ScalA16: Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems
Salt Lake City, 11/13/2016

Batched Generation of Incomplete Sparse Approximate Inverses on GPUs

Hartwig Anzt, Edmond Chow, Thomas Huckle, Jack Dongarra

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

SC16 Salt Lake City, Utah | hpc matters.

http://www.icl.utk.edu/~hanzt/talks/ISAI.pdf

Incomplete Sparse Approximate Inverse (ISAI) Preconditioner

Goal: Find solution to sparse linear problem $Ax = b,\, A \in \mathbb{R}^{n \times n}$.

Compute factorization

$$(A = LU)_{\mathcal{S}} \text{ for some sparsity pattern } \mathcal{S} \begin{cases} \mathcal{S} = \mathbb{R}^{n \times n} & \text{exact fact.} \\ \mathcal{S} = spy(A) & \text{ILU(0)} \end{cases}$$

Incomplete factorizations attractive for preconditioning iterative solvers.

Preconditioner application involves solving triangular systems $Ly = z\,,\, Ux = y$.

Incomplete Sparse Approximate Inverse (ISAI) Preconditioner

Goal: Find solution to sparse linear problem $Ax = b, A \in \mathbb{R}^{n \times n}$.

Compute factorization

$$(A = LU)_{\mathcal{S}} \text{ for some sparsity pattern } \mathcal{S} \begin{cases} \mathcal{S} = \mathbb{R}^{n \times n} & \text{exact fact.} \\ \mathcal{S} = spy(A) & \text{ILU(0)} \end{cases}$$

Incomplete factorizations attractive for preconditioning iterative solvers.

Preconditioner application involves solving triangular systems $Ly = z\,,\ Ux = y$.

- Exact triangular solves
  - Inherently sequential, level scheduling often provides little parallelism.

Incomplete Sparse Approximate Inverse (ISAI) Preconditioner

Goal: Find solution to sparse linear problem $Ax = b,\ A \in \mathbb{R}^{n \times n}$.

Compute factorization

$$(A = LU)_{\mathcal{S}} \text{ for some sparsity pattern } \mathcal{S} \begin{cases} \mathcal{S} = \mathbb{R}^{n \times n} & \text{exact fact.} \\ \mathcal{S} = spy(A) & \text{ILU(0)} \end{cases}$$

Incomplete factorizations attractive for preconditioning iterative solvers.

Preconditioner application involves solving triangular systems $Ly = z\,,\ Ux = y$.

- Exact triangular solves
  - Inherently sequential, level scheduling often provides little parallelism.
- Replace with approximate triangular solve
  - Relaxation steps like (Block) Jacobi iterations.

Incomplete Sparse Approximate Inverse (ISAI) Preconditioner

Goal: Find solution to sparse linear problem $Ax = b,\, A \in \mathbb{R}^{n\times n}$.

Compute factorization

$$(A = LU)_{\mathcal{S}} \text{ for some sparsity pattern } \mathcal{S} \begin{cases} \mathcal{S} = \mathbb{R}^{n\times n} & \text{exact fact.} \\ \mathcal{S} = spy(A) & \text{ILU(0)} \end{cases}$$

Incomplete factorizations attractive for preconditioning iterative solvers.

Preconditioner application involves solving triangular systems $Ly = z\,,\, Ux = y$.

- Exact triangular solves
  - Inherently sequential, level scheduling often provides little parallelism.
- Replace with approximate triangular solve
  - Relaxation steps like (Block) Jacobi iterations.
  - Incomplete Sparse Approximate Inverse (ISAI)[1]:

$$(L \cdot M_L = I)_{S^*} \text{ for some sparsity pattern } S^*\text{, e.g. } \begin{array}{ll} S^* = spy(A) & \text{ISAI(1)} \\ S^* = spy(A^2) & \text{ISAI(2)} \\ S^* = spy(A^3) & \text{ISAI(3)} \\ S^* = JAC(4) & \end{array}$$

[1]Huckle, Anzt, Dongarra **"Parallel Preconditioning"**. *In: SIAM PP 2016.*

Generating ISAI for triangular factor

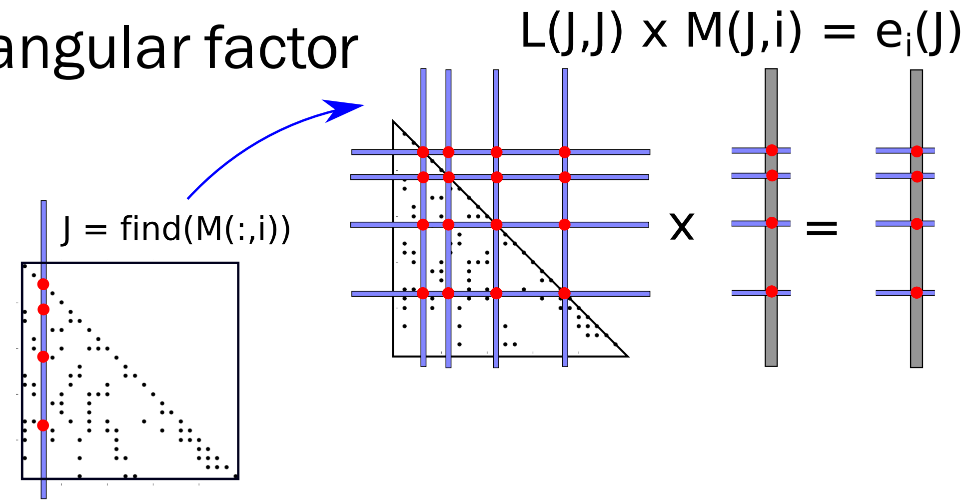
M_L with $(L \cdot M_L = I)_{S^*}$ for $S^* = spy(A)$

$$(L \cdot M_L(:, i) = e_i)_{S^*} \;\; \forall i = 1 \ldots n$$

```
for i=1:n
   J = find(M(:,i));
   generate L(J,J);
   solve L(J,J) M(J,i) = e_i(J);
   insert M(J,i) into M;
end
```

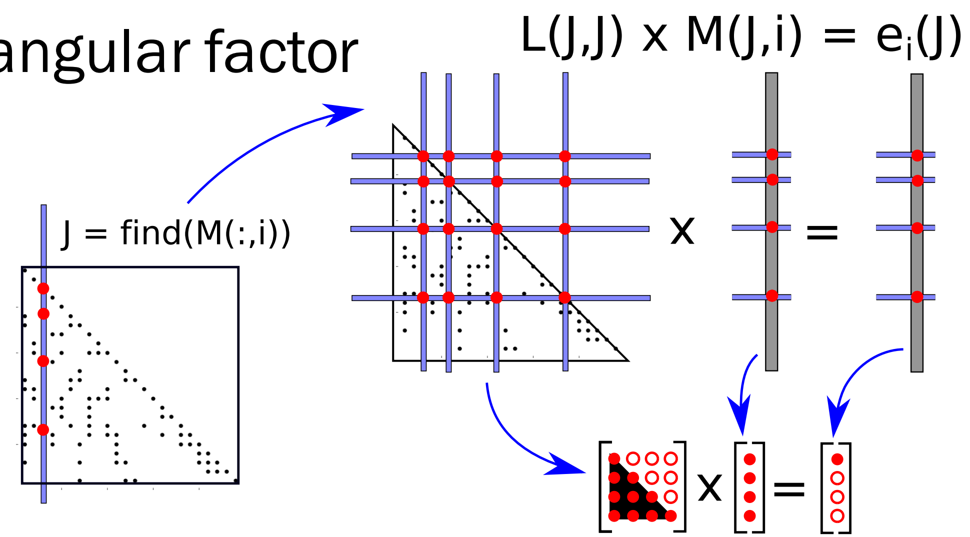
Algorithm composes into solving a set of small triangular systems

Generating ISAI for triangular factor




```
for i=1:n
   J = find(M(:,i));
   generate L(J,J);
   solve L(J,J) M(J,i) = e_i(J);
   insert M(J,i) into M;
end
```

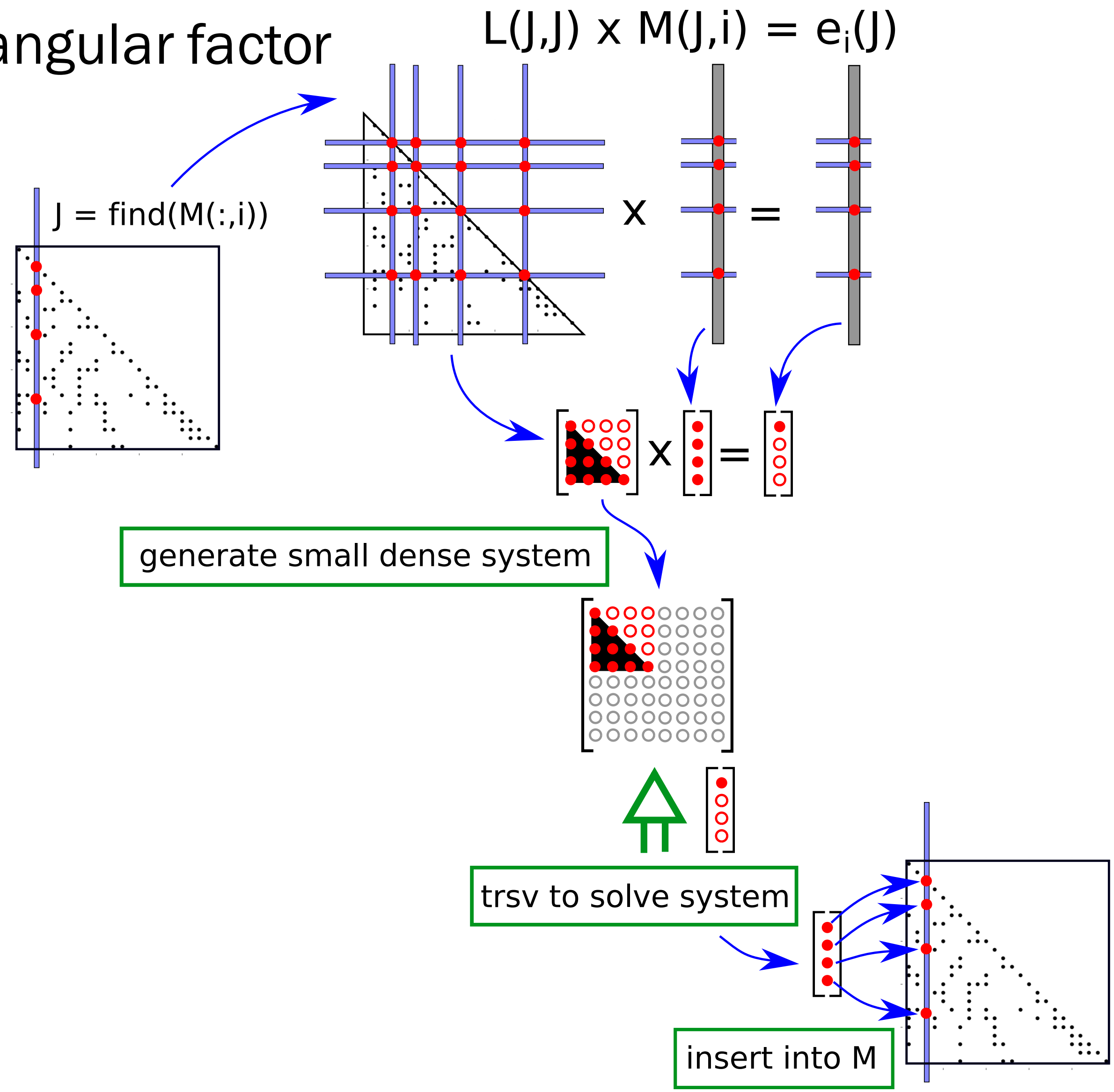
Generating ISAI for triangular factor




```
for i=1:n
   J = find(M(:,i));
   generate L(J,J);
   solve L(J,J) M(J,i) = e_i(J);
   insert M(J,i) into M;
end
```

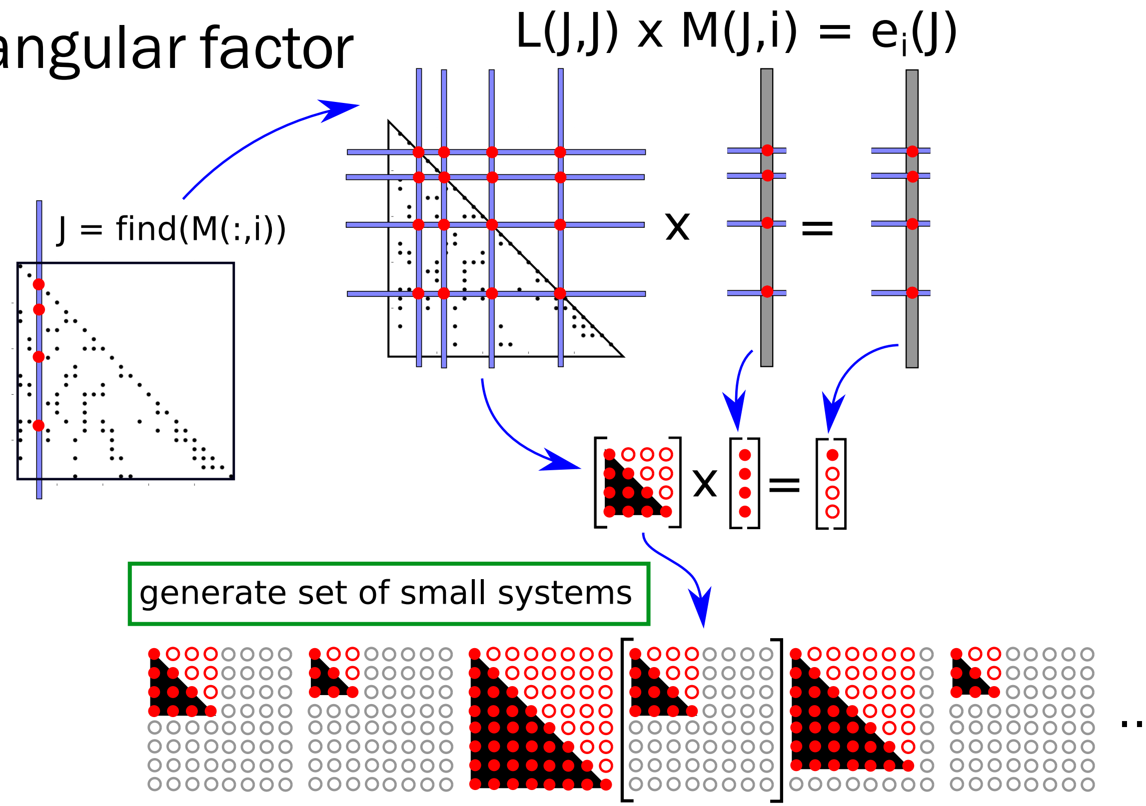
Generating ISAI for triangular factor

```
for i=1:n
   J = find(M(:,i));
   generate L(J,J);
   solve L(J,J) M(J,i) = e_i(J);
   insert M(J,i) into M;
end
```



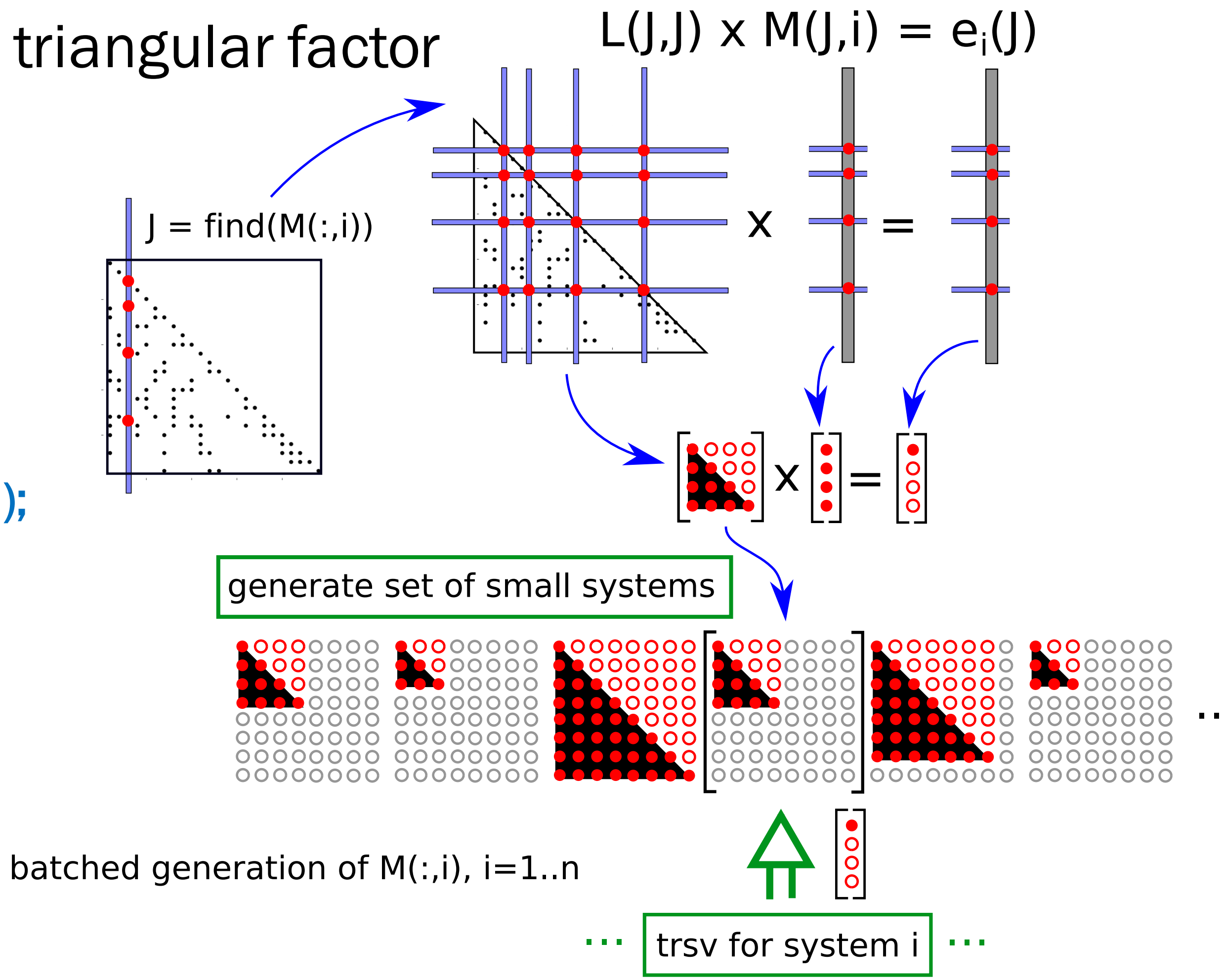
Generating ISAI for triangular factor

```
for i=1:n
   J = find(M(:,i));
   generate L(J,J);
   solve L(J,J) M(J,i) = e_i(J);
   insert M(J,i) into M;
end
```



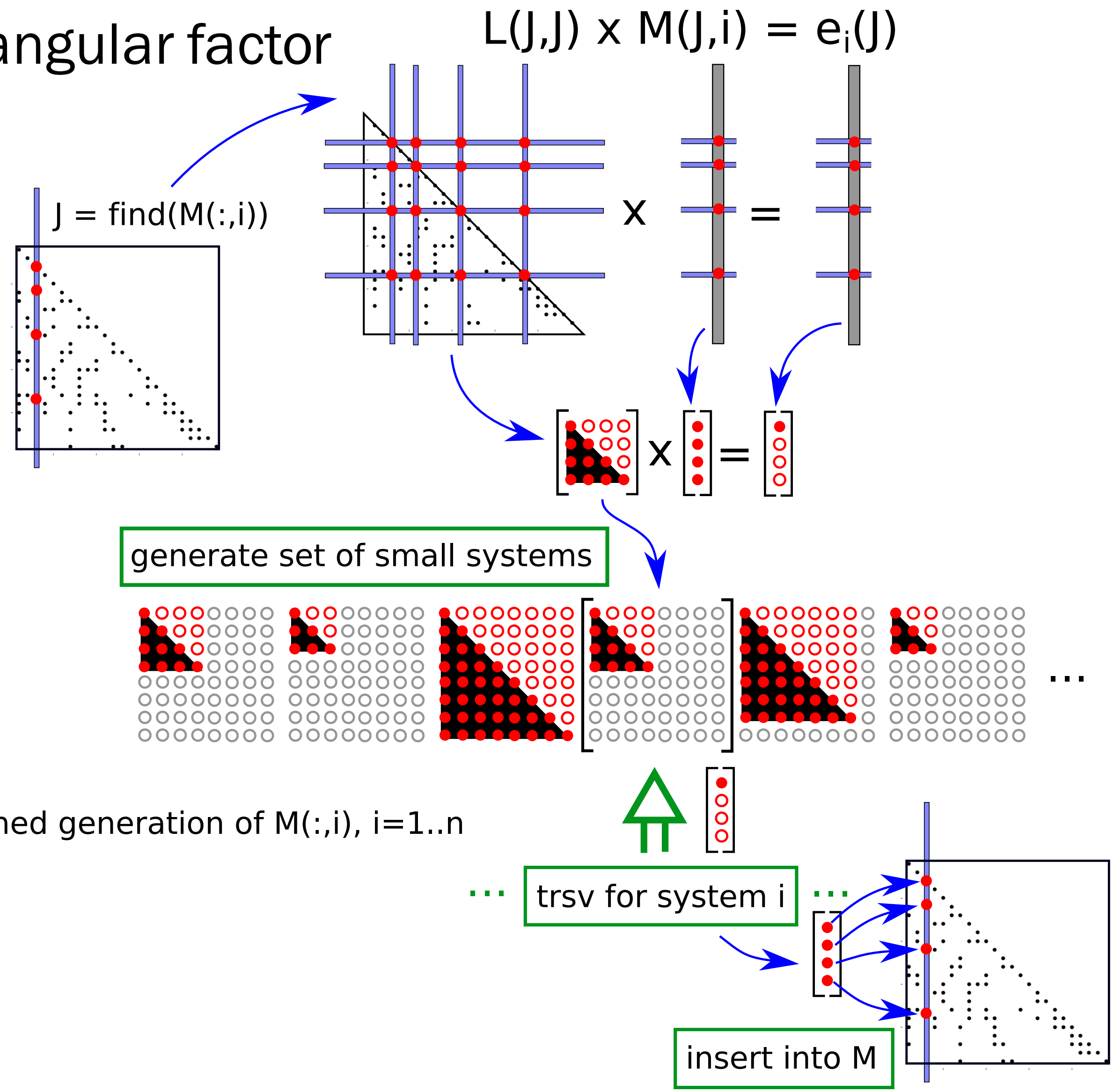
Generating ISAI for triangular factor

```
for i=1:n
   J = find(M(:,i));
   generate L(J,J);
   solve L(J,J) M(J,i) = e_i(J);
   insert M(J,i) into M;
end
```



Generating ISAI for triangular factor

```
for i=1:n
   J = find(M(:,i));
   generate L(J,J);
   solve L(J,J) M(J,i) = e_i(J);
   insert M(J,i) into M;
end
```



batched generation of M(:,i), i=1..n

Generating ISAI for triangular factor

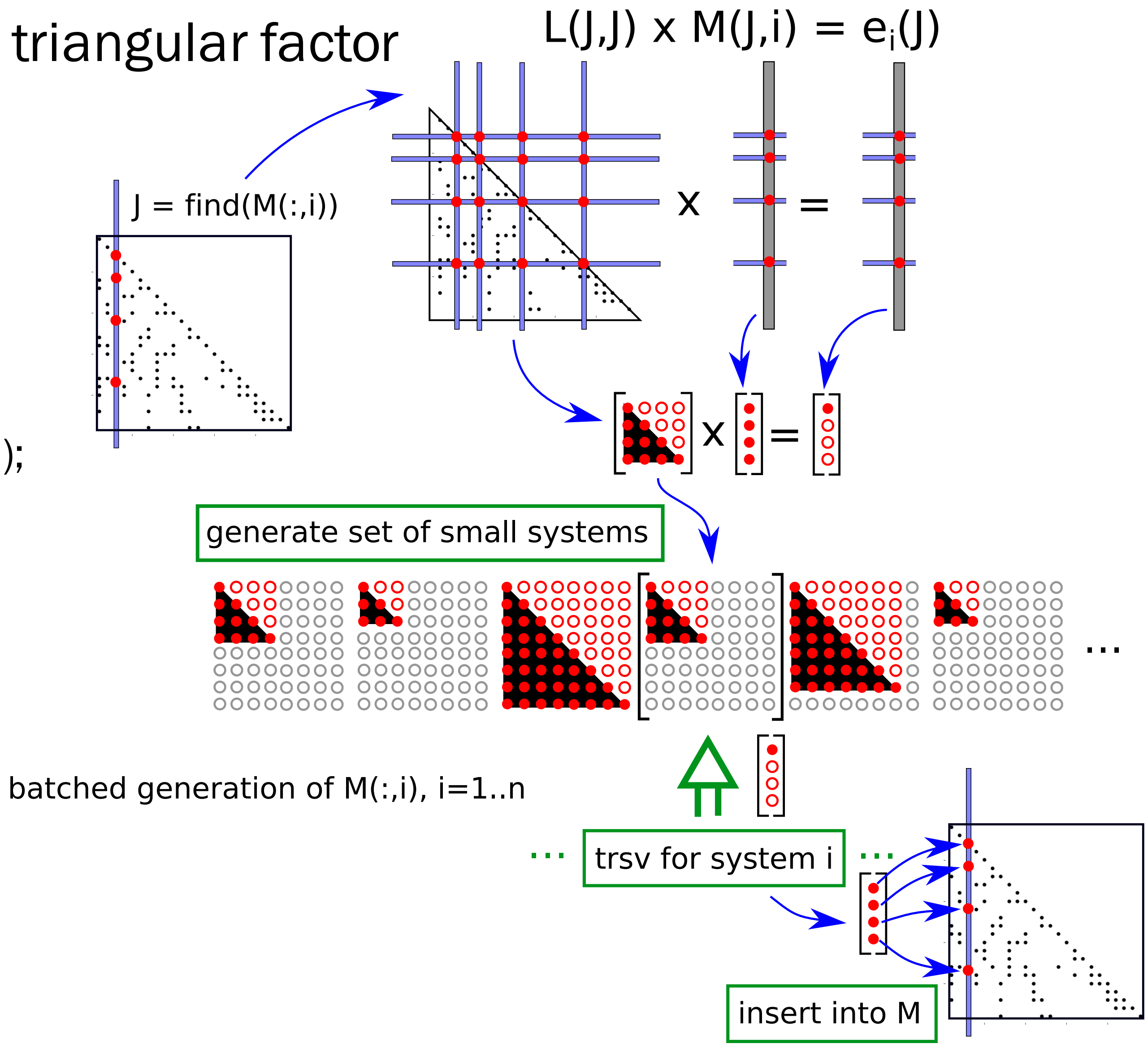
```
for i=1:n
   J = find(M(:,i));
   generate L(J,J);
   solve L(J,J) M(J,i) = e_i(J);
   insert M(J,i) into M;
end
```

Four Batched Routines:

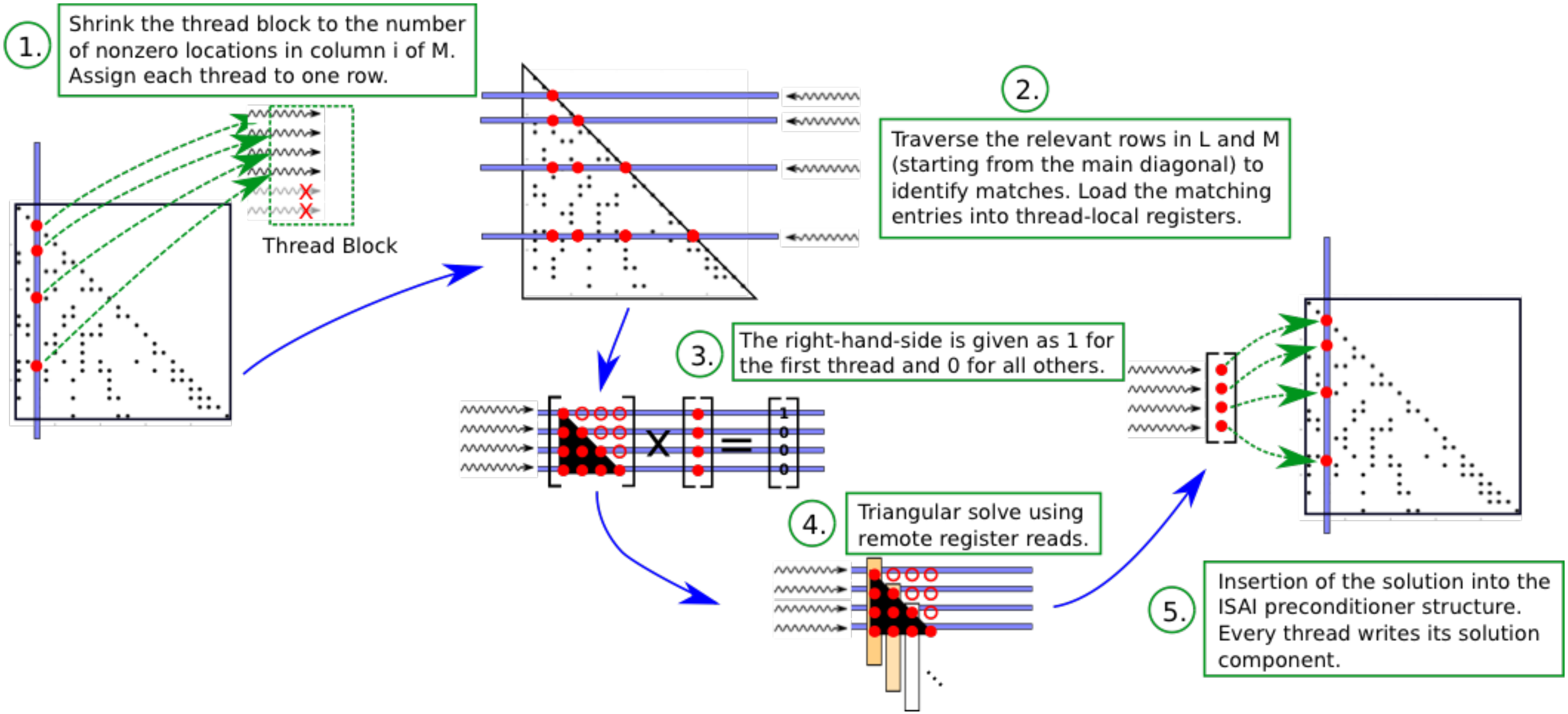
- Find the locations in each row
  - store size information for small tri-systems
  - store nonzero-locations to find matches
- Generate batch of small triangular systems
  - different sizes in uniformly-sized blocks
- Batched trsv
  - different sizes
  - non-coalescent in memory (uniform blocks)
  - use kernel-switch for hard-coded sizes
- Batched re-insertion into sparse ISAI matrix
  - non-coalescent reads/writes

Generating ISAI for triangular factor

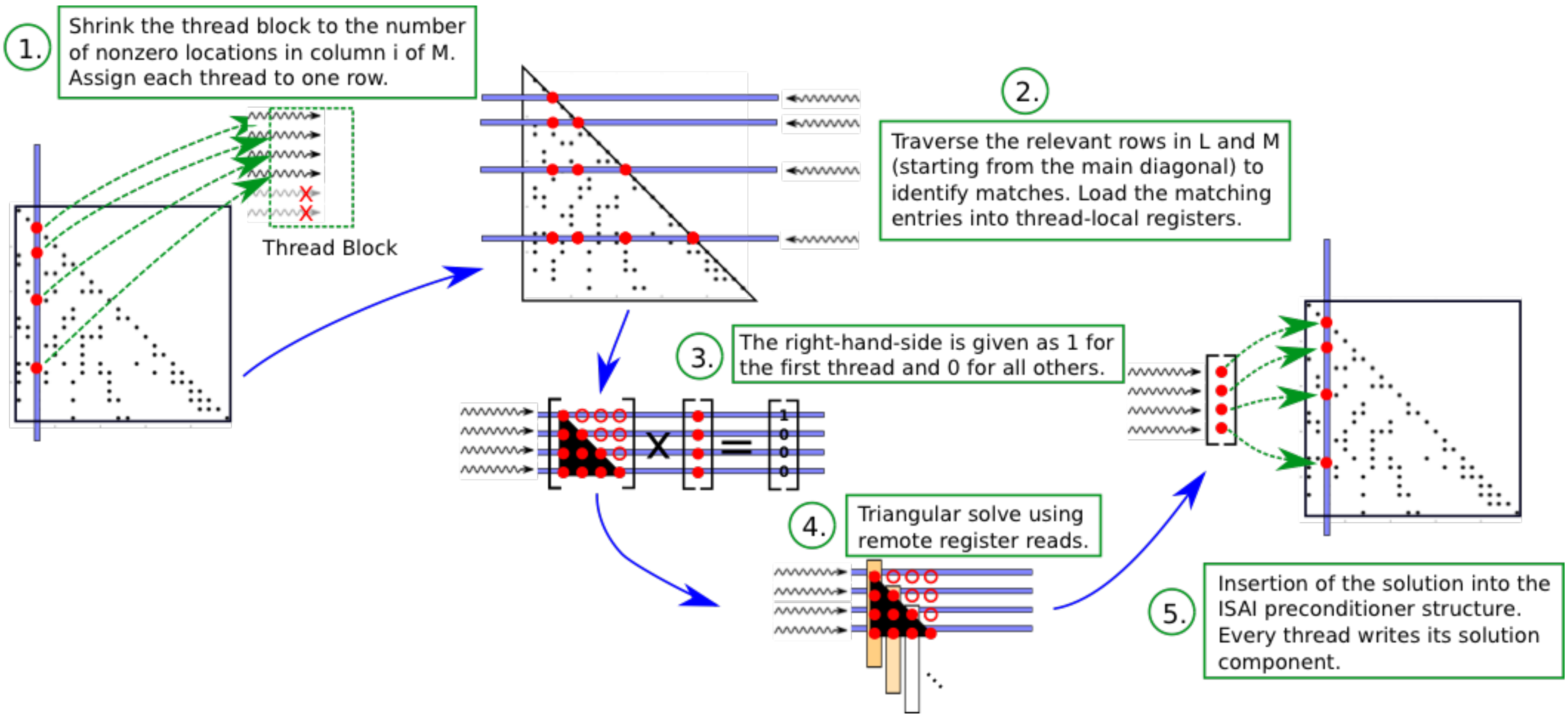
```
for i=1:n
   J = find(M(:,i));
   generate L(J,J);
   solve L(J,J) M(J,i) = e_i(J);
   insert M(J,i) into M;
end
```



Generating ISAI using one batched routine



Generating ISAI using one batched routine

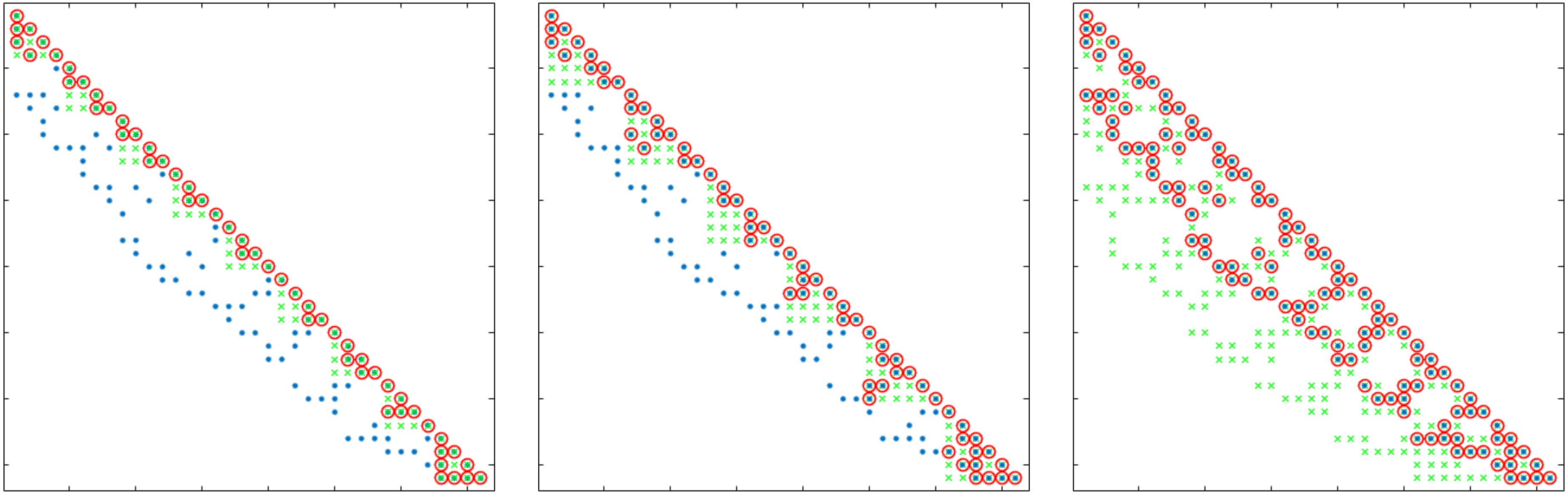


- Generate triangular system in registers only & batched triangular solve in registers[2].

[2]Kurzak et al. **"Implementation and Tuning of Batched Cholesky Factorization and Solve on NVIDIA GPUs".** *TPDS, 2015*.

Mapping sparsity pattern

Nonzero locations in L (blue) in M_L (green), and matching locations (red).



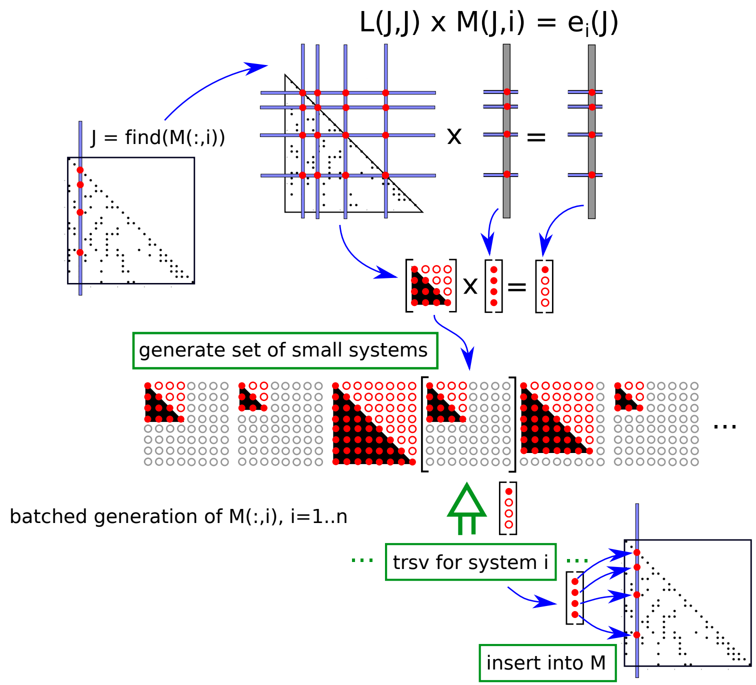
$S^* = JAC(2)$ $S^* = JAC(6)$ $S^* = spy(A^2)$

- Generate triangular system in registers only & batched triangular solve in registers[2].
- Identify matching locations by traversing triangular matrices from the diagonal.

Two strategies for generating ISAI

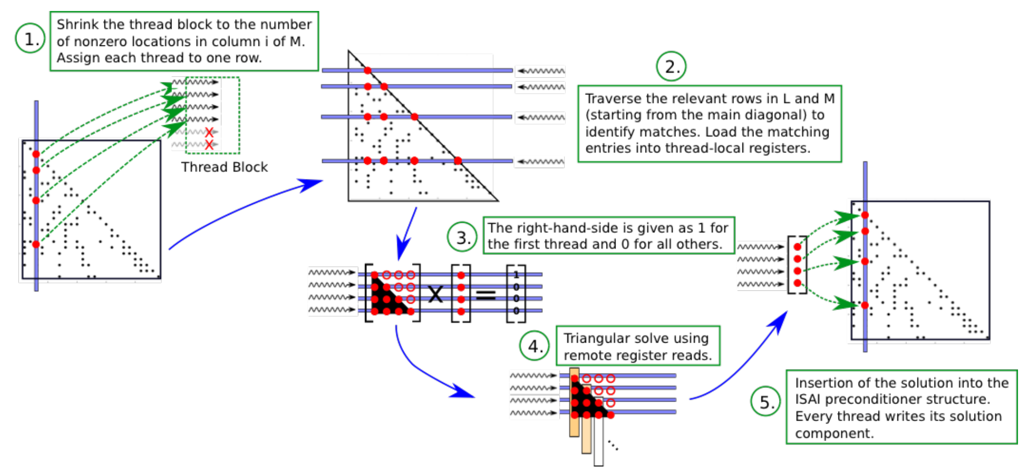
Combination of 4 batched routines

- Batch of systems in main memory
- Coalesced memory access



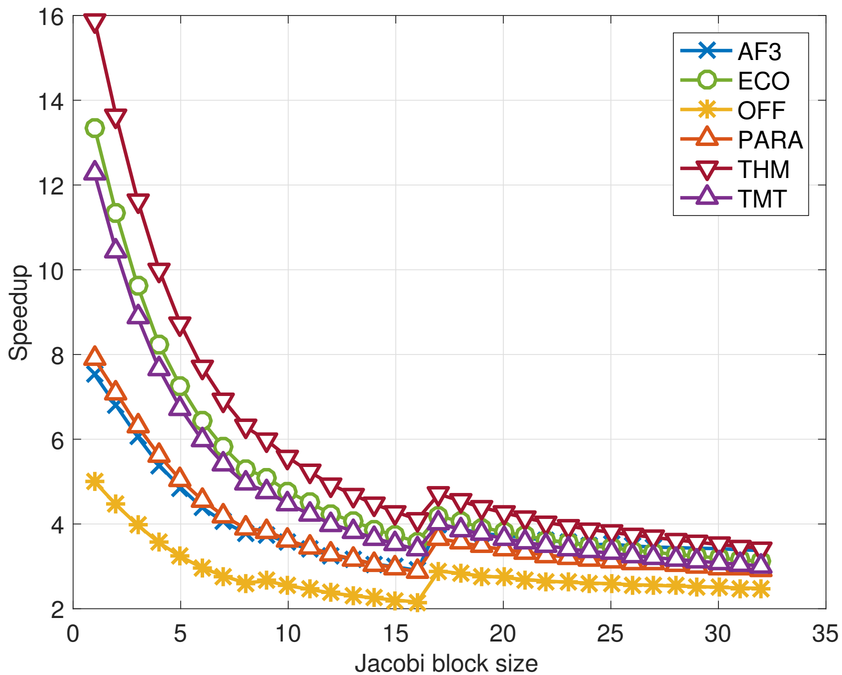
One batched routine

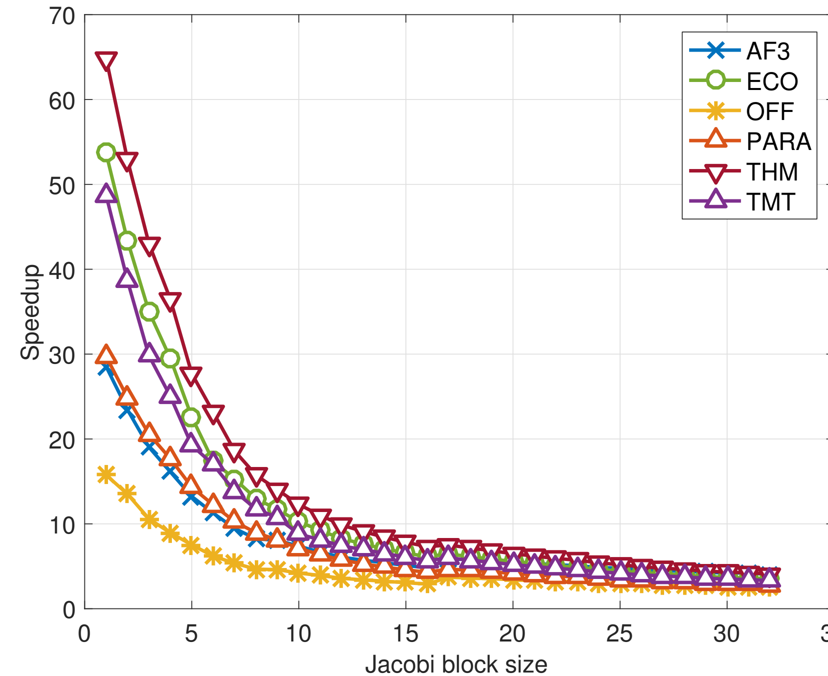
- Linear system in registers
- __shfl() for communication



Performance comparison of ISAI generation

Speedup of batched routine vs. sequence of batched routines for block Jacobi pattern.





Nvidia K40 GPU

- 1.4 TF DP, 280 GB/s (**4.86 : 1**)
- Local memory as L1 cache/shared memory

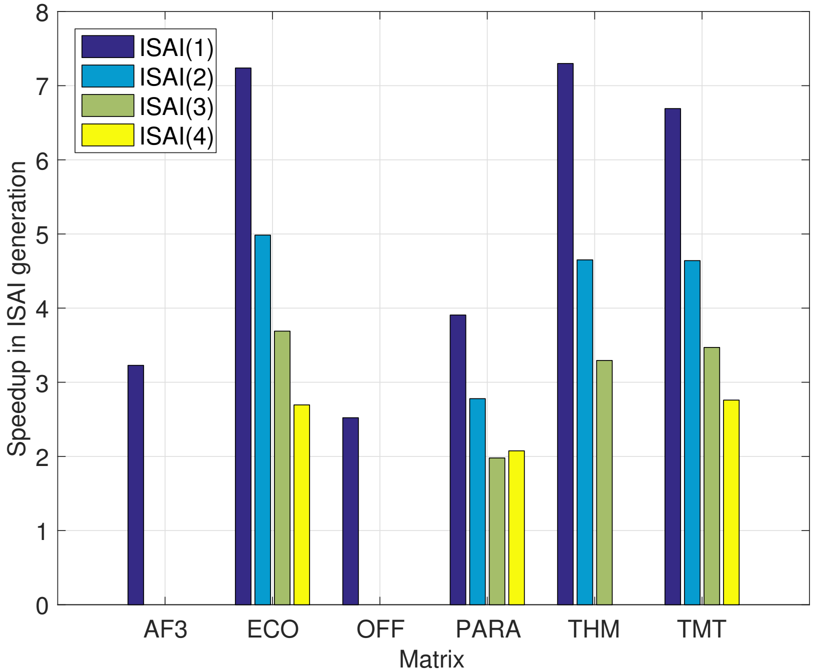
Nvidia P100 GPU

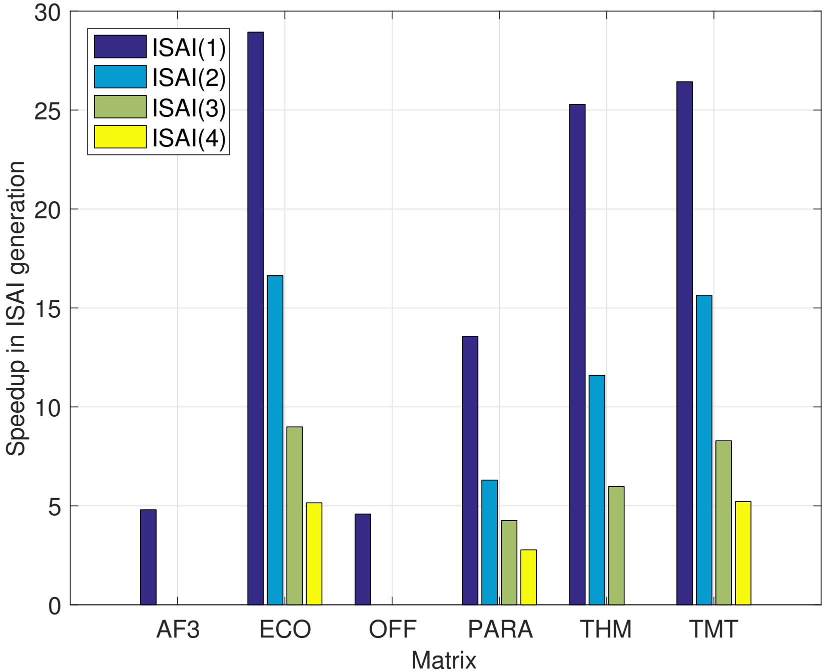
- 5.3 TF DP, 720 GB/s (**7.36 : 1**)
- No cache, only registers and shared memory

(MAGMA 2.1.0, ILU taken from cuSPARSE 8.0)

Performance comparison of ISAI generation

Speedup of batched routine vs. sequence of batched routines for ISAI pattern.





Nvidia K40 GPU

- 1.4 TF DP, 280 GB/s (**4.86 : 1**)
- Local memory as L1 cache/shared memory

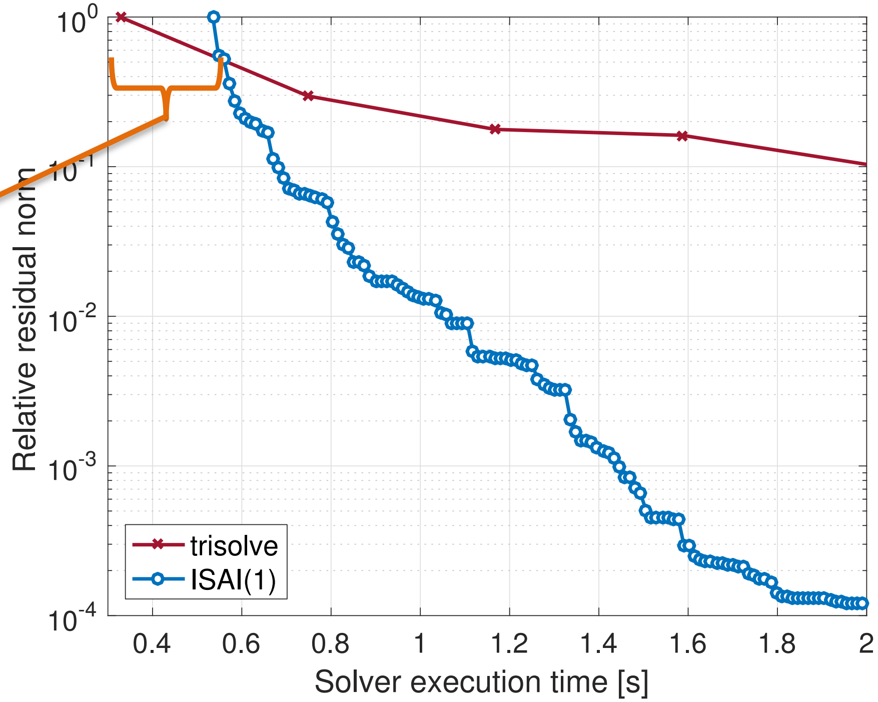
Nvidia P100 GPU

- 5.3 TF DP, 720 GB/s (**7.36 : 1**)
- No cache, only registers and shared memory

(MAGMA 2.1.0, ILU taken from cuSPARSE 8.0)

Incomplete Sparse Approximate Inverse in Iterative Solvers

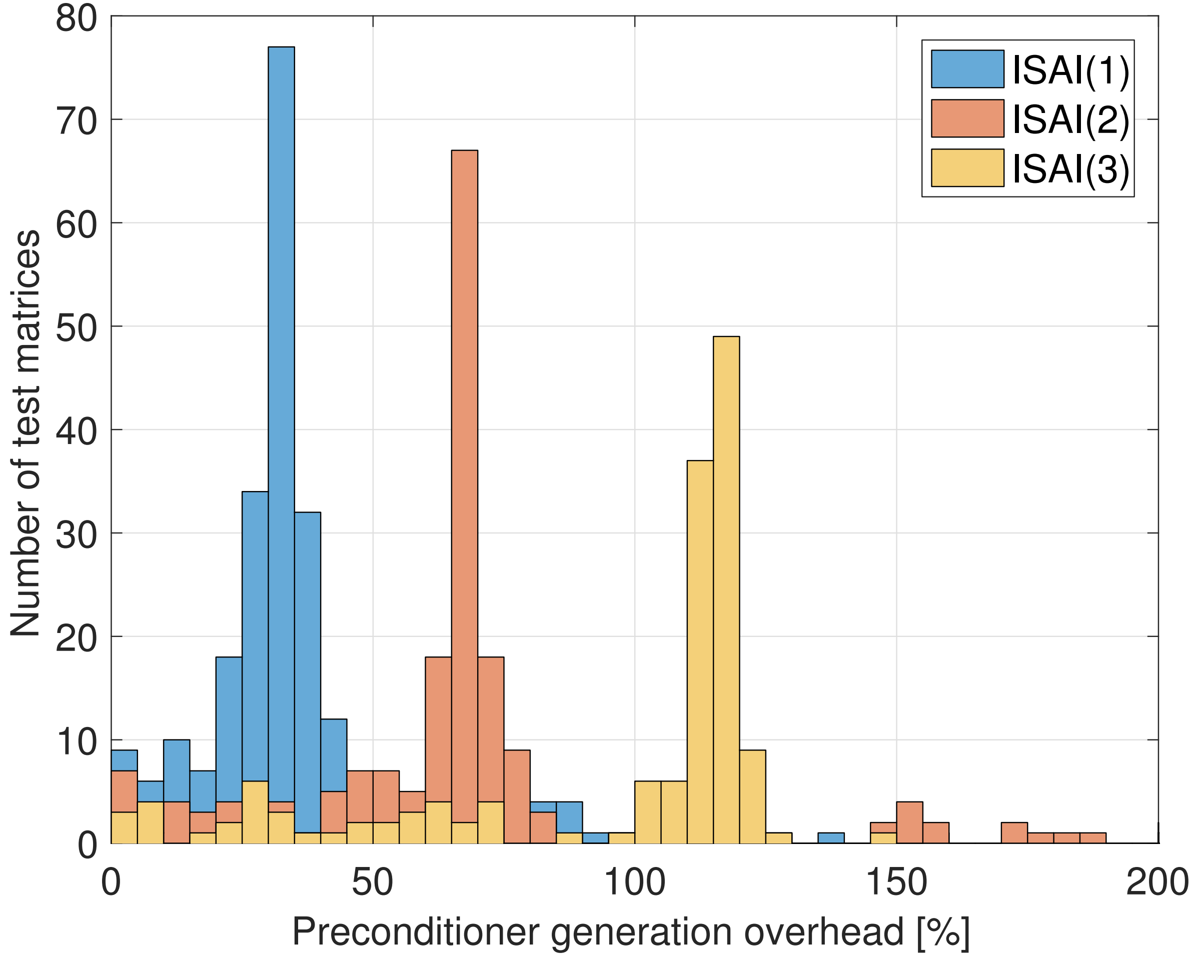
Overhead for ISAI generation.



Matrix	Preconditioner setup [s]			Preconditioner application [s]			IDR(4) Iterations			Solver execution [s]		
	ILU(0)	ISAI(1)	overhead	trisol	ISAI(1)	speedup	trisol	ISAI(1)	overhead	trisol	ISAI(1)	speedup
AF3	1.3672	0.7279	53.24%	0.6909	0.0129	53.46	329	1588	382.67%	234.2021	49.4370	**4.74**
ECO	0.3122	0.2889	92.56%	0.2116	0.0064	33.16	954	2933	207.44%	216.2284	65.1396	**3.32**
OFF	0.3276	0.2076	63.35%	0.4103	0.0044	92.74	148	440	197.30%	62.2157	5.8897	**10.56**
PARA	0.4857	0.1979	40.77%	0.2687	0.0045	60.09	139	730	425.18%	39.0847	11.0221	**3.55**
THM	0.8296	0.4543	54.77%	0.8084	0.0132	61.27	873	1755	101.03%	722.9693	60.0312	**12.04**
TMT	0.3009	0.2522	83.83%	0.1661	0.0060	27.90	755	1881	149.14%	136.0090	38.9953	**3.49**

(Nvidia K40 GPU, MAGMA 2.1.0, ILU taken from cuSPARSE 8.0)

Performance of Batched ISAI implementation



460 UFSMC matrices

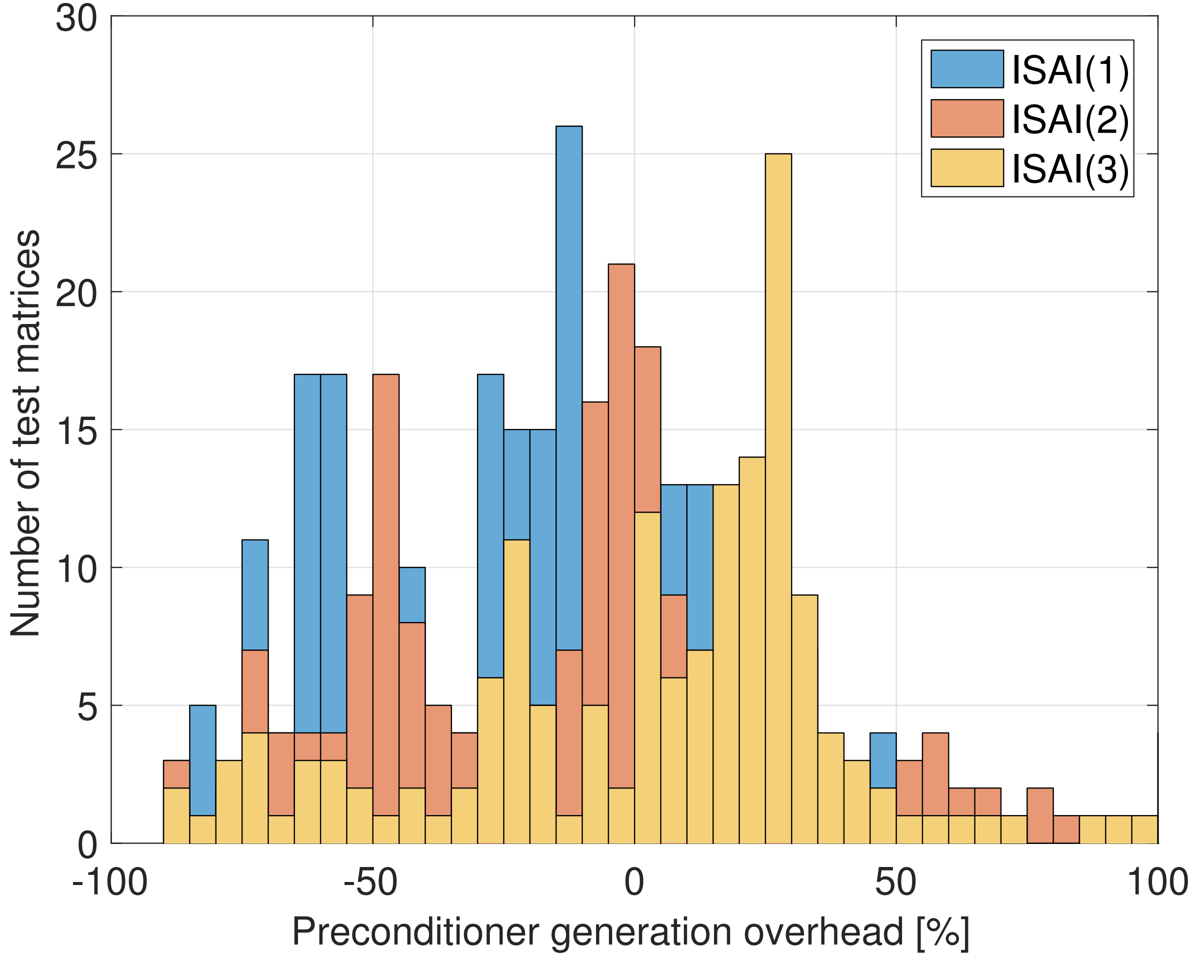
ISAI(1)
251 systems
overhead 37.8%

ISAI(2)
192 systems
overhead 77.4%

ISAI(3)
164 systems
overhead 113.9%

(Nvidia K40 GPU, MAGMA 2.1.0, ILU taken from cuSPARSE 8.0)

Performance of Batched ISAI implementation



458 UFSMC matrices

ISAI(1)
- 251 systems
- overhead -- 20.9%

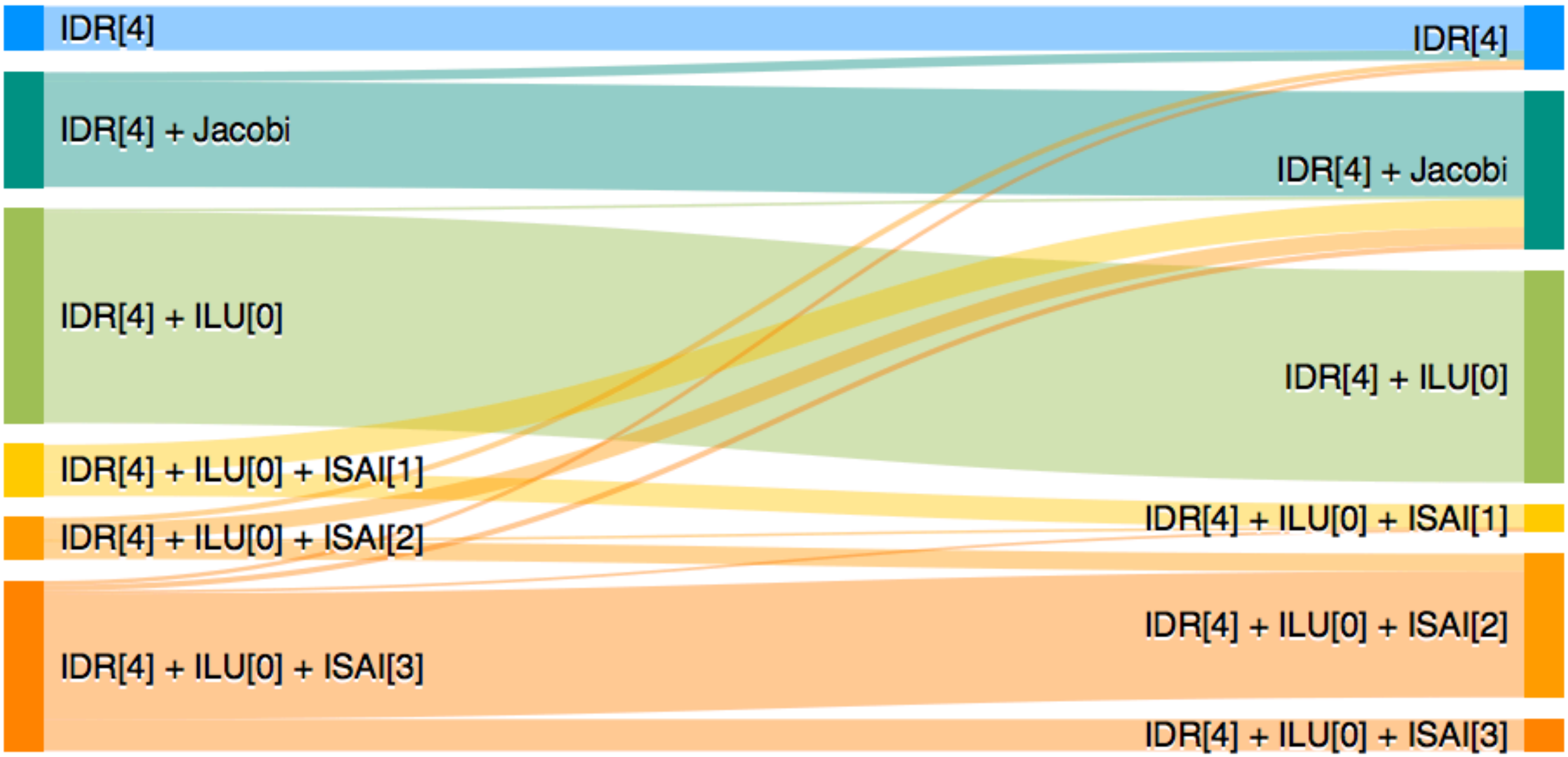
ISAI(2)
- 192 systems
- overhead -- 13.9%

ISAI(3)
- 164 systems
- overhead -- 6.6%

(Nvidia K40 GPU, MAGMA 2.1.0, ILU taken from cuSPARSE 8.0)

Performance of Batched ISAI implementation

ISAI generation successful for 251 of 460 UFSMC matrices



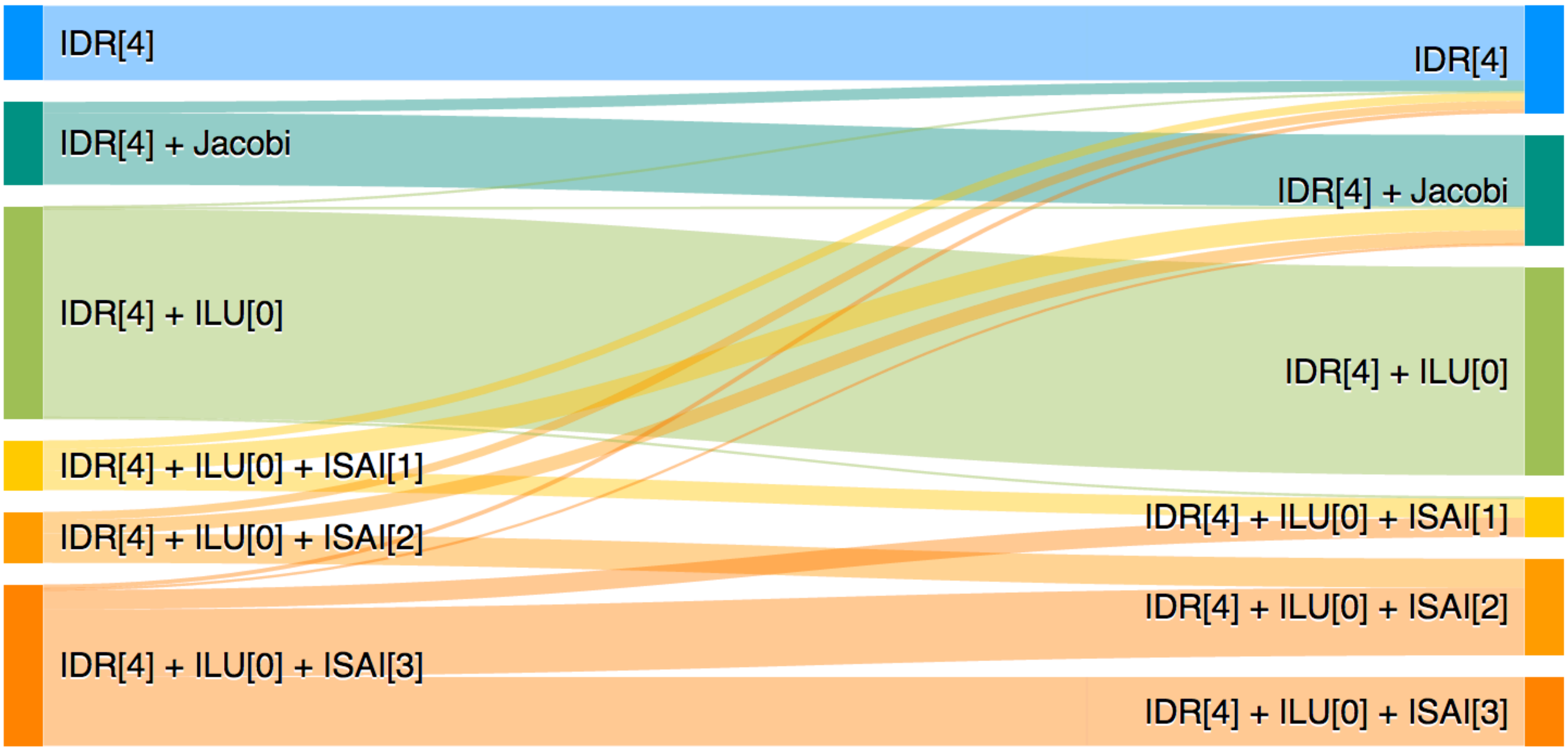
fastest solver
without prec. setup

fastest solver
with prec. setup

(Nvidia K40 GPU, MAGMA 2.1.0, ILU taken from cuSPARSE 8.0)

Performance of Batched ISAI implementation

ISAI generation successful for 251 of 460 UFSMC matrices



fastest solver without prec. setup

fastest solver with prec. setup

(Nvidia KP100 GPU, MAGMA 2.1.0, ILU taken from cuSPARSE 8.0) http://icl.cs.utk.edu/magma/

All functionalities are included in the MAGMA 2.2.0 release

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

This research is based on a cooperation between Hartwig Anzt (University of Tennessee), Edmond Chow (Georgia Tech), and Thomas Huckle (TU Munich), and partly funded by the Department of Energy.

http://www.icl.utk.edu/~hanzt/talks/ISAI.pdf