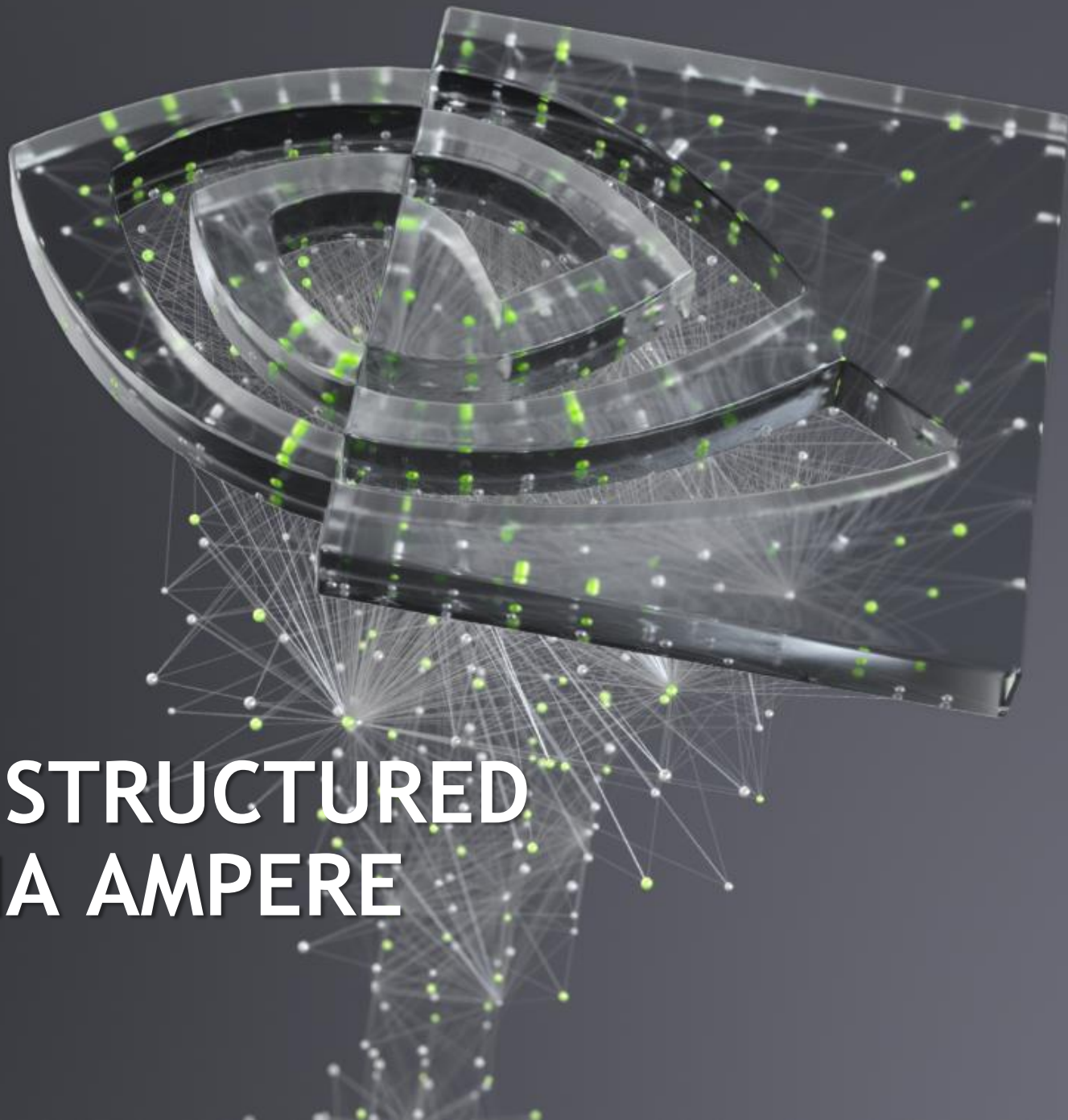




MAKING THE MOST OF STRUCTURED SPARSITY IN THE NVIDIA AMPERE ARCHITECTURE

Jeff Pool, Senior Architect





OUTLINE

Sparsity Review

Motivation

Taxonomy

Challenges

NVIDIA Ampere Architecture's 2:4 Sparsity

Sparsity Pattern

Sparse Tensor Cores

Performance

Training Recipe

Recipe steps

Empirical evaluation

Implementation in PyTorch

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!



SPARSITY REVIEW

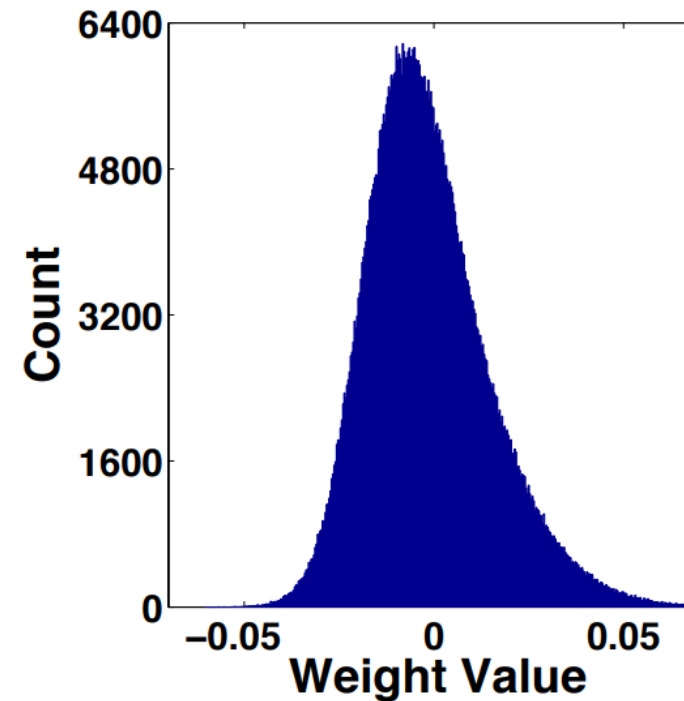
SPARSITY: ONE OF MANY OPTIMIZATION TECHNIQUES

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

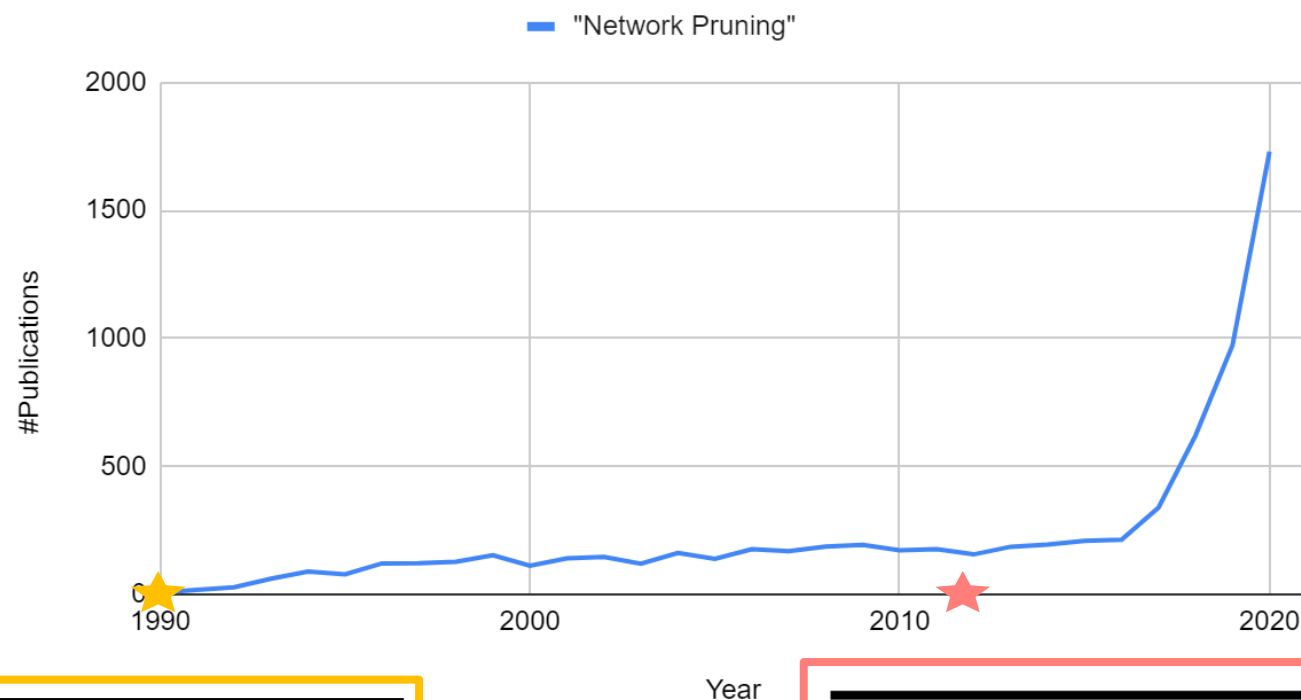
- Eliminate (prune) connections: set some weights to 0
- Eliminate (prune) neurons
- Etc.

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

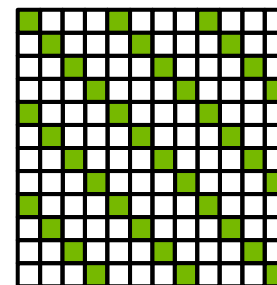
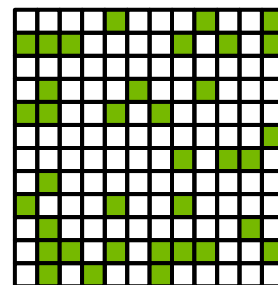
Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

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

SPARSITY TAXONOMY

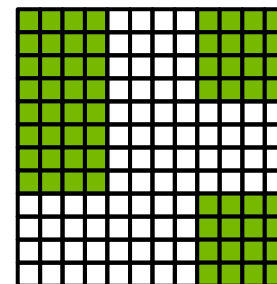
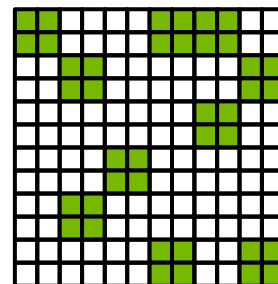
Structure:

- 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

2:4 FINE-GRAINED STRUCTURED SPARSITY

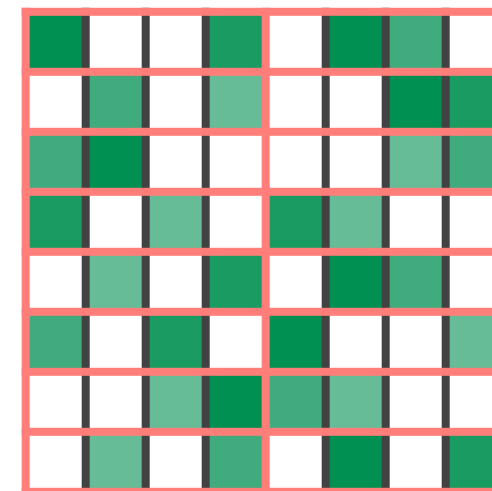
Fine-grained structured sparsity for Tensor Cores


- 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

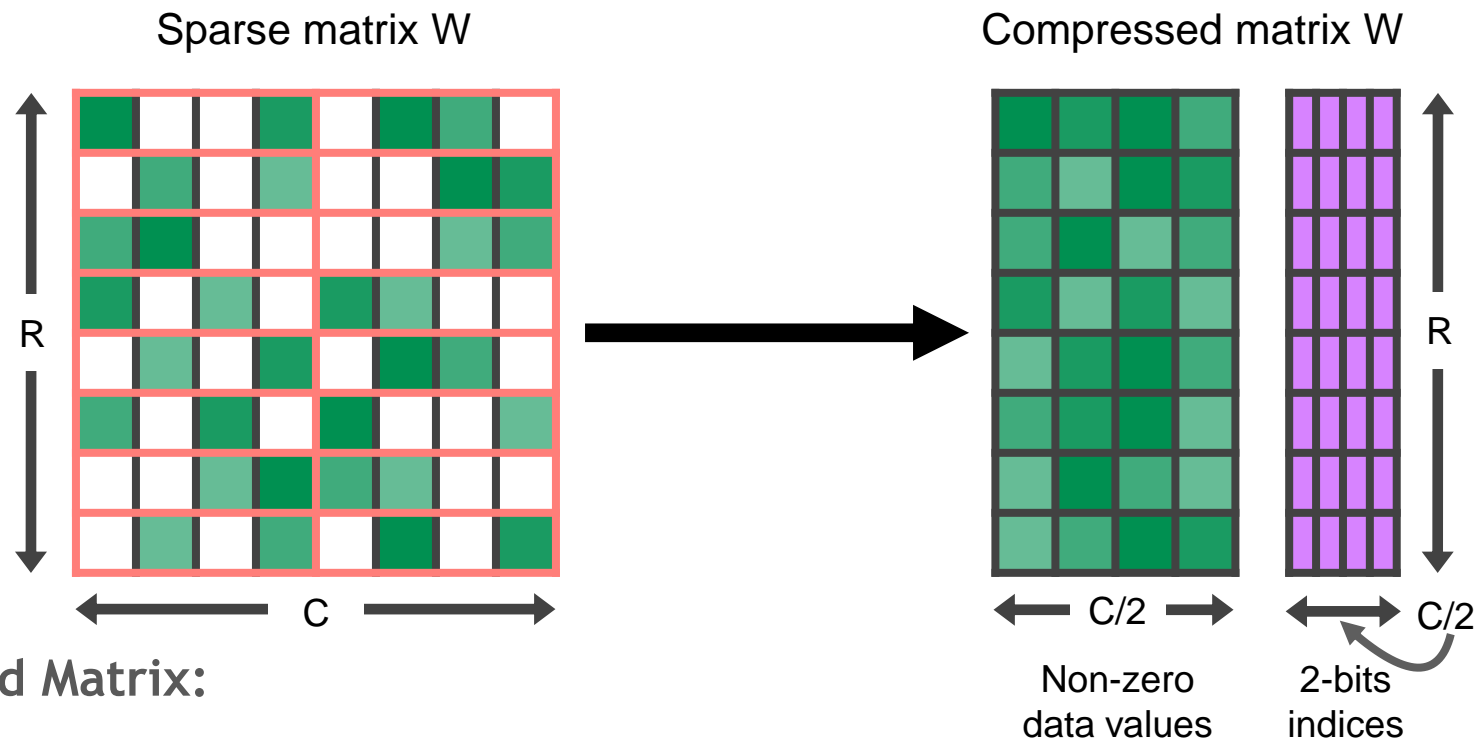
Output: dense activations

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



Compressed Matrix:

Data: $\frac{1}{2}$ size

Metadata: 2b per non-zero element

16b data \Rightarrow 12.5% overhead

8b data \Rightarrow 25% overhead

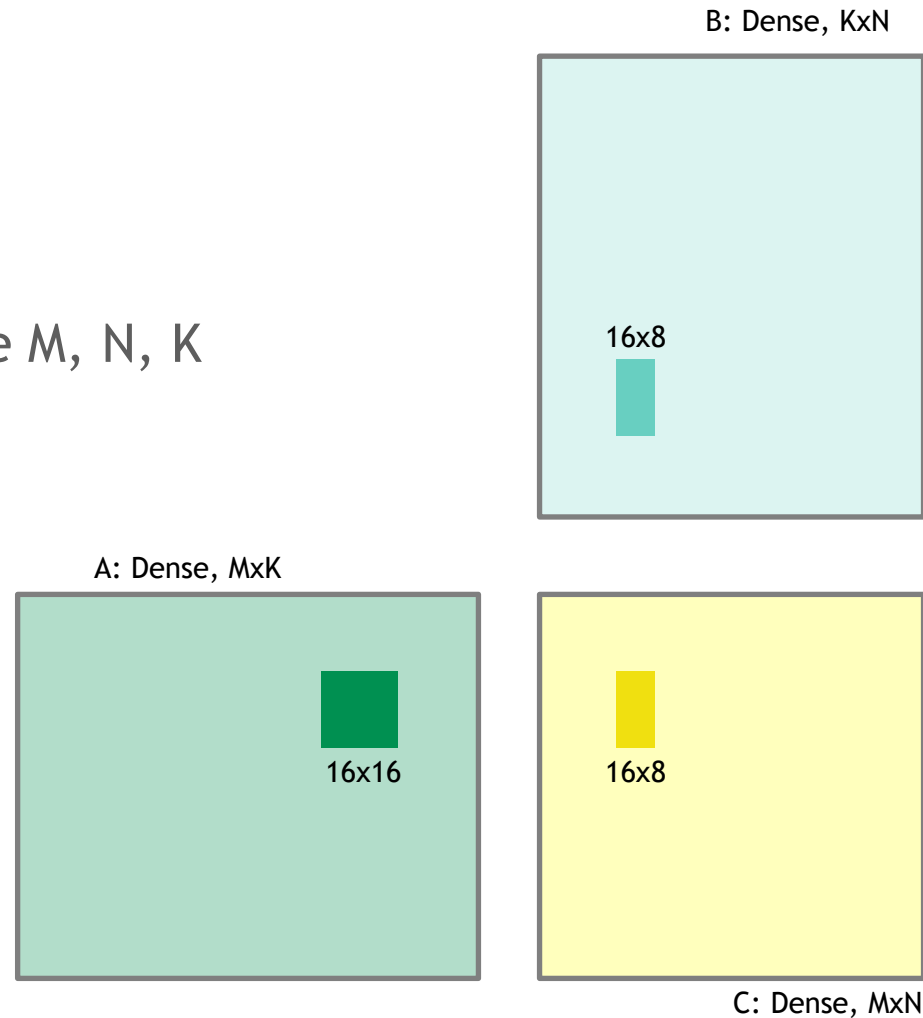
TENSOR CORE OPERATION

Tiling a Large GEMM

Dense Tensor Cores (FP16)

$16 \times 16 * 16 \times 8$ matrix multiplication

Replicated and repeated to support large M, N, K

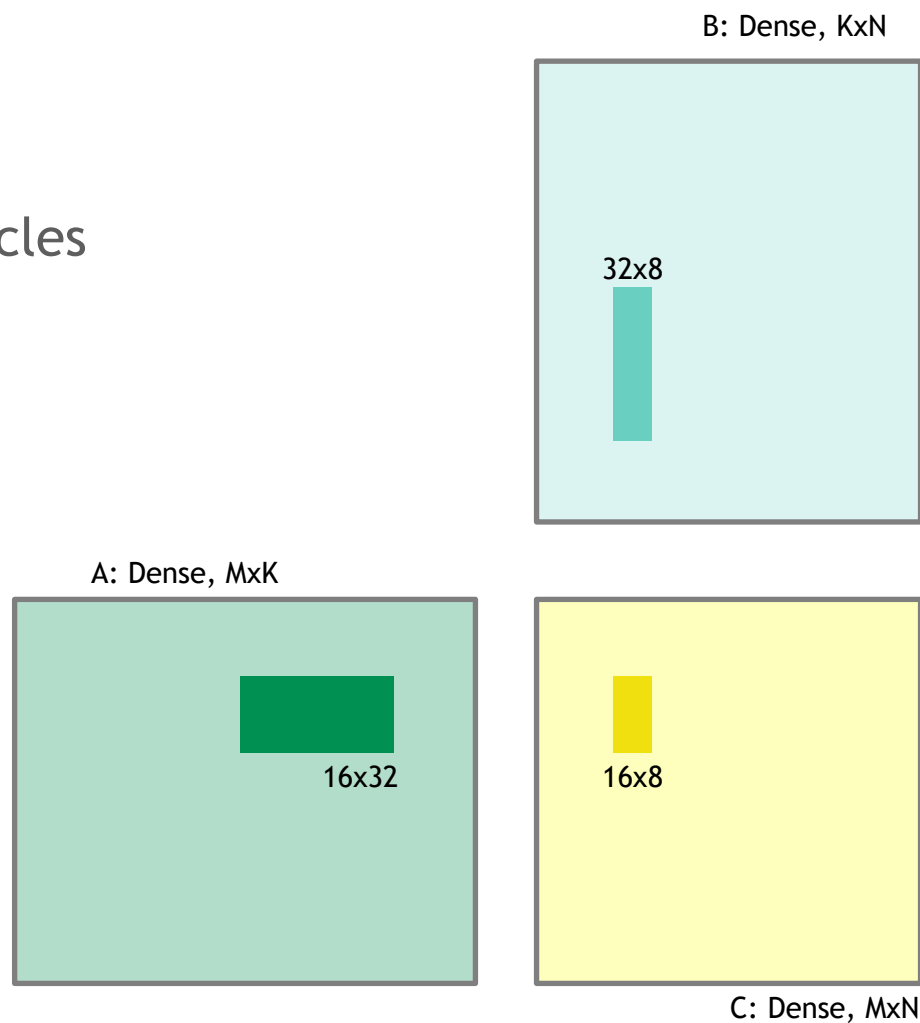


TENSOR CORE OPERATION

Larger Tile = More Cycles

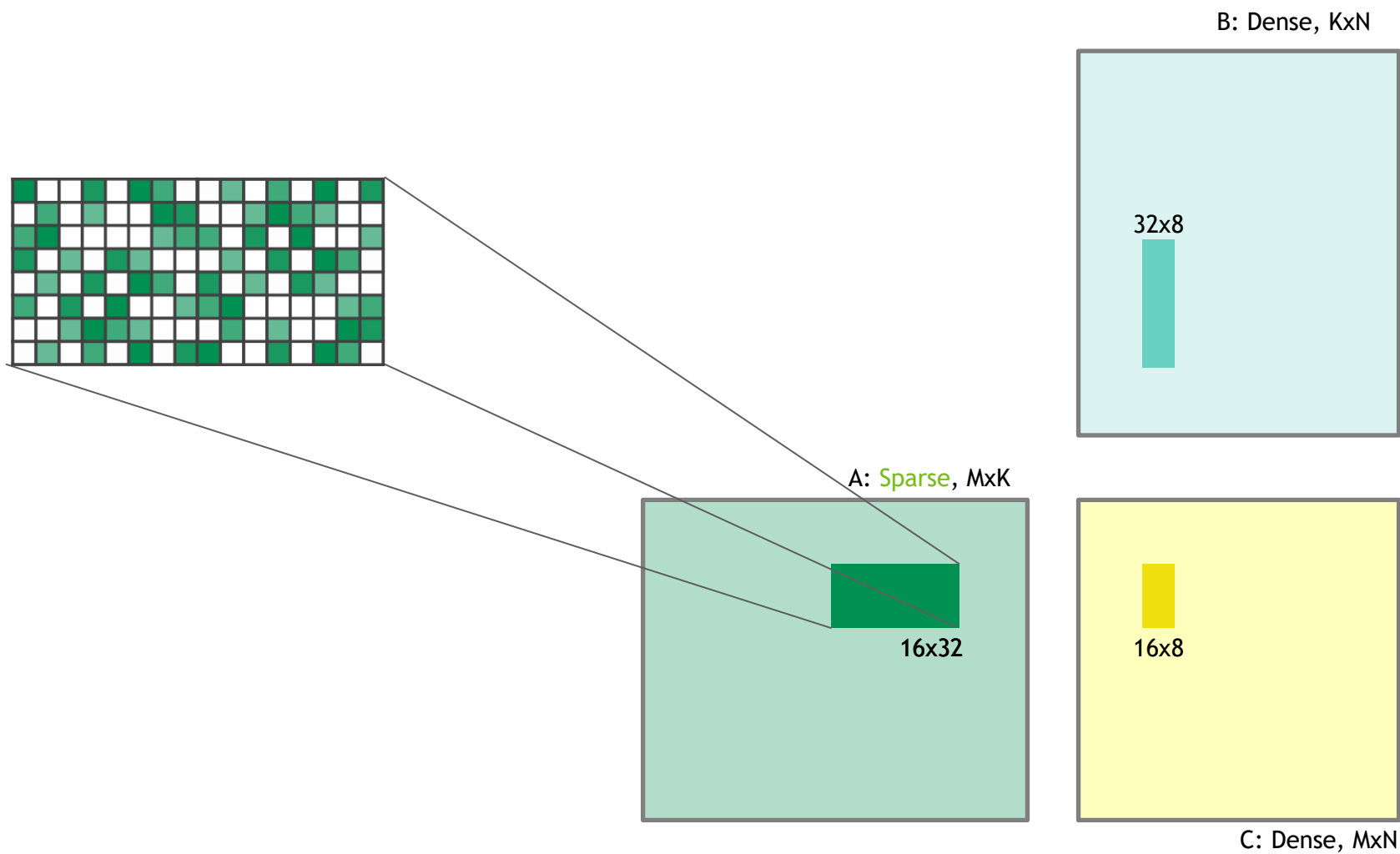
Dense Tensor Cores (FP16)

16x32 * 32x8 matrix multiplication - 2 cycles



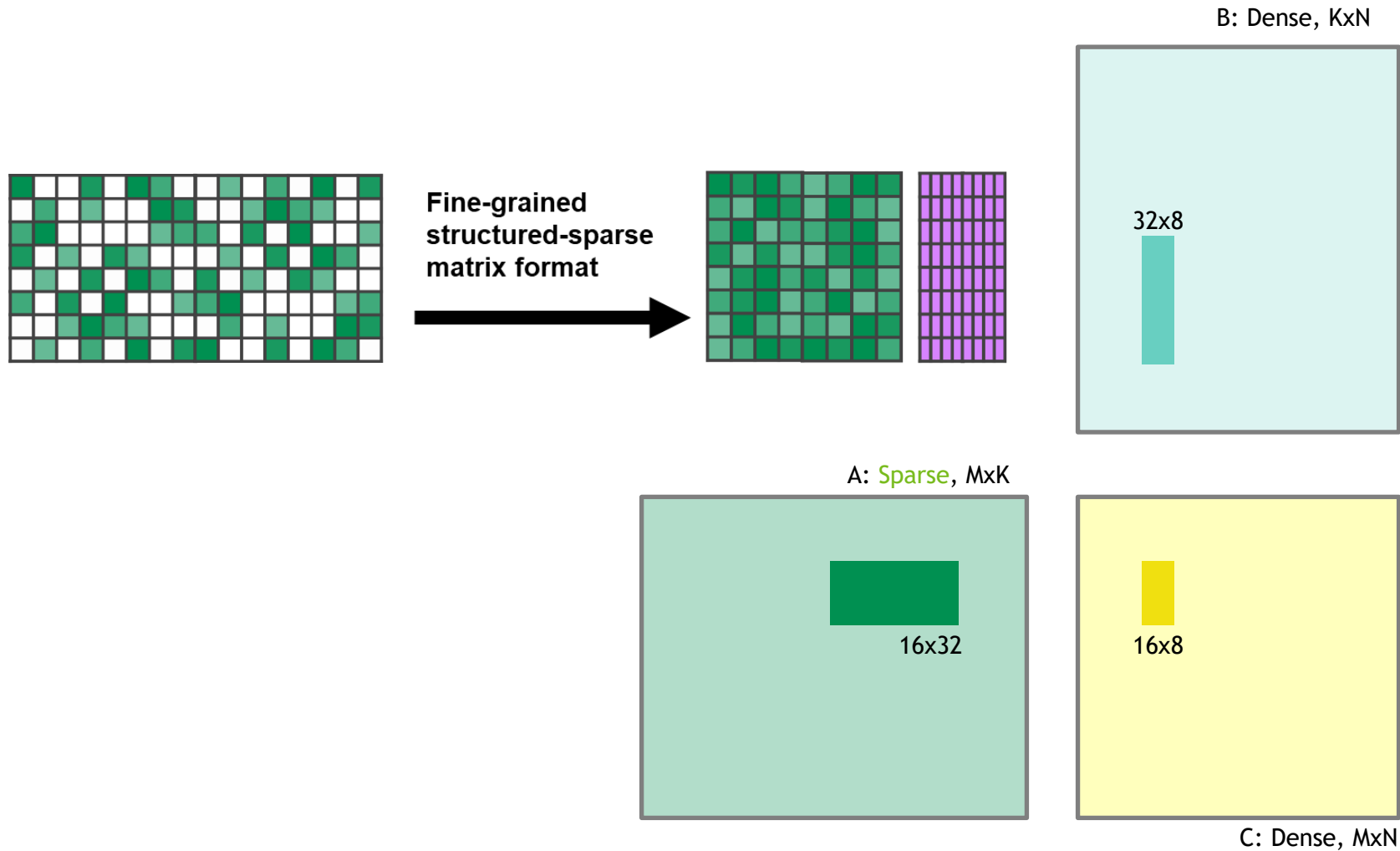
TENSOR CORE OPERATION

Pruned Weight Matrix



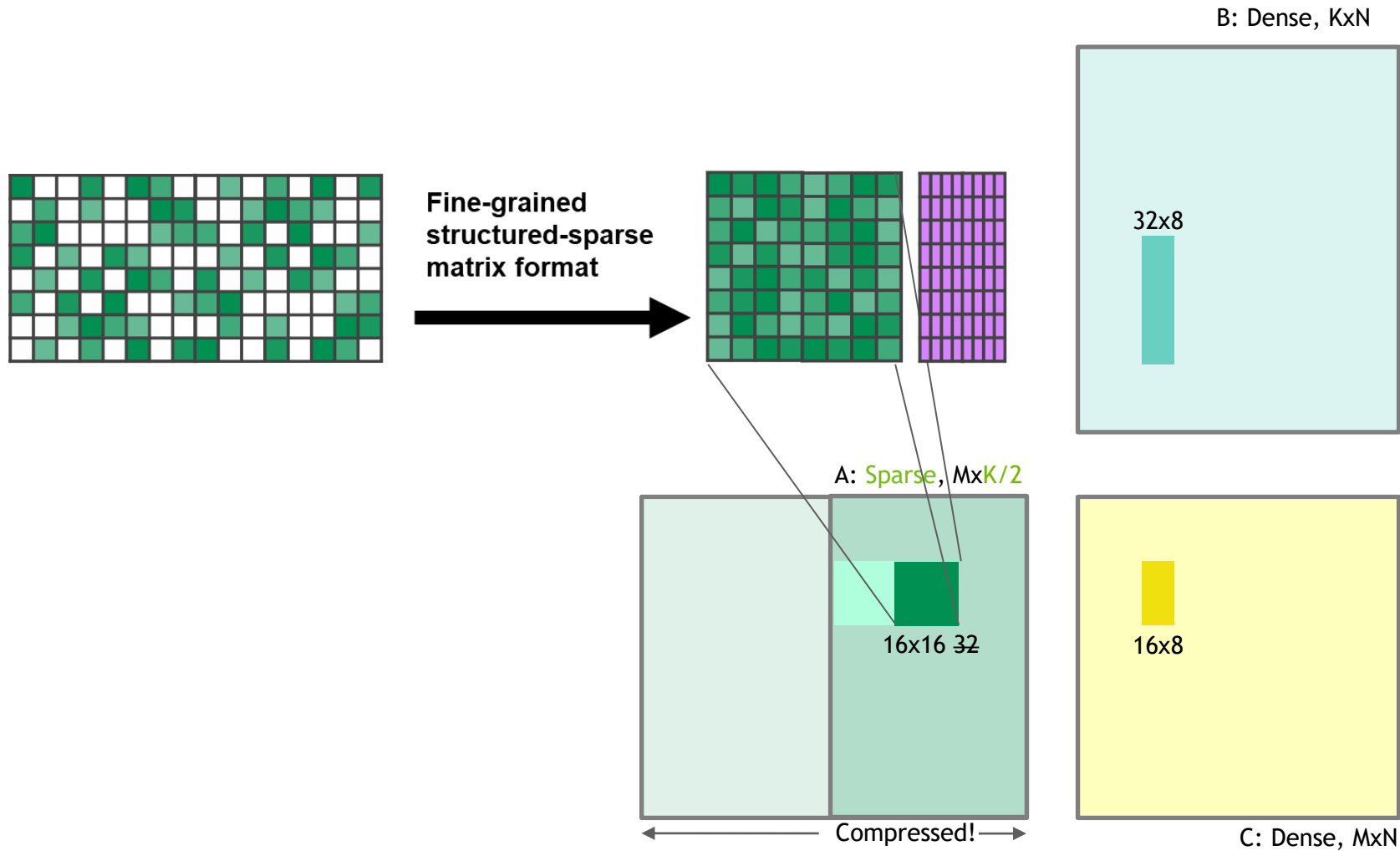
TENSOR CORE OPERATION

Pruned and Compressed Weight Matrix



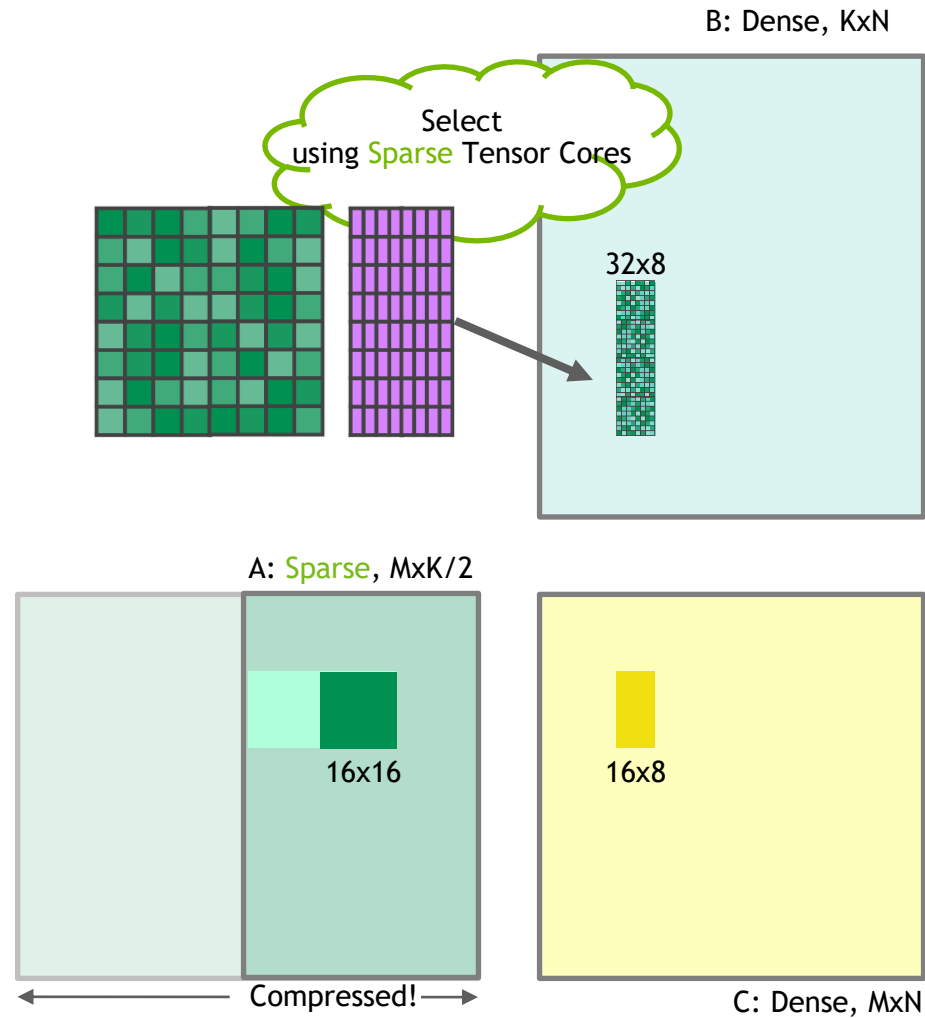
TENSOR CORE OPERATION

Tiling a Large, Sparse GEMM



TENSOR CORE OPERATION

Sparse Tensor Cores - Hardware Magic



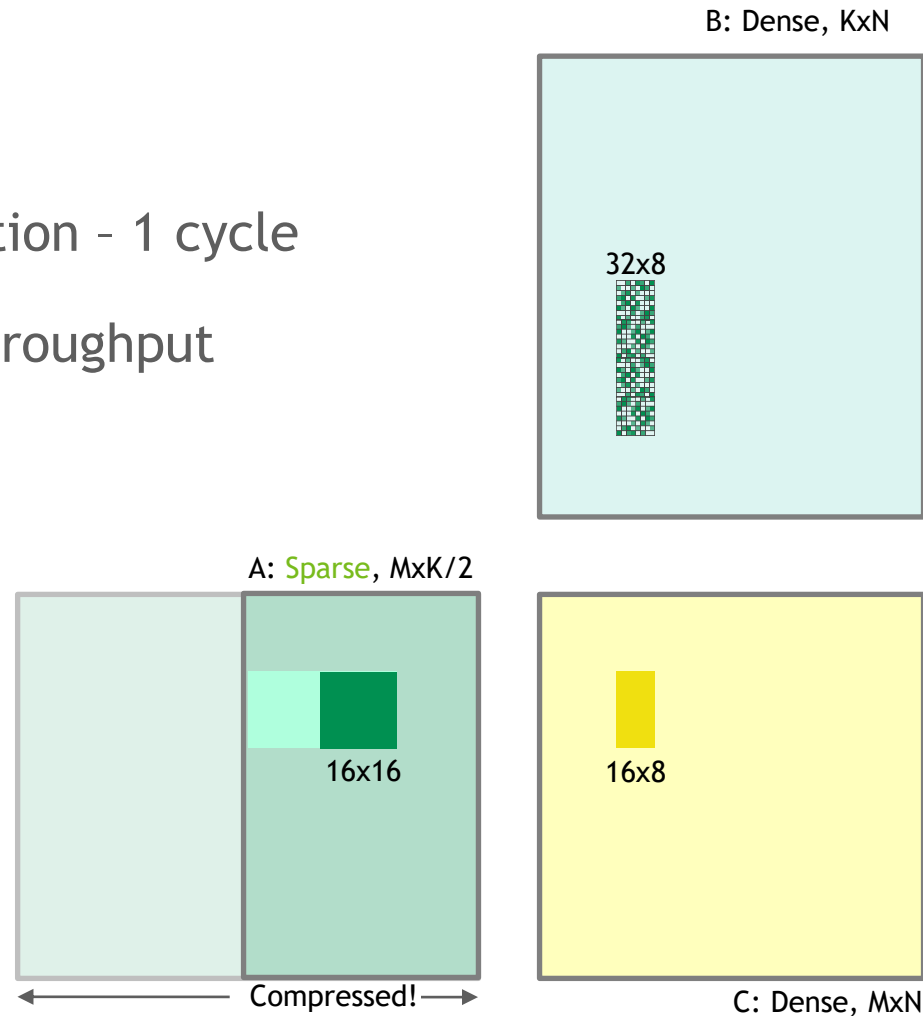
TENSOR CORE OPERATION

Sparse Tensor Cores

Sparse Tensor Cores (FP16)

$16 \times 32 * 32 \times 8$ effective matrix multiplication - 1 cycle

2x the work with the same instruction throughput



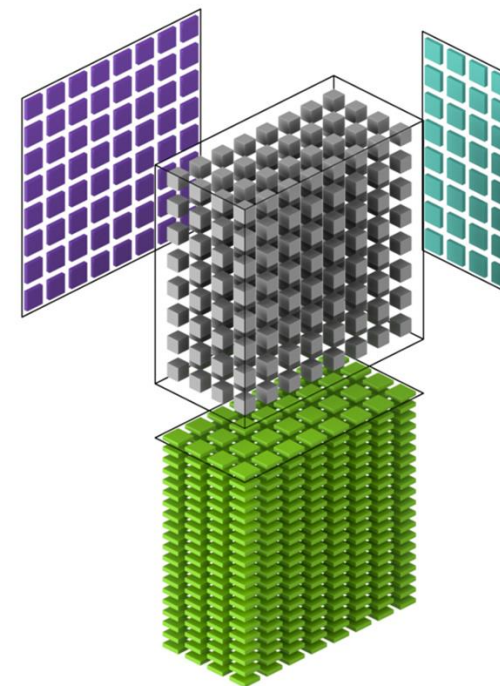


SPARSE TENSOR CORE PERFORMANCE

TENSOR CORE MATH THROUGHPUT

2x with Sparsity

INPUT OPERANDS	ACCUMULATOR	TOPS	Dense vs. FFMA	Sparse 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



CUSPARSELT

2:4 Structured Sparse GEMMs

Key Features

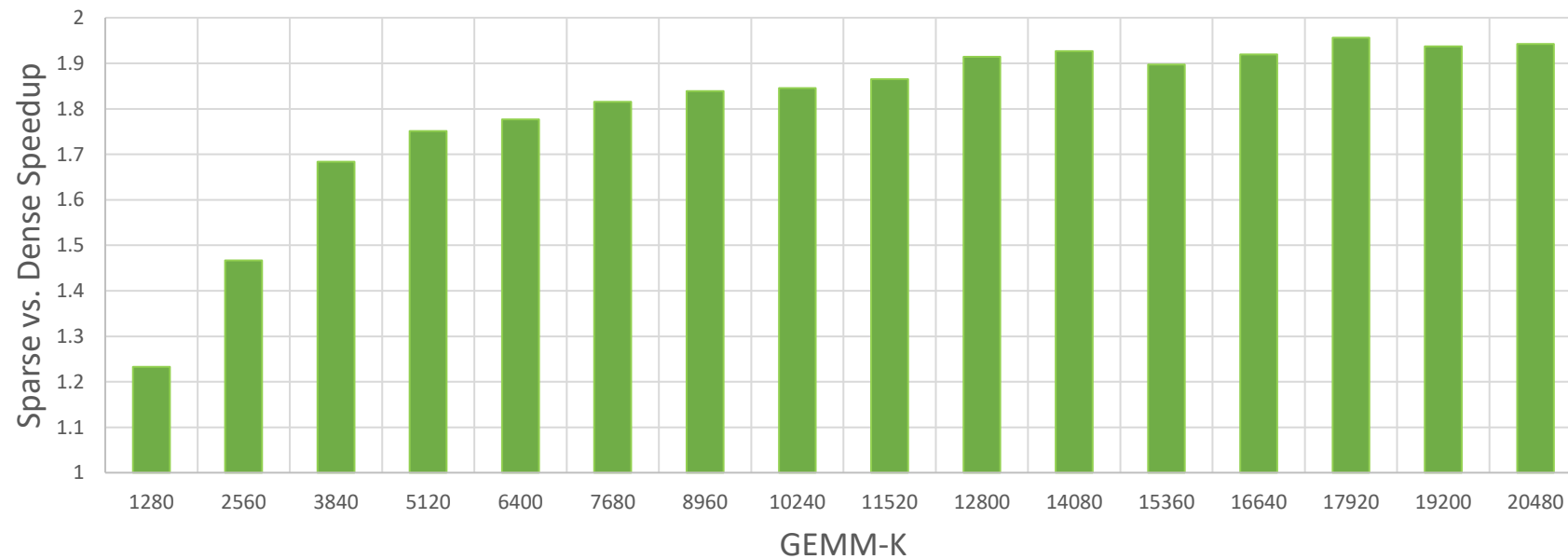
- GEMMs targeting Sparse Tensor Cores
- Mixed-precision
 - FP16 in/out, FP32 Tensor Core accumulation
 - BFLOAT16 in/out, FP32 Tensor Core accumulation
 - INT8 in/out, INT32 Tensor Core accumulation
- Flexible layouts: row-/column-major
- Pruning and compression utilities
- Auto-tuning for optimal kernel selection

<https://docs.nvidia.com/cuda/cusparselt/index.html>

CUSPARSELT

GEMMS up to 2x faster

INT8 (TN) cuSPARSELt vs. cuBLAS Performance
GEMM-M = GEMM-N = 10240

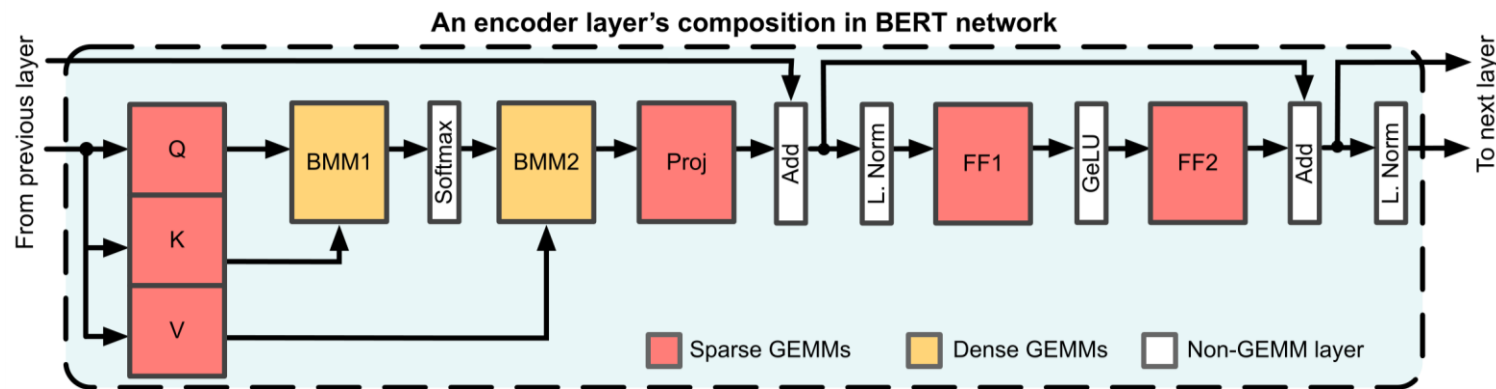


BERT-LARGE PERFORMANCE

MLPerf Inference 0.7

Our 2:4 sparse submission shows matching accuracy with an end-to-end speedup of 22% over a highly-optimized dense implementation

- INT8, SEQLen=384, pre-/post-LayerNorm residuals
- Four GEMM operations have weights (and can be pruned)
- Dense throughput: 26,625 samples/second
- Sparse throughput: **32,356 samples/second**



TENSORRT 8.0

Out-of-the-box Sparse Tensor Core support

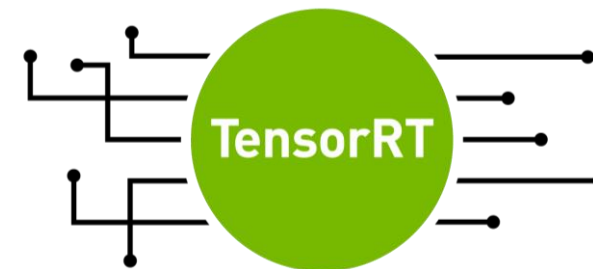
Support for Sparse Convolutions

After generating a sparse network (stay tuned!)

- Set the `kSPARSE_WEIGHTS` flag in the `IBuilderConfig` Inference Phase
- **For example:** `config->setFlag(BuilderFlag::kSPARSE_WEIGHTS);`
- That's it!

TensorRT will profile when building the engine and choose the best kernels automatically

An end-to-end example is coming in the NVIDIA Megatron github repo





TRAINING RECIPE

GOALS FOR A TRAINING RECIPE

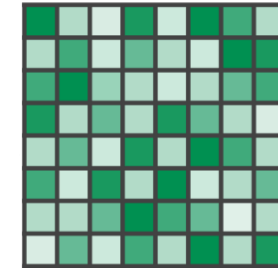
Maintains accuracy

Is applicable across various tasks, network architectures, and optimizers

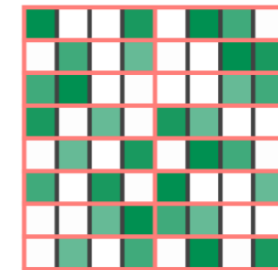
Does not require hyper-parameter searches

RECIPE FOR 2:4 SPARSE NETWORK TRAINING

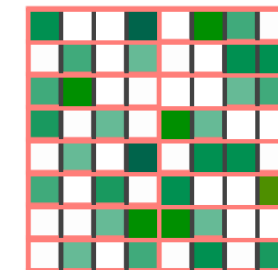
- 1) Train (or obtain) a dense network
- 2) Prune for 2:4 sparsity
- 3) Repeat the original training procedure
 - Same hyper-parameters as in step-1
 - Initialize to weights from step-2
 - Maintain the 0 pattern from step-2: no need to recompute the mask



Dense weights



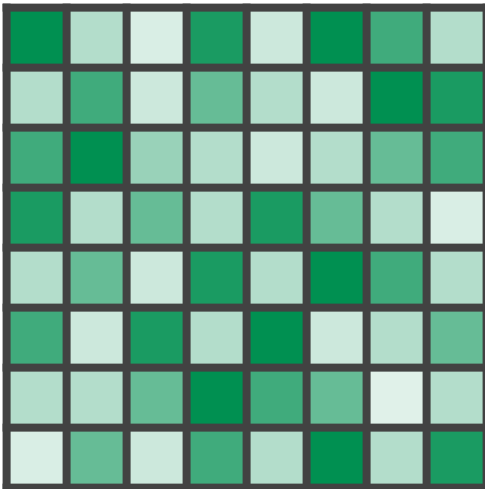
2:4 sparse weights



Retrained 2:4 sparse weights

RECIPE STEP 2: PRUNE WEIGHTS

Dense matrix W



Single-shot, magnitude-based pruning

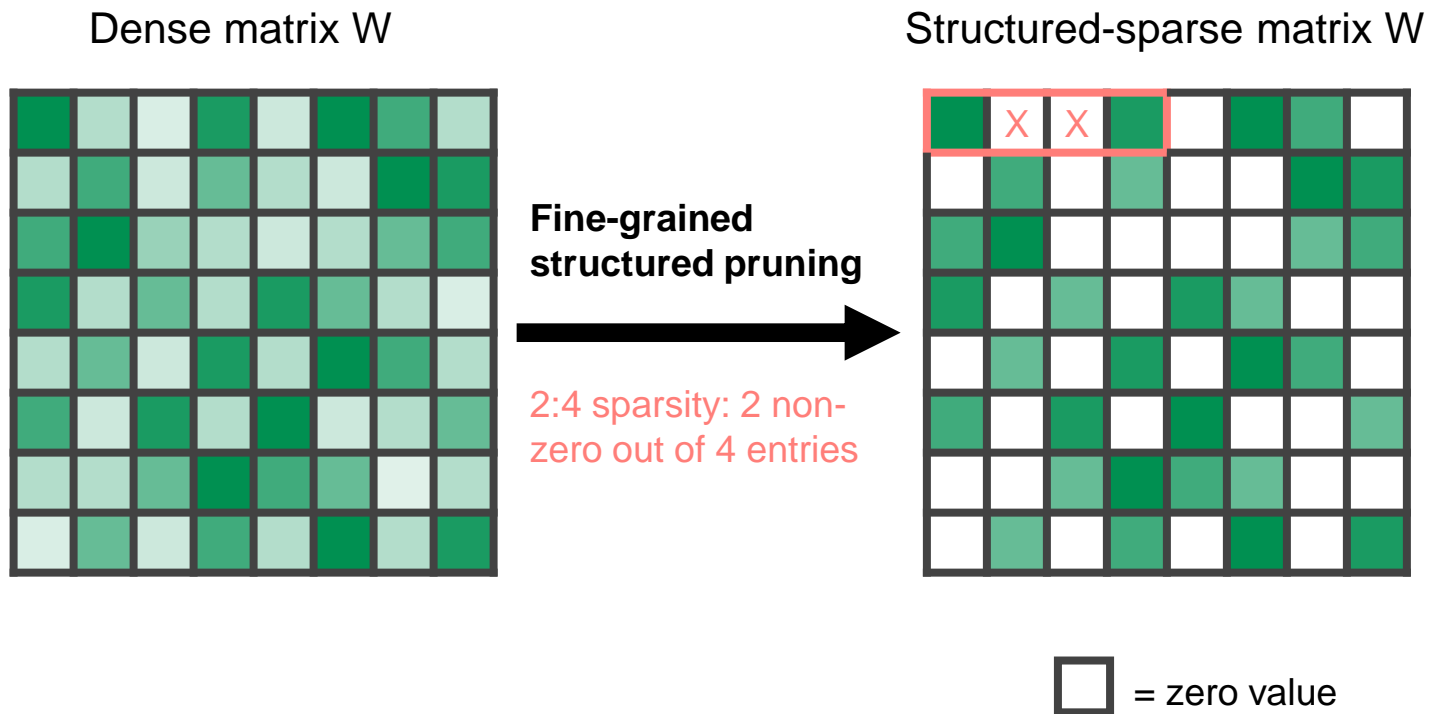
For each 1x4 block of weights:

- Set 2 weights with the smallest magnitudes to 0

Layer weights to prune: conv, linear

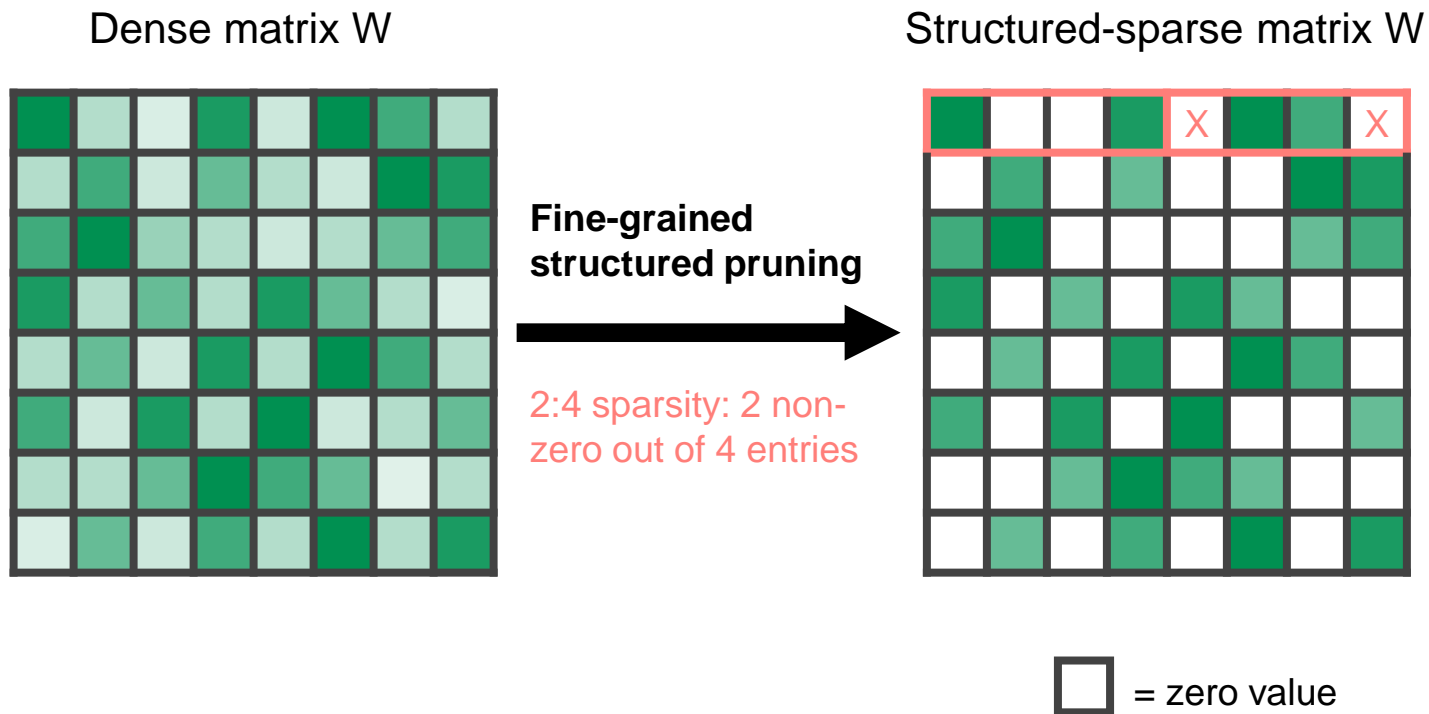
RECIPE STEP 2: PRUNE WEIGHTS

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



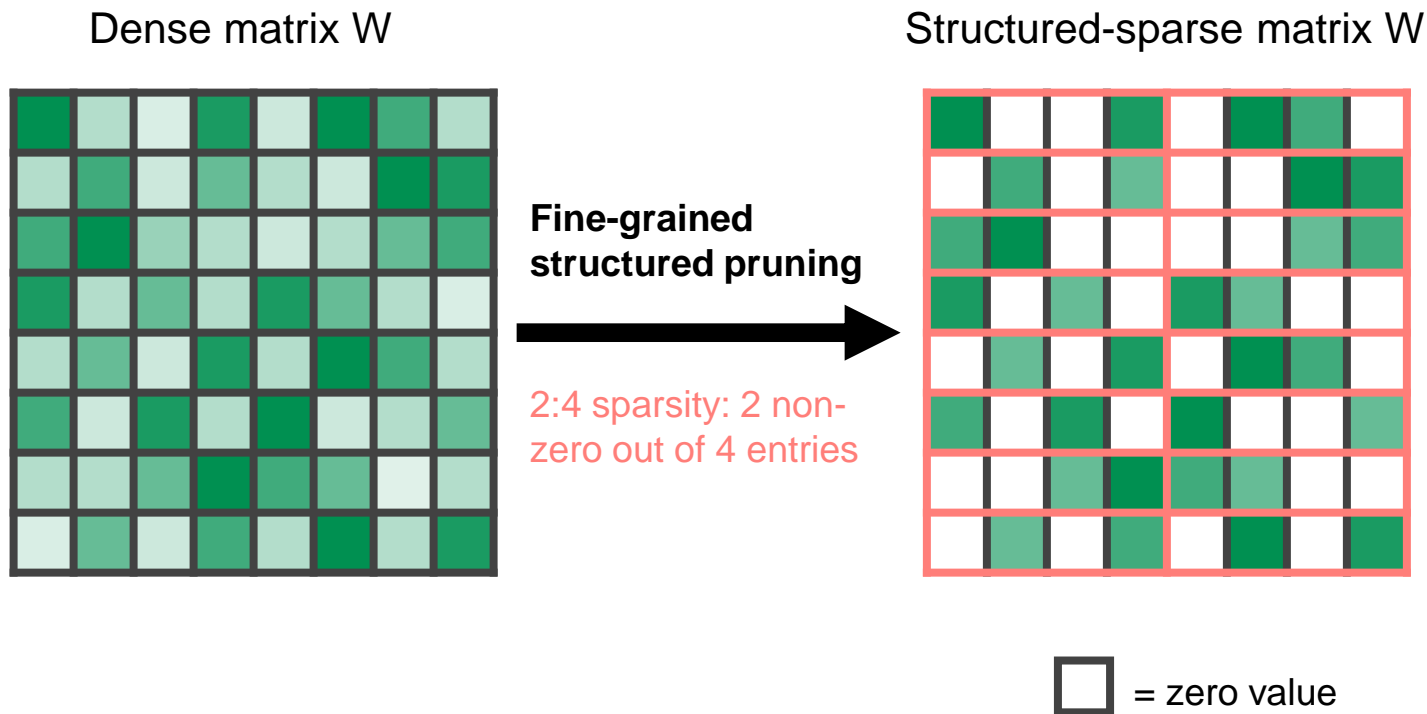
RECIPE STEP 2: PRUNE WEIGHTS

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



RECIPE STEP 2: PRUNE WEIGHTS

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



RECIPE STEP 2: CHANNEL PERMUTATIONS

Preserve Weight Magnitudes by Permuting Weight Tensors Before Pruning

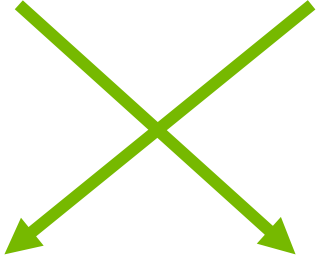
Default weight layout may cluster large values

- This example row loses 140 weight magnitude

100	90	80	10	20	40	30	70
-----	----	---------------	---------------	---------------	----	---------------	----

Permuting along the channel dimension distributes large weights

- After permutation, the lost weight magnitude is only 100



100	90	20	10	80	40	30	70
50	80	50	30	140	5	60	40
15	16	17	10	100	50	40	20
...
80	130	6	49	12	120	60	50

Finding a good permutation for an entire layer is not trivial

- Efficient algorithms and implementations are important

RECIPE STEP 3: RETRAIN

Pruning out 50% of the weight values reduces model accuracy

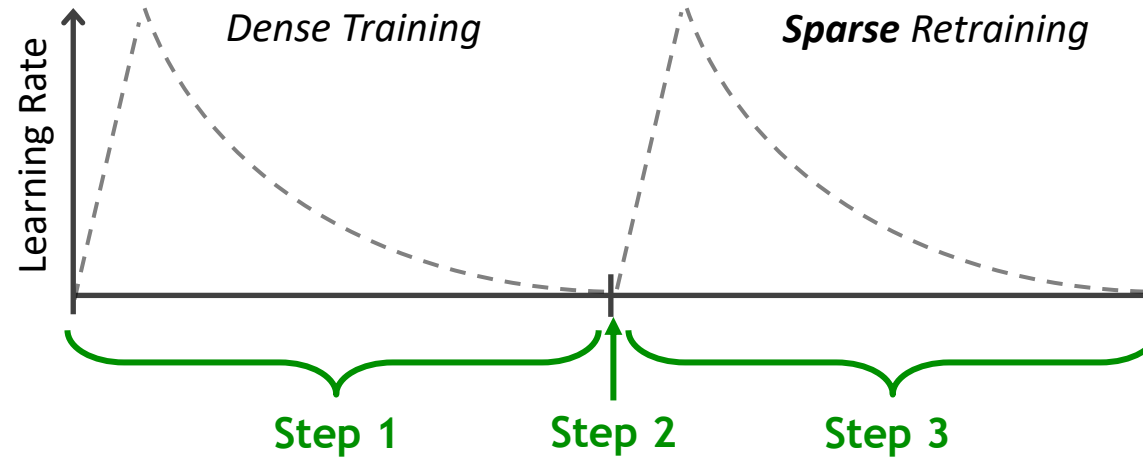
Retraining recovers accuracy

- 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

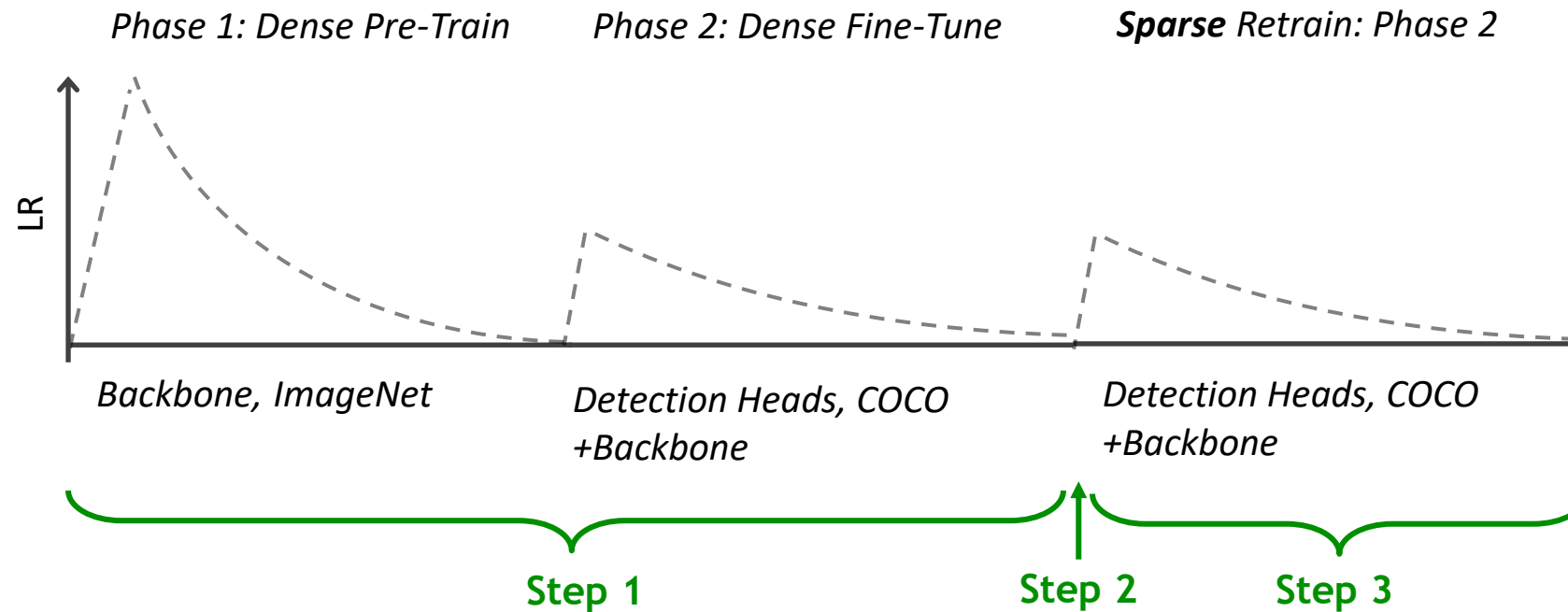
- Pretrain on one task and dataset, then train (fine-tune) on another task and dataset
- Examples:
 - Retinanet for object detection: 1) train for classification on ImageNet, 2) train for detection on COCO
 - BERT for question answering: 1) train for language modeling on BooksCorpus/Wikipedia, 2) train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

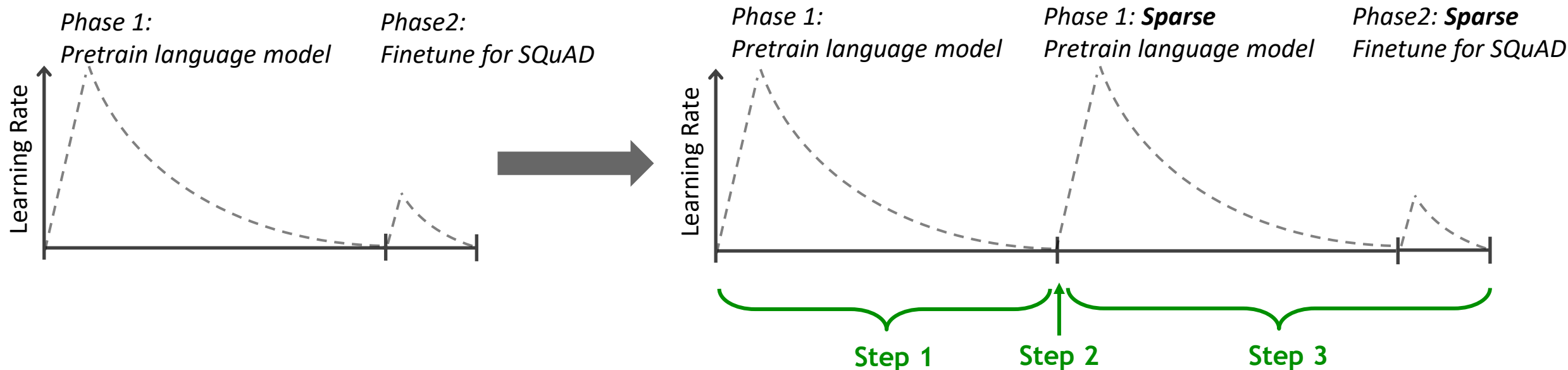
STEP3: DETECTOR EXAMPLE

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

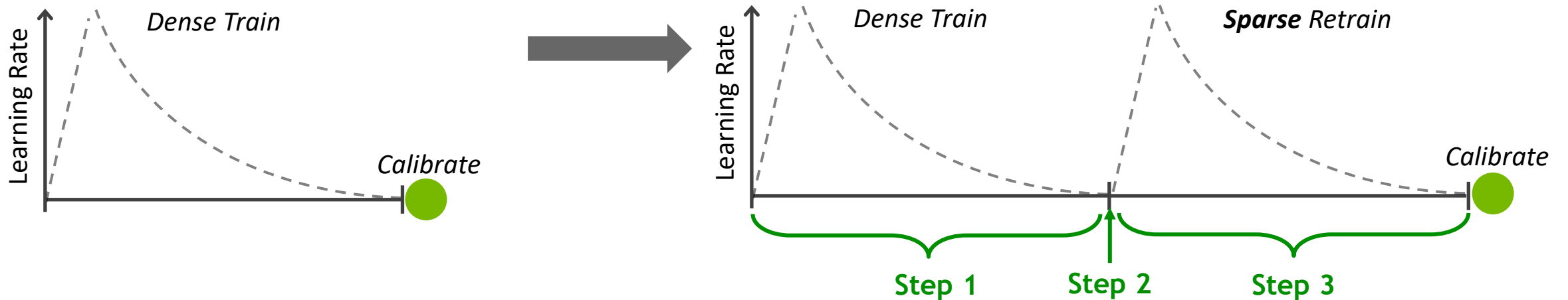
Apply Sparsity Before Quantizing

- ▶ 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

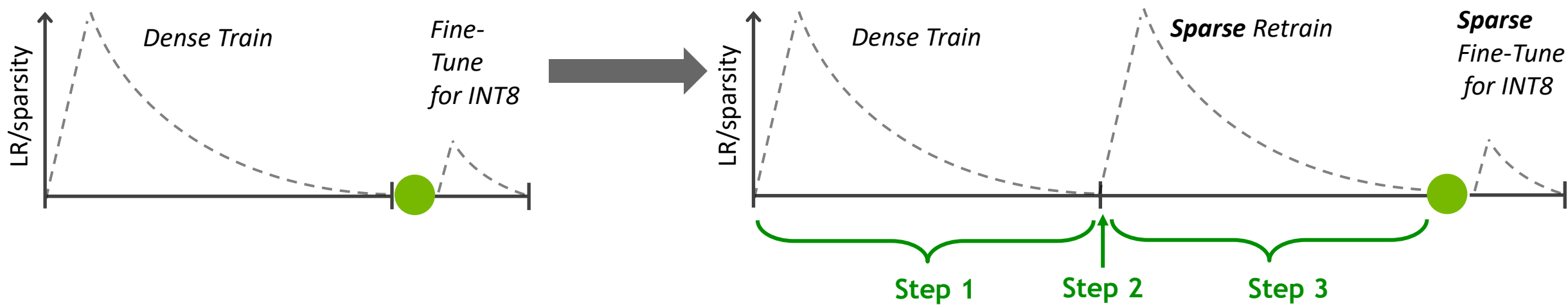
Post-training calibration follows the sparse fine-tuning



SPARSITY AND QUANTIZATION

Quantization Aware Training

Fine-tune for sparsity before fine-tuning for quantization





ACCURACY EVALUATION

ACCURACY

Overview

Tested 47 public networks, covering a variety of AI domains, with the described recipe

No channel permutations, except as noted

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:

1st: Retrain a sparse FP16 network first

2nd: 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	Dense FP16	Accuracy			
		Sparse FP16		Sparse INT8	
ResNet-50 (SWSL)	81.1	80.9	-0.2	80.9	-0.2
ResNeXt-101-32x8d (SWSL)	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	-

WSL = Weakly Supervised Learning
SWSL = Semi-Weakly Supervised Learning

IMAGE CLASSIFICATION

ImageNet

Network	Dense FP16	Accuracy			
		Naïve 2:4 Sparse FP16		Permuted 2:4 Sparse FP16	
MobileNet v2	71.6	69.6	-2.0	71.6	-
SqueezeNet v1.0	58.1	54.1	-4.0	58.4	0.3
SqueezeNet v1.1	58.2	57.0	-1.2	58.2	-
MNASNet 1.0	73.2	72.0	-1.2	73.3	0.1
ShuffleNet v2	68.3	67.0	-1.3	68.4	0.1

Small, parameter-efficient networks may see accuracy loss without extended fine-tuning.

By using channel permutations to retain more important weights, such networks can be pruned with no loss in accuracy using the simple recipe.

Permutations may also help shorten the fine-tuning samples needed to recover accuracy: finding a consistent and universal method is future work.

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

GENERATIVE NETWORKS

Variety of GAN Architectures and Tasks

Network	Task	Dataset	FID Score (lower is better)		
			Dense FP16	Sparse FP16	
DCGAN	Image Synthesis	MNIST	50.4	50.5	0.1
Pix2Pix	Domain Translation	Satellite -> Map	17.6	17.9	0.3
Pix2Pix	Domain Translation	Map -> Satellite	30.8	30.7	-0.1
CycleGAN	Style Transfer	Monet -> Photo	63.2	63.0	-0.2
CycleGAN	Style Transfer	Photo -> Monet	32.0	32.4	0.4
CycleGAN	Image-Image Translation	Zebra -> Horse	60.9	61.0	0.1
CycleGAN	Image-Image Translation	Horse->Zebra	52.9	52.5	-0.4
StarGAN	Image-Image Translation	CelebA	6.1	6.9	0.8
SRGAN	Super Resolution	DIV2K	14.7	16.6	1.9

QUALITATIVE RESULTS: STARGAN

Image-Image Translation (CelebA)

Dense



2:4
Sparse



NLP - TRANSLATION

EN-DE WMT'14

Network	Metric	Dense FP16	Accuracy			
			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
Levenshtein Transformer	Validation Loss	6.16	6.18	-0.2	6.16	-

NLP - LANGUAGE MODELING

Transformer-XL, BERT

Network	Task	Metric	Dense FP16	Accuracy			
				Sparse FP16	Dense INT8	Sparse INT8	
Transformer-XL	enwik8	BPC	1.06	1.06	-	-	-
BERT-Large	SQuAD v1.1	F1	91.9	91.9	-	90.9	90.8 -0.1



ASP: AUTOMATIC SPARSITY FOR RETRAINING IN PYTORCH

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 2:4 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('model.pth')) # Load model

optimizer = optim.SGD(model.parameters()) # Define optimizer

ASP.prune_trained_model(model, optimizer) # Prune model

x, y = DataLoader(...) #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!

GENERATE A STRUCTURED SPARSE NETWORK

Channel Permutations by Default

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
```

Coming soon:
prune_trained_model() will
efficiently search for a good
permutation and apply it to your
network



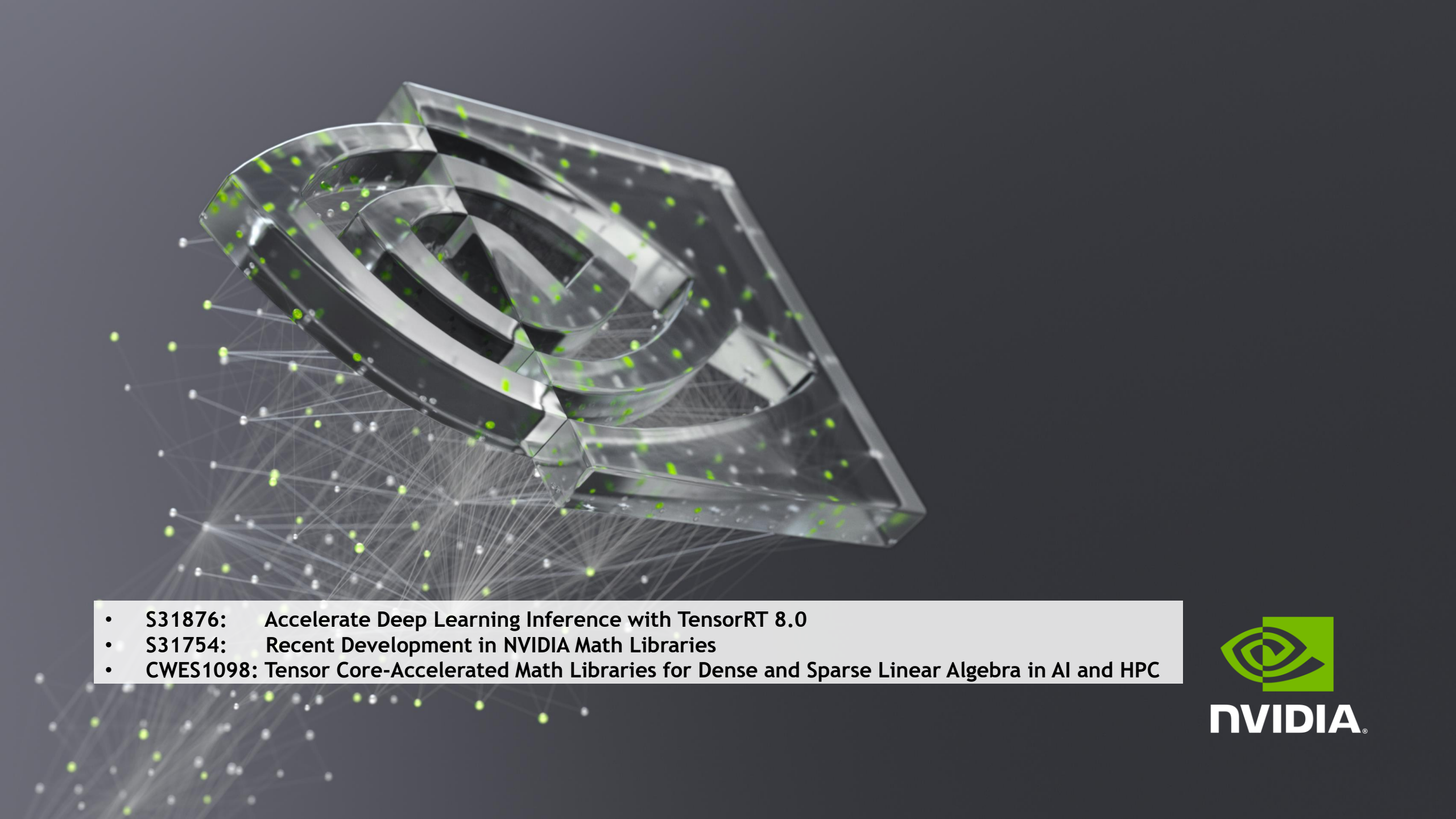
SUMMARY

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, permute+prune, re-train sparse
 - Across many tasks, networks, optimizers
- Fast with the NVIDIA Ampere Architecture's Sparse Tensor Cores
 - Up to 2x in GEMMs with cuSPARSELt
 - TensorRT 8.0 supports convolutional networks OOTB

- 
- S31876: Accelerate Deep Learning Inference with TensorRT 8.0
 - S31754: Recent Development in NVIDIA Math Libraries
 - CWES1098: Tensor Core-Accelerated Math Libraries for Dense and Sparse Linear Algebra in AI and HPC

