Euro-Par'17, Santiago de Compostela, 30 August

Balanced CSR Sparse Matrix-Vector Product on Graphics Processors

Goran Flegar, Enrique S. Quintana-Ortí

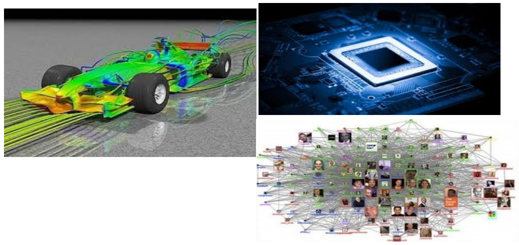
Scan me for slides!

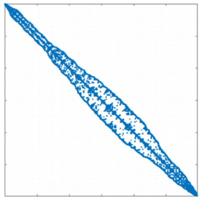
github.com/gflegar/talks/europar_2017

flegar@uji.es

MAGMA-sparse software library

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





$$Ax = b$$

MAGMA SPARSE

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

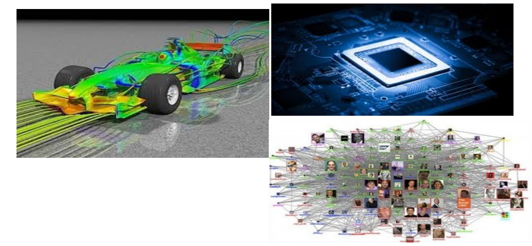
THE UNIVERSITY OF TENNESSEE KNOXVILLE

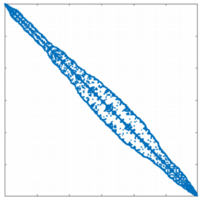
KIT Karlsruher Institut für Technologie

UNIVERSITAT 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





$$Ax = b$$

MAGMA SPARSE

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

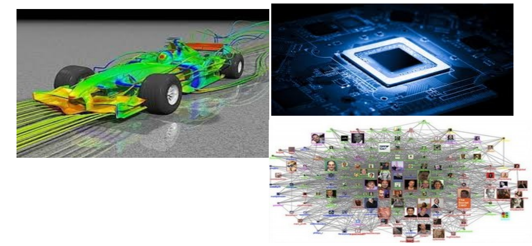
THE UNIVERSITY OF TENNESSEE KNOXVILLE

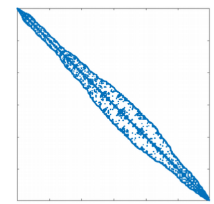
KIT Karlsruher Institut für Technologie

UNIVERSITAT 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





$$Ax = b$$

MAGMA SPARSE

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

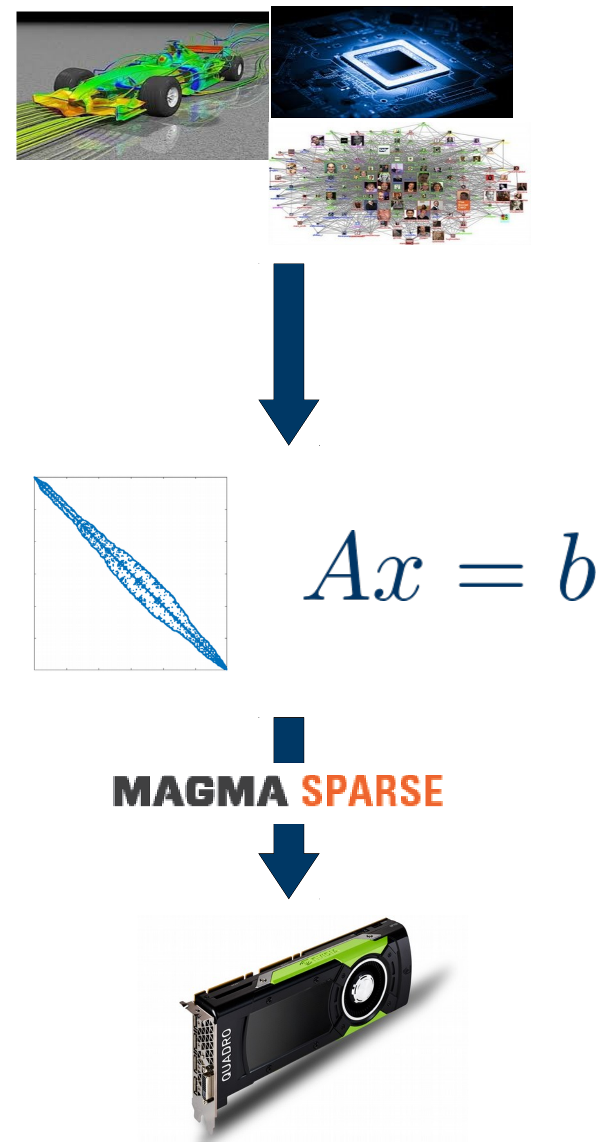
THE UNIVERSITY OF TENNESSEE KNOXVILLE

KIT Karlsruher Institut für Technologie

UNIVERSITAT 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, Knoxvile; Karlsruhe Institute of Technology; University Jaume I

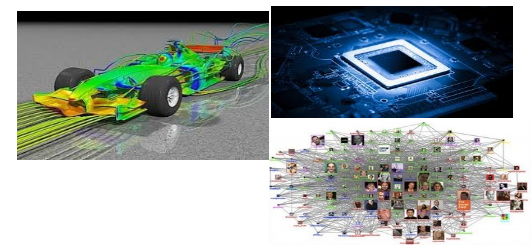
THE UNIVERSITY OF TENNESSEE KNOXVILLE

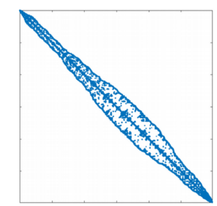
KIT Karlsruher Institut für Technologie

UNIVERSITAT 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





$$Ax = b$$

MAGMA SPARSE

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

THE UNIVERSITY OF TENNESSEE KNOXVILLE

KIT Karlsruher Institut für Technologie

UNIVERSITAT 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

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	0.4	2.7	1.3	4.1	0.1	2.7

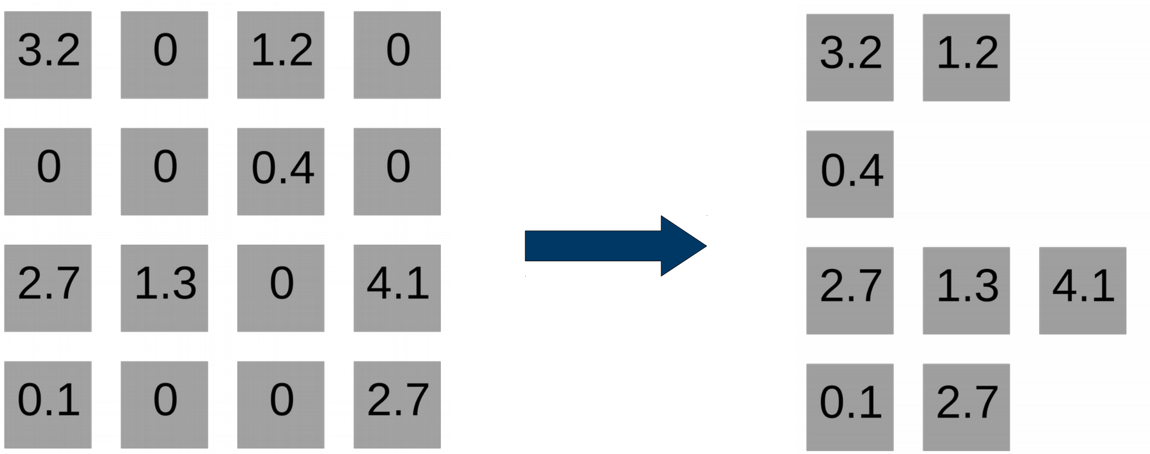
Sparse matrix formats

$$\begin{bmatrix} 3.2 & 0 & 1.2 & 0 \\ 0 & 0 & 0.4 & 0 \\ 2.7 & 1.3 & 0 & 4.1 \\ 0.1 & 0 & 0 & 2.7 \end{bmatrix} \rightarrow$$

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

COO

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	
0.4		
2.7	1.3	4.1
0.1	2.7	

0	2	
2		
0	1	3
0	3	

Sparse matrix formats

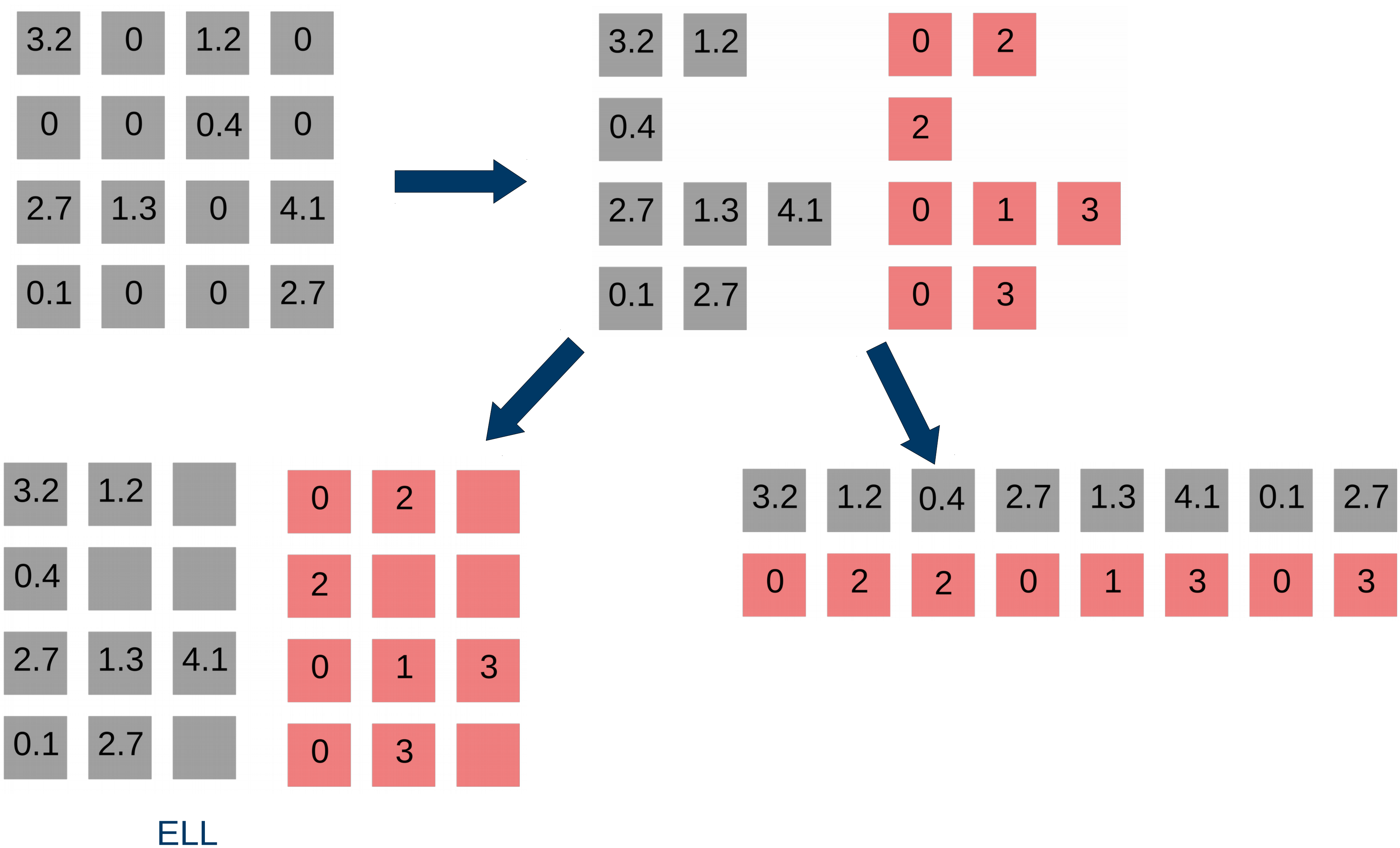
3.2	0	1.2	0
0	0	0.4	0
2.7	1.3	0	4.1
0.1	0	0	2.7

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

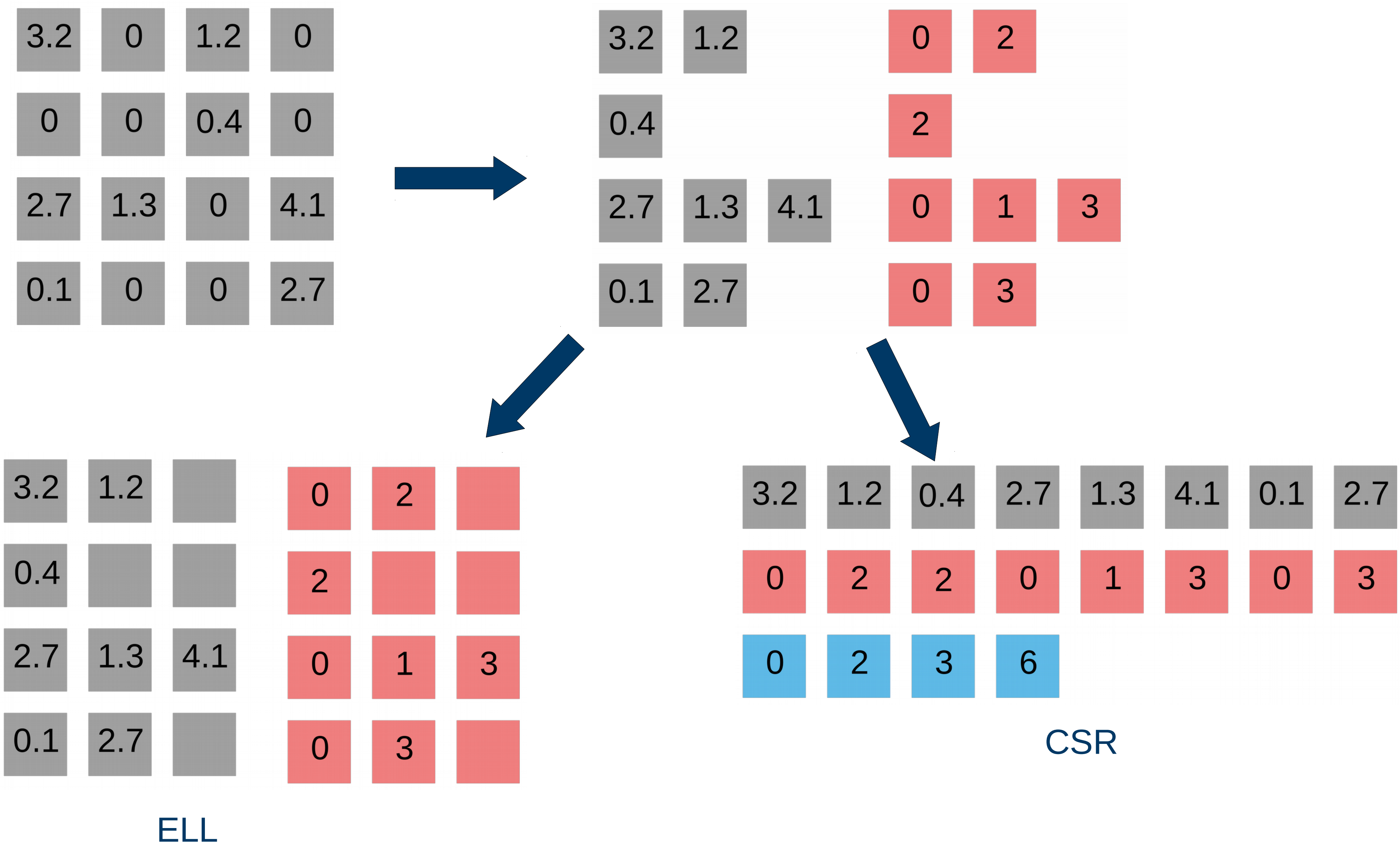
3.2	1.2		0	2	
0.4			2		
2.7	1.3	4.1	0	1	3
0.1	2.7		0	3	

ELL

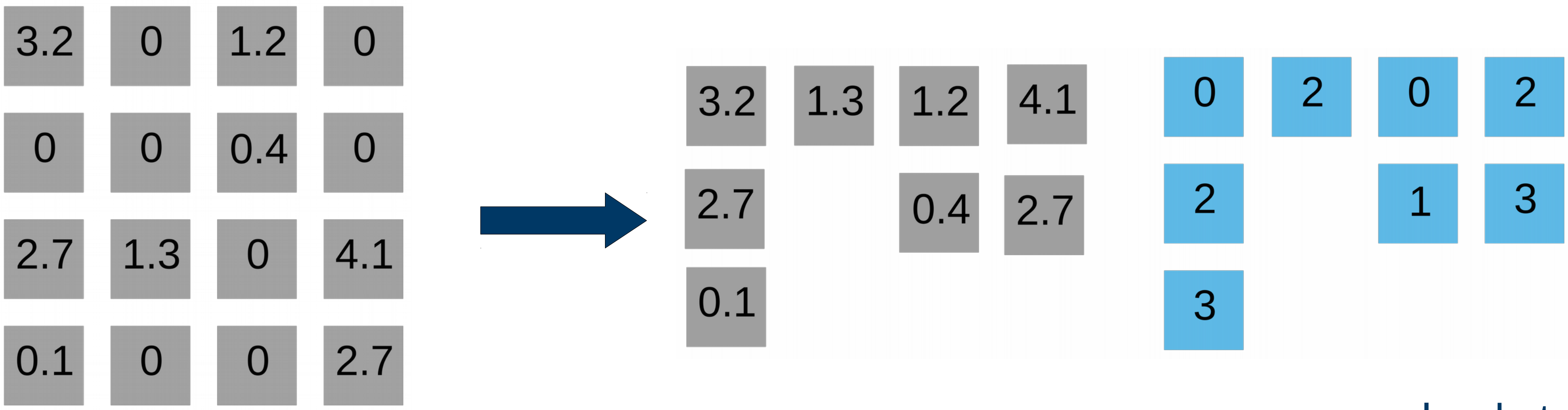
Sparse matrix formats



Sparse matrix formats

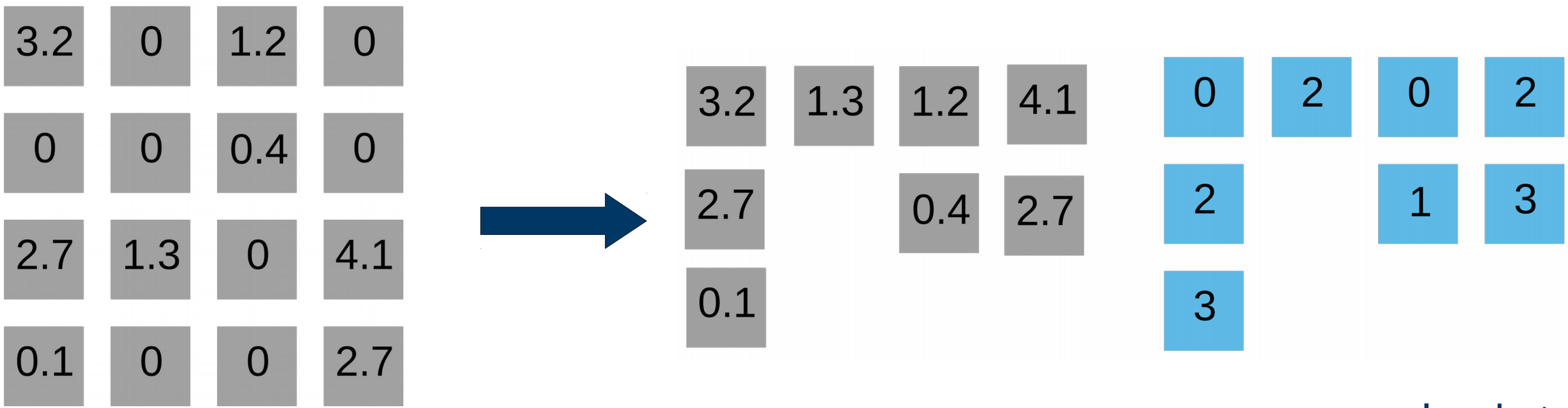


Sparse matrix formats

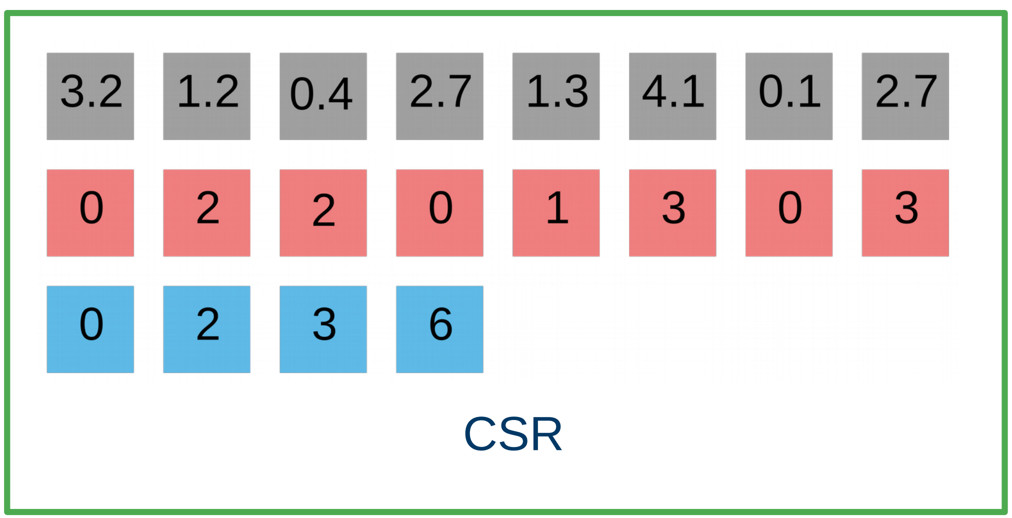


… leads to CSC

Sparse matrix formats



… leads to CSC



“Standard” approach

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)

$$y := Ax$$

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

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)

$$y := Ax$$

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

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)

$$y := Ax$$

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

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)

$$y := Ax$$

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

Bell & Garland '08

- parallelize outer loop

~ cuSPARSE SpMV

Load imbalance!
Non-coalescence!

Specialized formats

- HYB – ELL + COO
- SELL-P – 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)

$$y := Ax$$

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

Bell & Garland ‘08
- parallelize outer loop

~ cuSPARSE SpMV

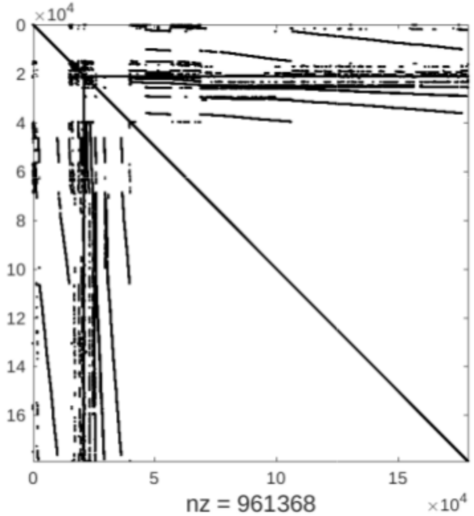
Load imbalance!
Non-coalescence!

Specialized formats
- HYB – ELL + COO
- SELL-P – good memory access, parallelizes well
- … a few new ones every year

Library code – SpMV is not the only kernel!

Motivating example

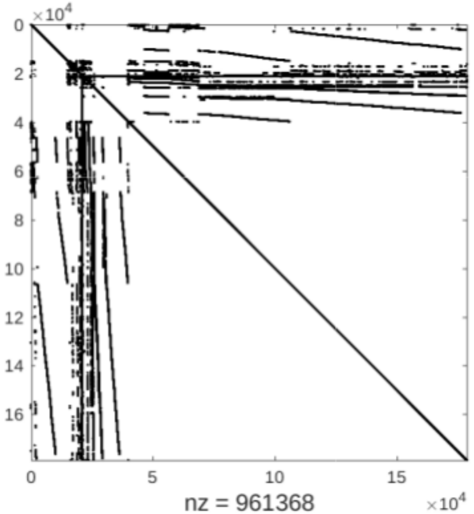
Freescale/transient

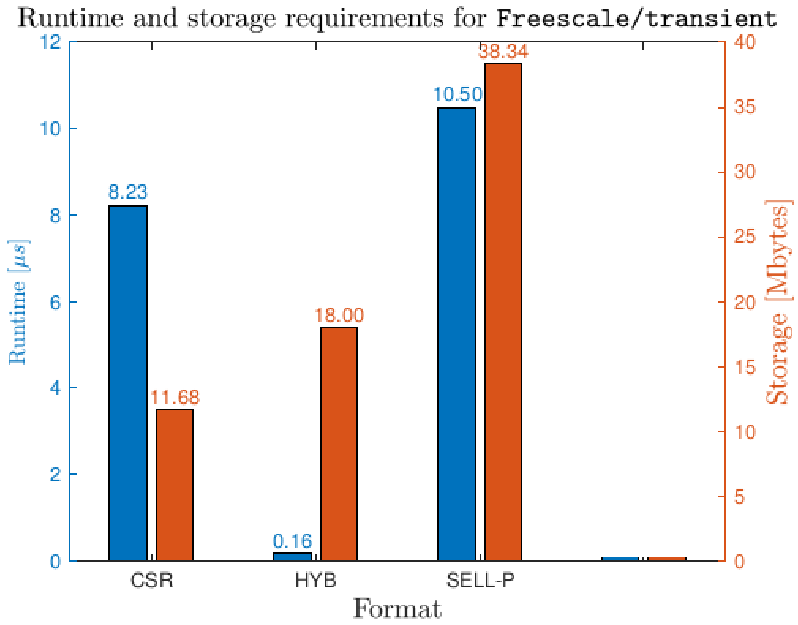


Motivating example

* GTX 1080

Freescale/transient



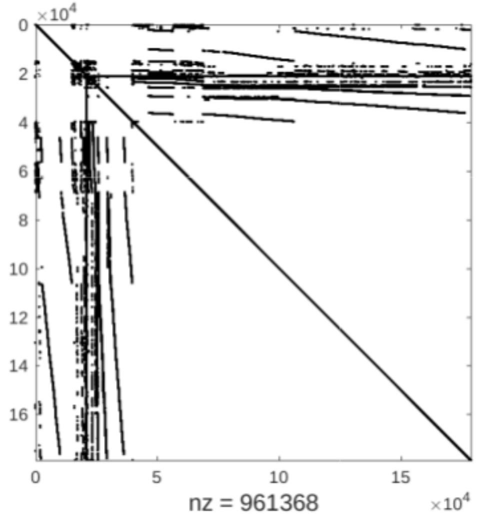


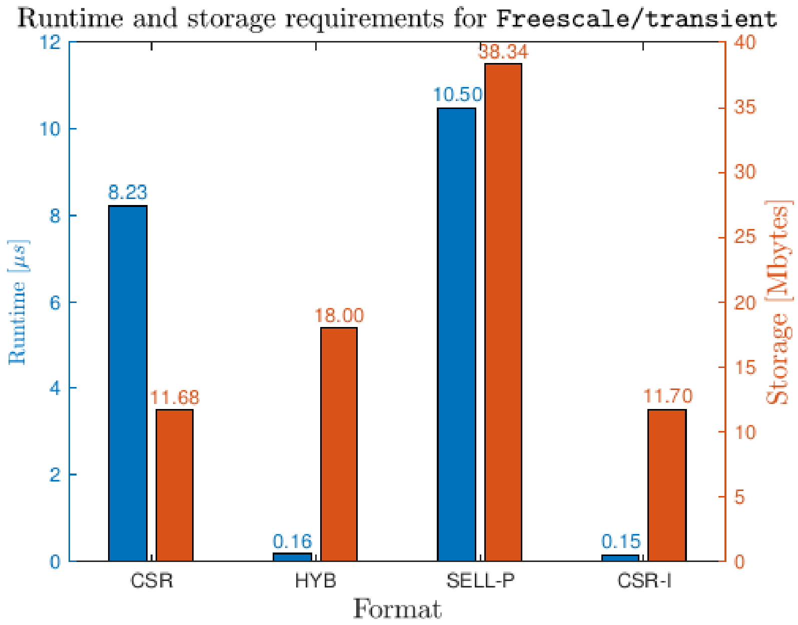
Can we do better than HYB using CSR?

Motivating example

* GTX 1080

Freescale/transient





55x speedup

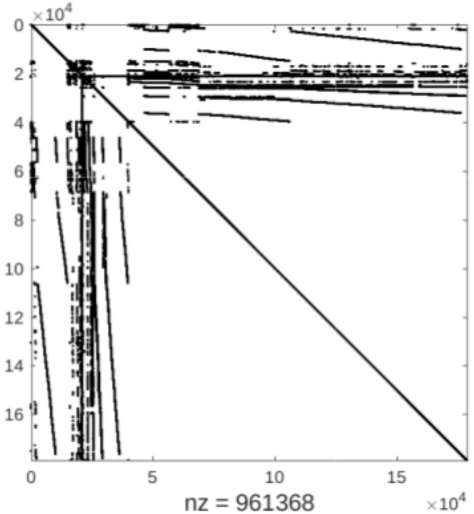
Can we do better than HYB using CSR?

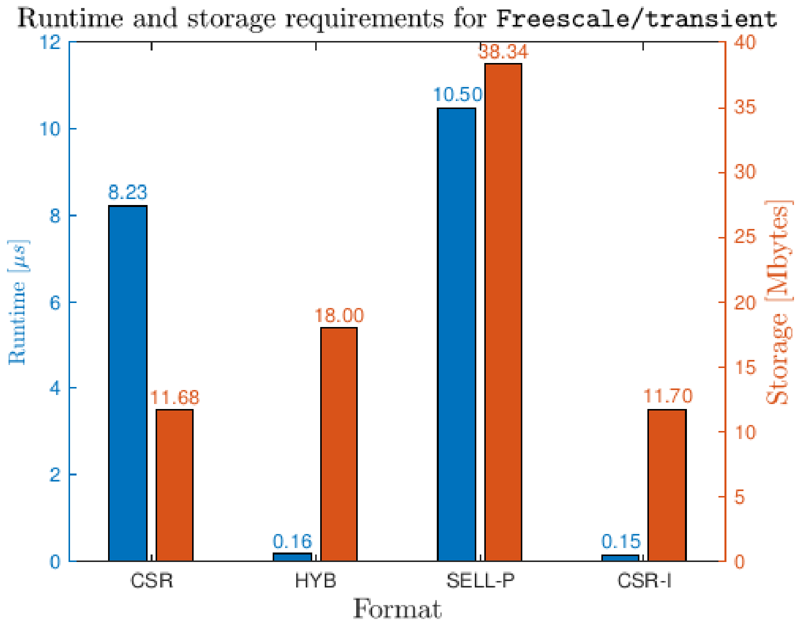
YES!

Motivating example

* GTX 1080

Freescale/transient





55x speedup

Can we do better than HYB using CSR?

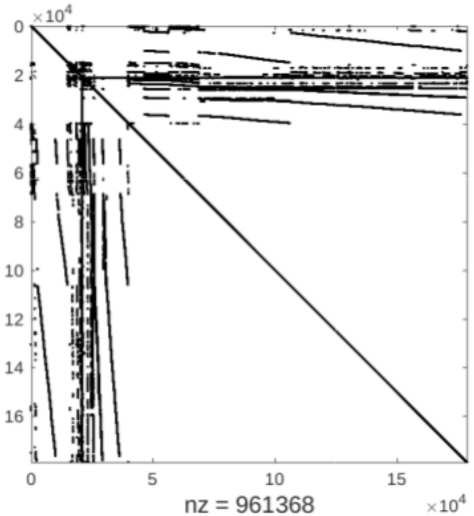
YES!

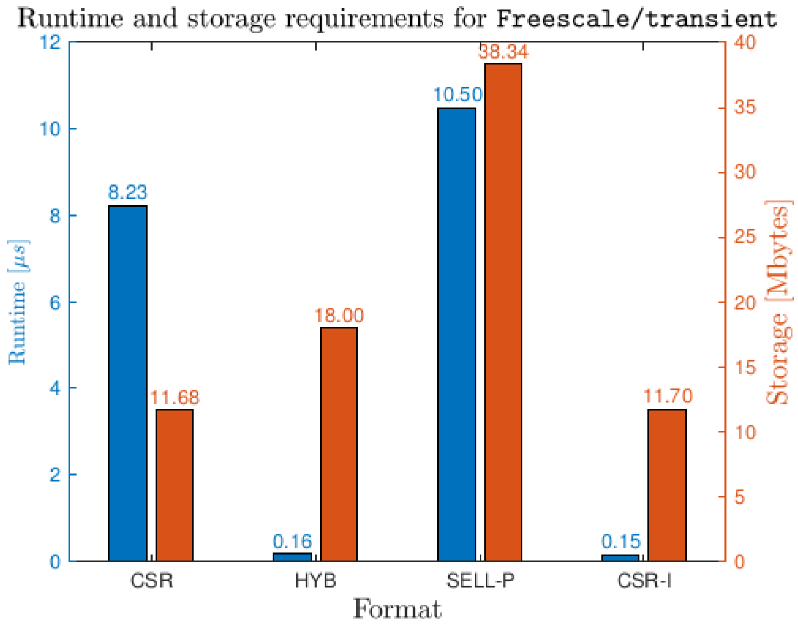
New **CSR-I** algorithm (and format).

Motivating example

* GTX 1080

Freescale/transient





55x speedup

Can we do better than HYB using CSR?

YES!

New **CSR-I** algorithm (and format).

Only need to add a small amount of data to vanilla CSR. Add - but not modify the existing format!

How to do it? By force!

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

How to do it? By force!

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

Merge the two loops into one.

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

How to do it? By force!

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

Merge the two loops into one.

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

Split the loop into equal chunks.

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

How to do it? By force!

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

Merge the two loops into one.

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

Split the loop into equal chunks.

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

Parallelize this!

How to do it? By force!

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

Merge the two loops into one.

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

Split the loop into equal chunks.

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

Parallelize this!

Race conditions!

How to do it? By force!

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

Merge the two loops into one.

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

Split the loop into equal chunks.

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

Parallelize this!

Race conditions!

- Use atomics
- Accumulate partial result into registers

How to do it? By force!

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

Merge the two loops into one.

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

Split the loop into equal chunks.

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

Parallelize this!

State between outer loop iterations!

Race conditions!

- Use atomics
- Accumulate partial result into registers

How to do it? By force!

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

Merge the two loops into one.

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

Split the loop into equal chunks.

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

Parallelize this!

State between outer loop iterations!

Race conditions!

- Use atomics
- Accumulate partial result into registers

Precompute starting value of “row” for each thread.

Getting good performance on GPUs

CUDA thread = 1 lane of a 32-wide SIMD unit (warp)

Getting good performance on GPUs

CUDA thread = 1 lane of a 32-wide SIMD unit (warp)

Spreading out threads causes strided memory access.

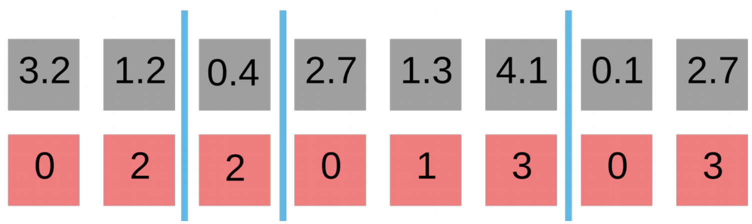
Assign one warp per chunk.

Getting good performance on GPUs

CUDA thread = 1 lane of a 32-wide SIMD unit (warp)

Spreading out threads causes strided memory access.

Assign one warp per chunk.

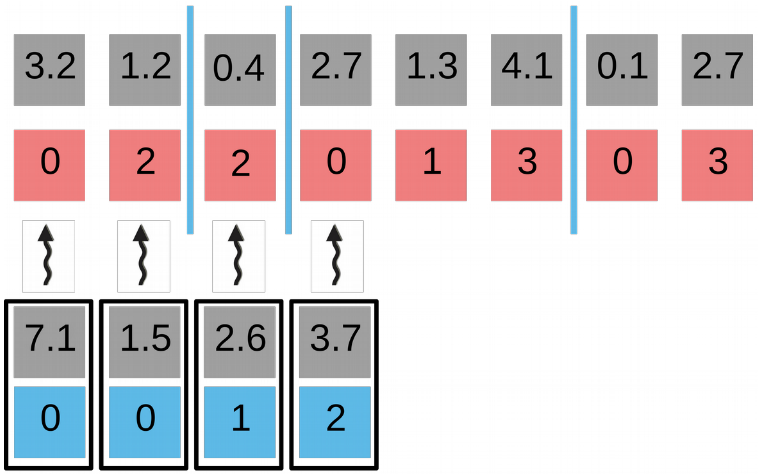


Getting good performance on GPUs

CUDA thread = 1 lane of a 32-wide SIMD unit (warp)

Spreading out threads causes strided memory access.

Assign one warp per chunk.

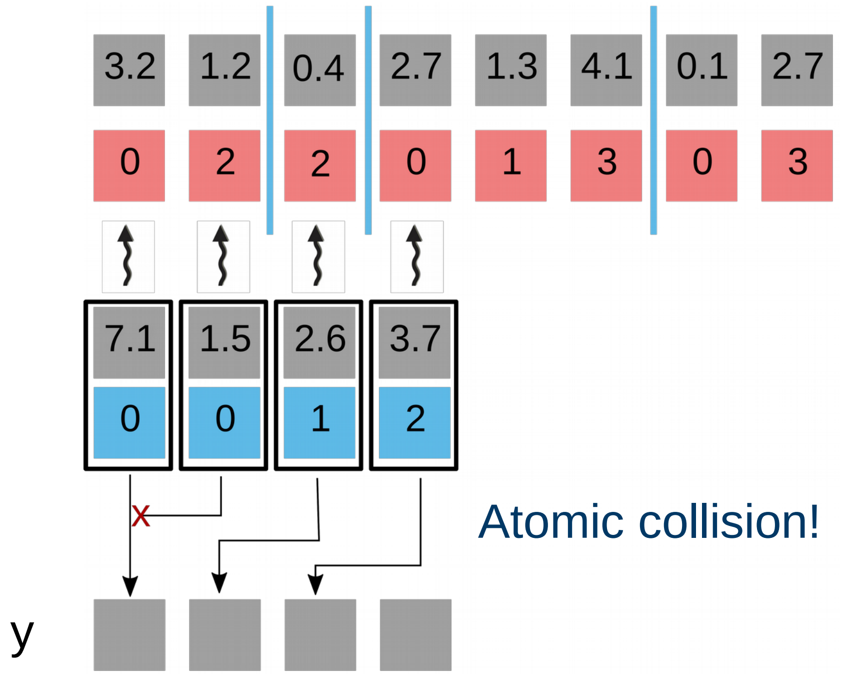


Getting good performance on GPUs

CUDA thread = 1 lane of a 32-wide SIMD unit (warp)

Spreading out threads causes strided memory access.

Assign one warp per chunk.



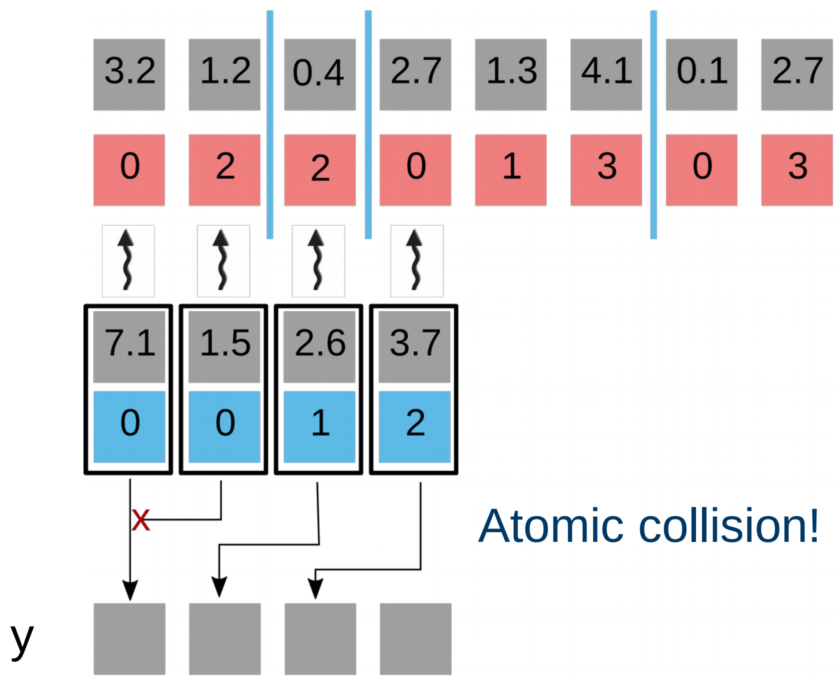
Atomic collision!

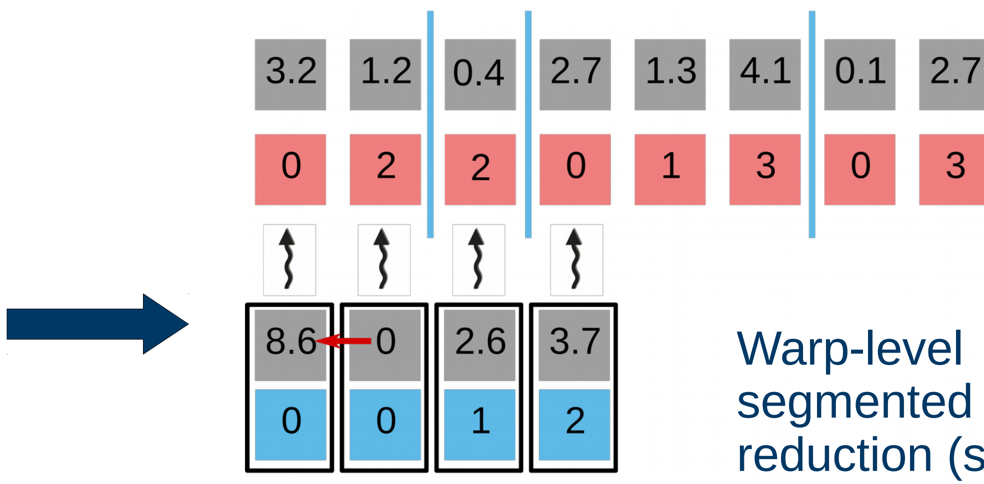
Getting good performance on GPUs

CUDA thread = 1 lane of a 32-wide SIMD unit (warp)

Spreading out threads causes strided memory access.

Assign one warp per chunk.





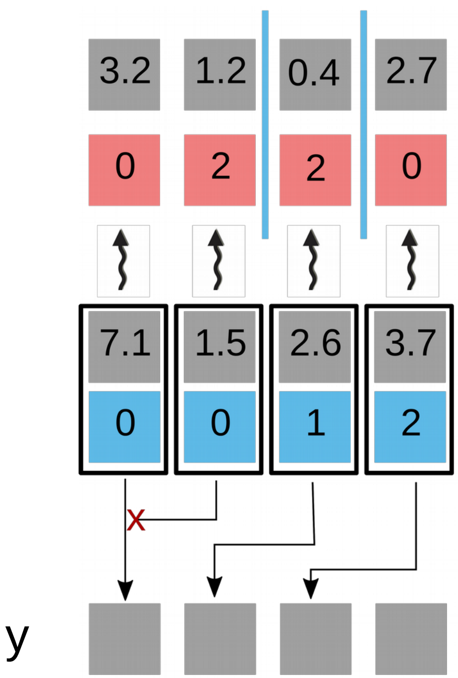
Warp-level segmented reduction (scan) using shuffles

Getting good performance on GPUs

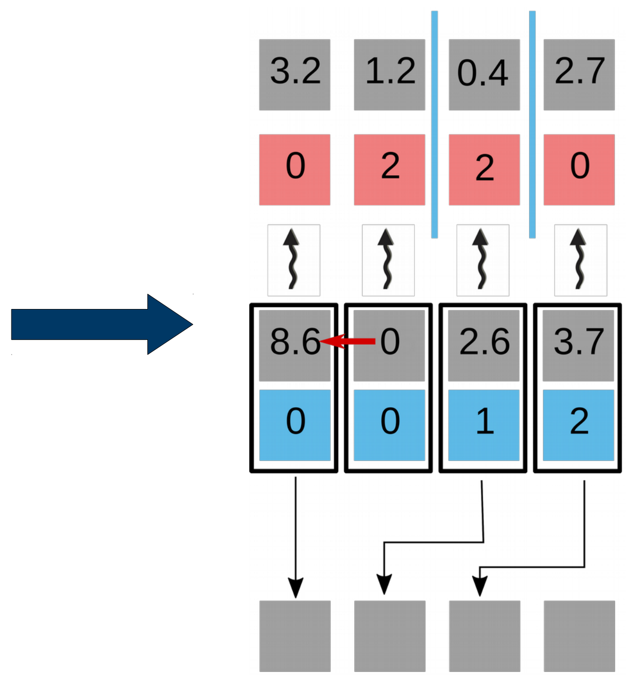
CUDA thread = 1 lane of a 32-wide SIMD unit (warp)

Spreading out threads causes strided memory access.

Assign one warp per chunk.



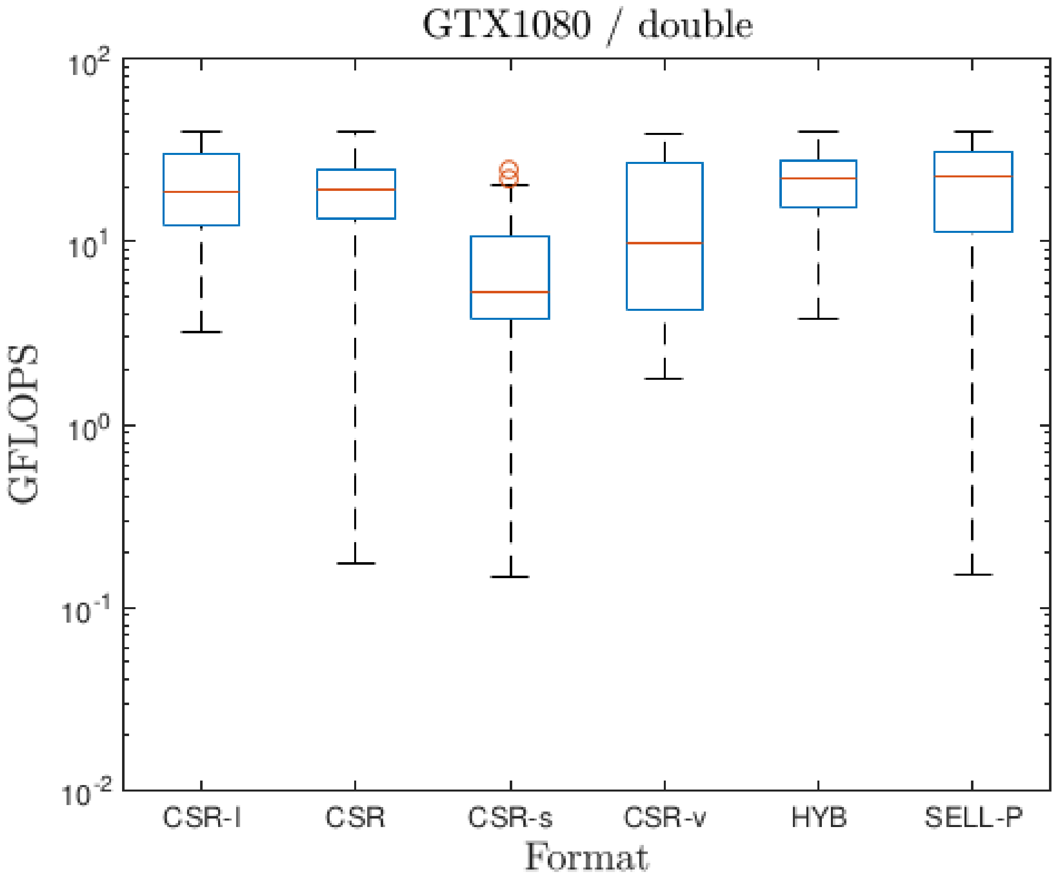
Atomic collision!



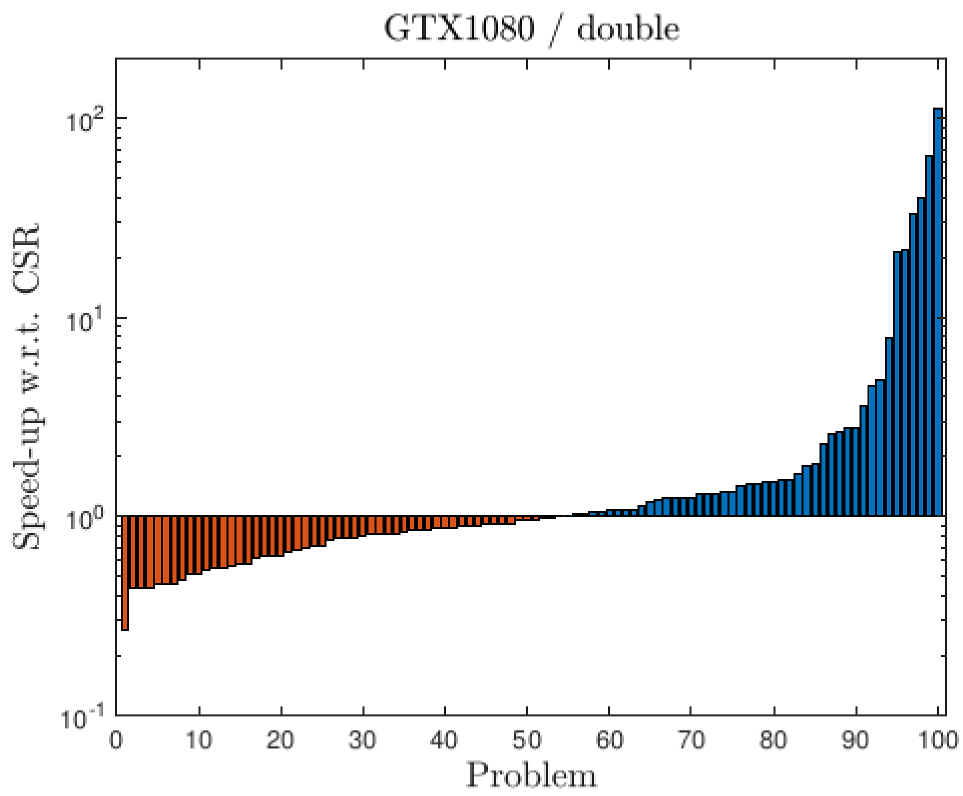
Warp-level segmented reduction (scan) using shuffles

Performance of CSR-I

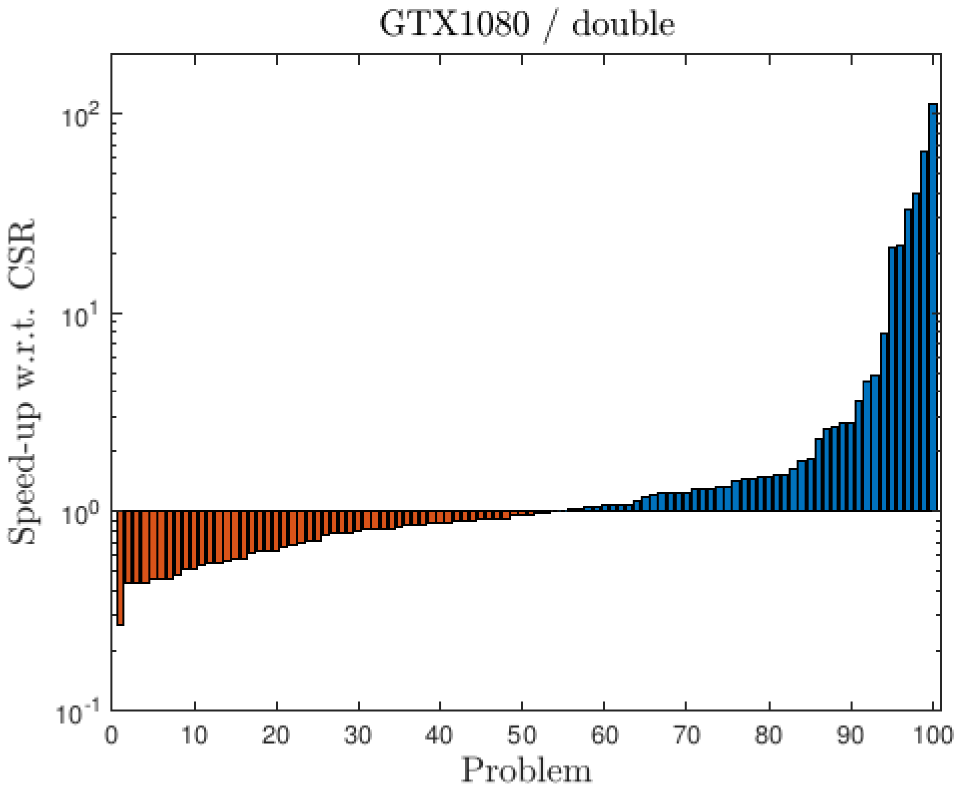
100 matrices from SuiteSparse



Speed-up / slowdown over cuSPARSE CSR



Speed-up / slowdown over cuSPARSE CSR



No format conversion!

- try both, and use the fastest later on!
- sometimes 1 cuSPARSE SpMV = 100 CSR-I SpMVs

Choosing the winner a priori

CSR-I designed for irregular patterns

Choosing the winner a priori

CSR-I designed for irregular patterns

How to measure irregularity?

Deviation of row lengths from the mean.

Choosing the winner a priori

CSR-I designed for irregular patterns

How to measure irregularity?

Deviation of row lengths from the mean.

Is “5” regular or irregular?

Depends on the density of the matrix (mean #rows)

Choosing the winner a priori

CSR-I designed for irregular patterns

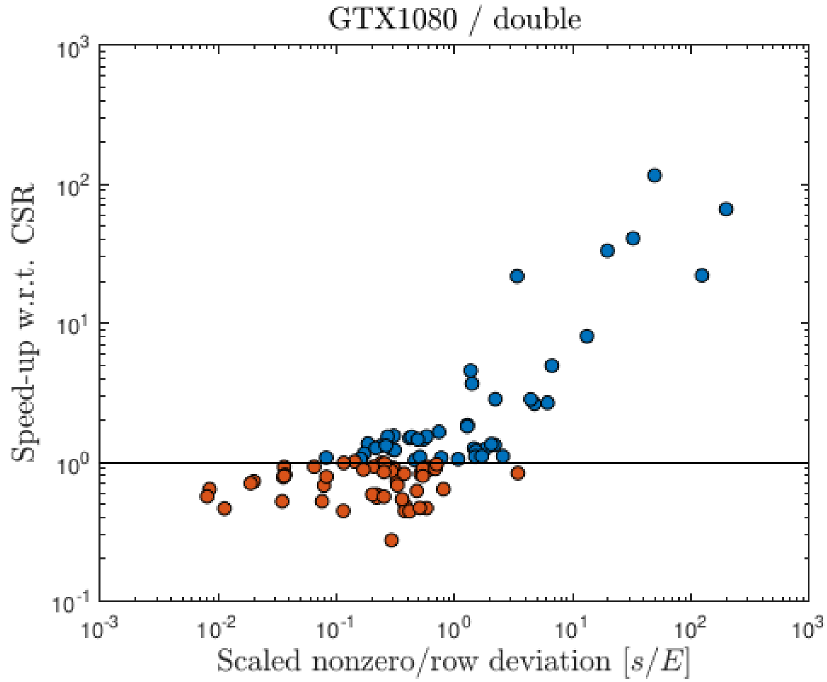
How to measure irregularity?

Deviation of row lengths from the mean.

Is “5” regular or irregular?

Depends on the density of the matrix (mean #rows)

Scatter plot of speedup vs normalized std. dev.



Choosing the winner a priori

CSR-I designed for irregular patterns

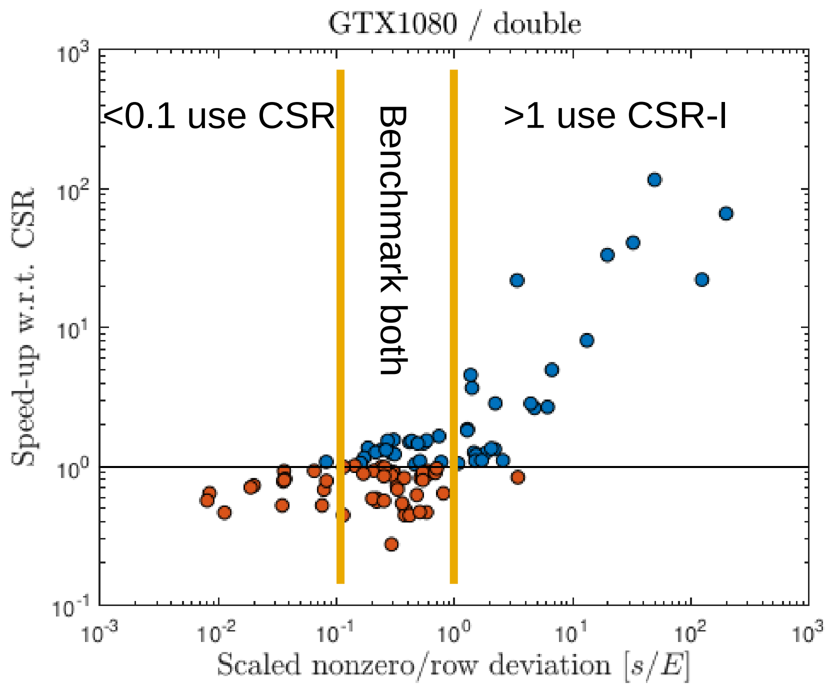
How to measure irregularity?

Deviation of row lengths from the mean.

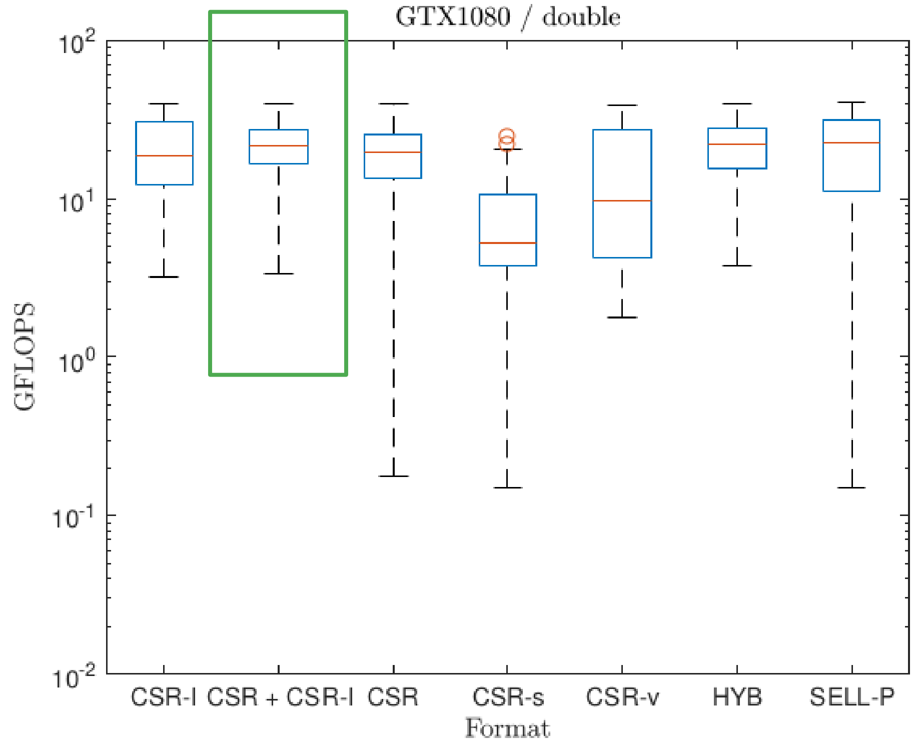
Is “5” regular or irregular?

Depends on the density of the matrix (mean #rows)

Scatter plot of speedup vs normalized std. dev.



Combining both approaches



Conclusion

Use atomics and warp shuffles to tackle irregular sparsity patterns.

Determine a priori when CSR-I is faster than standard algorithm.

- keep cuSPARSE performance for regular patterns
- CSR-I for irregular ones

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/europar_2017

This research is based on a cooperation between Hartwig Anzt, Jack Dongarra (University of Tennessee), Goran Flegar and Enrique S. Quintana-Ortí (Universidad Jaume I).

THE UNIVERSITY OF TENNESSEE KNOXVILLE

UNIVERSITAT JAUME I