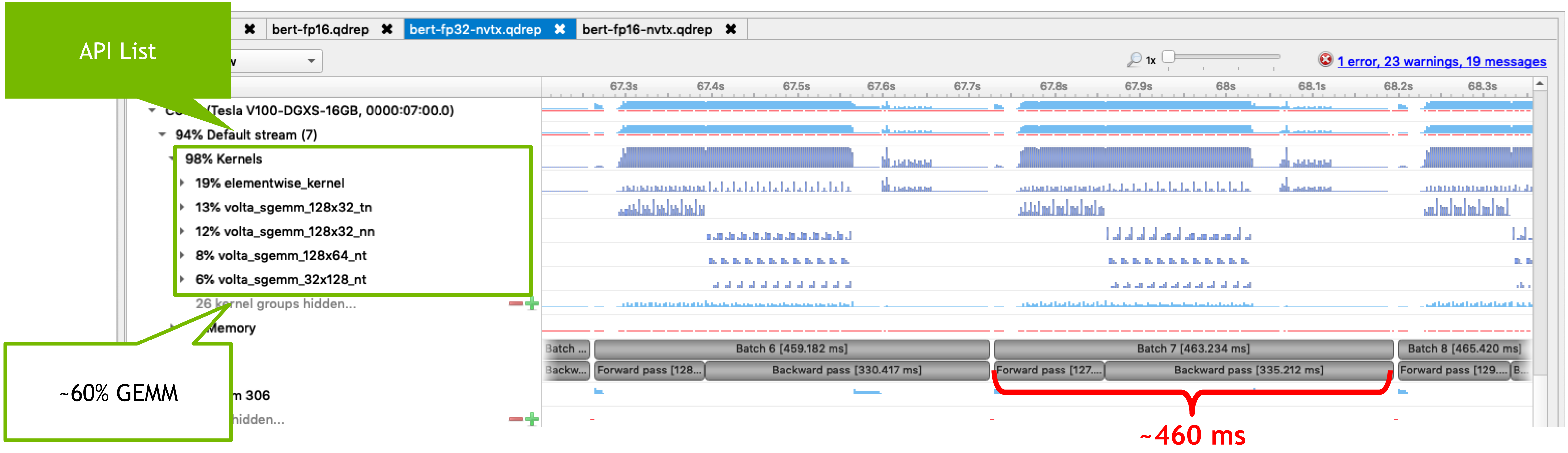
AM I USING TENSOR CORES?”

- cuBLAS and cuDNN are optimized for Tensor Cores, coverage always increasing
- Run with nvprof and look for “s[some digits]” in kernel name
  - Eg: volta_fp16_s884gemm_fp16_128x128_ldg8_f2f_nn



BERT FP32 BENCHMARK

HuggingFace's pretrained BERT



NVIDIA

BERT FP16 BENCHMARK

HuggingFace's pretrained BERT

2.1x Speed up

NVIDIA NGC MODEL SCRIPTS

NEW

Tensor Core Optimized Deep Learning Examples

16 Available today!

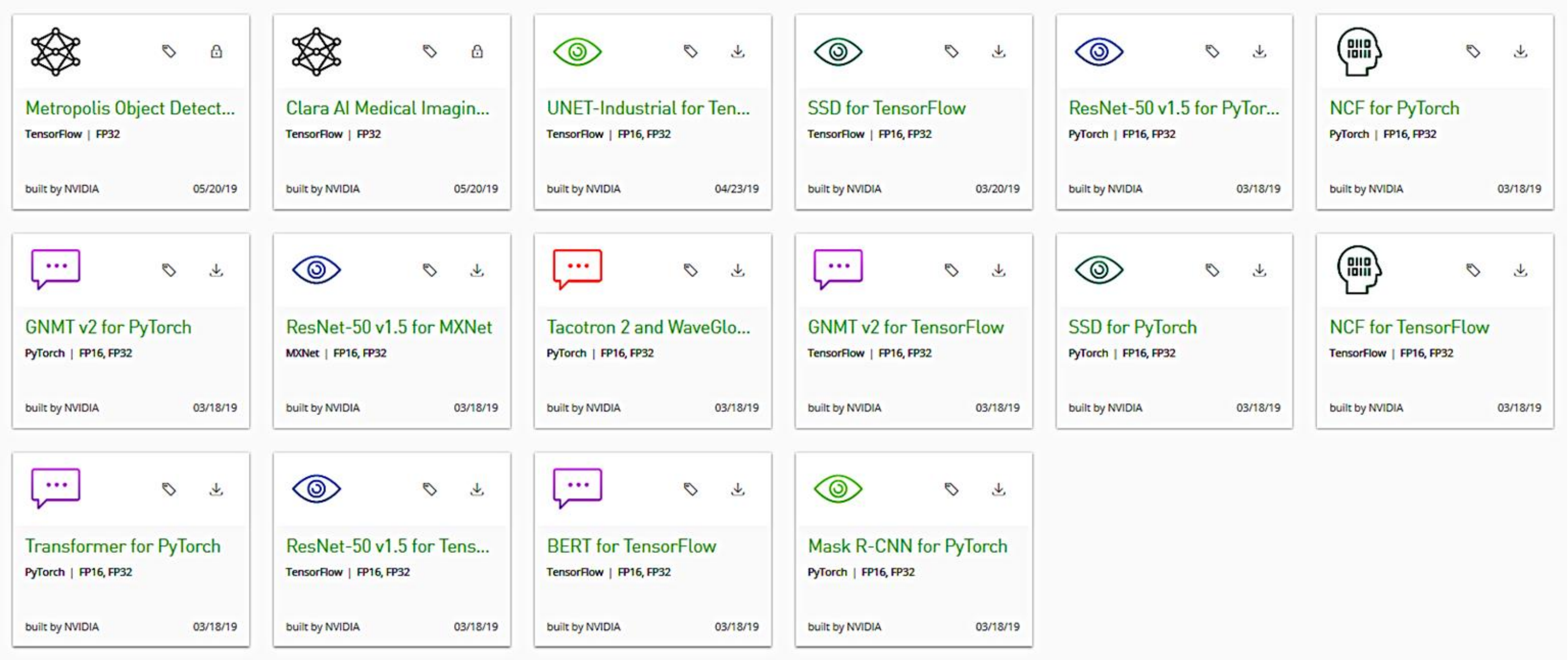
- Tensor Core optimized for greater performance
- Test drive automatic mixed precision
- Actively updated by NVIDIA
- State-of-the-art accuracy using Tensor Cores
- Serves as a reference implementation
- Exposes hyperparameters and source code for further adjustment

Accessible via:

- NVIDIA NGC https://ngc.nvidia.com/catalog/model-scripts
- GitHub https://www.github.com/NVIDIA/deeplearningexamples
- NVIDIA NGC Framework containers https://ngc.nvidia.com/catalog/containers

NVIDIA NGC MODEL SCRIPTS

Tensor Core Examples Built for Multiple Use Cases and Frameworks



A dedicated hub to download Tensor Core Optimized Deep Learning Examples on NGC

https://ngc.nvidia.com/catalog/model-scripts?quickFilter=deep-learning

MODEL SCRIPTS FOR VARIOUS APPLICATIONS

https://developer.nvidia.com/deep-learning-examples

Computer Vision

- SSD PyTorch
- SSD TensorFlow
- UNET-Industrial TensorFlow
- UNET-Medical TensorFlow
- ResNet-50 v1.5 MXNet
- ResNet-50 PyTorch
- ResNet-50 TensorFlow
- Mask R-CNN PyTorch

Speech & NLP

- GNMT v2 TensorFlow
- GNMT v2 PyTorch
- Transformer PyTorch
- **BERT** (Pre-training and Q&A) TensorFlow

Recommender Systems

- NCF PyTorch
- NCF TensorFlow

Text to Speech

- Tacotron2 and WaveGlow PyTorch

ENABLING AUTOMATIC MIXED PRECISION

Add Just A Few Lines of Code, Get Upto 3X Speedup

TensorFlow — GA

```
os.environ['TF_ENABLE_AUTO_MIXED_PRECISION'] = '1'
OR
export TF_ENABLE_AUTO_MIXED_PRECISION=1

NVIDIA Container 19.07+ and TF 1.14+, explicit optimizer wrapper available:

opt = tf.train.experimental.enable_mixed_precision_graph_rewrite(opt)
```

PyTorch — GA

```
model, optimizer = amp.initialize(model, optimizer,
opt_level="O1")
with amp.scale_loss(loss, optimizer) as scaled_loss:
    scaled_loss.backward()
```

MXNet — GA Coming Soon

```
amp.init()
amp.init_trainer(trainer)
with amp.scale_loss(loss, trainer) as scaled_loss:
   autograd.backward(scaled_loss)
```

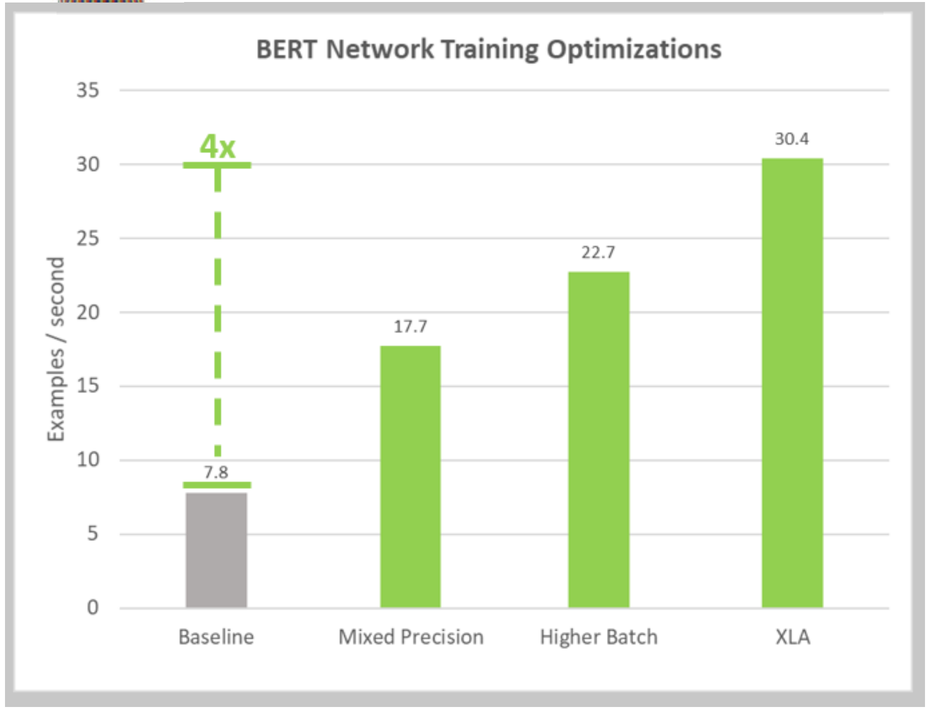
More details: https://developer.nvidia.com/automatic-mixed-precision

GTC SESSION RECORDINGS 2019

Recommended on-demand-gtc.gputechconf.com Talks

- Overview
  - (E8494) Mixed precision training with Deep Neural Networks
  - (S91022) Text-to-speech: Overview of latest research using Tacotron and Waveglow
- PyTorch
  - (S9998) Automatic Mixed Precision in PyTorch
  - (S9832) Taking advantage of mixed precision to accelerate training in PyTorch
- TensorFlow
  - (S91029) Automatic mixed precision tools for TensorFlow Training
- MXNet
  - (S91003) MXNet Computer Vision and Natural Language Processing Models Accelerated with NVIDIA Tensor Cores

TAKEAWAY



- Getting 3x math performance with Tensor Cores
- Reduced memory usage → larger batch size
- Achieves the same accuracy of FP32 training
- Just need a 2-3 lines of codes
- 16 samples in NGC, nvidia/DeepLearningExamples in CV, NLP, Speech, and Recommendation
- Increasing providing samples, we do first for you

https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/LanguageModeling/BERT

GETTING MORE IN TRAINING

Deep learning training acceleration

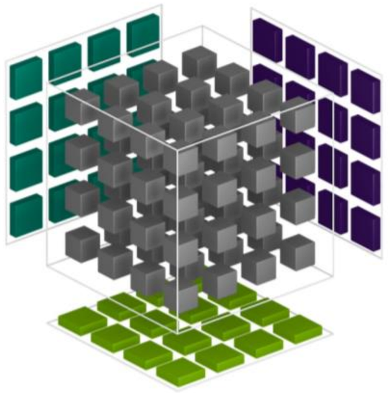
- Deep Learning Research of NAVER Clova for AI-Enhanced Business
    - Mixed Precision’s contribution to LarVa (Language Representations by **Clova**) research
    - Track 1, Session 2 (13:50 - 14:30), 하정우 리더 (CLOVA AI Research 리더)
- GPU를 활용한 Image Augmentation 가속화 방안 - DALI
    - Track 1, Session 4 (15:40 - 16:20), 한재근 과장 (NVIDIA Solutions Architect)
- GPU Profiling 기법을 통한 Deep Learning 성능 최적화 기법 소개
    - Track 3, Session 5 (16:30 - 17:10), 홍광수 과장 (NVIDIA Solutions Architect)

NVIDIA®

SUPPLEMENTS:
TENSOR CORES API

TENSOR CORE

Mixed Precision Matrix Math
4x4 matrices



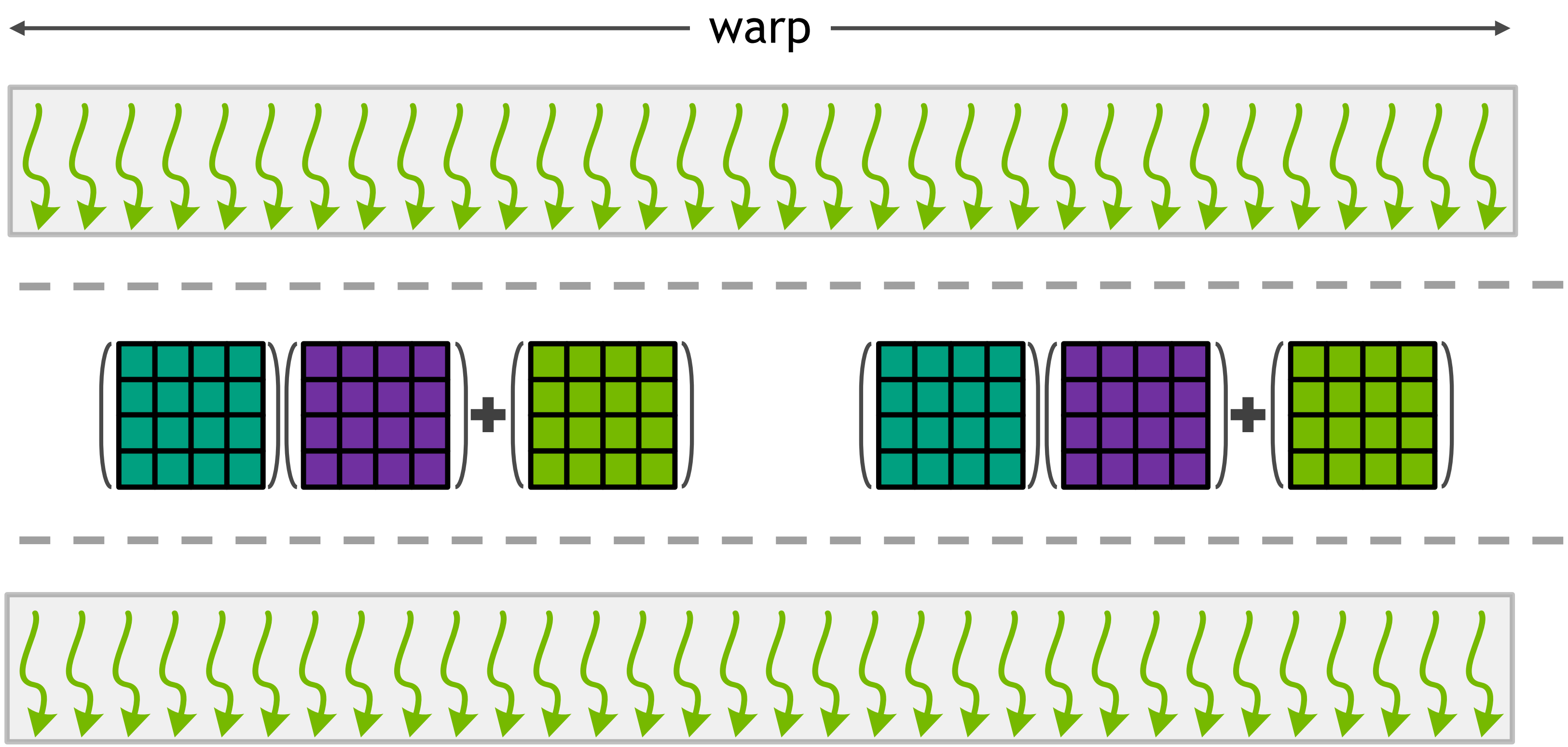
$$
D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}
$$

D	A	B	C
FP16 or FP32	FP16	FP16	FP16 or FP32

$$D = AB + C$$

TENSOR SYNCHRONIZATION

Full Warp 16x16 Matrix Math



Warp-synchronizing operation

Composed Matrix Multiply and Accumulate for **16x16** matrices

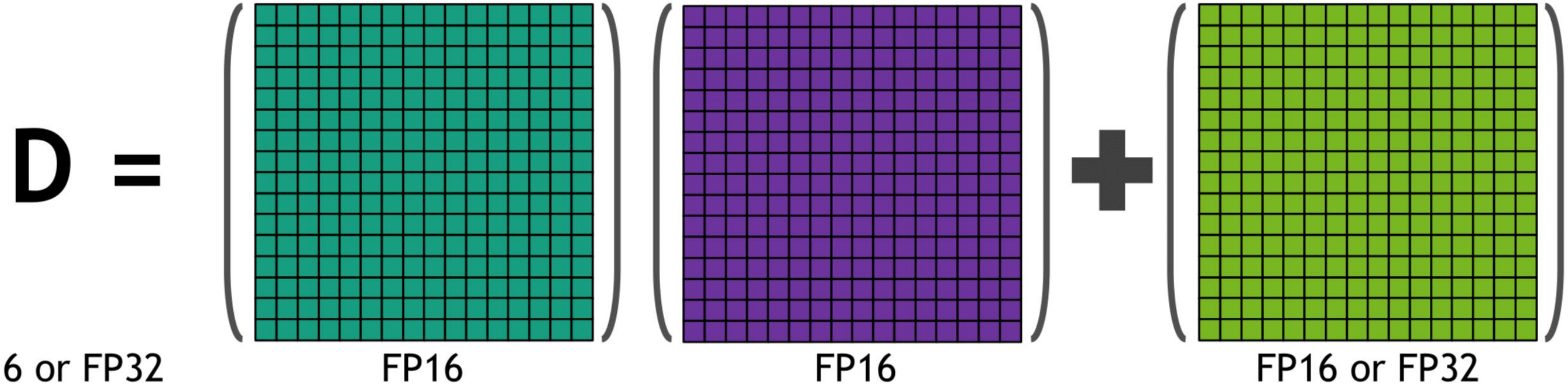
Result distributed across warp

NVIDIA

CUDA TENSOR CORE PROGRAMMING

16x16x16 Warp Matrix Multiply and Accumulate (WMMA)

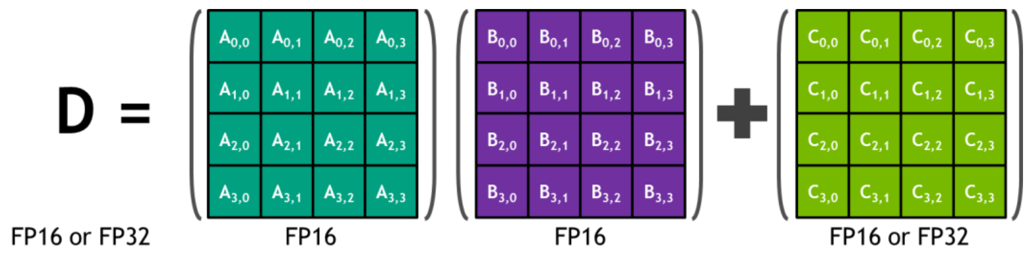
```
wmma::mma_sync9Dmat, Amat, Bmat, Cmat);
```



$$D = AB + C$$

WARP MATRIX API

Overview

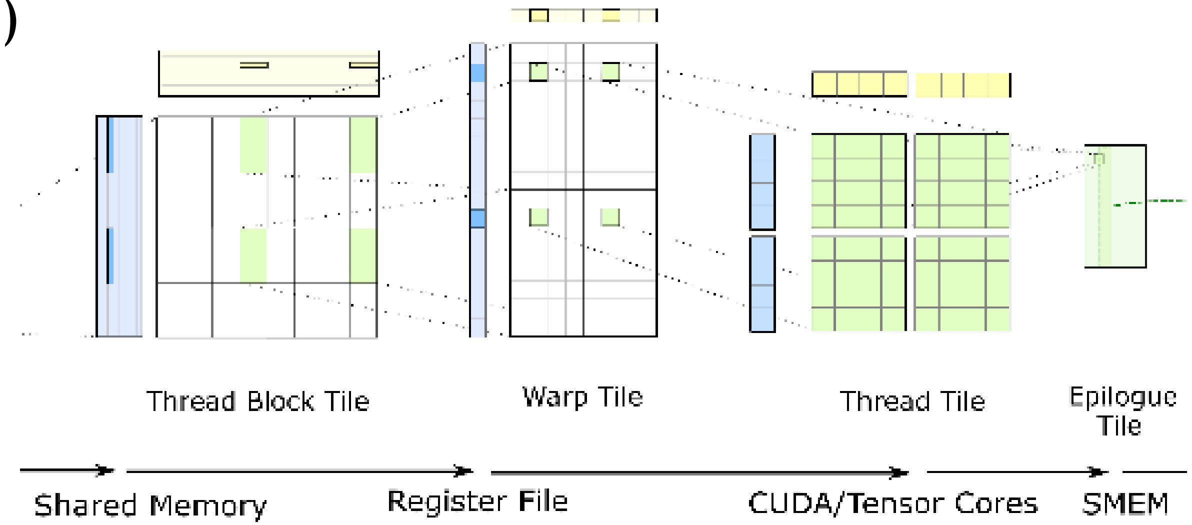


$$D = AB + C$$

Introduced in **Volta** as an abstraction layer

Provides CUDA C++ API that:

- Defines **fragment abstraction** (array of values)
- **Load Matrix A/B** from SMEM to Registers
- Perform the ***MMA operation**
- **Store Accumulators** from Registers to SMEM
- More information in Programming Guide

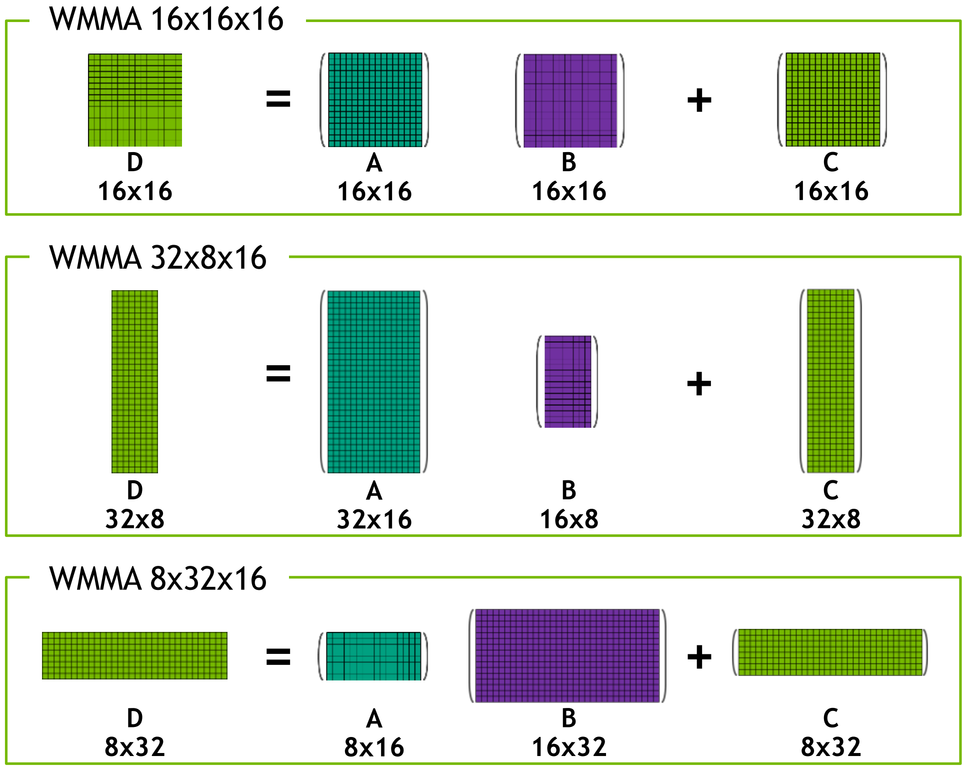


TURING TENSOR CORE

New Warp Matrix Functions

API operations now include 8-bit integer

- Turing (sm_75) only
- Signed & unsigned 8-bit input
- 32-bit integer accumulator
- Match input/output dimensions with *half*
- Experimental Sub-Byte Operations (4-bit, 1-bit)

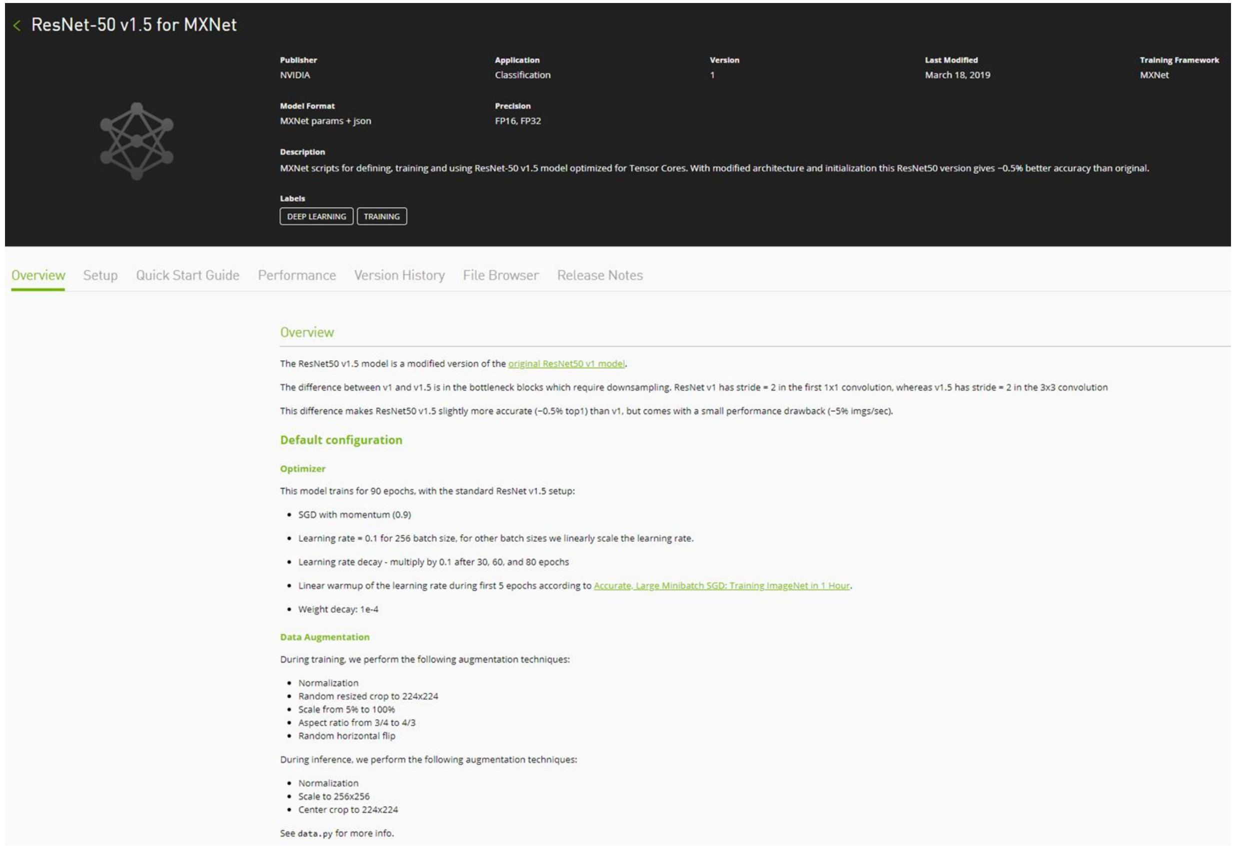


SUPPLEMENTS:
MIXED PRECISION PERFORMANCE

IMAGE CLASSIFICATION: MXNet ResNet-50 v1.5

https://ngc.nvidia.com/catalog/model-scripts/nvidia:resnet_50_v1_5_for_mxnet

DGX-1V **8GPU 16G**	MXNet ResNet FP32	MXNet ResNet Mixed Precision
Time to Train [Hours]	**11.1**	**3.3**
Train AccuracyTop 1%	**76.67%**	**76.49%**
Perf.	**2,957** Img/sec	**10,263** Img/sec
Data set	ImageNet	



ResNet-50 v1.5 for MXNet

Publisher: NVIDIA
Application: Classification
Version: 1
Last Modified: March 18, 2019
Training Framework: MXNet
Model Format: MXNet params + json
Precision: FP16, FP32

Description
MXNet scripts for defining, training and using ResNet-50 v1.5 model optimized for Tensor Cores. With modified architecture and initialization this ResNet50 version gives ~0.5% better accuracy than original.

Labels
DEEP LEARNING | TRAINING

Overview | Setup | Quick Start Guide | Performance | Version History | File Browser | Release Notes

Overview

The ResNet50 v1.5 model is a modified version of the original ResNet50 v1 model.

The difference between v1 and v1.5 is in the bottleneck blocks which require downsampling. ResNet v1 has stride = 2 in the first 1x1 convolution, whereas v1.5 has stride = 2 in the 3x3 convolution

This difference makes ResNet50 v1.5 slightly more accurate (~0.5% top1) than v1, but comes with a small performance drawback (~5% imgs/sec).

Default configuration

Optimizer

This model trains for 90 epochs, with the standard ResNet v1.5 setup:

- SGD with momentum (0.9)
- Learning rate = 0.1 for 256 batch size, for other batch sizes we linearly scale the learning rate.
- Learning rate decay - multiply by 0.1 after 30, 60, and 80 epochs
- Linear warmup of the learning rate during first 5 epochs according to Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour.
- Weight decay: 1e-4

Data Augmentation

During training, we perform the following augmentation techniques:

- Normalization
- Random resized crop to 224x224
- Scale from 5% to 100%
- Aspect ratio from 3/4 to 4/3
- Random horizontal flip

During inference, we perform the following augmentation techniques:

- Normalization
- Scale to 256x256
- Center crop to 224x224

See data.py for more info.

NGC 18.12+ MXNet container

Source: https://github.com/NVIDIA/DeepLearningExamples/tree/master/MxNet/Classification/RN50v1.5

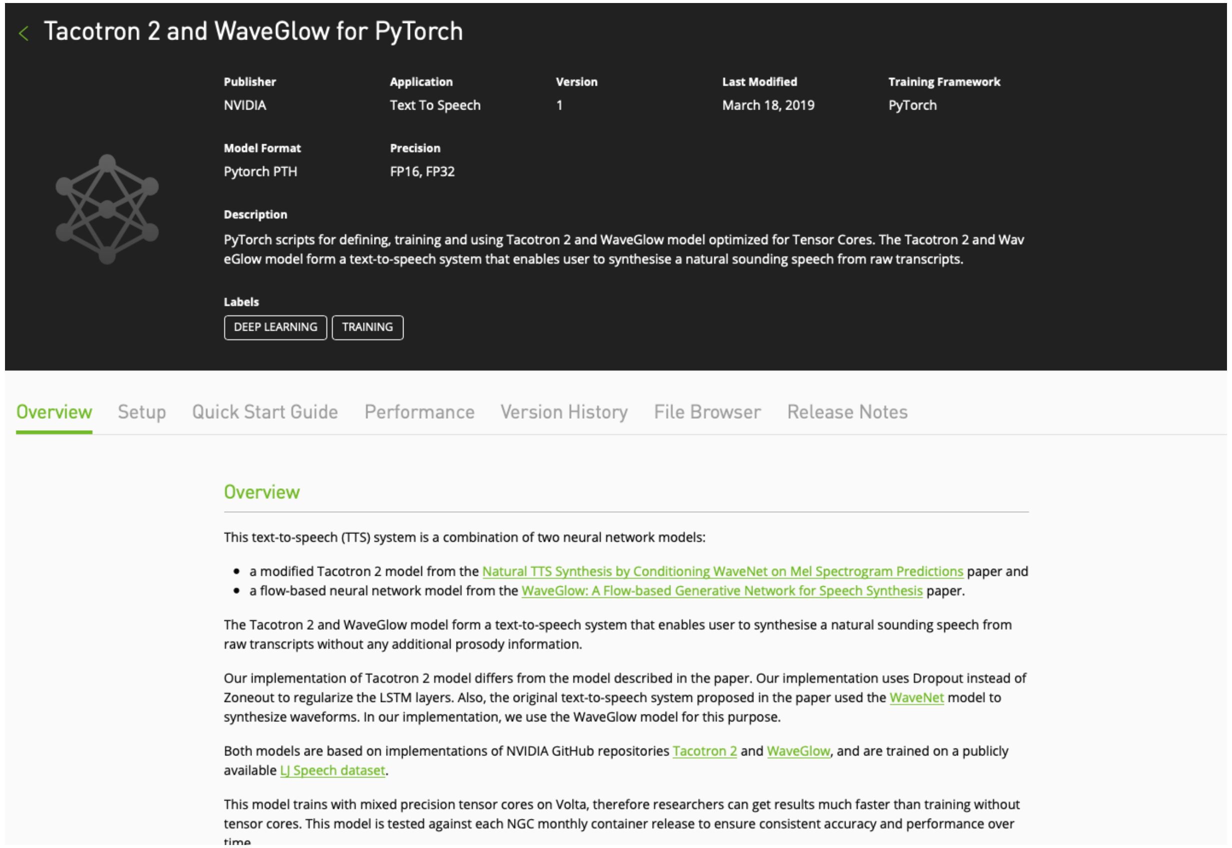
GPU:1xV100-16GB | DGX-1V | Batch Size: 208 (FP16), 96 (FP16)

SPEECH SYNTHESIS: Tacotron 2 And WaveGlow v1.0

https://ngc.nvidia.com/catalog/model-scripts/nvidia:tacotron_2_and_waveglow_for_pytorch

DGX-1V 16G	Tacotron 2 FP32	Tacotron 2 Mixed Precision	WaveGlow FP32	WaveGlow Mixed Precision
Time to Train [Hours]	**44** @ 1500 epochs	**33.14** @ 1500 epochs	**109.96** @ 1000 epochs	**54.83** @ 1000 epochs
Train Accuracy Loss (@1000 Epochs)	**0.3629**	**0.3645**	**-6.1087**	**-6.0258**
Perf.	**10,843** tokens/sec	**12,742** tokens/sec	**257,687(*)** samples/sec	**500,375(*)** samples/sec
Data set	LJ Speech Dataset			

(*) With sampling rate equal to 22050, one second of audio is generated from 22050 samples



Source: https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/SpeechSynthesis/Tacotron2

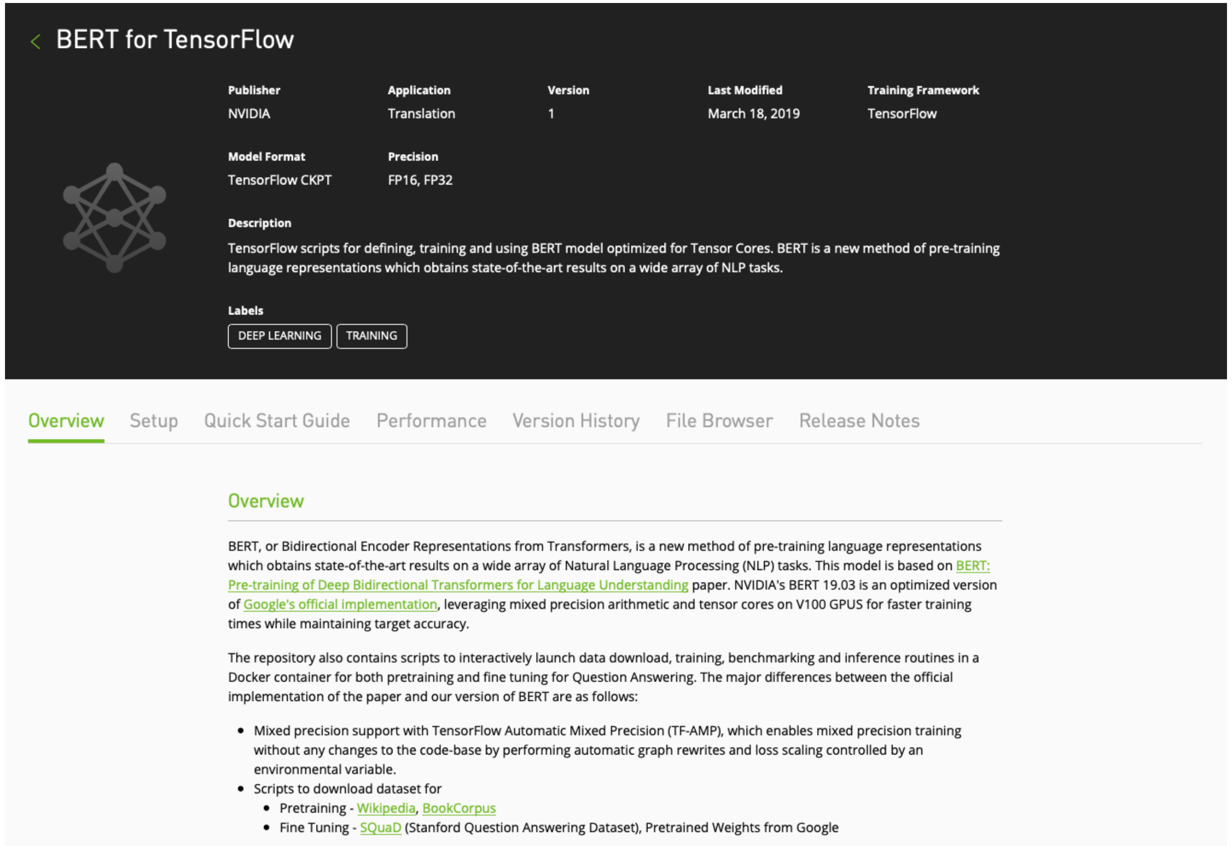
GPU:1xV100-16GB | DGX-1V | Batch Size: 208 (FP16), 96 (FP16)

LANGUAGE MODELING: BERT for TensorFlow

https://ngc.nvidia.com/catalog/model-scripts/nvidia:bert_for_tensorflow

DGX-1V **8GPU 32G**	TF BERT FP32	TF BERT Mixed Precision
Time to Train [Hours]	**0.77** (BSxGPU = 4)	**0.51** (BSxGPU = 4)
Train F1 (mean)	**90.83**	**90.99**
Perf. (BSxGPU = 4)	**66.65** sentences/sec	**129.16** sentences/sec
Data set	SQuaD (fine-tuning)	

NGC 19.03 TensorFlow container



Source: https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/LanguageModeling/BERT

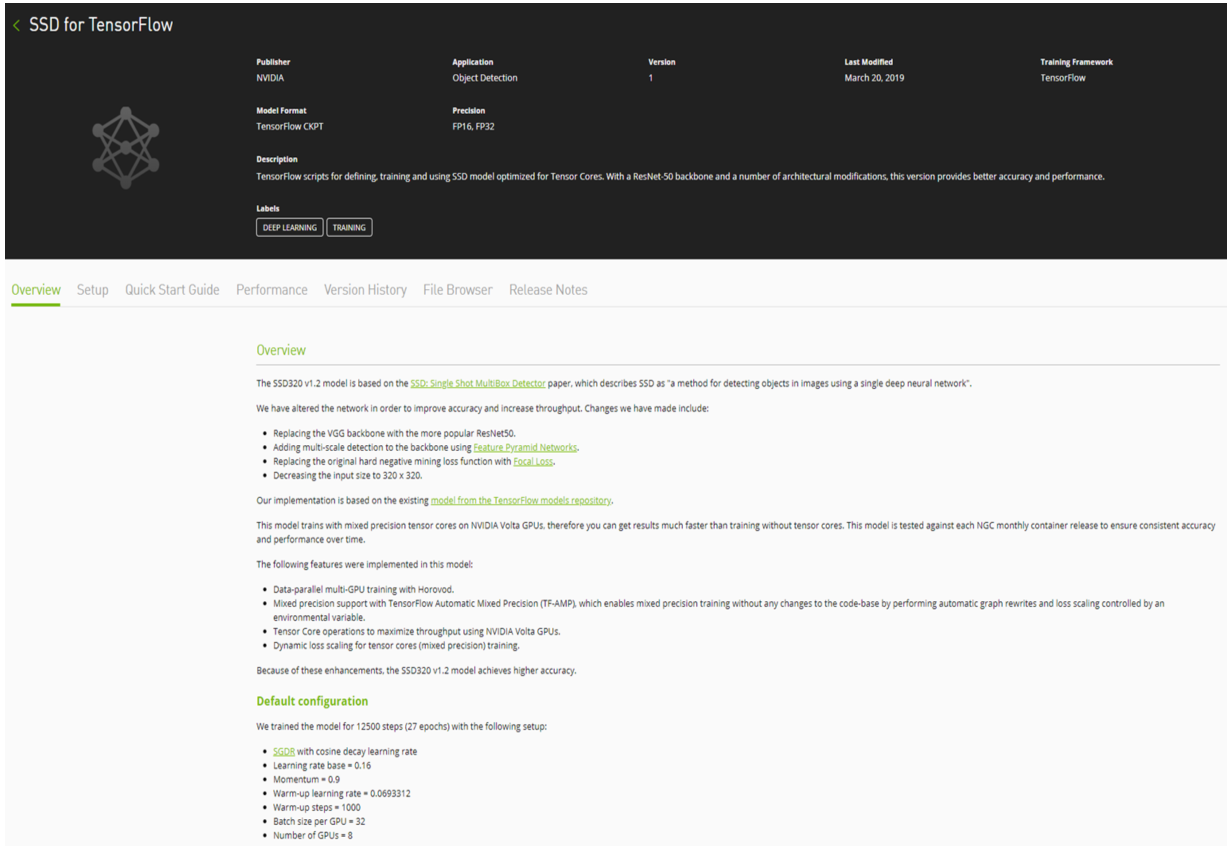
GPU:8xV100-32GB | DGX-1 | Batch size per GPU: 4

OBJECT DETECTION: TensorFlow SSD

https://ngc.nvidia.com/catalog/model-scripts/nvidia:ssd_for_tensorflow

DGX-1V **8GPU 16G**	TF SSD FP32	TF SSD Mixed Precision
Time to Train	**1h 37min**	**1h 19min**
Accuracy (map)	**0.268**	**0.269**
Perf. (BSxGPU = 32)	**569** Img/sec	**752** Img/sec
Data set	COCO 2017	

NGC 19.03 TensorFlow container



Source: https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/Detection/SSD

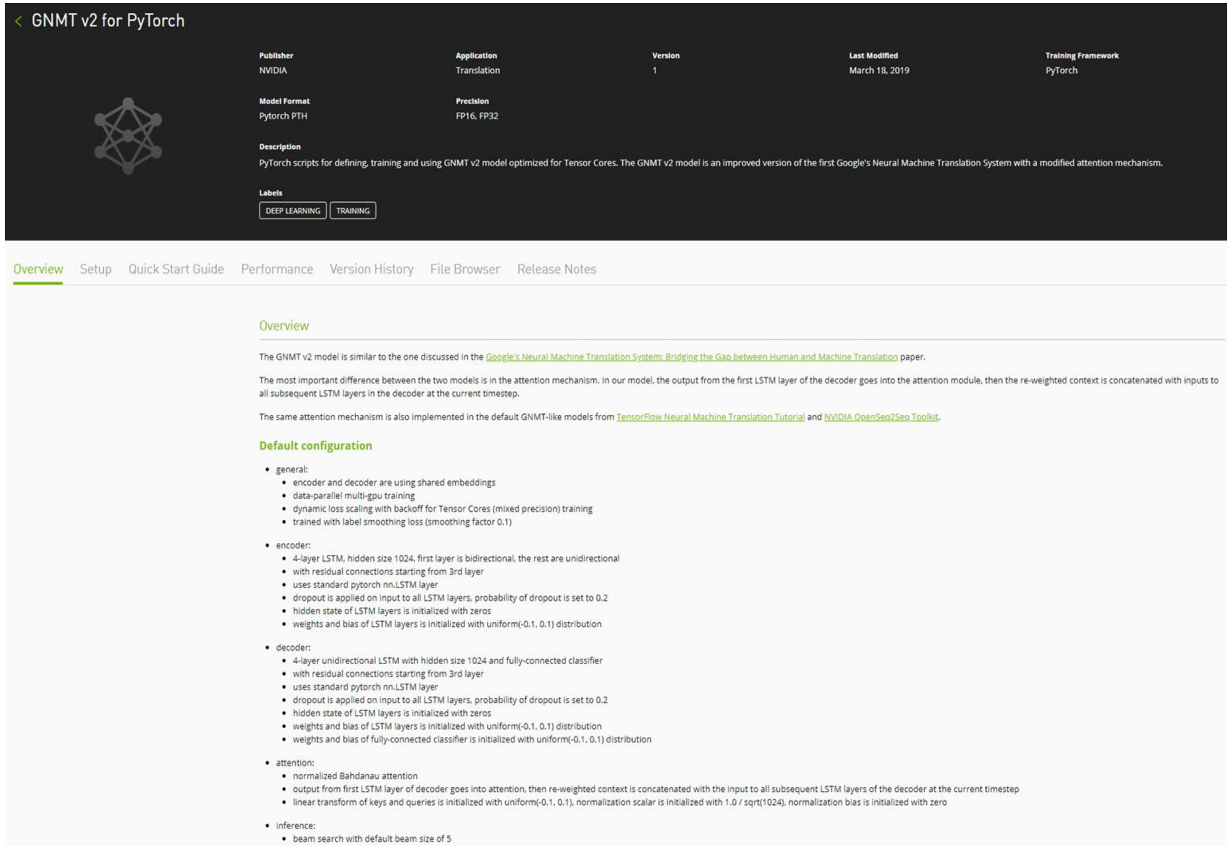
GPU:8xV100-16GB | DGX-1V | Batch Size: 32 (FP32, Mixed)

TRANSLATION: PyTorch GNMT

https://ngc.nvidia.com/catalog/model-scripts/nvidia:gnmt_v2_for_pytorch

DGX-2V 16GPU 32G	PyTorch GNMT FP32	PyTorch GNMT Mixed Precision
Time to Train [min]	**58.6**	**26.3**
Train Accuracy BLEU score	**24.16**	**24.22**
Perf.	**314.831** tokens/sec	**738,521** tokens/sec
Data set	WMT16 English to German	

NGC 19.01 PyTorch container



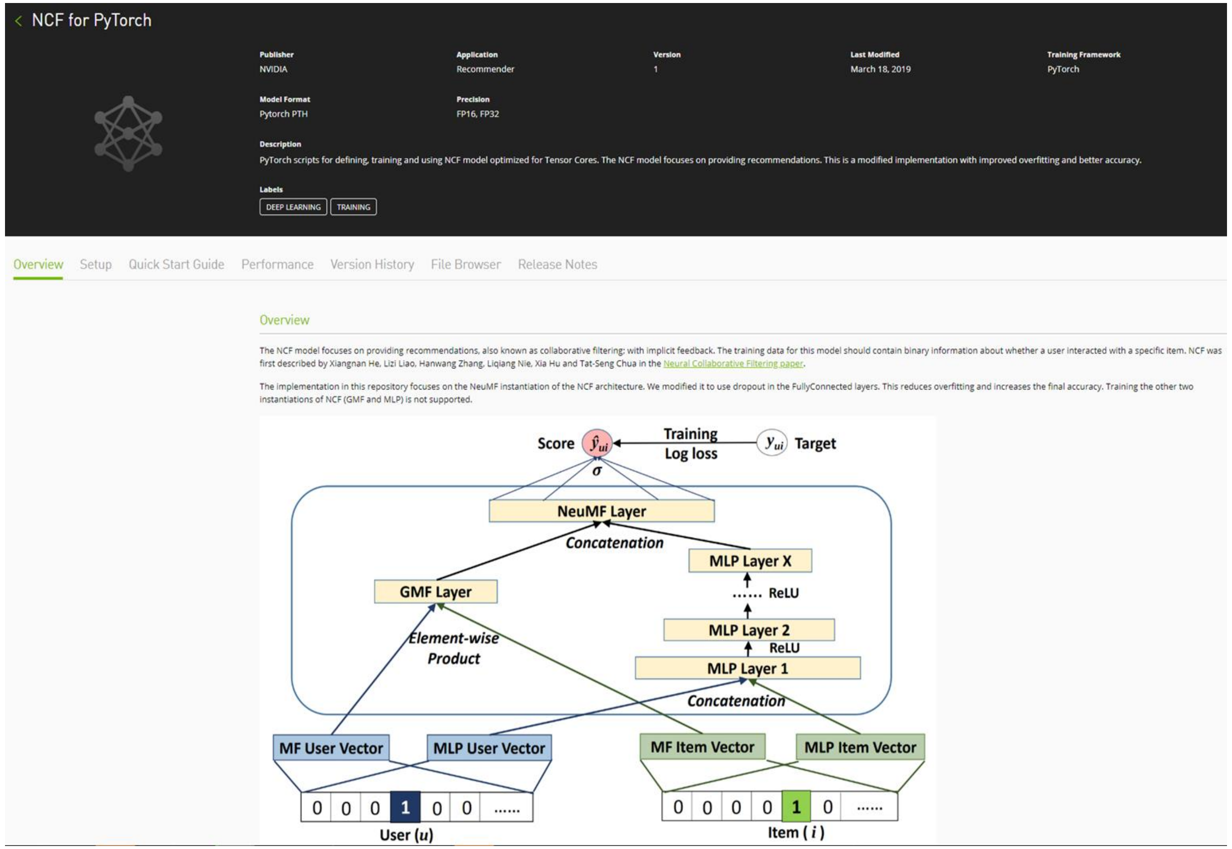
Source: https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Translation/GNMT

GPU:16xV100-32GB | DGX-2 | Batch size: 128 (FP32, Mixed)

RECOMMENDER: PyTorch Neural Collaborative Filter

https://ngc.nvidia.com/catalog/model-scripts/nvidia:ncf_for_pytorch

DGX-1V **8GPU 16G**	PyTorch NCF FP32	PyTorch NCF Mixed Precision
Time to Accuracy [seconds]	**32.68**	**20.42**
Accuracy Hit Rate @10	**0.96**	**0.96**
Perf.	**55,004,590** smp/sec	**99,332,230** smp/sec
Data set	MovieLens 20M	



NGC 18.12 PyTorch container

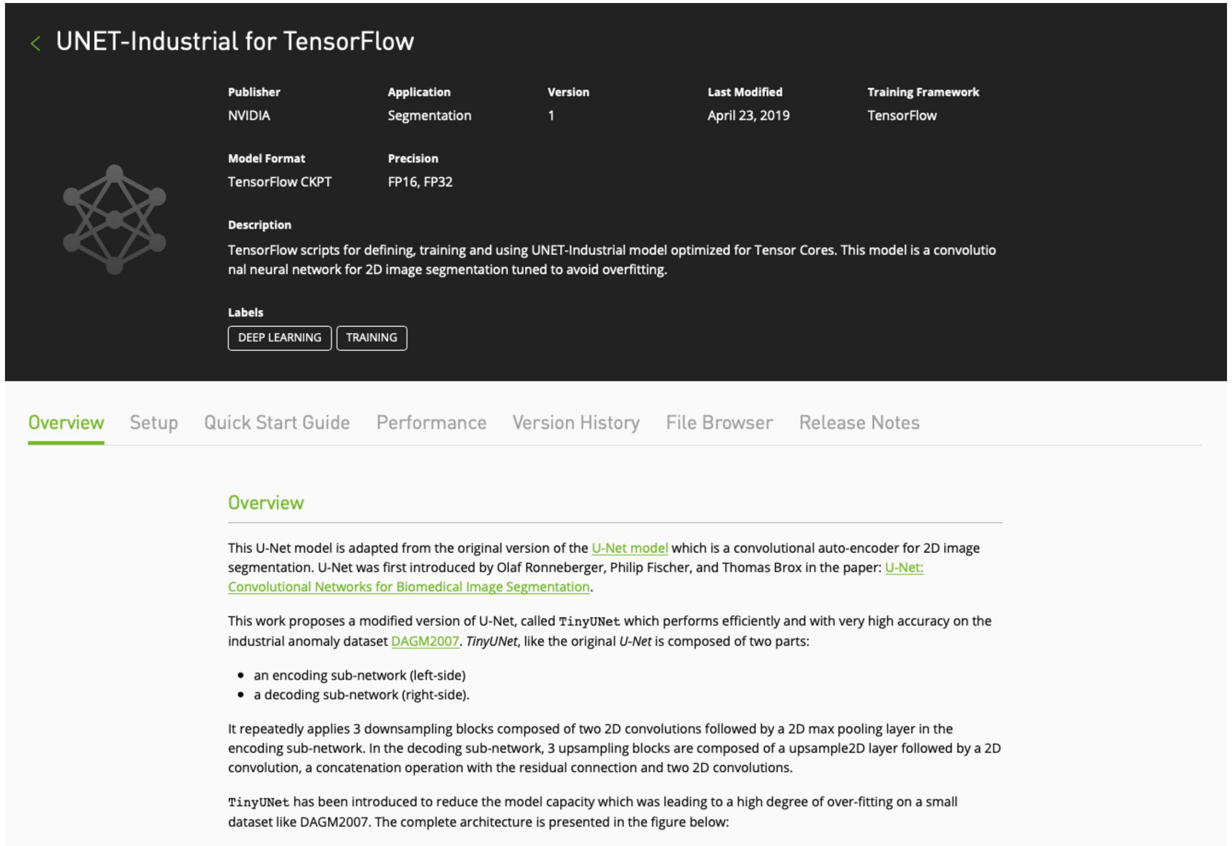
Source: https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Recommendation/NCF

GPU:8xV100-16GB | DGX-1 | Batch size: 1,048,576

INDUSTRIAL DEFECT DETECTION: TensorFlow U-Net

https://ngc.nvidia.com/catalog/model-scripts/nvidia:unet_industrial_for_tensorflow

DGX-1V **8GPU 16G**	TF U-Net FP32	TF U-Net Mixed Precision
Time to Train	**1 min 44 sec**	**1 min 36 sec**
IOU (Th=0.75 Class #4)	**0.965**	**0.960**
IOU (Th=0.75 Class #9)	**0.988**	**0.988**
Perf.	**445** Img/sec	**491** Img/sec
Data set	DAGM 2007	



NGC 19.03 TensorFlow container

Source: https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/Segmentation/UNet_Industrial

GPU:8xV100-16GB | DGX-1 | Batch size: 16

DAGM 2007 has 10 classes (for the competition). Each class has an independent IOU.

Matching Accuracy for FP32 and Mixed Precision

Model Script	Framework	Data Set	Automatic or Manual Mixed-Precision	FP32 Accuracy	Mixed-Precision Accuracy	FP32 Throughput	Mixed-Precision Throughput	Speedup
BERT Q&A (2)	**TensorFlow**	SQuaD	AMP	90.83 Top 1	90.99 Top 1	66.65 sentences/sec	129.16 sentences/sec	1.94
SSD w/RN50 (1)	**TensorFlow**	COCO 2017	AMP	0.268 mAP	0.269 mAP	569 images/sec	752 images/sec	1.32
GNMT (3)	**PyTorch**	WMT16 English to German	Manual	24.16 BLEU	24.22 BLEU	314,831 tokens/sec	738,521 tokens/sec	2.35
Neural Collaborative Filter (1)	**PyTorch**	MovieLens 20M	Manual	0.959 HR	0.960 HR	55,004,590 samples/sec	99,332,230 items/sec	1.81
U-Net Industrial (1)	**TensorFlow**	DAGM 2007	AMP	0.965-0.988	0.960-0.988	445 images/sec	491 images/sec	1.10
ResNet-50 v1.5 (1)	**MXNet**	ImageNet	Manual	76.67 Top 1%	76.49 Top 1%	2,957 images/sec	10,263 images/sec	3.47
Tacotron 2 / WaveGlow 1.0 (1)	**PyTorch**	LJ Speech Dataset	AMP	0.3629/ -6.1087	0.3645/ -6.0258	10,843 tok/s 257,687 smp/s	12,742 tok/s 500,375 smp/s	1.18/ 1.94

Values are measured with model running on (1) **DGX-1V 8GPU 16G,** (2) **DGX-1V 8GPU 32G or** (3) **DGX-2V 16GPU 32G**