

第8讲 稀疏矩阵



目标

- 学习通过以下方法来正则化非正则数据
 - 收紧上下限Limiting variations with clamping
 - 排序Sorting
 - 转置Transposition
- 学习编写基于JDS转置格式的高性能SpMV内核
- 学习在并行稀疏方法中压缩输入数据以减少内存带宽消耗的关键技术
 - 更好地利用片上存储器
 - 更少的字节被传输到片上存储器
 - 更好地利用全局内存
 - 挑战：保持正则性(regularity)

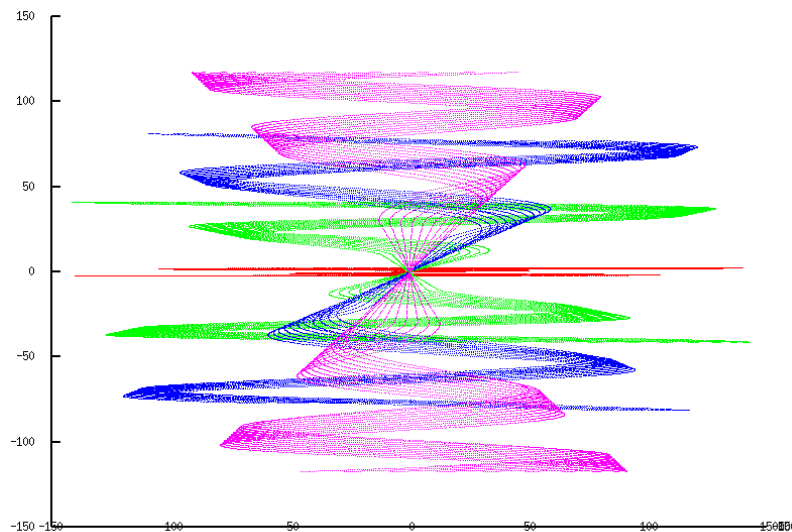


稀疏矩阵

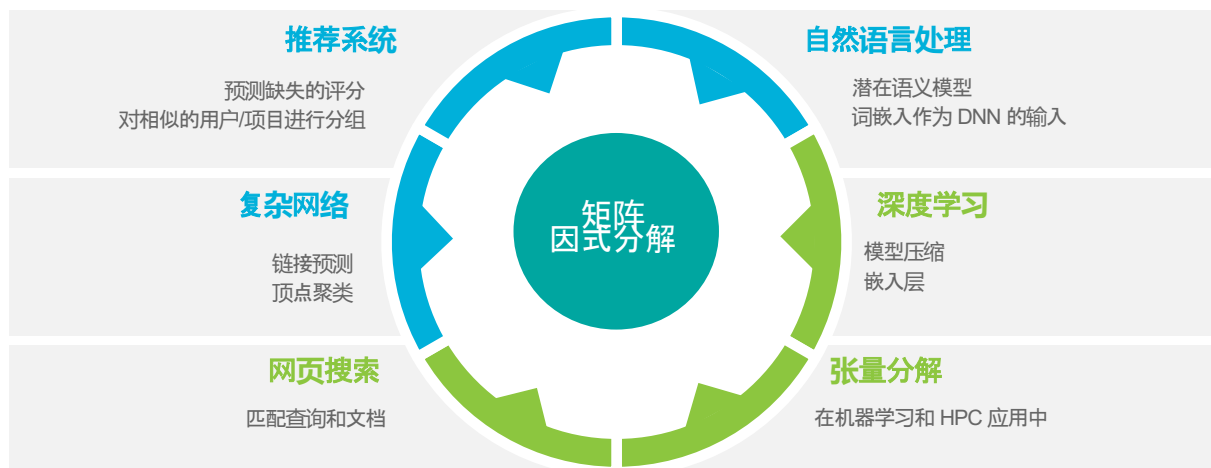
- 许多现实世界的系统本质上是稀疏的
 - 描述为稀疏矩阵的线性系统
- 求解稀疏线性系统
 - 高斯消元等传统反演算法会产生过多的“填充”元素，导致矩阵大小爆炸
 - 基于稀疏矩阵向量乘法的迭代共轭梯度求解器是最佳选择
- PDE 系统的解可以由表示为稀疏矩阵向量乘法的线性运算来表述

稀疏数据 压缩的动机

- 许多现实世界的输入是稀疏/不均匀的
- 信号样本、网格模型、交通网络、通信网络等



数据分析和AI中的稀疏矩阵



	2			4	5
	5	3		1	
		4		2	
1	3		3		4
			2		4

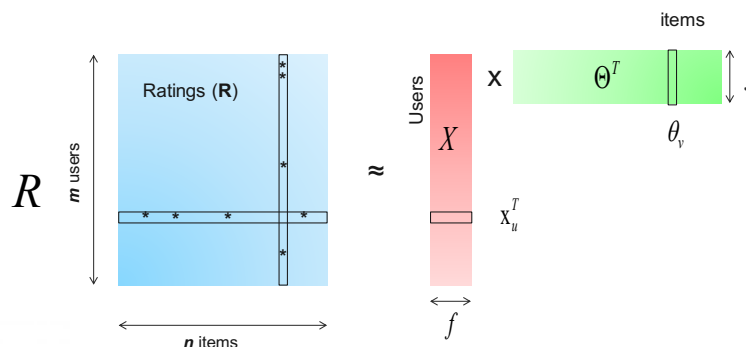
NETFLIX



amazon.com

Quora

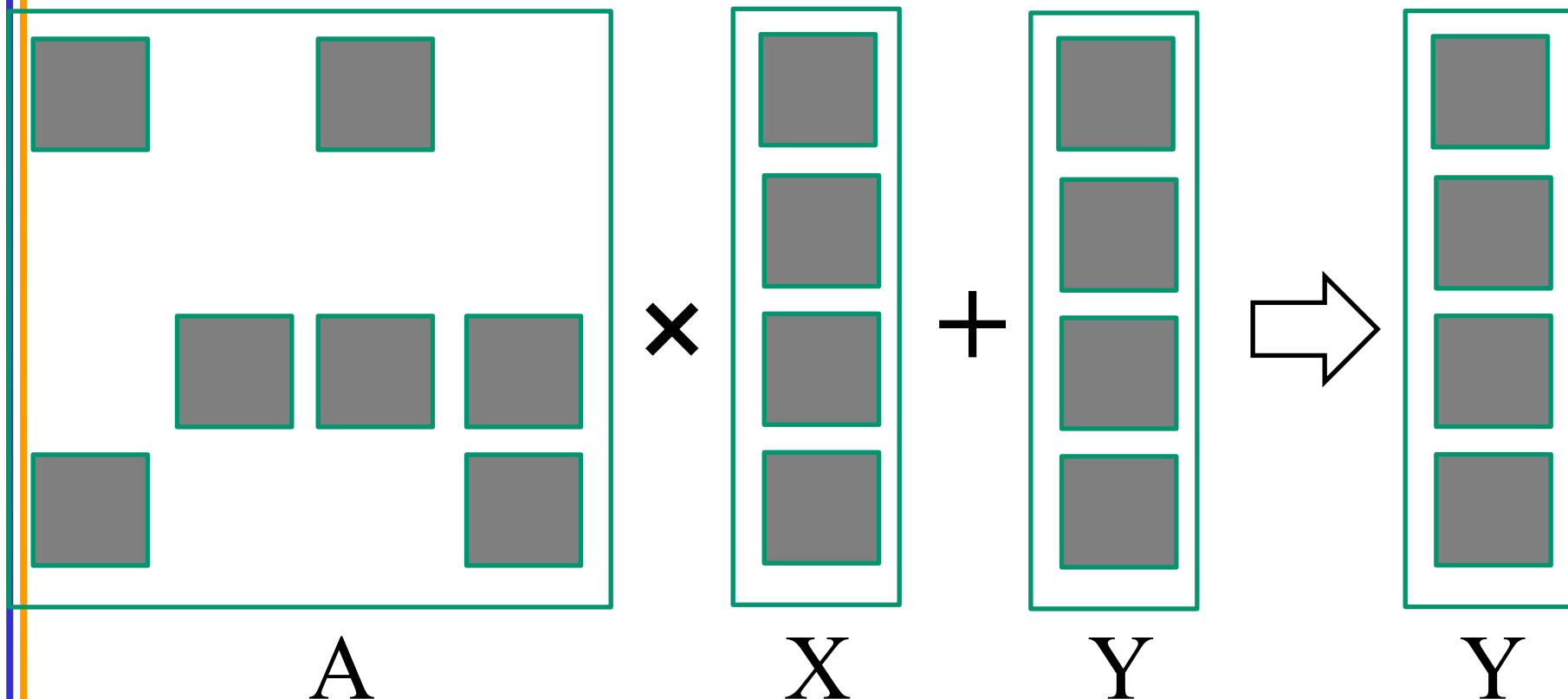
Apple Music



电子科技大学
University of Electronic Science and Technology of China

科学领域	团队数量	代码	结构化 网格	非结构化 网格	稠密矩 阵	稀疏矩 阵	N- Body	Monte Carlo	FFT	PIC	Sig I/O
气候与天气	3	CESM, GCRM, CM1/WRF, HOMME	X	X		X		X			X
等离子体/ 磁层	2	H3D(M),VPIC, OSIRIS, Magtail/UPIC	X				X		X		X
恒星大气和超新星	5	PPM, MAESTRO, CASTRO, SEDONA, ChaNGa, MS-FLUKSS	X			X	X	X		X	X
宇宙学	2	Enzo, pGADGET	X			X	X				
燃烧/ 涡流	2	PSDNS, DISTUF	X						X		
广义相对论	2	Cactus, Harm3D, LazEV	X			X					
分子动力学	4	AMBER, Gromacs, NAMD, LAMMPS				X	X		X		
量子化学	2	SIAL, GAMESS, NWChem			X	X	X	X			X
材料科学	3	NEMOS, OMEN, GW, QMCPACK			X	X	X	X			
地震/ 地震学	2	AWP-ODC, HERCULES, PLSQR, SPECFEM3D	X	X			X				X
量子色动力学	1	Chroma, MILC, USQCD	X		X	X					
社会网络	1	EPISIMDEMICS									
进化论	1	Eve									
工程/ 系统的系统	1	GRIPS,Revisit						X			6
计算机科学	1			X	X	X			X		X

稀疏矩阵向量乘法(SpMV)



挑战

- 相较于稠密矩阵乘法，SpMV
 - 是不规则的或非结构化的
 - 很少有输入数据重用
- 最大化性能的关键
 - 最大化正则性regularity（通过减少发散divergence和负载不平衡）
 - 最大化 DRAM 突发(burst)利用率（布局安排）

一个简单的并行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

- 每个线程处理一行



压缩稀疏行 (CSR) 格式

CSR表示法

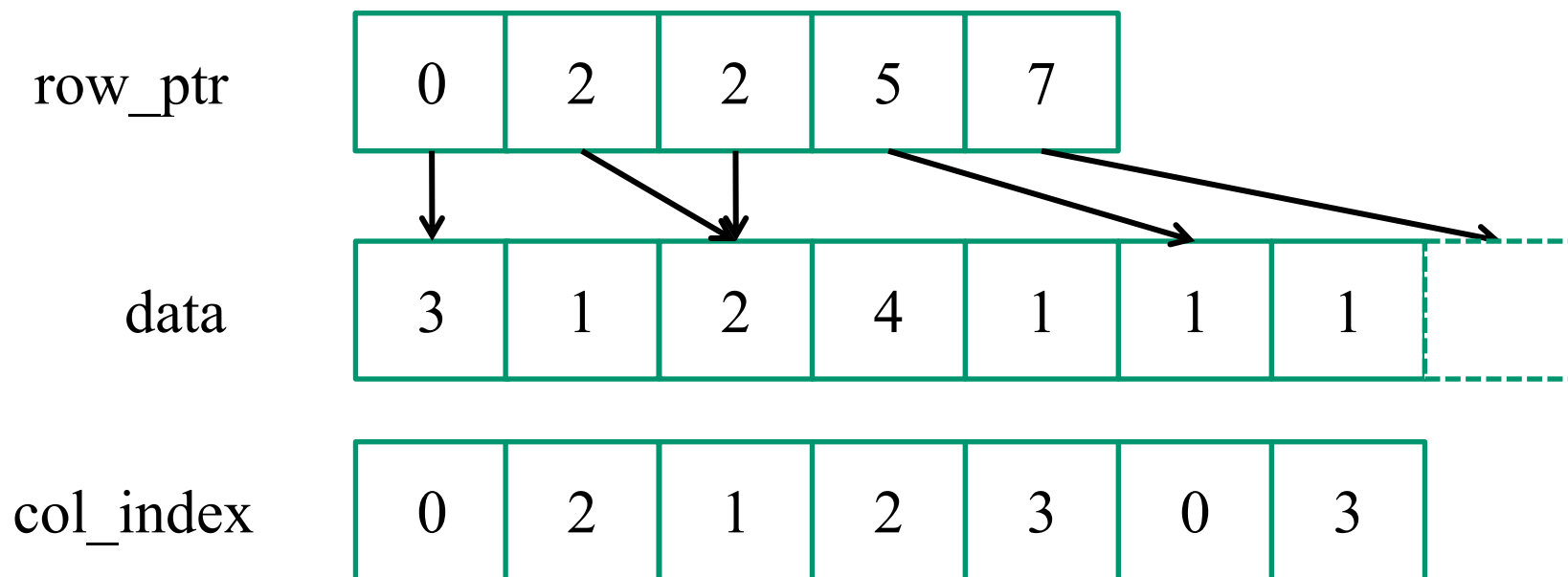
		Row 0	Row 2	Row 3
非0值	data[7]	{ 3, 1,	2, 4, 1,	1, 1 }
列号	col_index[7]	{ 0, 2,	1, 2, 3,	0, 3 }
行指针	ptr[5]	{ 0, 2, 2, 5, 7 }		

稠密表示法

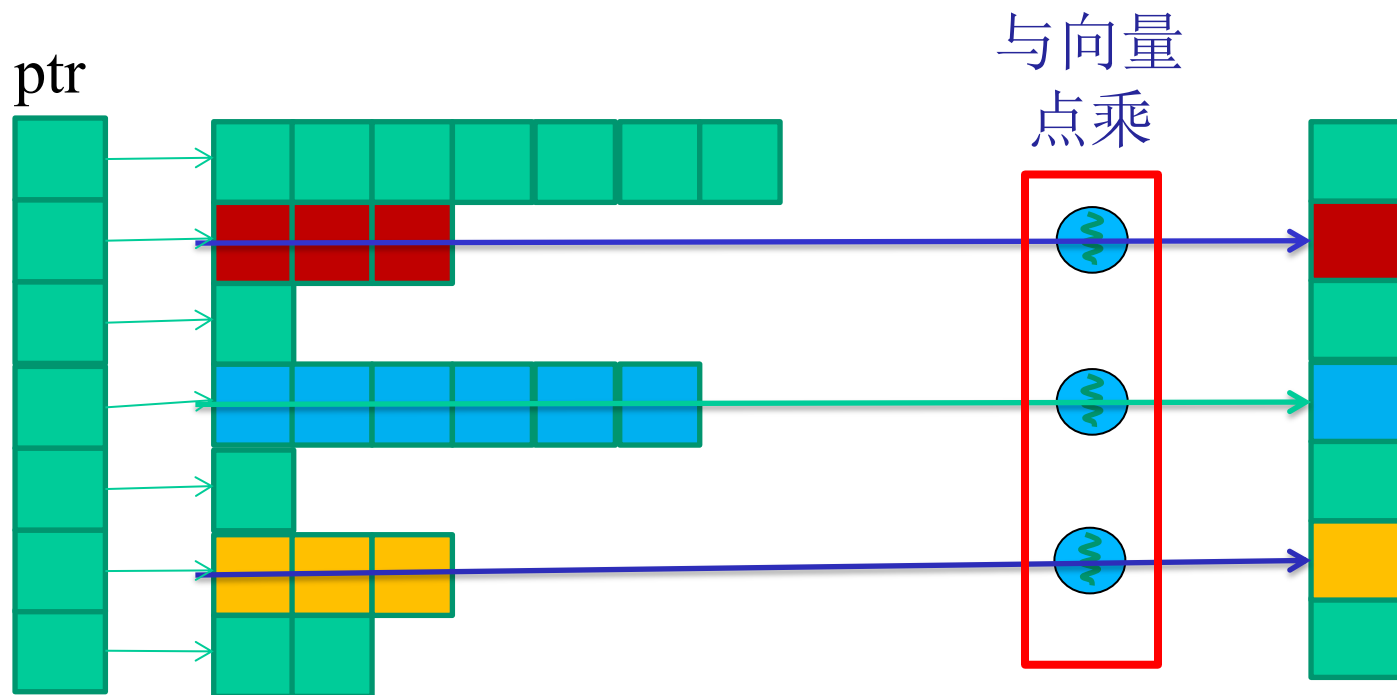
Row 0	3	0	1	0	Thread 0
Row 1	0	0	0	0	Thread 1
Row 2	0	2	4	1	Thread 2
Row 3	1	0	0	1	Thread 3



CSR数据布局



CS内核设计



并行SpMV/CSR内核 (CUDA)

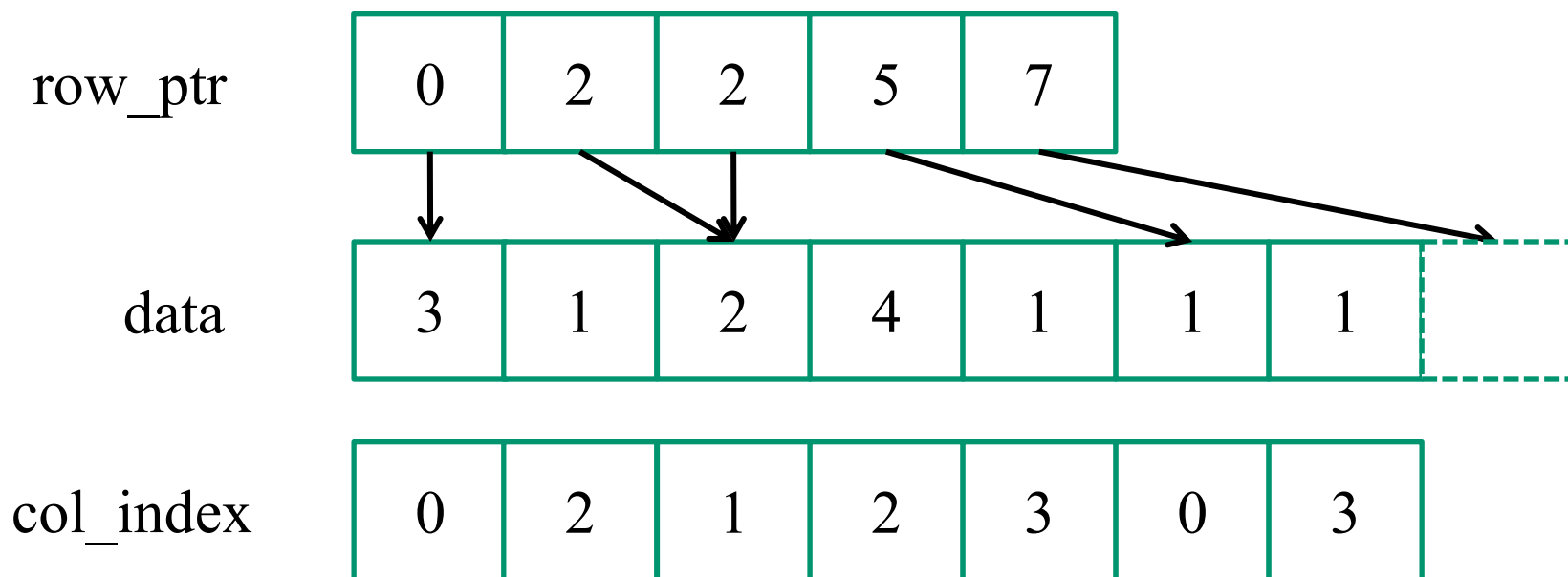
```
1. __global__ void SpMV_CSR(int num_rows, float *data,  
    int *col_index, int *row_ptr, float *x, float *y) {  
2.     int row = blockIdx.x * blockDim.x + threadIdx.x;  
3.     if (row < num_rows) {  
4.         float dot = 0;  
5.         int row_start = row_ptr[row];  
6.         int row_end = row_ptr[row+1];  
7.         for (int elem = row_start; elem < row_end; elem++) {  
8.             dot += data[elem] * x[col_index[elem]];  
9.         }  
        y[row] = dot;  
    }  
}
```

		Row 0	Row 2	Row 3
非0值	data[7]	{ 3, 1,	2, 4, 1,	1, 1 }
列号	col_index[7]	{ 0, 2,	1, 2, 3,	0, 3 }
行指针	row_ptr[5]	{ 0, 2,	2, 5, 7 }	



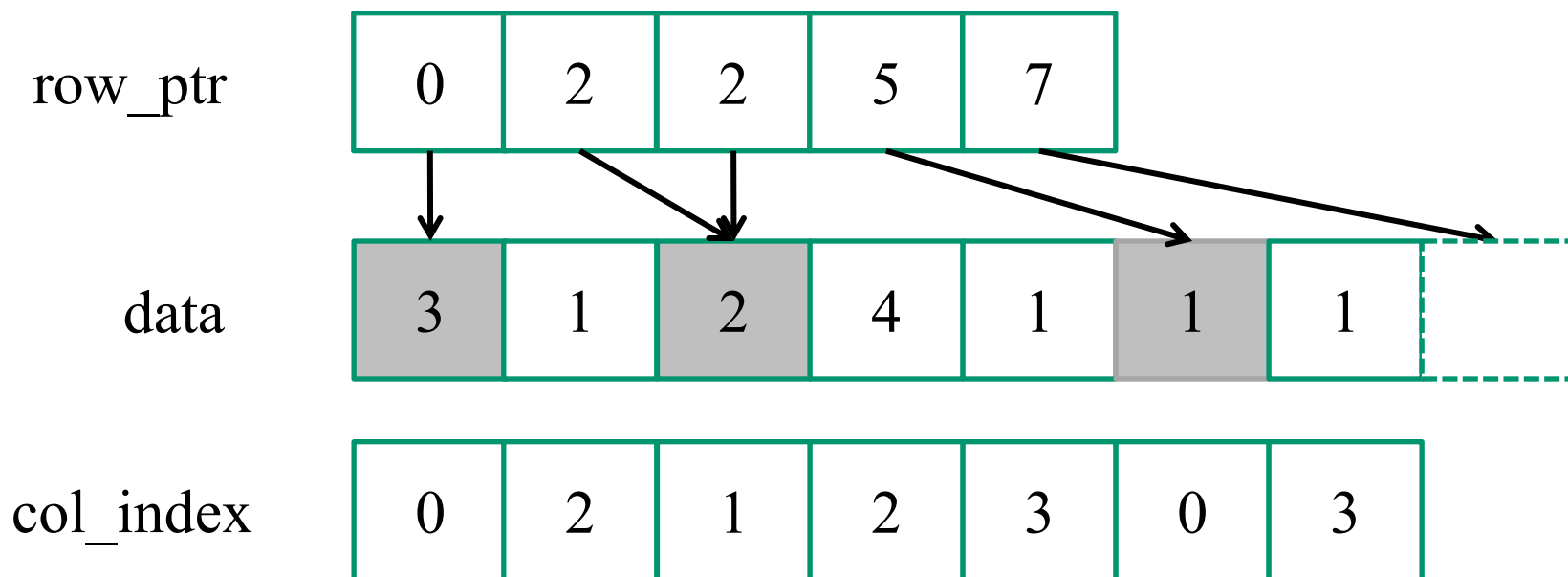
CSR内核控制发散

- 线程在内核 for 循环中执行不同次数的迭代



CSR 内核内存发散（未合并访问）

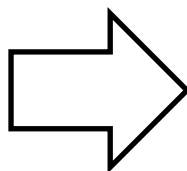
- 相邻线程访问不相邻的内存位置
 - 灰色元素在第0次迭代中被所有线程访问



使用 ELL(PACK) 格式正则化 SpMV

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

带有填充的 CSR



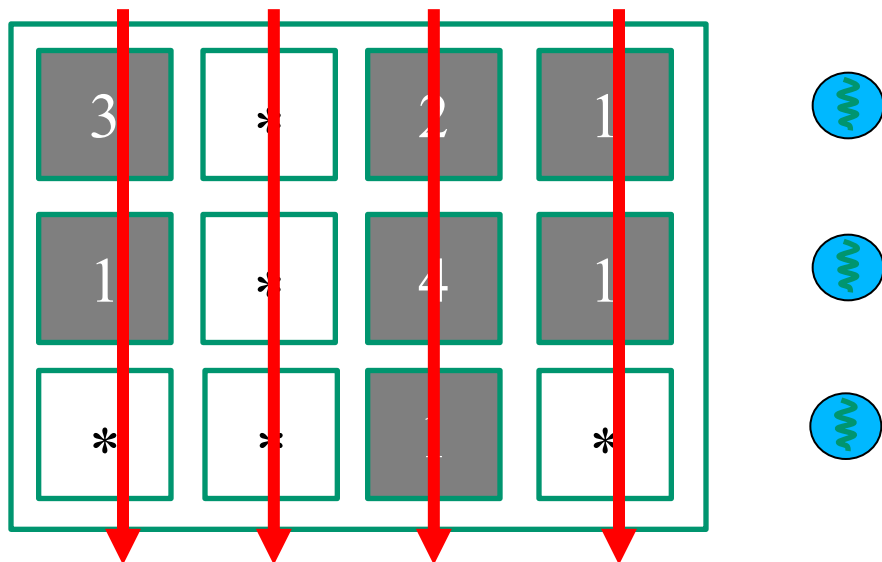
3	*	2	1
1	*	4	1
*	*	1	*

转置后

- 将所有行填充至相同长度
 - 当少数行比其他行长得多时效率很低
- **转置** (列优先) 以提高DRAM效率
- 数据及col_index都被填充或转置



ELL 内核设计

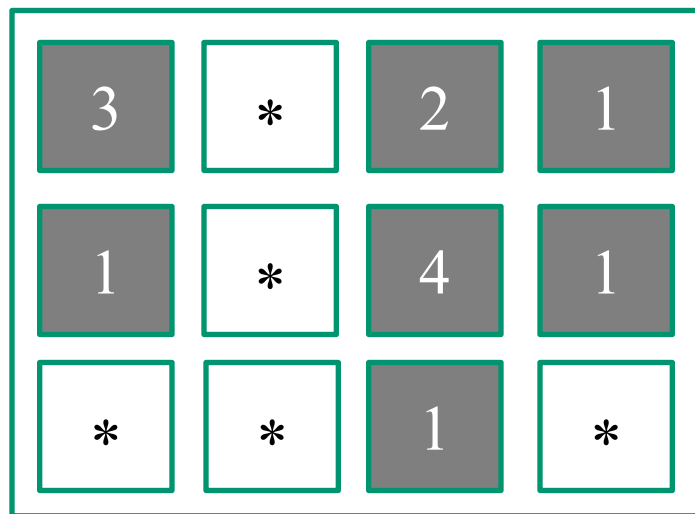
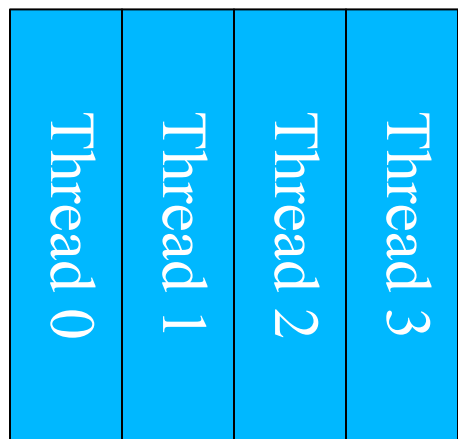


一个并行SpMV/ELL内核

```
1. __global__ void SpMV_ELL(int num_rows, float *data,
    int *col_index, int num_elem, float *x, float *y) {
2.     int row = blockIdx.x * blockDim.x + threadIdx.x;
3.     if (row < num_rows) {
4.         float dot = 0;
5.         for (int i = 0; i < num_elem; i++) {
6.             dot += data[row+i*num_rows]*x[col_index[row+i*num_rows]];
7.             }
8.         y[row] = dot;
9.     }
10. }
```



ELL 的合并访存



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

col_index



电子科技大学
University of Electronic Science and Technology of China

坐标 (Coordinate, COO) 格式

- 显式列出每个非零元素的列和行索引

		Row 0	Row 2	Row 3
非0值	data[7]	{ 3, 1,	2, 4, 1,	1, 1 }
列号	col_index[7]	{ 0, 2,	1, 2, 3,	0, 3 }
行号	row_index[7]	{ 0, 0,	2, 2, 2,	3, 3 }

COO 允许重新排序元素

		Row 0	Row 2	Row 3
非0值	data[7]	{ 3, 1,	2, 4, 1,	1, 1 }
列号	col_index[7]	{ 0, 2,	1, 2, 3,	0, 3 }
行号	row_index[7]	{ 0, 0,	2, 2, 2,	3, 3 }

非0值	data[7]	{ 1 1, 2, 4, 3, 1 1 }
列号	col_index[7]	{ 0 2, 1, 2, 0, 3, 3 }
行号	row_index[7]	{ 3 0, 2, 2, 0, 2, 3 }



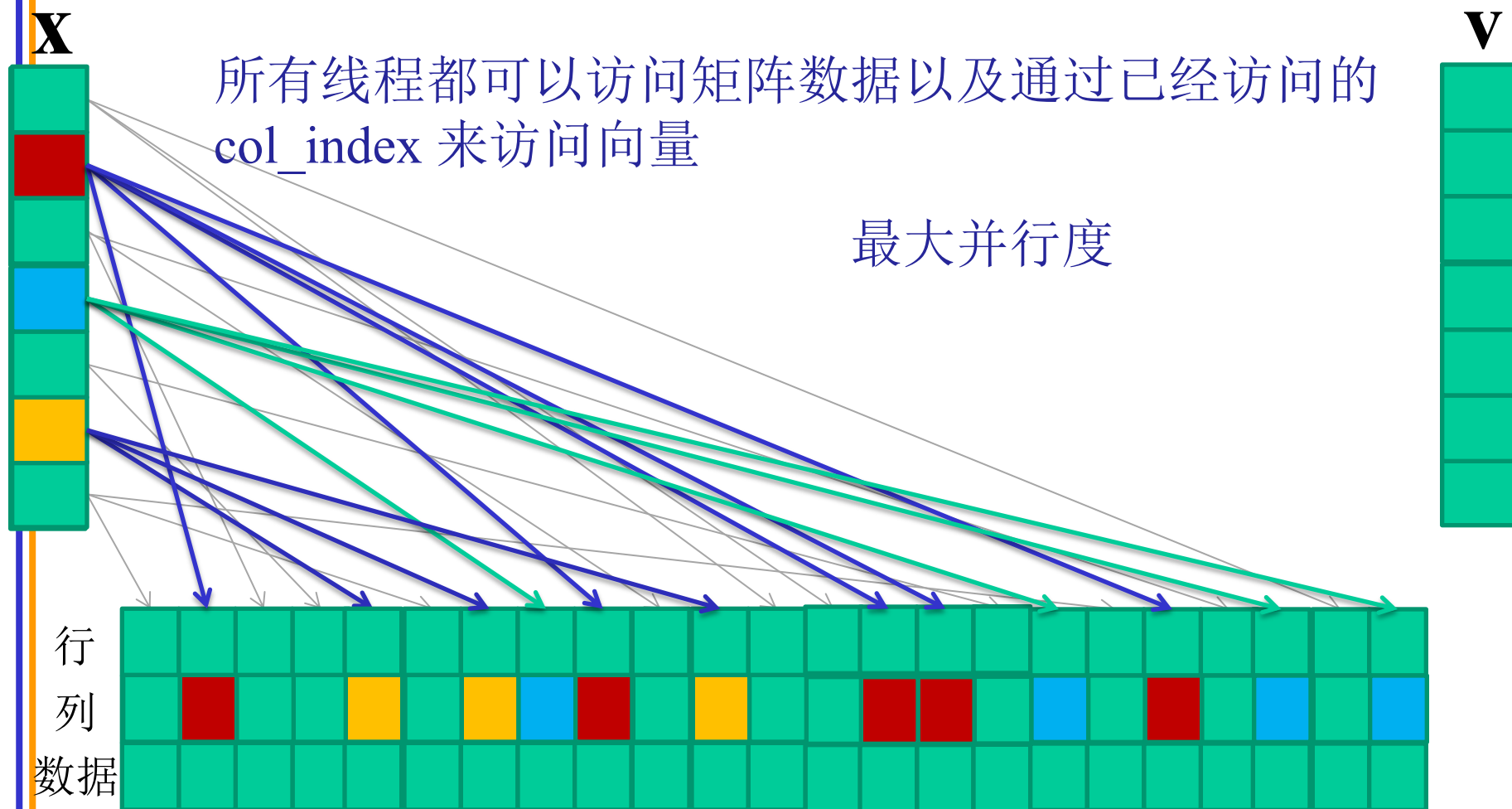
```
1.    for (int i = 0; i < num_elem; row++)  
2.        y[row_index[i]] += data[i] * x[col_index[i]];
```

实现 SpMV/COO 的顺序循环



COO内核设计

访问输入矩阵和向量



非0值	data[7]	{	3, 1,	2, 4, 1,	1, 1 }
列号	col_index[7]	{	0, 2,	1, 2, 3,	0, 3 }
行号	row_index[7]	{	0, 0,	2, 2, 2,	3, 3 }

COO内核设计 累加到输出向量

X



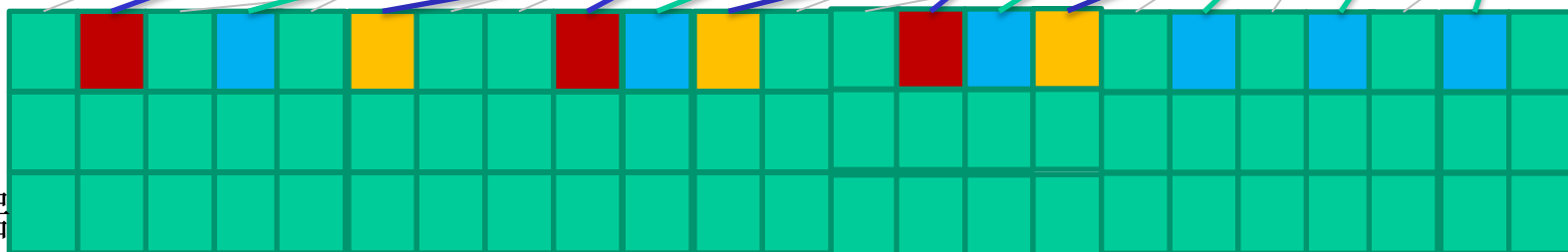
每个线程使用其元素的 row_index 累加到输出 向量Y 的其中一个元素

需要原子操作！

V



行
列
数据

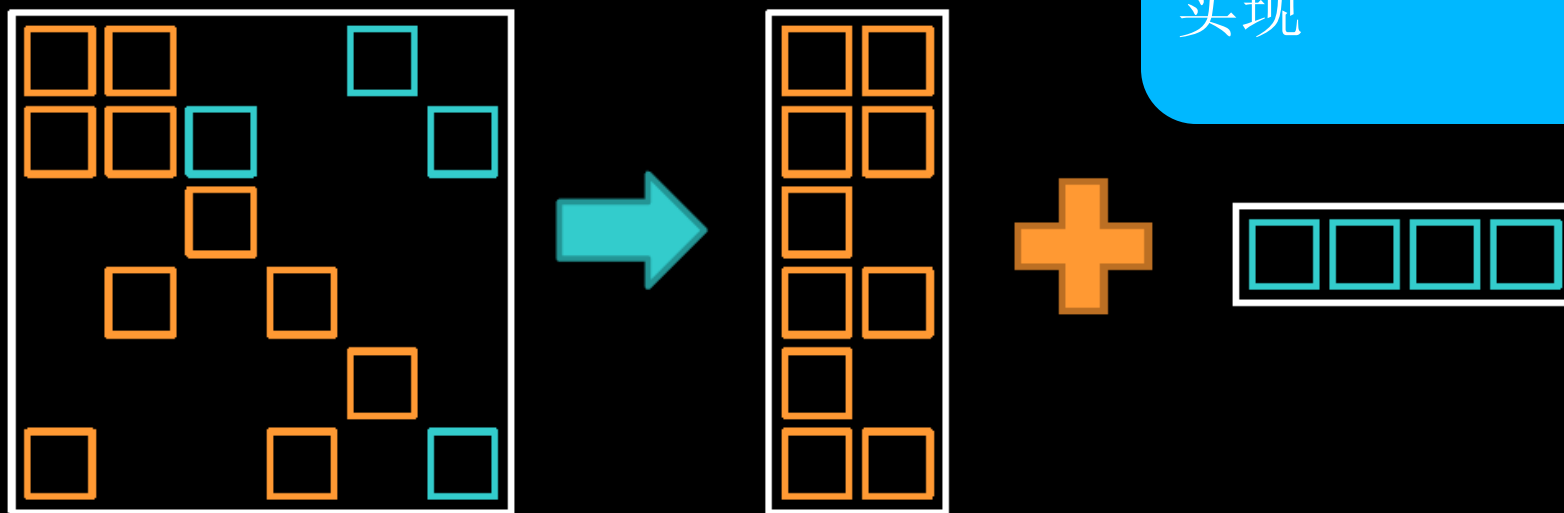


Hybrid Format

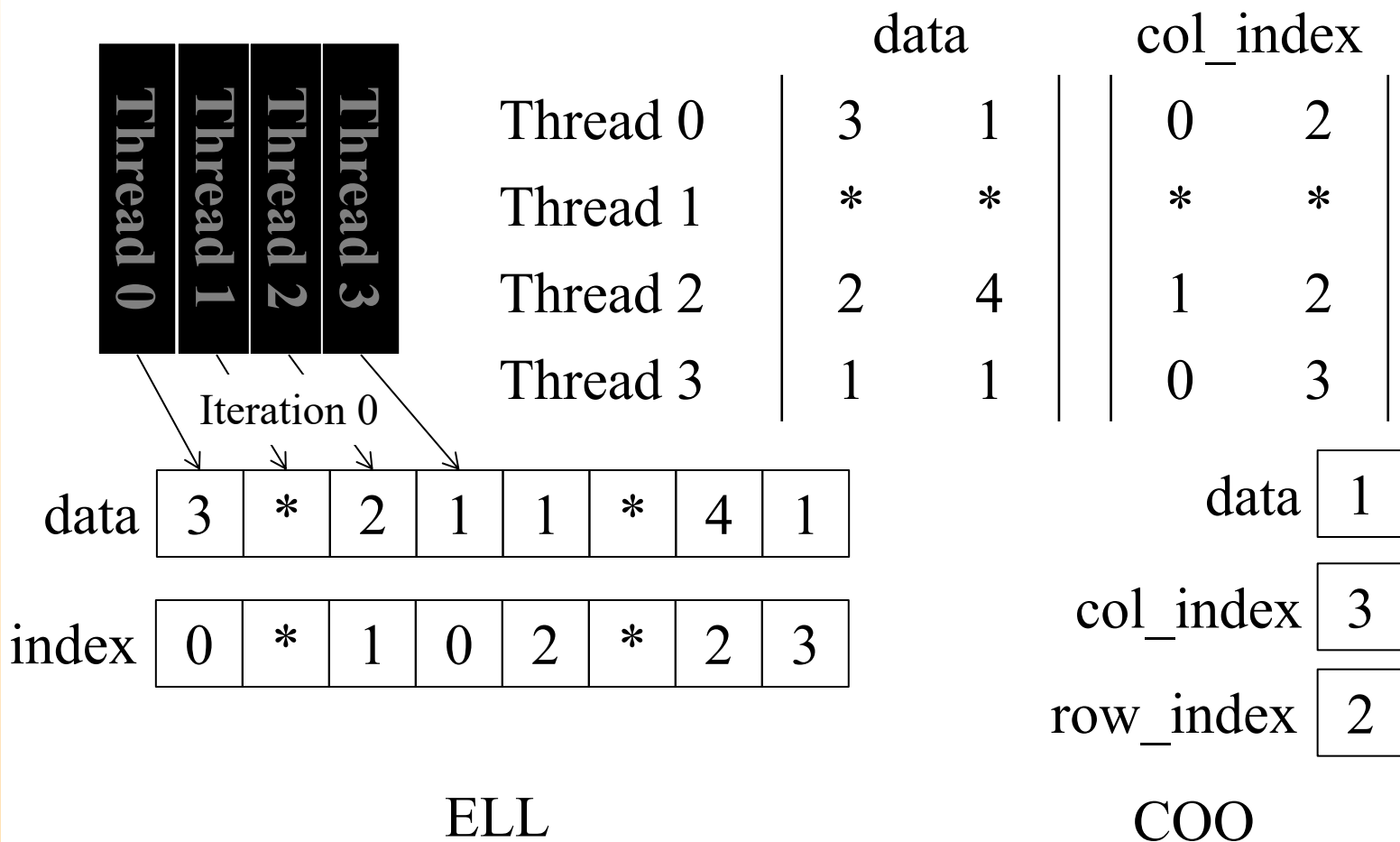


- ELL handles *typical* entries
- COO handles *exceptional* entries
 - Implemented with segmented reduction

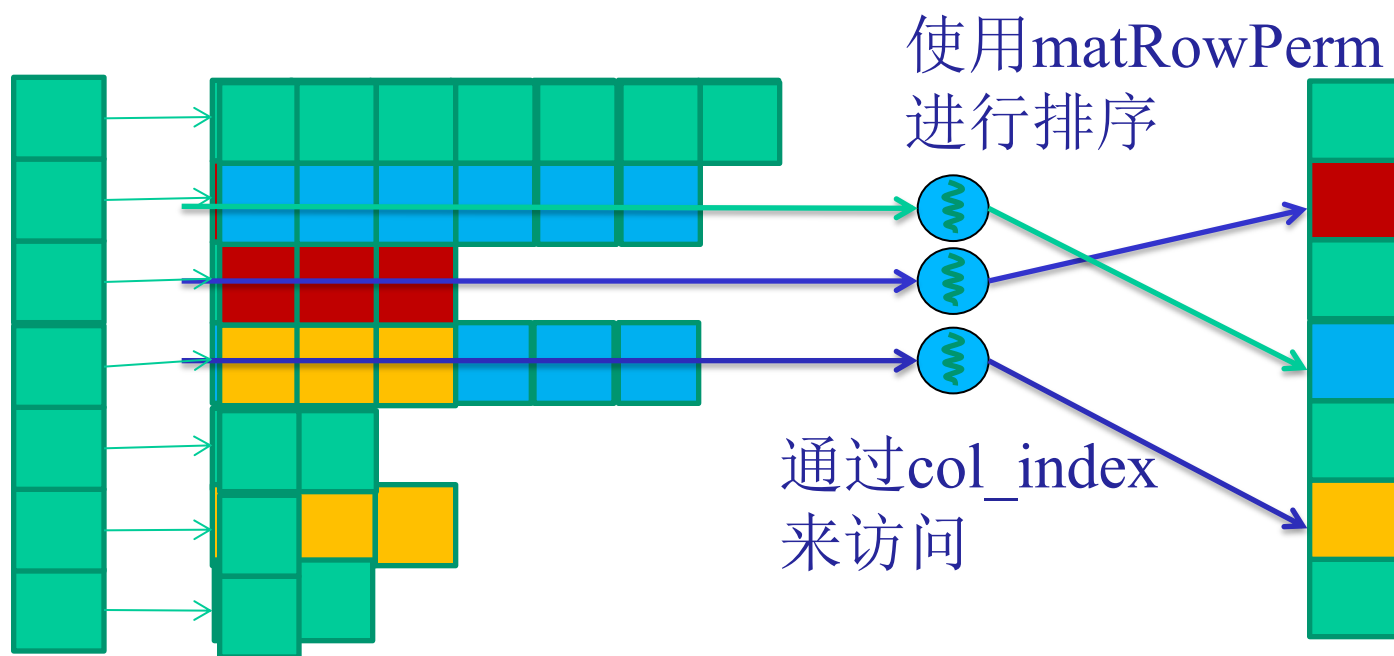
在实践中通常在
顺序主机代码中
实现



使用混合格式减少填充



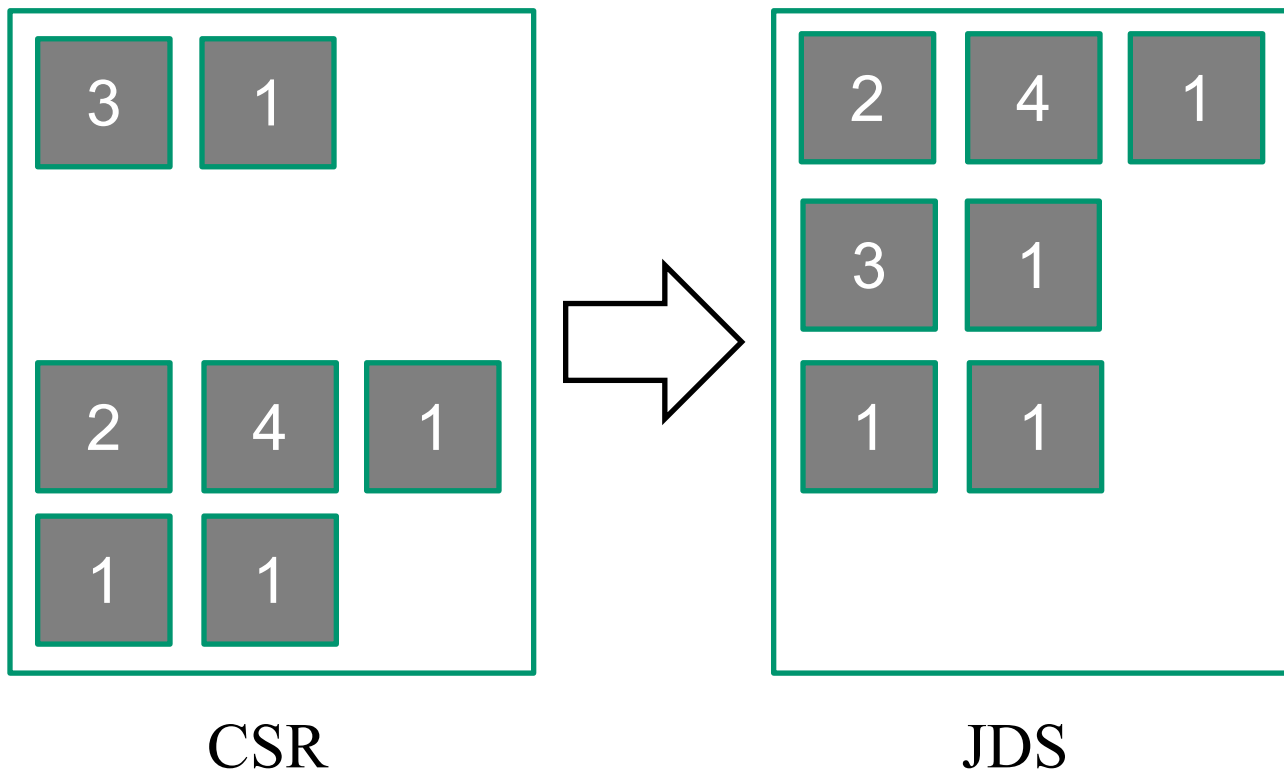
负载均衡的JDS (锯齿状对角线稀疏, Jagged Diagonal Sparse) 内核设计



根据非零的数量将行降序排列。
跟踪原始行号，以便正确生成输出向量。



根据长度对行进行排序（正则化）



CSR 到 JDS 的转换

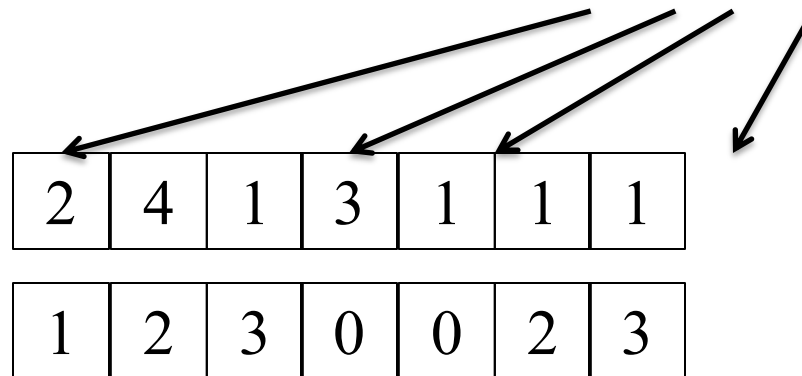
	Row 0	Row 2	Row 3
非0值 data[7]	{ 3, 1, }	{ 2, 4, 1, }	{ 1, 1 }
列号 col_index[7]	{ 0, 2, }	{ 1, 2, 3, }	{ 0, 3 }
行指针 row_ptr[5]	{ 0, 2, 2, }		{ 5, 7 }

	Row 2	Row 0	Row 3
非0值 data[7]	{ 2, 4, 1, }	{ 3, 1, }	{ 1 1 }
列号 col_index[7]	{ 1, 2, 3, }	{ 0, 2, }	{ 0, 3 }
JDS行指针 jds_row_ptr[5]	{ 0, 3, 5, 7, 7 }		
JDS 行号 jds_row_perm[4]	{ 2, 0, 3, 1 }		



JDS 总结

非0值	data[7]	{ 2, 4, 1, 3, 1, 1, 1 }
列号	Jds_col_index[7]	{ 1, 2, 3, 0, 2, 0, 3 }
JDS 行号	Jds_row_perm[4]	{ 2, 0, 3, 1 }
JDS 行指针	Jds_row_ptr[5]	{ 0, 3, 5, 7, 7 }



一个并行SpMV/JDS内核

```
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.         }
10.        y[jds_row_perm[row]] = dot;
11.    }
12. }
```

	Row 2	Row 0	Row 3	
非0值 data[7]	{ 2, 4, 1,	3, 1,	1 1	}
列号 col_index[7]	{ 1, 2, 3,	0, 2,	0, 3	}
JDS行指针 jds_row_ptr[5]	{0,	3,	5,	7,7 }
JDS行号 jds_row_perm[4]	{2,	0,	3,	1 }

JDS vs. CSR – 控制发散

- 在 JDS 内核 for-loop 中，线程仍然执行不同次数的迭代
 - 但由于排序，相邻线程倾向于执行相似数量的迭代。
 - 更好的线程利用率，更少的控制发散

非0值 data[7] { 2, 4, 1, 3, 1, 1, 1 }
列号 col_index[7] { 1, 2, 3, 0, 2, 0, 3 }
JDS行号 Jds_row_perm[4] { 2, 0, 3, 1 }
JDS行指针 Jds_row_ptr[5] { 0, 3, 5, 7, 7 }

data



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

col_index

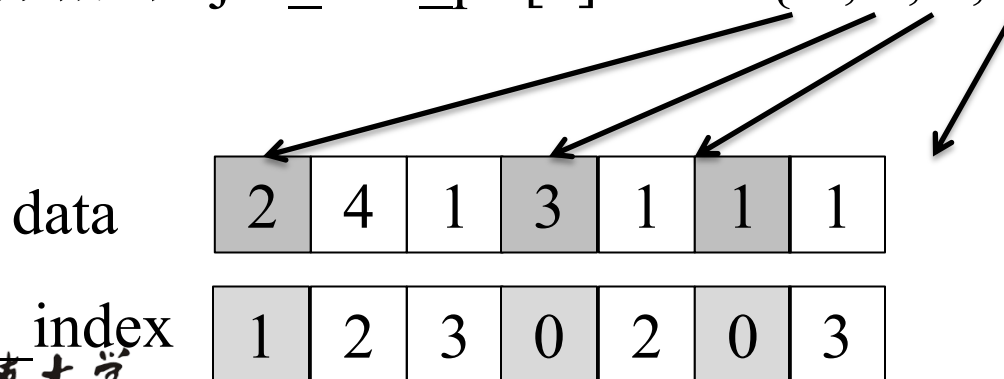
1	2	3	0	2	0	3
---	---	---	---	---	---	---



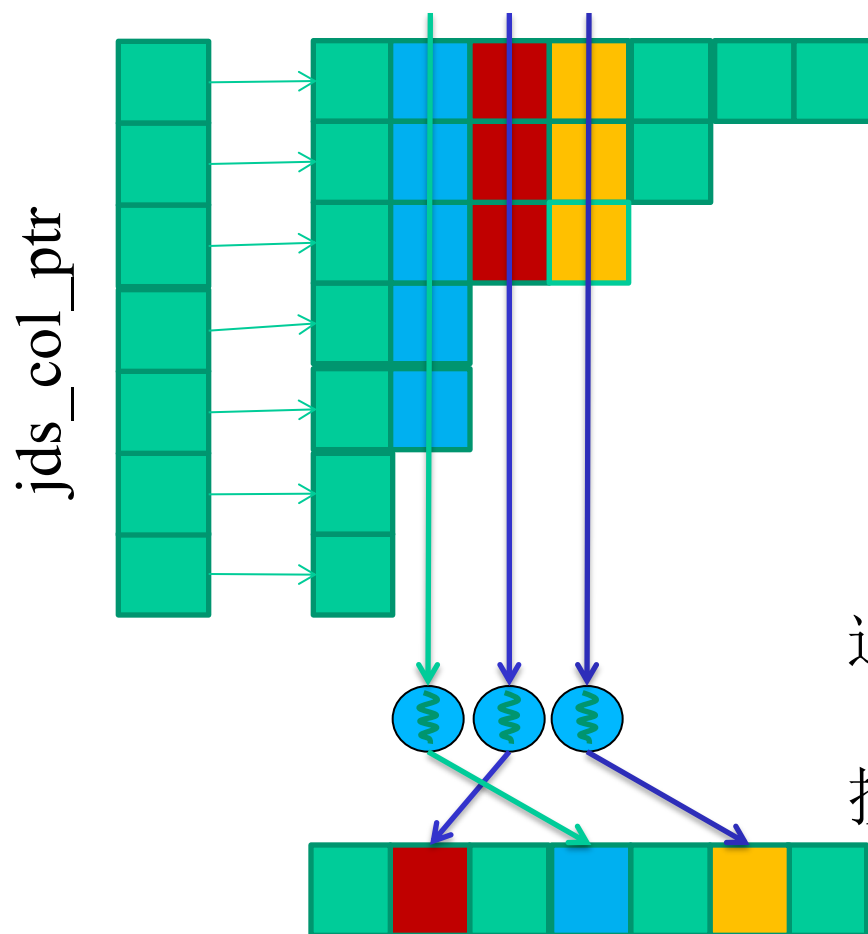
JDS vs. CSR 内存发散

- 相邻线程仍然访问不相邻的内存位置

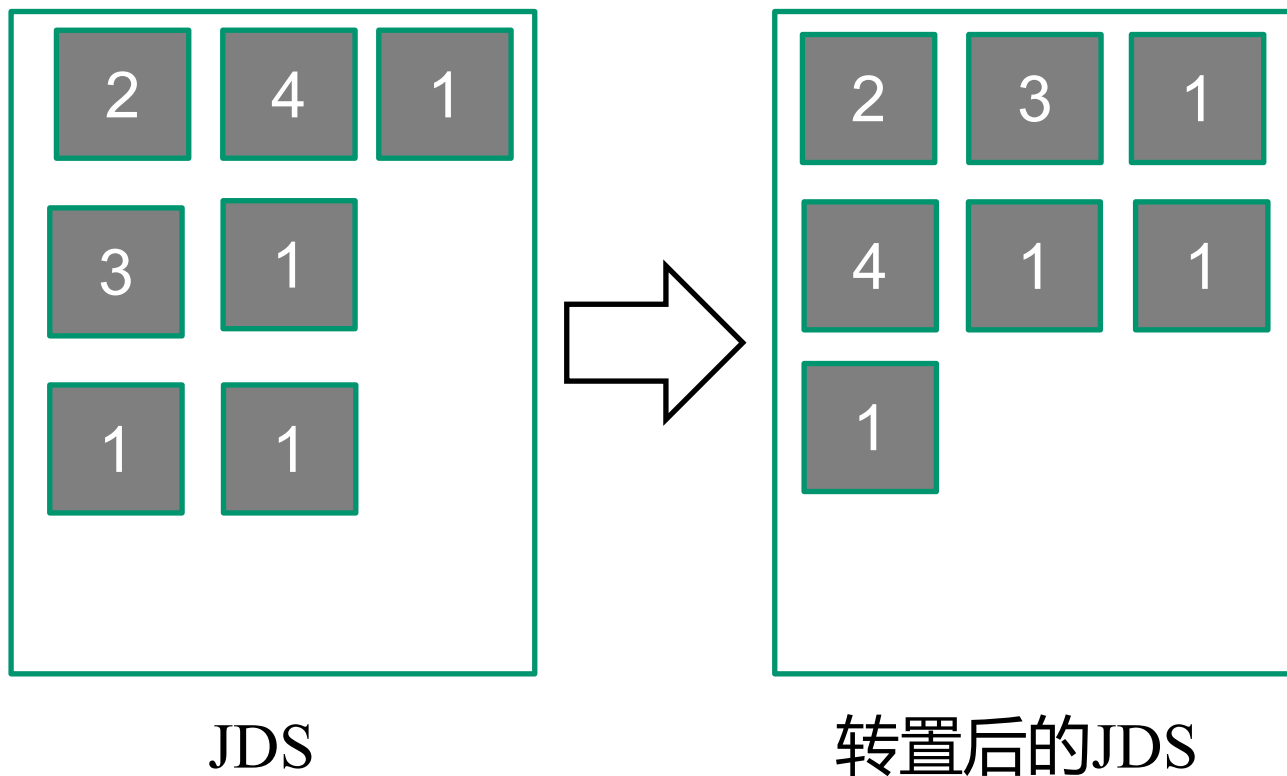
非0值 data[7] { 2, 4, 1, 3, 1, 1, 1 }
列号 col_index[7] { 1, 2, 3, 0, 2, 0, 3 }
JDS行号 jds_row_perm[4] { 2, 0, 3, 1 }
JDS行指针 jds_row_ptr[5] { 0, 3, 5, 7, 7 }



带转置的 JDS



合并访存的转置

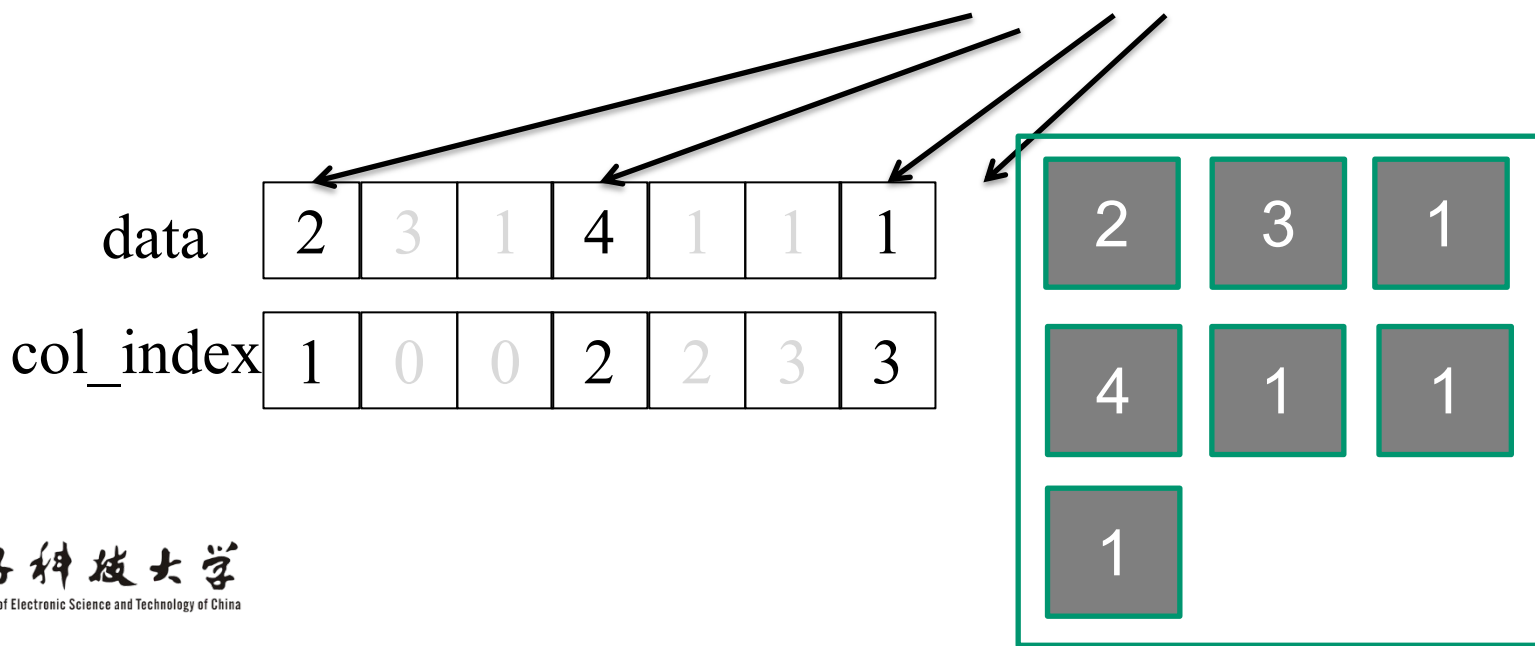


带有转置布局的 JDS 格式

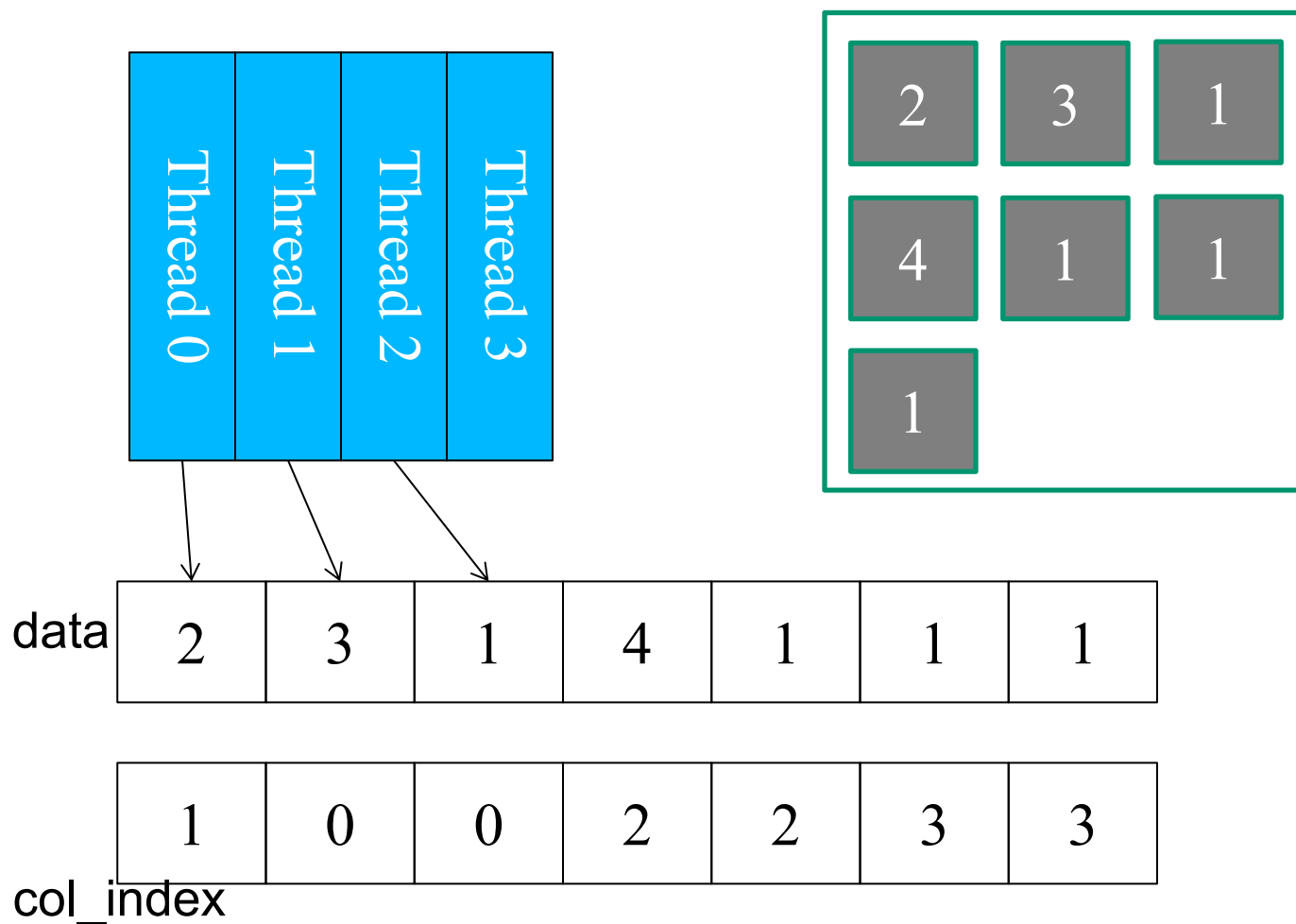
Row 0	3	0	1	0	Thread 0
Row 1	0	0	0	0	Thread 1
Row 2	0	2	4	1	Thread 2
Row 3	1	0	0	1	Thread 3

JDS 行号 Jds_row_perm[4] { 2, 0, 3, 1 }

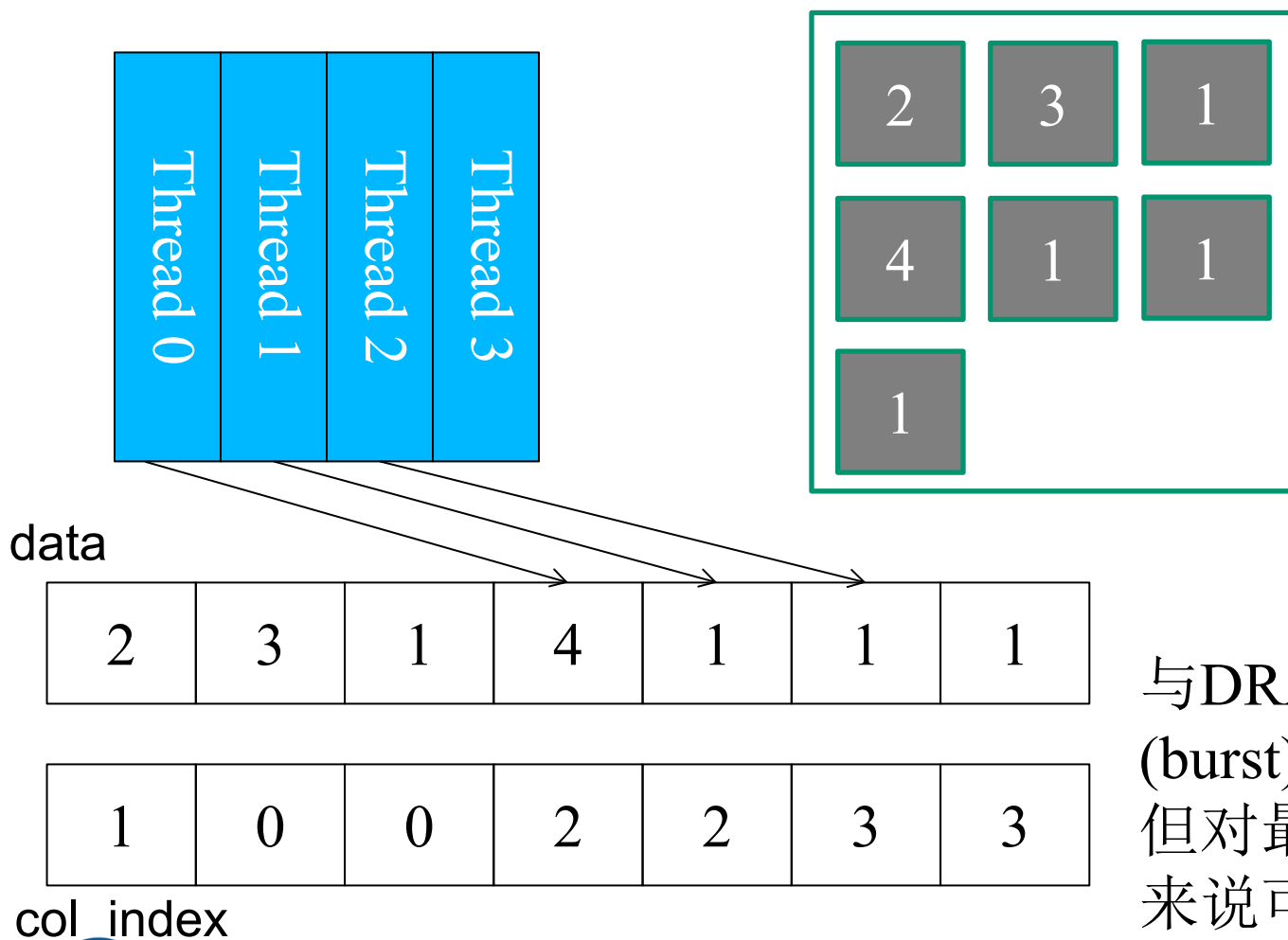
JDS 列指针 jds_t_col_ptr[4] { 0, 3, 6, 7 }



具有转置合并访存的 JDS



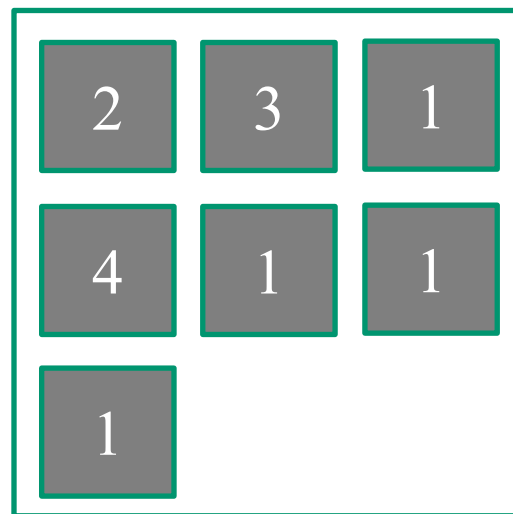
具有转置合并访存的 JDS



与DRAM突发
(burst)不一致，
但对最近的GPU
来说可以接受



具有转置合并访存的 JDS



data

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

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

col_index



电子科技大学
University of Electronic Science and Technology of China

一个SpMV/JDS_T内核

```
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;
5.     unsigned in_sec = 0;
6.     while (jds_t_col_ptr[in_sec+1]-jds_t_col_ptr[in_sec] > row) {
7.       dot += data[jds_t_col_ptr[in_sec]+row] *
8.             x[col_index[jds_t_col_ptr[in_sec]+row]];
9.       in_sec++;
10.    }
11.    y[jds_row_perm[row]] = dot;
12.  }
13. }
```

列号	col_index[7]	{ 1, 0, 3, 2, 2, 3, 3 }
JDS_T 列指针	jds_t_col_ptr[5]	{ 0, 3, 6, 7, 7 }
JDS 行号	jds_row_perm[4]	{ 2, 0, 3, 14 }

MP7 变量名称

JDS_T 列长度 matRows[4] {3, 2, 2, 0 }

	Sec 0	Sec 1	Sec 2
非零值 matData[7]	{ 2, 3, 1,	4, 1, 1	1 }

列号 matCols[7]	{ 1, 0, 3,	2, 2, 3	3 }
---------------	------------	---------	-----

JDS_T 列指针 matColStart[4] {0, 3, 6, 7 }

JDS 行号 matRowPerm[4] {2, 0, 3, 1 }



稀疏矩阵作为高级算法技术的基础

- 图通常表示为稀疏邻接矩阵
 - 广泛用于社交网络分析、自然语言处理等
- 分箱(Binning)技术通常使用稀疏矩阵进行数据压缩
 - 广泛用于光线追踪、基于粒子的流体动力学方法和游戏



ANY QUESTIONS?

