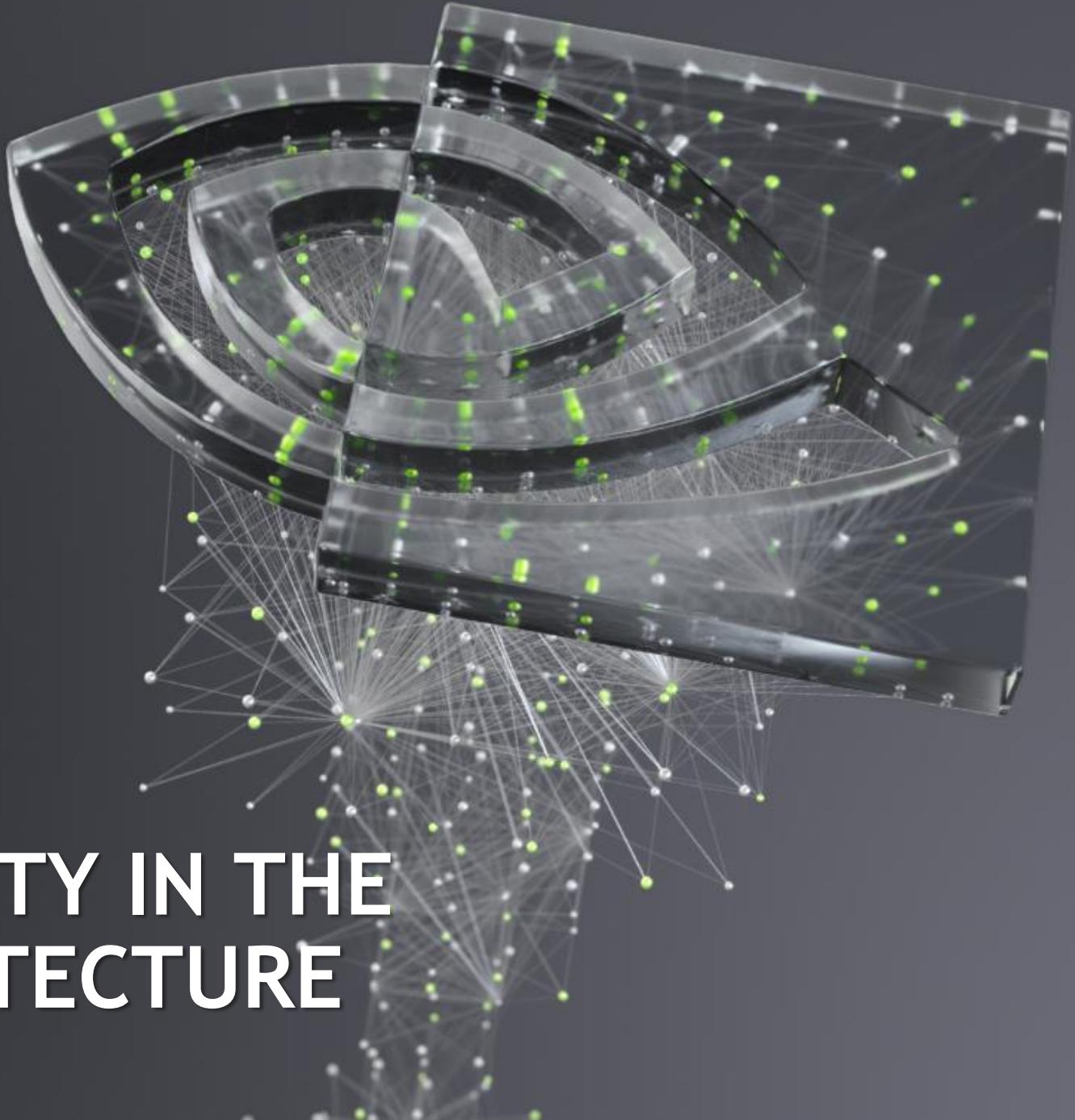




# ACCELERATING SPARSITY IN THE NVIDIA AMPERE ARCHITECTURE

Jeff Pool, Senior Architect





# OUTLINE

Sparsity Review

Motivation

Taxonomy

Challenges

---

NVIDIA A100 GPU 2:4 Sparsity

Sparsity pattern

Sparse Tensor Cores

Inference Speedups

---

Training Recipe

Recipe steps

Empirical evaluation

Implementation in frameworks

# SPARSITY - INFERENCE ACCELERATION VS TRAINING ACCELERATION

Focus of this talk is Inference acceleration

- Including training methods that enable accelerated inferencing with no loss of accuracy

Using sparsity to accelerate training is very interesting - but not the focus of this talk!

- At the end of the talk, we'll touch briefly on accelerating training

The background of the slide features a network graph with numerous small, semi-transparent green and white circular nodes connected by thin, light gray lines. The nodes are distributed across the frame, with a higher density in the upper left and lower right areas, creating a sense of a sparse but interconnected system.

# SPARSITY REVIEW

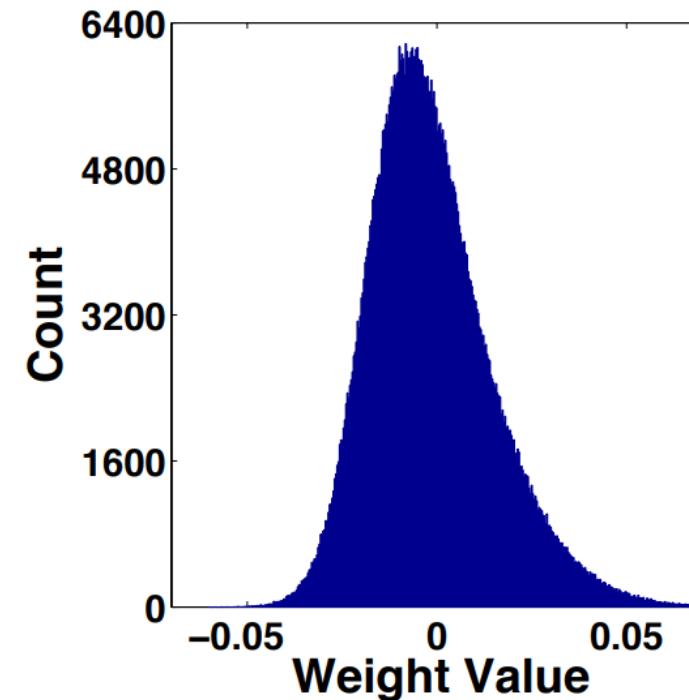
# SPARSITY: ONE OF MANY OPTIMIZATION TECHNIQUES

## Optimization goals for inference:

- Reduce network model size
- Speed up network model execution

## Observations that inspire sparsity investigations

- Biology: neurons are not densely connected
- Neural networks:
  - Trained model weights have many small-magnitude values
  - Activations may have 0s because of ReLU



# SPARSITY AND PERFORMANCE

**Do not store or process 0 values -> smaller and hopefully faster model**

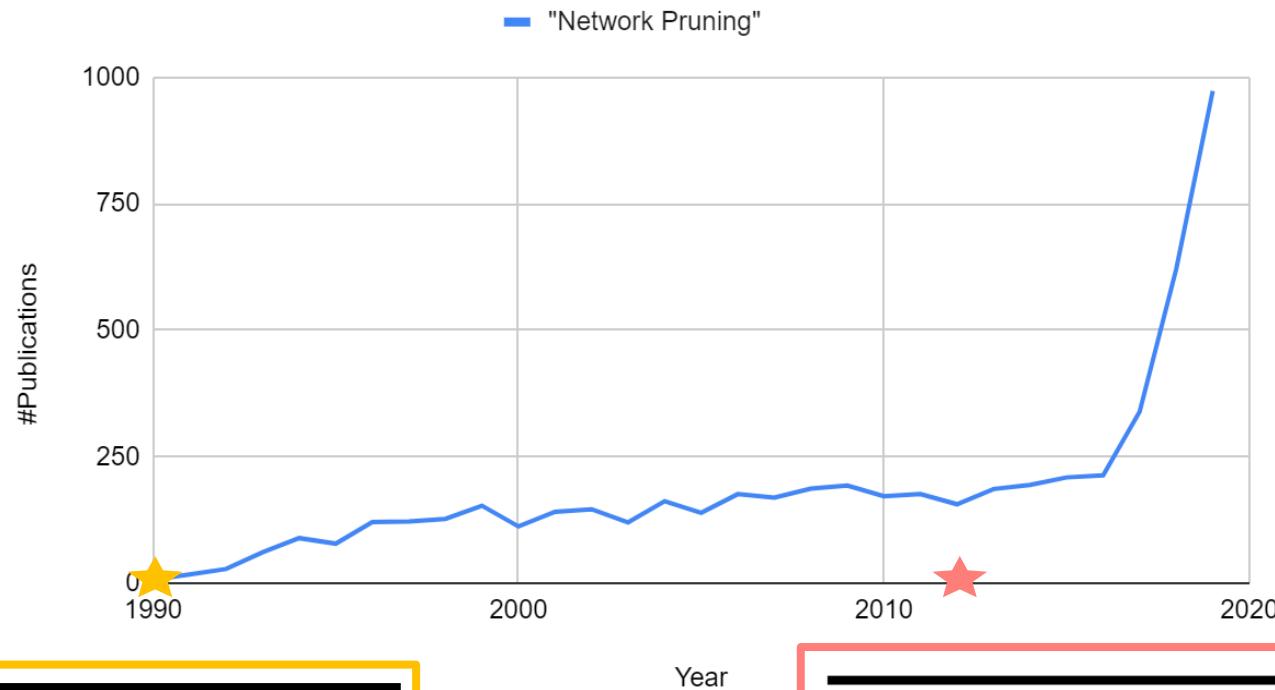
- Eliminate (prune) connections: set some weights to 0
- Eliminate (prune) neurons
- Etc.

**But, must also:**

- Maintain model accuracy
- Efficiently execute on hardware to gain speedup

# PRUNING/SPARSITY IS AN ACTIVE RESEARCH AREA

Publications per Year



*Optimal Brain Damage*

Yann Le Cun, John S. Denker and Sara A. Solla  
AT&T Bell Laboratories, Holmdel, N. J. 07733

**ImageNet Classification with Deep Convolutional Neural Networks**

Alex Krizhevsky  
University of Toronto  
kriz@cs.utoronto.ca

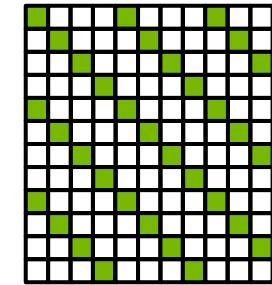
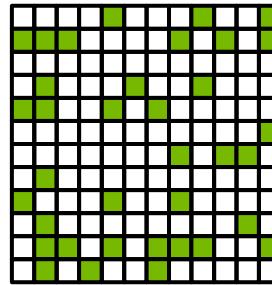
Ilya Sutskever  
University of Toronto  
ilya@cs.utoronto.ca

Geoffrey E. Hinton  
University of Toronto  
hinton@cs.utoronto.ca

# SPARSITY TAXONOMY

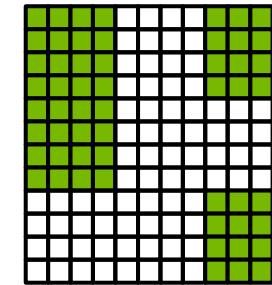
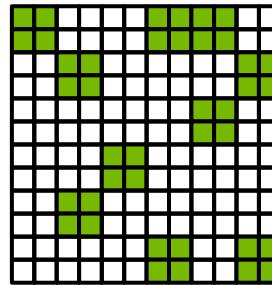
## Structure:

- Unstructured: irregular, no pattern of zeros
- Structured: regular, fixed set of patterns to choose from



## Granularity:

- Finest: prune individual values
- Coarser: prune blocks of values
- Coarsest: prune entire layers



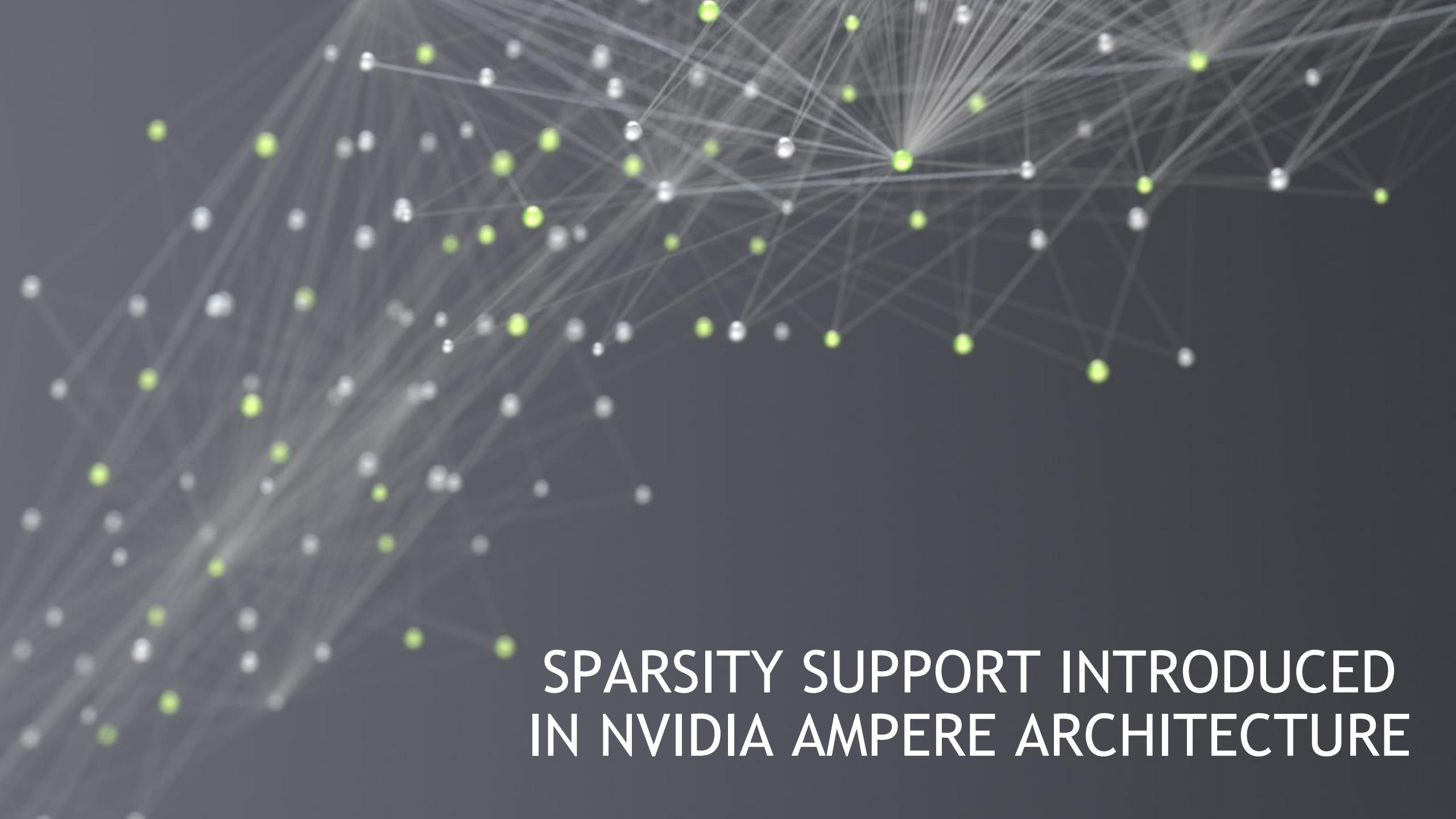
# STATE OF SPARSITY RESEARCH

Lots of research in two areas:

- High amounts (80-95%) unstructured, fine-grained sparsity
- Coarse-grained sparsity for simpler acceleration

Challenges not resolved for these approaches:

- Accuracy loss
  - High sparsity often leads to accuracy loss of a few percentage points, even after advanced training techniques
- Absence of a training approach that works across different tasks and networks
  - Training approaches to recover accuracy vary from network to network, often require hyper-parameter searches
- Lack of speedup
  - Math: unstructured data struggles to take advantage of modern vector/matrix math instructions
  - Memory access: unstructured data tends to poorly utilize memory buses, increases latency due to dependent sequences of reads
  - Storage overheads: metadata can consume 2x more storage than non-zero weights, undoing some of compression benefits



SPARSITY SUPPORT INTRODUCED  
IN NVIDIA AMPERE ARCHITECTURE

# SPARSITY IN A100 GPU

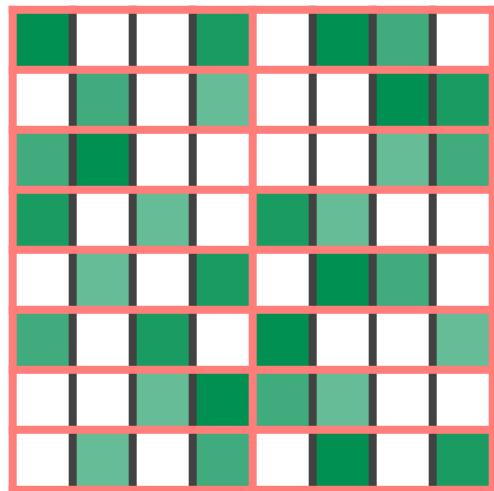
# Fine-grained structured sparsity for Tensor Cores

- 50% fine-grained sparsity
  - 2:4 pattern: 2 values out of each contiguous block of 4 must be 0

## Addresses the 3 challenges:

- **Accuracy:** maintains accuracy of the original, unpruned network
    - Medium sparsity level (50%), fine-grained
  - **Training:** a recipe shown to work across tasks and networks
  - **Speedup:**
    - Specialized Tensor Core support for sparse math
    - Structured: lends itself to efficient memory utilization

## 2:4 structured-sparse matrix



 = zero value

# SPARSE TENSOR CORES

**Applicable for:**

- Convolutions
- Matrix multiplies (linear layers, MLPs, recurrent cells, transformer blocks, etc.)

**Inputs: sparse weights, dense activations**

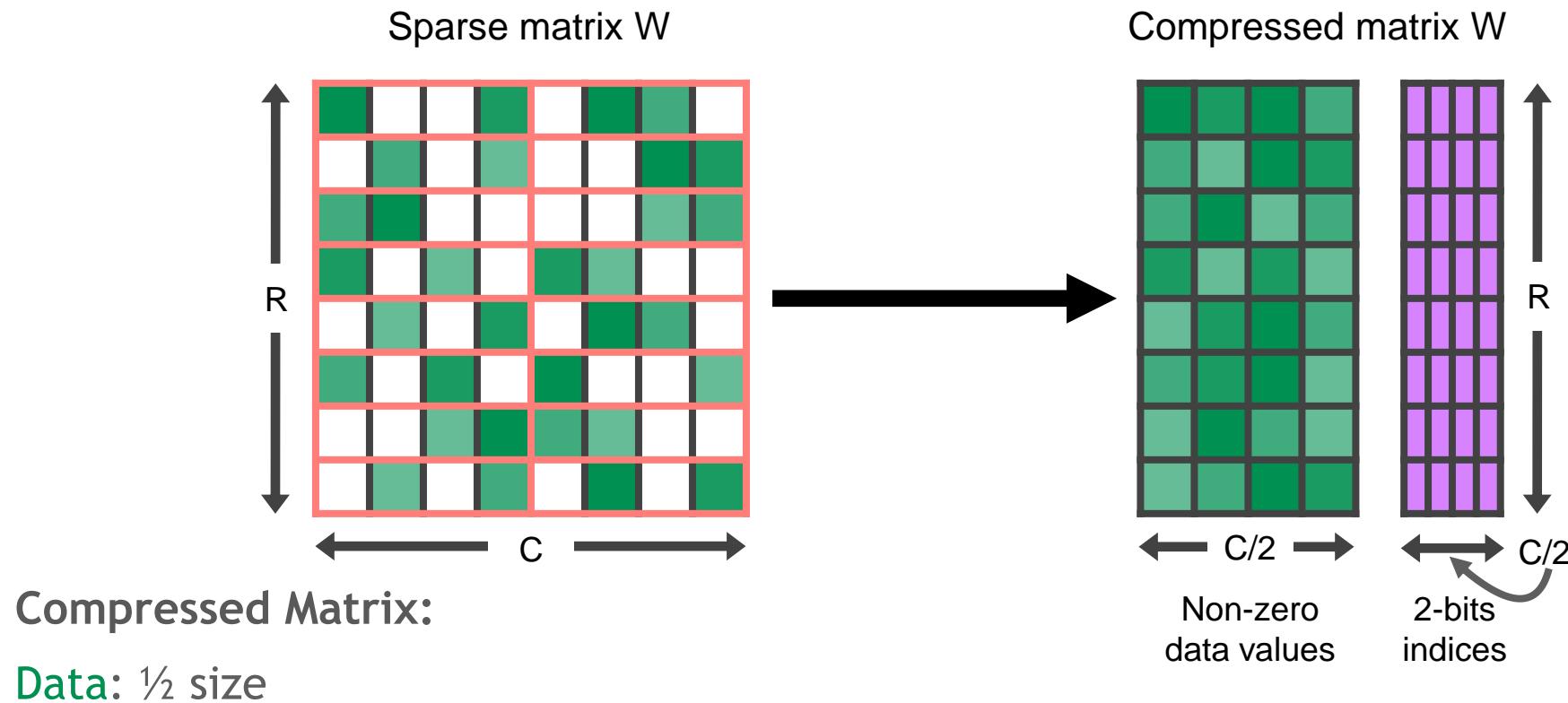
**Output: dense activations**

**Compressed format for the sparse matrix:**

- Do not store two 0s in each block of 4 values -> 50% of original storage
  - If a block contains more than two 0s, some of the 0s will be stored
- Metadata to index the remaining 2 values - needed for accessing the dense activations
  - 2 bits per value
  - 12.5% overhead for fp16, compared to 100-200% for CSR format

# 2:4 COMPRESSED MATRIX FORMAT

At most 2 non-zeros in every contiguous group of 4 values



16b data => 12.5% overhead

8b data => 25% overhead

# TENSOR CORE OPERATION

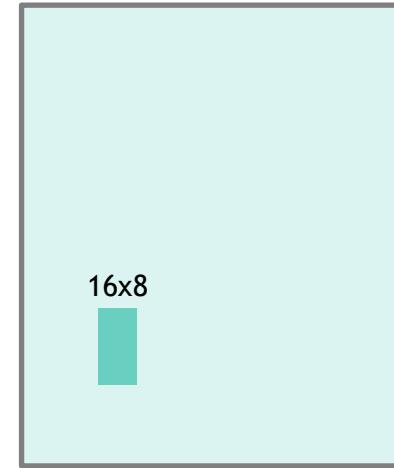
## Tiling a Large GEMM

Dense Tensor Cores (FP16)

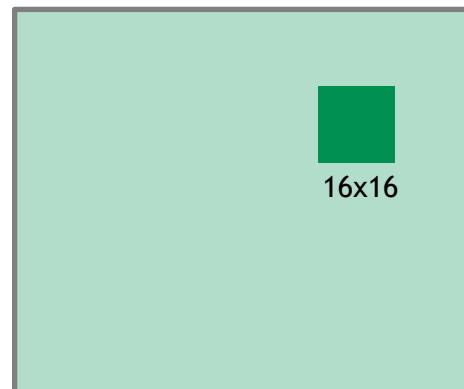
16x16 \* 16x8 matrix multiplication

Replicated and repeated to support large M, N, K

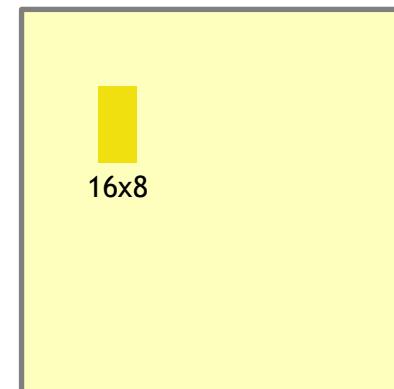
B: Dense, KxN



A: Dense, MxK



16x8



C: Dense, MxN

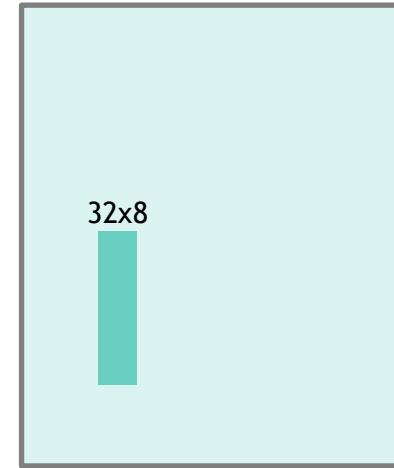
# TENSOR CORE OPERATION

Larger Tile = More Cycles

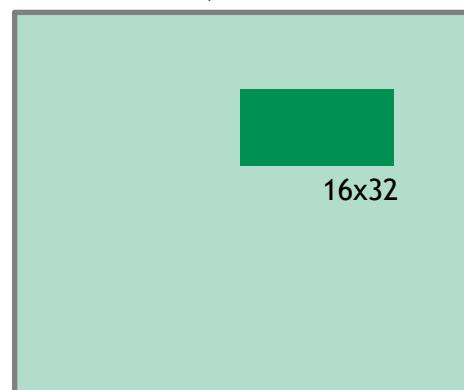
Dense Tensor Cores (FP16)

16x32 \* 32x8 matrix multiplication - 2 cycles

B: Dense, KxN

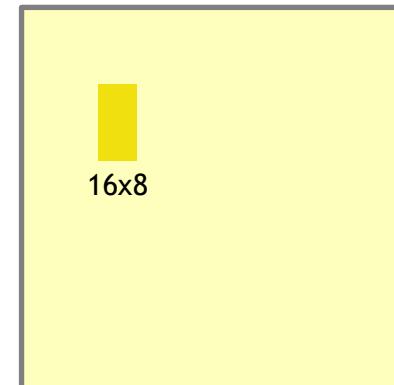


A: Dense, MxK



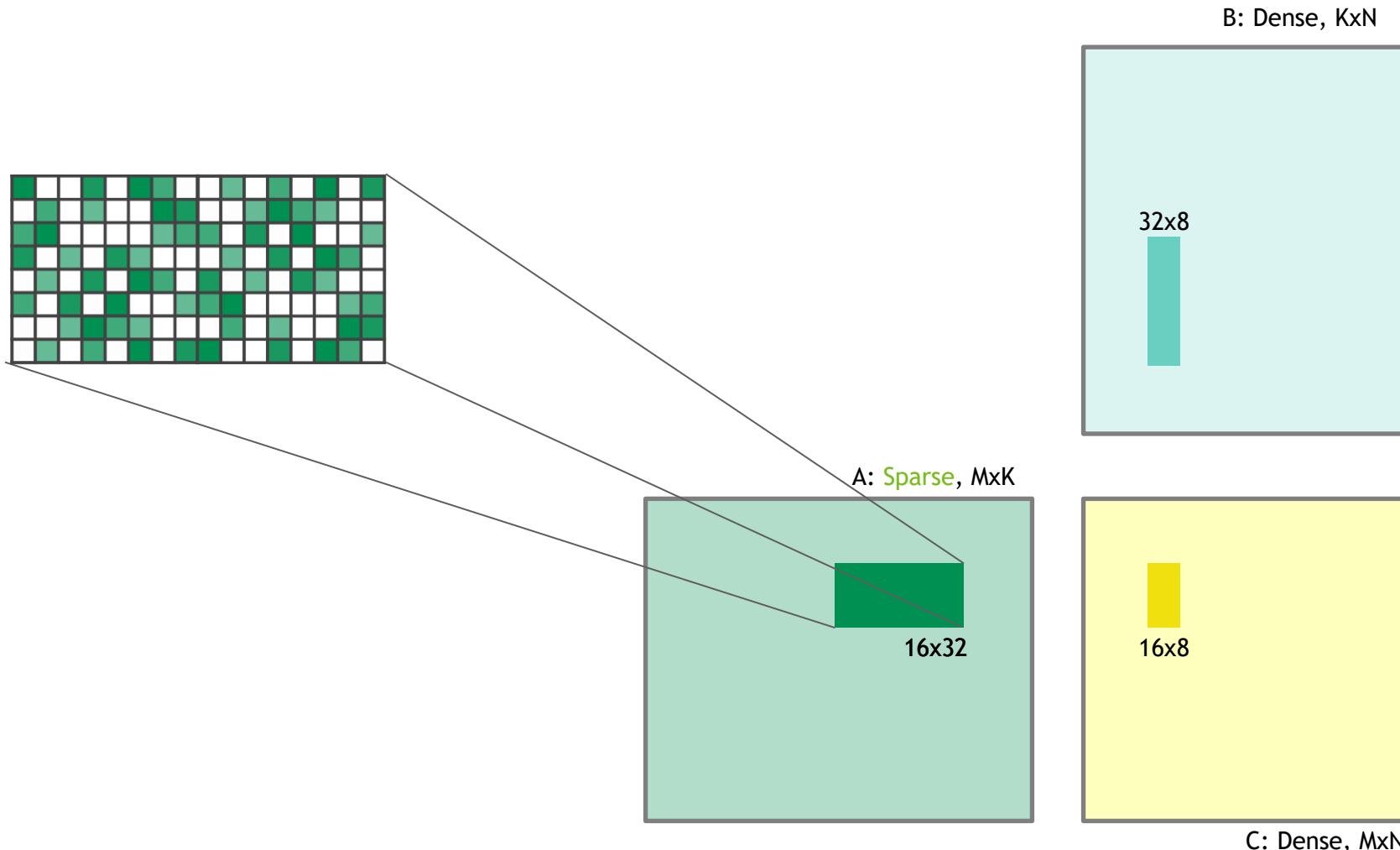
16x8

C: Dense, MxN



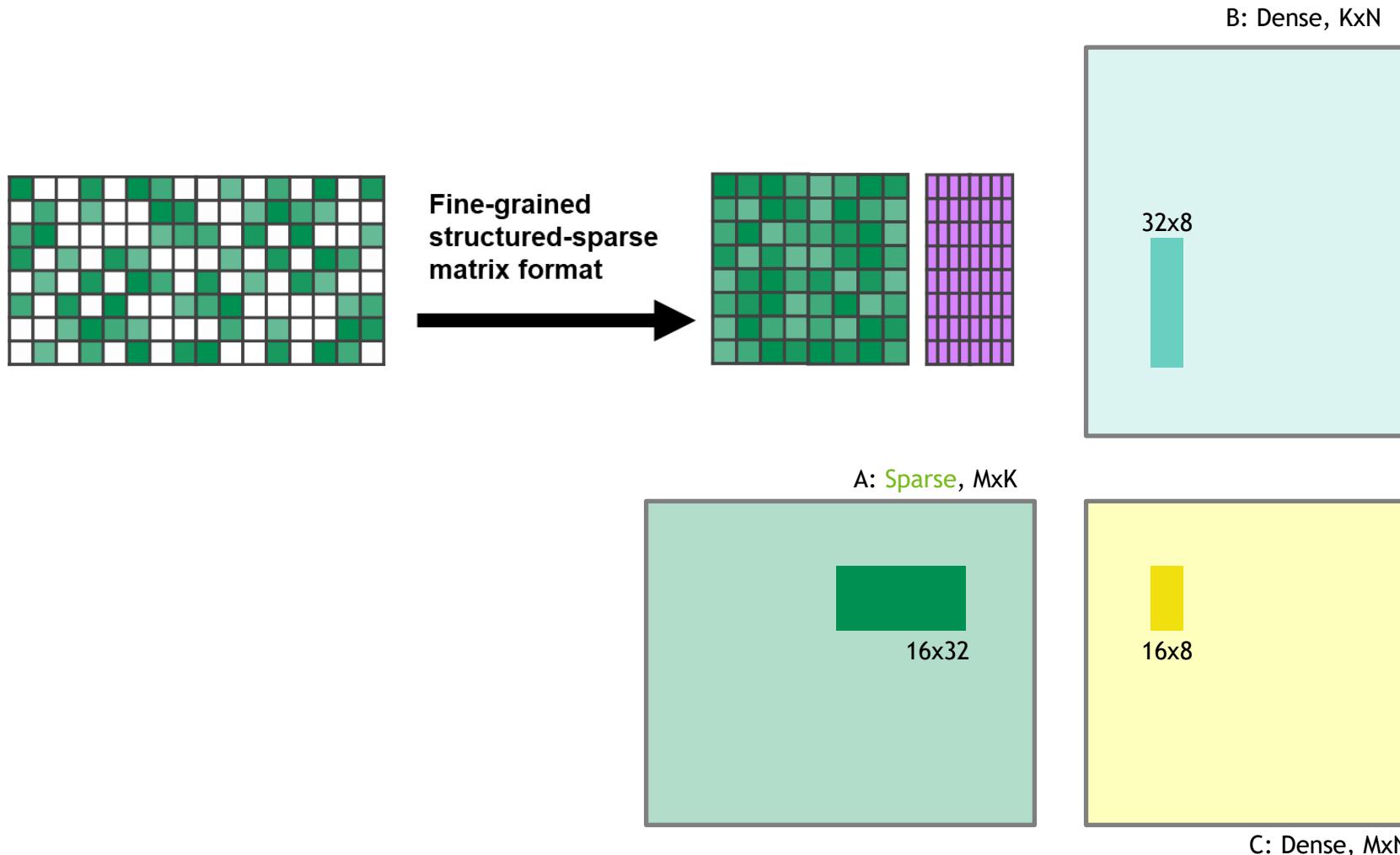
# TENSOR CORE OPERATION

Pruned Weight Matrix



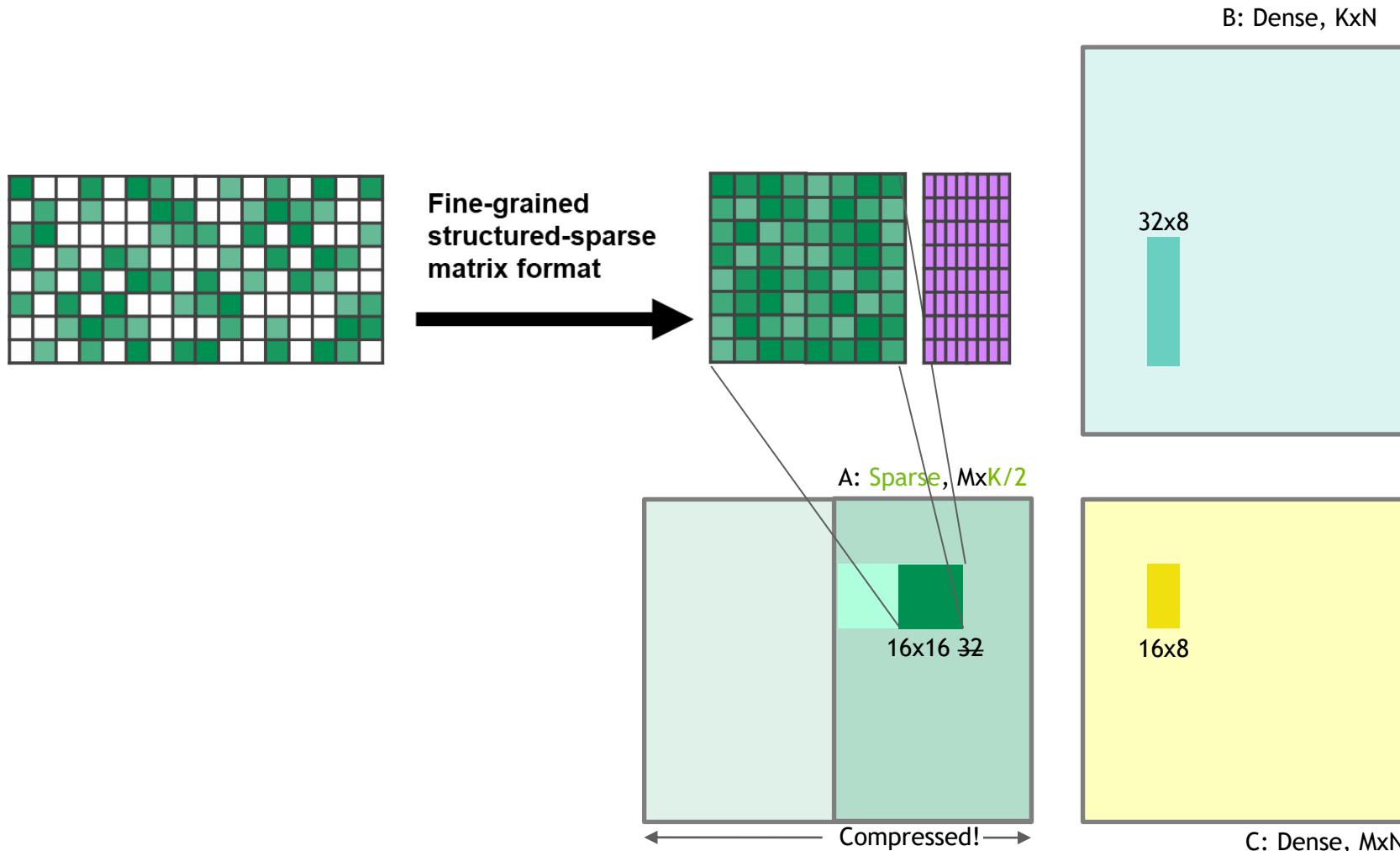
# TENSOR CORE OPERATION

Pruned and Compressed Weight Matrix



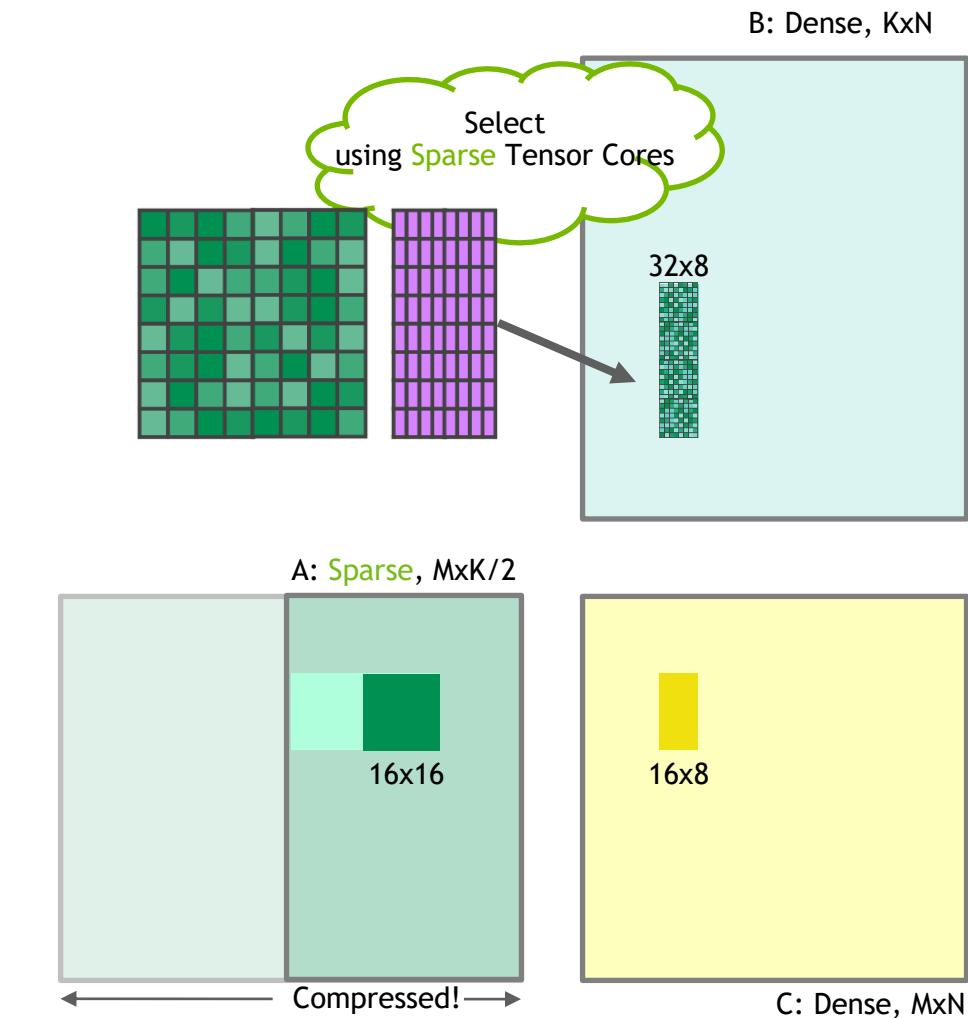
# TENSOR CORE OPERATION

## Tiling a Large, Sparse GEMM



# TENSOR CORE OPERATION

## Sparse Tensor Cores - Hardware Magic



# TENSOR CORE OPERATION

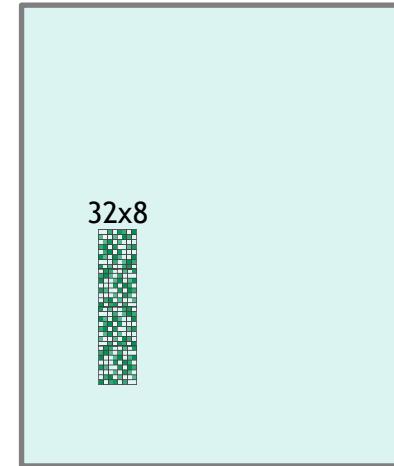
## Sparse Tensor Cores

Sparse Tensor Cores (FP16)

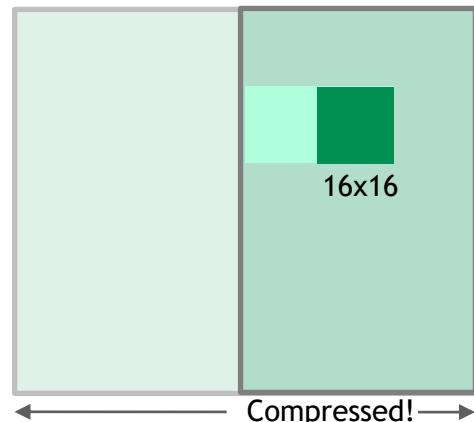
16x32 \* 32x8 effective matrix multiplication - 1 cycle

2x the work with the same instruction throughput

B: Dense, KxN

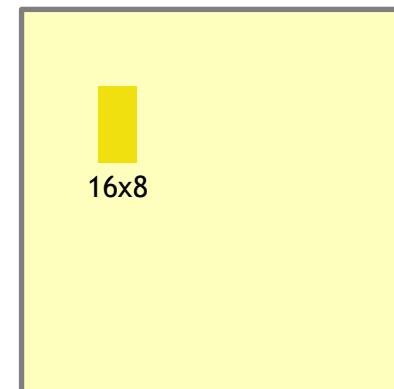


A: Sparse, MxK/2



16x8

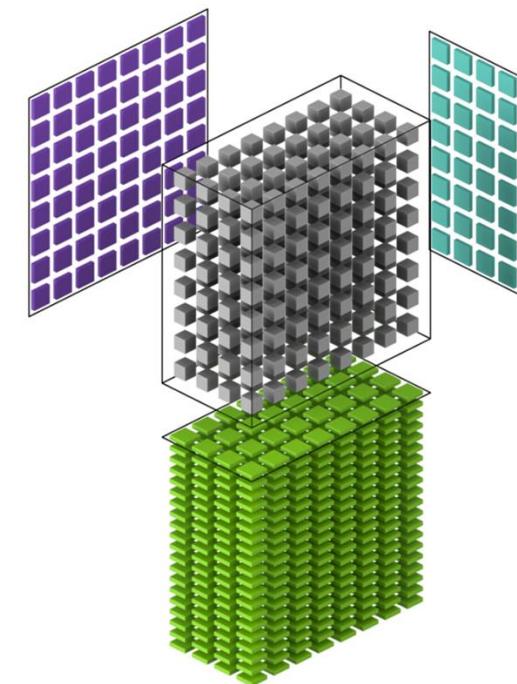
C: Dense, MxN



# TENSOR CORE MATH THROUHPUT

2x with Sparsity

INPUT OPERANDS	ACCUMULATOR	TOPS	Dense	Sparse
			vs. FFMA	Vs. FFMA
FP32	FP32	19.5	-	-
TF32	FP32	156	8X	16X
FP16	FP32	312	16X	32X
BF16	FP32	312	16X	32X
FP16	FP16	312	16X	32X
INT8	INT32	624	32X	64X
INT4	INT32	1248	64X	128X
BINARY	INT32	4992	256X	-



# SPARSE TENSOR CORES

Measured GEMM Performance with Current Software

M	N	K	Speedup
1024	8192	1024	1.44x
1024	16384	1024	1.73x
4096	8192	1024	1.53x
4096	16384	1024	1.78x

GEMM sizes selected from BERT-Large

# SPARSE TENSOR CORES

Measured Convolution Performance With Current Software

N	C	K	H,W	R,S	Speedup
32	1024	2048	14	1	1.52x
32	2048	1024	14	1	1.77x
32	2048	4096	7	1	1.64x
32	4096	2048	7	1	1.75x
256	256	512	7	3	1.85x

Kernel sizes selected from ResNeXt-101\_32x16d/ResNet-50

# NETWORK PERFORMANCE

## End to End Inference Speedup

NETWORK	DATA TYPE	SCENARIO	PERFORMANCE
BERT-Large	INT8	BS=256, SeqLen=128	6200 seq/s
		BS=1-256, SeqLen=128	1.3X-1.5X

# NETWORK PERFORMANCE

## End to End Inference Speedup

NETWORK	DATA TYPE	SCENARIO	PERFORMANCE
BERT-Large	INT8	BS=256, SeqLen=128	6200 seq/s
		BS=1-256, SeqLen=128	1.3X-1.5X
ResNeXt-101_32x16d	FP16	BS=256	2700 images/second
		BS=1-256	Up to 1.3X

# NETWORK PERFORMANCE

## End to End Inference Speedup

NETWORK	DATA TYPE	SCENARIO	PERFORMANCE
BERT-Large	INT8	BS=256, SeqLen=128	6200 seq/s
		BS=1-256, SeqLen=128	1.3X-1.5X
ResNeXt-101_32x16d	FP16	BS=256	2700 images/second
		BS=1-256	Up to 1.3X
	INT8	BS=256	4400 images/second
		BS=1-256	Up to 1.3X

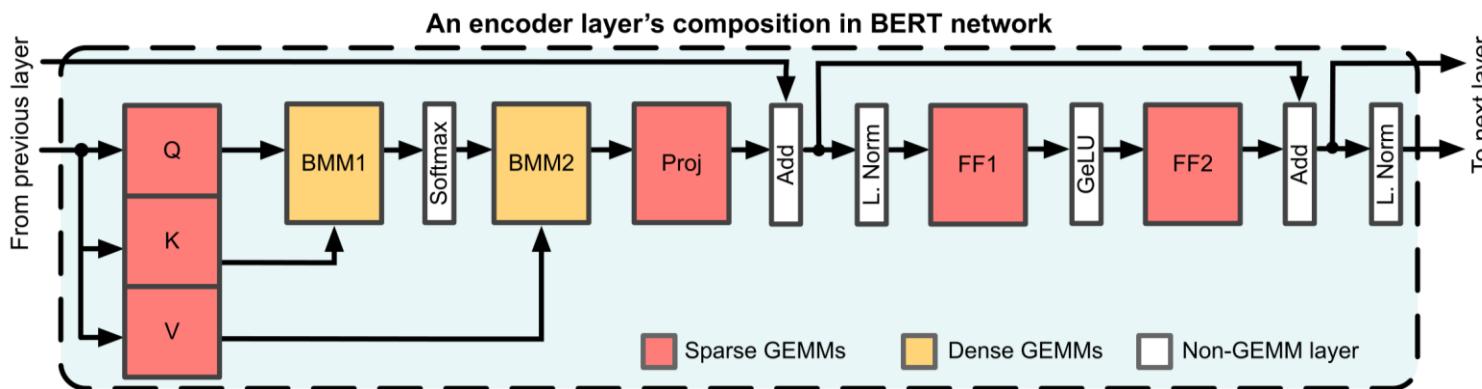
# NETWORK PERFORMANCE

## BERT-Large

1.8x GEMM Performance -> 1.5x Network Performance

Some operations remain dense:

Non-GEMM layers (Softmax, Residual add, Normalization, Activation functions, ...)  
GEMMs without weights to be pruned - Attention Batched Matrix Multiples

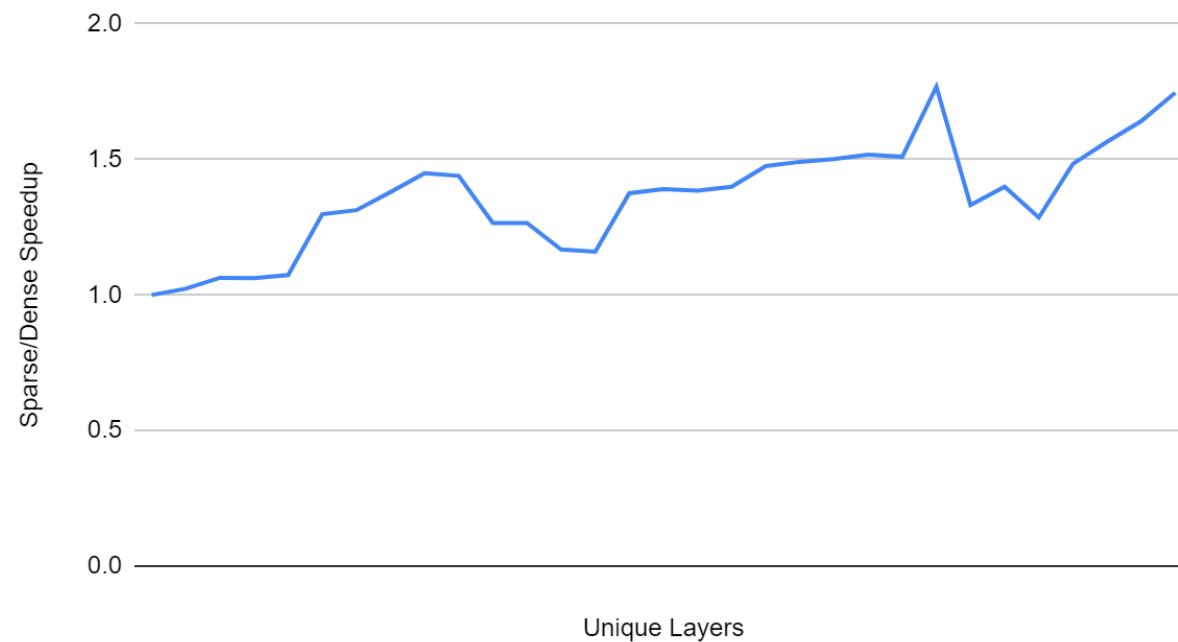


# CONVOLUTION SPEEDUPS

## Layers of ResNeXt-101

Some layers are less compute-limited than others

INT8 ResNeXt-101\_32x16d Convolutions





TRAINING RECIPE

# GOALS FOR A TRAINING RECIPE

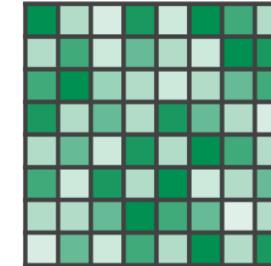
Maintains accuracy

Is applicable across various tasks, network architectures, and optimizers

Does not require hyper-parameter searches

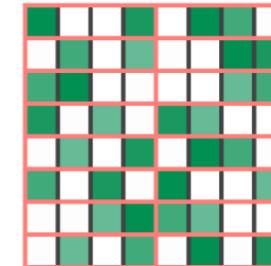
# RECIPE FOR 2:4 SPARSE NETWORK TRAINING

1) Train (or obtain) a dense network



Dense weights

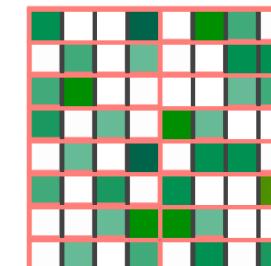
2) Prune for 2:4 sparsity



2:4 sparse weights

3) Repeat the original training procedure

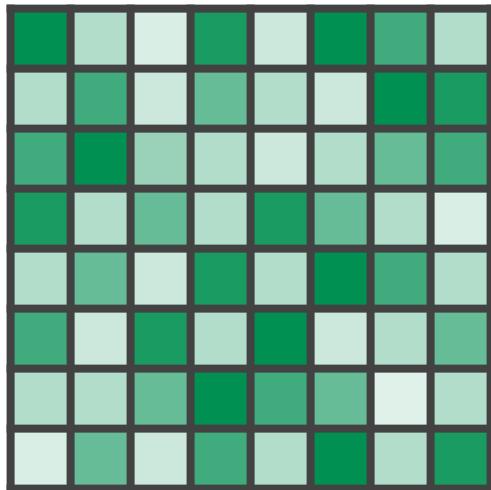
- Same hyper-parameters as in step-1
- Initialize to weights from step-2
- Maintain the 0 pattern from step-2: no need to recompute the mask



Retrained 2:4 sparse weights

# RECIPE STEP 2: PRUNE WEIGHTS

Dense matrix W



**Single-shot, magnitude-based pruning**

**For each 1x4 block of weights:**

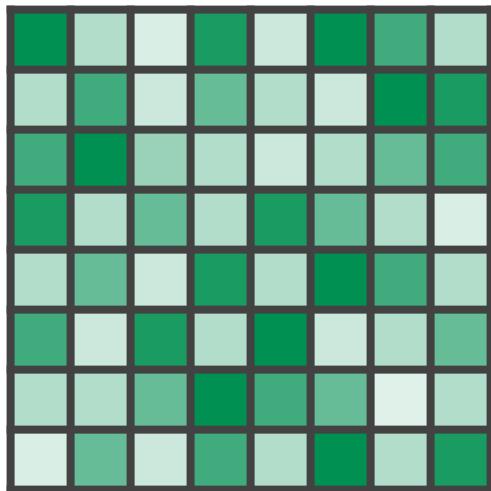
- Set 2 weights with the smallest magnitudes to 0

**Layer weights to prune: conv, linear**

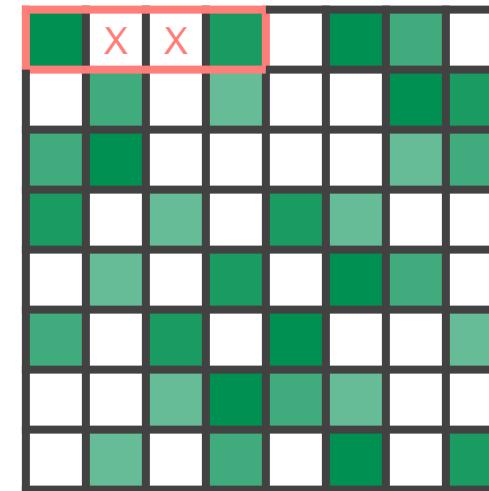
# RECIPE STEP 2: PRUNE WEIGHTS

At Most 2 Non-zeros in Every Contiguous Group of 4 Values

Dense matrix W



Structured-sparse matrix W



Fine-grained  
structured pruning

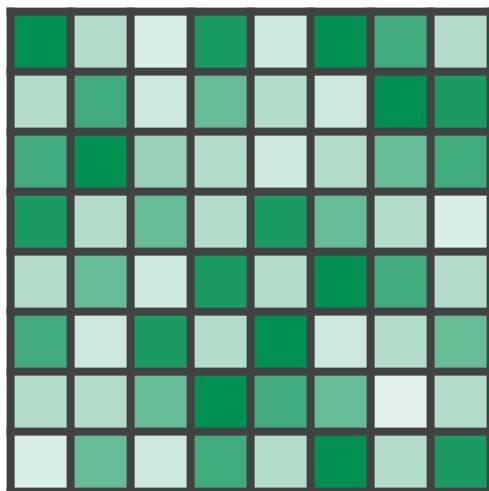
2:4 sparsity: 2 non-  
zero out of 4 entries

□ = zero value

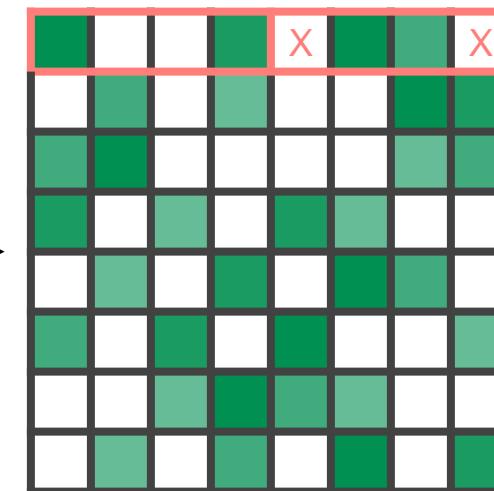
# RECIPE STEP 2: PRUNE WEIGHTS

At Most 2 Non-zeros in Every Contiguous Group of 4 Values

Dense matrix W



Structured-sparse matrix W



Fine-grained  
structured pruning

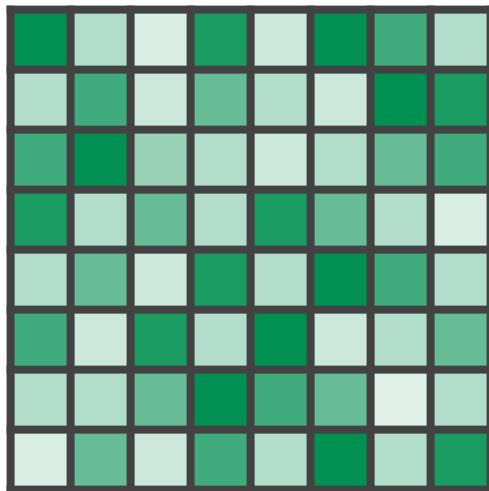
2:4 sparsity: 2 non-  
zero out of 4 entries

□ = zero value

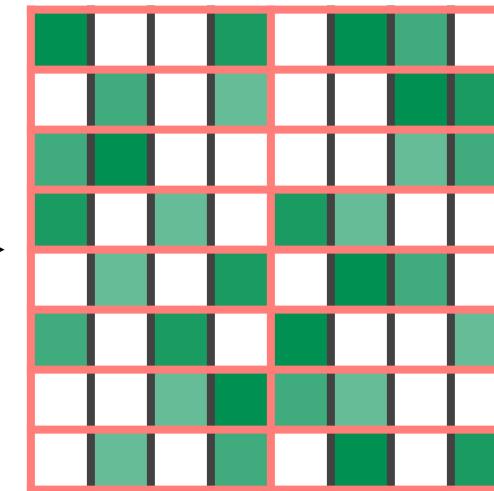
# RECIPE STEP 2: PRUNE WEIGHTS

At Most 2 Non-zeros in Every Contiguous Group of 4 Values

Dense matrix W



Structured-sparse matrix W



Fine-grained  
structured pruning

2:4 sparsity: 2 non-  
zero out of 4 entries

□ = zero value

# RECIPE STEP 3: RETRAIN

Pruning out 50% of the weight values reduces model accuracy

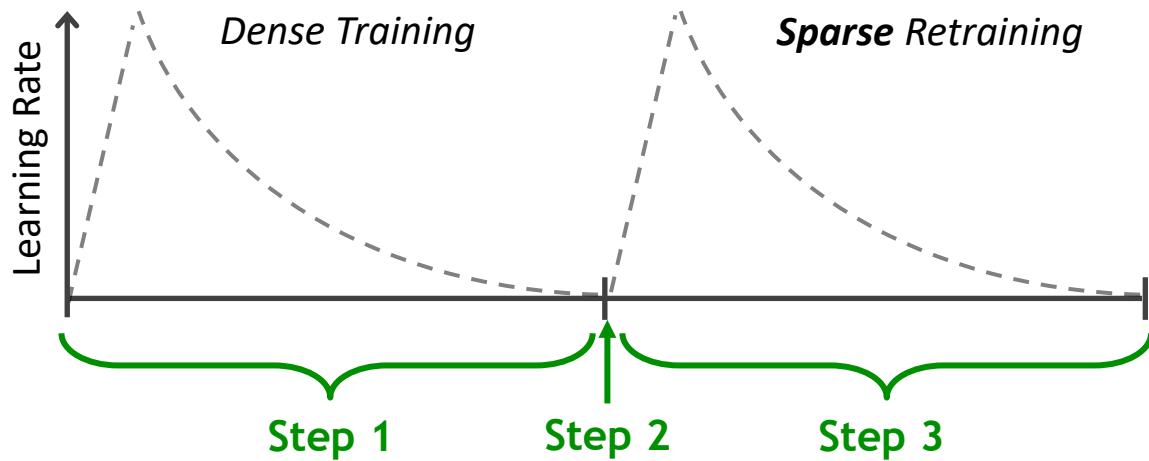
Retraining recovers accuracy

- Adjusts the remaining weights to compensate for pruning
- Requirement intuition:
  - Need **enough updates** by optimizer to compensate for pruning
  - Updates need **high-enough learning rates** to compensate

Simplest retraining:

- Repeat the training session, starting with weight values after pruning (as opposed to random initialization)
- All the same training hyper-parameters
- Do not update weights that were pruned out

# EXAMPLE LEARNING RATE SCHEDULE



# STEP 3 FOR NETWORKS TRAINED IN MULTIPLE PHASES

**Some networks are trained in multiple phases**

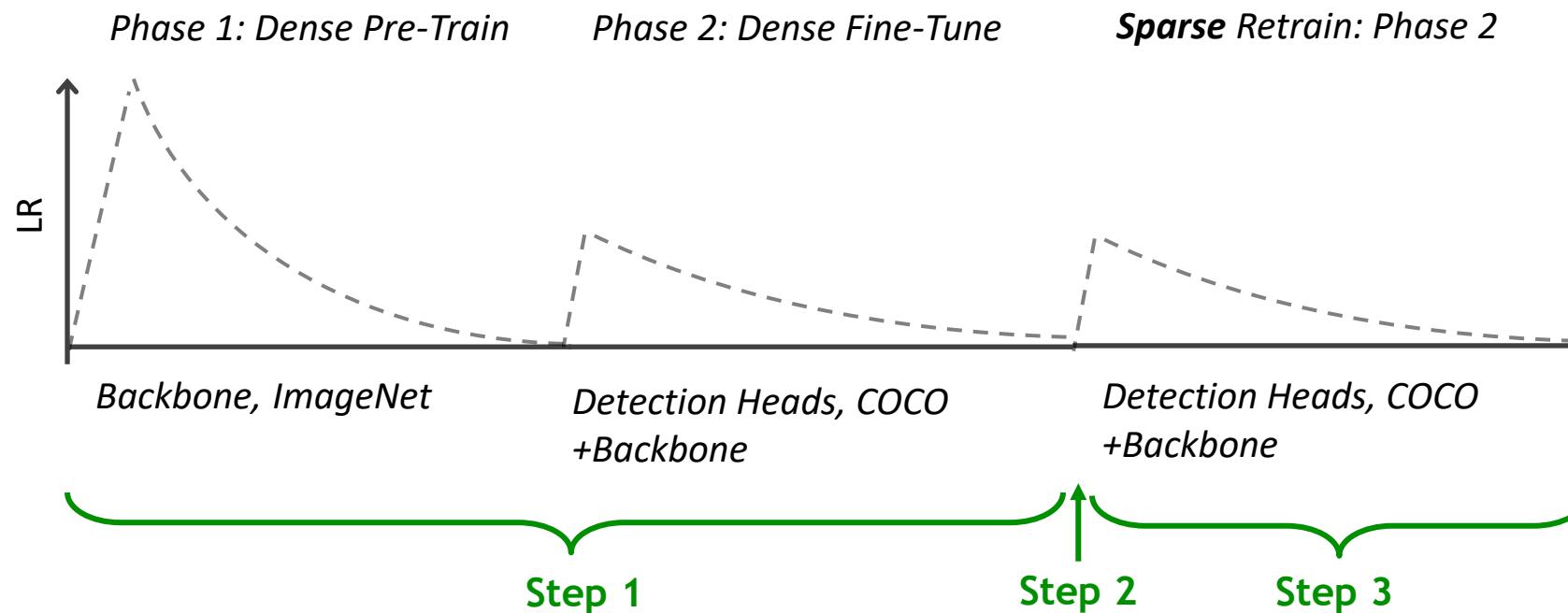
- Pretrain on one task and dataset, then train (fine-tune) on another task and dataset
- Examples:
  - Retinanet for object detection: 1) train for classification on ImageNet, 2) train for detection on COCO
  - BERT for question answering: 1) train for language modeling on BooksCorpus/Wikipedia, 2) train for question answering on SQuAD

**In some cases Step 3 can be applied to only the last phase of original training**

- Shortens retraining to recover accuracy
- Generally requires that the last phase(s):
  - Perform enough updates
  - Use datasets large enough to not cause overfitting
- When in doubt - retrain from the earliest phase, carry the sparsity through all the phases

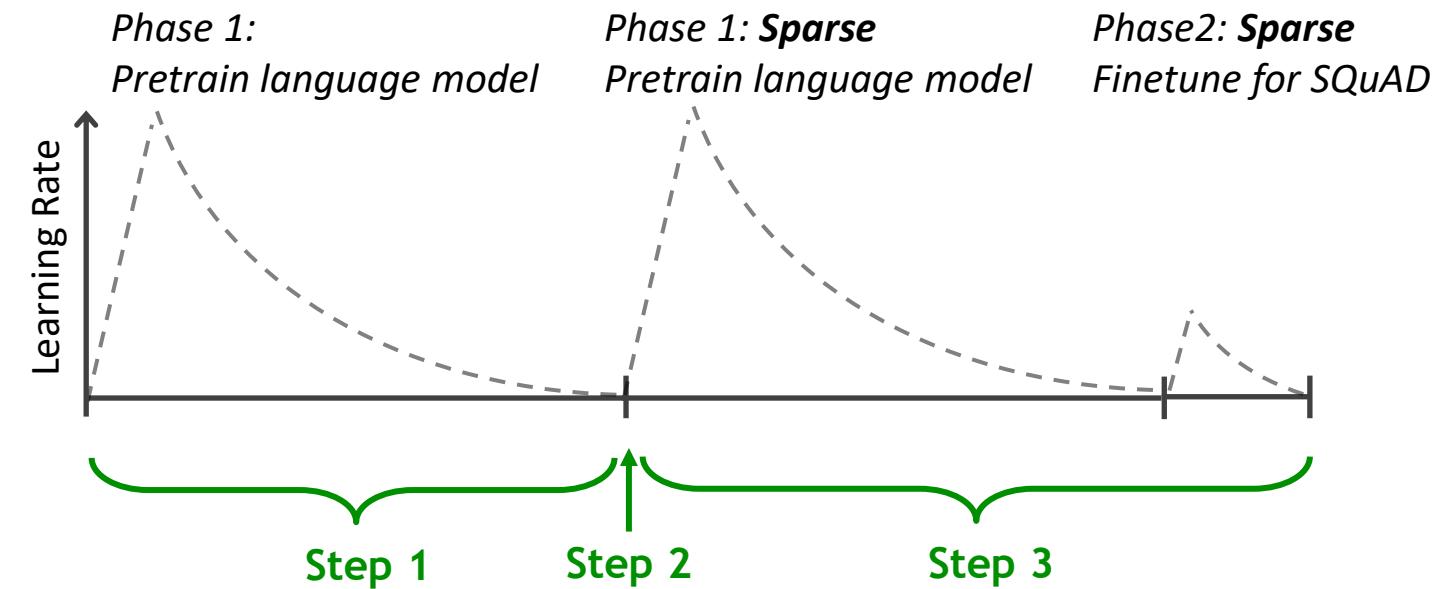
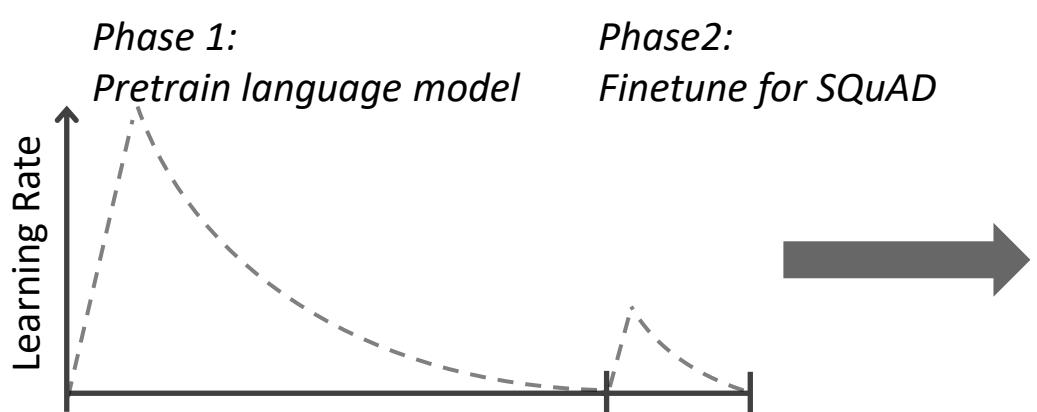
# STEP3: DETECTOR EXAMPLE

Detection Dataset is Large Enough to Provide Enough Updates and Not Overfit



# STEP3: BERT SQuAD EXAMPLE

Squad Dataset and Fine-tuning is Too Small to Compensate for Pruning on its Own



# SPARSITY AND QUANTIZATION

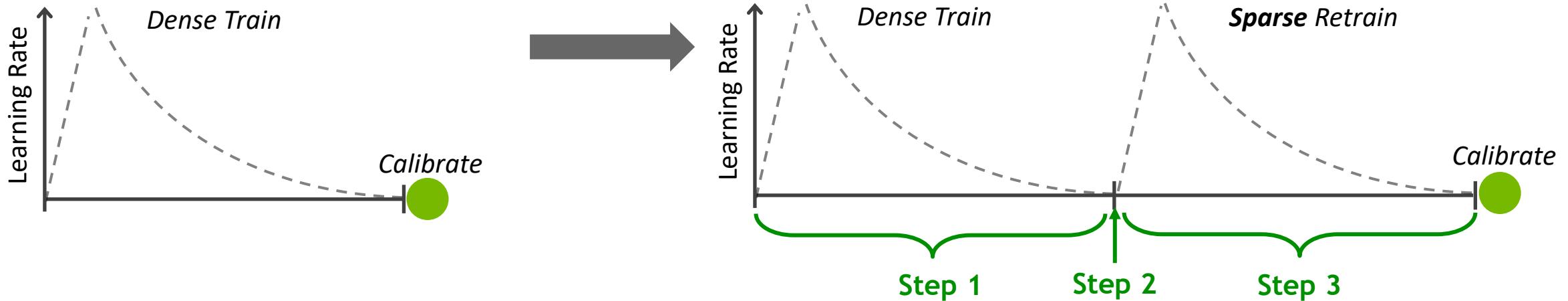
## Apply Sparsity Before Quantizing

- ▶ Quantization
- ▶ Generate a floating-point network
- ▶ Apply quantization (calibration, fine-tuning)
- ▶ Quantization+Sparsity
- ▶ Generate a floating-point network
- ▶ Prune
- ▶ Apply quantization (calibration, fine-tuning)

# SPARSITY AND QUANTIZATION

## Post-Training Quantization

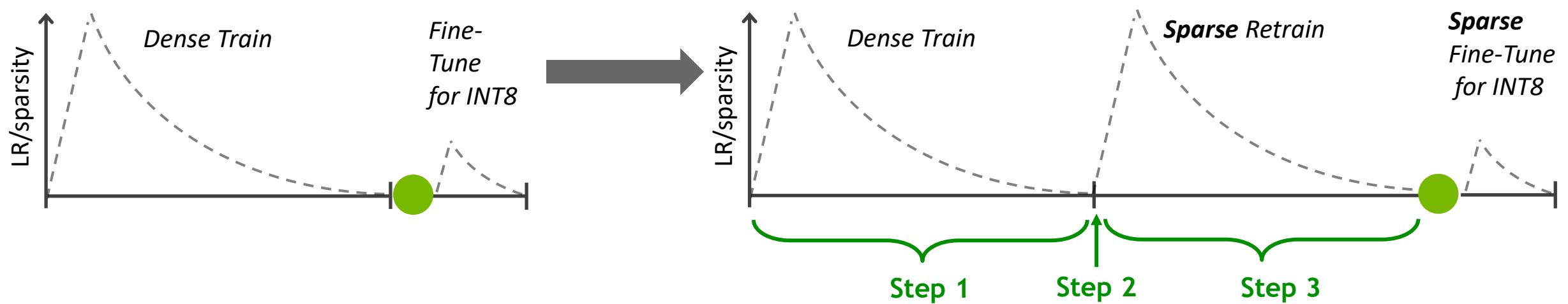
Post-training calibration follows the sparse fine-tuning



# SPARSITY AND QUANTIZATION

## Quantization Aware Training

Fine-tune for sparsity before fine-tuning for quantization



A network graph visualization featuring numerous nodes of two types: white and green. These nodes are interconnected by a dense web of thin, gray lines, representing relationships or connections between the entities. The graph is set against a dark, solid background.

ACCURACY EVALUATION

# ACCURACY

## Overview

Tested 34 networks, covering a variety of AI domains, with the described recipe

Run one test without sparsity and one test with sparsity, compare results

Results : accuracy is ~same (within prior observed run-to-run variation of networks)

FP16 networks trained with mixed precision training

INT8 networks generated by:

1<sup>st</sup>: Retrain a sparse FP16 network first

2<sup>nd</sup>: Apply traditional quantization techniques:

Post-training calibration

Quantization-Aware fine-tuning

# IMAGE CLASSIFICATION

ImageNet

Network	Accuracy				
	Dense FP16	Sparse FP16	Sparse INT8		
ResNet-34	73.7	73.9	0.2	73.7	-
ResNet-50	76.6	76.8	0.2	76.8	0.2
ResNet-101	77.7	78.0	0.3	77.9	-
ResNeXt-50-32x4d	77.6	77.7	0.1	77.7	-
ResNeXt-101-32x16d	79.7	79.9	0.2	79.9	0.2
DenseNet-121	75.5	75.3	-0.2	75.3	-0.2
DenseNet-161	78.8	78.8	-	78.9	0.1
Wide ResNet-50	78.5	78.6	0.1	78.5	-
Wide ResNet-101	78.9	79.2	0.3	79.1	0.2
Inception v3	77.1	77.1	-	77.1	-
Xception	79.2	79.2	-	79.2	-
VGG-16	74.0	74.1	0.1	74.1	0.1
VGG-19	75.0	75.0	-	75.0	-

# IMAGE CLASSIFICATION

ImageNet

Network	Accuracy			
	Dense FP16	Sparse FP16	Sparse INT8	
ResNet-50 (WSL)	81.1	80.9	-0.2	80.9 -0.2
ResNeXt-101-32x8d (WSL)	84.3	84.1	-0.2	83.9 -0.4
ResNeXt-101-32x16d (WSL)	84.2	84.0	-0.2	84.2 -
SUNet-7-128	76.4	76.5	0.1	76.3 -0.1
DRN-105	79.4	79.5	0.1	79.4 -

# SEGMENTATION/DETECTION

COCO 2017, bbox AP

Network	Accuracy				
	Dense FP16	Sparse FP16	Sparse INT8		
MaskRCNN-RN50	37.9	37.9	-	37.8	-0.1
SSD-RN50	24.8	24.8	-	24.9	0.1
FasterRCNN-RN50-FPN-1x	37.6	38.6	1.0	38.4	0.8
FasterRCNN-RN50-FPN-3x	39.8	39.9	-0.1	39.4	-0.4
FasterRCNN-RN101-FPN-3x	41.9	42.0	0.1	41.8	-0.1
MaskRCNN-RN50-FPN-1x	39.9	40.3	0.4	40.0	0.1
MaskRCNN-RN50-FPN-3x	40.6	40.7	0.1	40.4	0.2
MaskRCNN-RN101-FPN-3x	42.9	43.2	0.3	42.8	0.1
RetinaNet-RN50-FPN-1x	36.4	37.4	1.0	37.2	0.8
RPN-RN50-FPN-1x	45.8	45.6	-0.2	45.5	0.3

RN = ResNet Backbone

FPN = Feature Pyramid Network

RPN = Region Proposal Network

# NLP - TRANSLATION

EN-DE WMT'14

Network	Metric	Accuracy			
		Dense FP16	Sparse FP16	Sparse INT8	
GNMT	BLEU	24.6	24.9	0.3	24.9 0.3
FairSeq Transformer	BLEU	28.2	28.5	0.3	28.3 0.1
Levenstein Transformer	Validation Loss	6.16	6.18	-0.2	6.16 -

# NLP - LANGUAGE MODELING

Transformer-XL, BERT

Network	Task	Metric	Accuracy			
			Dense FP16	Sparse FP16	Sparse INT8	
Transformer-XL	enwik8	BPC	1.06	1.06	-	-
BERT-Base	SQuAD v1.1	F1	87.6	88.1	0.5	88.1 0.5
BERT-Large	SQuAD v1.1	F1	91.1	91.5	0.4	91.5 0.4

# COMPARING 2:4 TO OTHER ALTERNATIVES

Alternatives for 50% smaller models:

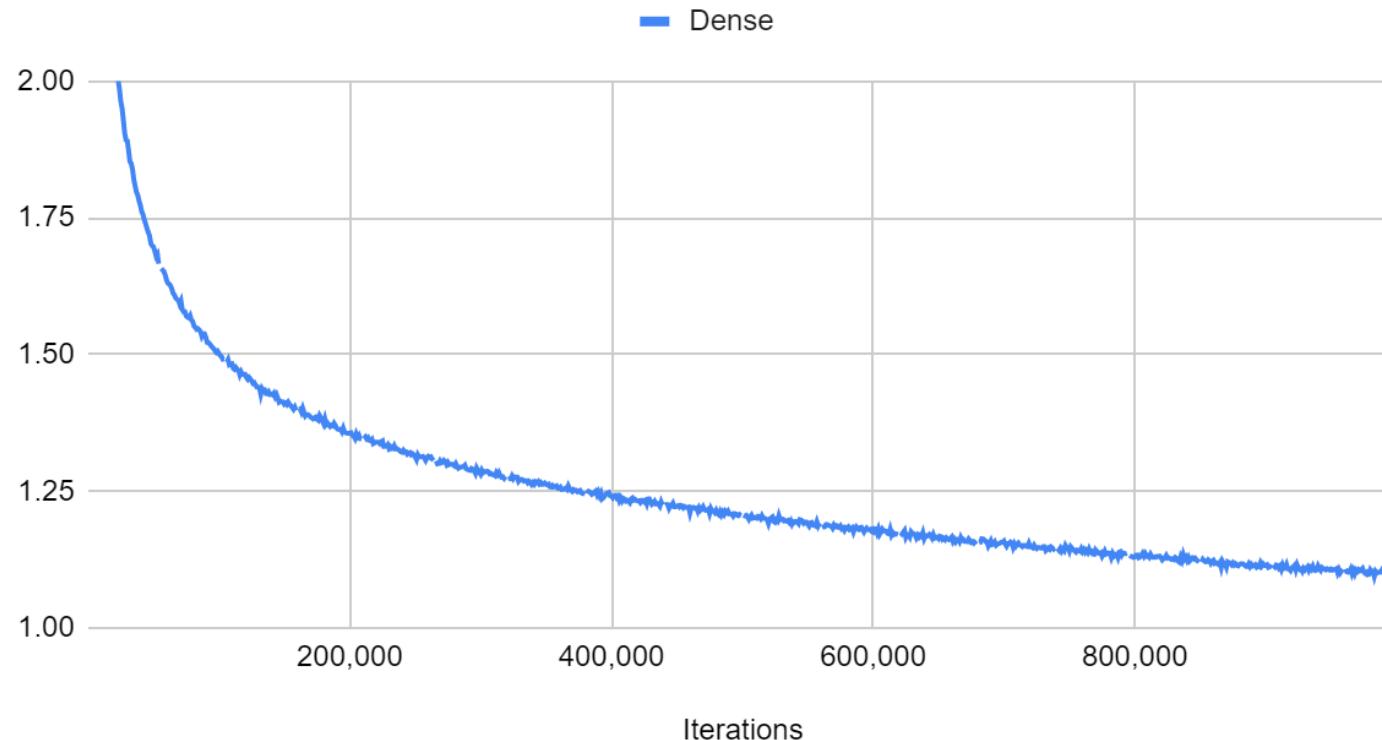
- Reduce layer width: model still dense, requires no special hardware
- Block-sparsity: easier to accelerate
- Unstructured fine-grained sparsity: upper bound on accuracy

Let's compare with 2:4 structured sparsity

# BERT-LARGE CASE STUDY

## Simpler Networks

BERT-Large Validation Loss

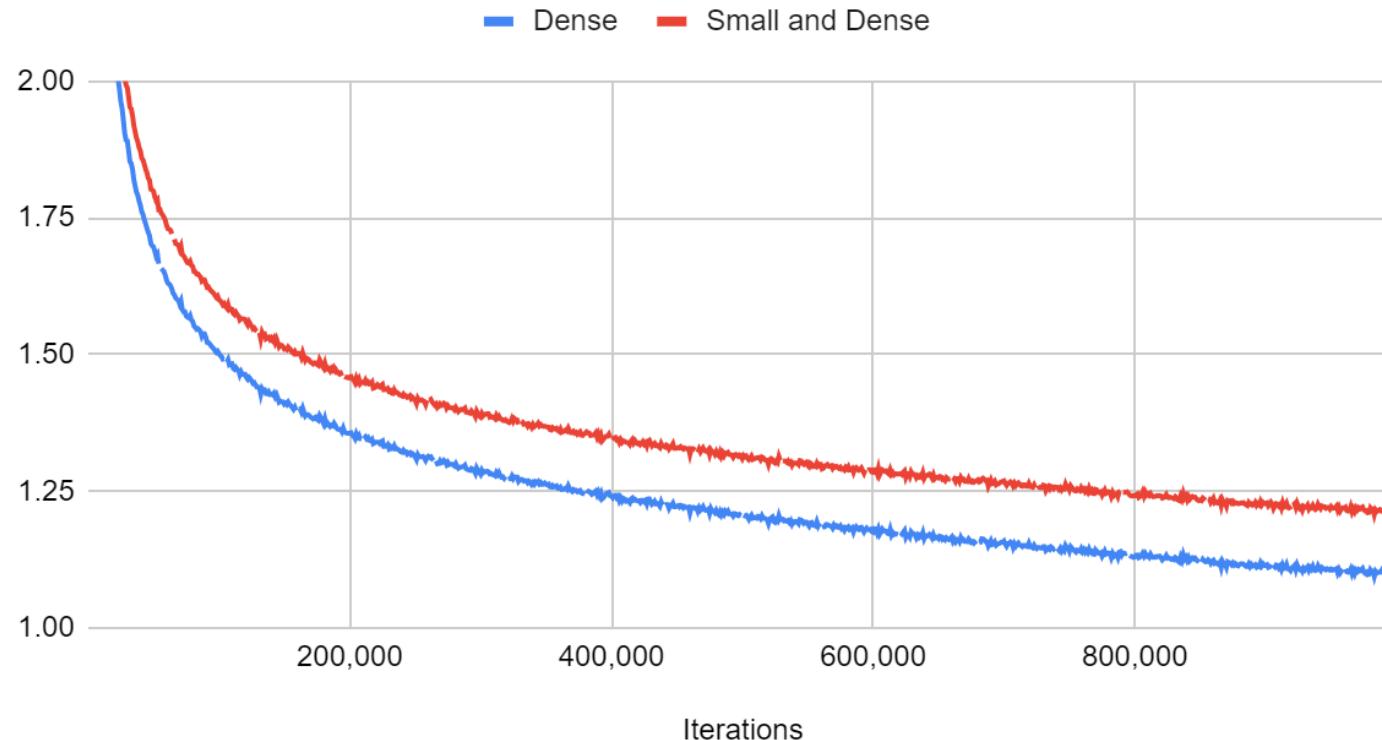


**Note:** Validation loss is *not* final accuracy, but it can show general trends in network quality.

# BERT-LARGE CASE STUDY

## Simpler Networks - From Scratch

BERT-Large Validation Loss

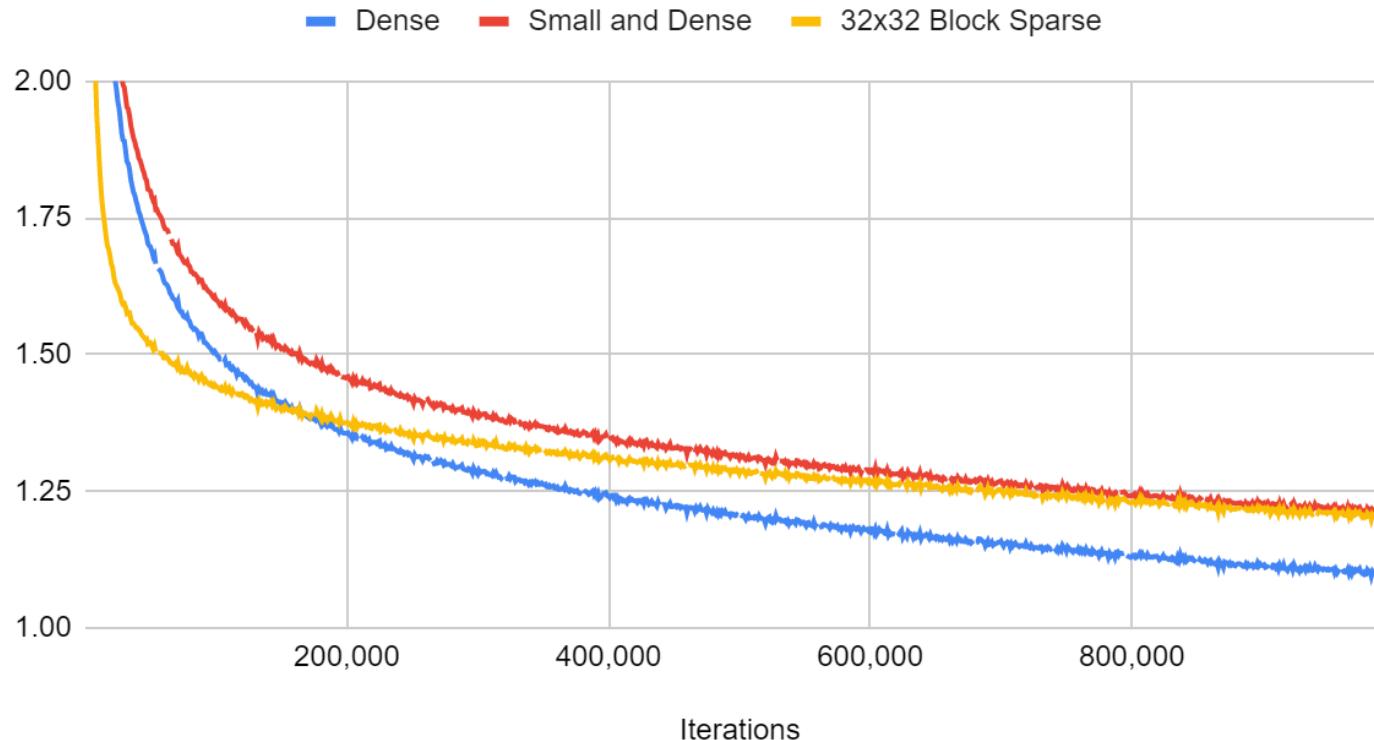


Halving the hidden size of encoders gives a smaller, dense network that is simple to accelerate, but the network itself is much worse.

# BERT-LARGE CASE STUDY

## Simpler Networks - Fine-Tuned

BERT-Large Validation Loss



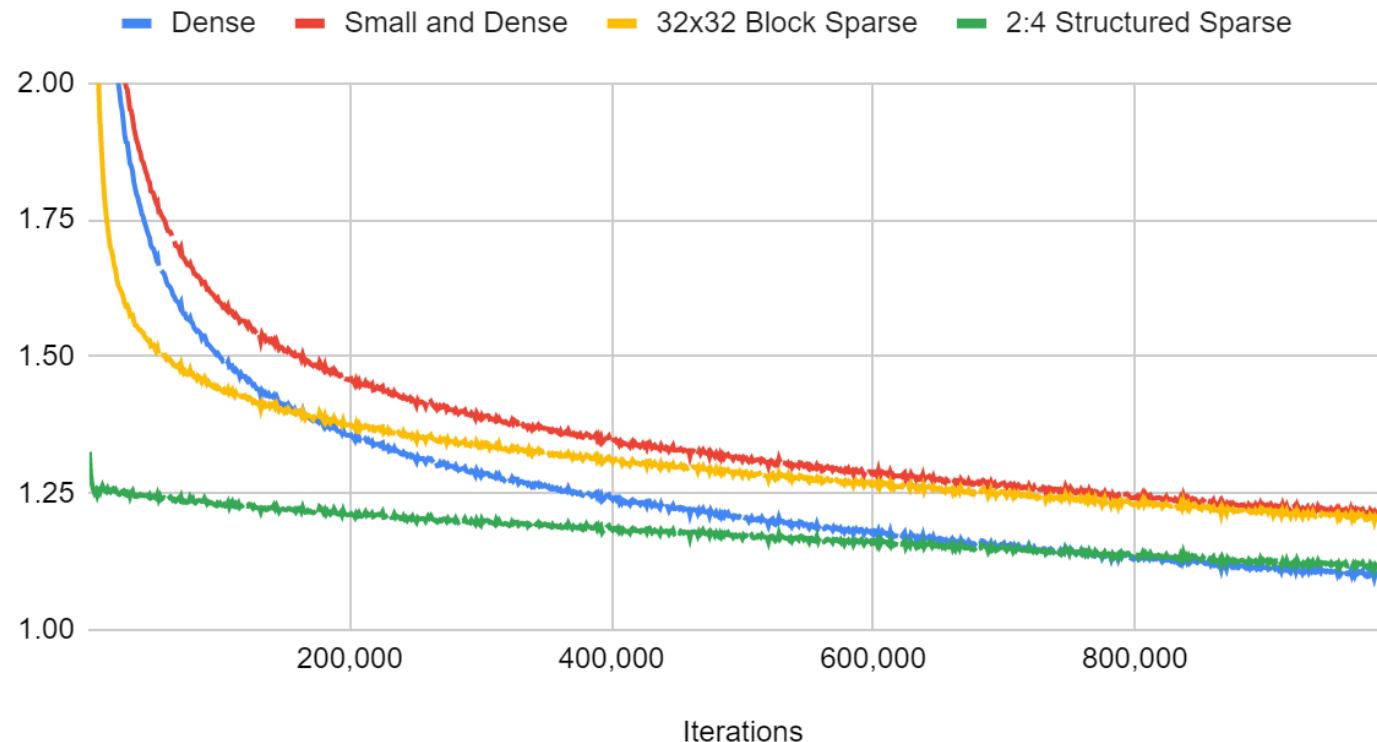
Pruning the full network to 50% sparsity with 32x32 blocks then fine tuning can be accelerated on most parallel hardware, but the network performs poorly.

Note: For this and the following pruning techniques, we use the same model size - no growing the model as we prune.

# BERT-LARGE CASE STUDY

## Simpler Networks - Fine-Tuned

BERT-Large Validation Loss

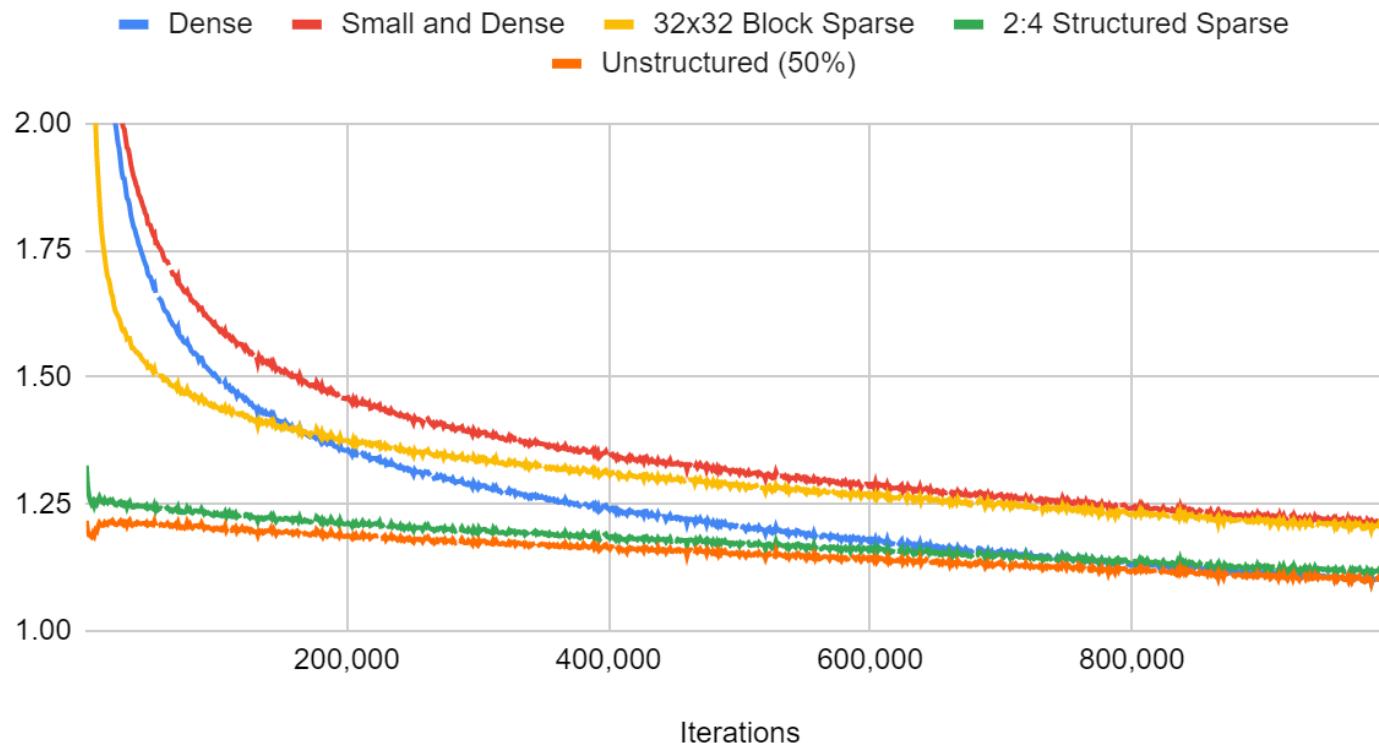


Structured Sparsity is easy to accelerate with A100 and converges to nearly the same loss - final accuracy on SQuAD v1.1 is equivalent to dense.

# BERT-LARGE CASE STUDY

## Simpler Networks - Fine-Tuned

### BERT-Large Validation Loss

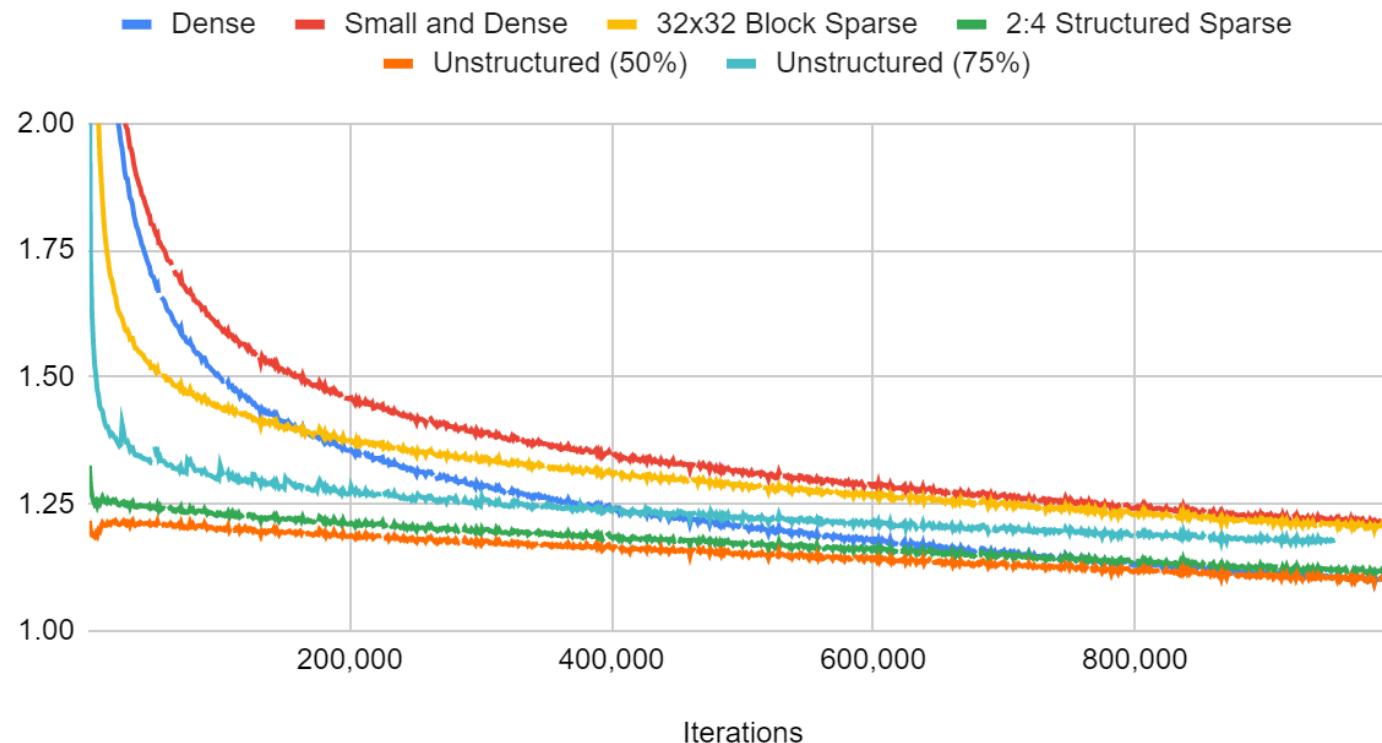


Completely unstructured, fine-grained sparsity has similar loss compared to enforcing a 2:4 structure, but at only 50% sparse, it is incredibly hard to exploit.

# BERT-LARGE CASE STUDY

## Simpler Networks - Fine-Tuned

### BERT-Large Validation Loss



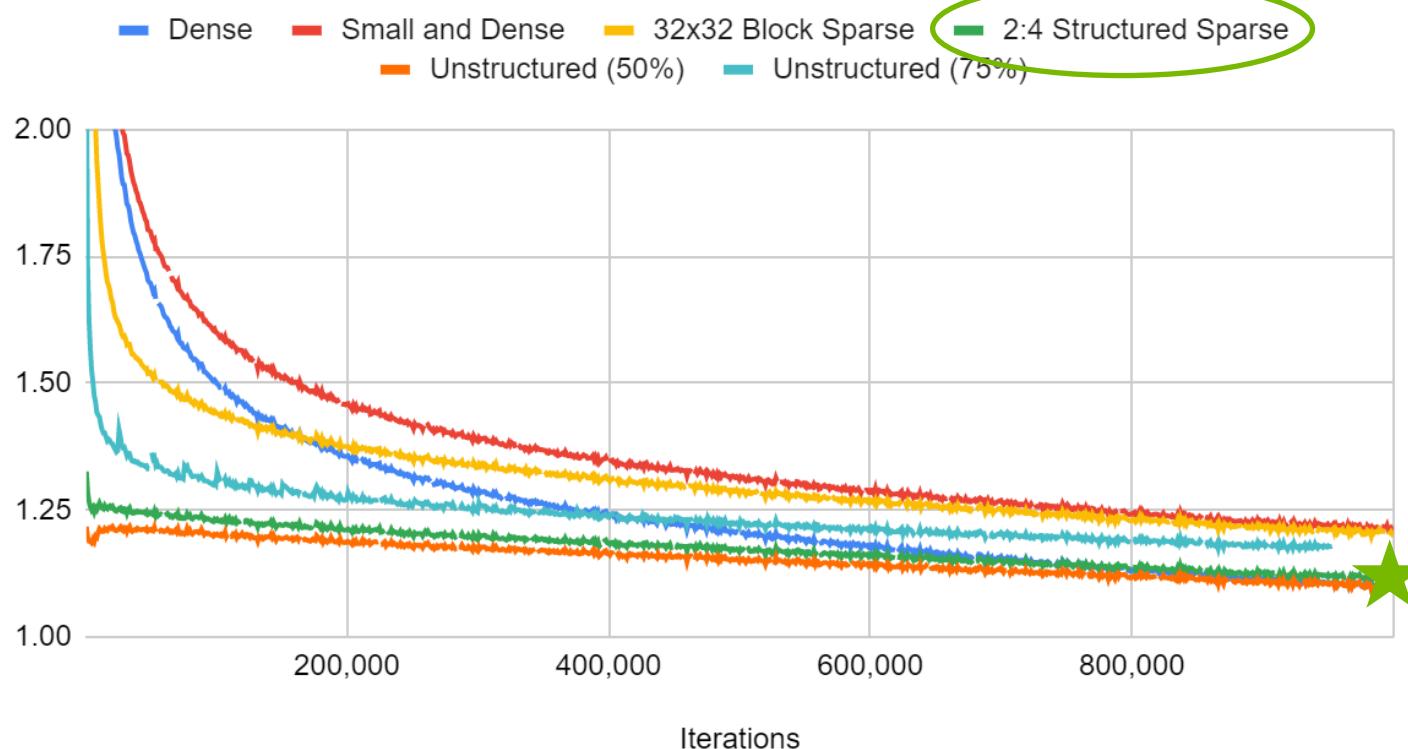
75% unstructured sparsity could be accelerated with standard techniques, but it is still tricky.

However, it does not approach the quality of the dense baseline.

# BERT-LARGE CASE STUDY

## Simpler Networks - Fine-Tuned

BERT-Large Validation Loss



Of these options, **2:4 structured sparsity** is the only technique that both maintains network quality *and* is easy to accelerate on A100



# ASP: AUTOMATIC SPARSITY FOR RETRAINING IN FRAMEWORKS

# GENERATE A STRUCTURED SPARSE NETWORK

APEX's Automatic SParsity: ASP

Conceptually simple - 3 step recipe

Simple in practice - 3 lines of code

NVIDIA's APEX library

AMP = Automatic Mixed Precision

ASP = Automatic SParsity

# GENERATE A STRUCTURED SPARSE NETWORK

## APEX's Automatic SParsity: ASP

Original PyTorch training loop

```
import torch

device = torch.device('cuda')

model = TheModelClass(*args, **kwargs) # Define model structure

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9) # Define optimizer

x, y = DataLoader(...) #load data samples and labels to train the model
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

torch.save(model.state_dict(), 'dense_model.pth')
```

# GENERATE A STRUCTURED SPARSE NETWORK

## APEX's Automatic SParsity: ASP

PyTorch sparse fine-tuning loop

```
import torch
from apex.contrib.sparsity import ASP
device = torch.device('cuda')

model = TheModelClass(*args, **kwargs) # Define model structure

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9) # Define optimizer

x, y = DataLoader(...) #load data samples and labels to train the model
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

torch.save(model.state_dict(), 'pruned_model.pth') # checkpoint has weights and masks
```

NVIDIA's Sparsity library

# GENERATE A STRUCTURED SPARSE NETWORK

## APEX's Automatic SParsity: ASP

PyTorch sparse fine-tuning loop

```
import torch
from apex.contrib.sparsity import ASP
device = torch.device('cuda')

model = TheModelClass(*args, **kwargs) # Define model structure
model.load_state_dict(torch.load('dense_model.pth'))

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9) # Define optimizer

x, y = DataLoader(...) #load data samples and labels to train the model
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

torch.save(model.state_dict(), 'pruned_model.pth') # checkpoint has weights and masks
```

Load the trained model

# GENERATE A STRUCTURED SPARSE NETWORK

## APEX's Automatic SParsity: ASP

PyTorch sparse fine-tuning loop

```
import torch
from apex.contrib.sparsity import ASP
device = torch.device('cuda')

model = TheModelClass(*args, **kwargs) # Define model structure
model.load_state_dict(torch.load('dense_model.pth'))

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9) # Define optimizer

ASP.prune_trained_model(model, optimizer)

x, y = DataLoader(...) #load data samples and labels to train the model
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

torch.save(model.state_dict(), 'pruned_model.pth') # checkpoint has weights and masks
```

Init mask buffers, tell optimizer to mask weights and gradients, compute sparse masks: Universal Fine Tuning

# GENERATE A STRUCTURED SPARSE NETWORK

## APEX's Automatic SParsity: ASP

PyTorch sparse fine-tuning loop

```
import torch
from apex.contrib.sparsity import ASP
device = torch.device('cuda')

model = TheModelClass(*args, **kwargs) # Define model structure
model.load_state_dict(torch.load('path_to_trained_model.pt')) # Load trained model
optimizer = optim.SGD(model.parameters(), lr=0.01) # Define optimizer
ASP.prune_trained_model(model) # Prune trained model

x, y = DataLoader(...).next() # load data
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

torch.save(model.state_dict(), 'pruned_model.pth') # checkpoint has weights and masks
```

3 Lines!



# DIRECTIONS FOR FURTHER RESEARCH

# SHORTEN RETRAINING

For some networks we were able to shorten retraining (Step-3) to a fraction of Step-1

However, these shortened hyper-parameters didn't apply to all networks

**Further research:** investigate shorter, universal recipes

Network	Fine-Tuning Epochs		Accuracy		
	Baseline	Reduced	Dense FP16	Sparse FP16	Short Sparse INT8
ResNet-50	90	15	76.6	76.8	76.6
Inception v3	90	30	77.1	77.1	77.0
DenseNet-161	90	15	78.8	78.8	78.8

# ACCELERATE TRAINING WITH SPARSITY

Sparse Tensor Cores can accelerate Step-3 (sparse retraining)

Can we eliminate Step-1?

- Recipe for training with sparsity from scratch (randomly initialized weights)

Research questions:

- How long to train densely (“dense warmup”)?
- Whether to periodically re-prune, if so: how frequently?
- How to use sparsity to accelerate weight gradient computation?
  - Input matrices are dense (activations and activation gradients), output is weight gradients (could be sparse)

Lots of active research, but still lacking a simple, general recipe



# SUMMARY

## Structured Sparsity gives Fast, Accurate Networks

We moved fine-grained weight sparsity from research to production

Fine-grained structured sparsity is:

- 50% sparse, 2 out of 4 elements are zero
- Accurate with our 3-step universal fine-tuning recipe
  - Simple recipe: train dense, prune, re-train sparse
  - Across many tasks, networks, optimizers
- Fast with the NVIDIA Ampere Architecture's Sparse Tensor Cores
  - Up to 1.85x in individual layers
  - Up to 1.5x in end-to-end networks

- S22082: Mixed-Precision Training of Neural Networks
- S21929: Tensor Core Performance on NVIDIA GPUs: The Ultimate Guide
- S21819: Optimizing Applications for NVIDIA Ampere GPU Architecture

5/20 2:45pm PDT  
5/21 9:00am PDT  
5/21 10:15am PDT

