INTRODUCTION TO MIXED PRECISION TRAINING

Dusan Stosic

NVIDIA

OUTLINE

1. Mixed Precision Hardware
2. What is Mixed Precision Training?
3. Considerations for Mixed Precision
4. Mixed Precision Software
5. Performance Guidelines

MIXED PRECISION HARDWARE

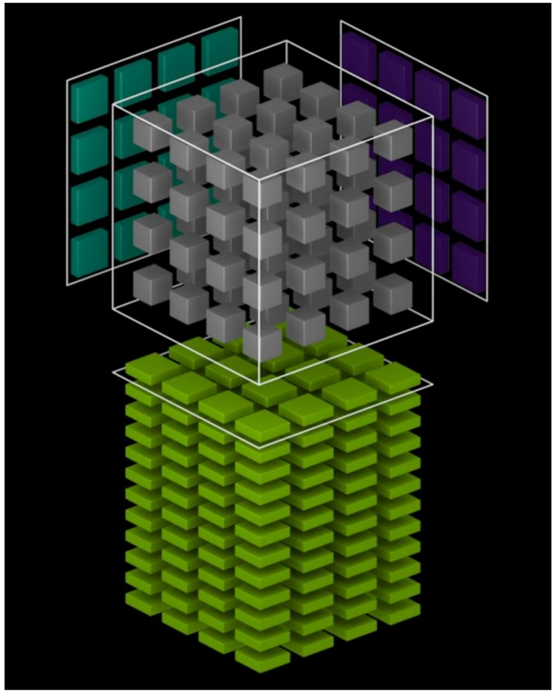
NVIDIA VOLTA AND TURING GPUS

Tensor Cores for mixed precision training and inference

Math Speed Ups

	FP32	FP16	INT8	INT4	INT1
VOLTA	1x	**8x**	4x		
TURING	1x	**8x**	**16x**	**32x**	**128x**

Tensor Cores are in bold



WHAT ARE TENSOR CORES?

Tensor cores are...

...special hardware execution units

...execute matrix multiply operations

...built to accelerate deep learning

Two flavors

- **Volta Tensor Cores** FP16
- **Turing Tensor Cores** FP16/INT8/INT4/INT1

VOLTA ARCHITECTURE

TENSOR CORES

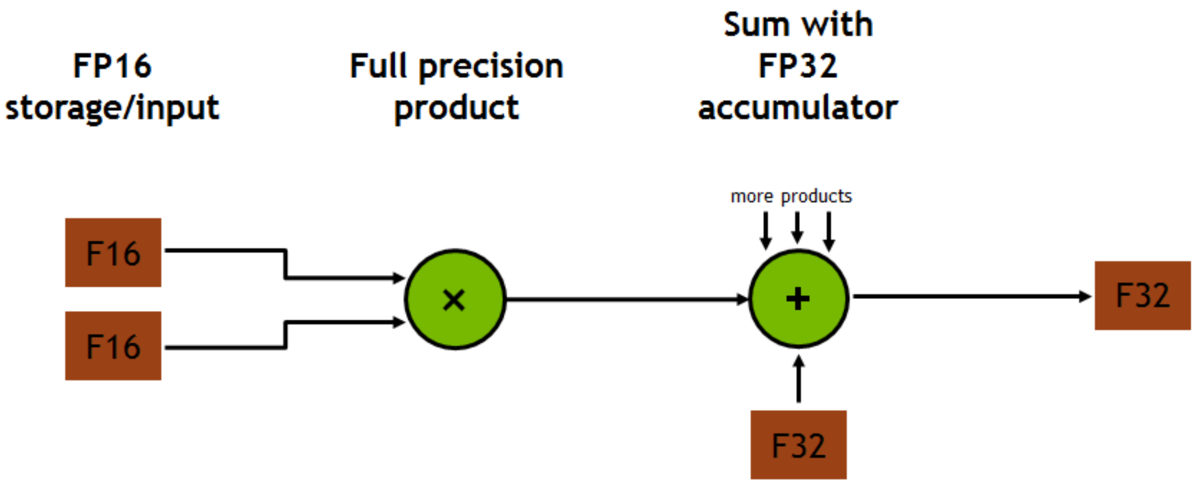
Mixed precision matrix math on 4x4 matrices

$$\underset{\text{FP16 or FP32}}{D} = \underset{\text{FP16}}{\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}} \underset{\text{FP16}}{\begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}} + \underset{\text{FP16 or FP32}}{\begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}}$$

$$D = AB + C$$

INTERNALS OF TENSOR CORES (VOLTA)

Inherently mixed precision



Accelerate matrix multiplications and convolutions

Tensor Core Optimized Libraries: cuDNN and cuBLAS

https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/

WHAT IS MIXED PRECISION TRAINING?

WHAT IS MIXED PRECISION TRAINING?

In a nutshell

Idea that you can train deep neural networks in multiple precisions:

- Make precision decisions per layer or operation
- Full precision (FP32) where needed to *maintain task-specific accuracy*
- Reduced precision (FP16) everywhere else for *speed* and *scale*

By using *multiple precisions*, we can have the best of both worlds: **speed** and **accuracy**

Goal: accelerate deep neural network training with mixed precision under the constraint of **matching accuracy of full precision training** and **no changes to how model is trained**

MIXED PRECISION TRAINING

Benefits

Accelerates math

- Tensor Cores are 8x faster than FP32

Reduces memory bandwidth pressure

- FP16 halves memory traffic compared to FP32

Reduces memory consumption

- FP16 halves the size of activation and gradient tensors
- Enables larger models, mini batches or inputs

WHY USE MIXED PRECISION?

Networks can’t always train properly in pure FP16

Need to keep some things in FP32 (need more mantissa)

...weight updates

- optimizer takes very ***small increments*** when search narrows into a solution
- late updates often cannot be represented in FP16, but can be crucial for accuracy

...reductions

- large sums of values, e.g. in linear layers and convolutions, can be ***too big*** for FP16
- adding small values to a large sum can lead to ***rounding errors***

MIXED PRECISION TRAINING

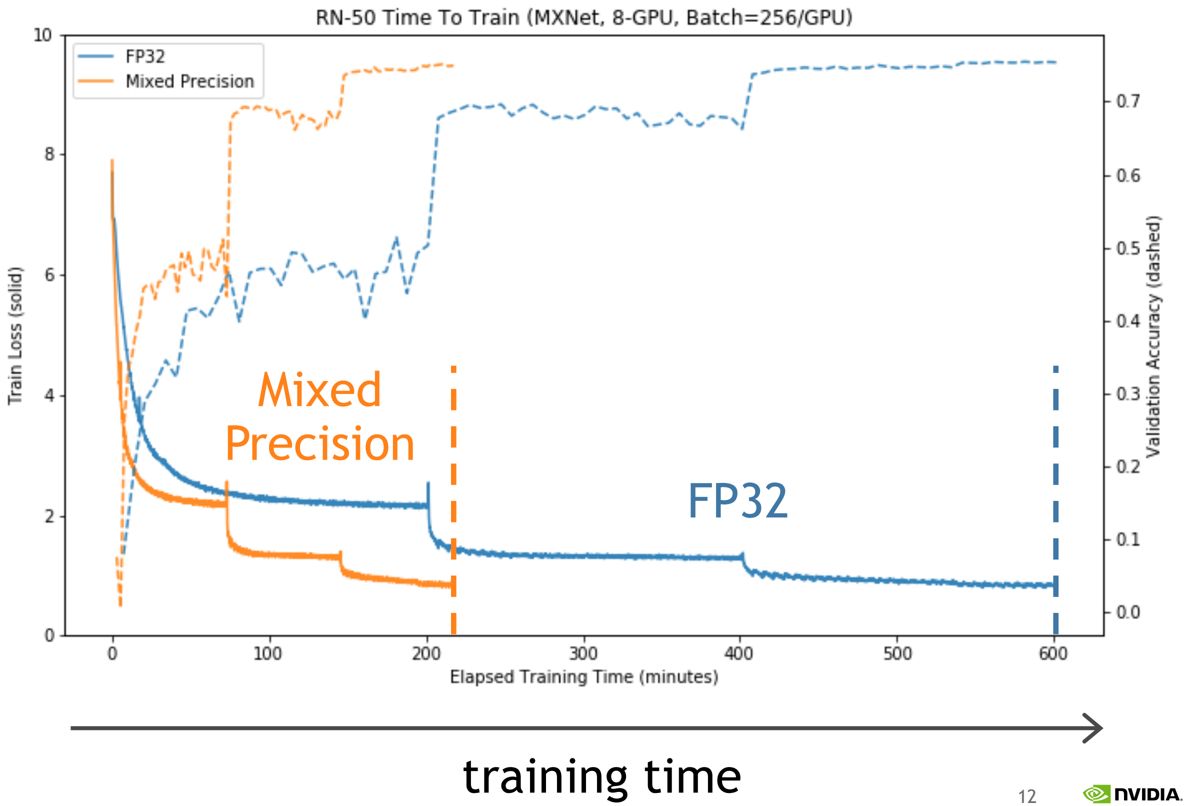
Example

ResNet-50 training for ImageNet classification

- 8 GPUs on DGX-1
- NVIDIA MXNet container

Comparing to FP32 training

- **>3x speedup**
- **equal accuracy**
- no hyperparameters changed



MIXED PRECISION IS GENERAL PURPOSE

Models trained to match FP32 results (same hyperparameters)

Works across a *wide* range of **tasks, problem domains, deep neural network architectures**

Image Classification
AlexNet
DenseNet
Inception
MobileNet
NASNet
ResNet
ResNeXt
ShuffleNet
SqueezeNet
VGG
Xception

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

Recommendation
DeepRecommender
NCF

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

Speech
Deep Speech 2
Jasper
Tacotron
Wave2vec
WaveNet
WaveGlow

Language Modeling
BERT
BigLSTM
Gated Convolutions
mLSTM
RoBERTa
Transformer XL

Translation
Convolutional Seq2Seq
Dynamic Convolutions
GNMT (RNN)
Levenshtein Transformer
Transformer (Self-Attention)

MIXED PRECISION TRAINS FASTER

Drastically reduces training time

Task	Model	Speedup
Image Classification	ResNet-50	3.6x
	DenseNet 201	2.2x*
	Xception	2.1x*
Detection / Segmentation	SSD	2.5x**
	Mask R-CNN	1.5x
	RetinaNet	2.0x

* 2x batch size ** Larger batch size

Task	Model	Speedup
Translation	GNMT	2.3x
	Transformer	2.9x 4.9x**
	Convolutional Seq2Seq	2.5x*
Speech	Deep Speech 2	4.5x**
	Wav2letter	3.0x*
	WaveGlow	1.9x
Language Modeling	mLSTM	4.0x**
	BERT	3.3x

...weeks to days

... days to hours

... hours to minutes

MIXED PRECISION ADVANCES DL RESEARCH

Both *accelerates* and *enables* novel research

“This paper shows that reduced mixed precision and large batch training can speedup training by nearly **5x** on a single 8-GPU machine with careful tuning and implementation.”

Scaling Neural Machine Translation, Facebook

“We train with mixed precision floating point arithmetic on DGX-1 machines [...] using 1024 V100 GPUs for approximately **one day**.”

RoBERTa, Facebook

“leverages mixed precision training [...] **largest transformer based language model ever trained** at 24x the size of BERT and 5.6x the size of GPT-2.”

MegatronLM, NVIDIA

CONSIDERATIONS FOR MIXED PRECISION

CONSIDERATIONS FOR MIXED PRECISION TRAINING

Goal #1: Make FP16 training general purpose, not only for limited class of applications

Goal #2: With no changes to hyperparameters or the network architecture

Three parts:

PRECISION OF OPS

Decide which operations to compute in FP16 and FP32.

MASTER WEIGHTS

Keep an FP32 copy of the model weights.

LOSS SCALING

Scale the loss value to retain small gradients.

PRECISION OF OPS

Precision choices for different classes of operations

Matrix Multiplication
linear, matmul, bmm, conv

8x performance boost from Tensor Cores

Pointwise
relu, sigmoid, tanh, exp, log

Reductions
batch norm, layer norm, sum, softmax

Loss Functions
cross entropy, l2 loss, weight decay

still get some speedup (e.g. 2x memory savings), but without sacrificing accuracy

PRECISION OF OPS

Conservative recommendations

Operations that can use FP16 storage

- matrix multiplications
- most pointwise operations (e.g. relu, tanh, add, sub, mul)

Operations that need FP32 mantissa

- reduction operations

Operations that need FP32 range

- pointwise operations where $|f(x)| \gg |x|$, e.g. exp, log, pow
- loss functions

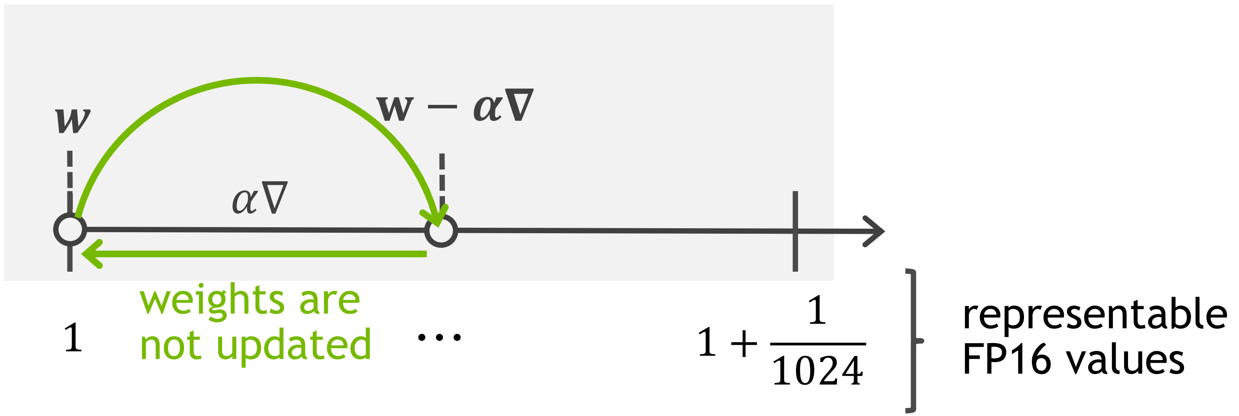
MASTER WEIGHTS

FP16 can be insufficient for weight updates

$$w_{t+1} = w_t - \alpha\nabla_t$$

Problem: in late stages of training, weight updates become too small for addition in FP16

Consequence: weight update gets clipped to zero when $w \gg \alpha\nabla$

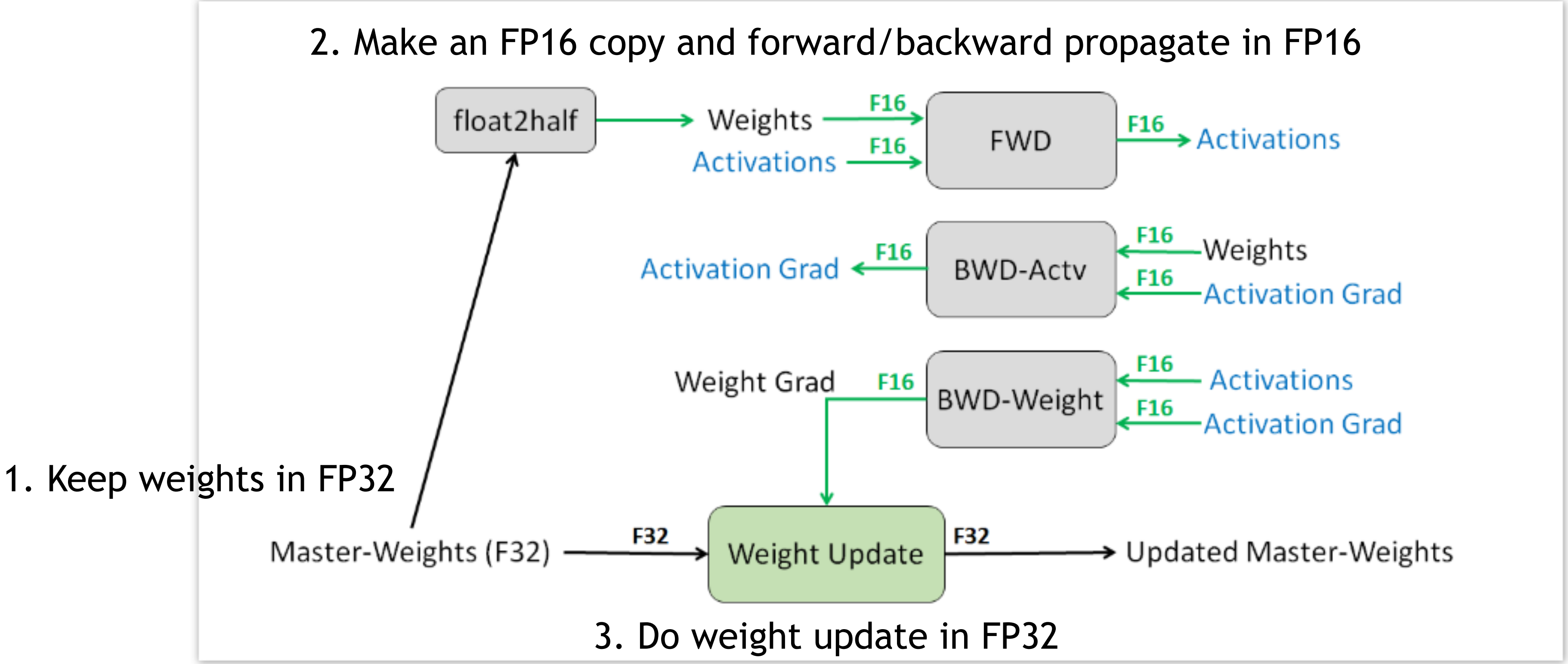


Example:
$w \times 0.01$ leads to $\sim 2^7$ ratio,
$w \times 0.001$ leads to $\sim 2^{10}$ ratio

Conservative solution: keep master copy of weights in FP32 so small updates can accumulate

MASTER WEIGHTS

Illustration



LOSS SCALING

Put all tensors in FP16 range

Range representable in FP16: ~40 powers of 2

Gradients are small:

some lost to zero

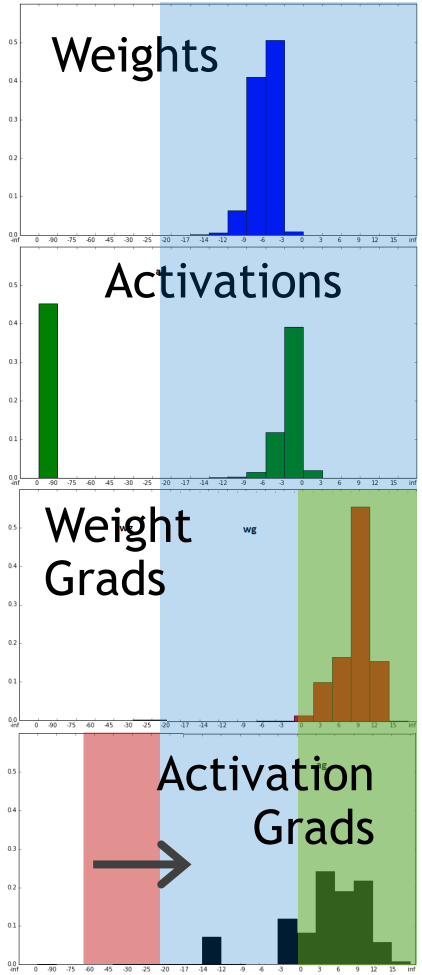
while ~15 powers of 2 remain unused

Solution:

move small gradient values to FP16 range

multiply loss by a constant factor

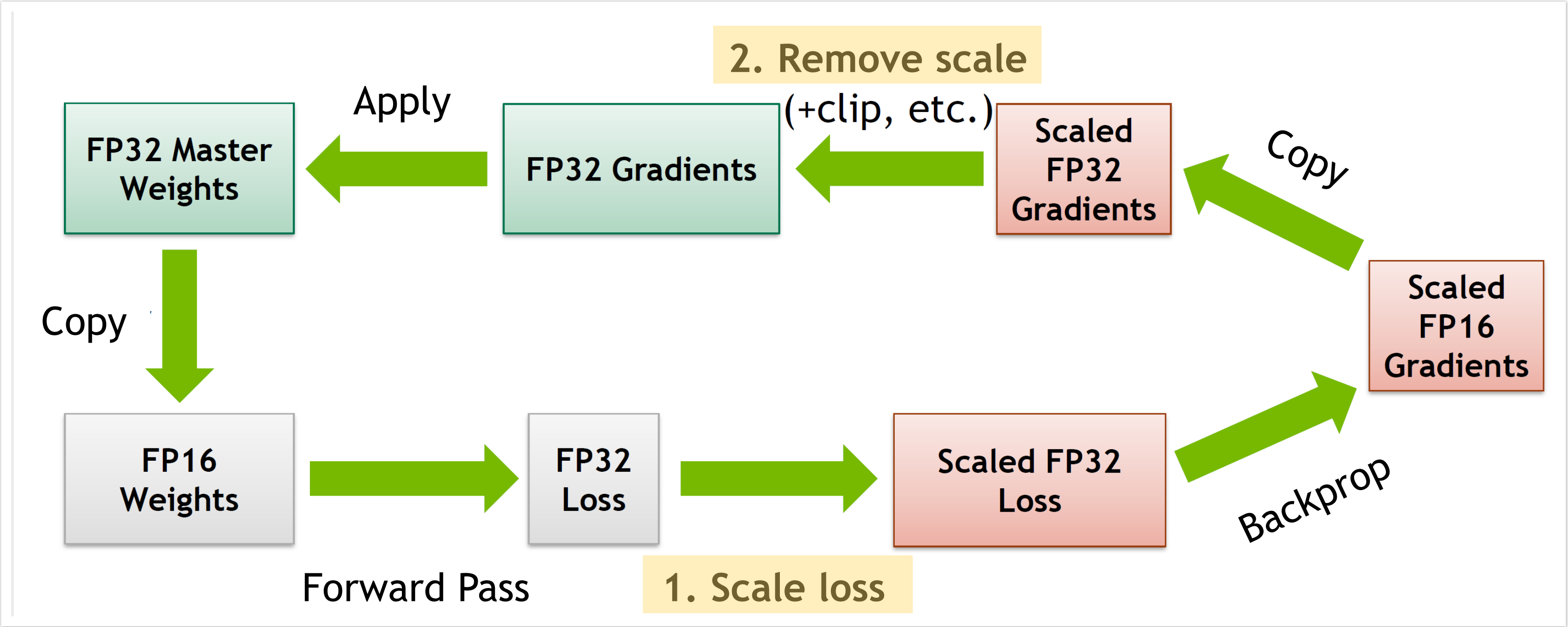
gradients are scaled (shifted) by chain rule



NVIDIA

LOSS SCALING

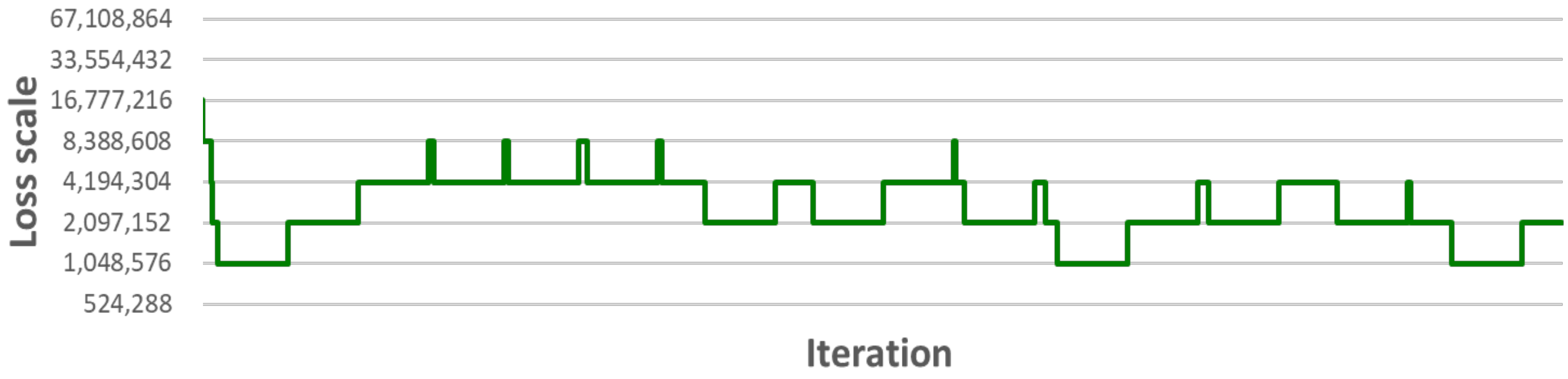
Algorithm



AUTOMATIC LOSS SCALING

Frees user from choosing a scaling factor

1. Start with a very large scale factor (e.g. **2^24**)
2. If gradient overflows (with ***Inf*** or a ***NaN***): decrease the scale by **2** and skip the update
3. If no overflows have occurred for some time (e.g. **2000** iterations): increase the scale by **2**



https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html#scalefactor

MIXED PRECISION SOFTWARE

WHAT IS MIXED PRECISION SOFTWARE?

Integration into deep learning frameworks

Goal: Make mixed precision training easy with minimum effort for the DL practitioner

Available for major Deep Learning Frameworks

TensorFlow | PyTorch | MXNet

Works with all types of optimizers

- SGD, Adam, AdaGrad, etc

Works with *multiple* models, optimizers, and losses

MIXED PRECISION SOFTWARE

Feedback

"Nuance Research advances and applies conversational AI technologies to power solutions that redefine how humans and computers interact. The rate of our advances reflects the speed at which we train and assess deep learning models. With Automatic Mixed Precision, we've realized a **50% speedup** in TensorFlow-based ASR model training without loss of accuracy via a minimal code change. We're eager to achieve a similar impact in our other deep learning language processing applications."

Wenxuan Teng, Senior Research Manager, Nuance Communications

"Automated mixed precision powered by NVIDIA Tensor Core GPUs on Alibaba allows us to instantly speedup AI models nearly **3X**. Our researchers appreciated the ease of turning on this feature to instantly accelerate our AI"

Wei Lin, Senior Director at Alibaba Computing Platform, Alibaba

"TensorFlow developers will greatly benefit from NVIDIA automatic mixed precision feature. This easy integration enables them to get up to **3X** higher performance with mixed precision training on NVIDIA Tensor Core GPUs while maintaining model accuracy."

Rajat Monga, Engineering Director, TensorFlow, Google

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

ENABLING MIXED PRECISION

Traditionally a lot of work (recap on methodology)

1. Model conversion

- switch everything to run on FP16 values
- cast to FP32 for loss functions, normalization, and pointwise ops that need full precision

2. Master weights

- keep FP32 copy of model parameters
- make FP16 copies during forward / backward passes

3. Loss scaling

- scale the loss value, unscale the gradients in FP32
- check gradients at each iteration to adjust loss scale and skip on overflow

AUTOMATIC MIXED PRECISION

Automates *everything* from the previous slide

Develop ***framework software*** to run mixed precision ***automatically***

Details vary by framework, but core idea is the same

Automatic **M**ixed **P**recision (**AMP**) does two things:

AUTOMATIC LOSS SCALING

Wraps the optimizer in order to:
- scale loss value & unscale gradients
- adjust scale & skip on gradient overflow

AUTOMATIC CASTING

Wraps model and operations in order to:
- cast data to FP16
- switch *everything* to run on FP16
- keep certain operations in FP32
- keep master copy of weights in FP32

AUTOMATIC CASTING

A primer

1. Make type decisions for each operation *a priori* (static graph) or at *runtime* (eager execution)

2. Define *conservative* set of rules to replace "by-hand" mixed precision

Divide the universe of operations into three kinds:

WHITELIST

FP16 enables Tensor Cores.

e.g. linear, bmm, convs

Rule: always run in FP16, cast if necessary.

BLACKLIST

FP32 is needed for accuracy.

e.g. loss, exp, sum, softmax

Rule: always run in FP32, cast if necessary.

EVERYTHING ELSE

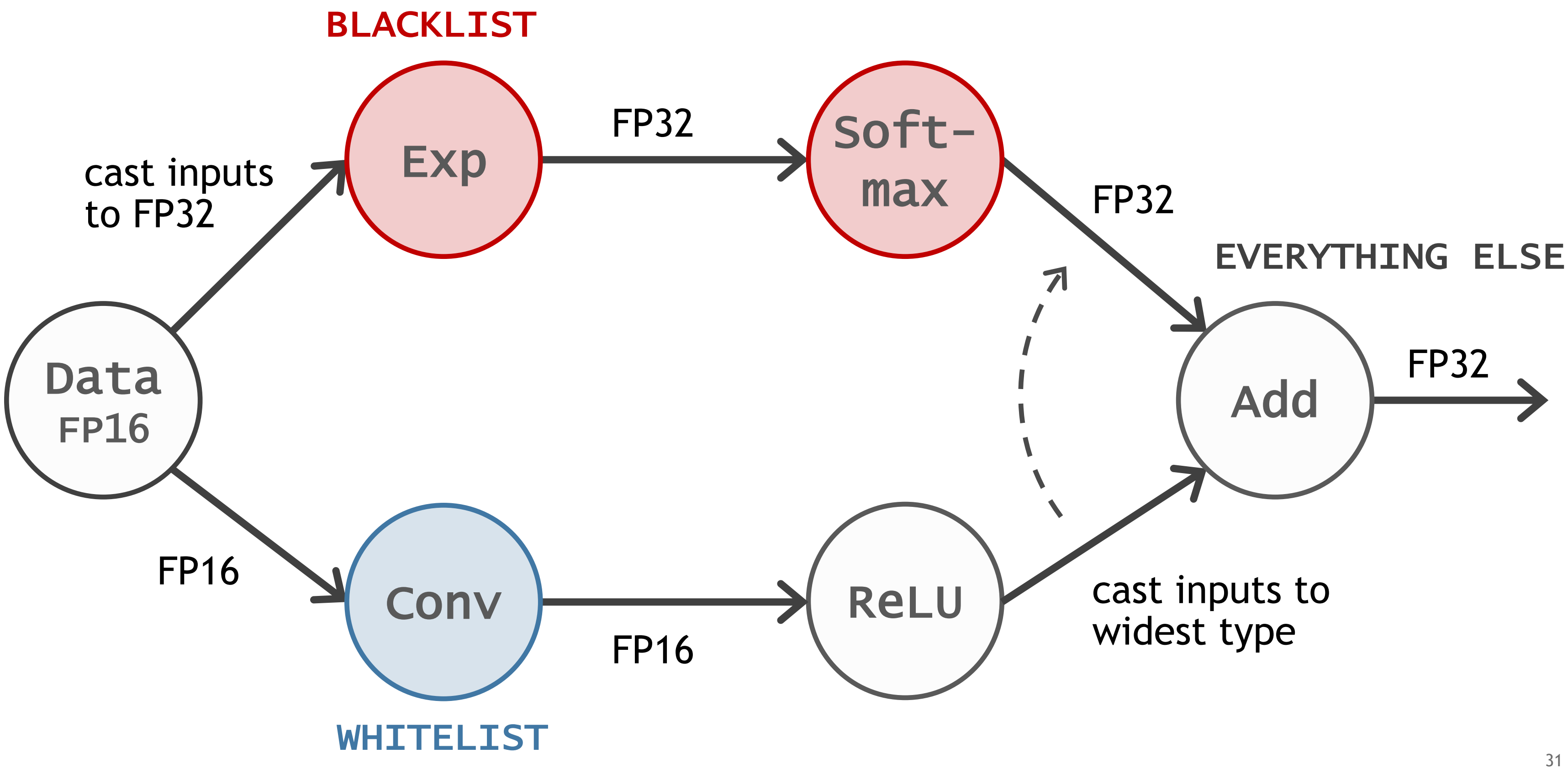
Can run in FP16, but only if inputs already in FP16.

e.g. relu, add, maxpool

Rule: run in existing input type.

AUTOMATIC CASTING

Graph example



AMP FOR TENSORFLOW

As simple as one environment variable

Easiest way is to set an environment variable

```
export TF_ENABLE_AUTO_MIXED_PRECISION=1
```

But can also wrap the optimizer (TensorFlow 1.14 or later)

```
opt = tf.train()
opt = tf.train.experimental.enable_mixed_precision_graph_rewrite(opt)
train_op = opt.minimize(loss)
```

AMP FOR TENSORFLOW

More advanced options

To separately enable automatic casting and automatic loss scaling

```
export TF_ENABLE_AUTO_MIXED_PRECISION_GRAPH_REWRITE=1
export TF_ENABLE_AUTO_MIXED_PRECISION_LOSS_SCALING=1
```

e.g. if your code already supports automatic loss scaling

To make AMP aware of custom operations

ops worth casting to FP16

```
export TF_AUTO_MIXED_PRECISION_GRAPH_REWRITE_WHITELIST_ADD=‘Op1’
```

ops required in FP32

```
export TF_AUTO_MIXED_PRECISION_GRAPH_REWRITE_BLACKLIST_ADD=‘Op2’
```

ops that can be in either FP16 or FP32

```
export TF_AUTO_MIXED_PRECISION_GRAPH_REWRITE_GRAYLIST_ADD=‘Op3’
```

AMP FOR PYTORCH

As simple as two lines of code

Wrap the model and optimizer

```
model, optimizer = amp.initialize(model, optimizer)
```

Apply automatic loss scaling and backpropagate with scaled loss

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

AMP FOR PYTORCH

More advanced options

To control the operations being casted

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

O0	O1	O2
FP32 Training Leave everything in FP32.	**Mixed Precision Training** FP16 whitelist and FP32 blacklist ops.	**FP16 Training** FP16 model/data with FP32 batchnorm.

AMP FOR PYTORCH

An example

allows AMP to perform automatic casting

```
import torch
import amp
model = ...
optimizer = ...
model, optimizer = amp.initialize(model, optimizer, opt_level="O1")
for data, label in data_iter:
    out = model(data)
    loss = criterion(out, label)
    optimizer.zero_grad()
    with amp.scaled_loss(loss, optimizer) as scaled_loss:
        scaled_loss.backward()
optimizer.step()
```

replaces `loss.backward()`

AMP FOR MXNET

As simple as three lines of code

Initialize AMP by changing behavior of operations

```
amp.init()
```

Wrap the Gluon trainer

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

Apply automatic loss scaling and calculate gradients with respect to scaled loss

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

AMP FOR MXNET

An example

```
from mxnet.contrib import amp
amp.init()
net = ...
trainer = mx.gluon.Trainer(...)
amp.init_trainer(trainer)
for data, label in data_iter:
    out = net(data)
    loss = criterion(out, label)
    optimizer.zero_grad()
    with amp.scaled_loss(loss, optimizer) as scaled_loss:
        mx.autograd.backward(scaled_loss)
trainer.step()
```

changes behavior of ops to follow AMP rules

initializes Gluon Trainer for AMP

replaces `mx.autograd.backward(loss)`

TRY AMP TODAY

For TensorFlow: https://docs.nvidia.com/deeplearning/frameworks/tensorflow-user-guide/index.html#tfamp

For PyTorch: https://nvidia.github.io/apex/amp.html

For MXNet: https://mxnet.apache.org/api/python/docs/tutorials/performance/backend/amp.html

AMP Examples: https://github.com/NVIDIA/DeepLearningExamples

PERFORMANCE GUIDELINES

DEBUGGING MIXED PRECISION

What to watch for

Bugs in code for mixed precision *often* manifest as slightly worse training accuracy

Common mistakes:

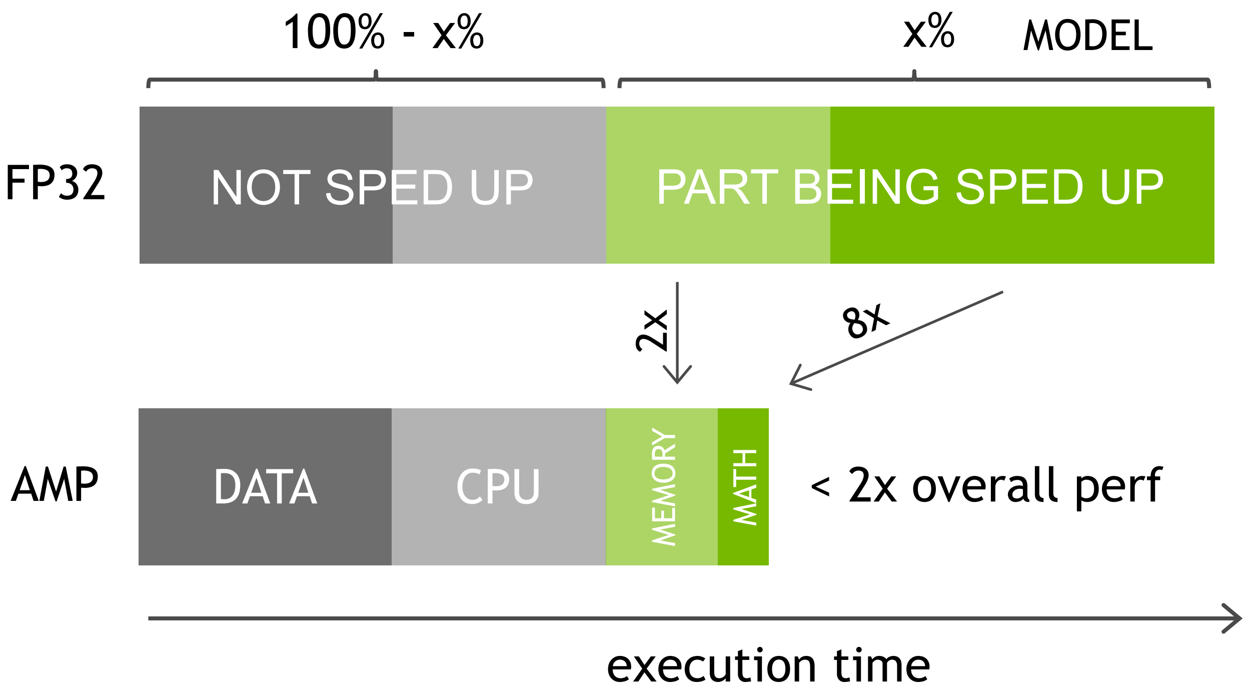
- Gradients not unscaled correctly before weight update
- Gradient clipping or regularization improperly using scaled gradients
- Not computing loss function (or some other op) in FP32

Highly recommend using automatic mixed precision tools

WHAT PERFORMANCE TO EXPECT?

Mixed precision speedup depends on where the time is being spent

Example: If 1/2 of your training routine runs on mixed precision, then the maximum speedup is 2x, even if mixed precision were infinitely fast



Amdhal’s Law
If you speed up x% of your training time, then the (100-x)% will limit your speedup

MIXED PRECISION PERFORMANCE

Varies across tasks/problem domains/architectures

What can you do to improve mixed precision performance?

A few guidelines on what to look for

DATA PIPELINE

Get the overhead from input data pipeline in your training session

MATH OPERATIONS

Find network time spent on math-bound operations (e.g. linear, convolutions)

TENSOR CORES

Analyze & improve Tensor Core utilization

DATA PIPELINE

Input data pipeline can be expensive for vision tasks

Set of operations performed on input data

- e.g. crop, resize, augment, and shuffle

Problem:

- **unoptimized** and/or **on cpu** and **slow**

Can make up +50% of training time

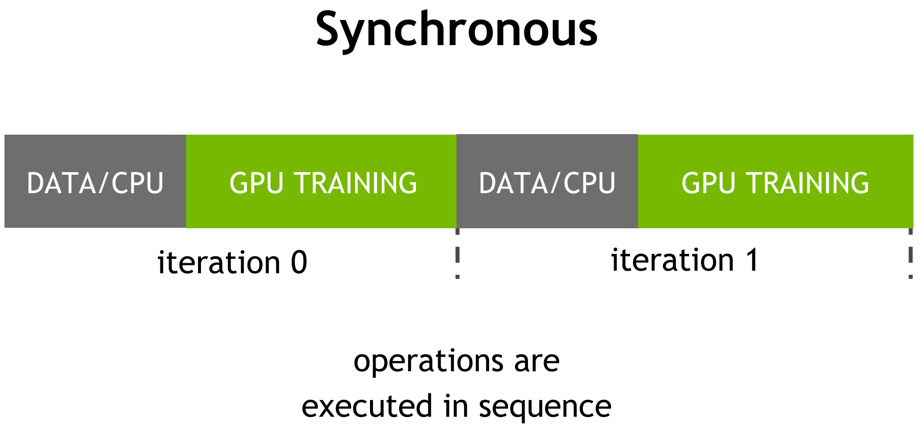
- limits the **achievable** speedup with mixed precision

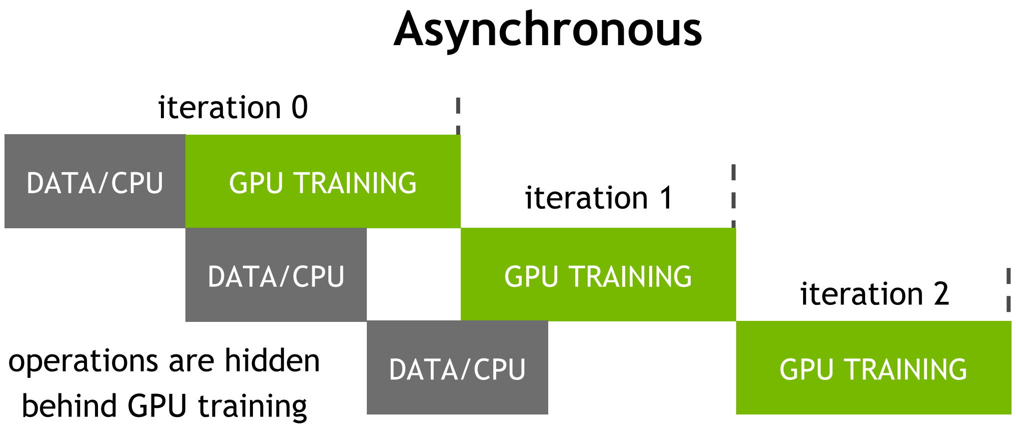
ANALYZING THE DATA PIPELINE

Need to be careful with asynchronous work

Data pipeline and CPU work are often ***asynchronous*** to model training

Hides I/O latency and CPU operations behind GPU model training



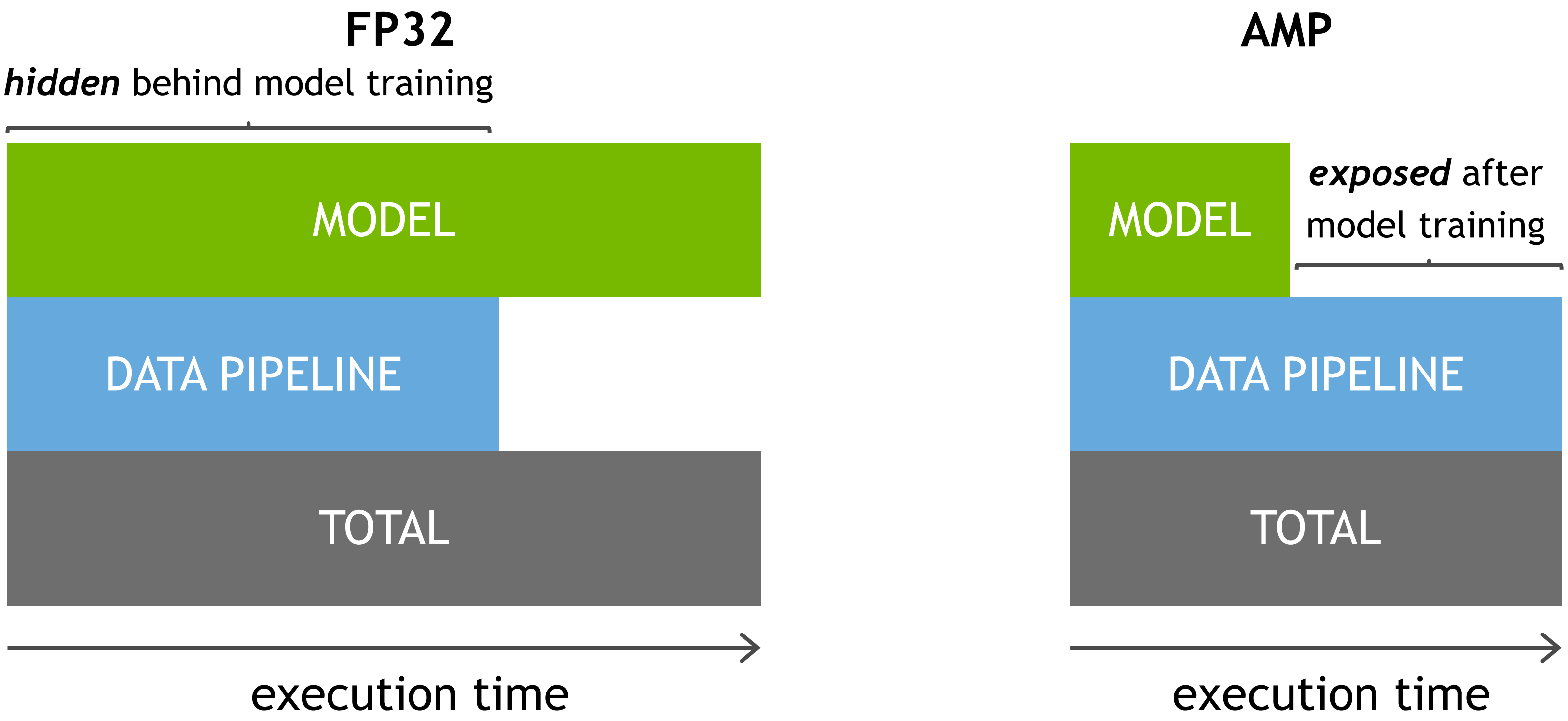


Make sure to synchronize *explicitly* with GPU work when using python timers

ANALYZING THE DATA PIPELINE

Mixed precision *exposes* the data pipeline

***Exposed* data pipeline time** is the difference between the total and model times



ANALYZING THE DATA PIPELINE

How to insert profiling?

Get the total time

```
start = time.time()
for (x,y) in enumerate(data_loader):
    loss = model(x, y)
    loss.backward()
    optimizer.step()
torch.cuda.synchronize()
total_time = time.time() - start
```

add python timers

wait for GPU work to finish

Get the model time

```
x, y = next(iter(data_loader))
start = time.time()
for i in range(len(data_loader)):
    loss = model(x, y)
    loss.backward()
    optimizer.step()
torch.cuda.synchronize()
model_time = time.time() - start
```

preload a single input of data

Make loops big enough to amortize overlap/variance

ANALYZING THE DATA PIPELINE

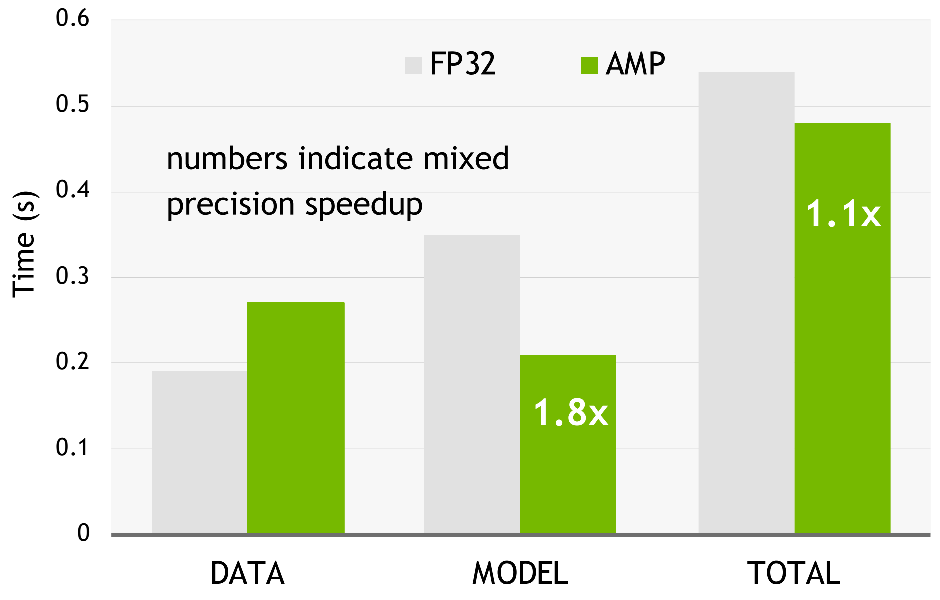
A profiling example

RetinaNet w/ RN50 backbone training on DGX-1, batch size 32 p/ GPU

Run for a few iterations, take average of times

Observations:

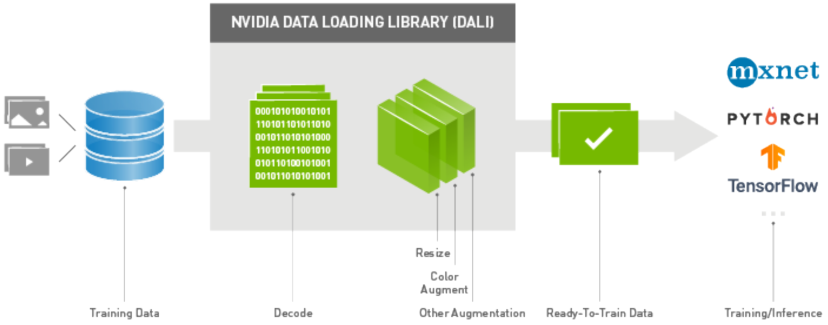
- 1/3 time spent on input data pipeline
- almost 2x speedup from model
- *low* end-to-end mixed precision perf
  is ***mostly*** because of the input data pipeline



IMPROVING THE DATA PIPELINE

Eliminating the input data pipeline costs

1. Increase the number of worker threads
2. Do data pre-processing offline before training
   - but need to store copy of data
3. Employ NVIDIA Data Loading Library (DALI)
   - GPU-accelerated data augmentation and image loading



https://developer.nvidia.com/DALI

Detectors	FP32	AMP	AMP + DALI
RetinaNet	1x	1.12x	2.1x
SSD	1x	1.01x	2x

MATH OPERATIONS

Mixed precision performance depends on layer composition

Not all the time is spent on the GPU

- framework overheads, CPU work, communication ...

Memory-bound layers

- can give up to 2x speedup - FP16 saves memory traffic
- e.g. losses, activations, normalizations, pointwise

Math-bound layers

- can get up to **8x** with Tensor Cores
- e.g. linear, matmul, batched gemms, convolutions

ANALYZING MATH OPERATIONS

Find model time spent on math-bound operations

Get most speedup from mixed precision

Need to profile *individual layers* of the network

Recommended tools

1. For PyTorch: PyTorch Profiling tool (PyProf)

   https://github.com/NVIDIA/apex/tree/master/apex/pyprof

2. For Tensorflow: Deep Learning Profiler (DLProf)

   https://docs.nvidia.com/deeplearning/frameworks/dlprof-user-guide/index.html

PYTORCH PROFILING TOOL

Correlates GPU kernels to network layers and operations

Provides layer-resolved breakdown of GPU time

Captures PyTorch API/layer name, tensor dimensions/precision, GPU kernel and duration

Determines various issues that limit mixed precision performance, e.g. Tensor Core usage

Tensor Core usage

Op	Params	TC	Time (ns)
add	T=(128,64,56,56),fp16	-	96545
conv2d	N=128,C=64,H=56,W=56,K=256,fp16	1	1600020
relu	T=(128,64,56,56),fp16	-	381028

Key idea is to provide the raw data and leave the user to organize, e.g. using excel

PYTORCH PROFILING TOOL

As simple as four steps

Add changes to training scripts

Initializes the library

```
from apex import pyprof
pyprof.nvtx.init()
```

Runs the training/inference loop with the PyTorch NVTX context manager

```
with torch.autograd.profiler.emit_nvtx():
    for iter in range(iters):
        output = net(data)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()
```

PYTORCH PROFILING TOOL

As simple as four steps

Run nvprof to generate SQL

```
nvprof -f -o net.sql python train.py
```

Parse the SQL to generate dictionaries containing information about each layer

```
python -m apex.pyprof.parse net.sql > net.dict
```

Run profiler to produce csv or columnated output

```
python -m apex.pyprof.prof --csv net.dict
```

DEEP LEARNING PROFILER

New profiling tool for data scientists and deep learning researchers

Meant to help understand and visualize performance of DNN training

- correlates with model layers
- shows which operations ran for how long

Can analyze output

- visually on TensorBoard
- text reports in csv/json

DEEP LEARNING PROFILER

A text report example

1. Time breakdown between GPU/CPU
2. Top N operations that were consuming the most time
3. Operations that are *using* or are *eligible to use* Tensor Cores

Model Summary

Total Wall Time: 29 s
Number of Found Iterations: 61

#1

Time Summary For All Iterations

	GPU Time	#Nodes
All Nodes	2.62 s	6734
Nodes Using TC	697 ms	95
Nodes Eligible For TC, But Not Using	33.6 ms	15
All Other Nodes	1.89 s	6624

TC stands for "Tensor Cores"
GPU Time is the cumulative time executing GPU kernels

Time Summary For All Kernels

	GPU Time	#Kernels
All Kernels	2.62 s	4454
Kernels Using TC	55.3 ms	95
All Other Kernels	2.57 s	4359

TC stands for "Tensor Cores"
GPU Time is the cumulative time executing GPU kernels

#3

Sort top 10 nodes by: GPU CPU

#2

GPU Time (µs)	CPU Time (µs)	Op Name	Op Type	Origin	Calls	TC Eligible	Using TC
2673185	80322	gradients/resnet50_v1.5/conv2d/conv2d/Conv2D_grad/Conv2DBackpropFilter	Conv2DBackpropFilter	GraphDef	103	true	true
868941	2723439	resnet50_v1.5/conv2d/conv2d/Conv2D	Conv2D	GraphDef	103	true	true
667262	9893	gradients/resnet50_v1.5/max_pooling2d/MaxPool_grad/MaxPoolGrad	MaxPoolGrad	GraphDef	103	false	false
294843	24971	gradients/resnet50_v1.5/btlnck_block_2_1/shortcut/conv2d/conv2d/Conv2D_grac	Conv2DBackpropInput	GraphDef	103	true	false
278675	12356	resnet50_v1.5/btlnck_block_2_1/shortcut/conv2d/conv2d/Conv2D	Conv2D	GraphDef	103	true	true
258061	49760	gradients/resnet50_v1.5/btlnck_block_2_1/bottleneck_1/conv2d/Conv2D_grad/Cc	Conv2DBackpropFilter	GraphDef	103	true	false
246756	7310	gradients/resnet50_v1.5/conv2d/BatchNorm/FusedBatchNormV2_grad/FusedBat	FusedBatchNormGradV2	GraphDef	103	false	false
245267	26611	gradients/resnet50_v1.5/btlnck_block_2_1/bottleneck_2/conv2d/Conv2D_grad/Cc	Conv2DBackpropInput	GraphDef	103	true	true
239516	60398	gradients/resnet50_v1.5/btlnck_block_2_1/shortcut/conv2d/conv2d/Conv2D_grac	Conv2DBackpropFilter	GraphDef	103	true	true

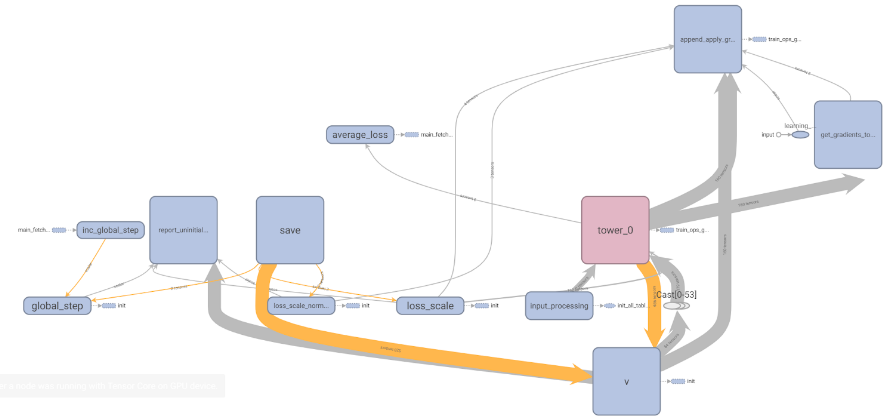
NVIDIA.

DEEP LEARNING PROFILER

A visual example on TensorBoard

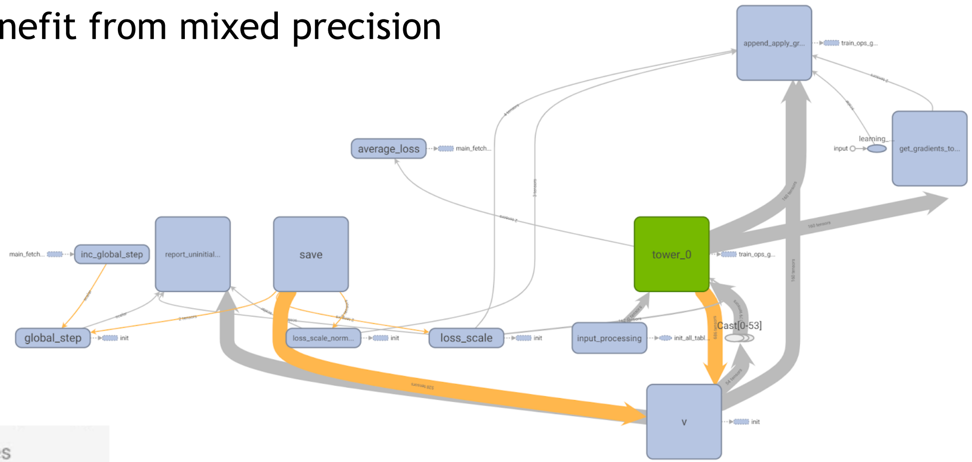
Visual cues on operations that are eligible to use (in red) or are using (in green) Tensor Cores

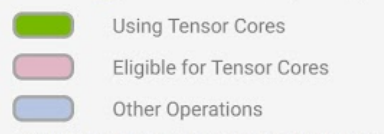
FP32



AMP

pinpoint operations that can benefit from mixed precision





DEEP LEARNING PROFILER

As simple as three steps

Obtain NVIDIA TensorFlow container (integration to official TF coming soon)

```
docker pull nvcr.io/nvidia/tensorflow:19.08-py3
```

Profile using a single command line

```
dlprof --out_detail_report_csv=report.csv python train.py
```

Analyze text report or import to TensorBoard

```
tensorboard --logdir ./events_file
```

ANALYZING MATH OPERATIONS

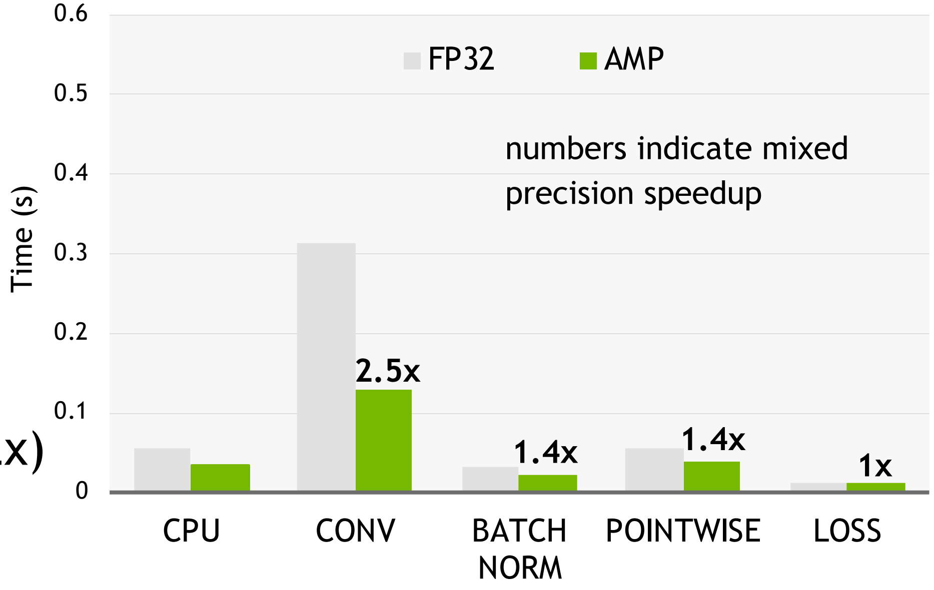
Going back to our profiling example

RetinaNet w/ RN50 backbone training on DGX-1, batch size 32 p/ GPU

Run for a few iterations using PyProf/DLProf

Observations:

- 90% of model time spent on GPU
- 2.5x from convolutions
- *good* end-to-end mixed precision performance (2x) because of speedup from convolutions



GETTING THE MOST FROM MATH OPERATIONS

Spend more time on Tensor Cores

Reduce time spent on operations that are not math bound

1. Can speed up ops by hand

- custom CUDA kernel + framework integration

2. Can fuse ops via DL framework compiler tools

- merge a set of small ops into a bigger operation
- TensorFlow XLA
- PyTorch JIT

```python
@torch.jit.script
def foo(x, y):
    s = torch.softmax(y)
    return x + s
z = foo(x, y)
```

fused into
a single op

GETTING THE MOST FROM MATH OPERATIONS

Increasing the batch size

FP16 activations and gradients can reduce memory consumption

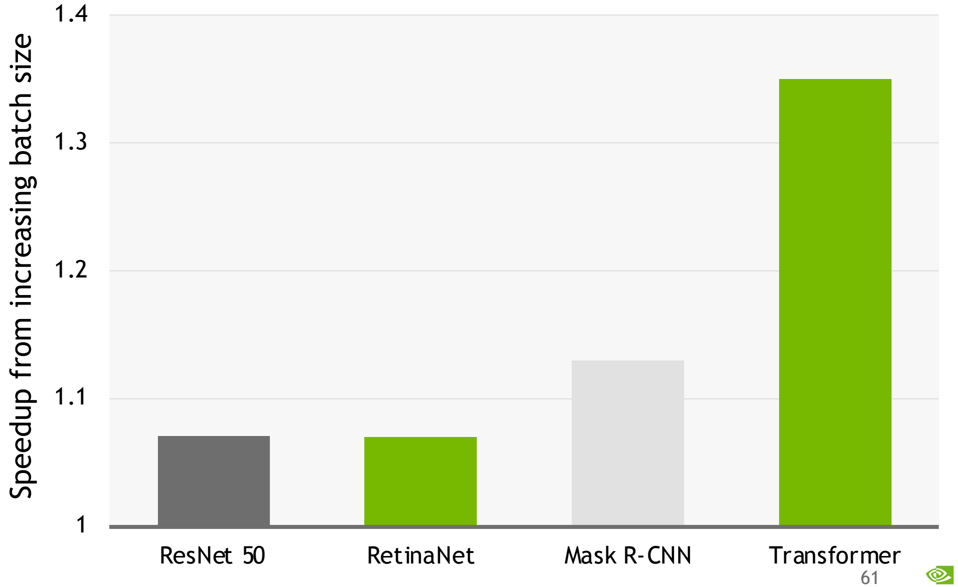
Allows for larger batch sizes

Reduces cost for forward / backward pass

Query GPU memory usage with `nvidia-smi`

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.87.01    Driver Version: 418.87.01    CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  On   | 00000000:06:00.0 Off |                    0 |
| N/A   60C    P0   300W / 300W |  14180MiB / 32480MiB |     96%      Default |
+-------------------------------+----------------------+----------------------+
```

Benchmarked on a single V100



TENSOR CORES

Make sure that Tensor Cores are being used

There are *several ways* to check Tensor Core usage

PyProf/DLProf, see slides 56 & 60

Kernel	Op	TC
volta_fp16_s884cudnn	conv2d	1
elementwise_kernel	relu	-

Sort top 10 nodes by: GPU CPU

GPU Time (µs)	CPU Time (µs)	Op Name	Op Type	Origin	Calls	TC Eligible	Using TC
2673185	80322	gradients/resnet50_v1.5/conv2d/conv2d/Conv2D_grad/Conv2DBackpropFilter	Conv2DBackpropFilter	GraphDef	103	true	true

NVIDIA Nsight Compute, next gen profiler for CUDA applications

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

Kernel Name	Metric Name	Metric Unit	Metric Vaue
volta_fp16_s884cudnn	sm__pipe_tensor_cycles_active.avg...	%	86.35
elementwise_kernel	sm__pipe_tensor_cycles_active.avg...	%	0

IMPROVING TENSOR CORE PERFORMANCE

Performance guidelines

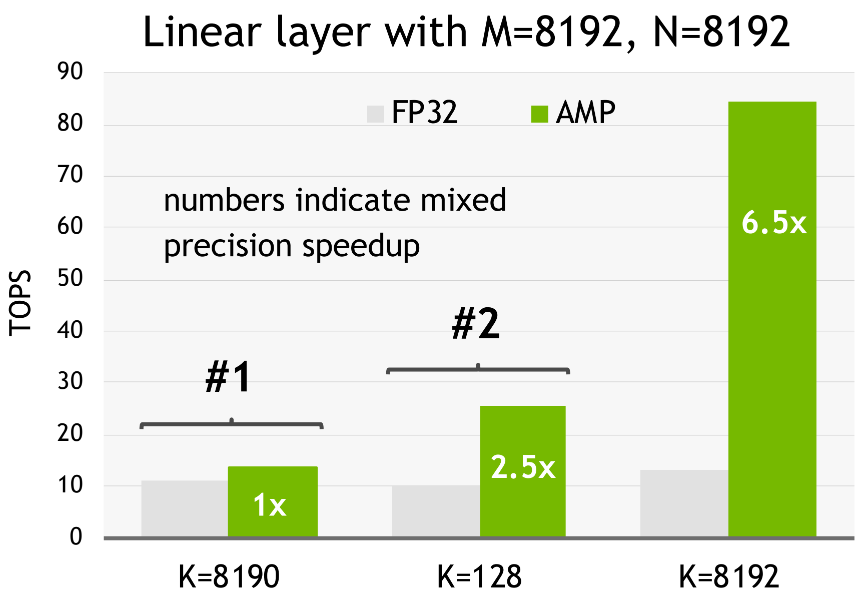
#1 Satisfy shape constraints to enable Tensor Cores

- 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

#2 Ensure Tensor Cores are doing enough math

- If any GEMM dimension is 128 or smaller, speedup is closer to 2x than 8x

Determine #1 and #2 with PyProf/DLProf



IMPROVING TENSOR CORE PERFORMANCE

Practical recommendations

1. Convert all dimensions to be multiples of 8

- mini batch, layer dimensions, pad vocabulary, pad sequence length

2. In model *implementation*

- concatenate matrix multiplies that share an input
- e.g. query/key/value projection matrices in transformers

3. In model *architecture*

- prefer dense math (vanilla convolutions vs depth separable)
- prefer wider layers – often little speed cost

SUMMARY: PERFORMANCE GUIDELINES

Following a few simple guidelines can maximize mixed precision performance:

1. Optimize input data pipeline
2. Make sure most of the time is spent on math-bound layers
3. Improve Tensor Core utilization with good parameter choices

Profile with python timers and recommended tools (PyProf/DLProf)

Visit the Deep Learning Performance Guide for more performance tips

https://docs.nvidia.com/deeplearning/sdk/dl-performance-guide/index.html

CONCLUSION

CONCLUSION

Mixed precision training is a general-purpose technique with enormous benefits

Mixed precision enables *faster* training and *larger* models, mini batches, or inputs

Matches FP32 training accuracy across a *wide* range of tasks/domains/architectures/optimizers

Frameworks have *automated* the mixed precision methodology (AMP)

- model conversion, master weights, and automatic loss scaling

Maximize mixed precision speedups with performance guidelines

RESOURCES

RESOURCES

Paper

- *Micikevicius et al. Mixed Precision Training. ICLR, 2018.*

Talks

- GTC 2018: Mixed Precision Training: Theory and Practice
- GTC 2019: Mixed Precision Training of Deep Neural Networks
- GTC 2019: Automatic Mixed Precision in PyTorch
- GTC 2019: Tensor Core DL Performance Guide

RESOURCES

Guide

- Training with Mixed Precision User Guide

Blogs

- Mixed-Precision Training of Deep Neural Networks
- Mixed-Precision Training Techniques Using Tensor Cores for Deep Learning

THANK YOU!

MODELS TRAINED IN MIXED PRECISION

IMAGE CLASSIFICATION

Top-1 Accuracy

	FP32	Mixed Precision
AlexNet	56.8%	56.9%
VGG-D	65.4%	65.4%
GoogleNet	69.3%	69.3%
Inception v2	70.0%	70.0%
Inception v3	77.3%	77.3%
SqueezeNet	61.0%	61.1%
MobileNet v2	71.6%	71.6%
ResNet 50	76.7%	76.7%
ResNeXt 50	77.6%	77.6%
DenseNet 161	78.3%	78.4%

A number of these train fine in mixed precision without loss scaling.

OBJECT DETECTION

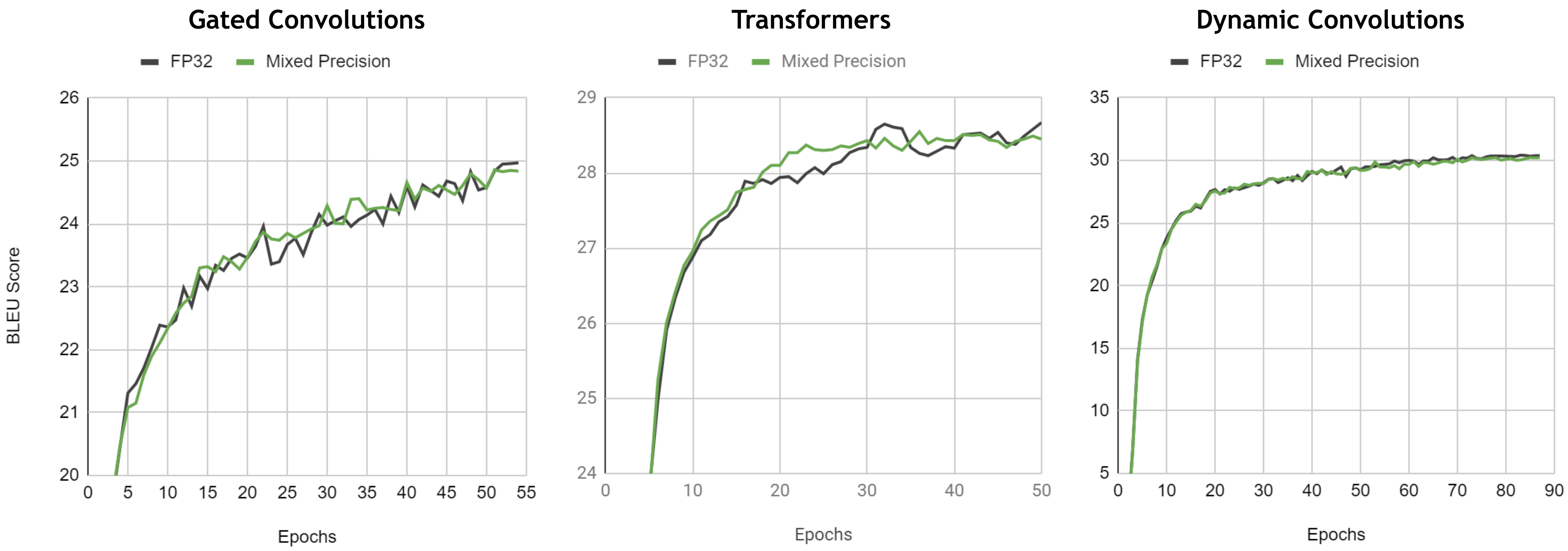
Mean Average Precision (MAP)

	Data Set	FP32	Mixed Precision
Faster R-CNN	VOC 07	69.1%	69.7%
MultiBox SSD	VOC 07+12	76.9%	77.1%
MultiBox SSD	COCO 17	24.8%	24.8%
RetinaNet	COCO 17	34.7%	34.8%
Mask R-CNN	COCO 17	37.8%	37.8%

NVIDIA propriertary automative networks train with mixed precision matching FP32 baseline accuracy.

LANGUAGE TRANSLATION (EN-DE)

Bilingual Evaluation Understudy (BLEU)



SPEECH

Character Error Rate (CER)

Courtesy of Baidu

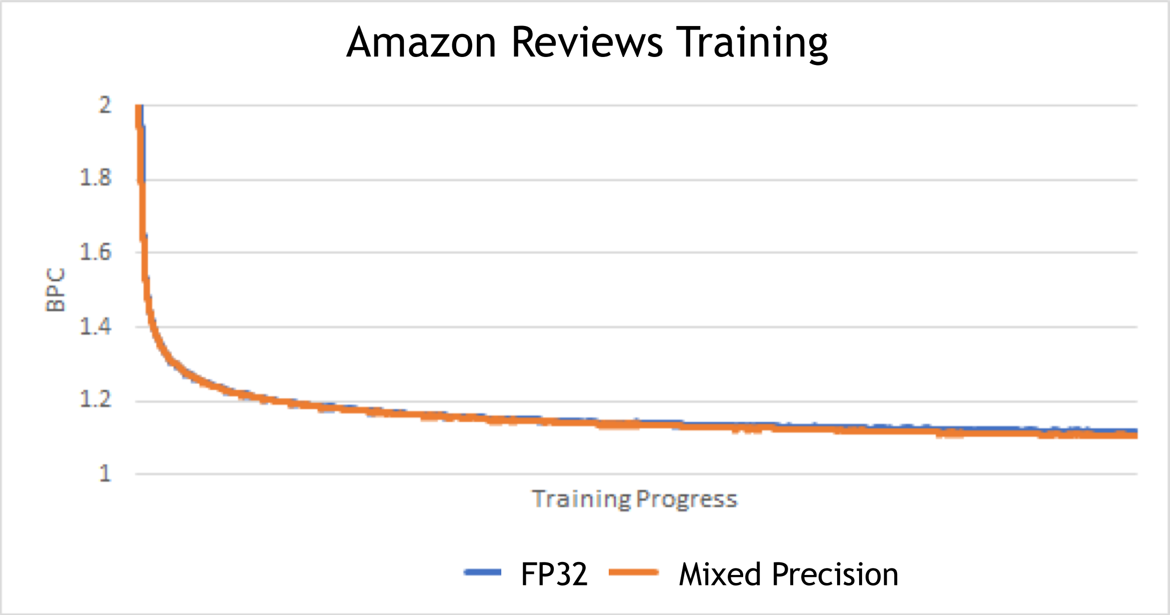
- Two 2D convolution layers + 3 GRU layers + 1D convolution
- Baidu internal datasets

	FP32	Mixed Precision
English	2.20	1.99
Mandarin	15.82	15.01

Lower is better

LANGUAGE MODELING

Bits Per Character (BPC)



	Train BPC	Val BPC	SST Acc.	IMDB Acc.
FP32	1.116	1.073	91.8%	92.8%
Mixed Precision	1.115	1.075	91.9%	92.8%

NVIDIA