中国科学院大学
University of Chinese Academy of Sciences

Introduction to CUDA

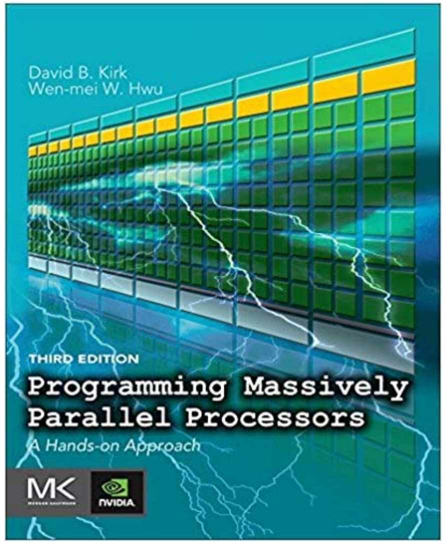
(8) Parallel Pattern: Sparse Matrix

Reference

- CUDA C Programming Guide,
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

- Programming Massively Parallel Processors,
  - A Hands-on Approach
  - Third Edition

  - Chapter 10



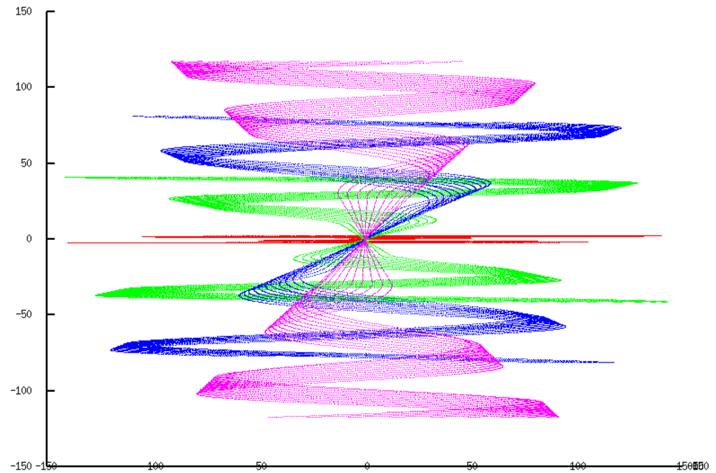
Content

- **Background**
- **Parallel SpMV Using CSR**
- **Padding and Transposition**
- **Hybrid Approach to Regulate Padding**
- **Sorting and Partitioning for Regularization**

Sparse Data
Motivation for Compaction

- Many real-world inputs are sparse/non-uniform

- Signal samples,
- mesh models,
- transportation networks,
- communication networks, etc.



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

Challenges

- Compared to dense matrix multiplication, SpMV
  - Is irregular/unstructured
  - Has little input data reuse
  - Benefits little from compiler transformation tools
- Key to maximal performance
  - Maximize regularity (by reducing divergence and load imbalance)
  - Maximize DRAM burst utilization (layout arrangement)

A Simple Parallel SpMV

$$\begin{array}{l} \text{Row 0} \\ \text{Row 1} \\ \text{Row 2} \\ \text{Row 3} \end{array} \left|\begin{array}{cccc} 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \\ 1 & 0 & 0 & 1 \end{array}\right|$$

$\boxed{Ax+Y=0}$

- Each row of the matrix represents one equation of the linear system.

- How to solve x?

A Simple Parallel SpMV

$$
\begin{array}{l}
\text{Row 0} \\
\text{Row 1} \\
\text{Row 2} \\
\text{Row 3}
\end{array}
\begin{vmatrix}
3 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 2 & 4 & 1 \\
1 & 0 & 0 & 1
\end{vmatrix}
$$

$\boxed{Ax+Y=0}$

- How to solve x? -- Direct method

$$\boxed{X=A^{-1}Y}$$

- Require inversion of the coefficient matrix
- Traditional inversion algorithms such as Gaussian elimination can create too many "fill-in" elements and explode the size of the matrix

A Simple Parallel SpMV

$$\begin{array}{l} \text{Row 0} \\ \text{Row 1} \\ \text{Row 2} \\ \text{Row 3} \end{array} \begin{vmatrix} 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \\ 1 & 0 & 0 & 1 \end{vmatrix} \qquad \boxed{Ax+Y=0}$$

- How to solve x? -- Iterative method (CG)

$$\boxed{Ax+Y} \qquad \text{If } x^TAx > 0$$

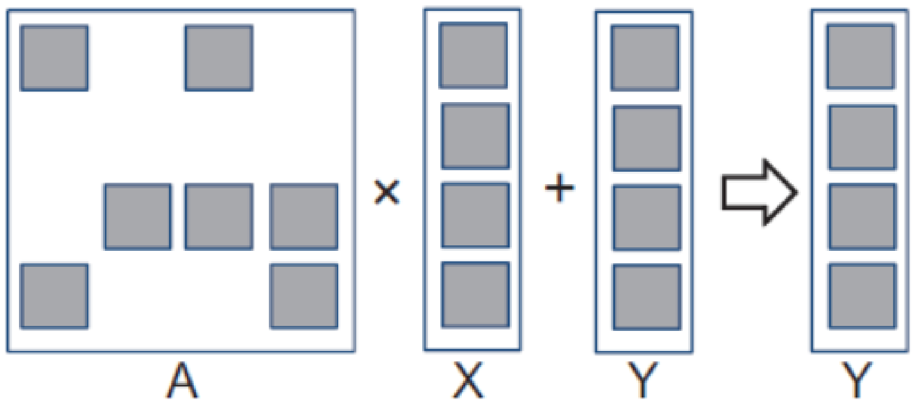
The conjugate gradient method CG is preferred.

A sparse matrix - vector multiplication and accumulation

Sparse Matrix-Vector Multiplication (SpMV)

SpMV

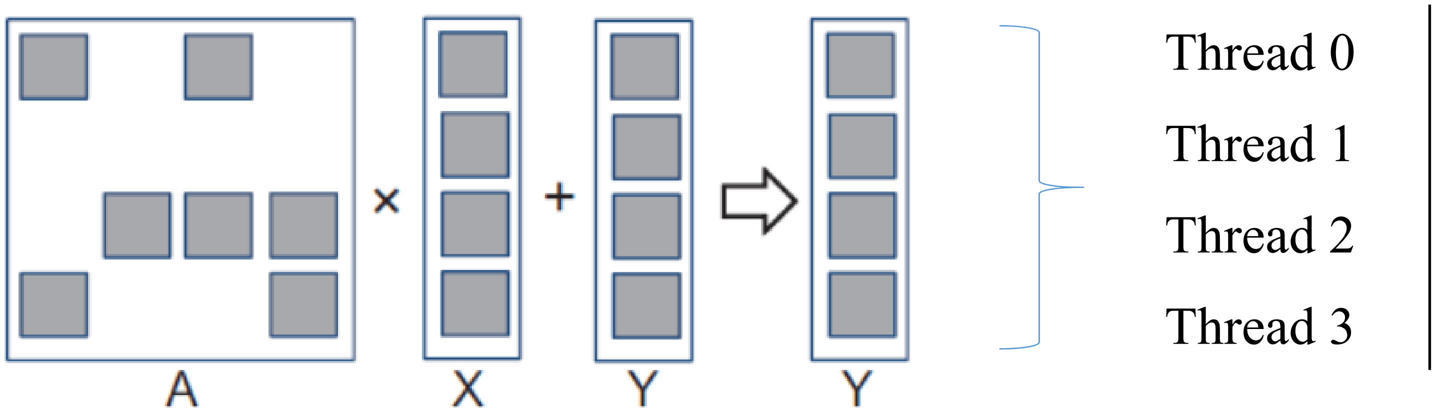
- Main task in CG:
  - matrix–vector multiplication and accumulation



- Parallel solution:
  - Each thread processes one row ?

A Simple Parallel SpMV

- Main task in CG:
  - matrix–vector multiplication and accumulation



- Main problem:
  - Storage and bandwidth considerations

We need a format (representation) that avoids storing zero elements.

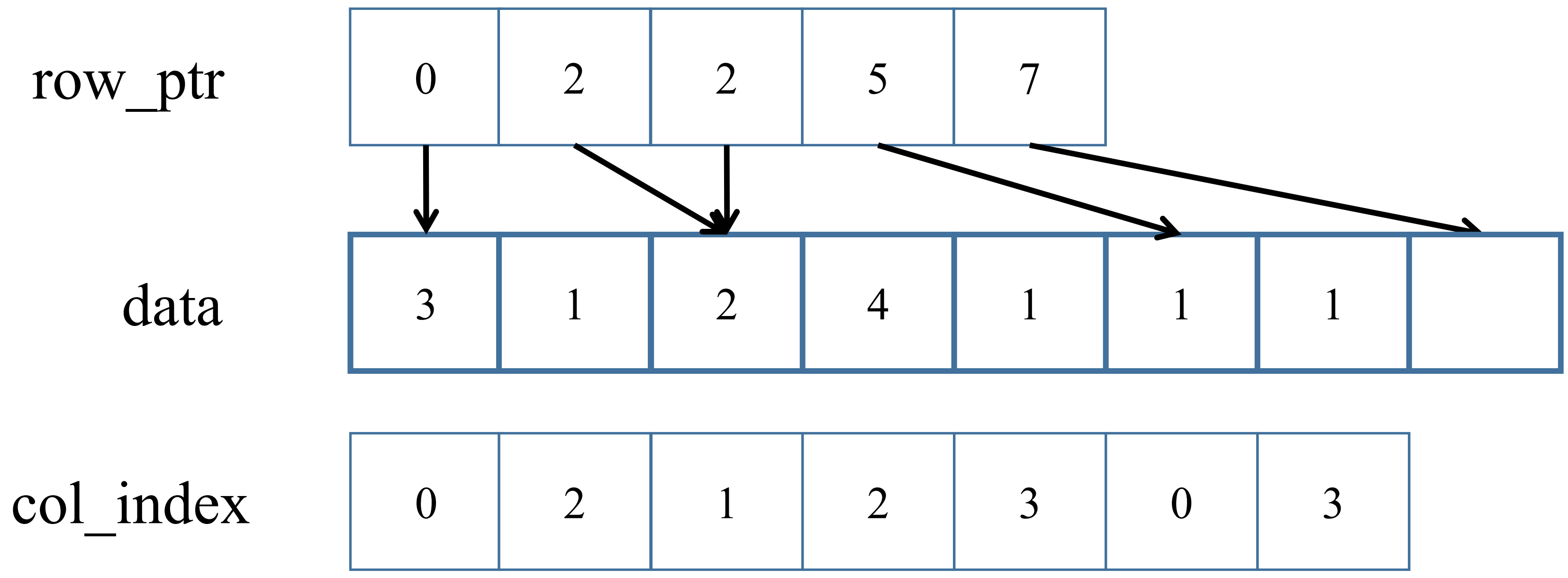
Compressed Sparse Row (CSR) Format

- CSR: stores only nonzero values in one-dimensional.

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

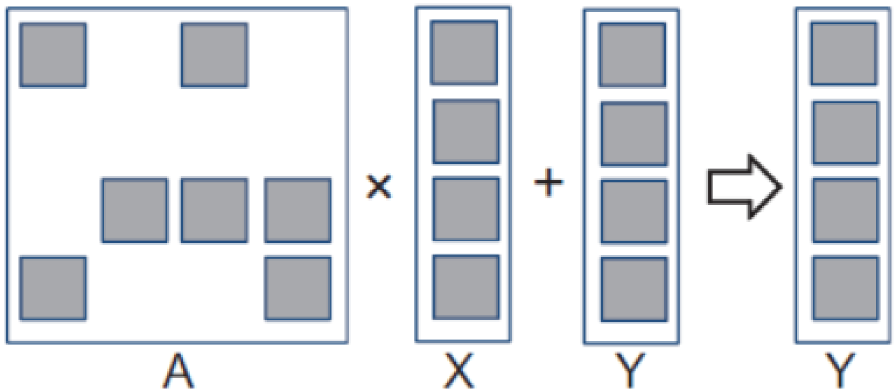
		Row 0	Row 2	Row 3
Nonzero values	data[7]	{ 3, 1,	2, 4, 1,	1, 1 }
Column indices	col_index[7]	{ 0, 2,	1, 2, 3,	0, 3 }
Row Pointers	ptr[5]	{ 0, 2,	2, 5,	7 }

CSR Data Layout



A sequential SpMV /CSR

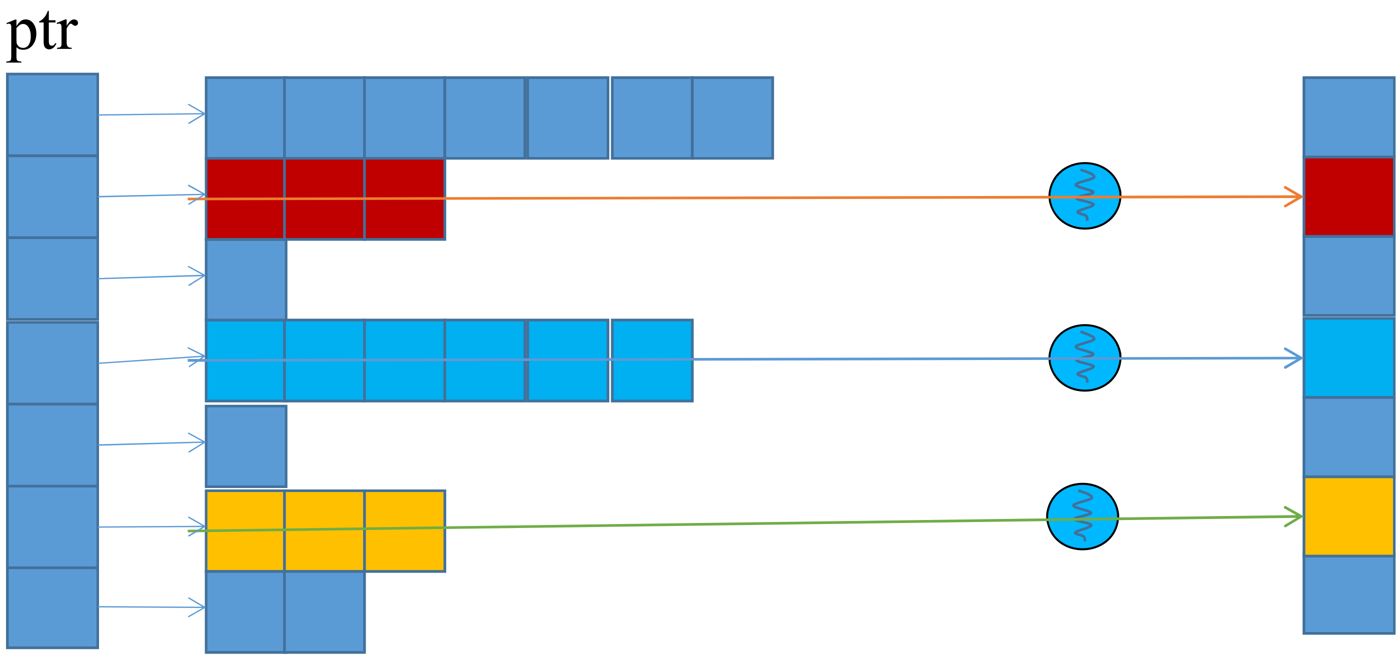
```
for (int row = 0; row < num_rows; row++)
 {
    float dot = 0;
    int row_start = row_ptr[row];
    int row_end = row_ptr[row+1];
   for (int elem = row_start; elem < row_end; elem++)
    {
      dot += data[elem] * x[col_index[elem]];
    }
    y[row] += dot;
 }
```



Content

- **Background**
- **Parallel SpMV Using CSR**
- **Padding and Transposition**
- **Hybrid Approach to Regulate Padding**
- **Sorting and Partitioning for Regularization**

CSR Kernel Design



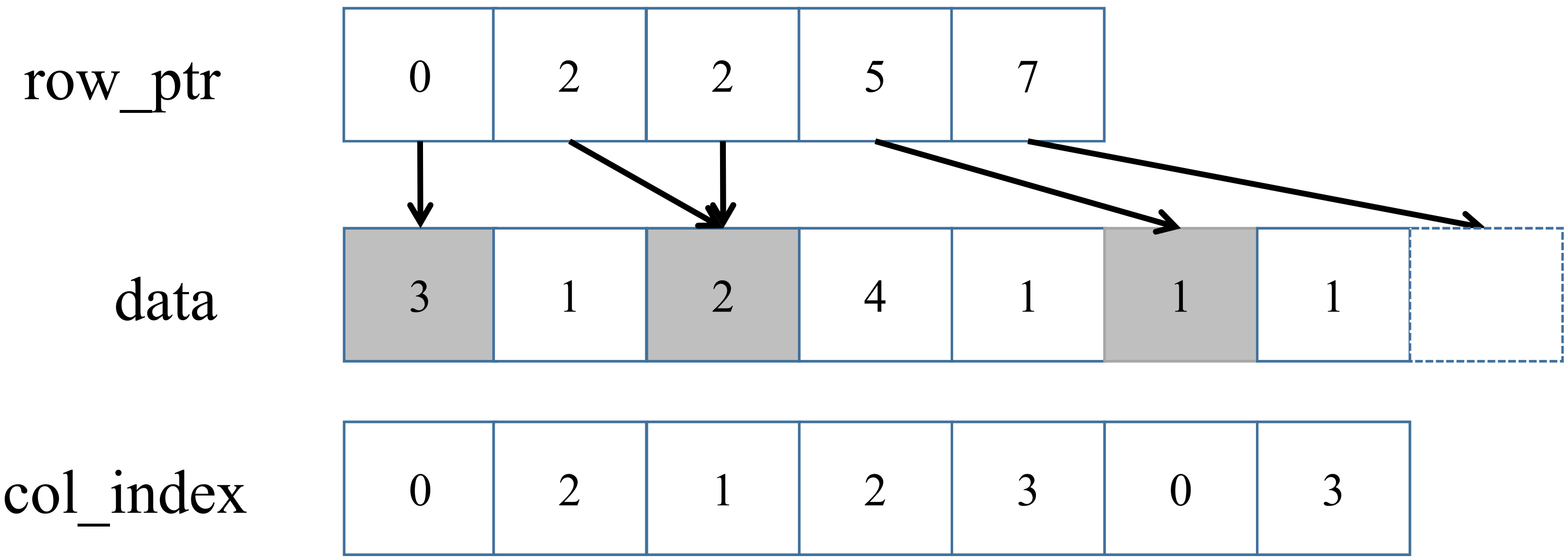
A Parallel SpMV/CSR Kernel (CUDA)

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

CSR Kernel Memory Divergence

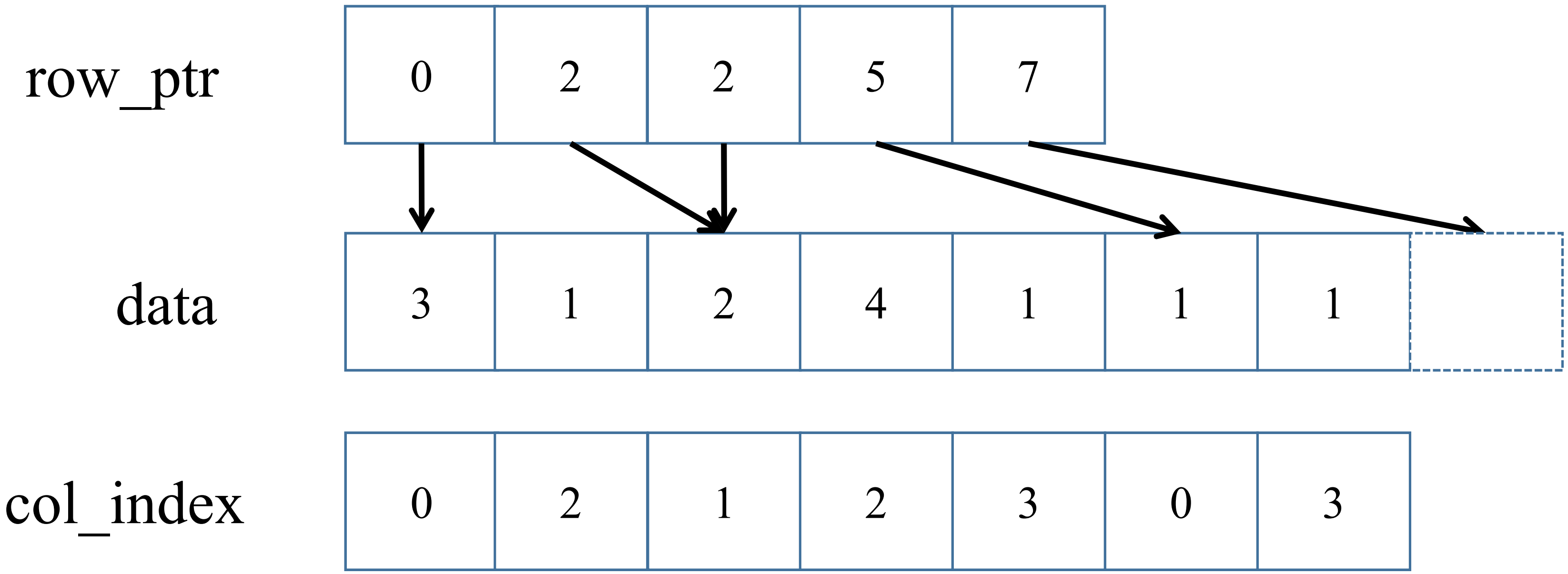
- Adjacent threads access <u>non-adjacent</u> memory locations
  - Grey elements are accessed by all threads in iteration 0



Memory Divergence

CSR Kernel Control Divergence

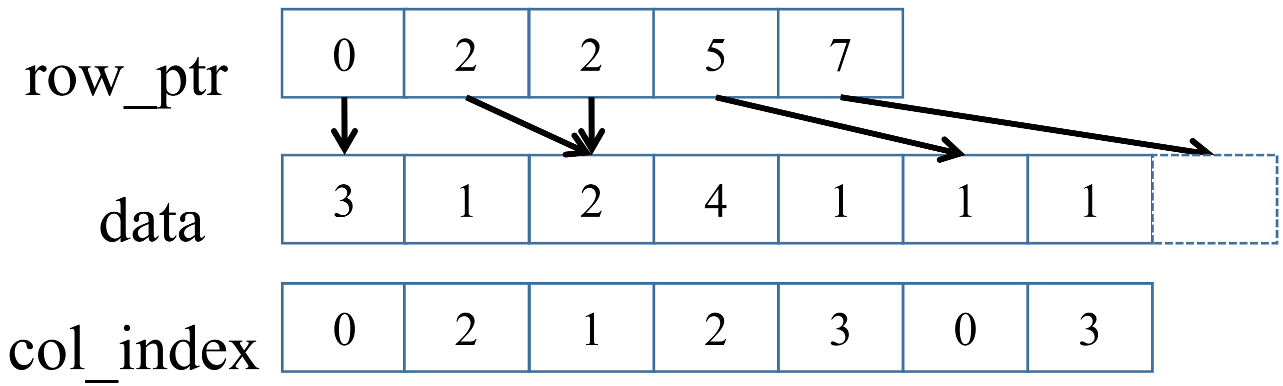
- Threads execute <u>different number of iterations</u> in the kernel for-loop



Control Divergence

CSR Kernel Data Dependence

- Both the execution efficiency and memory bandwidth efficiency of the parallel SpMV kernel depends on the distribution of the input data matrix.
- Commonly observed in real-world applications.



SpMV / CSR

- Advantages
  - Removing all zero elements from the storage
  - also eliminates the need to fetch these zero elements from memory
  - also eliminates the need to perform useless multiplication operations on these zero elements.
- Disadvantages
  - storage overhead by introducing the col_index and row_ptr arrays.
  - does not make coalesced memory accesses.
  - potential to incur significant control flow divergence in all warps.

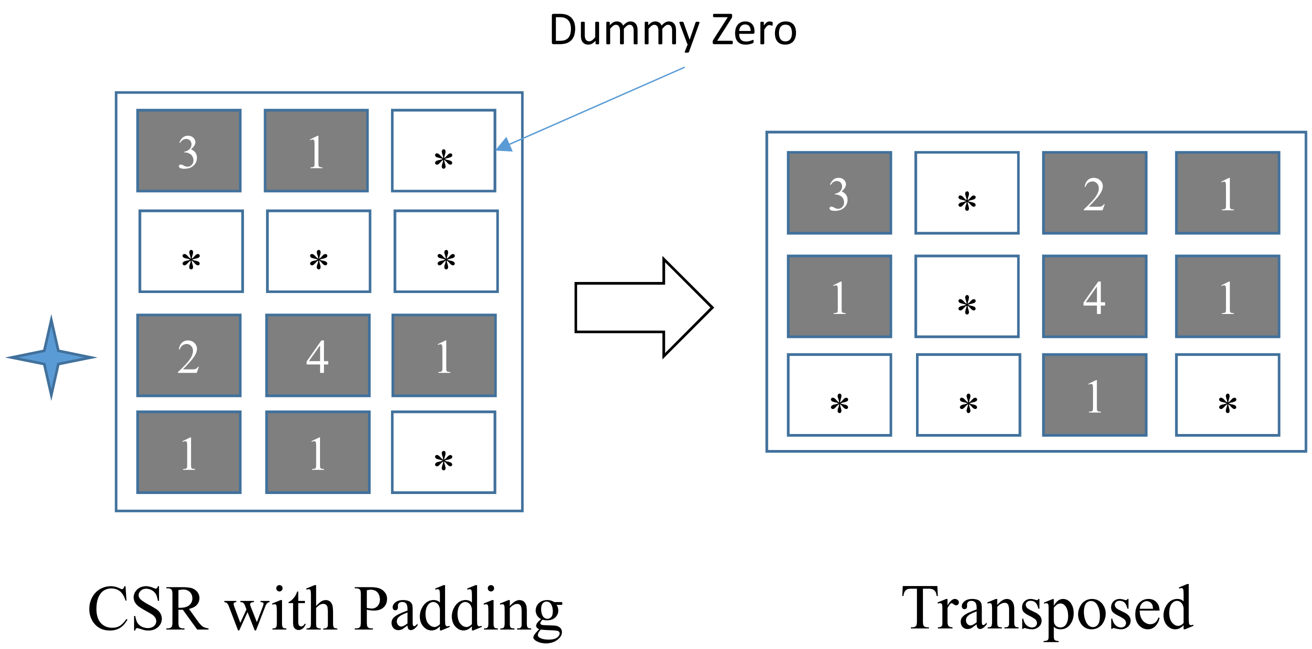
Content

- **Background**
- **Parallel SpMV Using CSR**
- **Padding and Transposition**
- **Hybrid Approach to Regulate Padding**
- **Sorting and Partitioning for Regularization**

ELL storage format

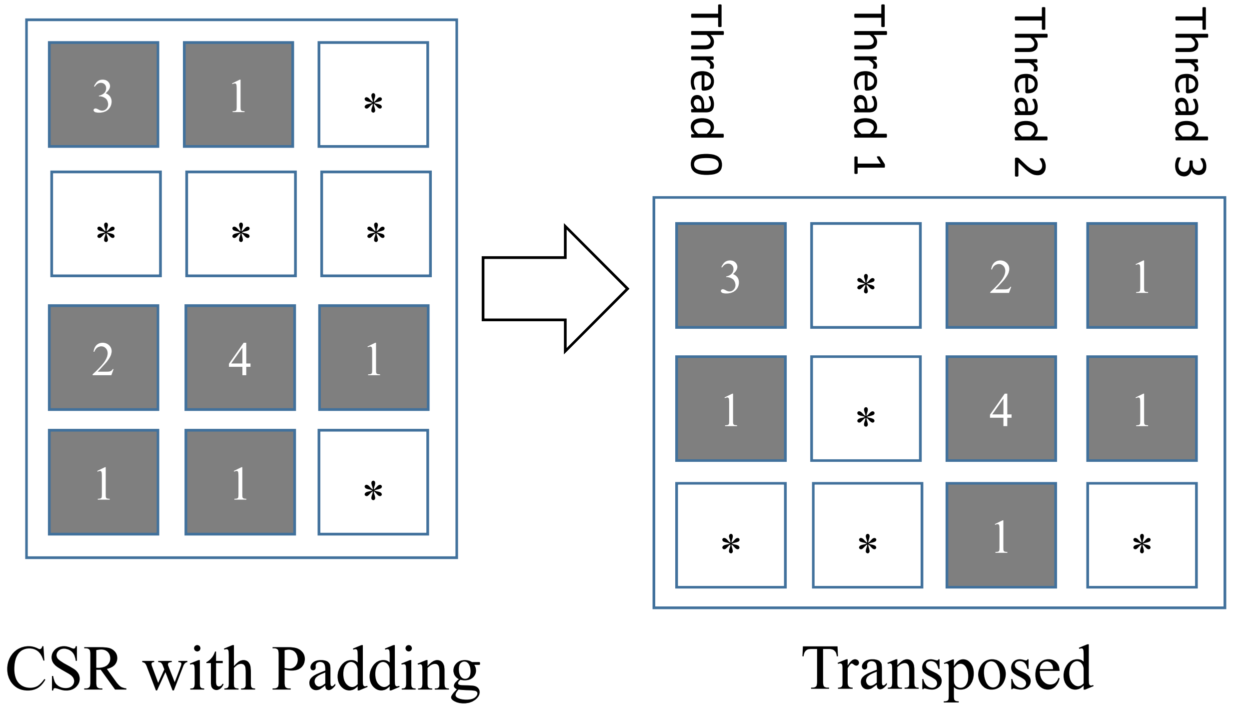
ELL: came from the sparse matrix package in ELLPACK.

- non-coalesced memory accesses and control divergence
  → applying data <u>padding</u> and <u>transposition</u>



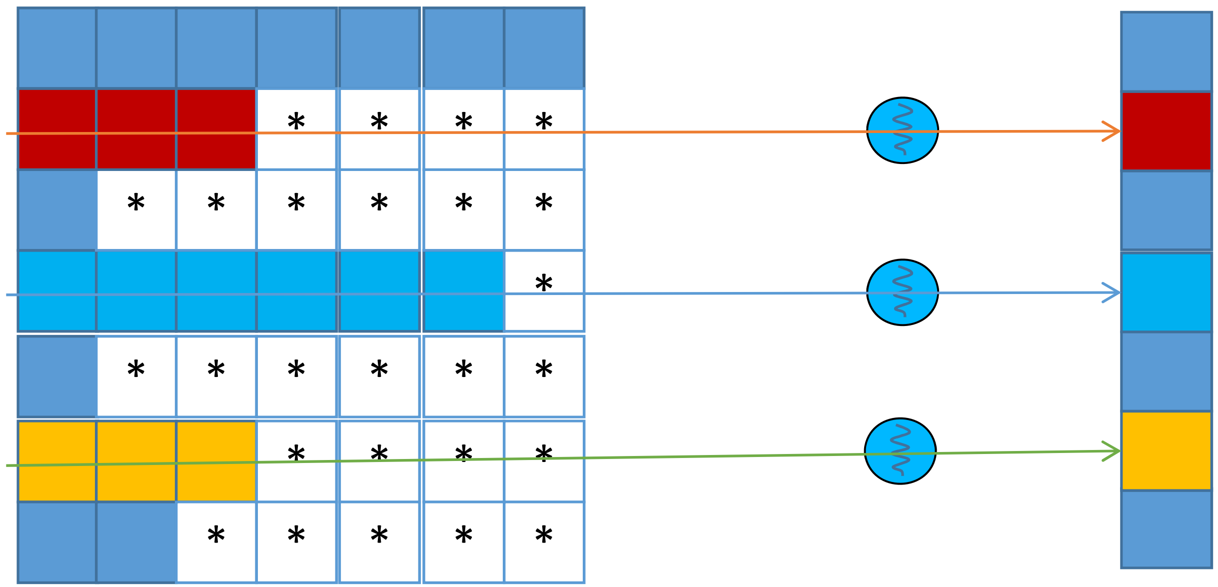
SpMV / ELL

- Pad all rows to the same length
- Transpose (Column Major) for DRAM efficiency



Both data and col_index padded/transposed

ELL Kernel Design

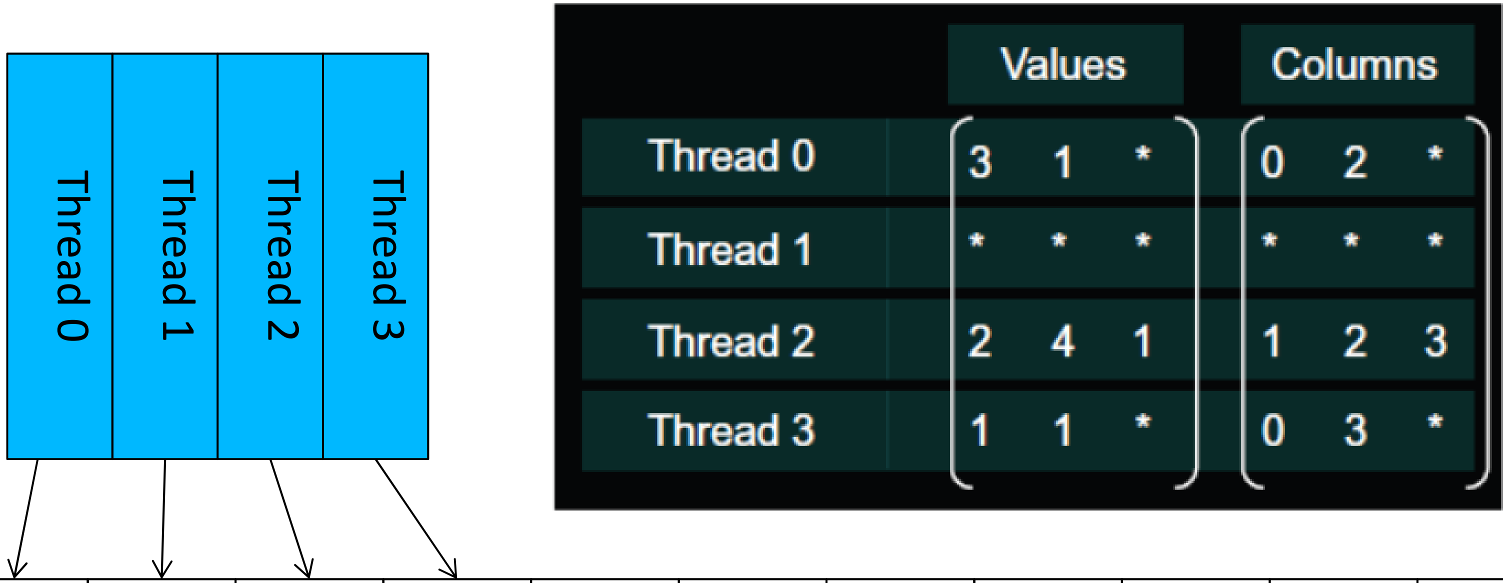


CSR with Padding

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
col_index	0	*	1	0	2	*	2	3	*	*	3

row_ptr is no longer need

SpMV / ELL

- Advantages
  - coalesced memory accesses.
    - all adjacent threads are now accessing adjacent memory locations;
  - No control flow divergence in warps.
    - all threads now iterate exactly the same number of times in the dot product loop.
- Disadvantages
  - Inefficient if a few rows are much longer than others.
  - 1000×1000 sparse matrix has 1% of its elements of nonzero value:
    - Each row has 10 nonzero elements;
    - Several rows has 200 nonzero values while all other rows have less than 10.

Content

- **Background**
- **Parallel SpMV Using CSR**
- **Padding and Transposition**
- **Hybrid Approach to Regulate Padding**
- **Sorting and Partitioning for Regularization**

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	}

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	}

Re-order

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	}

SpMV/COO

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

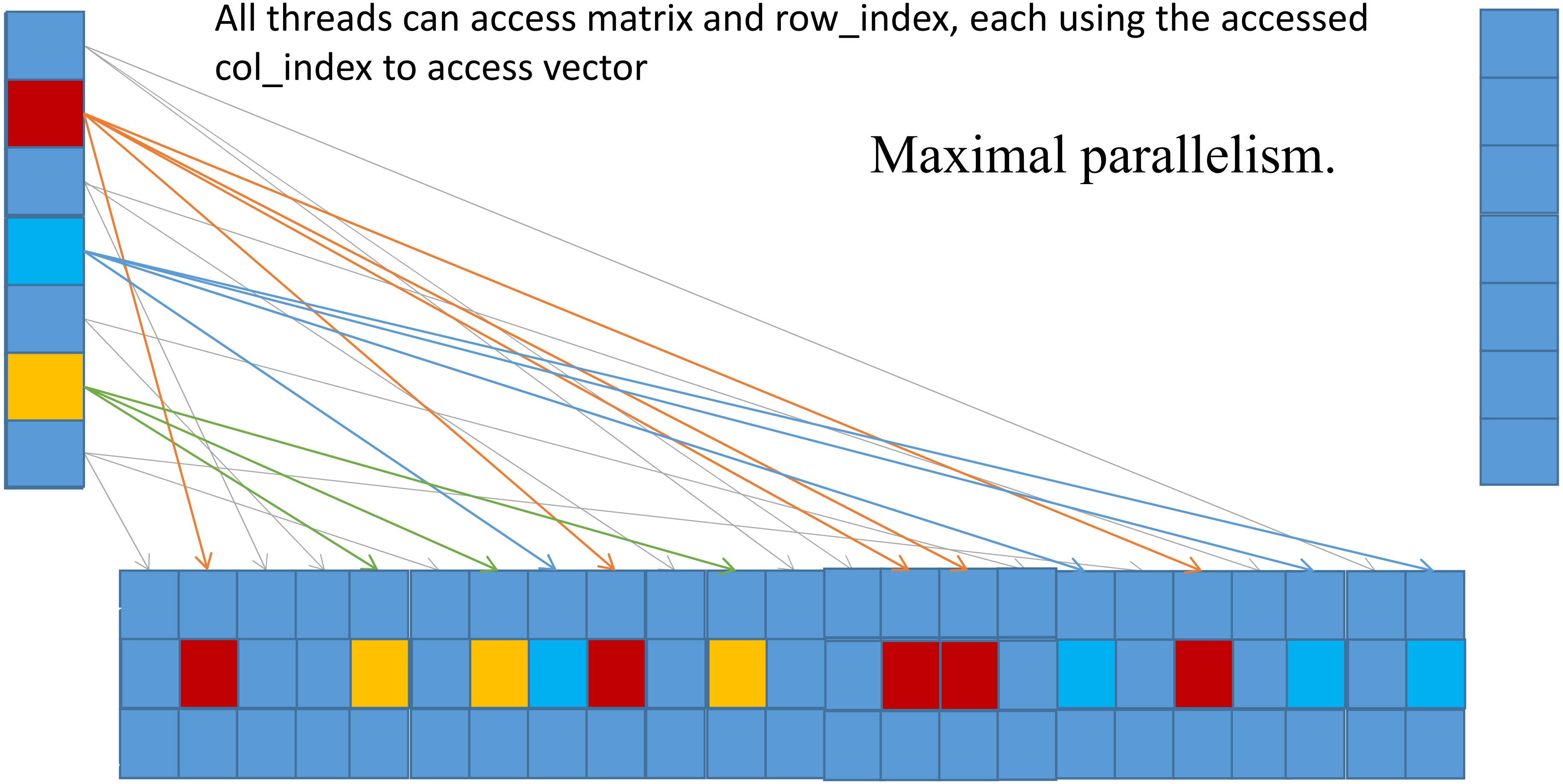
A sequential loop that implements SpMV/COO

The correct final answer will be obtained regardless of the order in which these elements are processed.

COO Kernel Design
Accessing Input Matrix and Vector

All threads can access matrix and row_index, each using the accessed col_index to access vector

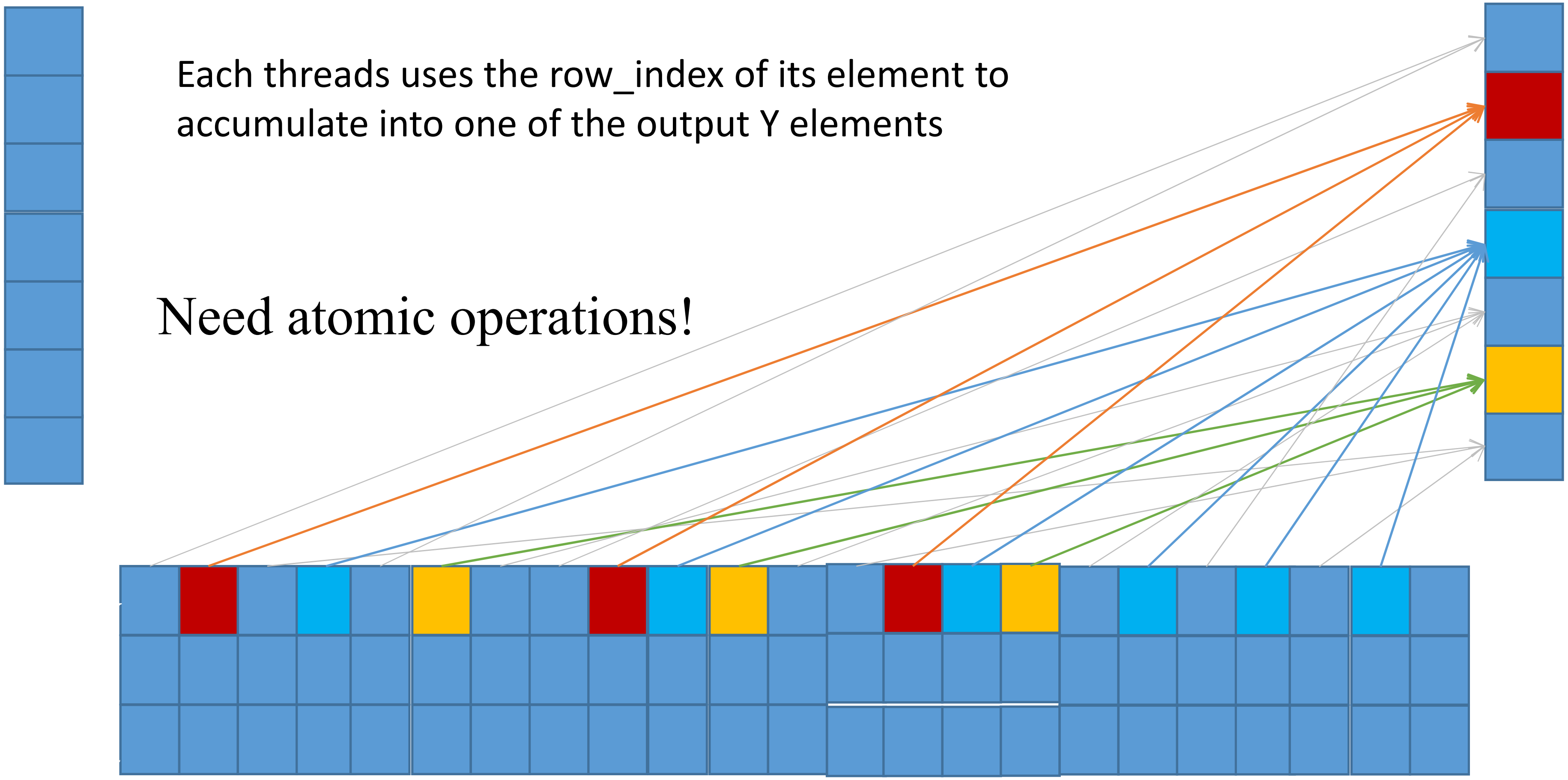
Maximal parallelism.



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!



SpMV / COO

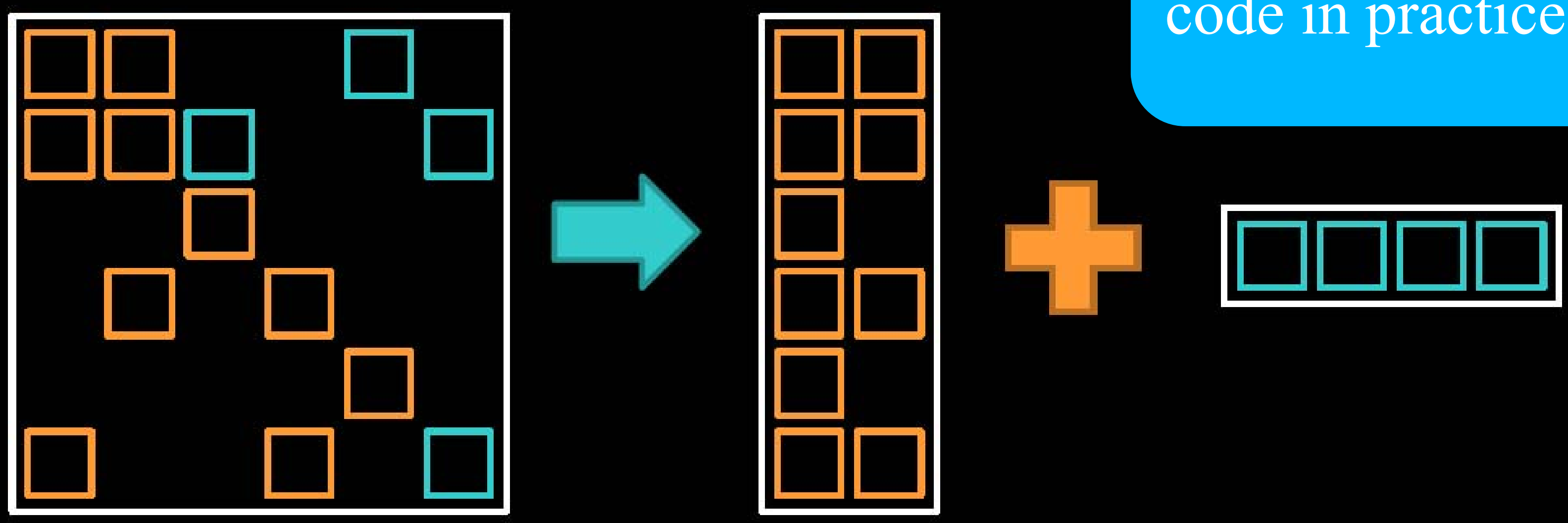
- Advantages
  - additional benefit of flexibility.

- Disadvantages
  - involves additional storage cost for the row_index array.
  - Need atomic operations.

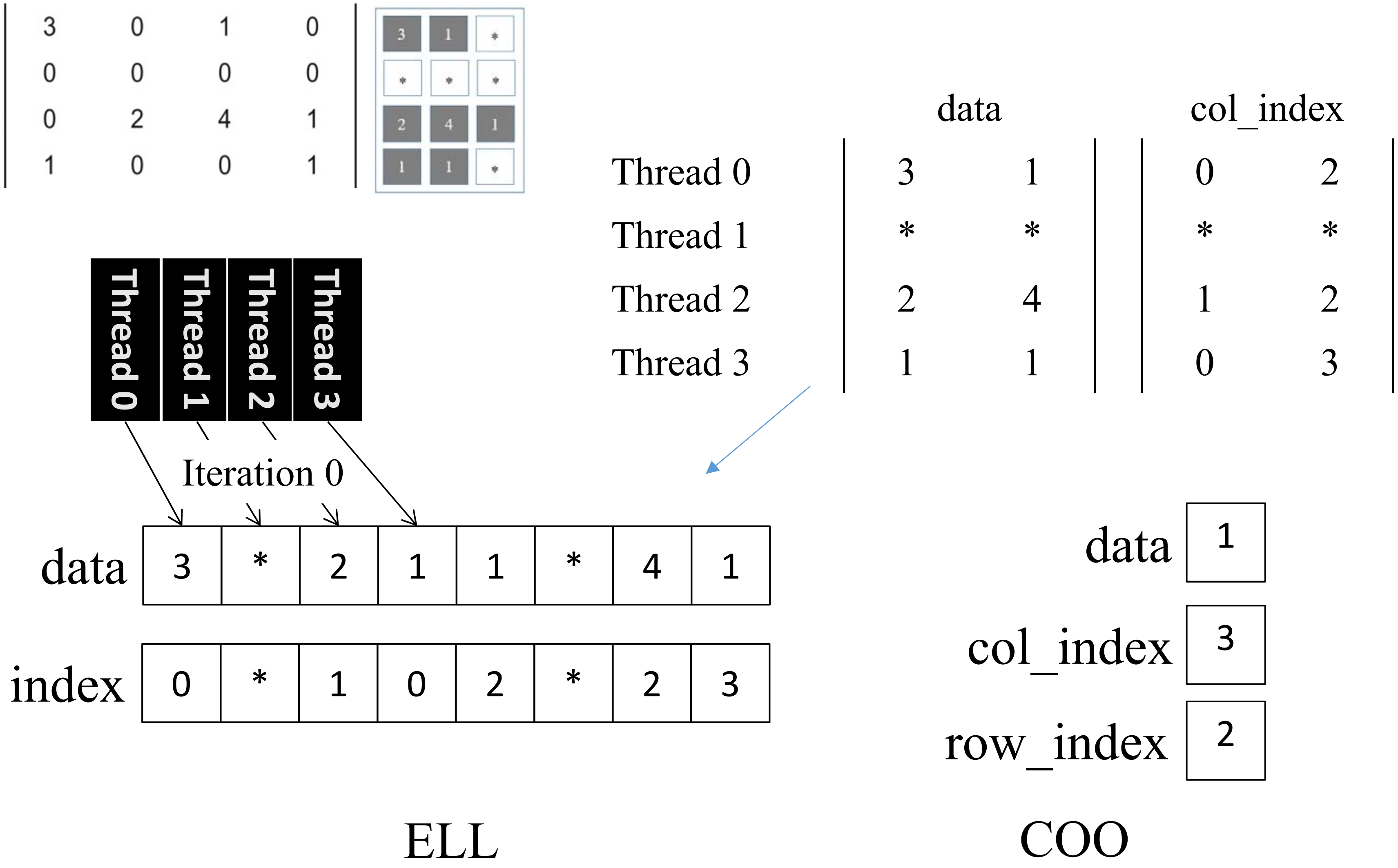
Hybrid Format

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

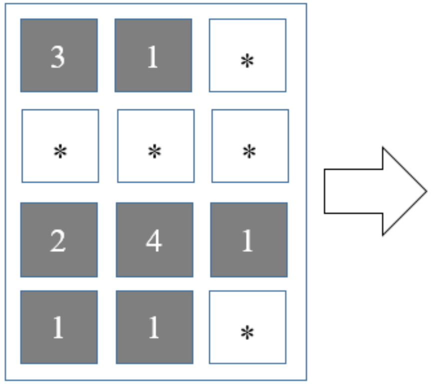
Often implemented in sequential host code in practice



Reduced Padding with Hybrid Format



Reduced Padding with Hybrid Format

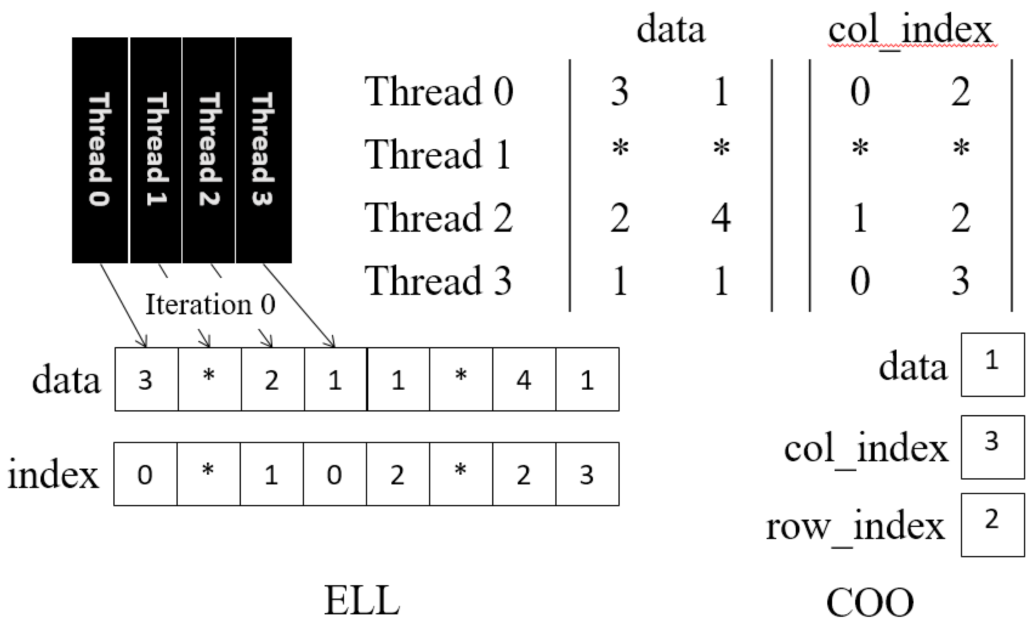


	data		col_index	
Thread 0	3	1	0	2
Thread 1	*	*	*	*
Thread 2	2	4	1	2
Thread 3	1	1	0	3

- The number of padded elements is reduced from 5 to 2.

- More importantly, all of the threads only need to perform 2 rather than 3 iterations.

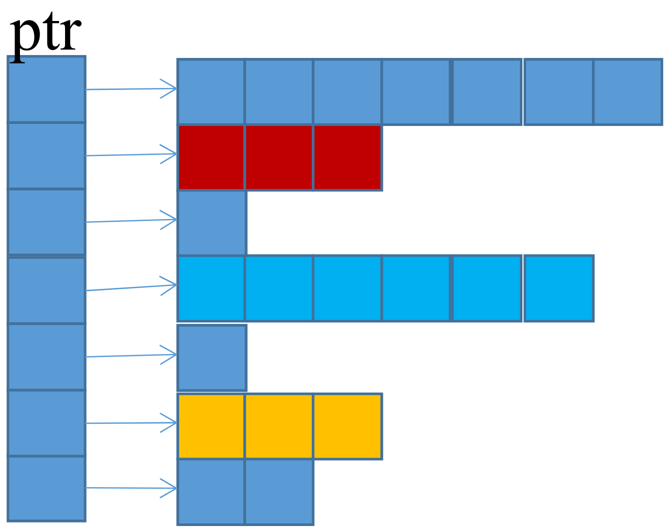
Question



- Whether the additional work of the host to separate COO elements from an ELL format could incur excessive overhead?
  - the SpMV is performed on the same sparse kernel repeatedly in an iterative solver. Averaged the overhead.

SpMV / Hybrid Format

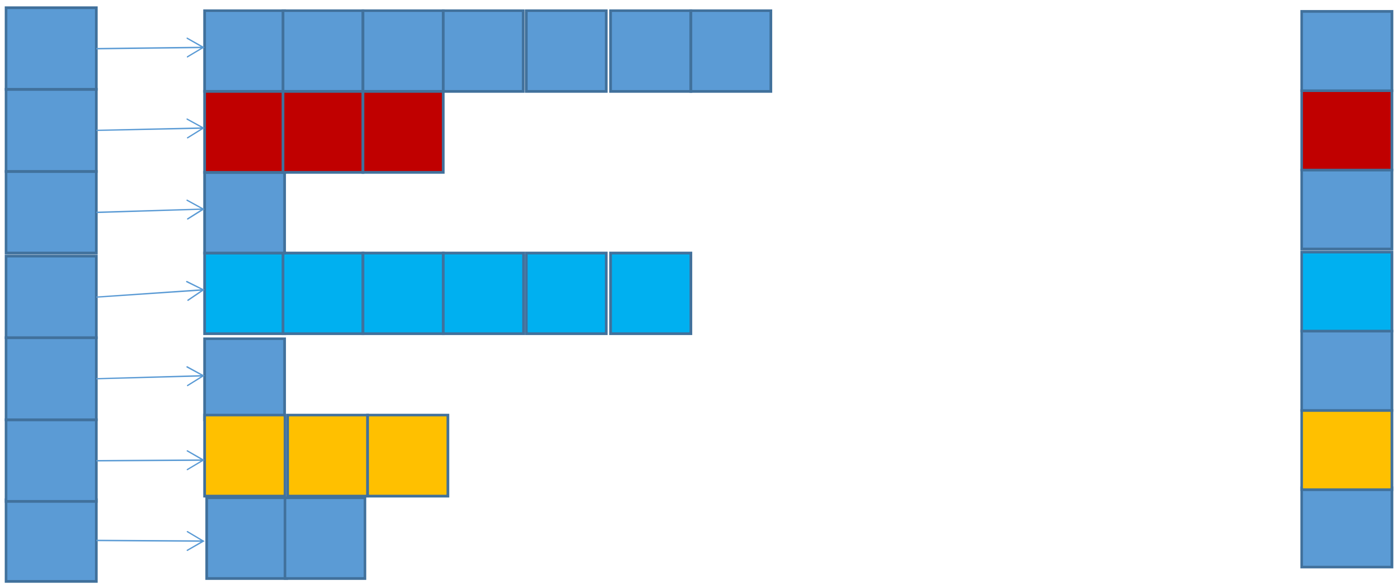
- Advantages
  - Avoid large number of zero padding by regularization.
  - CPU+GPU
- Disadvantages
  - Non-load-balance.



Content

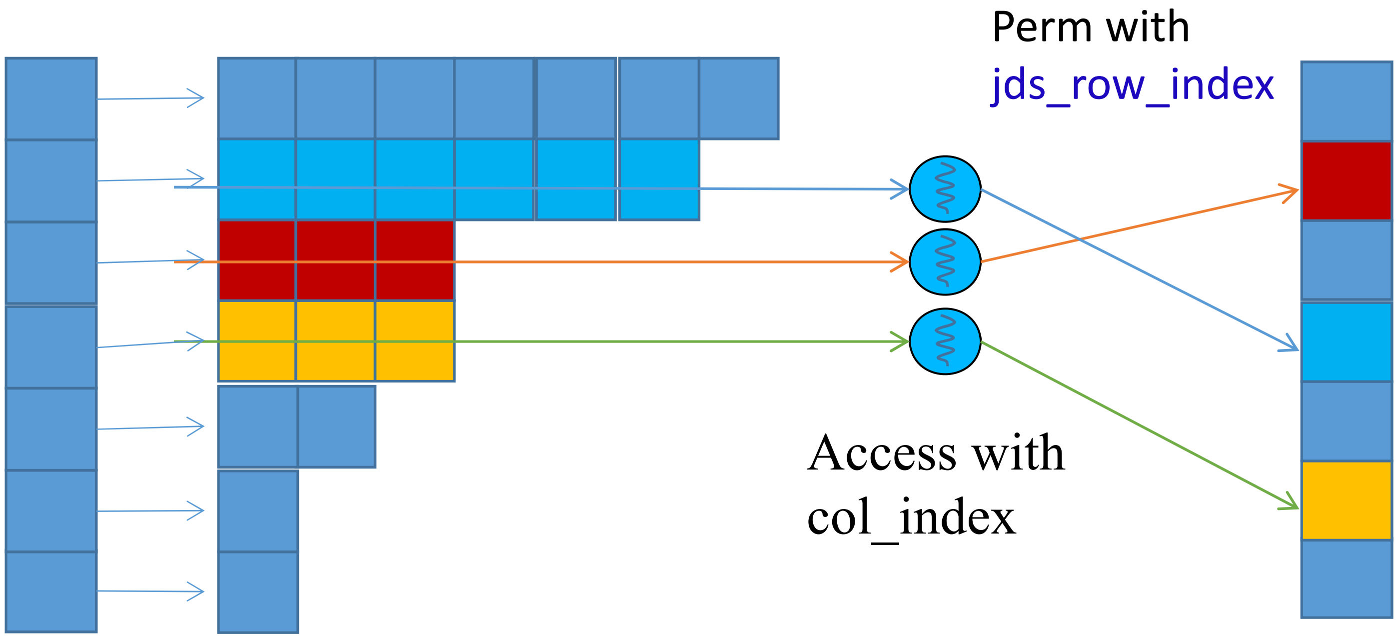
- **Background**
- **Parallel SpMV Using CSR**
- **Padding and Transposition**
- **Hybrid Approach to Regulate Padding**
- **<u>Sorting and Partitioning for Regularization</u>**

JDS (Jagged Diagonal Sparse) Kernel Design for Load Balancing



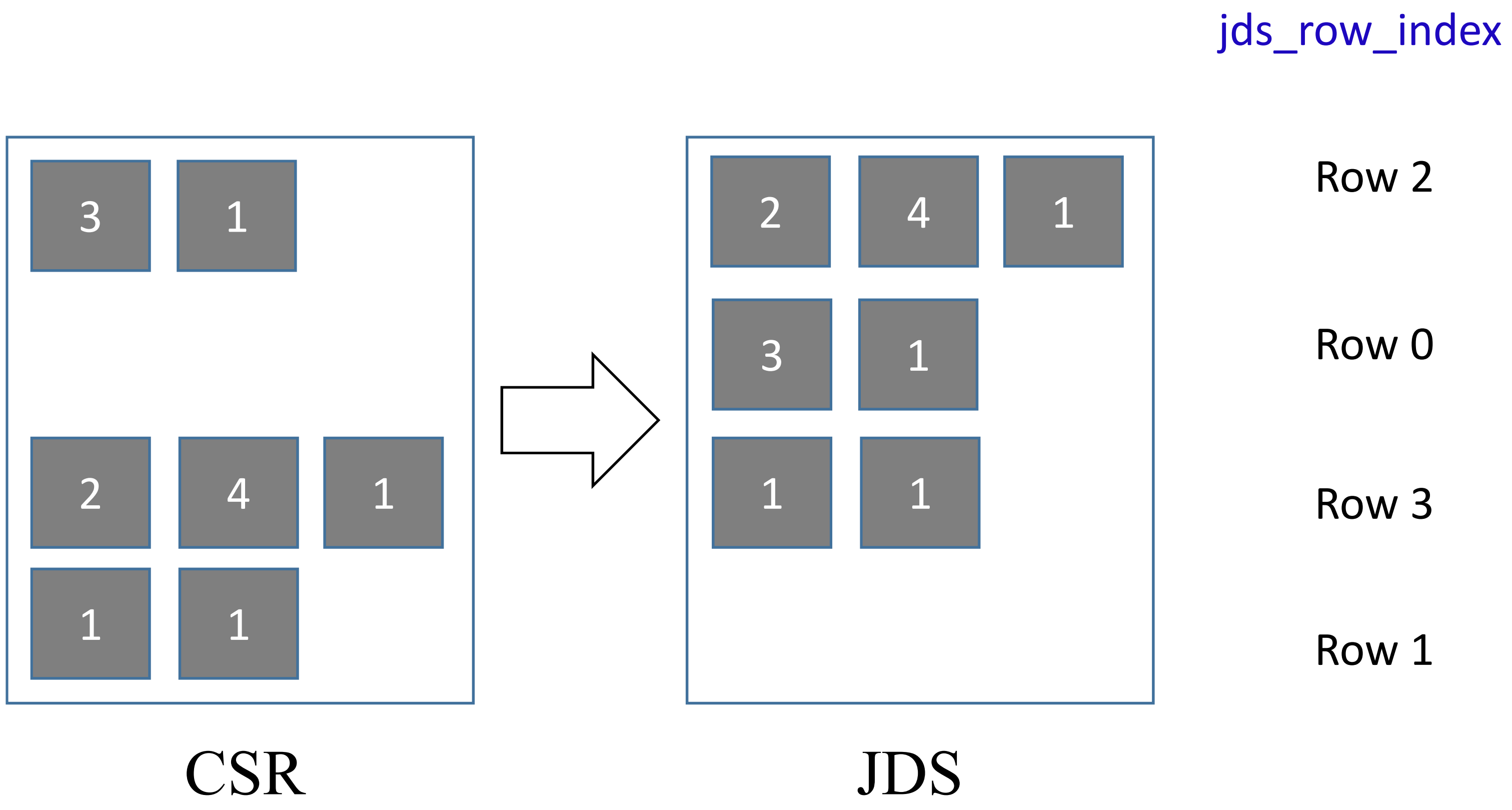
- Sort rows into descending order according to number of non-zero.

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. jds_row_index

Sorting Rows According to Length (Regularization)

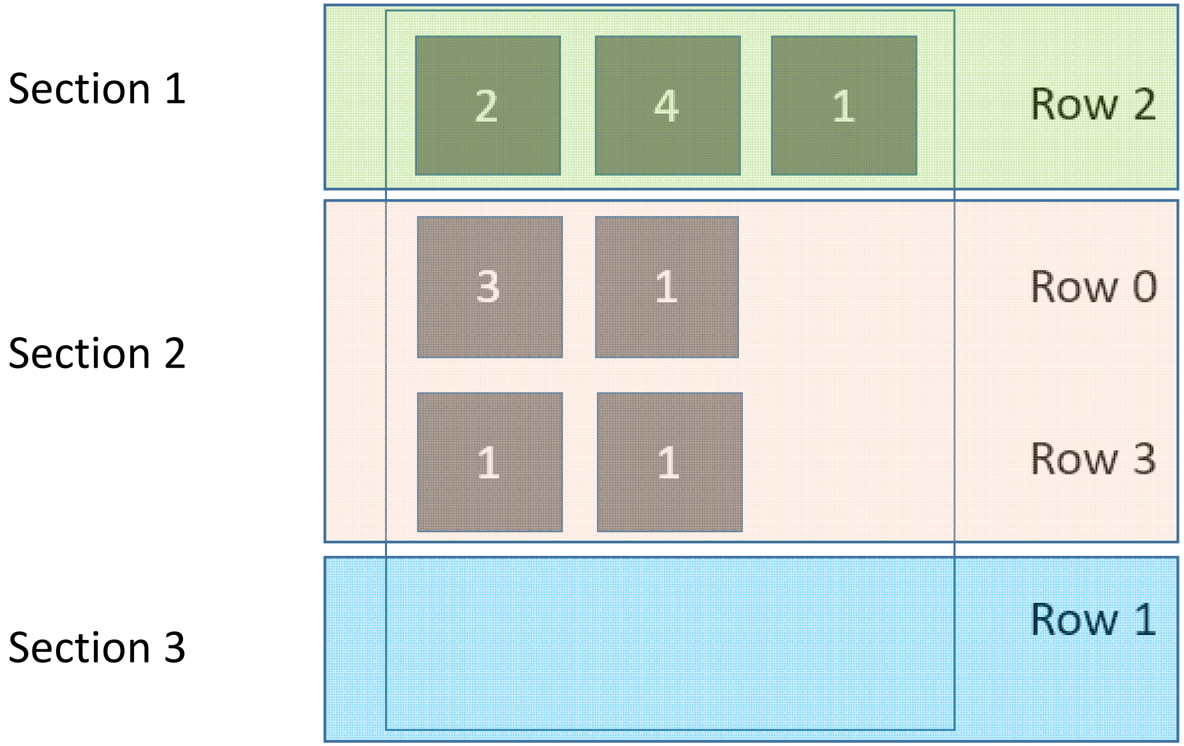


A Parallel SpMV/JDS Kernel

```
__global__ void SpMV_JDS(int num_rows, float *data,
      int *col_index, int *jds_row_ptr,int jds_row_index,
      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_index[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_index	{2,	0,	3,	1 }

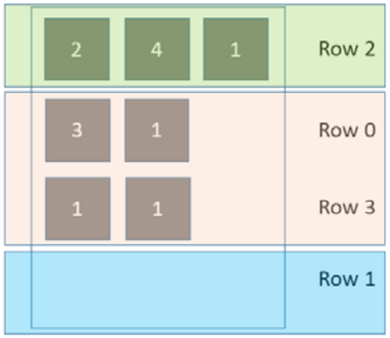
Divide into sections

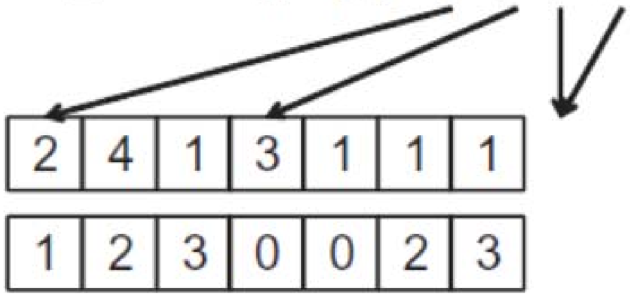


Within each section, we only need to pad the rows to match the row with the maximal number of elements in that section.

JDS format and sectioned ELL

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_index[4] { 2, 0, 3, 1 }
Section pointers Jds_section_ptr[4] { 0, 3, 7, 7 }





- We can then transpose each section independently
- and launch a separate kernel on each section.

Questions

- Whether sorting rows will lead to incorrect result?
  - No
  - The only extra step is to reorder the final solution back to the original order by using the jds_row_index array.

- Whether sorting will incur a significant overhead?
  - Iterative methods.

Take advantages of New Devices

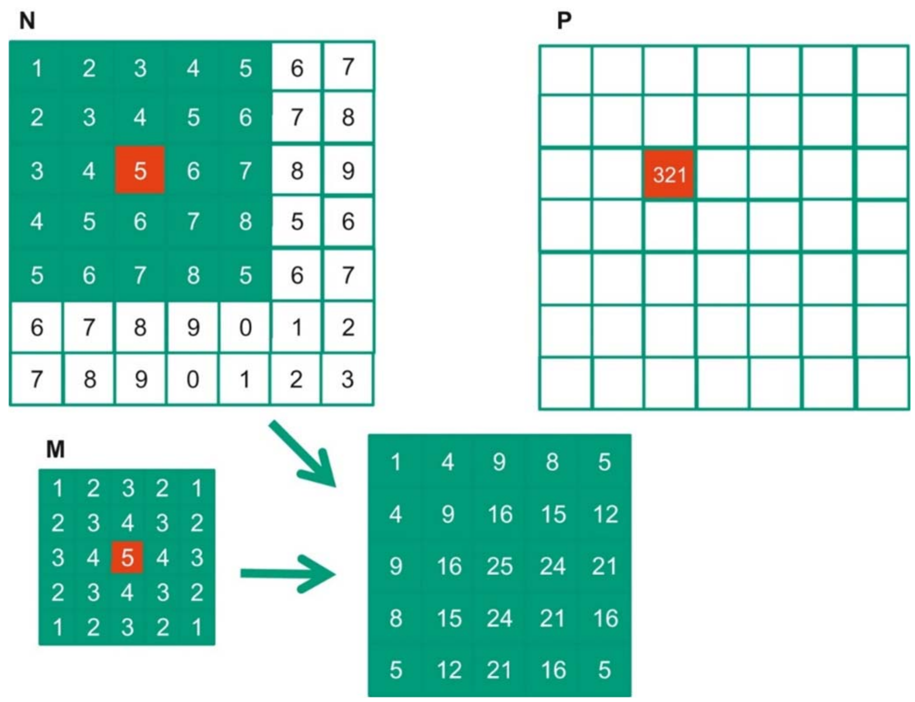
- In relatively recent devices:
  - the memory coalescing hardware has relaxed the address alignment requirement,
  - allowing the simple transposition of a JDS-CSR representation.
- In practice:
  - while sectioned JDS–ELL tends to exhibit the best performance on older CUDA devices,
  - transposed JDS-CSR tends to exhibit the best performance on Fermi and Kepler.

SpMV Summary

- 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
  - retaining regularity

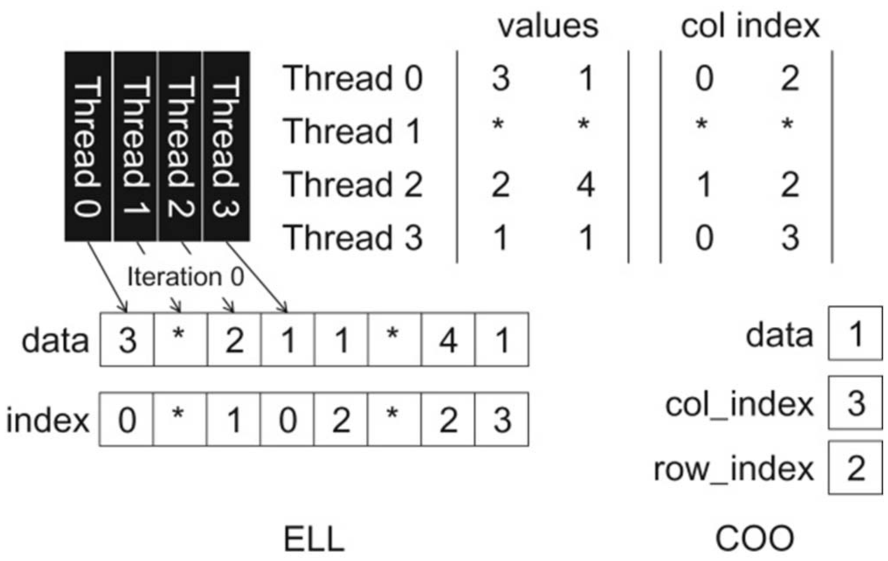
Parallel Patterns Summary

- **Convolution**
- Sparse matrix
- Prefix sum (scan)
- Histogram
- Merge sort
- Graph search



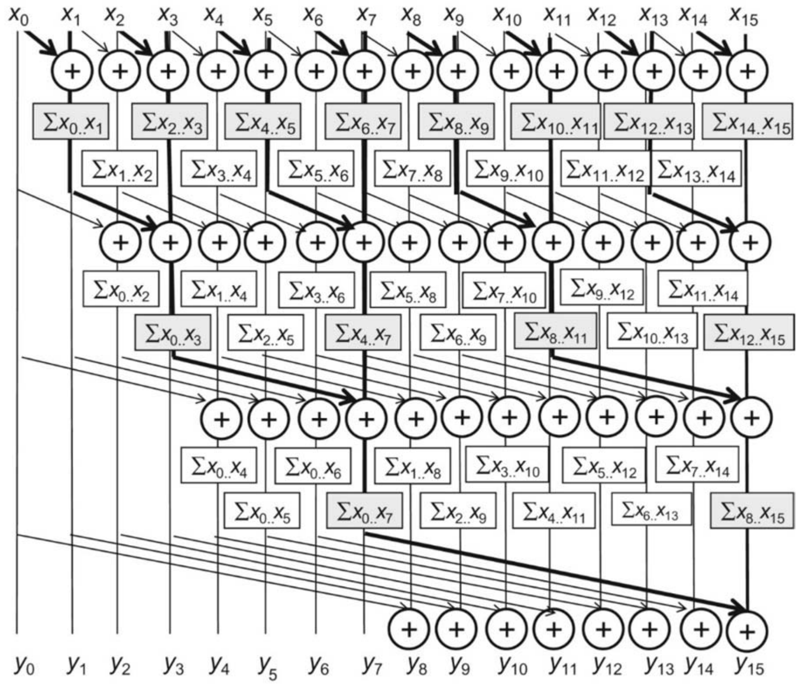
Parallel Patterns Summary

- Convolution
- **Sparse matrix**
- Prefix sum (scan)
- Histogram
- Merge sort
- Graph search



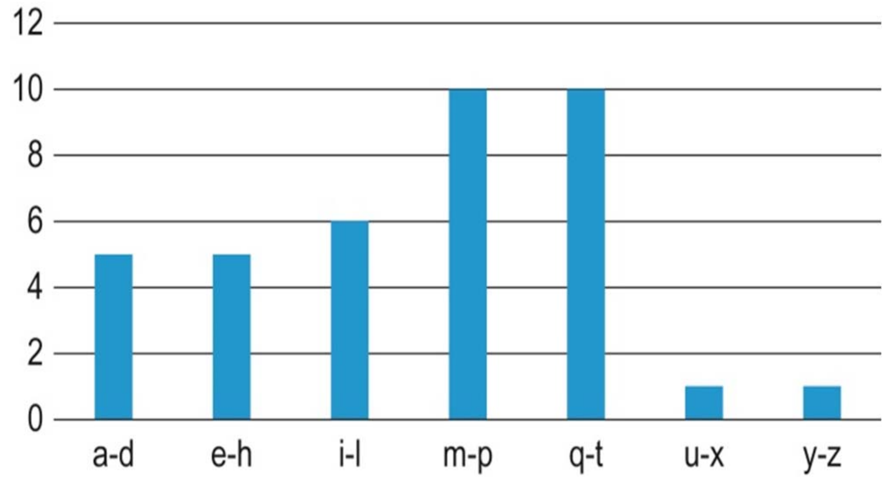
Parallel Patterns Summary

- Convolution
- Sparse matrix
- **[Prefix sum (scan)]()**
- Histogram
- Merge sort
- Graph search



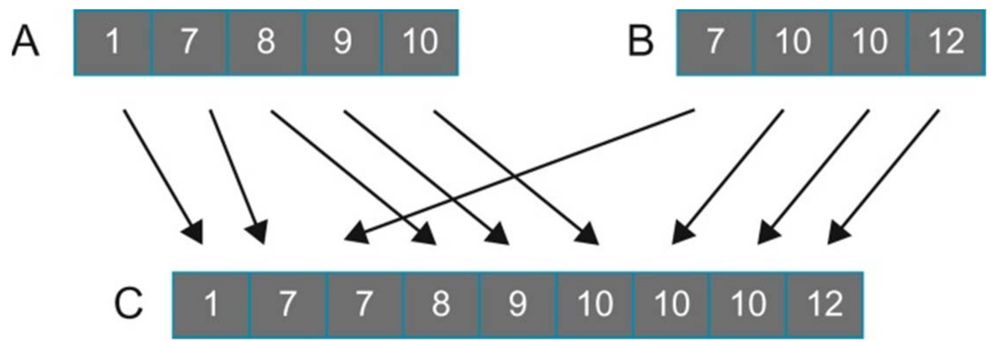
Parallel Patterns Summary

- Convolution
- Sparse matrix
- Prefix sum (scan)
- **Histogram**
- Merge sort
- Graph search



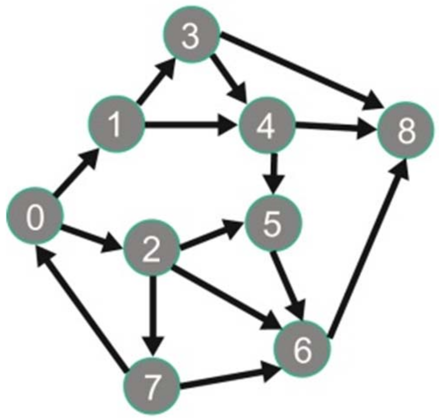
Parallel Patterns Summary

- Convolution
- Sparse matrix
- Prefix sum (scan)
- Histogram
- **[Merge sort]**
- Graph search



Parallel Patterns Summary

- Convolution
- Sparse matrix
- Prefix sum (scan)
- Histogram
- Merge sort
- **Graph search**



	0	1	2	3	4	5	6	7	8
0		1	1						
1				1	1				
2						1	1	1	
3					1				1
4						1			1
5							1		
6									1
7	1						1		
8									