



ECE408/CS483/CSE408 Fall 2021

Applied Parallel Programming

Lecture 20

Parallel Sparse Methods II

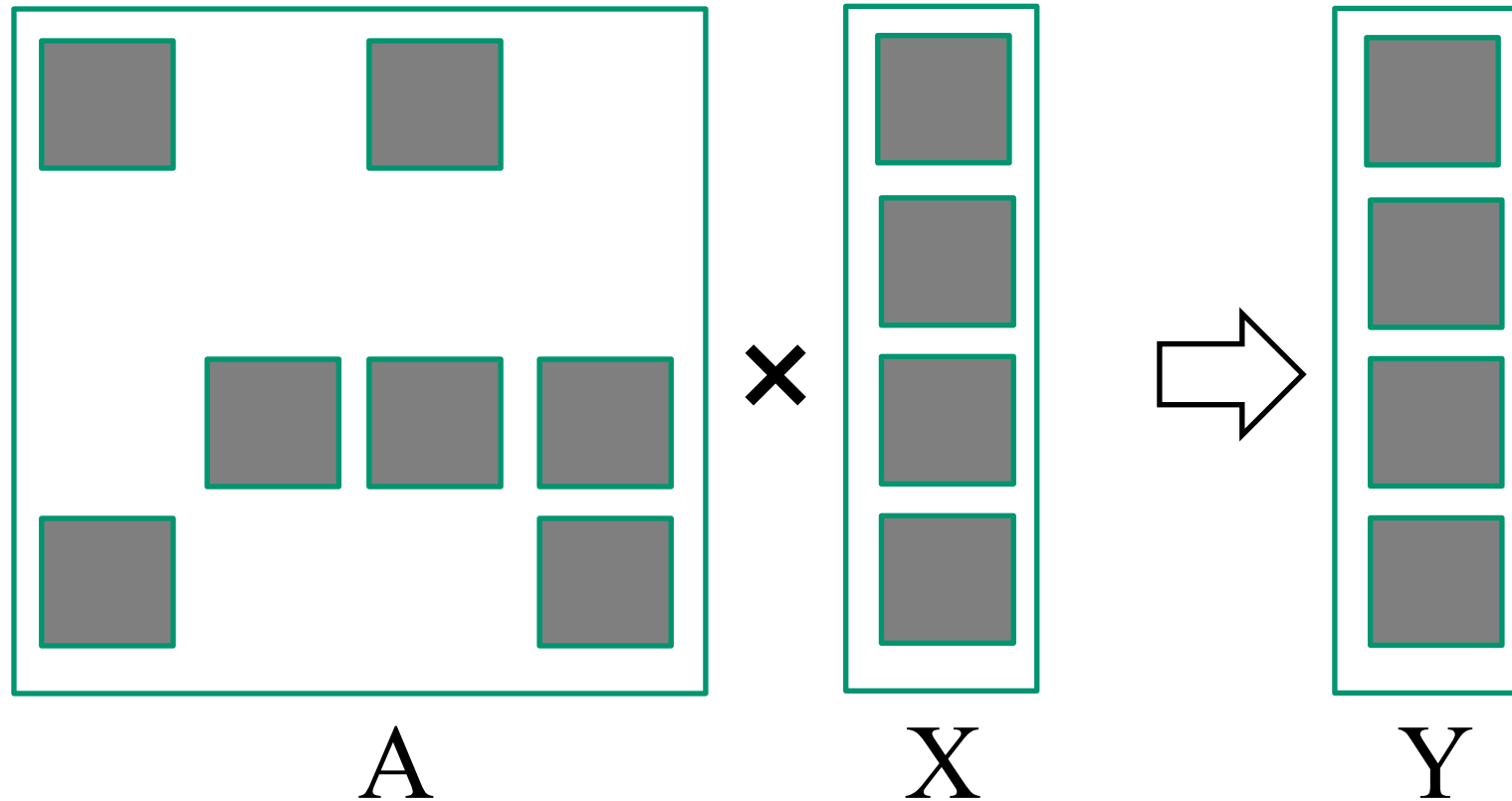
Course Reminders

- We are grading MP5.2 now
- PM1 grading is almost done
- PM2 is due this Friday

Objective

- To learn to regularize irregular data with
 - Limiting variations with clamping
 - Sorting
 - Transposition
- To learn to write a high-performance SpMV kernel based on JDS transposed format

Sparse Matrix-Vector Multiplication (SpMV)



Compressed Sparse Row (CSR) Format

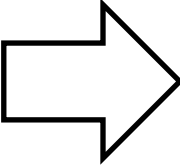
CSR Representation

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

Dense representation

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

Regularizing SpMV with ELL(PACK) Format



3	1	*
*	*	*
2	4	1
1	1	*

CSR with Padding

3	*	2	1
1	*	4	1
*	*	1	*

Transposed

- 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

Coordinate (COO) format

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

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

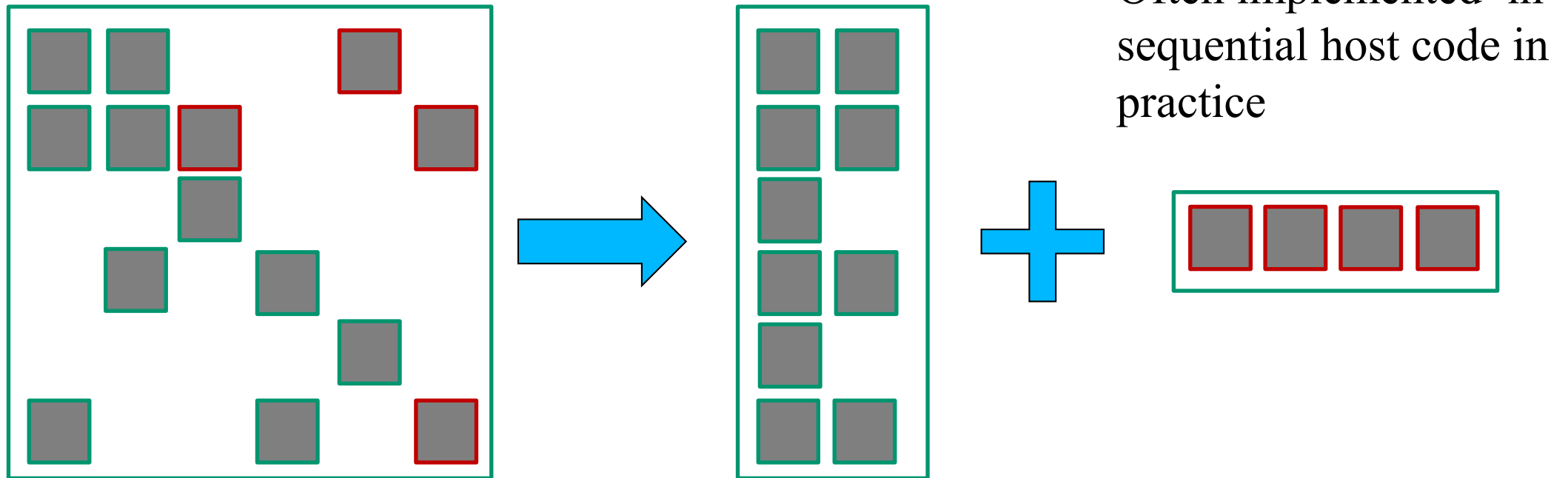
COO Allows Reordering of Elements

		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 }

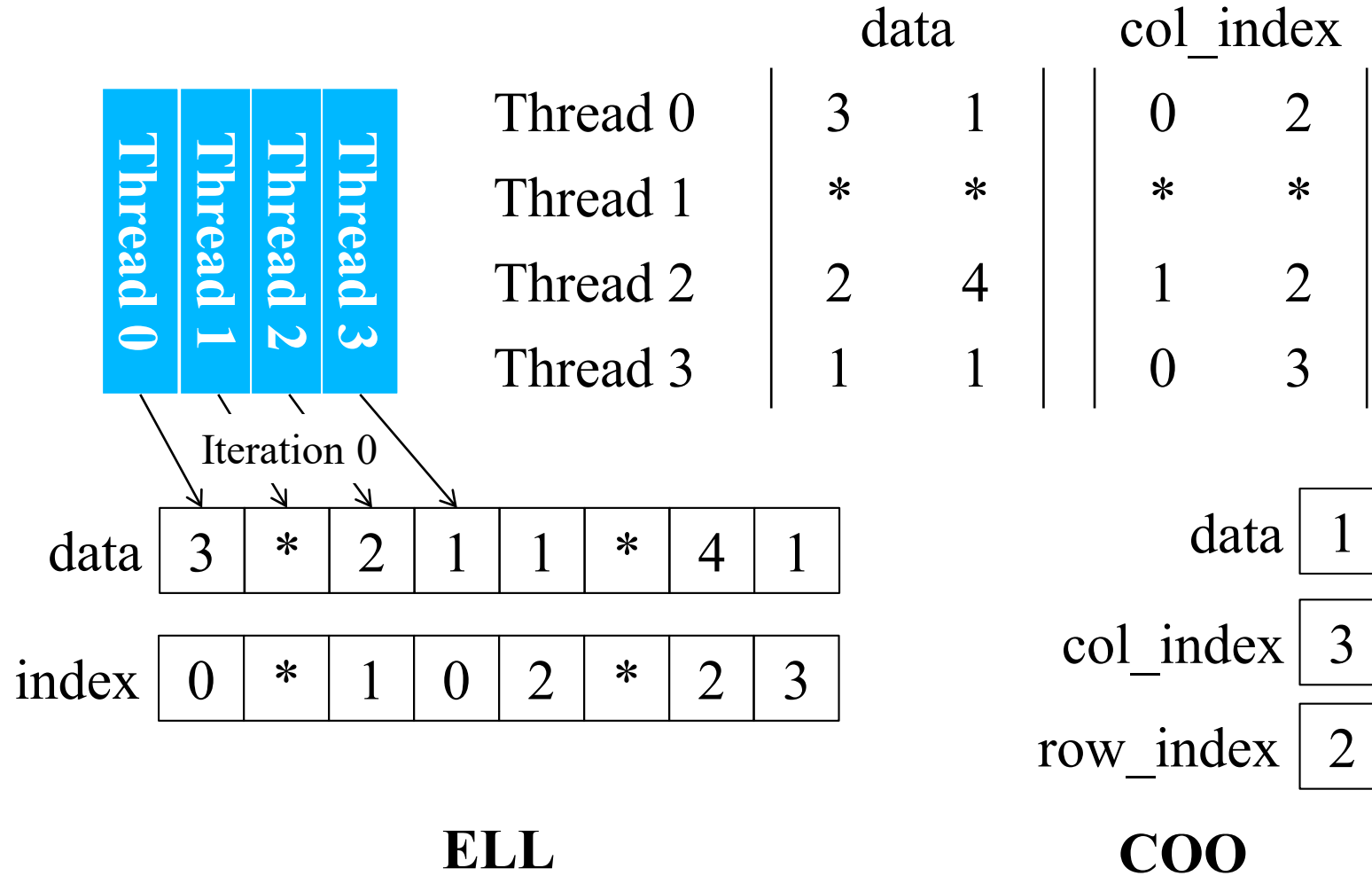
Nonzero values	data[7]	{ 1 1, 2, 4, 3, 1 1 }
Column indices	col_index[7]	{ 0 2, 1, 2, 0, 3, 3 }
Row indices	row_index[7]	{ 3 0, 2, 2, 0, 2, 3 }

Hybrid Format (ELL + COO)

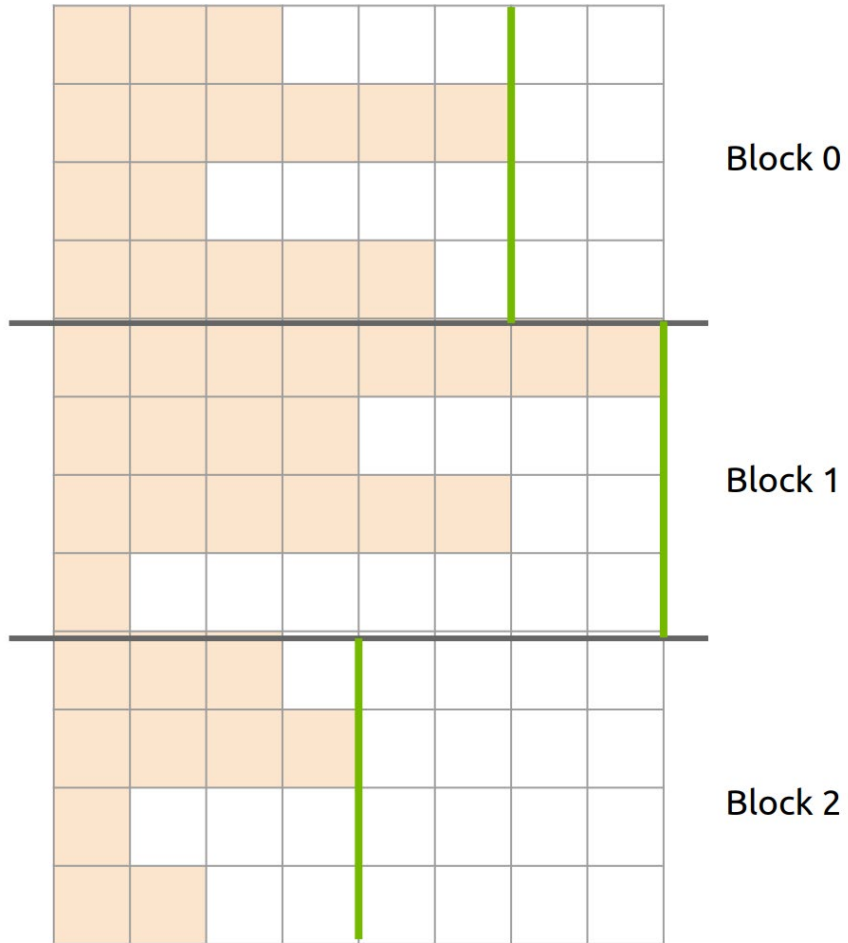
- ELL handles *typical* entries
- COO handles *exceptional* entries
 - Implemented with segmented reduction



Reduced Padding with Hybrid Format

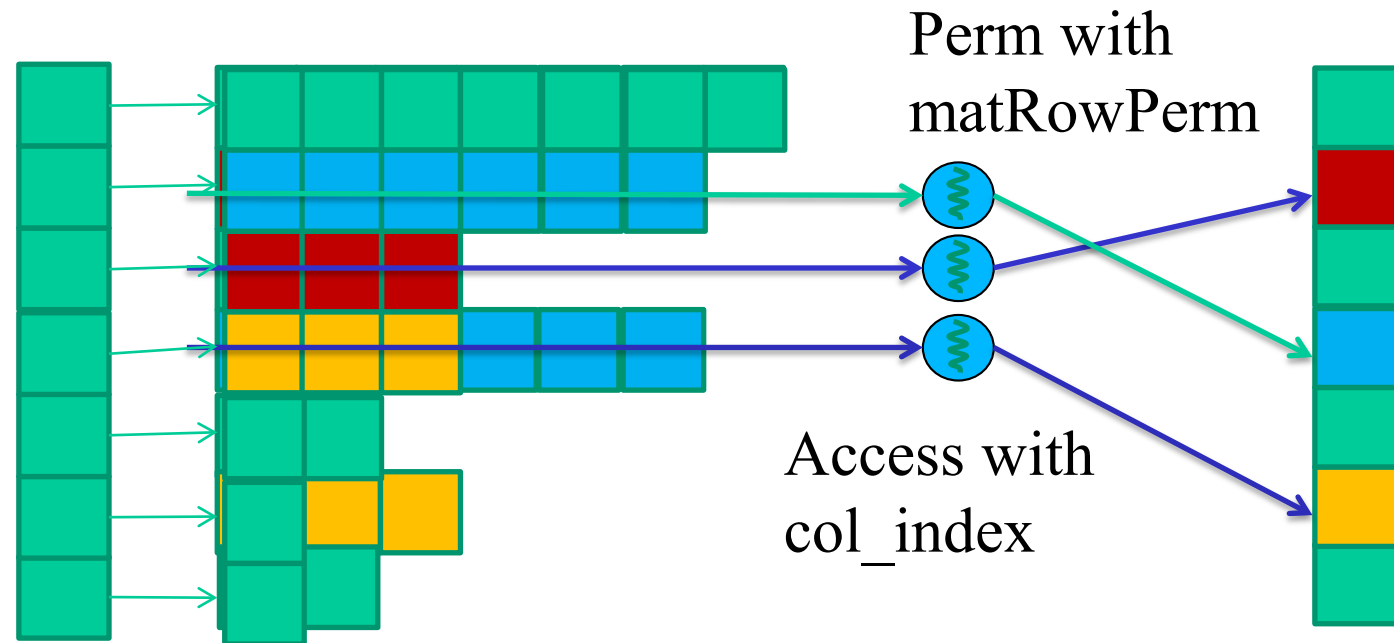


CSR Run-time



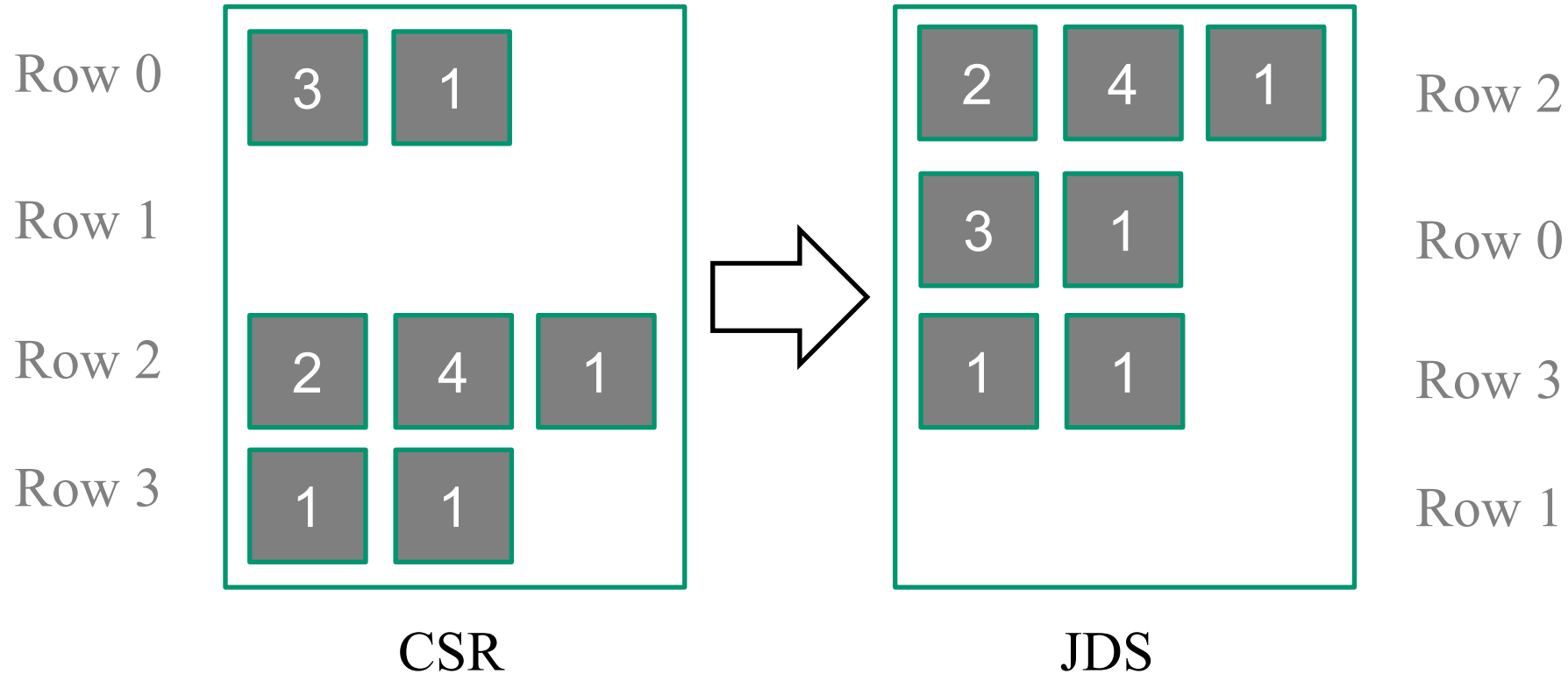
Block performance is determined by longest row

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)

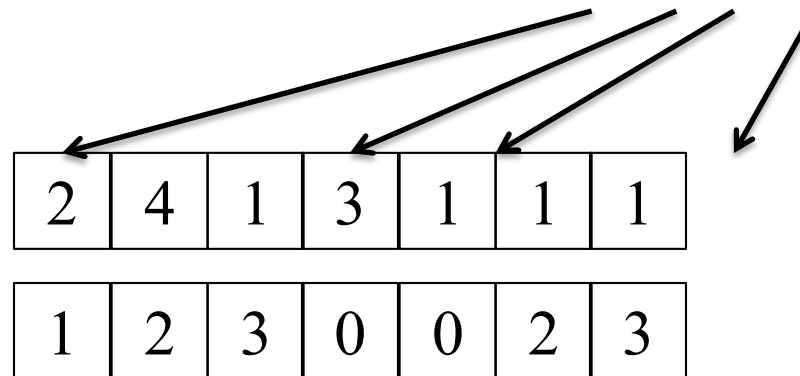


CSR to JDS Conversion

		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 }



A Parallel SpMV/JDS Kernel

```

1. __global__ void SpMV_JDS(int num_rows, float *data, int *col_index,
                           int *jds_row_ptr, int *jds_row_perm, 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 = jds_row_ptr[row];
6.         int row_end = jds_row_ptr[row+1];
7.         for (int elem = row_start; elem < row_end; elem++) {
8.             dot += data[elem] * x[col_index[elem]];
9.         }
10.        y[jds_row_perm[row]] = dot;
11.    }
12.}

```

	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

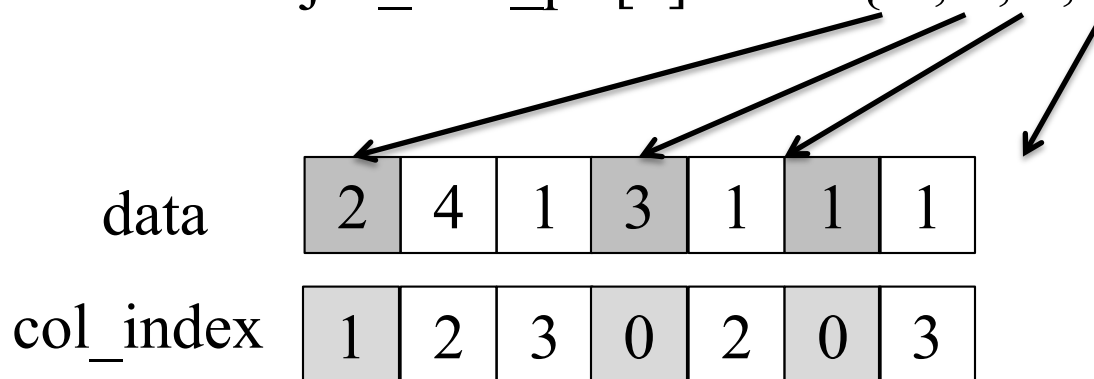
- Adjacent threads still access non-adjacent memory locations

Nonzero values data[7] { 2, 4, 1, 3, 1, 1, 1 }

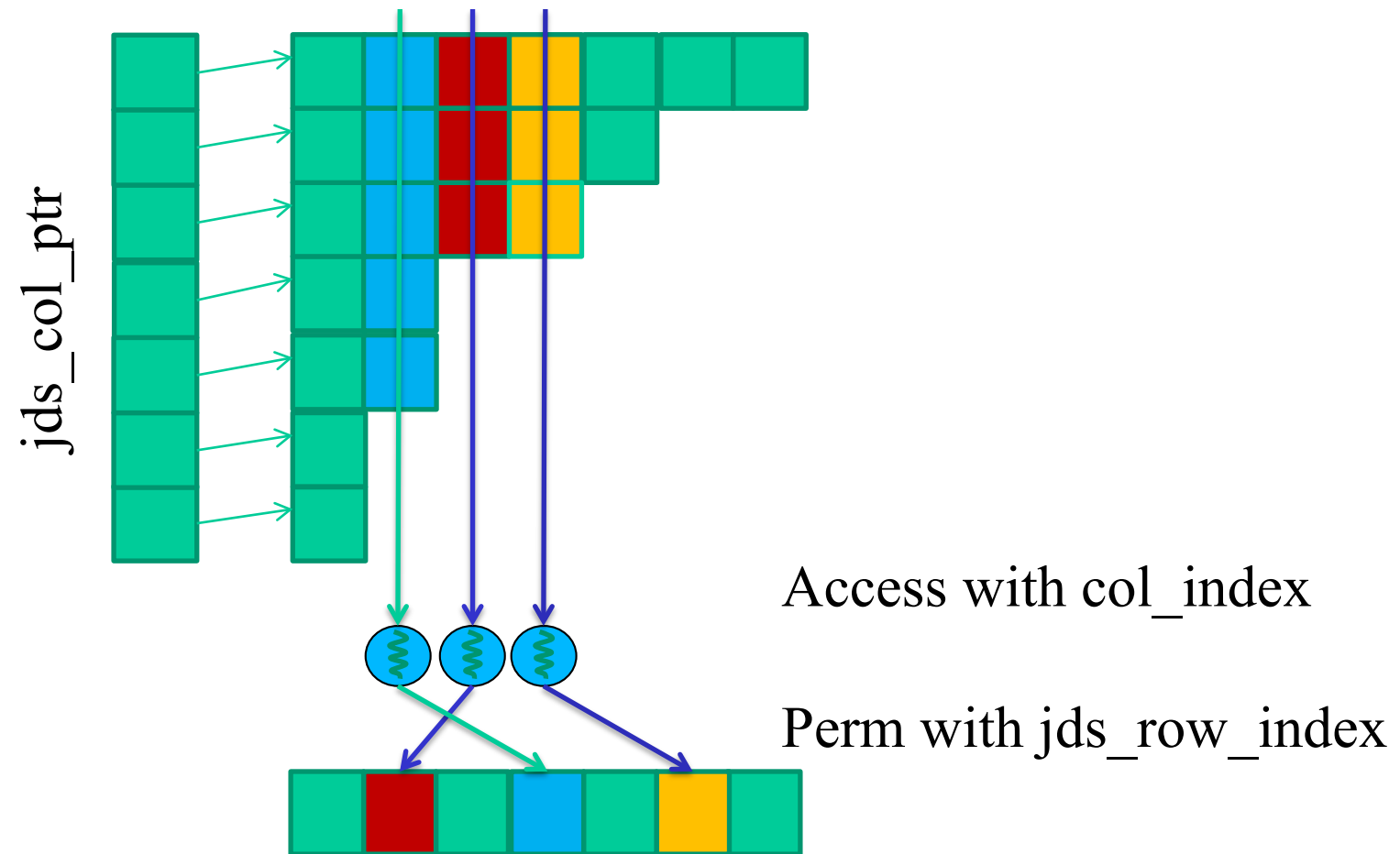
Column indices col_index[7] { 1, 2, 3, 0, 2, 0, 3 }

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

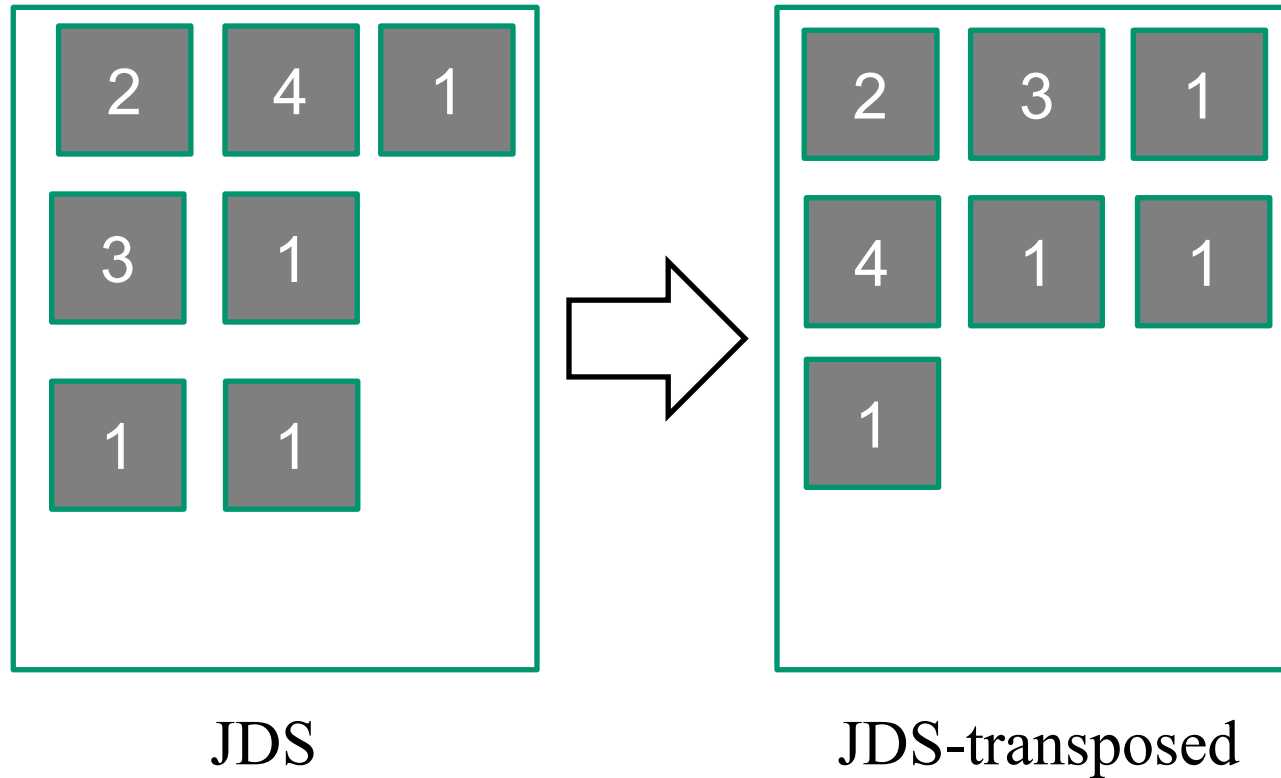
JDS Row Ptrs jds_row_ptr[5] { 0, 3, 5, 7, 7 }



JDS with Transposition



Transposition for Memory Coalescing

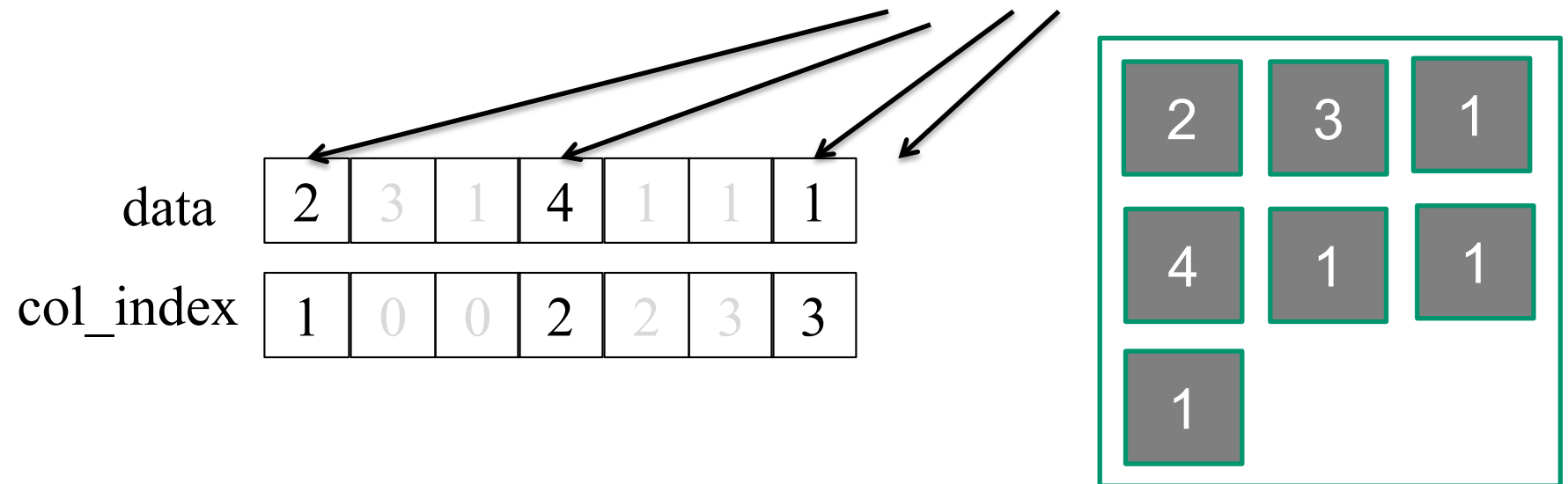


JDS Format with Transposed Layout

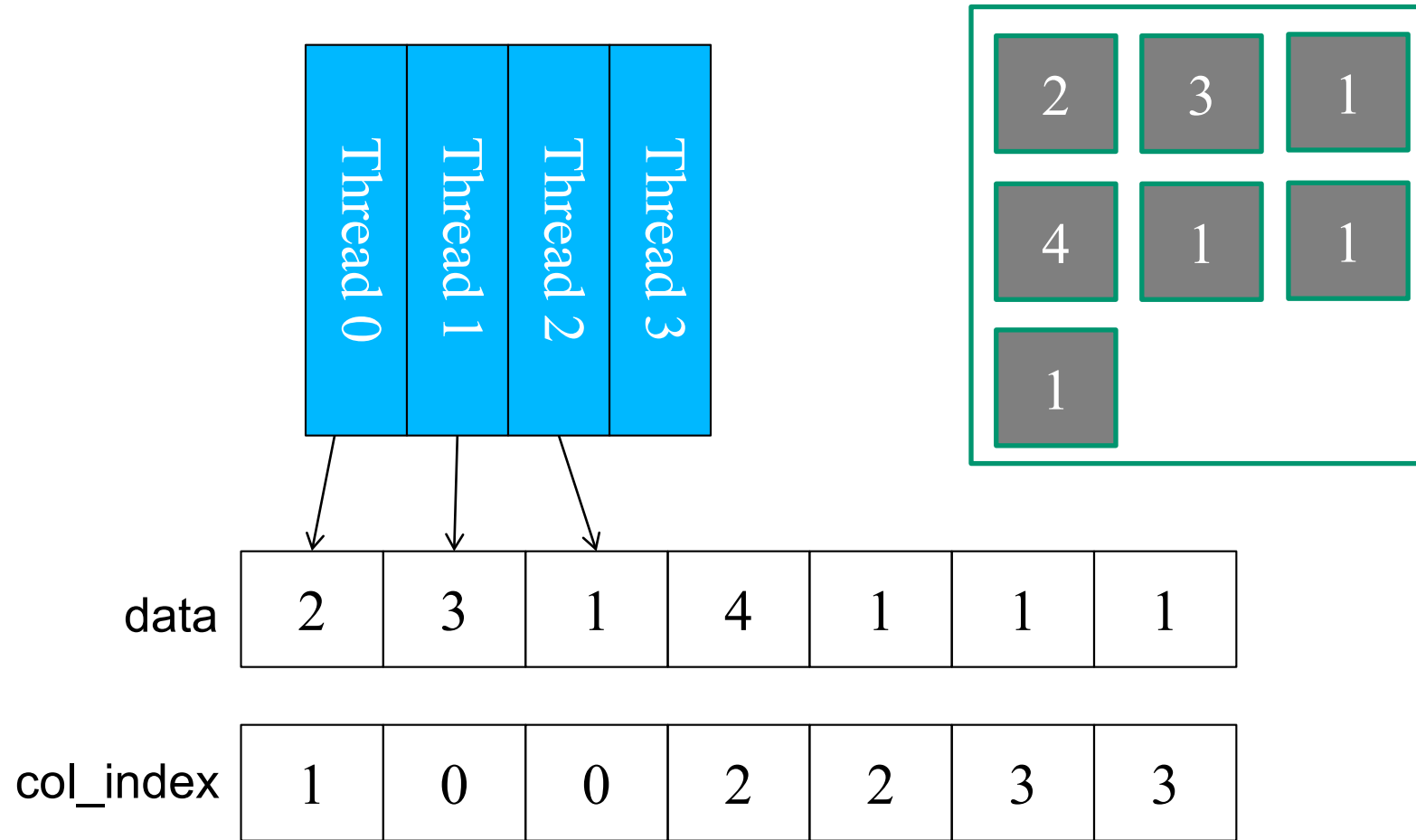
Row 0	3	0	1	0	Thread 0
Row 1	0	0	0	0	Thread 1
Row 2	0	2	4	1	Thread 2
Row 3	1	0	0	1	Thread 3

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

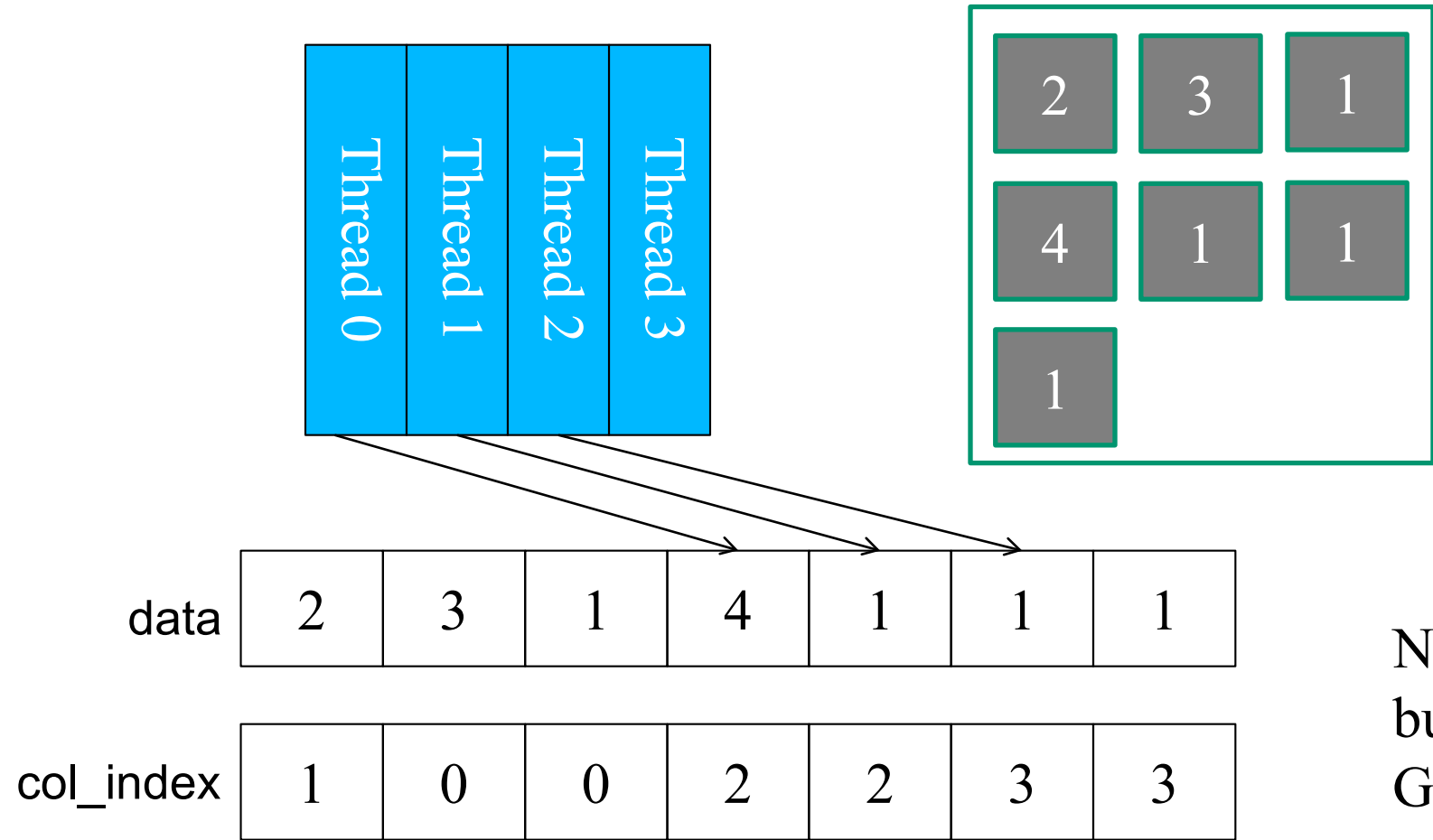
JDS column pointers `jds_t_col_ptr[4]` { 0, 3, 6, 7 }



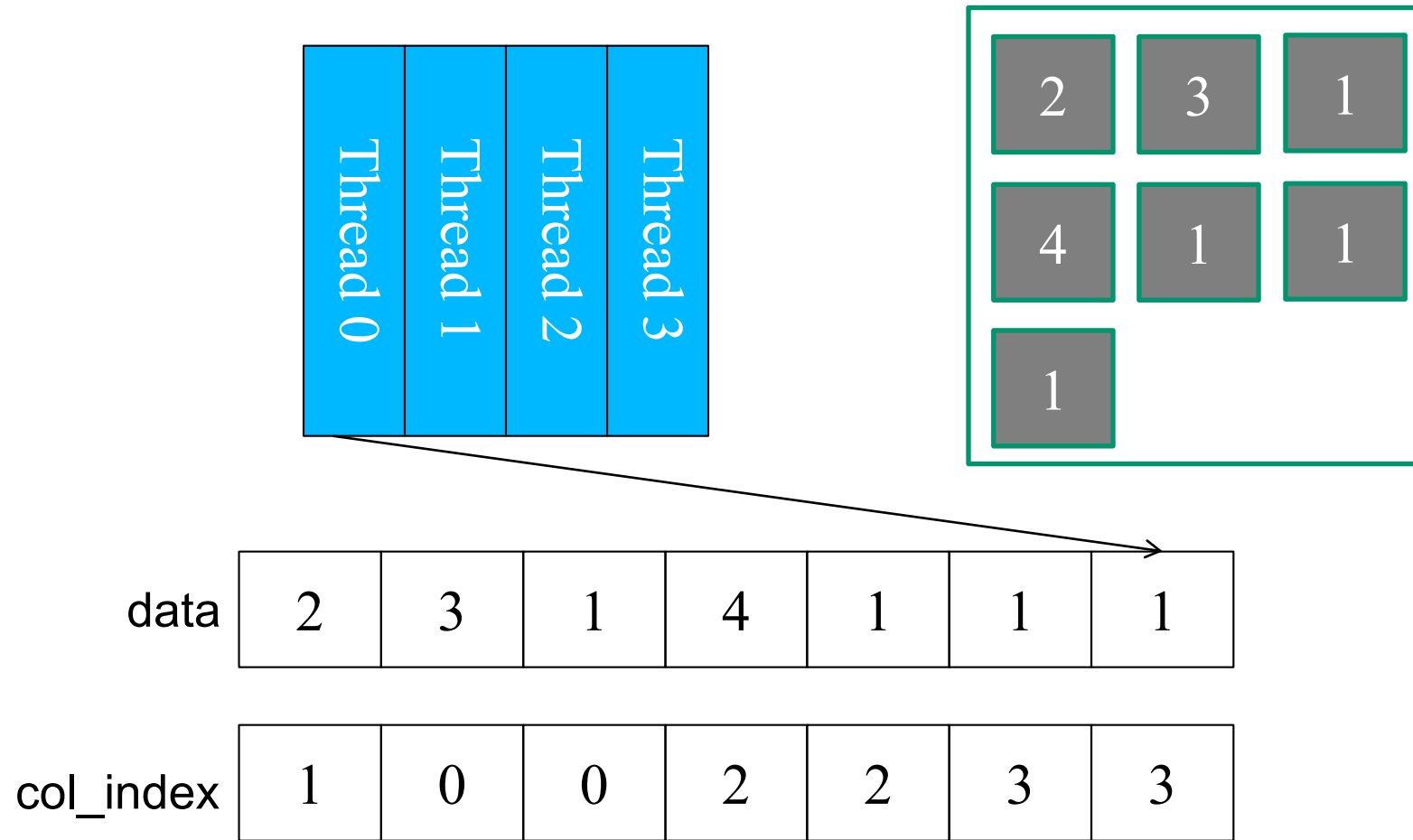
JDS with Transposition: Memory Coalescing



JDS with Transposition: Memory Coalescing



JDS with Transposition: Memory Coalescing



A Parallel SpMV/JDS_T Kernel

```

1. __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) {
2.     int row = blockIdx.x * blockDim.x + threadIdx.x;
3.     if (row < num_rows) {
4.         float dot = 0;
5.         unsigned int sec = 0;
6.         while (jds_t_col_ptr[sec+1] - jds_t_col_ptr[sec] > row) {
7.             dot += data[jds_t_col_ptr[sec]+row] *
                    x[col_index[jds_t_col_ptr[sec]+row]];
8.             sec++;
9.         }
10.    y[jds_row_perm[row]] = dot;
11. }

```

Nonzero values data[7]

Column indices col_index[7]

JDS_T Column Pointers jds_t_col_ptr[5]

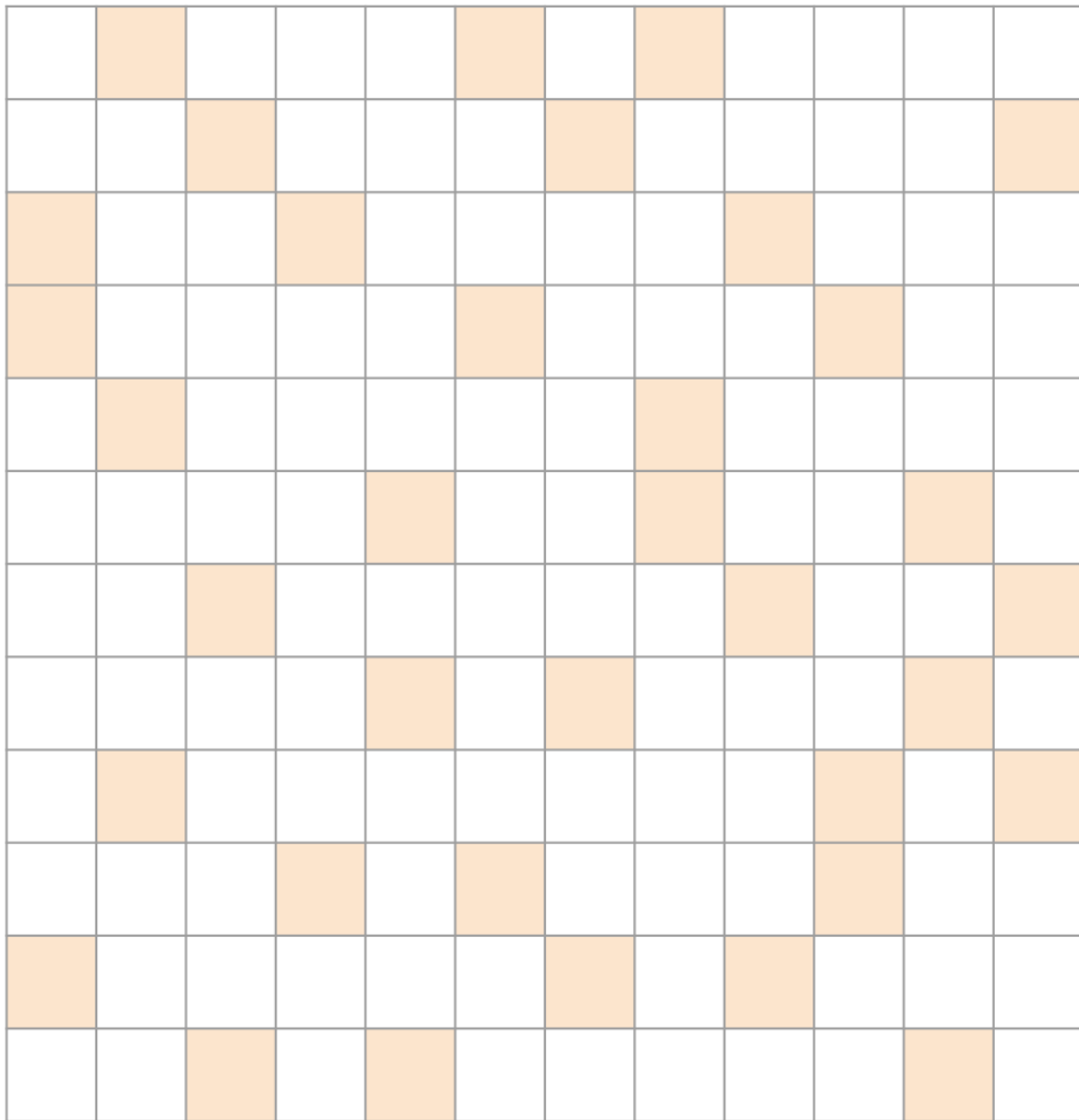
JDS Row Indices jds_row_perm[4]

Sec 0	Sec 1	Sec 2
{ 2, 3, 1,	4, 1, 1	1 }
{ 1, 0, 3,	2, 2, 3	3 }
{ 0,	3,	6,
		7,7 }
{ 2,	0,	3,
		1 }

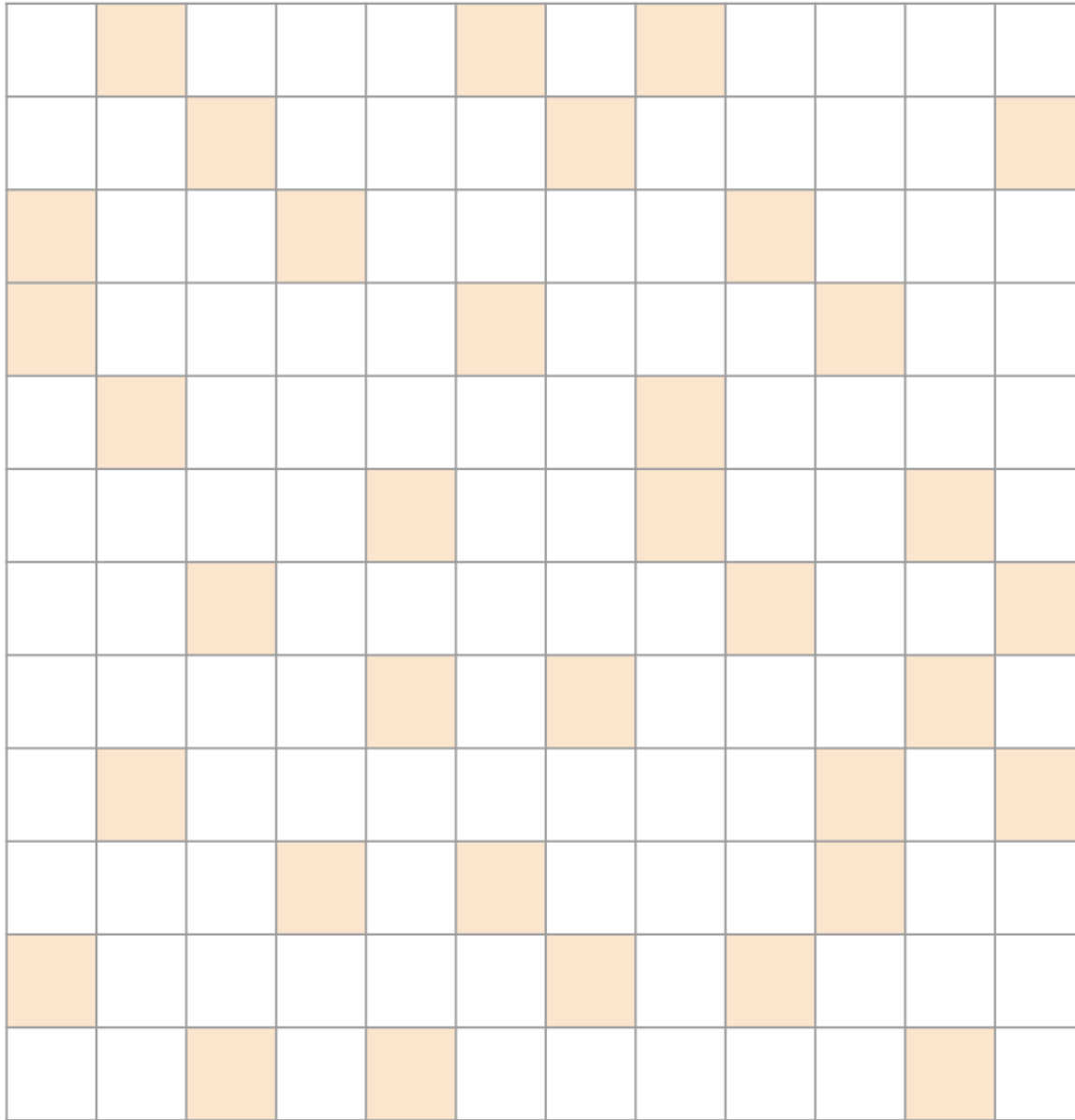
Lab 7 Variable Names

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, 3,	2, 2, 3	3 }
JDS_T Column Pointers matColStart[4]	{0,	3,	6, 7 }
JDS Row Indices matRowPerm[4]	{2,	0,	3, 1 }



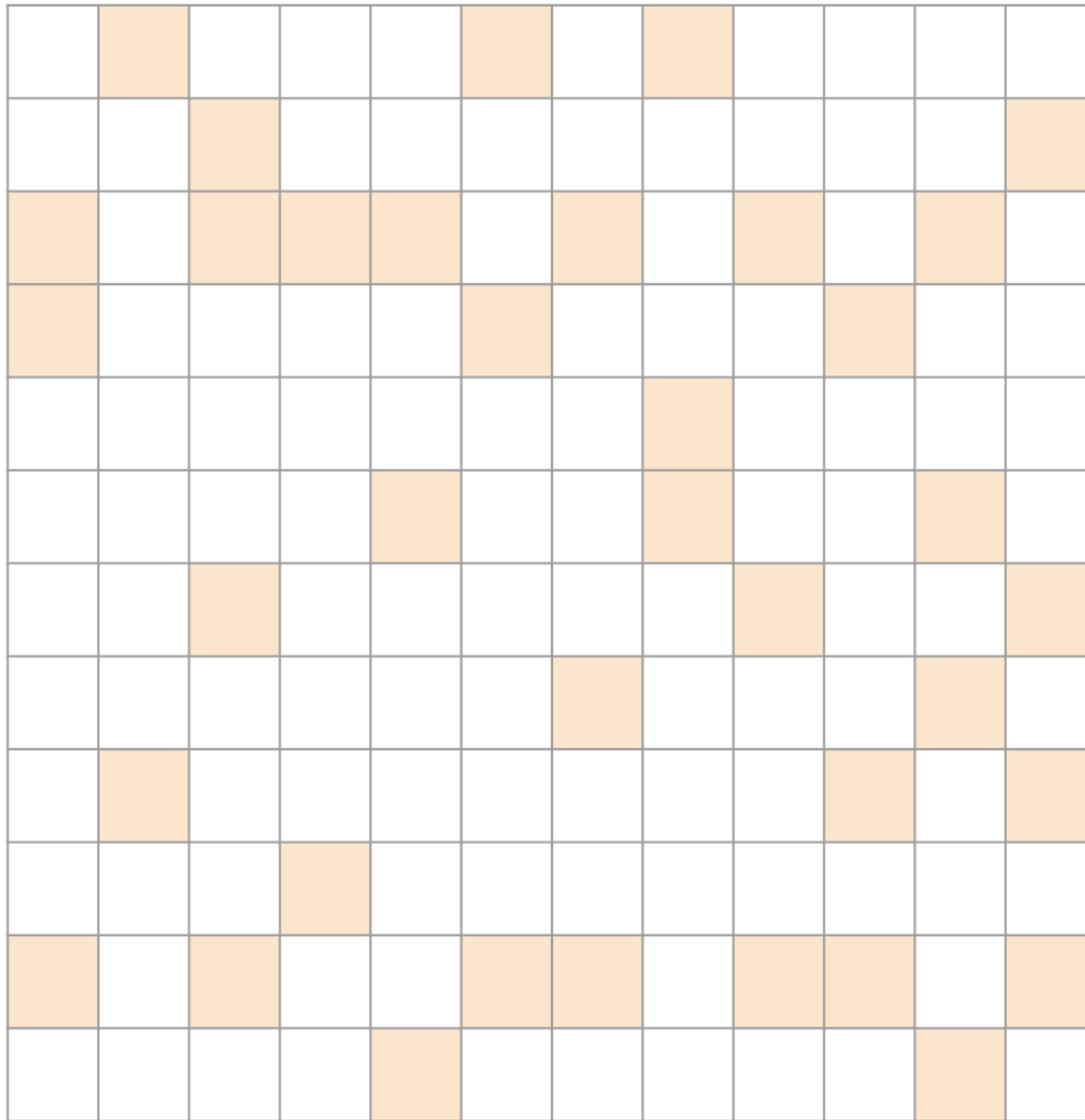
Roughly Random...



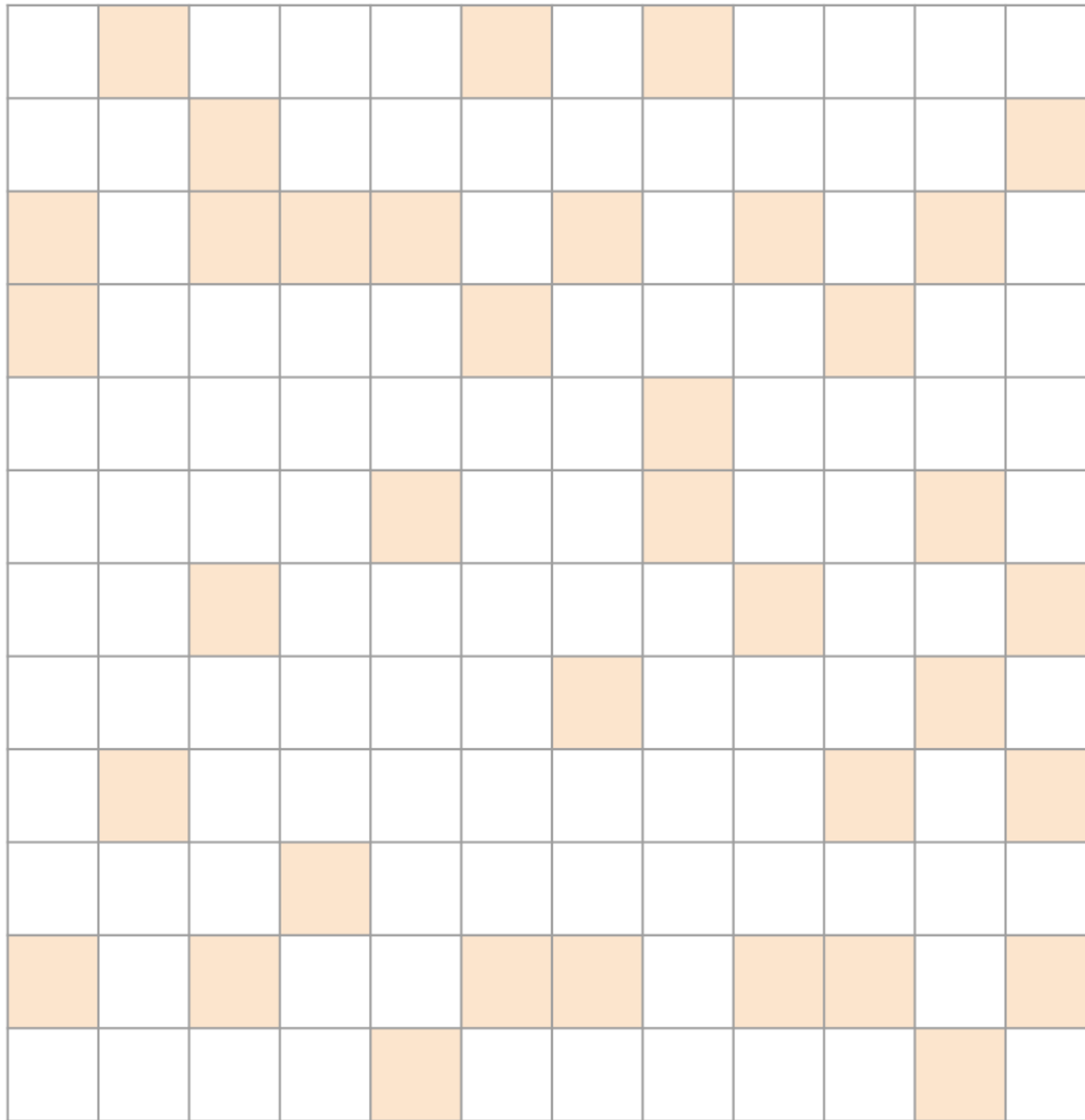
Roughly Random...

Probably best with ELL.

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



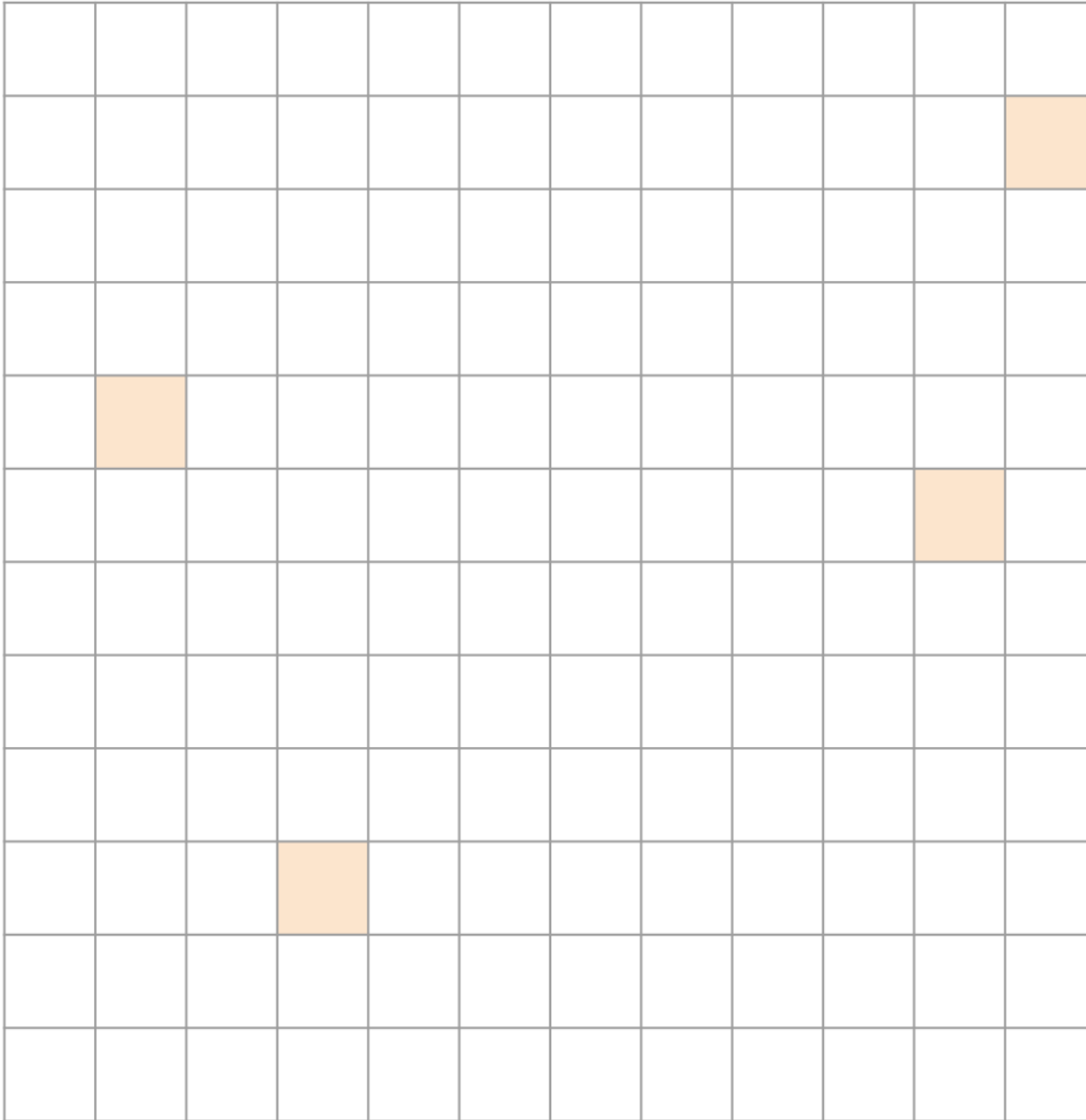
High variance in rows...



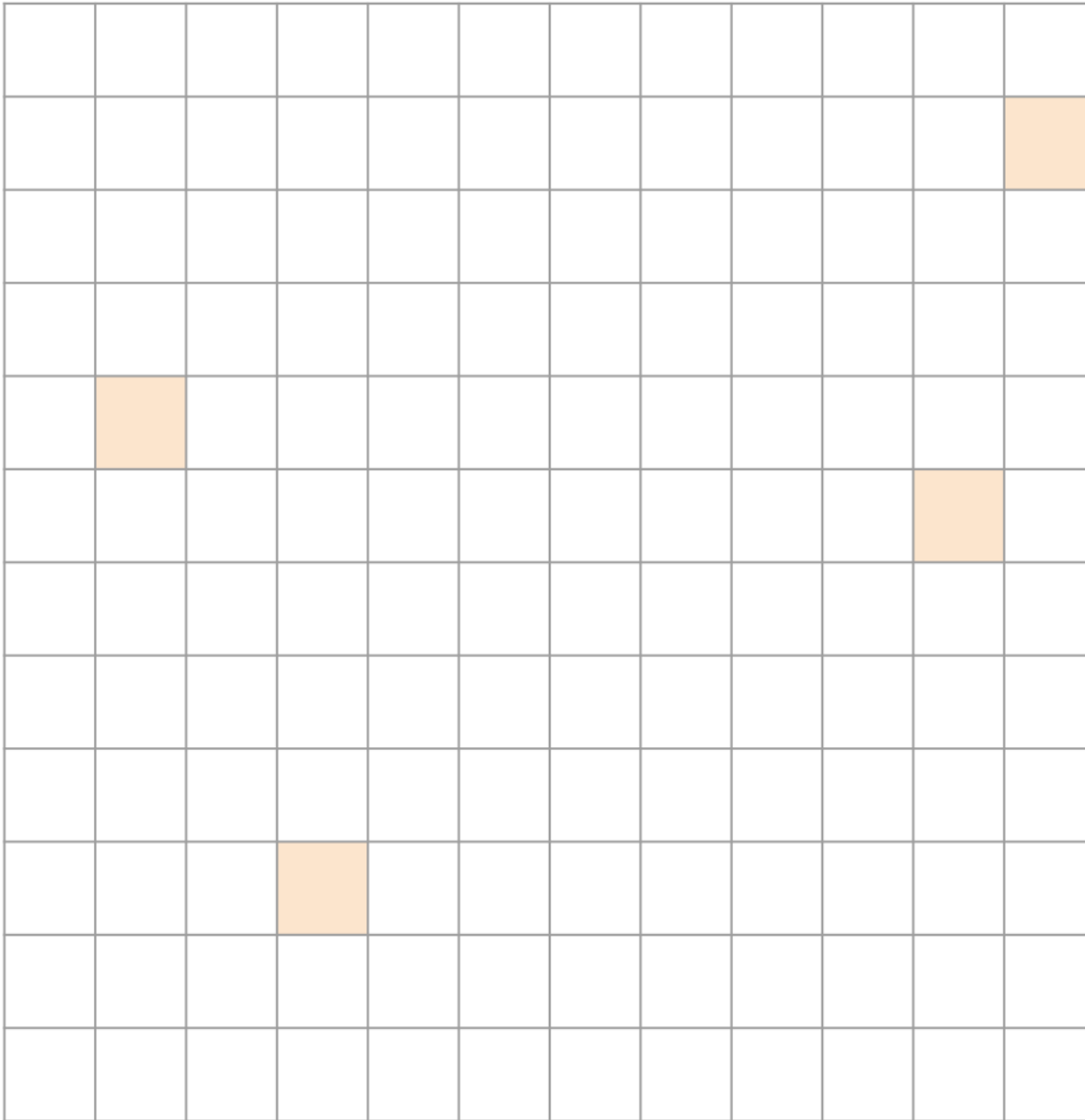
High variance in rows

Probably best with ELL/COO

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



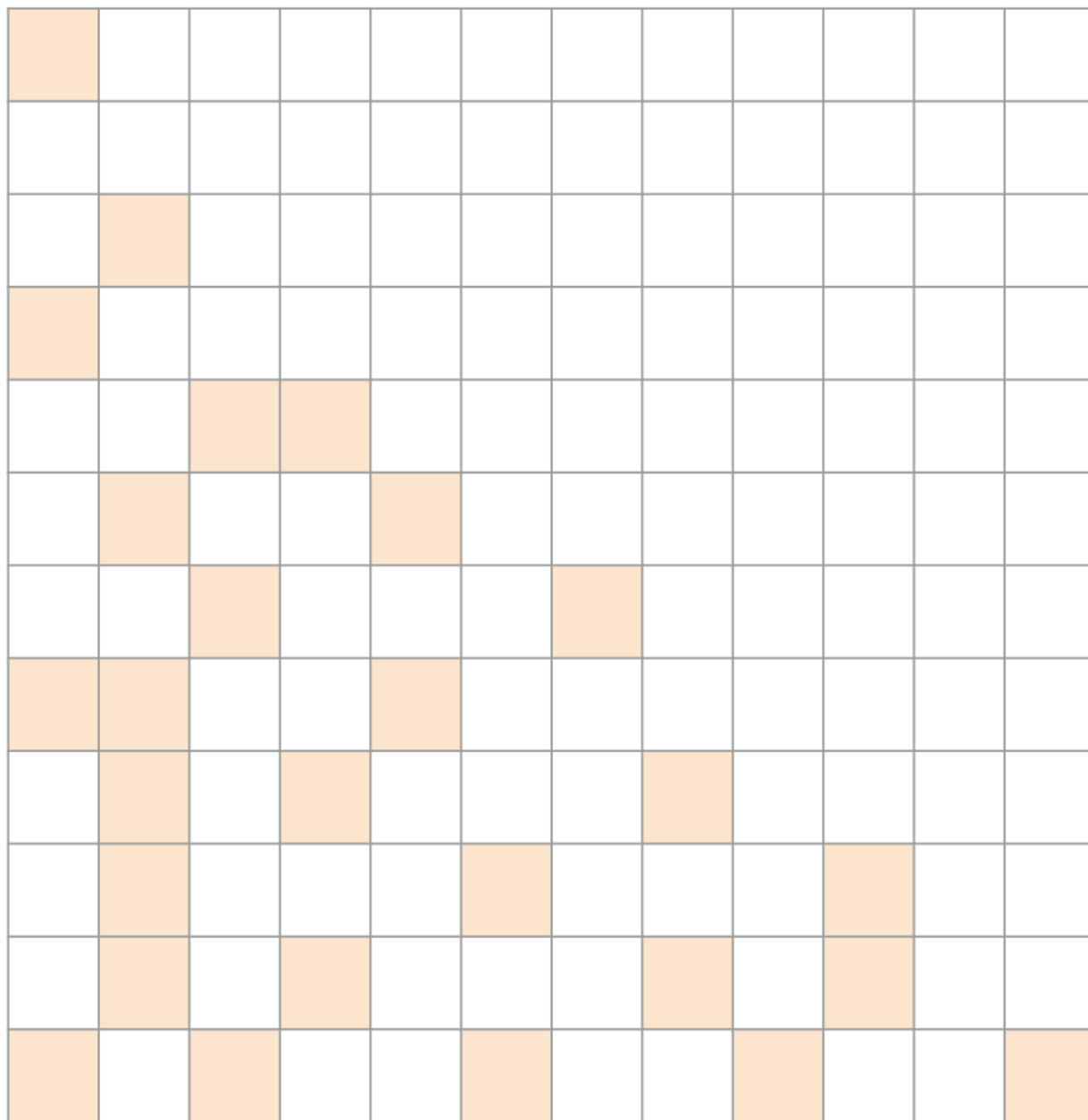
Very sparse...



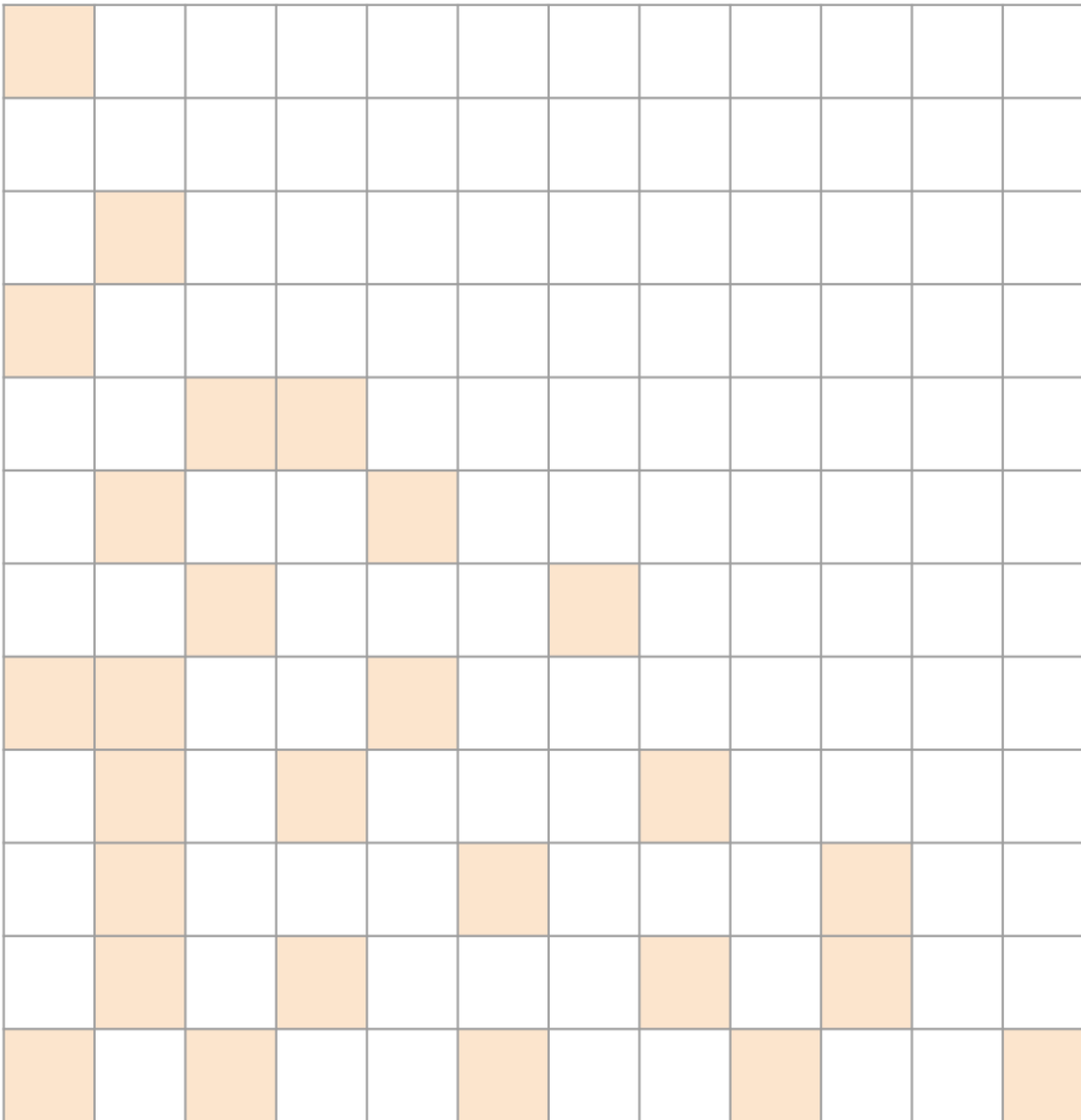
Very sparse

Probably best with COO

- Not a lot of data, compute is sparse



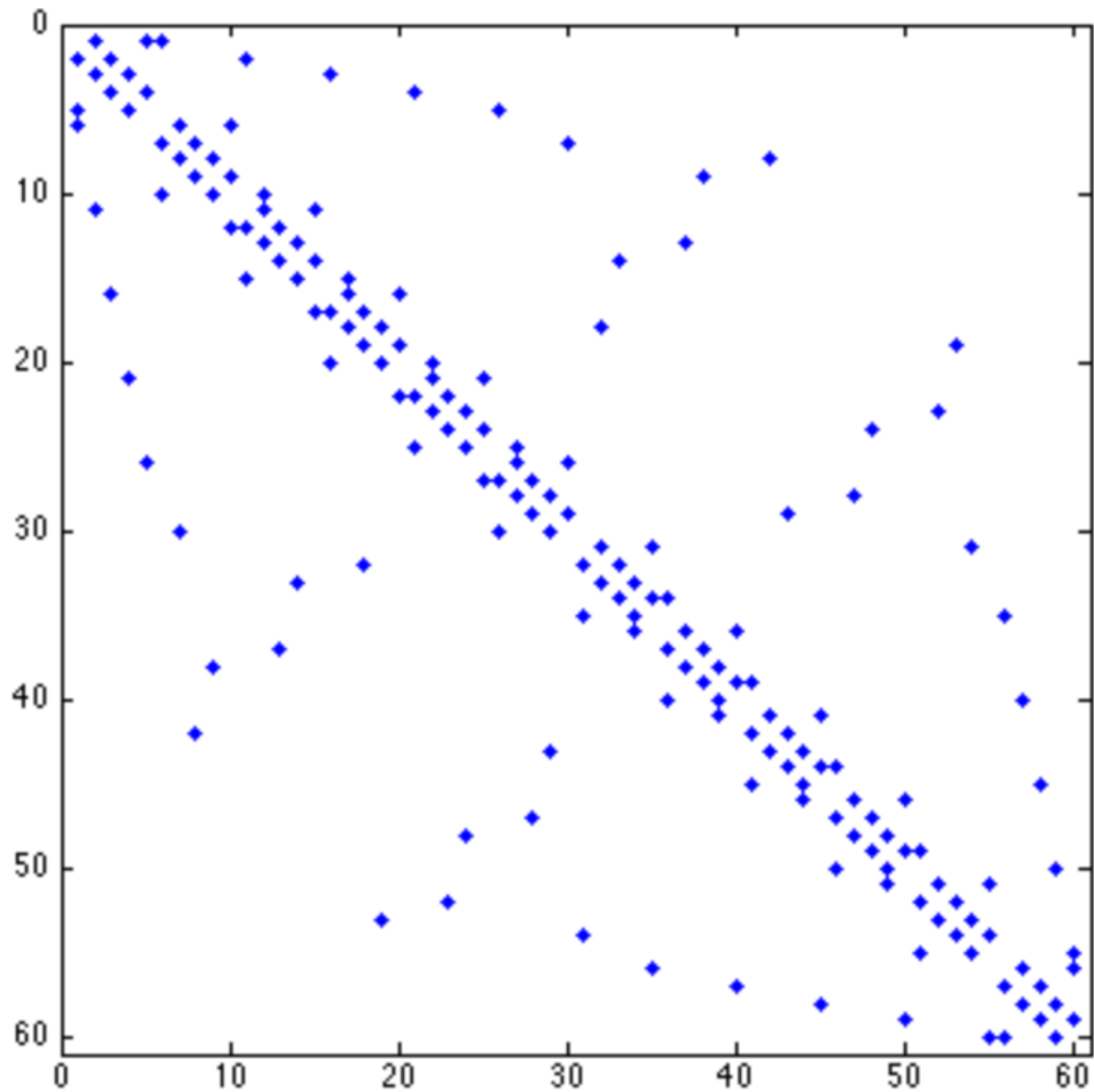
Roughly triangular...



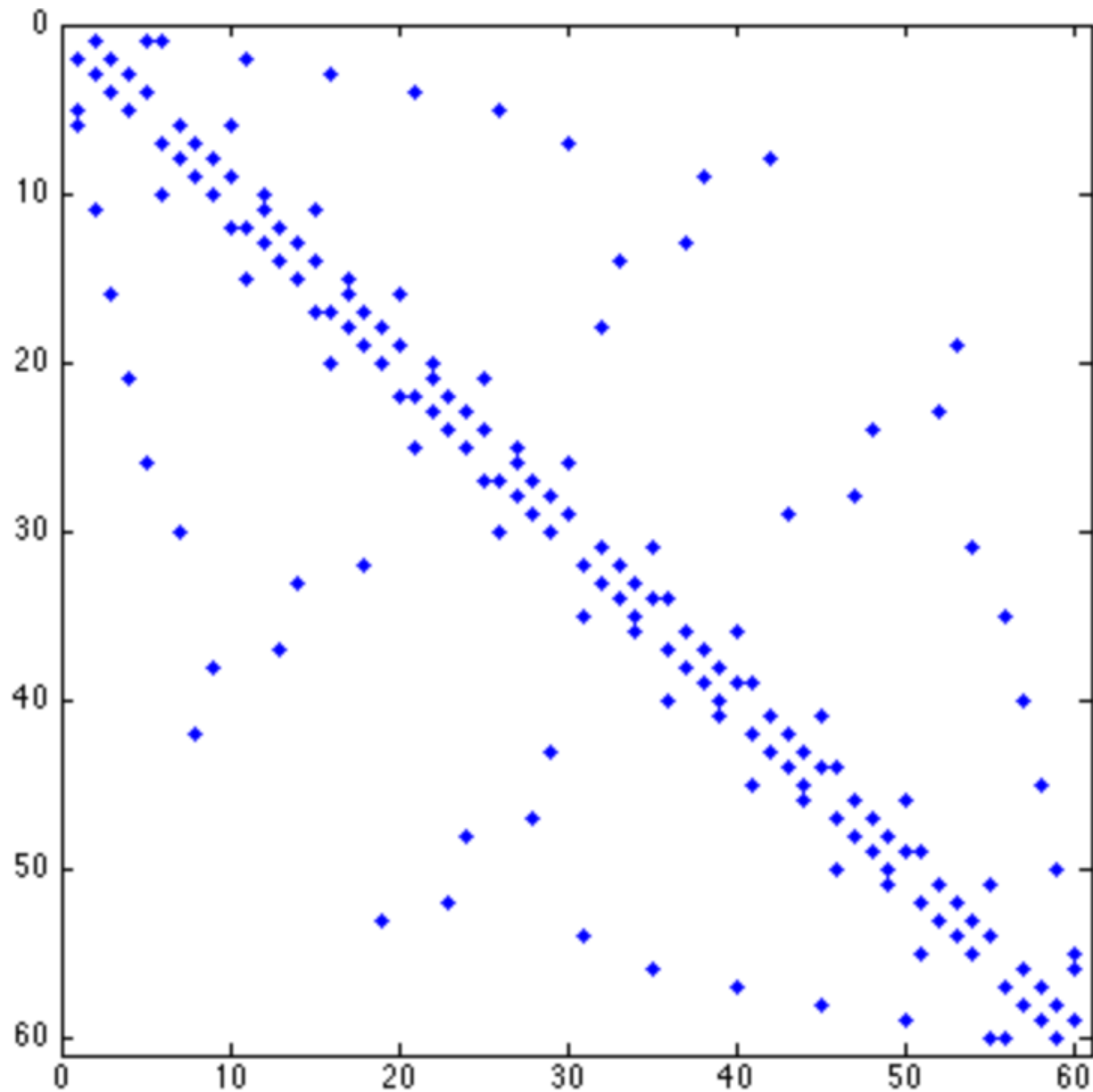
Roughly triangular...

Probably best with JDS

- Takes advantage of sparsity structure



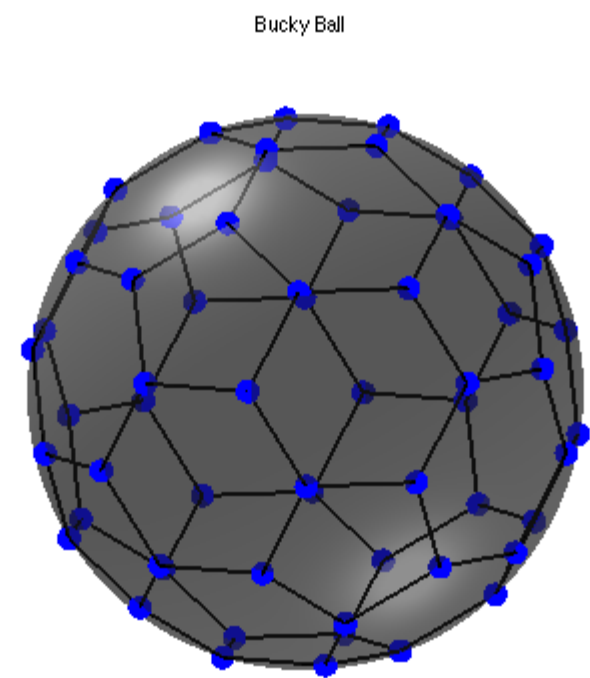
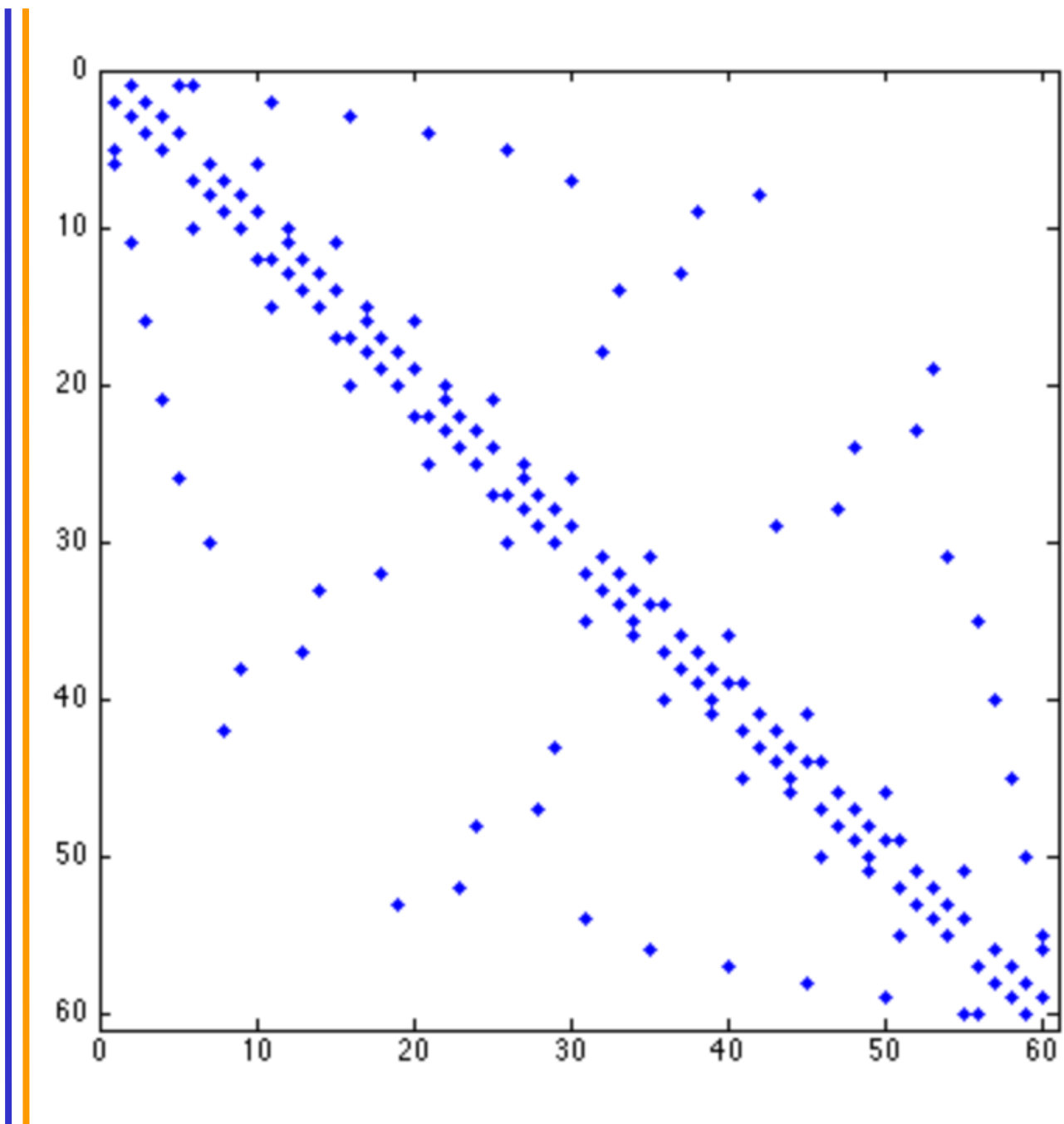
Banded Matrix...



Banded Matrix...

Probably best with ELL

- Small amount of variance in rows



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.
- 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

Two vertical lines, one blue and one orange, are positioned on the left side of the slide.

**ANY MORE QUESTIONS
READ CHAPTER 10**