Parallel Patterns: Sparse Matrix Multiplication

Prof. Seokin Hong

Sparse Matrix

- In a sparse matrix, the majority of the elements are zeros
- Many important real-word problems involve sparse matrix computation
  - Ex) graph (e.g., representation of social network), Neural Network
- Storing and processing zero elements are wasteful
  - Several sparse matrix storage formats and processing methods are proposed for efficient sparse matrix computation

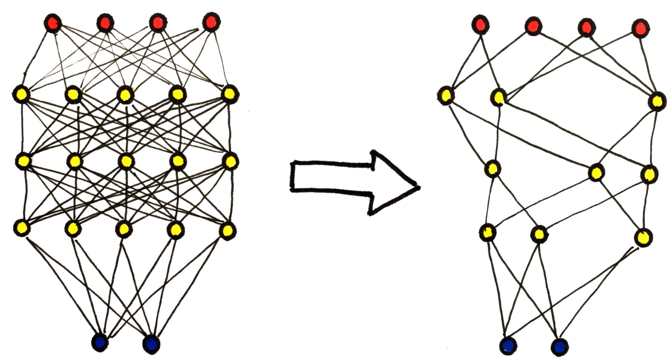
Dense Matrix

1	2	31	2	9	7	34	22	11	5
11	92	4	3	2	2	3	3	2	1
3	9	13	8	21	17	4	2	1	4
8	32	1	2	34	18	7	78	10	7
9	22	3	9	8	71	12	22	17	3
13	21	21	9	2	47	1	81	21	9
21	12	53	12	91	24	81	8	91	2
61	8	33	82	19	87	16	3	1	55
54	4	78	24	18	11	4	2	99	5
13	22	32	42	9	15	9	22	1	21

Sparse Matrix

1	.	3	.	9	.	3	.	.	.
11	.	4	.	.	.	.	.	2	1
.	.	1	.	.	.	4	.	1	.
8	.	.	.	3	1	.	.	.	.
.	.	.	9	.	.	1	.	17	.
13	21	.	9	2	47	1	81	21	9
.	.	.	.	.	.	.	.	.	.
.	.	.	.	19	8	16	.	.	55
54	4	.	.	.	11	.	.	.	.
.	.	2	.	.	.	.	22	.	21

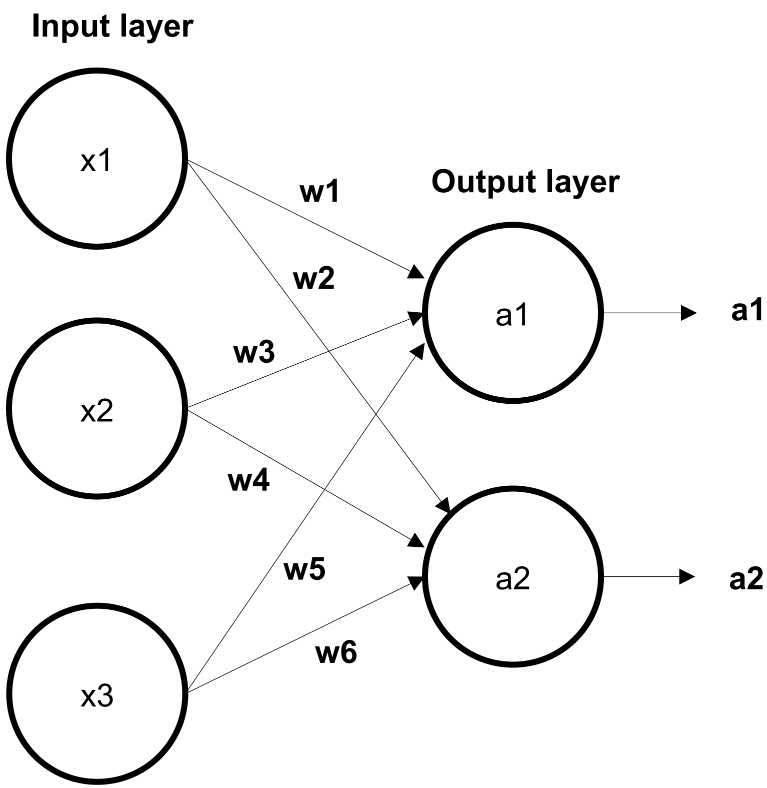
https://cmdlinetips.com/2018/03/sparse-matrices-in-python-with-scipy/



https://towardsdatascience.com/can-you-remove-99-of-a-neural-network-without-losing-accuracy-915b1fab873b

Matrix-Vector Multiplication in Neural Network

- Computations in a neural network can be performed with matrix-vector multiplications



a1 = $\boldsymbol{w1x1 + w3x2 + w5x3 + b1}$

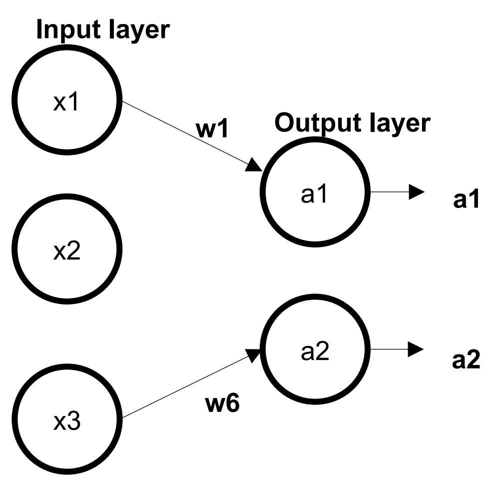
a2 = $\boldsymbol{w2x1 + w4x2 + w6x3 + b2}$

$$\begin{bmatrix} \boldsymbol{w1} & \boldsymbol{w3} & \boldsymbol{w5} \\ \boldsymbol{w2} & \boldsymbol{w4} & \boldsymbol{w6} \end{bmatrix} \mathrm{X} \begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix} + \begin{bmatrix} b1 \\ b2 \end{bmatrix} = \begin{bmatrix} a1 \\ a2 \end{bmatrix}$$

Sparse Matrix-Vector Multiplication (SpMV)

$$\begin{bmatrix} \boldsymbol{w1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{w6} \end{bmatrix} \text{X} \begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix} = \begin{bmatrix} a1 \\ a2 \end{bmatrix}$$

W X Y



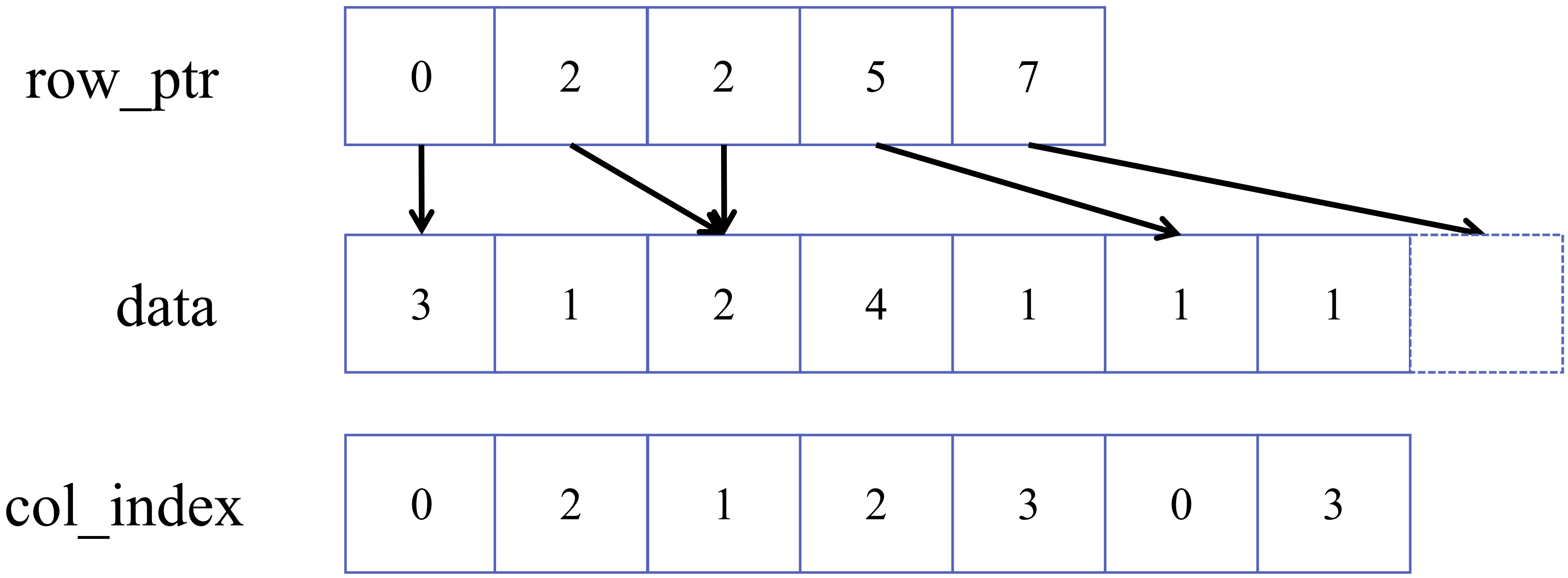
Compressed Sparse Row (CSR) Format

- A sparse matrix representation that avoids storing zero elements
- Stores only nonzero values and two sets of markers in three arrays
  - **data[]** : nonzero values
  - **col_index[]** : column of every nonzero value
  - **row_ptr[]** : Index of the beginning locations of each row in the data[] array

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

		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 Data Layout



Sequential SpMV

```
for (int row = 0; row < num_rows; row++) {
    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;
}
```

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

		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 Analysis

- It completely removes all zero elements from the storage
- But, it incurs storage overhead by introducing the column_index and row_ptr arrays

- For large and sparse matrices where the vast majority of elements are zero, the overhead is far small
  - If 1% of the elements are nonzero values, the total sotrage for the CSR would be around 2% of the space required to store both zero and nonzero elements

- Removing all zero elements also eliminates the need to
  - fetch zero elements from memory
  - perform useless multiplication operations on these zero elements.

A Simple Parallel SpMV

- Each thread processes one row

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

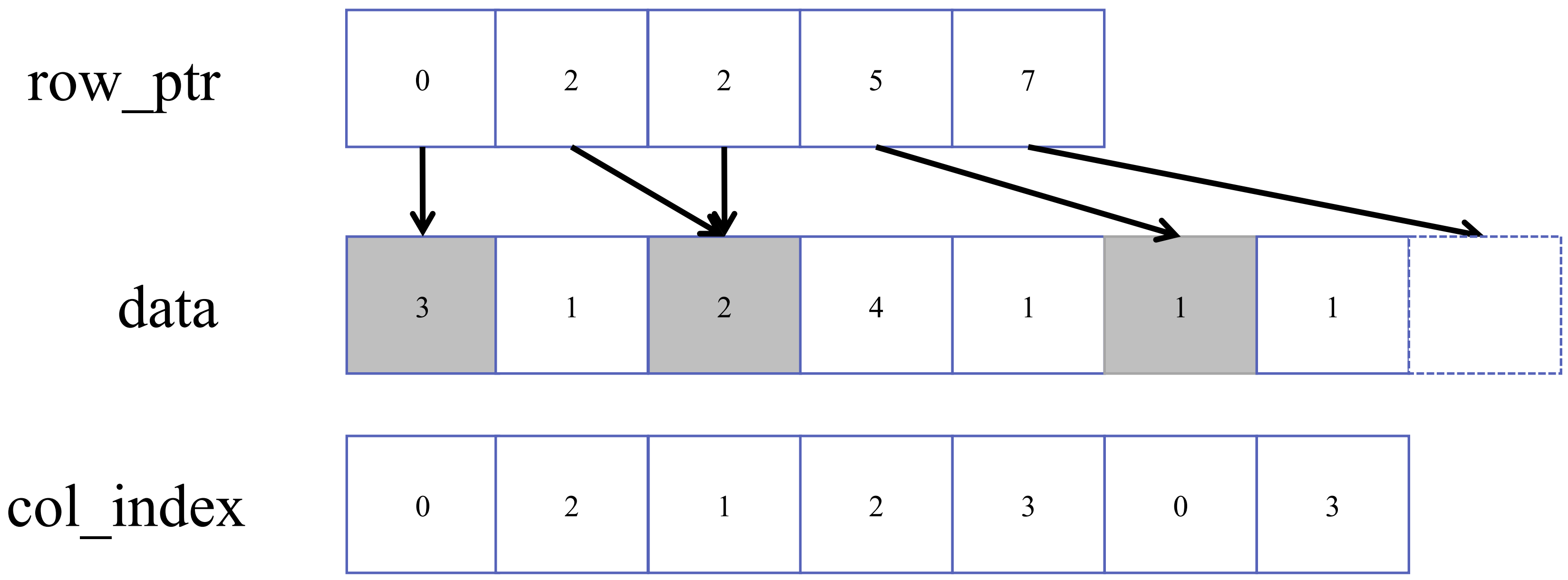
A Parallel SpMV/CSR Kernel

```
__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 }		

A Parallel SpMV/CSR Kernel

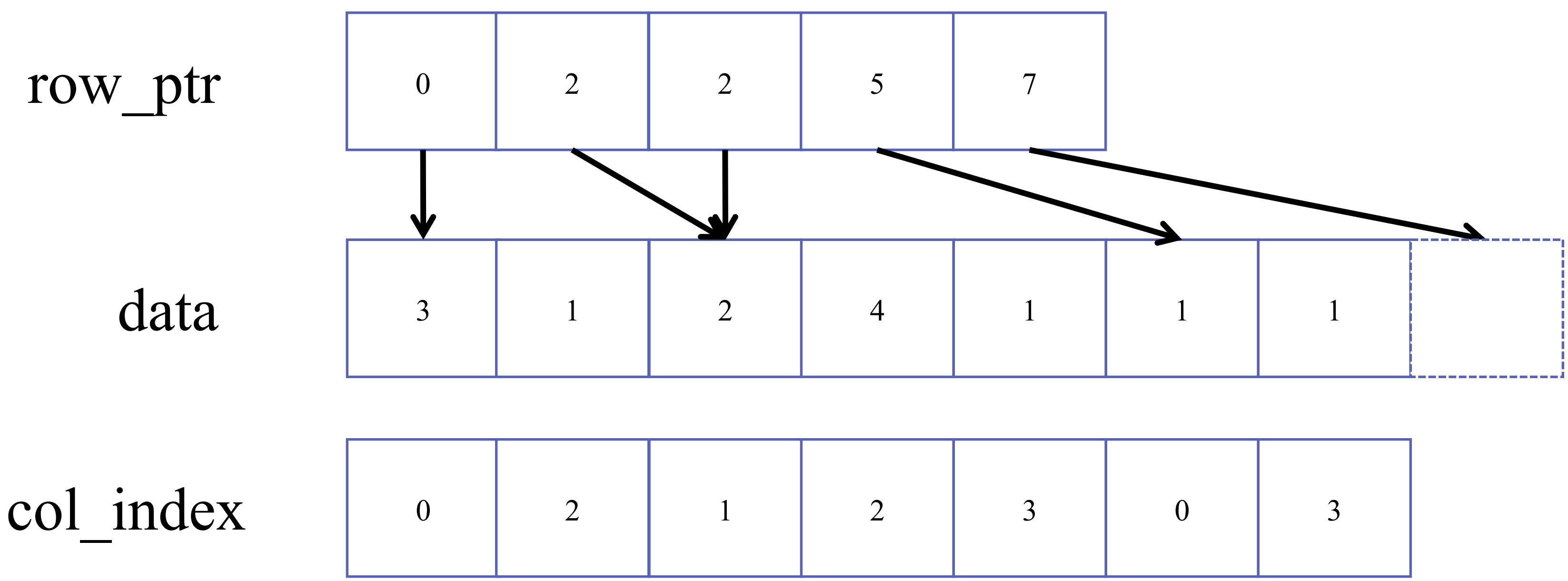
- **Shortcoming**
  - Kernel does not make coalesced memory accesses
    - Adjacent threads make simultaneous nonadjacent memory accesses
      - Ex) threads 0, 1, 2, and ,3 will access data[0], none, data[2], and data[5]



Grey elements are accessed by all threads in iteration 0

A Parallel SpMV/CSR Kernel (Cont'd)

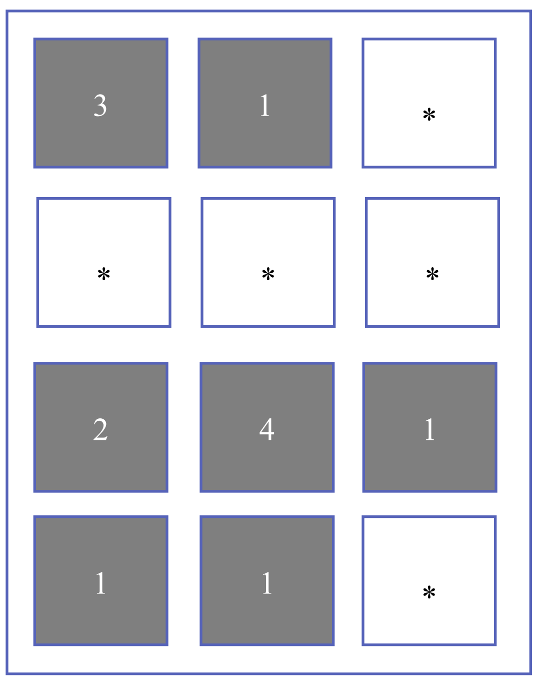
- **Shortcoming**
  - Kernel does not make coalesced memory accesses
  - Control flow divergence in all warps
    - The number of iterations performed by a thread depends on the number of nonzero elements in the row assigned to the thread
    - Distribution of nonzero elements among rows can be random
      - → Adjacent rows can have varying numbers of nonzero elements



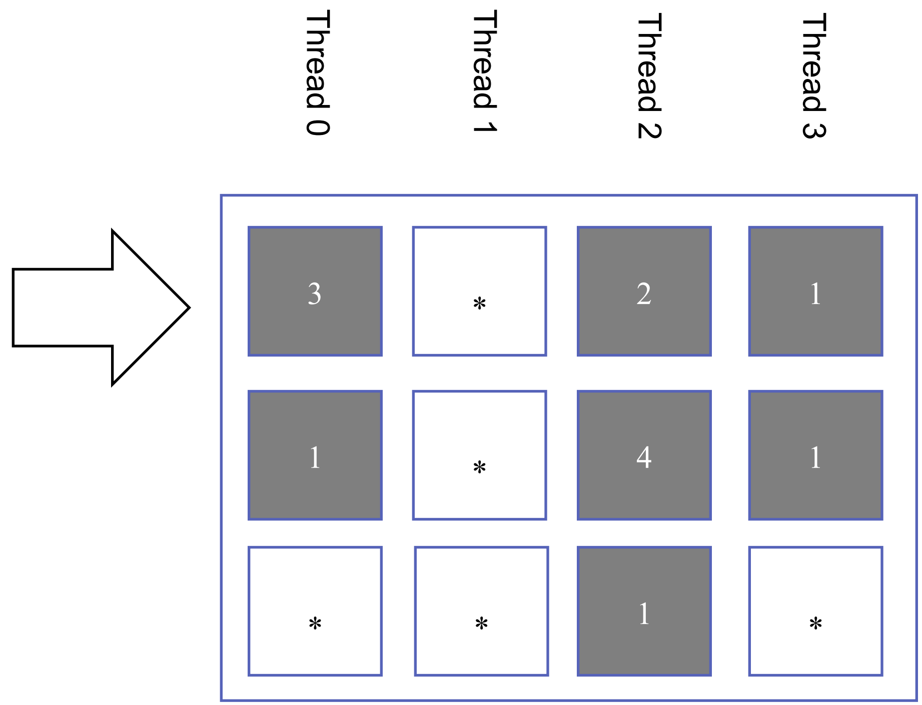
Regularizing SpMV with ELL(PACK) Format

- Pad all rows to the same length
- Transpose (Column Major)
- Both data and col_index padded/transposed

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

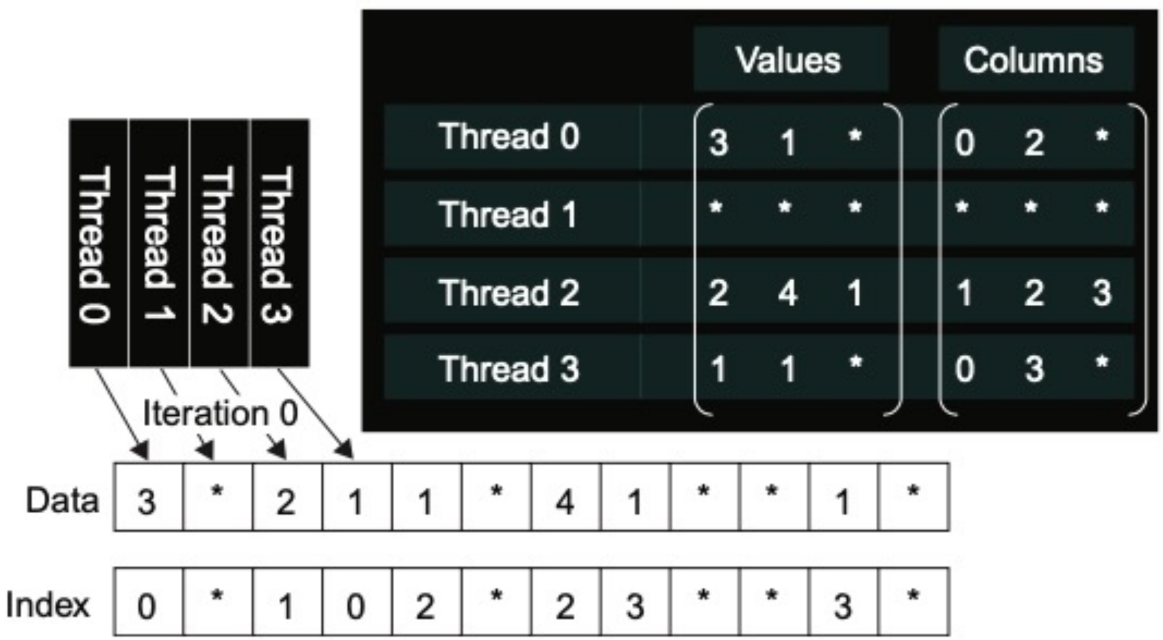


CSR with Padding



Transposed

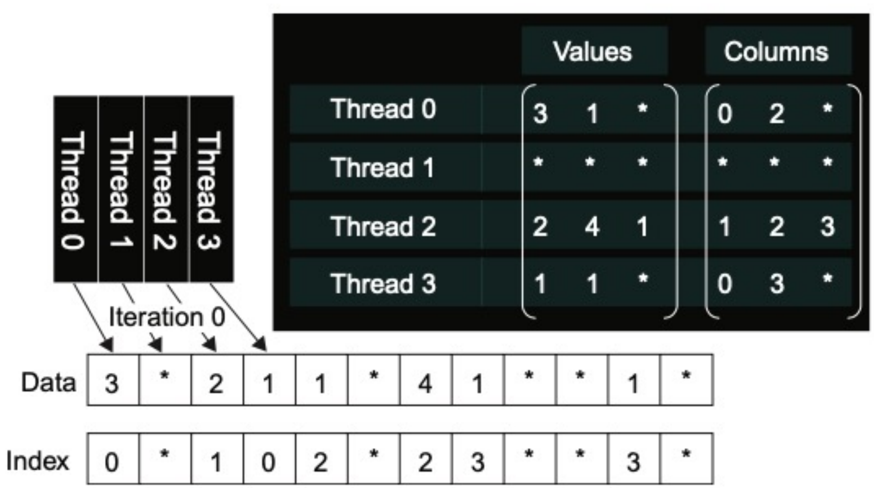
Regularizing SpMV with ELL(PACK) Format (cont'd)



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

A Parallel SpMV/ELL Kernel

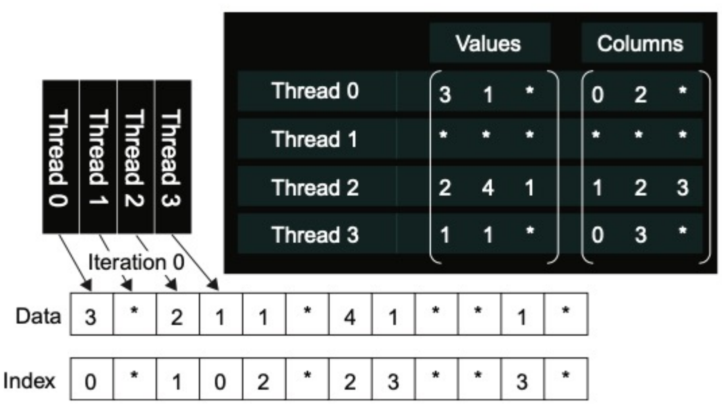
```
_global__ void SpMV_ELL(int num_rows, float *data,
   int *col_index, int num_elem, float *x, float *y) {

    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;
    }
 }
```



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

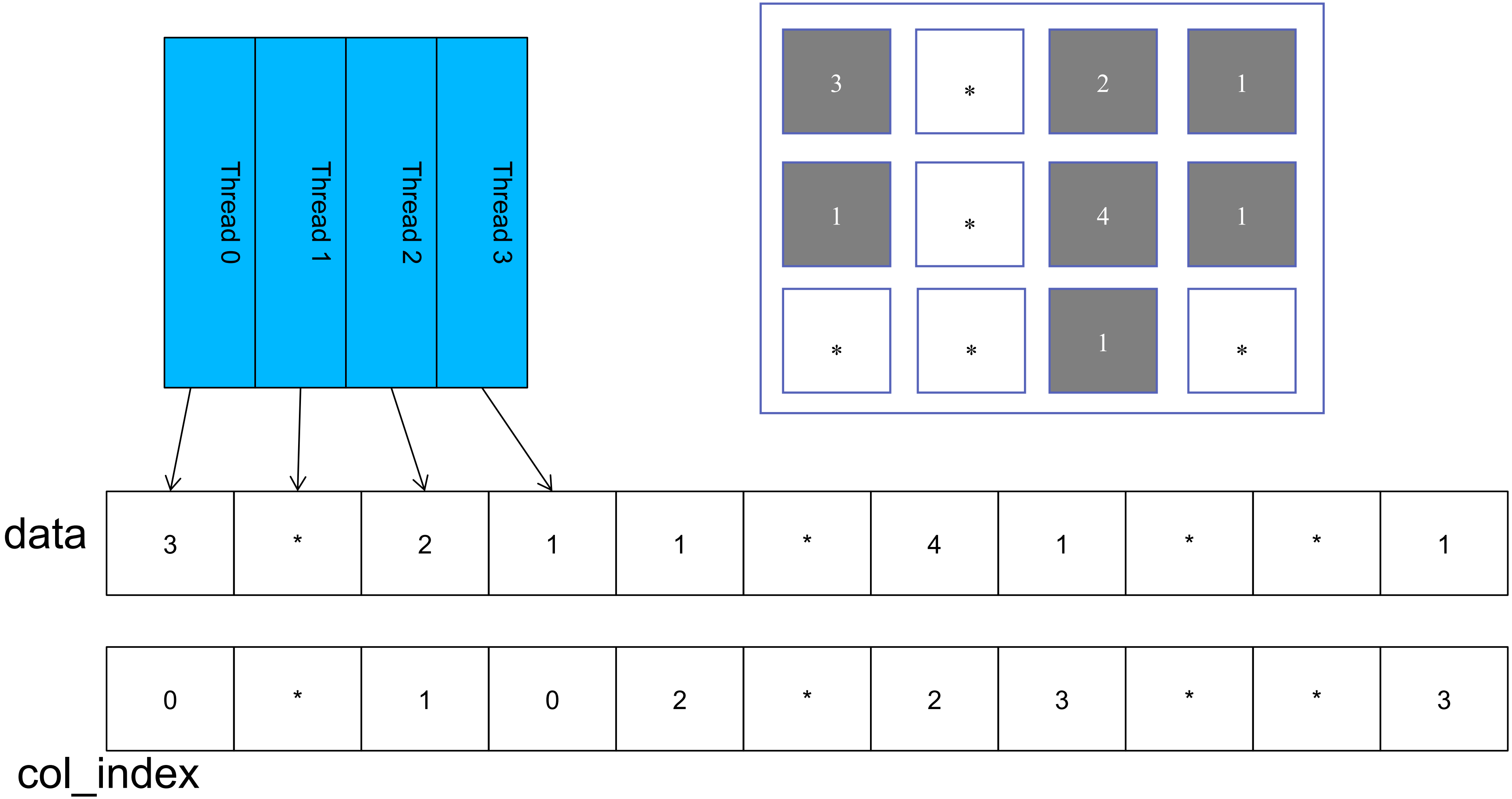
A Parallel SpMV/ELL Kernel (Cont'd)

- With padding, all rows are now of the same length (num_elem)
  - → **No control flow divergence**
- All adjacent thread access adjacent memory locations
  - → **Enable memory coalescing**
  - → **Use memory bandwidth more efficiently**
- Shortcoming
  - **if a small number of rows have an exceedingly large number of nonzero elements →excessive number of padded elements**



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

Memory Coalescing with ELL



Coordinate (COO) format

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

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

		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 }

- A sequential loop that implements SpMV/COO

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

Reordering Elements with COO format

- The elements in a COO format can be arbitrarily reordered

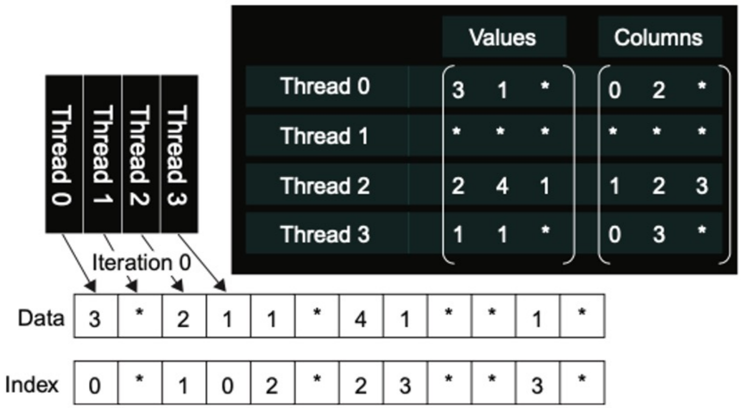
			Row 0			Row 2			Row 3		
Nonzero values	data[7]	{	3,	1,	2,	4,	1,	1,	1	}
Column indices	col_index[7]	{	0,	2,	1,	2,	3,	0,	3	}
Row indices	row_index[7]	{	0,	0,	2,	2,	2,	3,	3	}

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	}

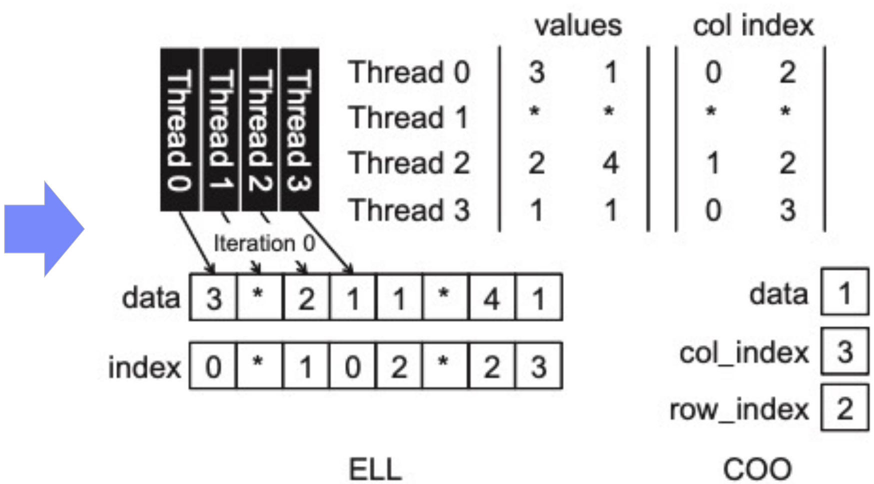
Hybrid ELL and COO method for SpMV

- Store rows with exceedingly large numbers of nonzero elements in a separate COO format
- Store remaining rows in ELL or CSR format

ELL



COO



Hybrid Format

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

Often implemented in sequential host code in practice