VOLTA TENSORコアで、高速かつ高精度にDLモデルをトレーニングする方法

成瀬 彰, シニアデベロッパーテクノロジーエンジニア, 2017/12/12

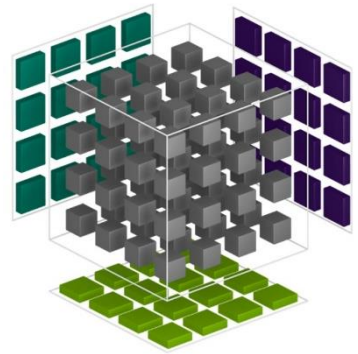
NVIDIA

アジェンダ

- Tensorコアとトレーニングの概要
- 混合精度(Tensorコア)で、FP32と同等の精度を得る方法
  - ウェイトをFP16とFP32を併用して更新する
  - ロス・スケーリング
  - DLフレームワーク対応状況
- ウェイトをFP16で更新する

VOLTA TENSORコア

4x4の行列の乗算を1サイクルで実行



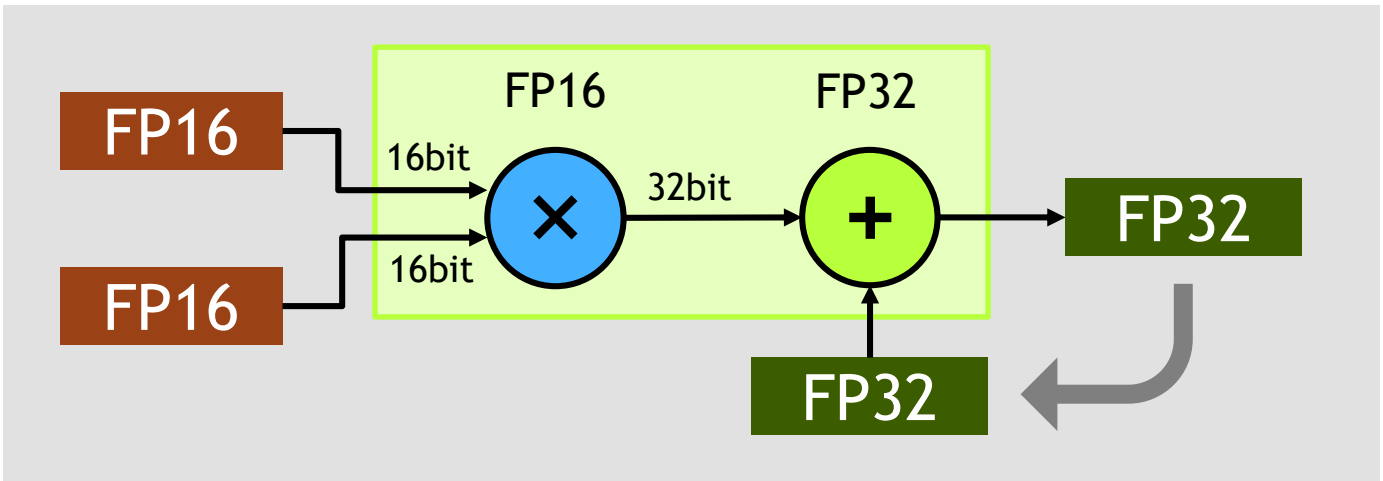
$$D = \underbrace{\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}}_{\text{FP16}} \underbrace{\begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0}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

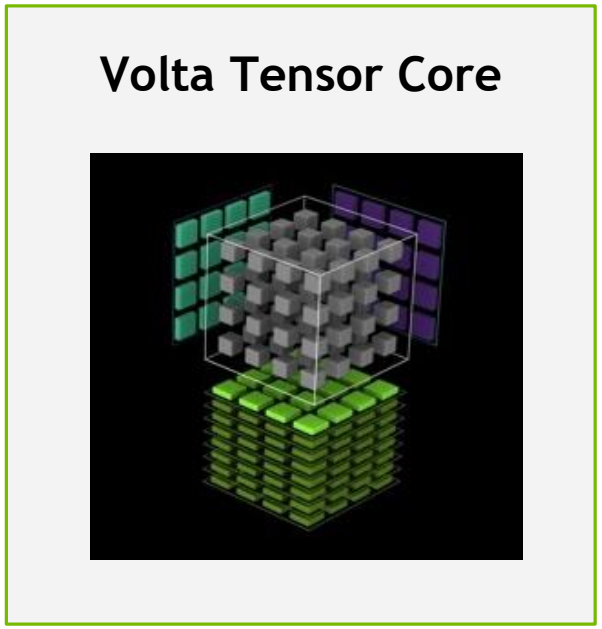
FP16 or FP32

$$D = AB + C$$

VOLTA TENSORコア

混合精度演算

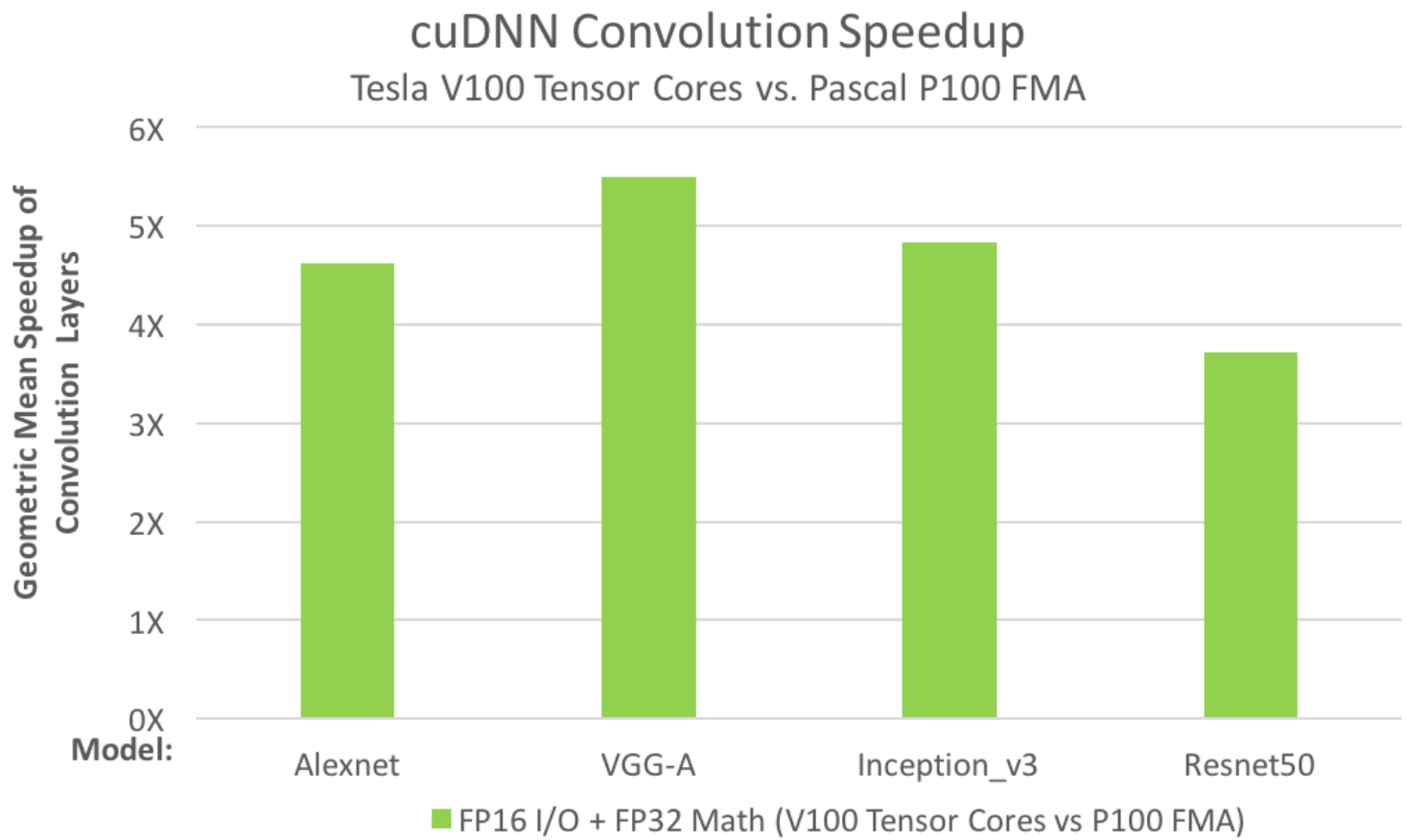




	P100	V100
FP16/Tensorコア	20 TFLOPS	125 TFLOPS
FP32	10 TFLOPS	15.6 TFLOPS

CUDNN: TENSORコアの実効性能

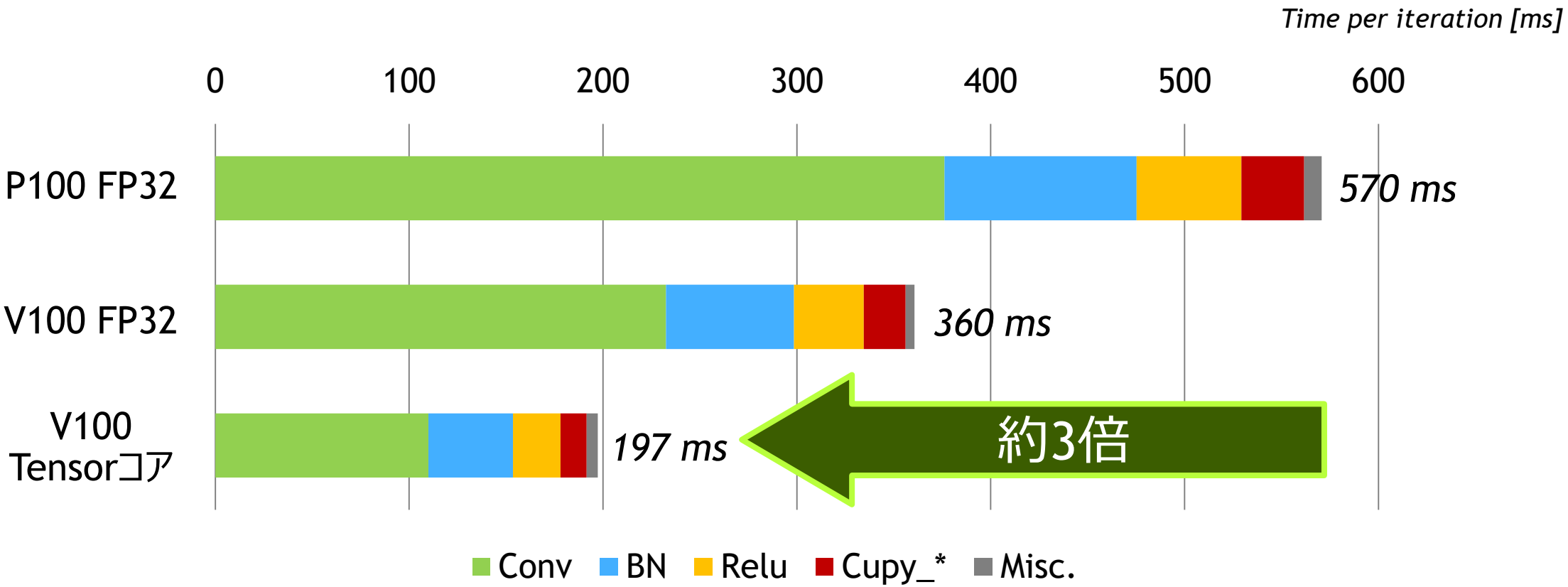
Pascal FP32 vs. V100 Tensorコア



Convolution層の性能比較

Resnet50, Imagenet, Batch:128

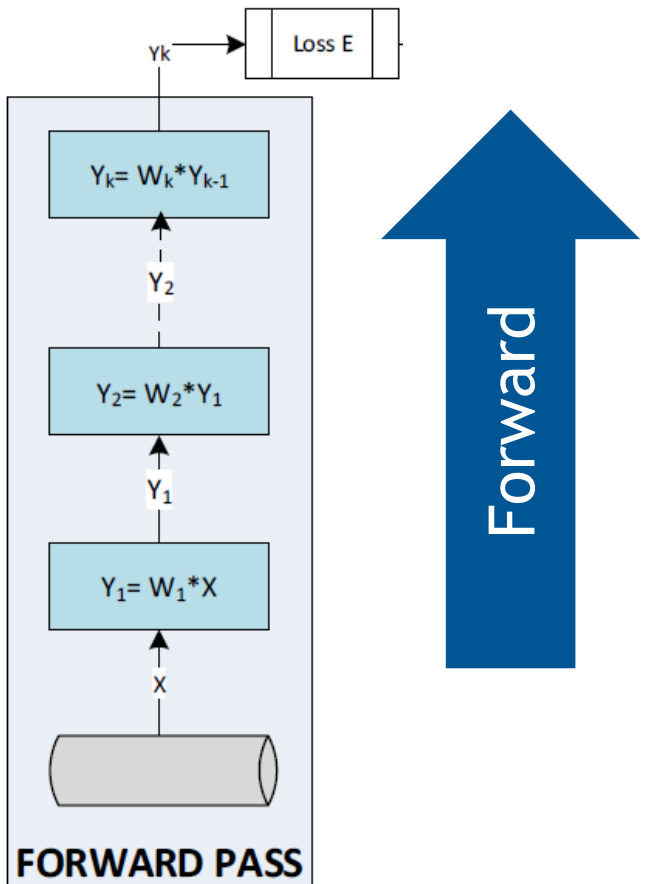
P100 FP32, V100 FP32 vs. V100 Tensorコア



(*) Chainer 3.0.0rc1+ と CuPy 2.0.0rc1+ を使用

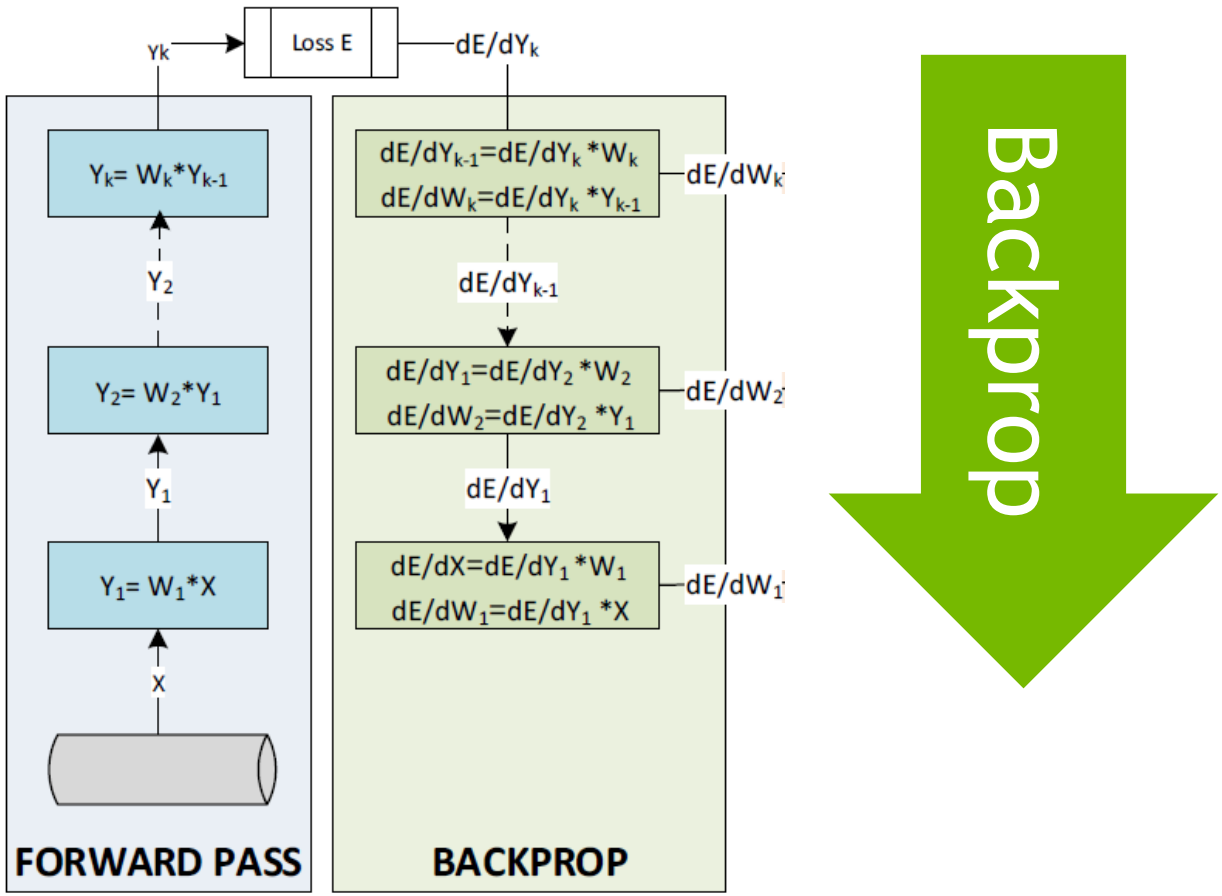
トレーニングの流れ

Forward



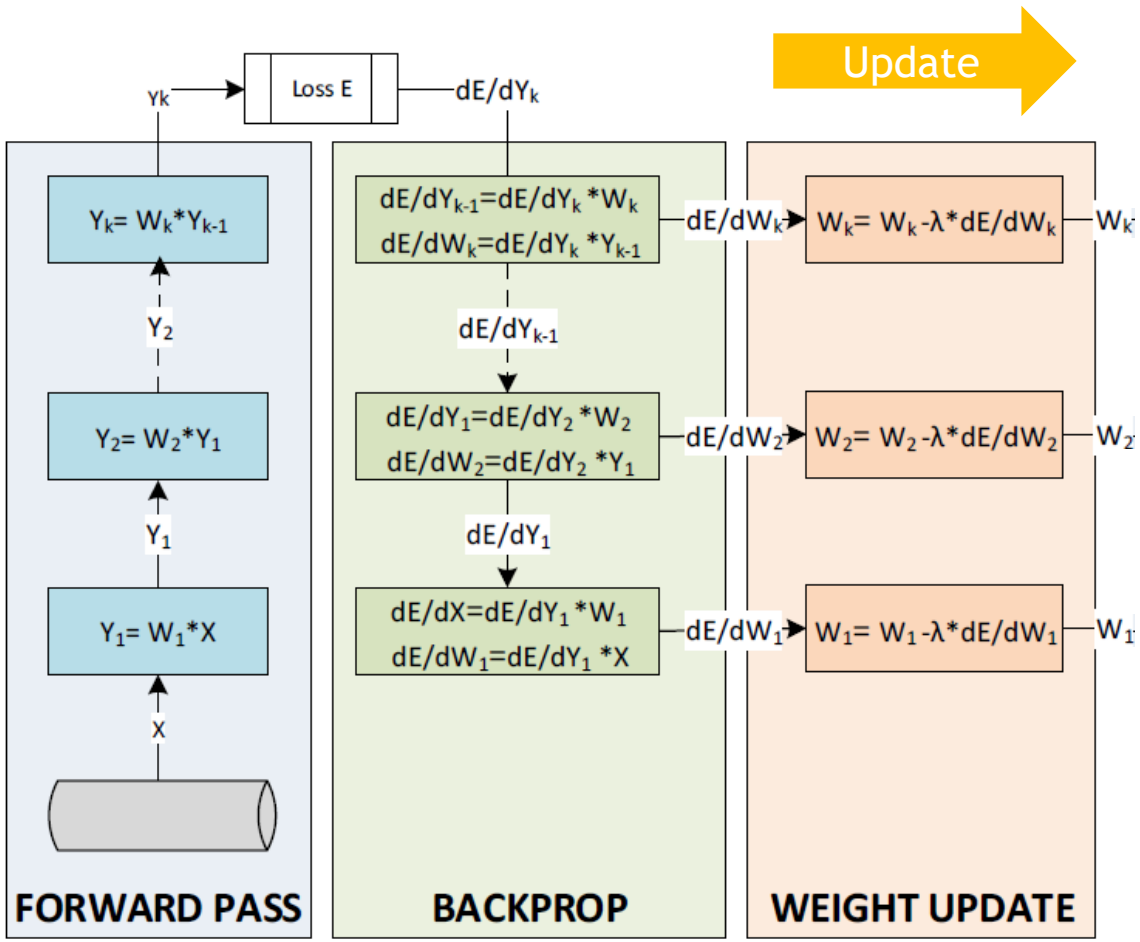
トレーニングの流れ

Backprop



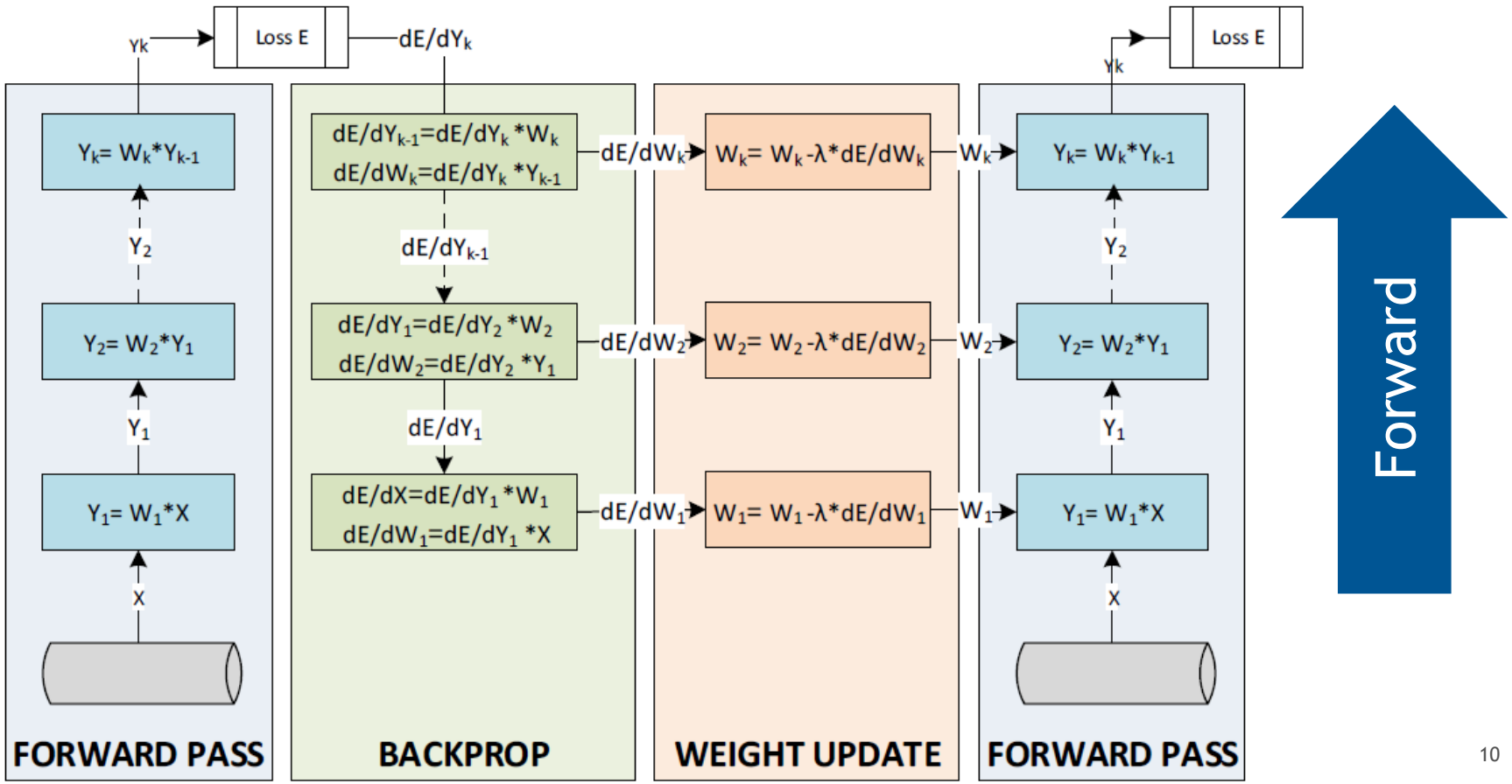
トレーニングの流れ

Update



トレーニングの流れ

Forward

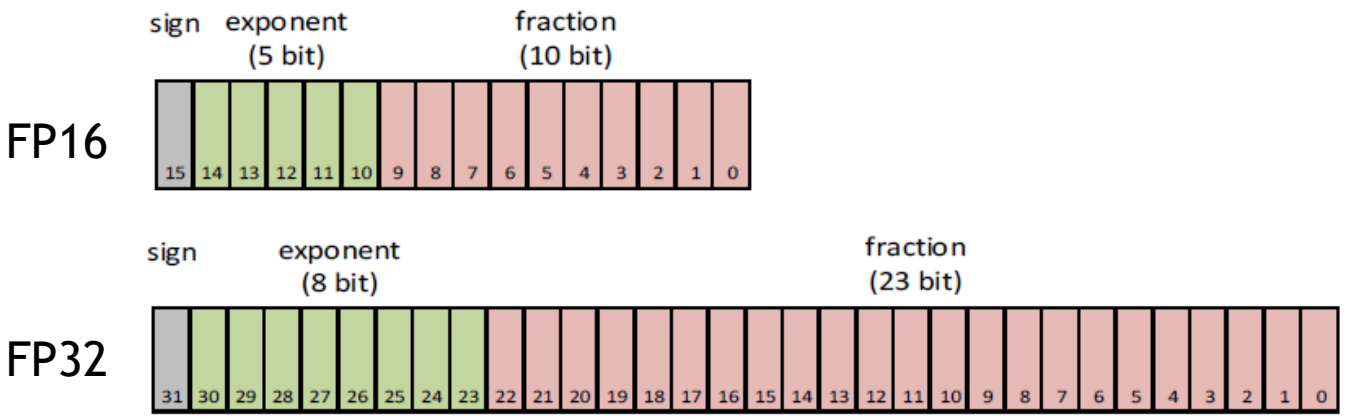


データ型に基づくトレーニングの分類

FP32, FP16, 混合精度

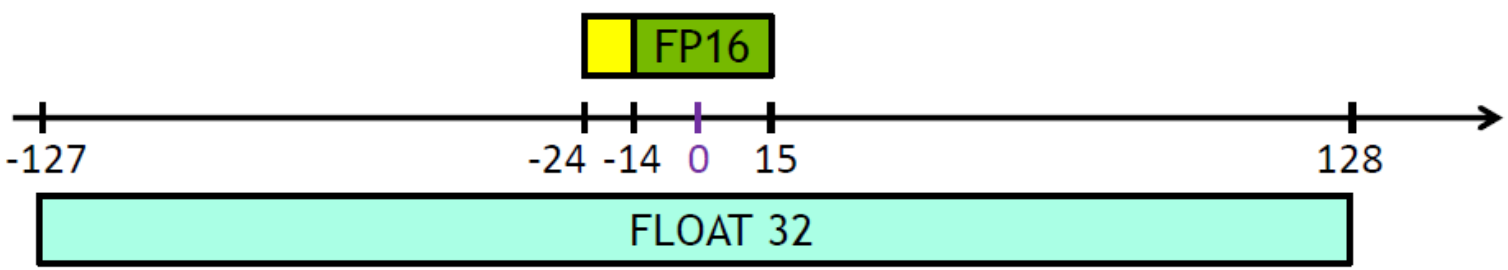
トレーニング	入力データ	行列演算 乗算 (x)	行列演算 加算 (+)	GPU
FP32	FP32	FP32	FP32	○

半精度浮動小数点(FP16)



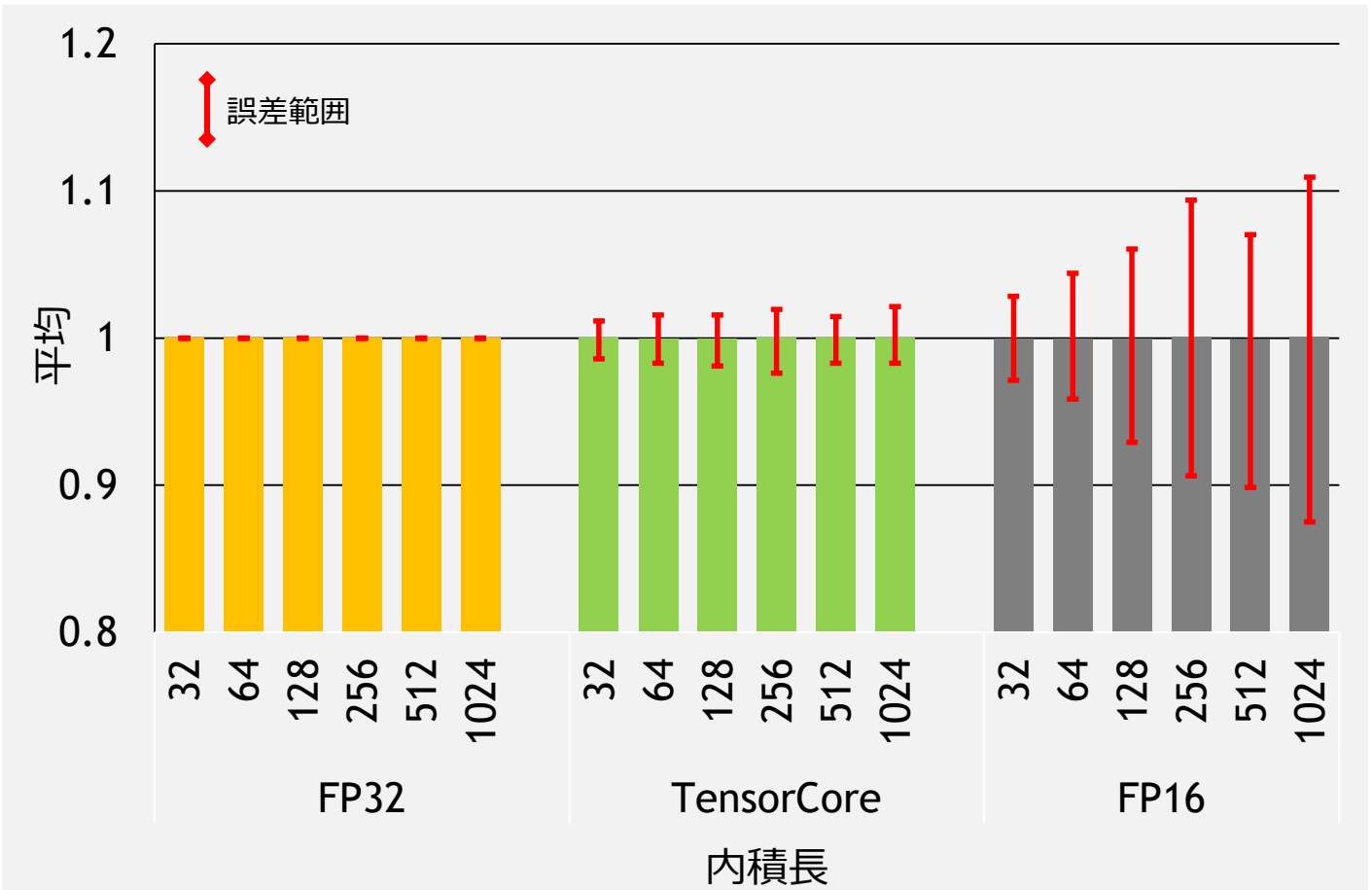
- IEEE754
- 単精度(FP32)と比べると、表現可能レンジが非常に狭い

Normal range: $[\ 6\times10^{-5}\ ,\ 65504\ ]$
Sub-normal range: $[\ 6\times10^{-8}\ ,\ 6\times10^{-5}\ ]$



TENSORコアの計算精度

FP32に近い結果



Tensorコアの演算結果は、FP16と比べて、FP32との誤差が小さい

- 行列A: 指数分布 (activation)
- 行列B: 正規分布 (weight)
  (平均0.0, 分散1.0)
- 内積長: 32 – 1024
- 1万サンプル
- 誤差区間: 99%

混合精度(TENSORコア)でトレーニング

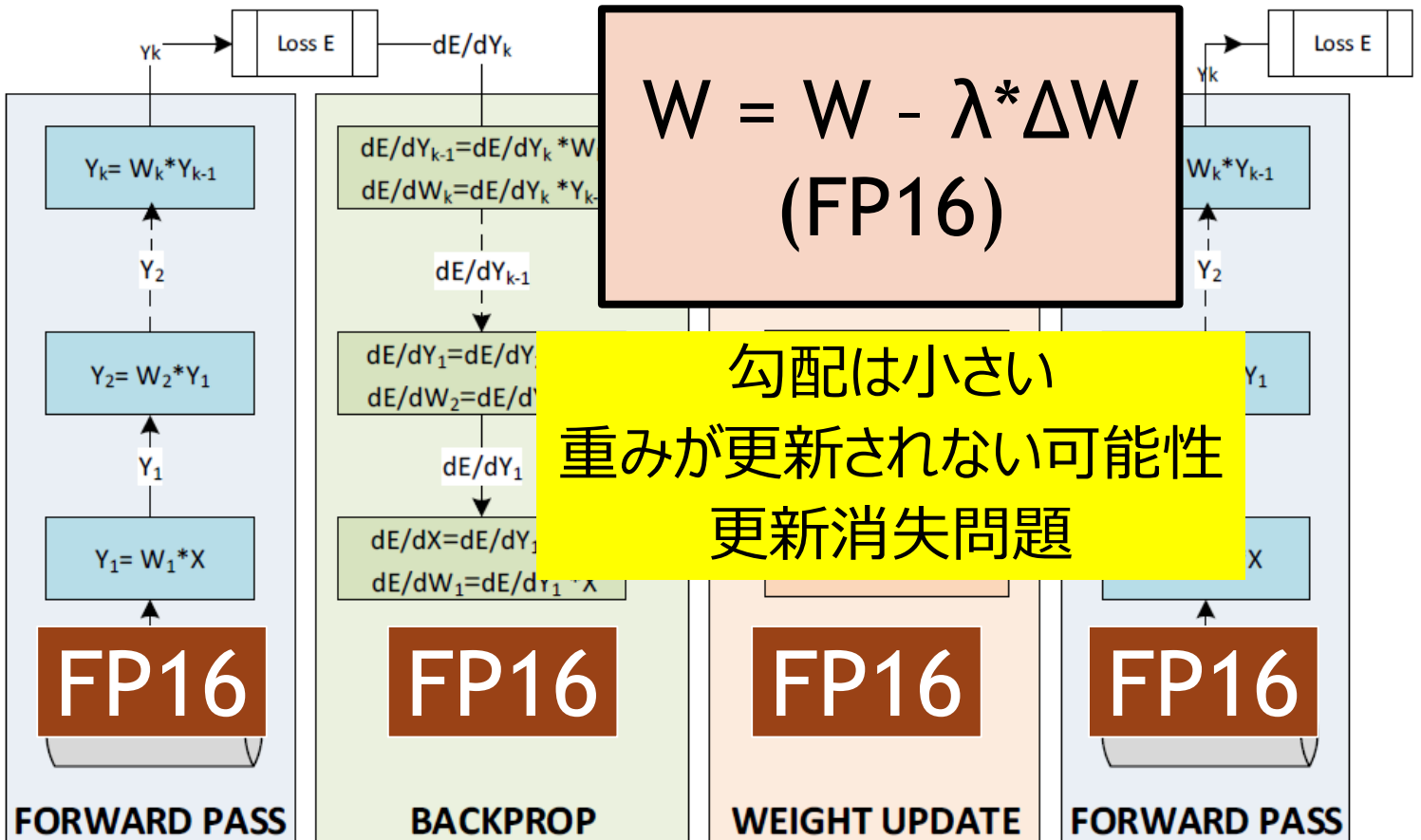
Q: FP32でトレーニングしたモデルと、同じ精度を得られるのか?

A: 可能です、その方法を説明します

ウェイトの更新には、FP16とFP32を併用する

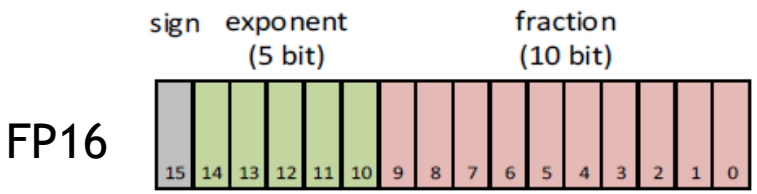
トレーニング (FP16、混合精度)

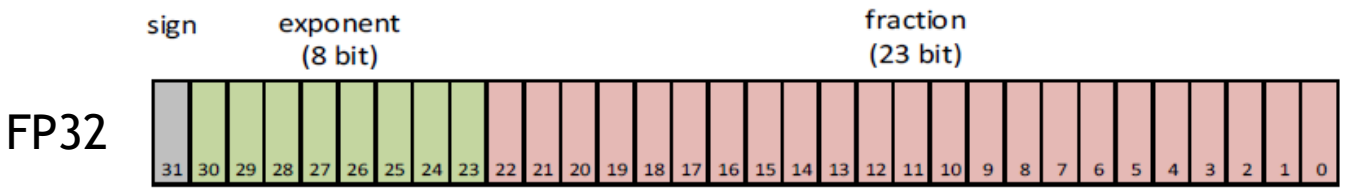
ストレージはFP16



ストレージの
データ型

半精度浮動小数点(FP16)



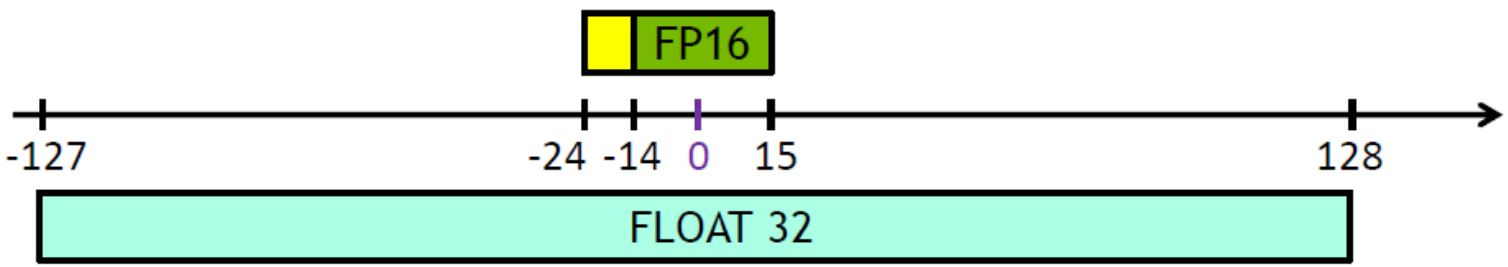


表現可能範囲が狭い

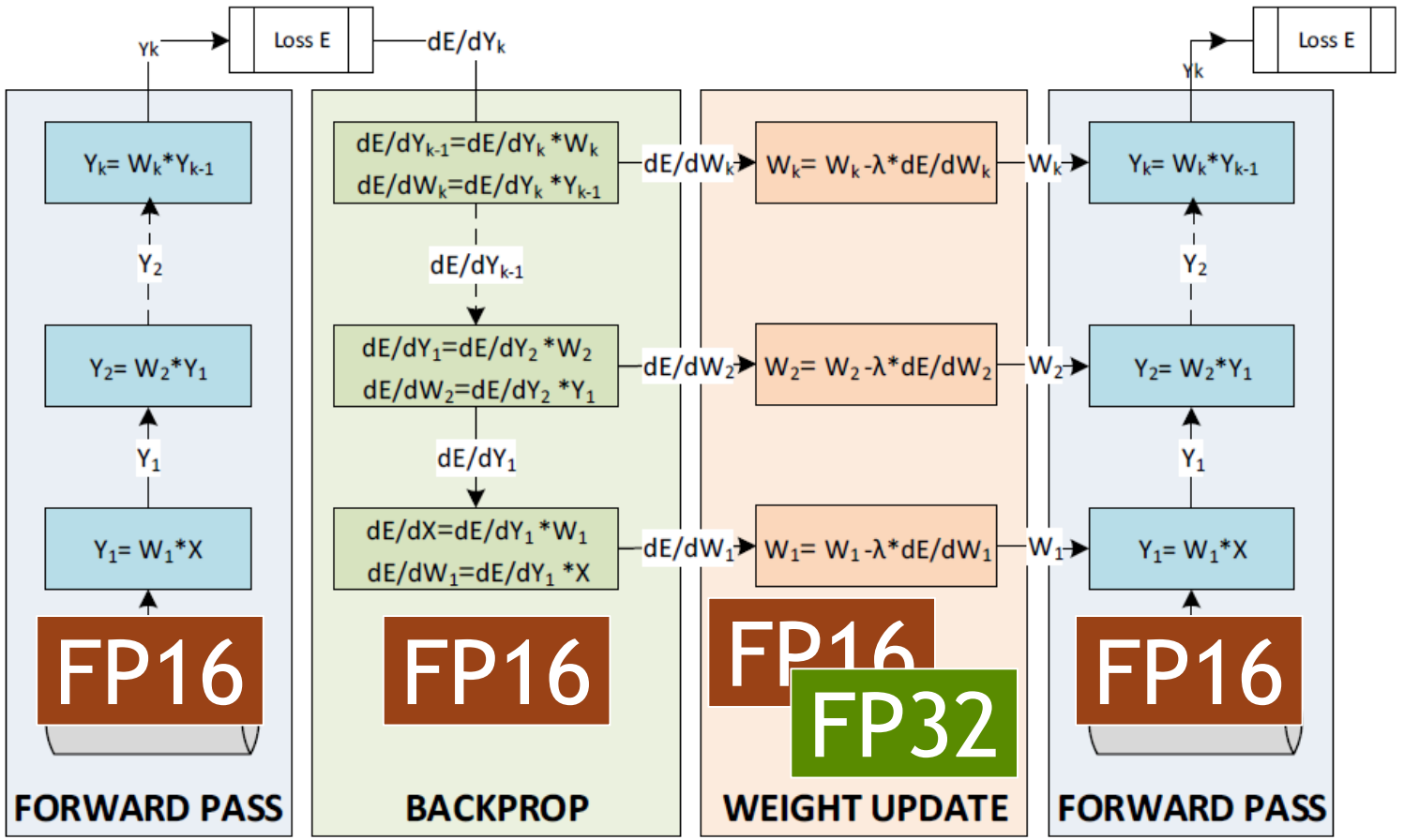
FP16の仮数部は10ビット

2048 + 1 = 2048

Normal range: [6×10^{-5} , 65504]
Sub-normal range: [6×10^{-8} , 6×10^{-5}]

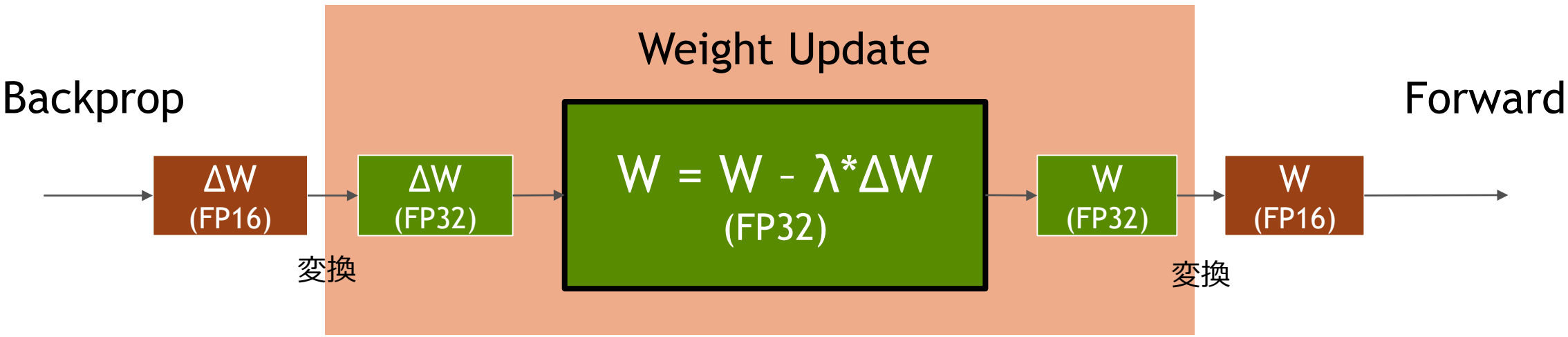


ウェイトはFP32で更新



ストレージの
データ型

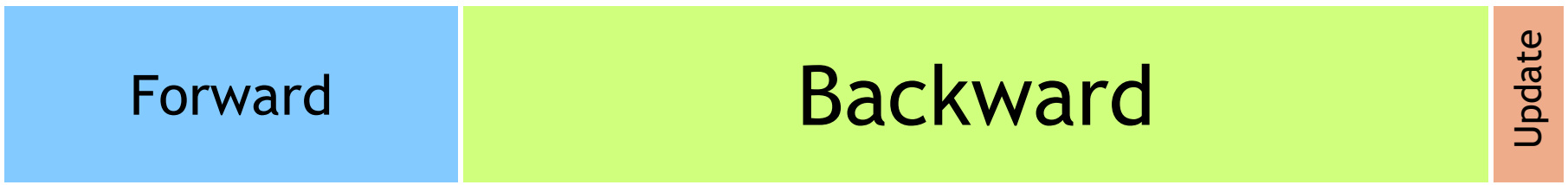
ウェイトはFP32で更新



- Updateは、FP32で計算する
  - FP16の勾配を、FP32に変換
  - FP32のウェイト(マスターコピー)を、FP32で更新
  - FP32のウェイトから、FP16のウェイトを作成

Q: FP32で更新すると遅くならないか?

トレーニングの時間比率



トレーニング時間の大部分は、BackwardとForward

Updateの時間は短い、FP32計算によるスピード低下は僅か

トレーニングの分類

トレーニング	入力データ	行列乗算 乗算 (x)	行列乗算 加算 (+)	ウェイト更新	GPU
FP32	FP32	FP32	FP32	FP32	
FP16	FP16	FP16	FP16	FP16/FP32	Pascal
混合精度	FP16	FP16	FP32	FP16/FP32	Volta

混合精度+ウェイトFP32更新

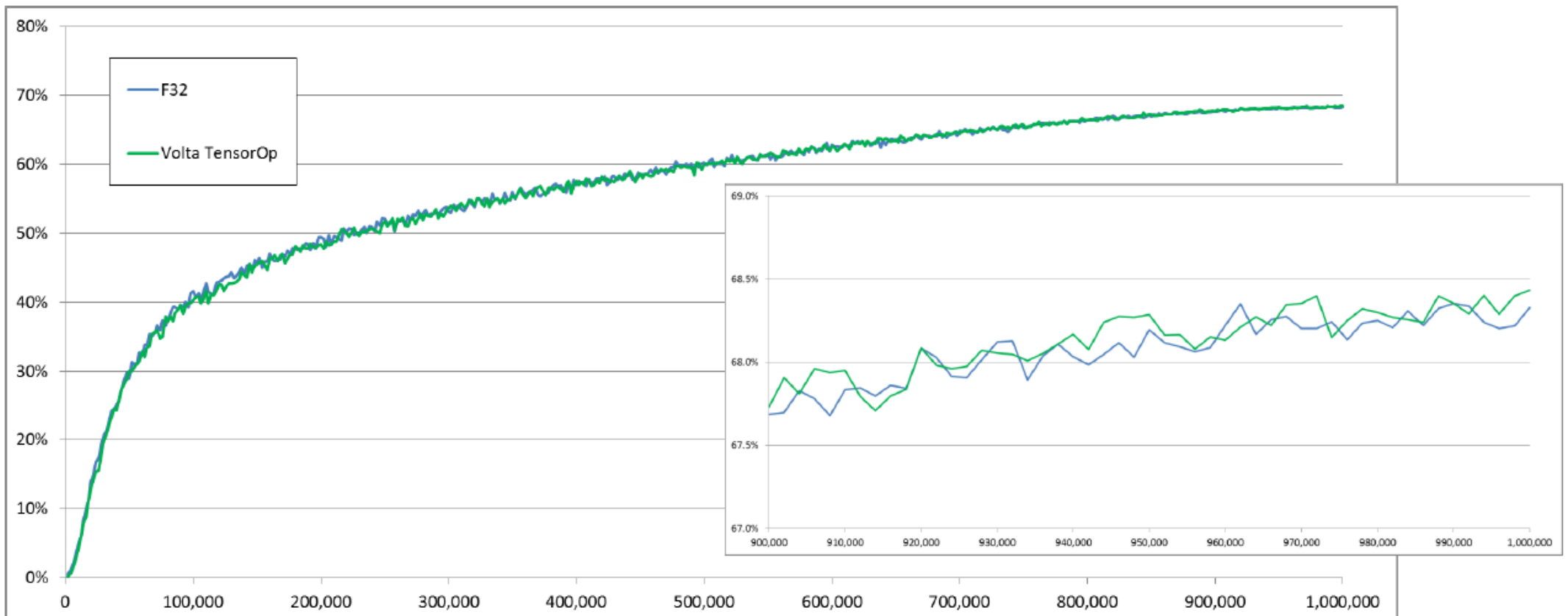
多くのモデルは、これで収束する

- FP32モデルと同等の精度が得られるケースも多い
  - 同じソルバー、同じハイパーパラメータ、同じ学習レートコントロール、...

- 画像分類 (ImageNet)
  - GoogleNet, VGG-D, Inception v3, ResNet-50
  - ソルバー: モメンタムSGD
- 言語モデル、機械翻訳
  - NMT
  - ソルバー: ADAM

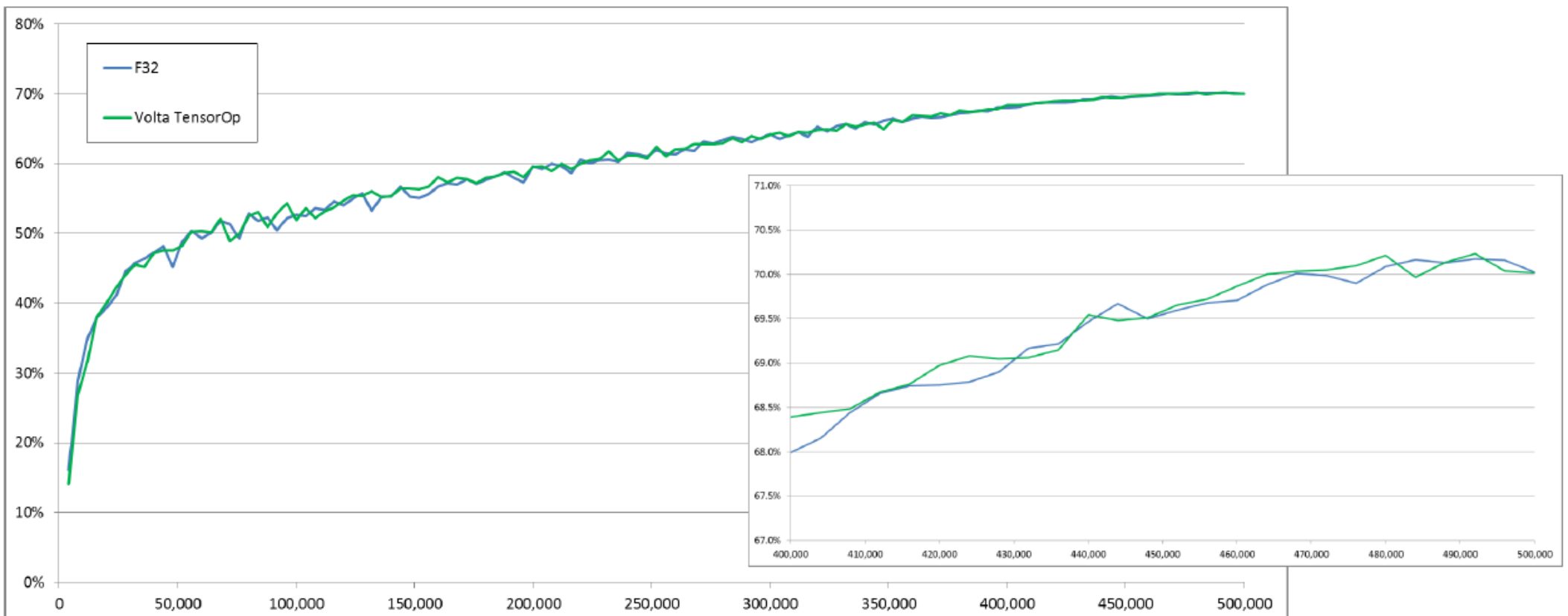
GOOGLENET

FP32の学習カーブと一致



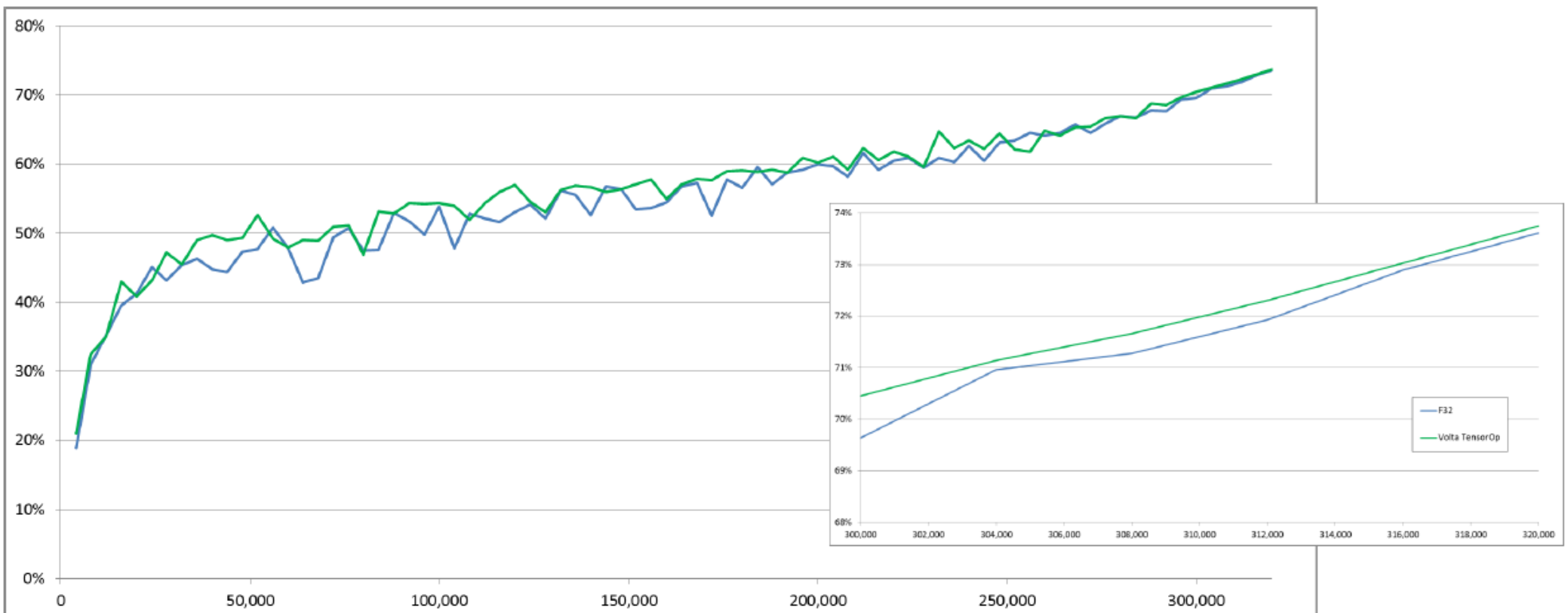
INCEPTION V1

FP32の学習カーブと一致



RESNET-50

FP32の学習カーブと一致



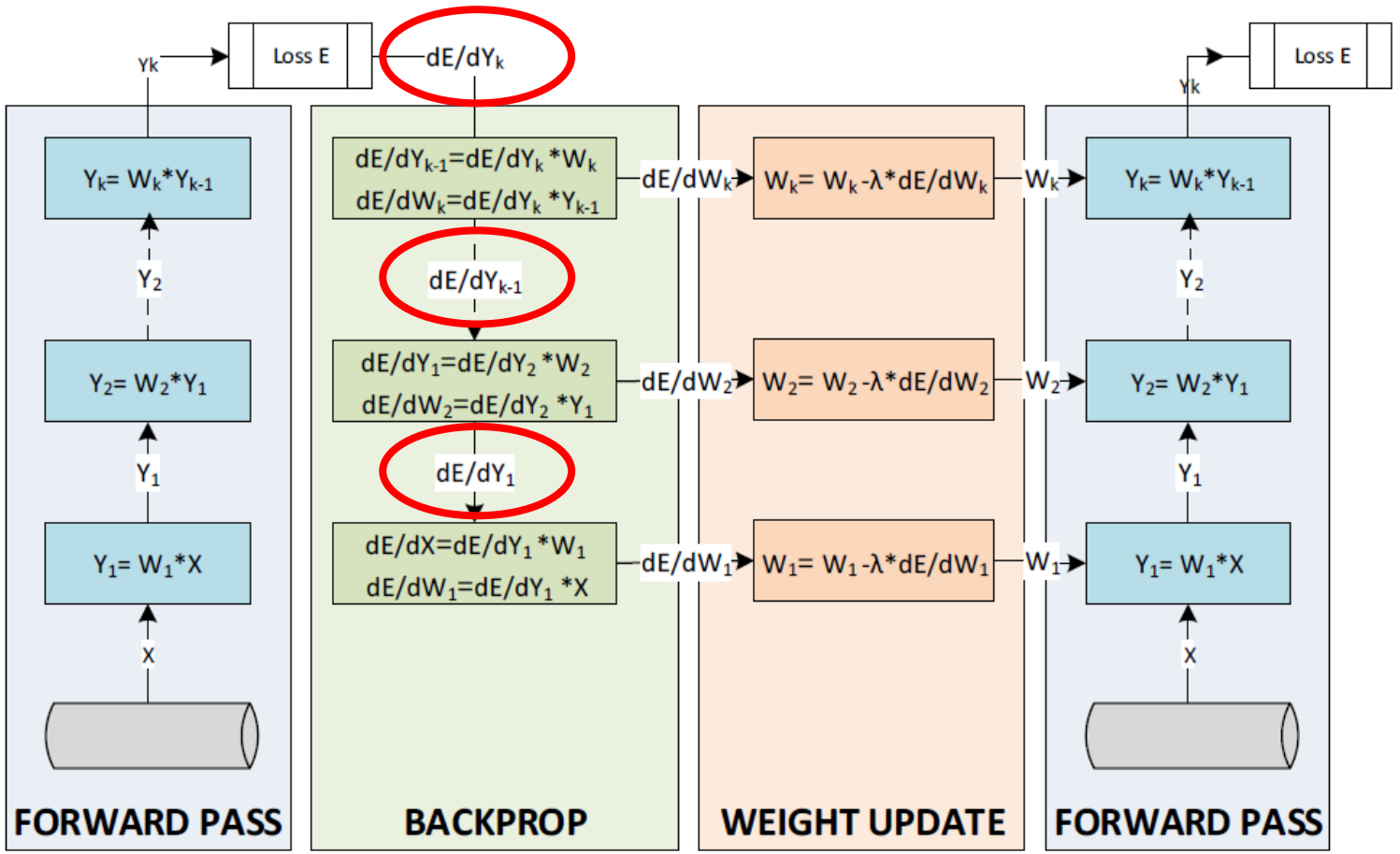
混合精度+ウェイトFP32更新

収束しないケース

- CNN (画像分類)
  - Alexnet, CaffeNet
- CNN (物体検出)
  - Multibox SSD (VGG-D): 学習できず
  - Faster R-CNN (VGG-D): 精度低下　mAP: 69.1% (FP32) → 68.5% (Tensorコア)
- RNN
  - Seq2seq (アテンション付): 収束が遅い
  - bigLSTM: 途中から発散

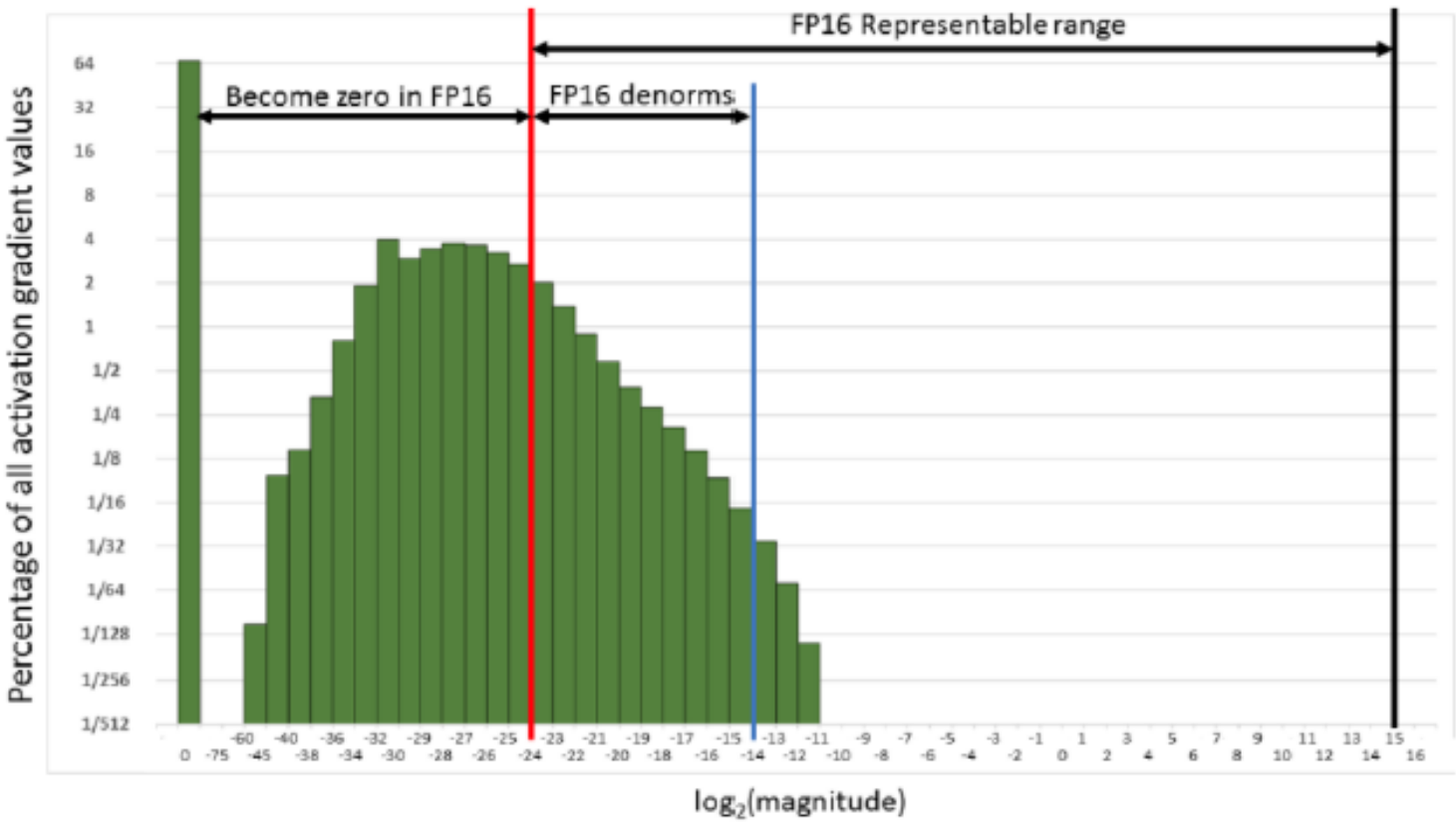
Q: 問題は何？

アクティベーションの勾配



アクティベーションの勾配のヒストグラム

Multibox SSD (VGG-D, FP32)



FP32:

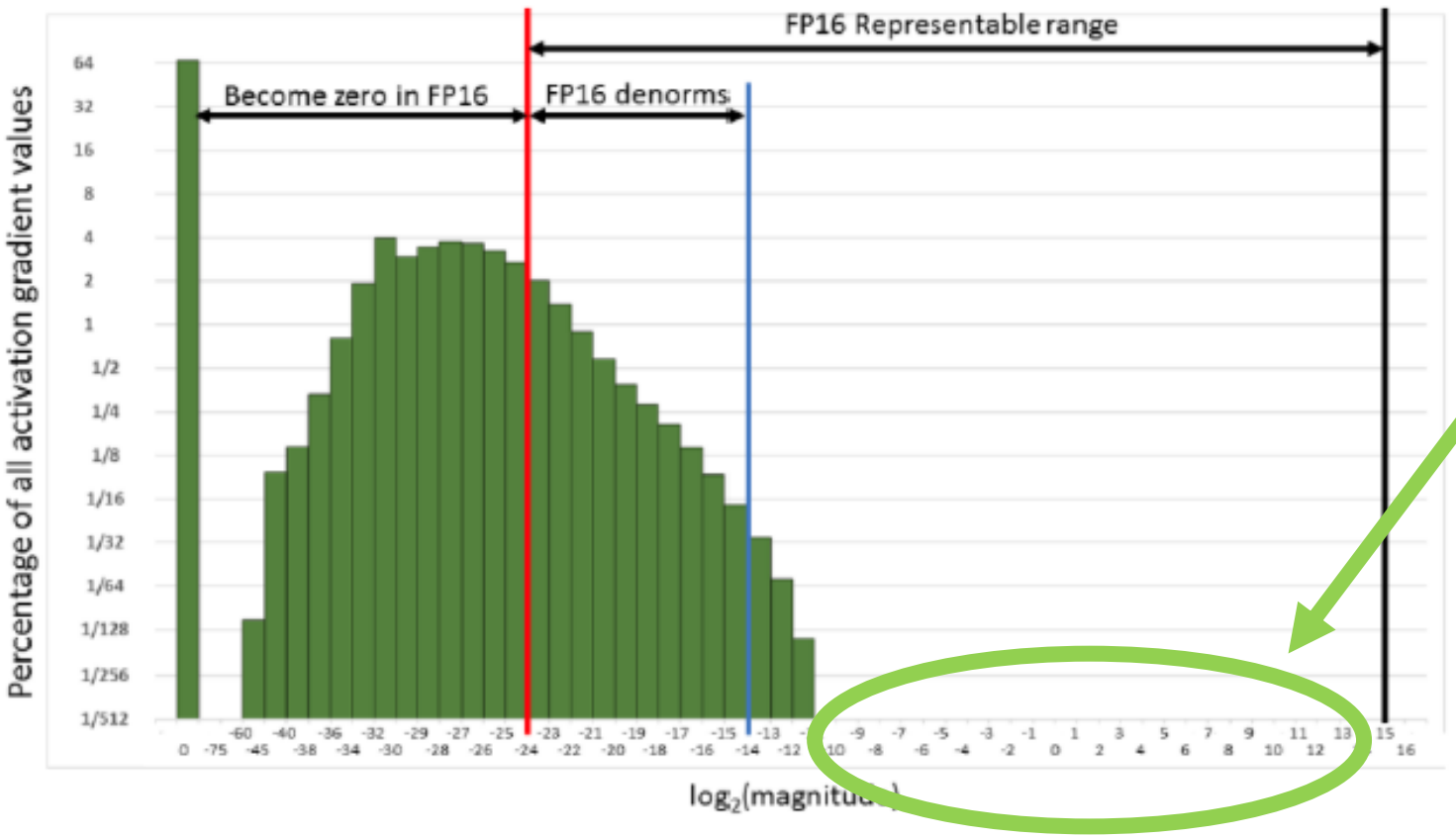
- ゼロ: 67%
- 非ゼロ: 33%

FP16:

- ゼロ: 94%
- 非ゼロ: 6%

アクティベーションの勾配のヒストグラム

Multibox SSD (VGG-D, FP32)



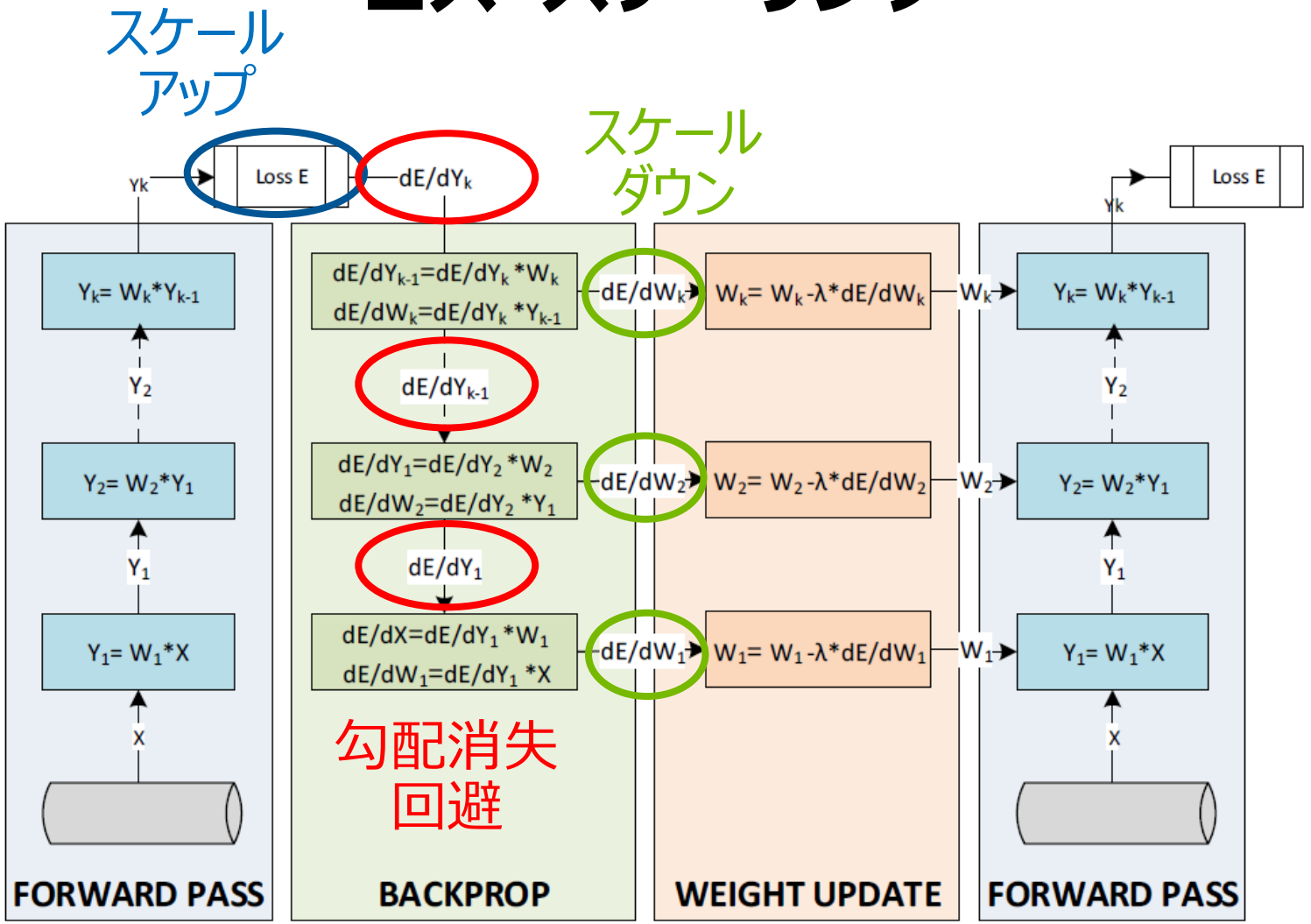
- FP16で表現可能なレンジが、ほとんど使われていない

ロス・スケーリング

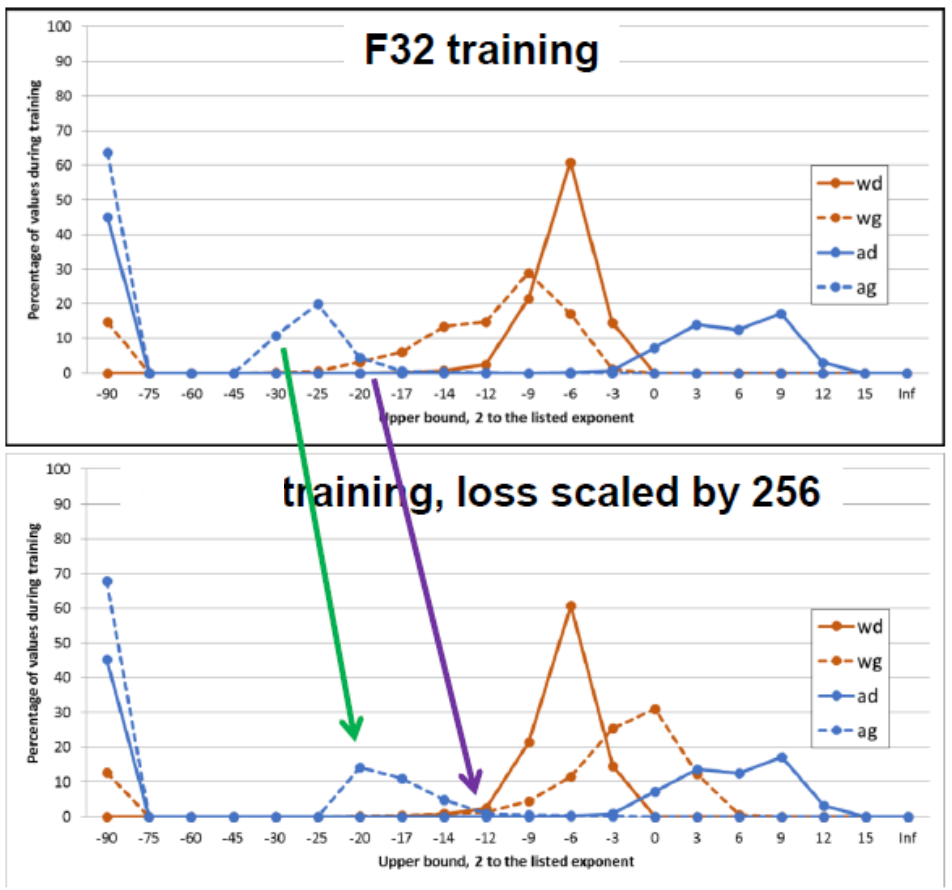
ロス・スケーリング

- 問題: 勾配消失
  - アクティベーションの勾配値は小さい、データ型をFP16にするとゼロになる
- 解決法: ロススケーリング
  - ロスの値をスケールアップ（大きく）してから、Backpropする
  - ウェイト更新の直前に、ウェイトの勾配をスケールダウン（小さく）する
  - スケーリングファクター: 新ハイパーパラメータ？

ロス・スケーリング


スケール
アップ
スケール
ダウン
Loss E
Yk
dE/dYk
Loss E
Yk
Yk= Wk*Yk-1
dE/dYk-1=dE/dYk *Wk
dE/dWk=dE/dYk *Yk-1
dE/dWk
Wk= Wk -λ*dE/dWk
Wk
Yk= Wk*Yk-1
Y2
dE/dYk-1
Y2
Y2= W2*Y1
dE/dY1=dE/dY2 *W2
dE/dW2=dE/dY2 *Y1
dE/dW2
W2= W2 -λ*dE/dW2
W2
Y2= W2*Y1
Y1
dE/dY1
Y1
Y1= W1*X
dE/dX=dE/dY1 *W1
dE/dW1=dE/dY1 *X
dE/dW1
W1= W1 -λ*dE/dW1
W1
Y1= W1*X
X
X
勾配消失
回避
FORWARD PASS
BACKPROP
WEIGHT UPDATE
FORWARD PASS

ロス・スケーリング



- (例) ロスの値を256倍
  - 勾配の値も256倍になる
- 効果:
  - アクティベーションの勾配値がFP16の表現可能域にシフト
  - ウェイトの勾配値はFP16の正規数領域に入る

ロス・スケーリングの効果

Alexnet

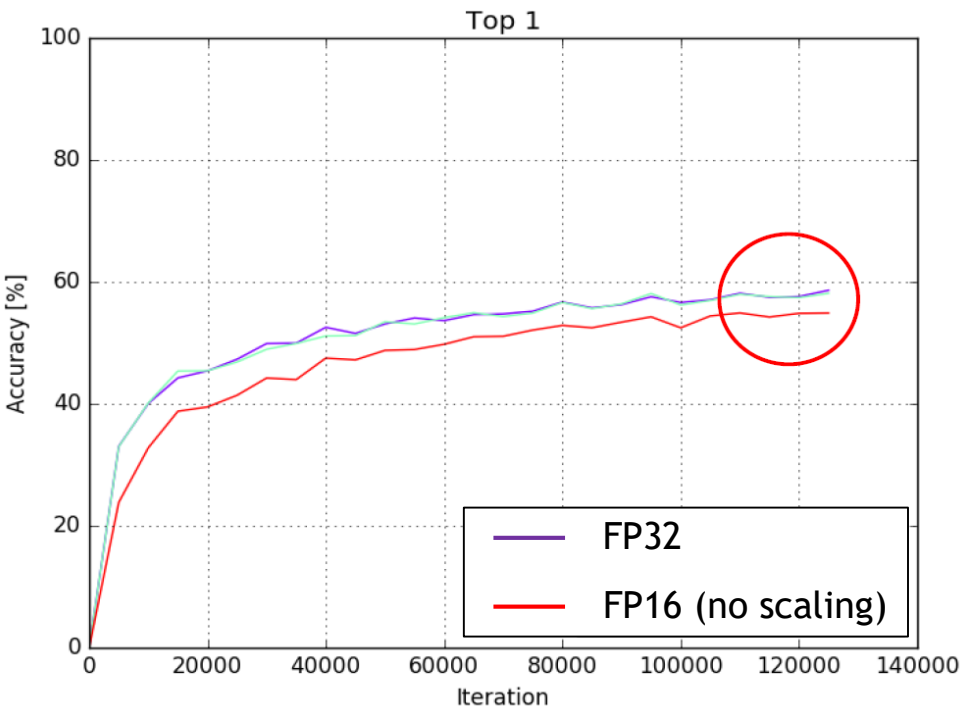
トレーニングモード	Top1 (%)	Top5 (%)
FP32	58.6	81.3
FP16 (スケーリング無し)	56.7	78.1

(*) Nvcaffe-0.16, momentum SGD, 100 epochs, 1024 batch, DGX1

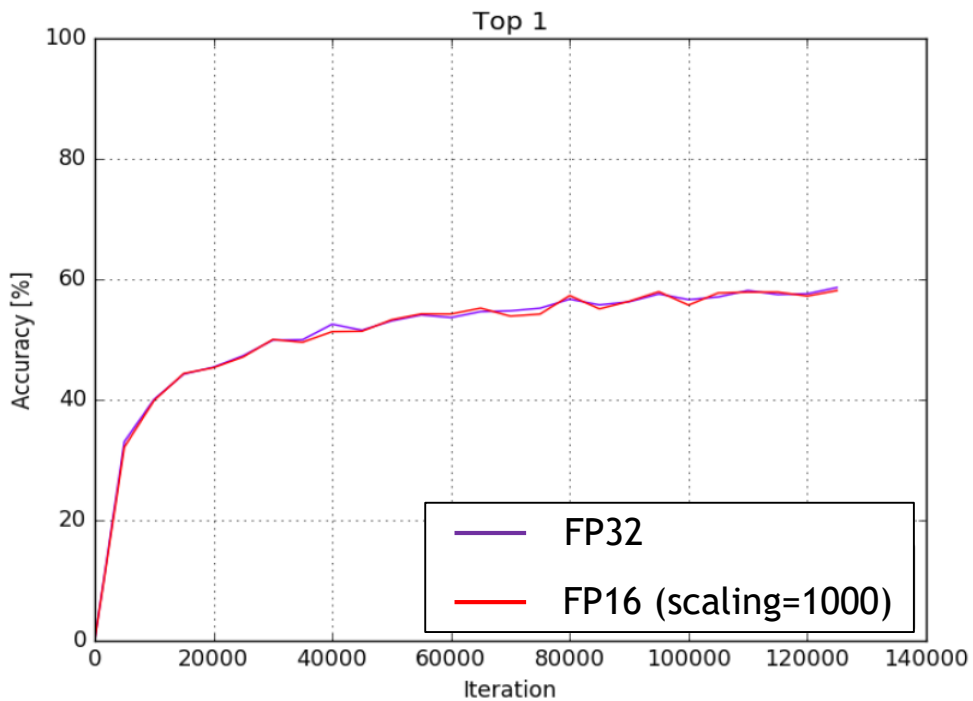
ロス・スケーリングの効果

Alexnet

ロス・スケーリング無し



ロス・スケーリング有り



ロス・スケーリングの効果

物体検出

トレーニングモード	Multibox SSD (mAP)	Facter-RCNN (mAP)
FP32	76.9%	69.1%
Tensorコア (スケーリング無し)	X	68.5%

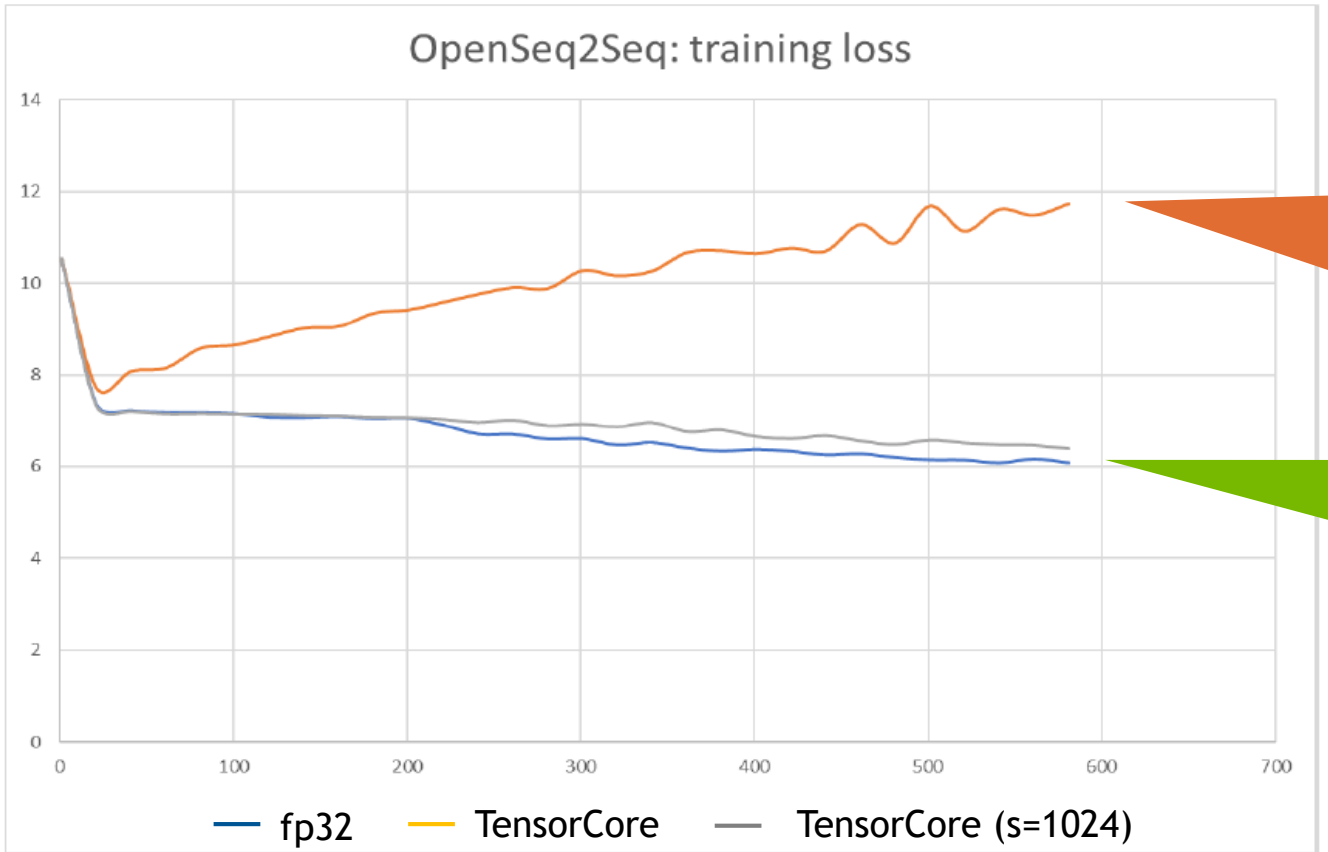
SEQ2SEQ

NMT: ドイツ語 - 英語

- OpenSeq2Seq
  - https://github.com/NVIDIA/OpenSeq2Sseq
- NMT_ONE model
  - Encoder: 2-layer bi-directional (512 LSTM)
  - Attention: Normalized Bahdanau
  - Decoder: 4-layer (512 LSTM)

SEQ2SEQ

OpenSeq2Seq

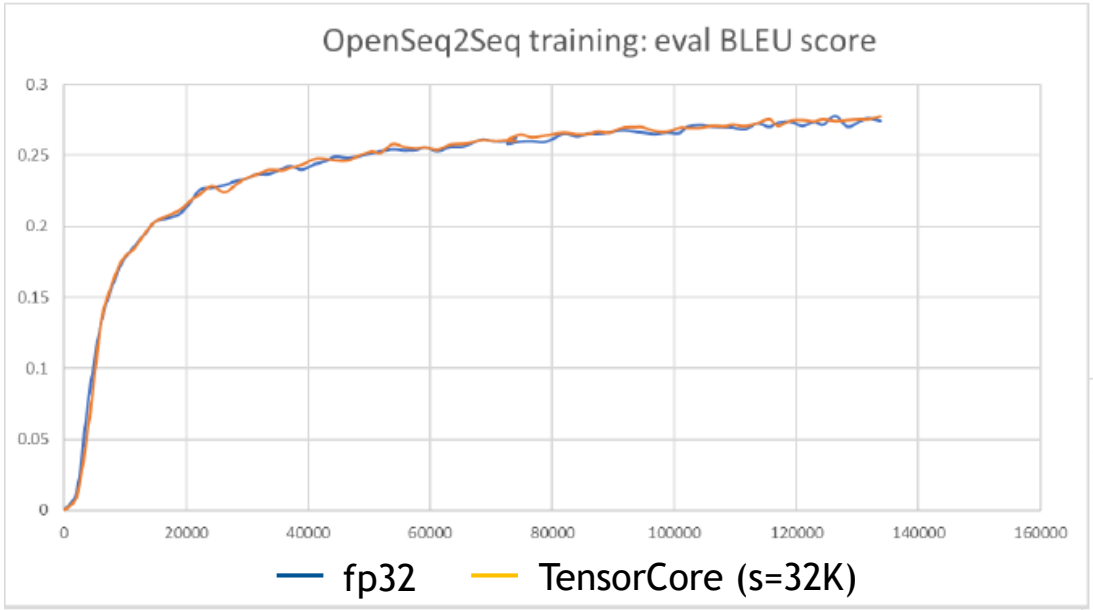


単にTensorコアを
使用するだけでは、
精度が低下

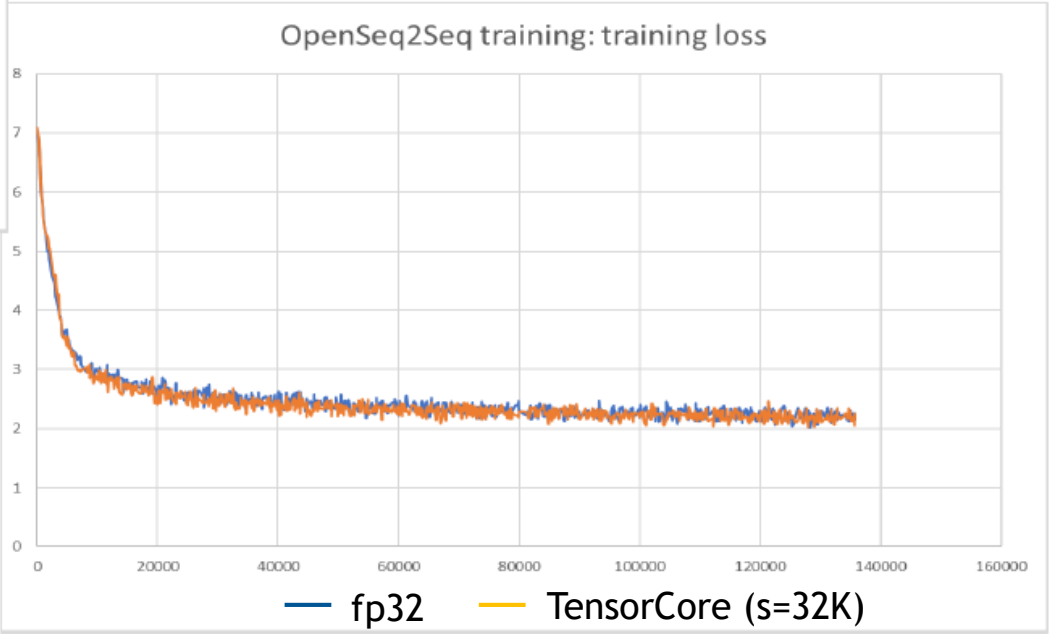
ロス・スケーリング
(1024)で、FP32と
同程度の精度

SEQ2SEQ

NMT_ONE



スケーリングファクター
小さくできないか?



ロス・スケーリング使用で、
FP32と同程度の精度
スケーリングファクター: 32K

ロス関数変更とラーニングレート調整

- Ave Loss → Sum Loss

$$Loss_{avg} = Avg_{batch}(\boldsymbol{Avg_{timesteps}}(crossentropy(logits, targets)))$$

$$Loss_{sum} = Avg_{batch}(\boldsymbol{SUM_{timesteps}}\ (crossentropy(logits, targets)))$$

- LARS (Layer-wise Adaptive Rate Scaling)

  レイヤー毎に学習率を調整

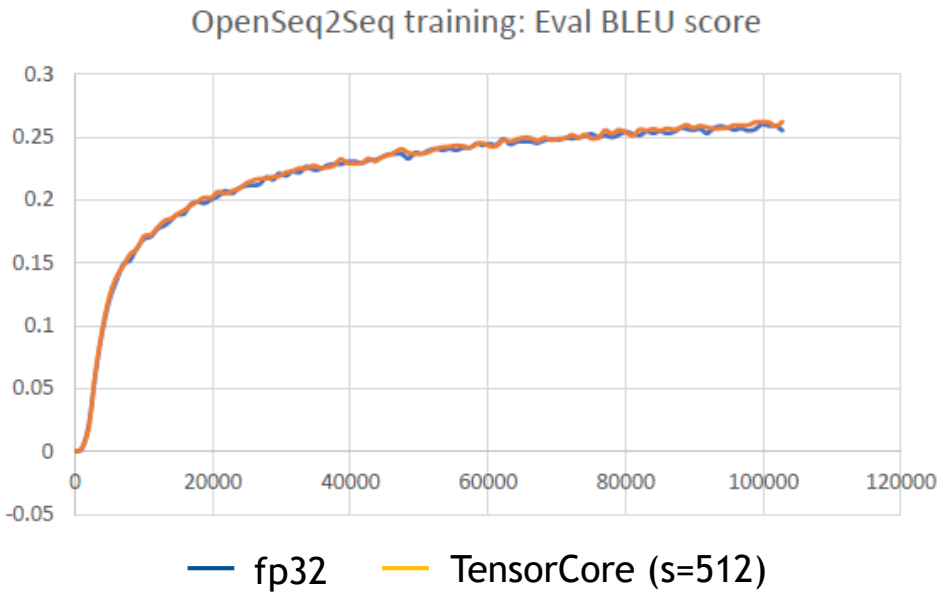
$$\triangle w_t^l = \gamma * \lambda^l * \nabla L(w_t^l)$$

$$\lambda^l = \eta \times \frac{||w^l||}{||\nabla L(w^l)||}$$

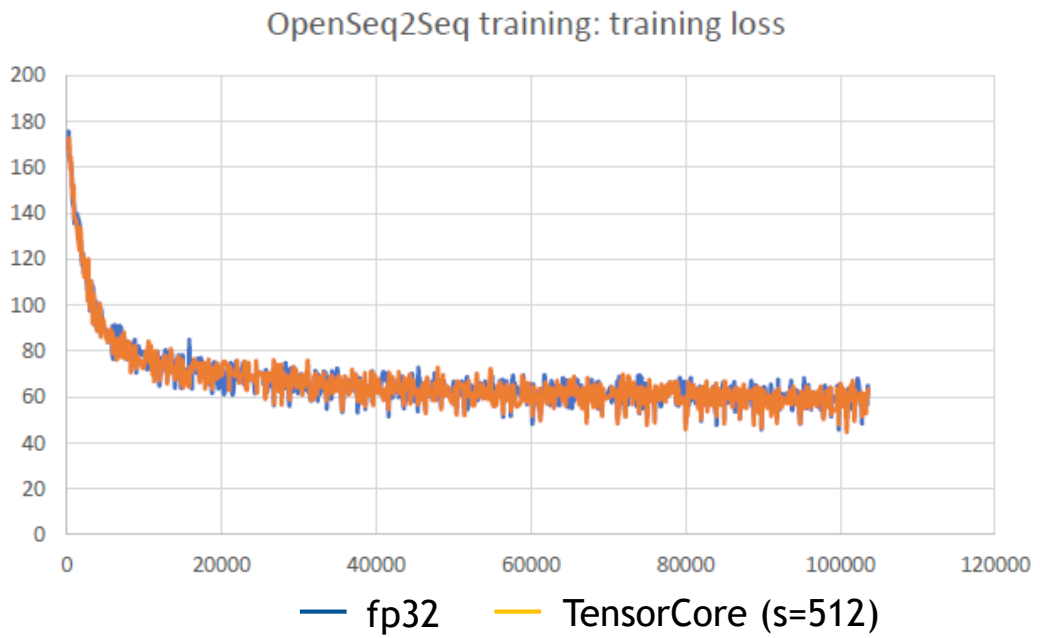
(*) Yang You, et al., “Large Batch Training of Convolutional Networks”, 2017

SEQ2SEQ

NMT_ONE

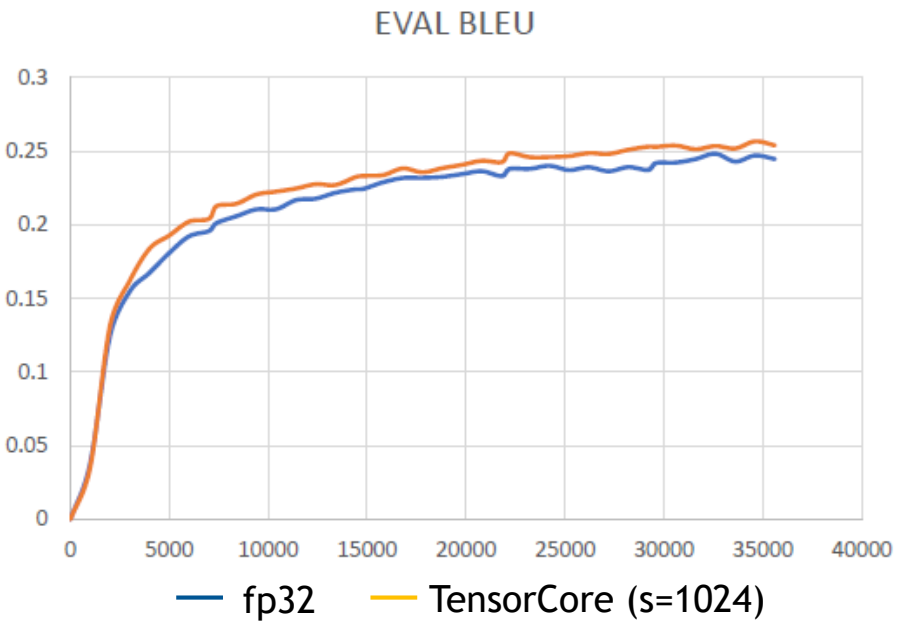


Sum LossとLARS使用
スケーリングファクター: 512
FP32と同程度の精度

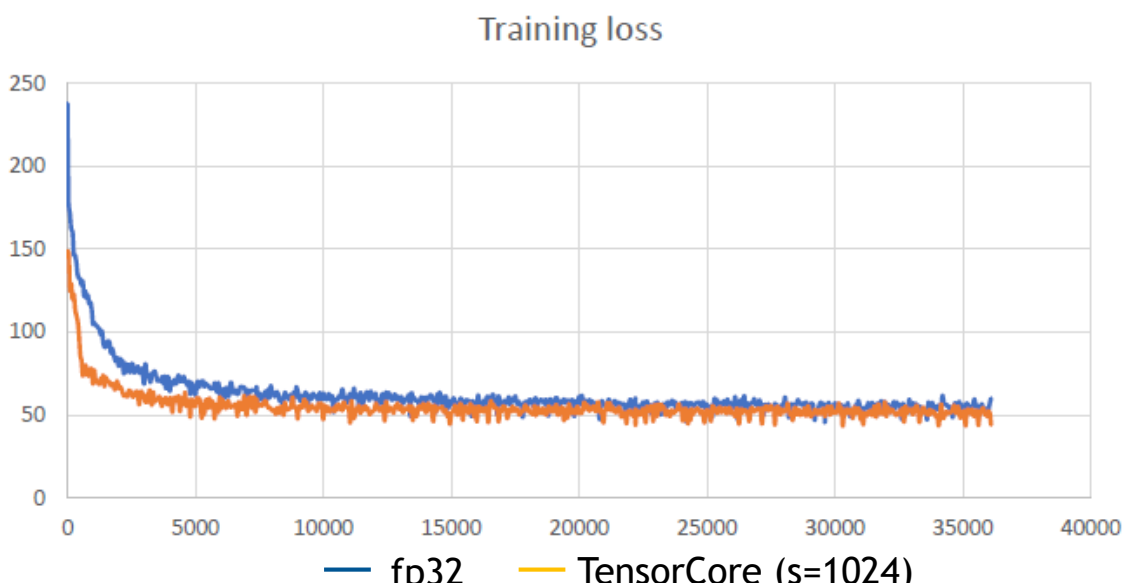


SEQ2SEQ

GNMT-like



Encoder: 8-layer bi-directional (1024 LSTM)
Attention: GNMT-style normalized Bahdanau
Decoder: 8-layer (1024 LSTM)



Sum LossとLARS使用
スケーリングファクター: 1024
FP32と同程度の精度

(*) Yonghui Wu, et al.: “Google’s neural machine translation system: Bridging the gap between human and machine translation.”, 2016

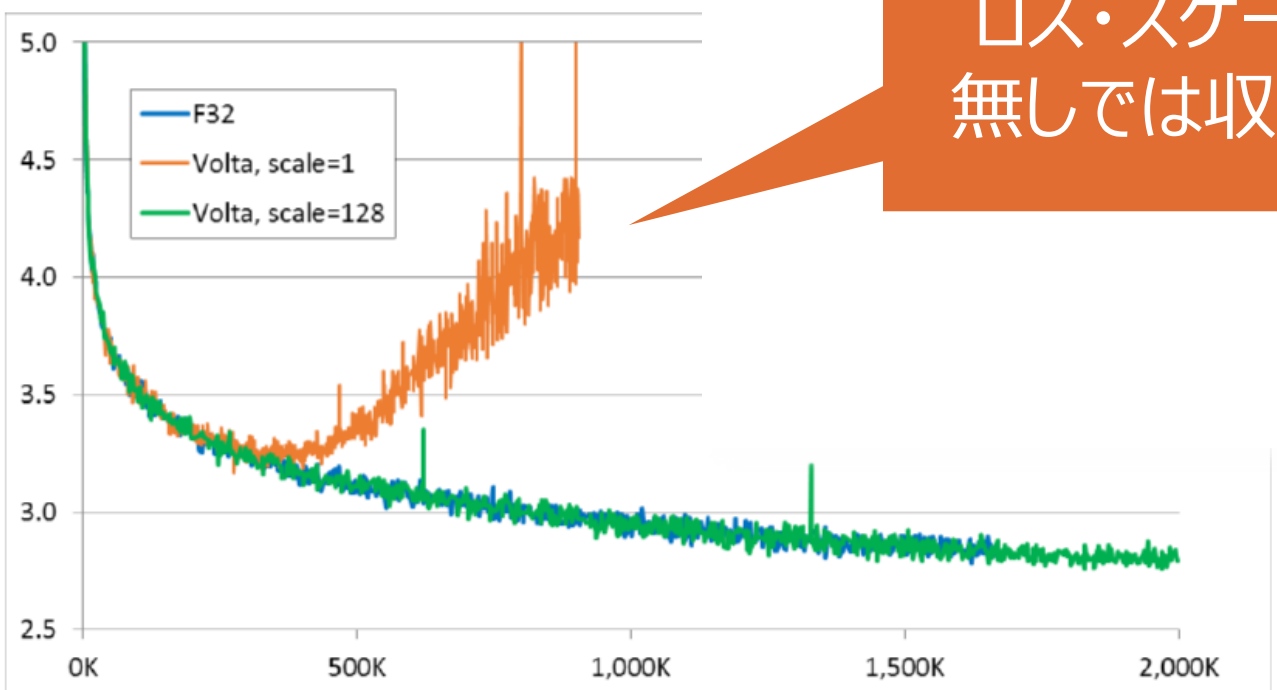
言語モデル

- 1 Billion Word Language Benchmark
- BigLSTM
  - 2 x 8192 LSTM, 1024 Projection
  - Vocabulary: 800K words
  - Solver: Adagrad

(*) Rafal Jozefowicz, et al.: “Exploring the Limits of Language Modeling.”, 2016

言語モデル

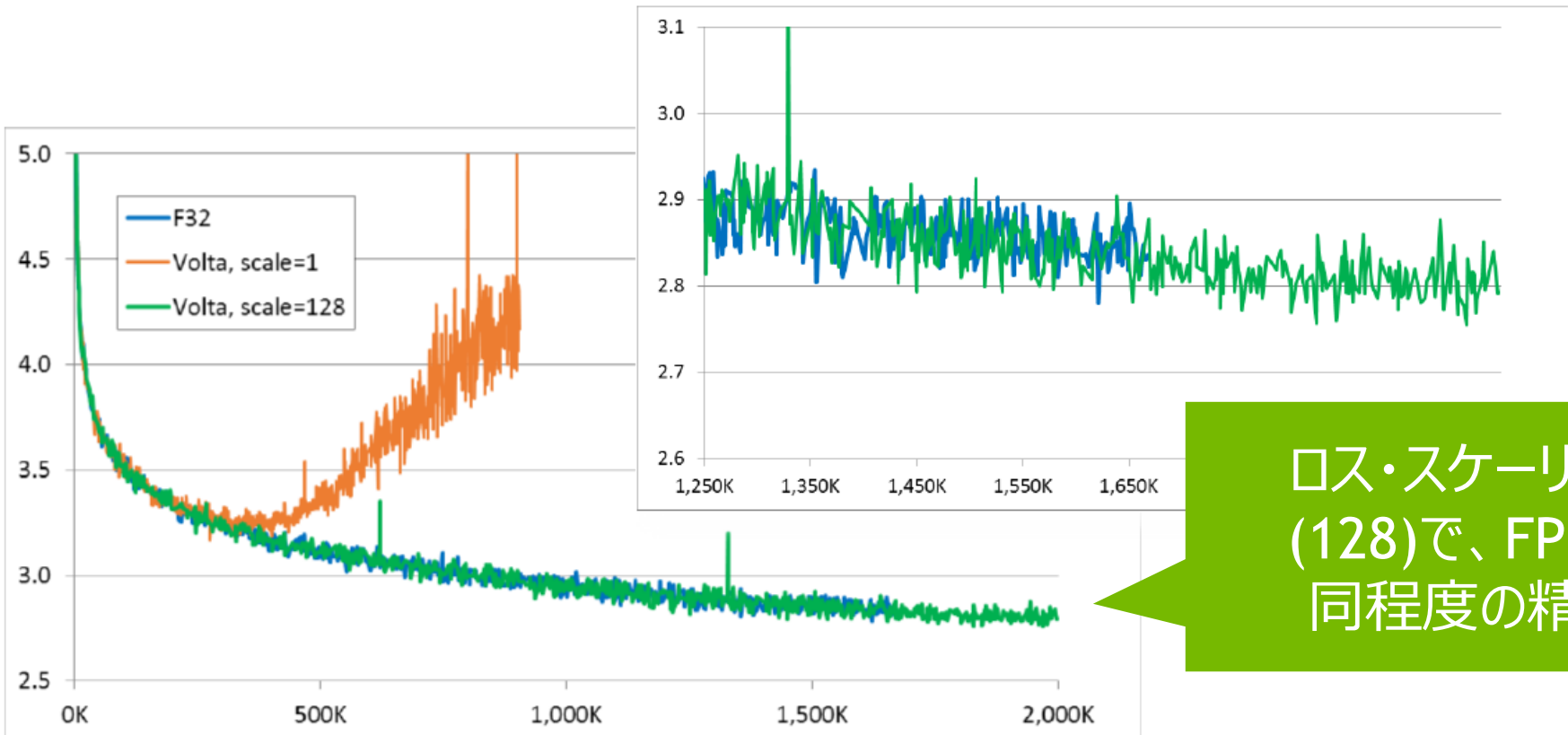
BigLSTM: 2 x 8192 LSTM, 1024 projection



ロス・スケーリング
無しでは収束せず

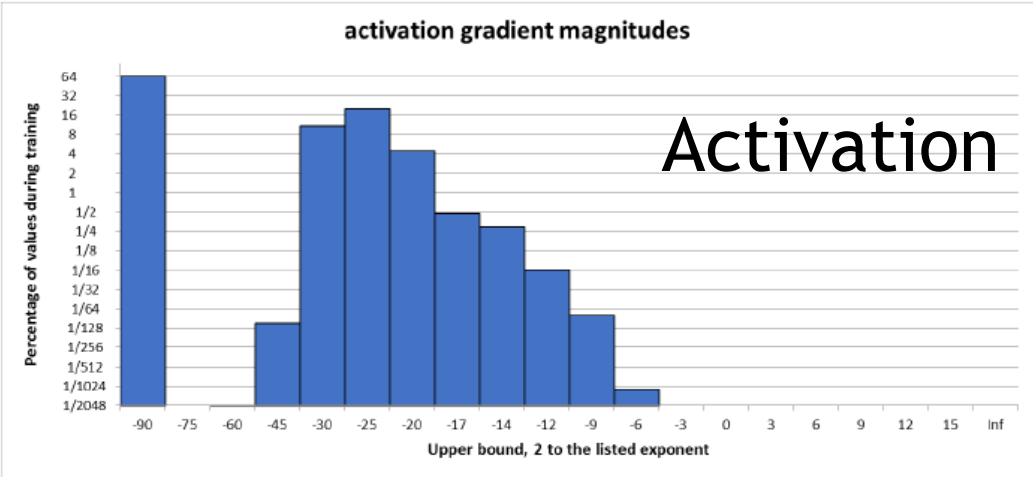
言語モデル

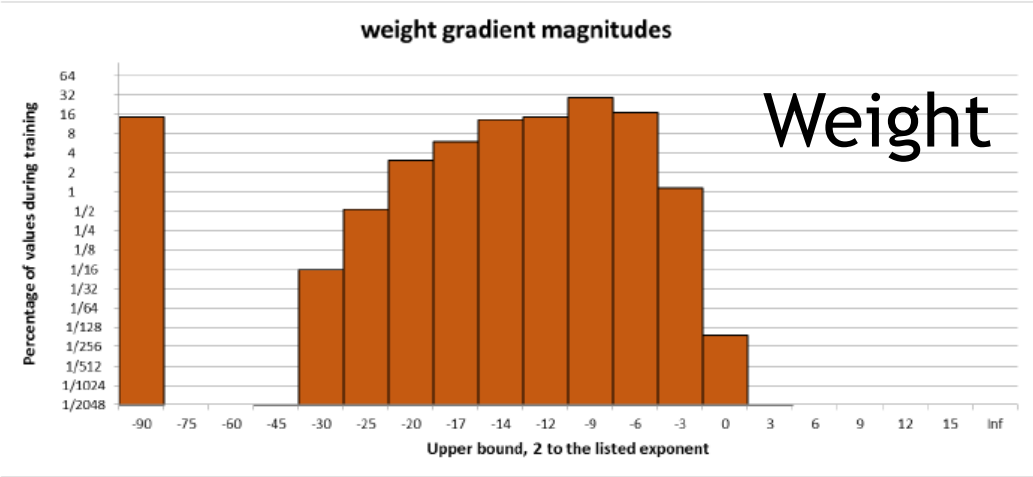
BigLSTM: 2 x 8192 LSTM, 1024 projection



ロス・スケーリング(128)で、FP32と同程度の精度

勾配値の特徴





勾配の範囲は、FP16の表現可能領域より、小さいほうに偏っている

- 最大値は高々10程度?
- オーバフローすることなく、スケールアップ可能 (～ 1024倍)

重みの勾配 >> Activationの勾配

- 消失しやすいのは、Activationの勾配
- ほぼ全てのモデルで共通の傾向

VOLTA 混合精度トレーニング

FP32と同じ精度のモデルをトレーニングする方法

- ストレージ (weights, activation, gradients): FP16
- ForwardとBackpropの計算: Tensorコア
  - Batch Normalizationの計算はFP32 (cuDNNは、FP16入力、FP32計算)
- Updateの計算: FP32 (weightsはfp16とfp32の両方で管理)

注意

- 勾配は、FP16で表現できないほど、小さくなることがある (勾配消失)
- 勾配消失は、ロススケーリングで解消できる

DLフレームワークの対応状況

NVIDIA CAFFE 0.16

https://github.com/NVIDIA/caffe/tree/caffe-0.16

FP16、Tensorコアに完全対応

ForwardとBackward: それぞれ、データ型、計算型を指定可能 (FP32 or FP16)

ウェイト更新: FP32更新対応

ロス・スケーリング対応

NVIDIA CAFFE 0.16

https://github.com/NVIDIA/caffe/tree/caffe-0.16

```
name: "AlexNet_fp16"

default_forward_type:  FLOAT16
default_backward_type: FLOAT16

# default_forward_math:  FLOAT    # GP100 only
# default_backward_math: FLOAT    # GP100 only

global_grad_scale: 1000.

layer {
  forward_math:  FLOAT16
  backward_math: FLOAT
  ...
}
```

```
solver_data_type: FLOAT16
```

TENSOR FLOW

Tensorコア: TensorFlow 1.4で対応

データ型をFP16にすると、Tensorコアを使用

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

ウェイトFP32更新: 可能

ロススケーリング: 可能

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

PYTORCH

Tensorコア: 対応

FP16ストレージにすると、Tensorコアを使用

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

ウェイトFP32更新: 可能

ロススケーリング: 可能

CHAINER

Tensorコア: Chainer V4で対応予定

データ型をFP16にすると、Tensorコア使用

```
x = F.cast(x, np.float16)
```

FP32パラメータ更新: 対応

```
optimizer = chinaer.optimizers.SGD()
optimizer.use_fp32_update()
```

ロススケーリング: 対応(予定)

```
loss = lossfunc(y, t)
loss.backward(loss_scale=1024)
```

ウェイトをFP16で更新できないか?

ウェイトをFP32で更新する問題

FP16とFP32の2種類のデータ型で、ウェイトを管理する必要がある

　　メモリ使用量の増加

FP16でウェイトを更新できないか?

SGD

FP16の問題: 更新消失

SGDによるウェイト更新

$$W(t+1) = W(t) - \lambda * \Delta W(t) \qquad (\lambda:学習率)$$

FP16を使うと、$\lambda * \Delta W(t)$ が小さくなりすぎることがある

- 学習初期: $\Delta W(t)$が非常に小さい ($\lambda<1$)
- 中盤以降: 学習初期より、$\Delta W(t)$は大きくなるが、λは小さくなる

モメンタムSGD

1.モメンタム計算:　$H(t+1) = m * H(t) - \lambda * \Delta W(t)$　　(m:モメンタム係数)

2.ウェイト更新:　　$W(t+1) = W(t) + H(t+1)$

FP16の場合、モメンタム計算、$\lambda * \Delta W(t)$ の減算で更新消失が起きやすい

モメンタムSGD

1.モメンタム計算:　$H(t+1) = m * H(t) - \lambda * \Delta W(t)$　　(m:モメンタム係数)

2.ウェイト更新:　　$W(t+1) = W(t) + H(t+1)$

モメンタム計算再考

$H(t+1) = -\lambda*\Delta W(t) + m*H(t)$

$= -\lambda*\Delta W(t) + m*(-\lambda*\Delta W(t-1) + m*H(t-1))$

$= -\lambda*\Delta W(t) + m*(-\lambda*\Delta W(t-1) + m*(-\lambda*\Delta W(t-2) + m*H(t-2)))$

$= -\lambda * (\Delta W(t) + m*\Delta W(t-1) + m^2*\Delta W(t-2) + \ldots + m^k*\Delta W(t-k) + \ldots)$

モメンタムは、勾配の蓄積と見なすことができる?

修正モメンタムSGD

FP32を使わなくても、更新消失を回避できる?

モメンタムSGD

1.モメンタム計算: $H(t+1) = m * H(t) - \lambda * \Delta W(t)$

2.ウェイト更新: $W(t+1) = W(t) + H(t+1)$

こう、解釈することも可能

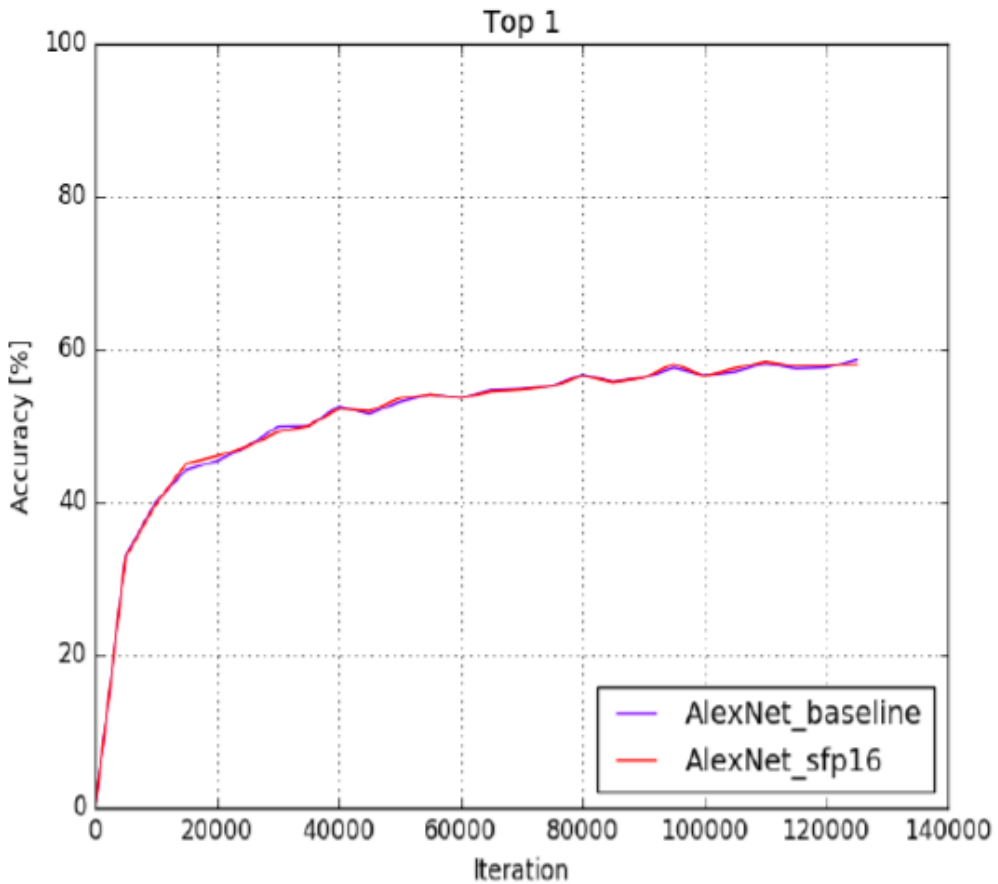
1.モメンタム計算: $G(t+1) = m * G(t) + \Delta W(t)$

2.ウェイト更新: $W(t+1) = W(t) - \lambda * G(t+1)$

G(t)は勾配の蓄積なので消失しにくい → ウェイトは正しく更新される?

ウェイトもFP16で更新

修正モメンタムSGD



AlexNet

- FP32と同じ精度を達成

ALEXNET

修正モメンタムSGD

トレーニングモード	Top1 (%)	Top5 (%)
FP32	58.6	81.3
FP16 (スケーリング無し)	56.7	78.1
FP16 (scaling=1000)	58.9	81.1
Tensorコア (scaling=1000)	59.1	81.2

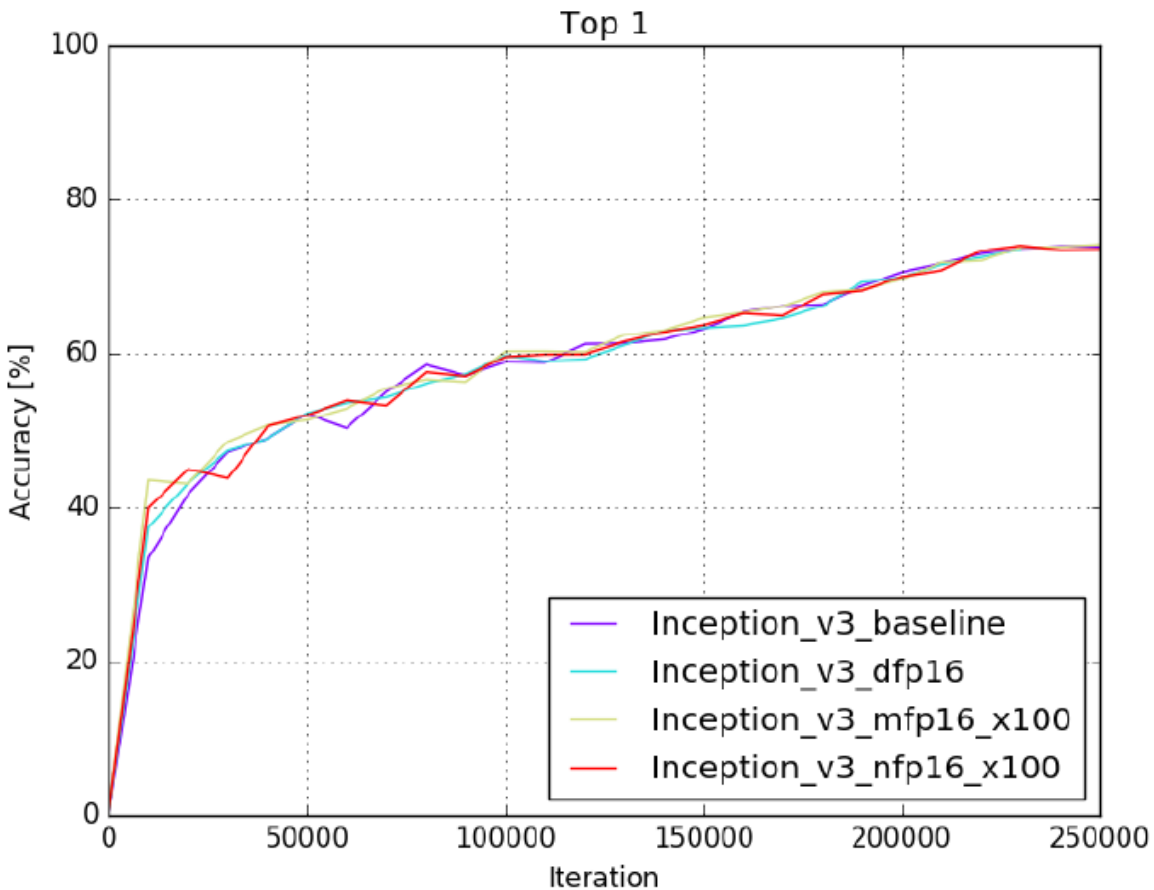
INCEPTION-V3

修正モメンタムSGD

トレーニングモード	Top1 (%)	Top5 (%)
FP32	73.8	91.4
FP16 (スケーリング無し)	51.4	90.8
FP16 (scaling=100)	74.1	91.5

INCEPTION-V3

修正モメンタムSGD



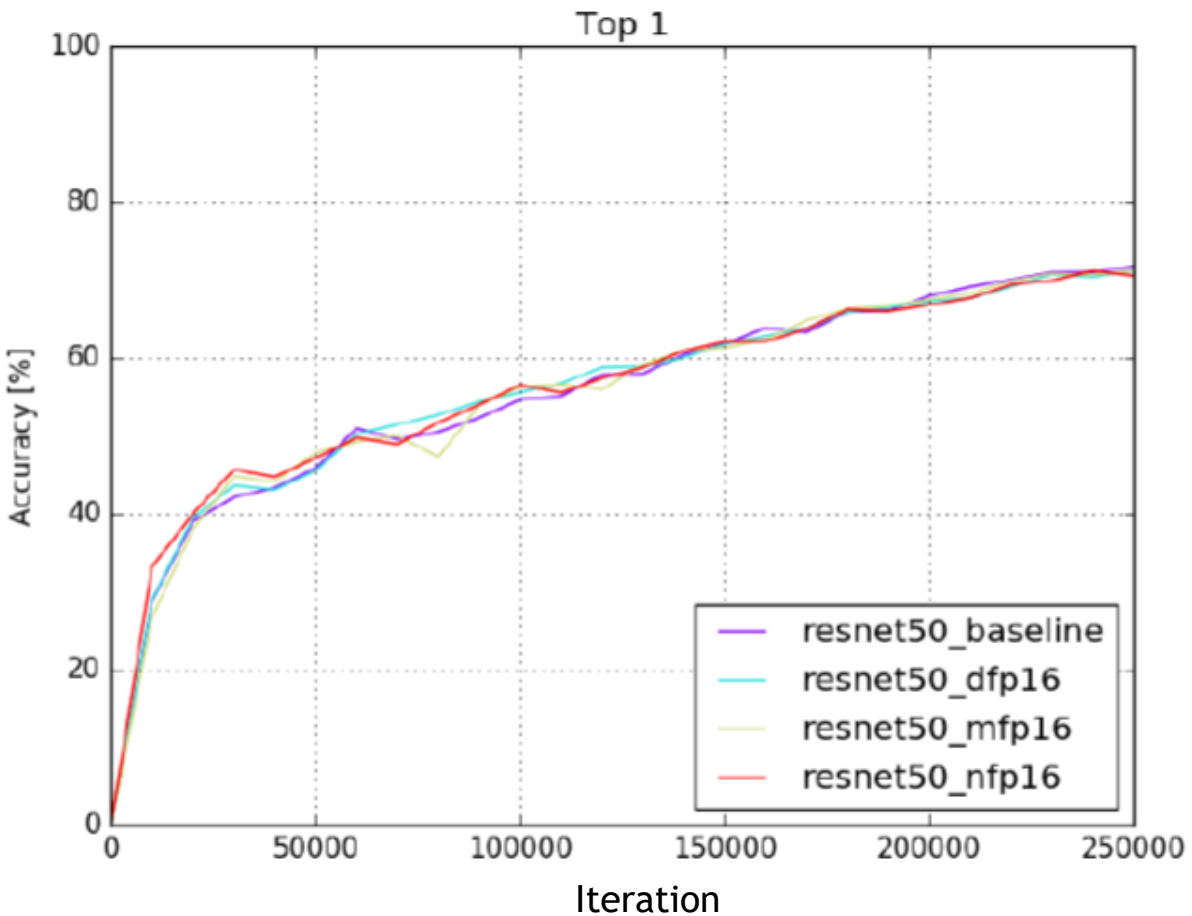
RESNET50

修正モメンタムSGD

トレーニングモード	Top1 (%)	Top5 (%)
FP32	73.2	91.2
FP16 (no scaling)	73.2	90.9

RESNET50

修正モメンタムSGD



まとめ

Tensorコア(混合精度)トレーニング

- ForwardとBackprop(計算の大部分)はTensorコアで計算する
- ウェイトをFP32で更新する
- 多くのモデルはこれで収束 (FP32と同程度の精度)
- それ以外も、ロス・スケーリング設定でFP32レベルの精度に回復

ウェイトもFP16で更新

- モメンタムSGDの修正で、CNNでFP32と同等の精度を確認

LINKS

"Mixed-Precision Training of Deep Neural Networks", NVIDIA blog post

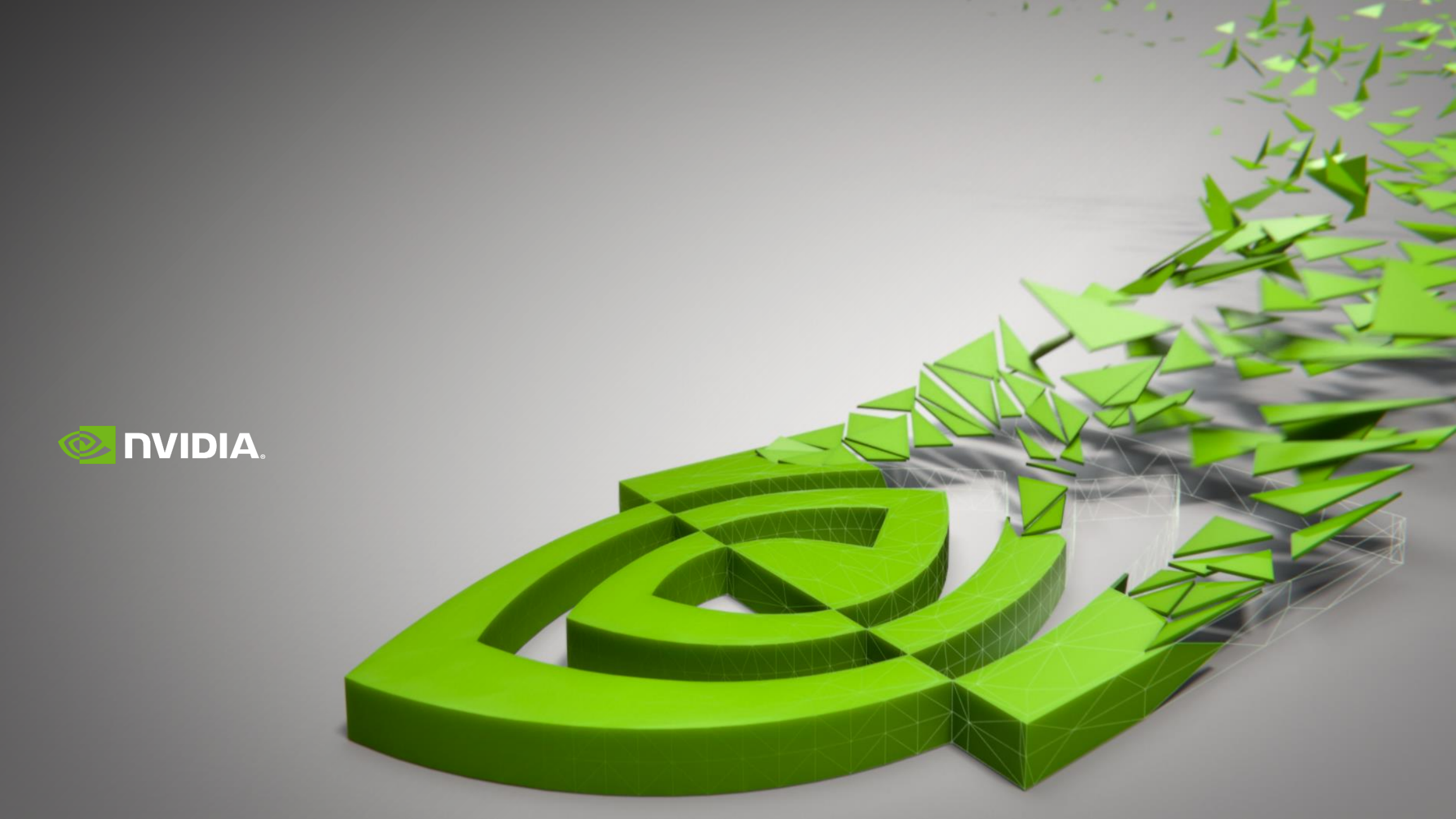
- devblogs.nvidia.com/parallelforall/mixed-precision-training-deep-neural-networks/

"Training with Mixed Precision", NVIDIA DL SDK doc

- docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html

Paulius Micikevicius, et al., "Mixed Precision Training"

- arxiv.org/abs/1710.03740


NVIDIA