Lecture 8: Parallel Sparse Methods

Objective

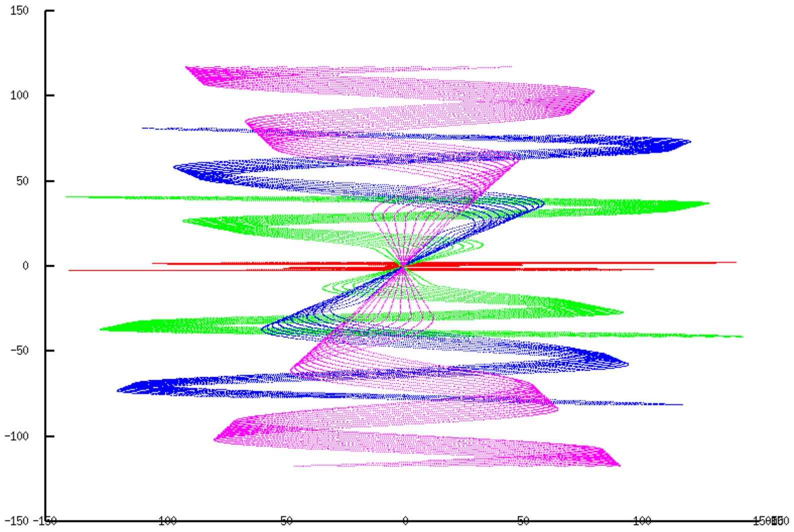
- To learn to regularize irregular data with
  - Limiting variations with clamping
  - Sorting
  - Transposition
- To learn to write a high-performance SpMV kernel based on JDS transposed format
- To learn the key techniques for compacting input data in parallel sparse methods for reduced consumption 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

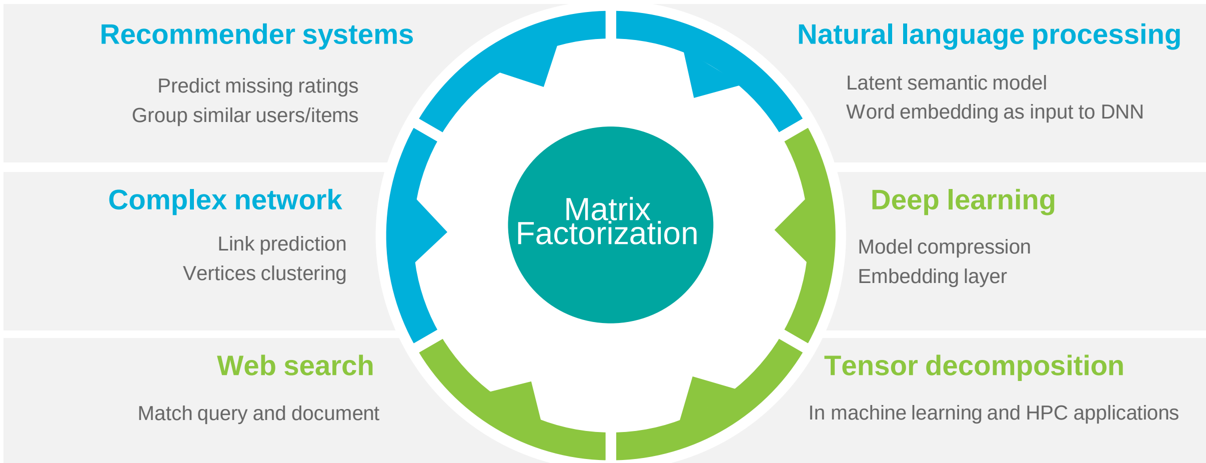
- Many real-world systems are sparse in nature
  - Linear systems described as sparse matrices
- Solving sparse linear systems
  - Traditional inversion algorithms such as Gaussian elimination can create too many "fill-in" elements and explode the size of the matrix
  - Iterative Conjugate Gradient solvers based on sparse matrix-vector multiplication is preferred
- Solution of PDE systems can be formulated into linear operations expressed as sparse matrix-vector multiplication

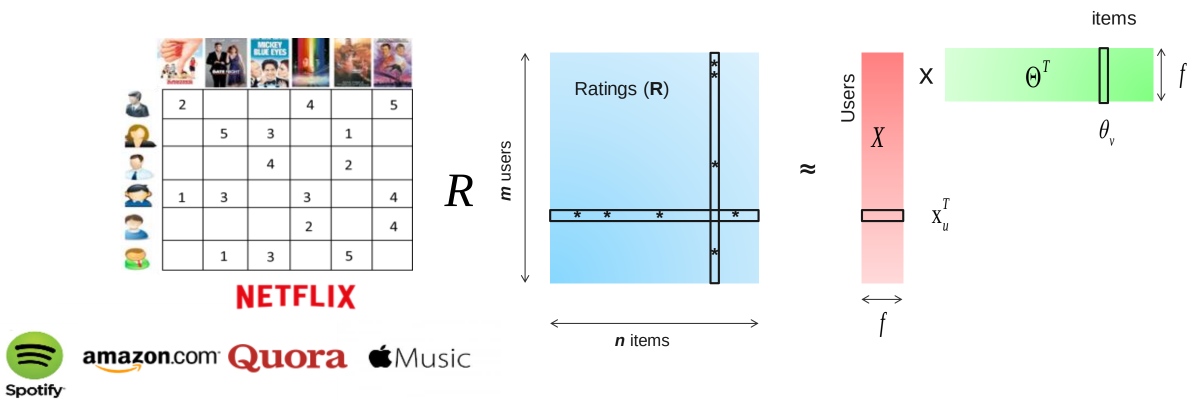
Sparse Data
Motivation for Compaction

- Many real-world inputs are sparse/non-uniform
- Signal samples, mesh models, transportation networks, communication networks, etc.



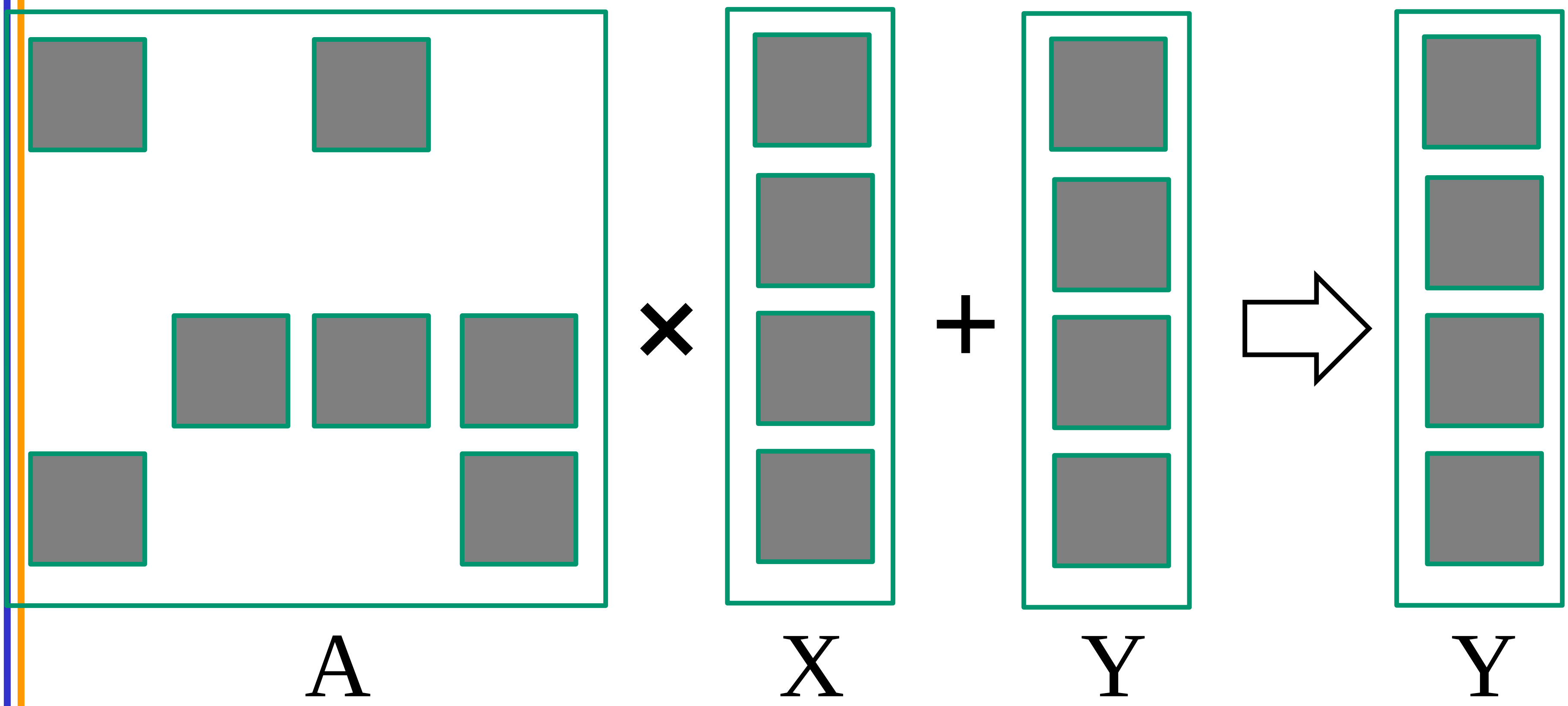
Sparse Matrix in Analytics and AI





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-Vector Multiplication (SpMV)



Challenges

- Compared to dense matrix multiplication, SpMV
  - Is irregular/unstructured
  - Has little input data reuse

- 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

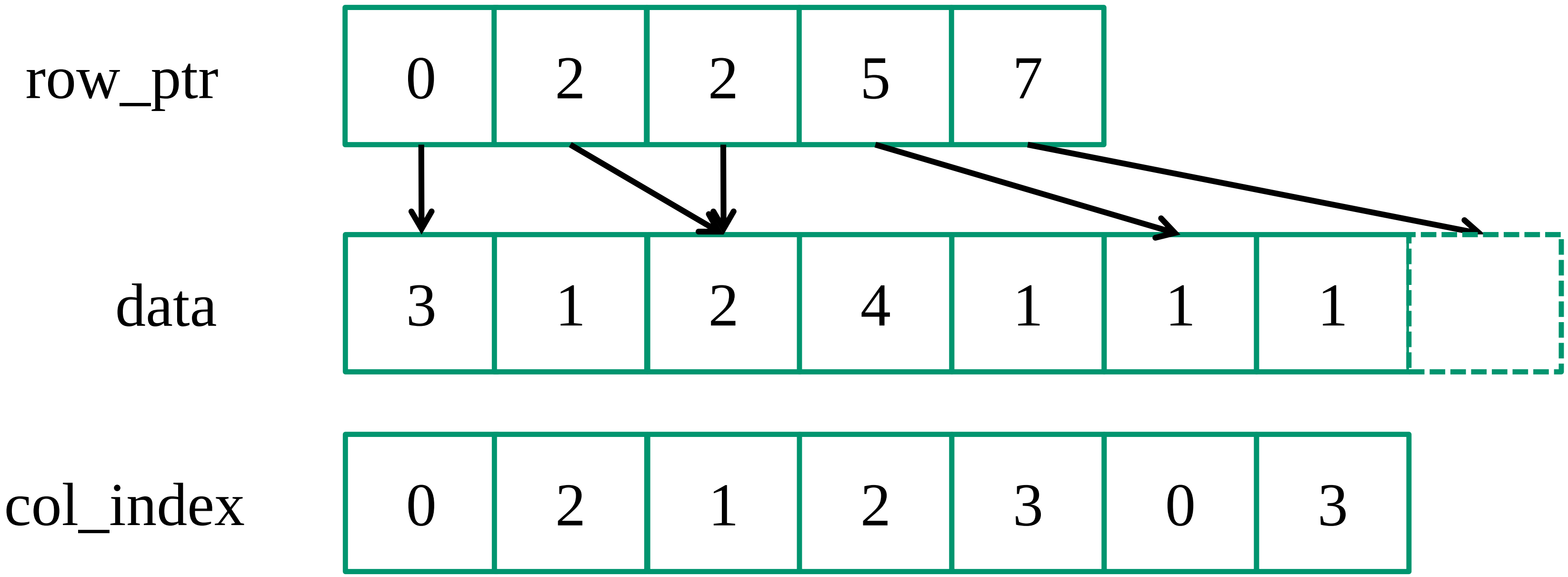
CSR Representation

			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	ptr[5]	{	0,	2,	2,	5,	7	}		

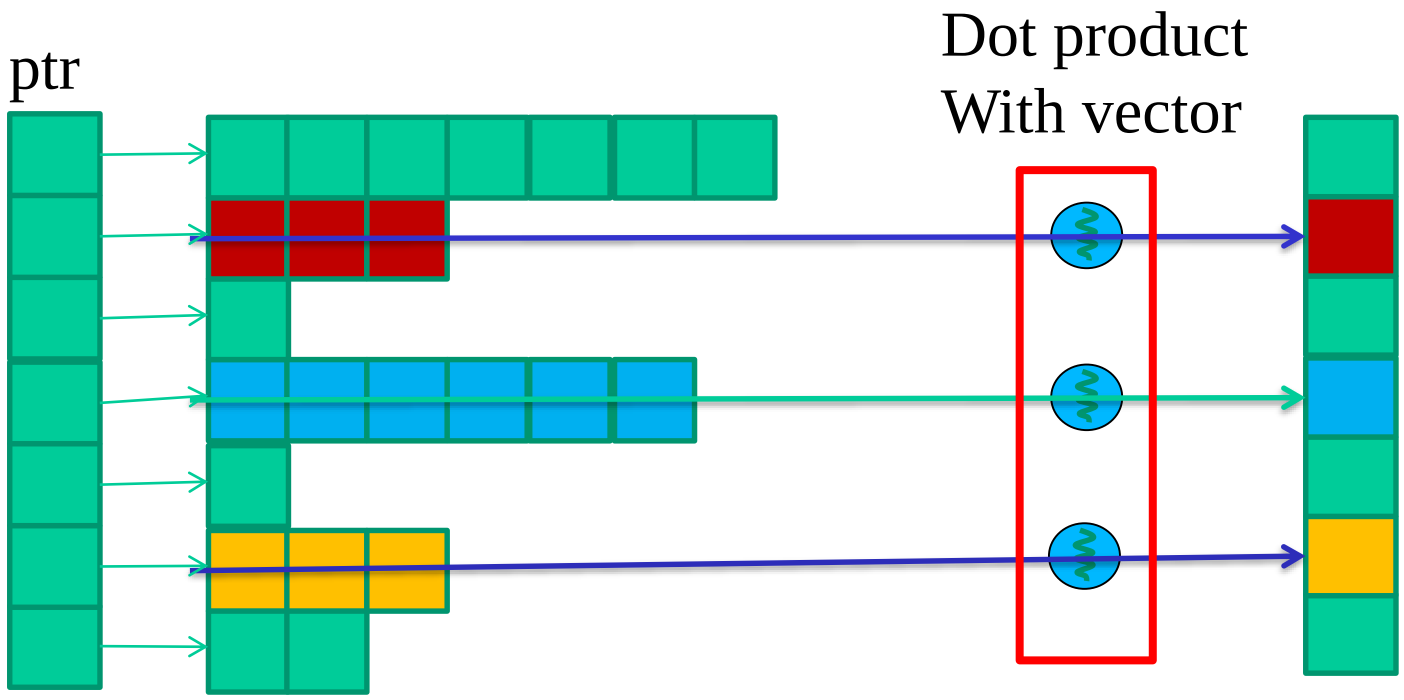
Dense representation

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

CSR Data Layout



CSR Kernel Design



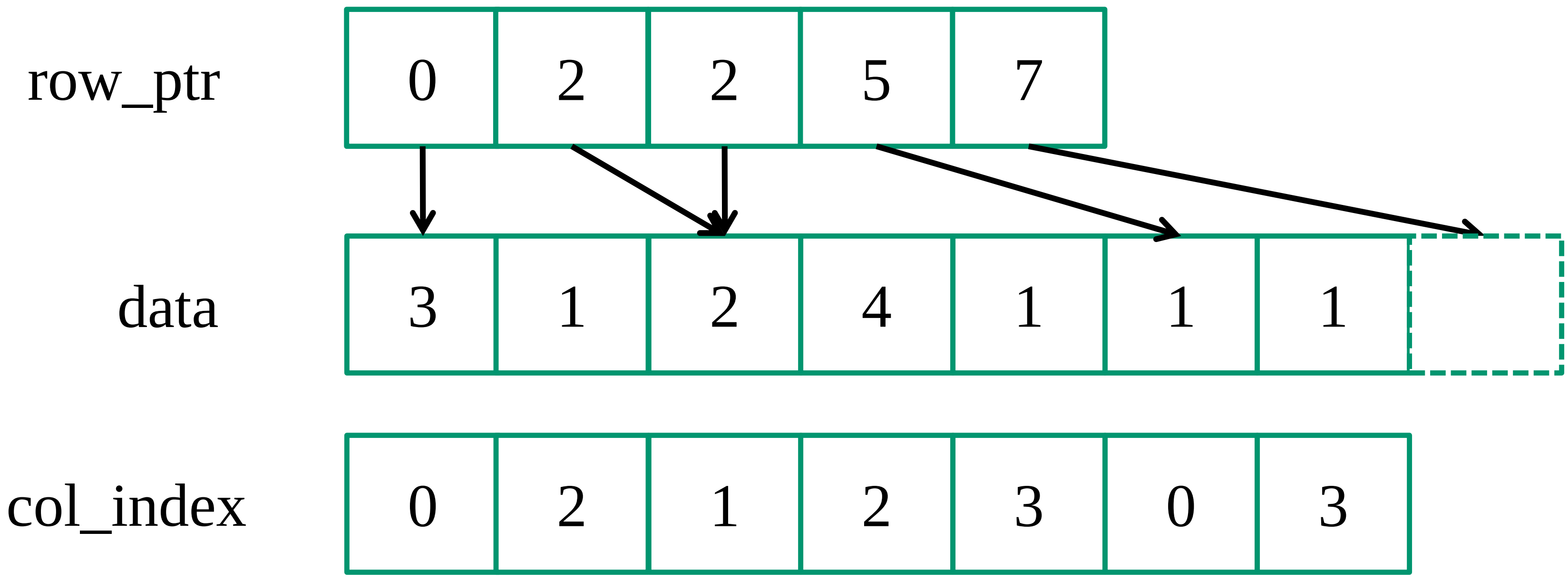
A Parallel SpMV/CSR Kernel (CUDA)

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

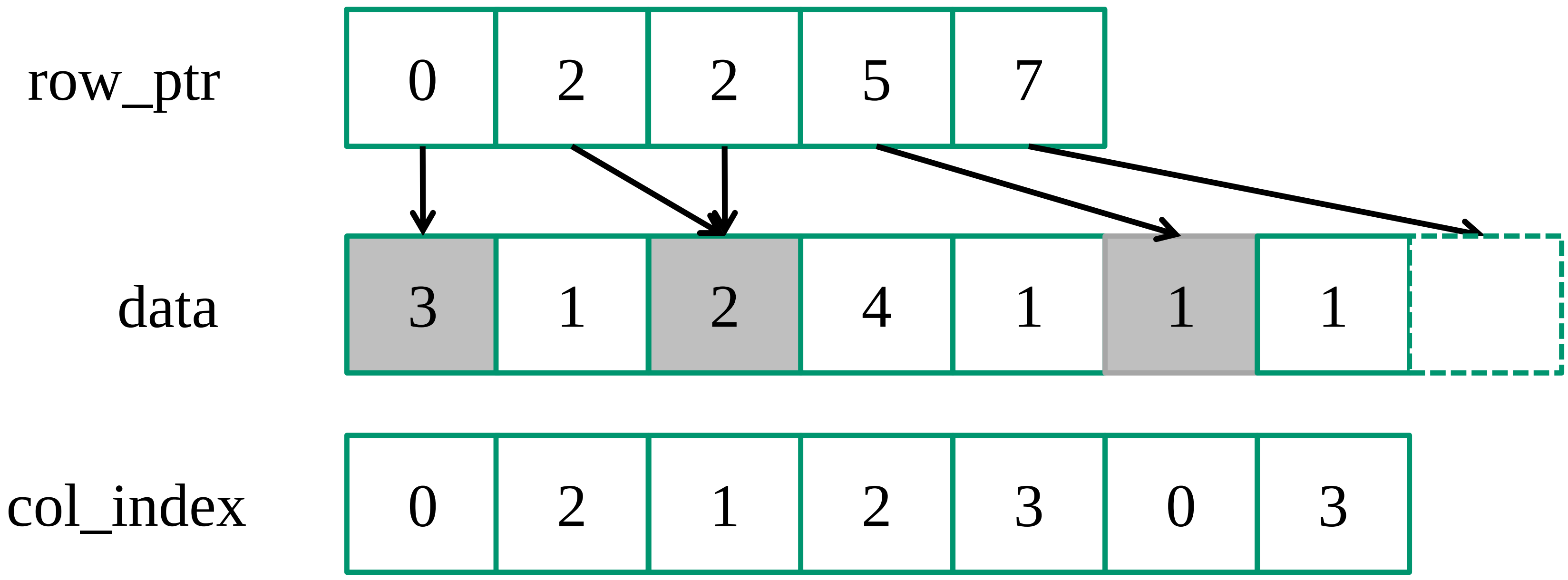
CSR Kernel Control Divergence

- Threads execute different number of iterations in the kernel for-loop

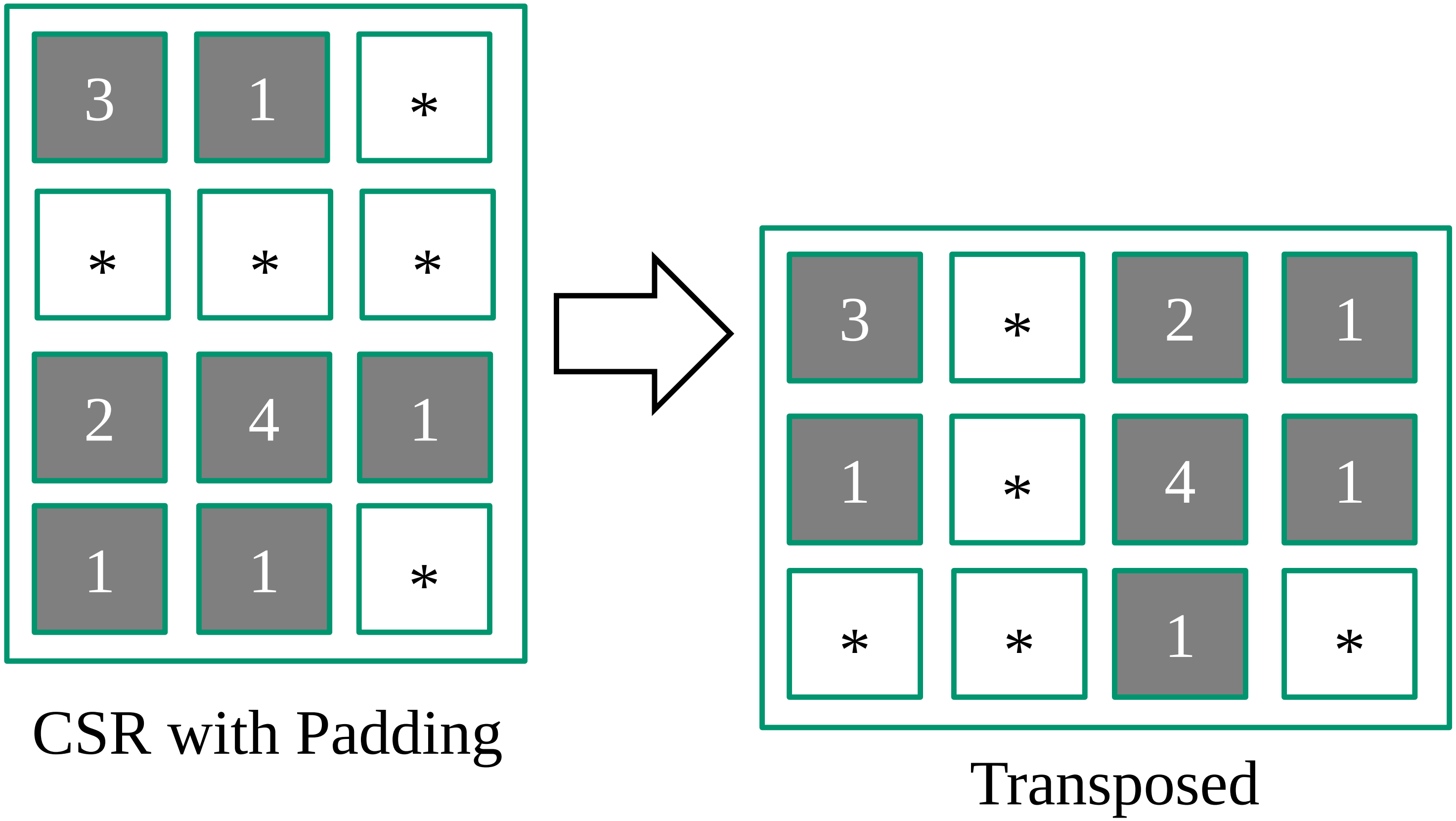


CSR Kernel Memory Divergence (Uncoalesced Accesses)

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

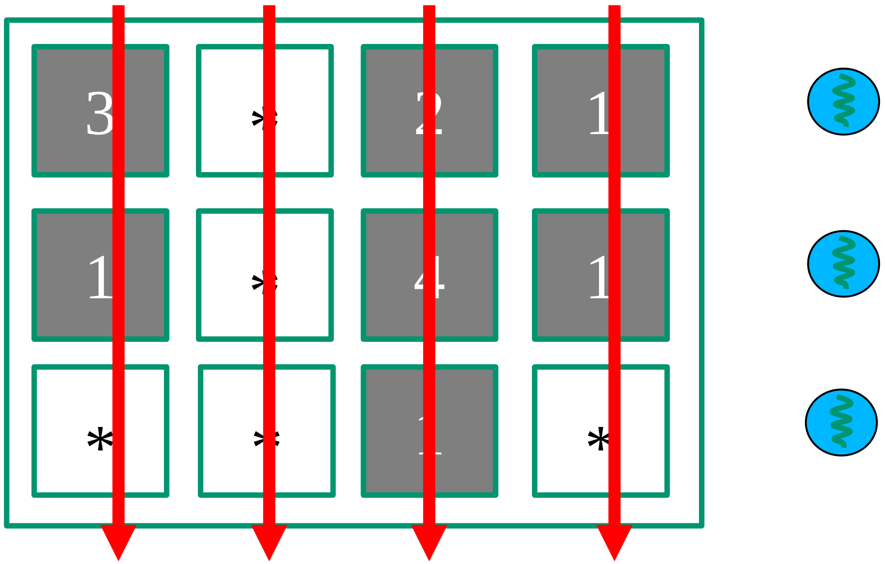


Regularizing SpMV with ELL(PACK) Format



- 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

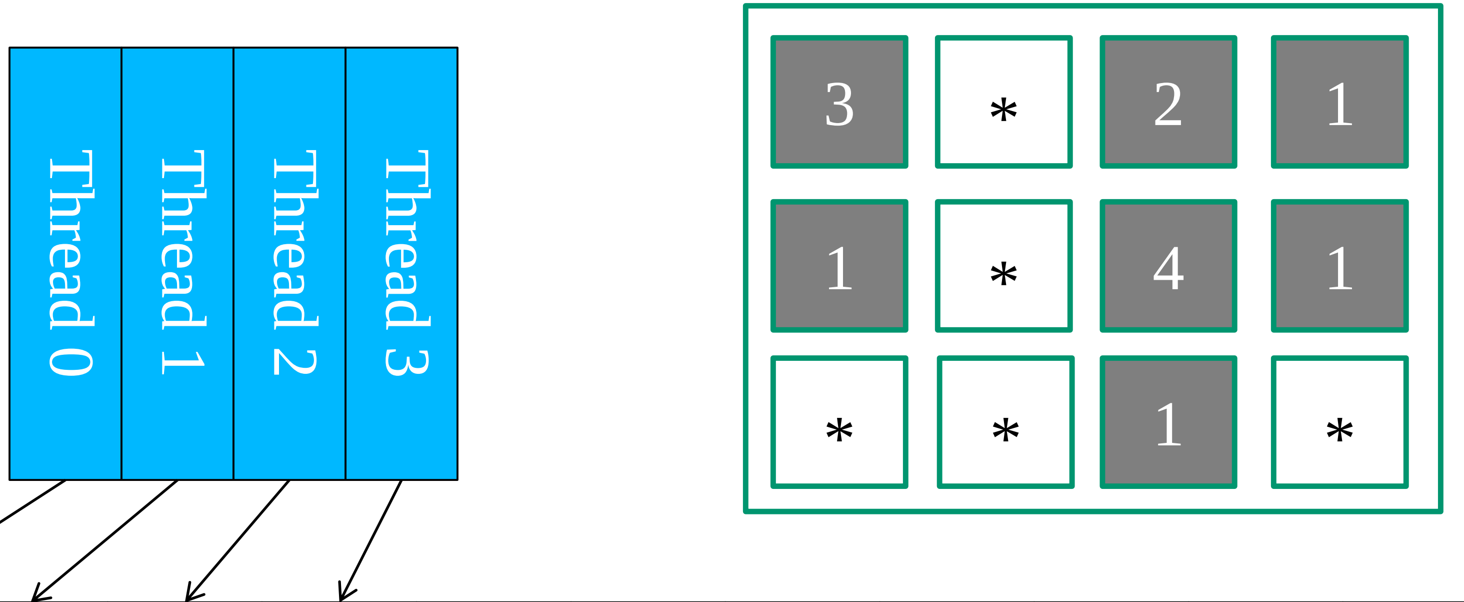


A parallel SpMV/ELL kernel

```
__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



data

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

0	0	1	0	2	1	2	3	3	2	3	1

col_index

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	}

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	}

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

a sequential loop that implements SpMV/COO

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	}

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 }

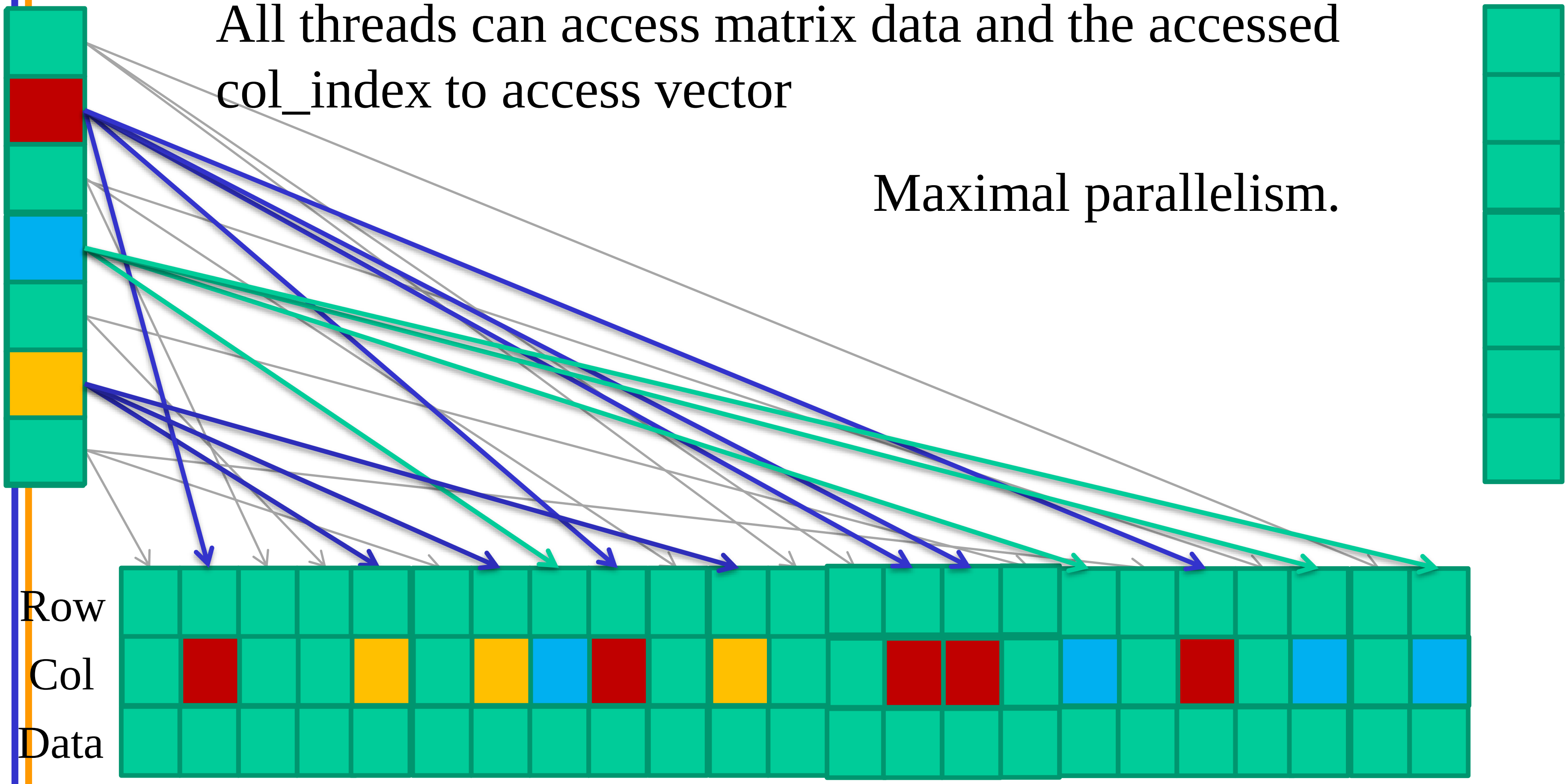
```
for (int i = 0; i < num_elem; row++)
  y[row_index[i]] += data[i] * x[col_index[i]];
```

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

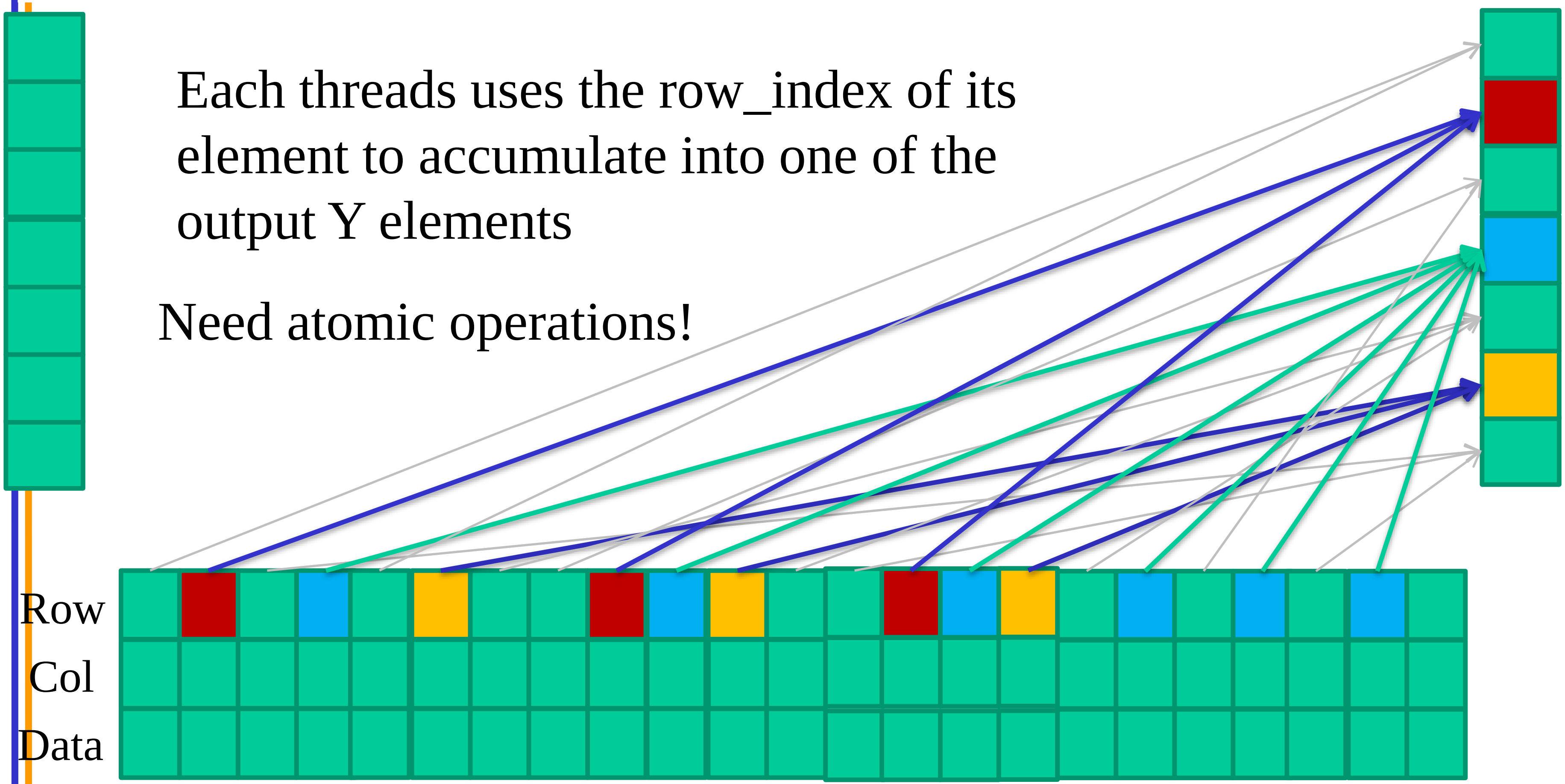
Maximal parallelism.



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

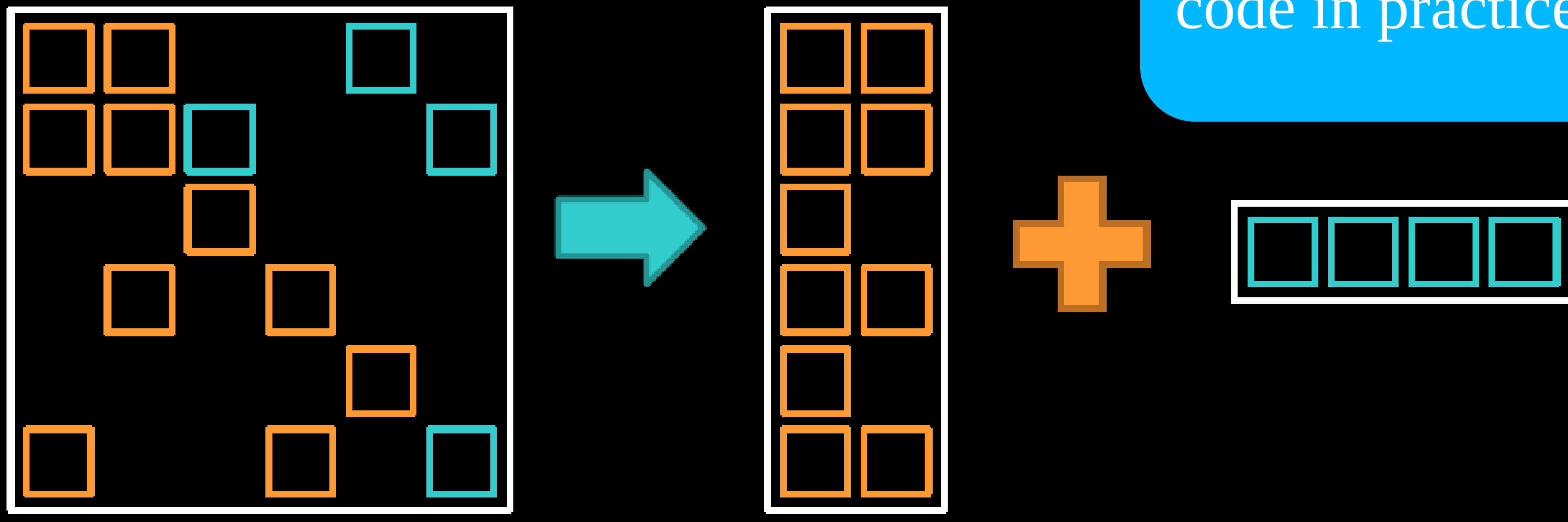
Need atomic operations!



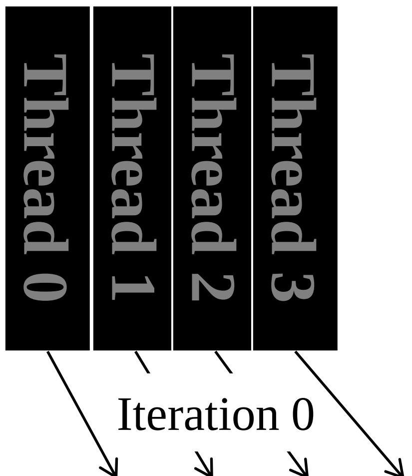
Hybrid Format

- ELL 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

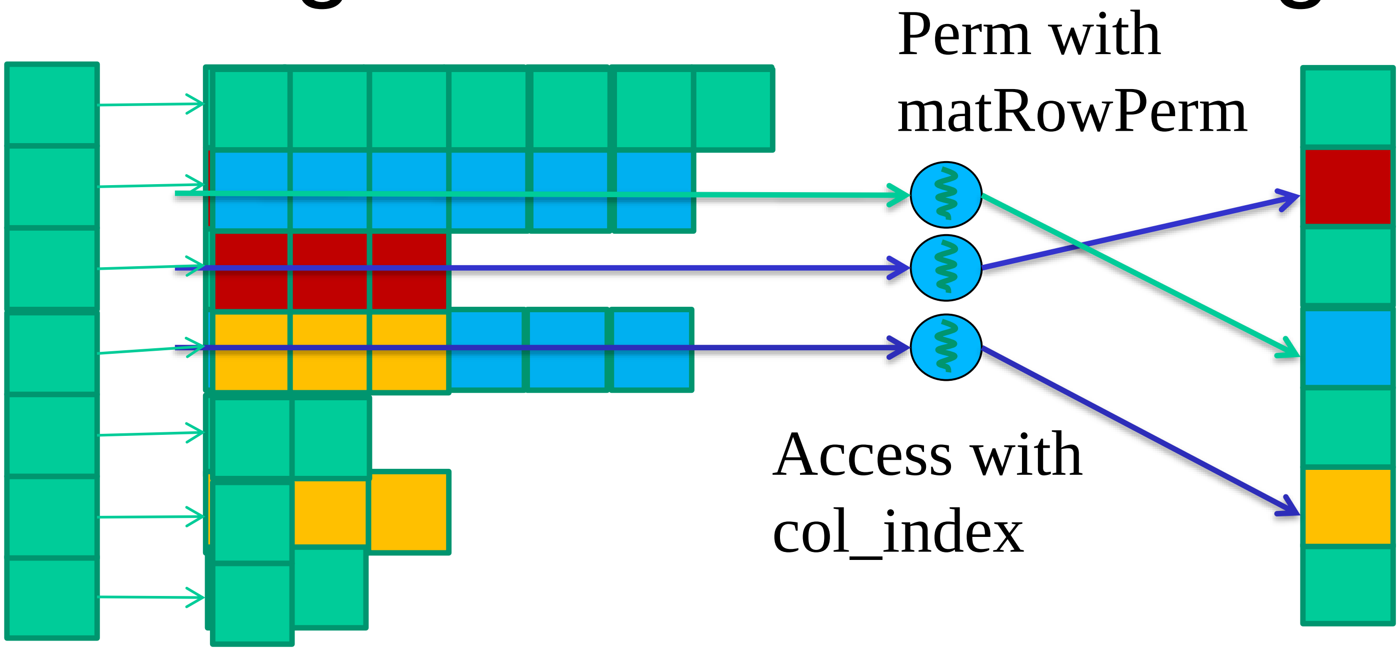
data	3	*	2	1	1	*	4	1
index	0	*	1	0	2	*	2	3

ELL

data	1
col_index	3
row_index	2

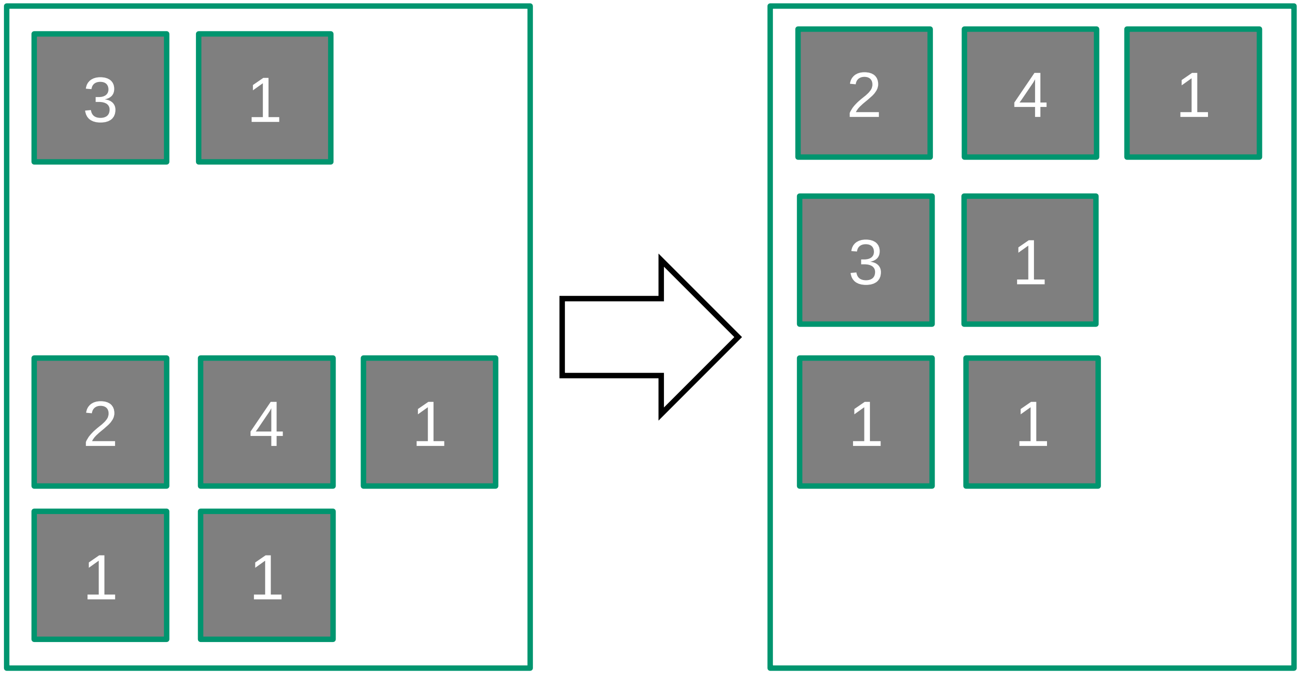
COO

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 JDS

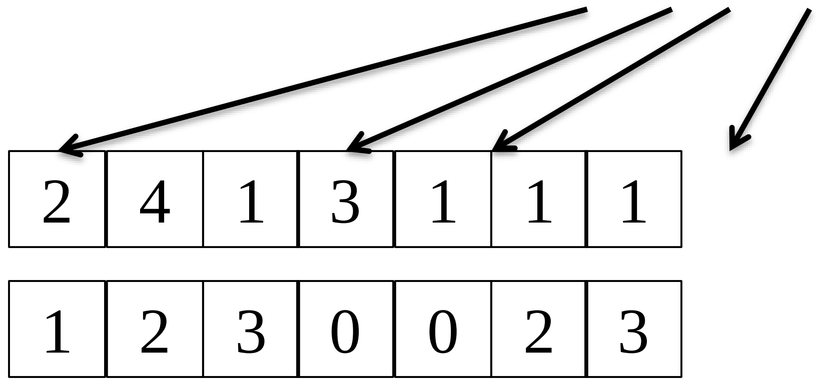
CSR to JDS Conversion

		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 }



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

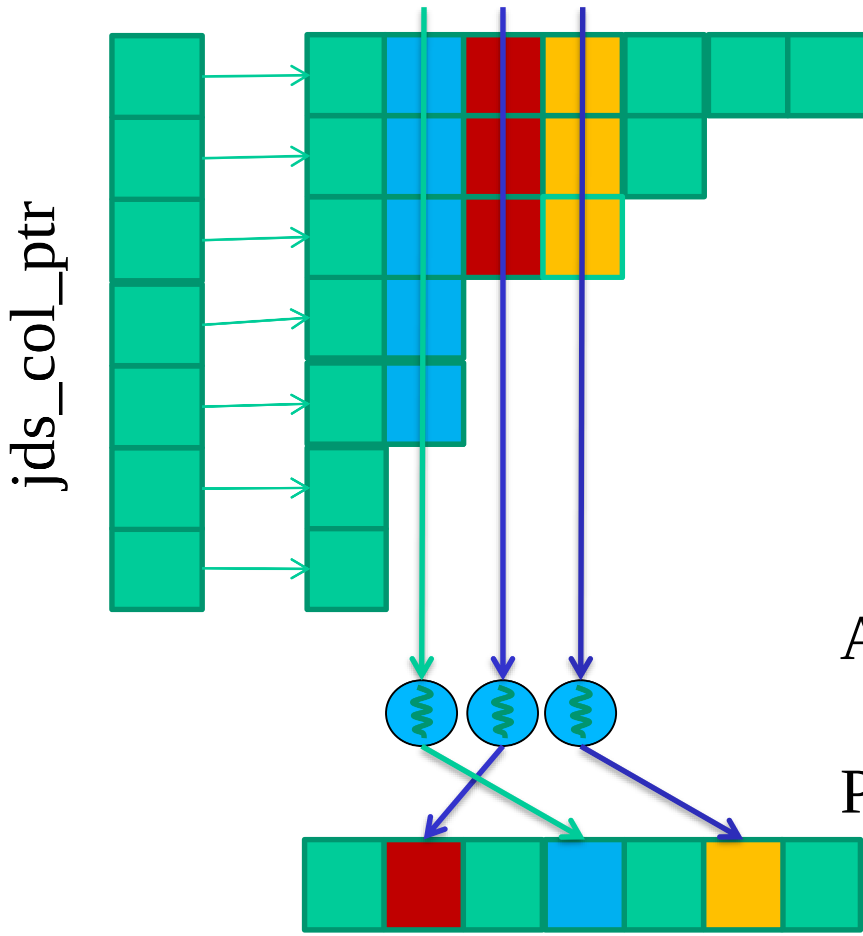
JDS vs. CSR Memory Divergence

- 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 }

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

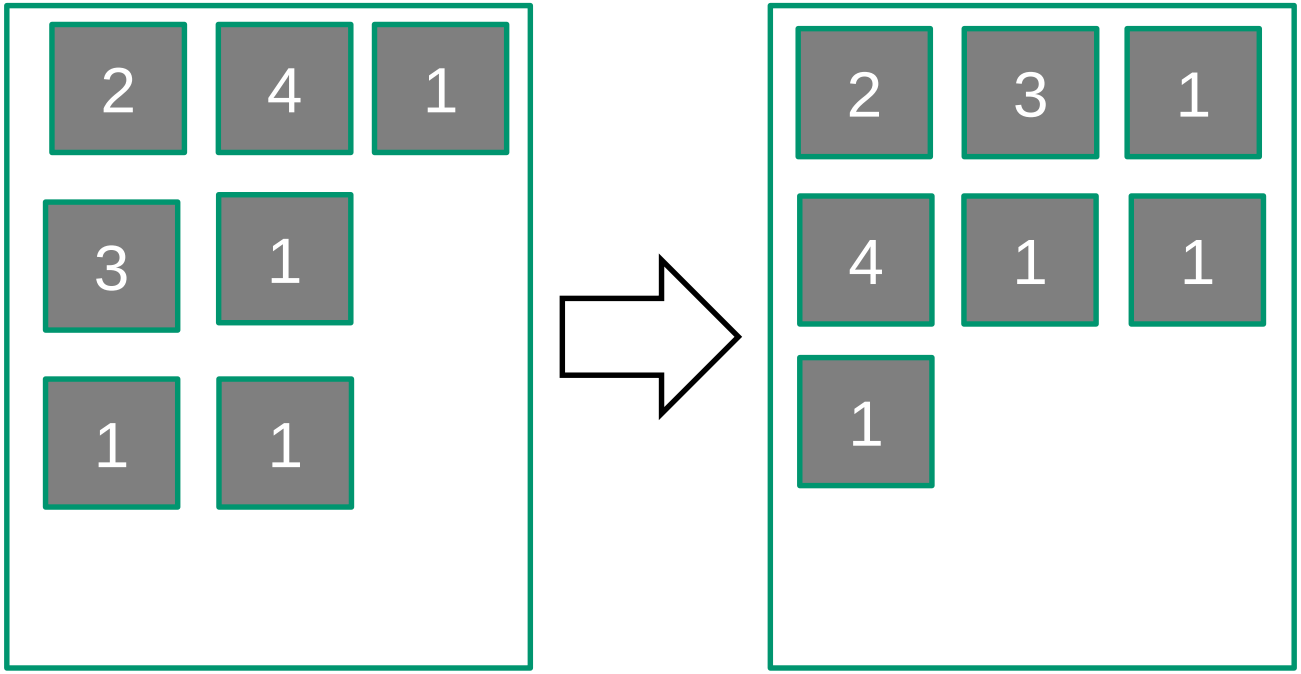
JDS with Trasposition



Access with col_index

Perm with jds_row_index

Transposition for Memory Coalescing



JDS

JDS-transposed

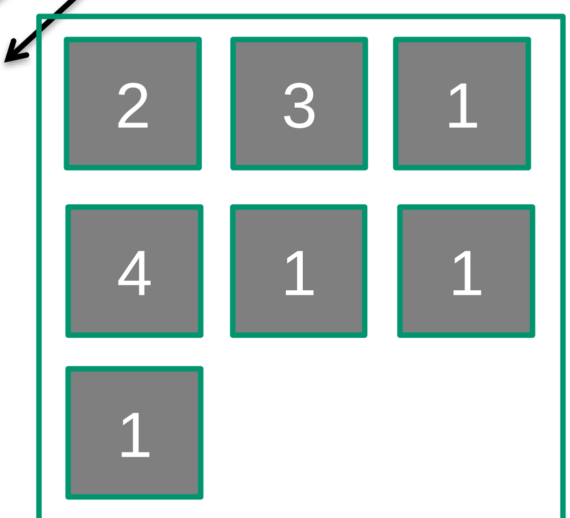
JDS Format with Transposed Layout

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

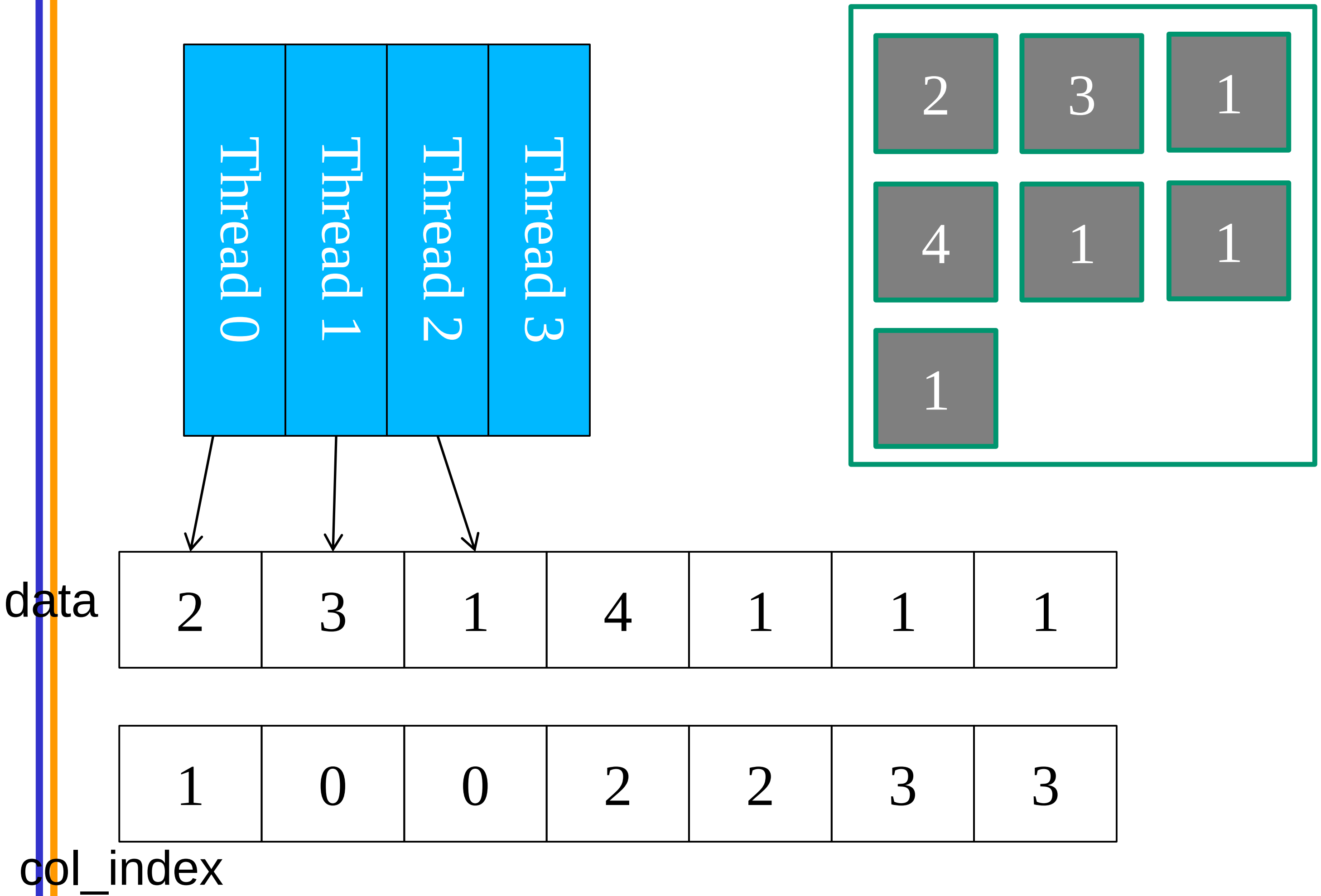
JDS row indices Jds_row_perm[4] { 2, 0, 3, 1 }

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

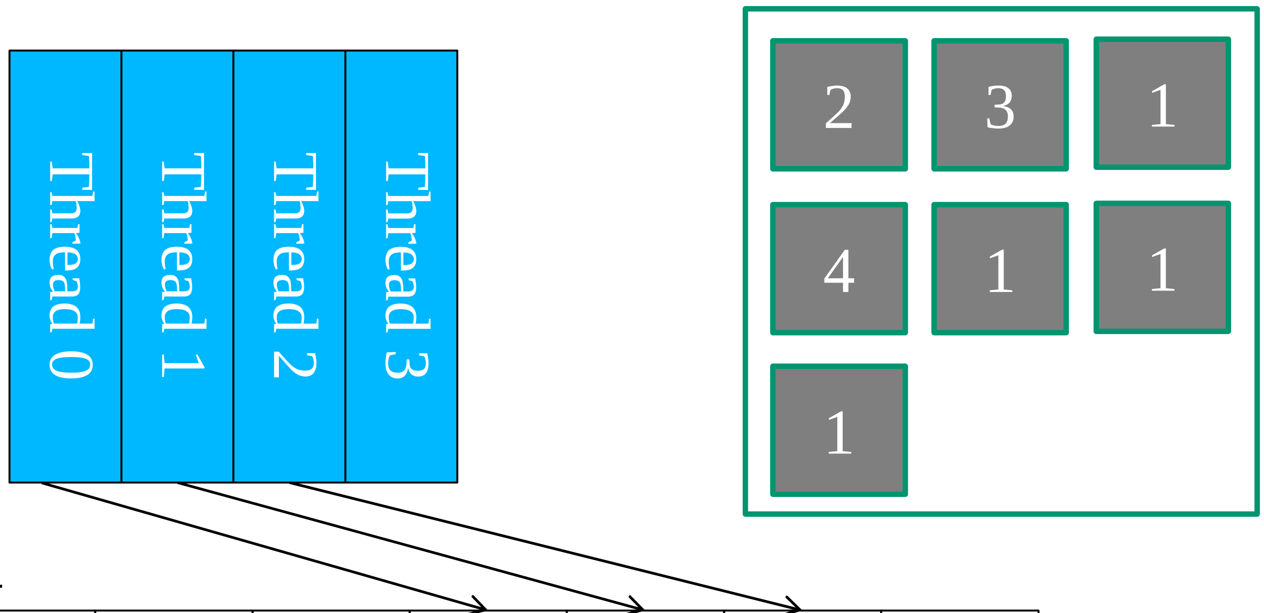
data	2	3	1	4	1	1	1
col_index	1	0	0	2	2	3	3



JDS with Transposition Memory Coalescing



JDS with Transposition Memory Coalescing



data

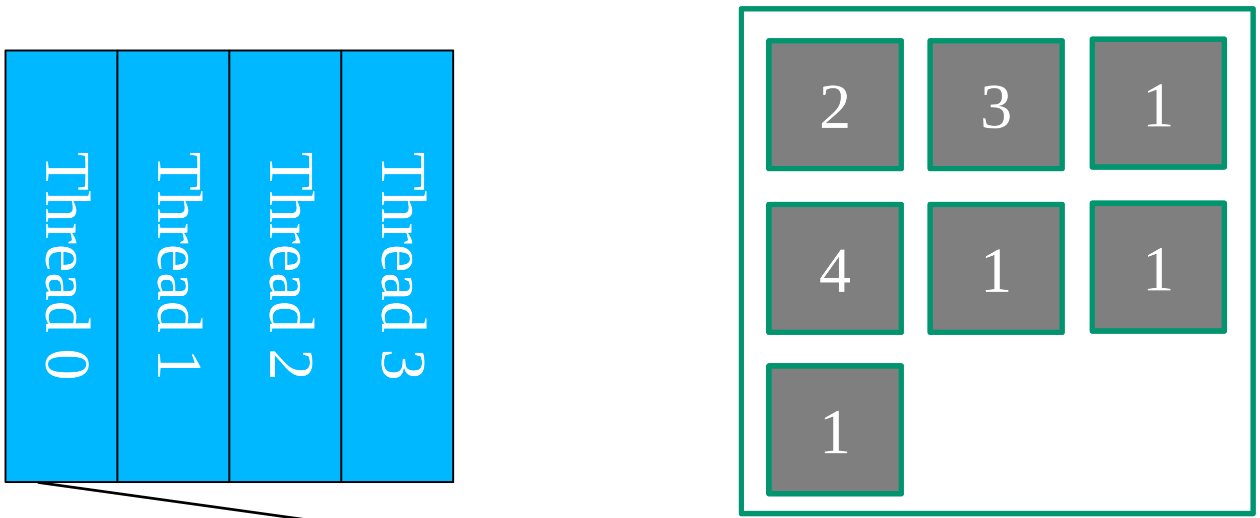
2	3	1	4	1	1	1

1	0	0	2	2	3	3

col_index

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 in 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}

MP7 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 }

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

ANY QUESTIONS?