ECE408 / CS483 / CSE408
Summer 2024

Applied Parallel Programming

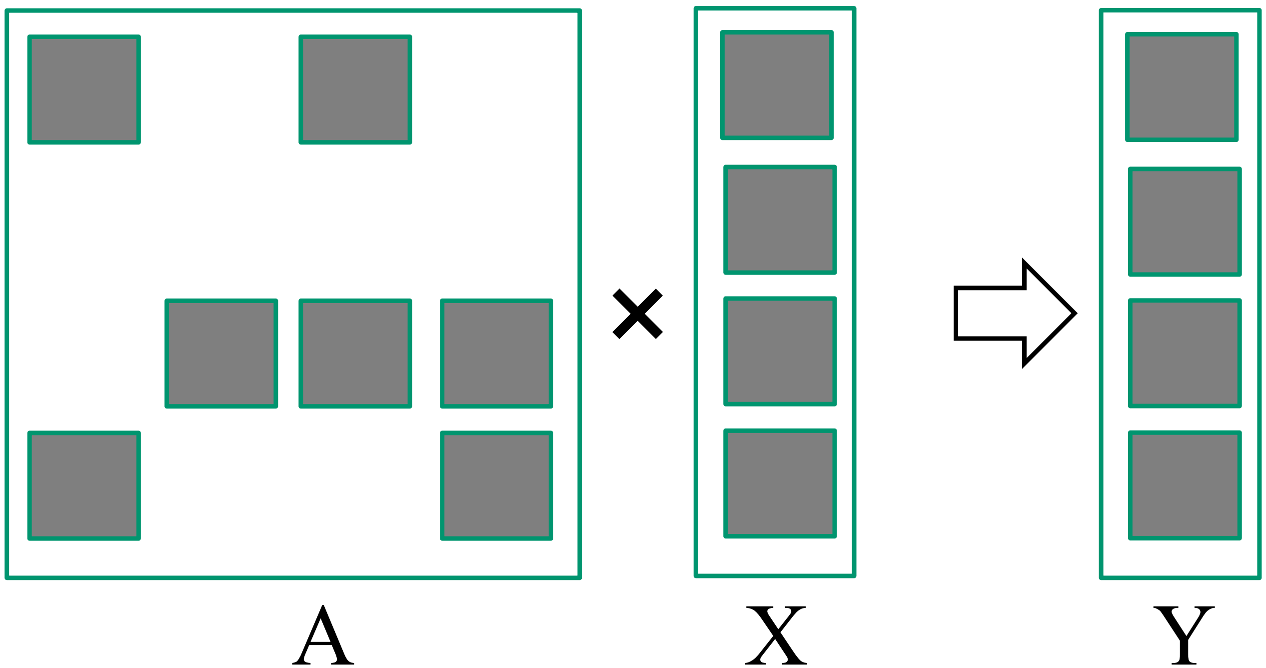
Lecture 15: Parallel Sparse Methods (Part 2)

What Will You Learn Today?

to regularize irregular data by
- limiting variations with clamping,
- sorting, and
- transposition

to write
- a high-performance SpMV kernel
- based on JDS transposed format

Review: Sparse Matrix-Vector Multiplication (SpMV)



Review: Compressed Sparse Row (CSR) Format

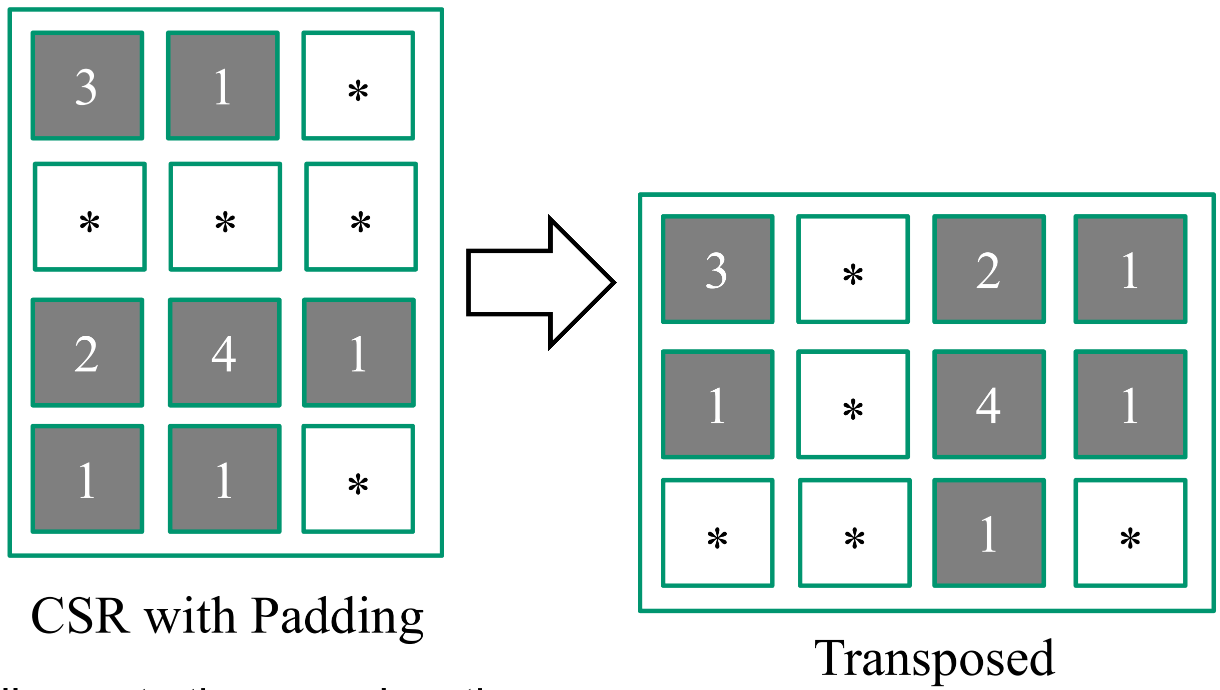
CSR Representation

		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

Review: 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

Review: Coordinate (COO) format

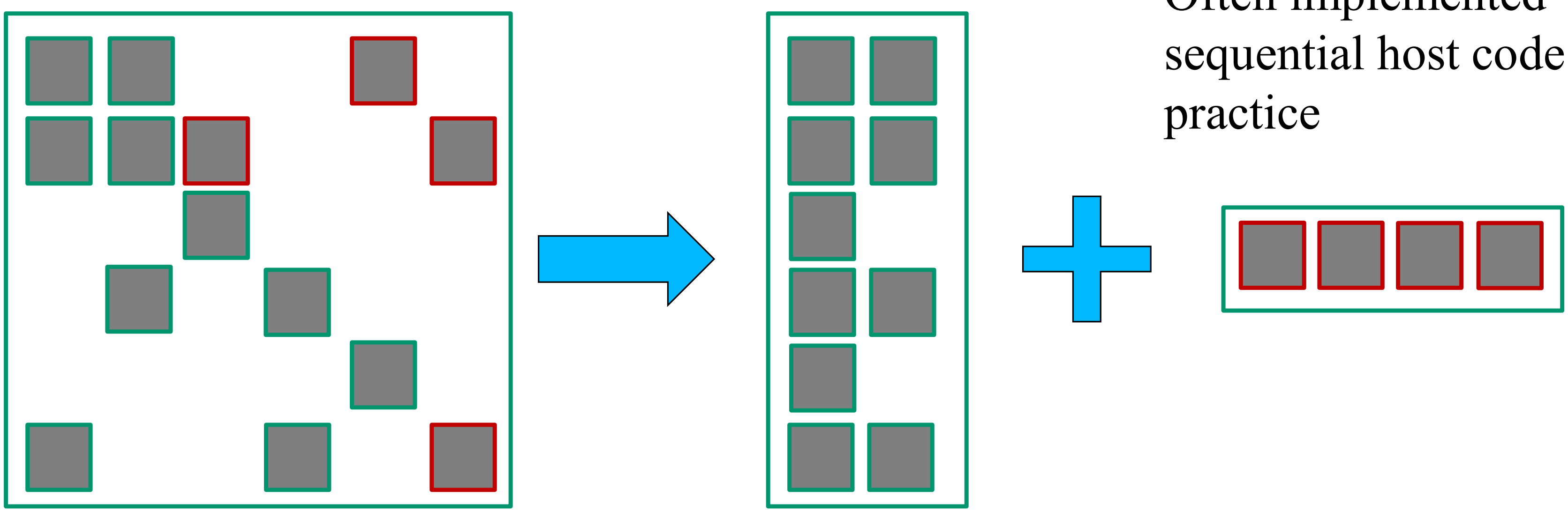
- Explicitly list the column and row indices for every non-zero element

		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	}

Review: Hybrid Format (ELL + COO)

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

Often implemented in sequential host code in practice



Review: Reduced Padding with Hybrid Format

Thread 0 | Thread 1 | Thread 2 | Thread 3

	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

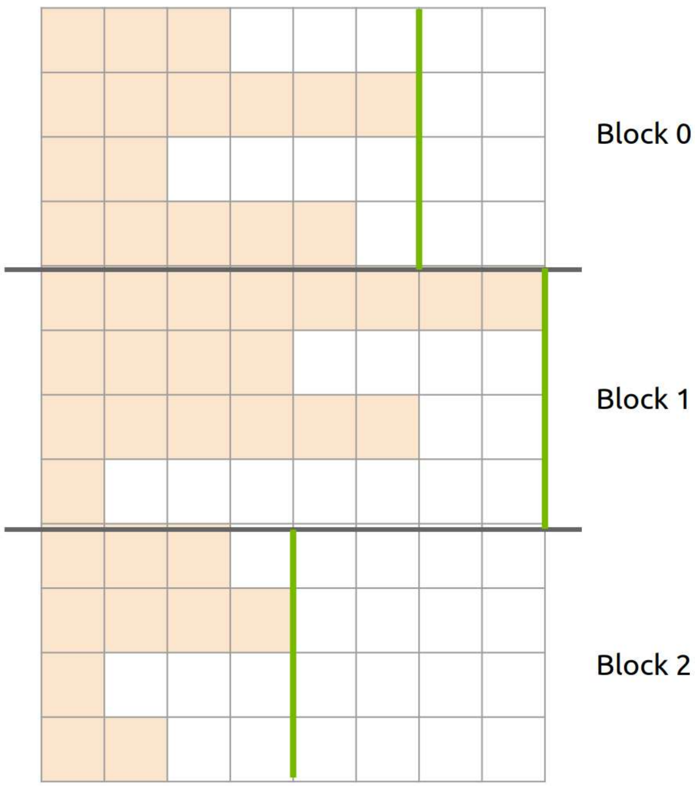
data	3	*	2	1	1	*	4	1
index	0	*	1	0	2	*	2	3

ELL

data	1
col_index	3
row_index	2

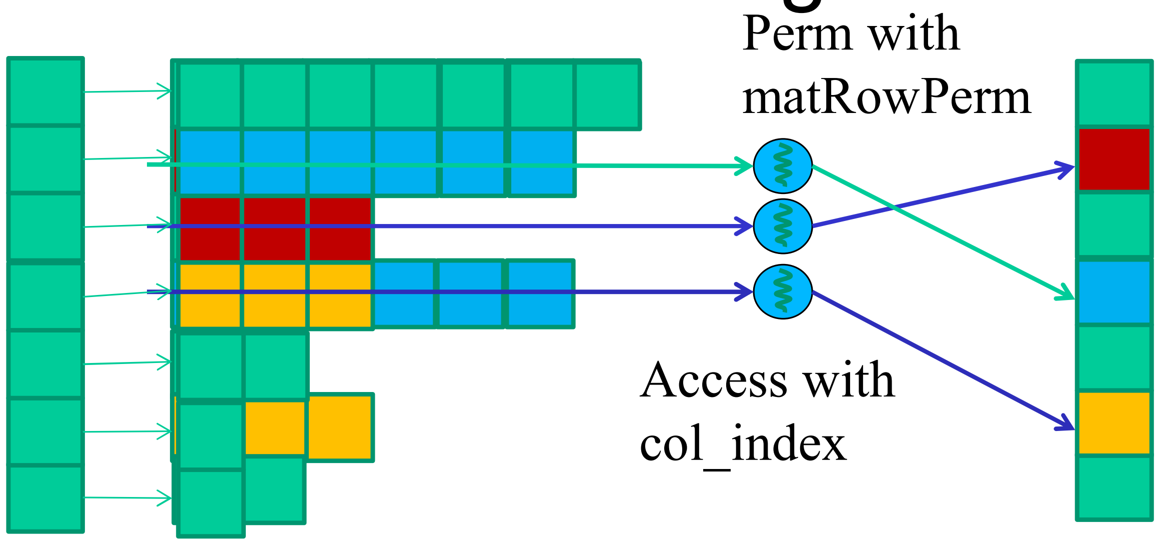
COO

CSR Run-time



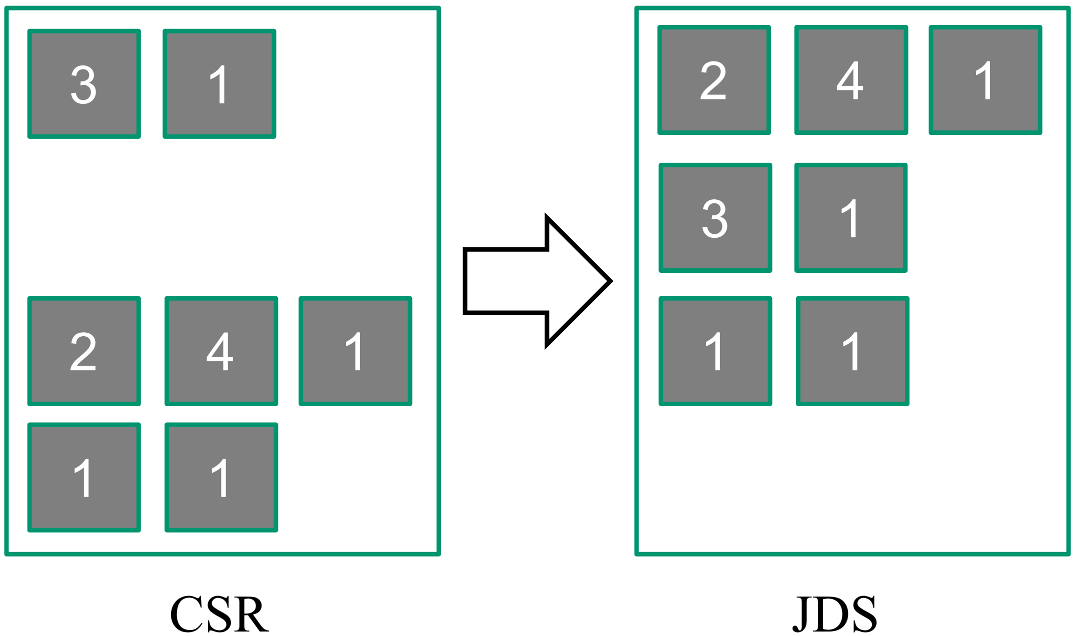
Block performance is determined by longest row

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)



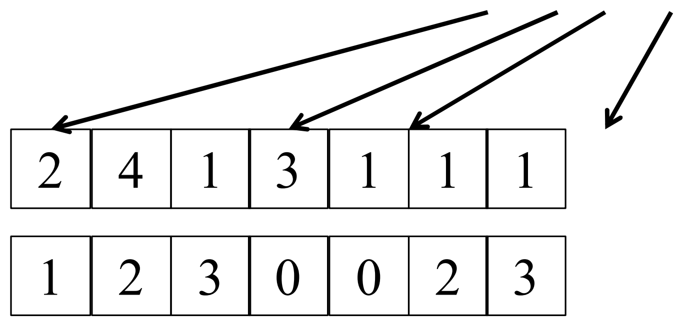
CSR to JDS Conversion

		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

```
__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

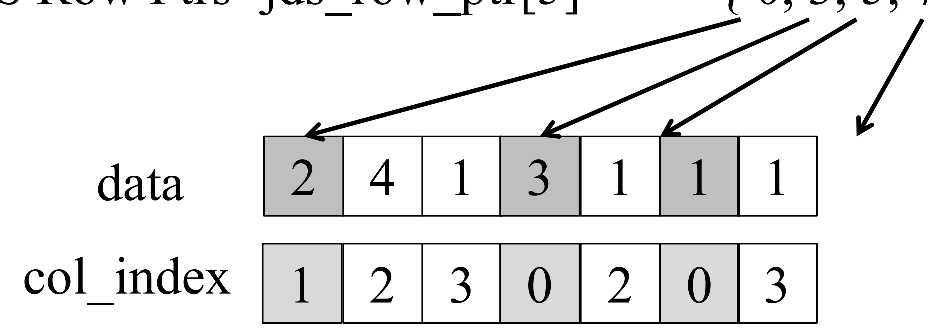
Nonzero values	data[7]	{ 2, 4, 1, 3, 1, 1, 1 }
Column indices	col_index[7]	{ 1, 2, 3, 0, 2, 0, 3 }
JDS row indices	Jds_row_perm[4]	{ 2, 0, 3, 1 }
JDS Row Ptrs	Jds_row_ptr[5]	{ 0, 3, 5, 7, 7 }

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

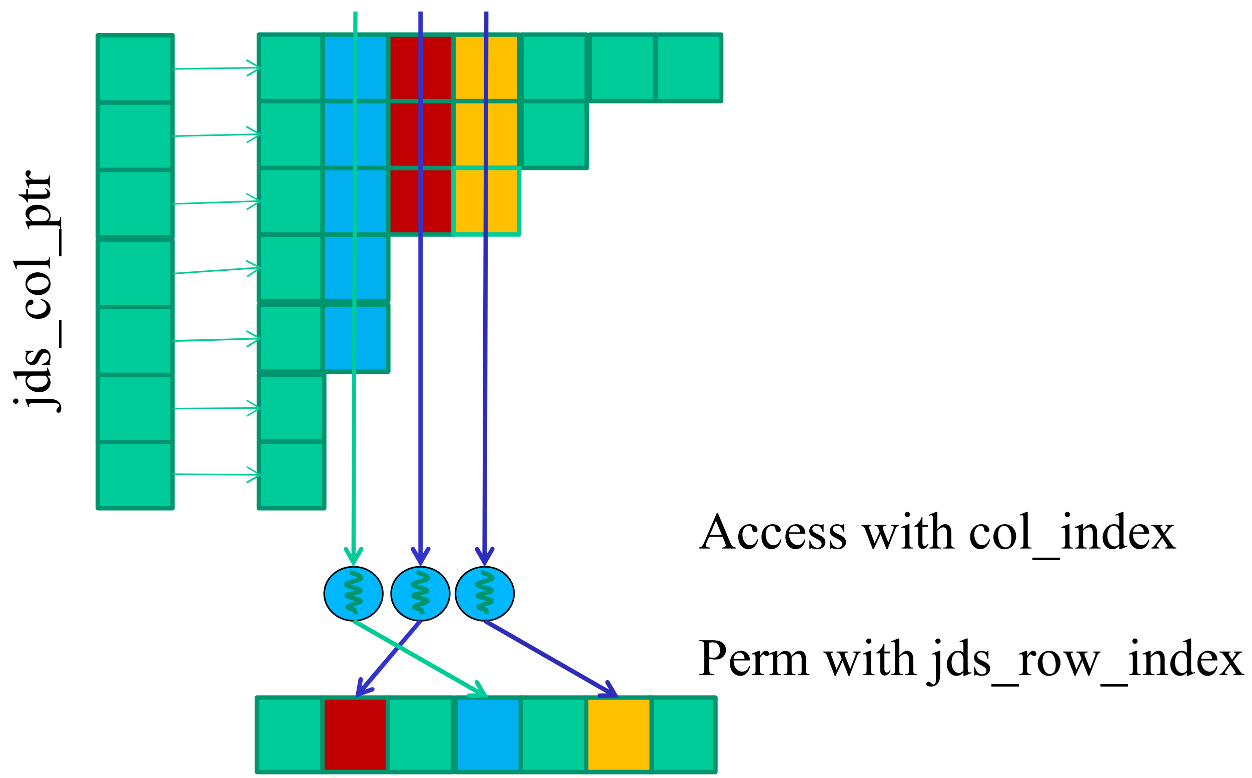
JDS vs. CSR Memory Divergence

- Adjacent threads still access non-adjacent memory locations

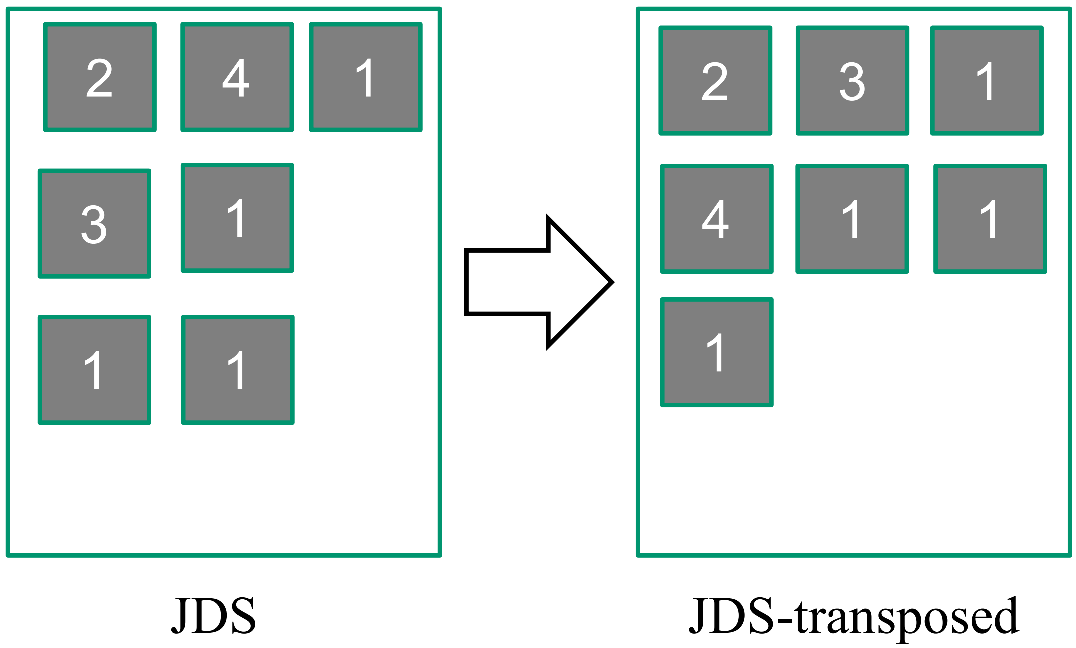
Nonzero values	data[7]	{ 2, 4, 1, 3, 1, 1, 1 }
Column indices	col_index[7]	{ 1, 2, 3, 0, 2, 0, 3 }
JDS row indices	jds_row_perm[4]	{ 2, 0, 3, 1 }
JDS Row Ptrs	jds_row_ptr[5]	{ 0, 3, 5, 7, 7 }



JDS with Transposition



Transposition for Memory Coalescing



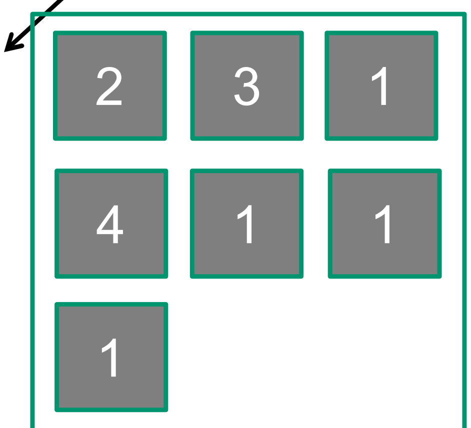
JDS Format with Transposed Layout

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

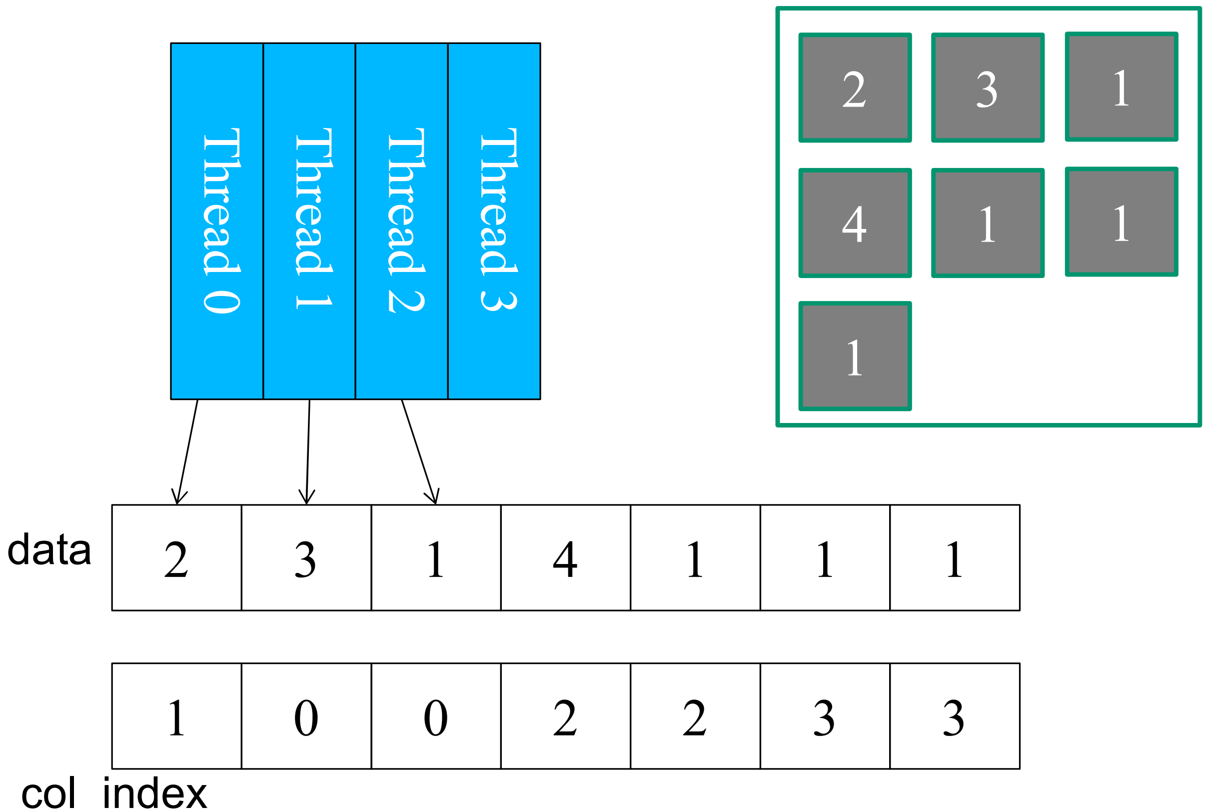
JDS row indices Jds_row_perm[4] { 2, 0, 3, 1 }

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

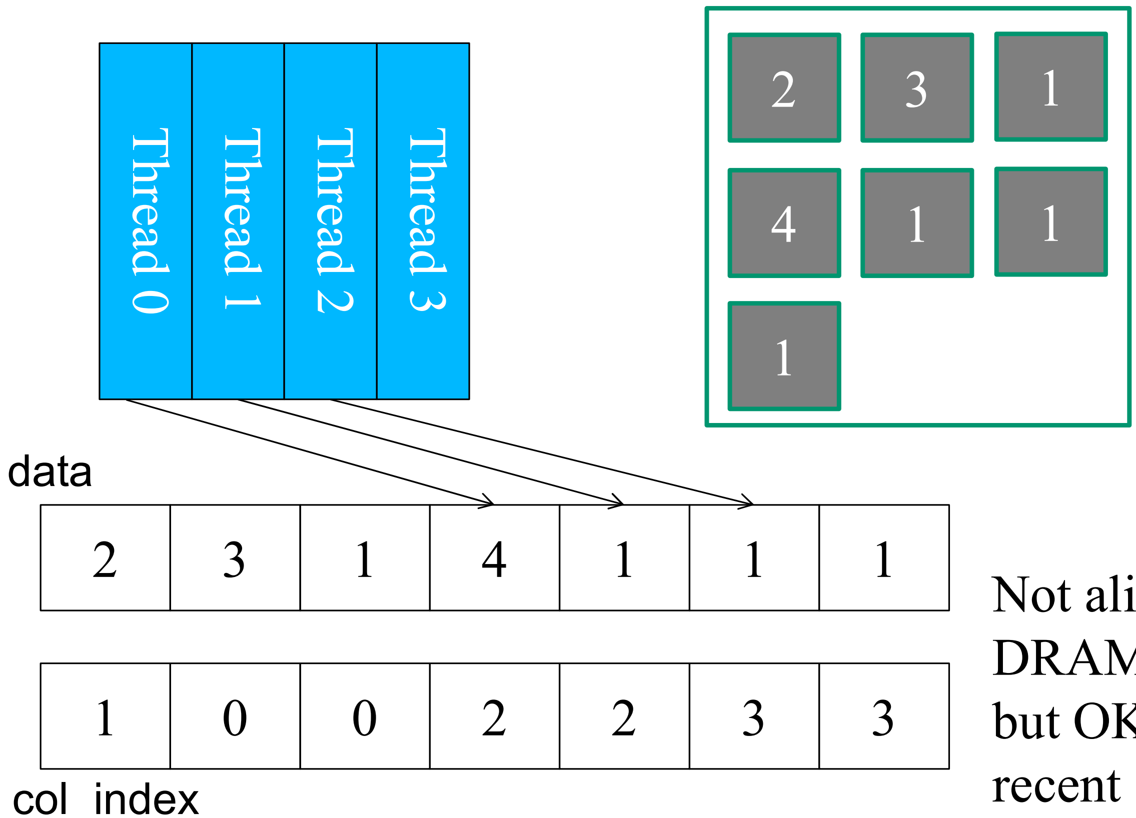
data	2	3	1	4	1	1	1
col_index	1	0	0	2	2	3	3



JDS with Transposition Memory Coalescing

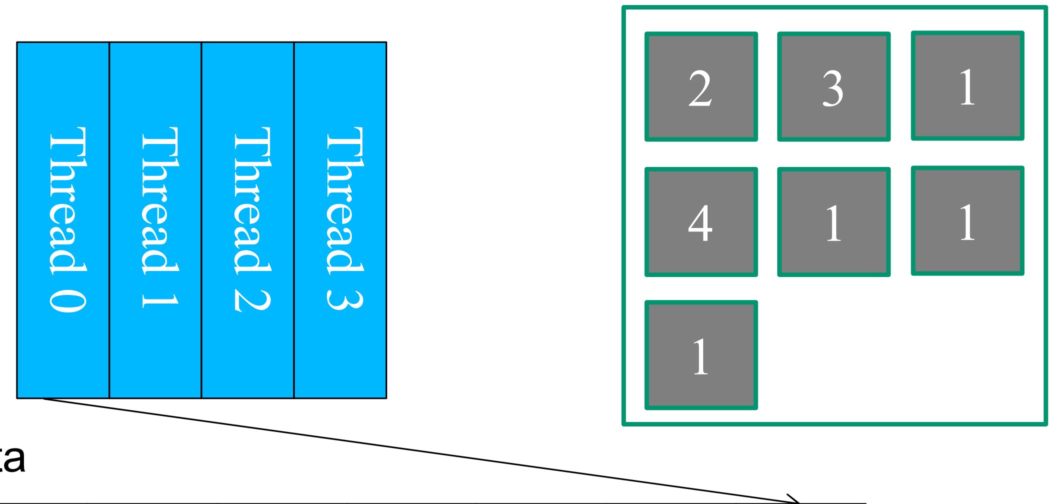


JDS with Transposition Memory Coalescing



Not aligned with DRAM bursts but OK with recent GPUS

JDS with Transposition Memory Coalescing



data

2	3	1	4	1	1	1

1	0	0	2	2	3	3

col_index

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;
       }
     }
```

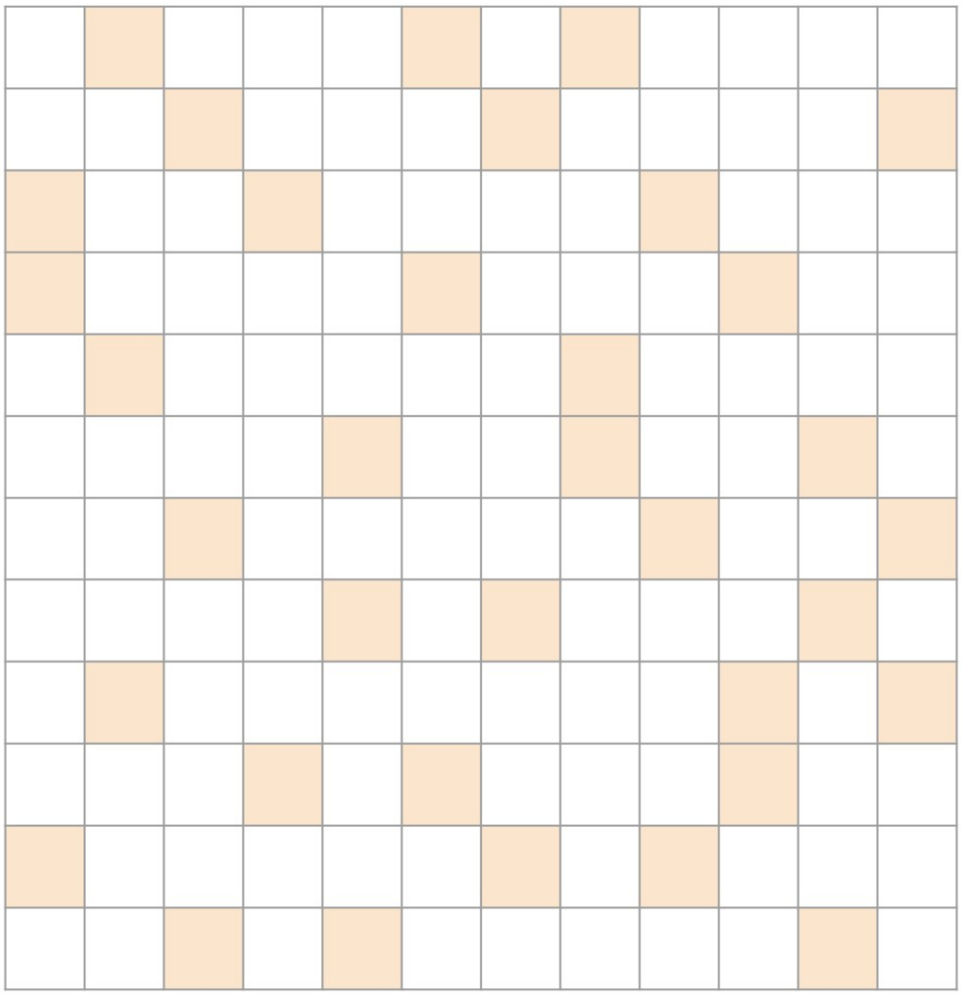
Column indices col_index[7] { 1, 0, 3, 2, 2, 3 3 }

JDS_T Column Pointers jds_t_col_ptr[5] {0, 3, 6, 7,7 }

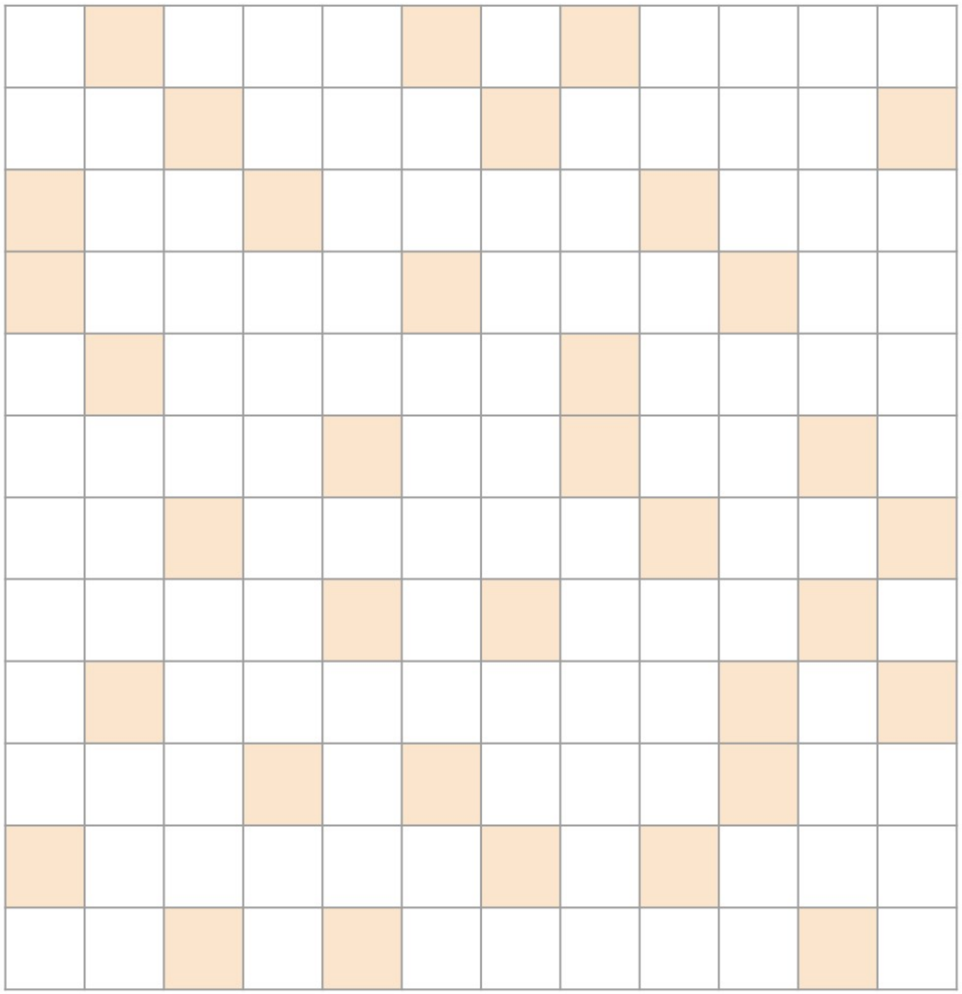
JDS Row Indices jds_row_perm[4] {2, 0, 3, 1 }

Lab 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 }



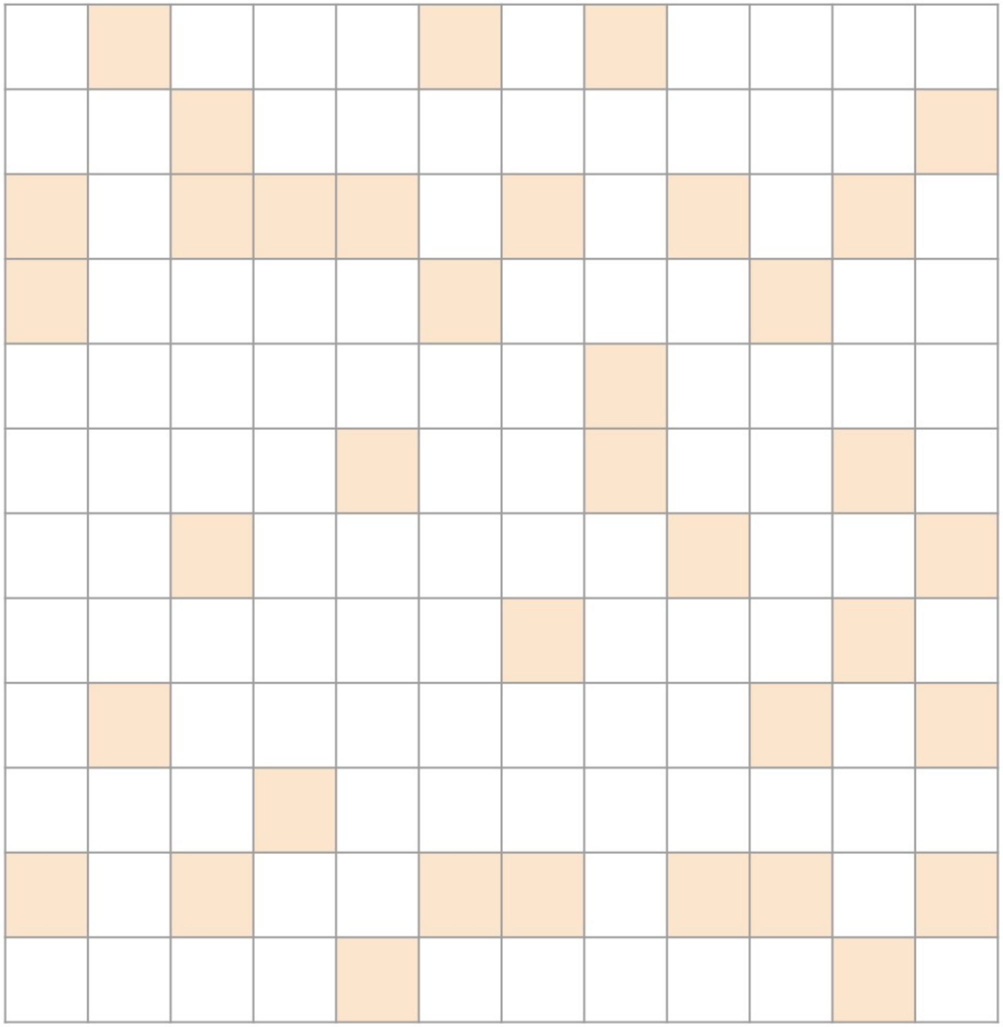
Roughly Random…



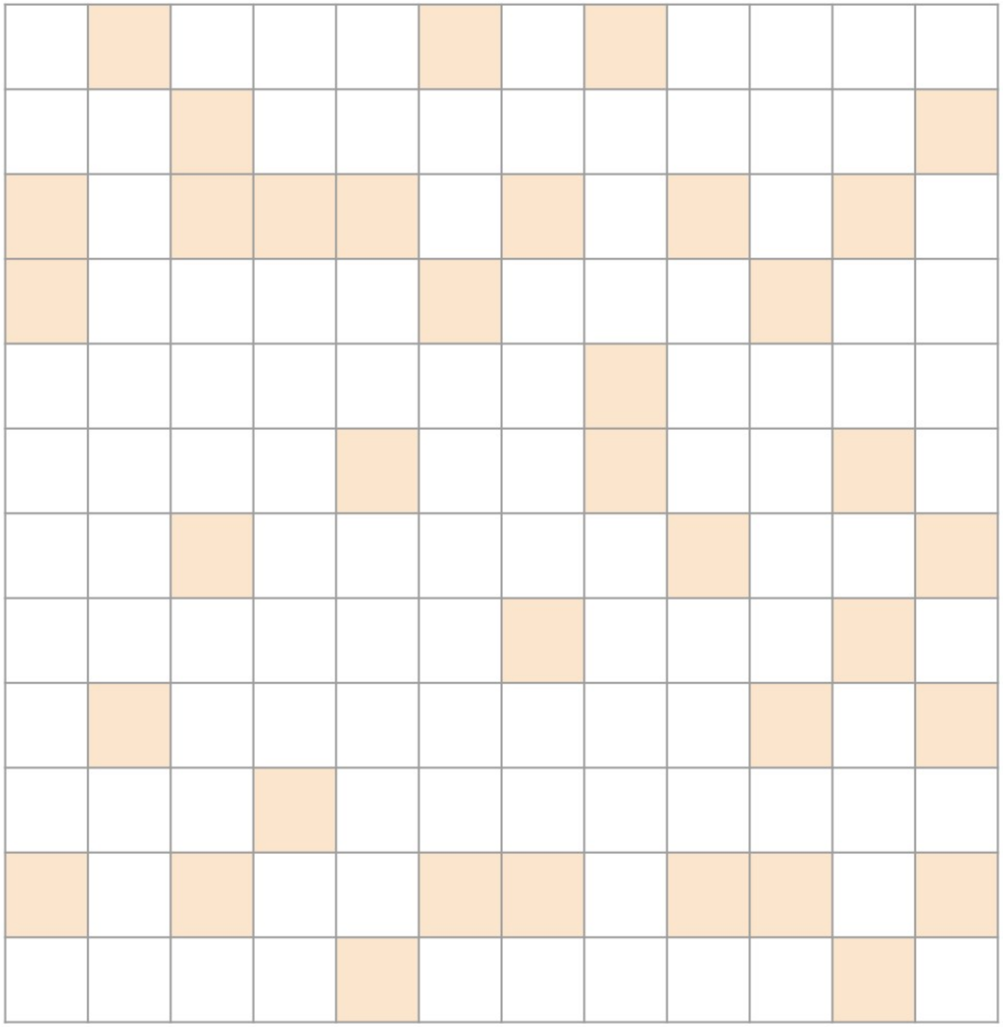
Roughly Random…

Probably best with ELL.

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



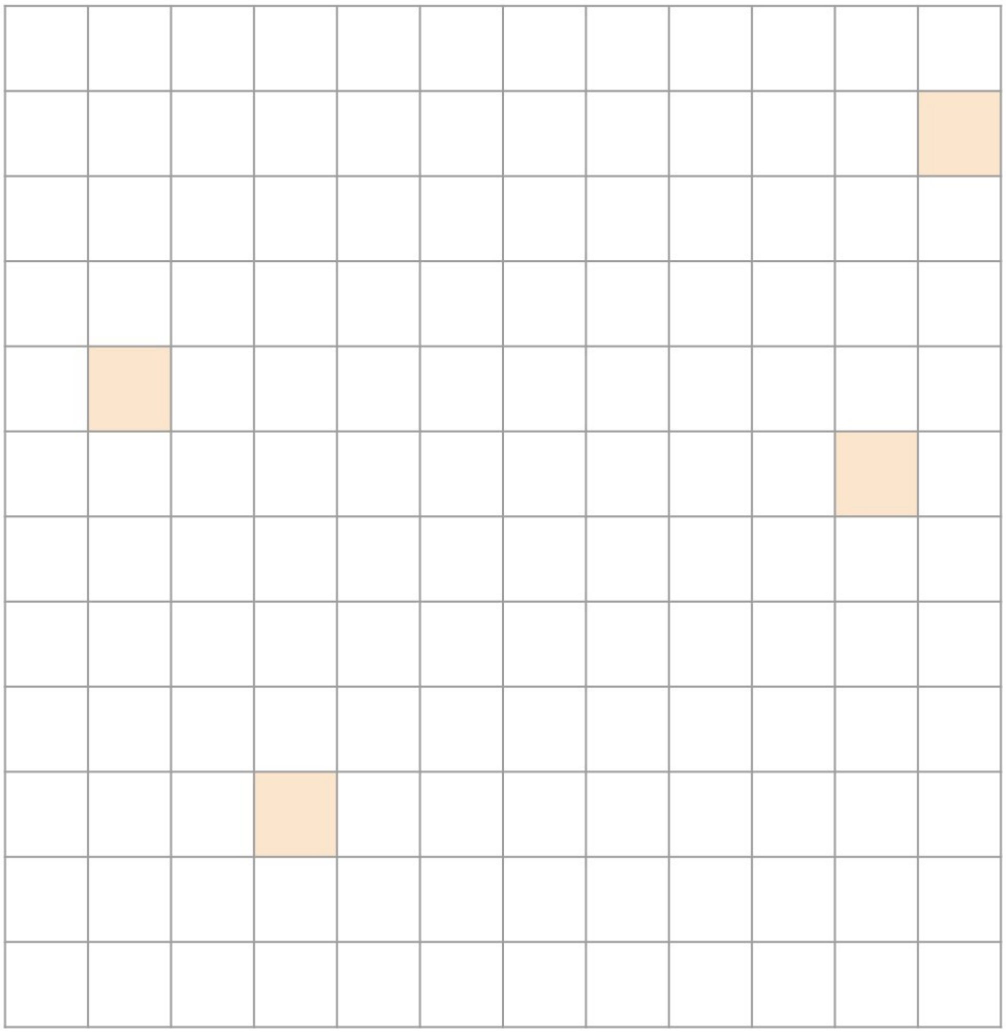
High variance in rows…



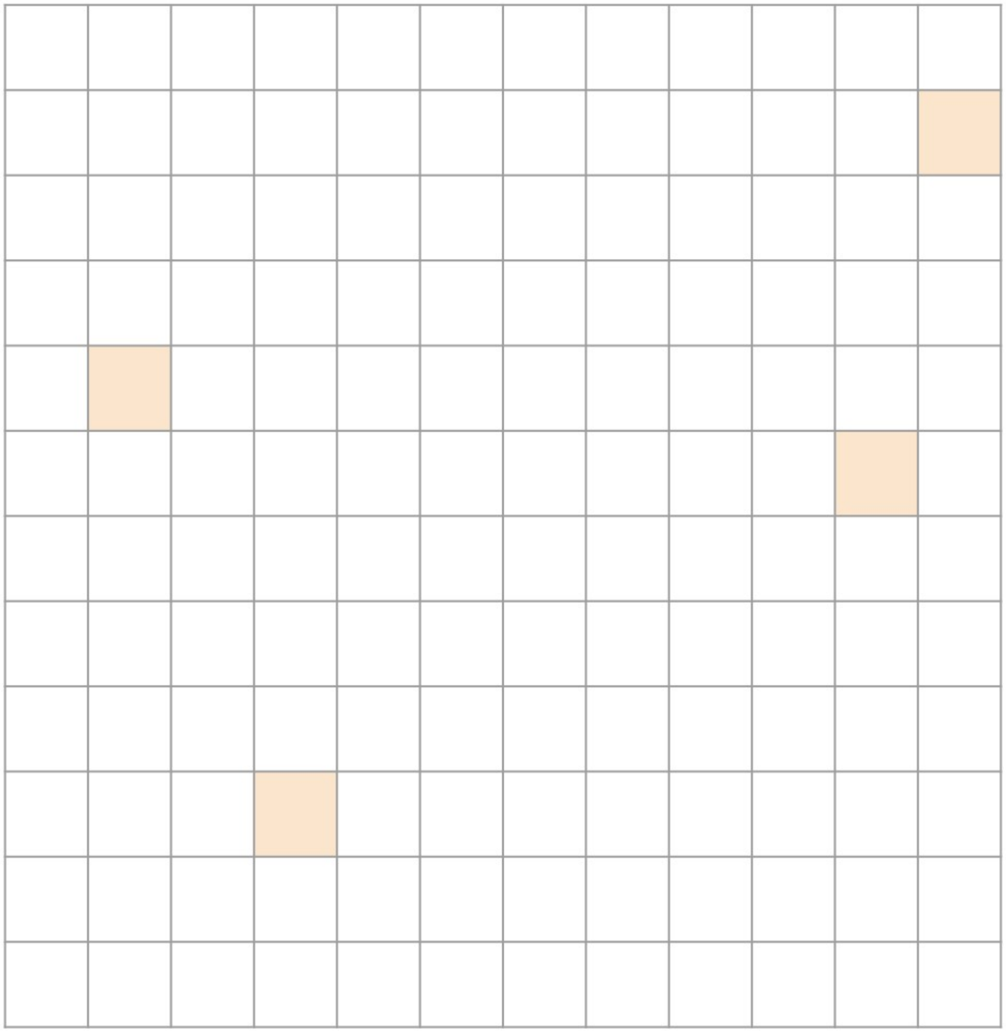
High variance in rows

Probably best with ELL/COO

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



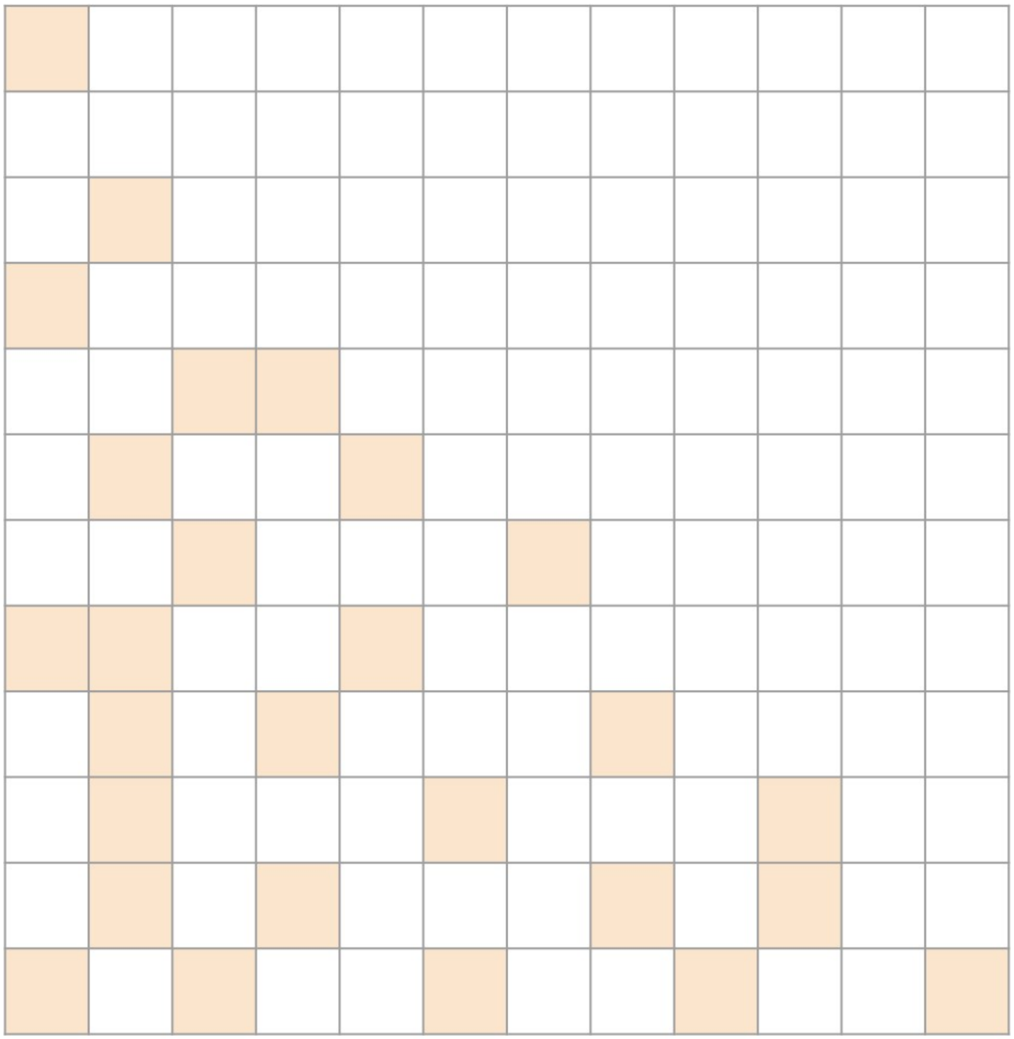
Very sparse…



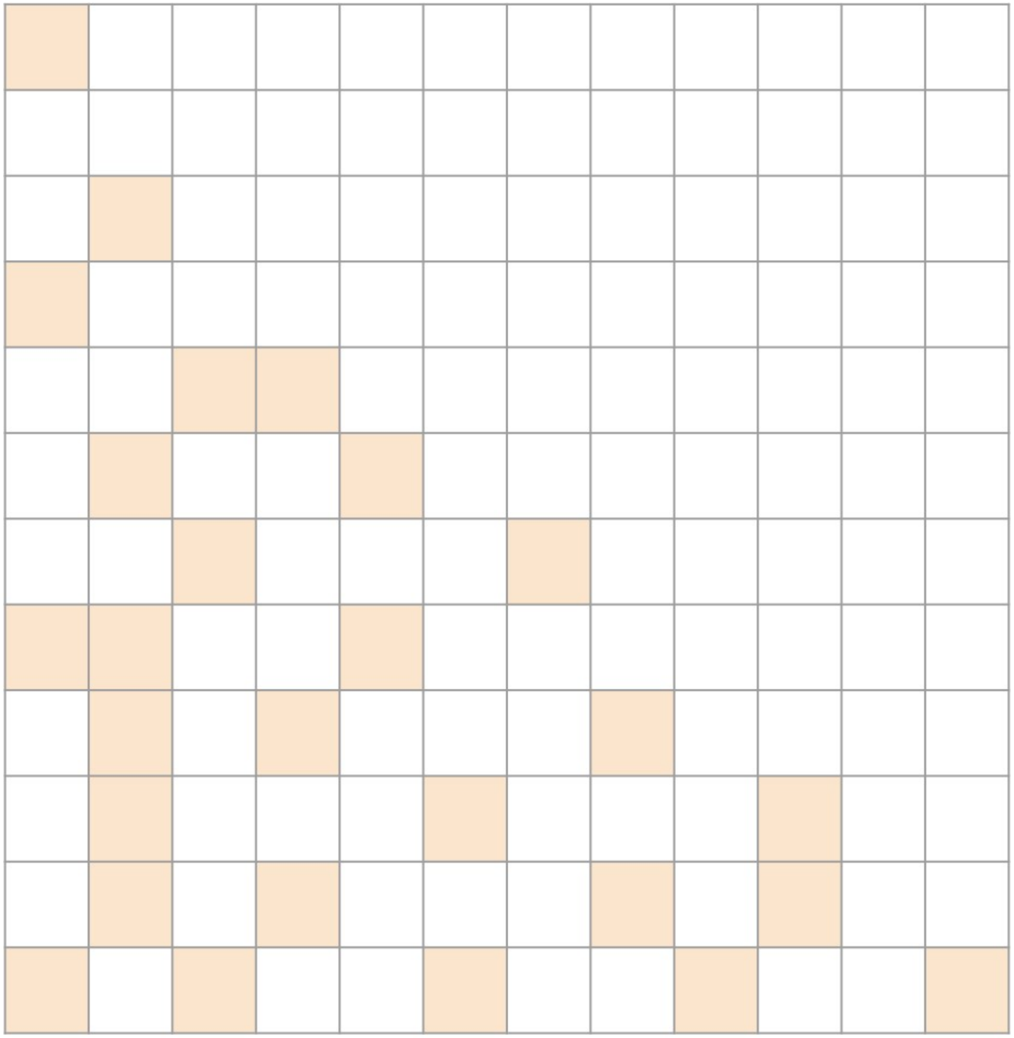
Very sparse

Probably best with COO

- Not a lot of data, compute is sparse



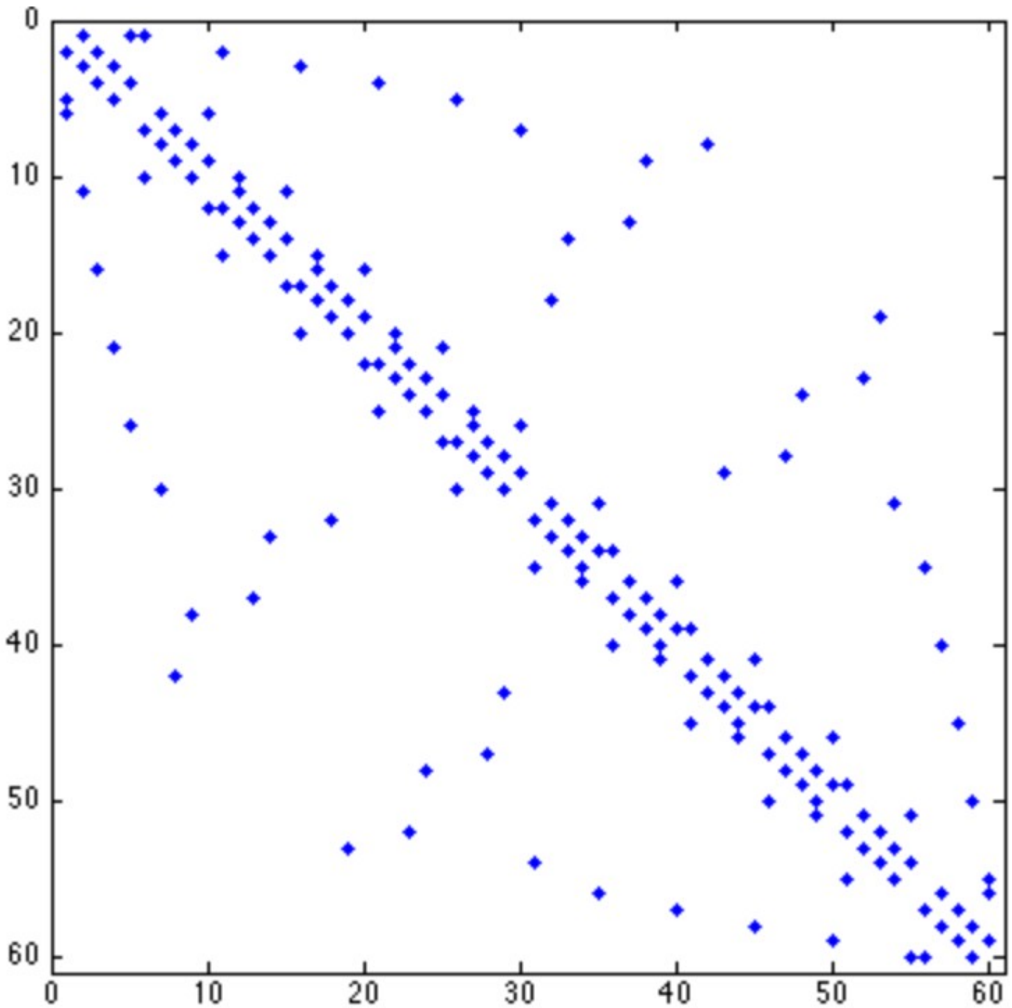
Roughly triangular…



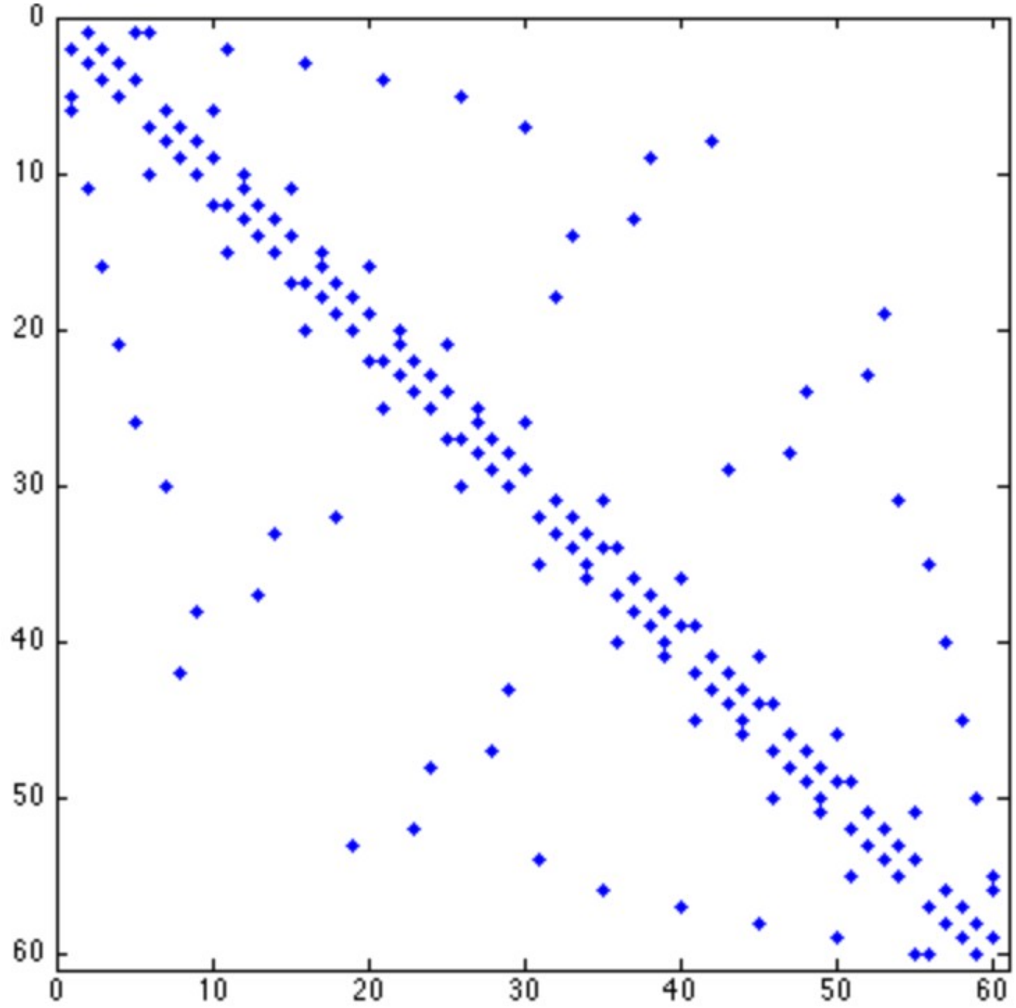
Roughly triangular…

Probably best with JDS

- Takes advantage of sparsity structure



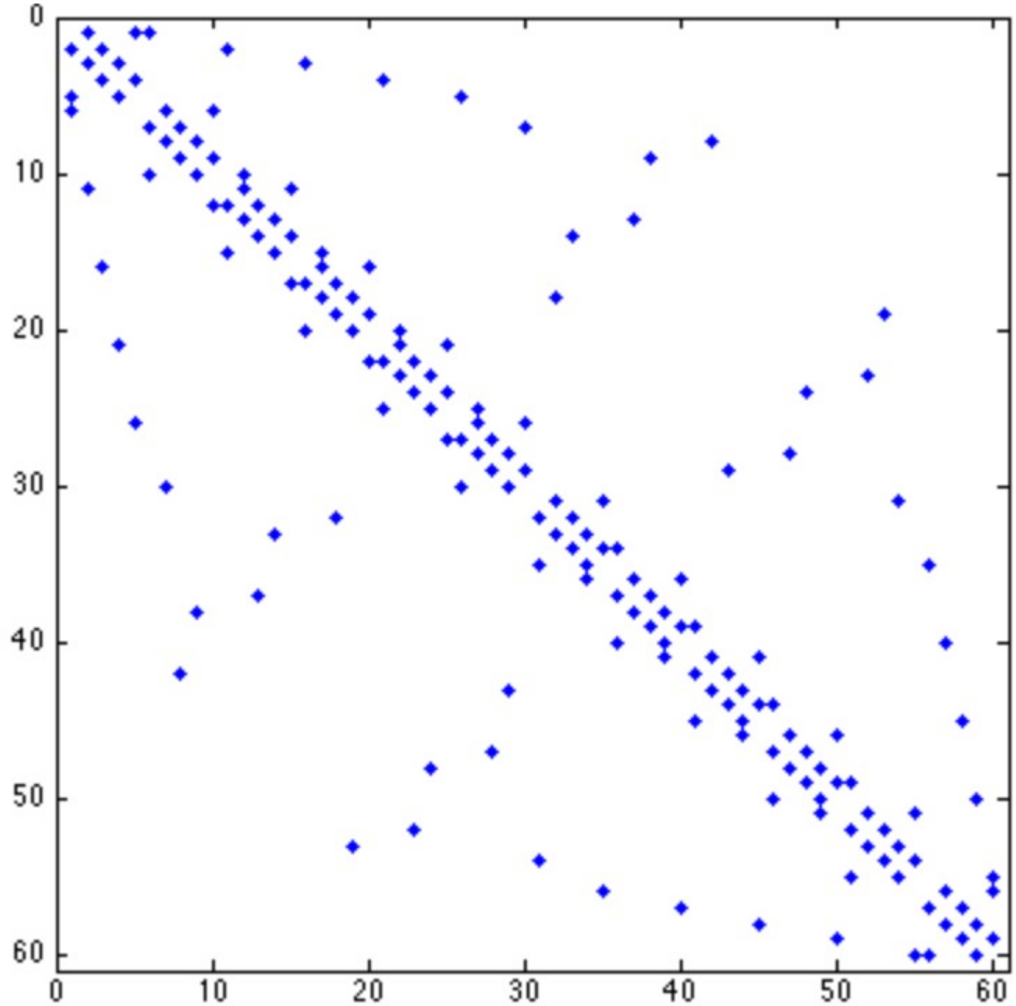
Banded Matrix…

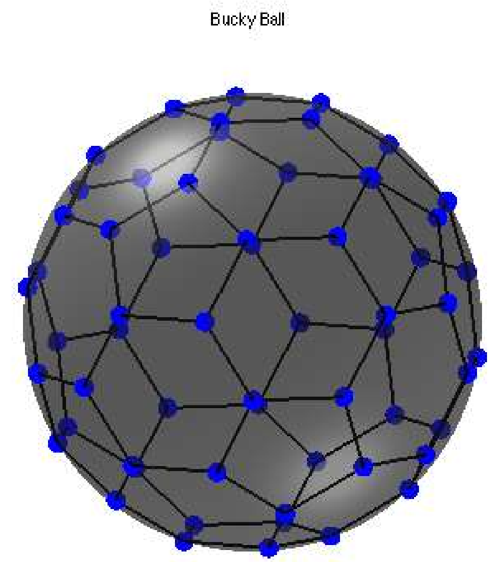


Banded Matrix…

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

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

QUESTIONS?

READ CHAPTER 10!

Problem Solving

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

- Q: Consider the following sparse Matrix:
- 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