Sparse Tensor Core:

Algorithm and Hardware Co-Design for Vector-wise Sparse Neural Networks on Modern GPUs

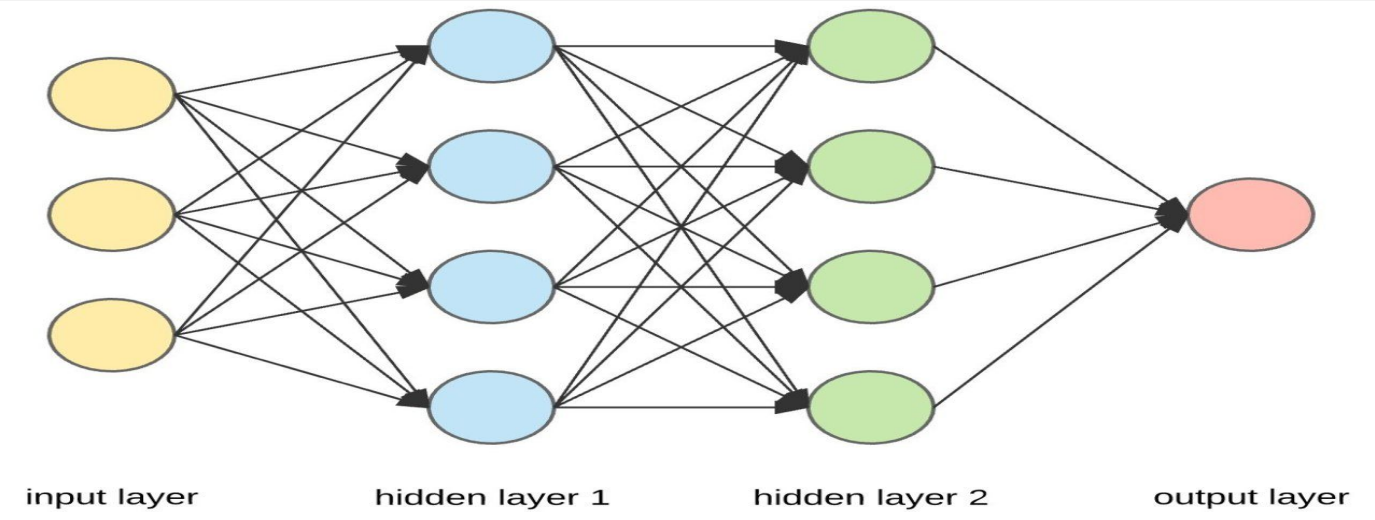
James Kelly, Tanay Kulkarni, Ashvin Kumar, Yu-Fan Teng, Ian Stewart

Agenda

1. Introduction
2. Background and motivation
3. VectorSparse pruning
4. Sparse Tensor core
5. Experimental Methodology
6. Experimental Results
7. Conclusion

Introduction

Neural Network

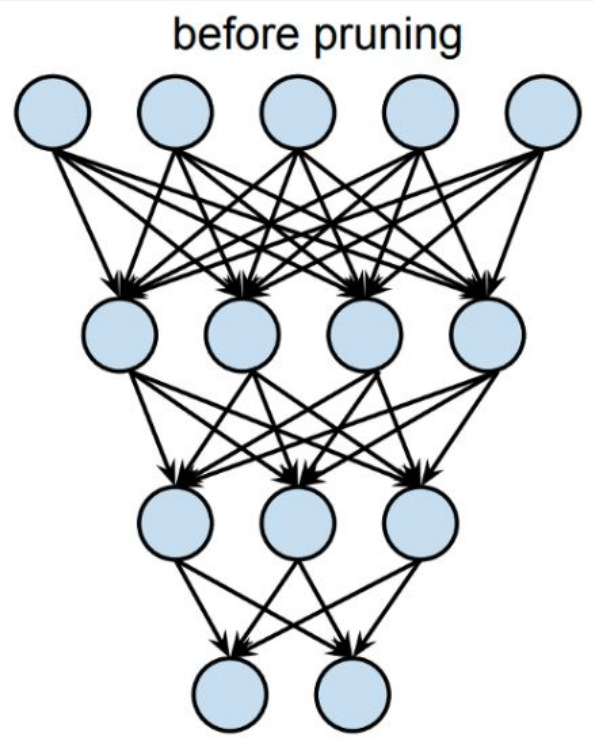


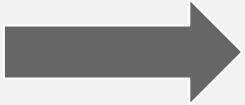
- Interconnected layers of neurons (visible and hidden)
- Inputs * Weights -> Sum -> Activation function

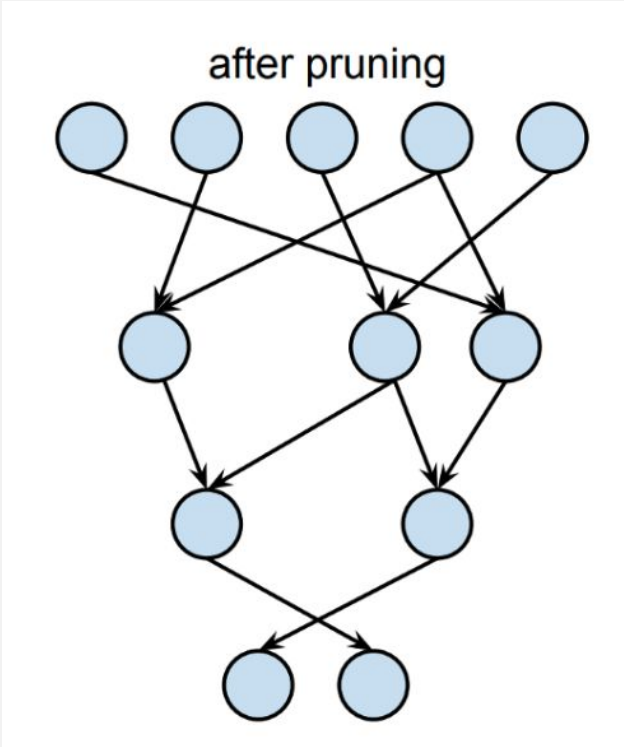
NN continued...

- Powerful solutions for complex applications
  - Image classification, Speech recognition, etc.
- Enormous computational requirements due to overparameterisation
  - Required for associative memory
  - Large number of multiply-accumulate (MAC) operations
- Improve efficiency?
  - Pruning
    - Rank and remove nodes
    - Retain accuracy
    - Leverage intrinsic redundancies

Pruning of NN







Effects of Pruning

- Sparsity in network
  - Reduced number of nodes
  - Irregularity in spacing
- Improved performance
- Problem?
  - Underutilization!
    - Significant accuracy drops on modern Graphics Processing Units (GPUs)
    - Optimisations for dense networks
      - e.g. Tensor Core, General Matrix Multiplication (GEMM)

Novel Approach

- Algorithm + Hardware Co-Design for improved sparse NN performance
- **Vector Sparse -**
  - New sparsifying technique
    - Single Instruction Multiple Data (SIMD)-friendly
    - “Vector-wise”
      - Divides matrix into individual vectors
  - Extend instruction set to Volta architecture
    - Usable on Tensor Core
- **Sparse Tensor Core -**
  - Tensor Core hardware modification

Background & Motivation

Sparsity-Centric Optimization for DNNs

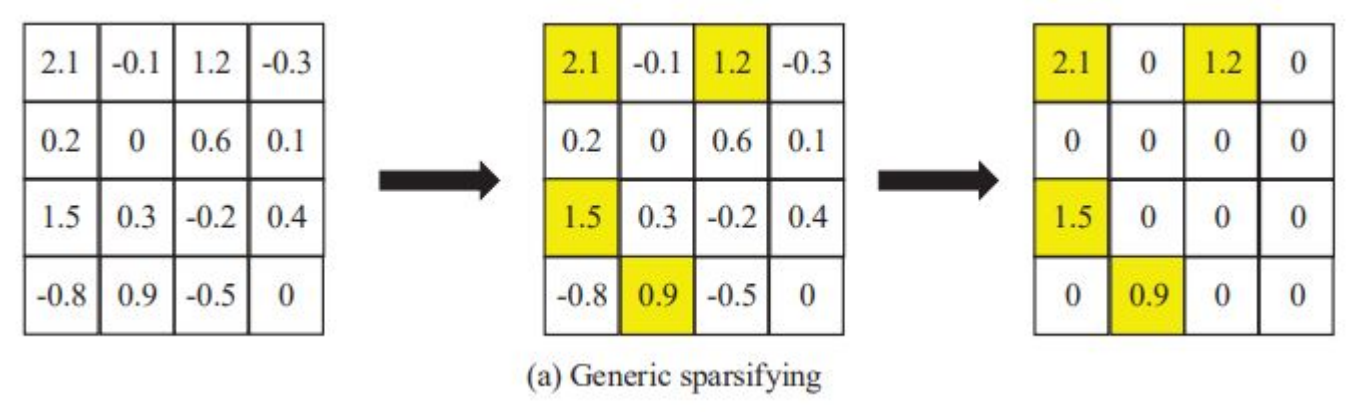
- Sparsity results from redundancy in neural network parameterization. This sparsity has the potential to be exploited for better computational efficiency and performance on GPUs.
- However, prior sparsifying techniques have several flaws. The top-k pruning method achieves compression but is unable to realize performance increases, whereas the structural sparsifying method increases performance but introduces inaccuracies.
- The authors seek to realize the best of both of these benefits.

Definition: Dynamic vs. Static Sparsity

- Static sparsity refers to the sparsity in the weight parameters, as it does not depend on the input data.
- Dynamic sparsity refers to the sparsity in the activations of each layer, as this depends on both the weights and the input data.
- This paper focuses on static sparsity.

Generic Sparsifying

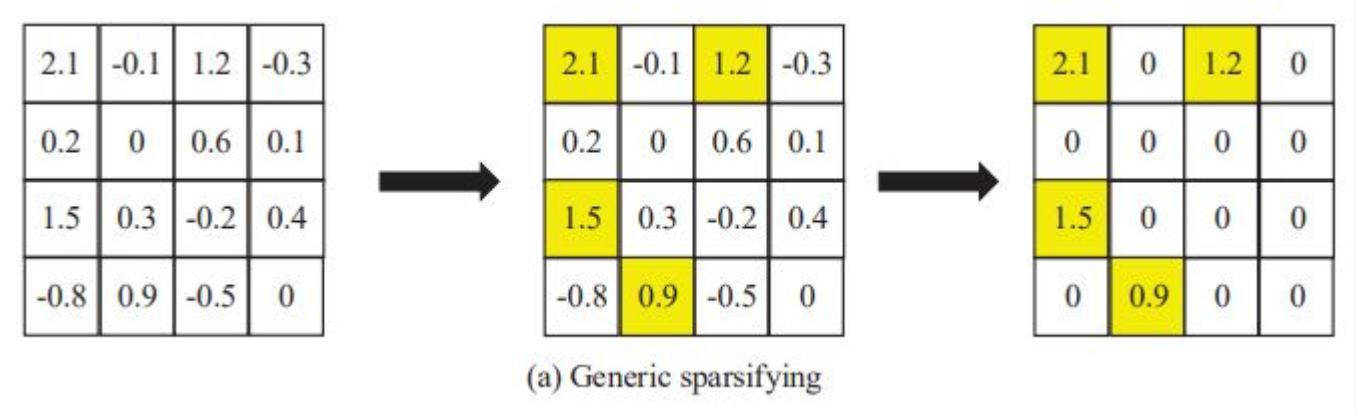
- Select the largest k values by absolute value, and leave them unchanged
- Force the other values to zero
- In this example, 75% sparsity for 4x4 matrix



(a) Generic sparsifying

Generic Sparsifying

- Several inefficiencies are exposed on GPUs:
  - Variation of row/column length impacts workload partitioning
  - Number of non-zero elements unknown until runtime
  - Not enough computation to hide memory access latency, therefore undoing sparsity benefit



(a) Generic sparsifying

Generic Sparsifying: Performance

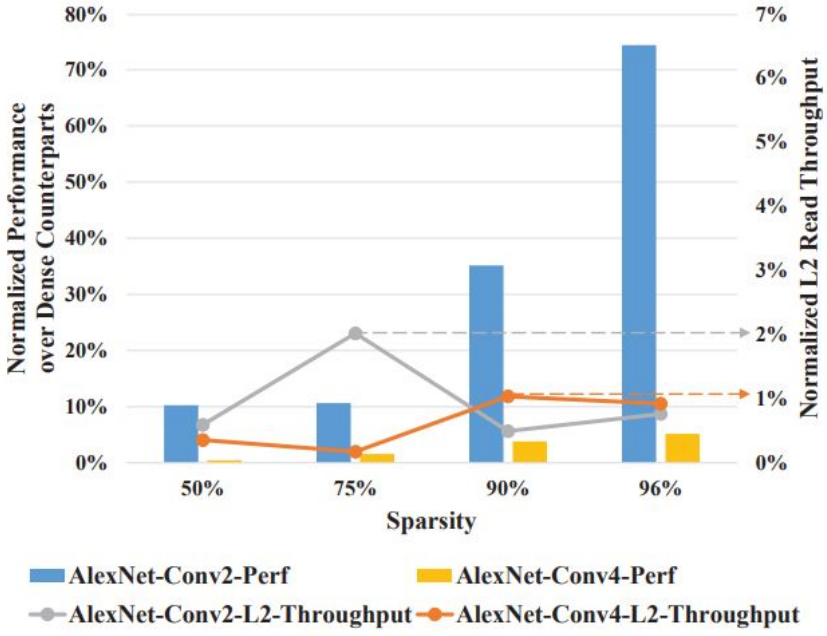


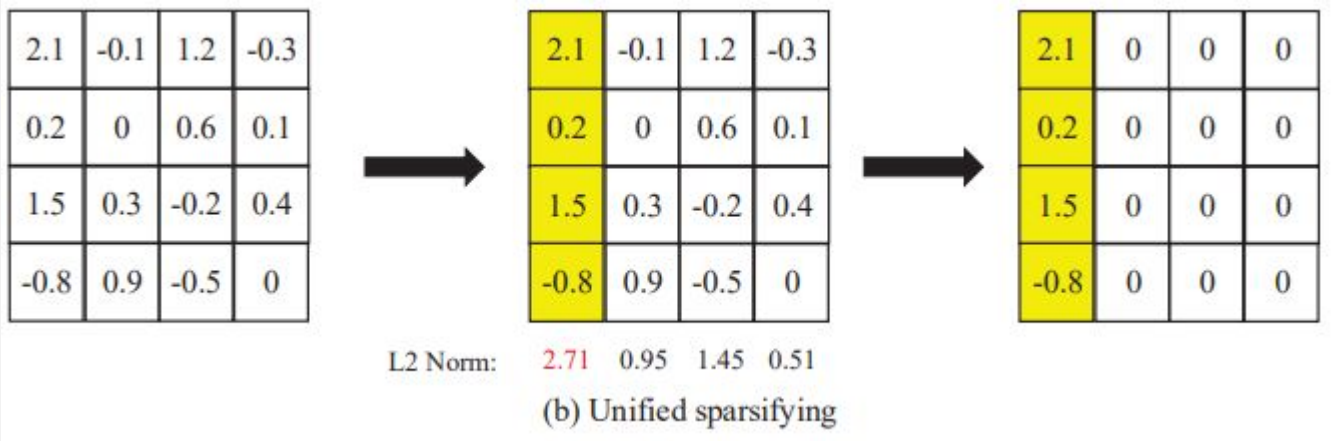
Figure 2: The normalized performance and L2 cache throughput of generic sparse CONV layers over dense CONV layers on a Tesla V100 GPU.

- Two sparse convolutional layers tested
- The sparse layers are less performant than dense layers
- Low L2 cache read throughput implies memory underutilization, meaning computation is a limiting factor

- Details: This example used Conv2 and Conv4 AlexNet layers on NVIDIA Tesla V100 GPU. Layers converted to GEMM with im2col transformation. Sparse layers based on CUSPARSE library. Normalized to dense CUBLAS layers.

Unified Sparsifying

- In this example, 75% sparsity achieved by keeping only one column
- This column was selected as the one with the largest L2 norm
  (Reminder : L2 norm = distance from origin or in this case, zero-vector)
- Inaccuracies from removing key elements that generic method keeps.



(b) Unified sparsifying

Characterization of Sparsity

- Goal: Characterize spatial locality of non-zero values
- Let Vector V(*y*,*x*) contain the elements with row index *y*, and column indices from *x* ✖ L to (*x* +1) ✖ (L-1) in the M✖N matrix W.
- Zeros are added if N is not divisible by L.
- Number of zeros in the vector in the range [0,L] divided by L gives the local sparsity degree of each vector.

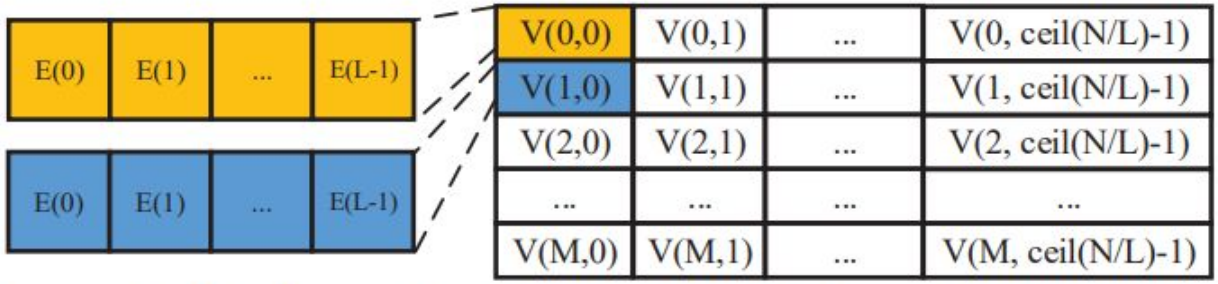


Figure 3: Dividing a $M \times N$ matrix into L-dim vectors for locality characterization.

Cumulative Distribution vs. Local Sparsity (ResNet-18 & NMT)

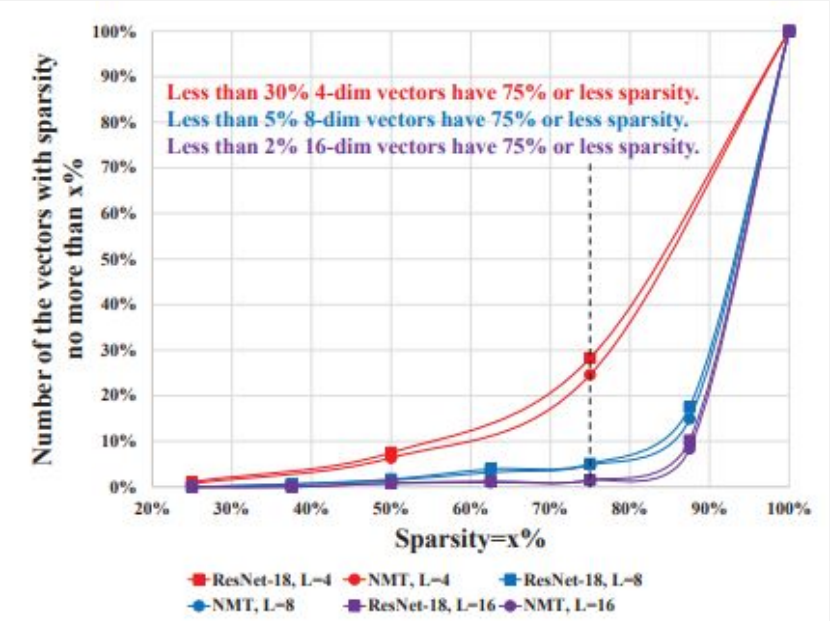


Figure 4: The cumulative distribution of the vectors vs. the local sparsity degree in a vector. The vertical axis is the number of the vectors with the sparsity not greater than x%. The horizontal axis is the sparsity x%.

- Only less than 30% of 4-dim vectors have ≤ 75% sparsity. Very few have more than 2 non-zero elements.
- Similarly, most 8-dim and 16-dim vectors have ≥ 75% sparsity.
- Less low sparsity vectors as the vector size increases, as the larger scope approaches the global sparsity.
- This inspires us to split the weight matrix into L-dim vectors and sparsify each one independently.

VectorSparse Pruning

Taking Advantage of Spatial Locality

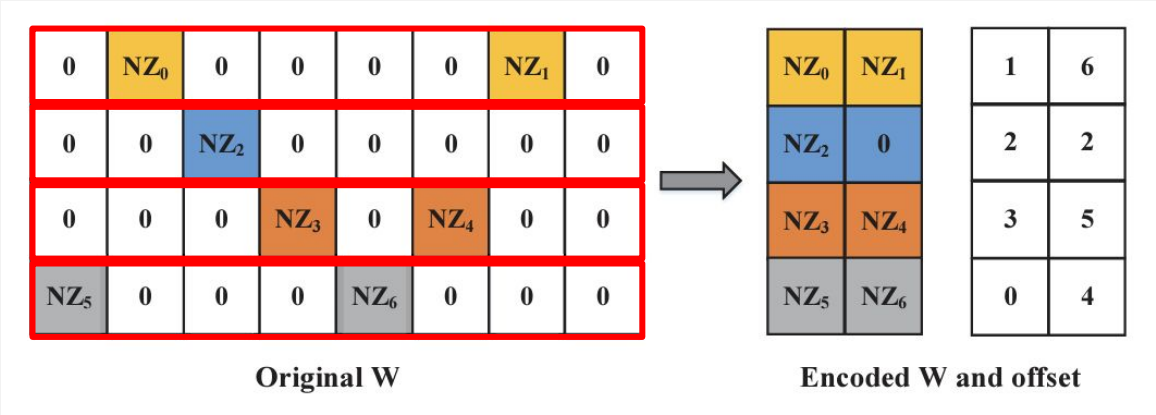
- We can take advantage of local sparsity to avoid the accuracy penalty of previous sparsifying methods
  - Split the weights matrix into vectors, and sparsify each vector independently
- Two parts to this solution:
  - Vector-wise encoding format for sparse matrices that simplifies workload partitioning on GPUs
  - VectorSparse algorithm to prune and retrain CNNs and RNNs

Vector-wise Sparse Matrix Encoding

1. Divide W into L-dim vectors
   - An M x N matrix will result in M x⌈N/L⌉ vectors
2. Count the maximum number of non-zero elements in each vector (K)
3. Compress each L-dim vector into a K-dim vector

L = 8

K = 2



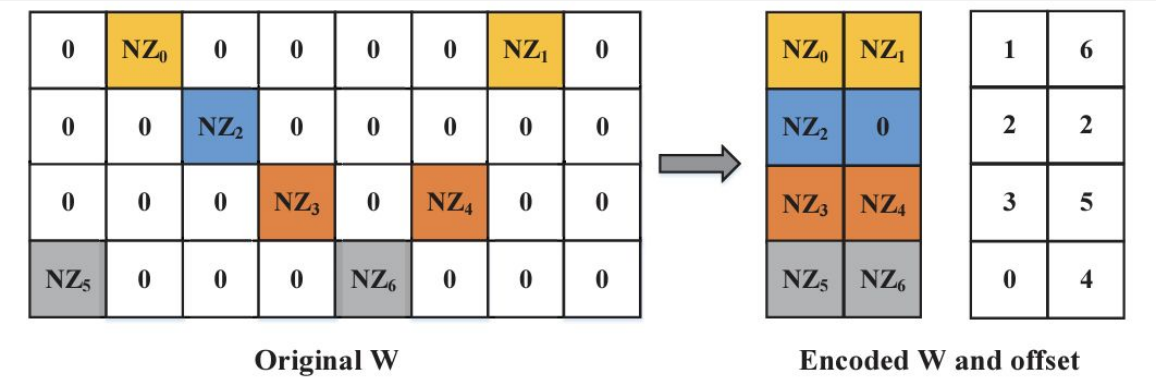
Vector-wise Sparse Matrix Encoding

- Encoding each index in a L-dim vector requires $\lceil \log_2 L \rceil$ bits
- Assuming each element takes P bits to store, each L-dim vector takes up:
  - P x L bits before compression
  - (P + $\lceil \log_2 L \rceil$) x K bits after compression

L = 8

K = 2

P = 16



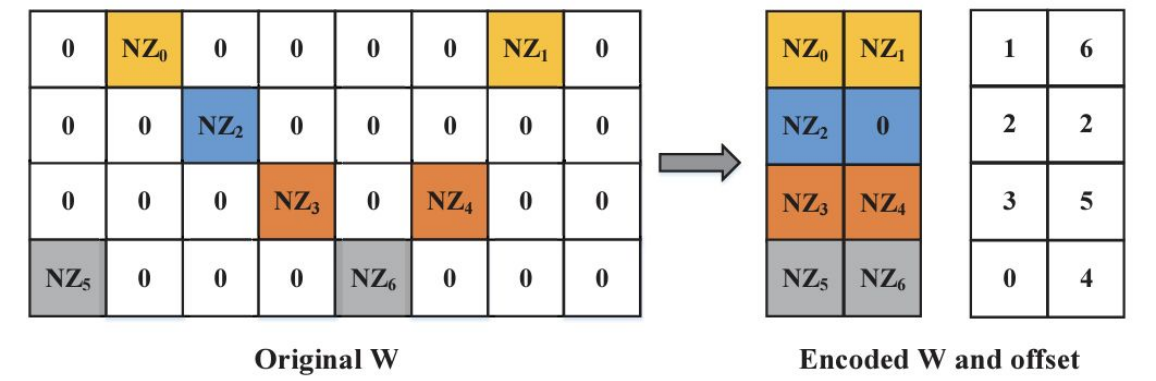
Vector-wise Sparse Matrix Encoding

- In order to achieve an ideal compression ratio, each L-dim vector should have the same number of non-zero elements
  - VectorSparse neural network pruning lets us adjust the topology of weights to achieve this!

L = 8

K = 2

P = 16



Implementing VectorSparse

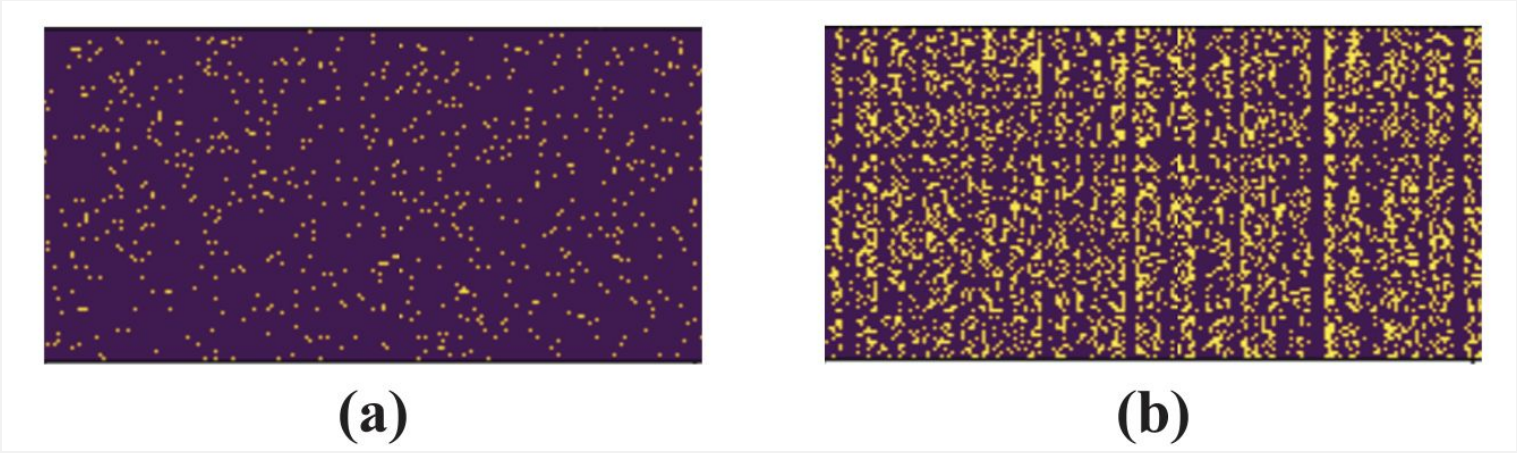
- VectorSparse is a pruning methodology for CNNs and RNNs used to realize a spatially even distribution of non-zero elements
  - Iteratively sparsify and retrain the weight matrices
- For convolutional layers, the learned filters are “unrolled” to produce a weight matrix that transforms the convolution operation into a regular matrix multiplication

Implementing VectorSparse

1. Split the weight matrices into L-dim vectors
2. The largest K elements in each vector are permitted, and the other elements are removed
3. The sparsified network is fine tuned
4. K is gradually reduced until the validation error falls below the maximum allowed error, E_δ

```
1  W=W0;
2  Divide W into vector_{i,j};
3  N_zero=0;
4  E0 = ValidationError(W);
5  E = E0;
6  while (E-E0)/E0 < Eδ do
7      N_zero=N_zero + 1;
8      for each i, j do
9          Sort absolute values of all elements in vector[i][j] in
             ascending order and save sorted elements in
             sorted[L];
10         T_ij=sorted[N_zero];
11         for each element in vector[i][j] do
12             Remove element if abs(element) < T_ij;
13         end
14     end
15     Fine tune the pruned W;
16     E = ValidationError(W);
17 end
18 Ws=W;
```

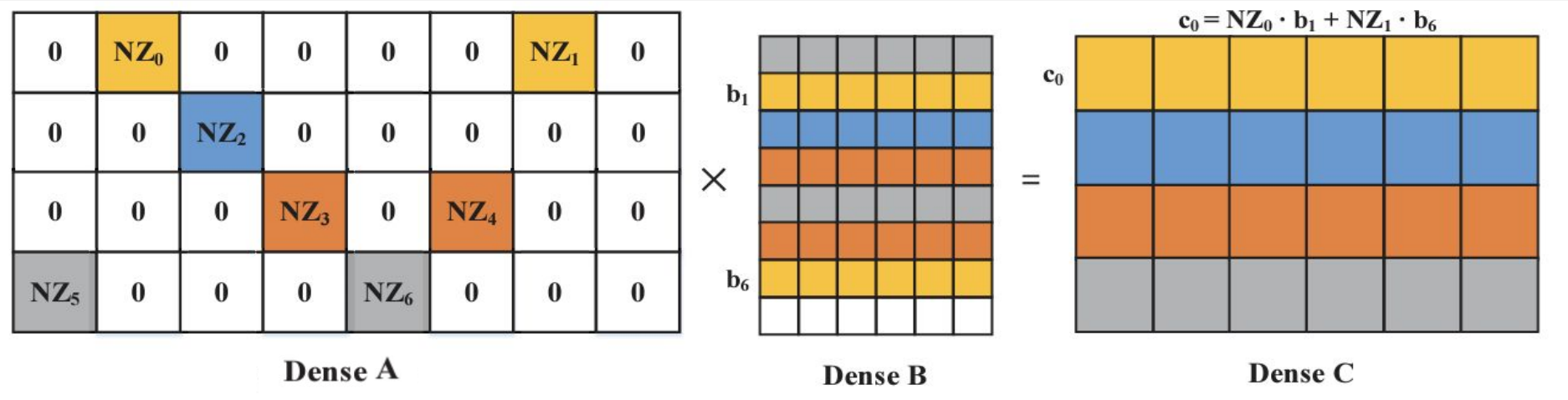
Visualizing VectorSparse



The spatial distribution of non-zero weights in neural networks pruned by (a) the generic method with 96% sparsity and (b) the vector-wise approach with 75% sparsity, respectively. Each yellow pixel represents a non-zero element.

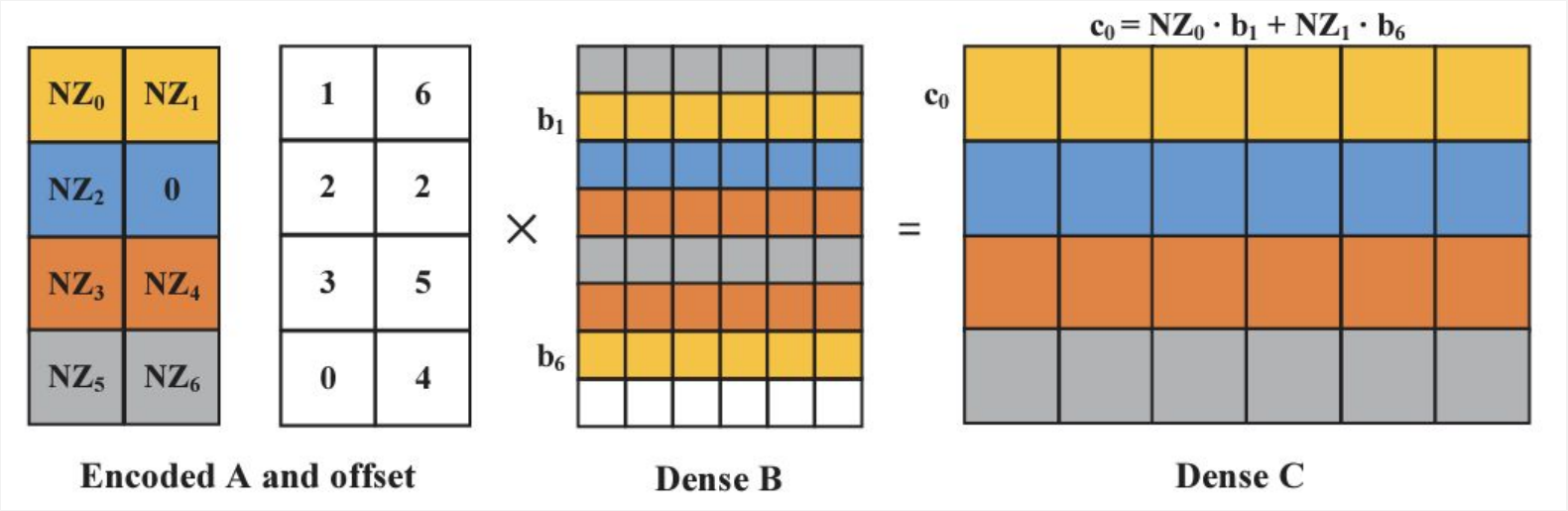
GPU Multiplication Kernel

- Let's assume a general matrix multiplication, C = A x B
- Using dense matrices, the product of every row of A and every column of B needs to be computed



GPU Multiplication Kernel

- Once the A matrix has been encoded using VectorSparse, only a subset of those calculations need to be completed
- 75% multiplication reduction in given example



GPU Multiplication Kernel

- The proposed matrix multiplication kernel is compatible with other common computation techniques for calculating matrix products on massively parallel hardware
  - Key example: tiling

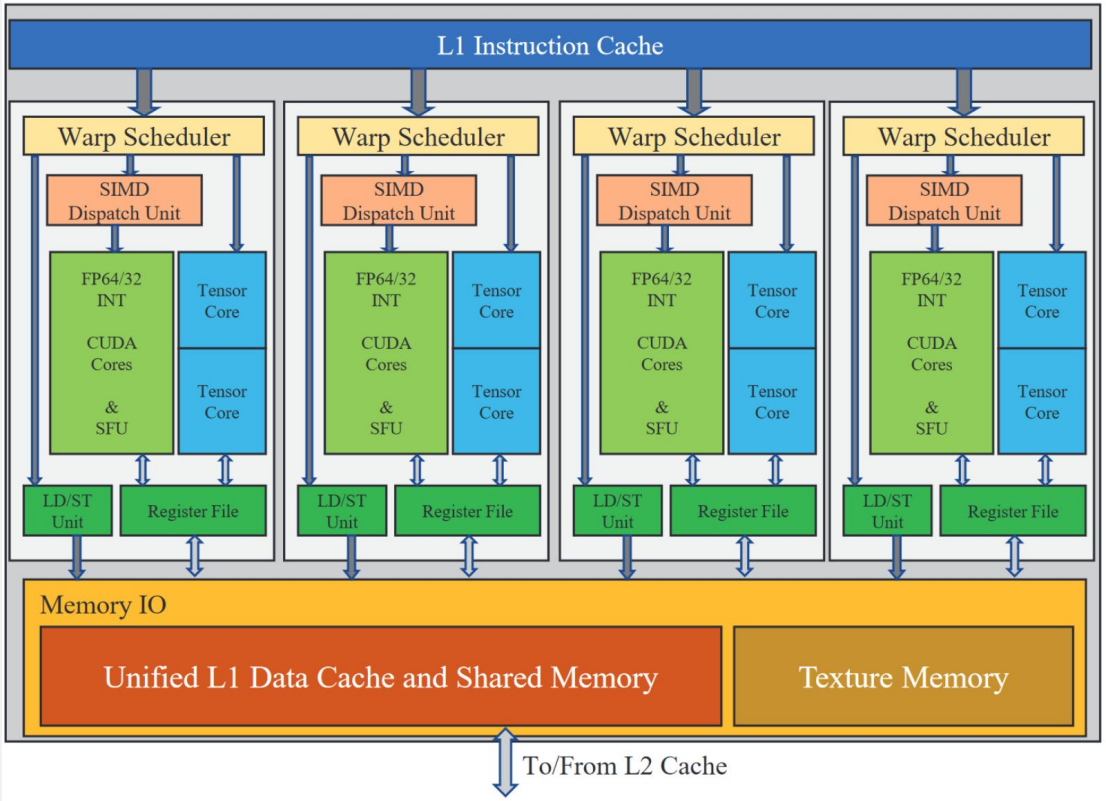
Sparse Tensor Core

What is a “Tensor Core”?

- Tensor Core is a hardware block in NVIDIA GPUs
  - Optimized for dense matrix MAC operations
    - Able to execute a 4x4x4 matrix MAC in one cycle
  - Can operate as 16-bit floating point or 32-bit mixed precision

- Lives within the steaming multi-processors
  - Each SMP has four subcores, each with two Tensor Cores
  - Both of these cores used by one “warp”
    - Each warp computes one tile of the product matrix

Streaming Multi-Procesor Diagram



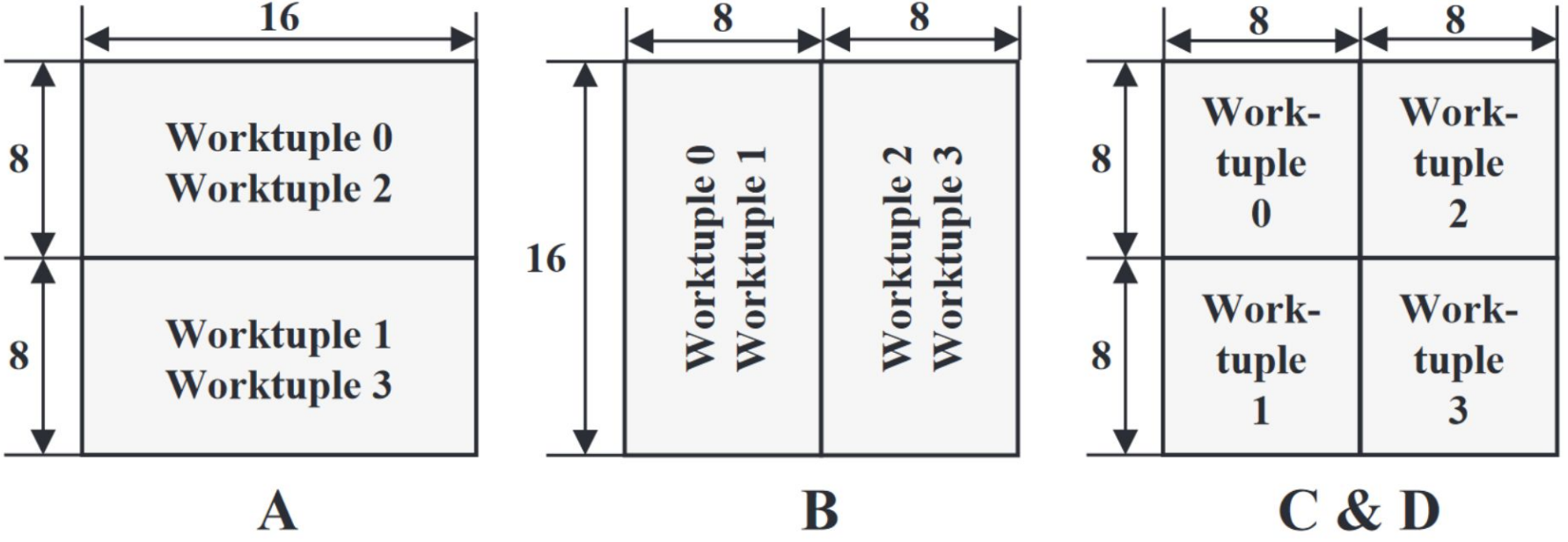
How does the actual math work?

- Each warp computes a matrix multiply-and-accumulate function
  - Paper generalizes this as **D**=**A**×**B**+**C** where all four are 16×16

- Warps broken up into 32 threads
  - Each thread put into one of eight sequential threadgroups
    - Work together to compute 4x4 tile multiplications

- Every threadgroup is in a worktuple with another threadgroup
  - Spaced out by 4 - ie, threadgroups 1 and 5 form a worktuple

Mapping of 16×16×16 matrix multiplication into 4 worktuples in a warp



How is this carried out?

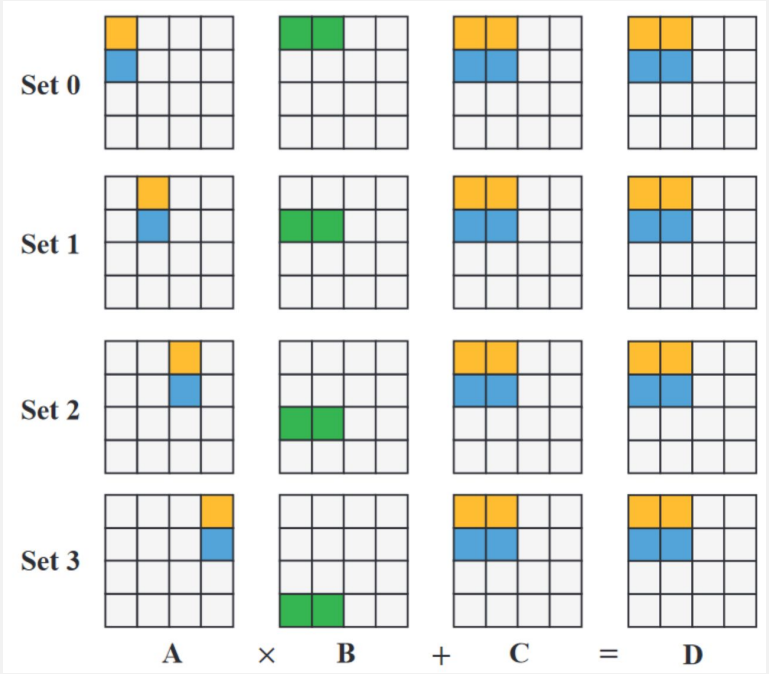
- Warp Matrix Multiply and Accumulate instructions (WMMA) in API
  - One WMMA instruction → four sets of HMMA instructions
  - Occurs at compile-time

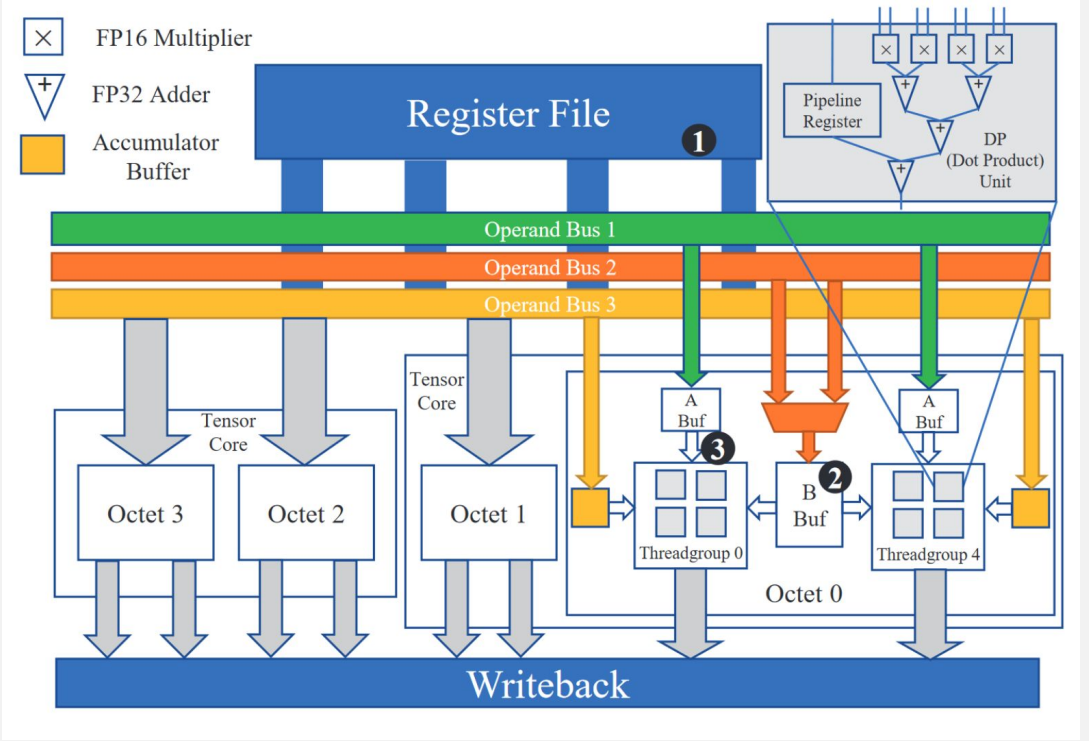
- Each set of instructions calculates one 4x8 tile of **D** in 8 cycles

- Each TC has eight dot product (DP) units per worktuple
  - Each can compute a 4-dimensional vector dot product per cycle
  - Each threadgroup takes 4 DP units

- Although buffers for **A** and **C** are unique, the buffer for **B** is shared

What does that look like?



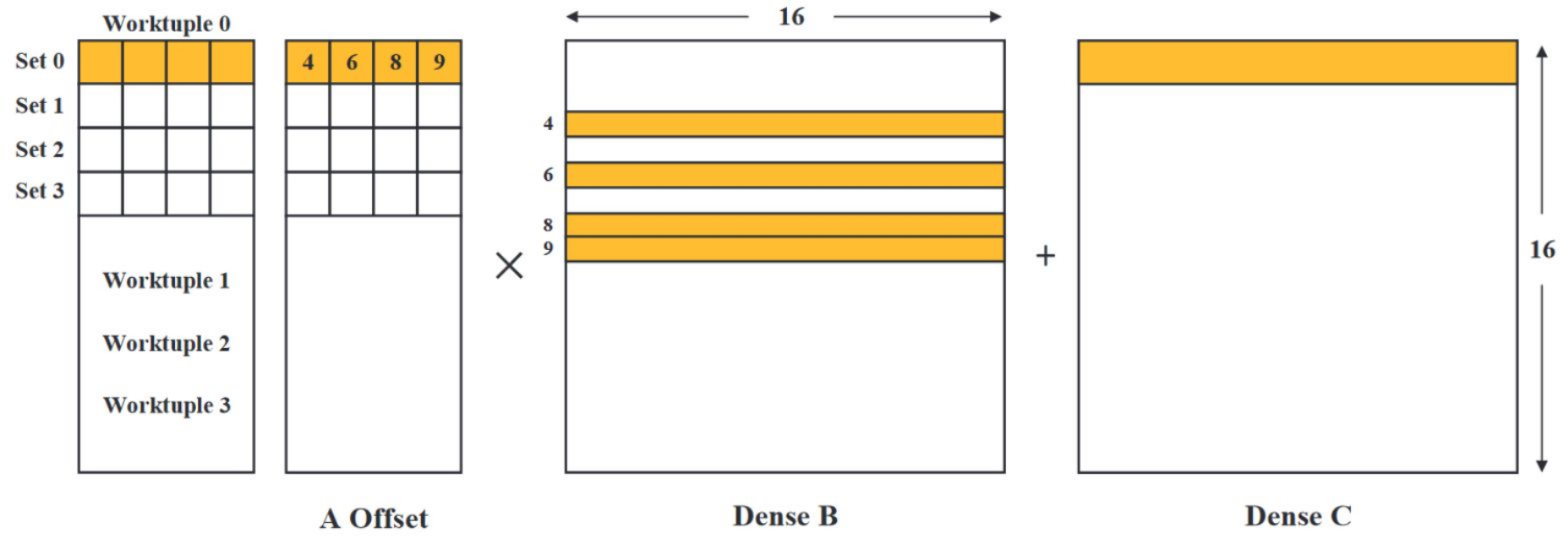


How does this need to change for VectorSparse?

- Creation of new *vector-wise sparse mode*
  - Current mode now referred to as "*dense mode*"

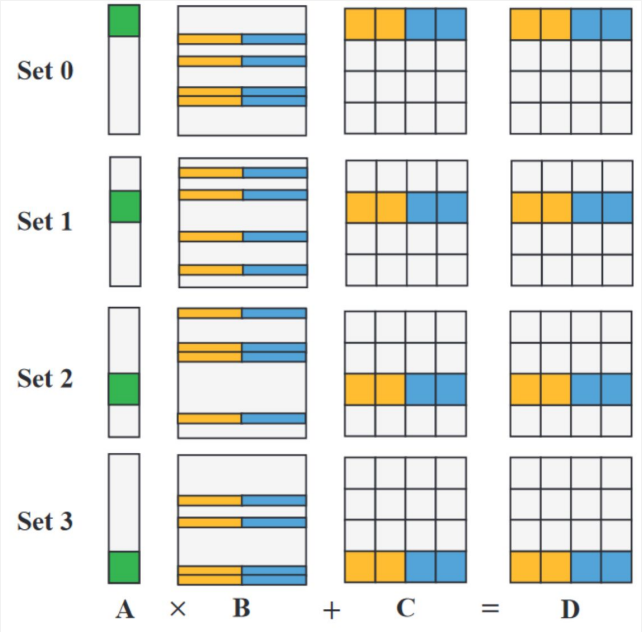
- Sets K to 4 for L of 16, still calculates 16x16 MMAC at warp level
  - Each worktuple gets four rows of the encoded **A**
  - Store offsets in a new special register

Sparse HMMA Tensor Core Execution Flow



VectorSparse Implementation Details

- Now both threadgroups in a worktuple calculate the same row of **A**
  - Rows of **B** to multiply by determined via the offset register



Sparse HMMA Instructions

- Two new SHMMA instructions added
  - One to store offset indices in implicit, dedicated offset register
  - One to decode the offset register, fetch rows of **B**, and calculate
  - These instructions form a new set, must work together in order

- Four sets of SHMMA instructions to complete the WMMA instruction

- Requires hardware modification to the Tensor Core

Sparse Tensor Core Modifications

- Add dedicated offset registers to register file
  - Can only be implicitly accessed by SHMMA instructions

- Double operand **B** buffer size and add a second one to hide latency
  - Total increase of 4x over standard buffer size

- Enable the broadcasting of operand buffer **A** to its four DP units

Theoretical Results

- In dense mode, one 16x16x16 GEMM takes 40 cycles
- In sparse mode, one 16x16x16 computation takes only 26 cycles
  - 1.54x speedup over dense mode
- With the quadrupled B buffer mentioned earlier, it's now 20 cycles
  - An additional 1.3x speedup
  - Technically also benefits dense mode to have this
    - Dense mode is compute-bound so speedup minimal
    - Probably not worth the die area for dense mode alone

Experimental Methodology

Methodology

- Trained Neural Networks on **3 domains** to evaluate model accuracy and performance (speedup)
  - Image classification
  - Image captioning
  - Machine translation
- Trained Neural Networks with **3 sparsifying methods**
  - Generic
  - Unified
  - VectorSparse (Proposed method)

Methodology

Domain	Model	Dataset	Evaluation	Sparse method
Image Classification	AlexNet, VGG-16, ResNet-50, ResNeXt-50	ImageNet	Accuracy	Trained Neural Networks with the following methods respectively: 1. Generic 2. Unified 3. VectorSparse
Image Captioning	Show and Tell (Inception V3 + LSTM)	MSCOCO	BLEU	
Machine Translation	NMT (2 LSTM + 4 LSTM + Attention)	WMT 16 English-German	BLEU	

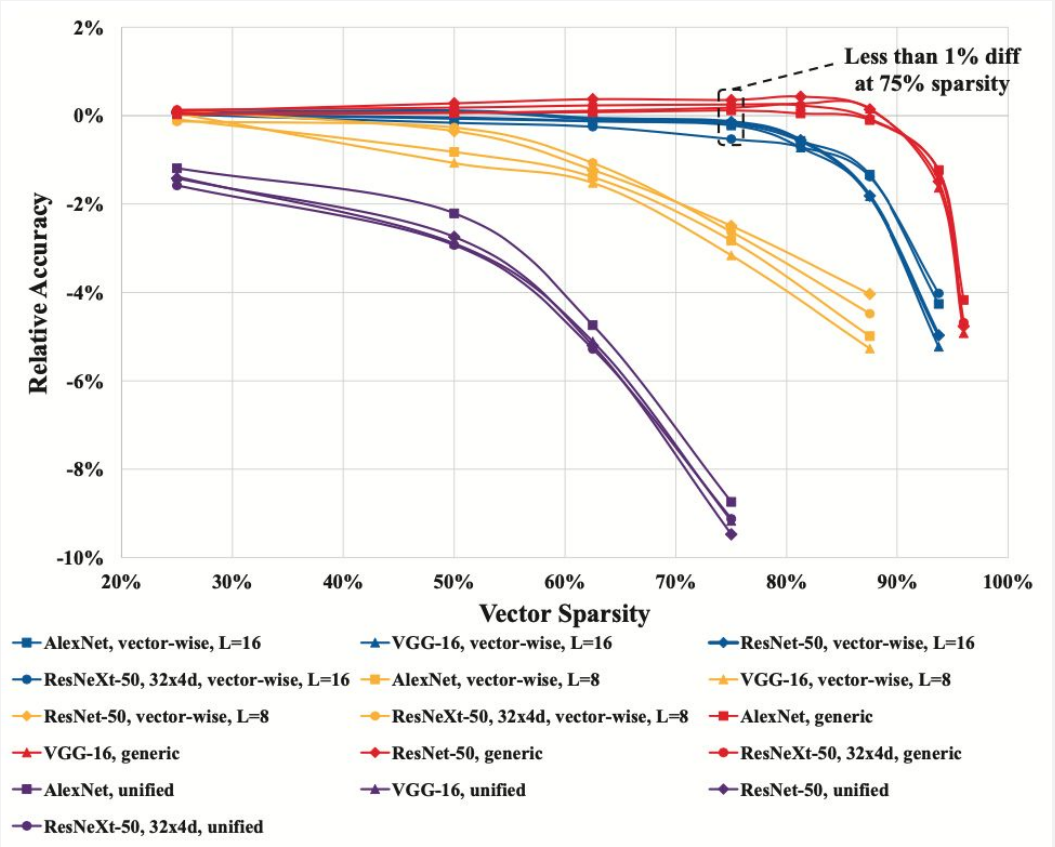
Environment & Implementation

- Ran the workloads under **6 different configurations** to evaluate performance
  - CUDA Core
    - No pruning method (baseline)
    - Generic (96% sparsity)
    - Unified (50% sparsity)
    - VectorSparse (75% sparsity)
  - Tensor Core
    - No pruning method
    - VectorSparse (75% sparsity)

Experimental Results

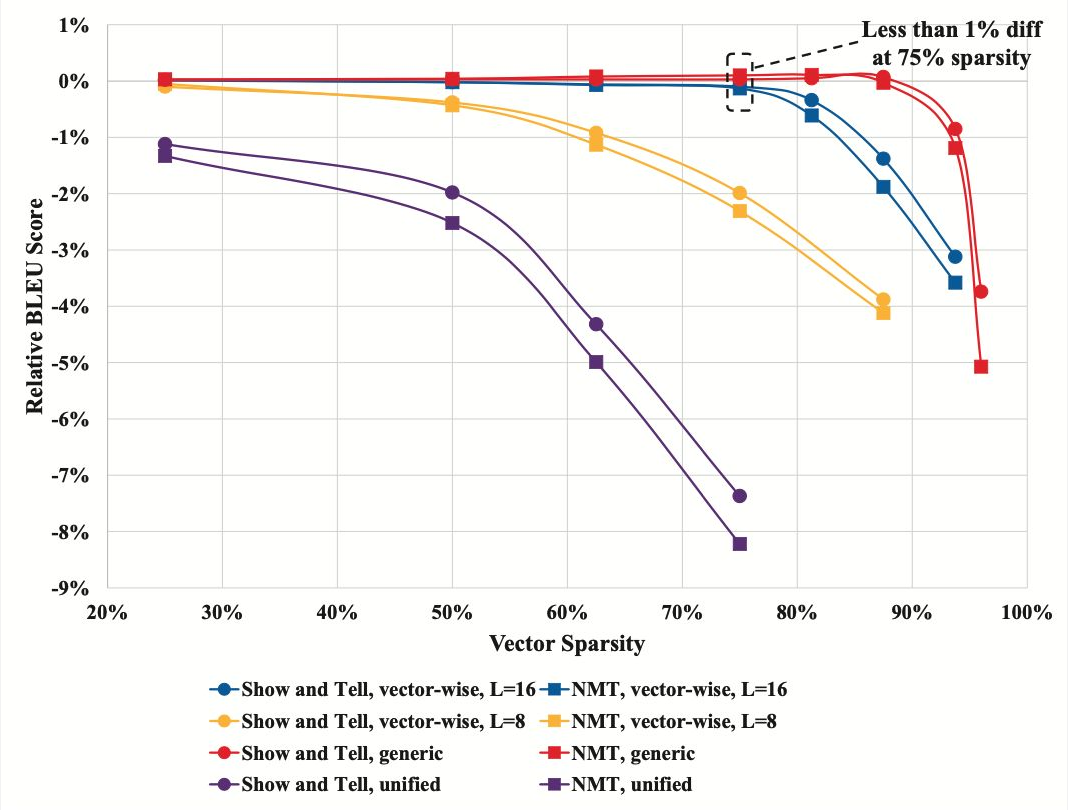
Accuracy Evaluation

- CNN
  - Image Classification

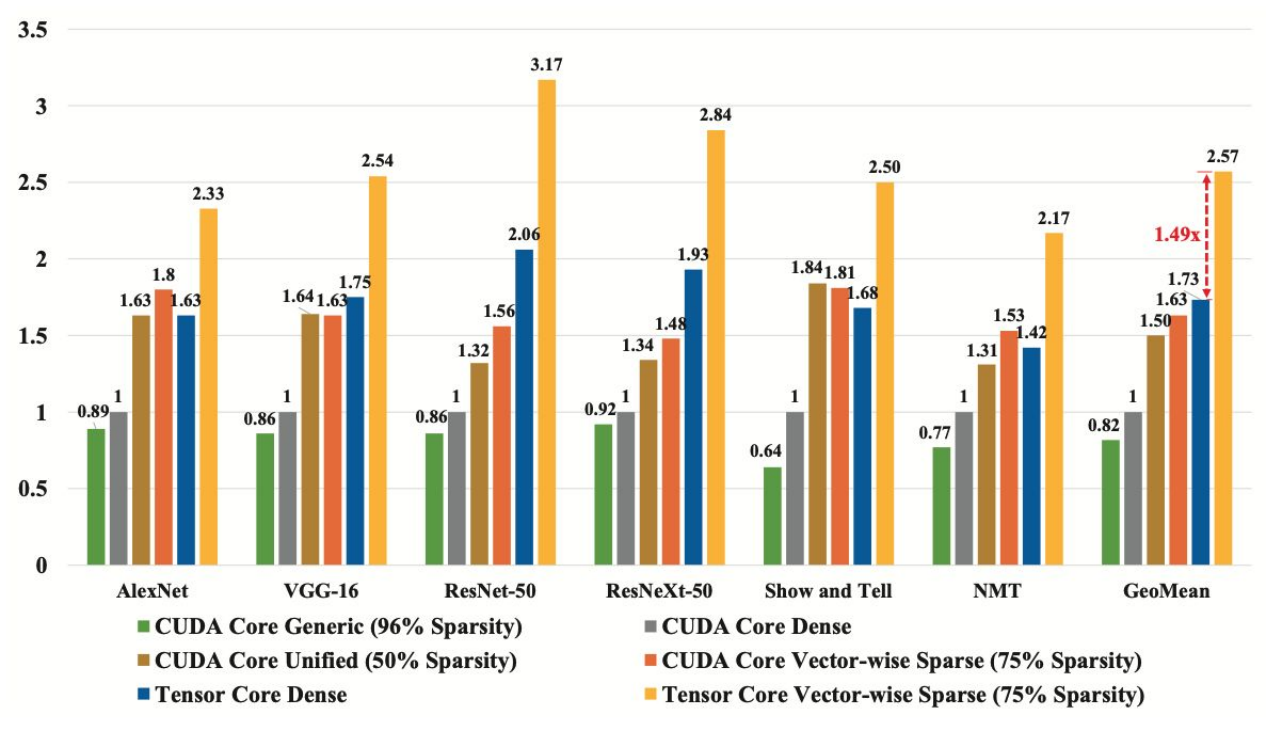


Accuracy Evaluation

- RNN
  - Image captioning
    - Show and Tell
  - Machine Translation
    - NMT



Performance Evaluation



Design overhead analysis

- Concern: adding VectorSparse method requires 4X large buffer (~4KB SRAM)
  - May cause design overhead on timing and area
- Use CACTI7 to evaluate the timing and area overhead on Tensor Core
- CACTI7 Result:
  - Timing and Area can be **negligible**
    - since V100's cycle period is 0.65 ns and area is 815 mm^2

Process	SRAM Size	Area	Cycle Time
22nm	4KB	0.019 mm^2	0.4ns

Conclusion

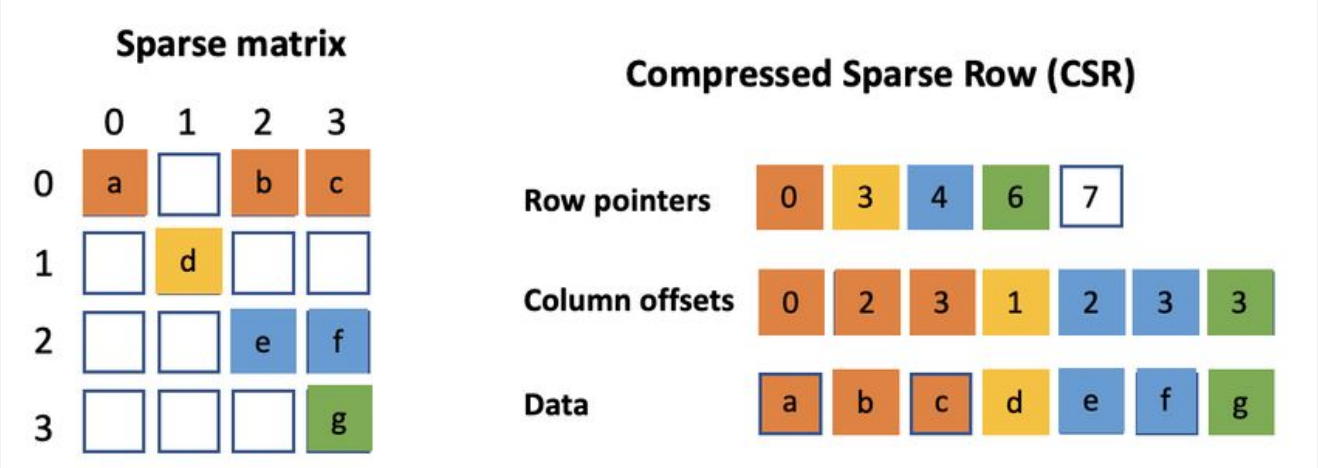
Summary and Conclusion

- Vector Sparse (Vector-wise) pruning
  - Exploit redundancy, Good workload balance
  - Enables sparse NN to run on GPUs
  - Allows uniform row splitting for thread parallelism
- Tensor Core - Extended instruction set and hardware improvements
  - Improved matrix multiplications, negligible area overhead
- Best performance - L = 16, 75%
- Hardware support for performance improvement
- Improvements -
  - Pruning Algorithm - 63%
  - Hardware modification - 58%

Questions?

Extra: Compressed Sparse Row (CSR) Format

- Encoding formats for generic sparse matrices don't contain information associated with the vectors necessary for VectorSparse pruning



Extra: Matrix Tiling