CMU, Pittsburgh, 9 November 2018

UNIVERSITAT JAUME I

How to Solve a Linear System

Goran Flegar

Hartwig Anzt, Yen-Chen Chen, Terry Cojean, Jack Dongarra, Nick Higham, Pratik Nayak, Enrique S. Quintana-Ortí, Mike Tsai

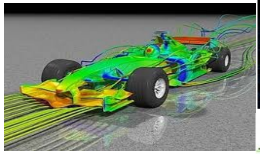
Scan me for slides!

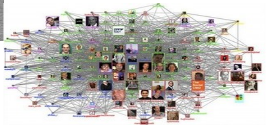
github.com/gflegar/talks/tree/master/cmu_pittsburgh_2018_11

flegar@uji.es

Sources of linear systems

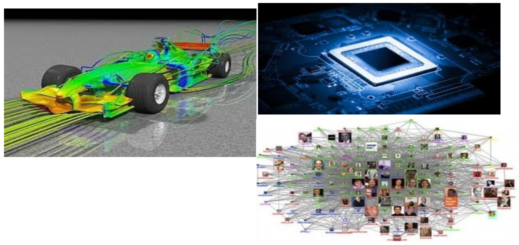
- Real-world problem transformed into a linear system via:
  - PDE discretizations, graph representations
  - Large number of unknowns (1M+, full matrix 8TB)
  - Most matrix elements are zero

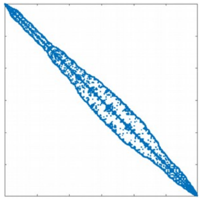




Sources of linear systems

- Real-world problem transformed into a linear system via:
  - PDE discretizations, graph representations
  - Large number of unknowns (1M+, full matrix 8TB)
  - Most matrix elements are zero
- Possible approach: iterative methods
  - 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

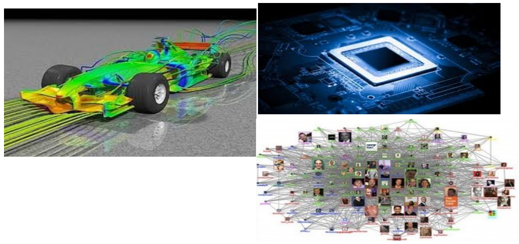


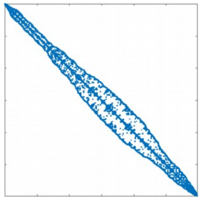


$$Ax = b$$

Sources of linear systems

- Real-world problem transformed into a linear system via:
  - PDE discretizations, graph representations
  - Large number of unknowns (1M+, full matrix 8TB)
  - Most matrix elements are zero
- Possible approach: iterative methods
  - 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
  - Special hardware (e.g. GPUs)
  - Probably not a good idea to implement everything from scratch…

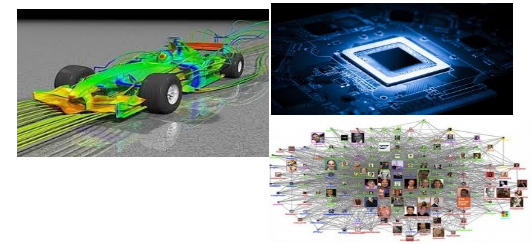


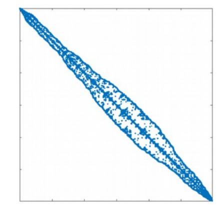


$$Ax = b$$

Sources of linear systems

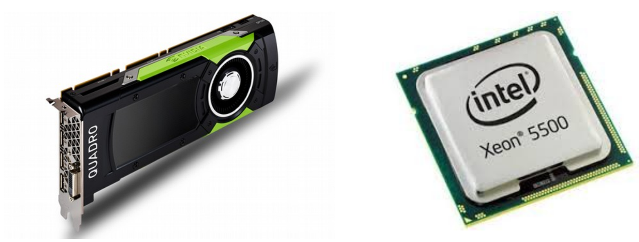
- Real-world problem transformed into a linear system via:
  - PDE discretizations, graph representations
  - Large number of unknowns (1M+, full matrix 8TB)
  - Most matrix elements are zero
- Possible approach: iterative methods
  - Krylov-subspace based linear solvers
    - SpMV
    - BLAS-1 operations
  - Sparse matrix formats & SpMV
    - accelerate each iteration of the solver
  - Preconditioners
    - reduce the number of iterations
  - Special hardware (e.g. GPUs)
  - Probably not a good idea to implement everything from scratch…
  - Use a library instead: Ginkgo





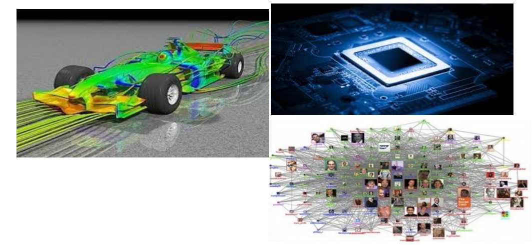
$$Ax = b$$

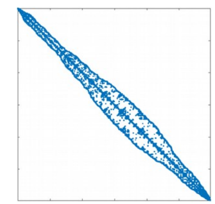
Ginkgo



Sources of linear systems

- Real-world problem transformed into a linear system via:
  - PDE discretizations, graph representations
  - Large number of unknowns (1M+, full matrix 8TB)
  - Most matrix elements are zero
- Possible approach: iterative methods
  - 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
  - Special hardware (e.g. GPUs)
  - Probably not a good idea to implement everything from scratch…
  - Use a library instead: Ginkgo





$$Ax = b$$

Ginkgo

The Ginkgo library

- Linear operator algebra library:
  - Matrix formats, preconditioners, (Krylov) solvers

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

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Karlsruher Institut für Technologie

UNIVERSITAT JAUME I

The Ginkgo library

- Linear operator algebra library:
  - Matrix formats, preconditioners, (Krylov) solvers
- Goals:
  - Backends for various devices (executors)
  - Easy composability – even when using building blocks from a different category (e.g. using a matrix or a solver as a preconditioner)
  - Easy extensibility
  - Support for matrix-free methods

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

Core idea

Matrix-vector product

$$x = Ab$$

Core idea

Matrix-vector product

$$x = Ab$$

Preconditioner

$$x = M^{-1}b$$

$$M \approx A$$

Core idea

Matrix-vector product

$$x = Ab$$

Preconditioner

$$x = M^{-1}b$$

$$M \approx A$$
$$M^{-1} = \Pi(A)$$

Core idea

Matrix-vector product

$$x = Ab$$

Preconditioner

$$x = M^{-1}b$$

$$M \approx A$$
$$M^{-1} = \Pi(A)$$

Solver

$$Ax = b$$

Core idea

Matrix-vector product

$$x = Ab$$

Preconditioner

$$x = M^{-1}b$$

$$M \approx A$$

$$M^{-1} = \Pi(A)$$

Solver

$$Ax = b$$

$$x = Sb$$

$$S = A^{-1}$$

Core idea

Matrix-vector product

$$x = Ab$$

Preconditioner

$$x = M^{-1}b$$

$$M \approx A$$
$$M^{-1} = \Pi(A)$$

Solver

$$Ax = b$$
$$x = Sb$$

$$S = A^{-1}$$
$$S = \Sigma(A)$$

Core idea

Matrix-vector product | Preconditioner | Solver

$$Ax = b$$

$$\boxed{x = Ab}$$

$$\boxed{x = M^{-1}b}$$

$$\boxed{x = Sb}$$

$$M \approx A$$
$$M^{-1} = \Pi(A)$$

$$S = A^{-1}$$
$$S = \Sigma(A)$$

All of them can be expressed as:

- applications of linear operators* (LinOp)

$$L : \mathbb{F}^m \to \mathbb{F}^n$$

* can be realized as a (non-linear) approximation

Core idea

Matrix-vector product	Preconditioner	Solver
		$Ax = b$
$\boxed{x = Ab}$	$\boxed{x = M^{-1}b}$	$\boxed{x = Sb}$
	$M \approx A$	$S = A^{-1}$
	$\boxed{M^{-1} = \Pi(A)}$	$\boxed{S = \Sigma(A)}$

All of them can be expressed as:

- applications of linear operators* (LinOp)
- (non-linear) transformations applied to linear operators (Factory)

$$L : \mathbb{F}^m \to \mathbb{F}^n$$
$$\Phi : \mathbb{L}^{mn}(\mathbb{F}) \to \mathbb{L}^{kl}(\mathbb{F})$$

* can be realized as a (non-linear) approximation

Example

- Solving a symmetric positive definite linear system using the conjugate gradient (CG) solver, preconditioned with a block-Jacobi (BJ) preconditioner, with a system matrix stored in CSR format

Example

- Solving a symmetric positive definite linear system using the conjugate gradient (CG) solver, preconditioned with a block-Jacobi (BJ) preconditioner, with a system matrix stored in CSR format

1. Read the system matrix operator, right hand side and initial guess from file, standard input, or use existing data from the application

$$\mathrm{A}_{\mathrm{CSR}} \;\; \mathrm{b}_{\mathrm{D}} \;\; \mathrm{x}_{\mathrm{D}}$$

Example

- Solving a symmetric positive definite linear system using the conjugate gradient (CG) solver, preconditioned with a block-Jacobi (BJ) preconditioner, with a system matrix stored in CSR format

1. Read the system matrix operator, right hand side and initial guess from file, standard input, or use existing data from the application

   $\mathrm{A}_{\mathrm{CSR}}\ \mathrm{b}_{\mathrm{D}}\ \mathrm{x}_{\mathrm{D}}$

2. Build the block-Jacobi preconditioner factory (no computation, binding parameters)

   BJ

Example

- Solving a symmetric positive definite linear system using the conjugate gradient (CG) solver, preconditioned with a block-Jacobi (BJ) preconditioner, with a system matrix stored in CSR format

1. Read the system matrix operator, right hand side and initial guess from file, standard input, or use existing data from the application

   $\mathrm{A}_{\mathrm{CSR}}\ \mathrm{b}_{\mathrm{D}}\ \mathrm{x}_{\mathrm{D}}$

2. Build the block-Jacobi preconditioner factory (no computation, binding parameters)

   BJ

3. Use the preconditioner factory to build the preconditioned conjugate gradient factory (no computation, binding parameters)

   $\mathrm{CG}_{\mathrm{BJ}}$

Example

- Solving a symmetric positive definite linear system using the conjugate gradient (CG) solver, preconditioned with a block-Jacobi (BJ) preconditioner, with a system matrix stored in CSR format

1. Read the system matrix operator, right hand side and initial guess from file, standard input, or use existing data from the application

   $\mathrm{A}_{\mathrm{CSR}}\ \mathrm{b}_{\mathrm{D}}\ \ \mathrm{x}_{\mathrm{D}}$

2. Build the block-Jacobi preconditioner factory (no computation, binding parameters)

   BJ

3. Use the preconditioner factory to build the preconditioned conjugate gradient factory (no computation, binding parameters)

   $\mathrm{CG}_{\mathrm{BJ}}$

4. Generate the solver operator from the system matrix using the solver factory (preprocessing needed for the solver and preconditioner)

   $\mathrm{S}_{\mathrm{A}} = \mathrm{CG}_{\mathrm{BJ}}(\mathrm{A}_{\mathrm{CSR}})$

Example

- Solving a symmetric positive definite linear system using the conjugate gradient (CG) solver, preconditioned with a block-Jacobi (BJ) preconditioner, with a system matrix stored in CSR format

1. Read the system matrix operator, right hand side and initial guess from file, standard input, or use existing data from the application

$$\mathrm{A}_{\mathrm{CSR}} \quad \mathrm{b}_{\mathrm{D}} \quad \mathrm{x}_{\mathrm{D}}$$

2. Build the block-Jacobi preconditioner factory (no computation, binding parameters)

$$\mathrm{BJ}$$

3. Use the preconditioner factory to build the preconditioned conjugate gradient factory (no computation, binding parameters)

$$\mathrm{CG}_{\mathrm{BJ}}$$

4. Generate the solver operator from the system matrix using the solver factory (preprocessing needed for the solver and preconditioner)

$$\mathrm{S}_{\mathrm{A}} = \mathrm{CG}_{\mathrm{BJ}}(\mathrm{A}_{\mathrm{CSR}})$$

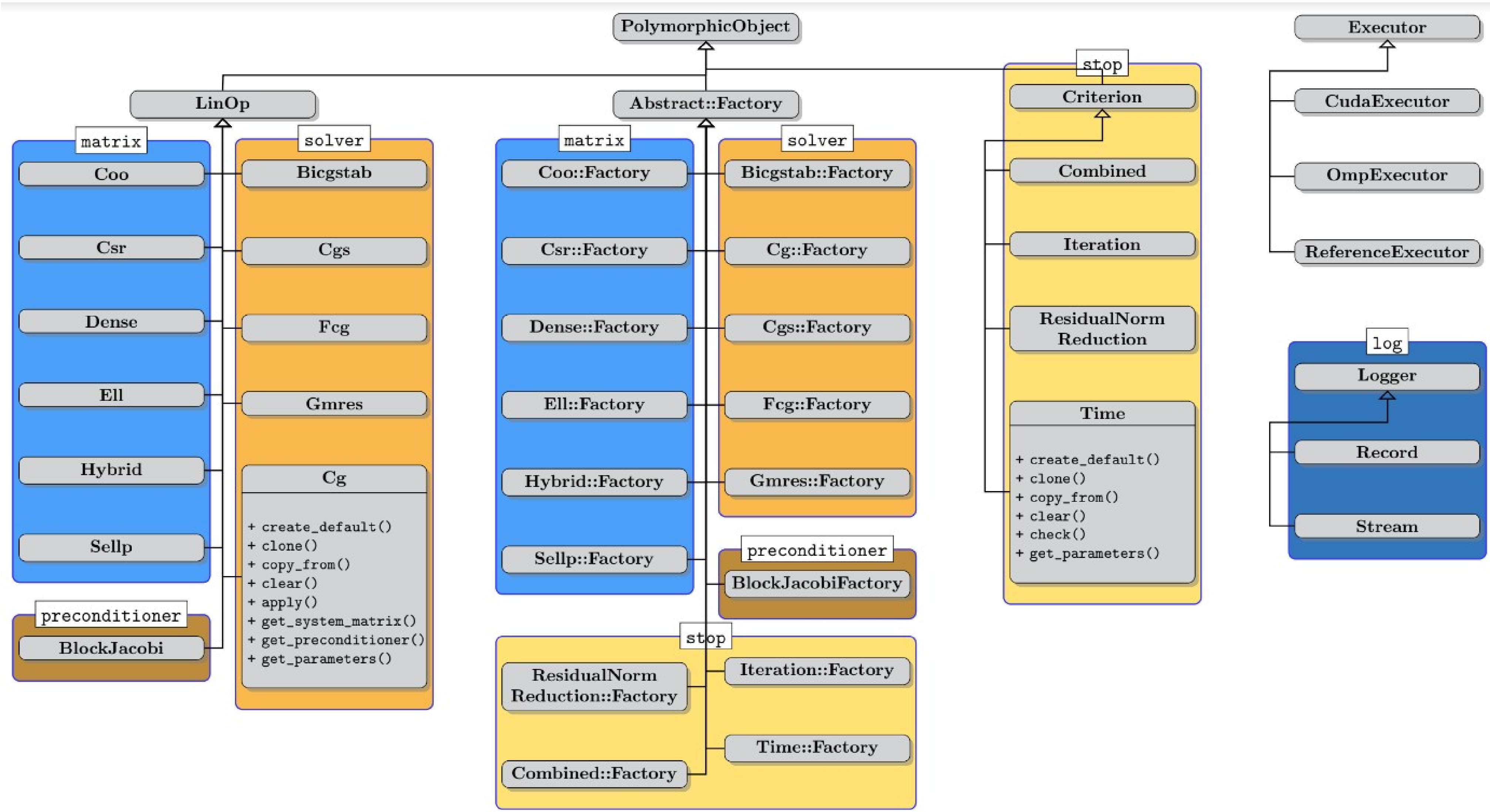
5. Solve the system (actual computation)

$$\mathrm{x}_{\mathrm{D}} = \mathrm{S}_{\mathrm{A}} \mathrm{b}_{\mathrm{D}}$$

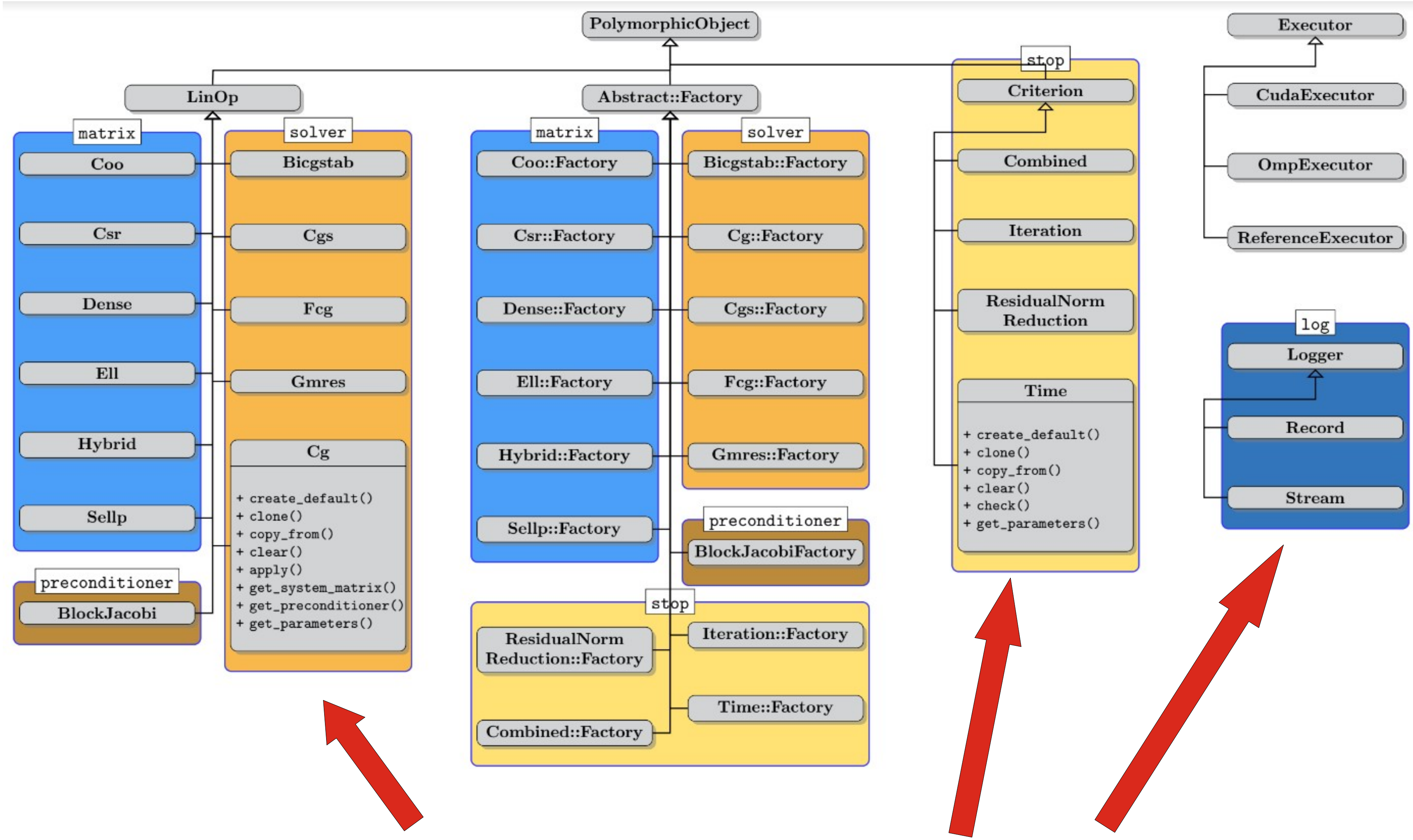
Example source code

```cpp
int main() {
  // Instantiate a CUDA executor
  auto gpu = gko::CudaExecutor::create(0, gko::OmpExecutor::create());
  // Read data
  auto A = gko::read<gko::matrix::Csr<>>(std::cin, gpu);
  auto b = gko::read<gko::matrix::Dense<>>(std::cin, gpu);
  auto x = gko::read<gko::matrix::Dense<>>(std::cin, gpu);
  // Create the solver
  auto solver = gko::solver::Cg<>::build()
    .with_preconditioner(
      gko::preconditioner::Jacobi<>::build().with_max_block_size(32).on(gpu))
    .with_criteria(
        gko::stop::Iteration::build().with_max_iters(20u).on(gpu),
        gko::stop::ResidualNormReduction<>::build()
          .with_reduction_factor(1e-15).on(gpu))
    .on(gpu);
  // Solve system
  solver->generate(give(A))->apply(lend(b), lend(x));
  // Write result
  write(std::cout, lend(x));
}
```

Library features



Library features: extensibility

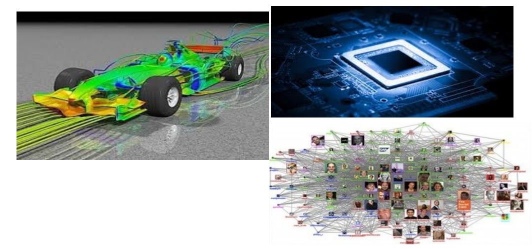


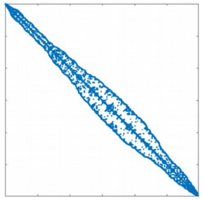
users can provide new matrices, solvers, preconditioners, stopping criteria, loggers

Without recompiling the library!

Sources of linear systems

- Real-world problem transformed into a linear system via:
  - PDE discretizations, graph representations
  - Large number of unknowns (1M+, full matrix 8TB)
  - Most matrix elements are zero
- Possible approach: iterative methods
  - 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
  - Special hardware (e.g. GPUs)
  - Probably not a good idea to implement everything from scratch…
  - Use a library instead: Ginkgo





$$Ax = b$$

Ginkgo

Preconditioning

$$Ax = b, \;\; A \in \mathbb{R}^{n \times n}$$

Preconditioning

$$Ax = b,\ A \in \mathbb{R}^{n \times n} \longrightarrow M^{-1}Ax = M^{-1}b$$

Replace the original system with an
equivalent preconditioned system

Preconditioning

$$Ax = b,\ A \in \mathbb{R}^{n \times n} \longrightarrow M^{-1}Ax = M^{-1}b$$

Replace the original system with an equivalent preconditioned system

$$M \approx A \qquad M^{-1} \text{ easy to compute}$$

Preconditioning

$$Ax = b,\ A \in \mathbb{R}^{n \times n} \quad \Rightarrow \quad M^{-1}Ax = M^{-1}b$$

Replace the original system with an equivalent preconditioned system

$$M \approx A \qquad M^{-1} \text{ easy to compute}$$

$$\cancel{M^{-1}A}$$

Do not compute the preconditioned system matrix explicitly!

Preconditioning

$$Ax = b, \; A \in \mathbb{R}^{n \times n} \quad \longrightarrow \quad M^{-1}Ax = M^{-1}b$$

Replace the original system with an equivalent preconditioned system

$$M \approx A \qquad M^{-1} \text{ easy to compute}$$

$$\cancel{M^{-1}A}$$

Do not compute the preconditioned system matrix explicitly!

$$y := (M^{-1}A)x$$

Preconditioning

$$Ax = b,\ A \in \mathbb{R}^{n\times n} \longrightarrow M^{-1}Ax = M^{-1}b$$

Replace the original system with an equivalent preconditioned system

$$M \approx A \qquad M^{-1} \text{ easy to compute}$$

$$\cancel{M^{-1}A}$$

Do not compute the preconditioned system matrix explicitly!

$$y := (M^{-1}A)x$$

$$\Downarrow$$

$$z := Ax$$

$$\boxed{y := M^{-1}z}$$

Preconditioner application

Preconditioning

$$Ax = b, \ A \in \mathbb{R}^{n \times n} \quad \longrightarrow \quad M^{-1}Ax = M^{-1}b$$

Replace the original system with an
equivalent preconditioned system

$$M \approx A \qquad M^{-1} \text{ easy to compute}$$

$$\cancel{M^{-1}A}$$

Do not compute the preconditioned system matrix explicitly!

$$y := (M^{-1}A)x$$

$$\downarrow$$

$$z := Ax$$

$$\boxed{y := M^{-1}z}$$

Generate the preconditioner matrix, and
store it in a form suitable for application

$$\boxed{A \rightsquigarrow M^{-1}} \longrightarrow \boxed{y := M^{-1}z}$$

Preconditioner setup

Preconditioner application

Preconditioning

$$Ax = b, \; A \in \mathbb{R}^{n \times n} \quad \longrightarrow \quad M^{-1}Ax = M^{-1}b$$

Replace the original system with an
equivalent preconditioned system

$$M \approx A \qquad M^{-1} \text{ easy to compute}$$

$$\cancel{M^{-1}A}$$

Do not compute the preconditioned system matrix explicitly!

$$y := (M^{-1}A)x$$

$$\downarrow$$

$$z := Ax$$

Generate the preconditioner matrix, and store it in a form suitable for application

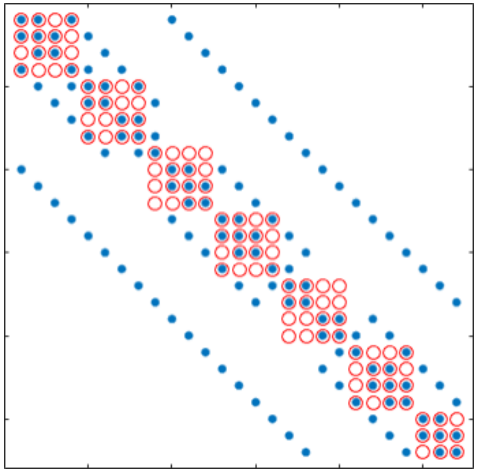
$$\boxed{A \rightsquigarrow M^{-1}} \quad \longrightarrow \quad \boxed{y := M^{-1}z}$$

Generation via factory

Linear operator application

Ginkgo linear operator abstraction

Example: Block-Jacobi preconditioning



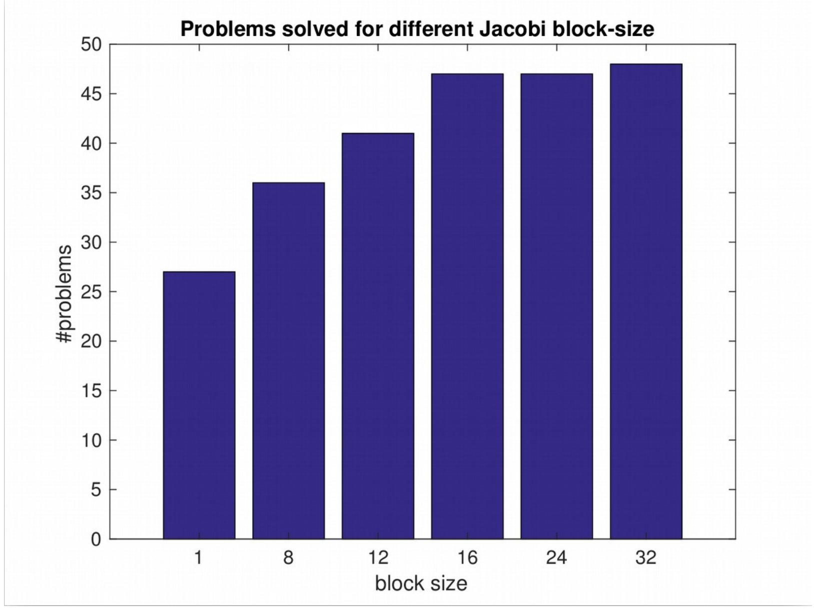
- Block-Jacobi preconditioning
  - Use only diagonal blocks for approximation

$$\operatorname{diag}(A) = [D_1, \ldots, D_k]$$
$$M := \operatorname{diag}(D_1, \ldots, D_k)$$

Anzt, Dongarra, Flegar, Quintana-Ortí, *Variable-size batched Gauss–Jordan elimination for block-Jacobi preconditioning on graphics processors, ParCo*

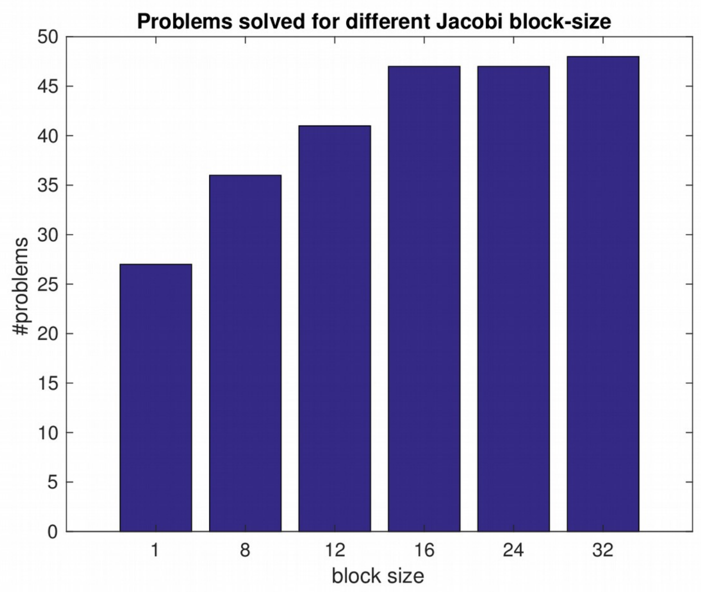
Benefits of block-Jacobi

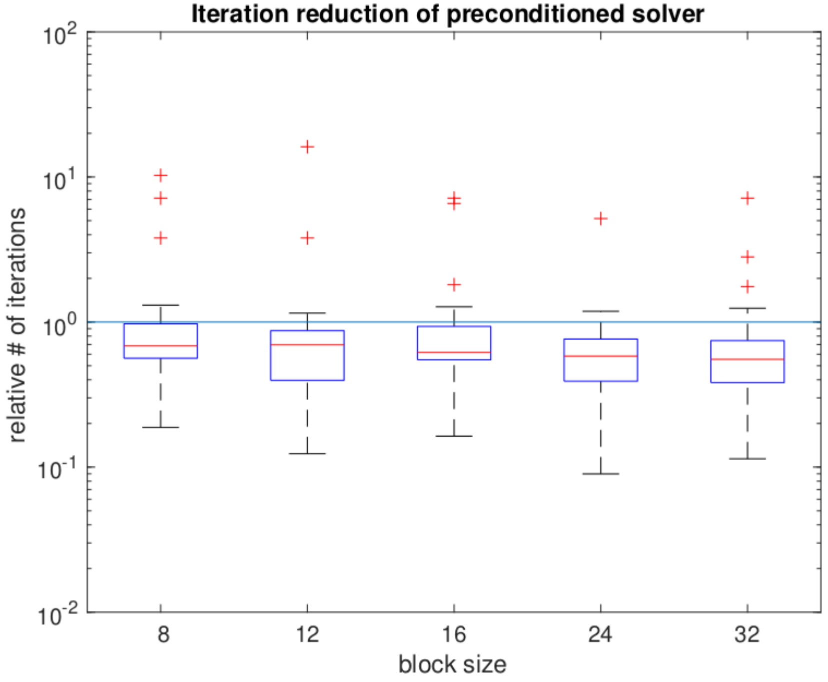
- 56 matrices from SuiteSparse with inherent block structure
- MAGMA-sparse open source library
  - IDR solver
  - Supervariable agglomeration
    - Detects block structure of the matrix
- Improves the robustness of the solver



Benefits of block-Jacobi

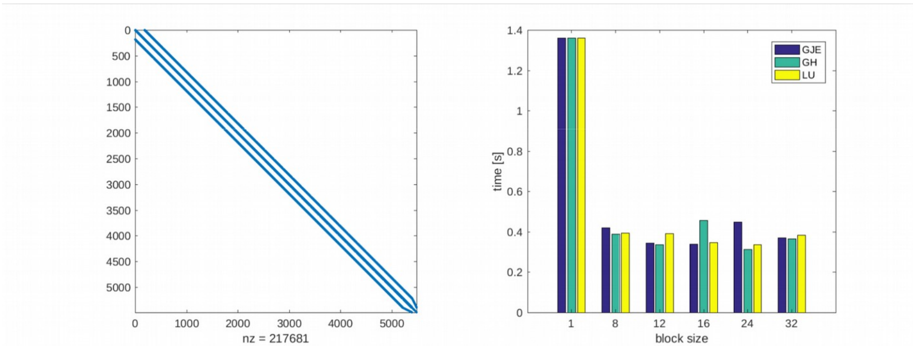
- 56 matrices from SuiteSparse with inherent block structure
- MAGMA-sparse open source library
  - IDR solver
  - Supervariable agglomeration
    - Detects block structure of the matrix
- Improves the robustness of the solver
- Improves convergence of the solver



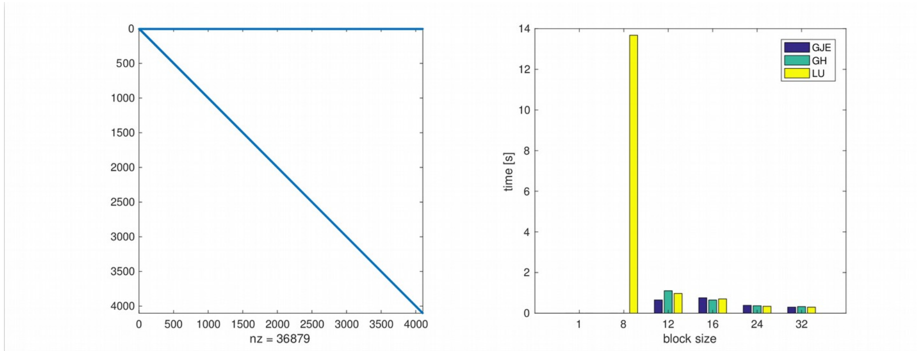


Complete solver runtime

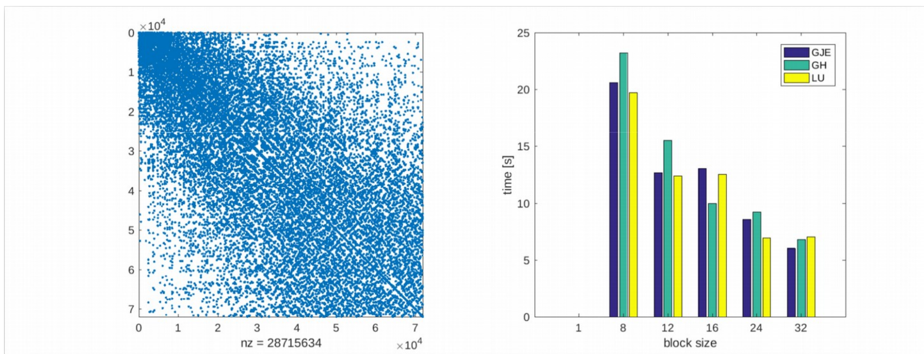
s2rmt3m1



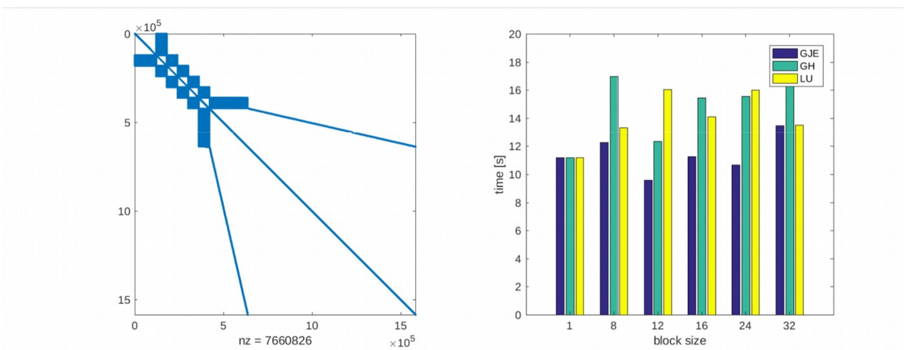
Chebyshev3



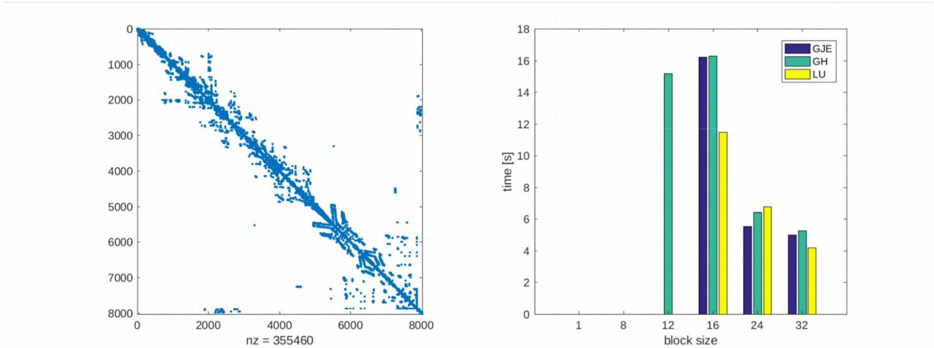
nd24k



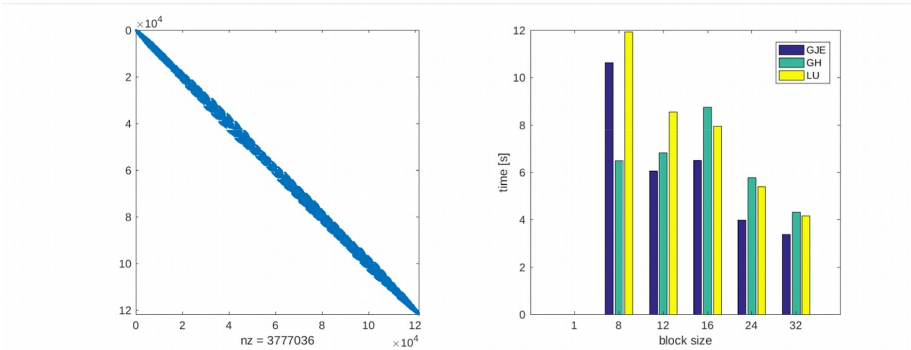
G3_circuit



bcsstk38



ship_003



Current Research: Adaptive precision block-Jacobi

Preconditioner is an approximation of the system matrix

- Applying a preconditioner inherently carries an error
- For block-Jacobi the relative error of z is usually around 0.01 – 0.1

$$z := M^{-1}y \approx A^{-1}y$$

Current Research: Adaptive precision block-Jacobi

Preconditioner is an approximation of the system matrix

- Applying a preconditioner inherently carries an error
- For block-Jacobi the relative error of z is usually around 0.01 – 0.1

$$z := M^{-1}y \approx A^{-1}y$$

Preconditioner application is memory bounded

- Most of the cost comes from reading the matrix from memory
- Idea: use lower precision to store the matrix

Current Research: Adaptive precision block-Jacobi

Preconditioner is an approximation of the system matrix

- Applying a preconditioner inherently carries an error
- For block-Jacobi the relative error of z is usually around 0.01 – 0.1

$$z := M^{-1}y \approx A^{-1}y$$

Preconditioner application is memory bounded

- Most of the cost comes from reading the matrix from memory
- Idea: use lower precision to store the matrix

Adaptive precision in inversion-based block-Jacobi:

- All computation is done in double precision
- Preconditioner matrix is stored in lower precision, with roundoff error “u”
- Error bound:

$$\frac{||\delta z_i||}{||z_i||} \lesssim (c_i \kappa(D_i) u_d + u)\kappa(D_i)$$

Current Research: Adaptive precision block-Jacobi

$$z := M^{-1}y \approx A^{-1}y$$

Preconditioner is an approximation of the system matrix

- Applying a preconditioner inherently carries an error
- For block-Jacobi the relative error of z is usually around 0.01 – 0.1

Preconditioner application is memory bounded

- Most of the cost comes from reading the matrix from memory
- Idea: use lower precision to store the matrix

Adaptive precision in inversion-based block-Jacobi:

- All computation is done in double precision
- Preconditioner matrix is stored in lower precision, with roundoff error “u”
- Error bound:

$$\frac{||\delta z_i||}{||z_i||} \lesssim (c_i \kappa(D_i) u_d + u)\kappa(D_i)$$

Assuming the preconditioner block is relatively well conditioned

- The error is determined by the product of u, and the condition number
- Choose the precision for each block **independently,** such that at least 1 digit of the result is correct

Experimental results

Determining the precision:

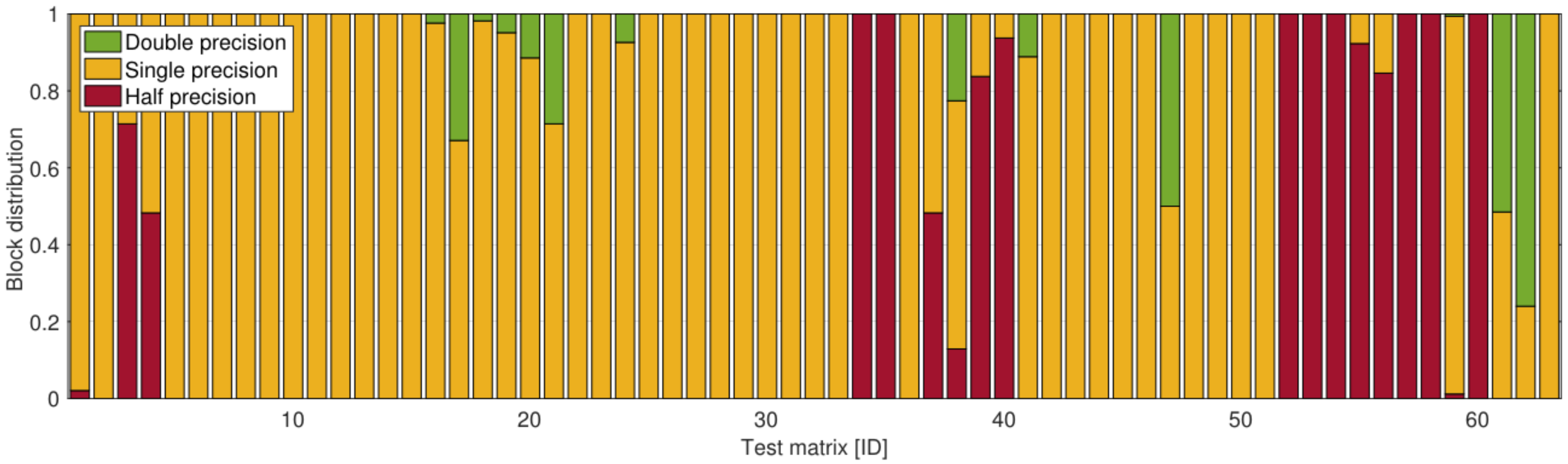
$$\begin{cases} \text{fp16} & \text{if } 0 < \kappa_1(D_i) \leq 10^2, \\ \text{fp32} & \text{if } 10^2 < \kappa_1(D_i) \leq 10^6, \text{ and} \\ \text{fp64} & \text{otherwise,} \end{cases}$$

Experimental results

Determining the precision:

$$\begin{cases} \text{fp16} & \text{if } 0 < \kappa_1(D_i) \leq 10^2, \\ \text{fp32} & \text{if } 10^2 < \kappa_1(D_i) \leq 10^6, \text{ and} \\ \text{fp64} & \text{otherwise,} \end{cases}$$

% of diagonal blocks stored in each precision *:

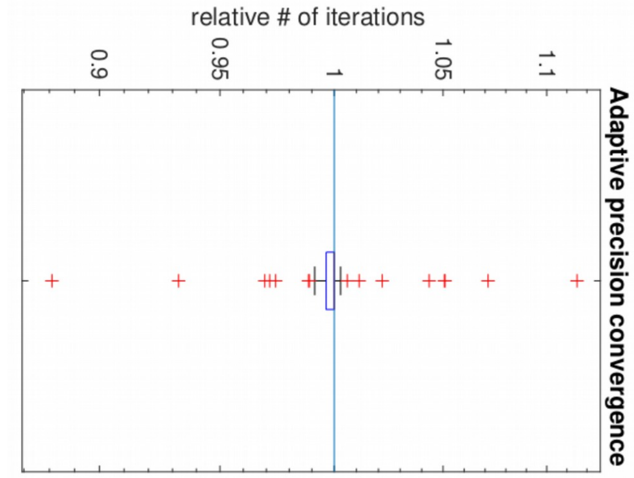


* Prototype implementation in MATLAB,
Results on 63 matrices from SuiteSparse

Experimental results

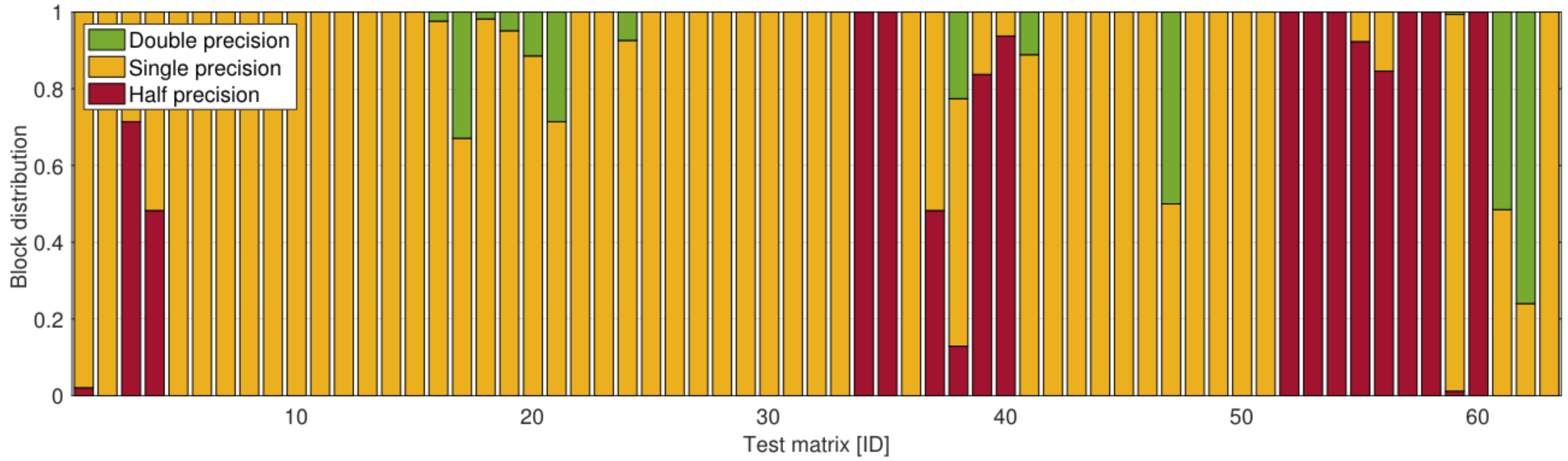
Determining the precision:

$$\begin{cases} \text{fp16} & \text{if } 0 < \kappa_1(D_i) \leq 10^2, \\ \text{fp32} & \text{if } 10^2 < \kappa_1(D_i) \leq 10^6, \text{ and} \\ \text{fp64} & \text{otherwise,} \end{cases}$$



Usually < 5% iteration increase

% of diagonal blocks stored in each precision *:



* Prototype implementation in MATLAB,
Results on 63 matrices from SuiteSparse

Energy efficiency of adaptive precision block-Jacobi

Memory access is an order of magnitude more expensive than computation! [Shalf 2013]

Energy efficiency of adaptive precision block-Jacobi

Memory access is an order of magnitude more expensive than computation! [Shalf 2013]

Energy model assumptions:

- Accessing 1 bit of data has a cost of 1 (energy unit)
- Disregard energy cost of arithmetic operations
- Total energy cost of each iteration:

$$\underbrace{14n \cdot \mathrm{fp64}}_{\text{vector memory transfers}} + \underbrace{(2n + n_z) \cdot \mathrm{fp64} + (n + n_z) \cdot \mathrm{int32}}_{\text{CSR}-\text{SpMV memory transfers}} + \underbrace{2n \cdot \mathrm{fp64} + \sum_{i=1}^{N} m_i^2 \cdot \mathrm{fpxx}_i}_{\text{preconditioner memory transfers}}$$

Energy efficiency of adaptive precision block-Jacobi

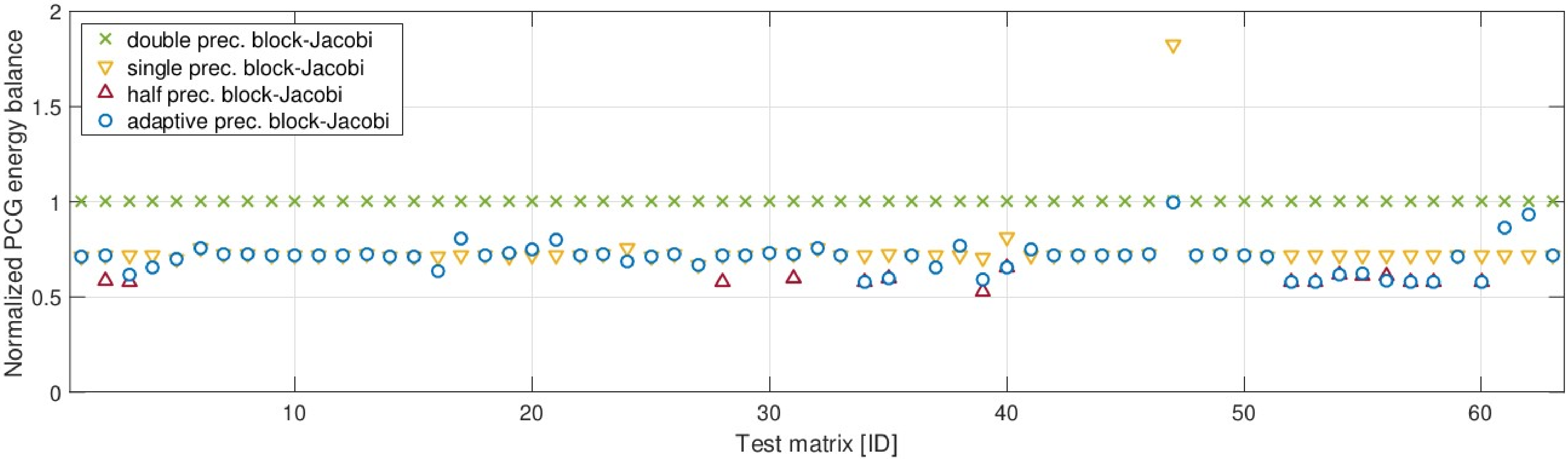
Memory access is an order of magnitude more expensive than computation! [Shalf 2013]

Energy model assumptions:

- Accessing 1 bit of data has a cost of 1 (energy unit)
- Disregard energy cost of arithmetic operations
- Total energy cost of each iteration:

$$\underbrace{14n \cdot \mathrm{fp64}}_{\text{vector memory transfers}} + \underbrace{(2n+n_z)\cdot \mathrm{fp64} + (n+n_z)\cdot \mathrm{int32}}_{\text{CSR-SpMV memory transfers}} + \underbrace{2n\cdot \mathrm{fp64} + \sum_{i=1}^{N} m_i^2 \cdot \mathrm{fpxx}_i}_{\text{preconditioner memory transfers}}$$

Predicted energy savings of adaptive precision block-Jacobi:



Overflow and underflow

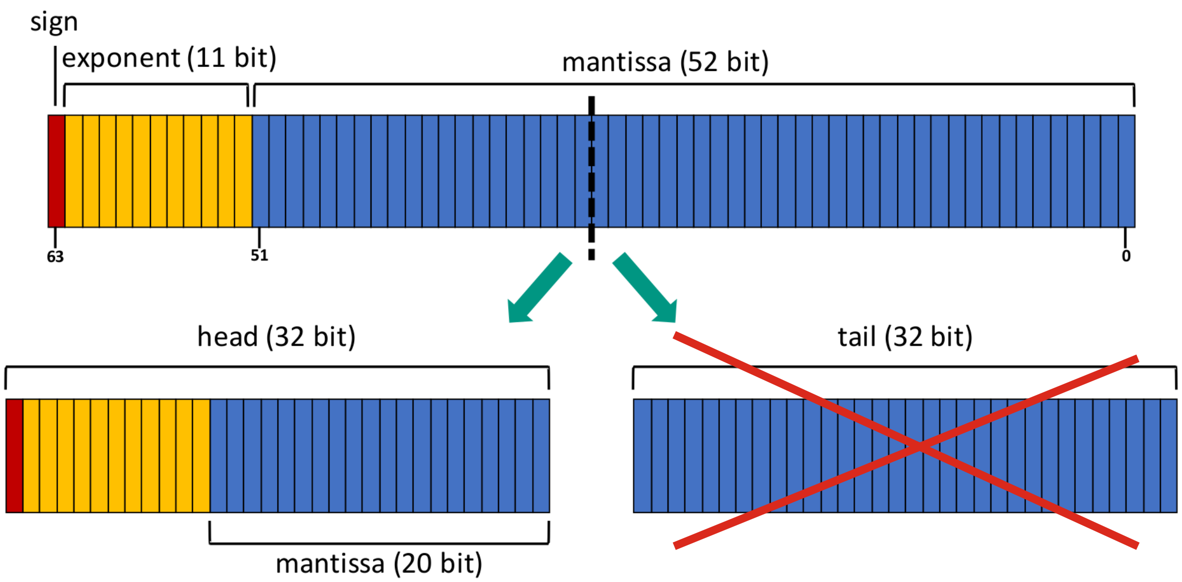
- Storing the block in lower precision can cause underflow or overflow in some values.
- This is not accounted for by the numerical analysis shown in the previous slides

Overflow and underflow

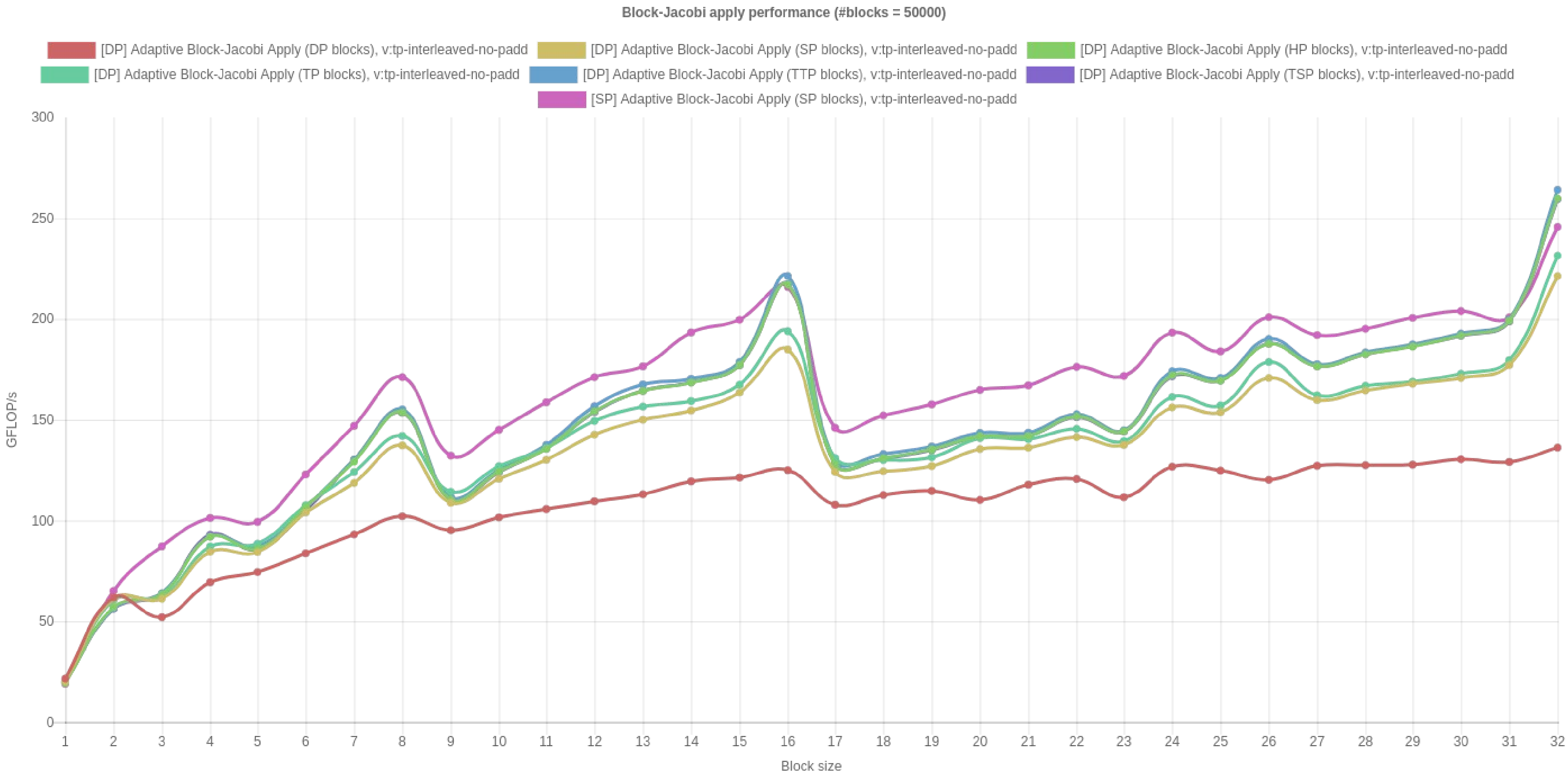
- Storing the block in lower precision can cause underflow or overflow in some values.
- This is not accounted for by the numerical analysis shown in the previous slides
- Two possible solutions:
  1. Check the condition number of the low precision block to verify there were no catastrophic overflows or underflows.

Overflow and underflow

- Storing the block in lower precision can cause underflow or overflow in some values.
- This is not accounted for by the numerical analysis shown in the previous slides
- Two possible solutions:
  1. Check the condition number of the low precision block to verify there were no catastrophic overflows or underflows.
  2. Use a custom storage format that preserves the number of exponent bits:

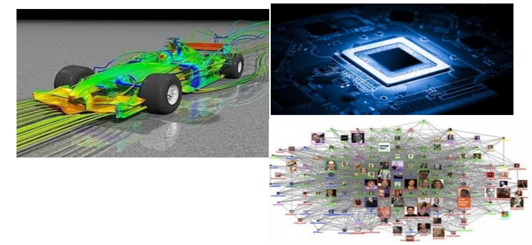


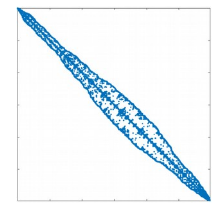
Initial GPU implementation



Sources of linear systems

- Real-world problem transformed into a linear system via:
  - PDE discretizations, graph representations
  - Large number of unknowns (1M+, full matrix 8TB)
  - Most matrix elements are zero
- Possible approach: iterative methods
  - 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
  - Special hardware (e.g. GPUs)
  - Probably not a good idea to implement everything from scratch…
  - Use a library instead: Ginkgo





$$Ax = b$$

Ginkgo

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

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	0	1	2	2	2	3	3	row
0	2	2	0	1	3	0	3	column

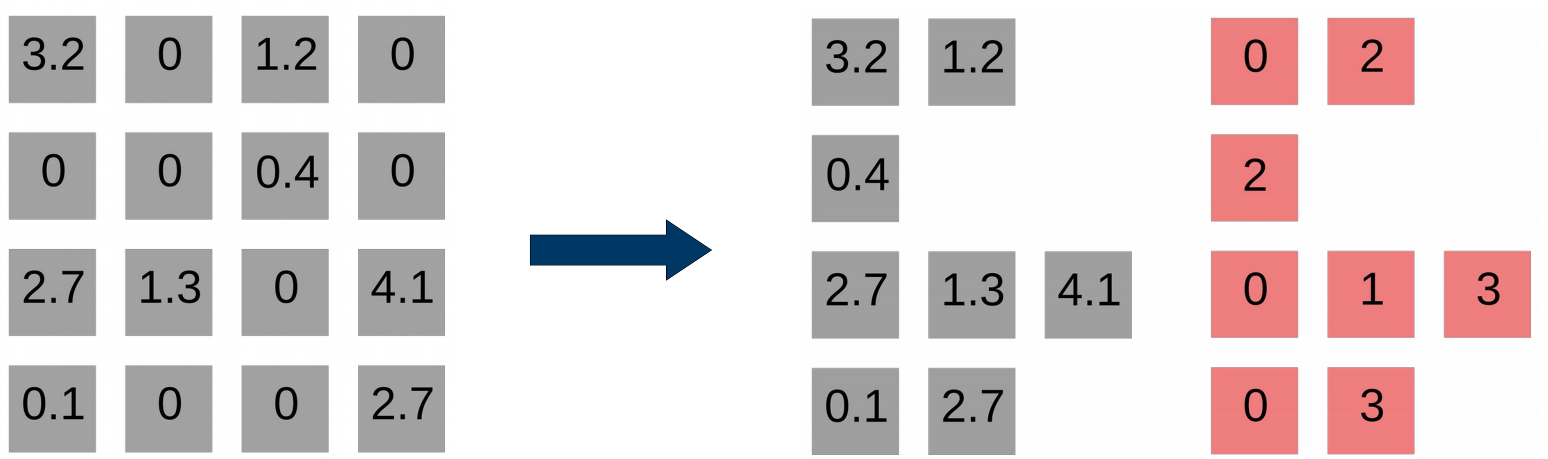
COO

Sparse matrix formats

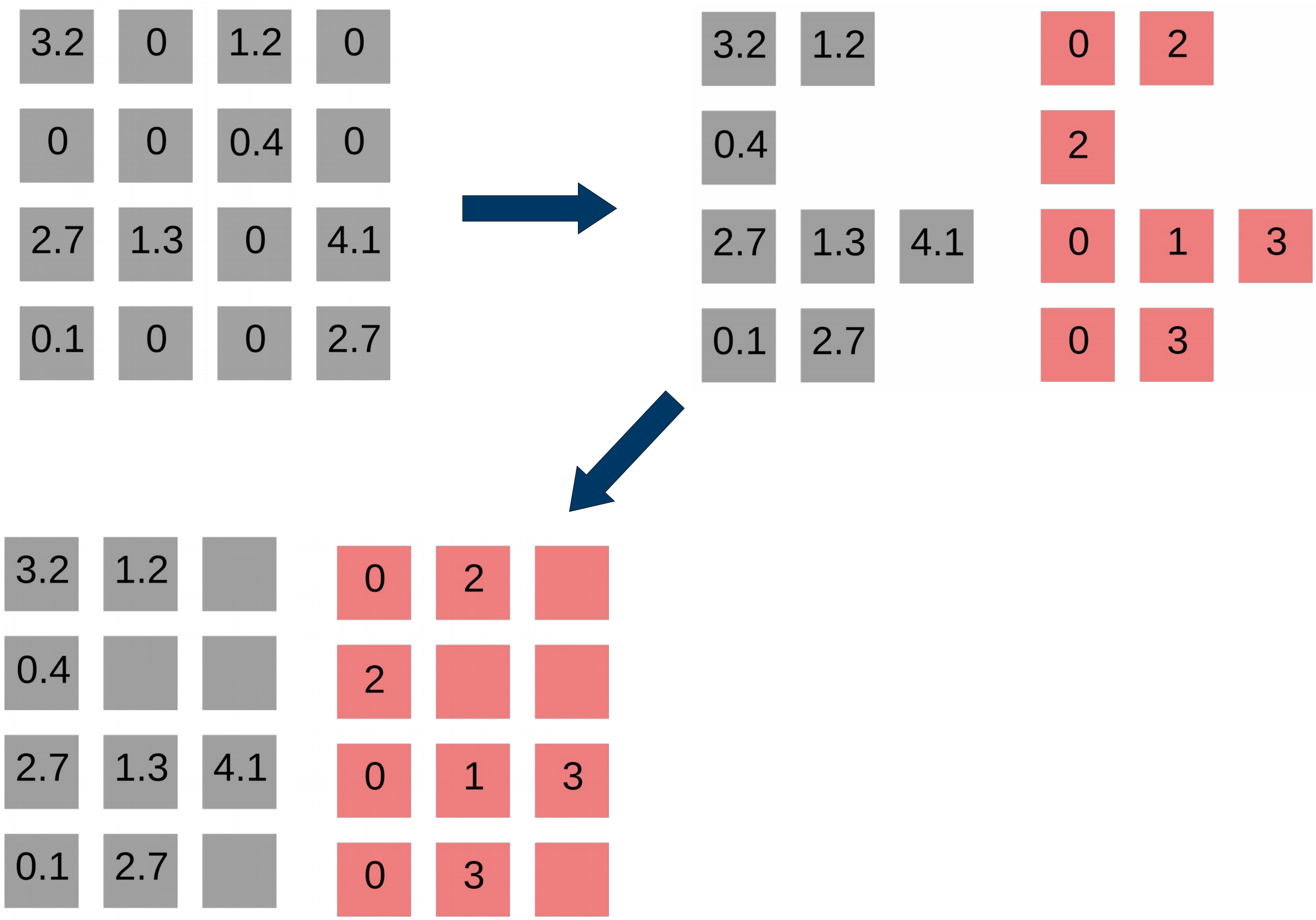
3.2	0	1.2	0
0	0	0.4	0
2.7	1.3	0	4.1
0.1	0	0	2.7

3.2	1.2	
0.4		
2.7	1.3	4.1
0.1	2.7	

Sparse matrix formats

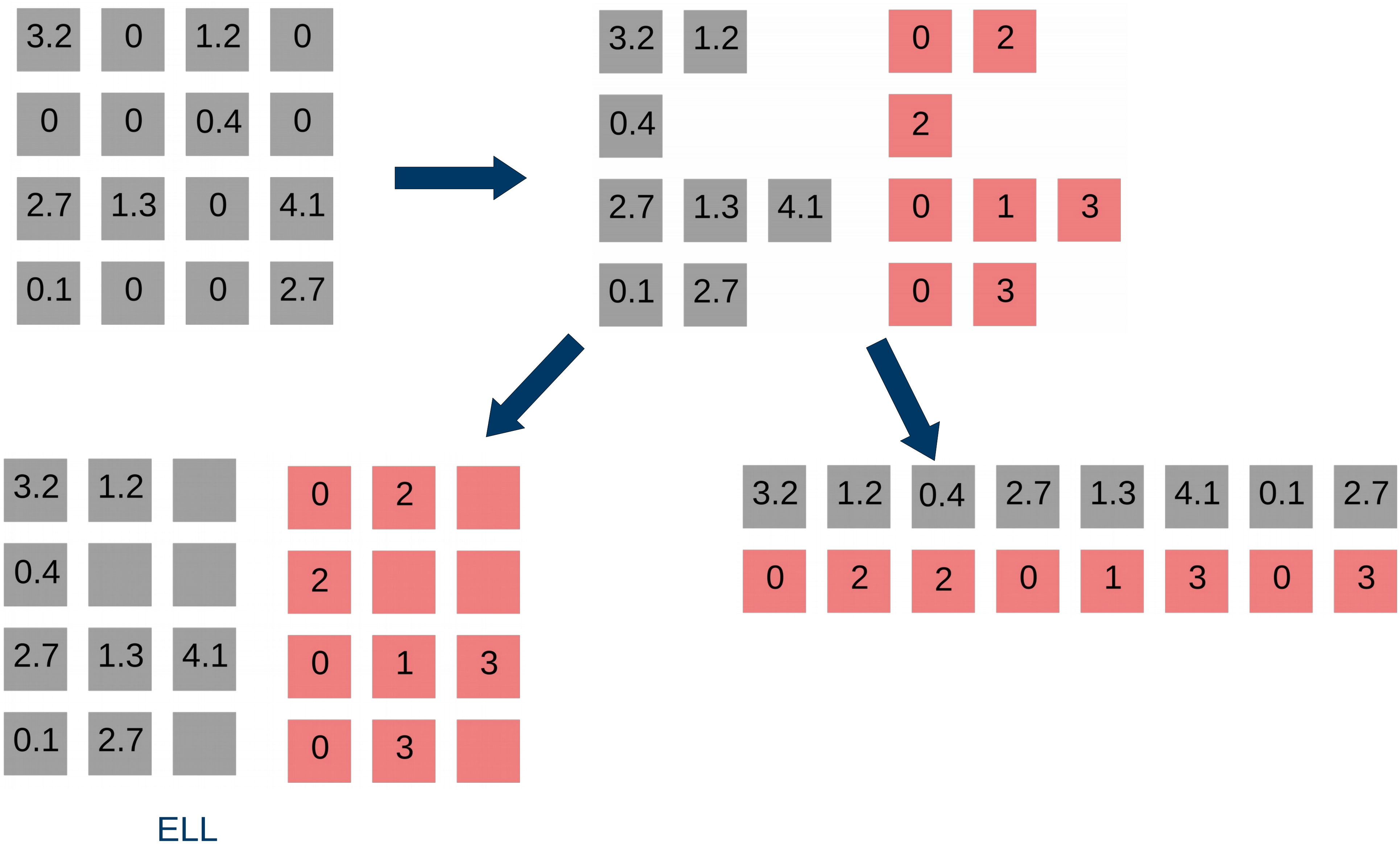


Sparse matrix formats

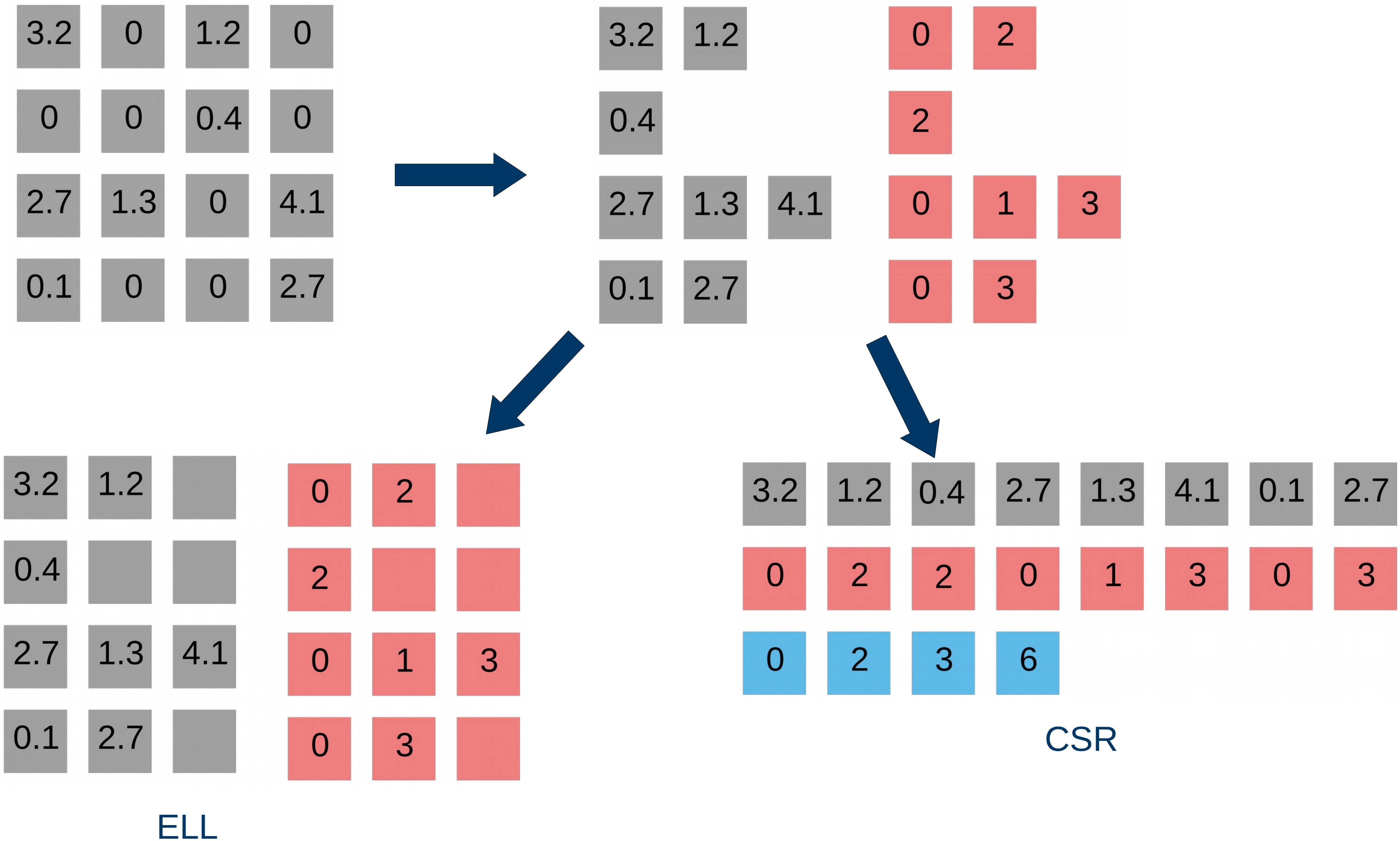


ELL

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	1.3	1.2	4.1
2.7		0.4	2.7
0.1			

0	2	0	2
2		1	3
3			

… leads to CSC

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	1.3	1.2	4.1
2.7		0.4	2.7
0.1			

0	2	0	2
2		1	3
3			

… leads to CSC

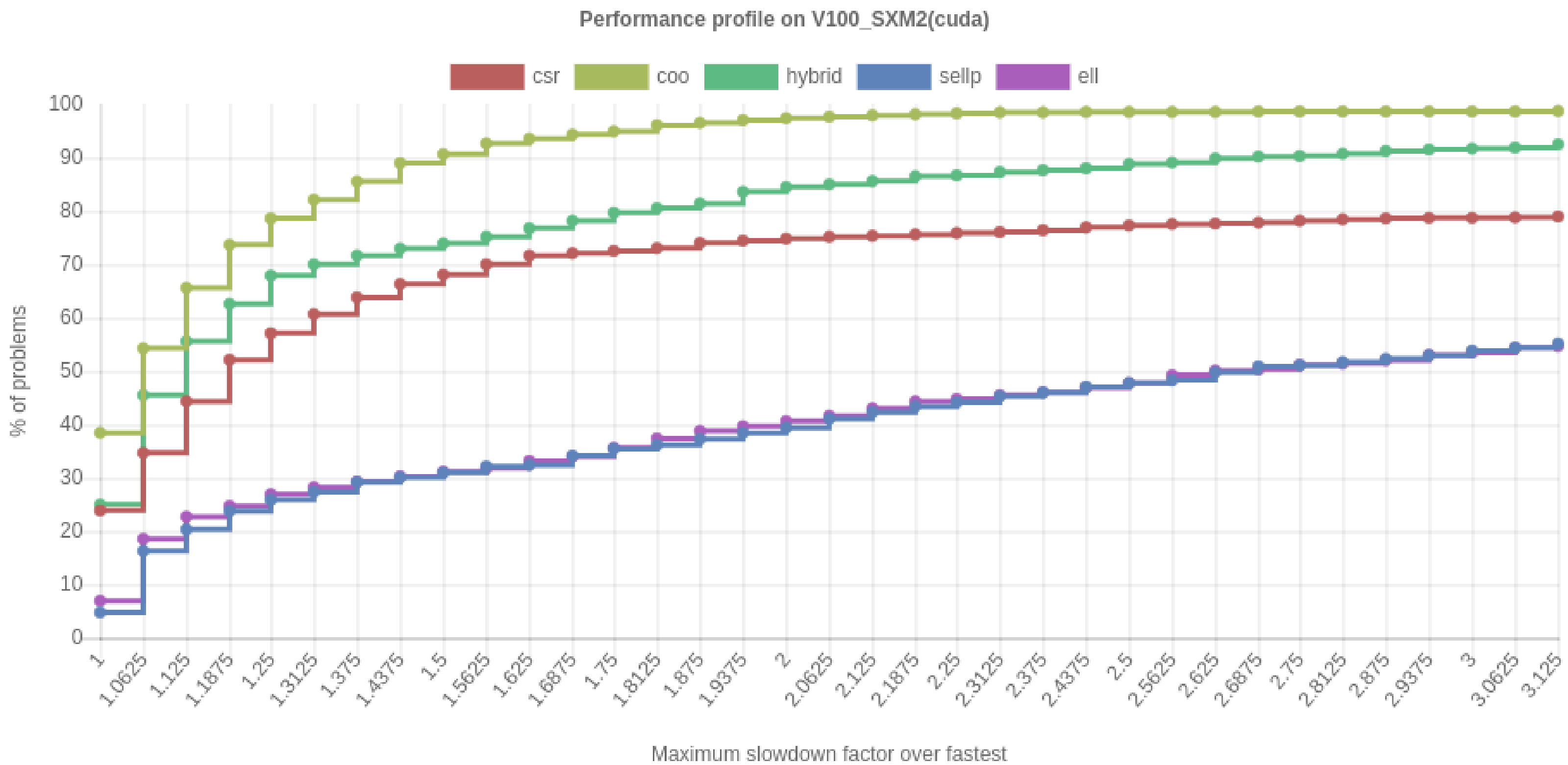
3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7
0	2	2	0	1	3	0	3
0	2	3	6				

CSR

“Standard” approach

First things first

THERE IS NO “BEST” SPARSE MATRIX FORMAT / SpMV ALGORITHM



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)

$$y := Ax$$

```
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] ];
  }
}
```

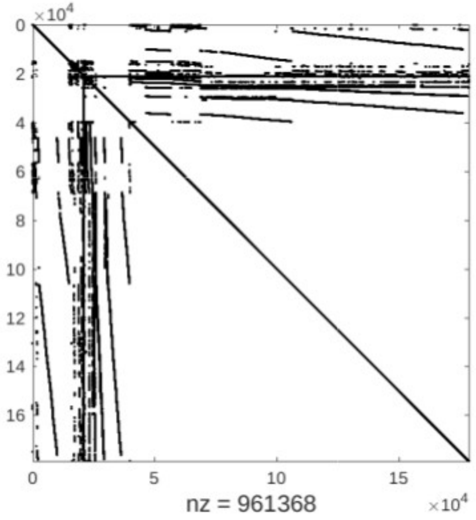
Bell & Garland ‘08
- parallelize outer loop

~ cuSPARSE SpMV

Load imbalance!

Example

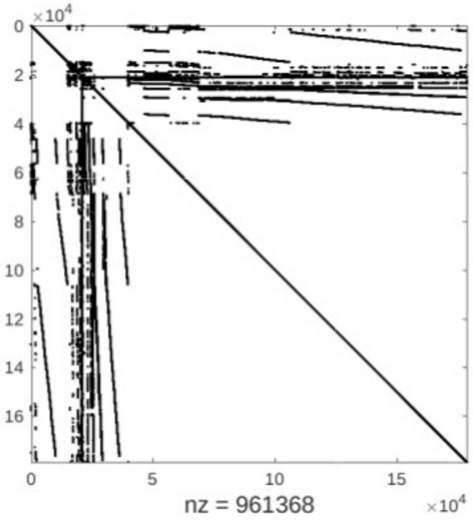
Freescale/transient

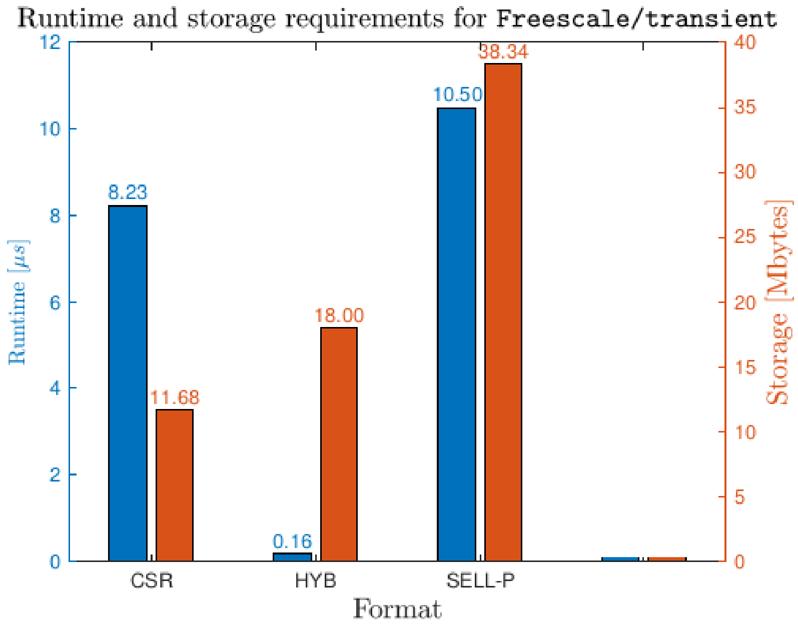


Example

* GTX 1080

Freescale/transient



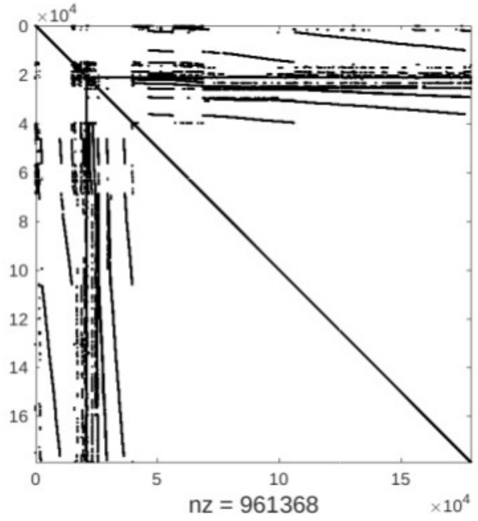


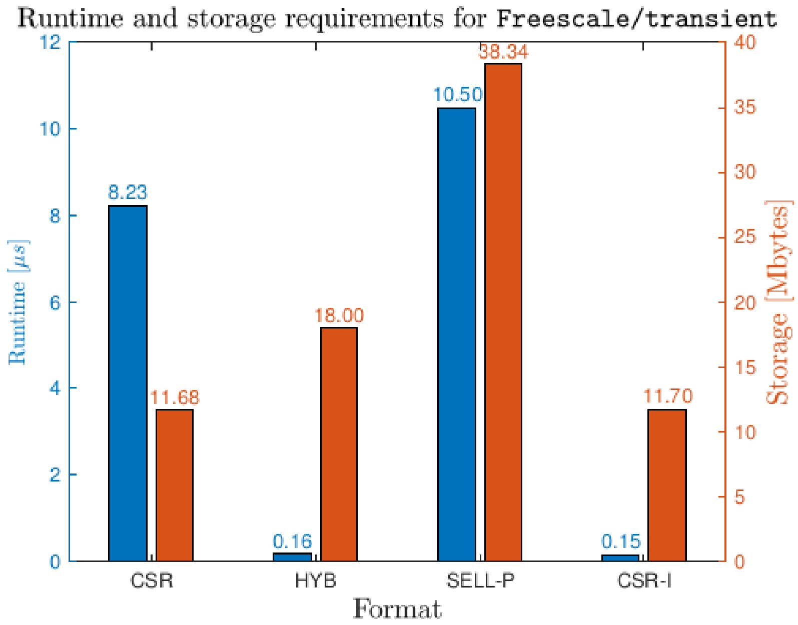
Can we do better than HYB using CSR?

Example

* GTX 1080

Freescale/transient





55x speedup

Can we do better than HYB using CSR?

YES!

Flegar, Anzt, *Overcoming Load Imbalance for Irregular Sparse Matrices,* IA3'17

Flegar, Quintana-Ortí, *Balanced CSR Sparse Matrix-Vector Product on Graphics Processors,* Euro-Par'17

How to do it

```
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

```
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

```
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

```
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

```
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

```
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 in the registers

How to do it

```
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 in the registers

How to do it

```
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 in the 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.

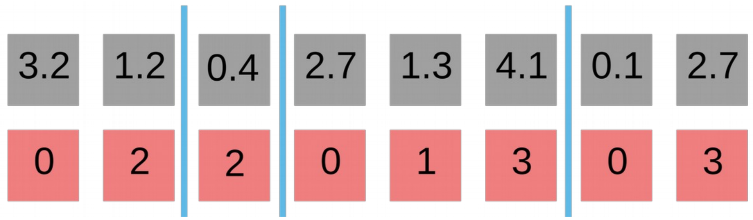
Assign one warp per chunk.

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.

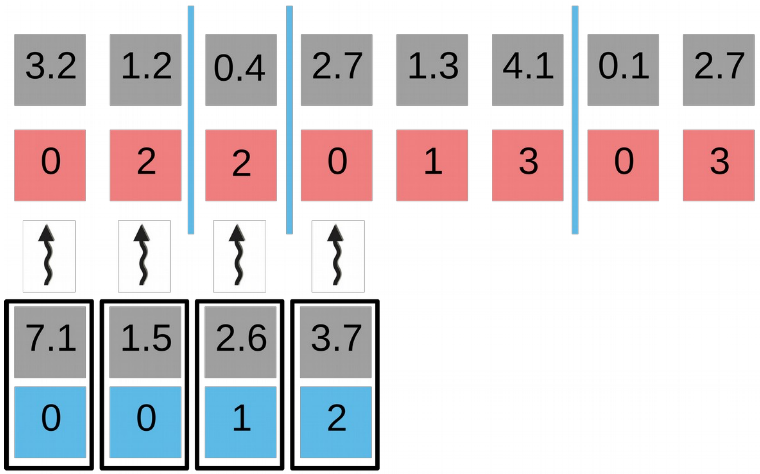


Getting good performance on GPUs

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

Spreading out threads causes strided memory access.

Assign one warp per chunk.

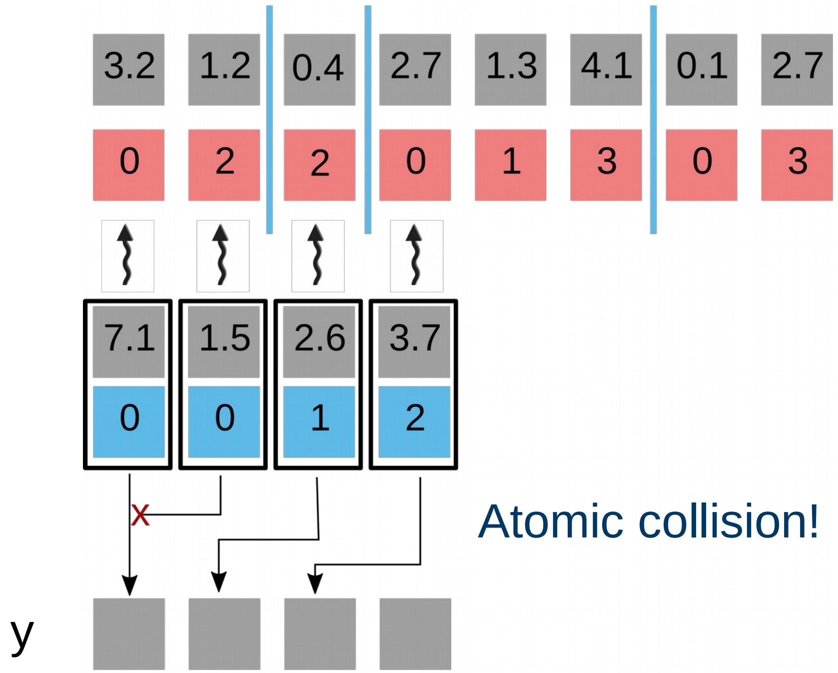


Getting good performance on GPUs

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

Spreading out threads causes strided memory access.

Assign one warp per chunk.



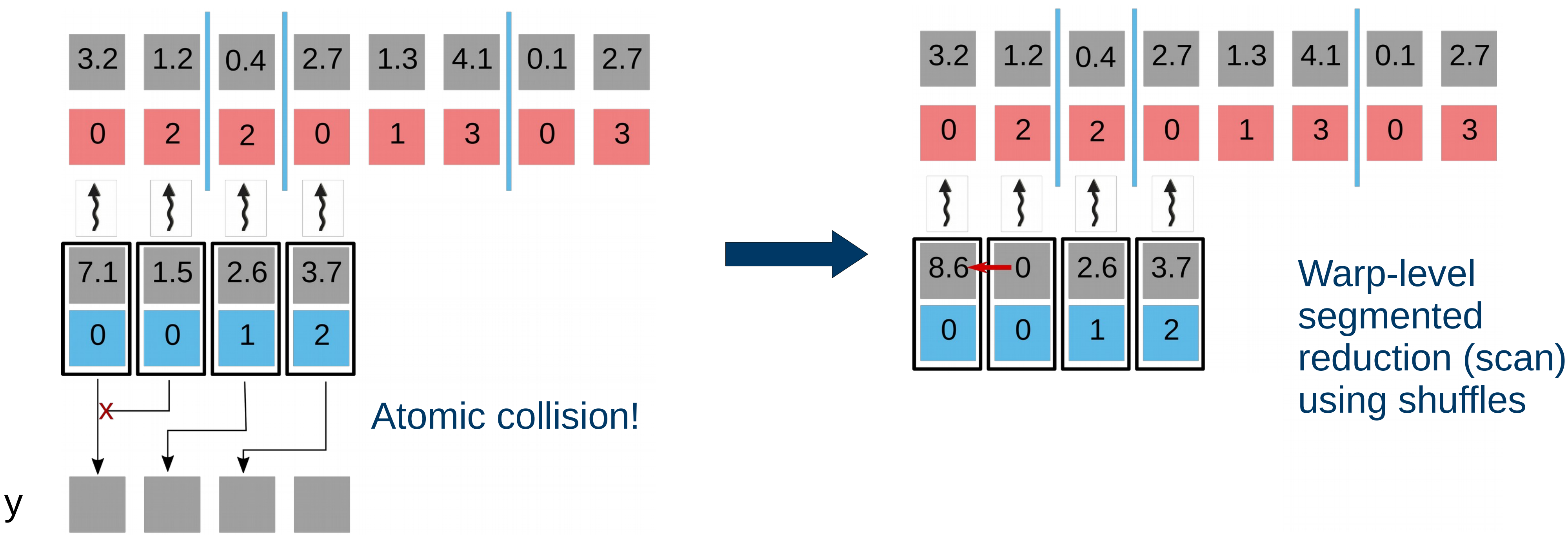
Atomic collision!

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.

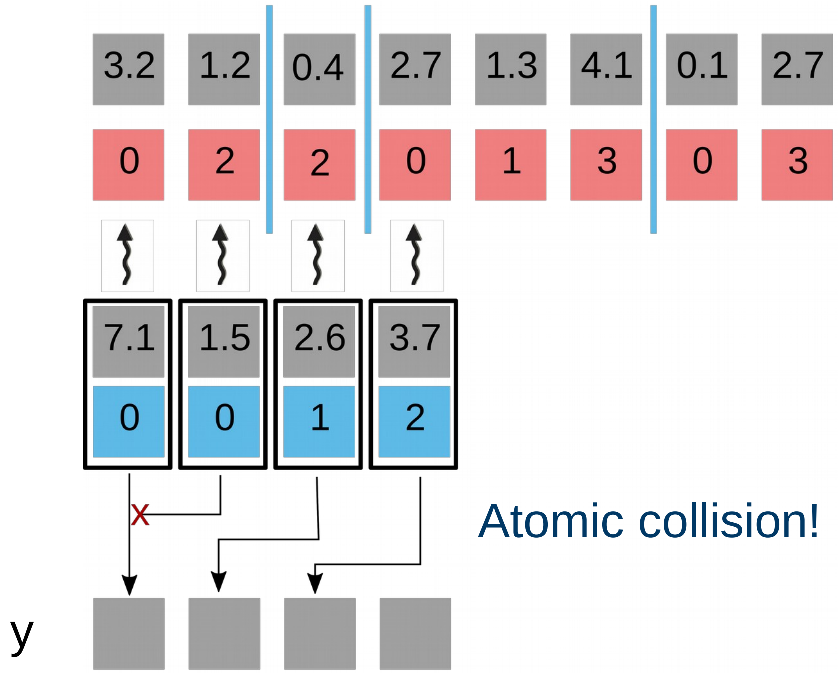


Getting good performance on GPUs

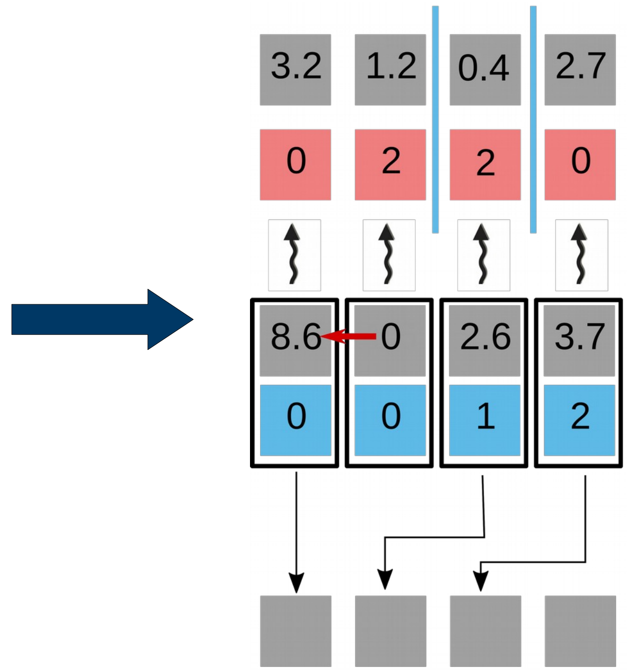
CUDA thread = 1 lane of a 32-wide SIMD unit (warp)

Spreading out threads causes strided memory access.

Assign one warp per chunk.



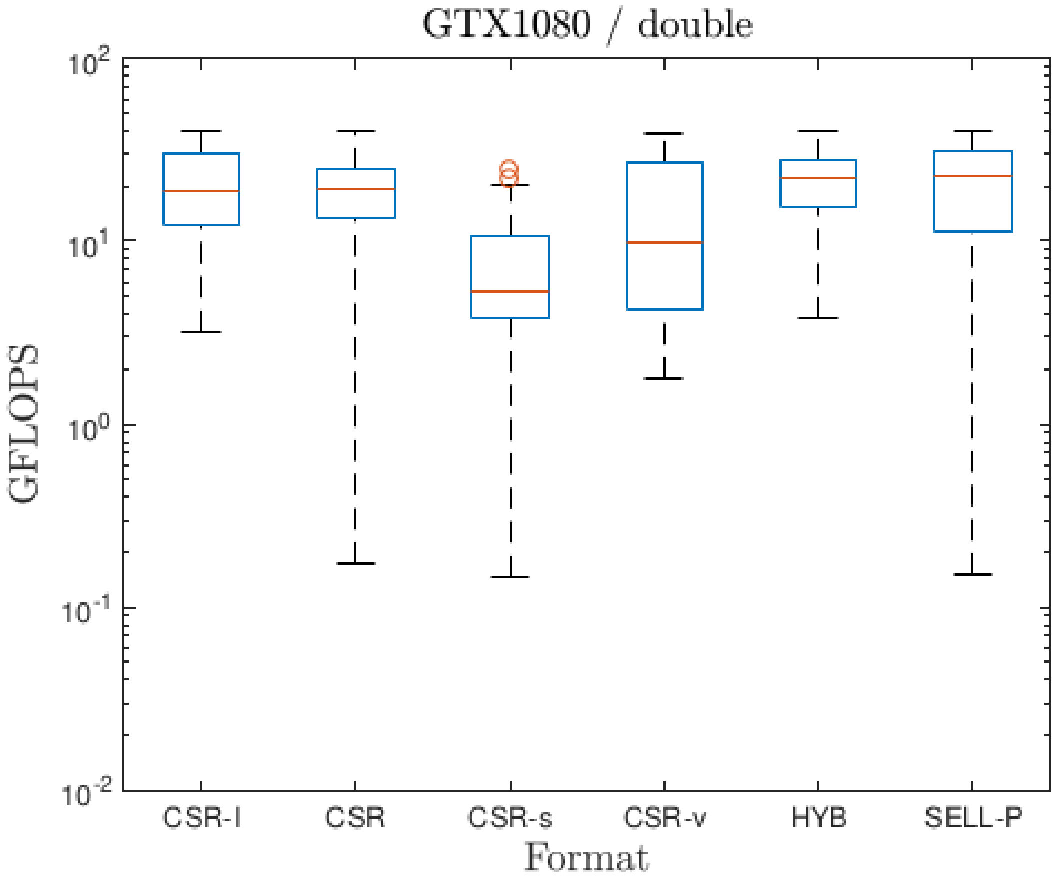
Atomic collision!



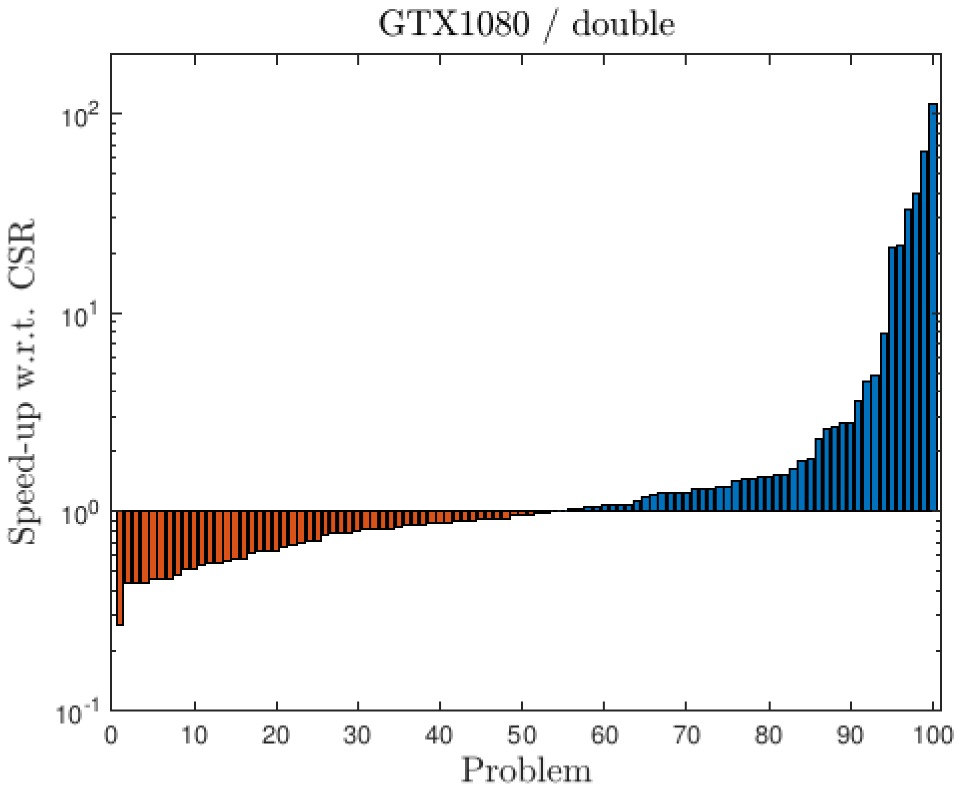
Warp-level segmented reduction (scan) using shuffles

Performance of CSR-I

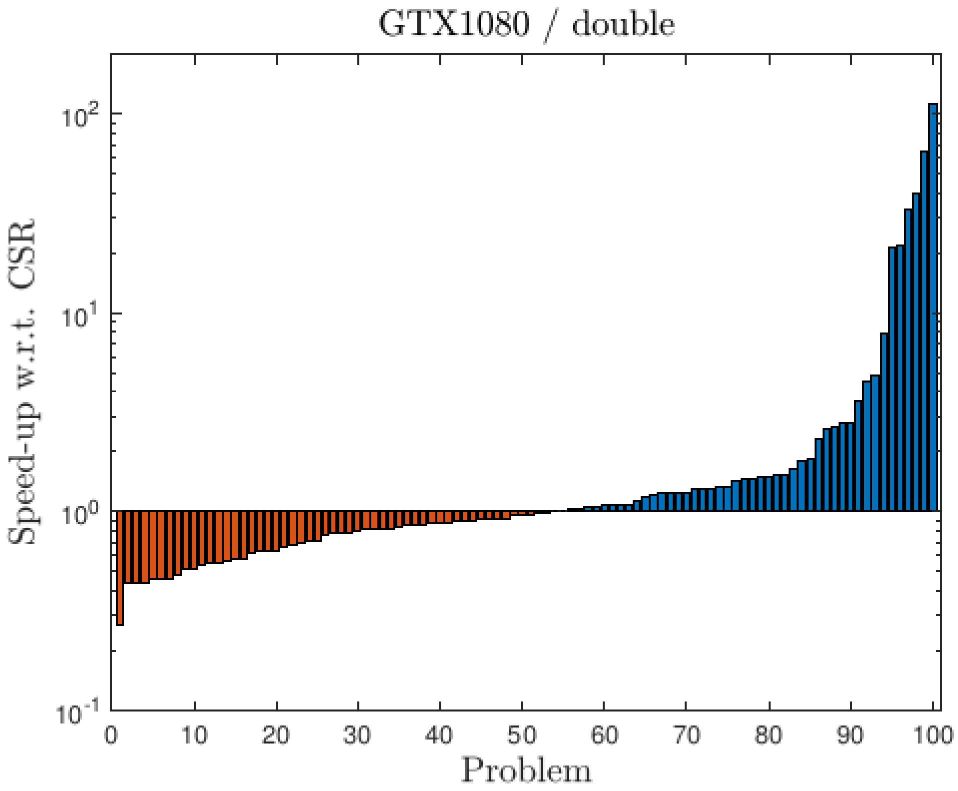
100 matrices from SuiteSparse



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

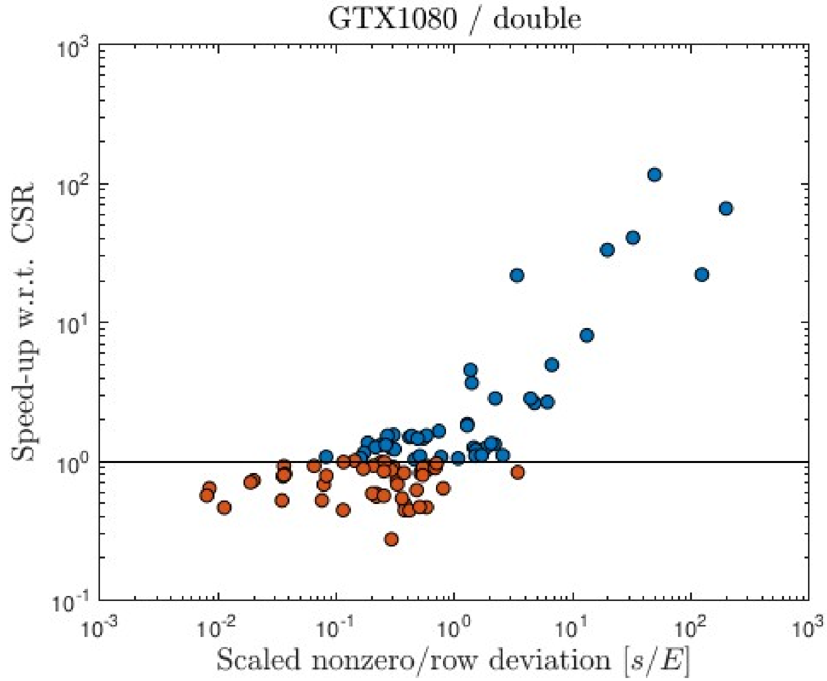
How to measure irregularity?

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

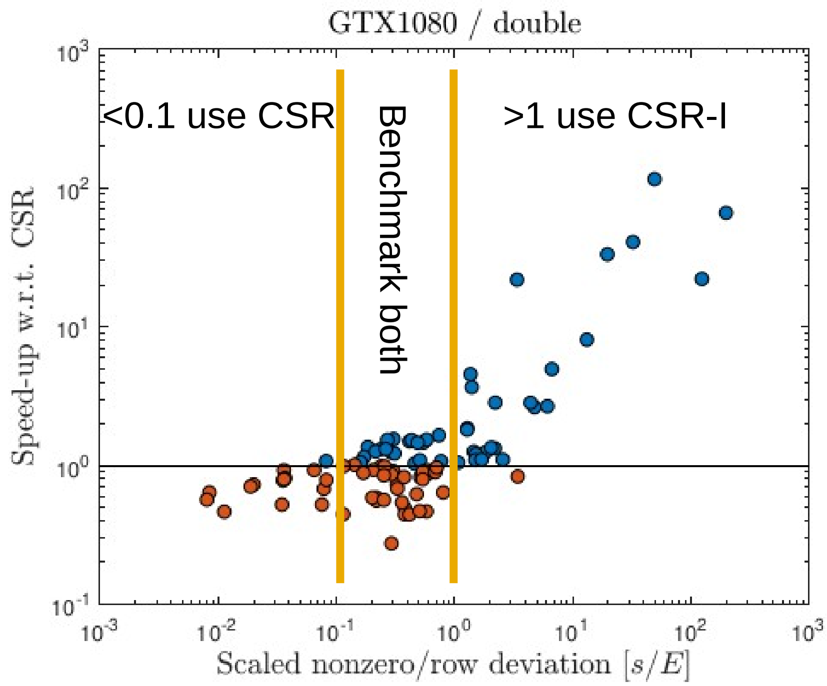
How to measure irregularity?

Deviation of row lengths from the mean.

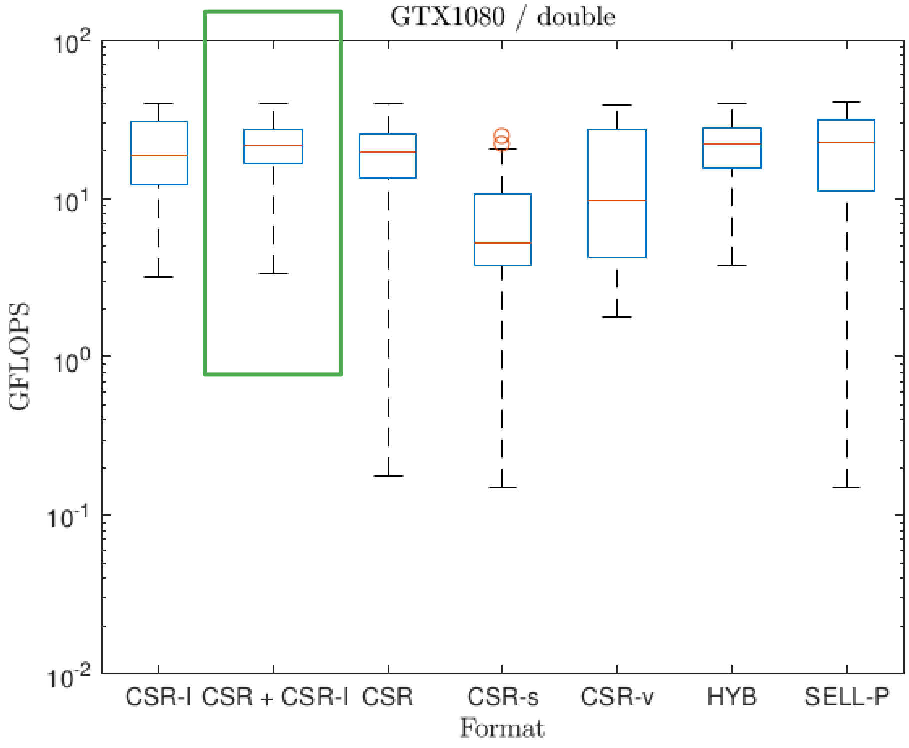
Is “5” regular or irregular?

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

Scatter plot of speedup vs normalized std. dev.



Combining both approaches



Choosing the winner a priori

CSR-I designed for irregular patterns

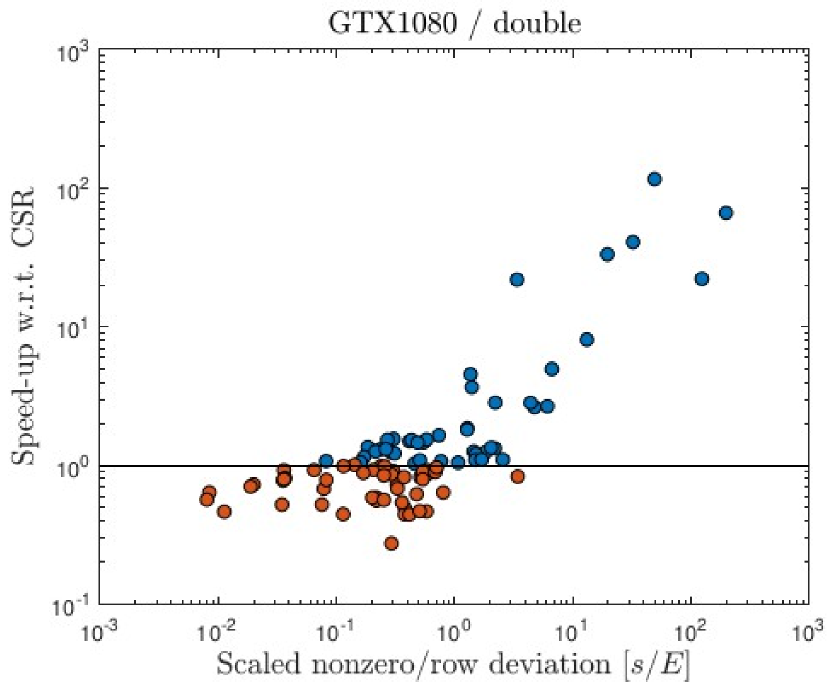
How to measure irregularity?

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

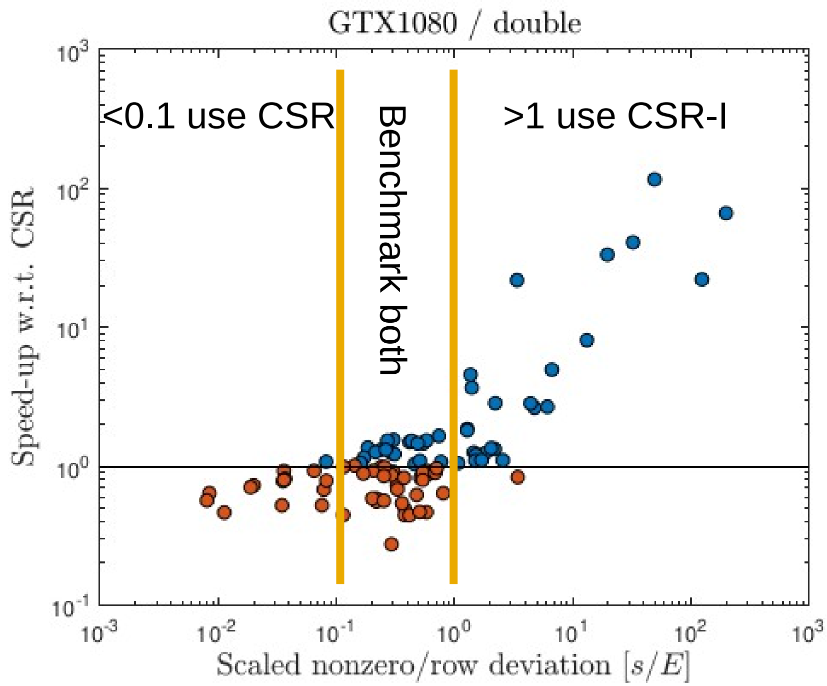
How to measure irregularity?

Deviation of row lengths from the mean.

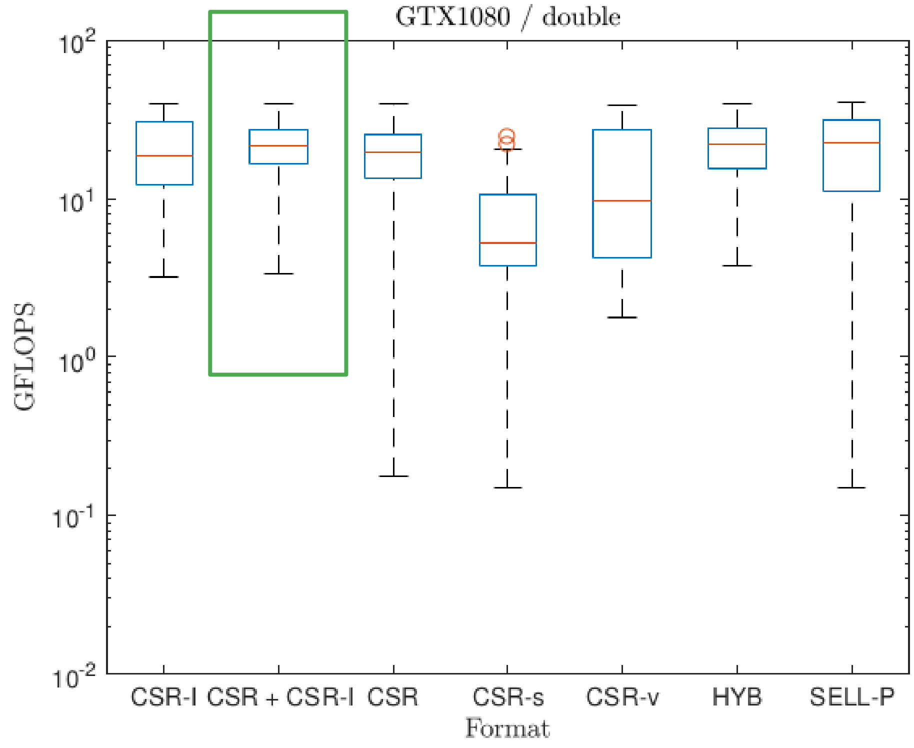
Is “5” regular or irregular?

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

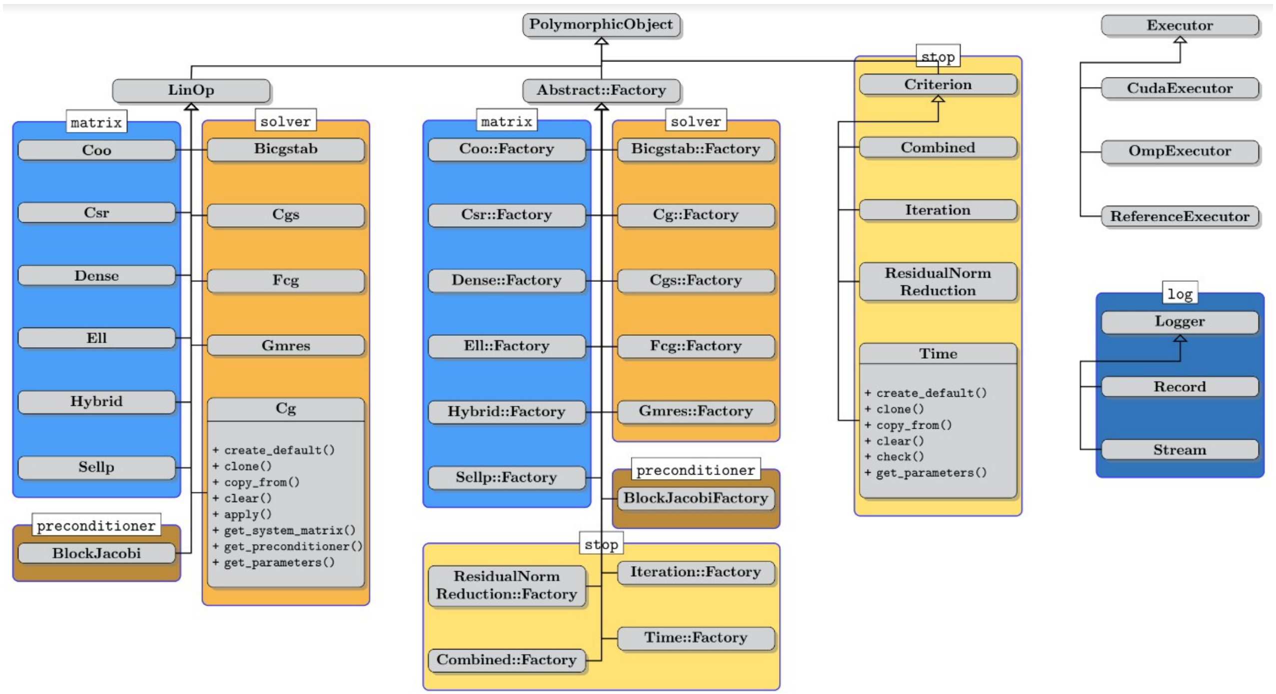
Scatter plot of speedup vs normalized std. dev.



Combining both approaches



Outlook



Choosing the correct combination of

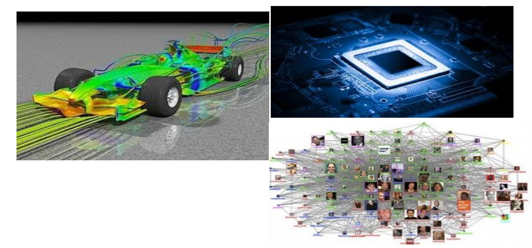
matrix format solver preconditioner

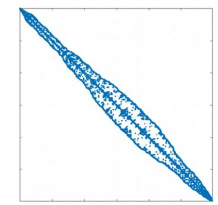
… requires expert knowledge or significant trial and error.

Design a tool that does it (semi-)automatically?

Thank you! Questions?

- Krylov-subspace based linear solvers
  - Performance depends on SpMV and Preconditioners
- Sparse matrix formats & SpMV
  - There is no “holy grail” - best format depends on the problem
- Preconditioners
  - Accelerate the solution of the solver
  - Only an approximation, can use lower precision storage
- Do not implement from scratch, use a library. For example:





$$Ax = b$$

Ginkgo

- github.com/ginkgo-project/ginkgo
- Open source license (BSD-3)
- Modern C++
- High performance GPU backend
- reference CPU implementation
- High performance CPU backend on the way

Slides:

github.com/gflegar/talks/tree/master/cmu_pittsburgh_2018_11