

How to Solve a Linear System

Goran Flegar

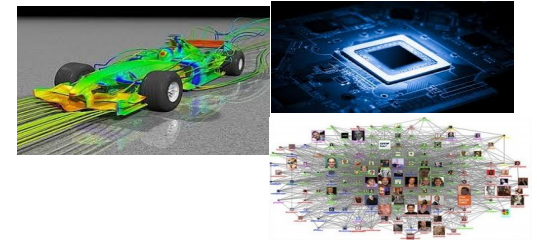
W/: Hartwig Anzt, Yen-Chen Chen, Terry Cojean, Pratik Nayak, Enrique S. Quintana-Ortí, Mike Tsai



Scan me
for slides!

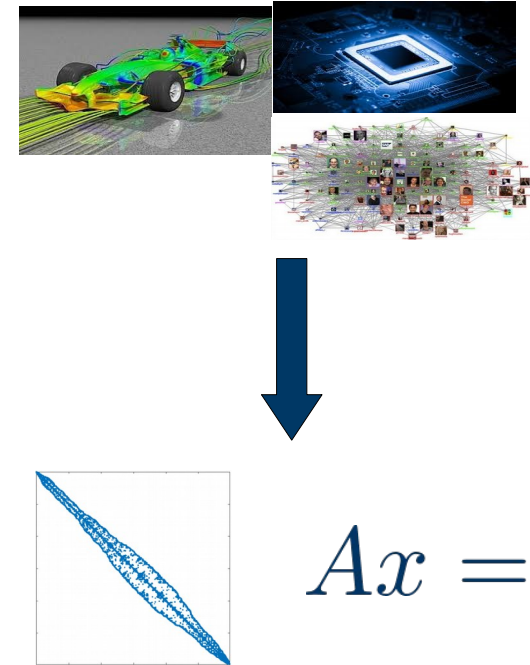
Sources of linear systems

- 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



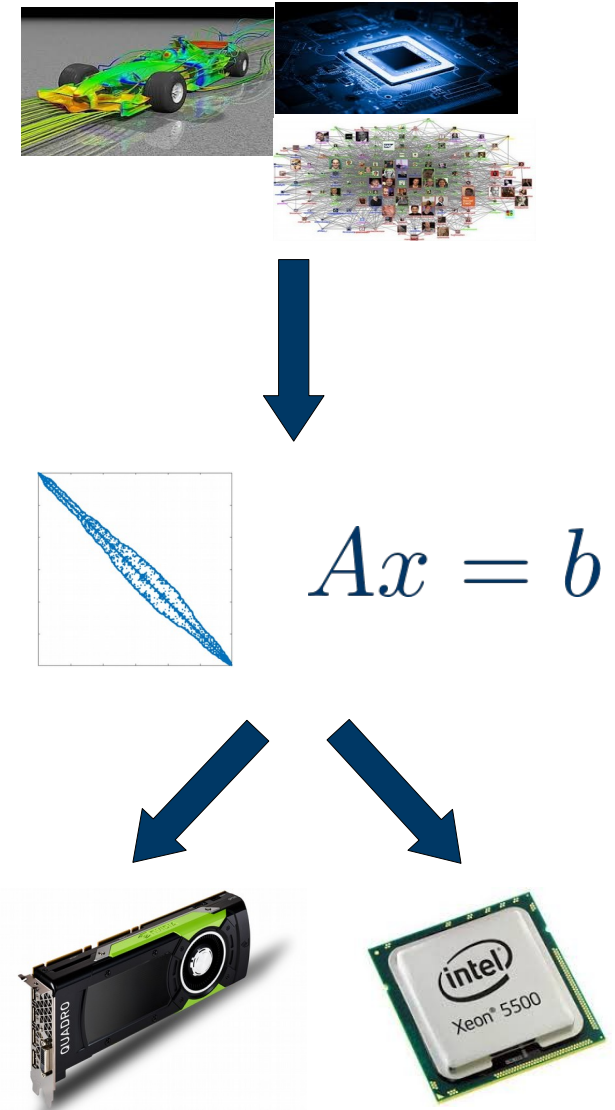
Sources of linear systems

- 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



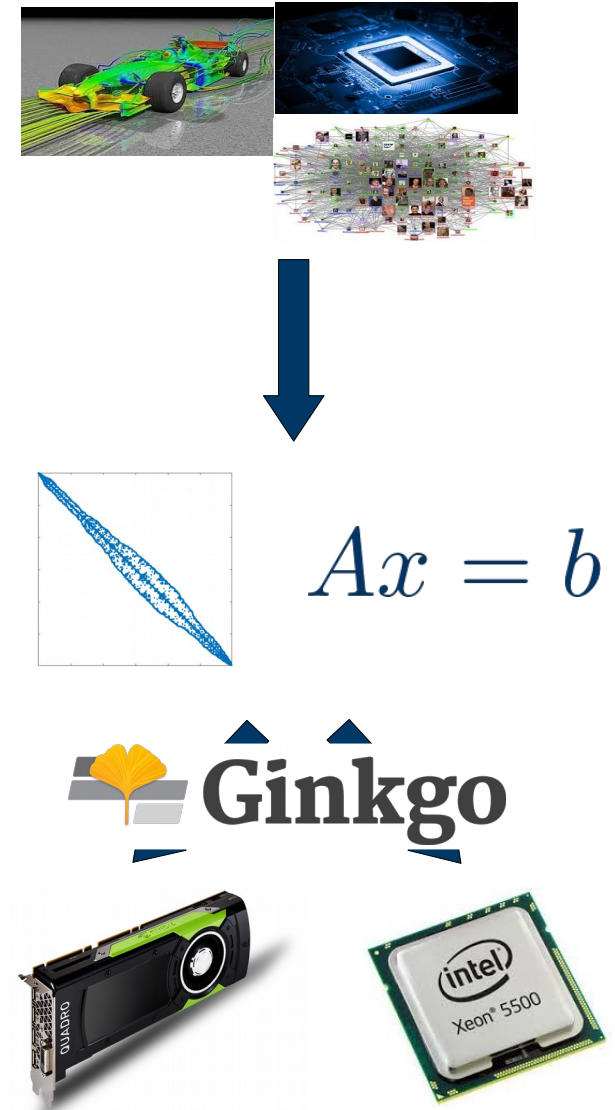
Sources of linear systems

- 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...



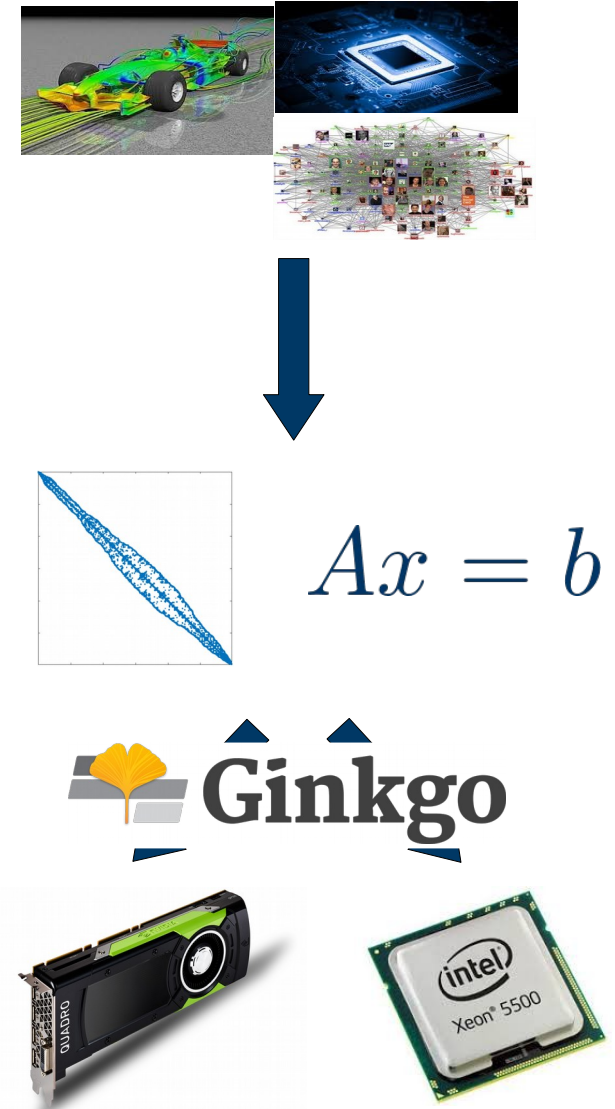
Sources of linear systems

- 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:



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...

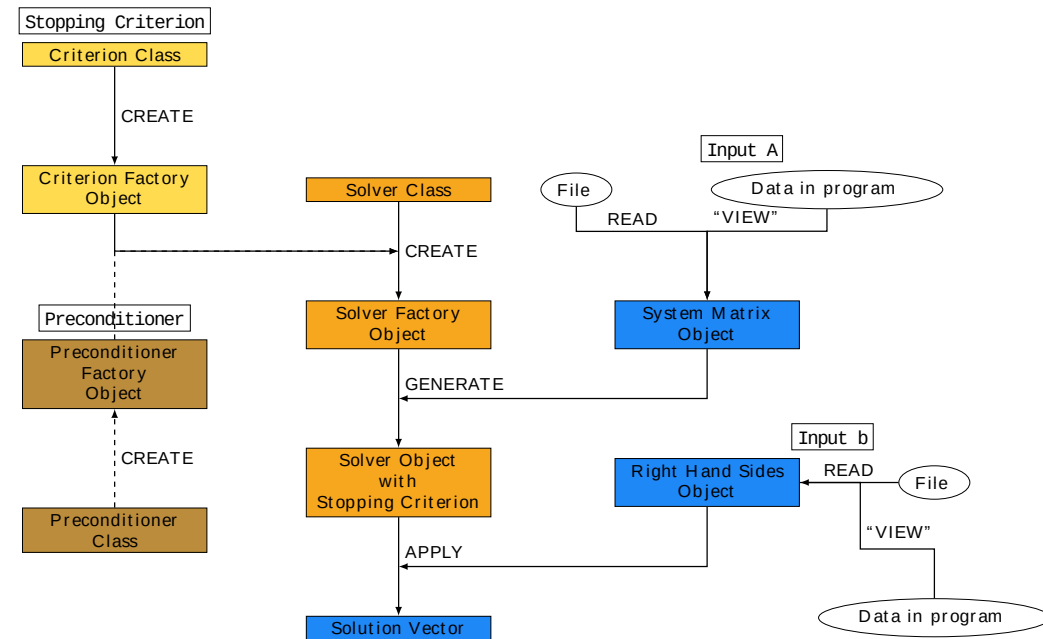


- Use a library instead:



The Ginkgo library

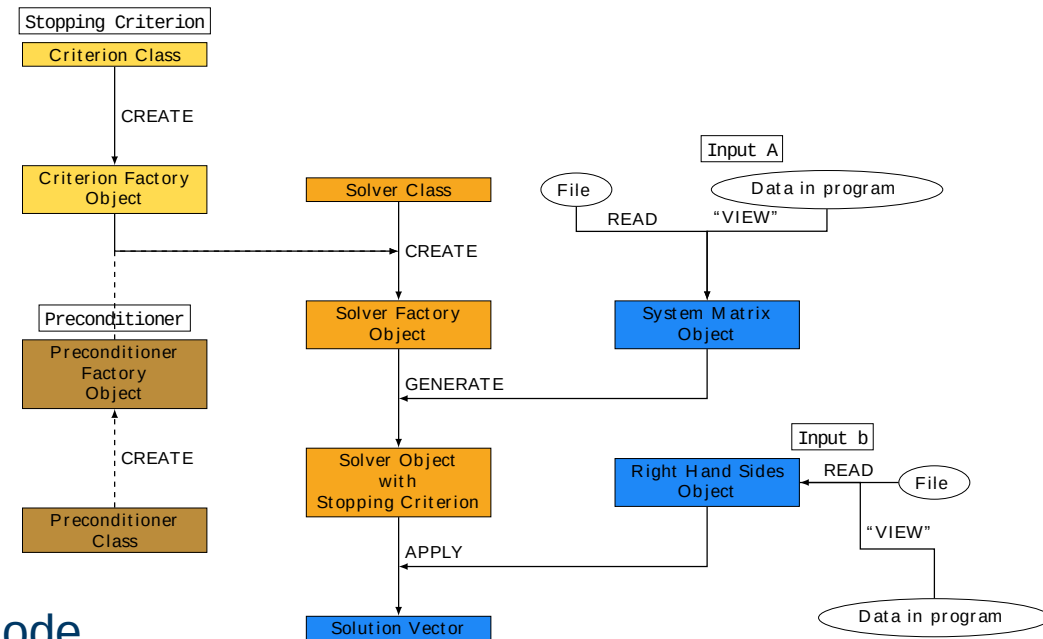
- Linear operator algebra library
 - Matrices, preconditioners, (Krylov) solvers



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

The Ginkgo library

- Linear operator algebra library
 - Matrices, preconditioners, (Krylov) solvers
- Supports execution on different devices
 - GPU
 - Sequential reference CPU
 - OpenMP under development
 - Plans for multi GPU, CPU + GPU, full node



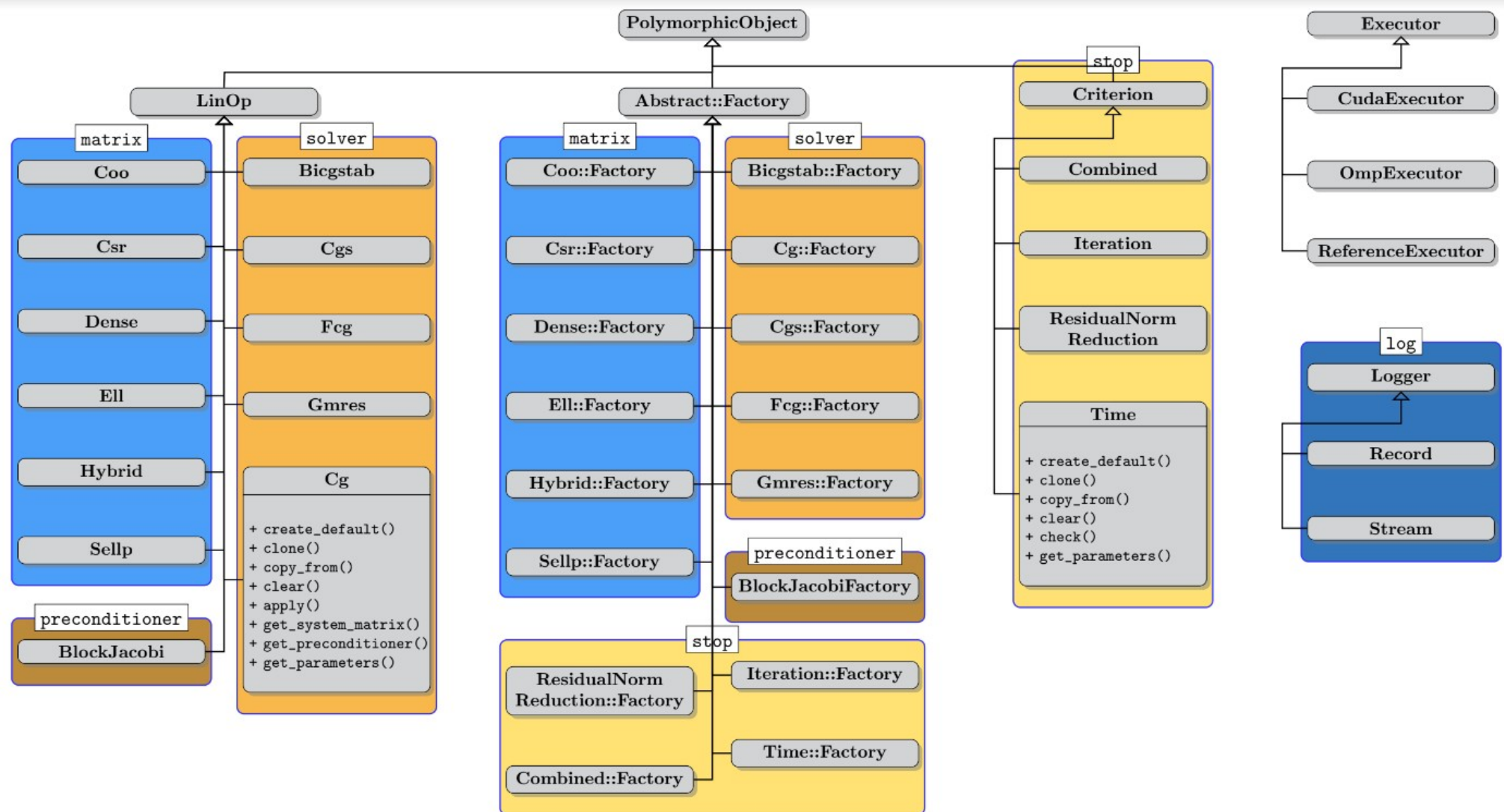
Joint effort: Innovative Computing Lab at University of Tennessee, Knoxville; Karlsruhe Institute of Technology; University Jaume I


```

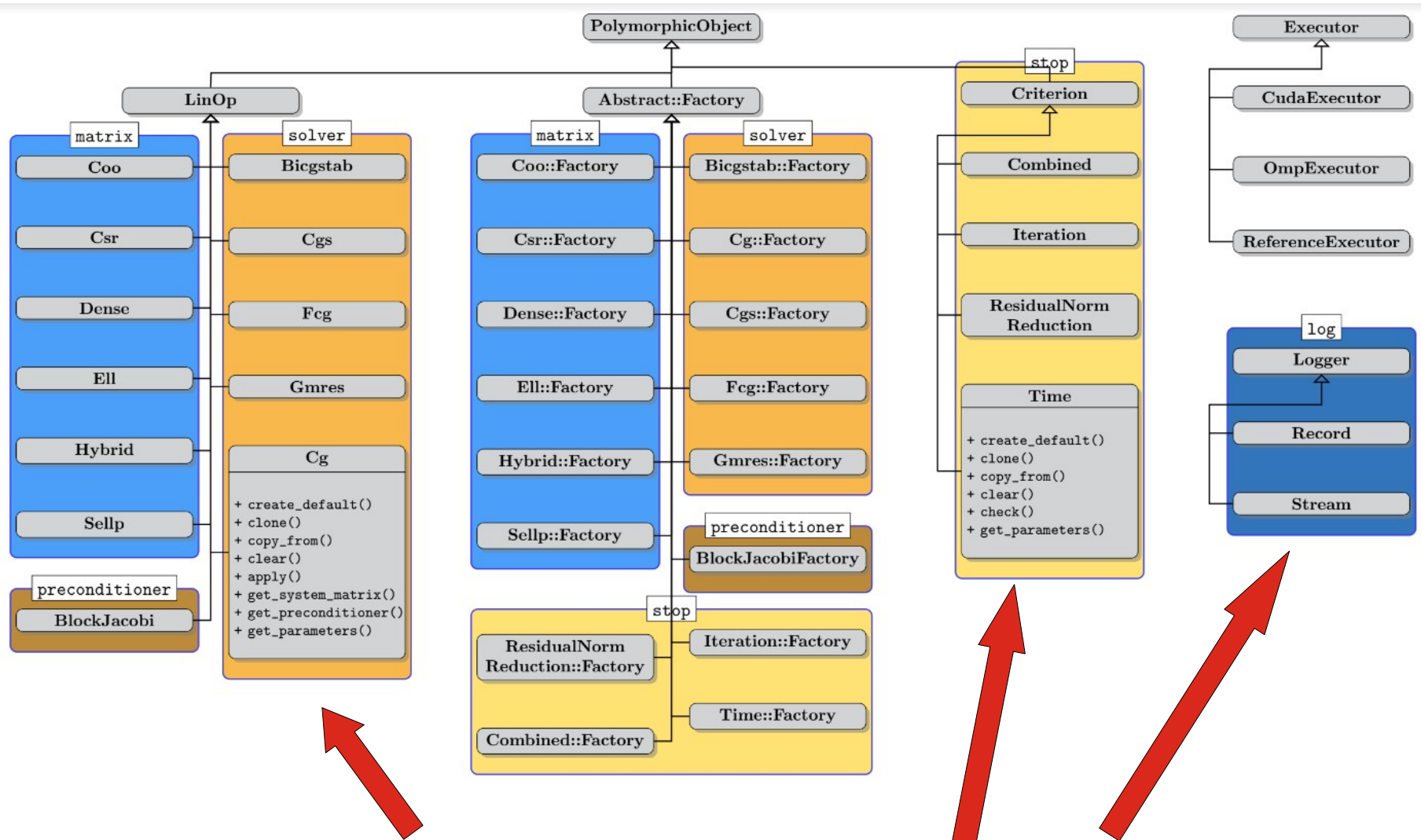
int main()
{
    // Instantiate a CUDA executor
    auto exec = gko::CudaExecutor::create(0, gko::OmpExecutor::create());
    // Read data
    auto A = gko::read<gko::matrix::Csr<>>(std::cin, exec);
    auto b = gko::read<gko::matrix::Dense<>>(std::cin, exec);
    auto x = gko::read<gko::matrix::Dense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

```

Library features



Library features: extensibility

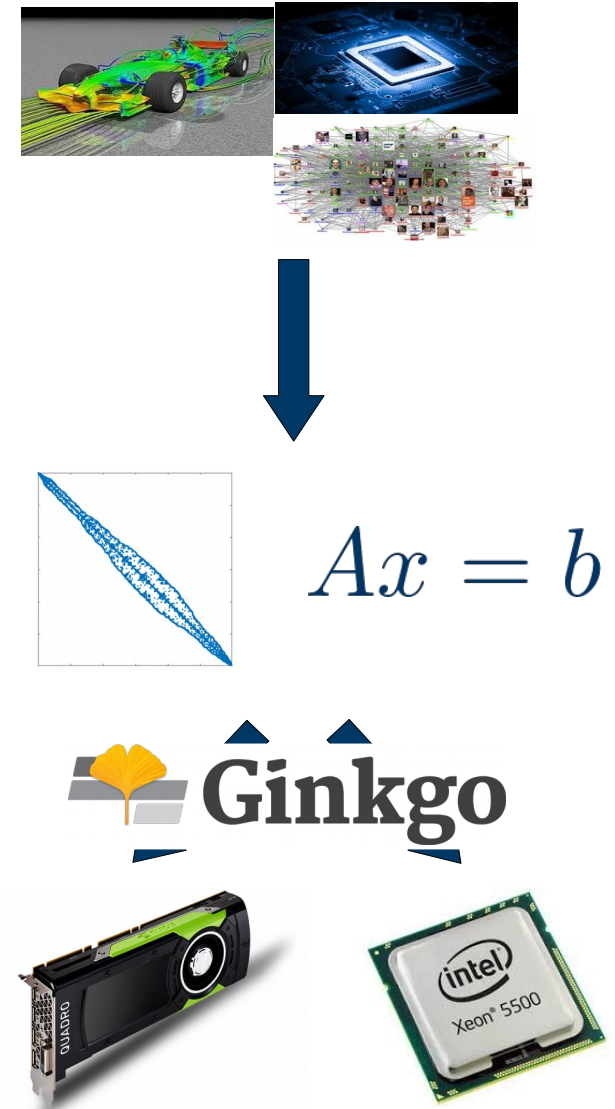


users can provide new matrices, solvers, preconditioners, stopping criteria, loggers

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:



Preconditioning

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

Preconditioning

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

Replace the original system with an
equivalent preconditioned system

Preconditioning

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

Replace the original system with an
equivalent preconditioned system

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

Preconditioning

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

Replace the original system with an
equivalent preconditioned system

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

~~$$M^{-1}A$$~~

**Do not compute the preconditioned
system matrix explicitly!**

Preconditioning

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

Replace the original system with an
equivalent preconditioned system

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

~~$$M^{-1}A$$~~

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

**Do not compute the preconditioned
system matrix explicitly!**

Preconditioning

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

Replace the original system with an equivalent preconditioned system

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

~~$$M^{-1}A$$~~

Do not compute the preconditioned system matrix explicitly!

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



$$z := Ax$$

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

Preconditioner application

Preconditioning

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

Replace the original system with an equivalent preconditioned system

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

~~$$M^{-1}A$$~~

Do not compute the preconditioned system matrix explicitly!

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



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

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

Preconditioner setup



$$\begin{aligned} z &:= Ax \\ y &:= M^{-1}z \end{aligned}$$

Preconditioner application

Preconditioning

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

Replace the original system with an equivalent preconditioned system

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

~~$$M^{-1}A$$~~

Do not compute the preconditioned system matrix explicitly!

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



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

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

Generation via factory



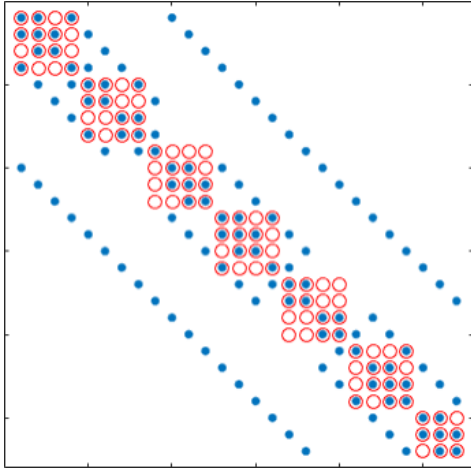
$$\begin{aligned} z &:= Ax \\ y &:= M^{-1}z \end{aligned}$$

Linear operator application



Ginkgo linear operator abstraction

Example: Block-Jacobi preconditioning



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

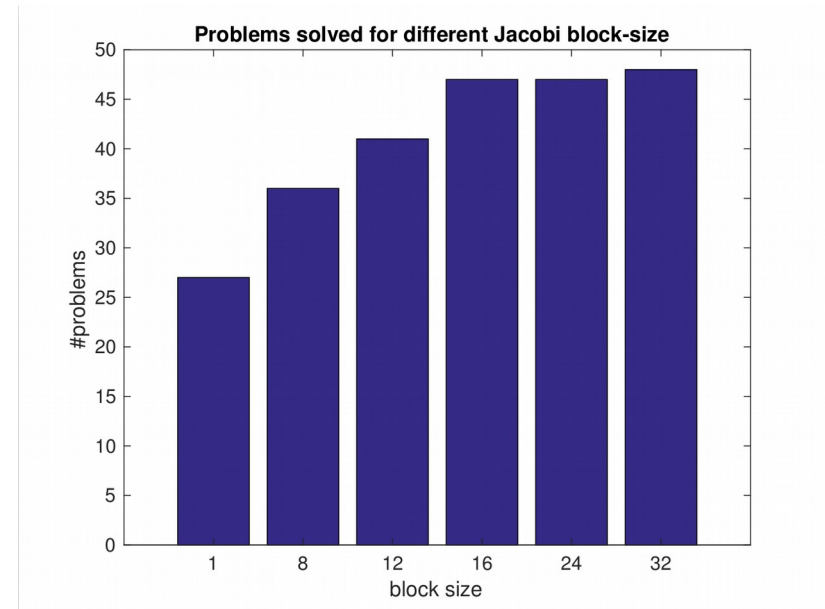
$$\text{diag}(A) = [D_1, \dots, D_k]$$

$$M := \text{diag}(D_1, \dots, D_k)$$

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

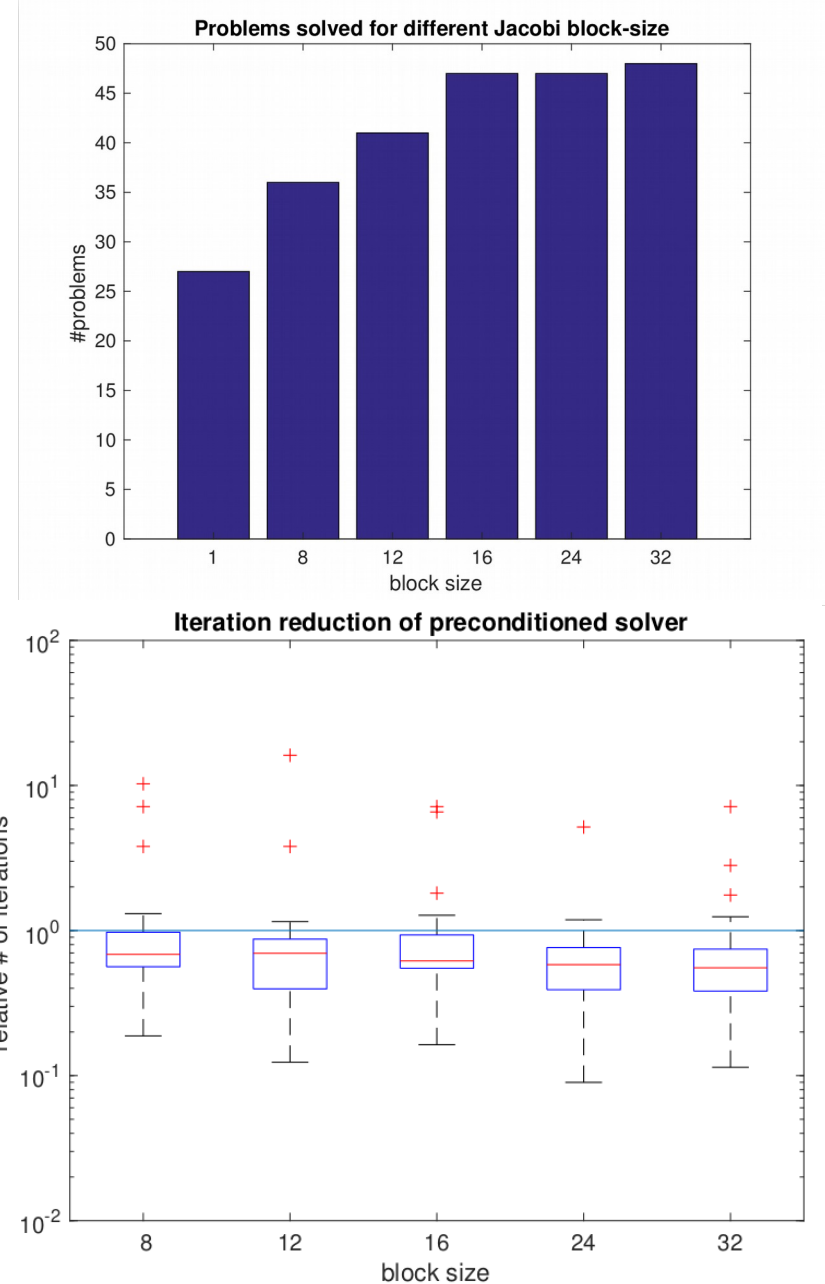
Benefits of block-Jacobi

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



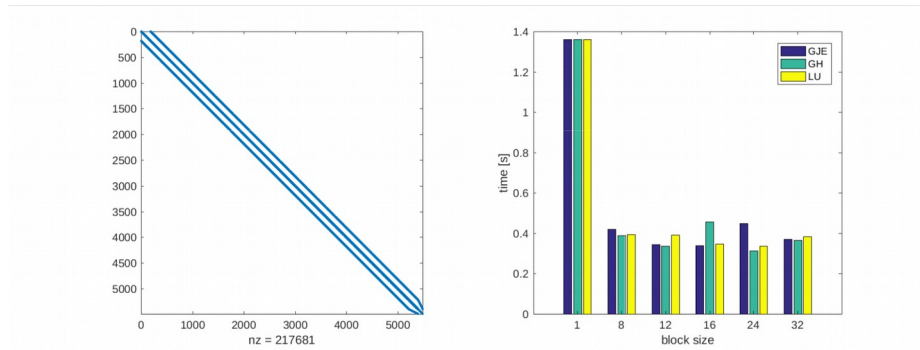
Benefits of block-Jacobi

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

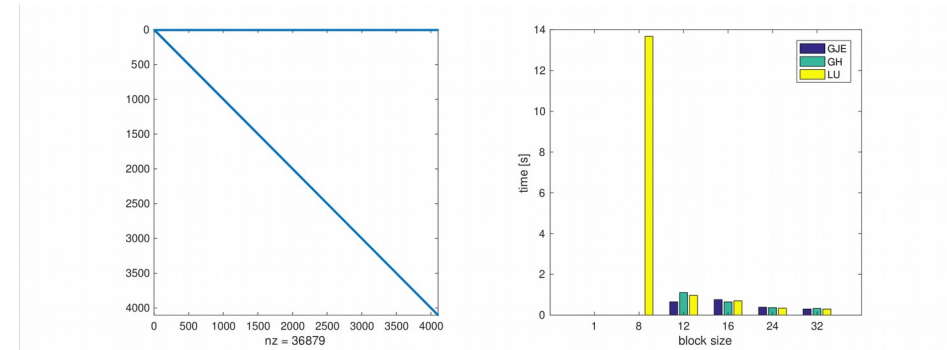


Complete solver runtime

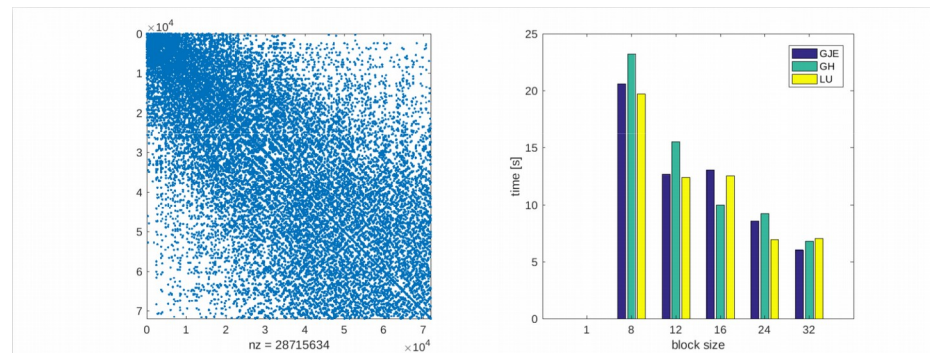
s2rmt3m1



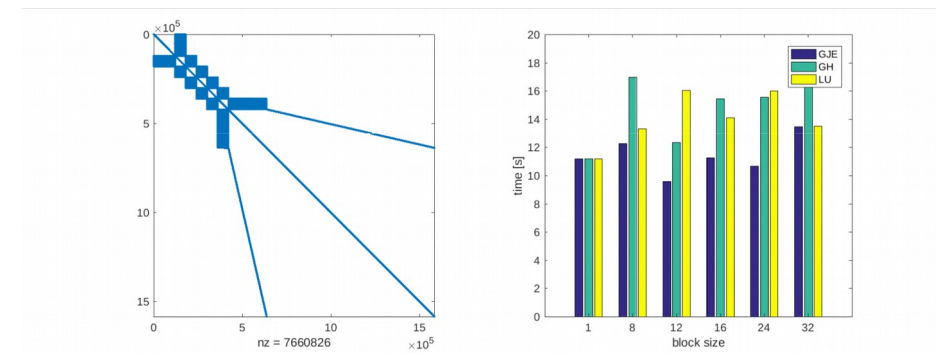
Chebyshev3



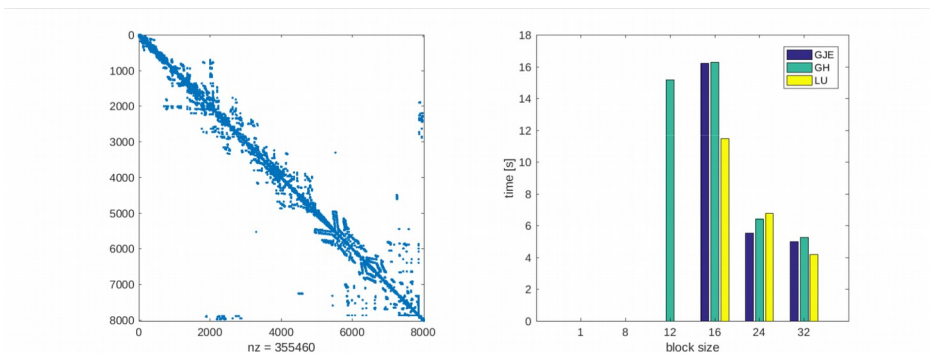
nd24k



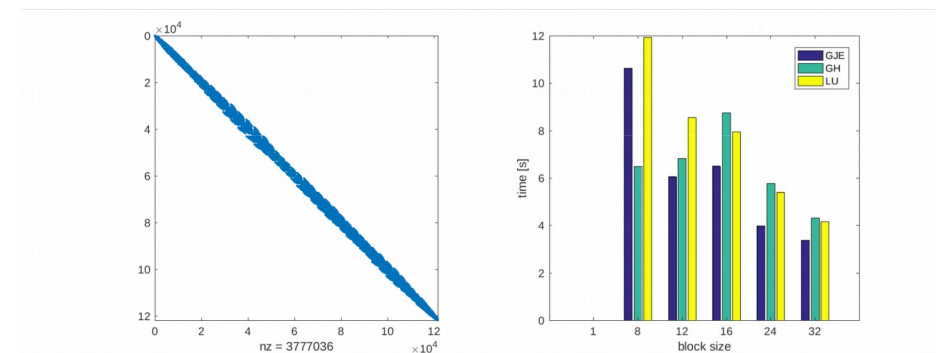
G3_circuit



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

Adaptive precision in inversion-based block-Jacobi:

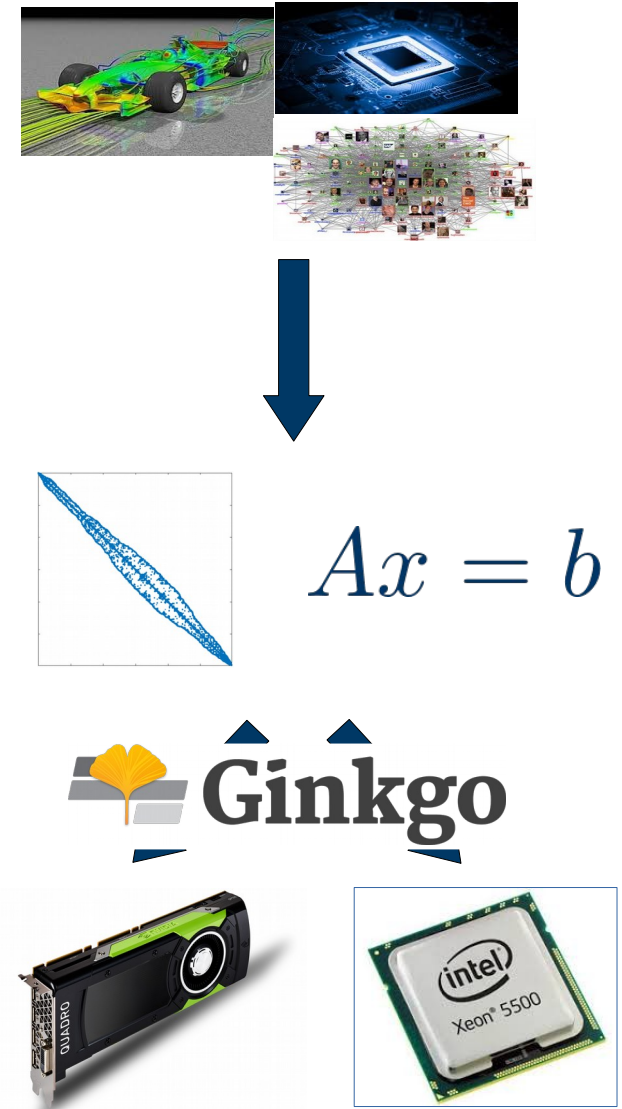
- All **computation** is done in **double precision**
- Preconditioner matrix is **stored** in **lower precision**, with roundoff error “ u ”
- Error bound:

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

Anzt, Dongarra, Flegar, Higham, Quintana-Ortí, *Adaptive precision in block-Jacobi preconditioning for iterative sparse linear system solvers*, CCPE

Sources of linear systems

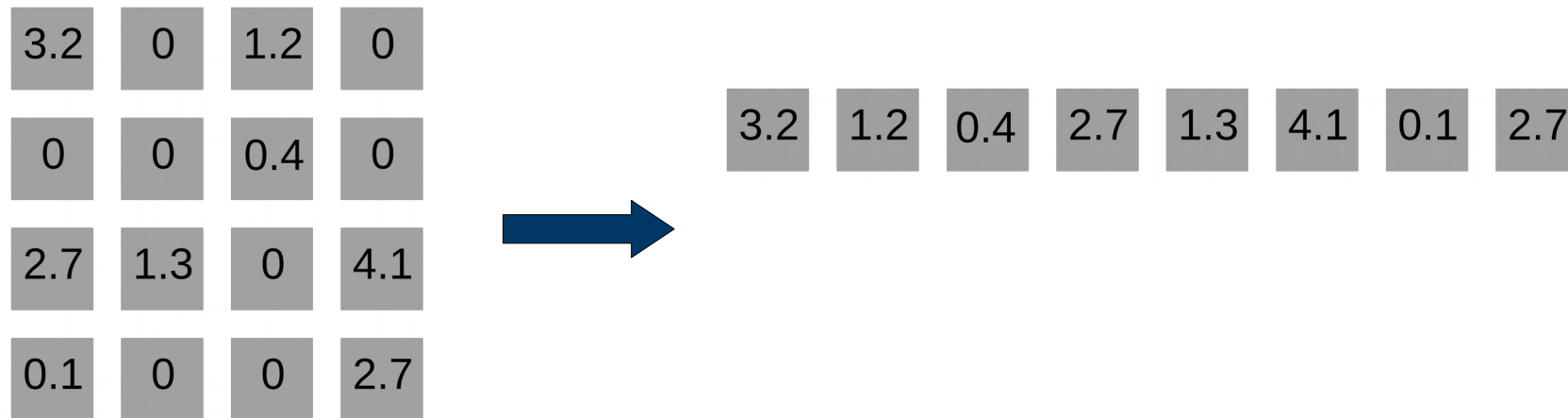
- Real-world problem transformed into a linear system via:
 - PDE discretizations, graph representations
 - Large number of unknowns (1M+, full matrix 8TB)
 - Most matrix elements are zero
- Possible approach: iterative methods
 - Krylov-subspace based linear solvers
 - SpMV
 - BLAS-1 operations
 - Sparse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



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

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



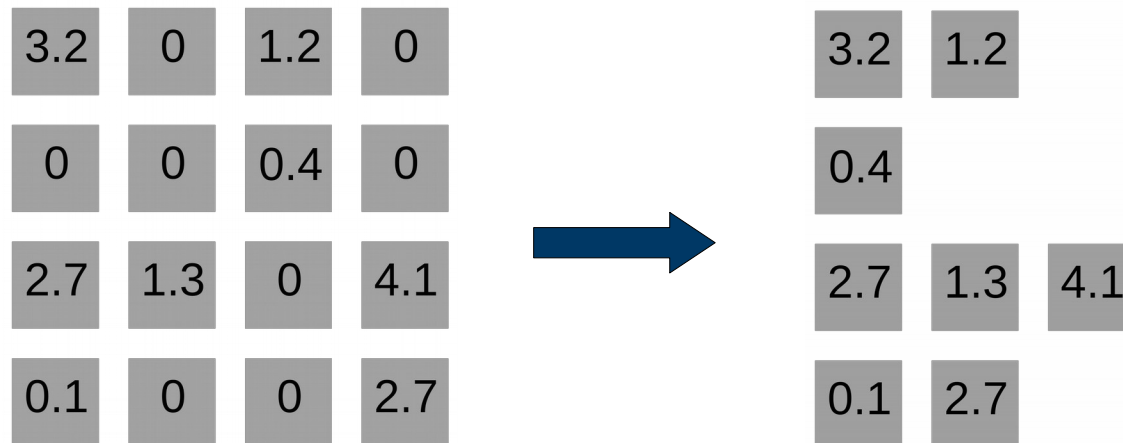
3.2	1.2	0.4	2.7	1.3	4.1	0.1	2.7
0	0	1	2	2	2	3	3
0	2	2	0	1	3	0	3

COO

row

column

Sparse matrix formats



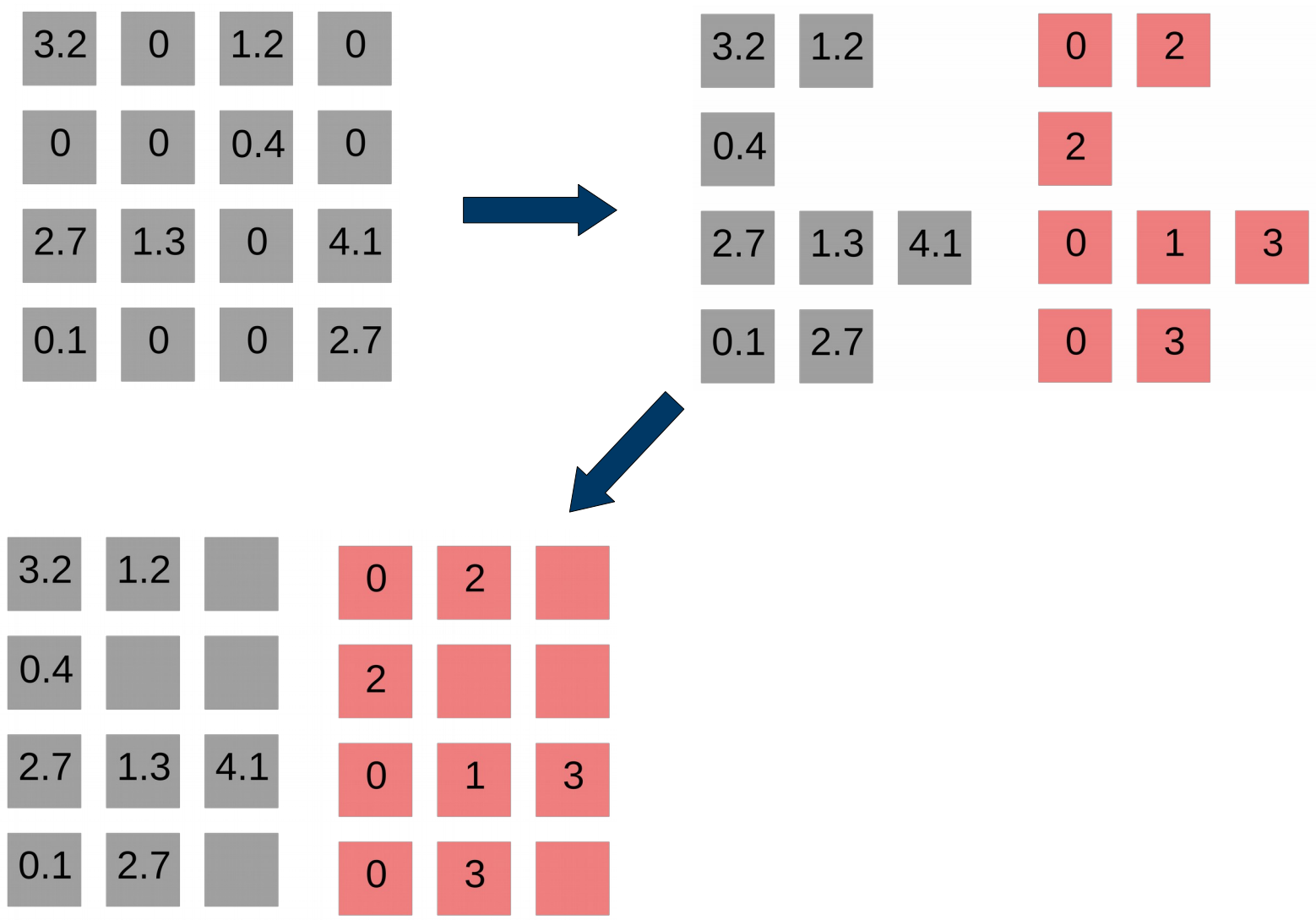
Sparse matrix formats

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



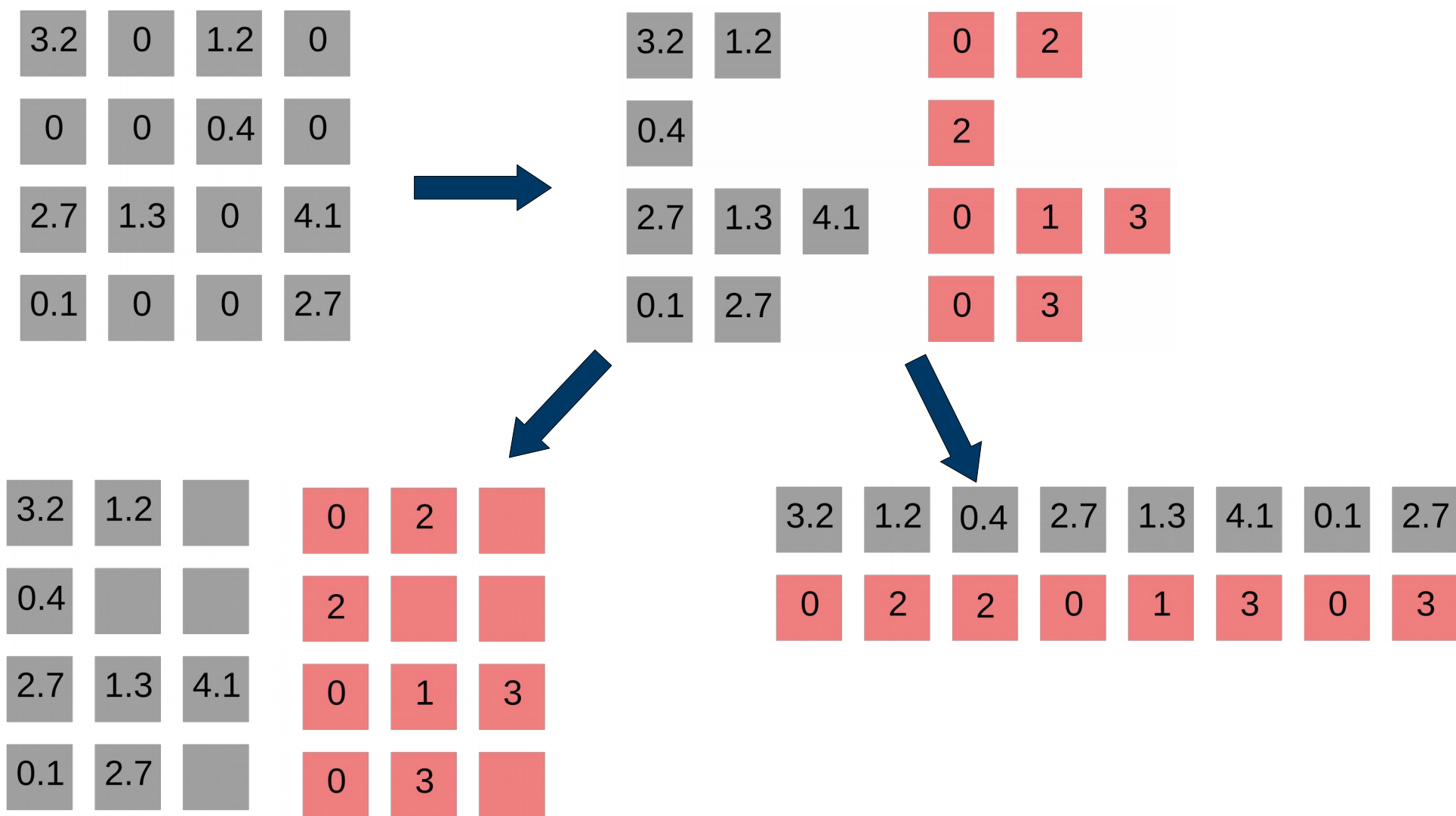
3.2	1.2			0	2	
0.4				2		
2.7	1.3	4.1		0	1	3
0.1	2.7			0	3	

Sparse matrix formats



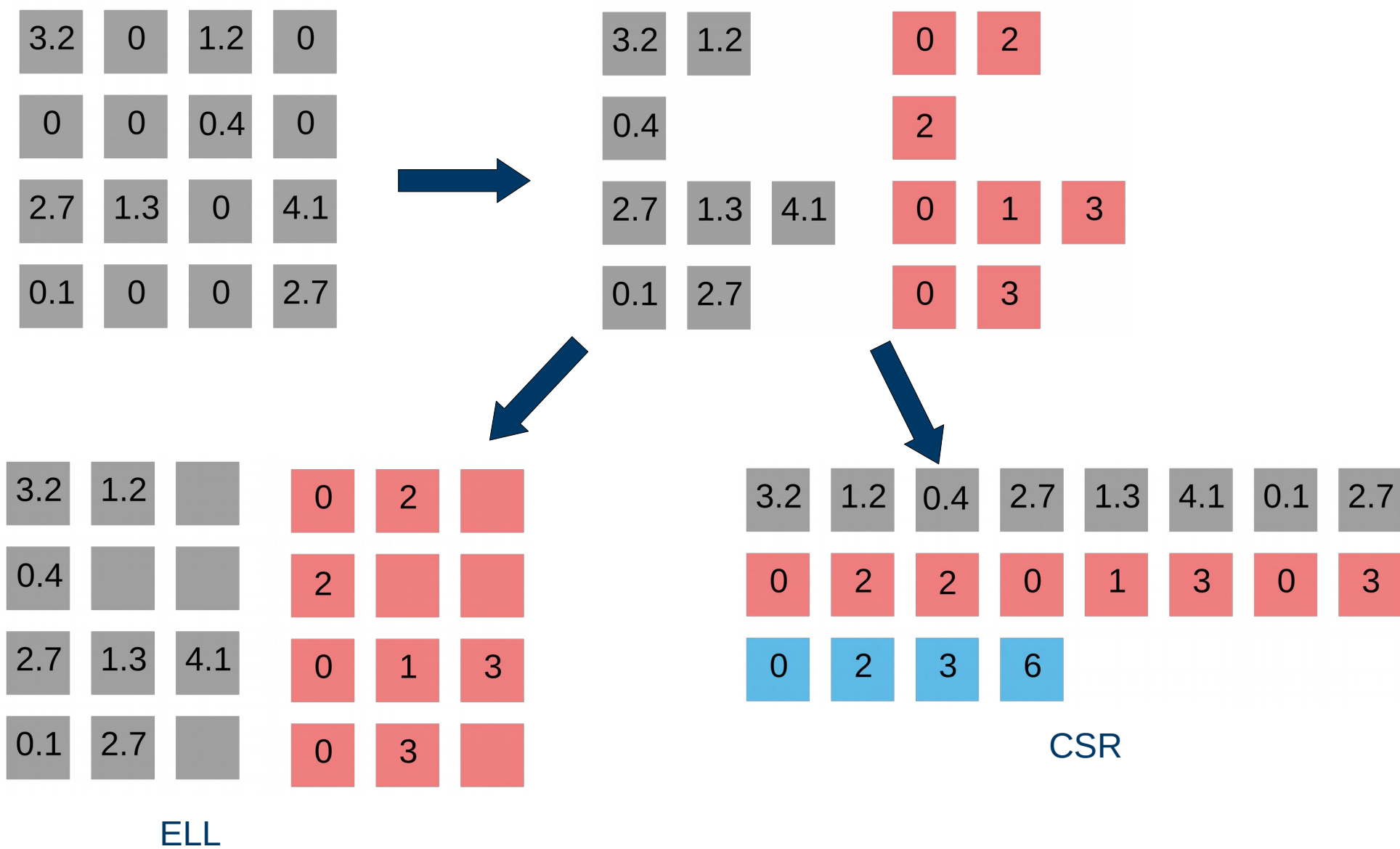
ELL

Sparse matrix formats



ELL

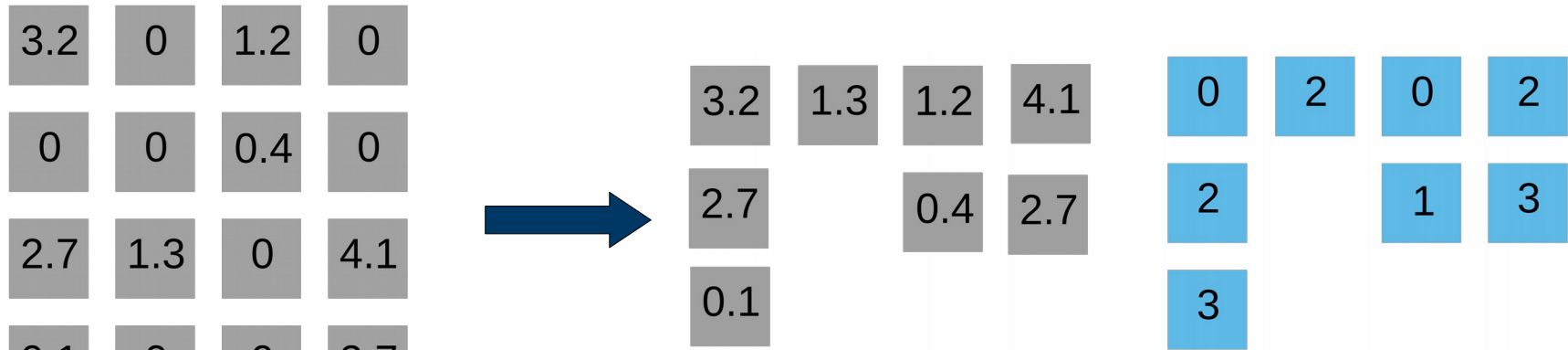
Sparse matrix formats



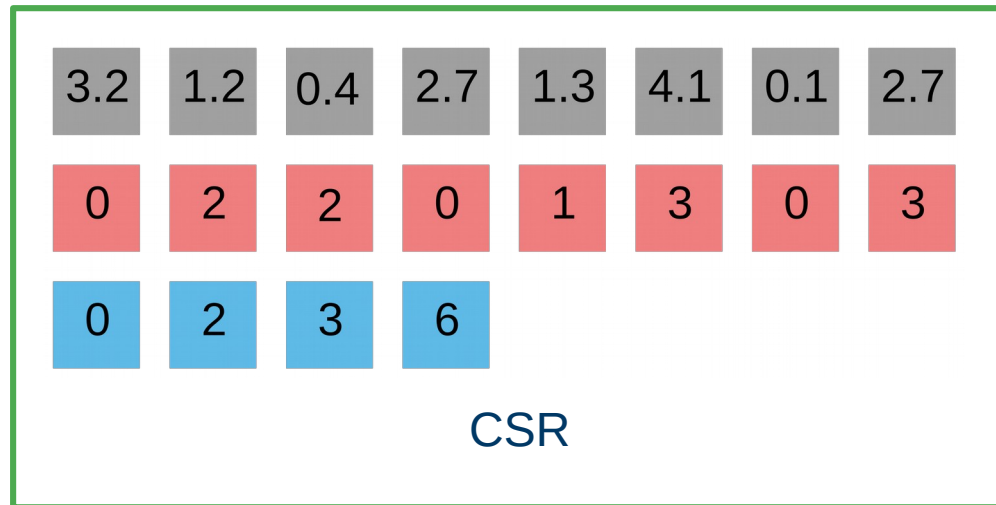
Sparse matrix formats



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)

$$y := Ax$$

```
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 }
```

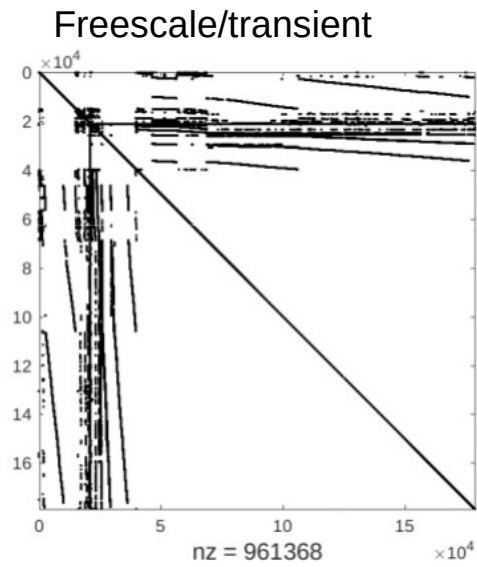
Bell & Garland '08

- parallelize outer loop

~ cuSPARSE SpMV

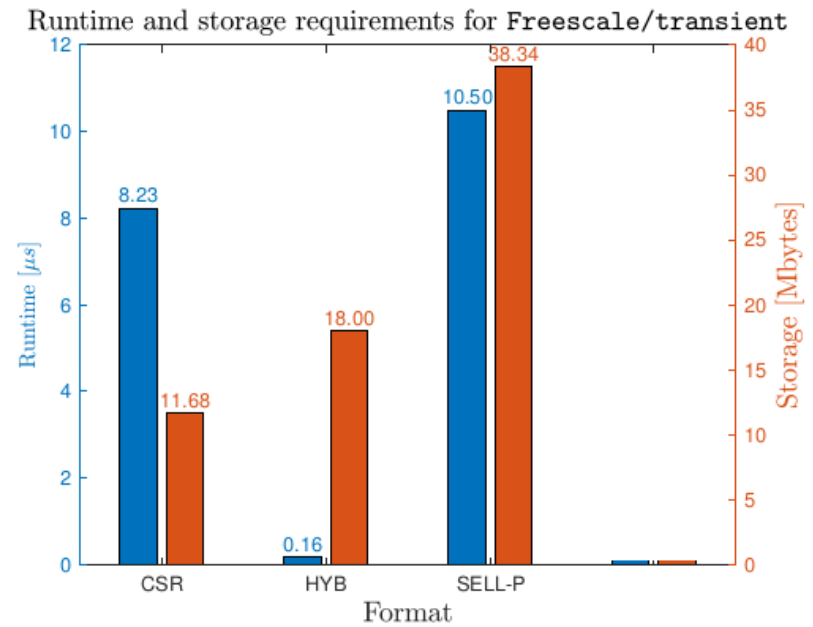
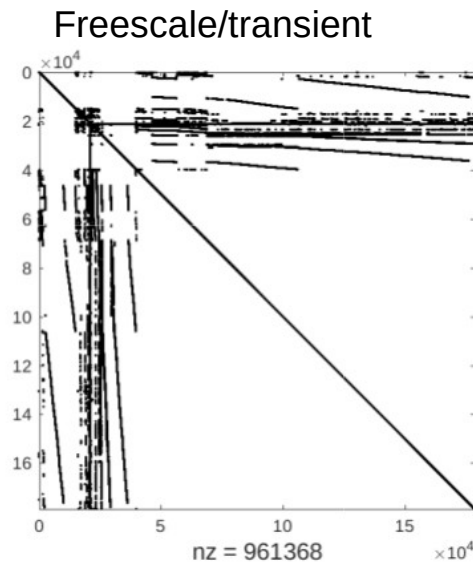
Load imbalance!

Example



Example

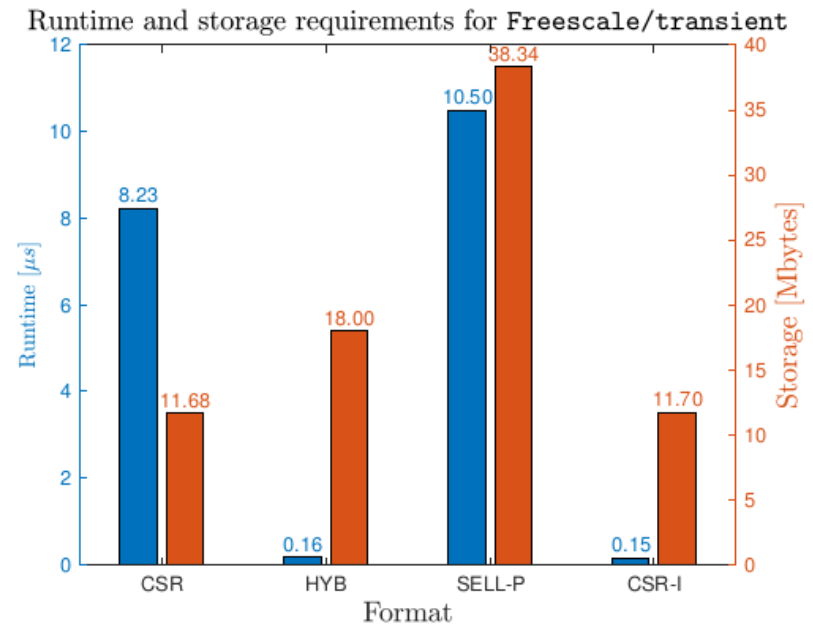
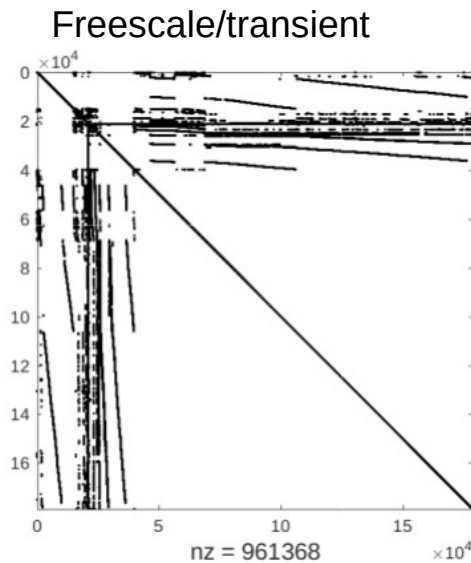
* GTX 1080



Can we do better than HYB using CSR?

Example

* GTX 1080



Can we do better than HYB using CSR?

55x speedup

YES!

Publish a paper about it?

You can...

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 publish an SpMV paper

<irony>

- Think of a “new” algorithm / format for sparse matrix-vector product.
 - Does not have to be great, can do stuff in software that the hardware will already do automatically, or not even give correct results (no one checks).

</irony>

Copyright notice: the “<irony>” tag was shamelessly stolen from Georg Hager’s “Thirteen modern ways to fool the masses with performance results on parallel computers” talk, see <https://blogs.fau.de/hager/archives/category/fooling-the-masses>

How to publish an SpMV paper

<irony>

- Think of a “new” algorithm / format for sparse matrix-vector product.
 - Does not have to be great, can do stuff in software that the hardware will already do automatically, or not even give correct results (no one checks).
- Find 10 - 20 matrices from the SuiteSparse collection where your algorithm is faster than any other algorithms / formats you compare.
 - Not that difficult, there’s 3000 matrices with different properties, no algorithm handles all the corner cases properly.

</irony>

Copyright notice: the “<irony>” tag was shamelessly stolen from Georg Hager’s “Thirteen modern ways to fool the masses with performance results on parallel computers” talk, see <https://blogs.fau.de/hager/archives/category/fooling-the-masses>

How to publish an SpMV paper

<irony>

- Think of a “new” algorithm / format for sparse matrix-vector product.
 - Does not have to be great, can do stuff in software that the hardware will already do automatically, or not even give correct results (no one checks).
- Find 10 - 20 matrices from the SuiteSparse collection where your algorithm is faster than any other algorithms / formats you compare.
 - Not that difficult, there’s 3000 matrices with different properties, no algorithm handles all the corner cases properly.
- Write a paper claiming that your algorithm is “on average 50% faster than the competitors”, on a “representative” subset.
- Send it to a conference / journal and hope the reviewers do not know a lot about SpMV (most likely true).

</irony>

Copyright notice: the “<irony>” tag was shamelessly stolen from Georg Hager’s “Thirteen modern ways to fool the masses with performance results on parallel computers” talk, see <https://blogs.fau.de/hager/archives/category/fooling-the-masses>

How to publish an SpMV paper

<irony>

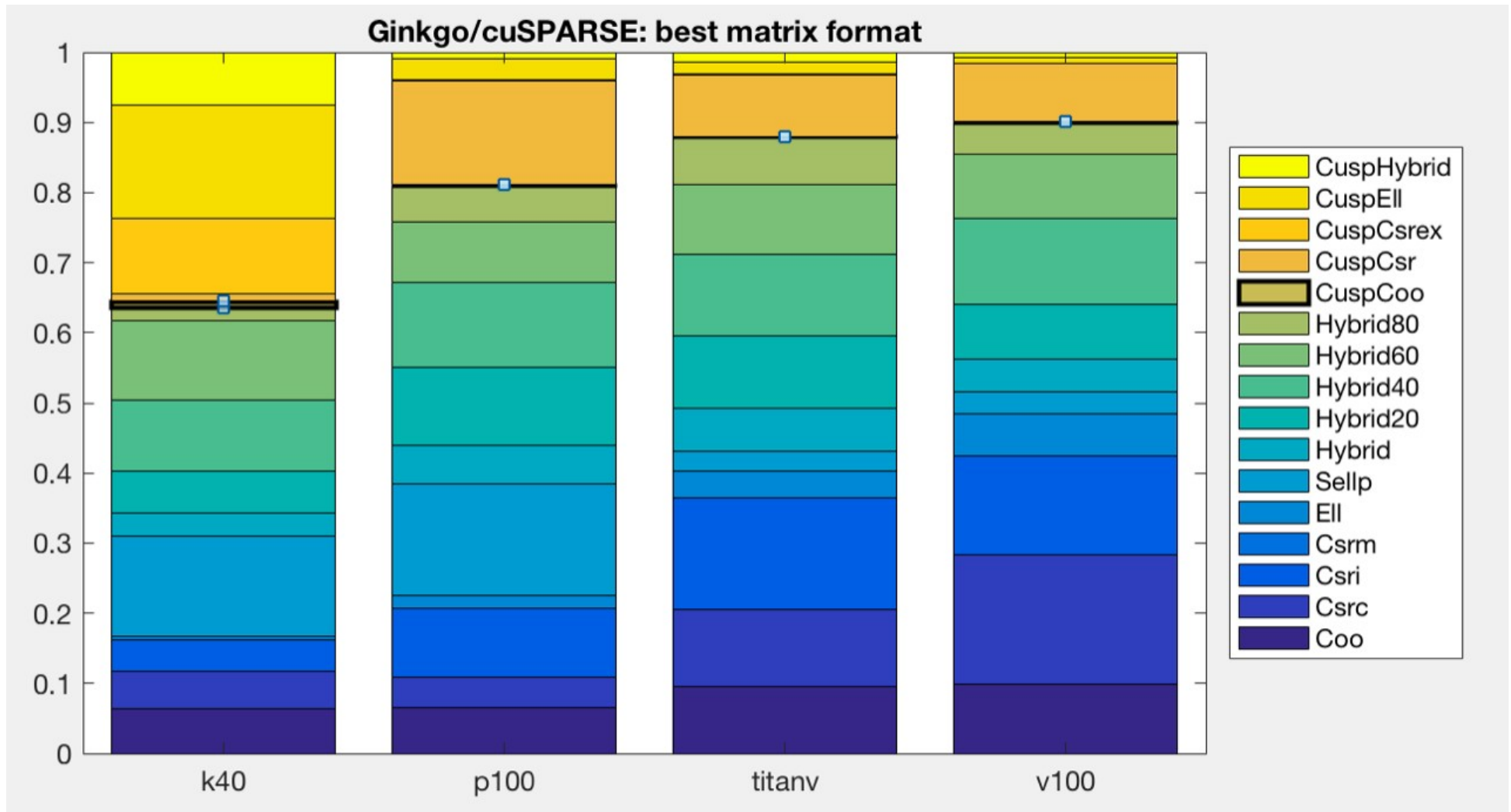
- Think of a “new” algorithm / format for sparse matrix-vector product.
 - Does not have to be great, can do stuff in software that the hardware will already do automatically, or not even give correct results (no one checks).
- Find 10 - 20 matrices from the SuiteSparse collection where your algorithm is faster than any other algorithms / formats you compare.
 - Not that difficult, there’s 3000 matrices with different properties, no algorithm handles all the corner cases properly.
- Write a paper claiming that your algorithm is “on average 50% faster than the competitors”, on a “representative” subset.
- Send it to a conference / journal and hope the reviewers do not know a lot about SpMV (most likely true).
- **Victory!** Think of another format and repeat.

</irony>

Copyright notice: the “<irony>” tag was shamelessly stolen from Georg Hager’s “Thirteen modern ways to fool the masses with performance results on parallel computers” talk, see <https://blogs.fau.de/hager/archives/category/fooling-the-masses>

In the real world...

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



Can we figure out which format is going to give best performance for a given problem?

Maybe...

Choosing the winner a priori

CSR-I designed for irregular patterns

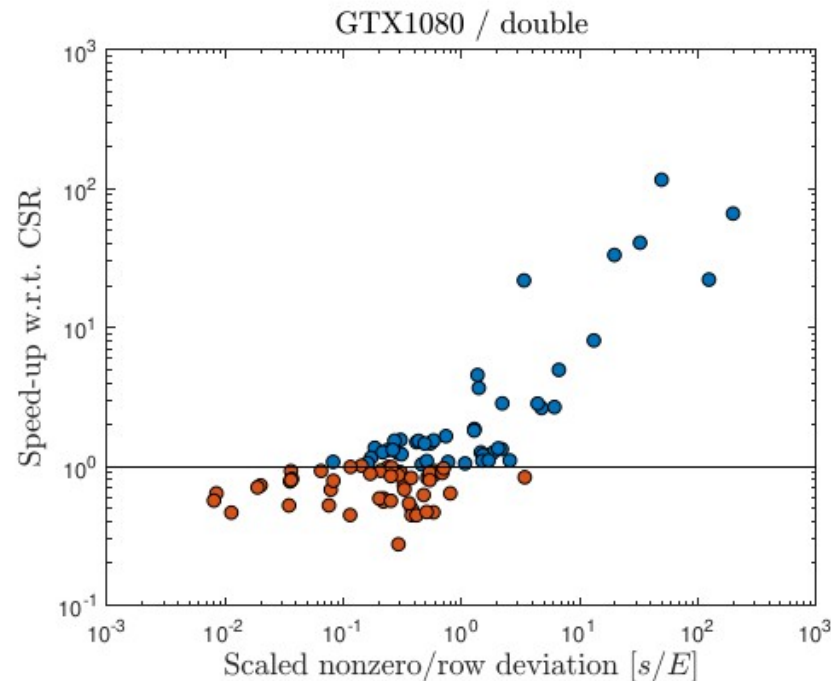
How to measure irregularity?

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

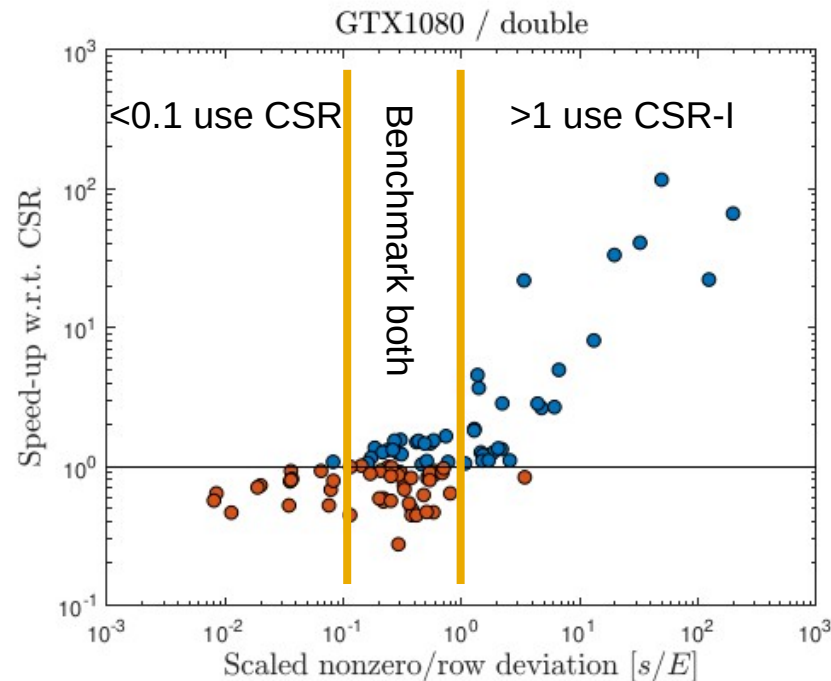
How to measure irregularity?

Deviation of row lengths from the mean.

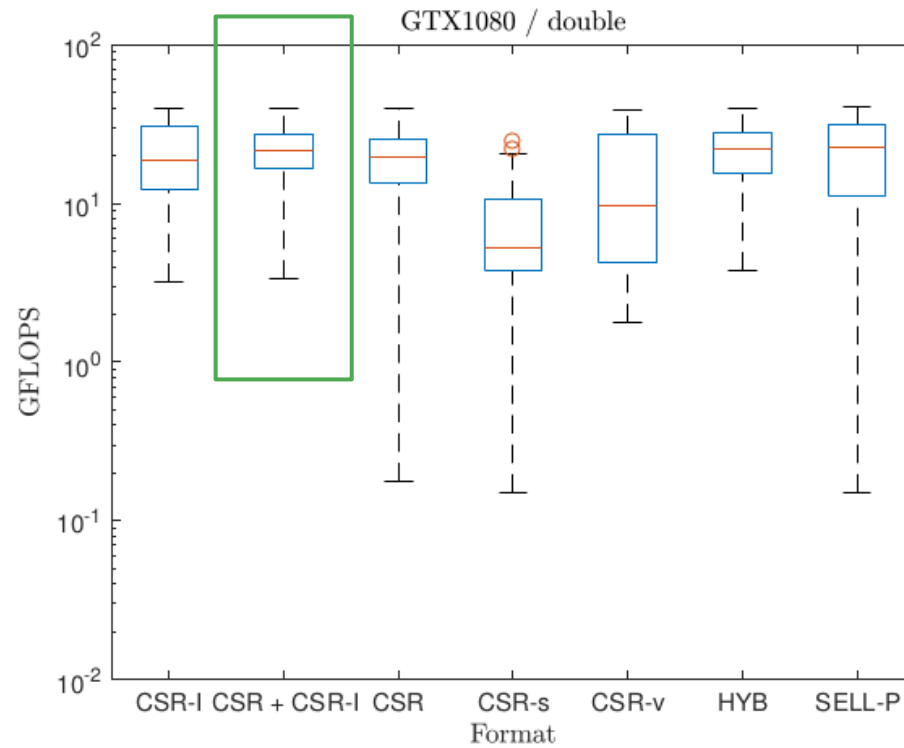
Is “5” regular or irregular?

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

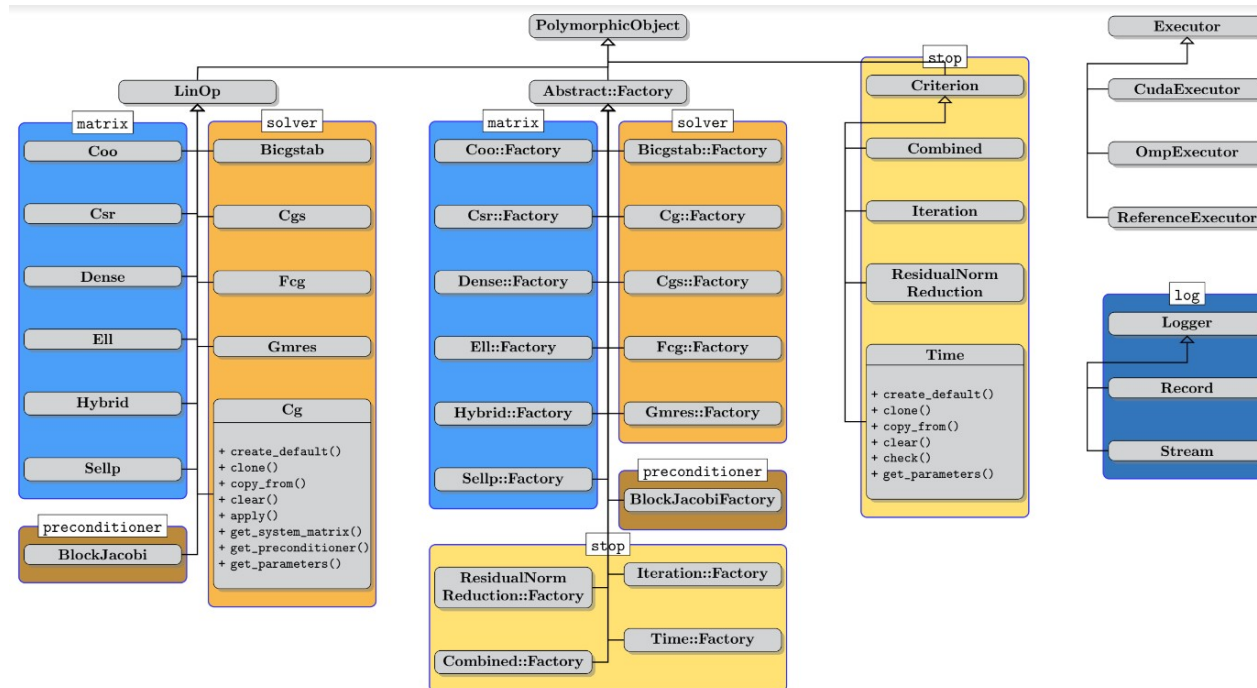
Scatter plot of speedup vs normalized std. dev.



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?