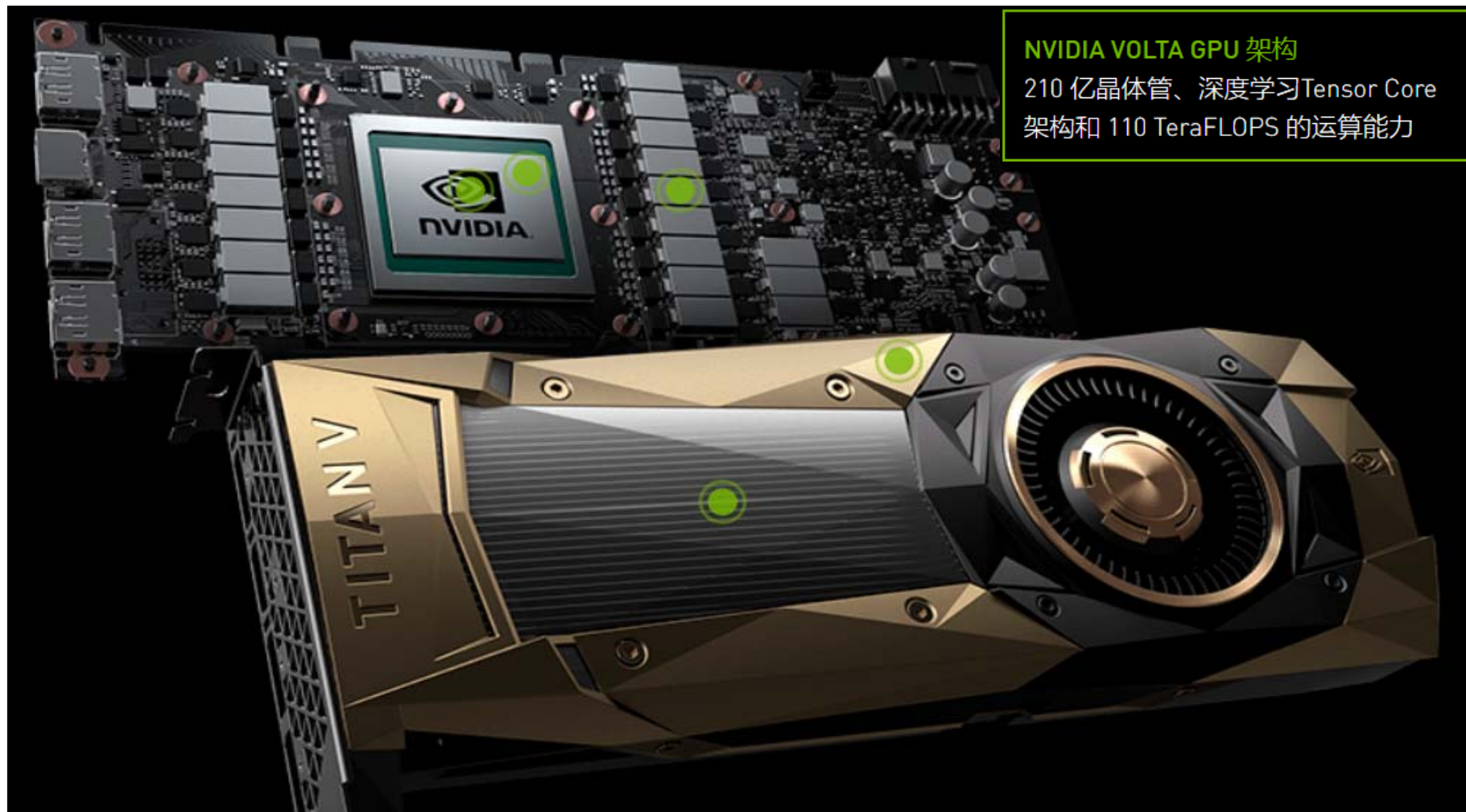


# Introduction to CUDA

*(9) Tensor Cores  
&  
Mixed Precision Programming*

# Pascal → Volta → Turing



# Pascal → Volta → Turing



# Content

## **1. Mixed-Precision Programming**

## 2. Tensor Cores

## 3. Mixed-Precision Training of DNN

# Using the right tool

- Numerical Computing:
  - tradeoffs between precision, accuracy, and performance.

INT4

INT8

INT16

FP16

FP32

FP64

FP128

- Pascal GPU architecture and CUDA 8:
  - mixed-precision computing with new 16-bit floating point and 8/16-bit integer computing capabilities.

# Half-precision Floating Point

- Common floating point formats include:
  - 32-bit, known as “single precision”
  - 64-bit, known as “double precision”.
- FP64:
  - 1 sign bit, 11 exponent bits, and 52 mantissa bits.
- FP32:
  - 1 sign bit, 8 exponent bits, and 23 mantissa bits.
  - about 2 billion values.
- FP16: half-precision floating point
  - 1 sign bit, 5 exponent bits, and 10 mantissa bits.
  - Range from  $2^{-14}$  to  $2^{15}$  , that's 30,720 values.

尾数

# Half-precision Floating Point

- FP16: half-precision floating point
  - twice the throughput of single-precision arithmetic,
  - four times the throughput of double precision.
  - 21.2 Teraflop/s of half-precision through NVLink.
- With support of:
  - GP100 GPU
  - supports a 2-way vector half-precision [fused multiply-add \(FMA\)](#) instruction

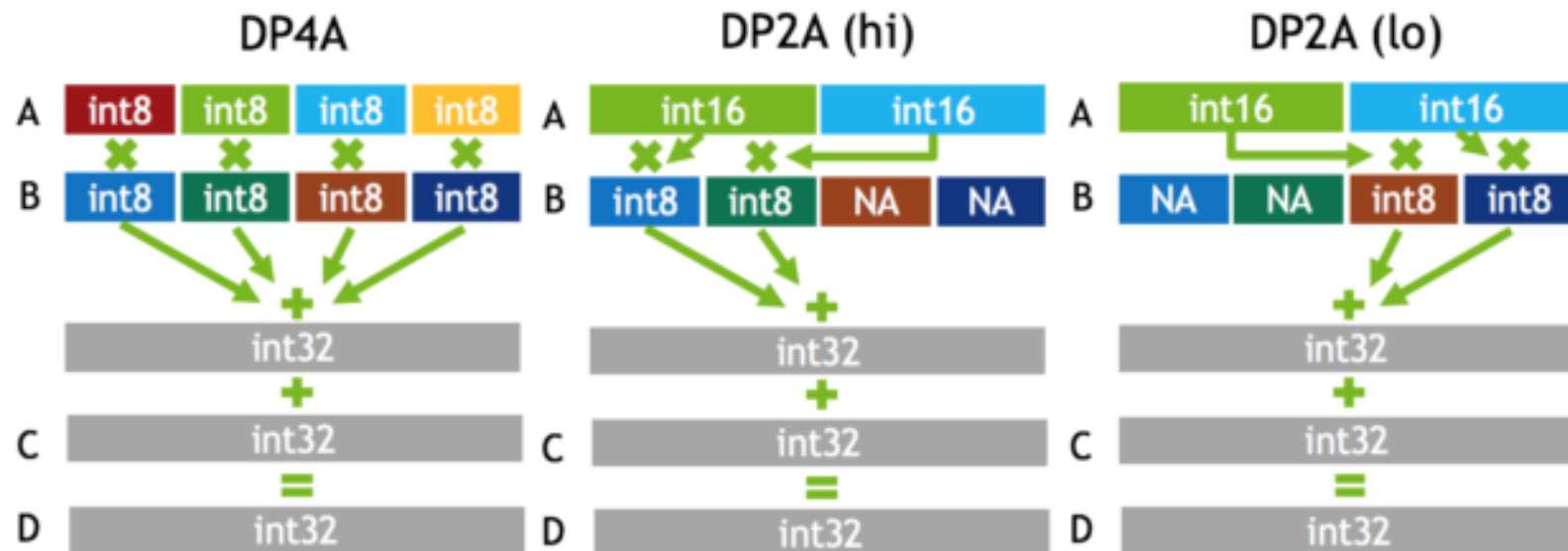
# Low-Precision Integers

- Floating point numbers combine high dynamic range with high precision;
- But there are also cases where [dynamic range is not necessary](#), so that integers may do the job.
- There are even applications where the data being processed has [low precision](#) so very [low-precision storage](#) (such as C short or char/byte types) can be used.



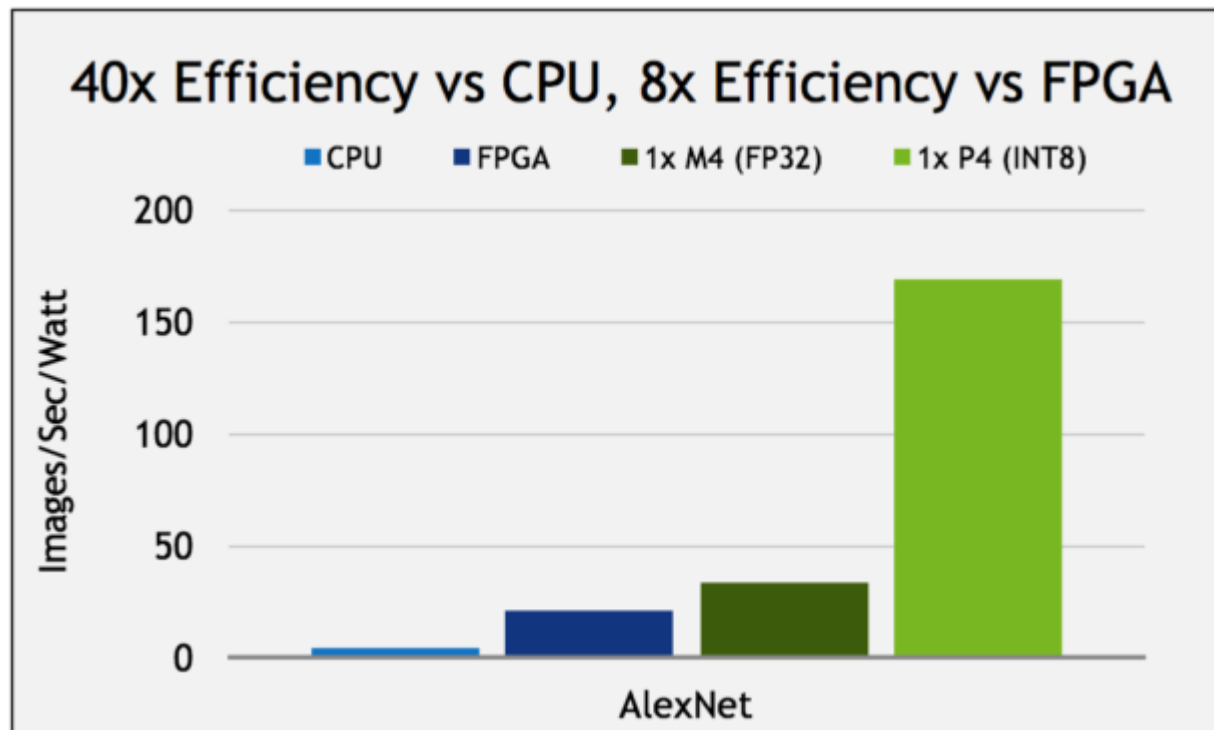
# Low-Precision Integers

- Pascal GPUs (GP102, GP104, and GP106) introduce new instructions :
  - 8-bit integer 4-element vector dot product (DP4A) :
  - 16-bit 2-element vector dot product (DP2A).



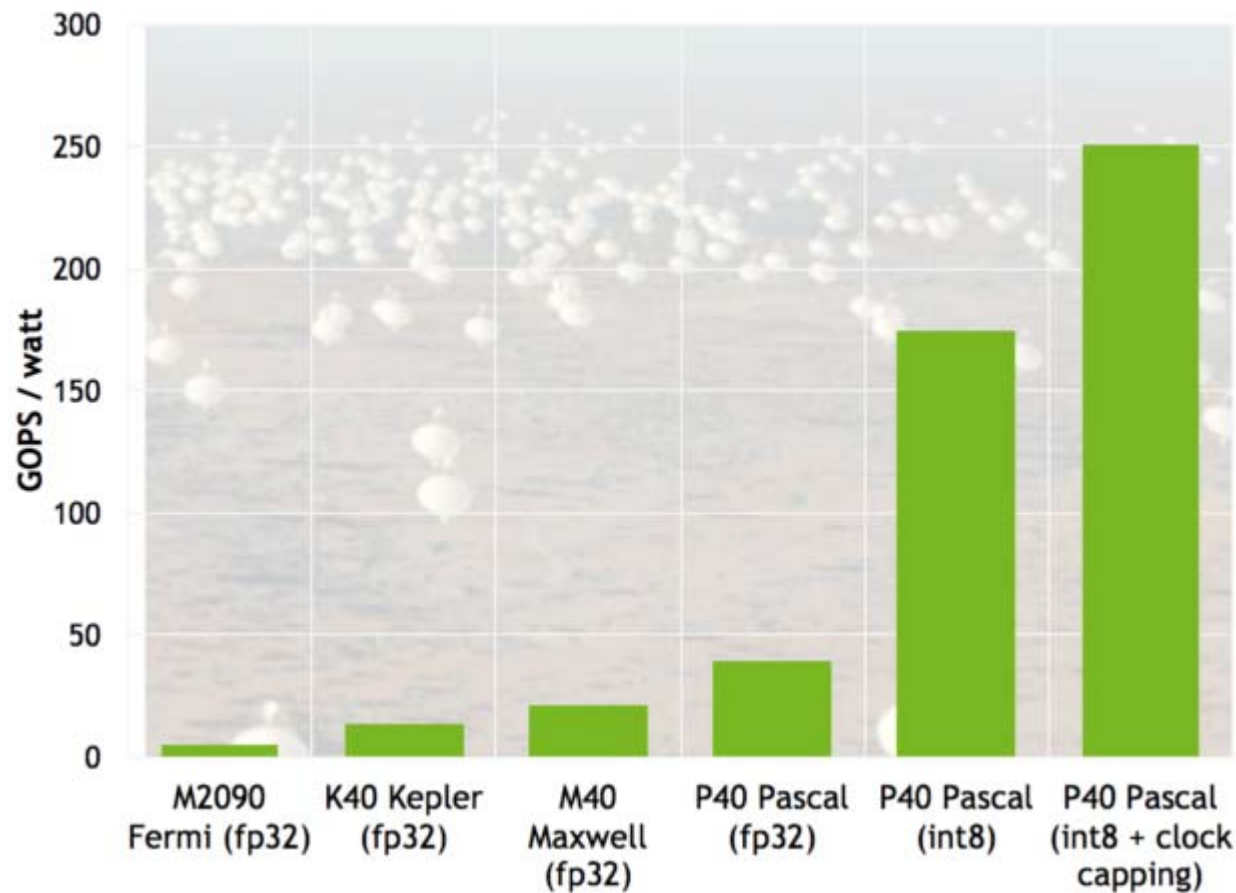
# Low-Precision Integers

- These flexible instructions are useful for linear algebraic computations such as [matrix multiplies](#) and [convolutions](#).
- They are particularly powerful for implementing 8-bit integer convolutions for [deep learning inference](#),



# Low-Precision Integers

- An example application of DP4A is the cross-correlation algorithm commonly used in radio telescope data processing pipelines.



# Performance on Pascal GPUs

- The [Pascal GPU architecture](#) implements general-purpose, IEEE 754 FP16 arithmetic.
- High performance FP16 is supported at full speed on Tesla P100 (GP100), and at lower throughput (similar to double precision) on other Pascal GPUs (GP102, GP104, and GP106)
- The 8-bit and 16-bit DP4A and DP2A dot product instructions are supported on GP102-GP106, but not on GP100.

# NVIDIA Libraries Support

- Key libraries from the NVIDIA SDK now support a variety of precisions for both computation and storage.

Feature	FP16x2 (HFMA2)	INT8/16 DP4A/DP2A
PTX instructions	CUDA 7.5	CUDA 8
CUDA C/C++ <u>intrinsic</u> s	CUDA 7.5	CUDA 8
<u>cuBLAS</u> GEMM	CUDA 7.5	CUDA 8
<u>cuFFT</u>	CUDA 8	I/O via <u>cuFFT</u> callbacks
cuDNN	5.1	6
<u>TensorRT</u>	v1	v2 Tech Preview

# Support of new GPUs

- The latest **Volta** and **Turing** GPUs incorporate **Tensor Cores**
- to accelerate certain types of FP16 matrix math.
- This enables **faster** and **easier** mixed-precision computation within popular AI frameworks.
- Making use of Tensor Cores requires CUDA 9 or later.
- **Automatic mixed precision** capabilities added to PyTorch, TensorFlow, and MXNet....

# Content

1. Mixed-Precision Programming

**2. Tensor Cores**

3. Mixed-Precision Training of DNN

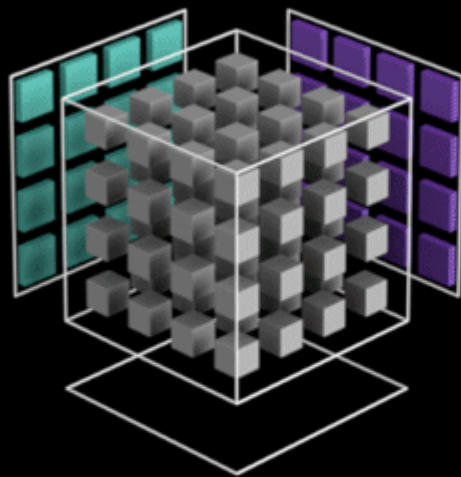
# What are Tensor Cores

- The Next Generation of Deep Learning
  - Accelerate large matrix operations (the heart of AI);
  - Perform mixed-precision matrix multiply and accumulate calculations in a single operation.
- With hundreds of Tensor Cores operating in parallel in one NVIDIA GPU, this enables massive increases in throughput and efficiency.

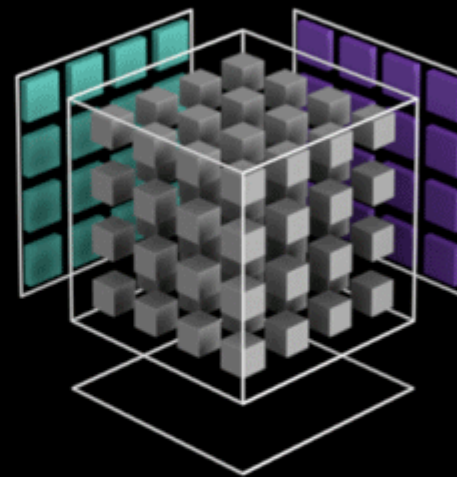


# Volta Tensor Cores

PASCAL



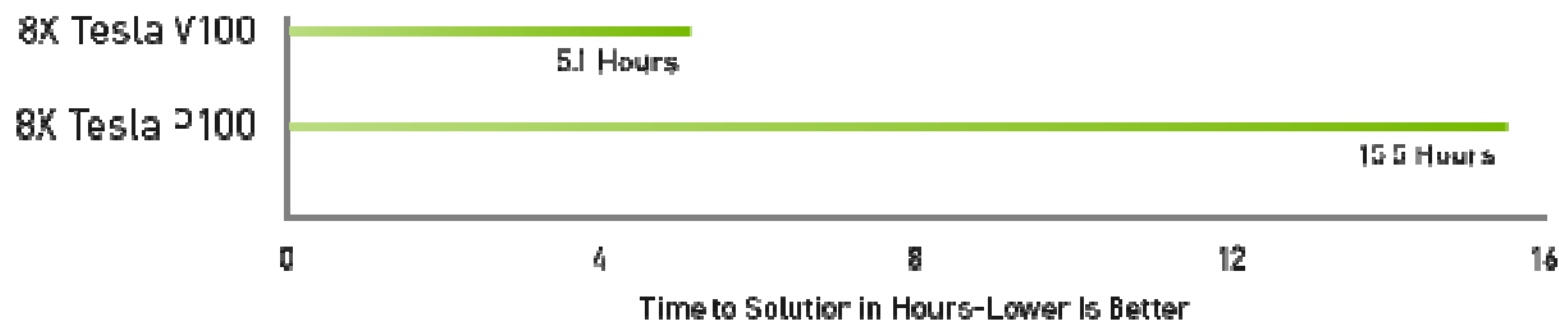
VOLTA TENSOR CORES



# Volta Tensor Cores

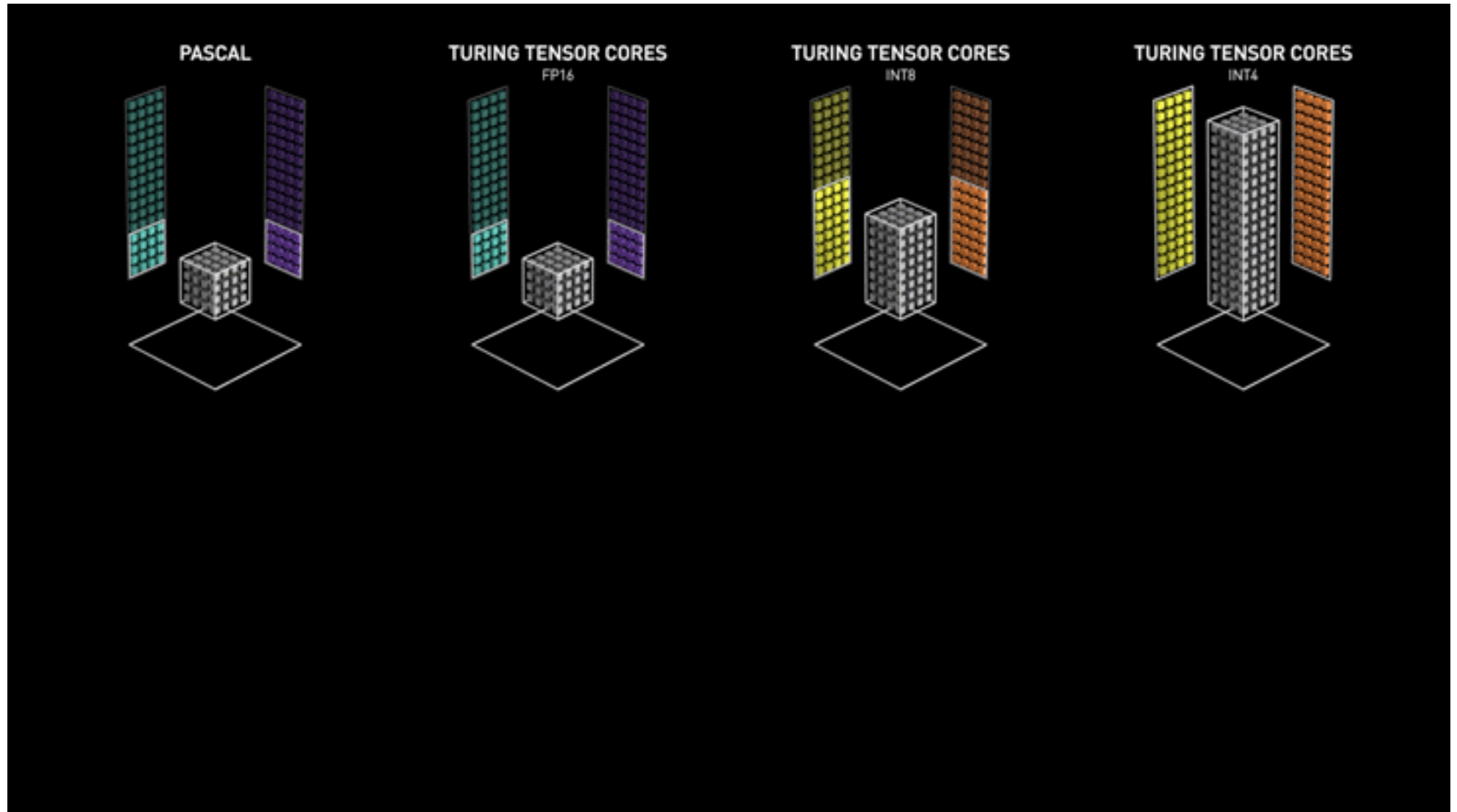
- Volta Tensor Cores
  - Each of Tesla V100's 640 Tensor Cores operates on a 4x4 matrix;
  - mixed-precision matrix multiply in FP16 and FP32;
  - up to 12X higher peak teraflops (TFLOPS) for training and 6X higher peak TFLOPS for inference over Pascal;

## Deep Learning Training in Less Than a Workday



Server Config: Dual Xeon E5-2699 v4 2.6 GHz | 8X NVIDIA® Tesla® P100 or V100 | ResNet-50 Training on MXNet for 90 Epochs with 1.28M ImageNet Dataset.

# Turing Tensor Cores



# Turing Tensor Cores

- Turing Tensor Cores
  - Tesla T4 introduces NVIDIA Turing Tensor Core technology with multi-precision computing for the world's most efficient [AI inference](#).
  - Provide a full range of precisions for inference, from FP32 to FP16 to INT8, as well as INT4;
  - Up to 40X higher performance compared to CPUs with just 60 percent of the power consumption

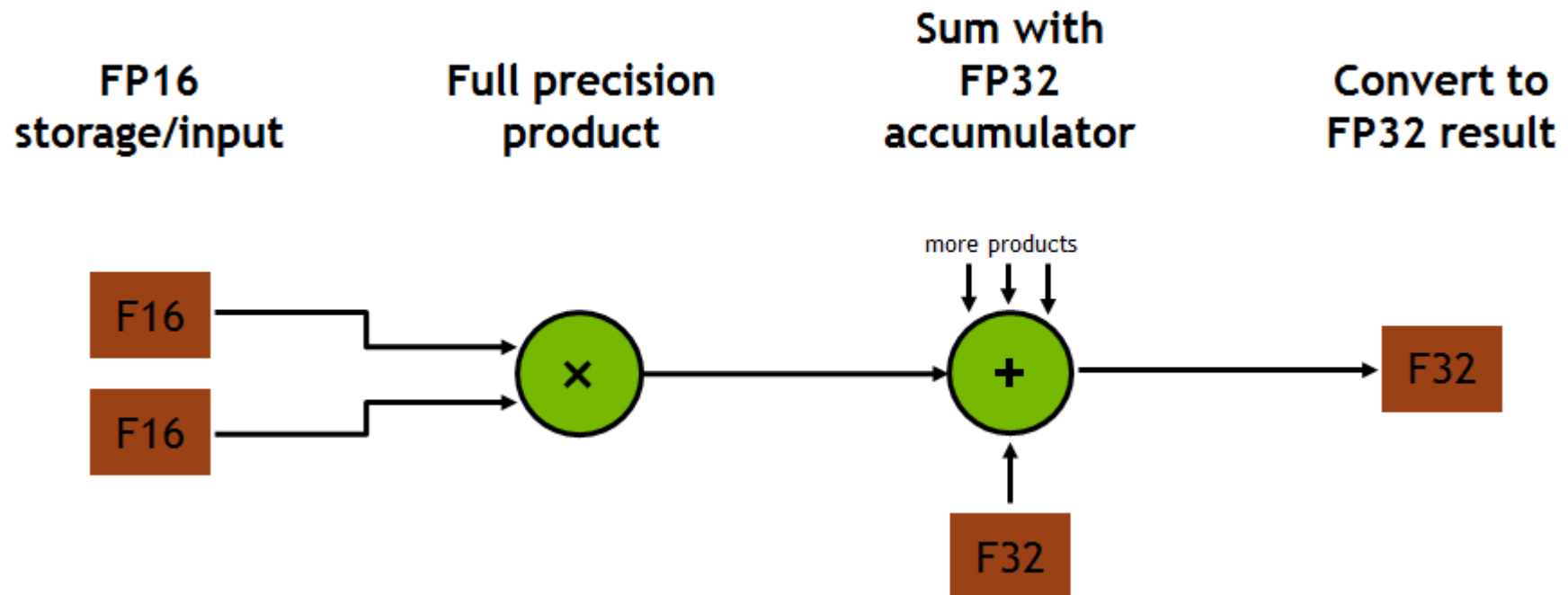
# Tensor Cores

- Each Tensor Core provides a matrix processing array which performs the operation  $D = A * B + C$ , where A, B, C and D are 4x4 matrices

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

# Tensor Cores

- Multiple Tensor Cores are used concurrently by a full warp of execution. The threads within a warp provide a larger 16x16x16 matrix operation to be processed by the Tensor Cores. CUDA exposes these operations as warp-level matrix operations in the CUDA C++ WMMA API.

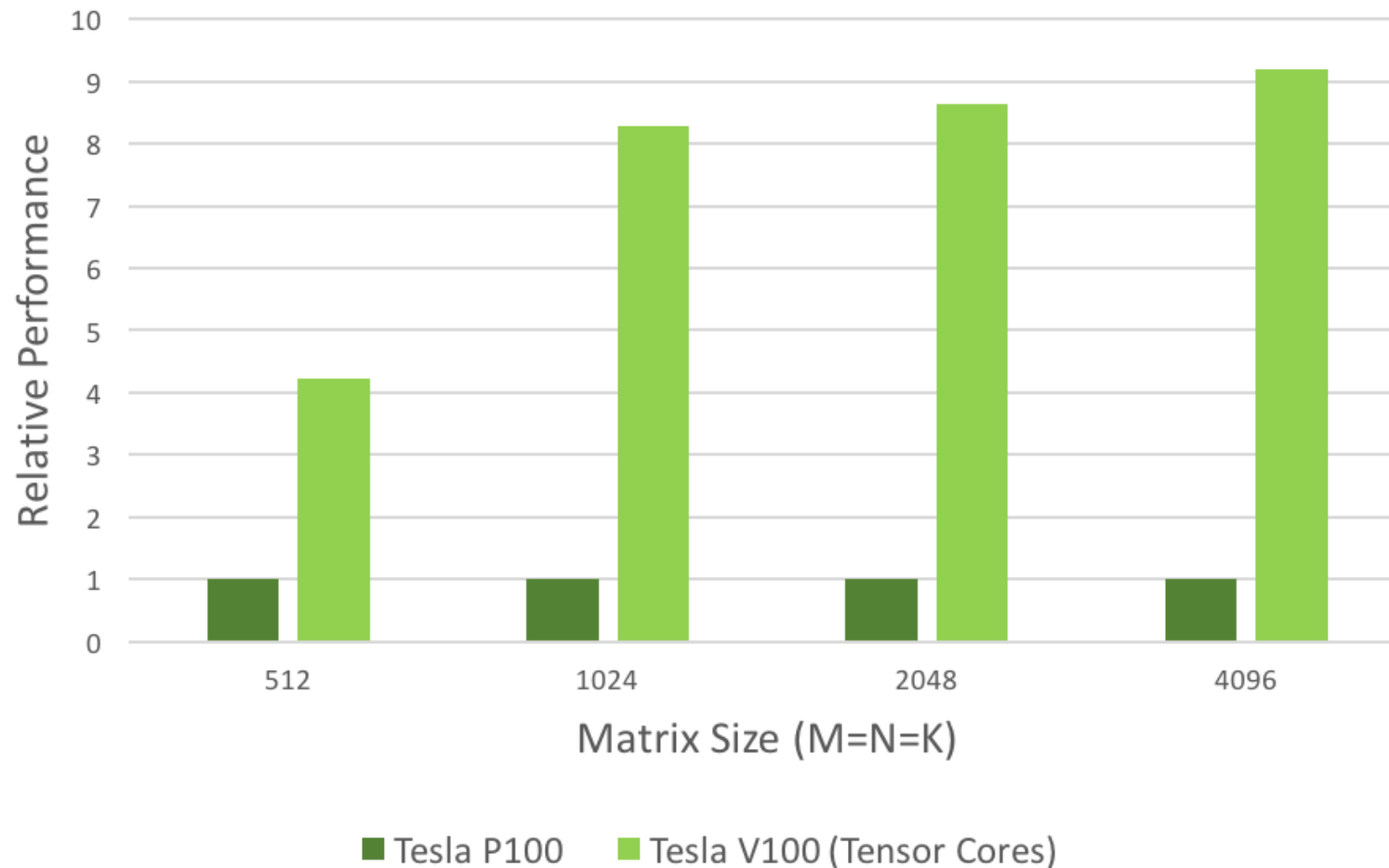


# Tensor Cores in CUDA Libraries

- Two CUDA libraries that use Tensor Cores are cuBLAS and cuDNN.
- cuBLAS uses Tensor Cores to speed up GEMM computations (GEMM is the BLAS term for a matrix-matrix multiplication);
- cuDNN uses Tensor Cores to speed up both convolutions and recurrent neural networks (RNNs).

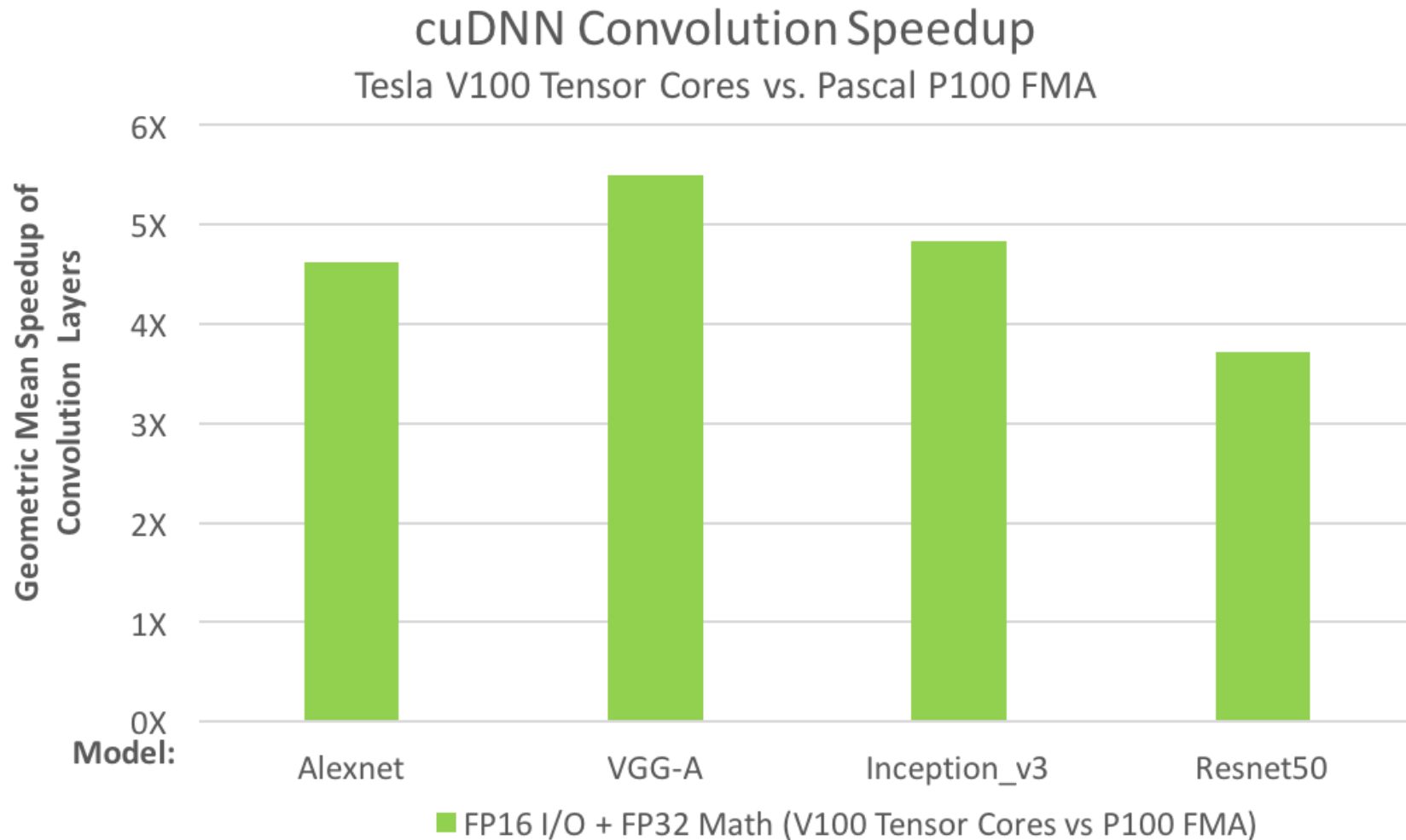
# Tensor Cores using cuBLAS

cuBLAS Mixed-Precision GEMM  
(FP16 Input, FP32 Compute)





# Tensor Cores using cuDNN



# Rules to use Tensor Cores

- The math type must be set to
  - CUDNN\_TENSOR\_OP\_MATH
  - CUBLAS\_TENSOR\_OP\_MATH
- Both input and output channel dimensions must be a multiple of eight.
- Others refer to user's manual.

# Tensor Cores for Scientific Computing

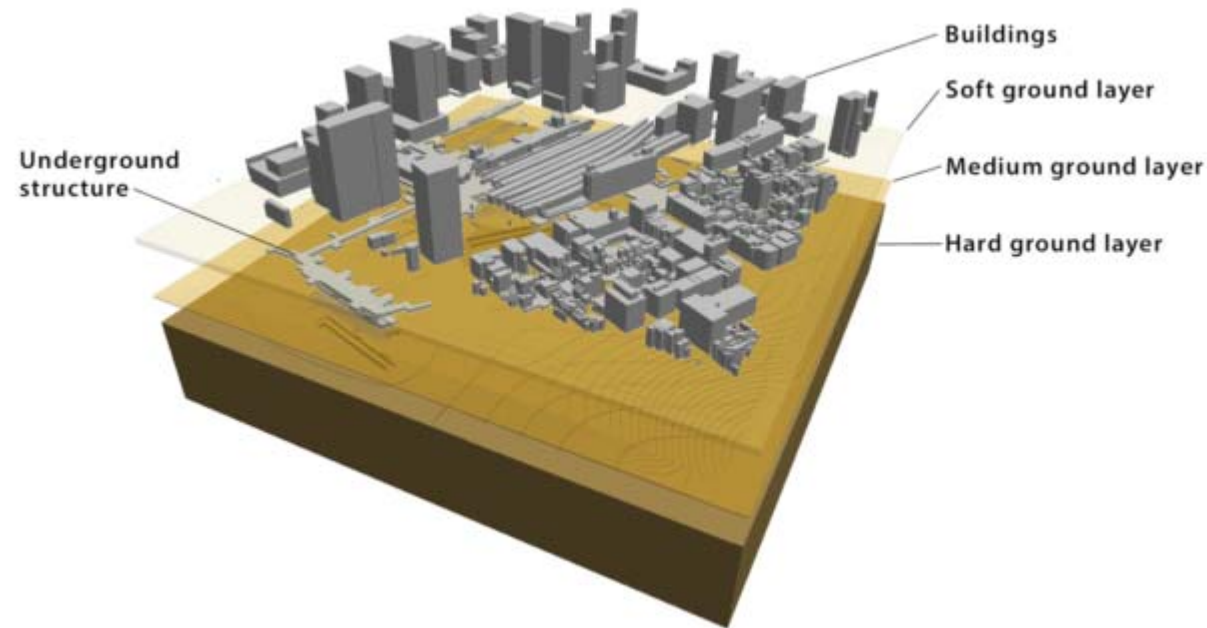
## ▪ Earthquake Simulation

- Using AI and transprecision computing
- [MOTHRA](#) (iMplicit sOlver wiTH artificial intelligence and tRAnsprecision computing).
- Running [Summit supercomputer](#) using a combination of AI and mixed-precision, MOTHRA achieved a 25x speed-up compared to the standard solver.

# Tensor Cores for Scientific Computing

## ■ Earthquake Simulation

- The simulation starts with 3D data of the various buildings in the city of Tokyo.



# Tensor Cores for Scientific Computing

## ▪ Earthquake Simulation













- The size of the domain and the physics involved requires a system of equations with 302 billion unknowns
- The researchers turned to AI to determine how to improve the effectiveness of their preconditioner by training a neural network on smaller models to help identify regions of slow convergence in their solver.
- [Use a combination FP64, FP32, FP21 and FP16](#)

# Tensor Cores for Scientific Computing

## ▪ Earthquake Simulation

- Using Tensor Core FP16 in Linear Algebra
- linear systems via LU factorization

achieved a 4x performance increase and 5x better energy efficiency versus the standard full FP64 implementation.

$Ax = b$			$=$	
$LUx = b$			$=$	
$Ly = b$			$=$	
then $Ux = y$			$=$	

# Content

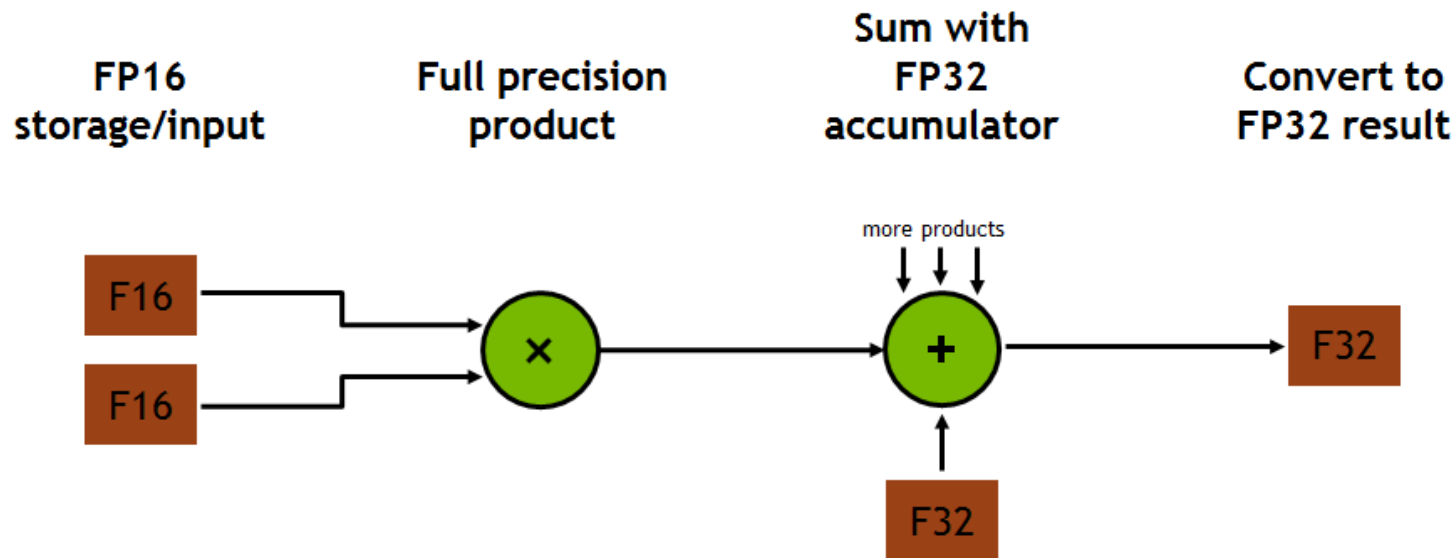
1. Mixed-Precision Programming

2. Tensor Cores

**3. Mixed-Precision Training of DNN**

# What is Mixed Precision Training?

- Reduced precision tensor math with:
  - FP32 accumulation, FP16 storage





# Benefits of Mixed Precision Training

- Accelerates math
  - TensorCores have 8x higher throughput than FP32
  - 125 Tflops theory
- Reduces memory bandwidth pressure:
  - FP16 halves the memory traffic compared to FP32
- Reduces memory consumption
  - Halve the size of activation and gradient tensors
  - Enables larger mini-batches or larger input sizes

# Considerations for Mixed Precision Training

- Which precision to use for storage, for math?
- Instructive to walk through by DNN operation type:
  - Weight update
  - Point-wise
  - Reduction
  - Convolution, Matrix multiply

# Guideline #1: weight update

- FP16 is sufficient for some networks, some require FP32
- Sum of FP16 values whose ratio is greater than  $2^{11}$  is just the larger value
  - FP16 has a 10-bit mantissa, binary points have to be aligned for addition
  - Weight update: if  $w \gg lr * dw$  then update doesn't change  $w$
- Conservative recommendation:
  - FP32 update:
    - Compute weight update in FP32
    - Keep a master copy of weights in FP32, make an FP16 copy for fwd/bwd passes
- If FP32 storage is a burden, try FP16 –it does work for some nets
  - i.e. convnets

## Guideline #2: pointwise

- FP16 is safe for most of these: ReLU, Sigmoid, Tanh, Scale, Add, ...
  - Inputs and outputs to these are value in a narrow range around 0
  - FP16 storage saves bandwidth -> reduces time
- FP32 math and storage is recommended for:
  - operations  $f$  where  $|f(x)| \gg |x|$ 
    - Examples: Exp, Square, Log, Cross-entropy
  - These typically occur as part of a normalization or loss layer that is unfused
  - FP32 ensures high precision, no perf impact since bandwidth limited
- Conservative recommendation :
  - Leave pointwise ops in FP32 (math and storage) unless they are known types
  - Pointwise op fusion is a good next step for performance
    - Use libraries for efficient fused pointwise ops for common layers (eg BatchNorm)

# Guideline #3: Reductions

- Examples:
  - Large sums of values: L1 norm, L2 norm, Softmax
- FP32 Math:
  - Avoids overflows
  - Does not affect speed –these operations are memory limited
- Storage:
  - FP32 output
  - Input can be FP16 if the preceding operation outputs FP16
    - If training frameworks supports different input and output types for an op
    - Saves bandwidth -> some speedup

# Guideline #4: Normalization and Loss

## ■ Normalizations:

- Usually constructed from primitive ops (reductions, squares, exp, scale)
- Storage:
  - Input and normalized output can be in FP16
  - Intermediate results should be stored in FP32
- Ideally should be fused into a single op:
  - Avoids round-trips to memory -> faster
  - Avoids intermediate storage

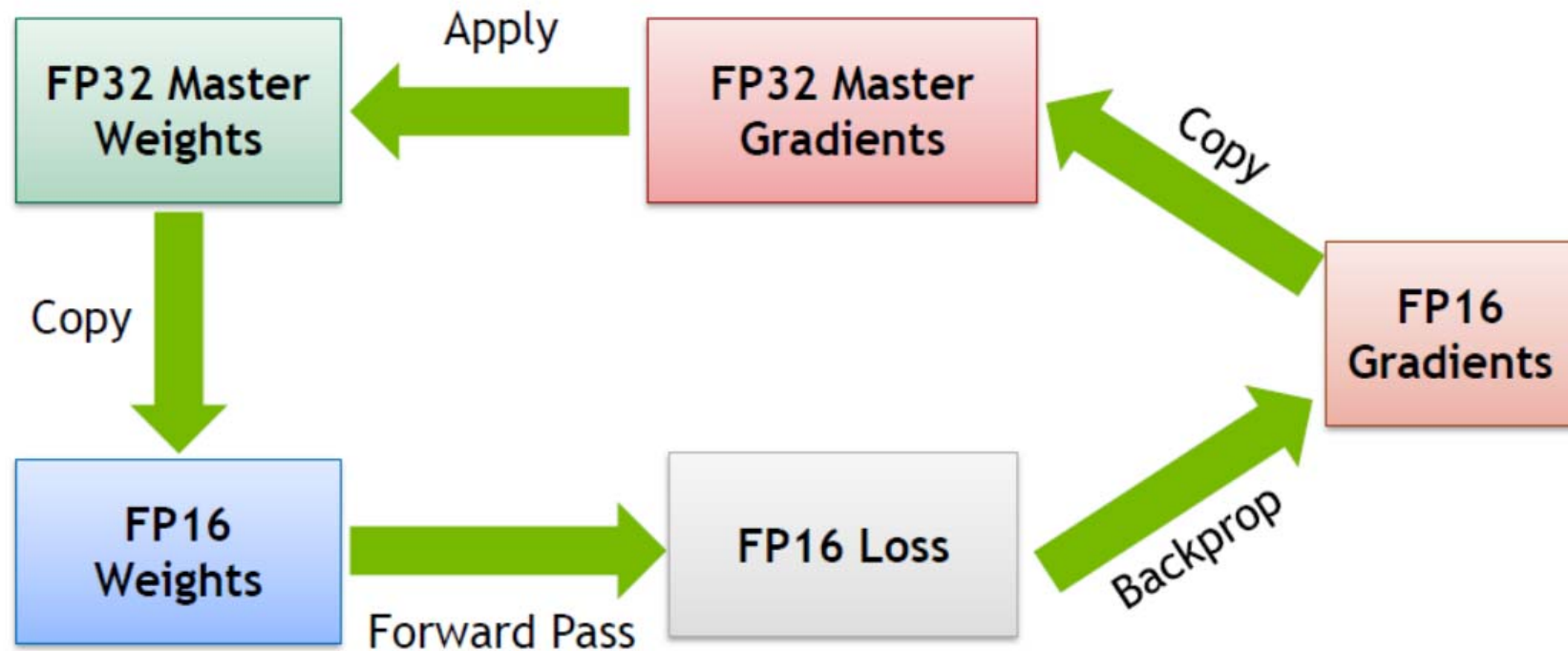
## ■ Loss, probability layers:

- Softmax, cross-entropy, attention modules
- FP32 math, FP32 output

# Guideline #5: Convolution, Matrix Multiply

- Fundamentally collections of dot-products
- Math: Tensor Cores starting with Volta GPUs
  - Training: use FP32 accumulation
  - Inference: FP16 accumulation can be used
- FP16 Storage (input and output)

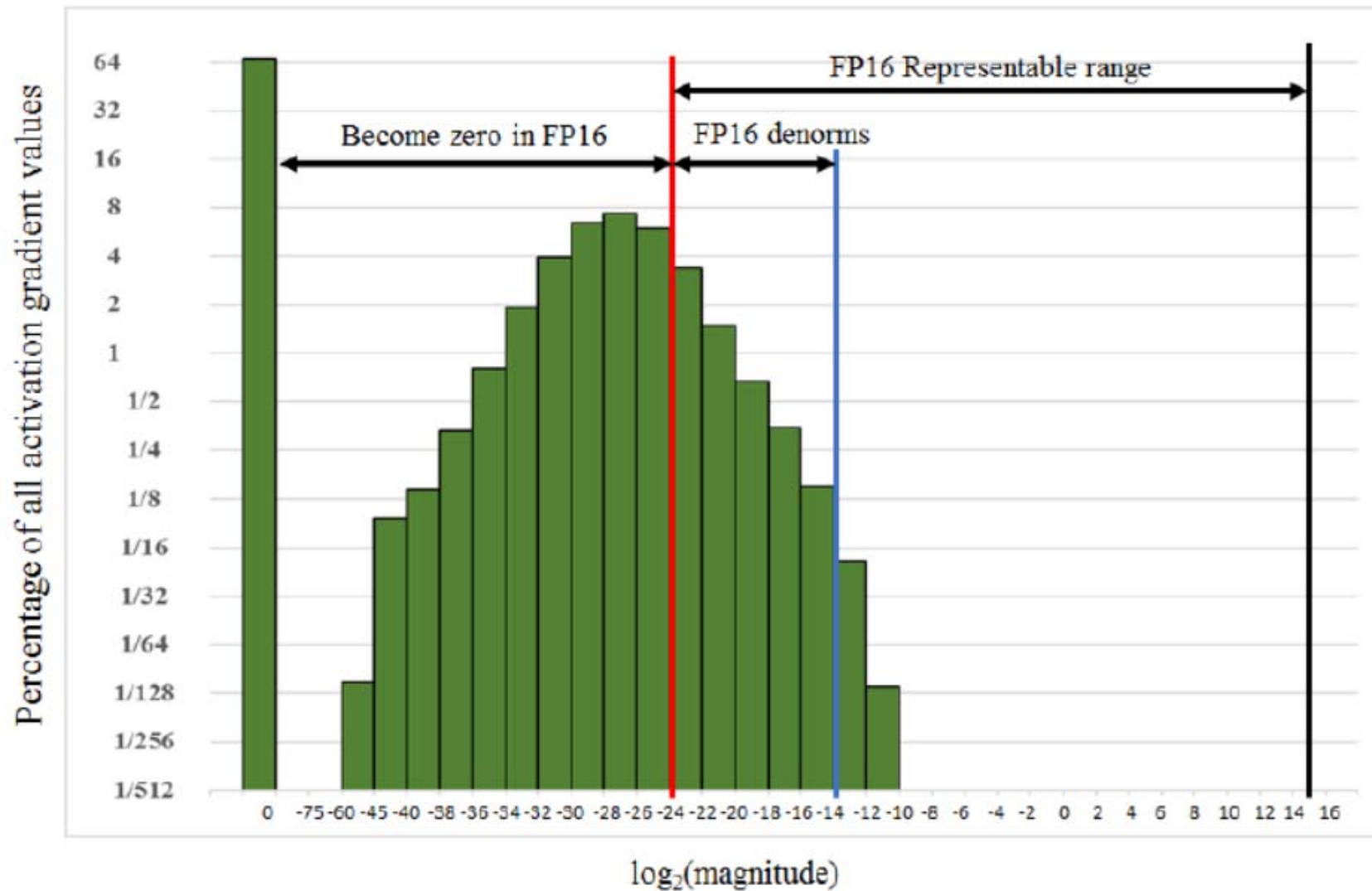
# Overview of Mixed Precision Training



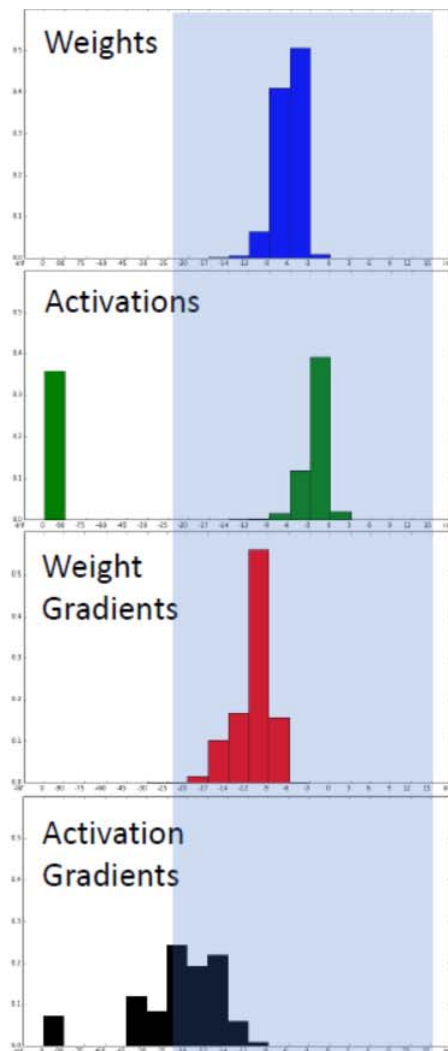
- One more precision consideration: [Loss Scaling](#)



# Gradient range offset

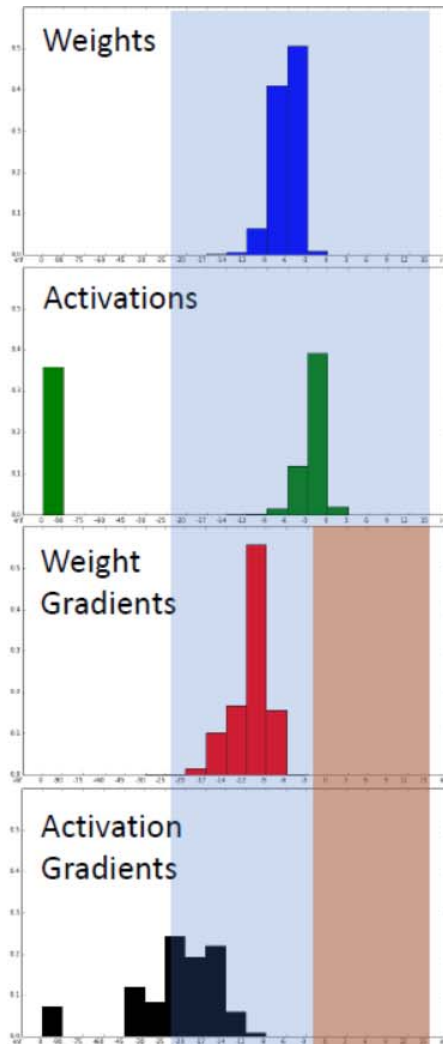


# Loss Scaling



Range representable in FP16: ~40 powers of 2

# Loss Scaling

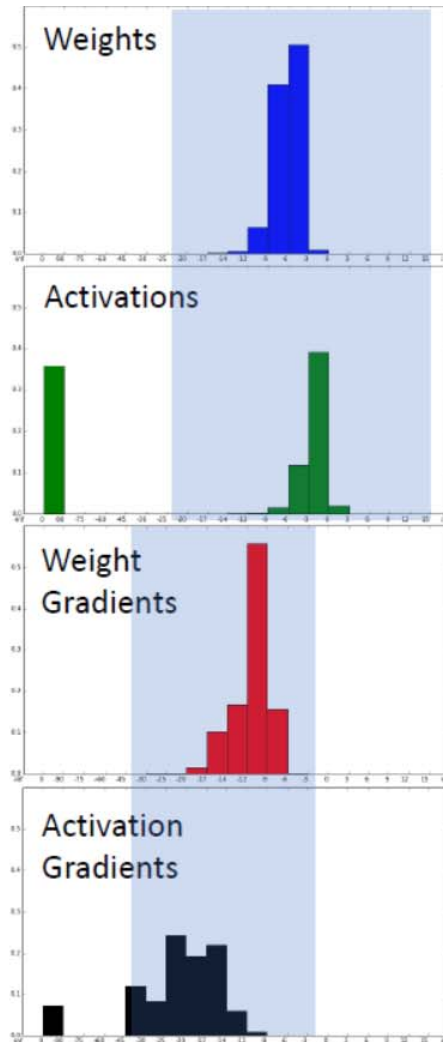


Range representable in FP16: ~40 powers of 2

Gradients are small, don't use much of FP16 range

FP16 range not used by gradients: ~15 powers of 2

# Loss Scaling



Range representable in FP16: ~40 powers of 2

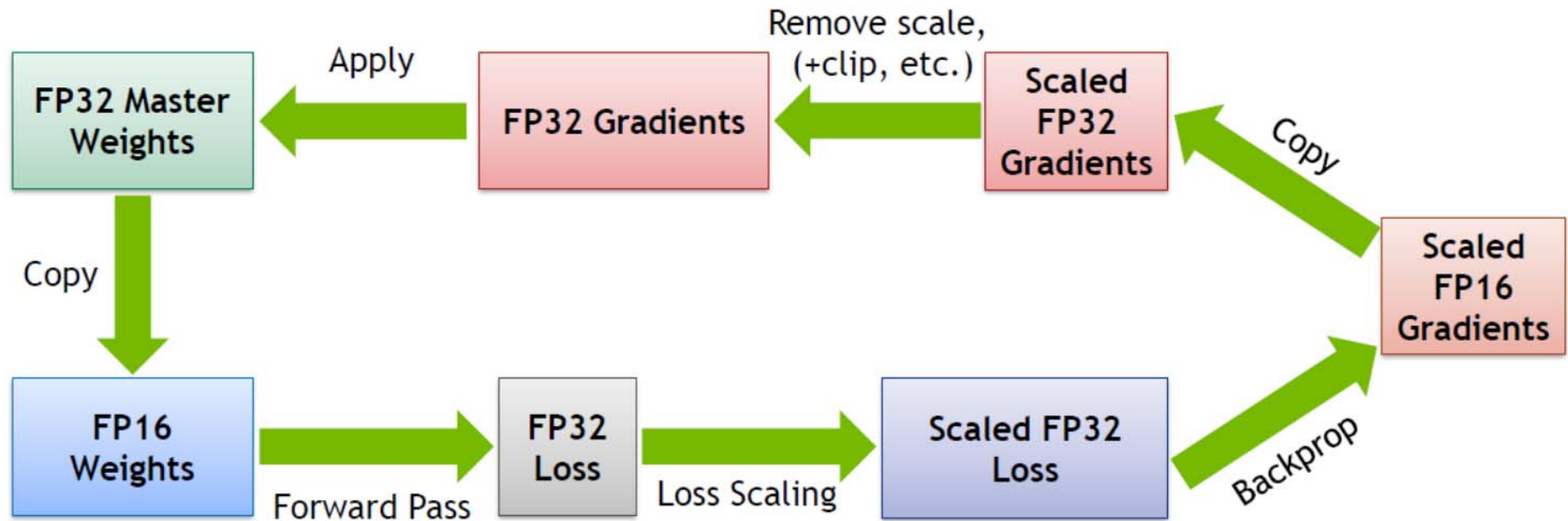
Gradients are small, don't use much of FP16 range

FP16 range not used by gradients: ~15 powers of 2

## Loss Scaling:

- multiply the loss by some constant  $s$
- by chain rule backprop scales gradients by  $s$
- preserves small gradient values
- unscale the weight gradient before update

# Overview of Mixed Precision Training



- Mixed precision consideration training with [Loss Scaling](#)

# Automatic Loss Scaling

- **Frees users from choosing a scaling factor**
  - Too small a factor doesn't retain enough small values
  - Too large a factor causes overflows
- **Algorithm**
  - Start with a large scaling factor  $s$
  - for each training iteration
    - Make an fp16 copy of weights
    - Fwd prop
    - Scale the loss by  $s$
    - Bwd prop
    - Update scaling factor  $s$ 
      - If  $dW$  contains Inf/NaN then reduce  $s$ , skip the update
      - If no Inf/NaN were detected for  $N$  updates then increase  $s$
    - Scale  $dW$  by  $1/s$
    - Update  $W$

} The automatic part

# ILSVRC12 Classification Networks

	FP32 Baseline	Mixed Precision
AlexNet	56.8%	56.9%
VGG-D	65.4%	65.4%
GoogLeNet	68.3%	68.4%
Inception v2	70.0%	70.0%
Inception v3	73.9%	74.1%
Resnet 50	75.9%	76.0%
ResNeXt 50	77.3%	77.5%

# Detection Networks, mAP

	<b>FP32 Baseline</b>	<b>Mixed Precision</b>
Faster R-CNN, VOC 07 data	69.1%	69.7%
Multibox SSD, VOC 07+12 data	76.9%	77.1%



# Speech

- Baidu
  - 2 2D-conv layers, 3 GRU layers, 1D conv
  - Baidu internal datasets
- FP32 Baseline

	FP32 Baseline	Mixed Precision
English	2.20	1.99
Mandarin	15.82	15.01

**Character Error Rate (lower is better)**

# Speedups

- **Memory limited ops:** should see  $\sim 2x$  speedup
- **Math limited ops:** will vary based on arithmetic intensity
- Some examples, mixed precision vs FP32 on GV100:
  - Resnet50:  $\sim 3.3x$
  - DeepSpeech2:  $\sim 4.5x$
  - FairSeq:  $\sim 4.0x$
  - Sentiment prediction:  $\sim 4.0x$
- Speedups to increase further:
  - libraries are continuously optimized
  - TensorCore paths are being added to more operation varieties

# TensorCore Performance Guidance

- For **matrix multiplication**:
  - On FP16 inputs, all three dimensions (M, N, K) must be multiples of 8.
  - On INT8 inputs (Turing only), all three dimensions must be multiples of 16.
- For **convolution**:
  - On FP16 inputs, input and output channels must be multiples of 8.
  - On INT8 inputs (Turing only), input and output channels must be multiples of 16.

# TensorCore Performance Guidance

- For mixed precision training:
  1. Choose mini-batch to be a multiple of 8;
  2. Choose linear layer dimensions to be a multiple of 8;
  3. Choose convolution layer channel counts to be a multiple of 8;
  4. For classification problems, pad vocabulary to be a multiple of 8;
  5. For sequence problems, pad the sequence length to be a multiple of 8;

# FRAMEWORKS

- Most major deep learning frameworks support for FP16 storage and tensor core math:
  - PyTorch, TensorFlow, MXNet, ...
  - <https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/#framework>
- PyTorch for example:
  - using AMP (Automatic Mixed Precision), which enables mixed precision in only 3 lines of Python.
    - AMP is available through NVIDIA's Apex repository of mixed precision and distributed training tools.
  - Examples and model scripts
  - Manual Conversion

# FRAMEWORKS

- PyTorch for example:

Model	Speedup
<a href="#">NVIDIA Sentiment Analysis</a>	4.5X speedup
<a href="#">FAIRSeq</a>	3.5X speedup
<a href="#">GNMT</a>	2X speedup
<a href="#">ResNet-50</a>	2X speedup

# Conclusions

- Mixed precision training benefits:
  - Math, memory speedups
  - Larger minibatches, larger inputs
- Automatic Loss Scaling simplifies mixed precision training
- Mixed precision matches FP32 training accuracy for a variety of:
  - Tasks: classification, regression, generation
  - Problem domains: images, language translation, language modeling, speech
  - Network architectures: feed forward, recurrent
  - Optimizers: SGD, Adagrad, Adam
- Note on inference:
  - Can be purely FP16: storage and math (use library calls with FP16 accumulation)
- More details: <http://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/>

# References

- Training-Mixed-Precision-User-Guide,
  - <https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html>
- Programming Tensor Cores in CUDA 9,
  - [Jeremy Appleyard](#) and [Scott Yokim](#) | [October 17, 2017](#)
  - <https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/>
- Tensor Ops Made Easier in cuDNN,
  - [Scott Yokim](#) | [August 20, 2018](#)
  - <https://devblogs.nvidia.com/tensor-ops-made-easier-in-cudnn/>