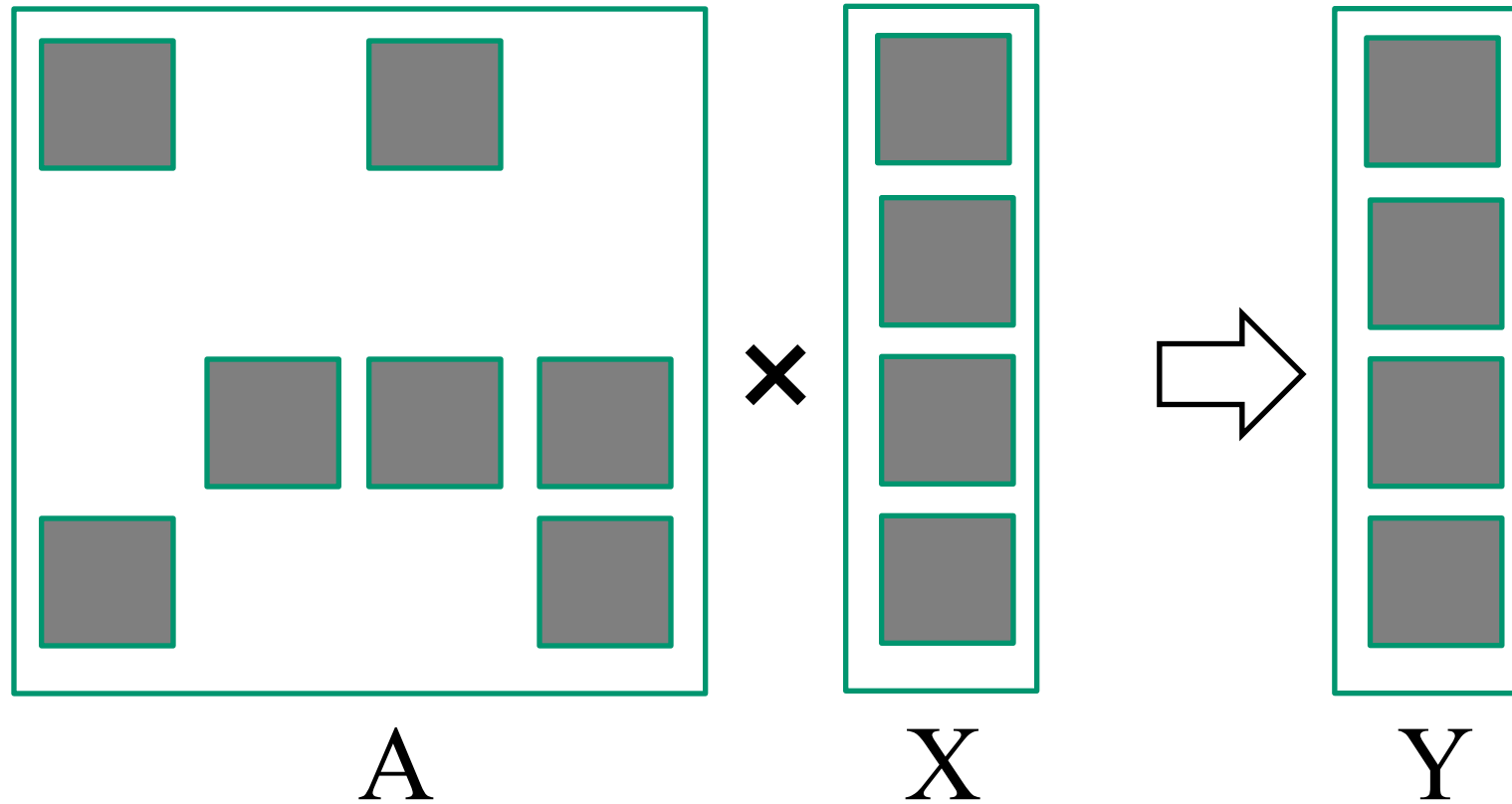# ECE408/CS483/CSE408 Fall 2022

# Applied Parallel Programming
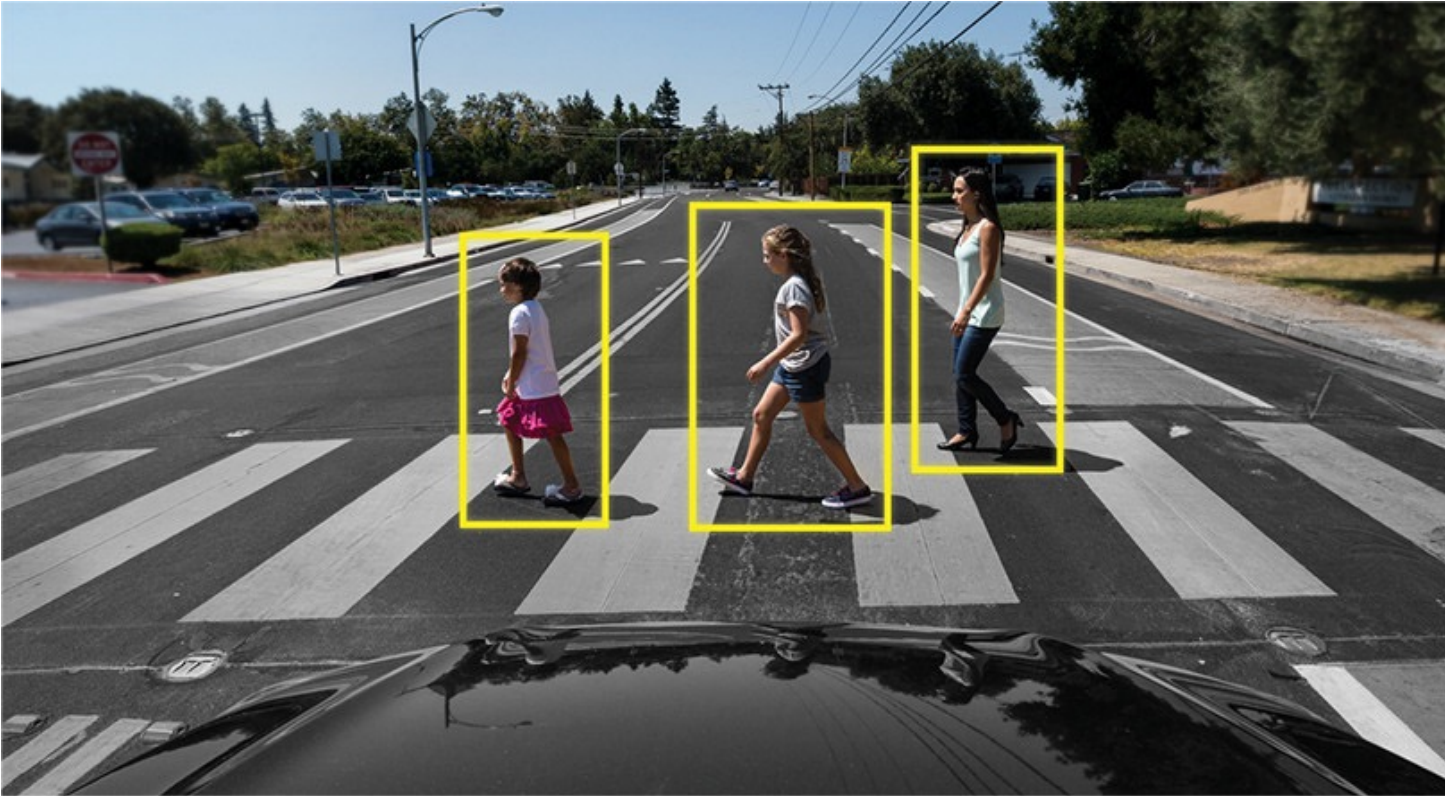
# Lecture 20
# Parallel Sparse Methods II

# Sparse Matrix-Vector Multiplication (SpMV)



A × X ⇒ Y

# Why are interesting data often sparse?



- Pedestrians occupy less than 5% of the pixels

- Let's consider the computation of a deep CNN that is detecting pedestrians.

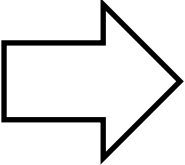- Zero weights and also lots of zero features

# 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 | 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



CSR with Padding

Transposed

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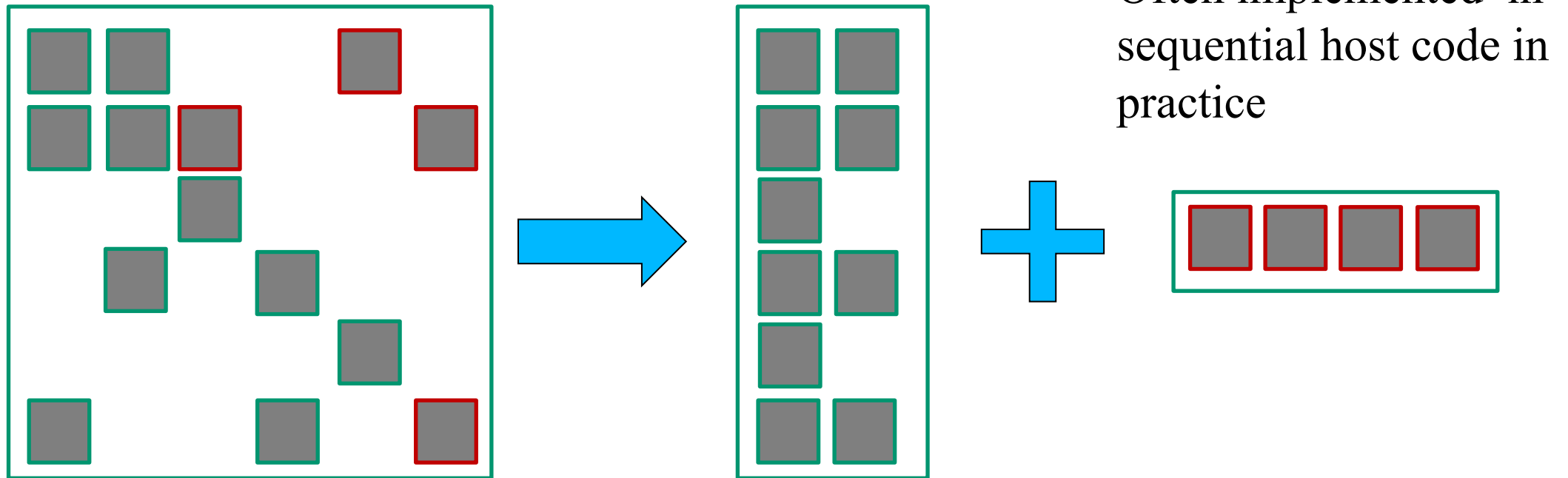- Both data and col_index padded/transposed

# Coordinate (COO) format

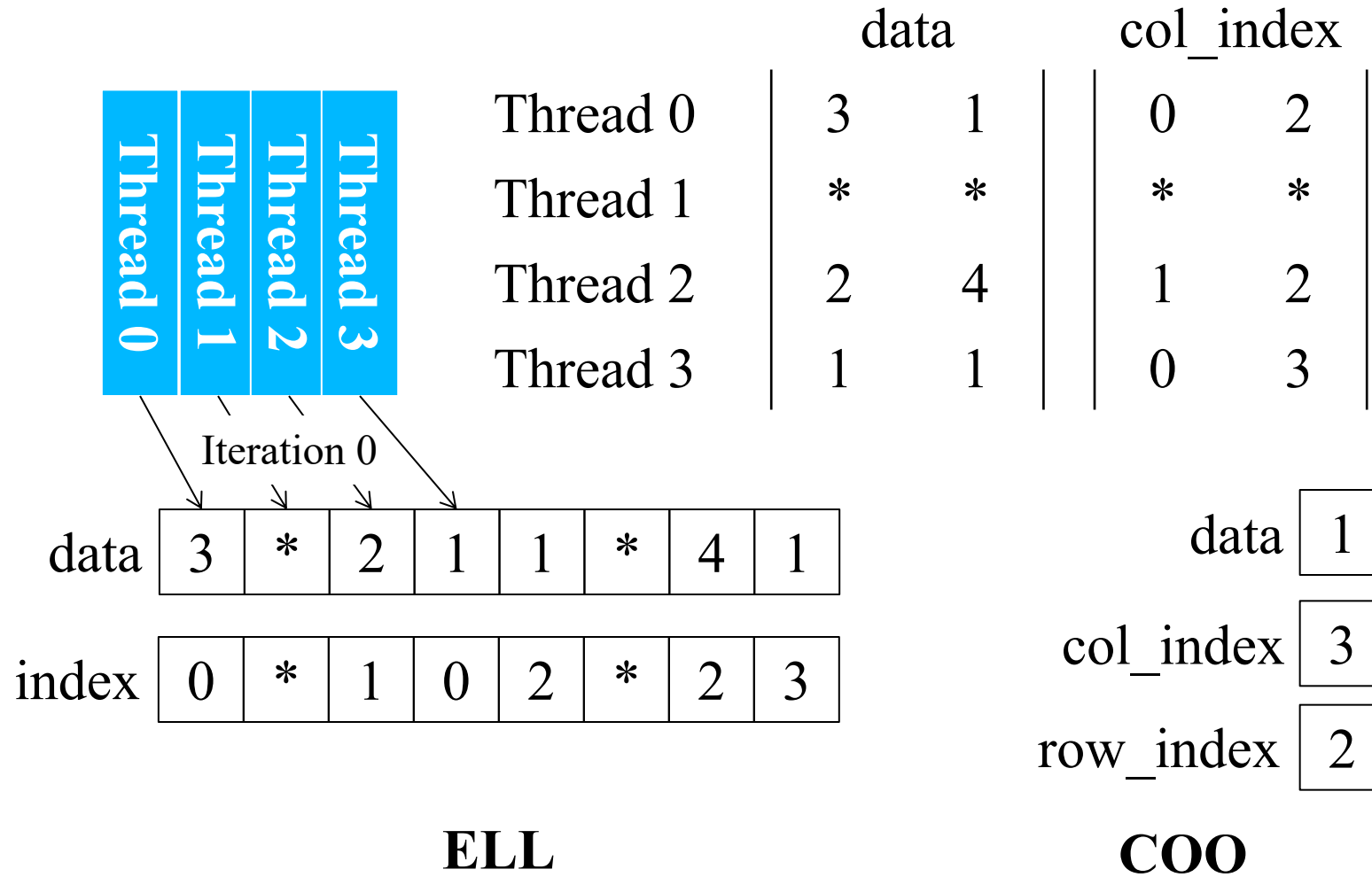- Explicitly list the column & row indices for every non-zero element

|  |  | Row 0 |  | Row 2 |  |  | Row 3 |  |  |
|---|---|---|---|---|---|---|---|---|---|
| Nonzero values | `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 | } |

# 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 |

Iteration 0

| data  | 3 | * | 2 | 1 | 1 | * | 4 | 1 |
|-------|---|---|---|---|---|---|---|---|
| index | 0 | * | 1 | 0 | 2 | * | 2 | 3 |

**ELL**

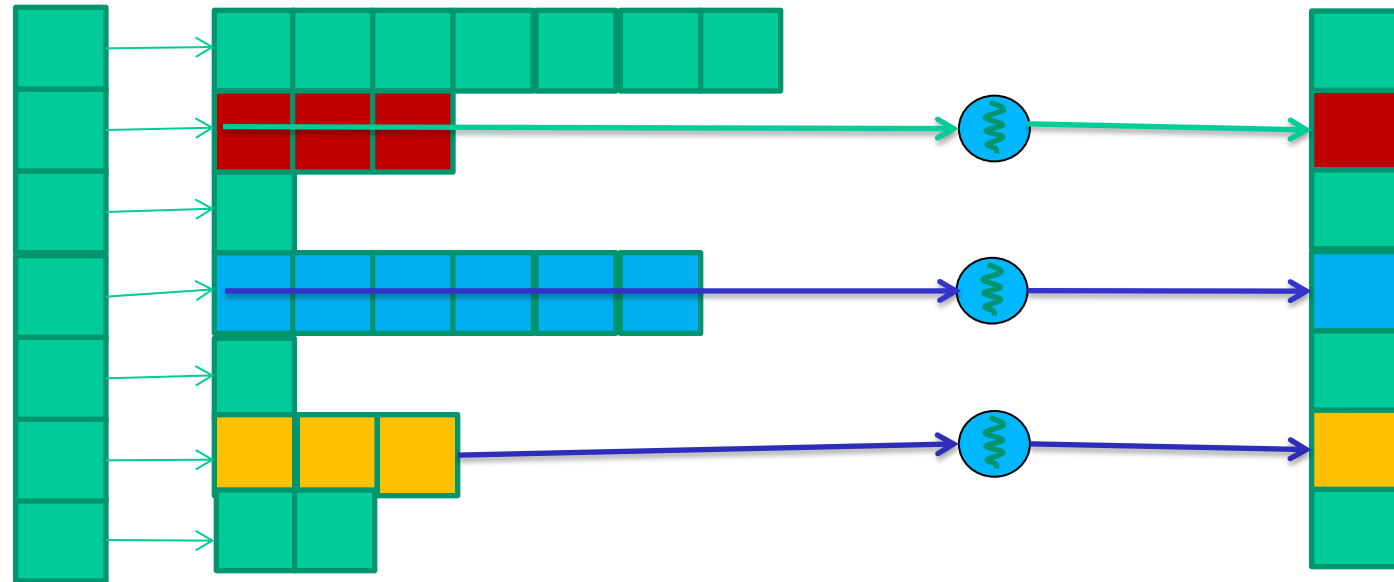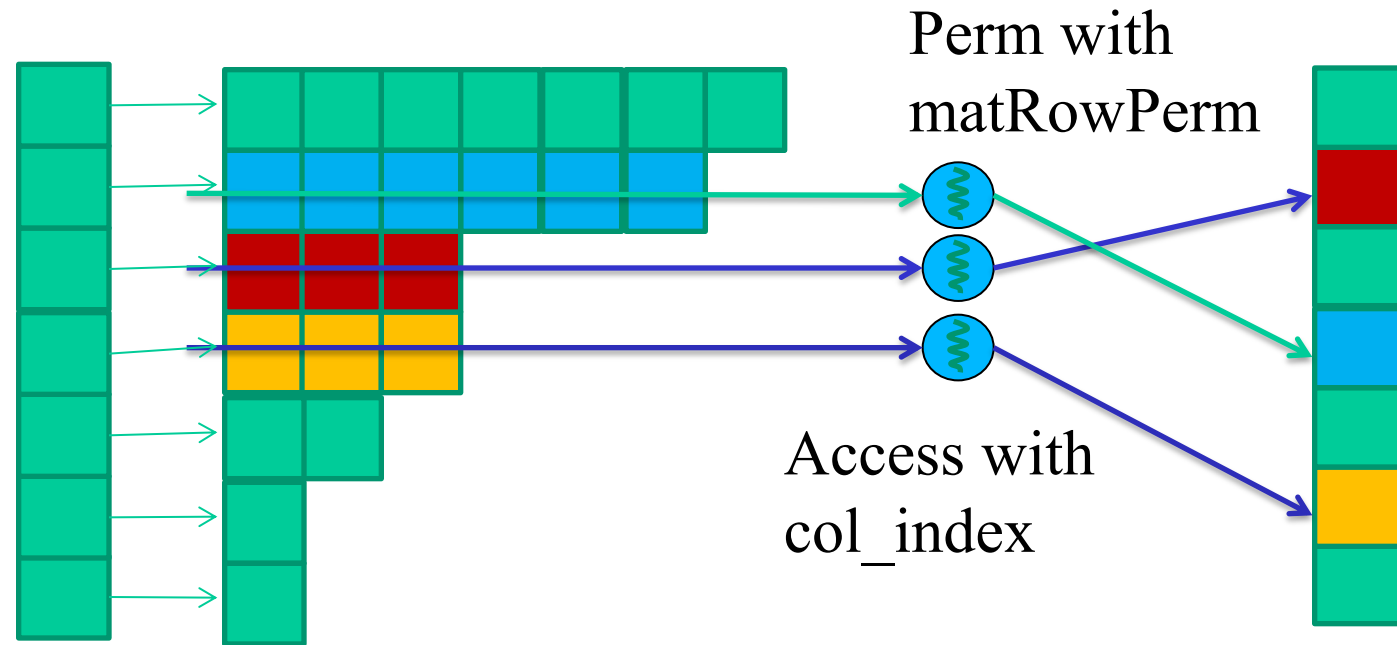| data      | 1 |
|-----------|---|
| col_index | 3 |
| row_index | 2 |

**COO**

# CSR Run-time

Block 0
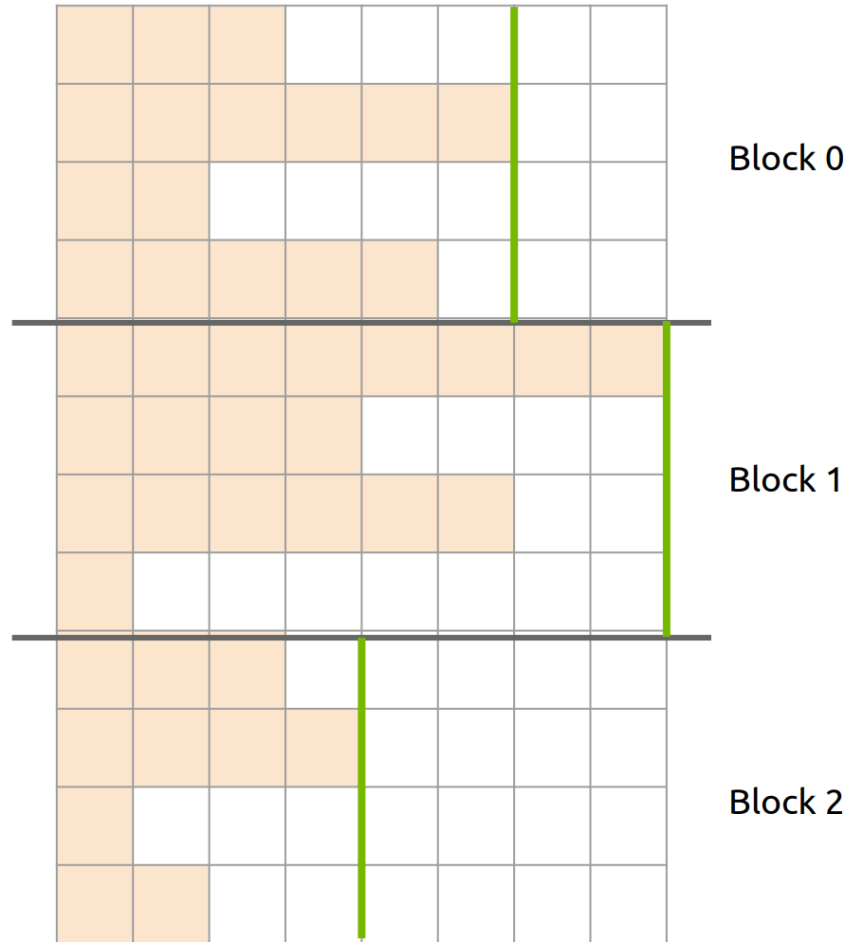
Block 1

Block 2

Block performance is
determined by longest row

# CSR-based SpMV

# JDS (Jagged Diagonal Sparse) Kernel Design for Load Balancing



Perm with matRowPerm

Access with col_index

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)



Row 0    | 3 | 1 |

Row 1

Row 2    | 2 | 4 | 1 |

Row 3    | 1 | 1 |

CSR

| 2 | 4 | 1 |    Row 2

| 3 | 1 |    Row 0

| 1 | 1 |    Row 3

Row 1

JDS
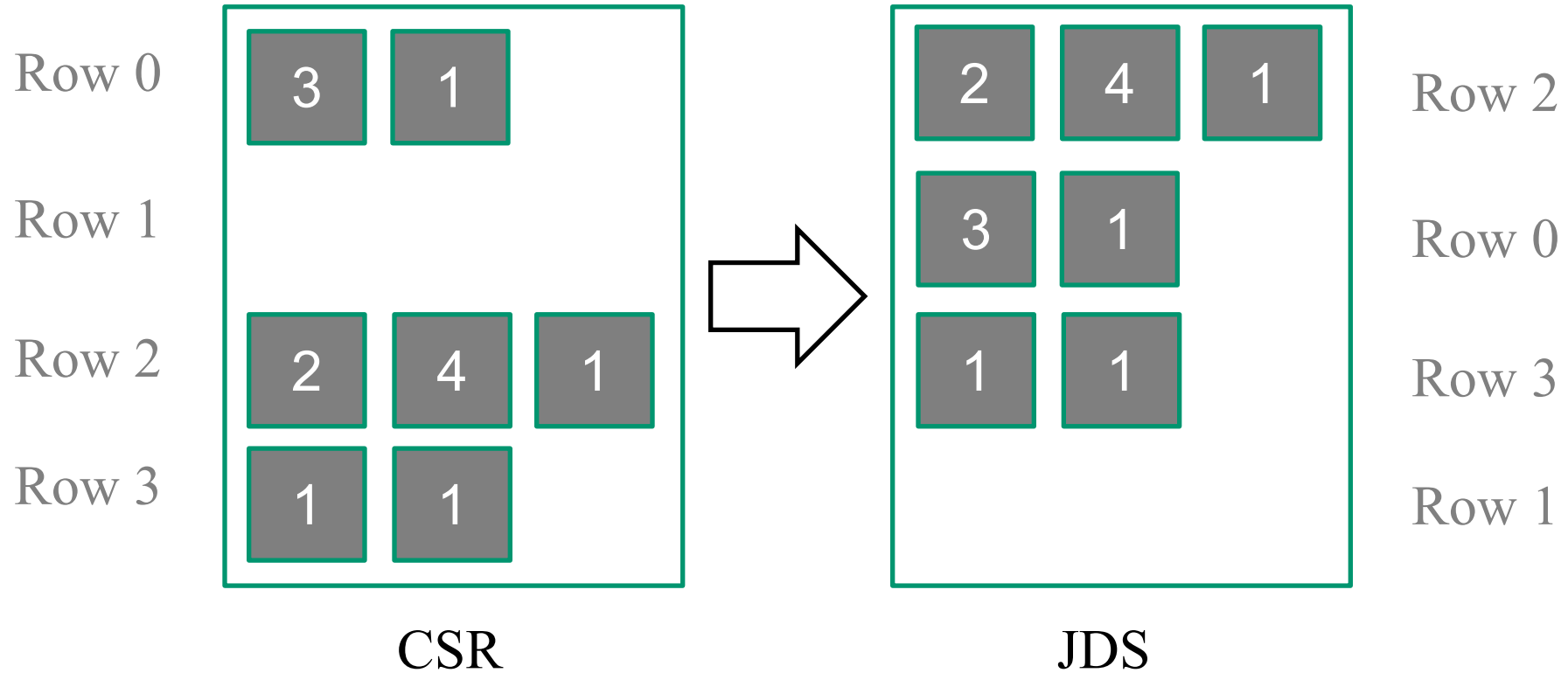
# CSR to JDS Conversion

|  | | 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 |
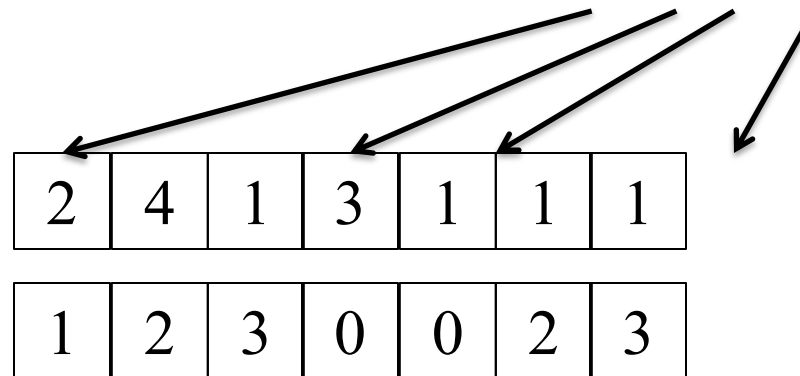
# A Parallel SpMV/JDS Kernel

```
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.    y[jds_row_perm[row]] = dot;
   }
 }
```

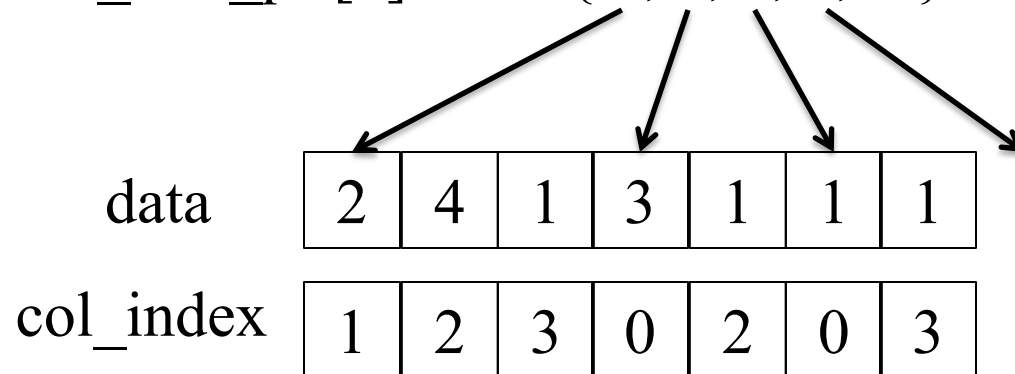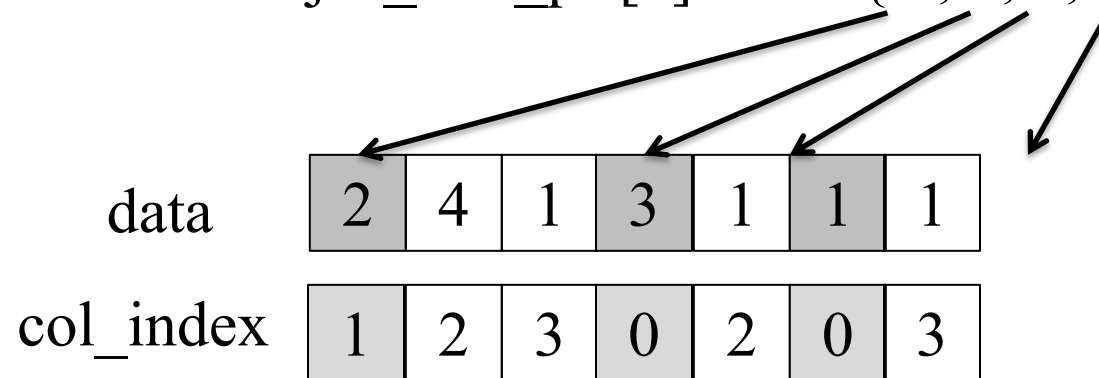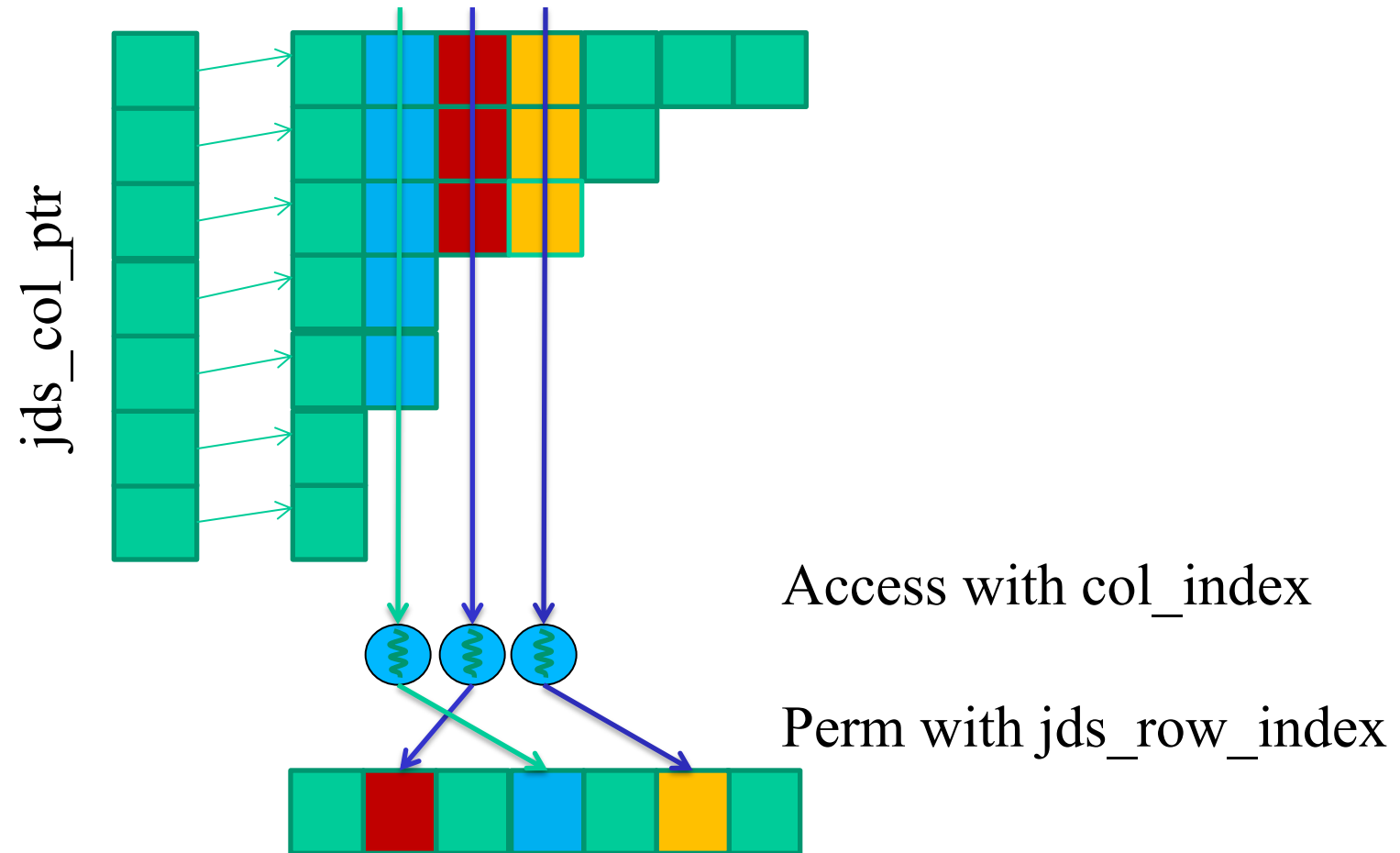|  | | Row 2 | | | Row 0 | | Row 3 | | |
|---|---|---|---|---|---|---|---|---|---|
| Nonzero values data[7] | { | 2, | 4, | 1, | 3, | 1, | 1 | 1 | } |
| Column indices col_index[7] | { | 1, | 2, | 3, | 0, | 2, | 0, | 3 | } |
| JDS Row Pointers jds_row_ptr[5] | {0, | | | 3, | | 5, | | 7,7 | } |
| JDS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

15

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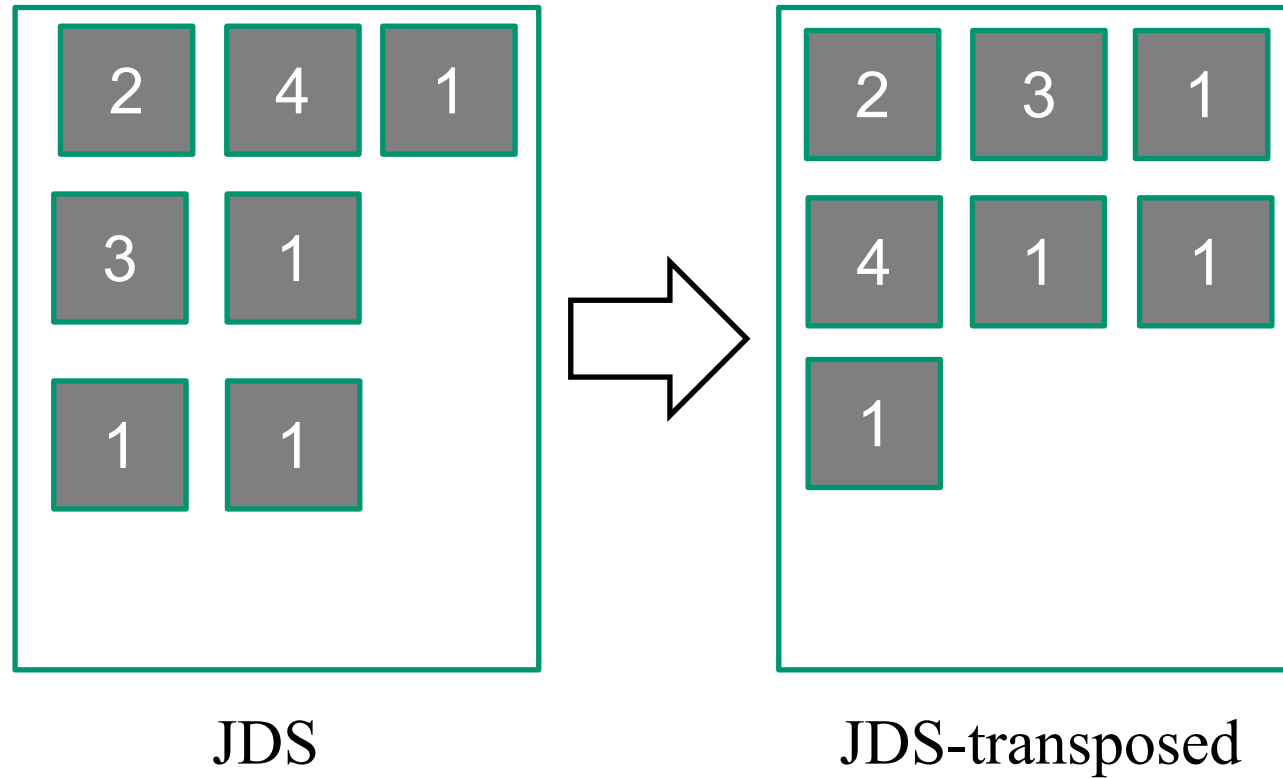
JDS row indices  jds_row_perm[4]  { 2, 0, 3, 1 }

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

data

| 2 | 4 | 1 | 3 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|

col_index

| 1 | 2 | 3 | 0 | 2 | 0 | 3 |
|---|---|---|---|---|---|---|

# JDS with Transposition



jds_col_ptr

Access with col_index

Perm with jds_row_index

# Transposition for Memory Coalescing



JDS

JDS-transposed

# 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 |
|---|---|---|---|---|---|---|---|

| | | |
|---|---|---|
| 2 | 3 | 1 |
| 4 | 1 | 1 |
| 1 | | |
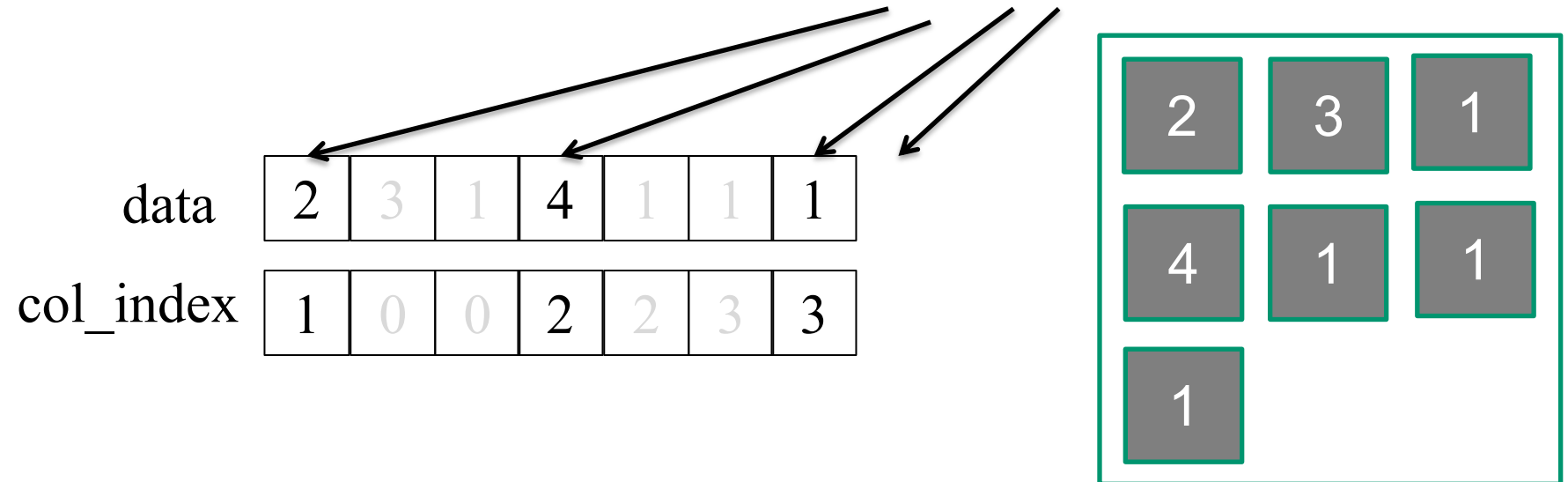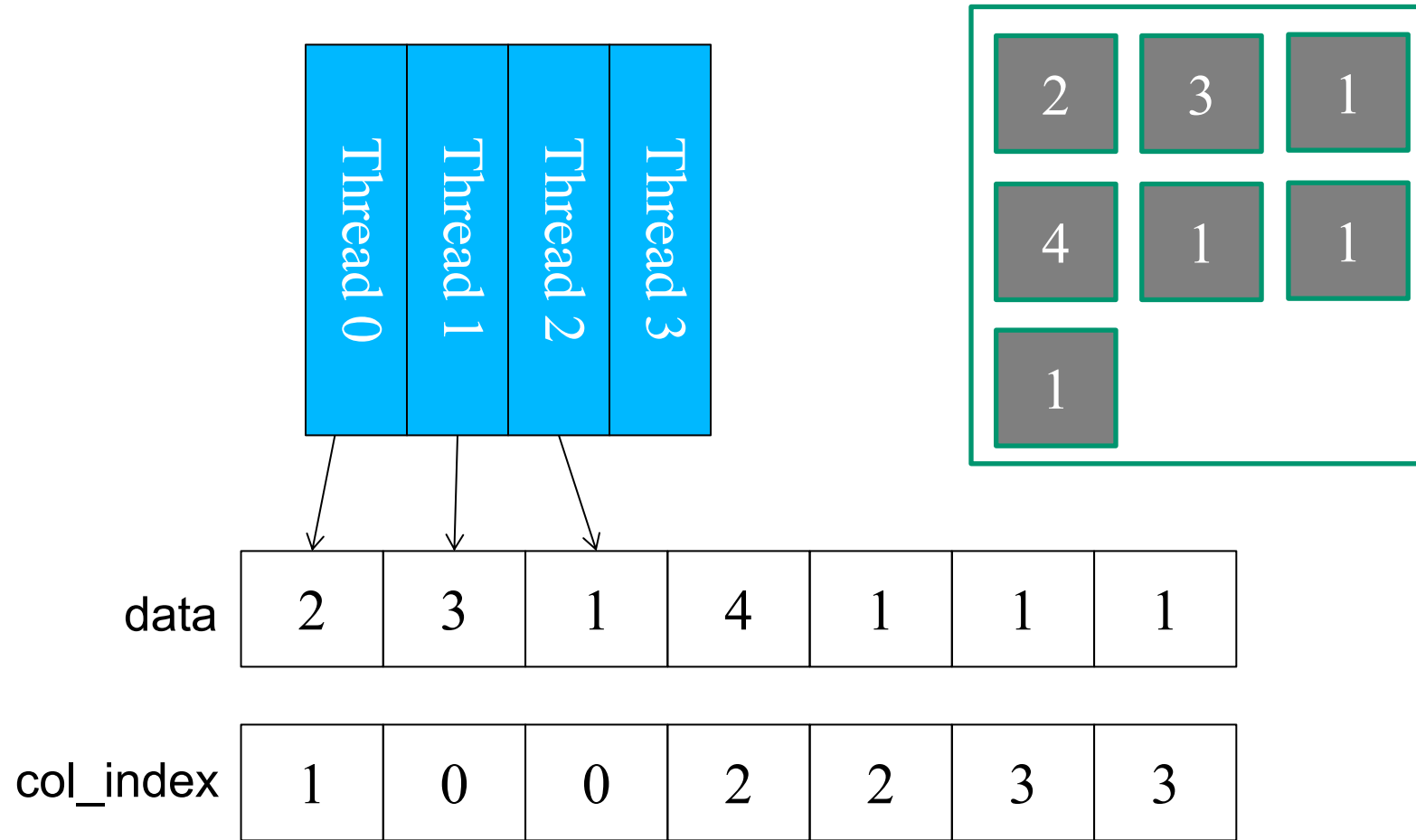
# JDS with Transposition: Memory Coalescing

# JDS with Transposition: Memory Coalescing

Thread 0  Thread 1  Thread 2  Thread 3

| 2 | 3 | 1 |
|---|---|---|
| 4 | 1 | 1 |
| 1 | | |

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
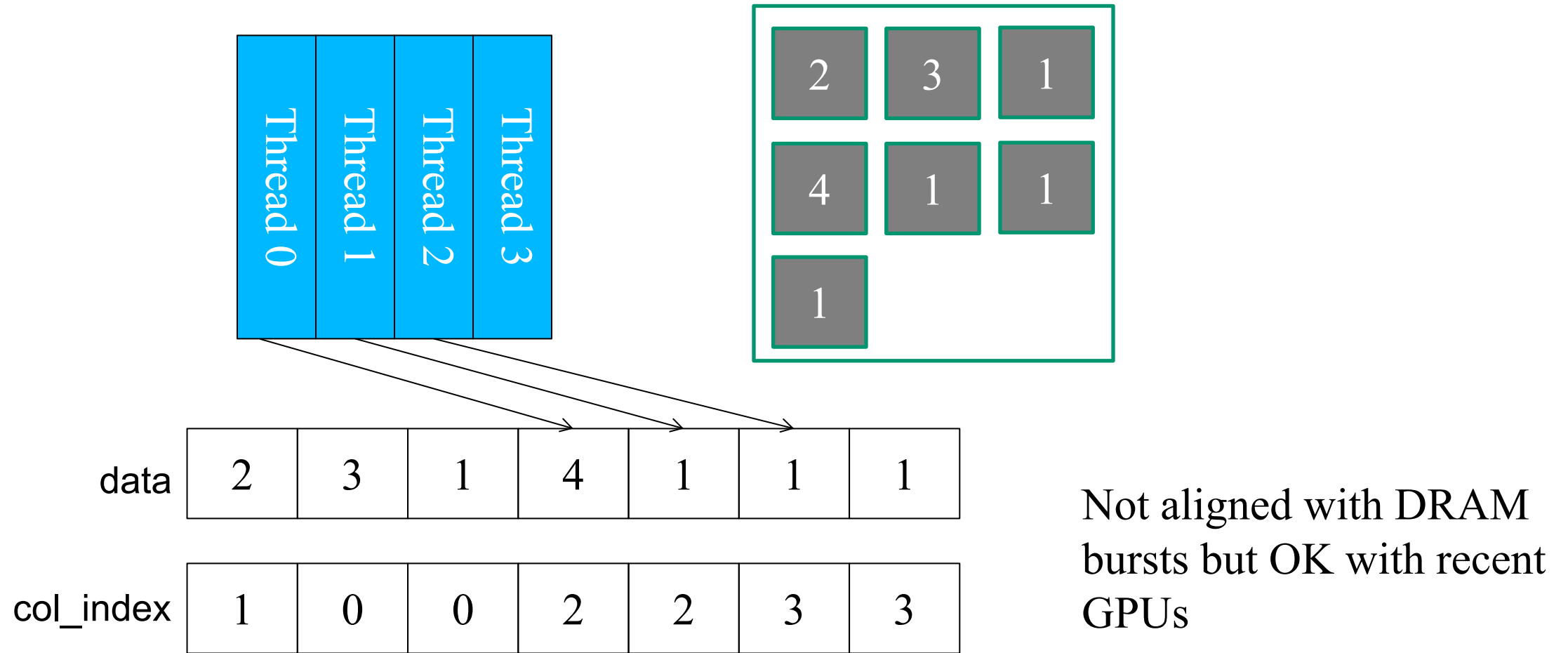
# JDS with Transposition: Memory Coalescing
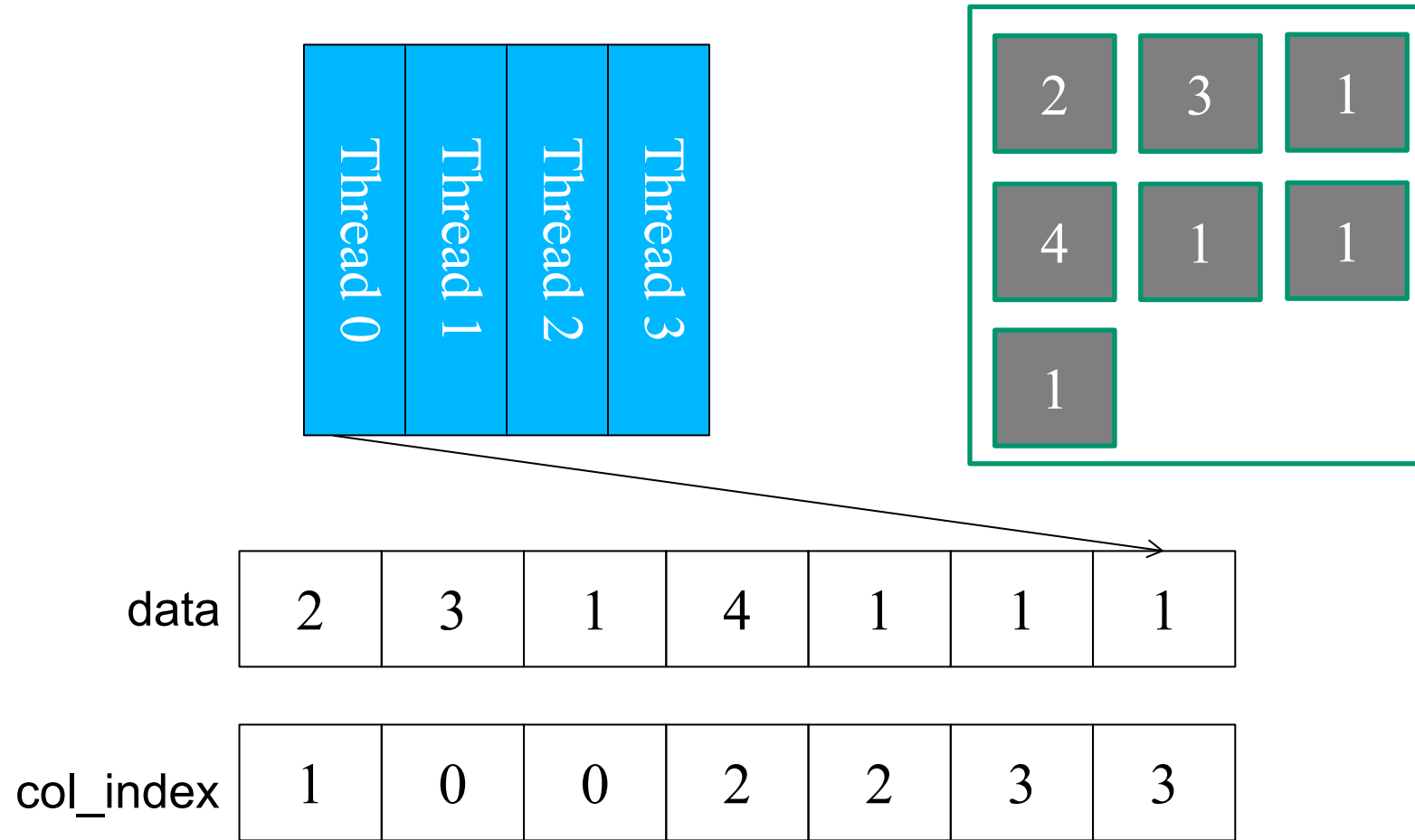
# A Parallel SpMV/JDS_T Kernel

```
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;
        unsigned int sec = 0;
5.      while (jds_t_col_ptr[sec+1] - jds_t_col_ptr[sec] > row) {
6.        dot += data[jds_t_col_ptr[sec]+row] *
                x[col_index[jds_t_col_ptr[sec]+row]];
7.        sec++;
        }
8.      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,  3, | 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 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 }

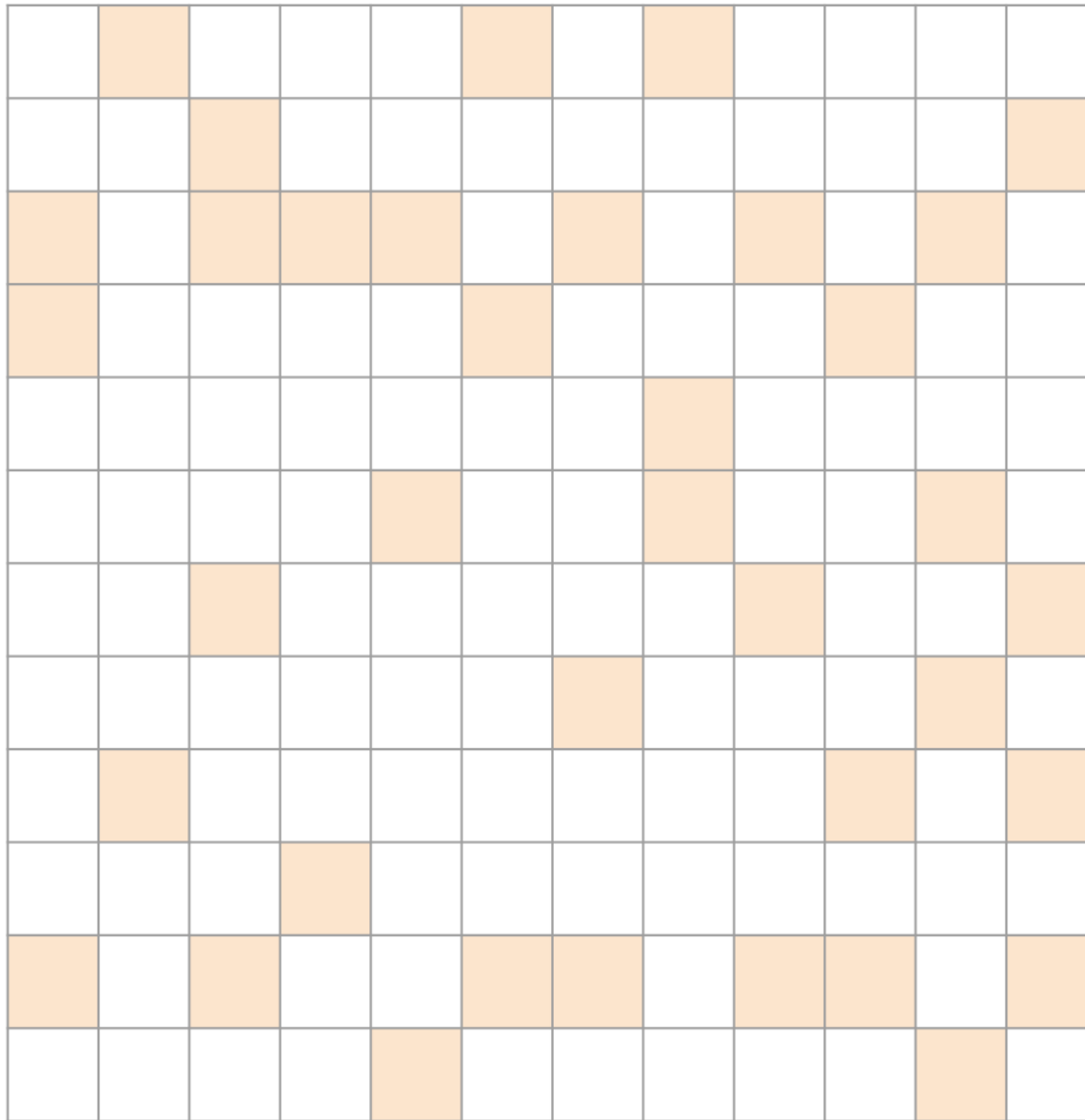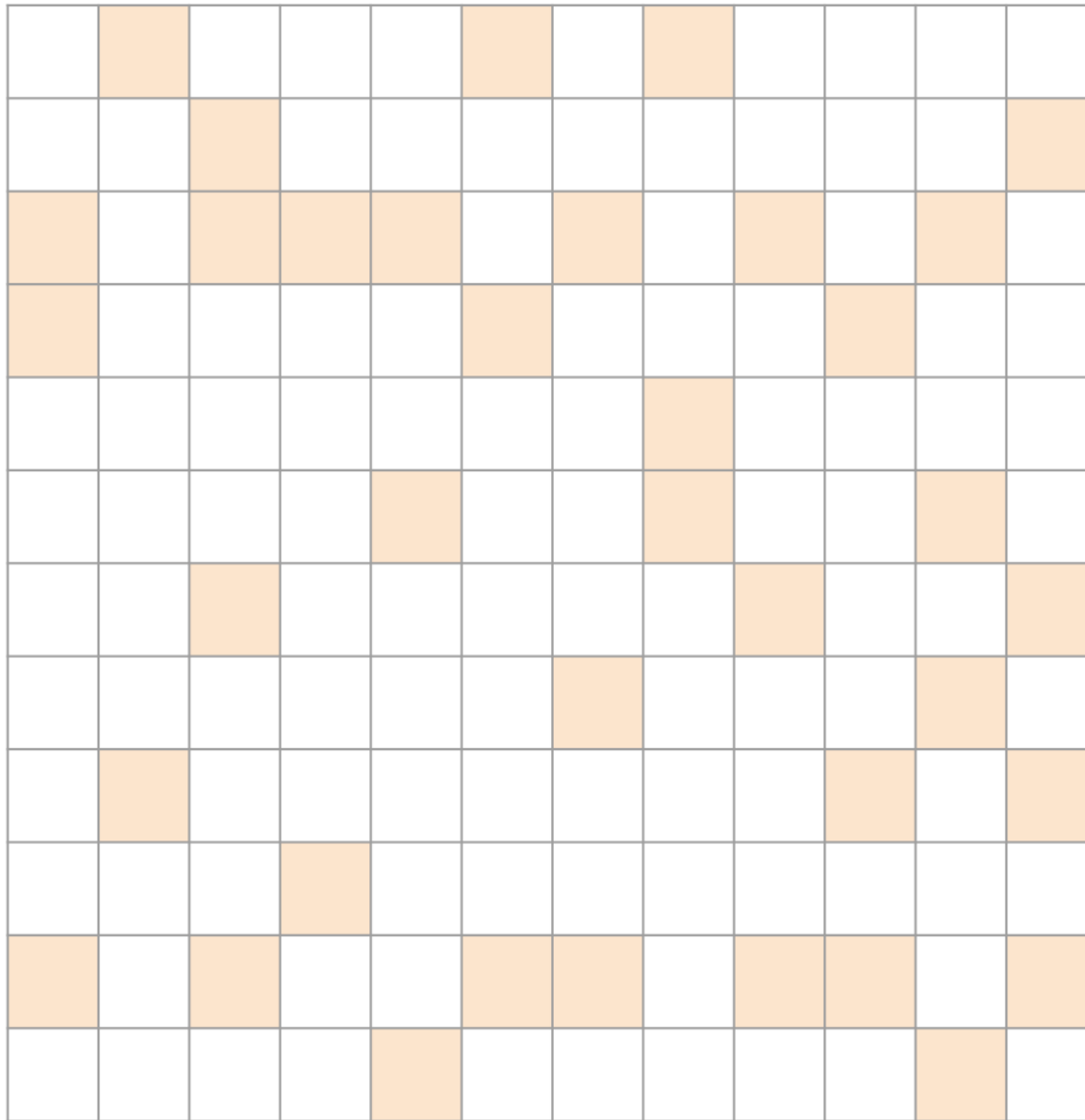JDS Row Indices  matRowPerm[4]     {2,          0,          3,        1 }
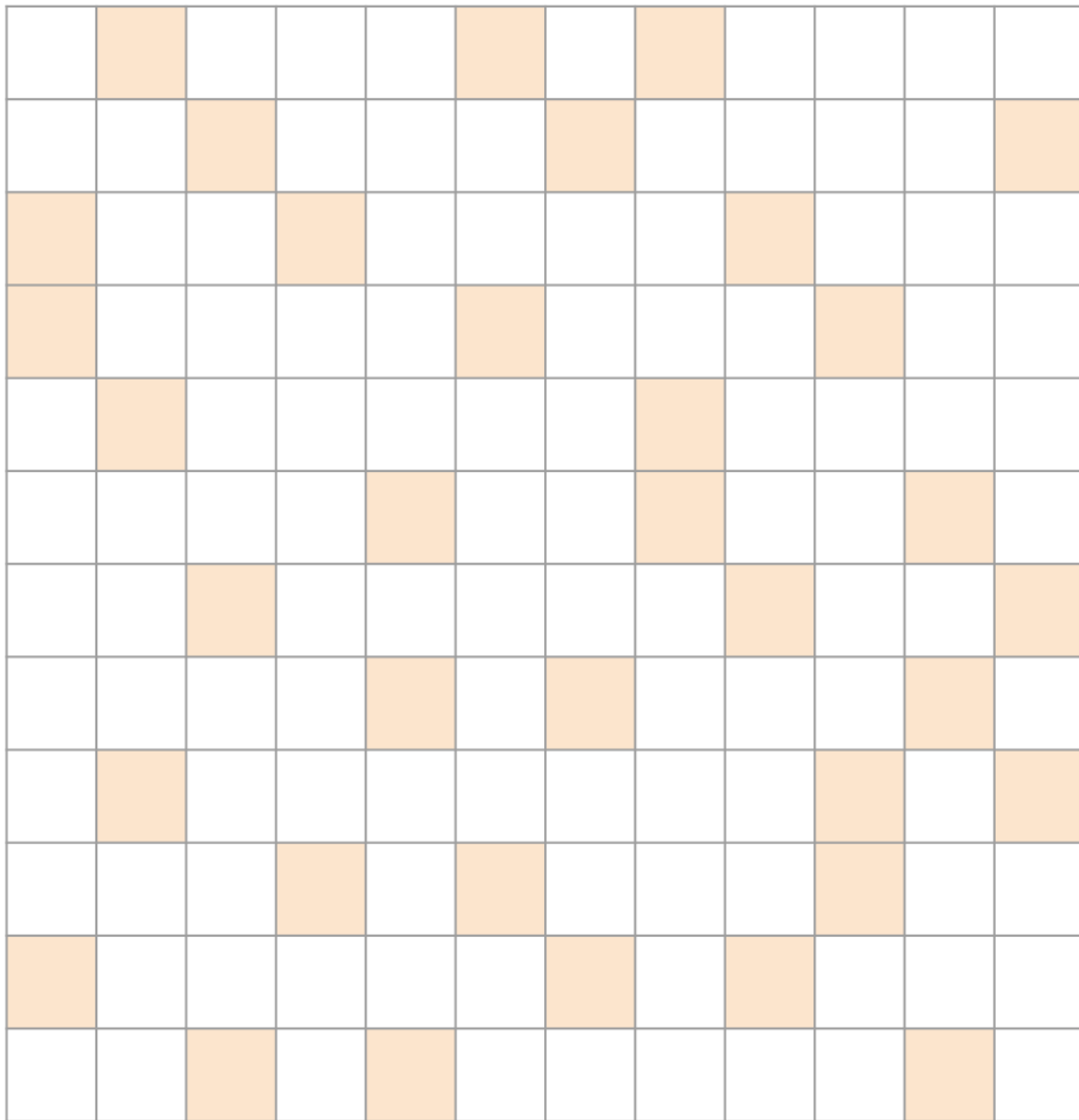
Roughly Random…
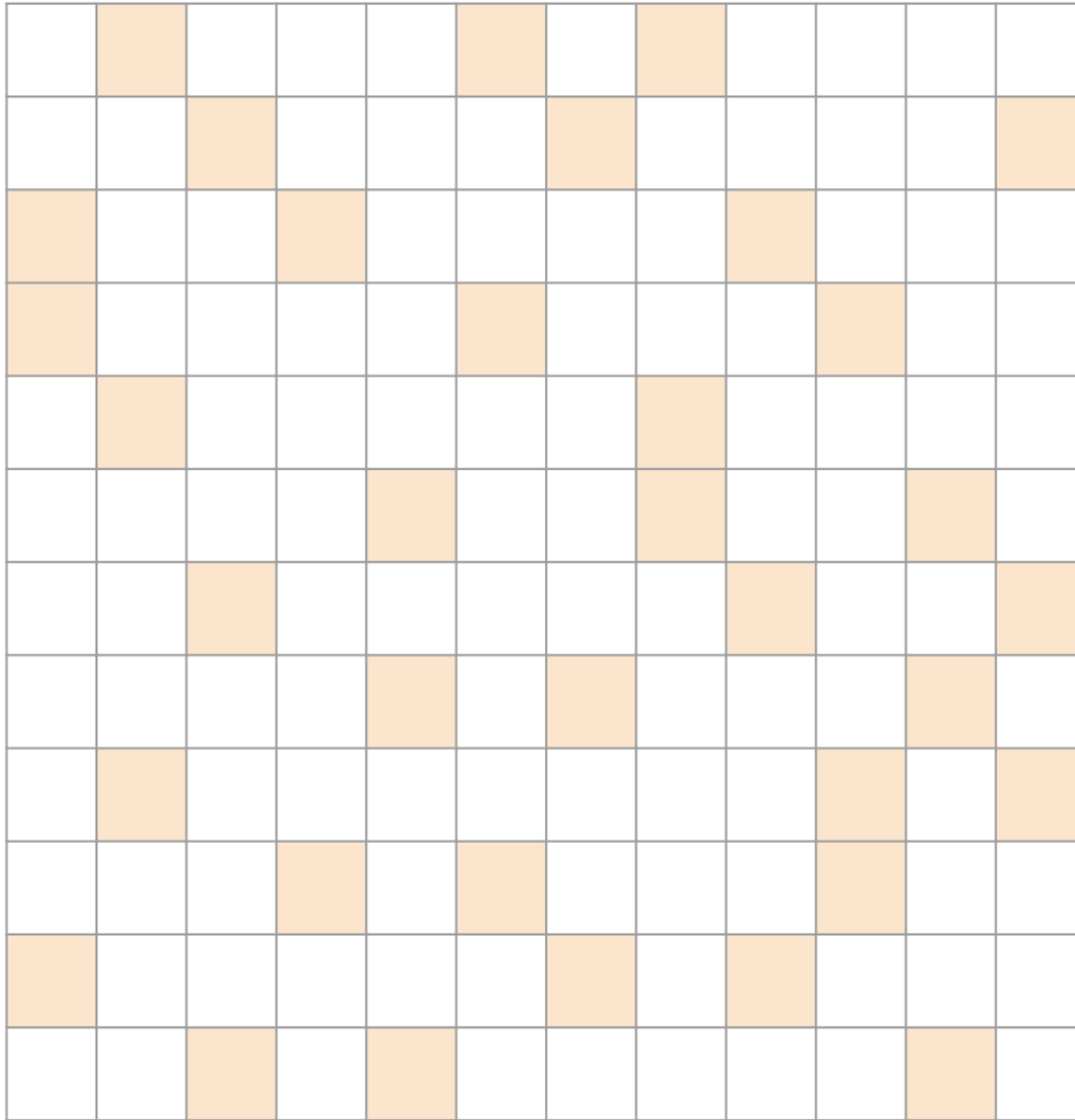
Roughly Random…

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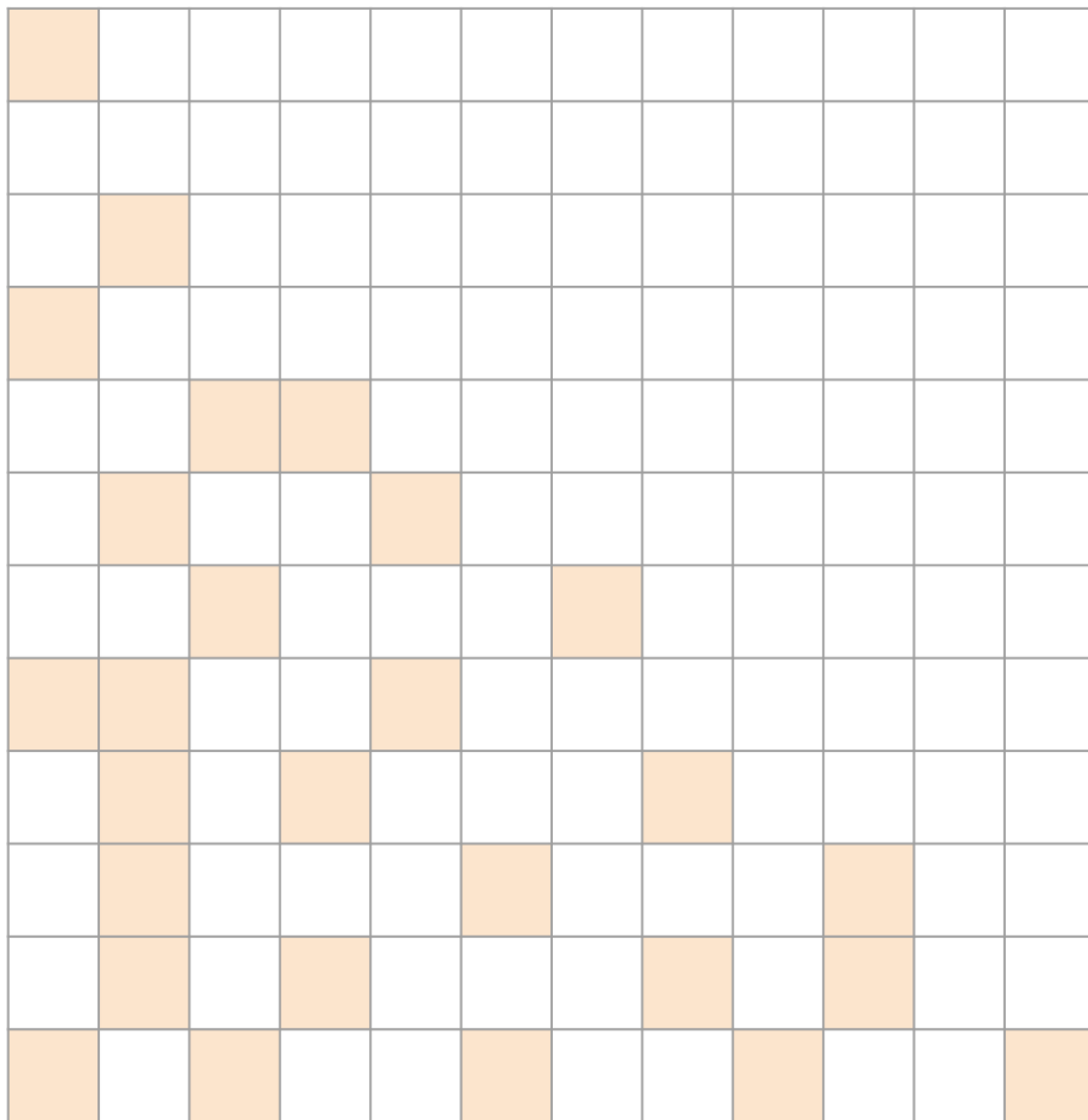- Benefit of ELL for most cases
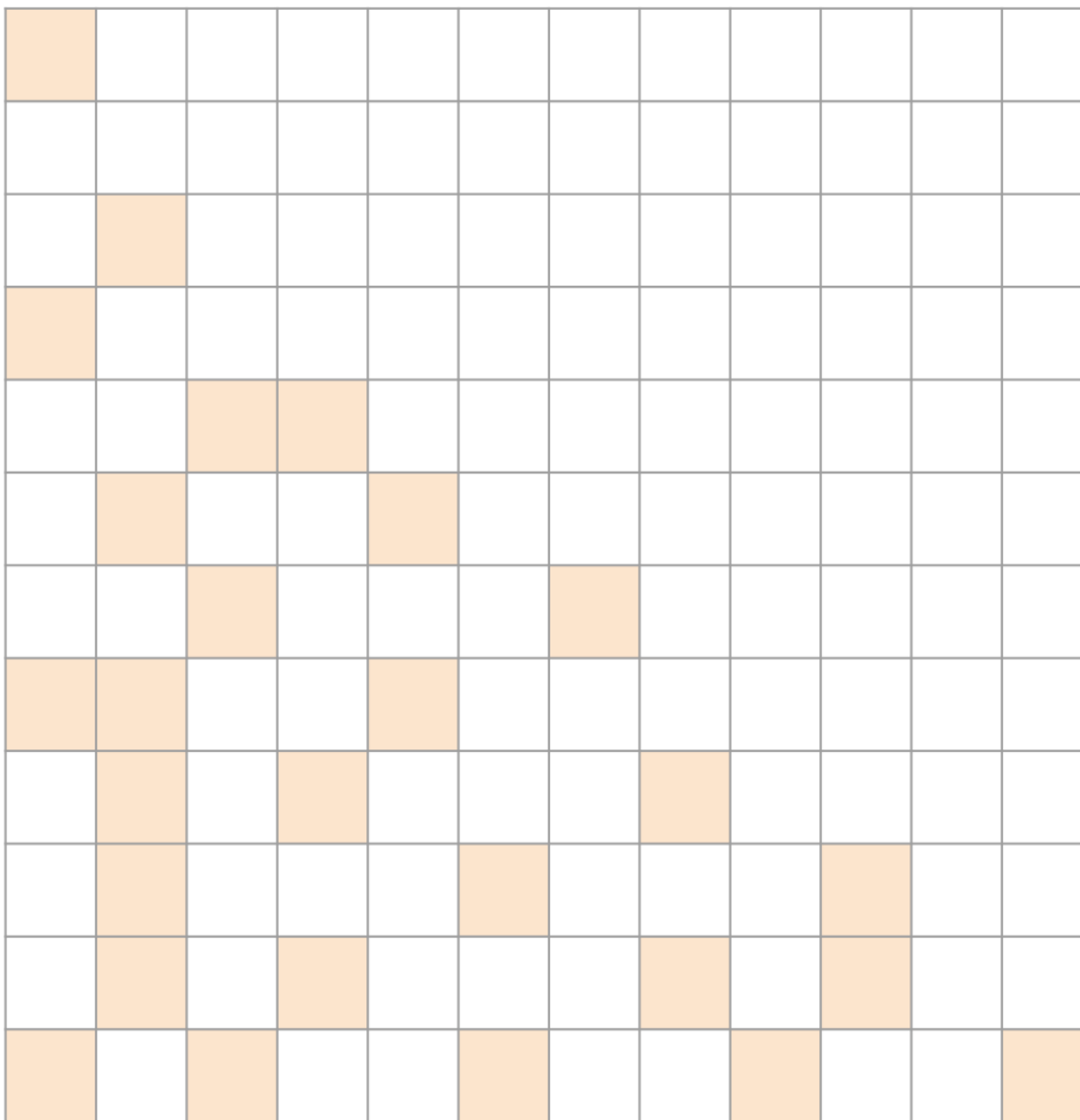- Outliers are captured with COO
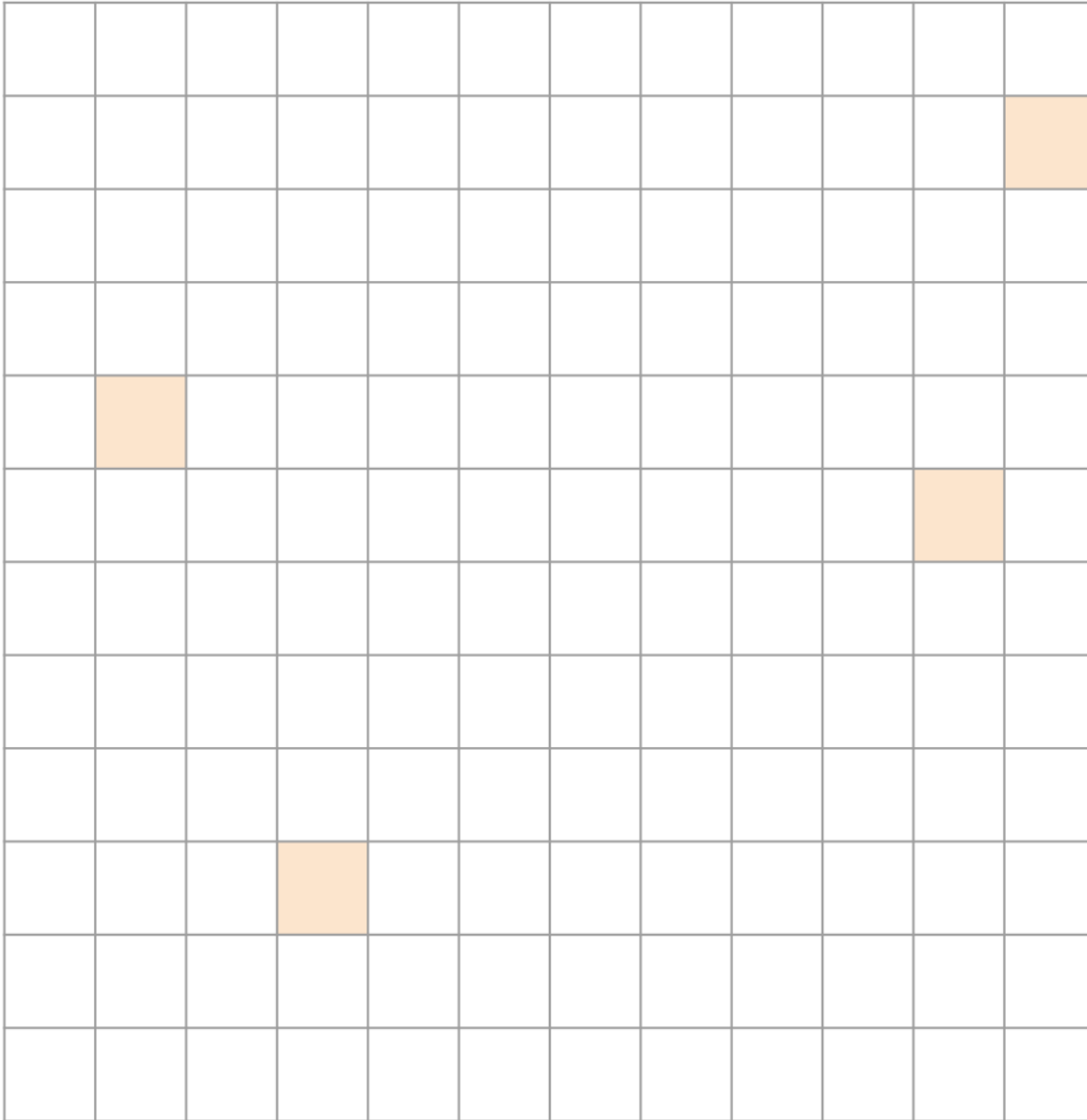
Very sparse…
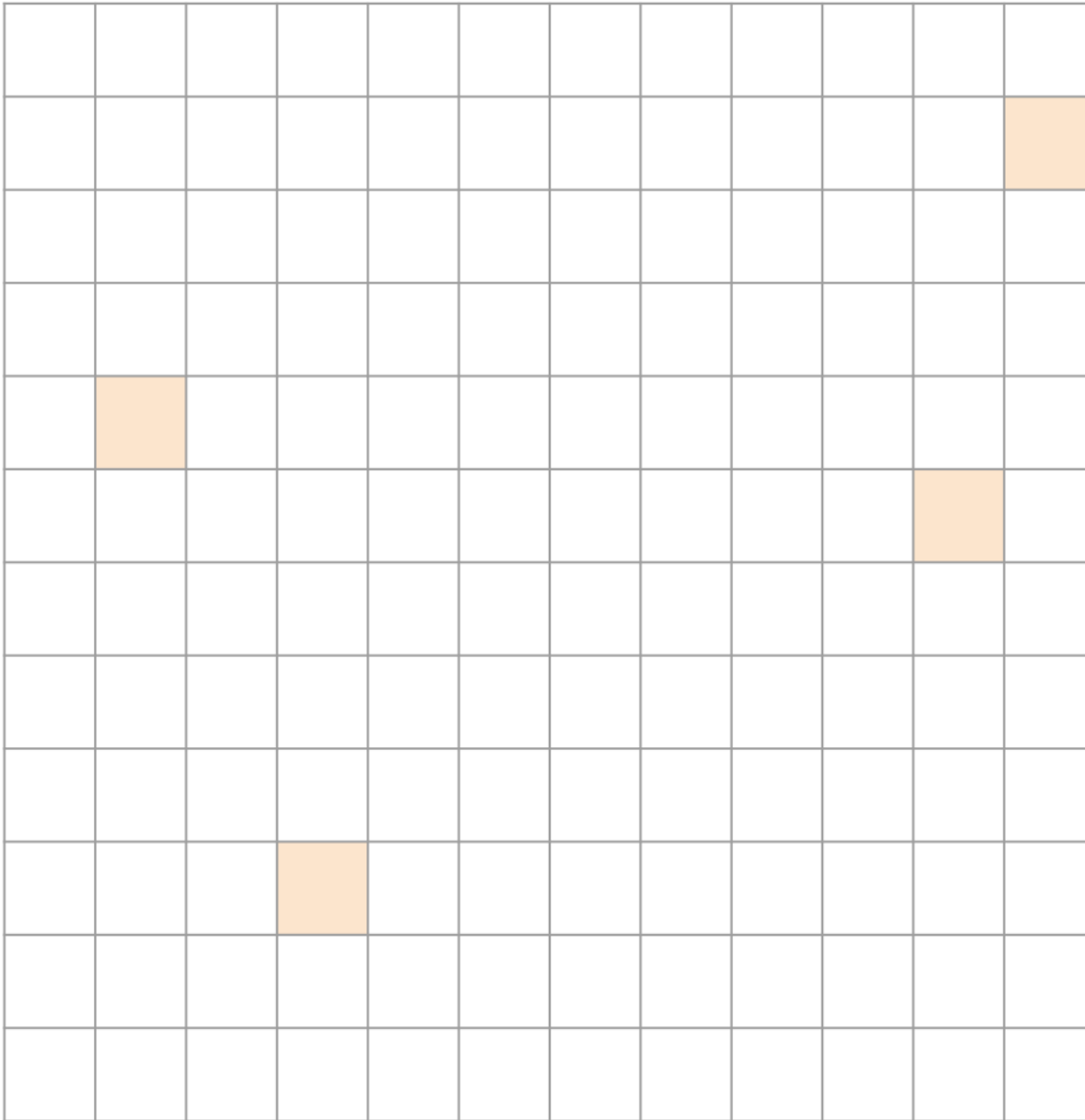
Very sparse

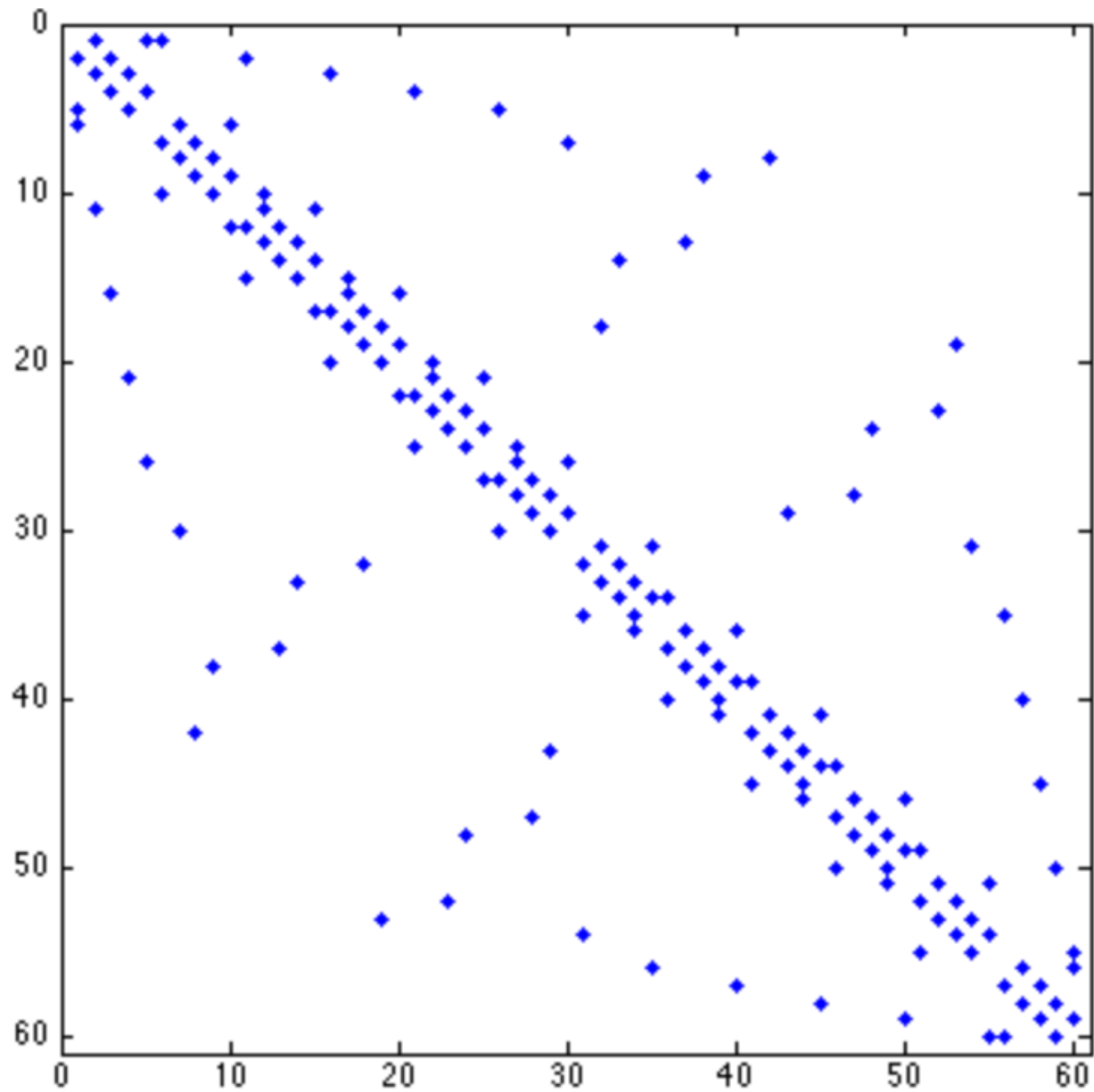Probably best with COO
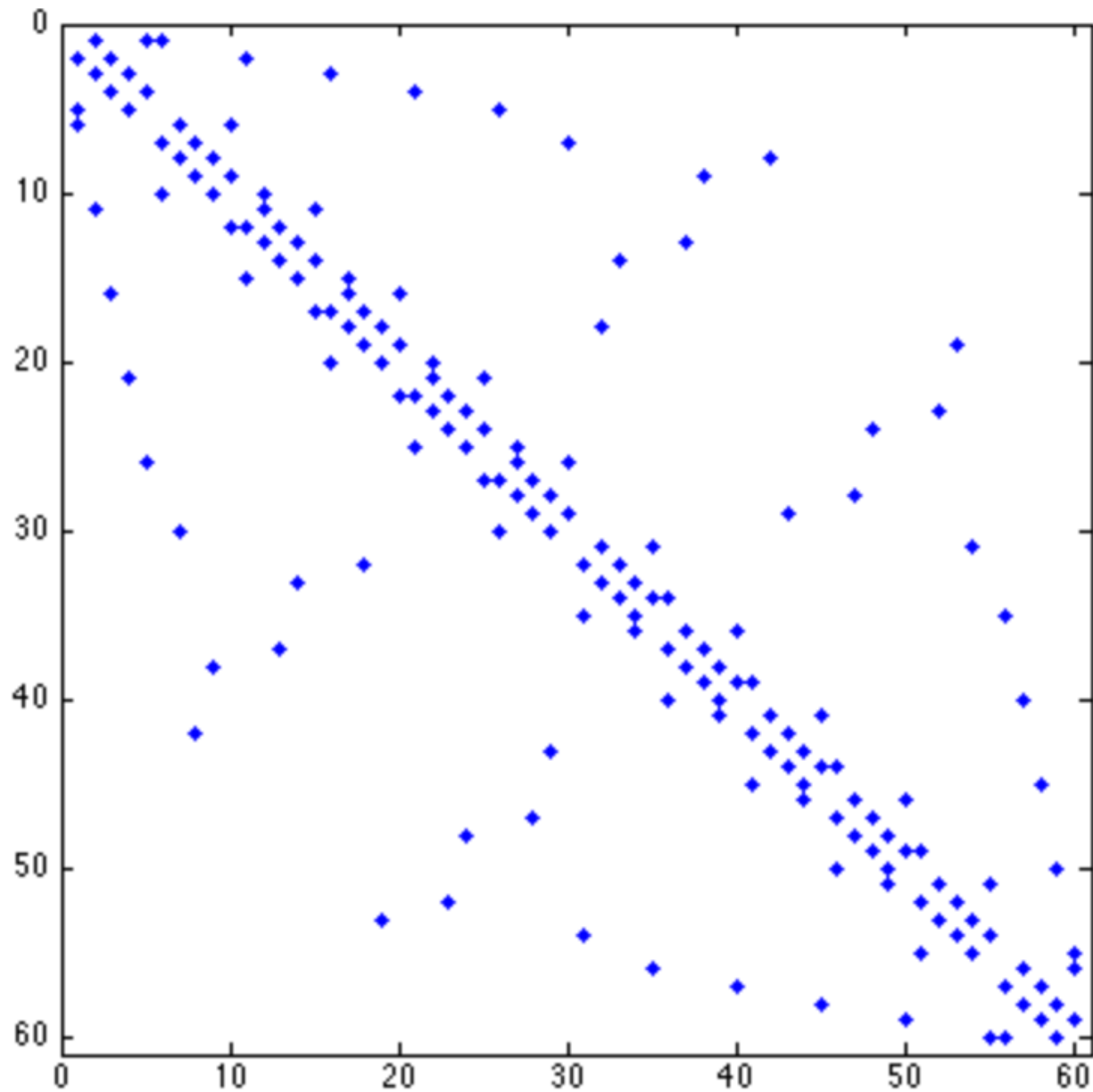- Not a lot of data, compute is sparse

Roughly triangular…

Roughly triangular…

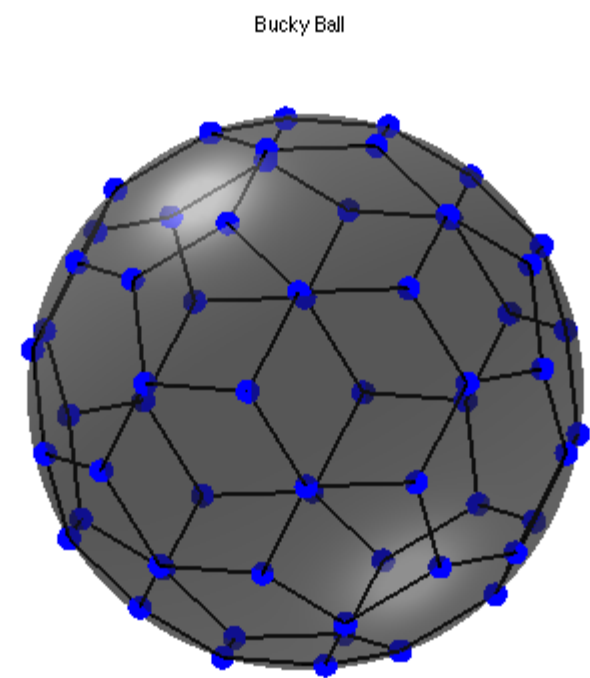Probably best with JDS
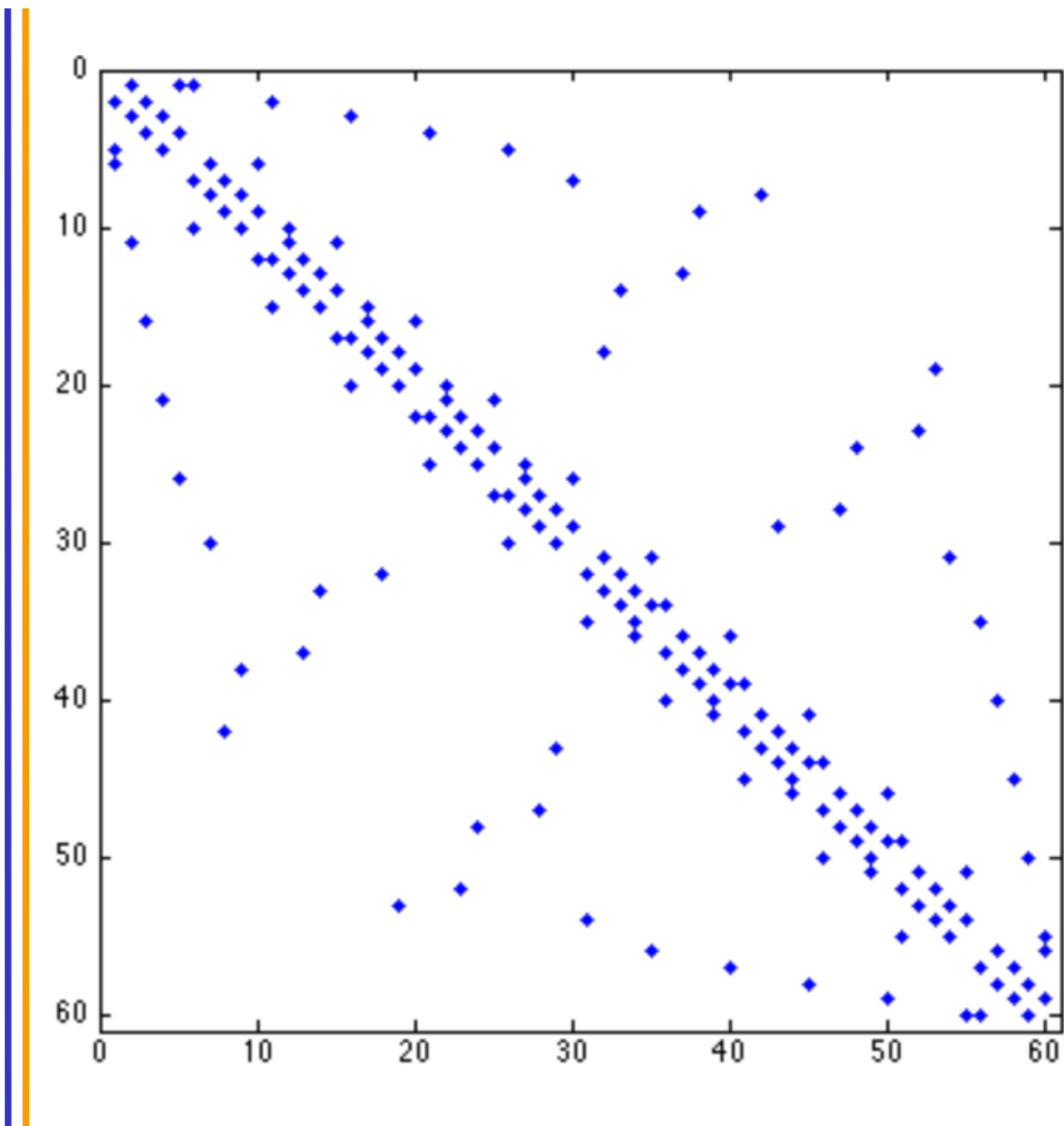- Takes advantage of sparsity structure

Banded Matrix…

Banded Matrix…

Probably best with ELL
- Small amount of variance in rows

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

- What about situations where sparsity is dynamic, i.e., the large sparse systems are dynamically generated at run-time?

# 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

# ANY MORE QUESTIONS
# READ CHAPTER 10