EECE695D: Efficient ML Systems

Pruning

(part 2)

Pruning without structure

$$\begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix}$$ 32bits x 4 = 128bits

$$\begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix} = \begin{bmatrix} a_1b_1 + a_2b_3 & a_1b_2 + a_2b_4 \\ a_3b_1 + a_4b_3 & a_3b_1 + a_4b_4 \end{bmatrix}$$

8 Multiplications, 4 Additions

$\downarrow$

$$\begin{bmatrix} a_1 & 0 \\ 0 & a_4 \end{bmatrix}$$ 32bits x 2 + α = 64bits + α

$$\begin{bmatrix} a_1 & 0 \\ 0 & a_4 \end{bmatrix} \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix} = \begin{bmatrix} a_1b_1+0 & a_1b_2+0 \\ 0+a_4b_3 & 0+a_4b_4 \end{bmatrix}$$

4 Multiplications, 0 Additions

NNs with sparse matrices are (often) not that efficient.

Memory. To load sparse matrices, we need to load both "weights" and "locations" and locate non-zeros according to the locations.

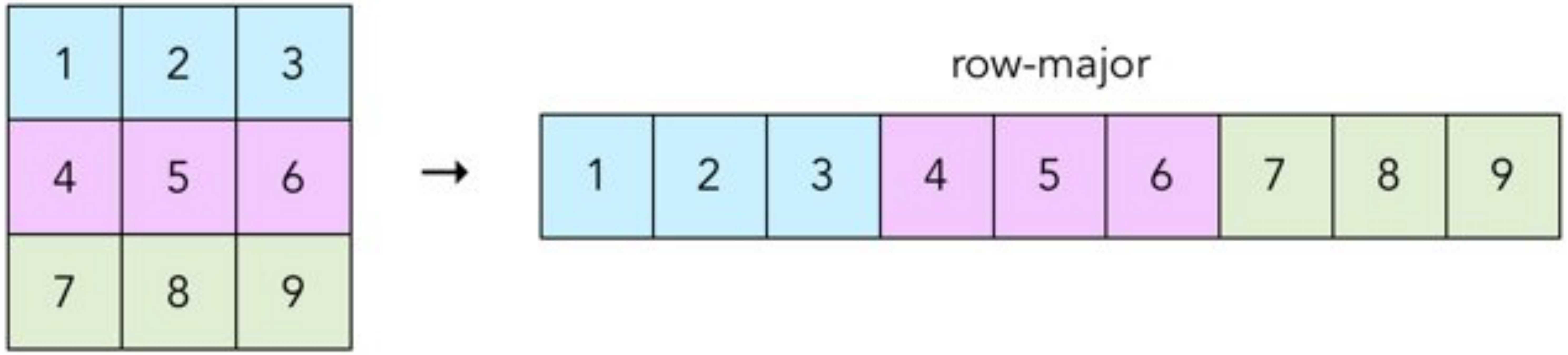
Compute. Need to know every location where we can skip the computation—not really fast without a specialized kernel (designed for each sparsity pattern?)

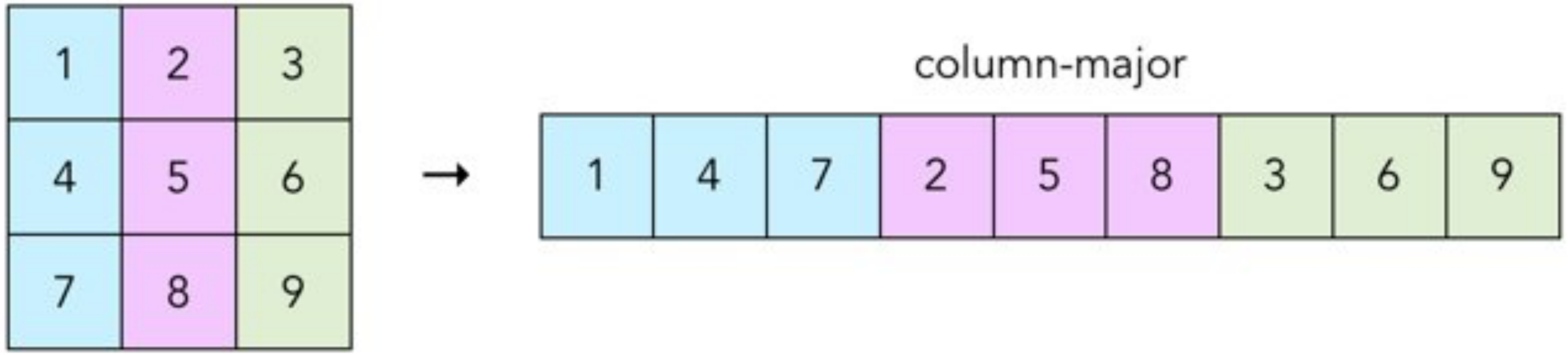
Worse, for GPUs, dot products are usually done as a group!

Matrix formats

First, let's take a look at how sparse matrices are stored.

If you ignore the sparsity, you may store the matrix in either row-major order or column-major order.
(Row-major: C, NumPy, PyTorch, …   Column-major: MATLAB, Julia, Fortran, …)
This choice makes the difference between the computing speeds of row-first loops and column-first loops.

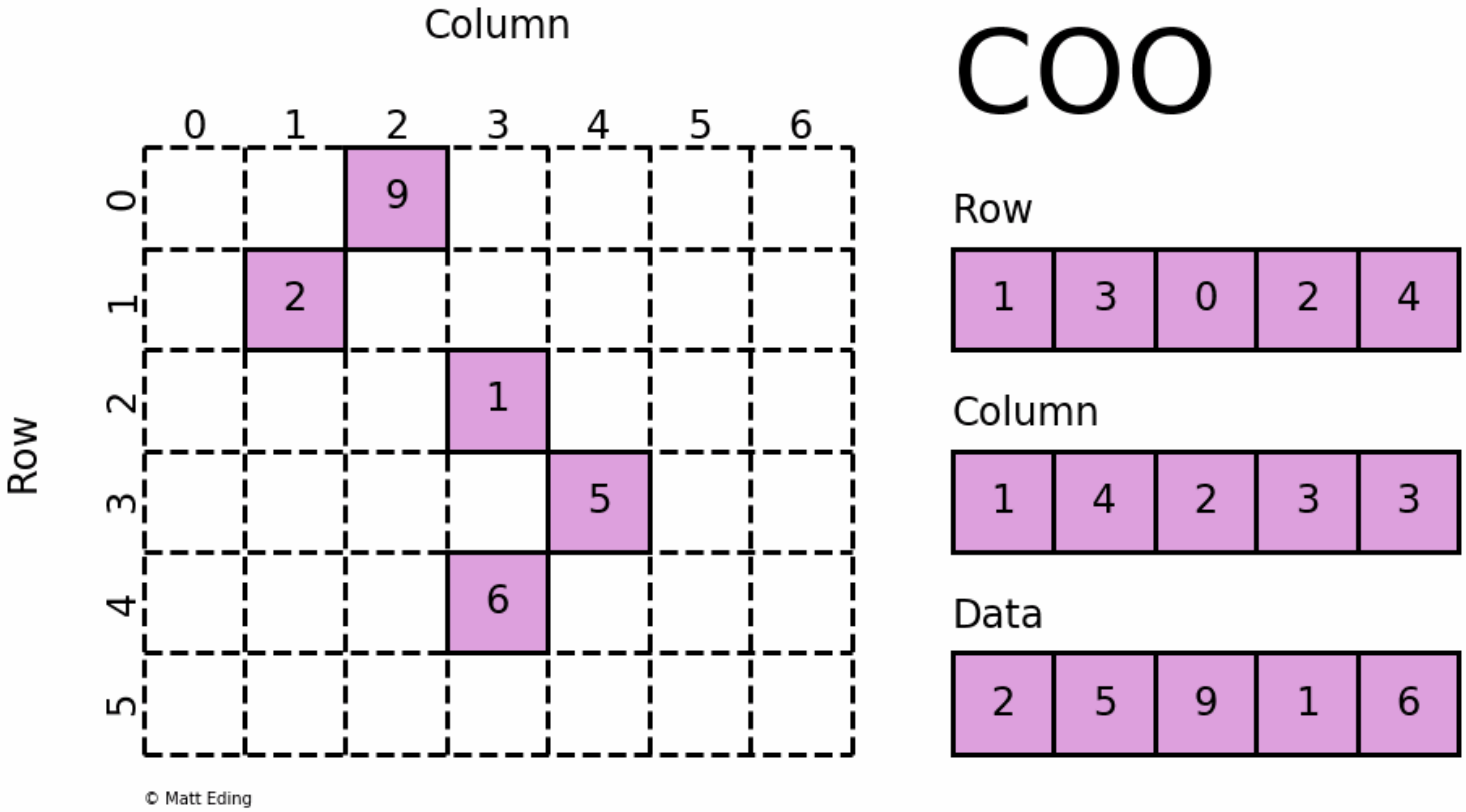




Note. There are formats for high-dimensional tensors (e.g., NCHW for activations in ConvNets), and this choice affects the wall-clock speed!

For sparse matrices, there are two popular formats: COO, and CSR (used in PyTorch / cuSPARSE)

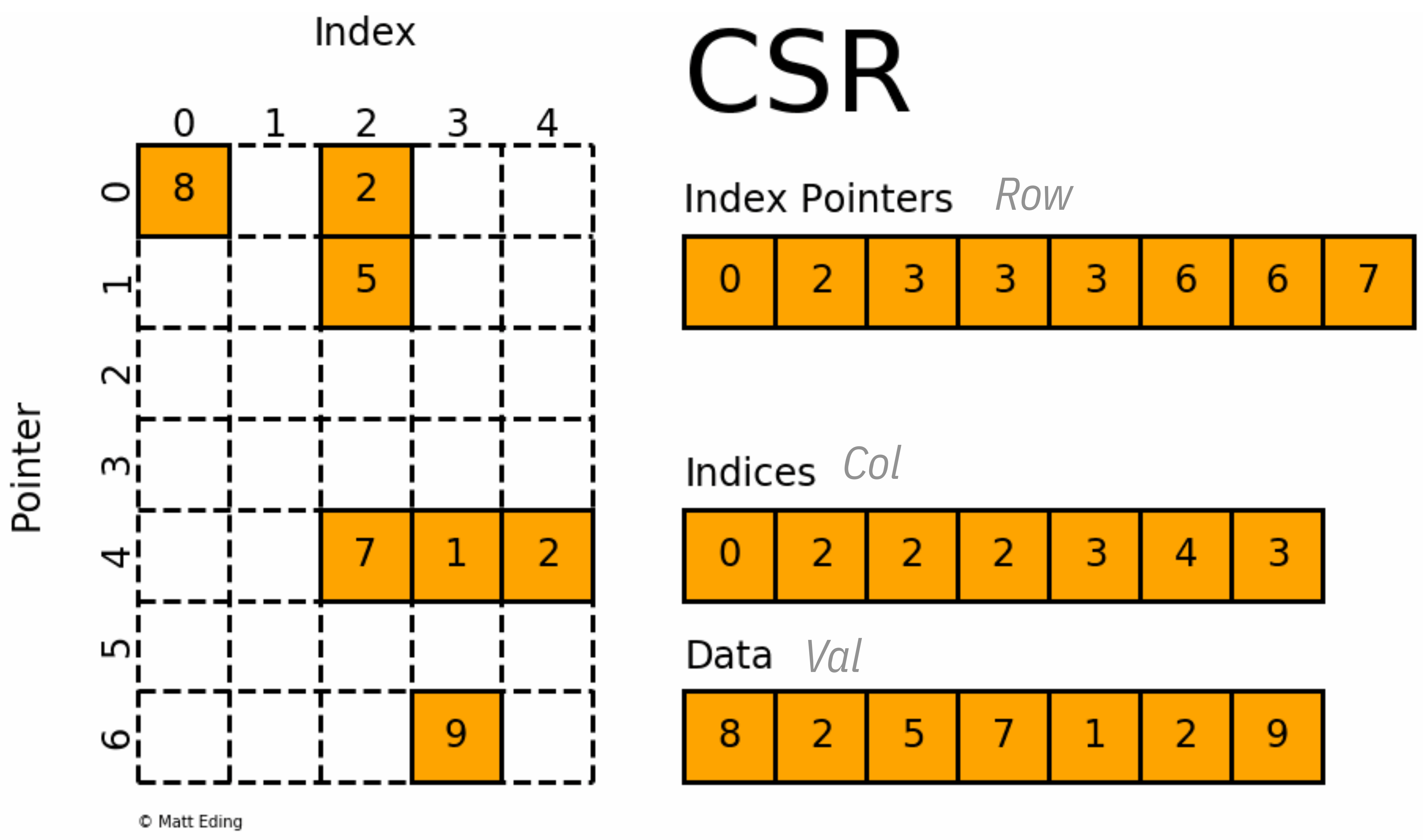
COOordinate**.** For each nonzero, store its weight, row, and column (small size, easy editing).



https://matteding.github.io/2019/04/25/sparse-matrices/

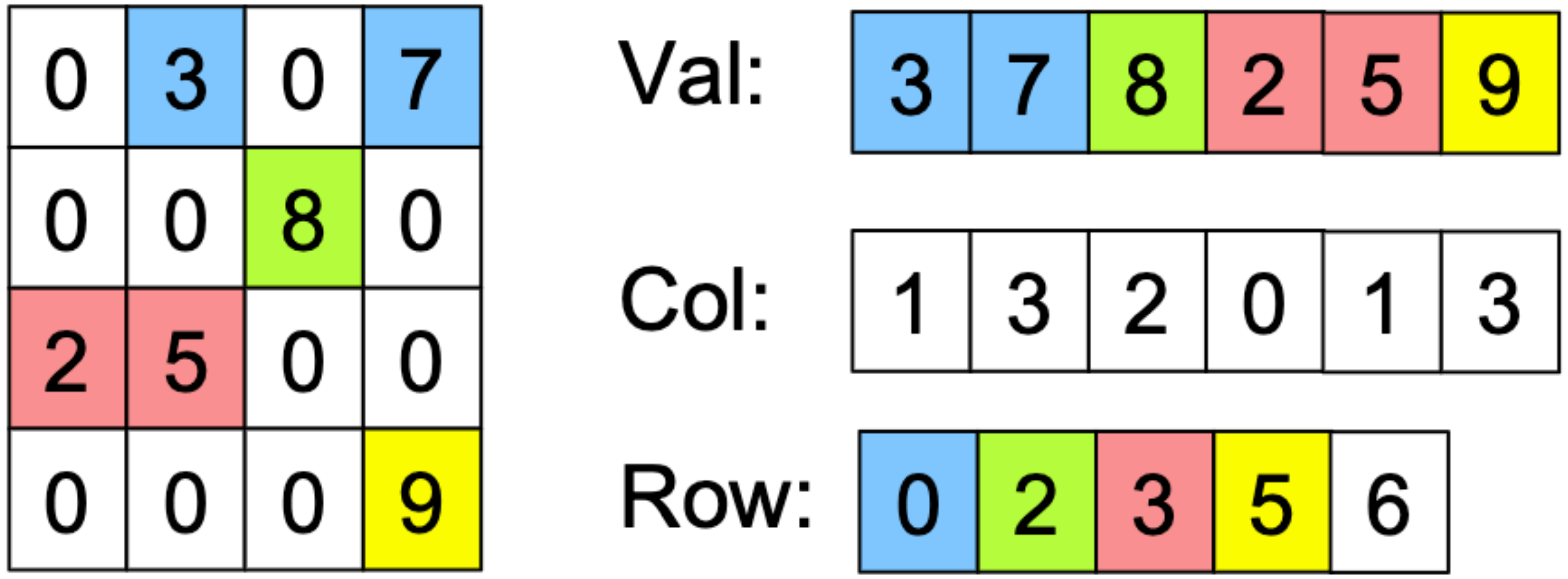
For sparse matrices, there are two popular formats: COO, and CSR (used in PyTorch / cuSPARSE)

Compressed **S**parse **R**ows. Instead of rows/columns, store the columns of each non-zero element, and which column indices belong to each row (better for computing).



https://matteding.github.io/2019/04/25/sparse-matrices/

CSR Overhead



Let us have an $N \times N$ matrix with K nonzero elements.

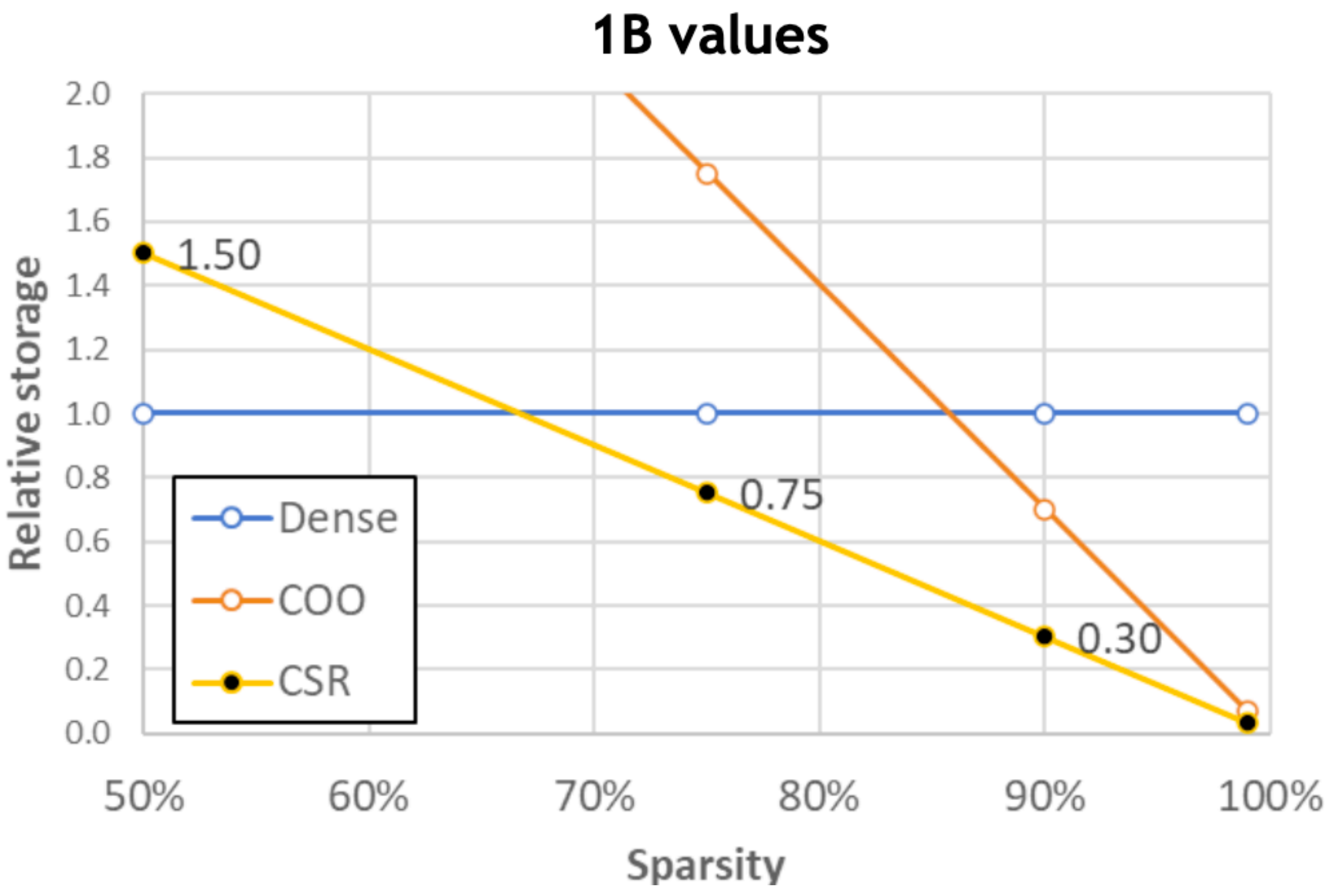
Val. K elements. Each are 1/2/4 bytes (int8, fp/bf16, fp32).

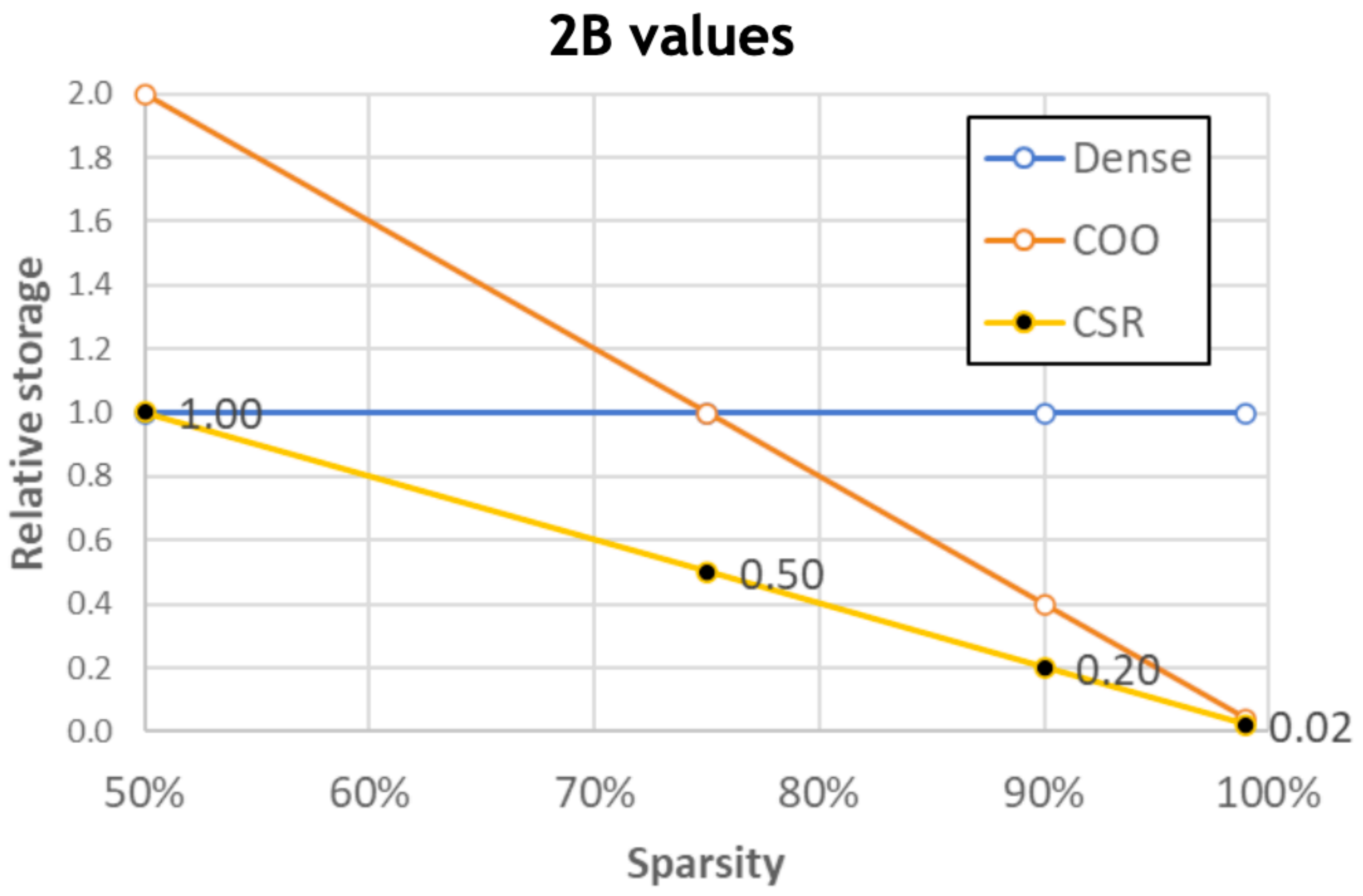
Col. K elements. Each are 1/2 bytes (1 if $N \leq 256$, 2 if $N \leq 65536$).

Row. $N + 1$ elements. Size dependent on K. (1 if $K \leq 256$, 2 if $K \leq 65536$, ...)

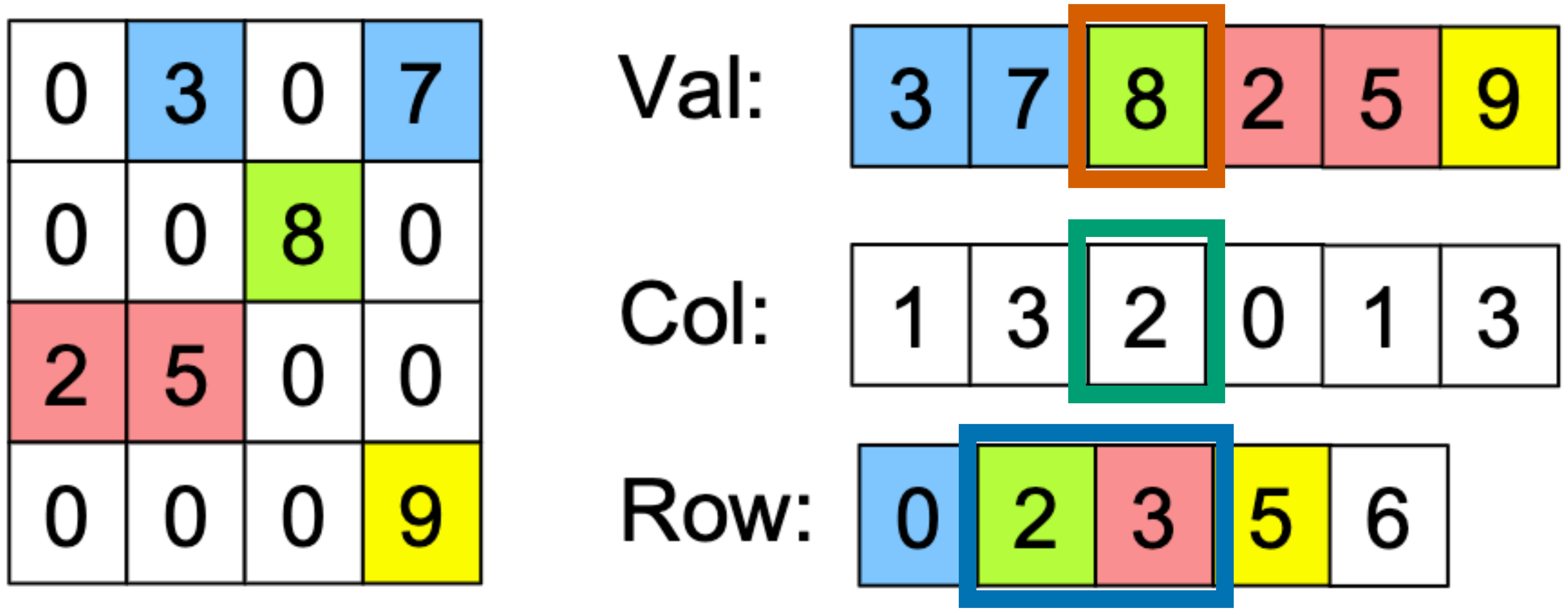
Overhead is quite big, especially if not that sparse!

Paulius Micikevicius, Invited Talk @ Sparse Neural Network Worshop 2021.





Paulius Micikevicius, Invited Talk @ Sparse Neural Network Worshop 2021.



There is an <u>extra latency</u> from the memory access.

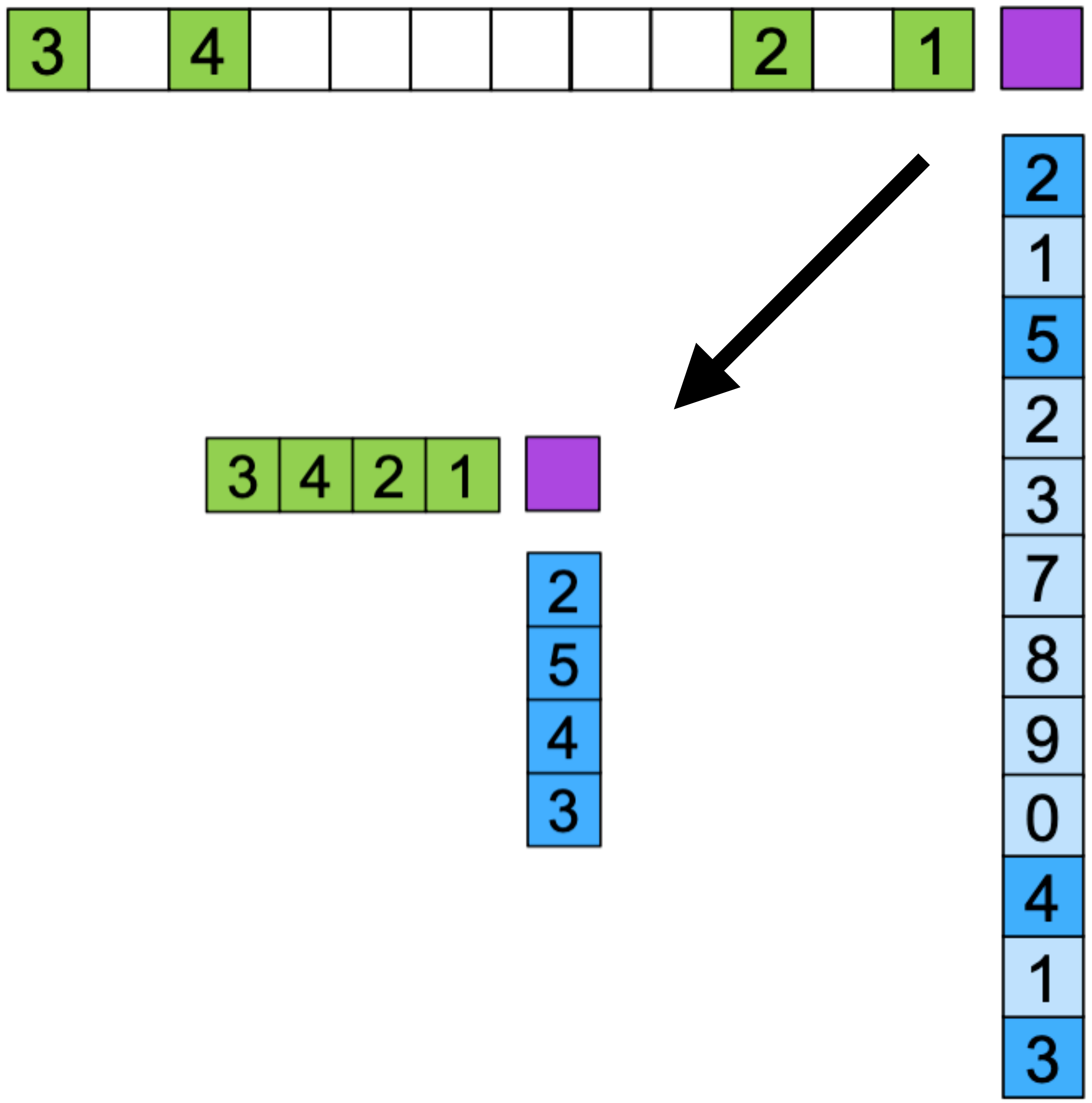
CSR introduces up to 2 dependent memory accesses: Suppose that we are trying to read row 2.

(1) Find out where **Row data** starts and ends.

(2) Find out which row to read from the **Col data**.

(3) Read the **Val data**.

Paulius Micikevicius, Invited Talk @ Sparse Neural Network Worshop 2021.



3*2 + 0*1 + 4*5 + 0* … + 2*4 + 0*1 + 1*3 = 37

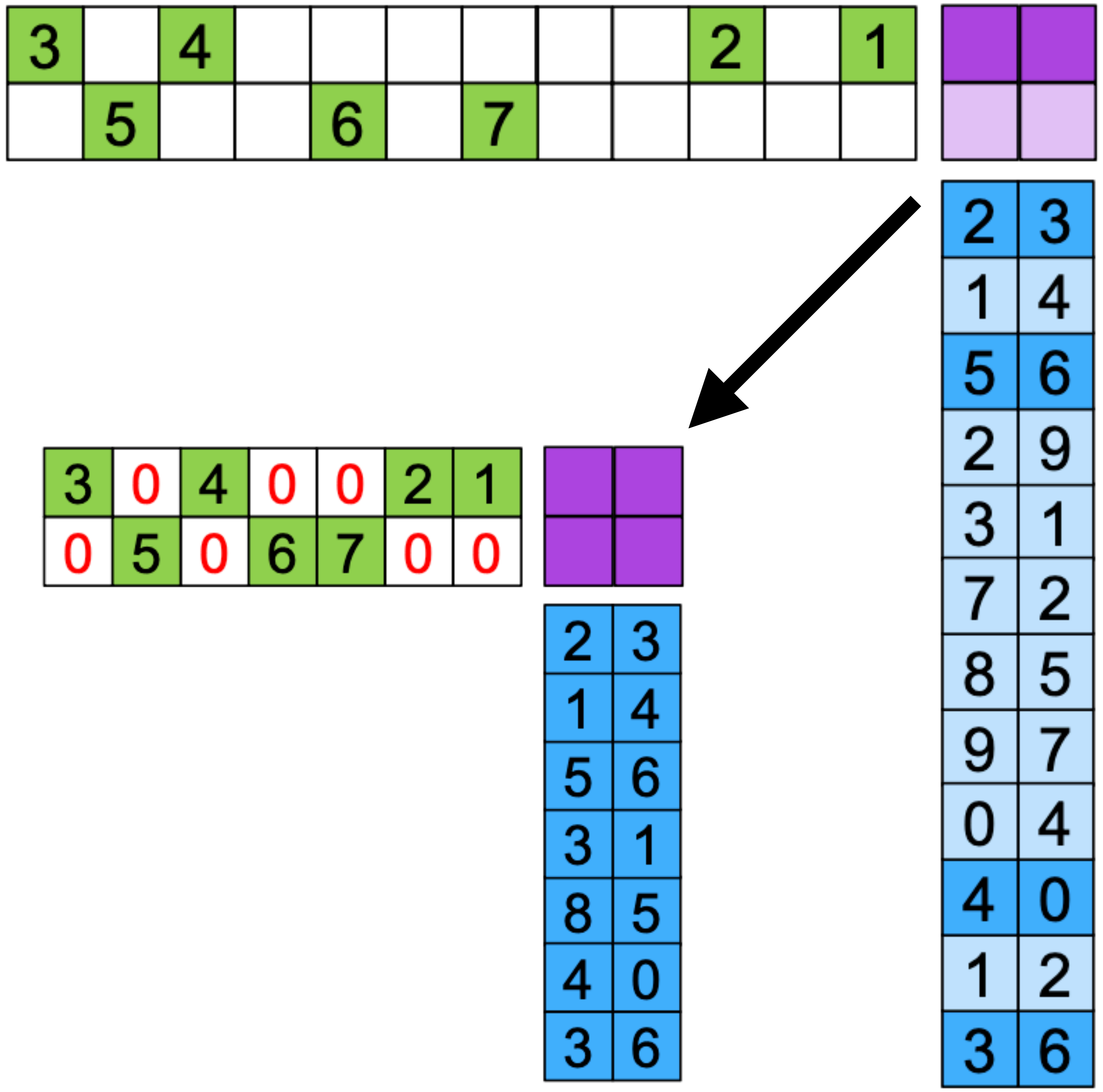
Typical routine for computing sparse-dense matmul:

(1) Fetch only nonzero weights from sparse matrix.

(2) Use location info of sparse to fetch subset of dense.

(3) Perform vector-wise operation (dot product)

Overhead: At (2)



Grouping multiple (unstructured) vectors lead to more number of wasted computation!

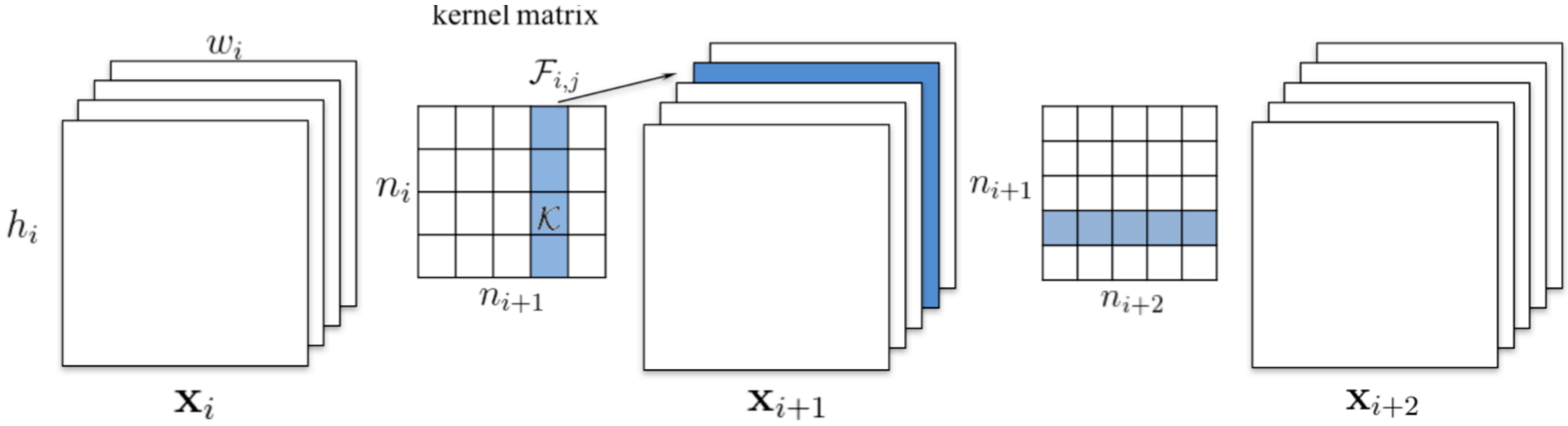
Here, total 14 multiplications are wasted.

Also, need to fetch more values from the dense matrix.

Structured Pruning

Idea. Prune the weight connections so that we can best exploit kernels/hardwares that are optimized for processing the dense networks.

Example. Filter/Channel Pruning (e.g., Li et al., 2017; Luo et al., 2017)
Remove convolutional filters with some saliency criterion;
a whole channel can be removed when all associated filters are gone.



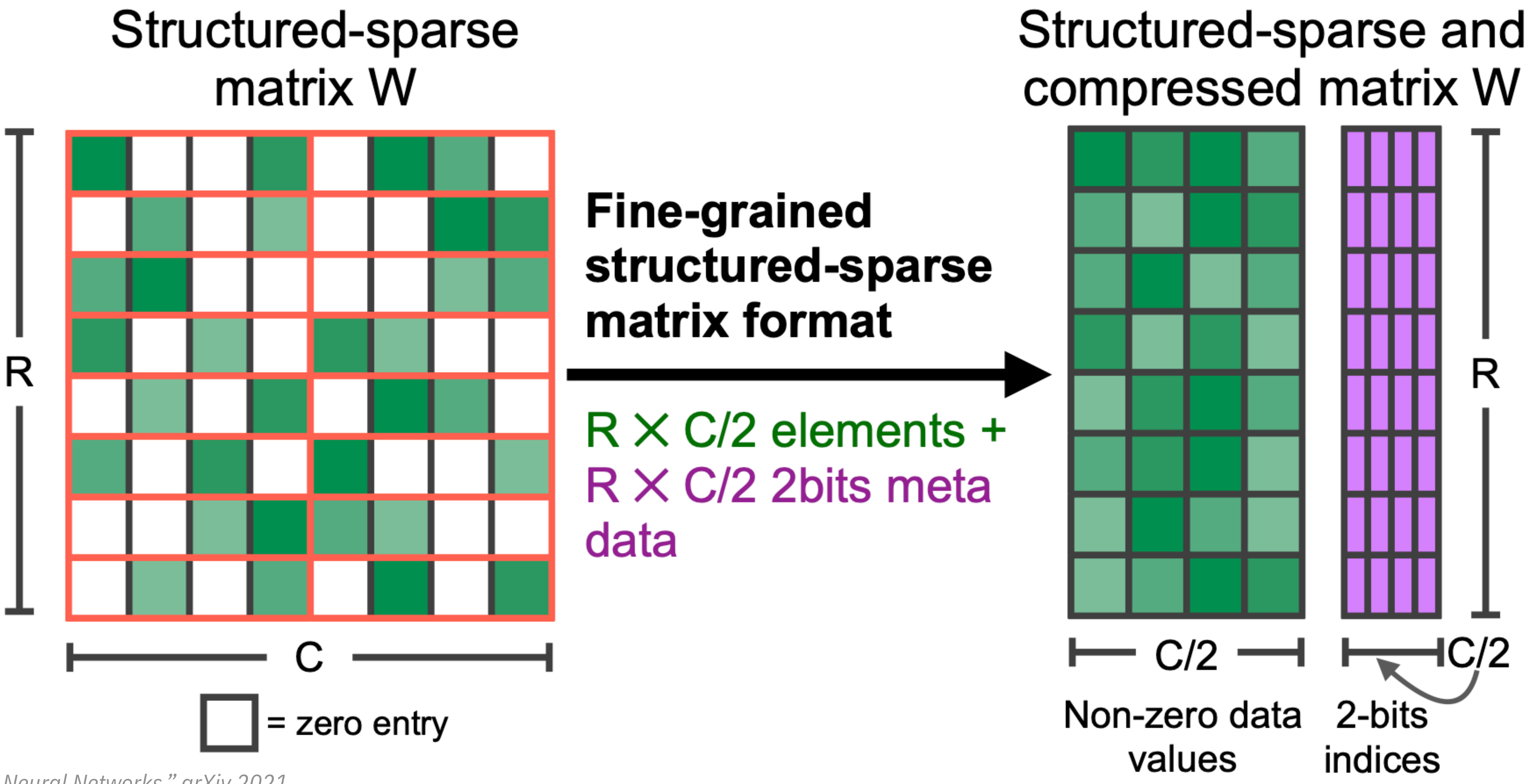
Li et al., "Pruning filters for efficient ConvNets," ICLR 2017
Luo et al., "ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression" ICCV 2017.

Example. 2:4 sparsity (NVIDIA A100 GPU)

Have at least 2 zeros in length-4 block

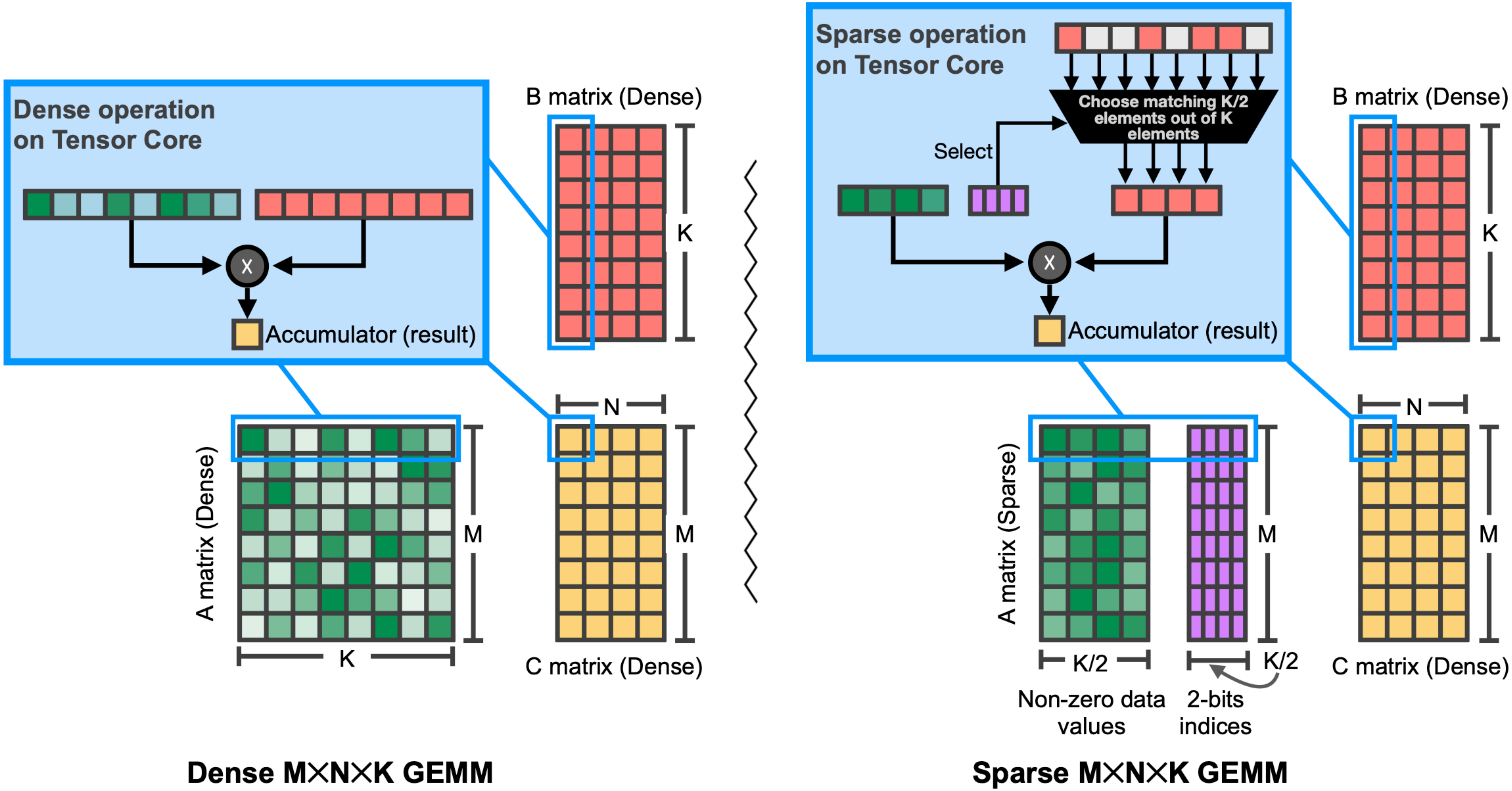
- 50% sparsity; no performance drop almost always.

- Meta-data can be very small; 2 bits per nonzero element.



Mishra et al., "Accelerating Sparse Deep Neural Networks," arXiv 2021

Provide specialized HW (A100 with Sparse Tensor Cores) and engine (TensorRT 8.0)

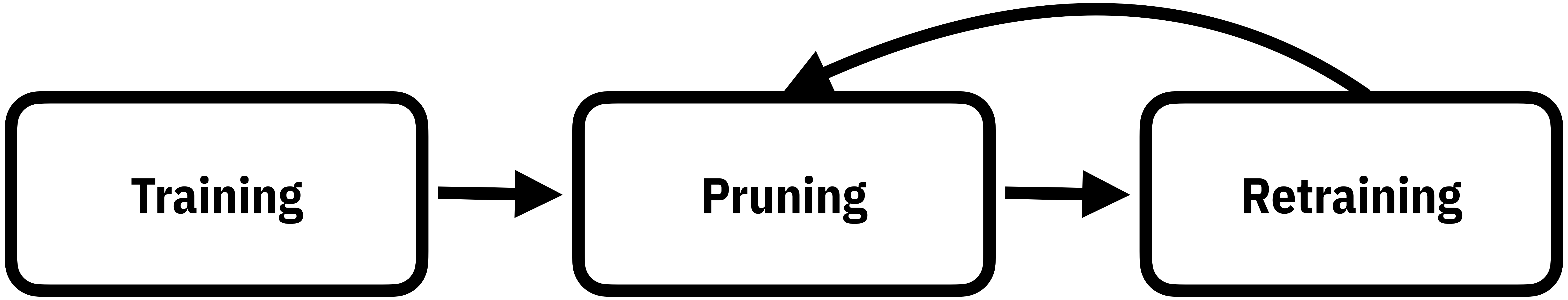


Mishra et al., "Accelerating Sparse Deep Neural Networks," arXiv 2021

Pruning at Initialization

Motivation. Typical network pruning pipeline require too many training FLOPs.

Training cost has not been a big issue before 2019, but now...?



Folk Knowledge. Without an initial training,

- The pruned model cannot be optimized well
- The final model does not generalize well.

ICLR'19. Three paper has been published:

Frankle et al., "The lottery ticket hypothesis: finding sparse, trainable NNs"

(proof of concept; not a real algorithm)

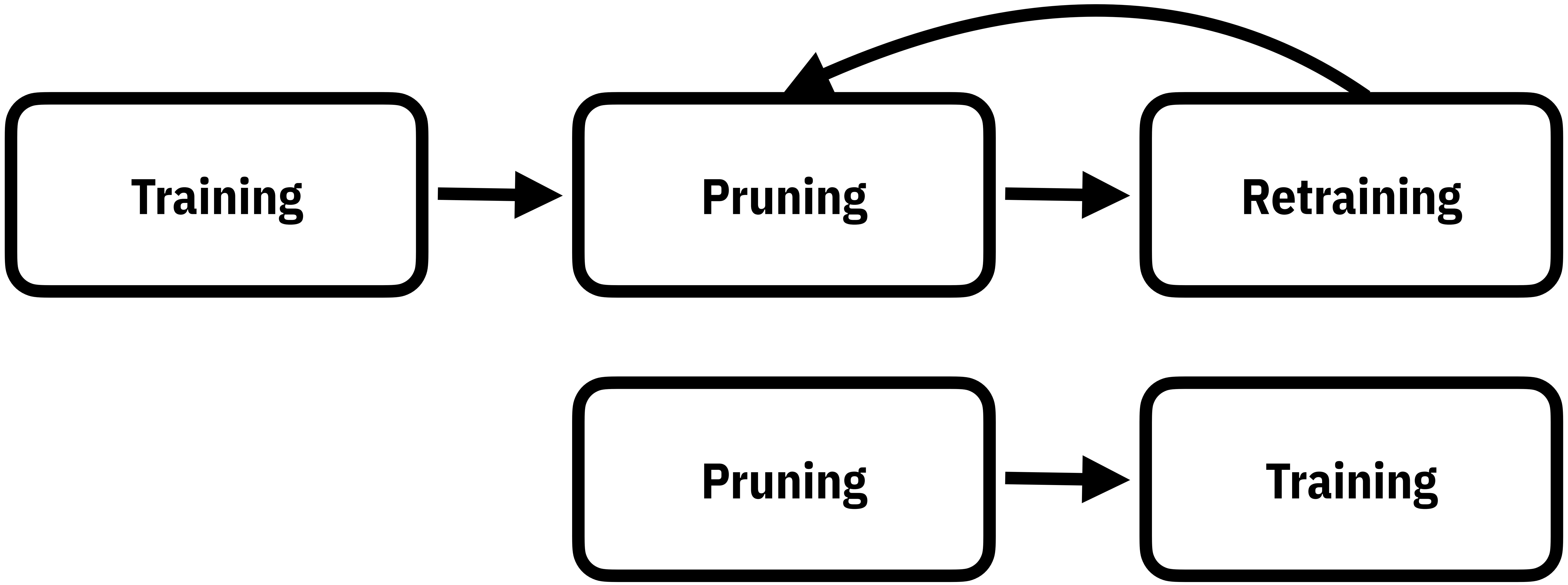
Liu et al., "Rethinking the value of network pruning"

(Structured-ly pruned networks can be optimized well with good hyperparams)

Lee et al., "SNIP: Single-shot network pruning based connection sensitivity"

(The first algorithm to prune at initialization)

Message: We can prune at the initial stage without performance drop (also note peak memory drop)



SNIP. We use the first-order derivative.

- Magnitude Pruning (0th order): $\text{score}_i(\theta) = |\theta_i|$

- SNIP (1st order): $\text{score}_i(\theta) = \left|\frac{\partial L}{\partial \theta_i}\right| \cdot |\theta_i|$

- Optimal Brain Damage (2nd order): $\text{score}_i(\mathbf{w}) = \left|\frac{\partial^2 L}{\partial \theta_i^2}\right| \cdot |\theta_i|^2$

Intuition. Recall the Taylor's approximation.

At the initial point, we can no longer ignore the second term!

$$L(\tilde{\theta}) \approx L(\theta) + (\tilde{\theta} - \theta)^\top G_\theta + \frac{1}{2}(\tilde{\theta} - \theta)^\top H_\theta(\tilde{\theta} - \theta)$$

Limitation. Typically can only achieve low sparsity; training usually takes longer than dense.

Note. There are other methods called GraSP / SynFlow, but SNIP performs best (in my personal experience)

Note. Recent experiments show that these methods are simply finding the right layerwise sparsity

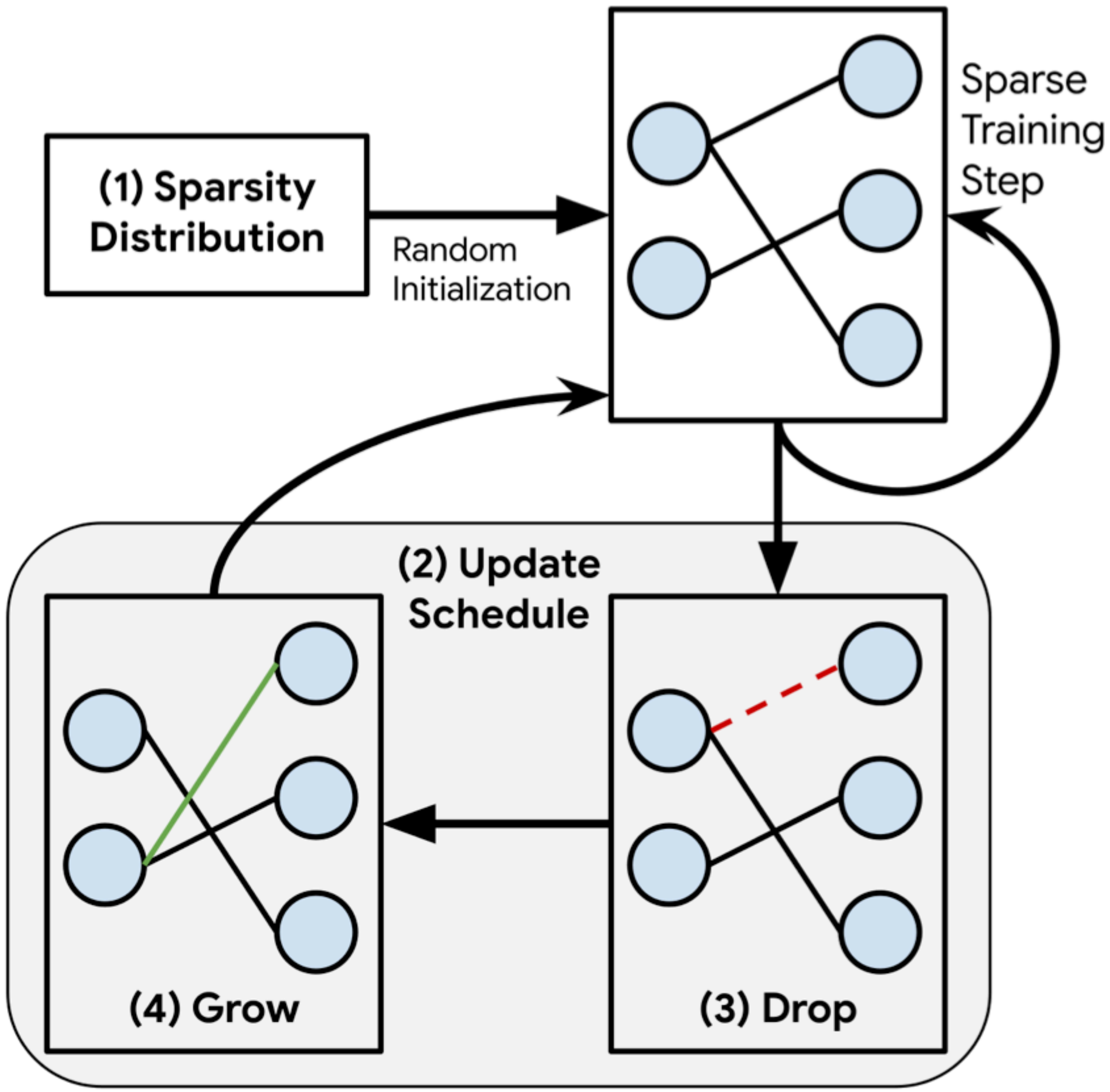
Sparse Training

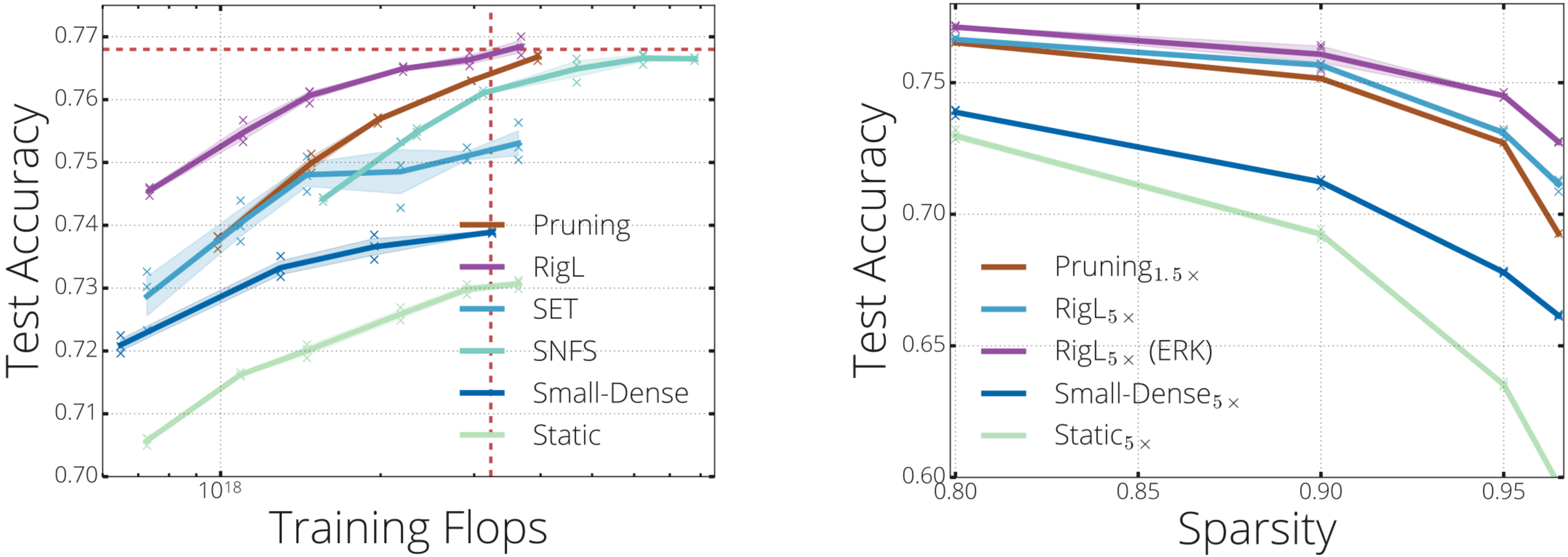
Motivation. Pruning at initialization methods can only have low sparsity.
But what if we allow for changing sparsity pattern during the training?

Method. (1) Initialize with random sparsity pattern.
(2) Train the sparse model
(3) Remove some small-magnitude connections.
(4) Re-grow some high-gradient connections.
(5) Go to step 2.
RigL (Evci et al., ICML'20)

Result. Almost as good as dense-to-sparse training.
(Note: using a handcrafted layerwise sparsity is critical)

Limitation. Peak memory is still the same as dense.
(Partially resolved by Top-KAST)
Training usually takes longer.





Test Accuracy
0.77
0.76
0.75
0.74
0.73
0.72
0.71
0.70
10^{18}
Training Flops
Pruning
RigL
SET
SNFS
Small-Dense
Static
Test Accuracy
0.75
0.70
0.65
0.60
0.80
0.85
0.90
0.95
Sparsity
Pruning$_{1.5\times}$
RigL$_{5\times}$
RigL$_{5\times}$ (ERK)
Small-Dense$_{5\times}$
Static$_{5\times}$

Personal Remarks.

Pruning is about both (1) Minimizing the loss after pruning <- good saliency score
(2) Maximizing the re-trainability <- good layerwise sparsity & schedule

In the past, people believed that (1) is much more important.
Nowadays, (2) is viewed as the most important decision criterion.

Optimizing (2) is very difficult, compute-heavy procedure...
how can we reduce the hyperparameter search cost? # my own papers are about these ;)

Next up. Conditional computation.
Activation sparsity.