Irregular Applications –Sparse Matrix Vector Multiplication

CS/EE217 GPU Architecture and Parallel Programming

Lecture 20

Sparse Matrix-Vector Multiplication

Many slides from presentation by:
Kai He[1], Sheldon Tan[1],
Esteban Tlelo-Cuautle[2], Hai Wang[3] and He Tang[3]

[1]University of California, Riverside

[2]Institute National Astrophysics, Optical and Electrics, Mexico

[3]UESTC, China

UC RIVERSIDE UNIVERSITY OF CALIFORNIA

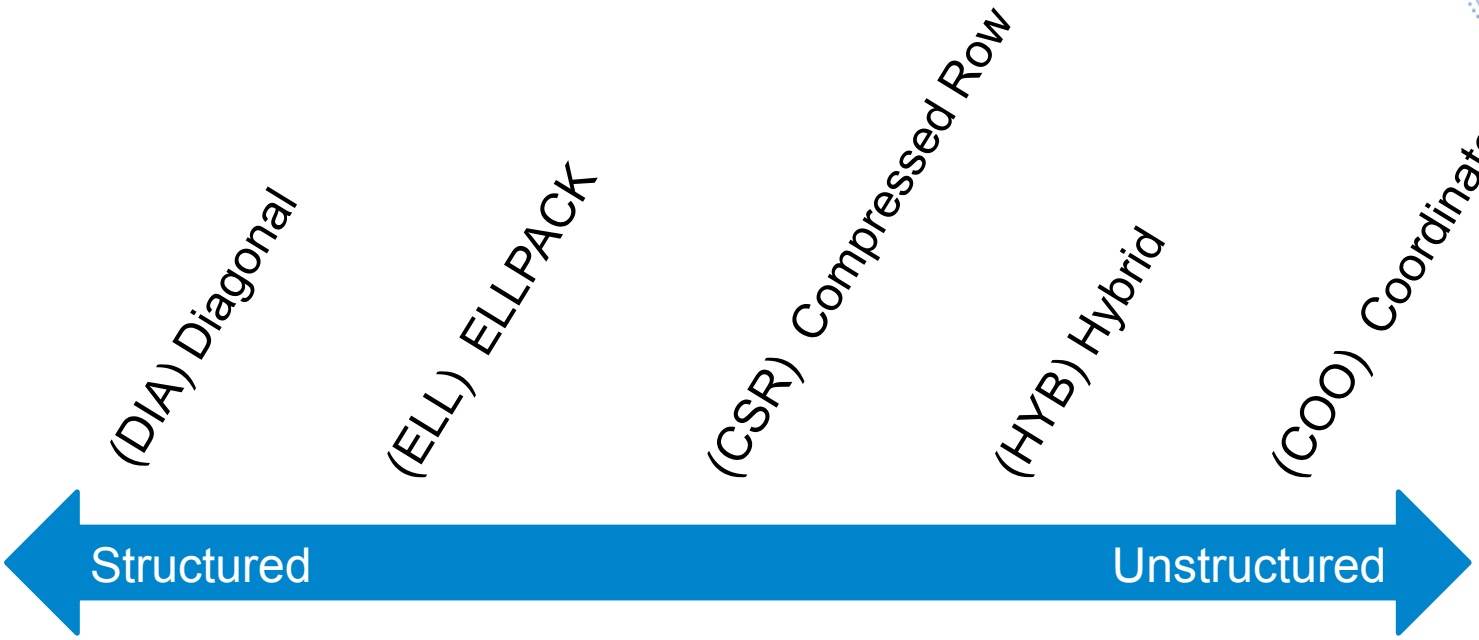
Outline

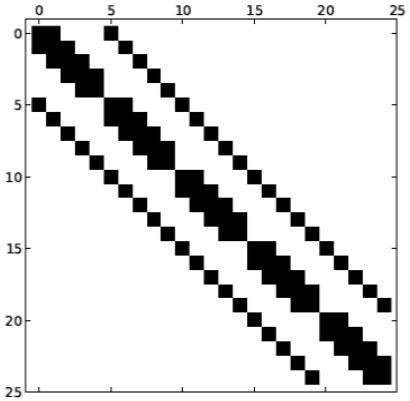
- Background
- Review of existing methods of Sparse Matrix-Vector (SpMV) Multiplication on GPU
- SegSpMV method on GPU
- Experimental results
- Conclusion

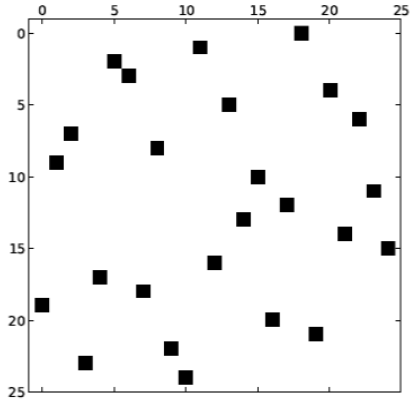
Sparse Matrices

- Matrices where most elements are zeros
- Come up frequently in many science, engineering, modeling, computer science, etc.. Problems
  - Example, solvers for linear systems of equations
  - Example, graphs (e.g., representation of the web connectivity)
- What impact/drawback do they have on computations?

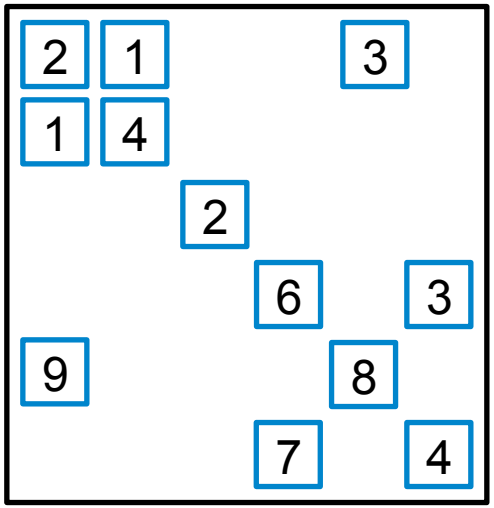
Sparse Matrix Format





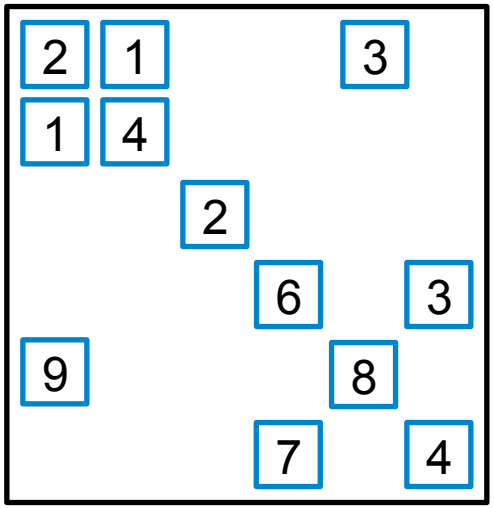


Compressed Sparse Row (CSR)



- Row Ptr 1 4 6 7 9 11 13
- Col Idx 1 2 5 | 1 2 | 3 | 4 6 | 1 5 | 4 6
- Values 2 1 3 1 4 2 6 3 9 8 7 4

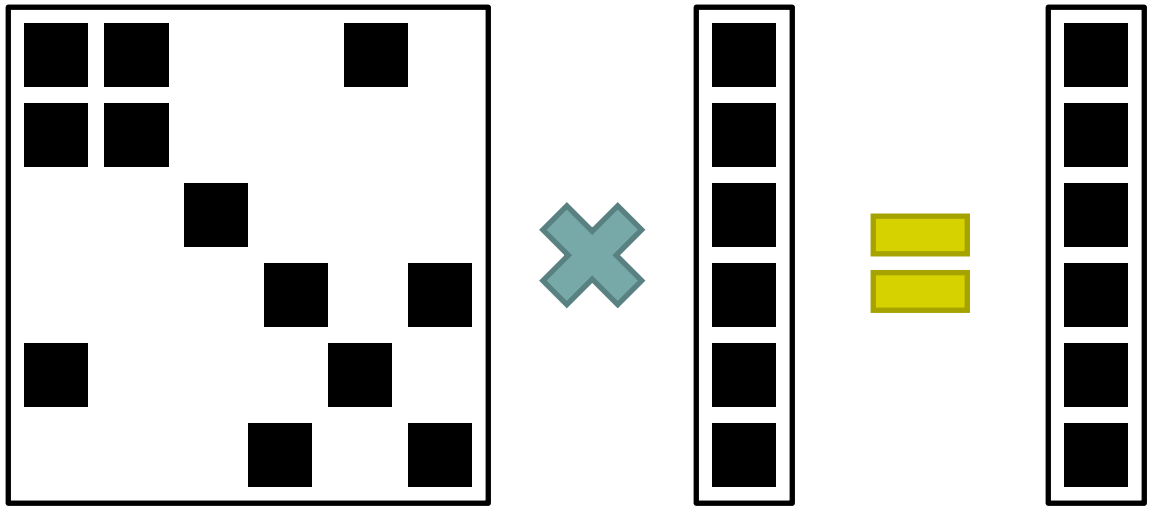
Compressed Sparse Column (CSC)



- Col Ptr 1 4 6 7 9 11 13
- Row Idx 1 2 5 | 1 2 | 3 | 4 6 | 1 5 | 4 6
- Values 2 1 9 1 4 2 6 7 3 8 3 4

Characteristics of SpMV Multiplication

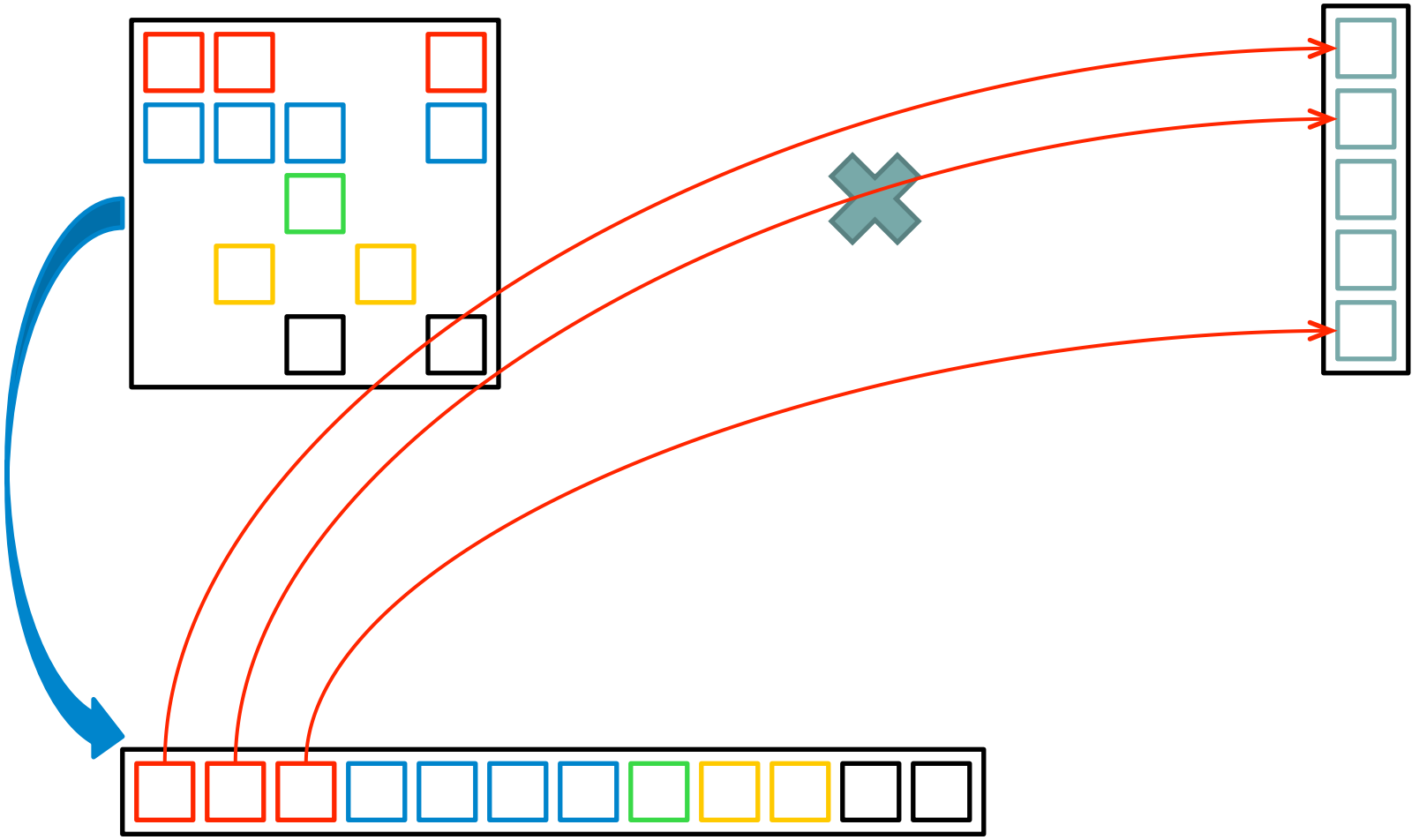
- Memory bound:
  - FLOP : Byte ratio is very low
- Generally irregular & unstructured
  - Unlike dense matrix operation



Objectives of SpMV Implementation on GPU

- Expose sufficient parallelism
  - Develop thousands of independent threads
- Minimize execution path divergence
  - SIMD utilization
- Minimize memory access divergence
  - Memory coalescing

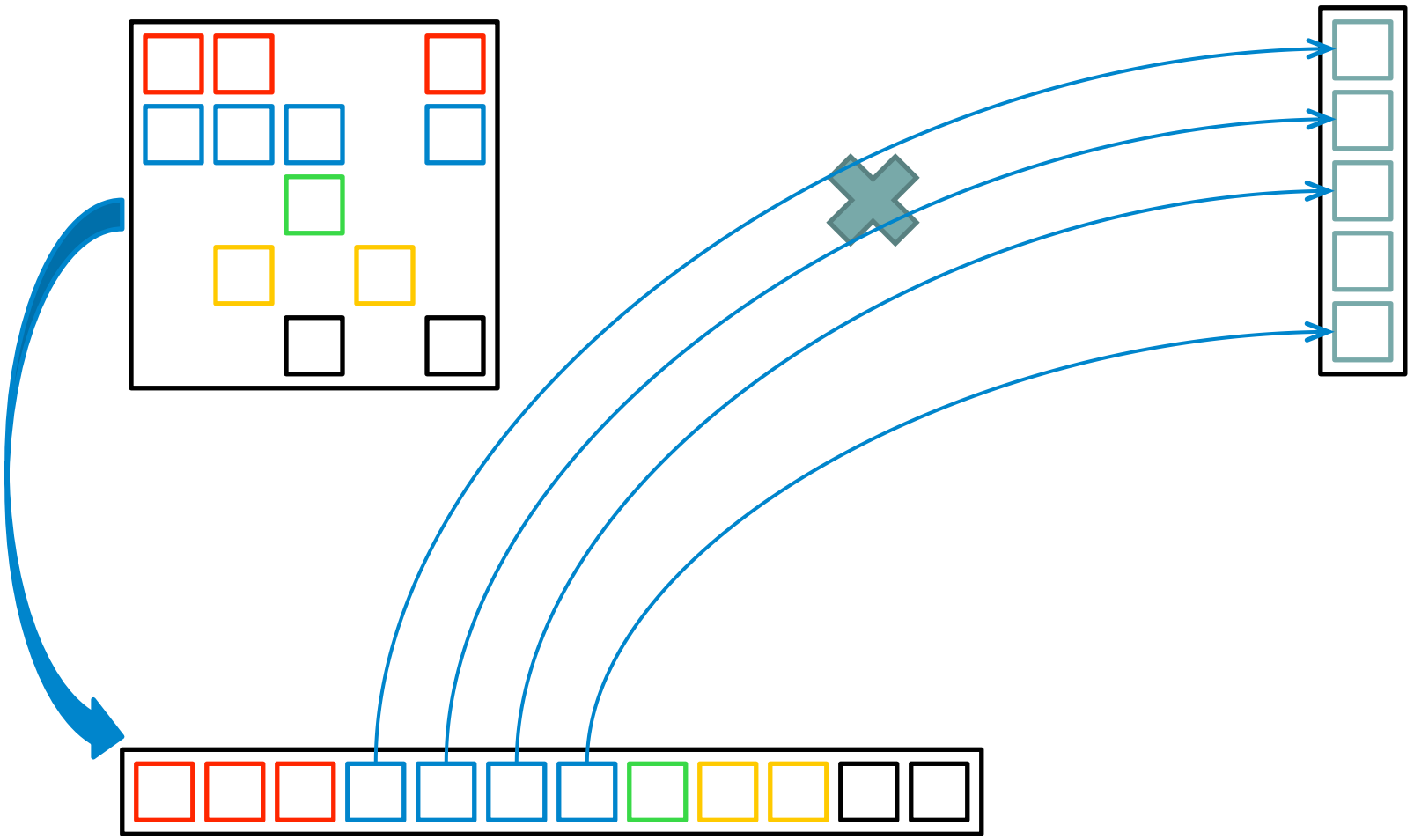
Computing Sparse Matrix-Vector Multiplication



Row laid out in sequence

Middle[elemid] = val[elemid] * vec[col[elemid]]

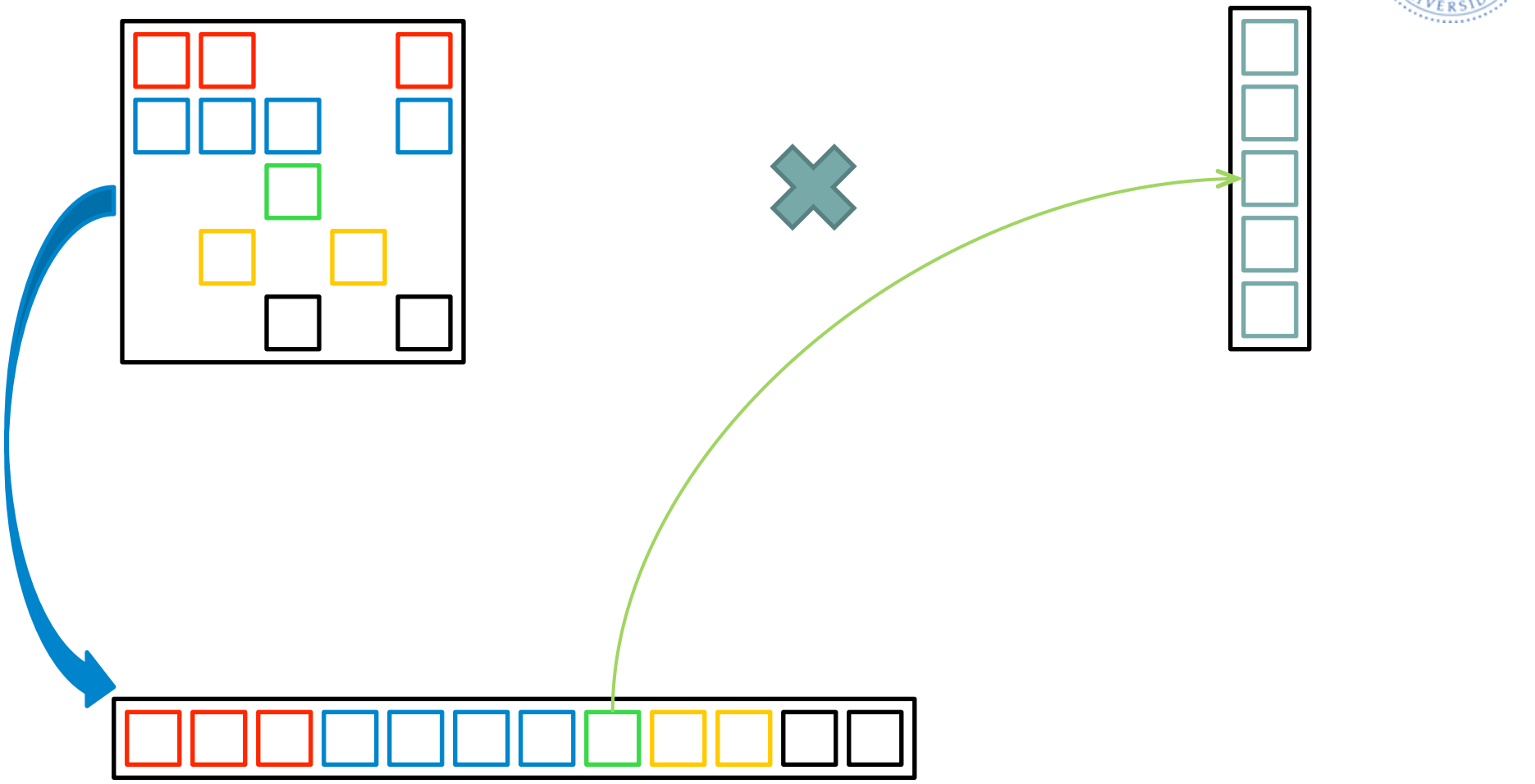
Computing Sparse Matrix-Vector Multiplication



Row laid out in sequence

Middle[elemid] = val[elemid] * vec[col[elemid]]

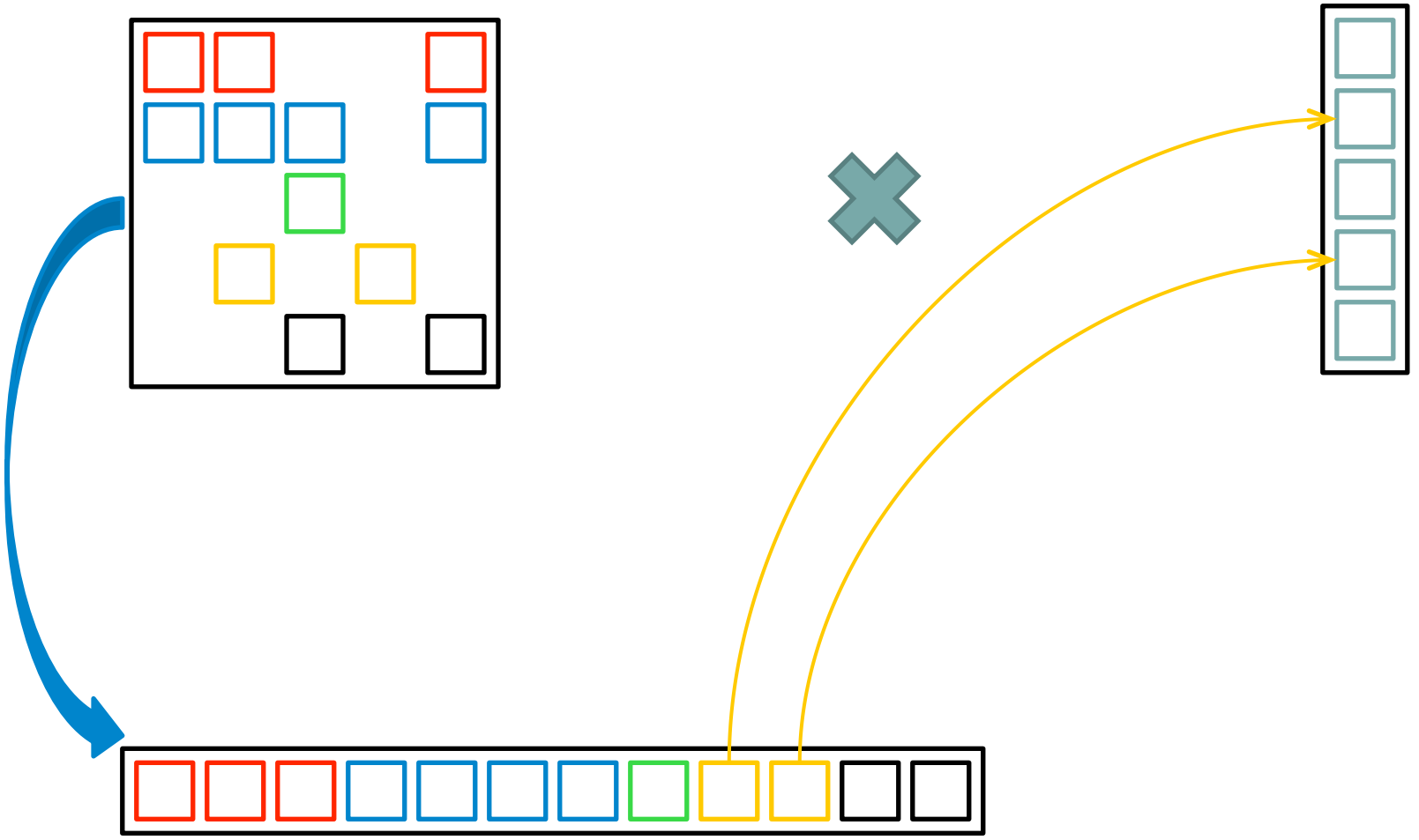
Computing Sparse Matrix-Vector Multiplication



Row laid out in sequence

Middle[elemid] = val[elemid] * vec[col[elemid]]

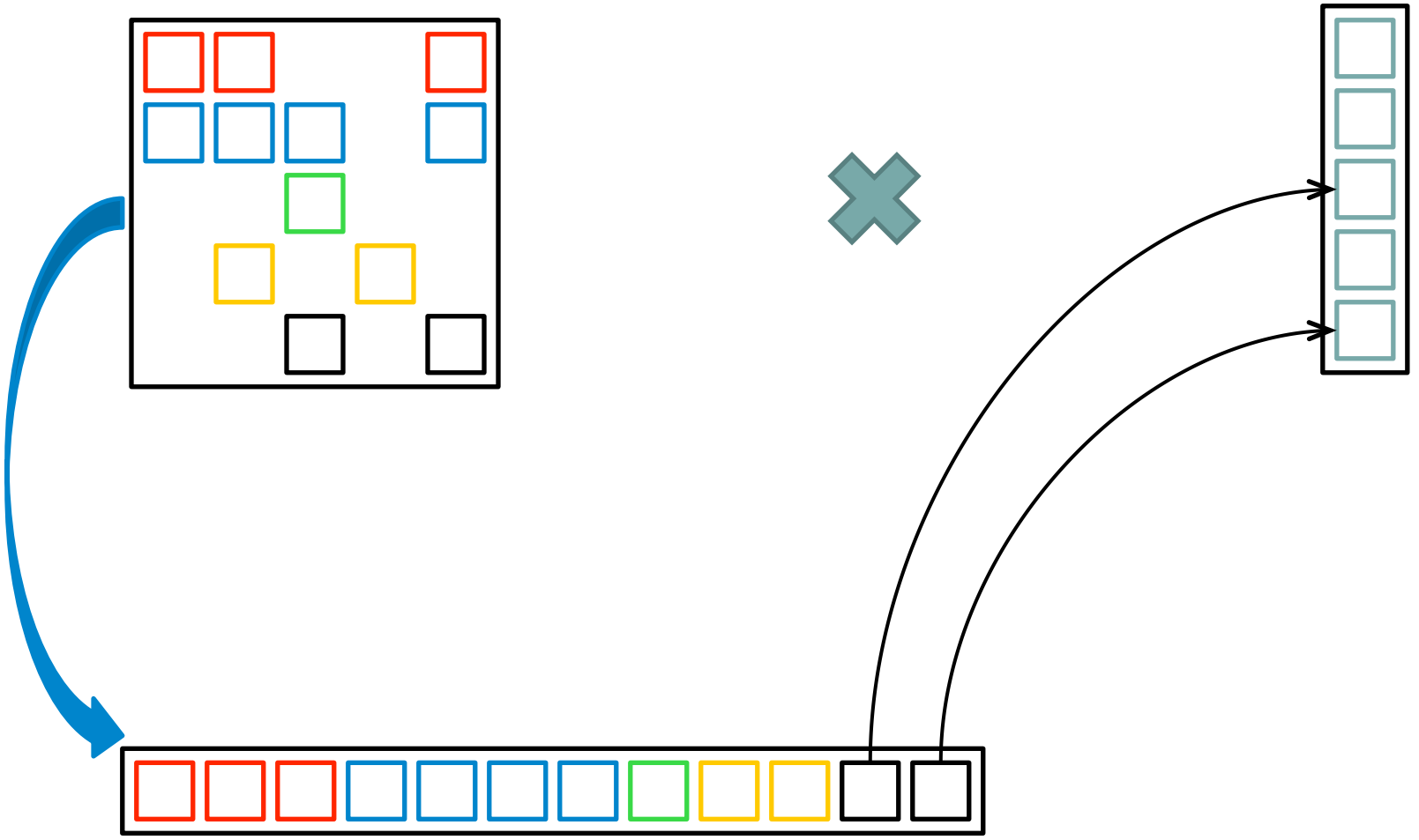
Computing Sparse Matrix-Vector Multiplication



Row laid out in sequence

Middle[elemid] = val[elemid] * vec[col[elemid]]

Computing Sparse Matrix-Vector Multiplication



Row laid out in sequence

Middle[elemid] = val[elemid] * vec[col[elemid]]

Sequential loop to implement SpMV

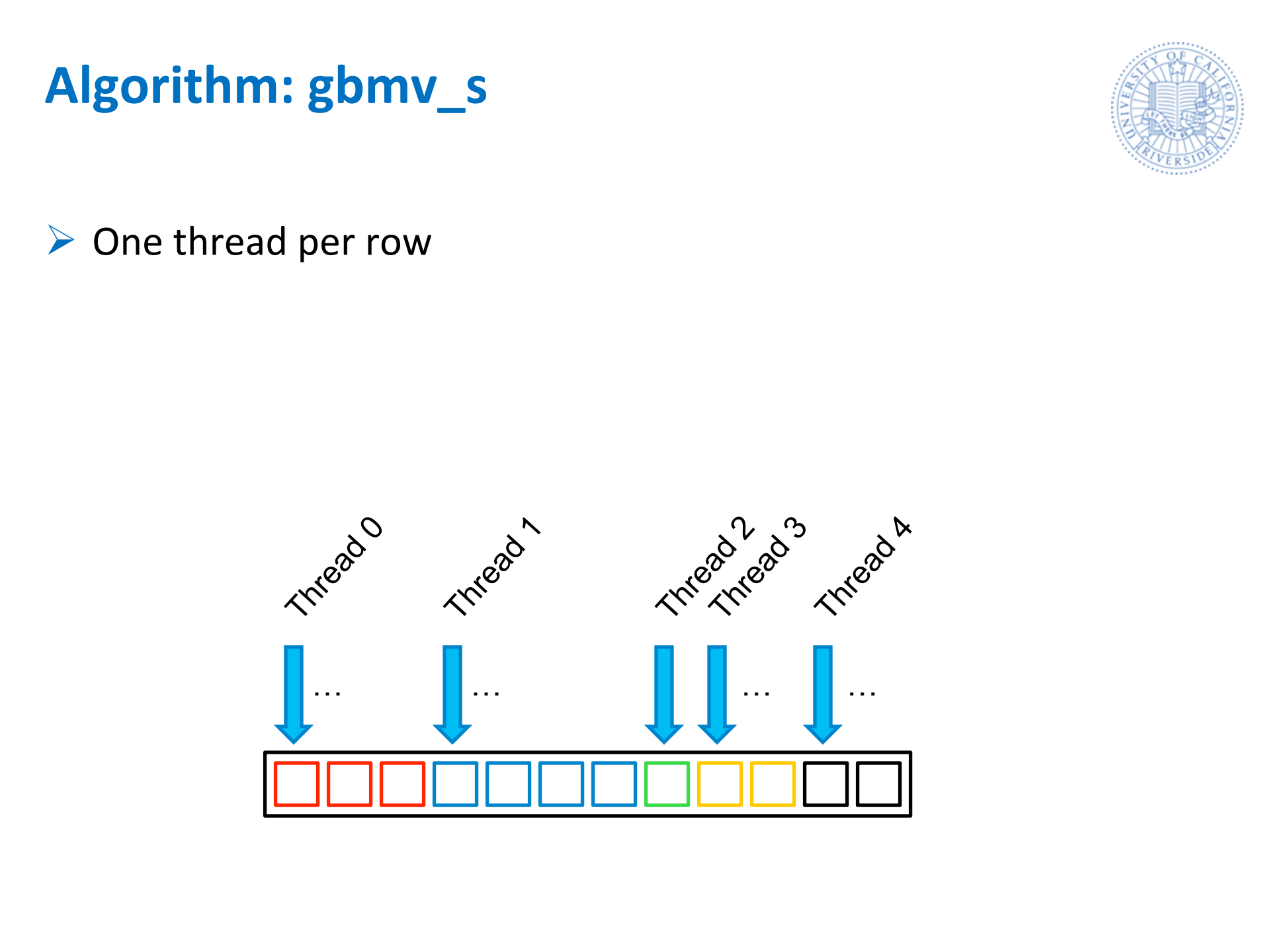
```
for (int row=0; row < num_rows; row++) {
        float dot = 0;
        int row_start = row_ptr[row];
        int row_end = row_ptr[row+1];

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

Discussion

- How do we assign threads to work?

- How will this parallelize?
  - Think of memory access patterns
  - Think of control divergence

Algorithm: gbmv_s



- One thread per row

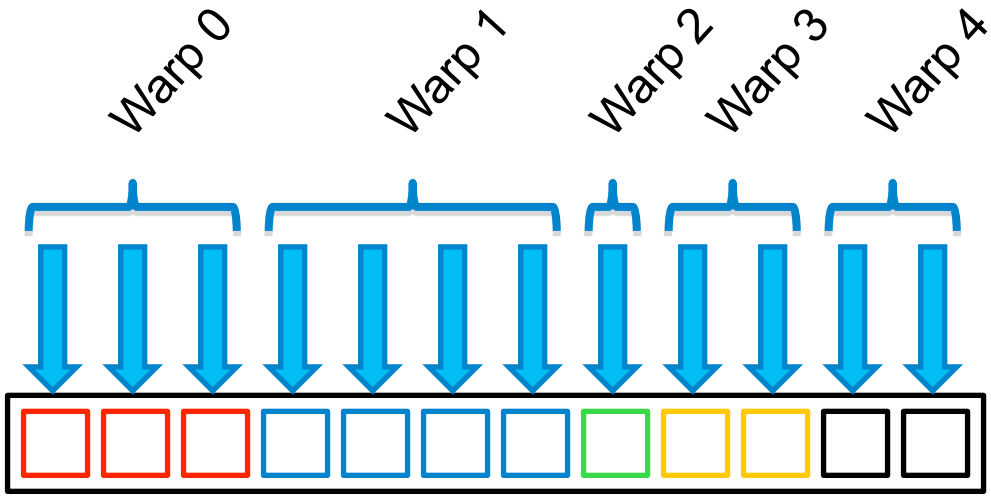
gbmv_s

```
__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;
        }
}
```

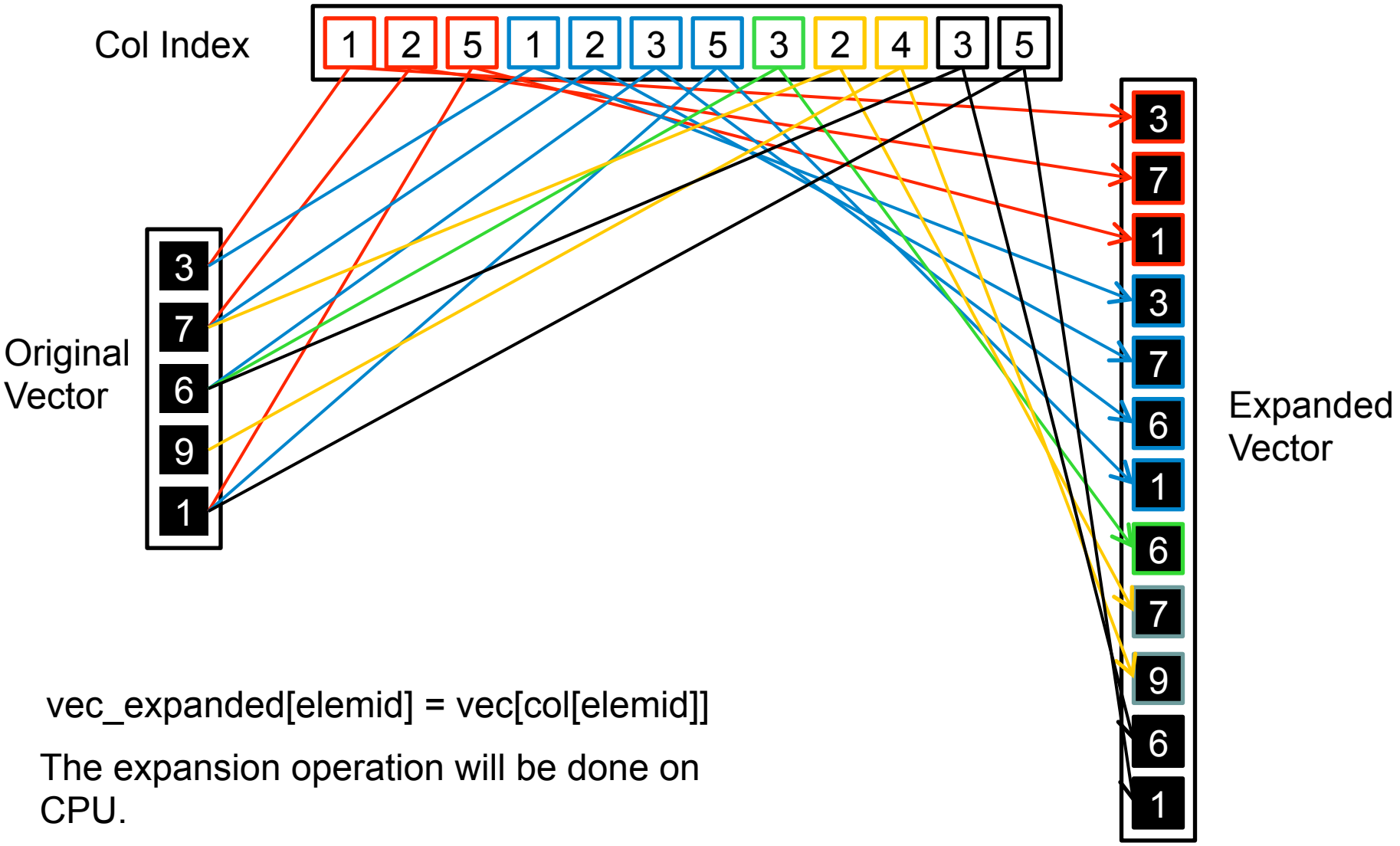
- Issues?
- Poor coalescing

Algorithm: gbmv_w

- One warp per row
  - Partial memory coalescing
  - vec[col[elemid]] cannot be coalesced because the col[elemid] can be arbitrary
  - How do you size the warps?



Vector Expansion

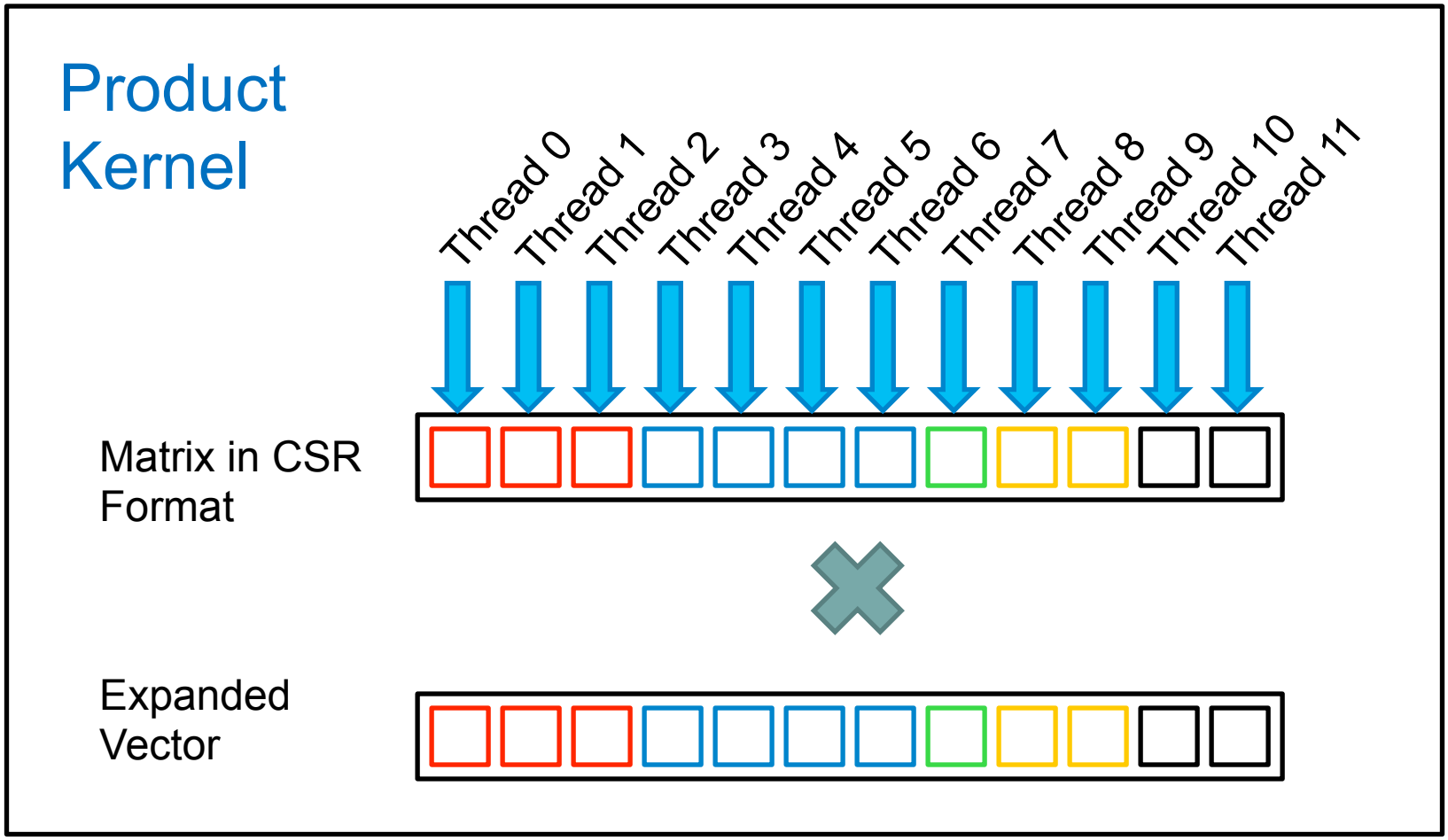


vec_expanded[elemid] = vec[col[elemid]]

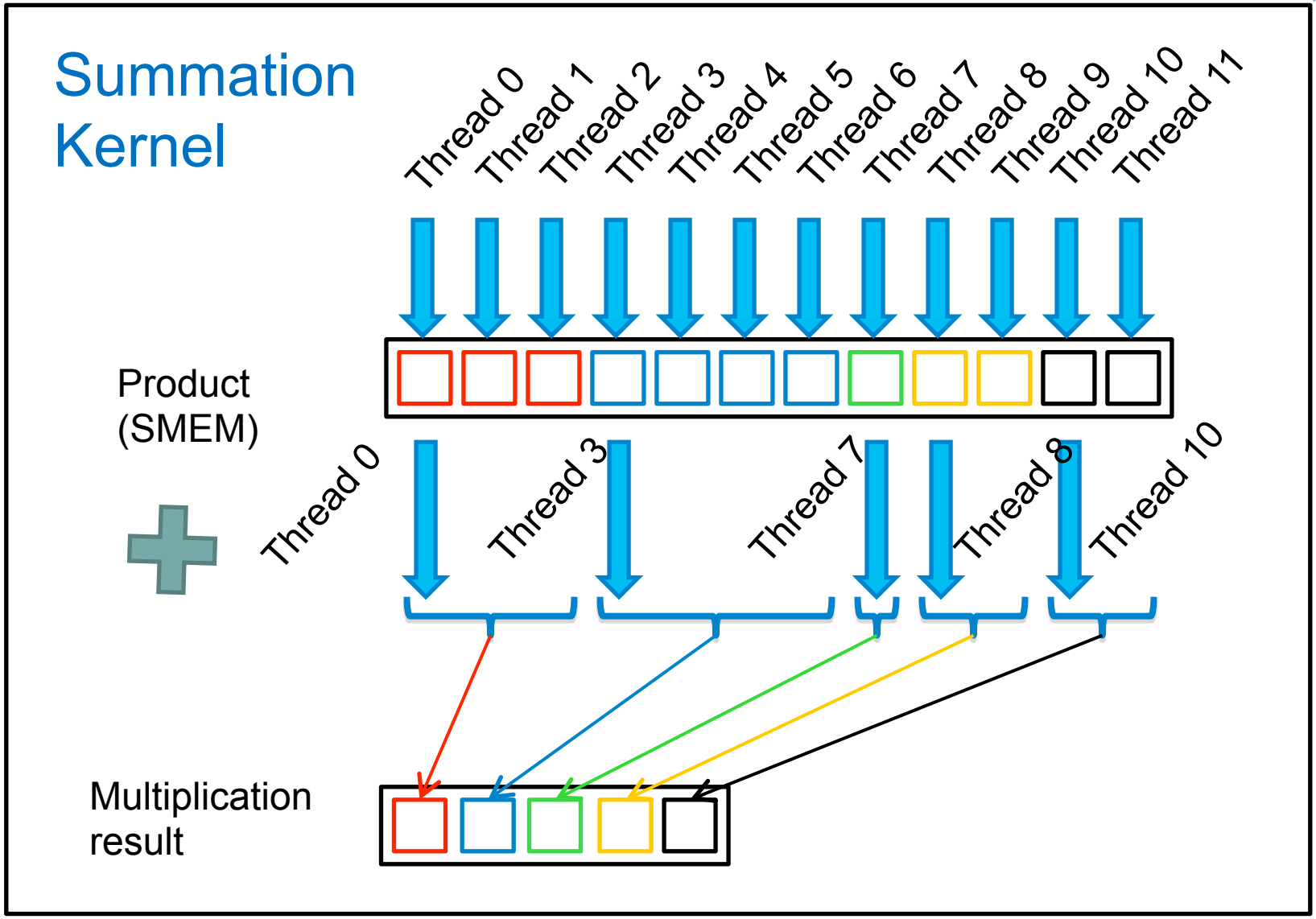
The expansion operation will be done on CPU.

Algorithm: PS (Product & Summation)

- One thread per non-zero element
  - Memory coalescing

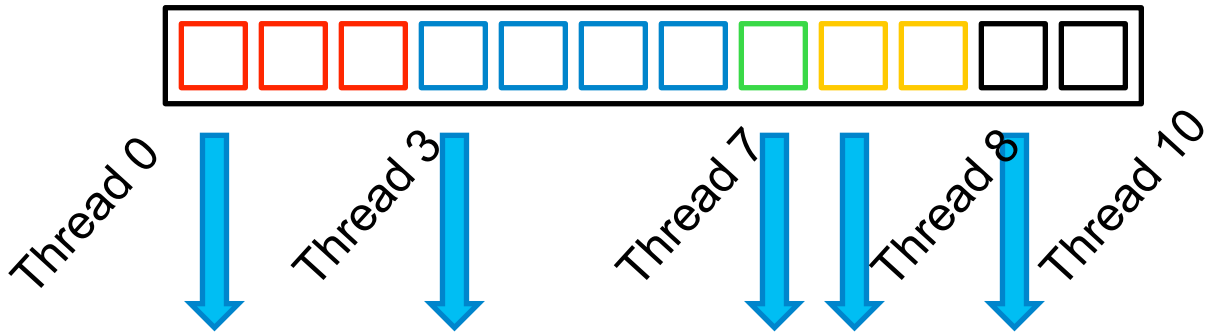


Algorithm: PS (Product & Summation)



Algorithm: PS (Product & Summation)

- We need to read row pointer array to get row_begin and row_end for summation
- The number of addition on each threads are different
- May not have enough shared memory when there are too many nonzeros in a row

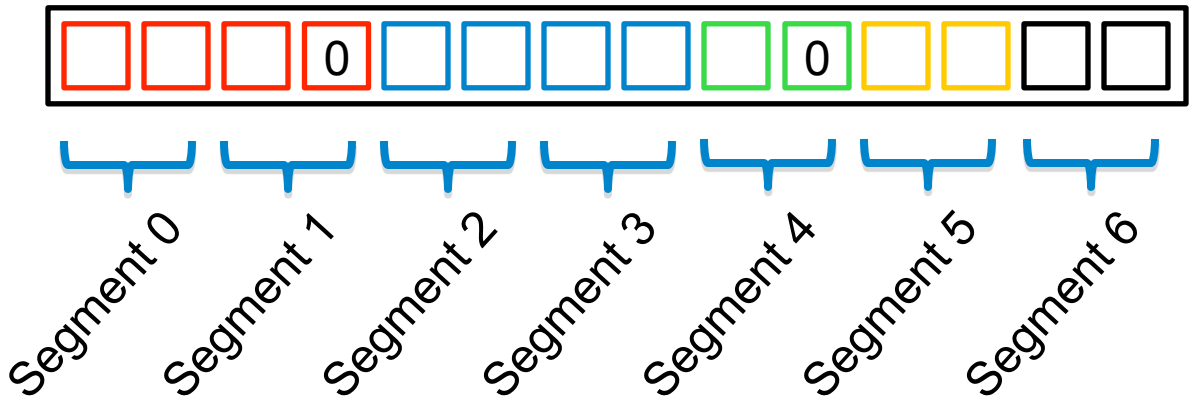


Algorithm: segSpMV

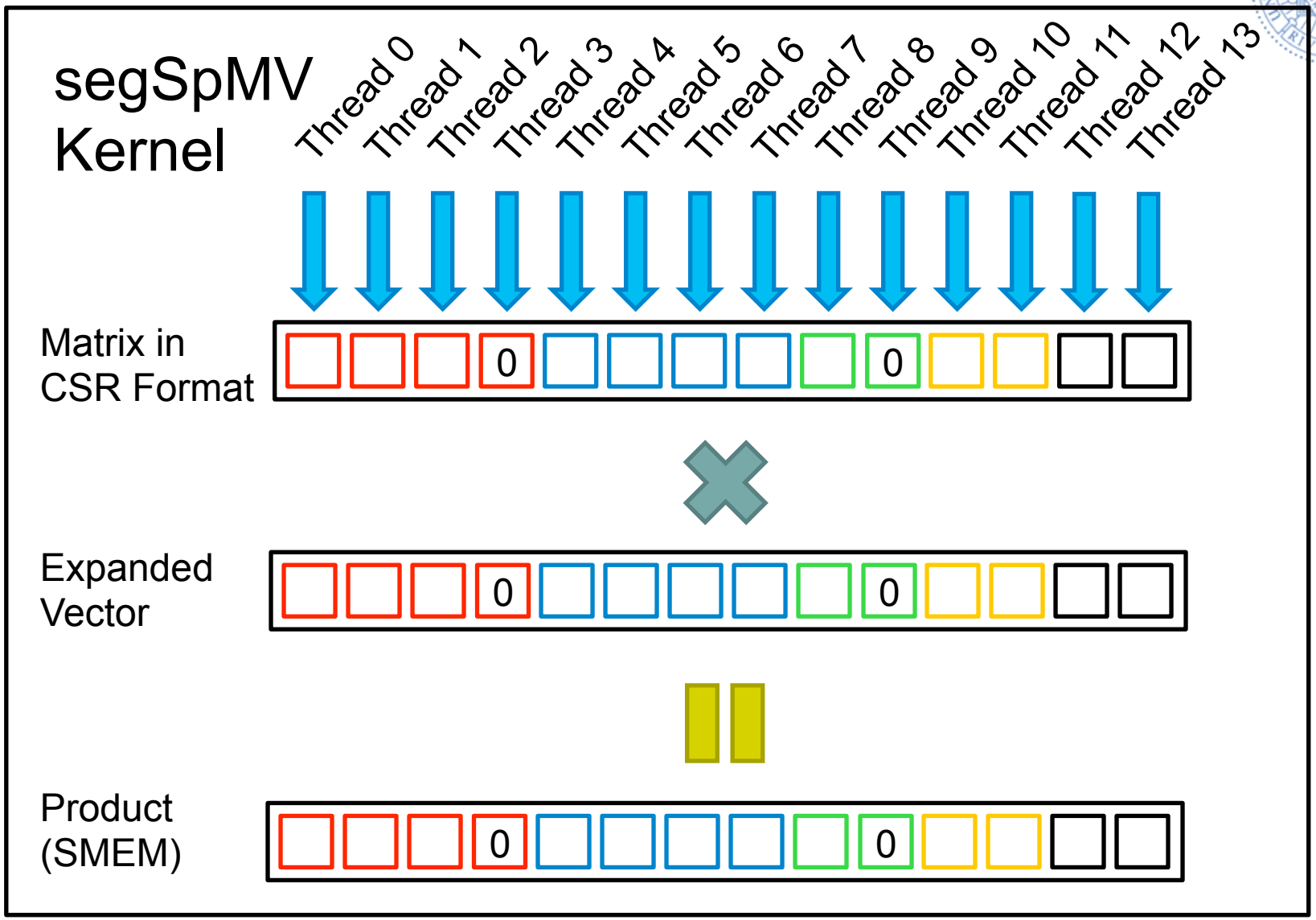
- Divide the matrix in CSR format and expanded vector into small segments with constant length

- Segments without enough elements will be appended zeros.

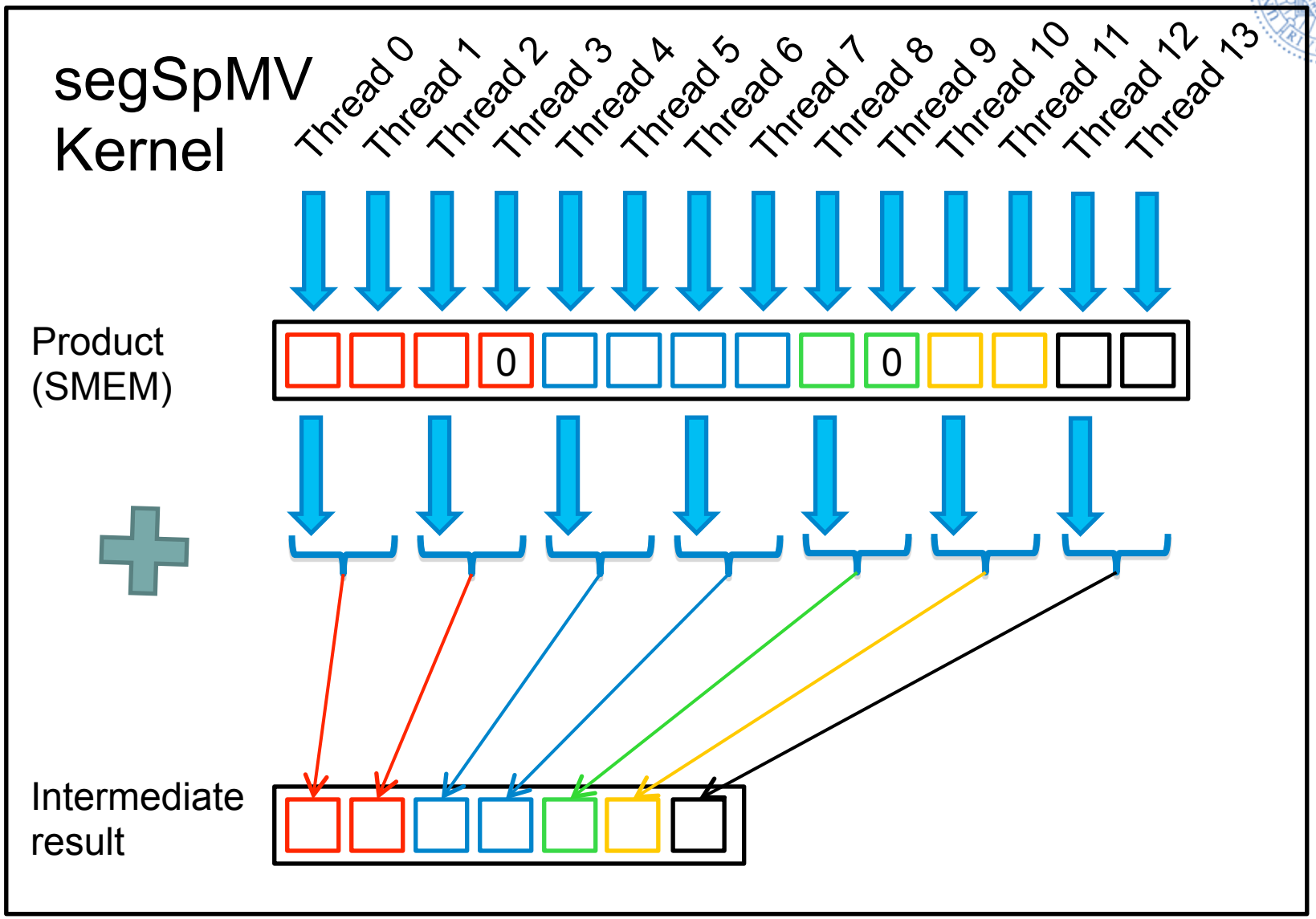
Segment length = 2



segSpMV Kernel



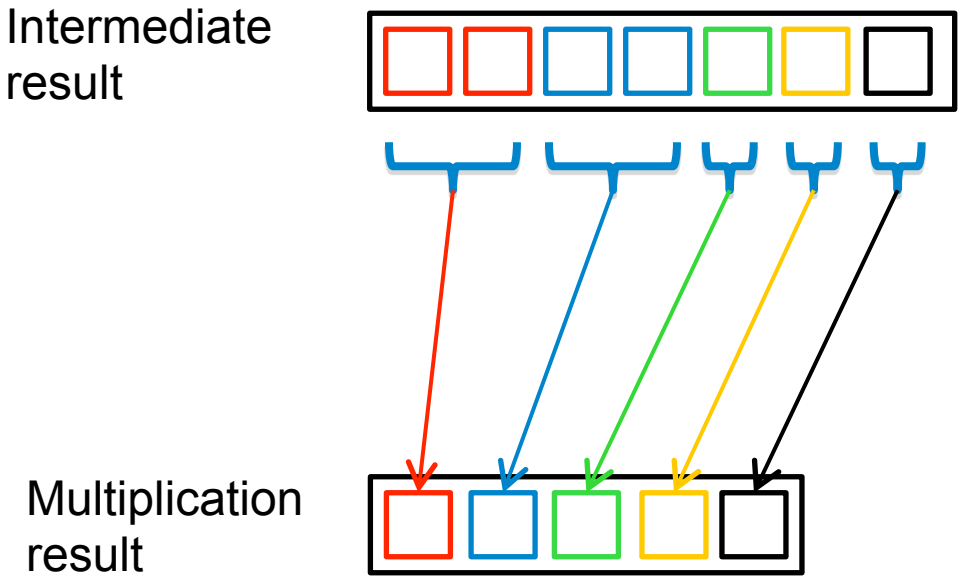
segSpMV Kernel (Cont.)



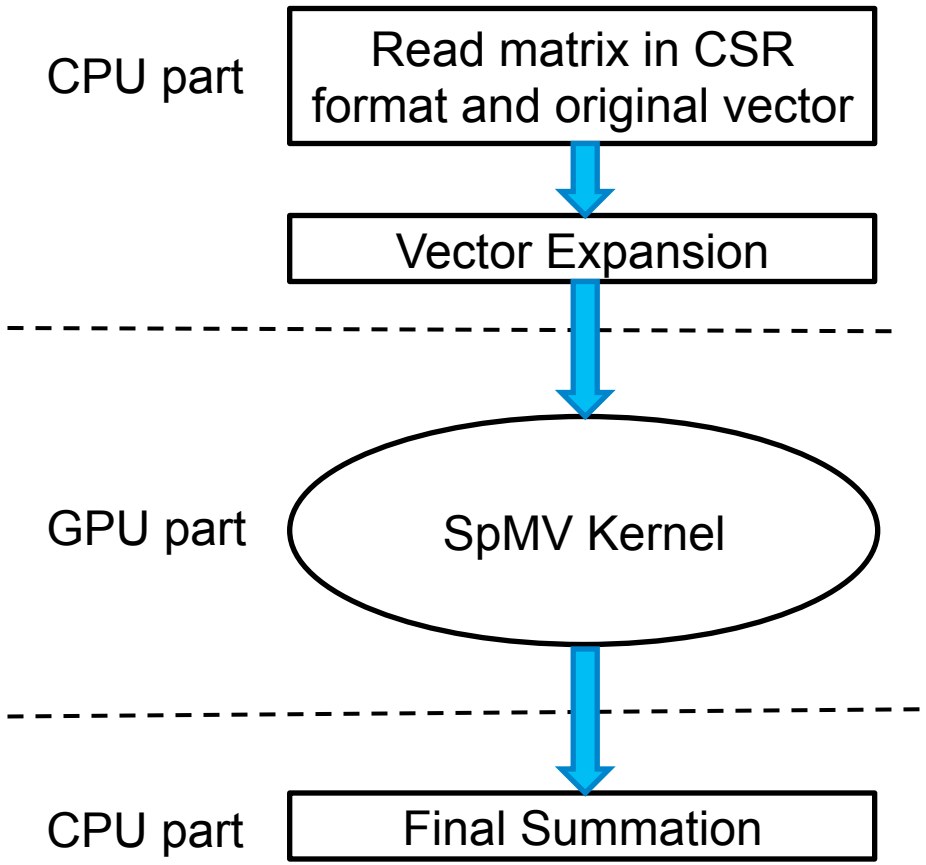
Final Summation

- Final summation is done on CPU

- There will be very few additions in final summation if we choose segment length carefully



Algorithm: segSpMV

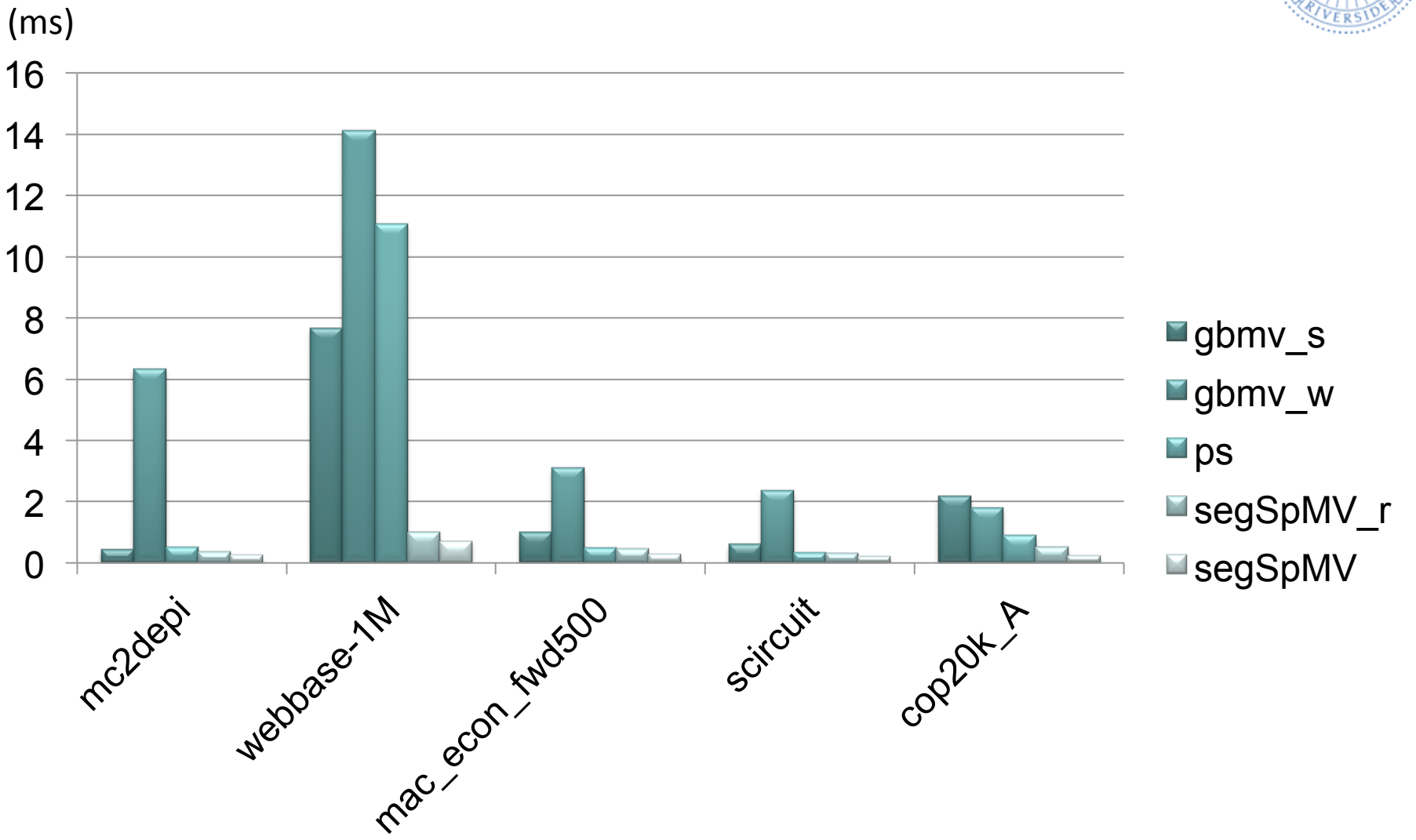


- Advantages:
  - Can enjoy full coalesced memory access
  - Less synchronization overhead because all segments have same length
  - Can store all intermediate results in shared memory

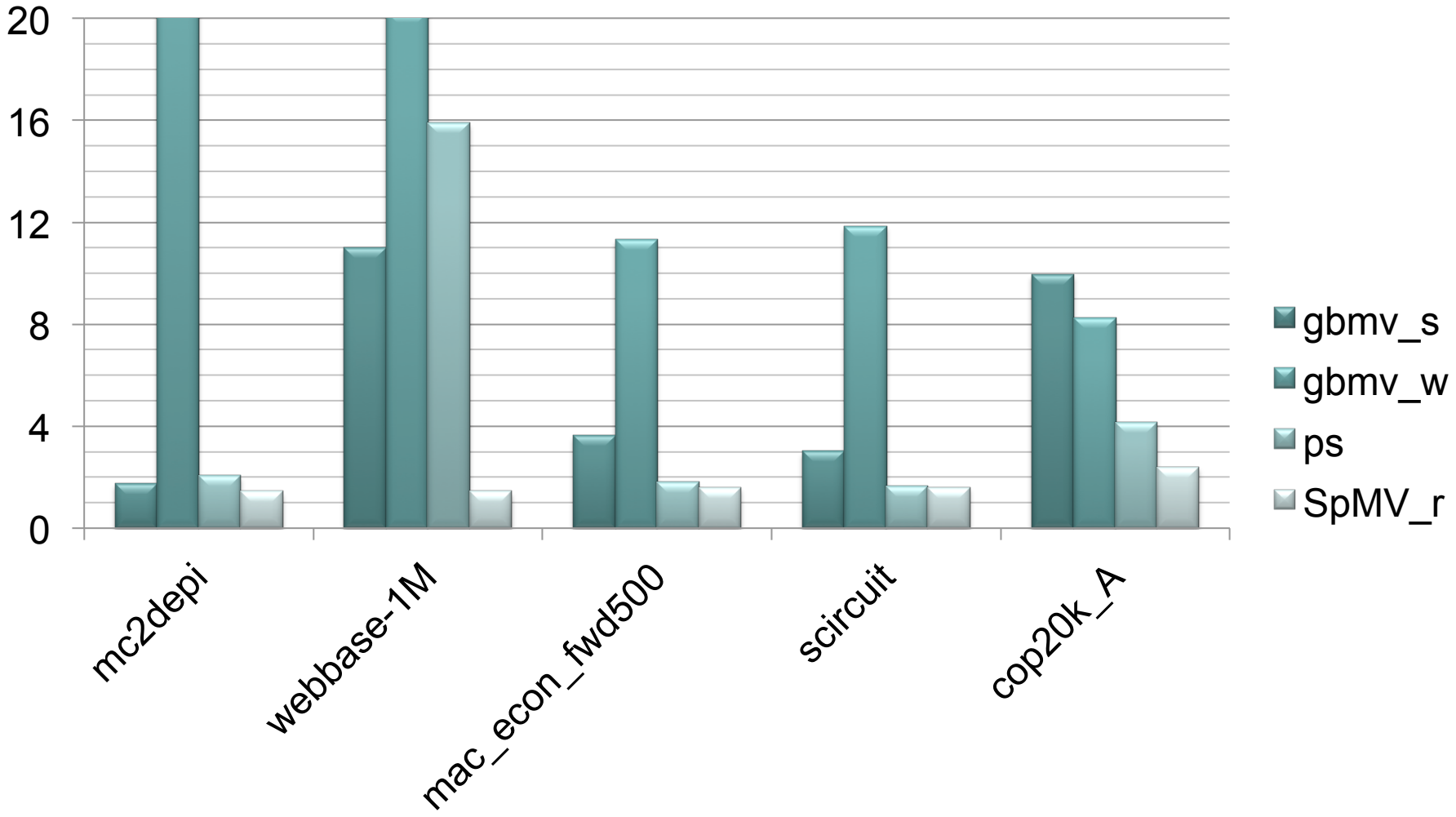
Experiment Results

Matrices	size	nnz	nzperrow	Seg_length
mc2depi	525825	2100225	3.9942	4
mac_econ_fwd500	206500	1273389	6.1665	8
scircuit	170998	958936	5.6079	8
webbase-1M	1000005	6105536	6.1055	8
cop20k_A	121192	1362087	11.2391	16
cant	62451	2034917	32.5842	32
consph	83334	3046907	36.5626	32
pwtk	217918	5926171	27.1945	32
qcd5_4	49152	1916928	39	32
shipsec1	140874	3977139	28.2319	32
pdb1HYS	36417	2190591	60.153	64
rma10	46835	2374001	50.6886	64

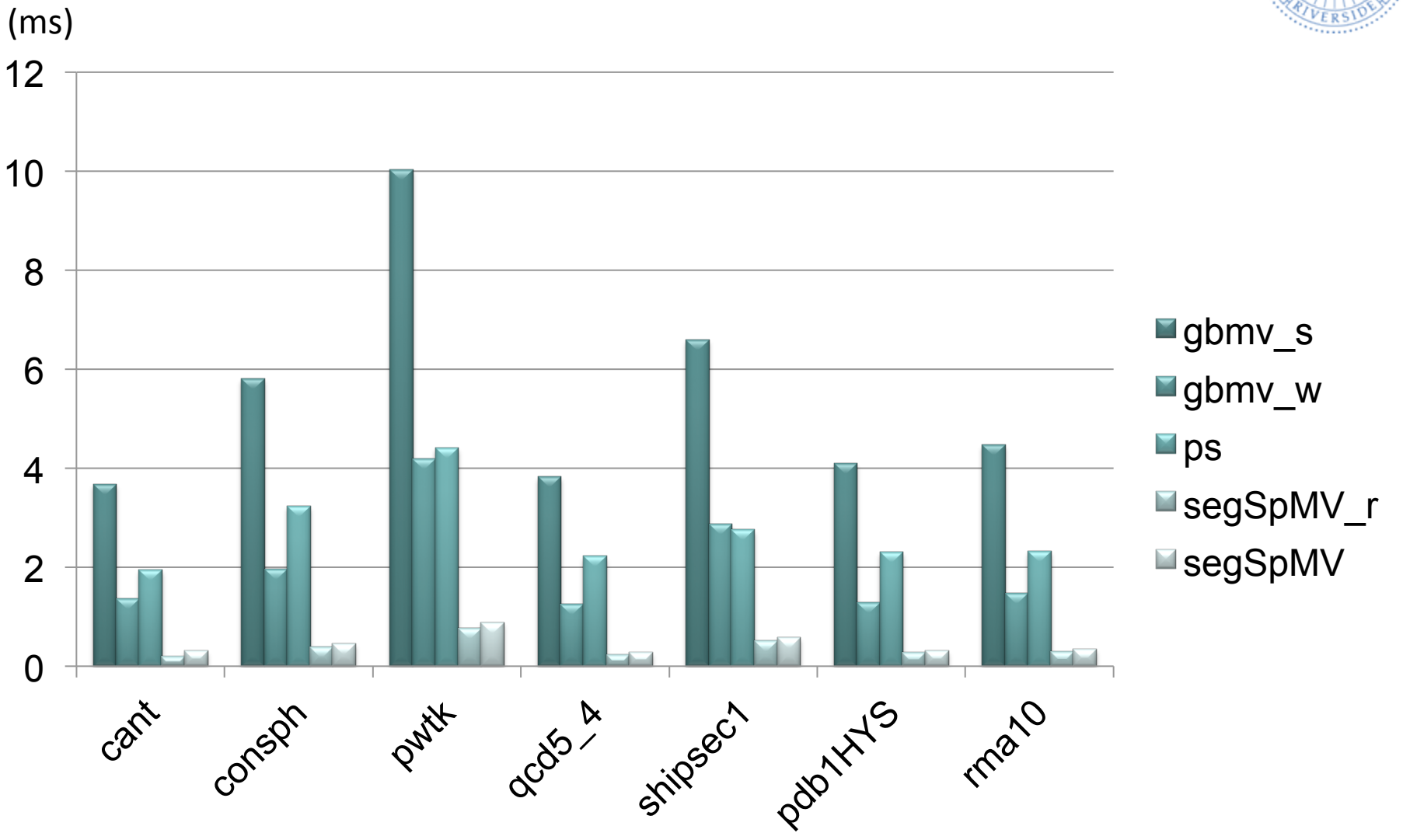
Results of Short Segment



Speedup of SpMV



Results of Long Segment



Reference

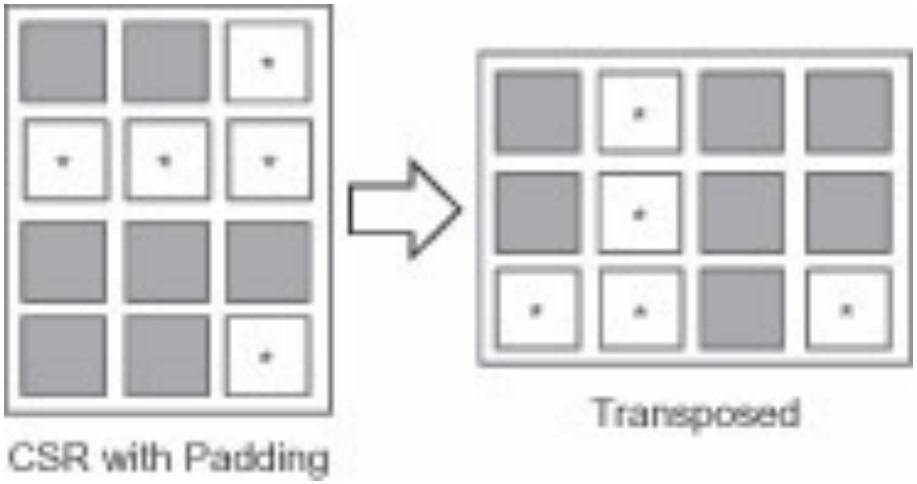
- [0] This paper: Kai He, Sheldon Tan, Esteban Tlelo-Cuautle, Hai Wang an He Tang, “A New Segmentation-Based GPU Accelerated Sparse Matrix Vector Multiplication”, IEEE MWCAS 2014

- [1] Wang BoXiong David and Mu Shuai, “Taming Irregular EDA Applications on GPUs”, ICCAD 2009.

- [2] Nathan Bell and Michael Garland, “Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors”, NVIDIA Research.

- [3] M. Naumov, “Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU”, NVIDIA Technical Report, NVR-2011-001, 2011.

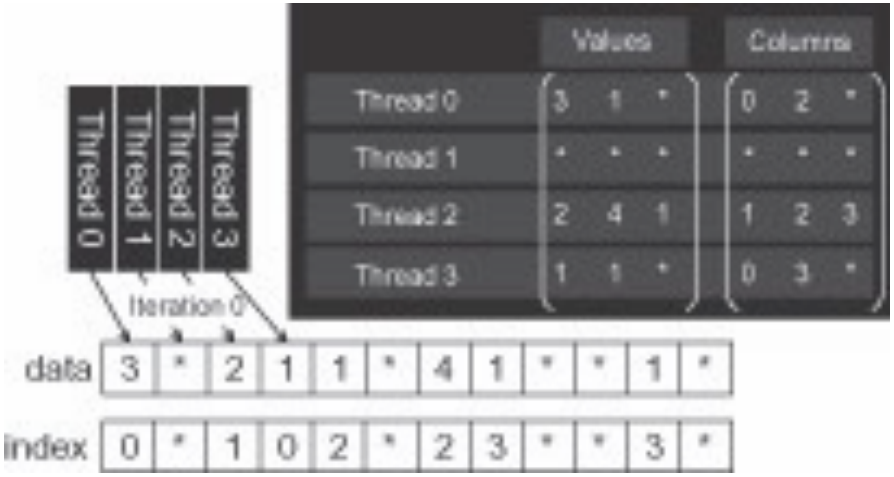
Another Alternative; ELL



- ELL representation; alternative to CSR
  - Start with CSR
  - PAD each row to the size of the longest row
  - Transpose it

```
__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;
        }
}
```

Example/Intuition



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

- Lets look at Thread 0
  - First iteration: 3 * col[0]
  - Second iteration 1 * col [2]
  - Third iteration?
  - Is this the correct result?
  - What do you think of this implementation?

Problems?

- What if one (or a few) rows are dense?
  - A lot of padding
  - Big storage/memory overhead
  - Shared memory overhead if you try to use it
  - Can actually become slower than CSR

- What can we do?
  - In comes COO (Coordinate Format)

Main idea (see 10.4 in the book)

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

- Start with ELL
- Store not only column index, but also row index
  - This now allow us to reorder elements
    - See code above
  - What can we gain from doing that?

Main idea (cont'd)

- Convert CSR to ELL
- As you convert, if there is a long row, you remove some elements from it and keep them in COO format
- Send ELL (missing the COO elements) to GPU for SpMV_ELL kernel to chew on
- Once the results are back, the CPU adds in the contributions from the missing COO elements
  - Which should be very few

Other optimizations?

- Sorting rows by length (see 10.5) in the book
  - Keep track of original row order
  - Create different kernels/grids for rows of similar size
  - Reshuffle y at the end to its original position

Questions?
Please check chapter 10
In the book