

# Overcoming Load Imbalance for Irregular Sparse Matrices

---

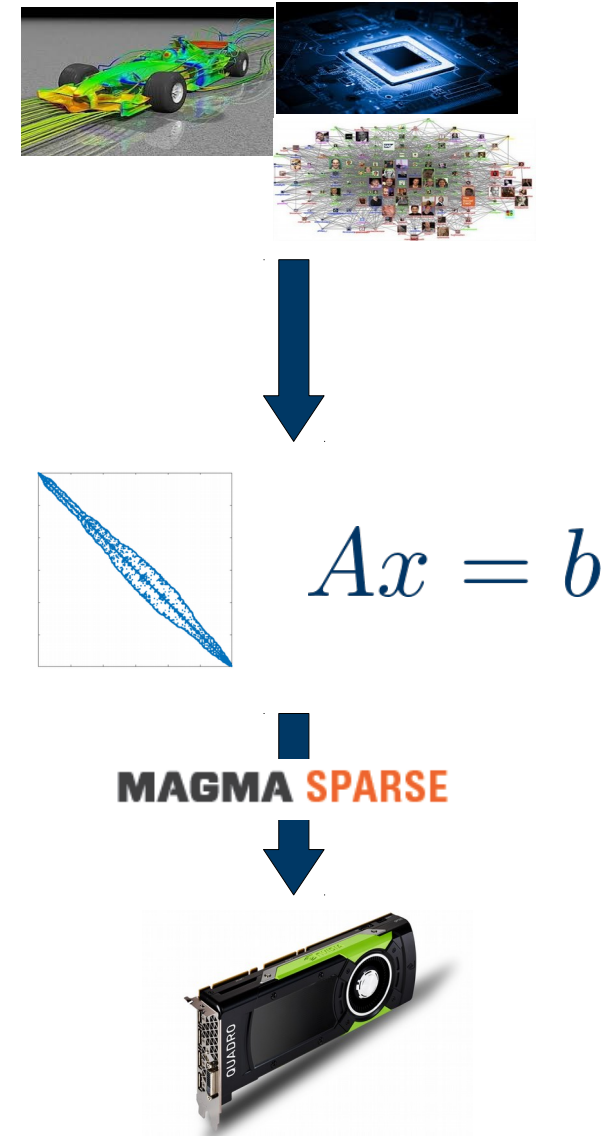
Goran Flegar, Hartwig Anzt



Scan me  
for slides!

# MAGMA-sparse software library

- GPU-accelerated sparse linear algebra library
  - Focus: linear systems

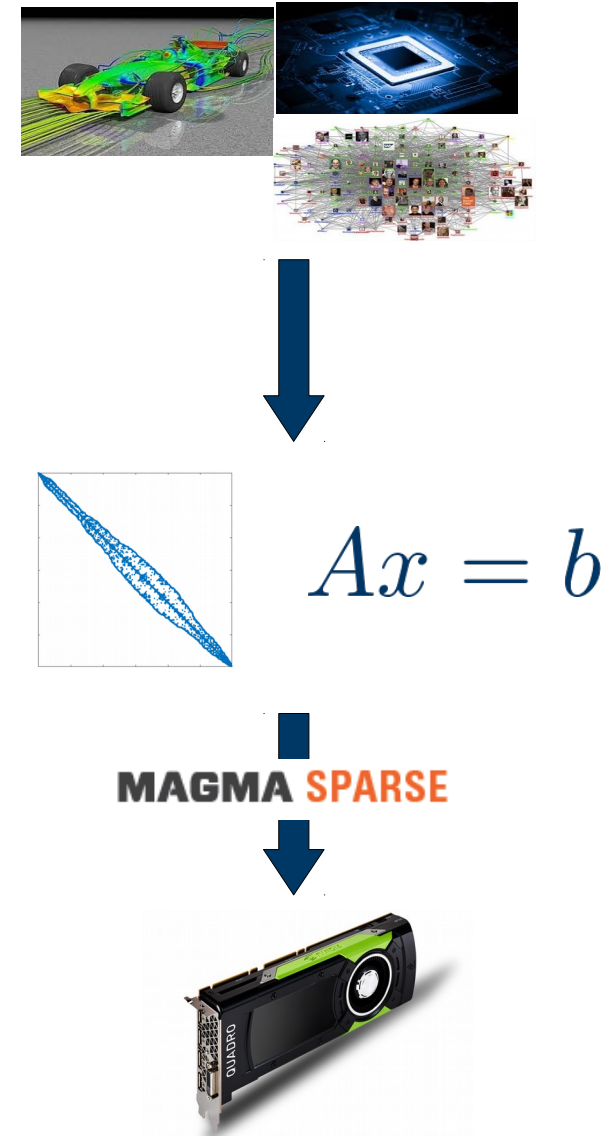


Joint effort: Innovative Computing Lab at University of Tennessee, Knoxville; Karlsruhe Institute of Technology; Universidad Jaume I



# MAGMA-sparse software library

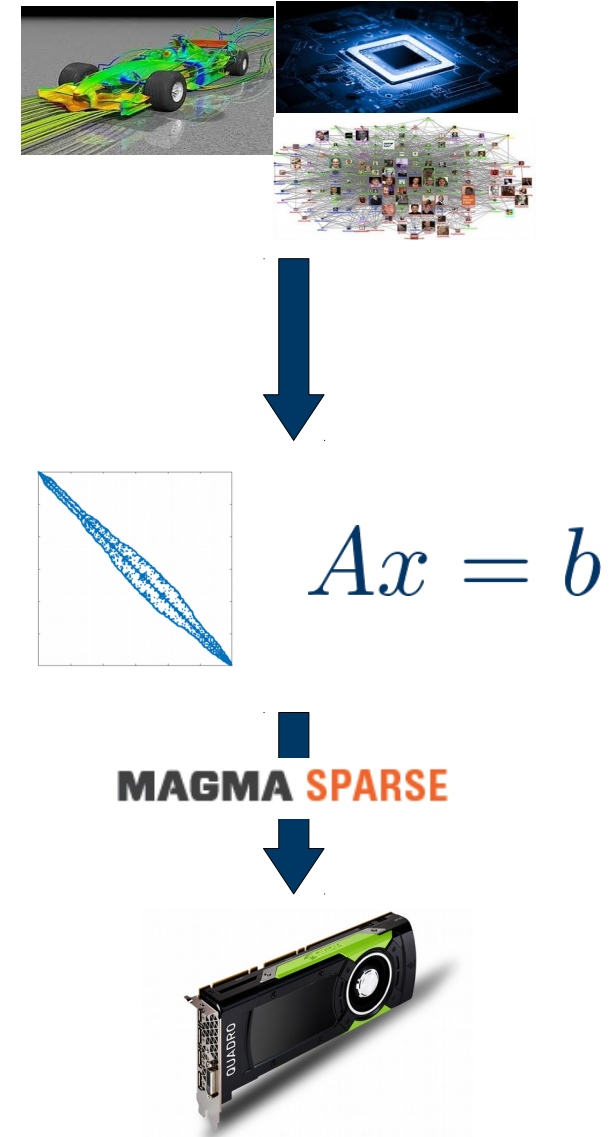
- GPU-accelerated sparse linear algebra library
  - Focus: linear systems
  - Iterative, Krylov-subspace based linear solvers
    - SpMV
    - BLAS-1 operations



Joint effort: Innovative Computing Lab at University of Tennessee, Knoxville; Karlsruhe Institute of Technology; Universidad Jaume I

# MAGMA-sparse software library

- GPU-accelerated sparse linear algebra library
  - Focus: linear systems
  - Iterative, Krylov-subspace based linear solvers
    - SpMV
    - BLAS-1 operations
  - Sparse matrix formats & SpMV
    - accelerate each iteration of the solver

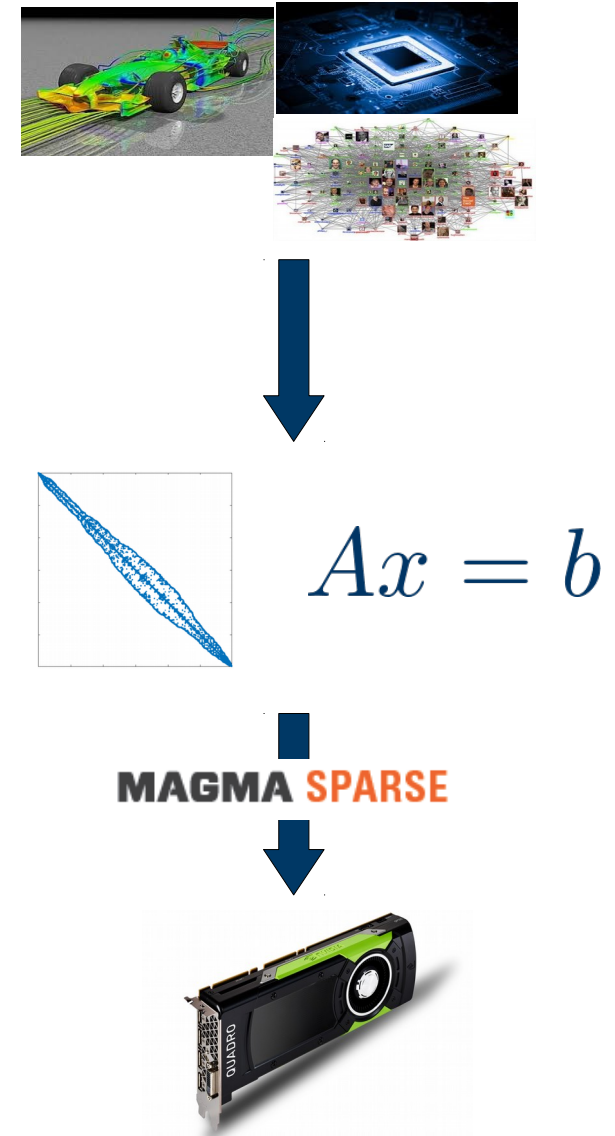


Joint effort: Innovative Computing Lab at University of Tennessee, Knoxville; Karlsruhe Institute of Technology; Universidad Jaume I



# MAGMA-sparse software library

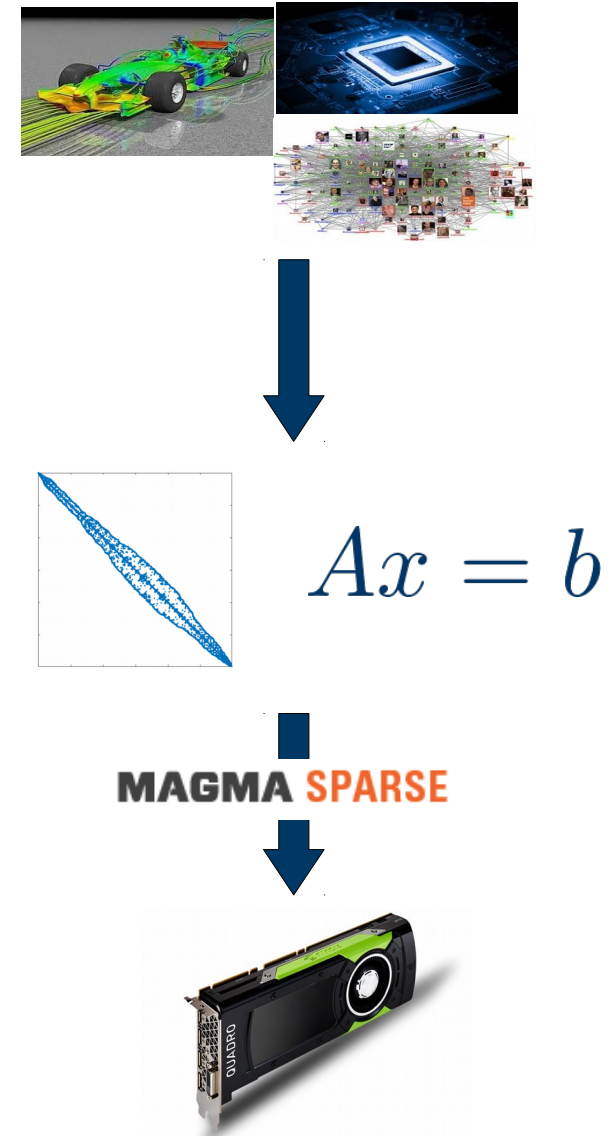
- GPU-accelerated sparse linear algebra library
  - Focus: linear systems
  - Iterative, Krylov-subspace based linear solvers
    - SpMV
    - BLAS-1 operations
  - Sparse matrix formats & SpMV
    - accelerate each iteration of the solver
  - Preconditioners
    - reduce the number of iterations



Joint effort: Innovative Computing Lab at University of Tennessee, Knoxville; Karlsruhe Institute of Technology; Universidad Jaume I

# MAGMA-sparse software library

- GPU-accelerated sparse linear algebra library
  - Focus: linear systems
  - Iterative, Krylov-subspace based linear solvers
    - SpMV
    - BLAS-1 operations
  - Sparse matrix formats & SpMV
    - accelerate each iteration of the solver
  - Preconditioners
    - reduce the number of iterations



Joint effort: Innovative Computing Lab at University of Tennessee, Knoxville; Karlsruhe Institute of Technology; Universidad Jaume I

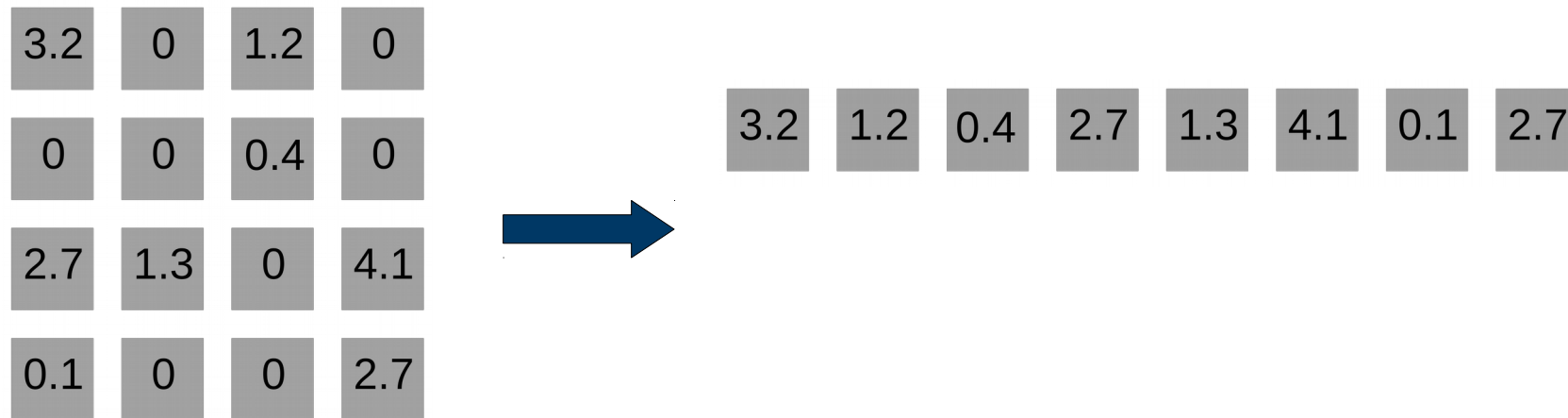


# Sparse matrix formats

---

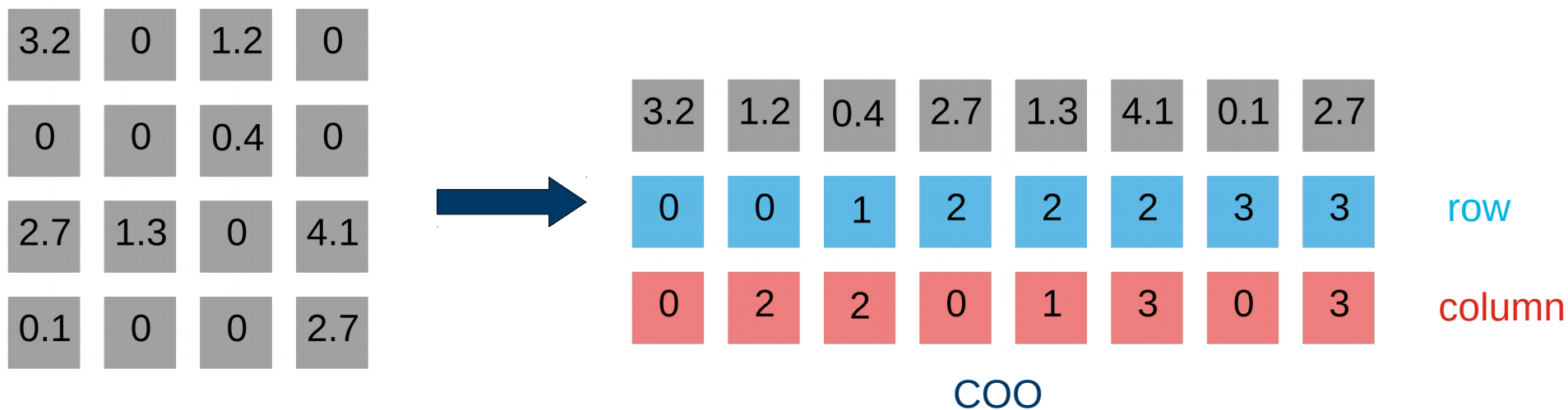
3.2	0	1.2	0
0	0	0.4	0
2.7	1.3	0	4.1
0.1	0	0	2.7

# Sparse matrix formats

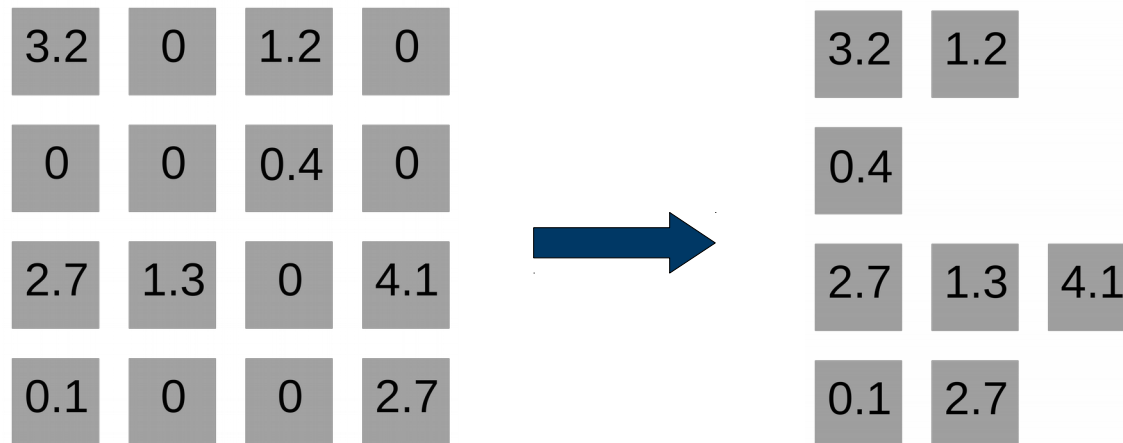




# Sparse matrix formats



# Sparse matrix formats



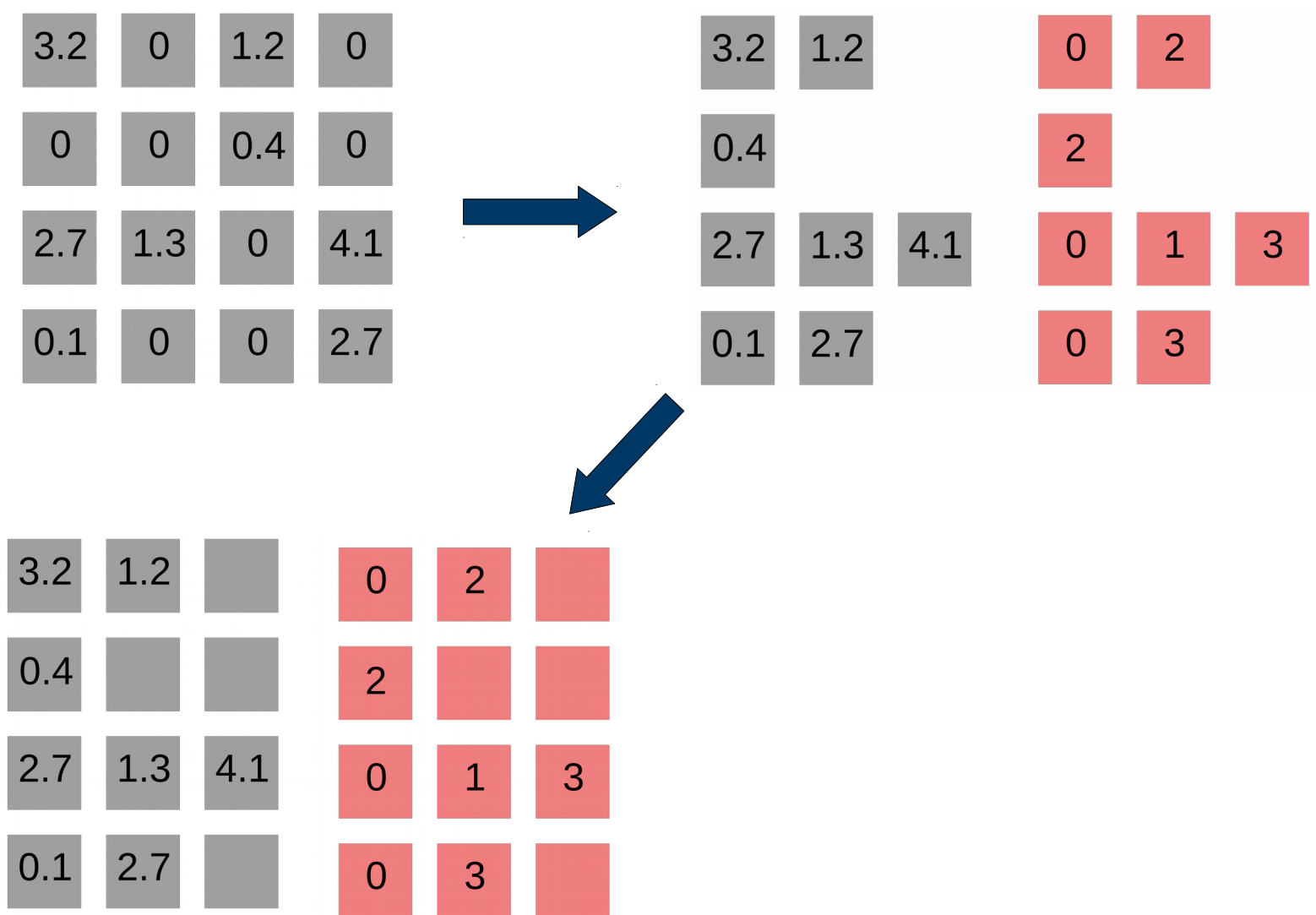
# Sparse matrix formats

3.2	0	1.2	0
0	0	0.4	0
2.7	1.3	0	4.1
0.1	0	0	2.7



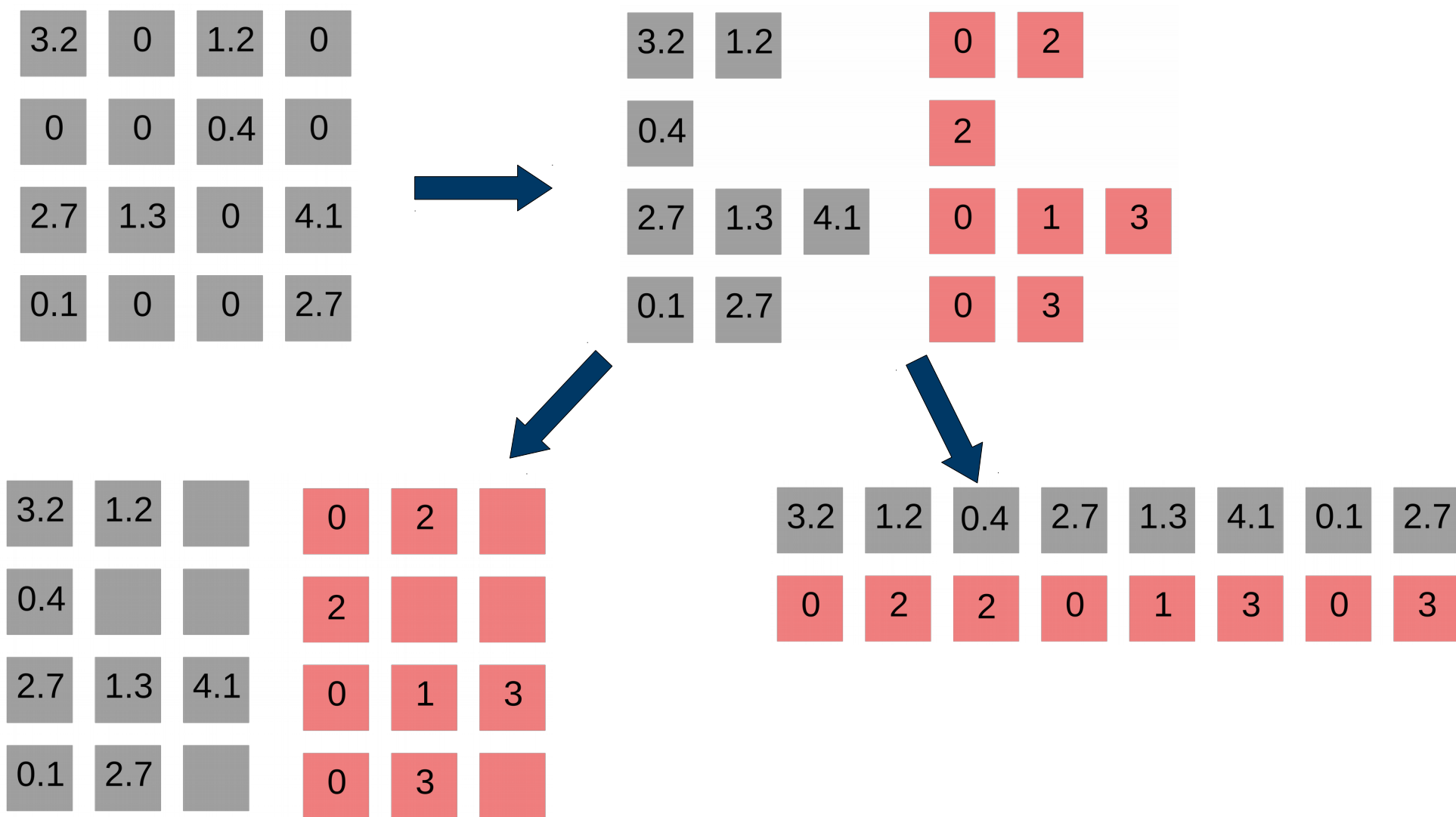
3.2	1.2			0	2	
0.4				2		
2.7	1.3	4.1		0	1	3
0.1	2.7			0	3	

# Sparse matrix formats



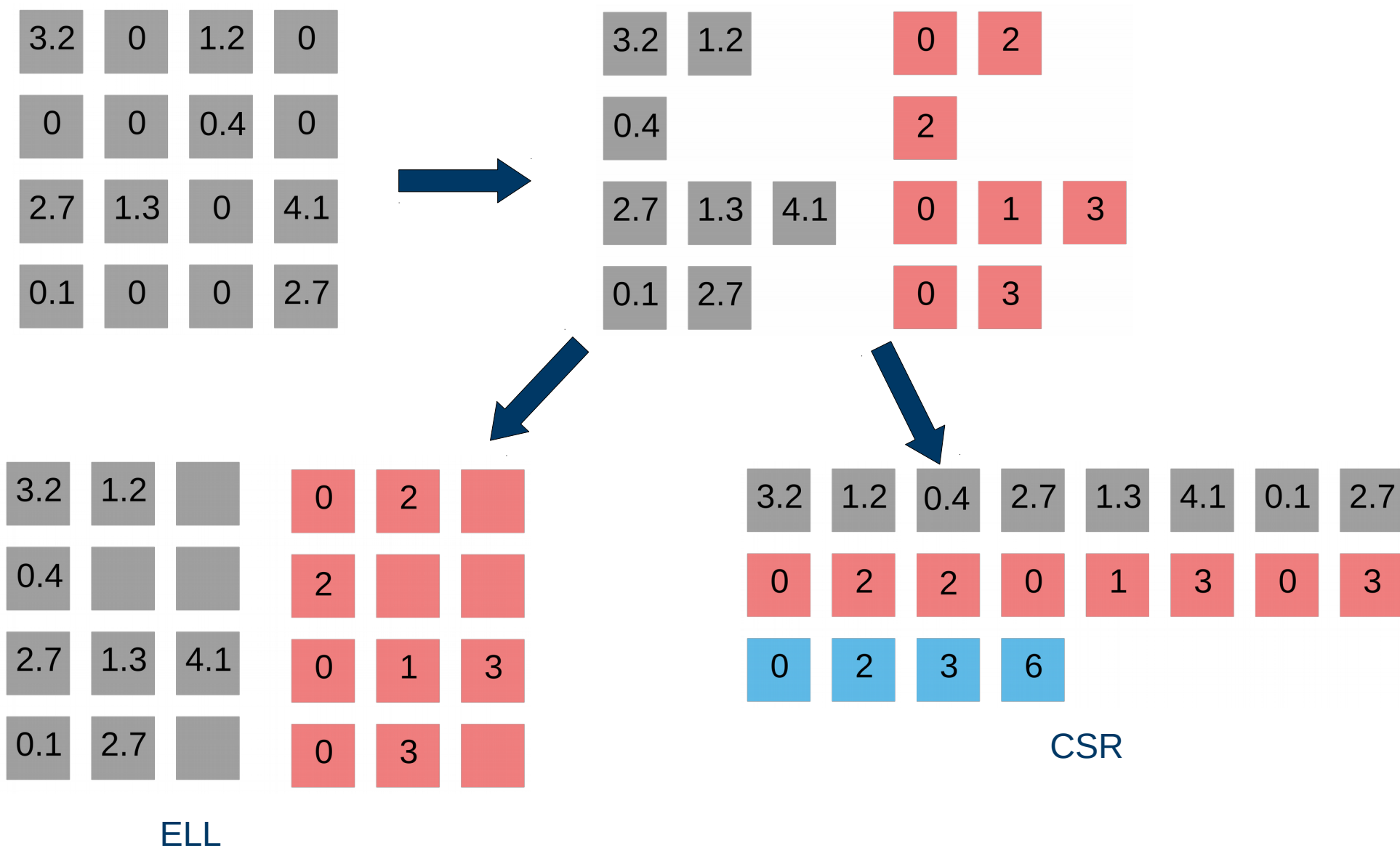
ELL

# Sparse matrix formats



ELL

# Sparse matrix formats



# Sparse matrix formats

3.2	0	1.2	0
0	0	0.4	0
2.7	1.3	0	4.1
0.1	0	0	2.7



3.2
2.7
0.1

1.3
-----

1.2
0.4

4.1
2.7

0
2
3

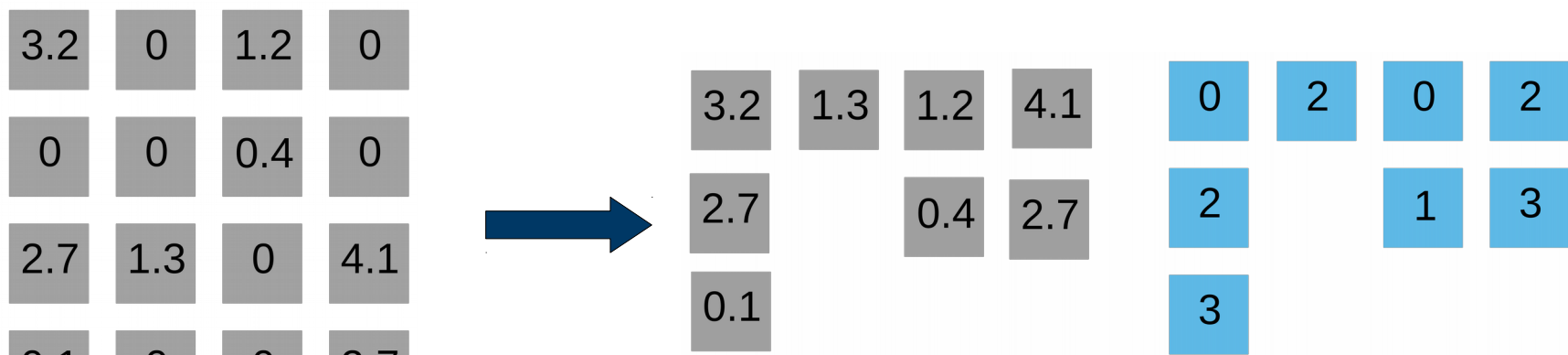
2
---

0
1

2
3

... leads to CSC

# Sparse matrix formats



... leads to CSC

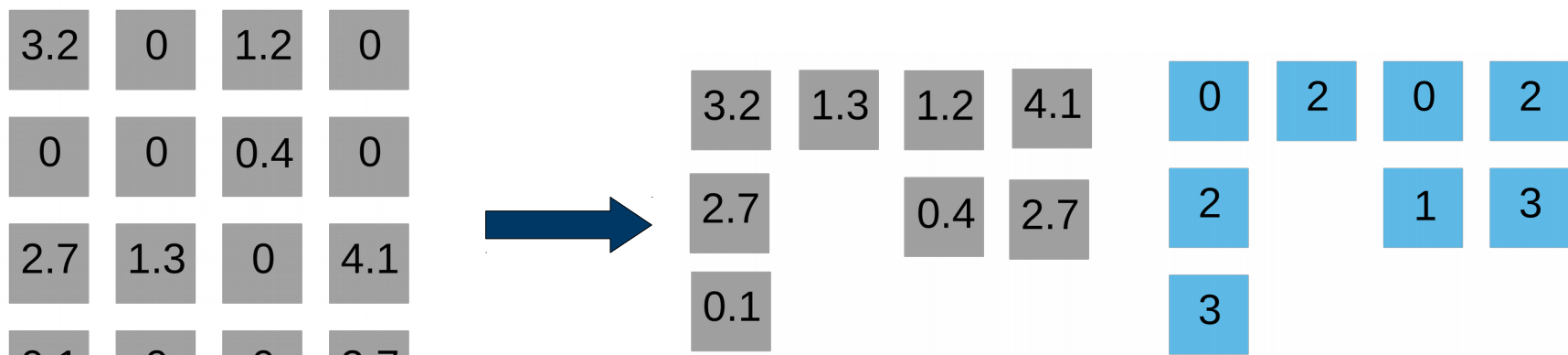
3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7
0	2	2	0	1	3	0	3
0	2	3	6				

CSR

“Standard” approach



# Sparse matrix formats



3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7
0	2	2	0	1	3	0	3
0	2	3	6				

CSR

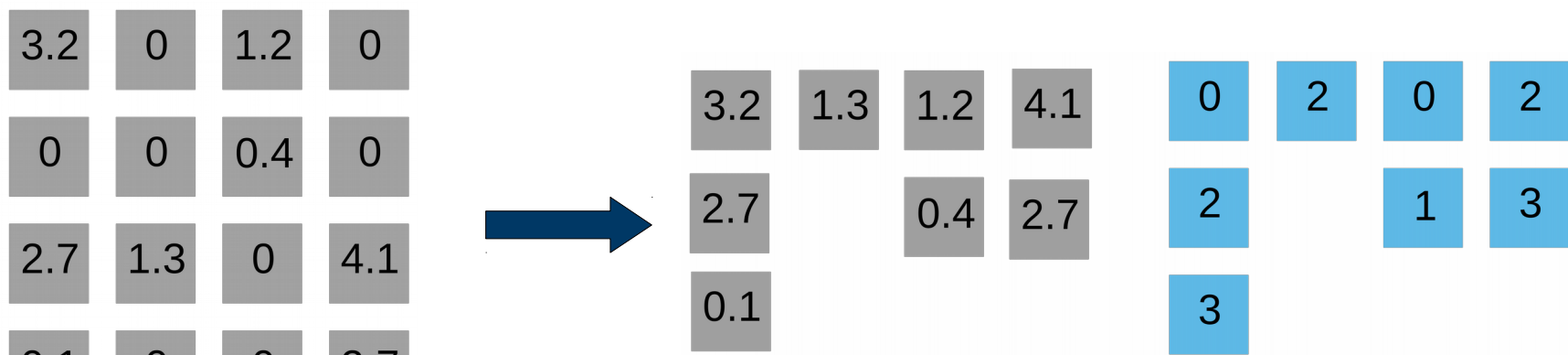
“Standard” approach

3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7
0	0	1	2	2	2	3	3
0	2	2	0	1	3	0	3

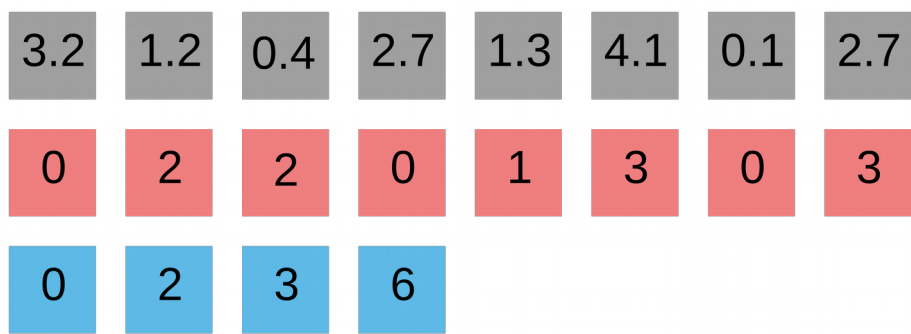
COO

Considered **inferior** to CSR (memory consumption)

# Sparse matrix formats

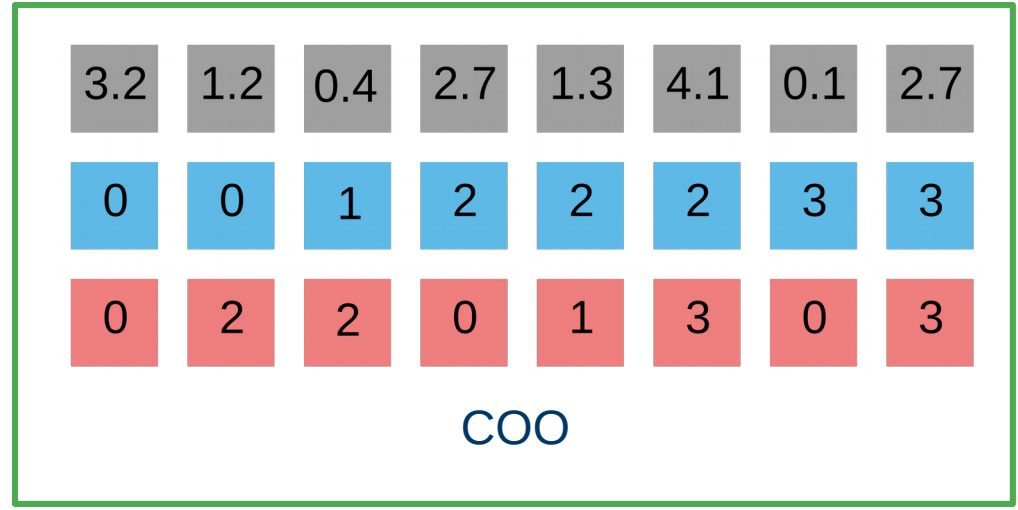


... leads to CSC



CSR

"Standard" approach



COO

Considered **inferior** to CSR (memory consumption)

# CSR SpMV

3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7	Values (val)
0	2	2	0	1	3	0	3	Column indexes (colidx)
0	2	3	6					Row pointers (rowptr)

# CSR SpMV

3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7	Values (val)
0	2	2	0	1	3	0	3	Column indexes (colidx)
0	2	3	6					Row pointers (rowptr)

$$y := Ax$$

# CSR SpMV

3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7	Values (val)
0	2	2	0	1	3	0	3	Column indexes (colidx)
0	2	3	6					Row pointers (rowptr)

$$y := Ax$$

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {  
2     for (int i = 0; i < m; ++i) {  
3         for (int j = rowptr[i]; j < rowptr[i+1]; ++j)  
4             y[i] += val[j] * x[colidx[j]];  
5     }  
6 }
```

# CSR SpMV

3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7
-----	-----	-----	-----	-----	-----	-----	-----

Values (val)

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

Column indexes (colidx)

0	2	3	6
---	---	---	---

Row pointers (rowptr)

$$y := Ax$$

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {  
2   for (int i = 0; i < m; ++i) {  
3     for (int j = rowptr[i]; j < rowptr[i+1]; ++j)  
4       y[i] += val[j] * x [ colidx[j] ];  
5   }  
6 }
```

Bell & Garland '08

- parallelize outer loop

~ cuSPARSE SpMV

# CSR SpMV

3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7
-----	-----	-----	-----	-----	-----	-----	-----

Values (val)

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

Column indexes (colidx)

0	2	3	6
---	---	---	---

Row pointers (rowptr)

$$y := Ax$$

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {  
2   for (int i = 0; i < m; ++i) {  
3     for (int j = rowptr[i]; j < rowptr[i+1]; ++j)  
4       y[i] += val[j] * x [ colidx[j] ];  
5   }  
6 }
```

Bell & Garland '08

- parallelize outer loop

~ cuSPARSE SpMV

Load imbalance!  
Non-coalescence!

# CSR SpMV

3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7	Values (val)
0	2	2	0	1	3	0	3	Column indexes (colidx)
0	2	3	6					Row pointers (rowptr)

$$y := Ax$$

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {  
2   for (int i = 0; i < m; ++i) {  
3     for (int j = rowptr[i]; j < rowptr[i+1]; ++j)  
4       y[i] += val[j] * x [ colidx[j] ];  
5   }  
6 }
```

Bell & Garland '08

- parallelize outer loop

~ cuSPARSE SpMV

Load imbalance!  
Non-coalescence!

## Specialized formats

- HYB (ELL + COO) [cuSPARSE]
- CSR5 [Liu, Vinter '15], CSR-I [Flegar, Quintana '17]
- SELL-P [Kreutzer et al.] – good memory access, parallelizes well
- ... a few new ones every year



# CSR SpMV

3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7	Values (val)
0	2	2	0	1	3	0	3	Column indexes (colidx)
0	2	3	6					Row pointers (rowptr)

$$y := Ax$$

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
2   for (int i = 0; i < m; ++i) {
3     for (int j = rowptr[i]; j < rowptr[i+1]; ++j)
4       y[i] += val[j] * x[colidx[j]];
5   }
6 }
```

Bell & Garland '08

- parallelize outer loop

~ cuSPARSE SpMV

Load imbalance!  
Non-coalescence!

Specialized formats

- HYB (ELL + COO) [cuSPARSE]
- CSR5 [Liu, Vinter '15], CSR-I [Flegar, Quintana '17]
- SELL-P [Kreutzer et al.] – good memory access, parallelizes well
- ... a few new ones every year

# CSR-I idea

---

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
2     for (int i = 0; i < m; ++i) {
3         for (int j = rowptr[i]; j < rowptr[i+1]; ++j)
4             y[i] += val[j] * x [ colidx[j] ];
5     }
6 }
```

# CSR-I idea

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {  
2   for (int i = 0; i < m; ++i) {  
3     for (int j = rowptr[i]; j < rowptr[i+1]; ++j)  
4       y[i] += val[j] * x[colidx[j]];  
5   }  
6 }
```



Collapse the two loops into one.

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {  
2   int row = -1, next_row = 0, nnz = rowptr[m];  
3   for (int i = 0; i < nnz; ++i) {  
4     while (i >= next_row) next_row = rowptr[++row+1];  
5     y[row] += val[i] * x[colidx[i]];  
6   }}
```

# CSR-I idea

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
2   for (int i = 0; i < m; ++i) {
3     for (int j = rowptr[i]; j < rowptr[i+1]; ++j)
4       y[i] += val[j] * x[colidx[j]];
5   }
6 }
```

Collapse the two loops into one.

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
2   int row = -1, next_row = 0, nnz = rowptr[m];
3   for (int i = 0; i < nnz; ++i) {
4     while (i >= next_row) next_row = rowptr[++row+1];
5     y[row] += val[i] * x[colidx[i]];
6   }}
```

Split the loop into equal chunks.

```
1 const int T = thread_count;
2 void SpMV_CSRI(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
3   int row = -1, next_row = 0, nnz = rowptr[m];
4   for (int k = 0; k < T; ++k) {
5     for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
6       while (i >= next_row) next_row = rowptr[++row+1];
7       y[row] += val[i] * x[colidx[i]];
8     }}
```

# CSR-I idea

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {  
2   for (int i = 0; i < m; ++i) {  
3     for (int j = rowptr[i]; j < rowptr[i+1]; ++j)  
4       y[i] += val[j] * x[colidx[j]];  
5   }  
6 }
```

Collapse the two loops into one.

```
1 void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {  
2   int row = -1, next_row = 0, nnz = rowptr[m];  
3   for (int i = 0; i < nnz; ++i) {  
4     while (i >= next_row) next_row = rowptr[++row+1];  
5     y[row] += val[i] * x[colidx[i]];  
6   }}
```

Split the loop into equal chunks.

```
1 const int T = thread_count;  
2 void SpMV_CSRI(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {  
3   int row = -1, next_row = 0, nnz = rowptr[m];  
4   for (int k = 0; k < T; ++k) {  
5     for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {  
6       while (i >= next_row) next_row = rowptr[++row+1];  
7       y[row] += val[i] * x[colidx[i]];  
8     }}
```

Parallelize this!

# COO SpMV

3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7	Values (val)
0	0	1	2	2	2	3	3	Row indexes (rowidx)
0	2	2	0	1	3	0	3	Column indexes (colidx)

# COO SpMV

3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7	Values (val)
0	0	1	2	2	2	3	3	Row indexes (rowidx)
0	2	2	0	1	3	0	3	Column indexes (colidx)

```
1 void SpMV_COO(int m, int *rowidx, int *colidx, float *val, float *x, float *y) {  
2     for (int i = 0; i < nnz; ++i) {  
3         y[rowidx[i]] += val[i] * x[colidx[i]];  
4     }}
```

# COO SpMV

3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7	Values (val)
0	0	1	2	2	2	3	3	Row indexes (rowidx)
0	2	2	0	1	3	0	3	Column indexes (colidx)

```
1 void SpMV_COO(int m, int *rowidx, int *colidx, float *val, float *x, float *y) {  
2     for (int i = 0; i < nnz; ++i) {  
3         y[rowidx[i]] += val[i] * x[colidx[i]];  
4     }}
```



Split the loop into equal chunks.

```
1 const int T = thread_count;  
2 void SpMV_COO(int m, int *rowidx, int *colidx, float *val, float *x, float *y) {  
3     for (int k = 0; k < T; ++k) {  
4         for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {  
5             y[rowidx[i]] += val[i] * x[colidx[i]];  
6         }}
```



# COO SpMV

3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7	Values (val)
0	0	1	2	2	2	3	3	Row indexes (rowidx)
0	2	2	0	1	3	0	3	Column indexes (colidx)

```
1 void SpMV_COO(int m, int *rowidx, int *colidx, float *val, float *x, float *y) {  
2     for (int i = 0; i < nnz; ++i) {  
3         y[rowidx[i]] += val[i] * x[colidx[i]];  
4     }}
```



Split the loop into equal chunks.

```
1 const int T = thread_count;  
2 void SpMV_COO(int m, int *rowidx, int *colidx, float *val, float *x, float *y) {  
3     for (int k = 0; k < T; ++k) { Parallelize this!  
4         for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {  
5             y[rowidx[i]] += val[i] * x[colidx[i]];  
6         }}
```

# COO vs CSR-I SpMV

COO:

```
1 | const int T = thread_count;
2 | void SpMV_COO(int m, int *rowidx, int *colidx, float *val, float *x, float *y) {
3 |     for (int k = 0; k < T; ++k) {
4 |         for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
5 |             y[rowidx[i]] += val[i] * x[colidx[i]];
6 |     }}
```

CSR-I:

```
1 | const int T = thread_count;
2 | void SpMV_CSRI(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
3 |     int row = -1, next_row = 0, nnz = rowptr[m];
4 |     for (int k = 0; k < T; ++k) {
5 |         for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
6 |             while (i >= next_row) next_row = rowptr[++row+1];
7 |             y[row] += val[i] * x[colidx[i]];
8 |     }}
```

# COO vs CSR-I SpMV

COO:

```
1 | const int T = thread_count;
2 | void SpMV_COO(int m, int *rowidx, int *colidx, float *val, float *x, float *y) {
3 |     for (int k = 0; k < T; ++k) {
4 |         for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
5 |             y[rowidx[i]] += val[i] * x[colidx[i]];
6 |     }}
```

CSR-I:

```
1 | const int T = thread_count;
2 | void SpMV_CSRI(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
3 |     int row = -1, next_row = 0, nnz = rowptr[m];
4 |     for (int k = 0; k < T; ++k) {
5 |         for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
6 |             while (i >= next_row) next_row = rowptr[++row+1];
7 |             y[row] += val[i] * x[colidx[i]];
8 |     }}
```

Race conditions!

# COO vs CSR-I SpMV

COO:

```
1 | const int T = thread_count;
2 | void SpMV_COO(int m, int *rowidx, int *colidx, float *val, float *x, float *y) {
3 |     for (int k = 0; k < T; ++k) {
4 |         for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
5 |             y[rowidx[i]] += val[i] * x[colidx[i]];
6 |     }}
```

CSR-I:

```
1 | const int T = thread_count;
2 | void SpMV_CSRI(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
3 |     int row = -1, next_row = 0, nnz = rowptr[m];
4 |     for (int k = 0; k < T; ++k) {
5 |         for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
6 |             while (i >= next_row) next_row = rowptr[++row+1];
7 |             y[row] += val[i] * x[colidx[i]];
8 |     }}
```

Race conditions!

- Use atomics
- Accumulate partial result into registers

# Getting good performance on GPUs

---

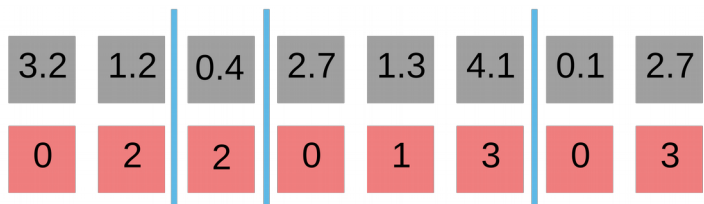
CUDA thread = 1 lane of a 32-wide SIMD unit (warp) – shared cache lines!

Spreading out threads causes strided memory access.

Assign one warp per chunk.

# Getting good performance on GPUs

CSR-I



# Getting good performance on GPUs

CSR-I

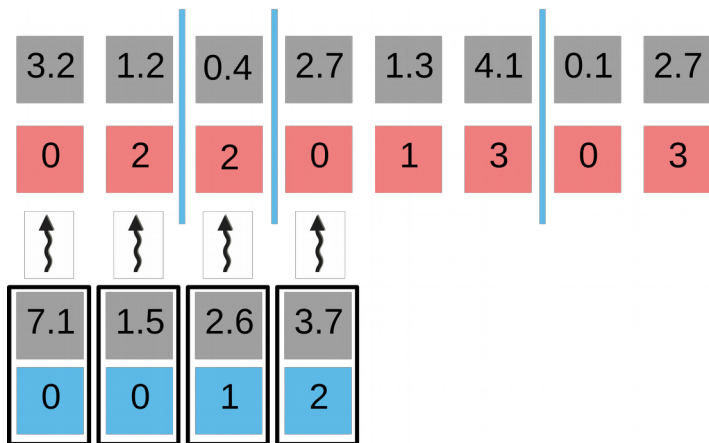
3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7
0	2	2	0	1	3	0	3

COO

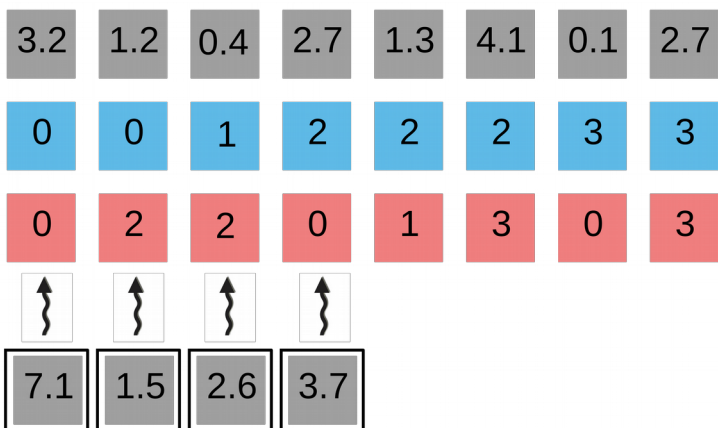
3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7
0	0	1	2	2	2	3	3
0	2	2	0	1	3	0	3

# Getting good performance on GPUs

CSR-I



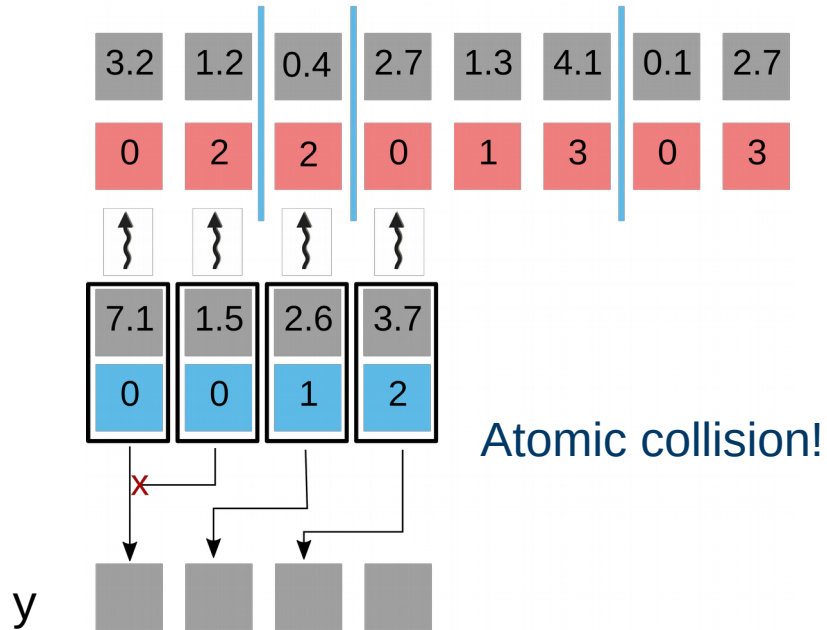
COO



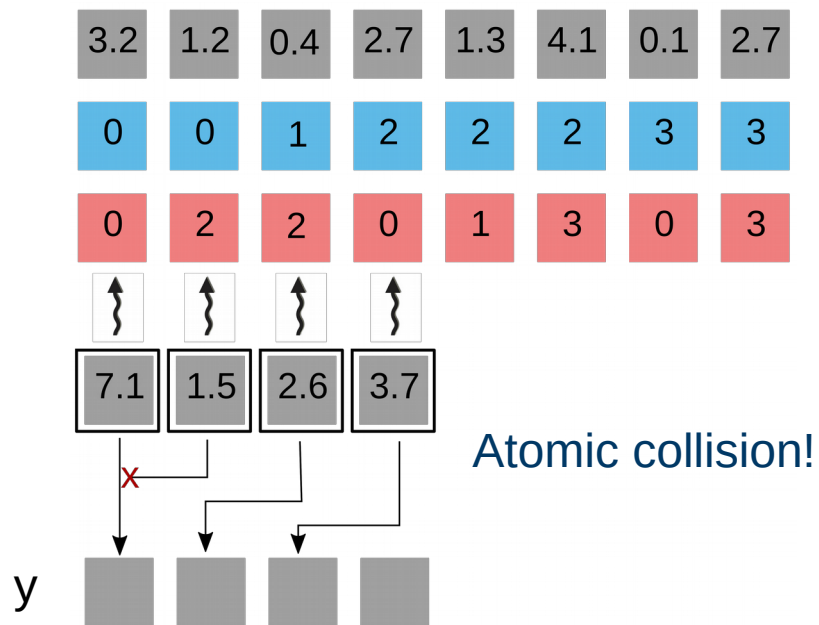


# Getting good performance on GPUs

CSR-I

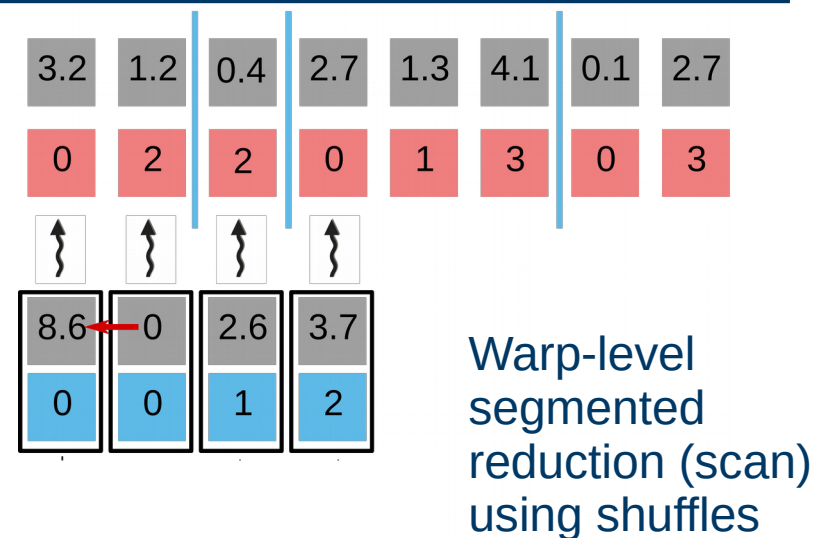
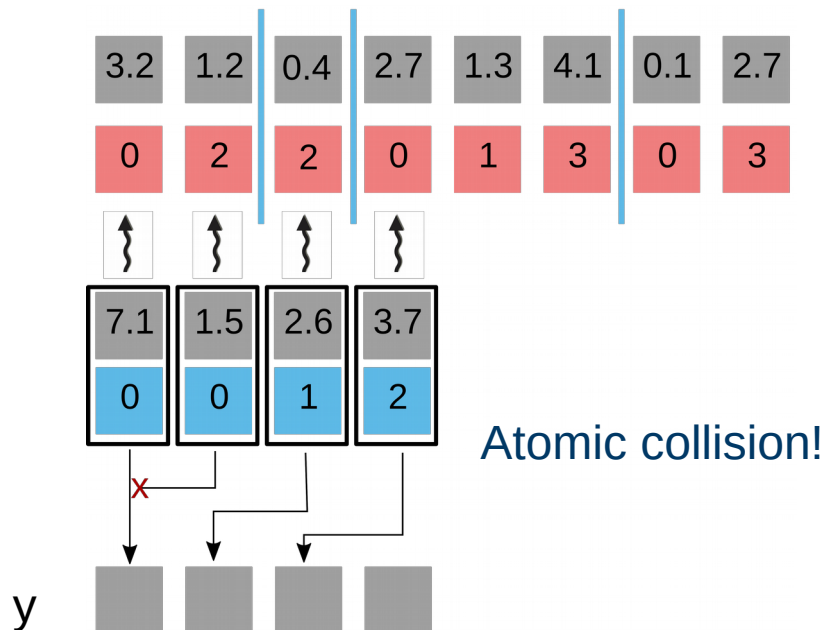


COO

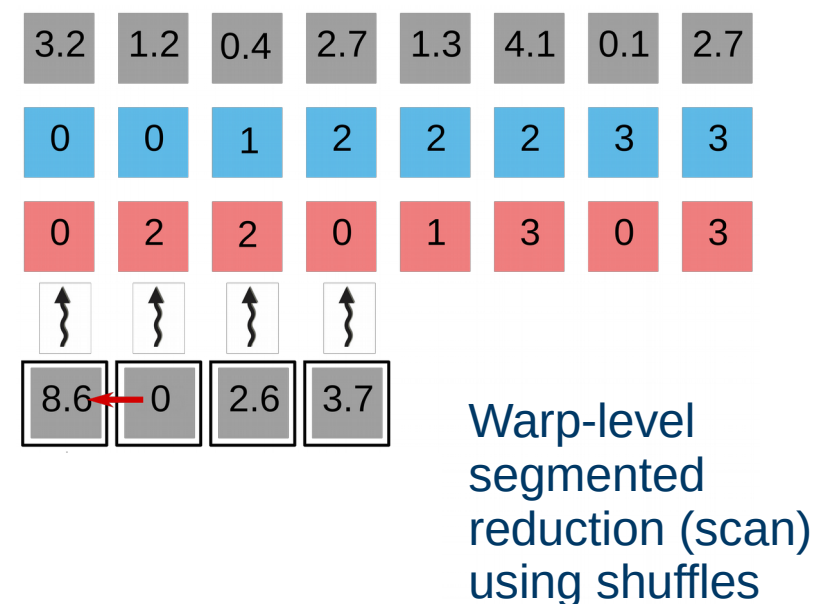
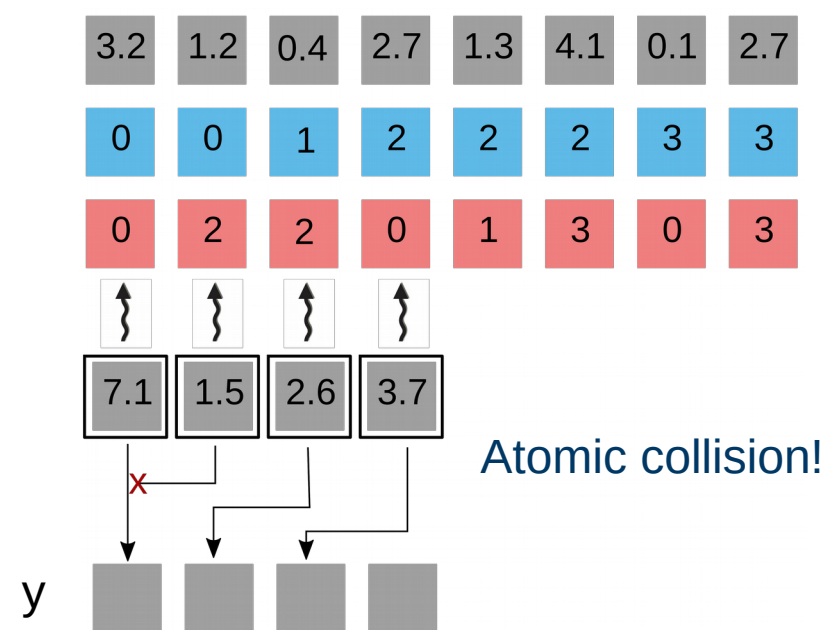


# Getting good performance on GPUs

CSR-I

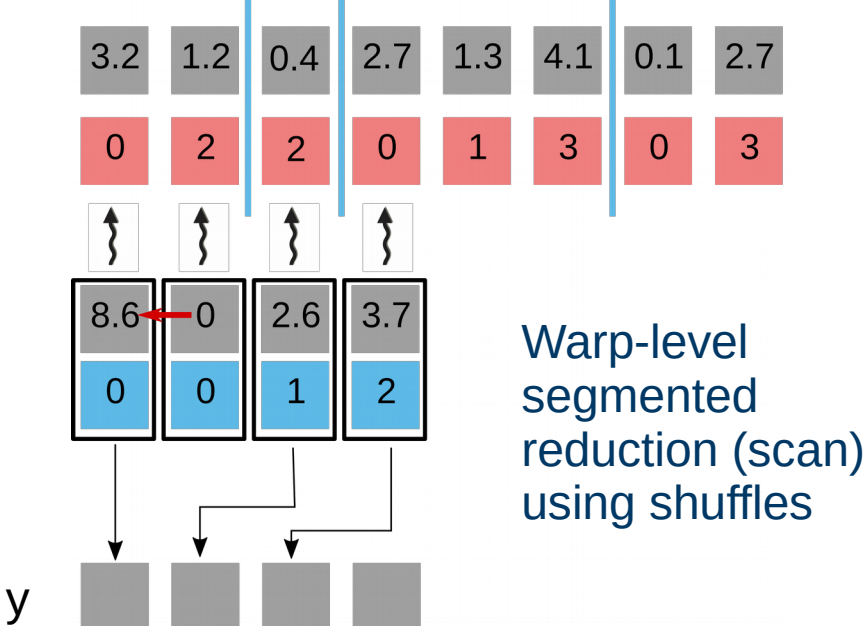
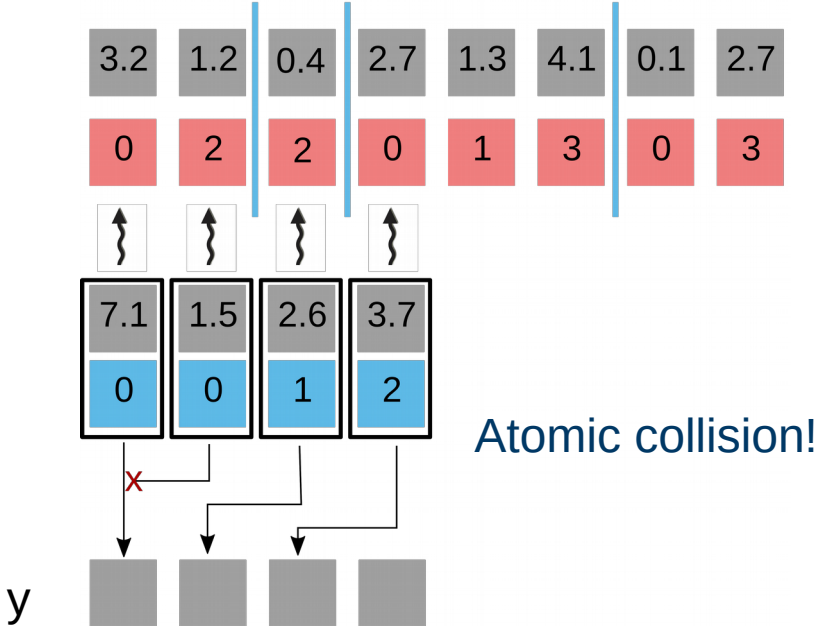


COO

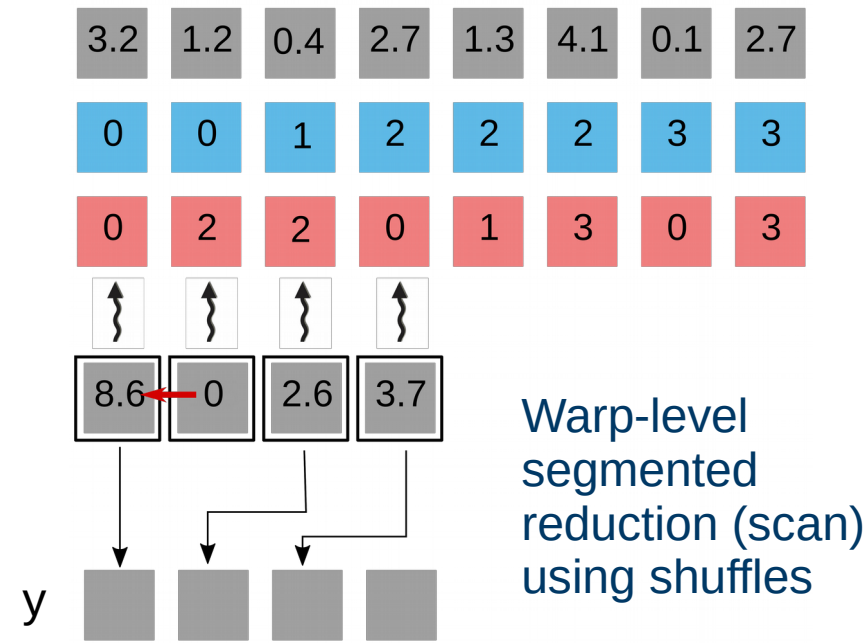
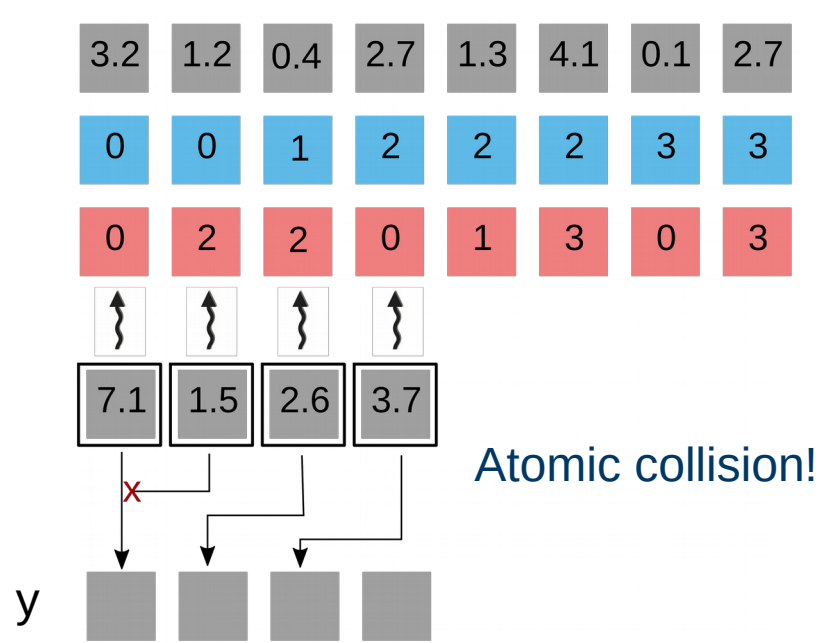


# Getting good performance on GPUs

CSR-I

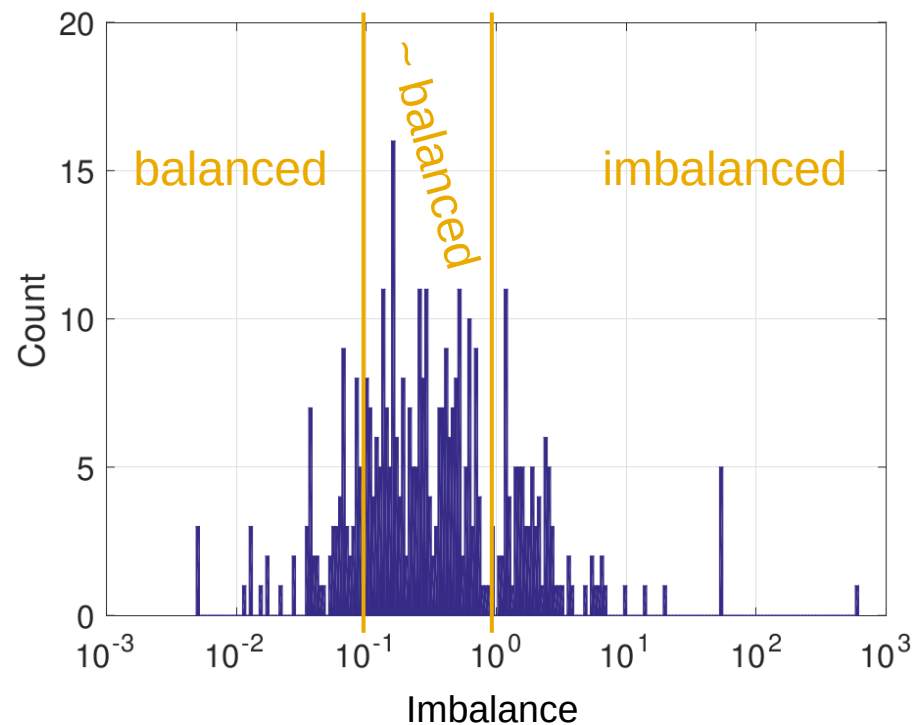
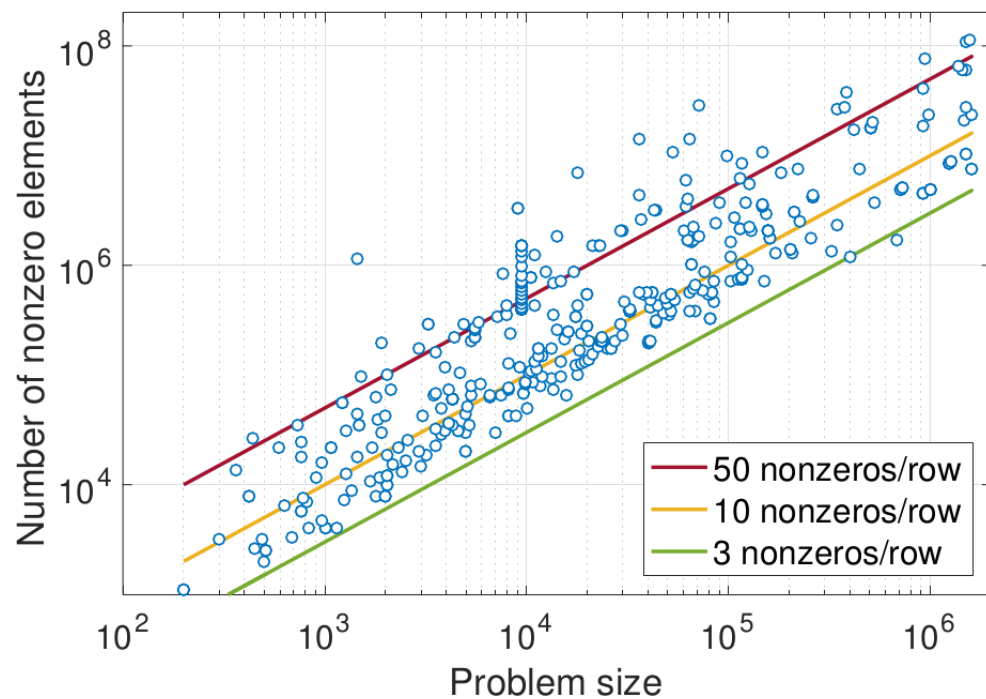


COO

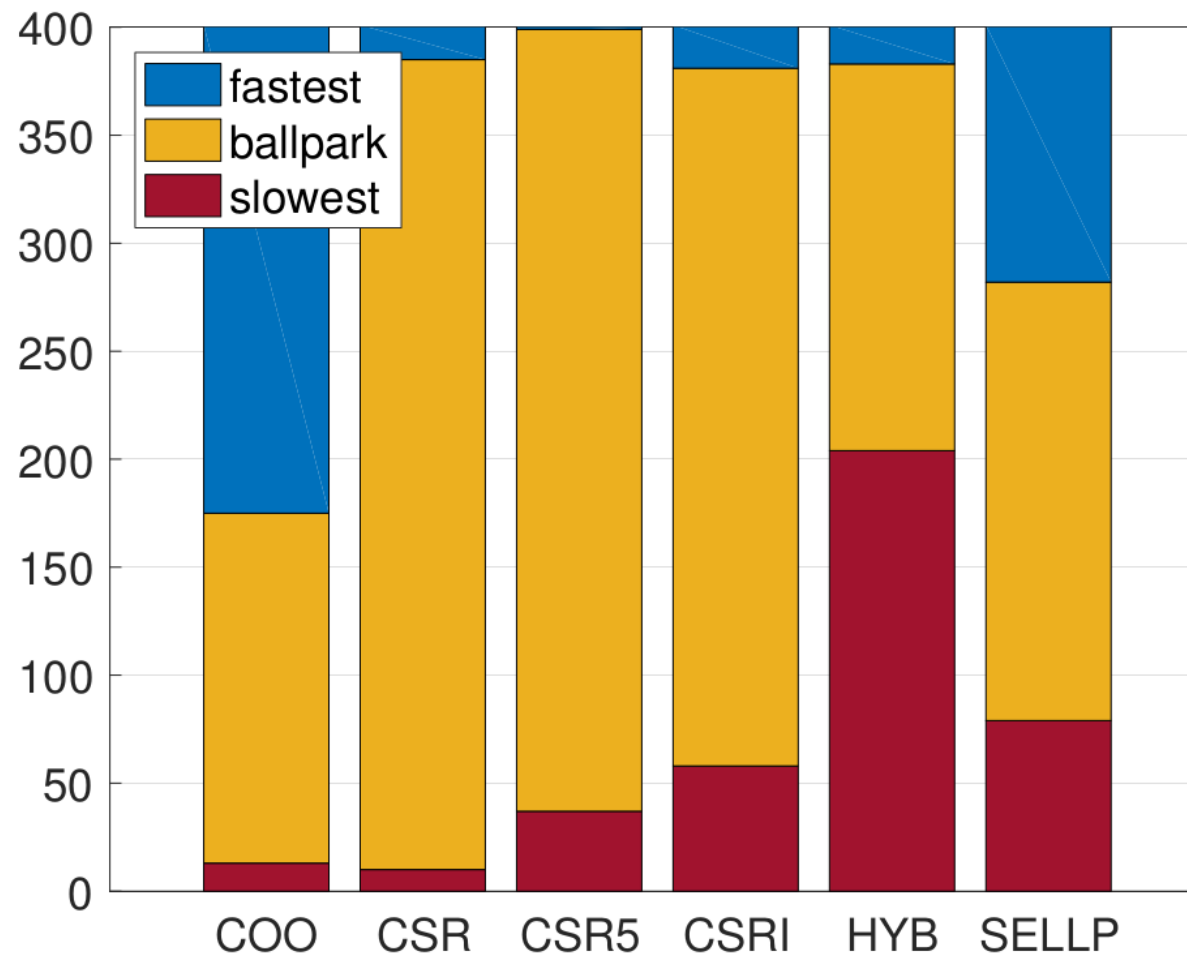


# Test matrices

400 matrices from SuiteSparse matrix collection



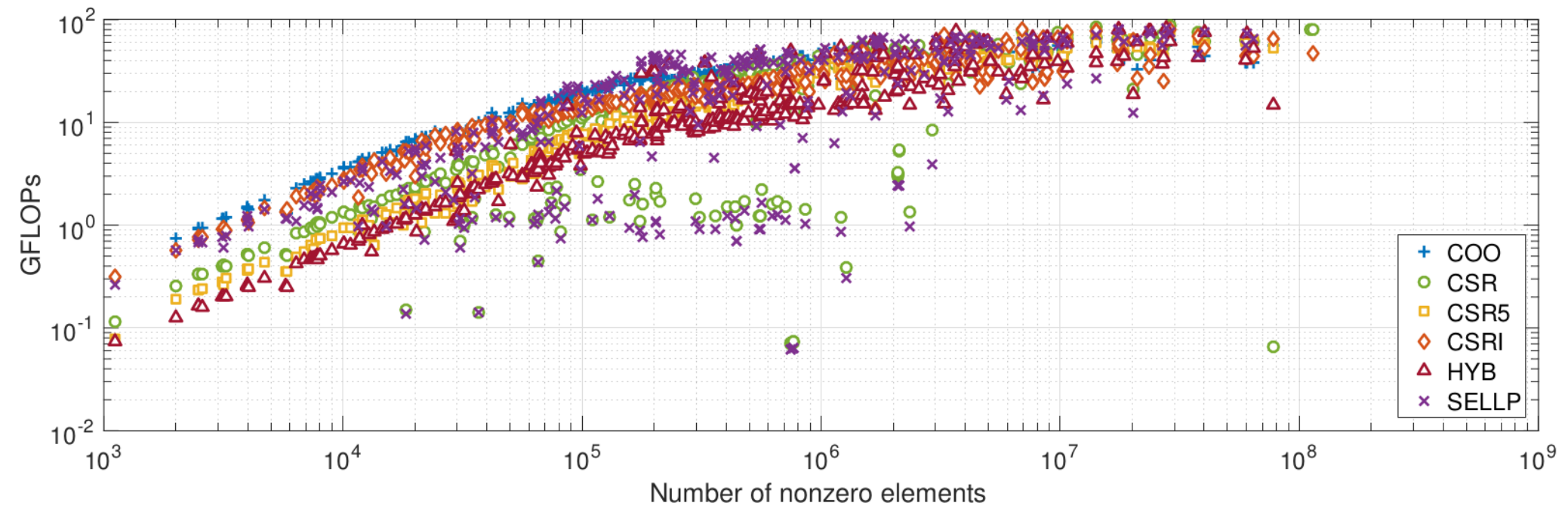
# Performance / format histogram



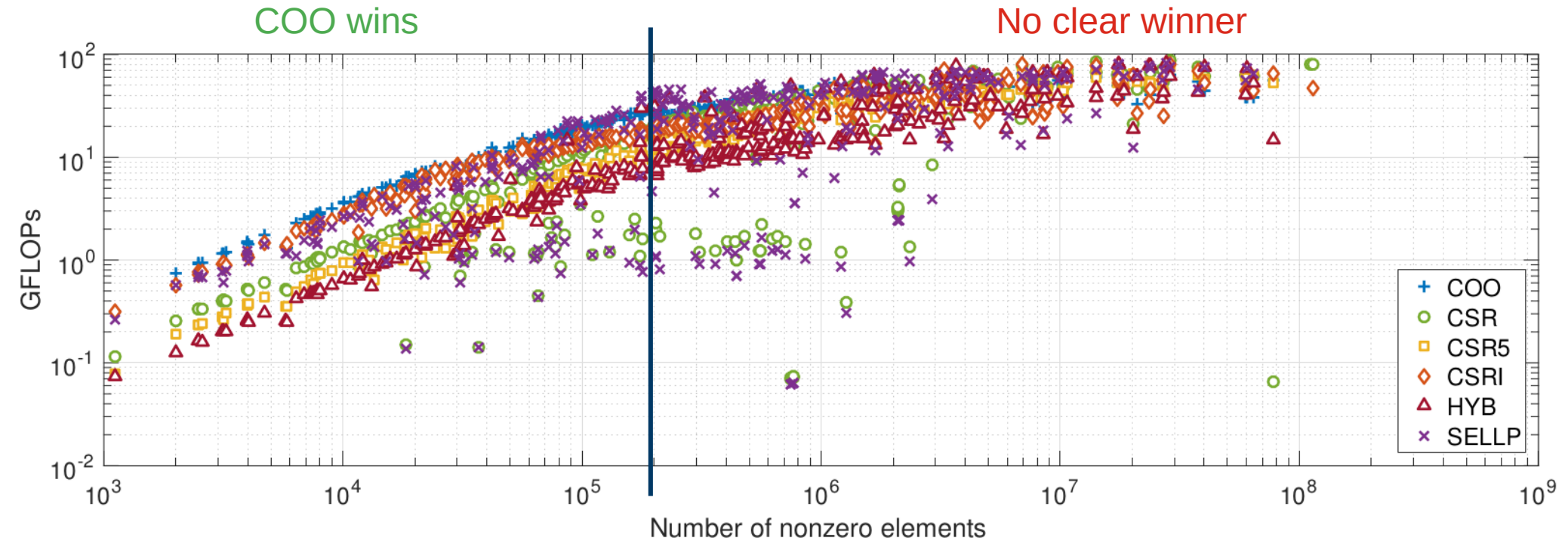
COO wins most of the cases!

\* P100 on Piz-Daint supercomputer @ CSCS

# (Too) detailed performance plot



# (Too) detailed performance plot



COO is superior for small matrices!

# Basic statistics

---

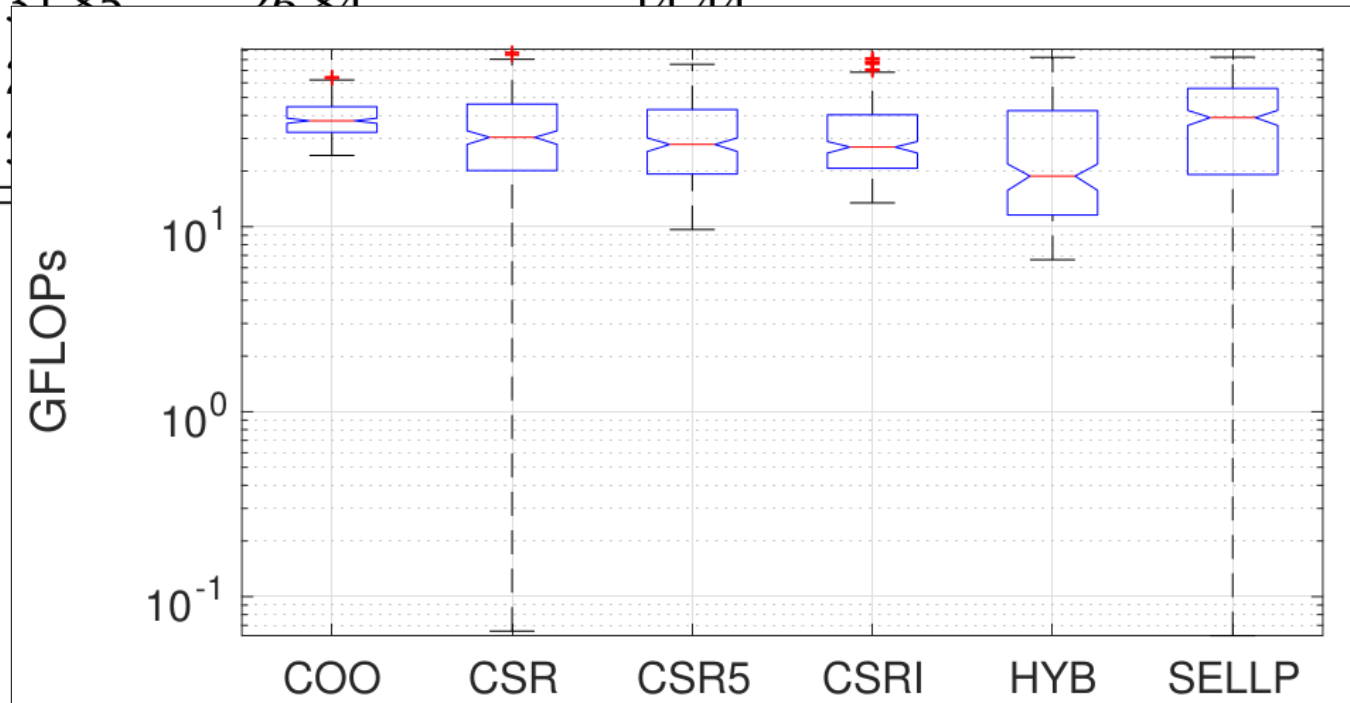
Kernel	min	max	average	median	standard-dev.
COO	24.29	64.32	38.86	37.24	9.16
CSR	0.07	87.43	32.77	30.43	20.07
CSR5	9.66	75.56	31.79	27.15	15.58
CSRI	13.47	81.21	31.85	26.84	14.44
HYB	6.64	82.43	27.98	18.74	20.22
SELLP	0.06	82.62	36.42	38.64	22.46

COO has good average & small deviation!



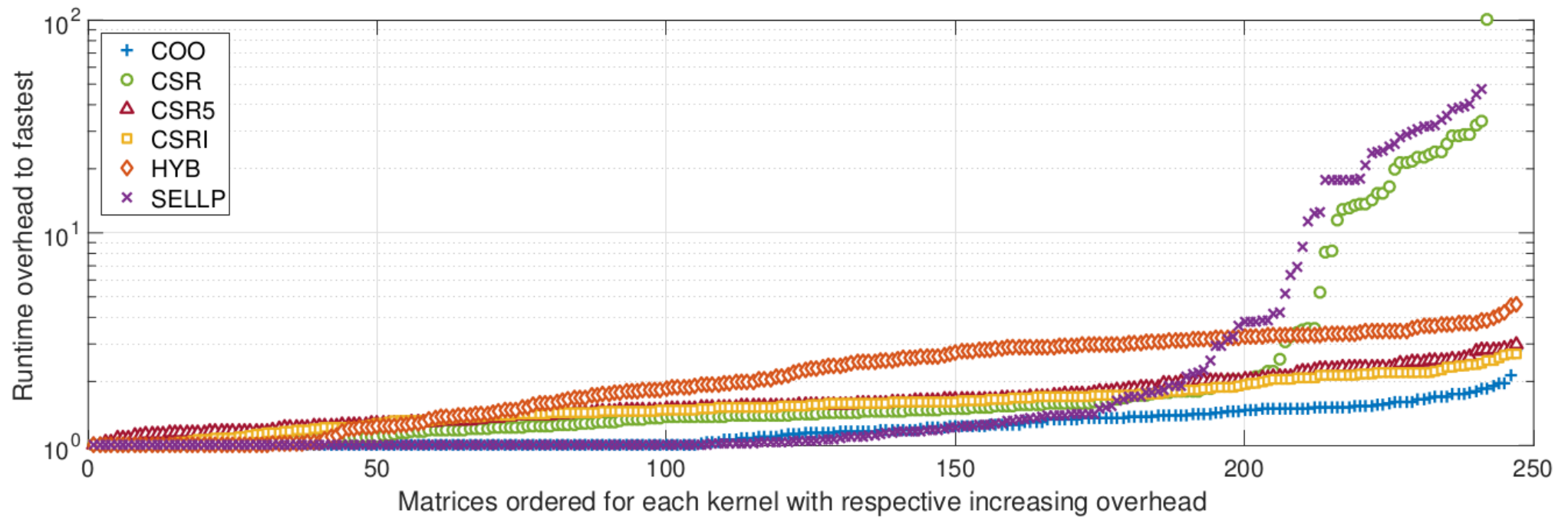
# Basic statistics

Kernel	min	max	average	median	standard-dev.
COO	24.29	64.32	38.86	37.24	9.16
CSR	0.07	87.43	32.77	30.43	20.07
CSR5	9.66	75.56	31.79	27.15	15.58
CSRI	13.47	81.21	21.85	26.84	14.44
HYB	6.64	82.43			
SELLP	0.06	82.62			



COO has good average & small deviation!

# Runtime comparison



COO is never “slow”!

# Conclusion

---

No “holy grail” format / algorithm for SpMV.

- COO is a good all-rounder.

Re-visiting “forgotten” formats may pay off.

Atomics + warp shuffles are sometimes a good alternative to reduction.

# Thank you! Questions?

All functionalities are part of the MAGMA-sparse project.

## MAGMA SPARSE

**ROUTINES** BiCG, BiCGSTAB, Block-Asynchronous Jacobi, CG, CGS, GMRES, IDR, Iterative refinement, LOBPCG, LSQR, QMR, TFQMR

**PRECONDITIONERS** ILU / IC, Jacobi, ParILU, ParILUT, Block Jacobi, ISAI

**KERNELS** SpMV, SpMM

**DATA FORMATS** CSR, ELL, SELL-P, CSR5, HYB

<http://icl.cs.utk.edu/magma/>

Scan me  
for slides!



[github.com/gflegar/talks/tree/master/sc\\_2017](https://github.com/gflegar/talks/tree/master/sc_2017)

*This research is based on a cooperation between Hartwig Anzt (Karlsruhe Institute of Technology, University of Tennessee) and Goran Flegar (Universidad Jaume I).*

