ECE408/CS483/CSE408 Spring 2025

Applied Parallel Programming

Lecture 17
Parallel Sparse Methods

Course Reminders

- Lab 6 is due this Friday
- Project Milestone 2 is due 4/11/25
- Take a note of the day/time of midterm 2

Objective

- To learn the key techniques for compacting input data in parallel sparse methods for reduced consumption of memory bandwidth
  - Better utilization of on-chip memory
  - Fewer bytes transferred to on-chip memory
  - Better utilization of global memory
  - Challenge: retaining regularity

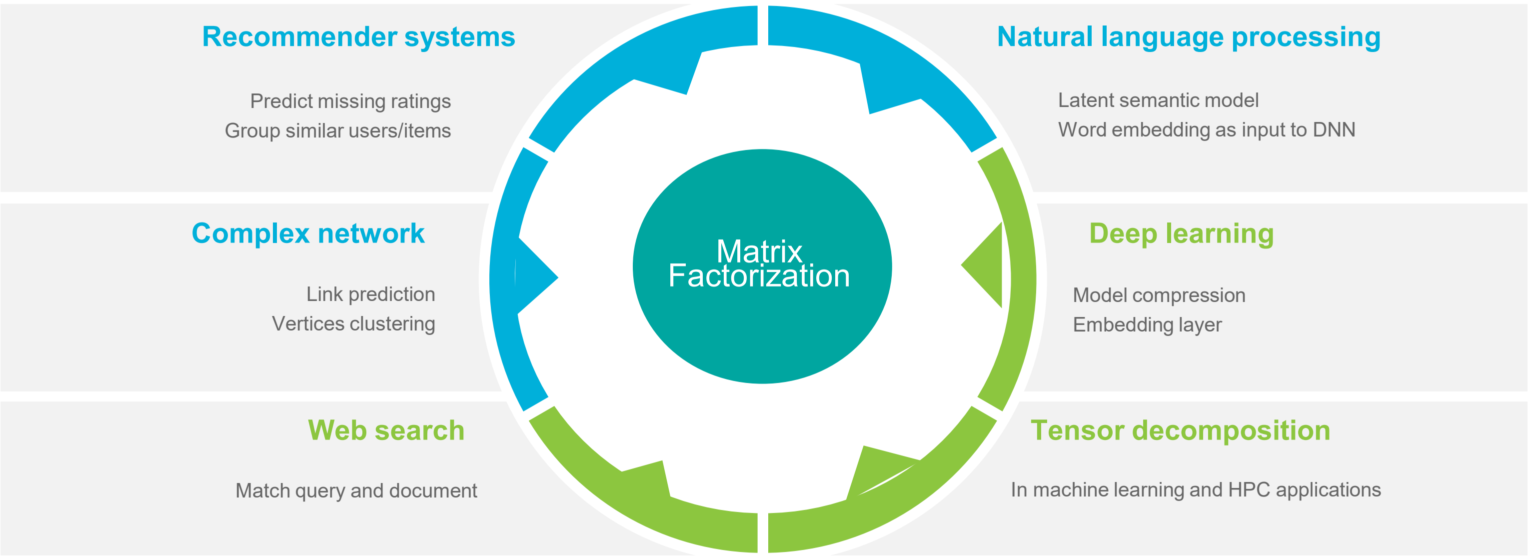
Sparse Matrix

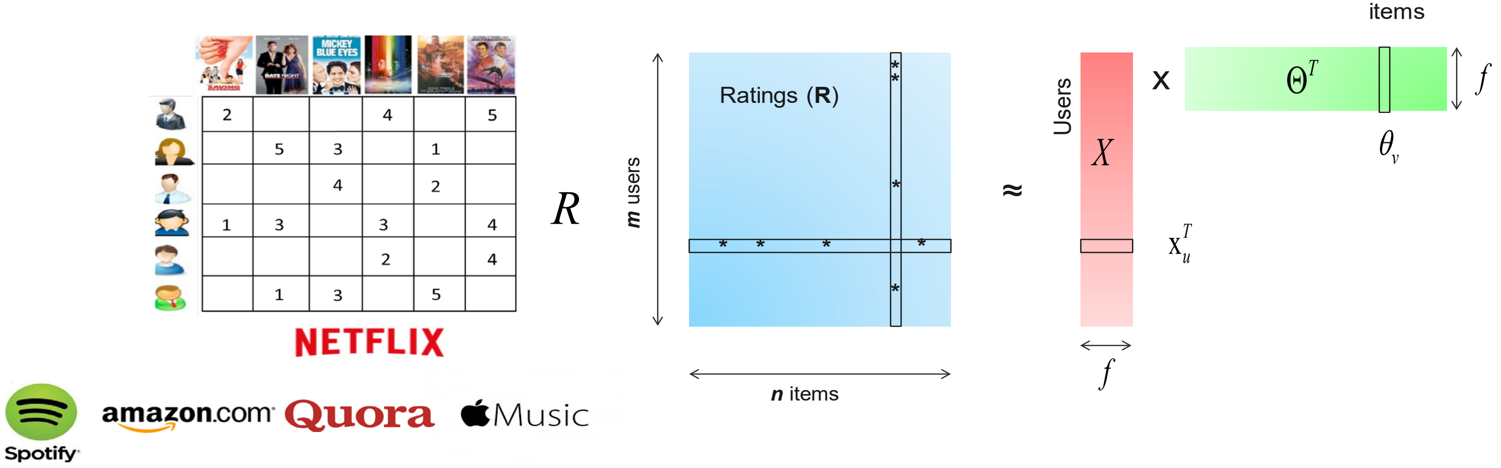
- Many real-world systems are sparse in nature
  - Linear systems described as sparse matrices
- Solving sparse linear systems
  - Iterative Conjugate Gradient solvers based on sparse matrix-vector multiplication is a common method
- Solution of PDE systems can be formulated into linear operations expressed as sparse matrix-vector multiplication

Sparse Matrix in Scientific Computing

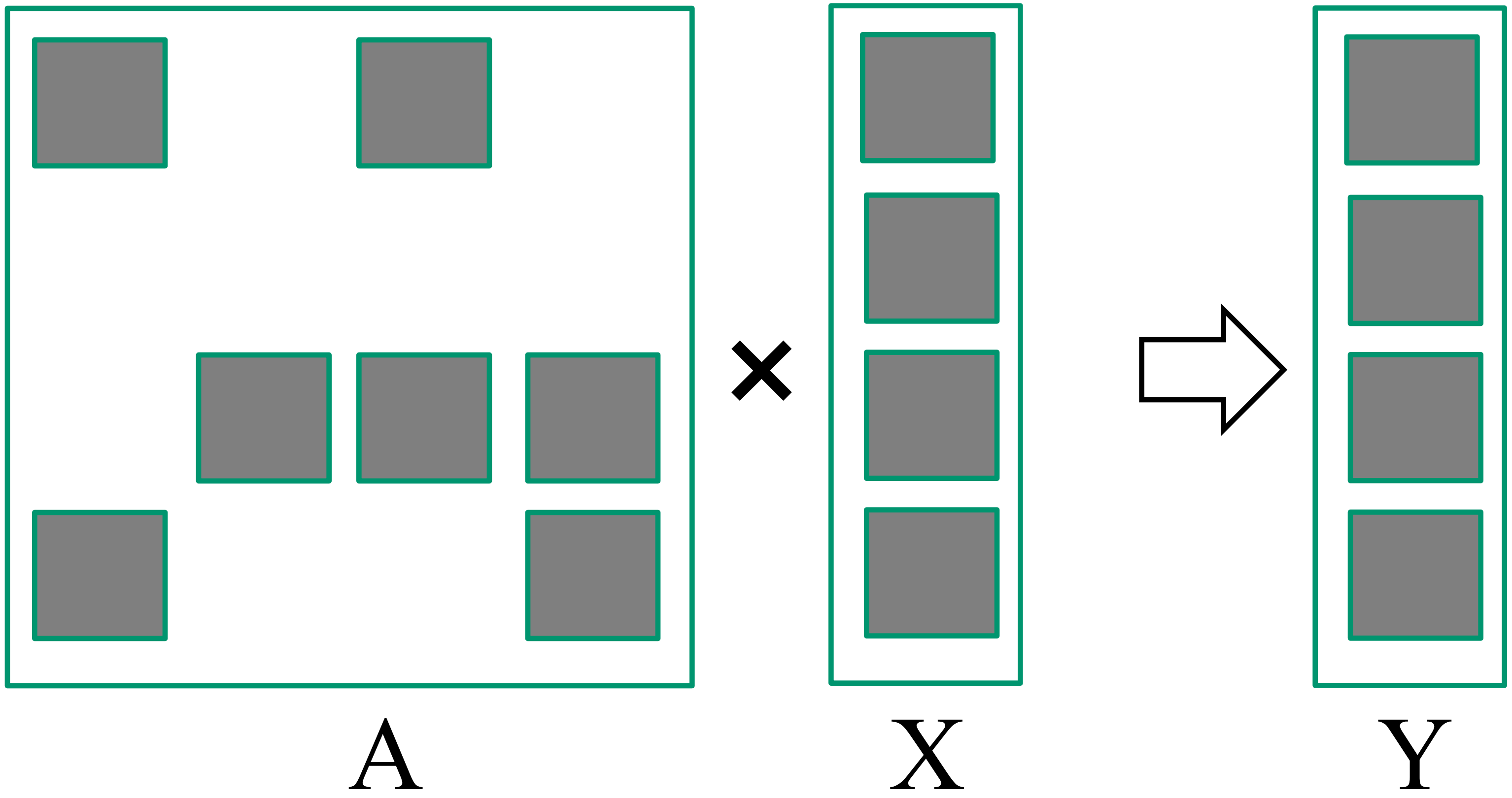
Science Area	Number of Teams	Codes	Struct Grids	Unstruct Grids	Dense Matrix	Sparse Matrix	N-Body	Monte Carlo	FFT	PIC	Sig I/O
Climate and Weather	3	CESM, GCRM, CM1/WRF, HOMME	X	X		X		X			X
Plasmas/Magnetosphere	2	H3D(M),VPIC, OSIRIS, Magtail/UPIC	X				X		X		X
Stellar Atmospheres and Supernovae	5	PPM, MAESTRO, CASTRO, SEDONA, ChaNGa, MS-FLUKSS	X			X	X	X		X	X
Cosmology	2	Enzo, pGADGET	X			X	X				
Combustion/Turbulence	2	PSDNS, DISTUF	X						X		
General Relativity	2	Cactus, Harm3D, LazEV	X			X					
Molecular Dynamics	4	AMBER, Gromacs, NAMD, LAMMPS				X	X		X		
Quantum Chemistry	2	SIAL, GAMESS, NWChem			X	X	X	X			X
Material Science	3	NEMOS, OMEN, GW, QMCPACK			X	X	X	X			
Earthquakes/Seismology	2	AWP-ODC, HERCULES, PLSQR, SPECFEM3D	X	X			X				X
Quantum Chromo Dynamics	1	Chroma, MILC, USQCD	X		X	X					
Social Networks	1	EPISIMDEMICS									
Evolution	1	Eve									
Engineering/System of Systems	1	GRIPS,Revisit						X			
Computer Science	1			X	X	X			X		X

Sparse Matrix in Analytics and AI





Sparse Matrix-Vector Multiplication (SpMV)



Challenges

- Compared to dense matrix multiplication, SpMV
  – Is irregular/unstructured
  – Has little input data reuse
  – Benefits little from compiler transformation tools

- Key to maximal performance
  – Maximize regularity (by reducing divergence and load imbalance)
  – Maximize DRAM burst utilization (layout arrangement)

A Simple Parallel SpMV

Row 0	3	0	1	0	Thread 0
Row 1	0	0	0	0	Thread 1
Row 2	0	2	4	1	Thread 2
Row 3	1	0	0	1	Thread 3

- Each thread processes one row

Compressed Sparse Row (CSR) Format

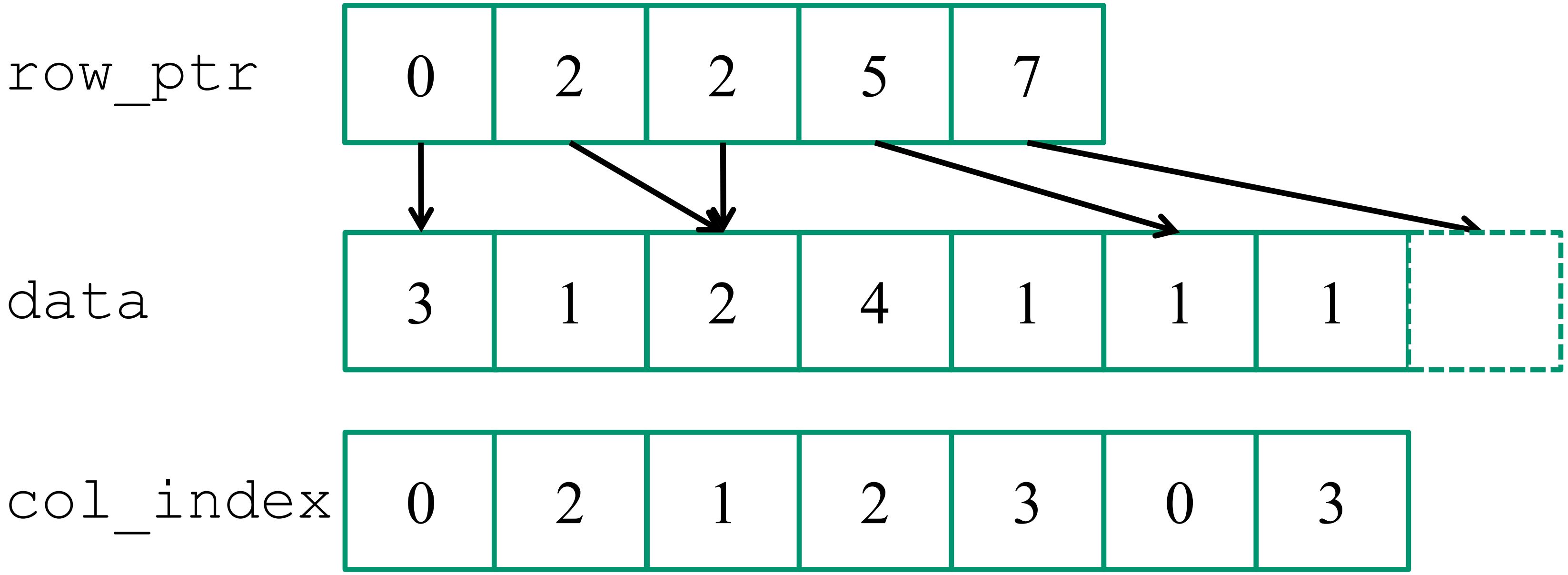
CSR Representation

		Row 0		Row 2			Row 3	
Nonzero values	`data[7]`	{ 3,	1,	2,	4,	1,	1,	1 }
Column indices	`col_index[7]`	{ 0,	2,	1,	2,	3,	0,	3 }
Row Pointers	`row_ptr[5]`	{ 0,	2,	2,	5,	7 }		

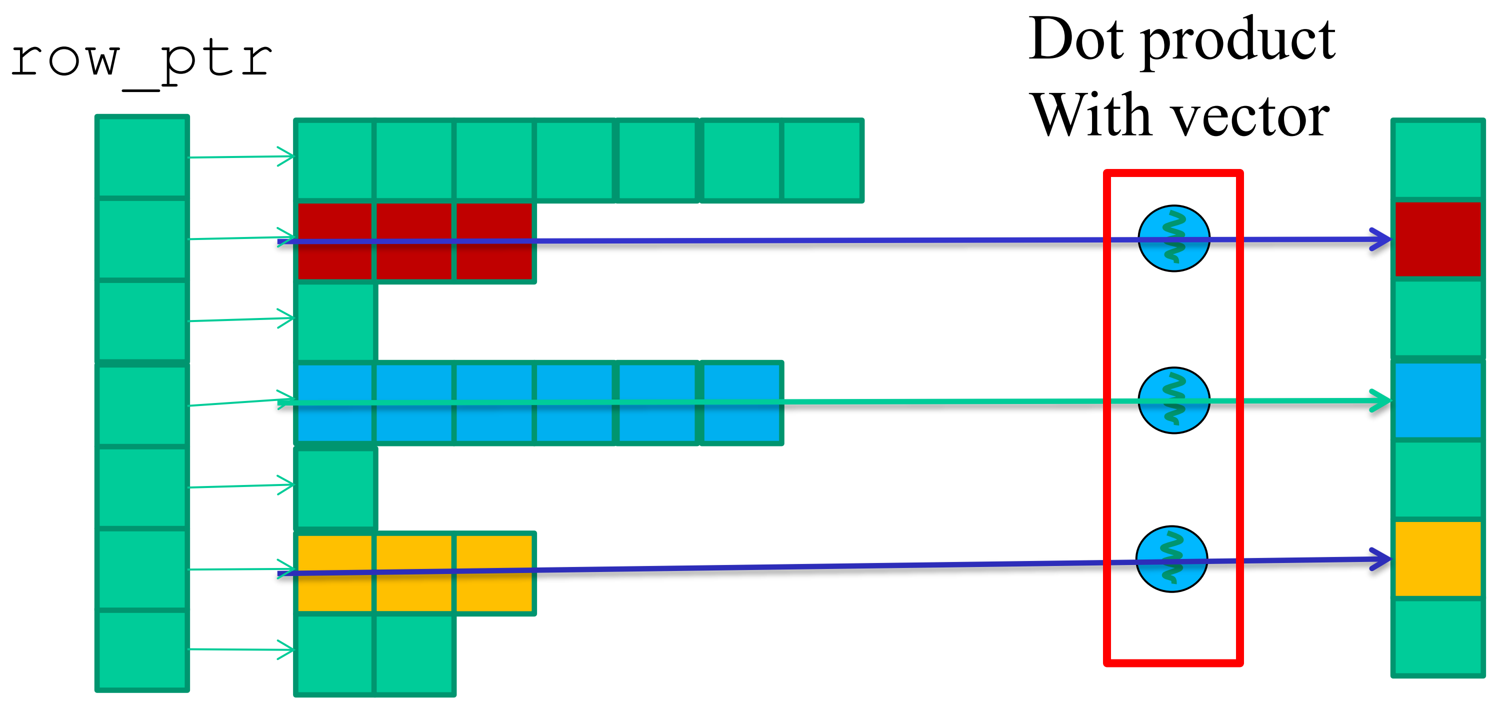
Dense representation

Row 0	3	0	1	0
Row 1	0	0	0	0
Row 2	0	2	4	1
Row 3	1	0	0	1

CSR Data Layout



CSR Kernel Design



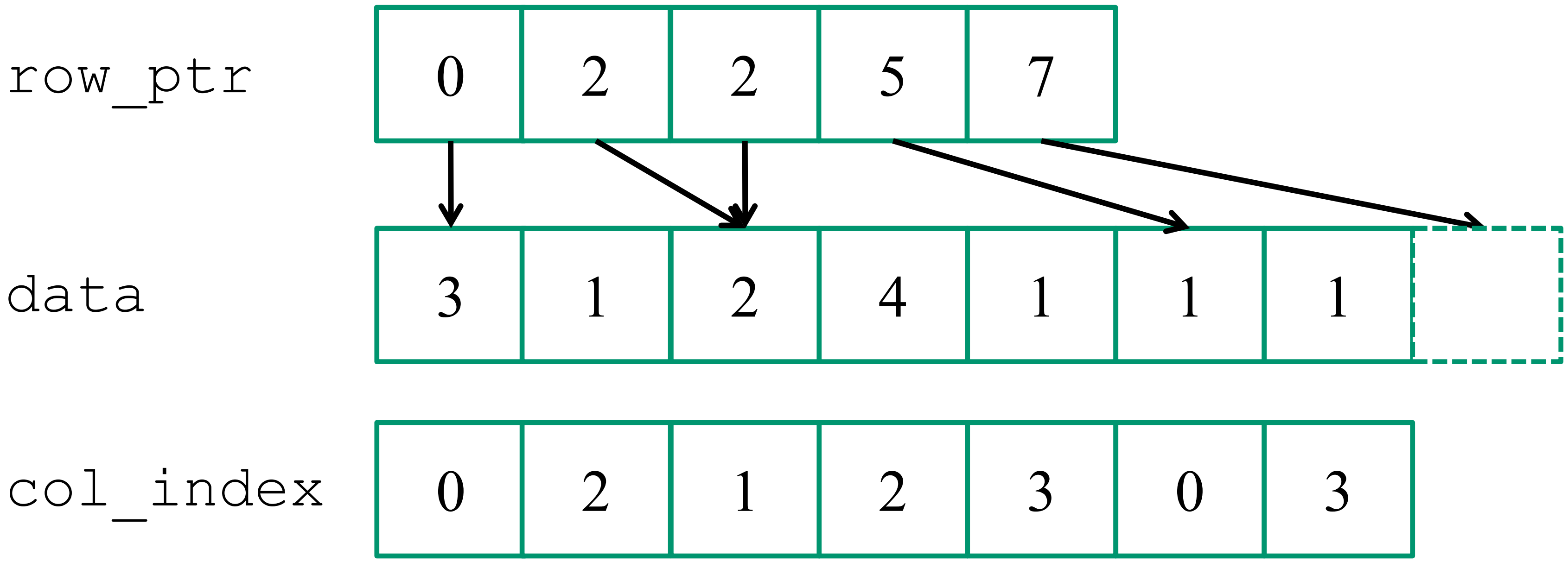
A Parallel SpMV/CSR Kernel (CUDA)

```
1. __global__ void SpMV_CSR(int num_rows, float *data, int
   *col_index, int *row_ptr, float *x, float *y)
   {
2.     int row = blockIdx.x * blockDim.x + threadIdx.x;
3.     if (row < num_rows) {
4.       float dot = 0;
5.       int row_start = row_ptr[row];
6.       int row_end =   row_ptr[row+1];
7.       for (int elem = row_start; elem < row_end; elem++)
8.           dot += data[elem] * x[col_index[elem]];
9.       y[row] = dot;
       }
   }
```

		Row 0	Row 2	Row 3
Nonzero values	data[7]	{ 3, 1,	2, 4, 1,	1, 1 }
Column indices	col_index[7]	{ 0, 2,	1, 2, 3,	0, 3 }
Row Pointers	row_ptr[5]	{ 0, 2,	2, 5, 7 }	

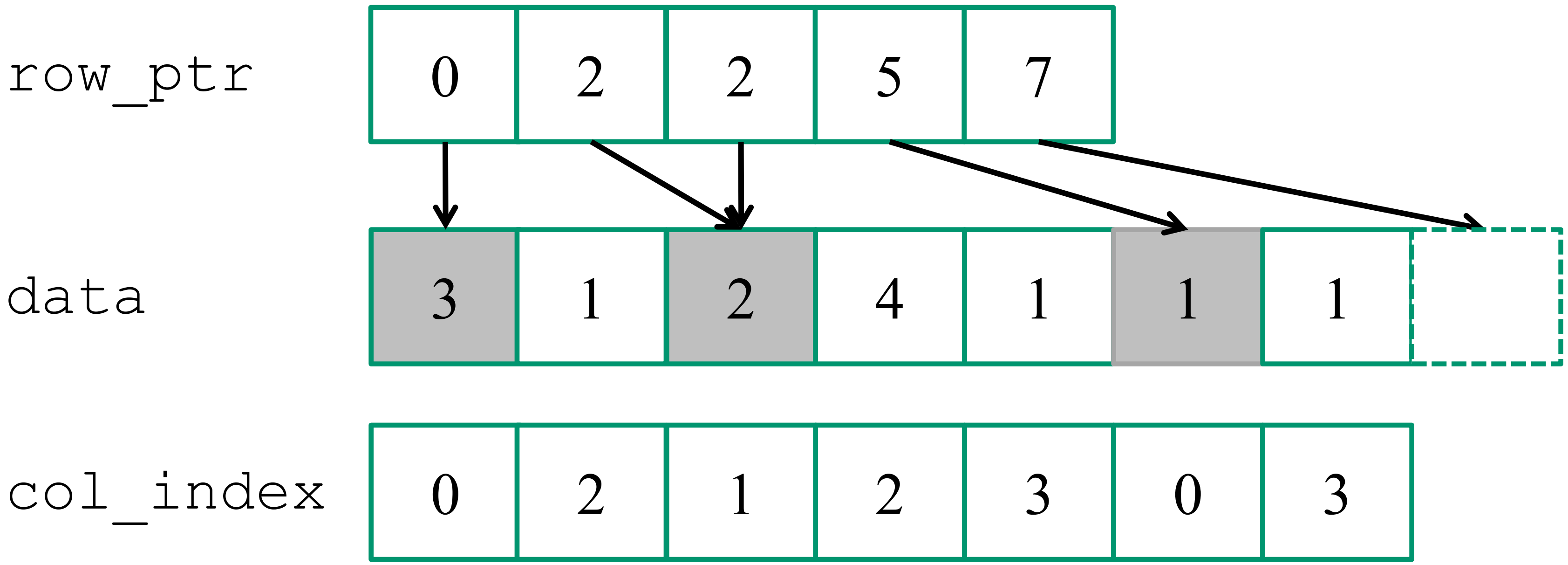
CSR Kernel Control Divergence

- Threads execute different number of iterations

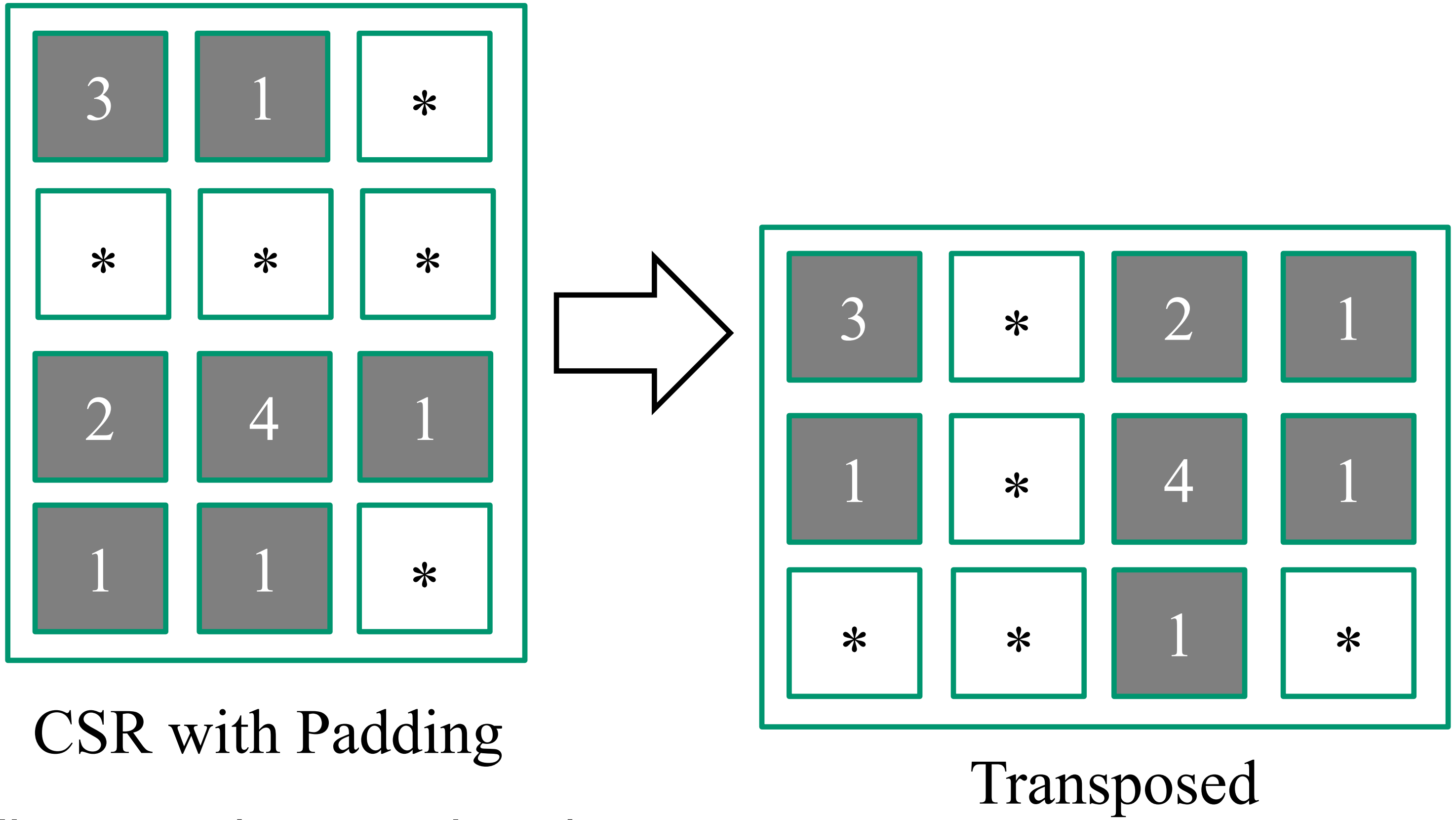


CSR Kernel Memory Divergence (Uncoalesced Accesses)

- Adjacent threads access non-adjacent memory locations
  - Grey elements are accessed by all threads in iteration 0

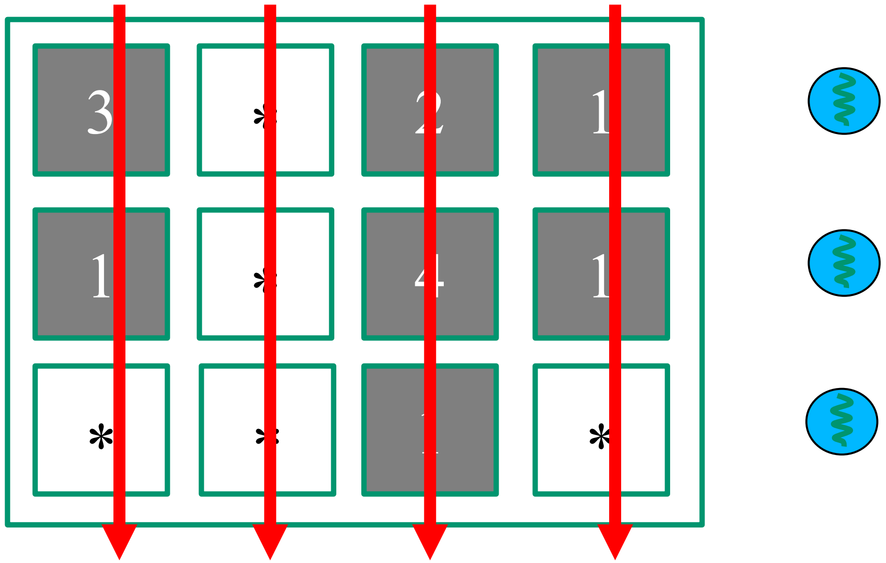


Regularizing SpMV with ELL(PACK) Format



- Pad all rows to the same length
  - Inefficient if a few rows are much longer than others
- Transpose (Column Major) for DRAM efficiency
- Both data and col_index padded/transposed

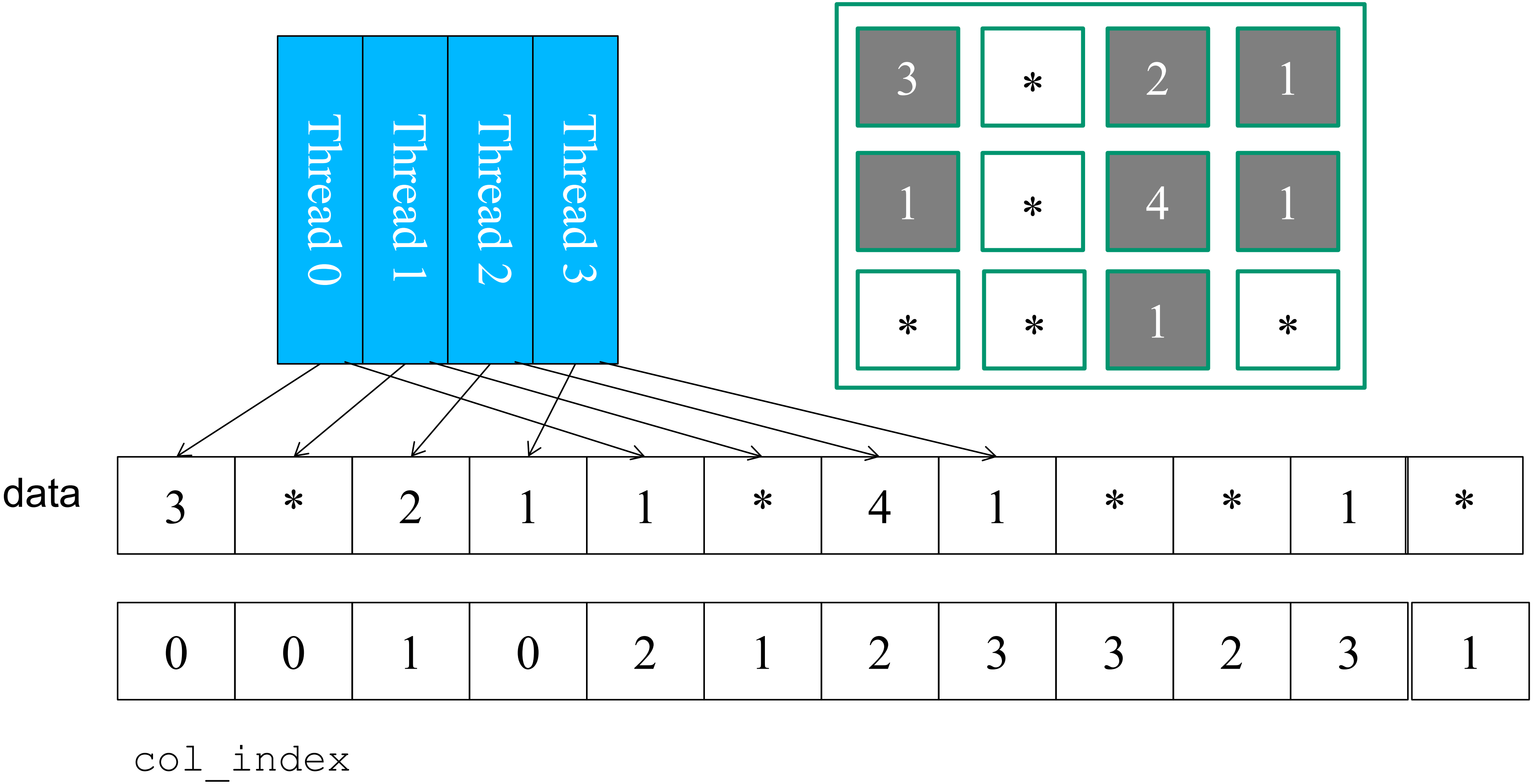
ELL Kernel Design



A parallel SpMV/ELL kernel

```
__global__ void SpMV_ELL(int num_rows, float *data,
      int *col_index, int num_elem, float *x, float *y)
 {
   int row = blockIdx.x * blockDim.x + threadIdx.x;
   if (row < num_rows) {
     float dot = 0;
     for (int i = 0; i < num_elem; i++)
        dot += data[row+i*num_rows] * x[col_index[row+i*num_rows]];
     y[row] = dot;
   }
 }
```

Memory Coalescing with ELL



Coordinate (COO) format

- Explicitly list the column & row indices for every non-zero element

		Row 0		Row 2			Row 3	
Nonzero values	`data[7]`	{ 3,	1,	2,	4,	1,	1,	1 }
Column indices	`col_index[7]`	{ 0,	2,	1,	2,	3,	0,	3 }
Row indices	`row_index[7]`	{ 0,	0,	2,	2,	2,	3,	3 }

COO Allows Reordering of Elements

			Row 0		Row 2			Row 3		
Nonzero values	data[7]	{	3,	1,	2,	4,	1,	1,	1	}
Column indices	col_index[7]	{	0,	2,	1,	2,	3,	0,	3	}
Row indices	row_index[7]	{	0,	0,	2,	2,	2,	3,	3	}

Nonzero values	data[7]	{ 1	1,	2,	4,	3,	1	1	}
Column indices	col_index[7]	{ 0	2,	1,	2,	0,	3,	3	}
Row indices	row_index[7]	{ 3	0,	2,	2,	0,	2,	3	}

COO Kernel

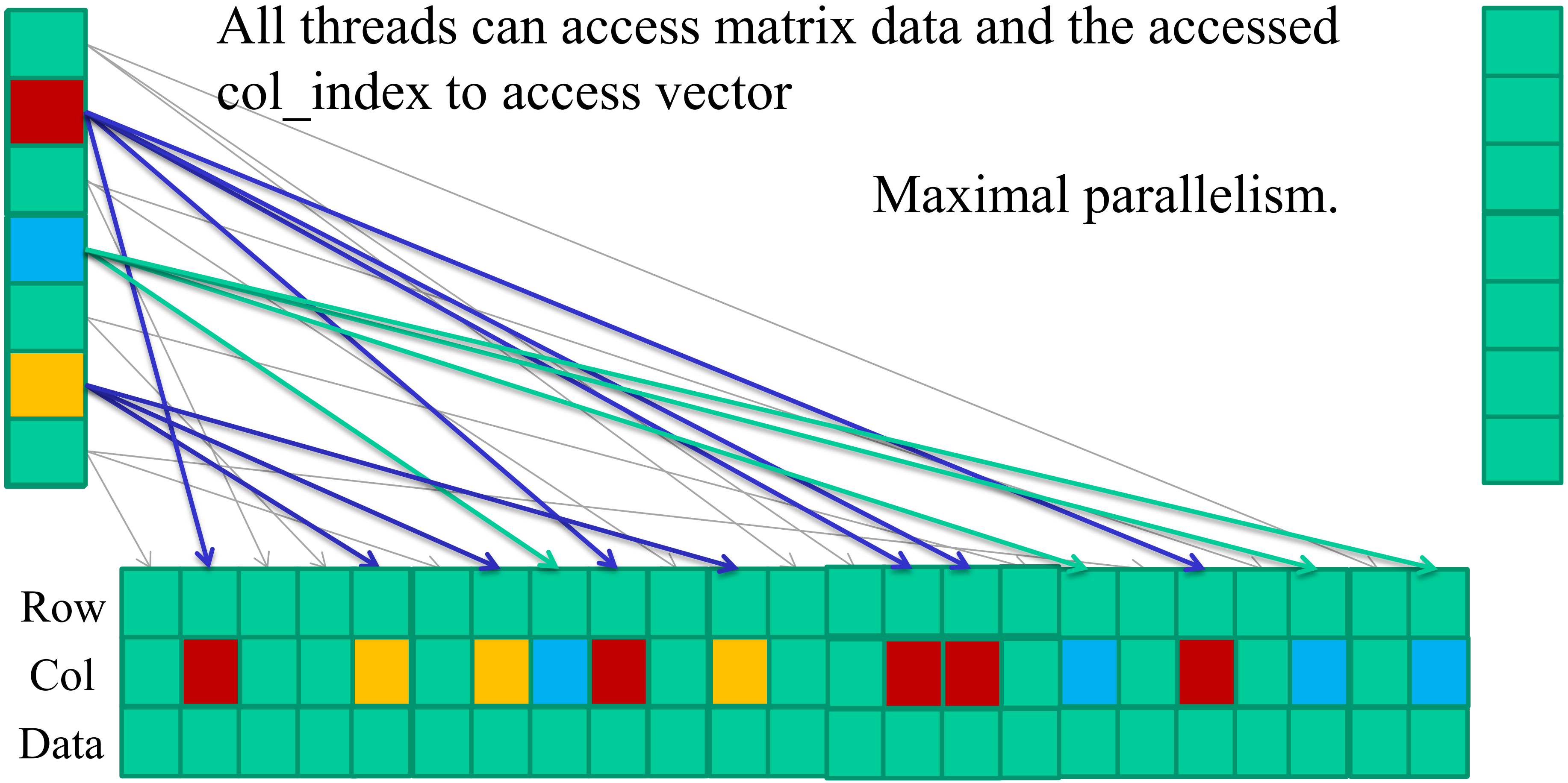
```
for (int i = 0; i < num_elem; i++)
  y[row_index[i]] += data[i] * x[col_index[i]];
```

a sequential loop that implements SpMV/COO

COO Kernel Design
Accessing Input Matrix and Vector

All threads can access matrix data and the accessed col_index to access vector

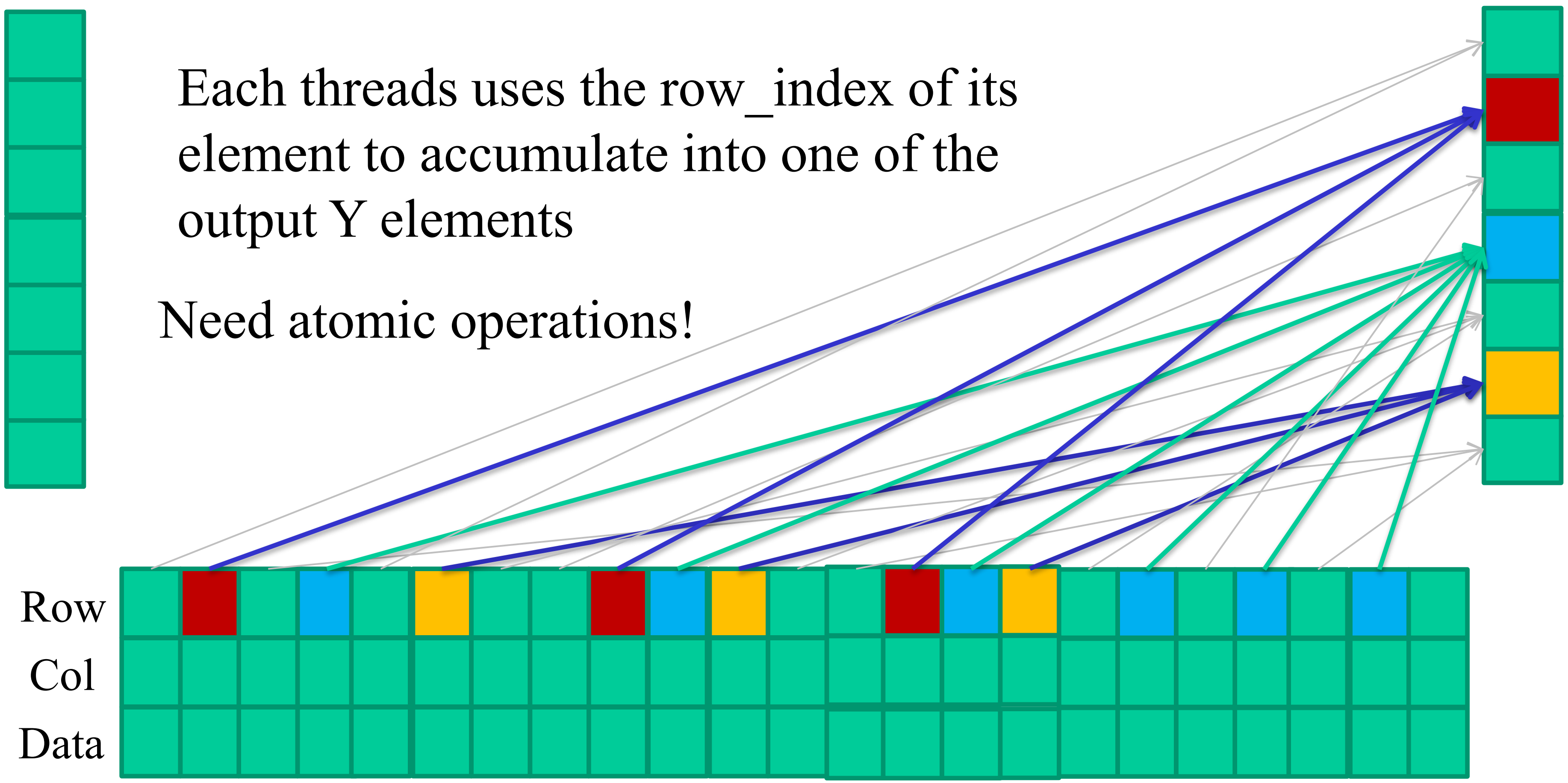
Maximal parallelism.



COO kernel Design
Accumulating into Output Vector

Each threads uses the row_index of its element to accumulate into one of the output Y elements

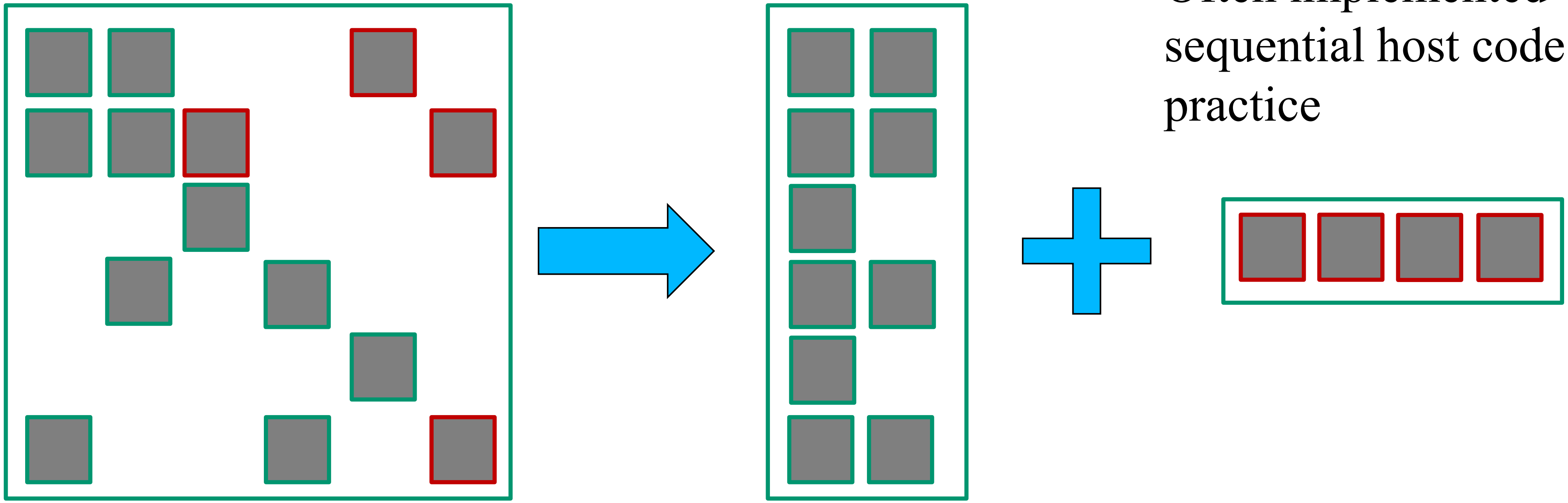
Need atomic operations!



Hybrid Format (ELL + COO)

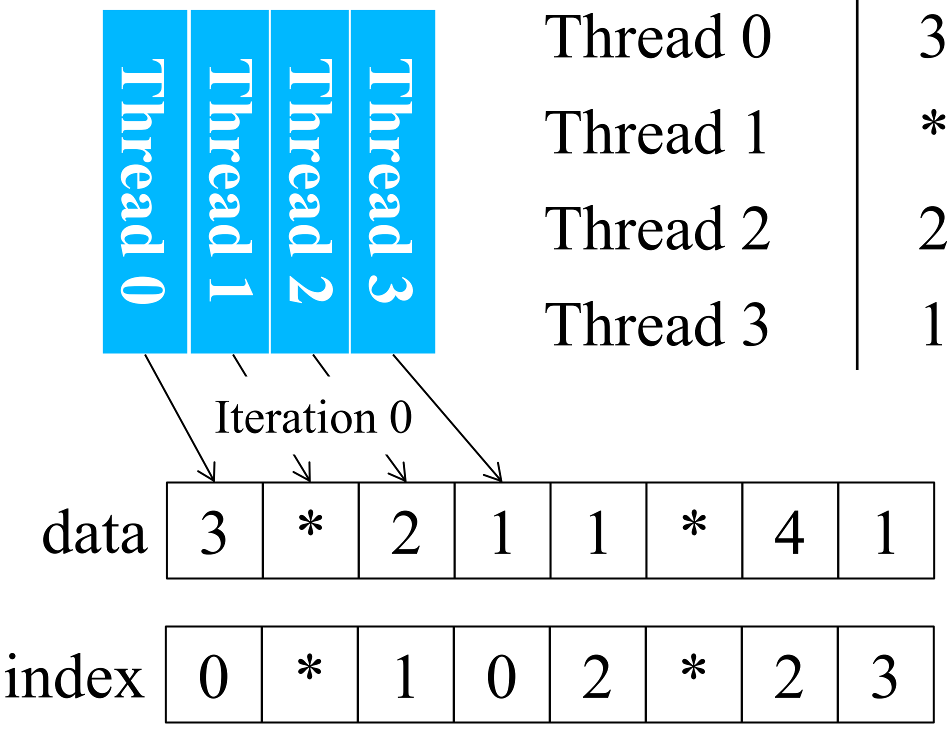
- ELL handles *typical* entries
- COO handles *exceptional* entries
  - Implemented with segmented reduction

Often implemented in sequential host code in practice



Reduced Padding with Hybrid Format

	data		col_index	
Thread 0	3	1	0	2
Thread 1	*	*	*	*
Thread 2	2	4	1	2
Thread 3	1	1	0	3

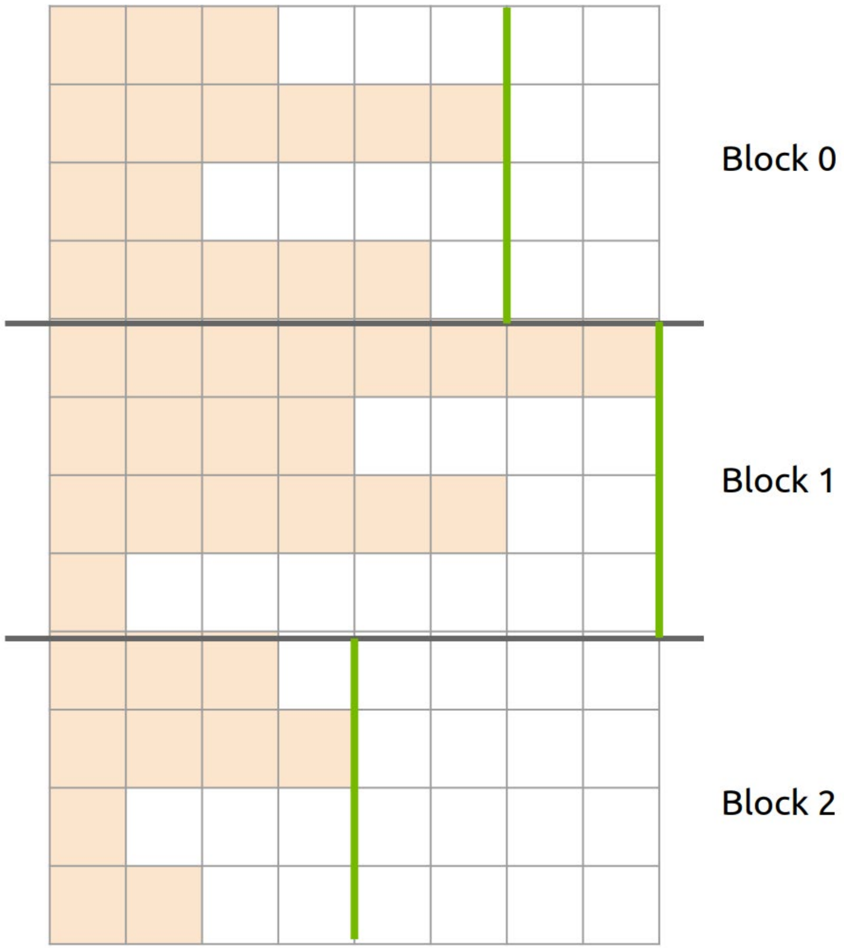


ELL

data	1
col_index	3
row_index	2

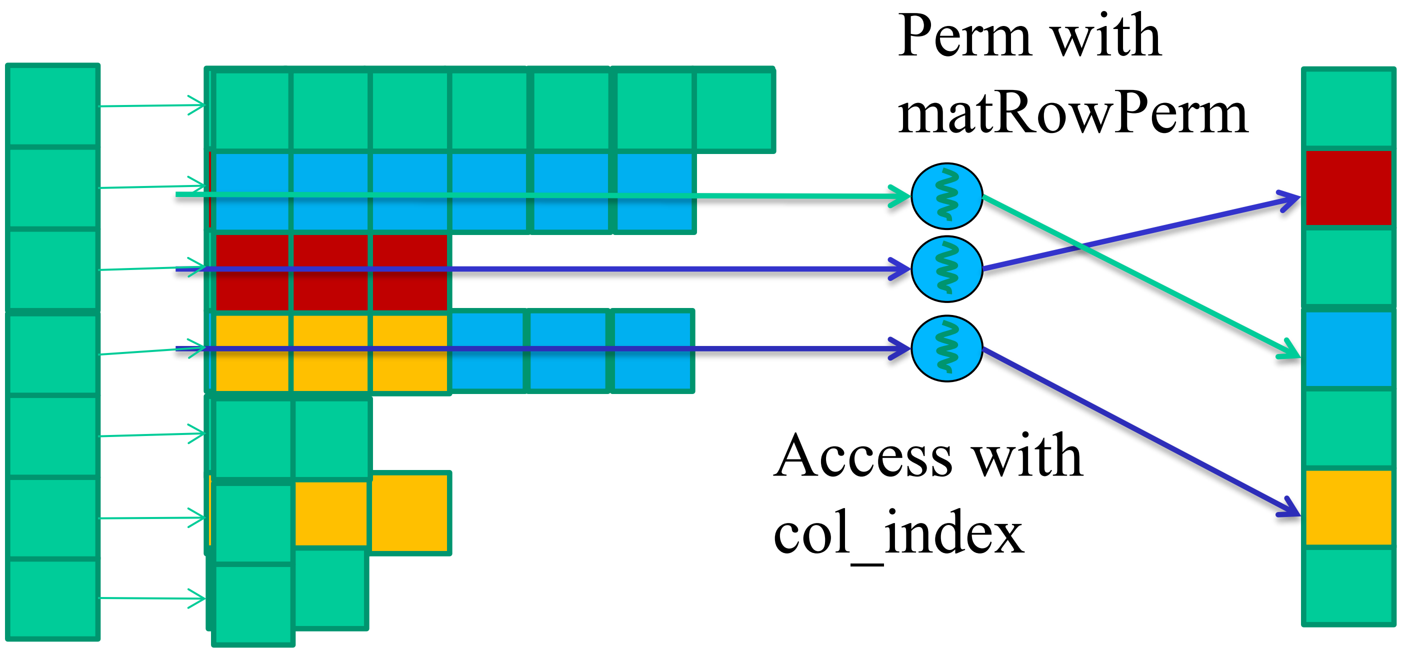
COO

CSR Run-time



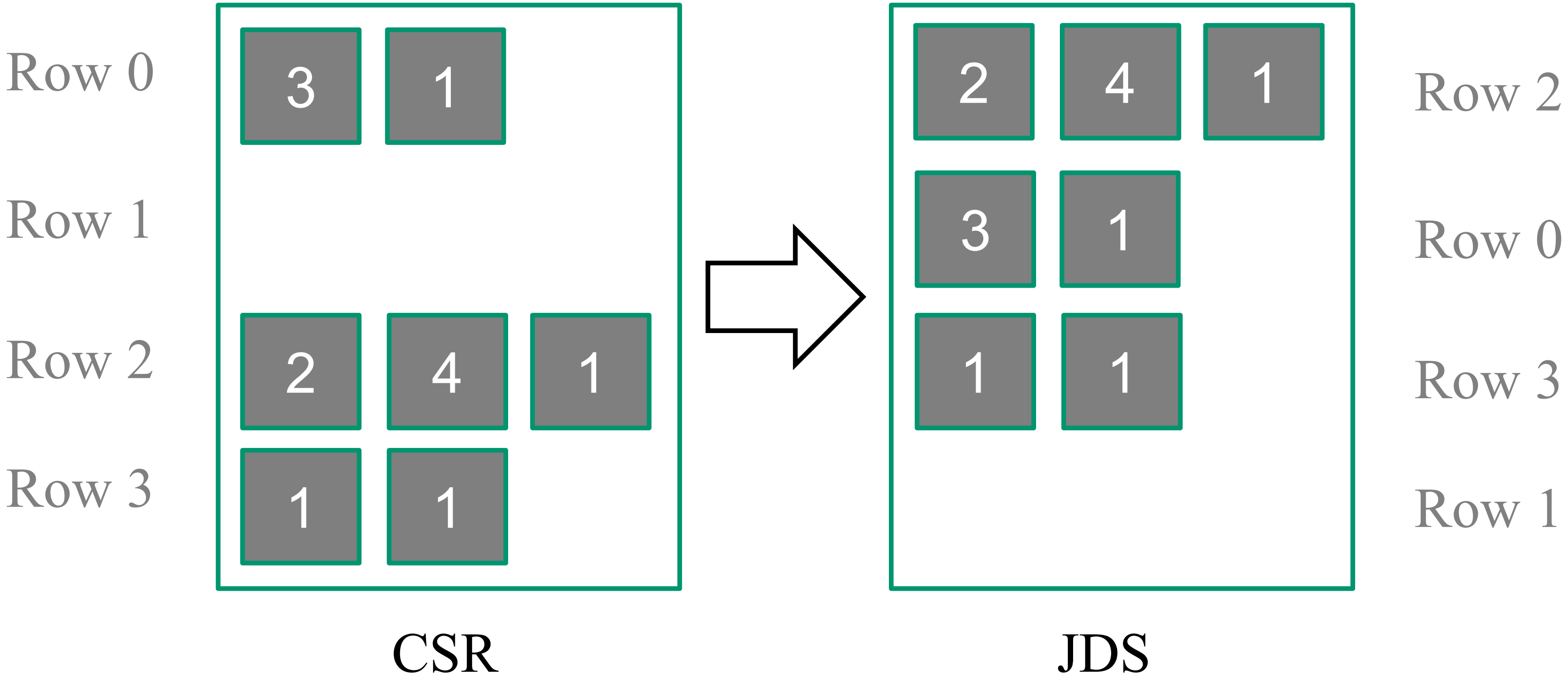
Block performance is determined by longest row

JDS (Jagged Diagonal Sparse) Kernel Design for Load Balancing



Sort rows into descending order according to number of non-zero. Keep track of the original row numbers so that the output vector can be generated correctly.

Sorting Rows According to Length (Regularization)



CSR to JDS Conversion

		Row 0	Row 2	Row 3	
Nonzero values	data[7]	{ 3, 1,	2, 4, 1,	1, 1	}
Column indices	col_index[7]	{ 0, 2,	1, 2, 3,	0, 3	}
Row Pointers	row_ptr[5]	{ 0, 2,	2,	5, 7	}

		Row 2	Row 0	Row 3	
Nonzero values	data[7]	{ 2, 4, 1,	3, 1,	1 1	}
Column indices	col_index[7]	{ 1, 2, 3,	0, 2,	0, 3	}
JDS Row Pointers	jds_row_ptr[5]	{0,	3,	5,	7, 7 }
JDS Row Indices	jds_row_perm[4]	{2,	0,	3,	1 }

JDS Summary

Nonzero values	data[7]	{ 2, 4, 1, 3, 1, 1, 1 }
Column indices	jds_col_index[7]	{ 1, 2, 3, 0, 2, 0, 3 }
JDS row indices	jds_row_perm[4]	{ 2, 0, 3, 1 }
JDS Row Ptrs	jds_row_ptr[5]	{ 0, 3, 5, 7, 7 }

2	4	1	3	1	1	1

1	2	3	0	0	2	3

A Parallel SpMV/JDS Kernel

```
__global__ void SpMV_JDS(int num_rows, float *data, int *col_index,
                int *jds_row_ptr, int *jds_row_perm, float *x, float *y)
  {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < num_rows) {
      float dot = 0;
      int row_start = jds_row_ptr[row];
      int row_end =   jds_row_ptr[row+1];
      for (int elem = row_start; elem < row_end; elem++) {
        dot += data[elem] * x[col_index[elem]];
      }
      y[jds_row_perm[row]] = dot;
    }
  }
```

		Row 2	Row 0	Row 3	
Nonzero values	data[7]	{ 2, 4, 1,	3, 1,	1 1	}
Column indices	col_index[7]	{ 1, 2, 3,	0, 2,	0, 3	}
JDS Row Pointers	jds_row_ptr[5]	{0,	3,	5,	7,7 }
JDS Row Indices	jds_row_perm[4]	{2,	0,	3,	1 }

JDS vs. CSR - Control Divergence

- Threads still execute different number of iterations in the JDS kernel for-loop
  - However, neighboring threads tend to execute similar number of iterations because of sorting.
  - Better thread utilization, less control divergence

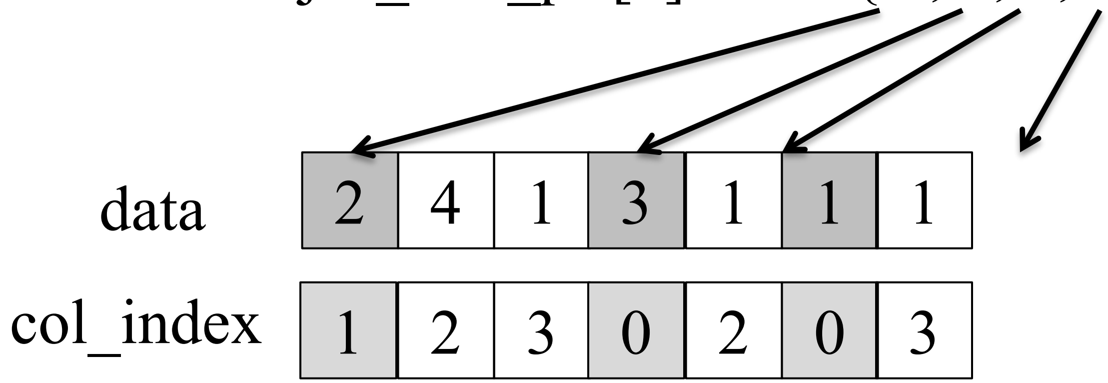
Nonzero values	data[7]	{ 2, 4, 1, 3, 1, 1, 1 }
Column indices	col_index[7]	{ 1, 2, 3, 0, 2, 0, 3 }
JDS row indices	jds_row_perm[4]	{ 2, 0, 3, 1 }
JDS Row Ptrs	jds_row_ptr[5]	{ 0, 3, 5, 7, 7 }

data	2	4	1	3	1	1	1
col_index	1	2	3	0	2	0	3

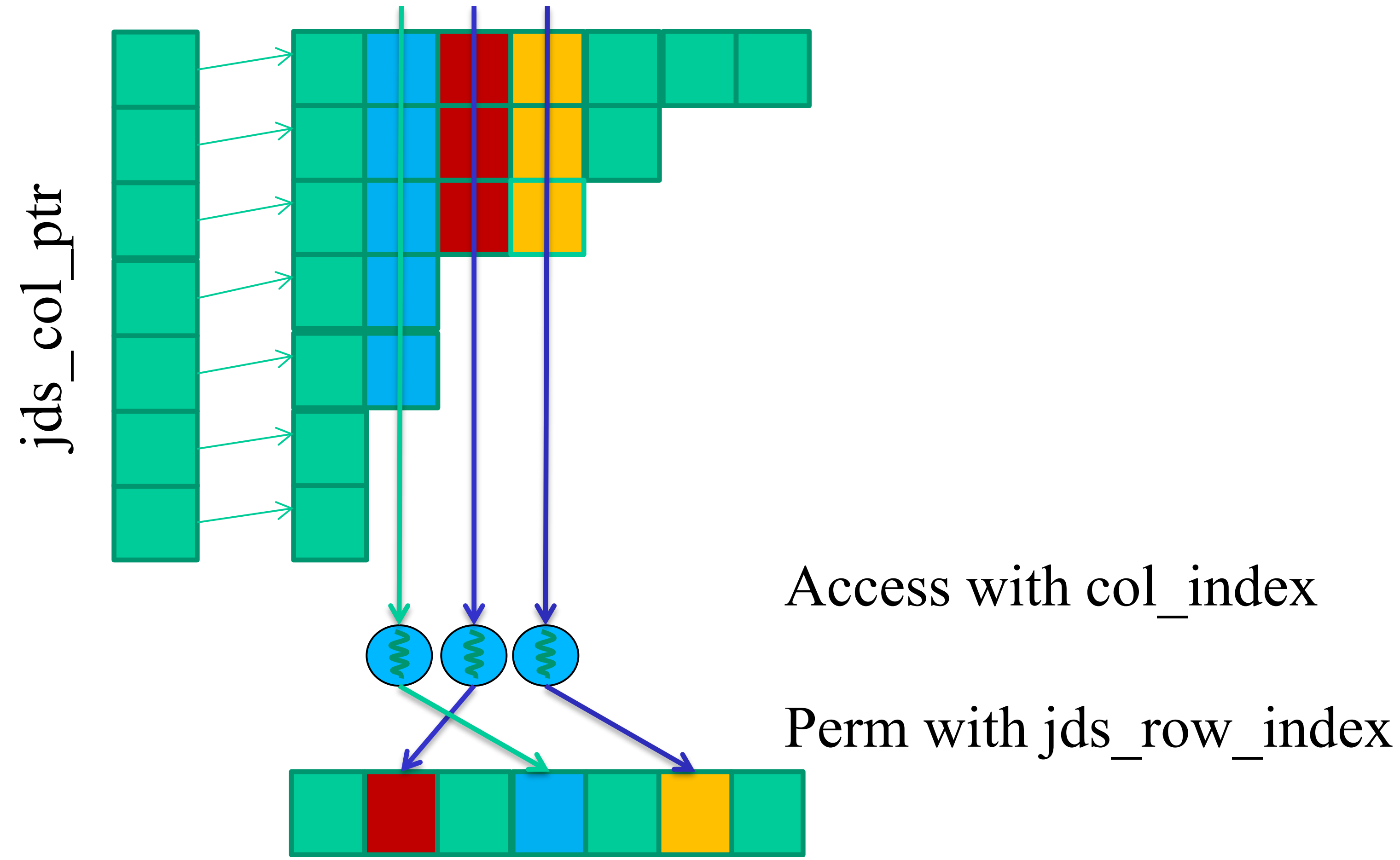
JDS vs. CSR Memory Divergence

- Adjacent threads still access non-adjacent memory locations

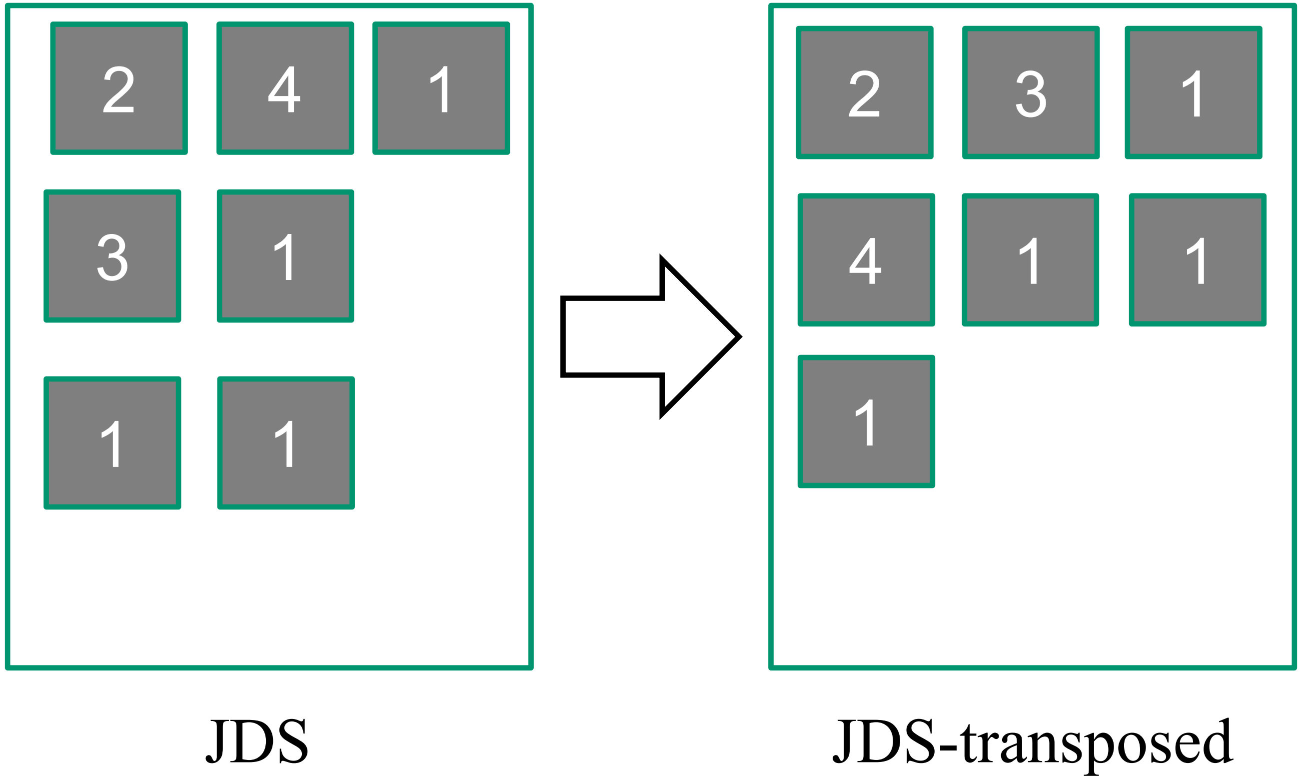
Nonzero values	data[7]	{ 2, 4, 1, 3, 1, 1, 1 }
Column indices	col_index[7]	{ 1, 2, 3, 0, 2, 0, 3 }
JDS row indices	jds_row_perm[4]	{ 2, 0, 3, 1 }
JDS Row Ptrs	jds_row_ptr[5]	{ 0, 3, 5, 7, 7 }



JDS with Transposition



Transposition for Memory Coalescing



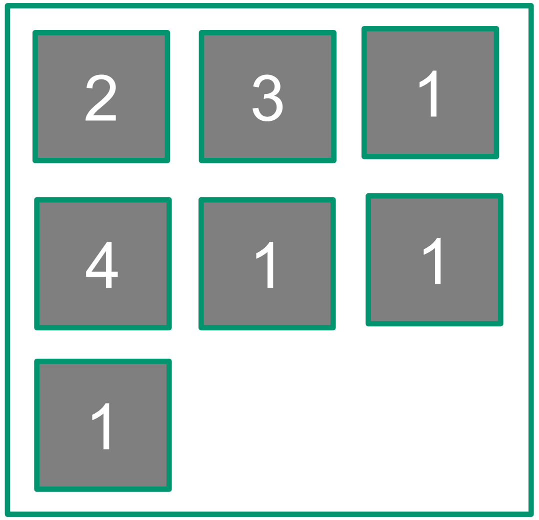
JDS Format with Transposed Layout

Row 0	3	0	1	0	Thread 0
Row 1	0	0	0	0	Thread 1
Row 2	0	2	4	1	Thread 2
Row 3	1	0	0	1	Thread 3

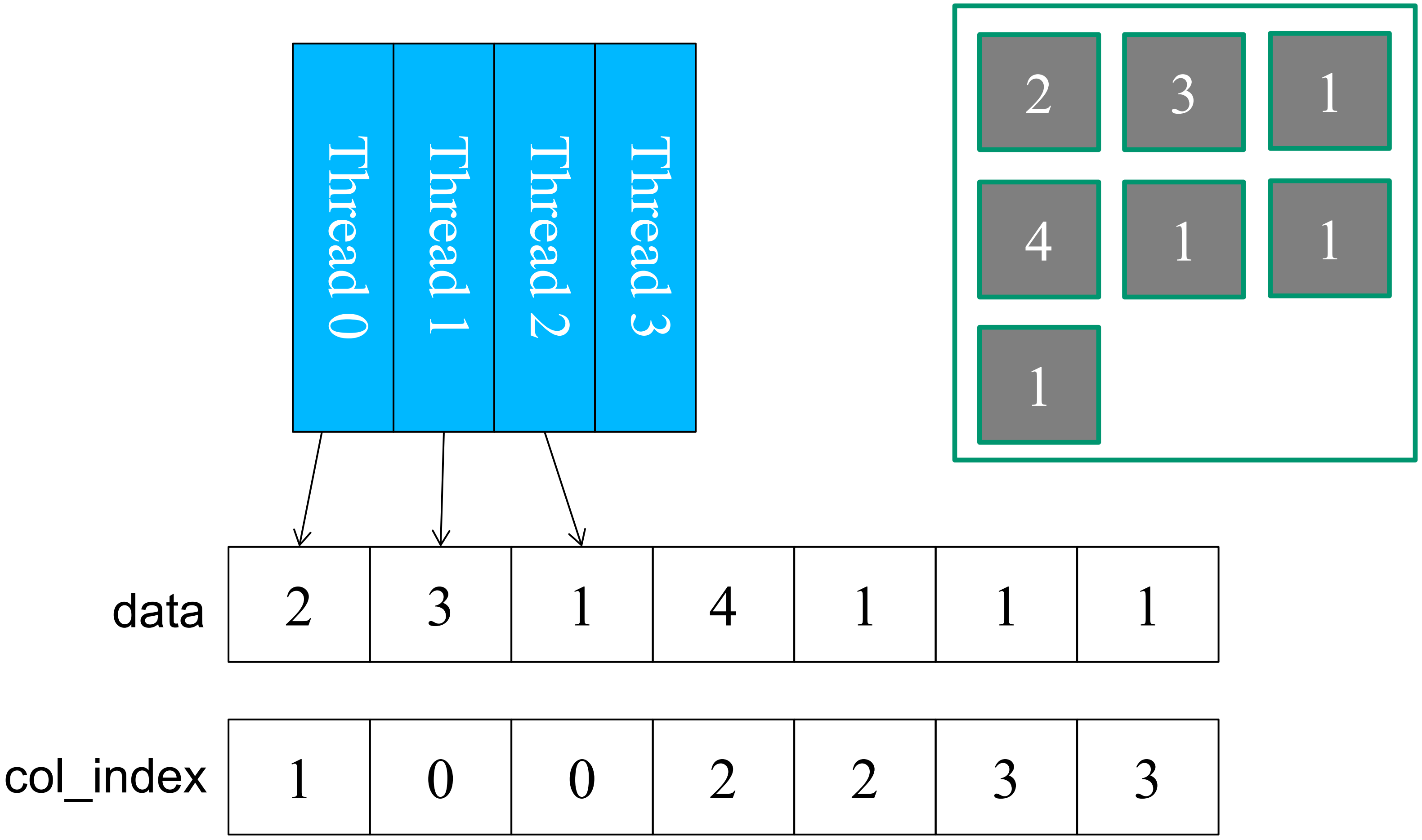
JDS row indices jds_row_perm[4] { 2, 0, 3, 1 }

JDS column pointers jds_t_col_ptr[4] { 0, 3, 6, 7 }

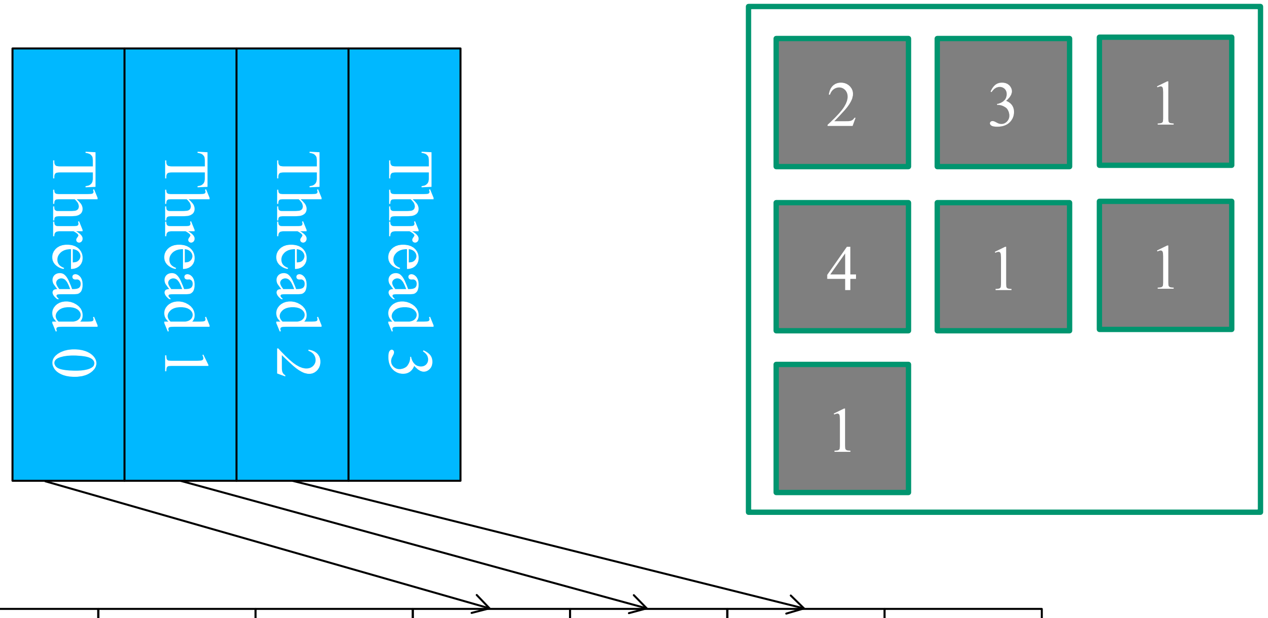
data	2	3	1	4	1	1	1
col_index	1	0	0	2	2	3	3



JDS with Transposition: Memory Coalescing



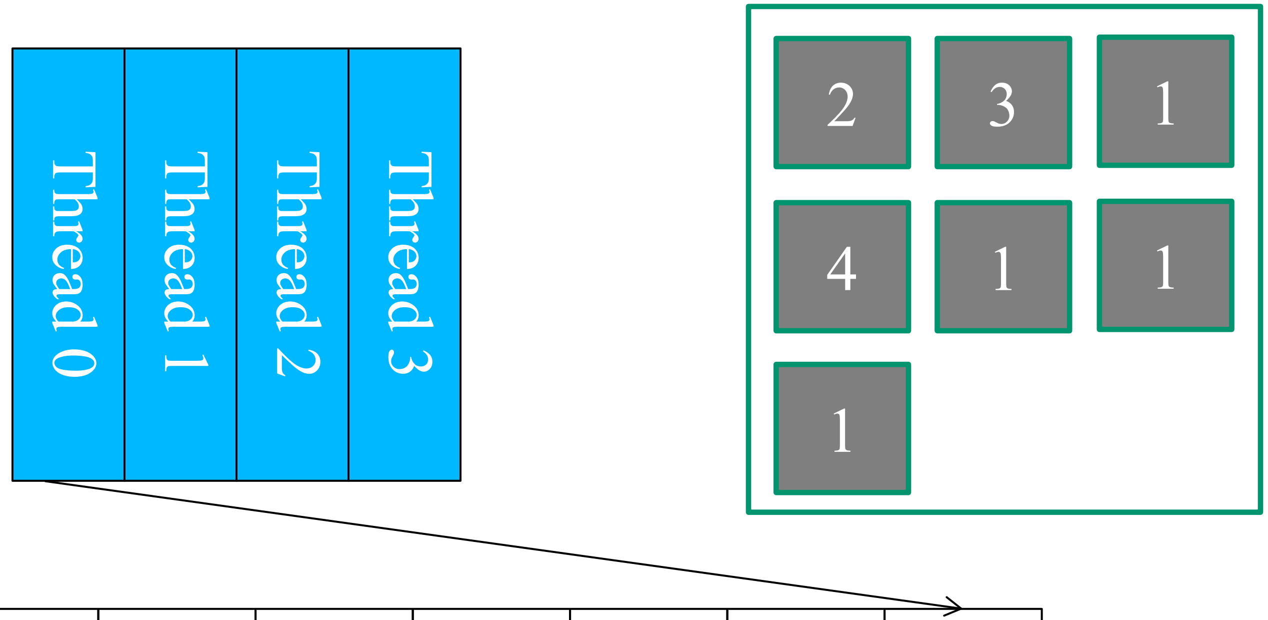
JDS with Transposition: Memory Coalescing



data	2	3	1	4	1	1	1
col_index	1	0	0	2	2	3	3

Not aligned with DRAM bursts but OK with recent GPUs

JDS with Transposition: Memory Coalescing



data	2	3	1	4	1	1	1
col_index	1	0	0	2	2	3	3

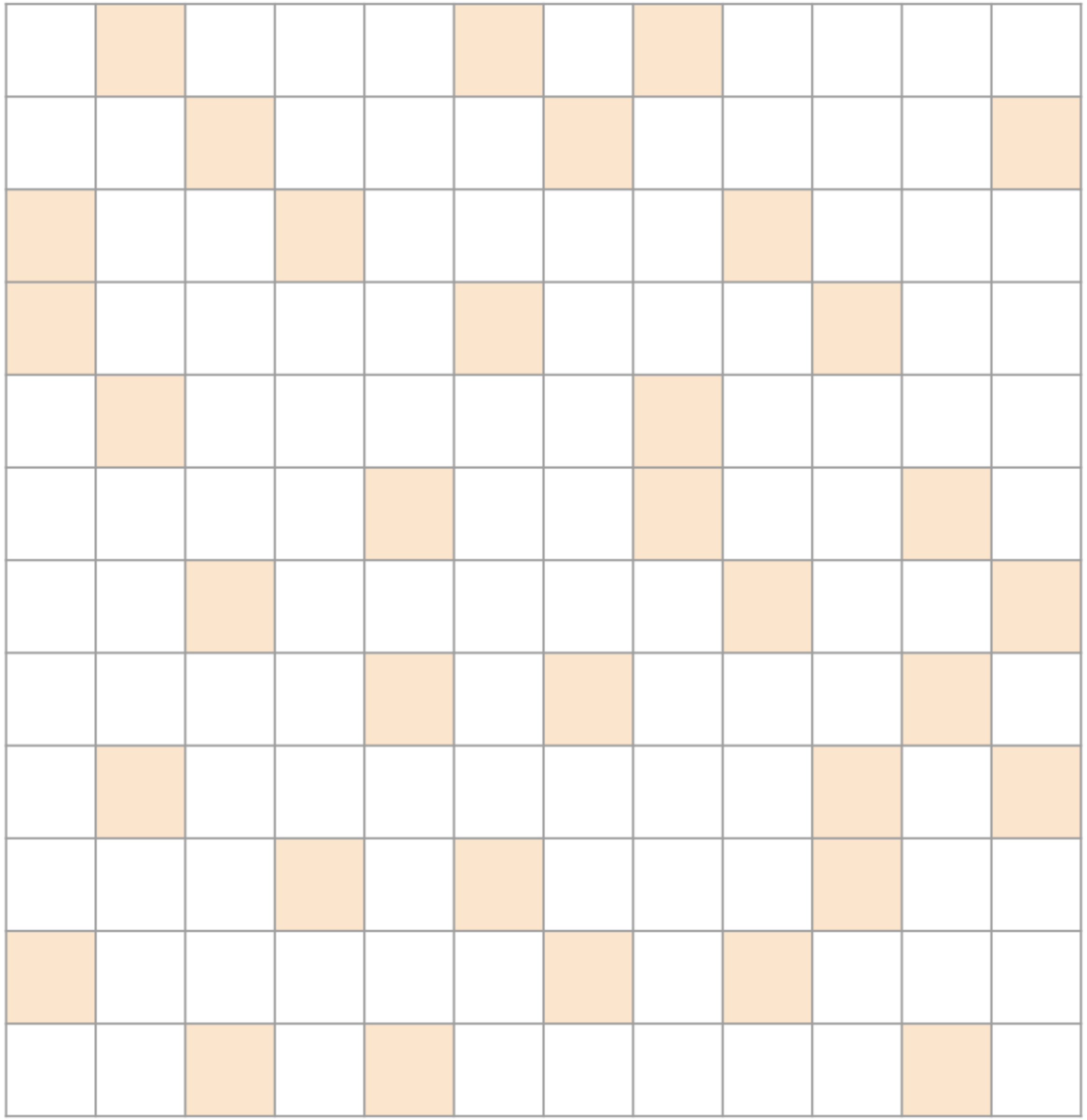
A Parallel SpMV/JDS_T Kernel

```
__global__ void SpMV_JDS_T(int num_rows, float *data, int *col_index,
          int *jds_t_col_ptr, int *jds_row_perm, float *x, float *y)
 {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < num_rows) {
      float dot = 0;
      unsigned int sec = 0;
      while (jds_t_col_ptr[sec+1] - jds_t_col_ptr[sec] > row) {
         dot += data[jds_t_col_ptr[sec]+row] *
                x[col_index[jds_t_col_ptr[sec]+row]];
         sec++;
      }
      y[jds_row_perm[row]] = dot;
    }
 }
```

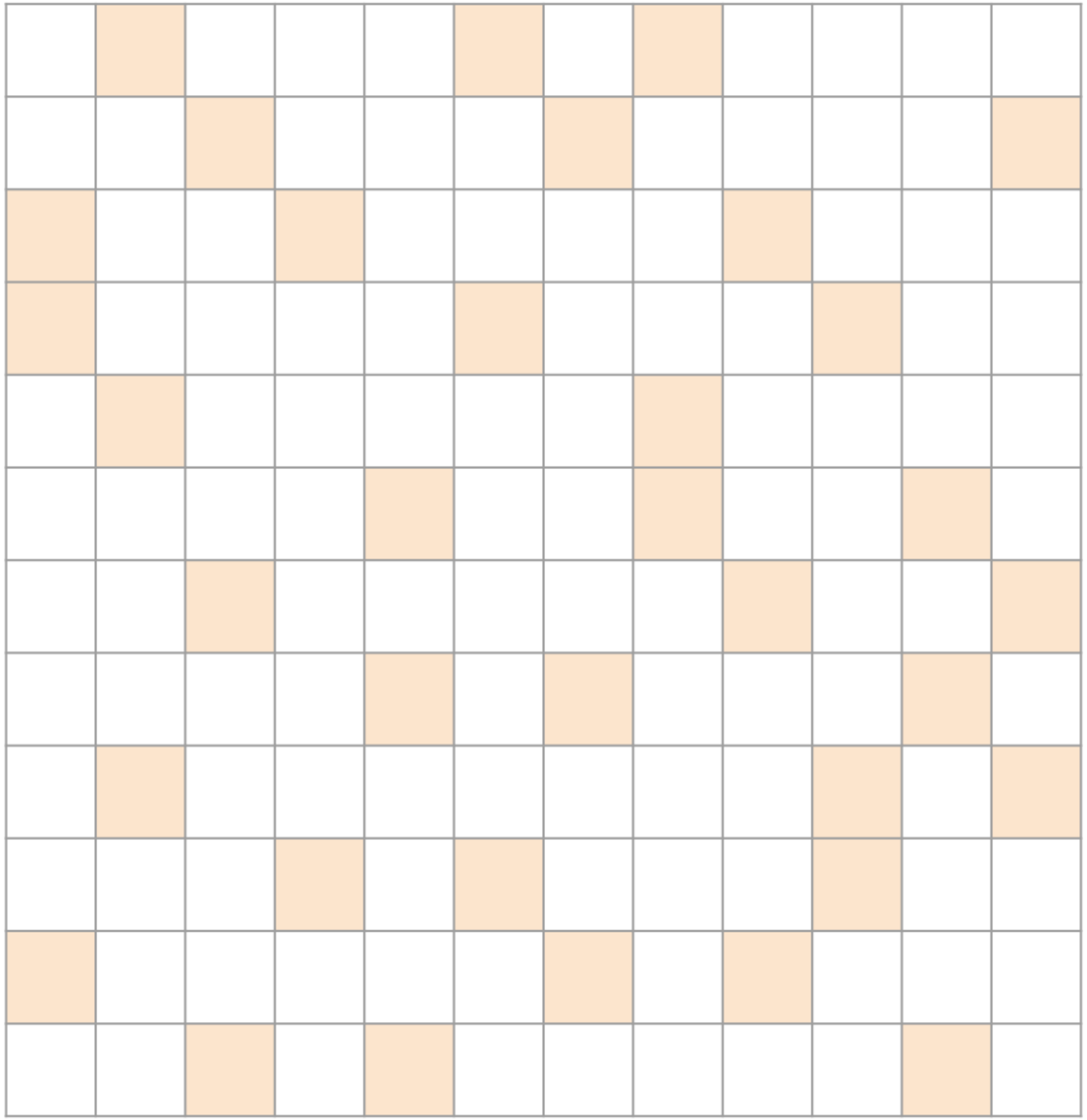
		Sec 0	Sec 1	Sec 2
Nonzero values	data[7]	{ 2, 3, 1,	4, 1, 1	1 }
Column indices	col_index[7]	{ 1, 0, 0,	2, 2, 3	3 }
JDS_T Column Pointers	jds_t_col_ptr[5]	{0,	3, 6,	7,7 }
JDS Row Indices	jds_row_perm[4]	{2,	0, 3,	1 }

Lab 8 Variable Names

JDS_T Length of Cols matRows[4]	{3,	2,	2,	0 }
	Sec 0	Sec 1	Sec 2	
Nonzero values matData[7]	{ 2, 3, 1,	4, 1, 1	1 }	
Column indices matCols[7]	{ 1, 0, 0,	2, 2, 3	3 }	
JDS_T Column Pointers matColStart[4]	{0,	3,	6,	7 }
JDS Row Indices matRowPerm[4]	{2,	0,	3,	1 }



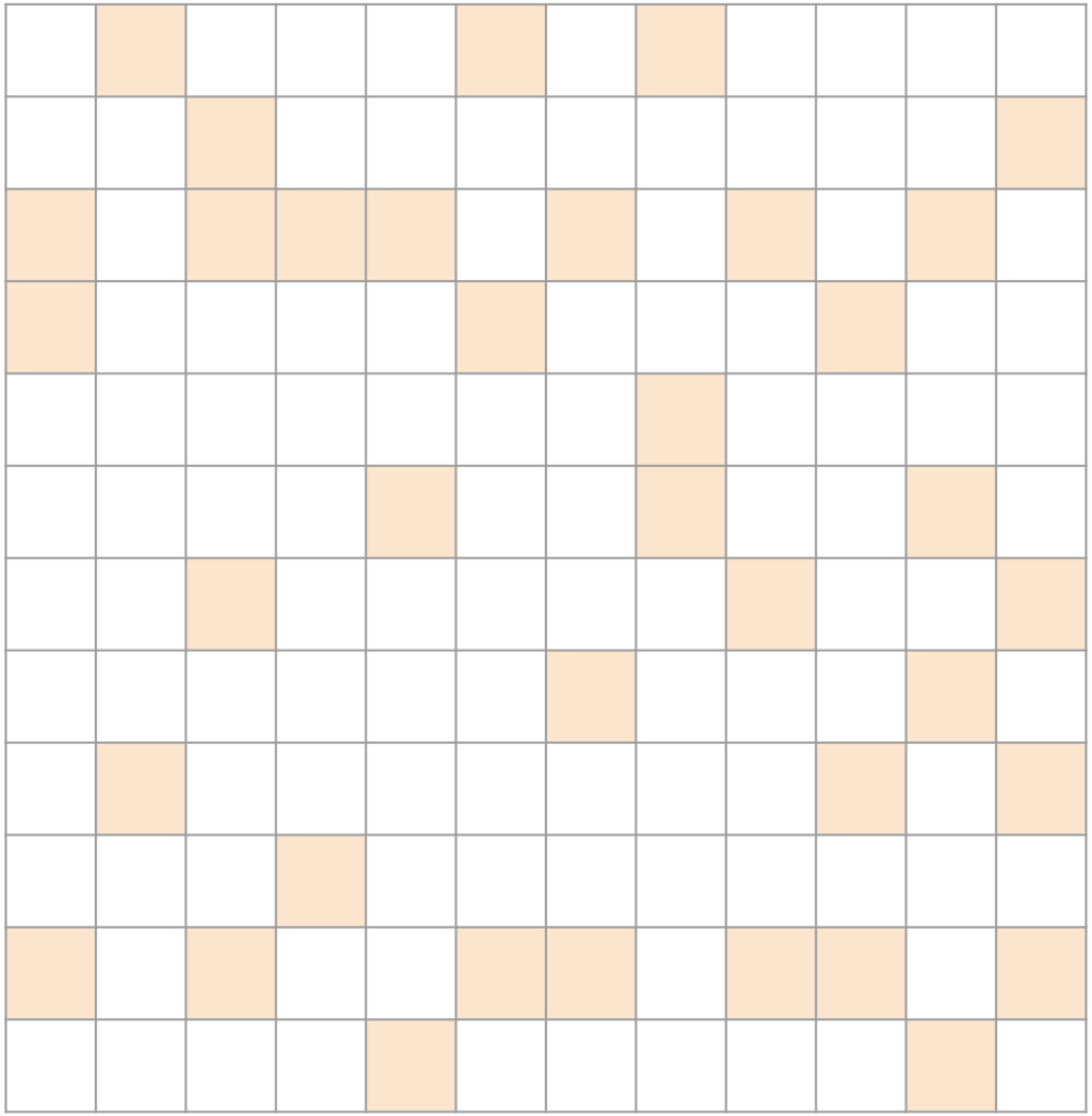
Roughly Random…



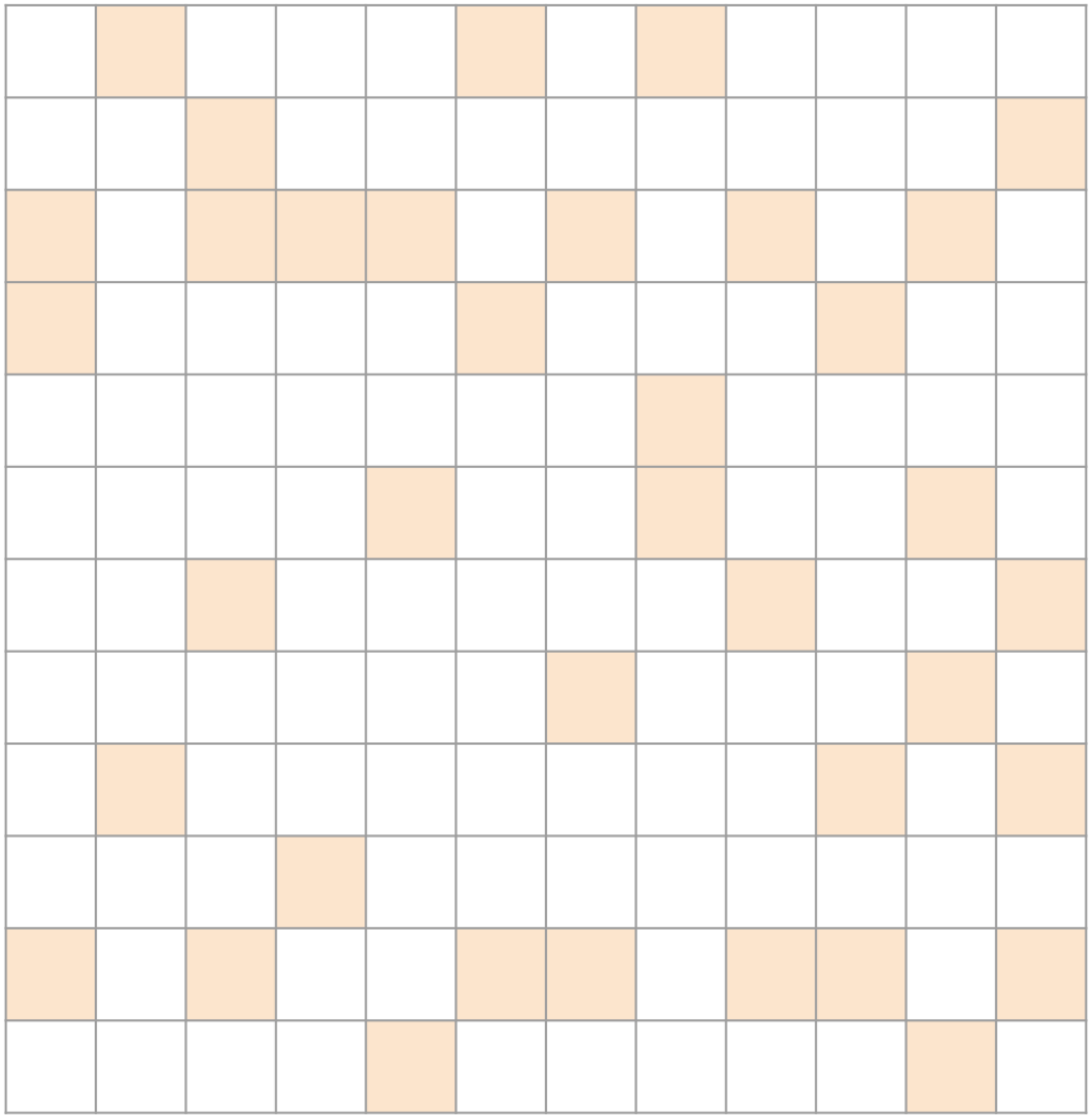
Roughly Random…

Probably best with ELL.

- Padding will be uniformly distributed
- Sparse representation will be uniform



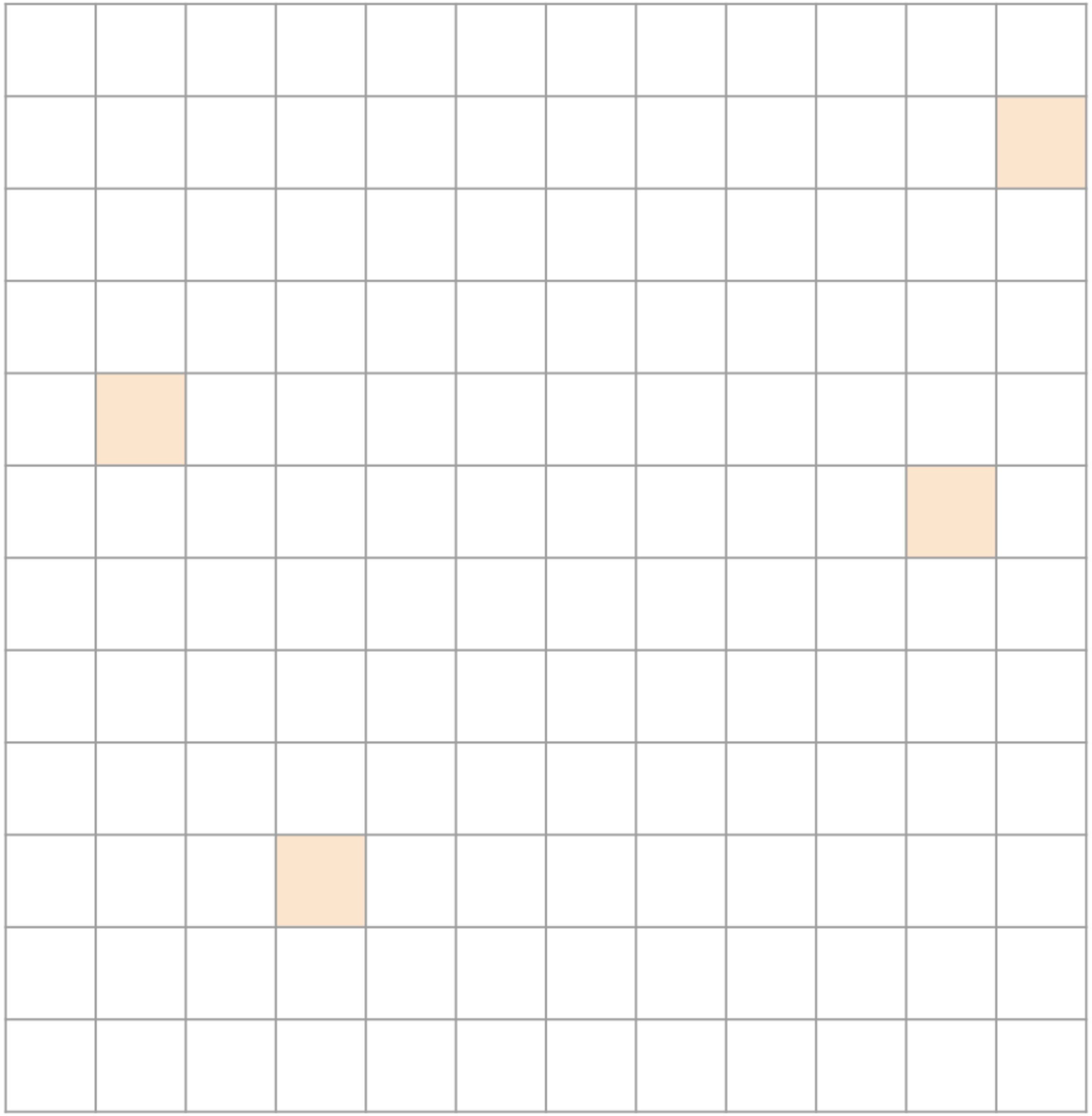
High variance in rows…



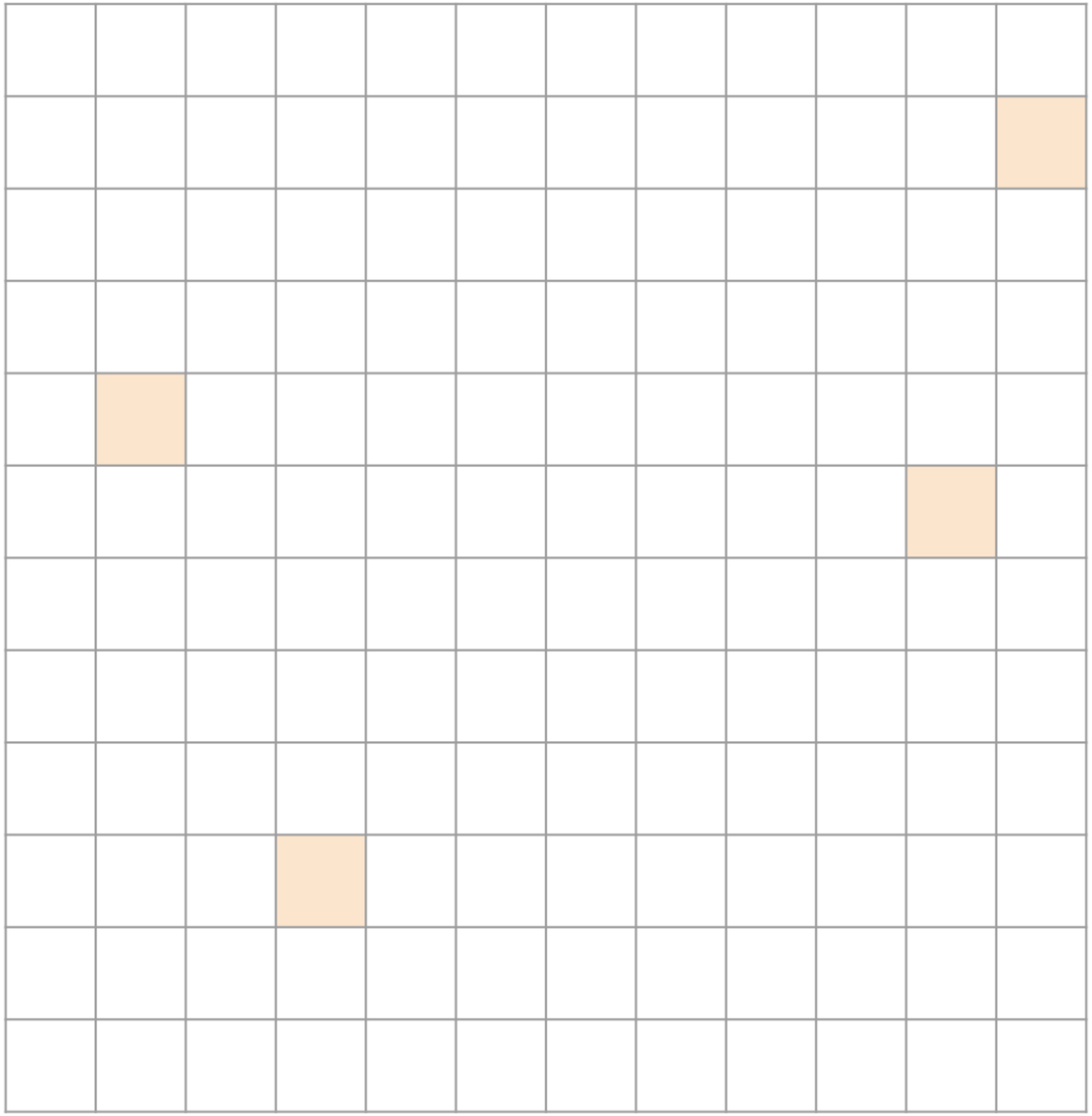
High variance in rows

Probably best with ELL/COO

- Benefit of ELL for most cases
- Outliers are captured with COO



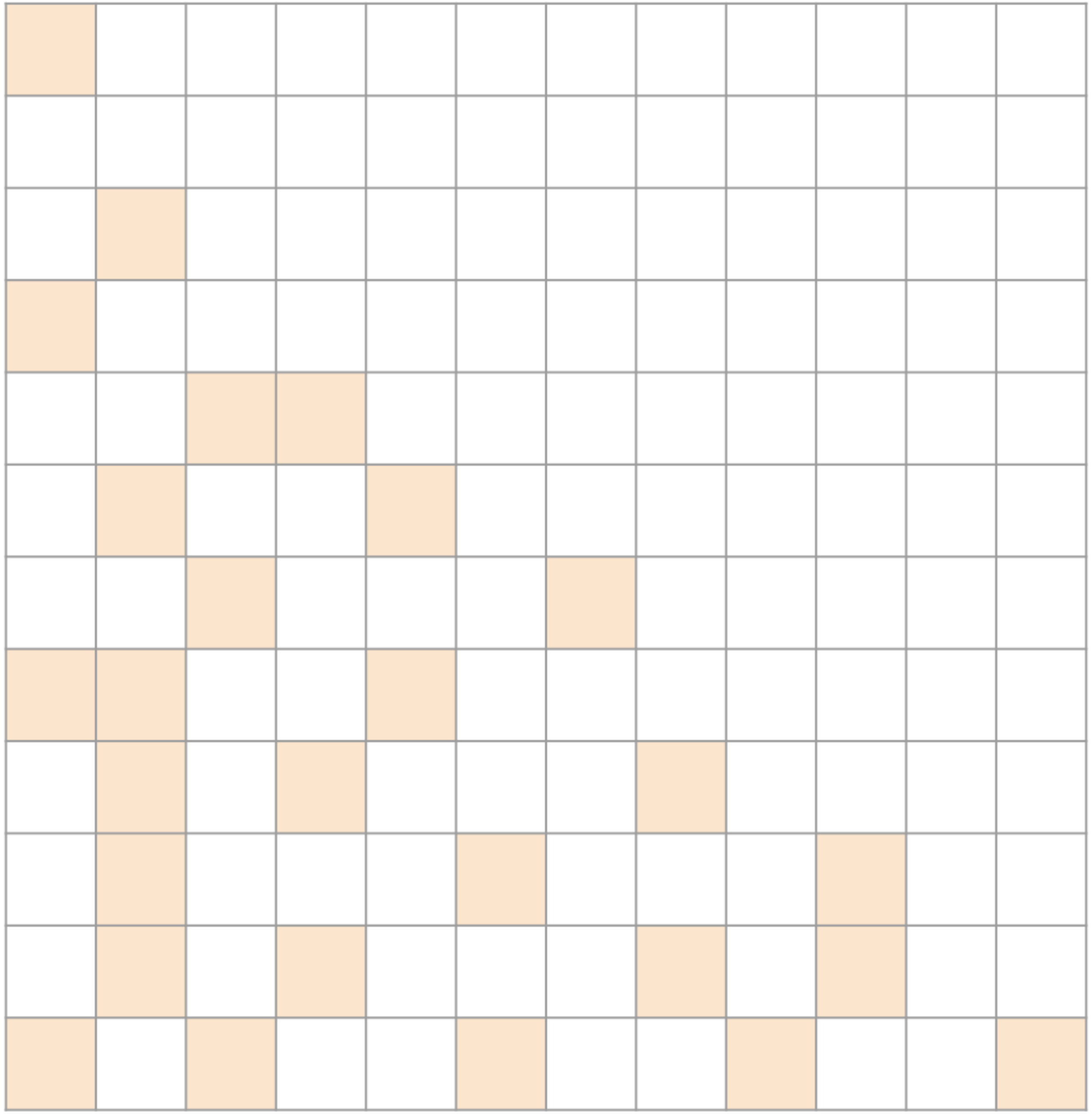
Very sparse…



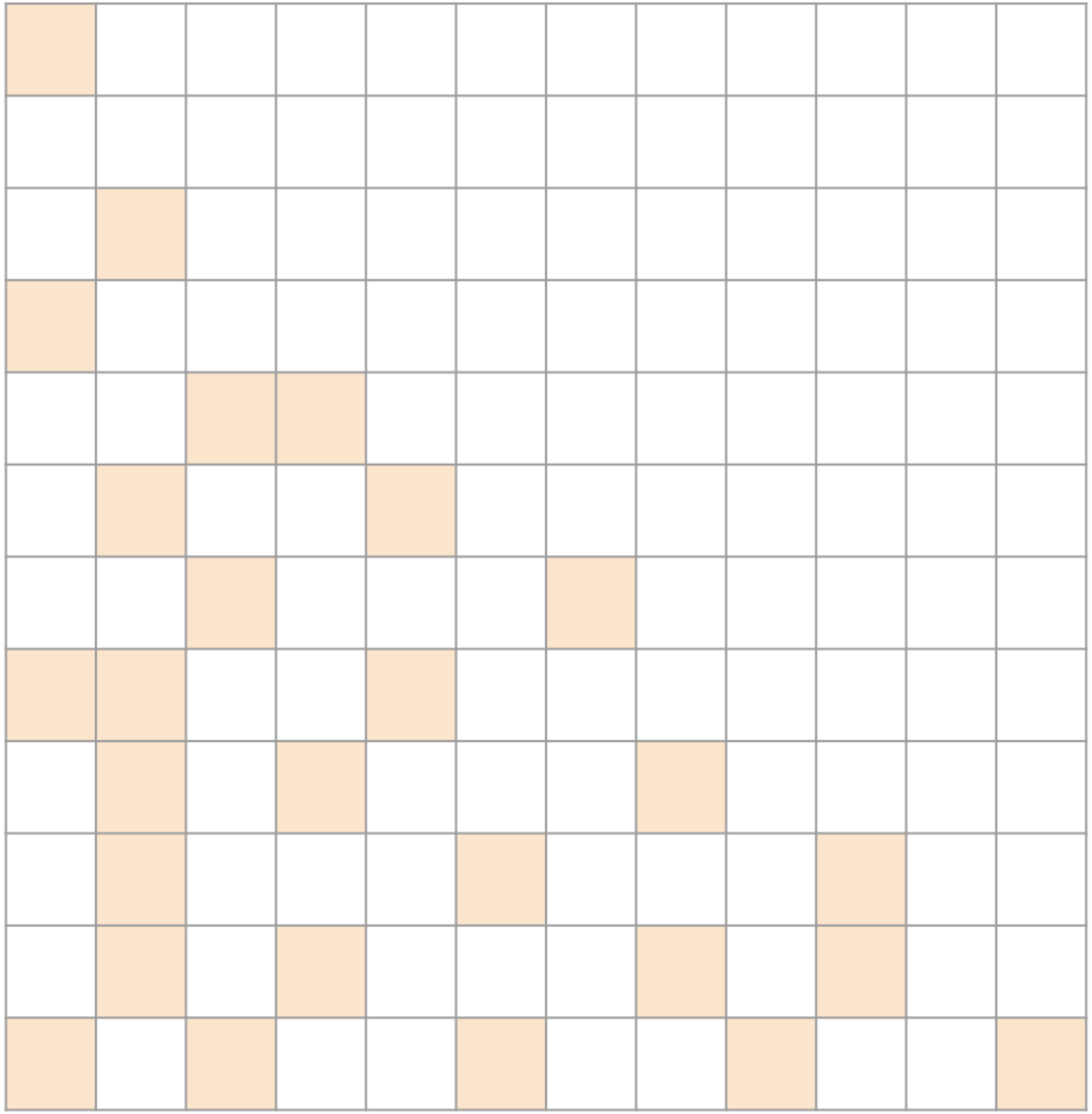
Very sparse

Probably best with COO

- Not a lot of data, compute is sparse



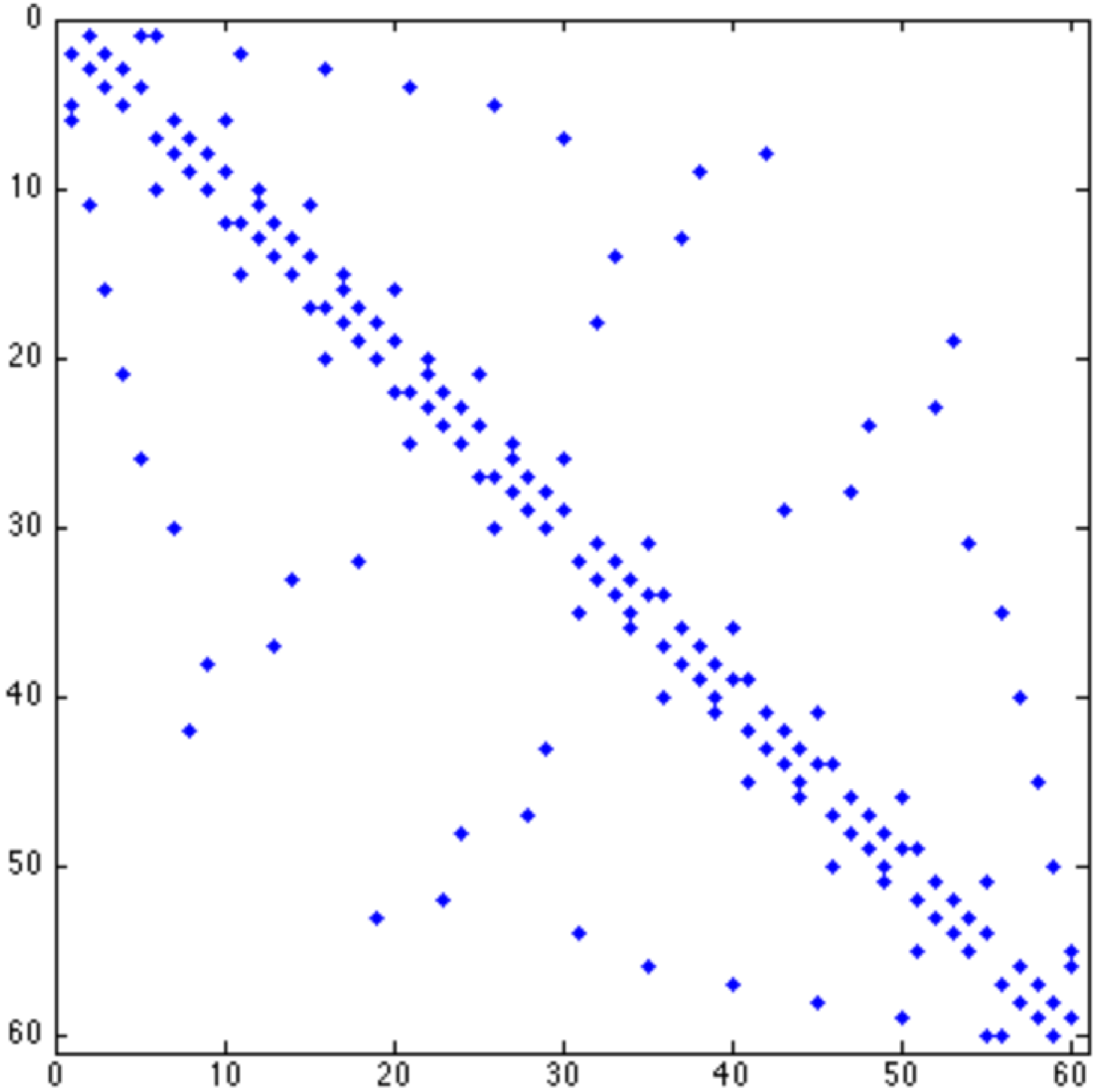
Roughly triangular…



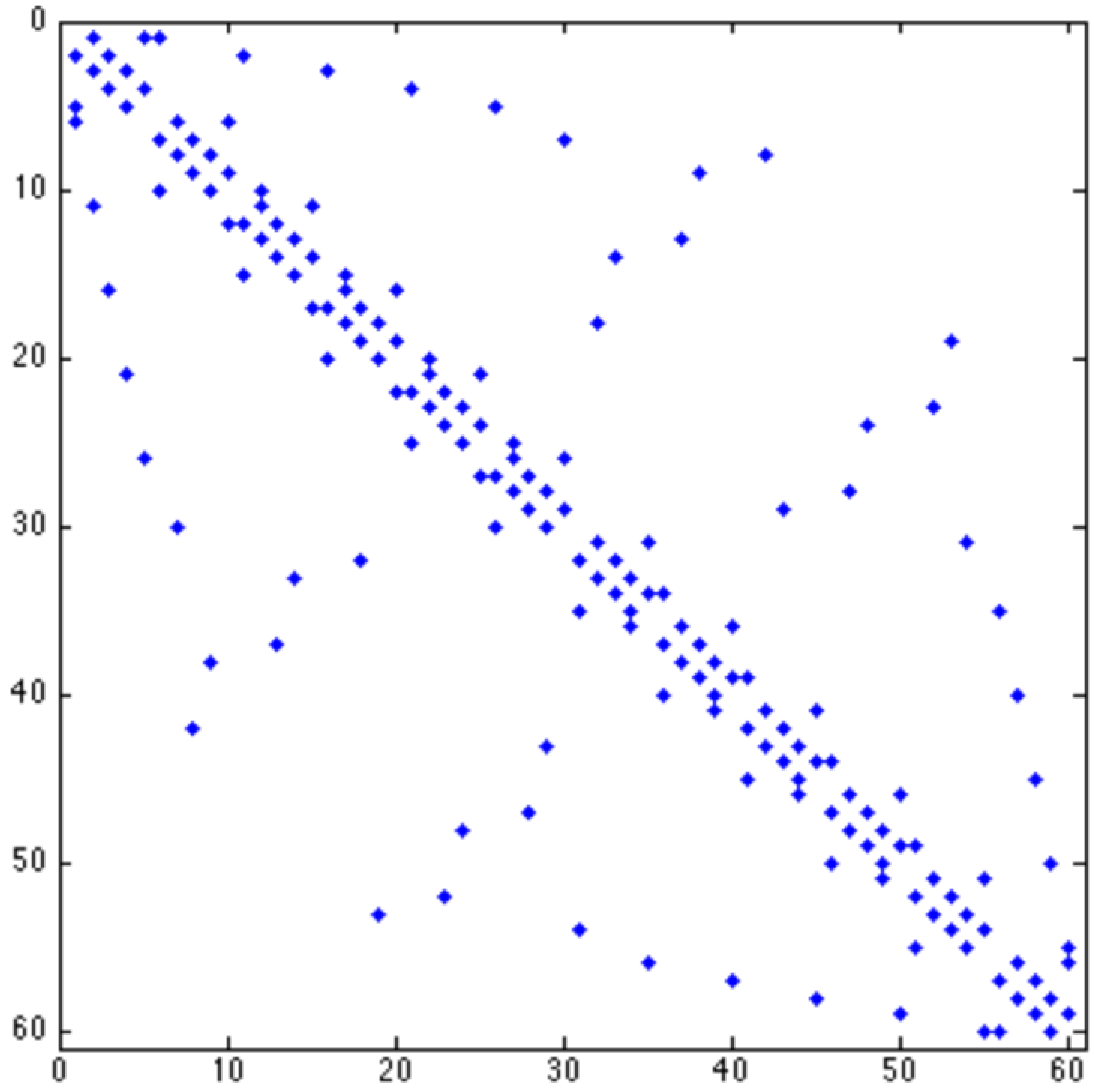
Roughly triangular…

Probably best with JDS

- Takes advantage of sparsity structure

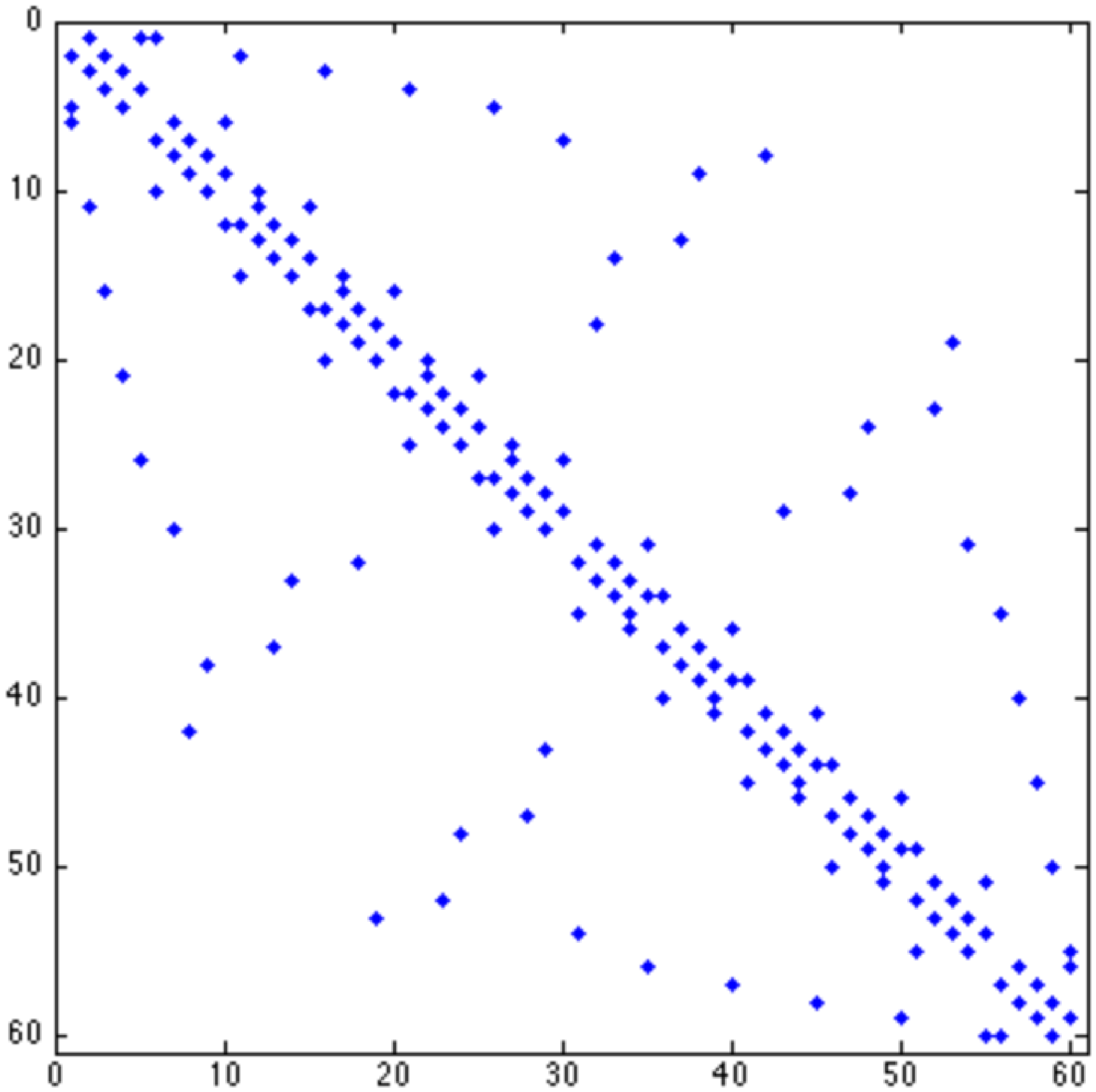


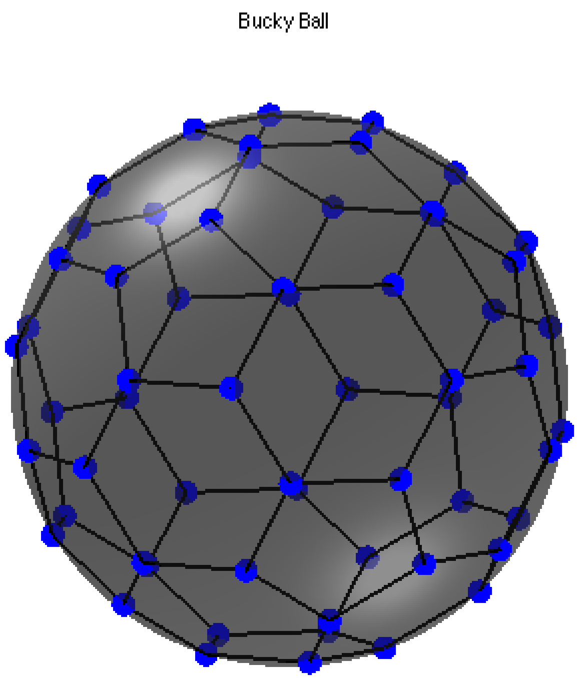
Banded Matrix…



Banded Matrix…

Probably best with ELL
- Small amount of variance in rows


0
10
20
30
40
50
60
0
10
20
30
40
50
60


Bucky Ball

Other formats

- Diagonal (DIA): for strictly banded/diagonal matrices
- Packet (PKT): create diagonal submatrices by reordering rows/cols
- Dictionary of Keys (DOK): map of (row/col) to data
- Compressed Sparse Column (CSC): when to use over CSR?
- Blocked CSR: useful for block sparse matrices
- Hybrids of these…

Sparse Matrices as Foundation for Advanced Algorithm Techniques

- Graphs are often represented as sparse adjacency matrices
  - Used extensively in social network analytics, natural language processing, etc.
  - Sparse Matrix-Matrix multiplication (SpMM) is a fundamental operator in GNNs, which performs a multiplication between a sparse matrix and a dense matrix.
- Binning techniques often use sparse matrices for data compaction
  - Used extensively in ray tracing, particle-based fluid dynamics methods, and games
- These will be covered in ECE508/CS508

ANY MORE QUESTIONS
READ CHAPTER 10

Problem Solving

- Q: Given matrix A, which of the following are correct?

$$A = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 0 & 0 & 1 \end{vmatrix}$$

CSR representation:

Data = [1,2,1,1,2,3,4,1,1]

Col_idx = [0,2,3,0,1,2,3,0,3]

Row_ptr = [0,1,3,7,9]

COO representation

Data = [1,2,1,1,2,3,4,1,1]

Col_idx = [0,2,3,0,1,2,3,0,3]

Row_idx = [0,1,1,3,3,3,3,7,7]

- A: only CSR

Problem Solving

- Q: Consider the following sparse Matrix:

$$\begin{bmatrix} 1 & 0 & 4 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 7 & 0 & 9 & 3 \\ 0 & 0 & 0 & 0 & 0 \\ 6 & 5 & 0 & 0 & 8 \end{bmatrix}$$

- For each of the following **data** layouts in memory, select the option that best matches all the sparse matrix formats that can store the data in memory as depicted.
- A:
  - 1) CSR, COO
  - 2) ??
  - 3) JDS, COO
  - 4) COO
  - 5) JDS-Transposed, COO

Layout 1:

1	4	2	7	9	3	6	5	8

Layout 2:

1	2	7	6	4	0	9	5	0	0	3	8

Layout 3:

7	9	3	6	5	8	1	4	2

Layout 4:

9	7	1	2	4	3	5	8	6

Layout 5:

7	6	1	2	9	5	4	3	8