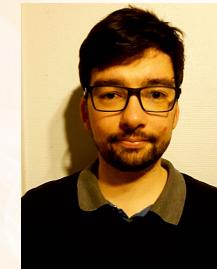


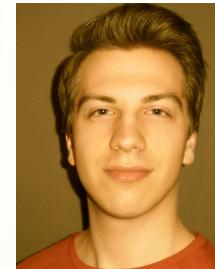
# Algorithm Design in the Advent of Exascale Computing

4th International Symposium on Research and Education of Computational Science (RECS)  
University of Tokyo, October 2<sup>nd</sup>, 2019

Hartwig Anzt, Terry Cojean, Goran Flegar, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel  
Steinbuch Centre for Computing (SCC)



Terry Cojean



Thomas  
Grützmacher



Pratik Nayak



Tobias Ribizel

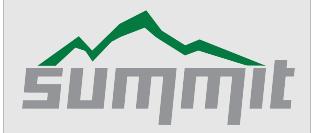


Mike Tsai

# Where do we stand?



LEADERSHIP  
COMPUTING  
FACILITY



- Node: 2 IBM POWER9 + 6 NVIDIA V100 GPUs
- 4,608 nodes, 9,216 IBM Power9 CPUs
- 27,648 V100 GPUs (**8 TFLOPs / GPU**)
- Peak performance of **200 Pflop/s** for modeling & simulation
- Peak performance of **3.3 Eflop/s** ( $10^{18}$ ) for 16 bit floating point used in data analytics and artificial intelligence



Copyright@ORNL

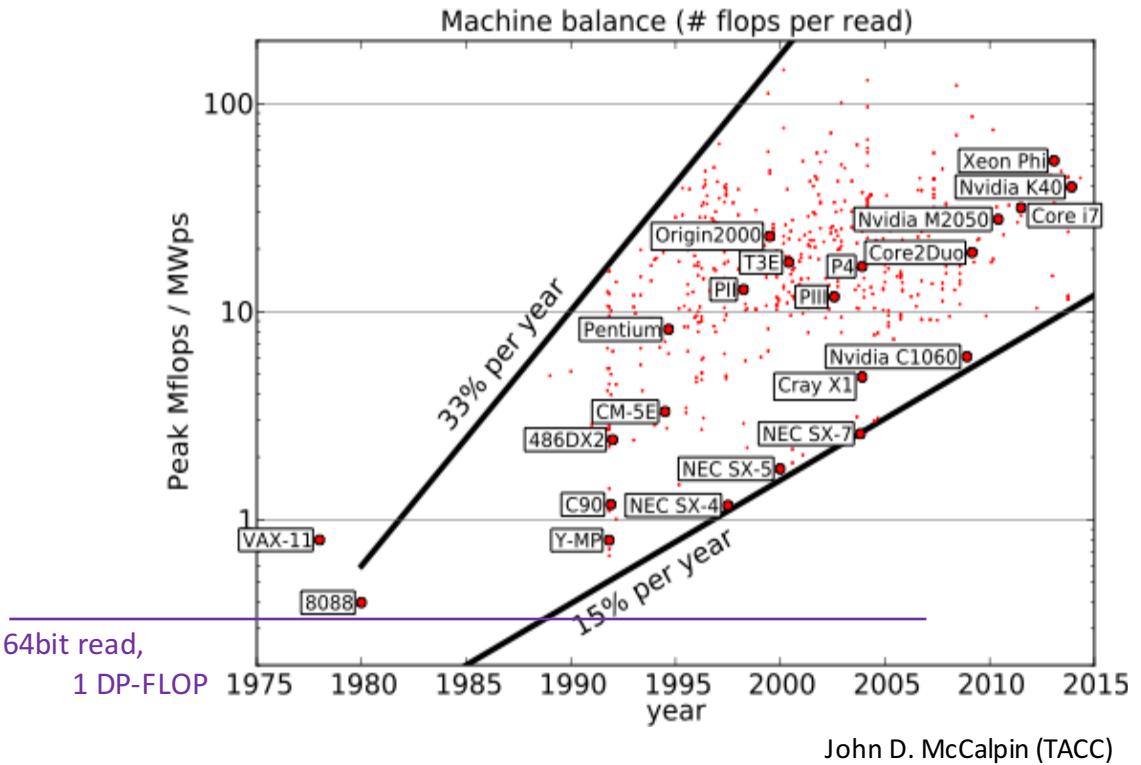
# Where do we stand?



- Node: 2 IBM POWER9 + 6 NVIDIA V100 GPUs
- 4,608 nodes, 9,216 IBM Power9 CPUs
- 27,648 V100 GPUs (**8 TFLOPs / GPU**)
- Peak performance of **200 Pflop/s** for modeling & simulation
- Peak performance of **3.3 Eflop/s** ( **$10^{18}$** ) for 16 bit floating point used in data analytics and artificial intelligence



Copyright@ORNL



1. Compute power (#FLOPs) grows much faster than bandwidth.  
*"Operations are free, mem access and comm is what counts."*

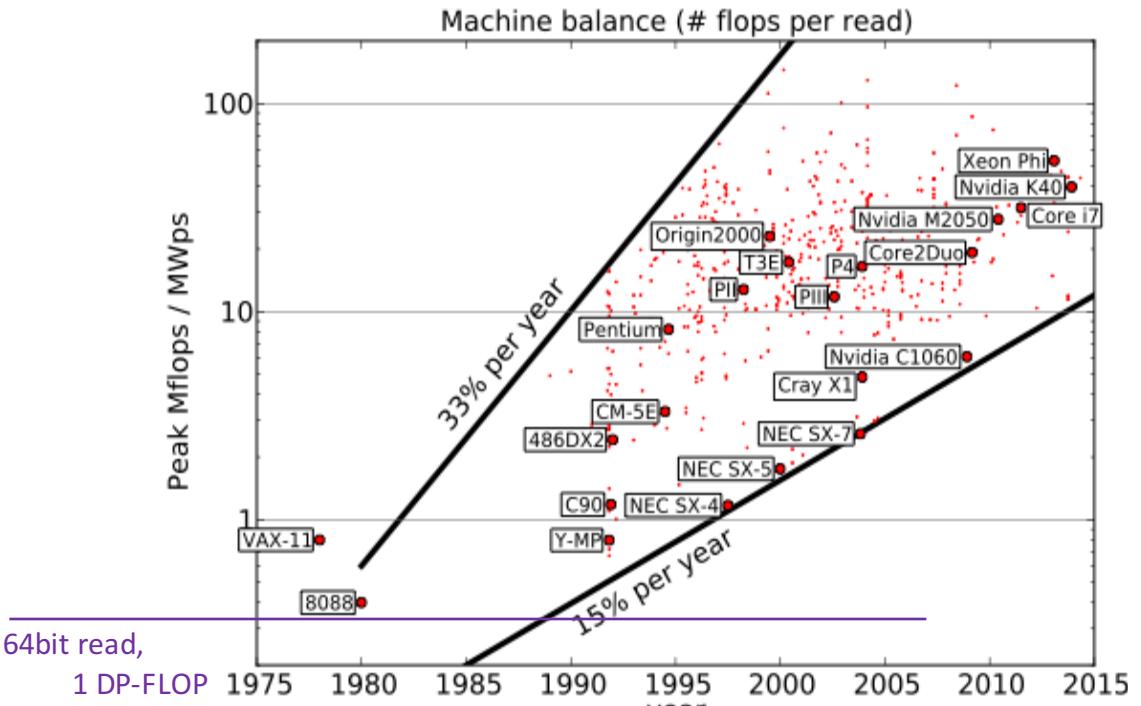
# Where do we stand?



- Node: 2 IBM POWER9 + 6 NVIDIA V100 GPUs
- 4,608 nodes, 9,216 IBM Power9 CPUs
- 27,648 V100 GPUs (**8 TFLOPs / GPU**)
- Peak performance of **200 Pflop/s** for modeling & simulation
- Peak performance of **3.3 Eflop/s** ( **$10^{18}$** ) for 16 bit floating point used in data analytics and artificial intelligence



Copyright@ORNL



John D. McCalpin (TACC)

1. Compute power (#FLOPs) grows much faster than bandwidth.  
*"Operations are free, mem access and comm is what counts."*
2. Manycore architectures need new algorithmic approaches.  
*"Sync-Free fine-grained parallelism needed."*

# Where do we stand?



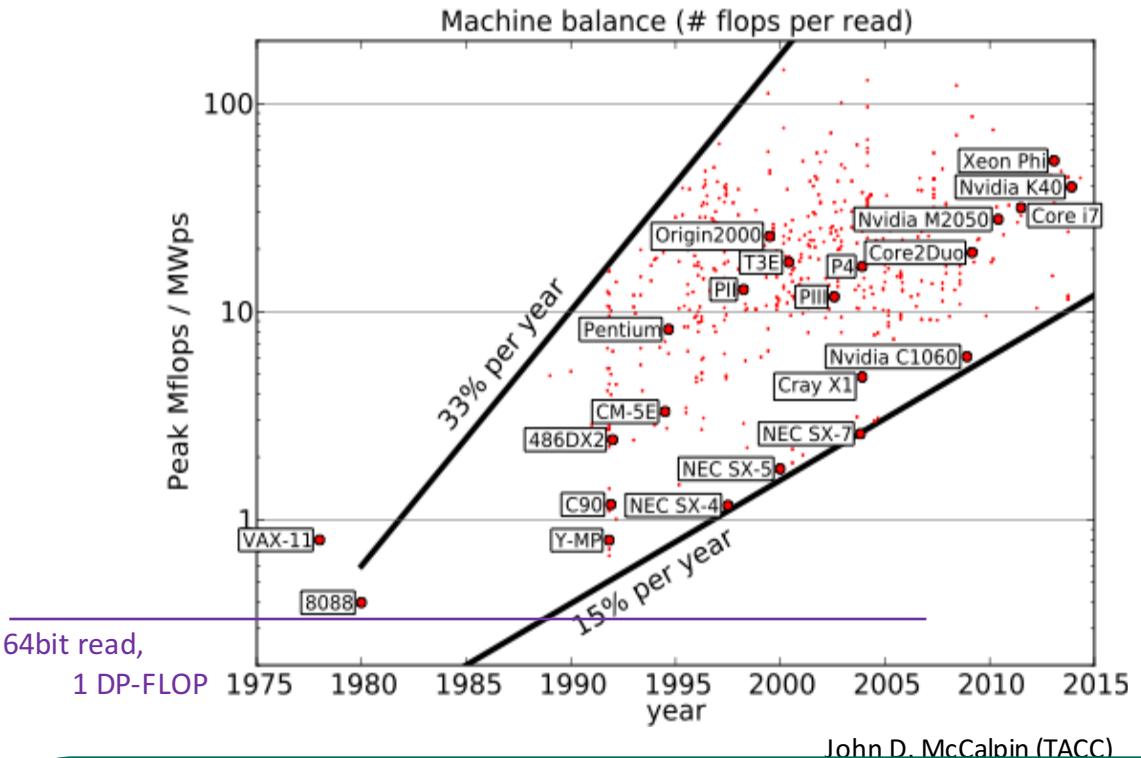
LEADERSHIP  
COMPUTING  
FACILITY



- Node: 2 IBM POWER9 + 6 NVIDIA V100 GPUs
- 4,608 nodes, 9,216 IBM Power9 CPUs
- 27,648 V100 GPUs (**8 TFLOPs / GPU**)
- Peak performance of **200 Pflop/s** for modeling & simulation
- Peak performance of **3.3 Eflop/s** ( **$10^{18}$** ) for 16 bit floating point used in data analytics and artificial intelligence



Copyright@ORNL



John D. McCalpin (TACC)

1. Compute power (#FLOPs) grows much faster than bandwidth.  
*"Operations are free, mem access and comm is what counts."*
2. Manycore architectures need new algorithmic approaches.  
*"Sync-Free fine-grained parallelism needed."*
3. Software lives longer than hardware.  
*"We need a paradigm change to embrace software development."*

# Where do we stand?



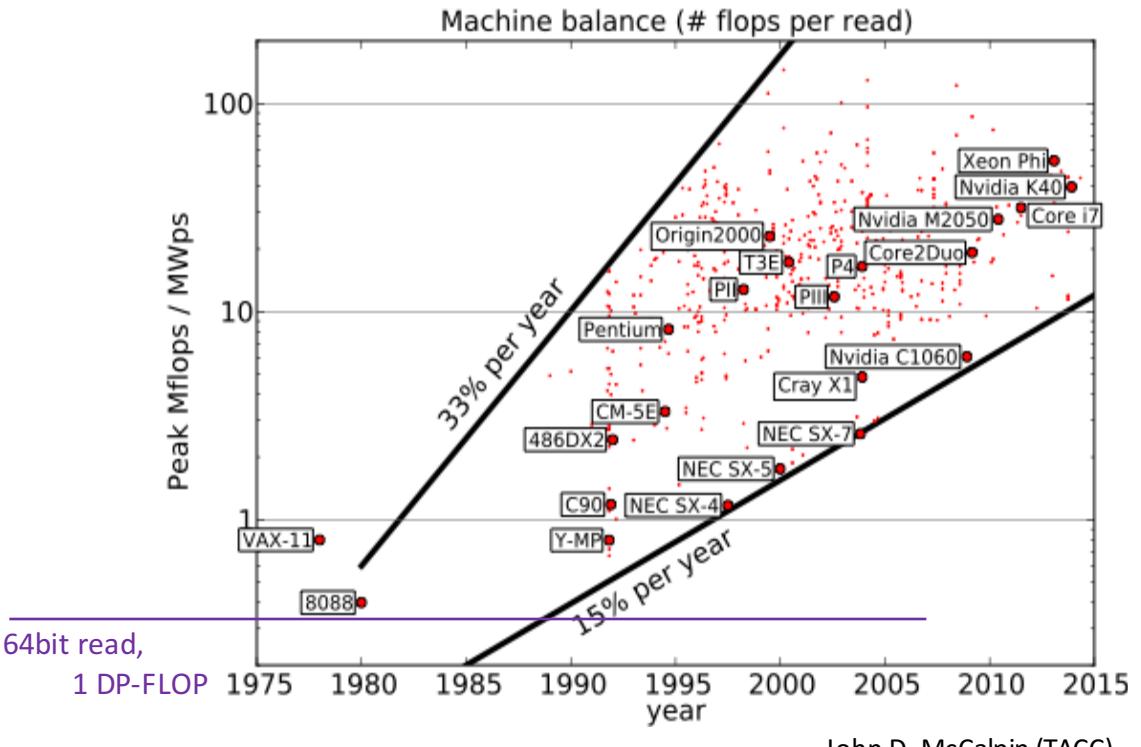
LEADERSHIP  
COMPUTING  
FACILITY



- Node: 2 IBM POWER9 + 6 NVIDIA V100 GPUs
- 4,608 nodes, 9,216 IBM Power9 CPUs
- 27,648 V100 GPUs (**8 TFLOPs / GPU**)
- Peak performance of **200 Pflop/s** for modeling & simulation
- Peak performance of **3.3 Eflop/s** ( **$10^{18}$** ) for 16 bit floating point used in data analytics and artificial intelligence



Copyright@ORNL



1. **Bandwidth Challenge**: Compute power (#FLOPs) grows much faster than bandwidth.  
*"Operations are free, mem access and comm is what counts."*
2. **Manycore Challenge**: Manycore architectures need new algorithmic approaches.  
*"Sync-Free fine-grained parallelism needed."*
3. **Software Challenge**: Software lives longer than hardware.  
*"We need a paradigm change to embrace software development."*

# The Communication Bottleneck



- Node: 2 IBM POWER9 + 6 NVIDIA V100 GPUs
- 4,608 nodes, 9,216 IBM Power9 CPUs
- 27,648 V100 GPUs (**8 TFLOPs / GPU**)
- Peak performance of **200 Pflop/s** for modeling & simulation
- Peak performance of **3.3 Eflop/s** ( $10^{18}$ ) for 16 bit floating point used in data analytics and artificial intelligence



Copyright@ORNL

# The Communication Bottleneck



- Node: 2 IBM POWER9 + 6 NVIDIA V100 GPUs
- 4,608 nodes, 9,216 IBM Power9 CPUs
- 27,648 V100 GPUs (**8 TFLOPs / GPU**)
- Peak performance of **200 Pflop/s** for modeling & simulation
- Peak performance of **3.3 Eflop/s** ( **$10^{18}$** ) for 16 bit floating point used in data analytics and artificial intelligence



Copyright@ORNL

## Roofline Model

Given certain **hardware characteristics**:

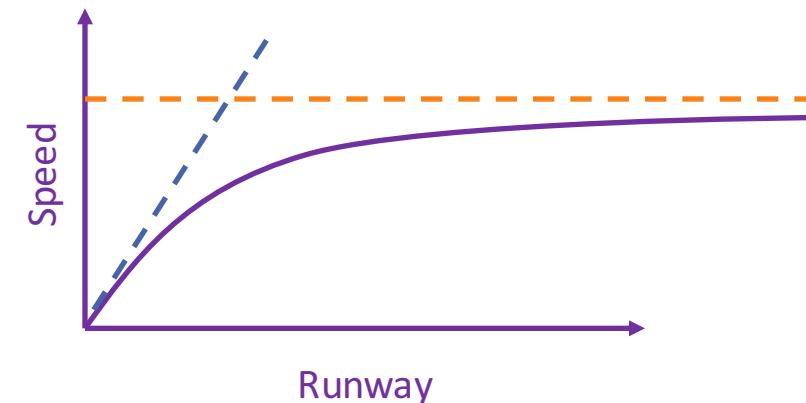
*memory bandwidth,*  
*arithmetic power,*

Acceleration  
Top Speed



the performance of any operation is

- either bound by the data access/communication (*memory bound*),
- or by the arithmetic operations (*compute bound*).



# The Communication Bottleneck



- Node: 2 IBM POWER9 + 6 NVIDIA V100 GPUs
- 4,608 nodes, 9,216 IBM Power9 CPUs
- 27,648 V100 GPUs (**8 TFLOPs / GPU**)
- Peak performance of **200 Pflop/s** for modeling & simulation
- Peak performance of **3.3 Eflop/s** ( **$10^{18}$** ) for 16 bit floating point used in data analytics and artificial intelligence



Copyright@ORNL

## Rooftline Model

Given certain **hardware characteristics**:

*memory bandwidth,*  
*arithmetic power,*

**Acceleration  
Top Speed**



the performance of any operation is

- either bound by the data access/communication (*memory bound*),
- or by the arithmetic operations (*compute bound*).

Matrix-Matrix Product (GEMM):  $C = A \times B$        $A, B, C \in \mathbb{R}^{n \times n}$

$3n^2$  Memory operations  
 $2n^3$  Arithmetic operations

*We just need to increase the size, and at some point the operation becomes compute bound.*

*"we infinitely extend the acceleration runway"*

# The Communication Bottleneck



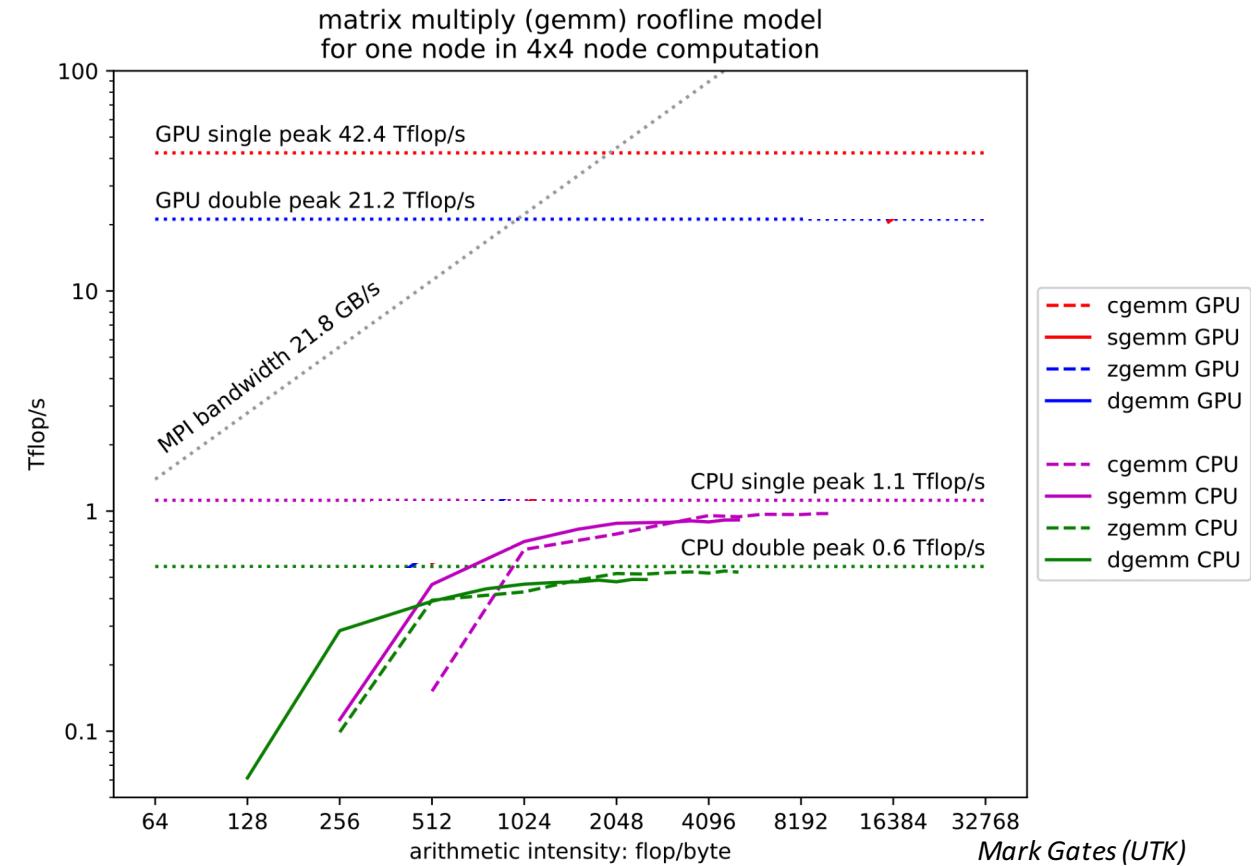
- Node: 2 IBM POWER9 + 6 NVIDIA V100 GPUs
- 4,608 nodes, 9,216 IBM Power9 CPUs
- 27,648 V100 GPUs (**8 TFLOPs / GPU**)
- Peak performance of **200 Pflop/s** for modeling & simulation
- Peak performance of **3.3 Eflop/s (10^18)** for 16 bit floating point used in data analytics and artificial intelligence



Copyright@ORNL

## Dense Matrix Operations?

- The inter-node communication is the limiting resource;
- Each node has more computational power than what we can leverage;



# The Communication Bottleneck



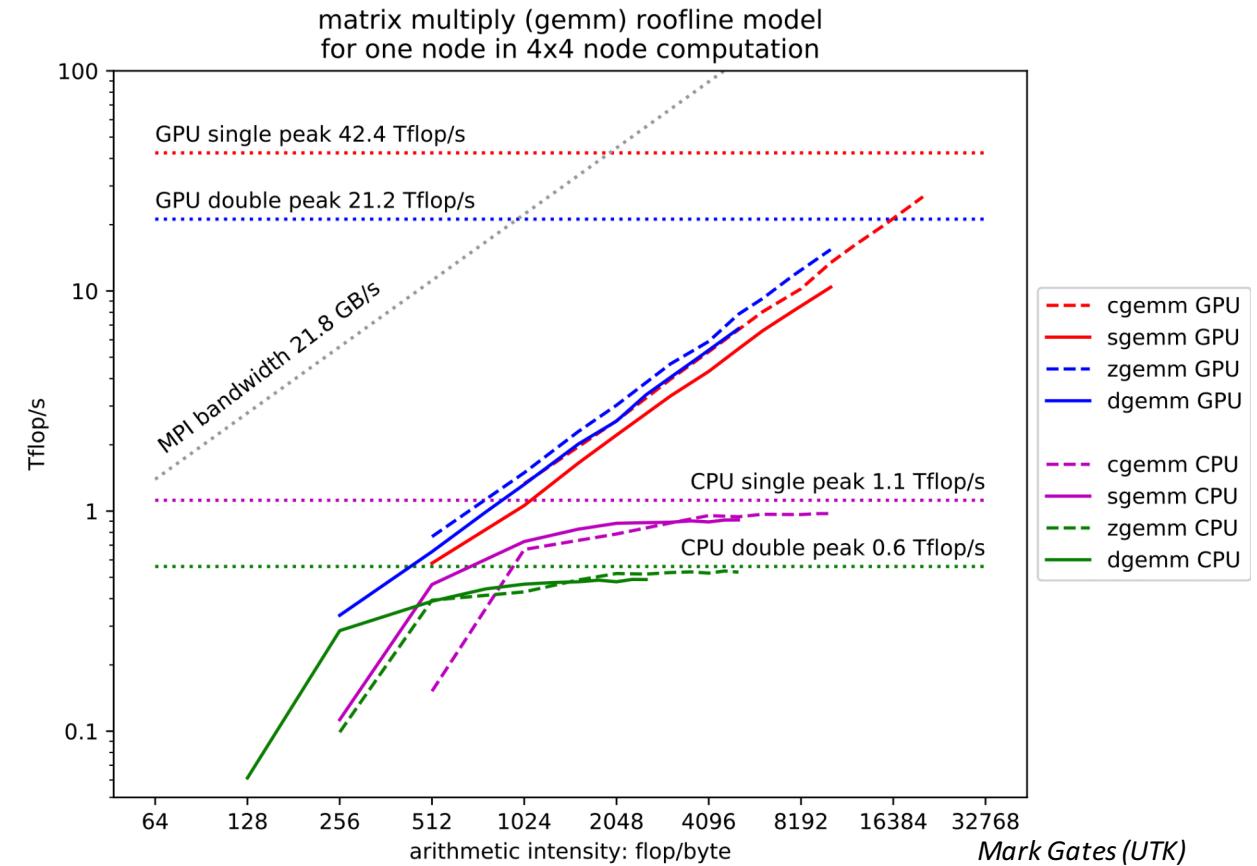
- Node: 2 IBM POWER9 + 6 NVIDIA V100 GPUs
- 4,608 nodes, 9,216 IBM Power9 CPUs
- 27,648 V100 GPUs (**8 TFLOPs / GPU**)
- Peak performance of **200 Pflop/s** for modeling & simulation
- Peak performance of **3.3 Eflop/s (10<sup>18</sup>)** for 16 bit floating point used in data analytics and artificial intelligence



Copyright@ORNL

## Dense Matrix Operations?

- The inter-node communication is the limiting resource;
- Each node has more computational power than what we can leverage;



# The Communication Bottleneck



- Node: 2 IBM POWER9 + 6 NVIDIA V100 GPUs
- 4,608 nodes, 9,216 IBM Power9 CPUs
- 27,648 V100 GPUs (**8 TFLOPs / GPU**)
- Peak performance of **200 Pflop/s** for modeling & simulation
- Peak performance of **3.3 Eflop/s** ( **$10^{18}$** ) for 16 bit floating point used in data analytics and artificial intelligence



Copyright@ORNL

## Dense Matrix Operations?

- The inter-node communication is the limiting resource;
- Each node has more computational power than what we can leverage;

## Sparse / Graph Problems?

- *Sparse Matrix Vector Product* (SpMV) is a central building block;

$$\begin{pmatrix} \times & \times & \times & \times & & \times & \times \\ \times & \times & \times & & & & \\ \times & \times & \times & \times & & & \\ \times & & \times & \times & \times & & \\ \times & & \times & \times & \times & & \\ \times & & \times & \times & \times & \times & \\ \times & & & \times & \times & & \\ \times & & & & \times & \times & \\ \end{pmatrix} \cdot \begin{pmatrix} \times \\ \times \end{pmatrix}$$

# The Communication Bottleneck



- Node: 2 IBM POWER9 + 6 NVIDIA V100 GPUs
- 4,608 nodes, 9,216 IBM Power9 CPUs
- 27,648 V100 GPUs (**8 TFLOPs / GPU**)
- Peak performance of **200 Pflop/s** for modeling & simulation
- Peak performance of **3.3 Eflop/s** ( **$10^{18}$** ) for 16 bit floating point used in data analytics and artificial intelligence



Copyright@ORNL

## Dense Matrix Operations?

- The inter-node communication is the limiting resource;
- Each node has more computational power than what we can leverage;

## Sparse / Graph Problems?

- *Sparse Matrix Vector Product* (SpMV) is a central building block;
- For many of the problems in the SuiteSparse Matrix Collection<sup>1</sup>, a Multi-node SpMV is slower than a Single-node SpMV;
- The inter-node communication is an order of magnitude slower than the local computations.

$$\begin{pmatrix} \times & \times & \times & \times & & \times & \times \\ \times & \times & \times & & & & \\ \times & \times & \times & \times & & & \\ \times & & \times & \times & \times & & \times \\ & & & \times & \times & \times & \\ \times & & \times & \times & \times & \times & \times \\ \times & & & \times & \times & & \\ \end{pmatrix} \cdot \begin{pmatrix} \times \\ \times \\ \times \\ \times \\ \times \\ \times \\ \times \end{pmatrix}$$

<sup>1</sup>SuiteSparse Matrix Collection: <https://sparse.tamu.edu/>

# The Communication Bottleneck



- Node: 2 IBM POWER9 + 6 NVIDIA V100 GPUs
- 4,608 nodes, 9,216 IBM Power9 CPUs
- 27,648 V100 GPUs (**8 TFLOPs / GPU**)
- Peak performance of **200 Pflop/s** for modeling & simulation
- Peak performance of **3.3 Eflop/s** ( $10^{18}$ ) for 16 bit floating point used in data analytics and artificial intelligence



**Radically decouple storage format from arithmetic format.**

## Dense Matrix Operations?

- The inter-node communication is the limiting resource;
- Each node has more computational power than what we can leverage;

## Sparse / Graph Problems?

- *Sparse Matrix Vector Product* (SpMV) is a central building block;
- Many of the problems in the SuiteSparse Matrix Collection<sup>1</sup>, multi-node SpMV is slower than a Single-node SpMV;

The inter-node communication is an order of magnitude slower than the local computations.

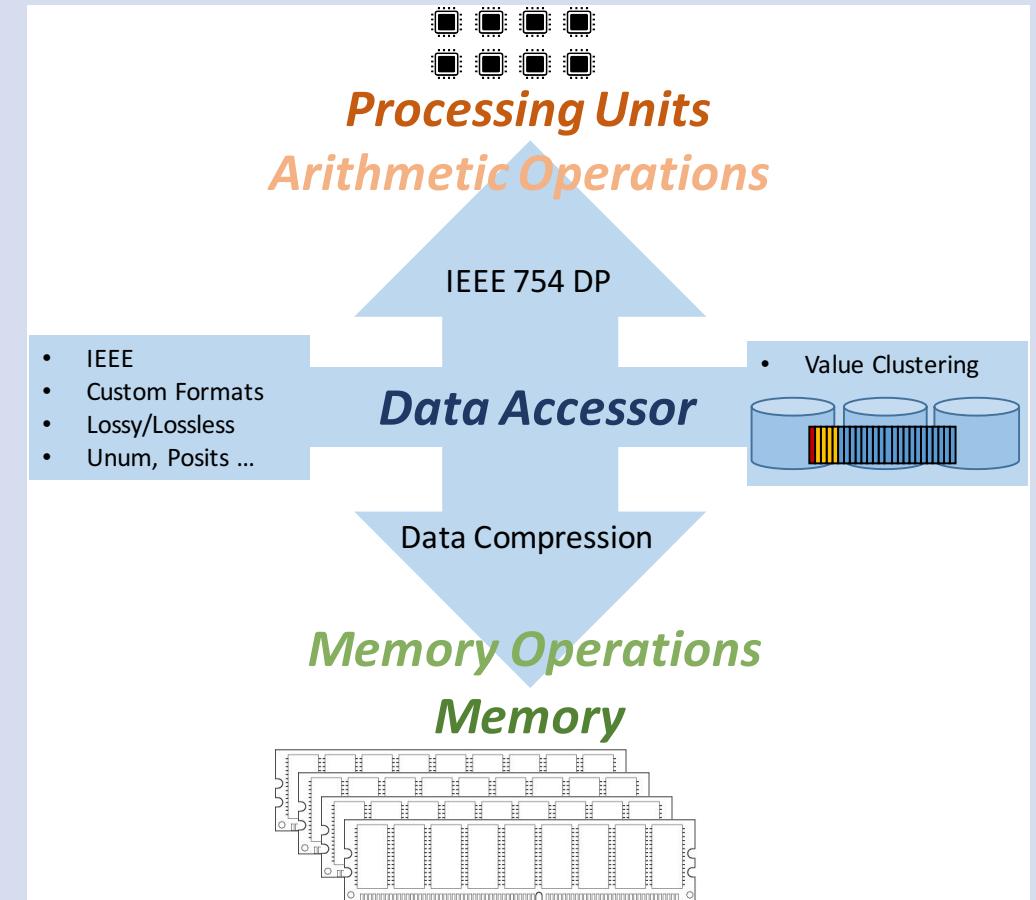
$$\begin{pmatrix} \times & \times & \times & \times & & \times & \times \\ \times & \times & \times & & & & \\ \times & \times & \times & \times & & & \\ \times & & \times & \times & \times & & \\ \times & & \times & \times & \times & & \\ \times & & \times & \times & \times & \times & \\ \times & & & \times & \times & & \\ \times & & & \times & \times & & \\ \end{pmatrix} \cdot \begin{pmatrix} \times \\ \times \end{pmatrix}$$

<sup>1</sup>SuiteSparse Matrix Collection: <https://sparse.tamu.edu/>

# The Communication Bottleneck

Radically decouple storage format from arithmetic format.

- The arithmetic operations should use high precision formats natively supported by hardware.
- Data access should be as cheap as possible, reduced precision.
- Consider a wide range of memory formats:
  - IEEE standard precision formats
  - Customized formats (configuring mantissa/exponent)
  - Lossy compression
  - ...



Copyright@ORNL

<sup>1</sup>SuiteSparse Matrix Collection: <https://sparse.tamu.edu/>

## Spotlight Example: Use reduced precision for “approximate Operators”

- **Solve sparse linear system**  $Ax = b$
- **Preconditioners for iterative solvers.**
  - Idea: Approximate inverse of system matrix to make the system “easier to solve”:  $P^{-1} \approx A^{-1}$   
 $\tilde{A} = P^{-1}A$ ,  $\tilde{b} = P^{-1}b$ , and we solve  $Ax = b \Leftrightarrow \tilde{A}x = \tilde{b}$ .

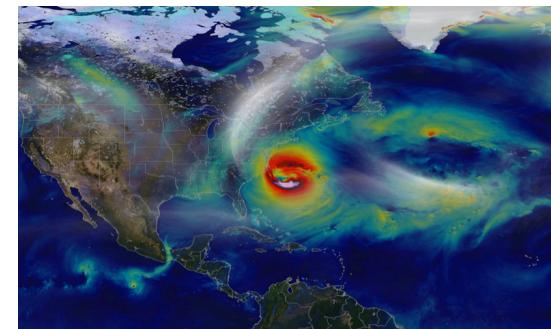
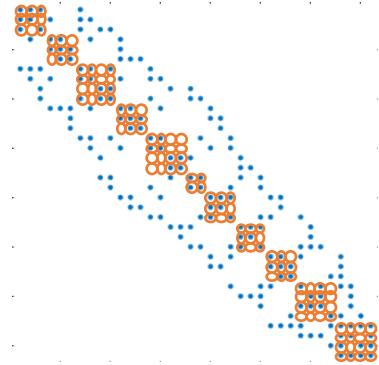
## Spotlight Example: Use reduced precision for “approximate Operators”

- Solve sparse linear system  $Ax = b$
- Preconditioners for iterative solvers.
  - Idea: Approximate inverse of system matrix to make the system “easier to solve”:  $P^{-1} \approx A^{-1}$   
 $\tilde{A} = P^{-1}A$ ,  $\tilde{b} = P^{-1}b$ , and we solve  $Ax = b \Leftrightarrow \tilde{A}x = \tilde{b}$ .
  - Why should we store the preconditioner matrix  $P^{-1}$  in full (high) precision?
    - We have to ensure regularity! (Reducing precision can turn matrix singular)

# Spotlight Example: Use reduced precision for “approximate Operators”

- Solve sparse linear system  $Ax = b$
- Preconditioners for iterative solvers.
  - Idea: Approximate inverse of system matrix to make the system “easier to solve”:  $P^{-1} \approx A^{-1}$   
 $\tilde{A} = P^{-1}A$ ,  $\tilde{b} = P^{-1}b$ , and we solve  $Ax = b \Leftrightarrow \tilde{A}x = \tilde{b}$ .
  - Why should we store the preconditioner matrix  $P^{-1}$  in full (high) precision?
    - We have to ensure regularity! (Reducing precision can turn matrix singular)
- Jacobi method based on diagonal scaling  $P = \text{diag}(A)$
- Block-Jacobi is based on block-diagonal scaling:  $P = \text{diag}_B(A)$ 
  - Large set of small diagonal blocks.
  - Each block corresponds to one (small) linear system.
    - Larger blocks typically improve convergence.
    - Larger blocks make block-Jacobi more expensive.

*Extreme case: one block of matrix size.*

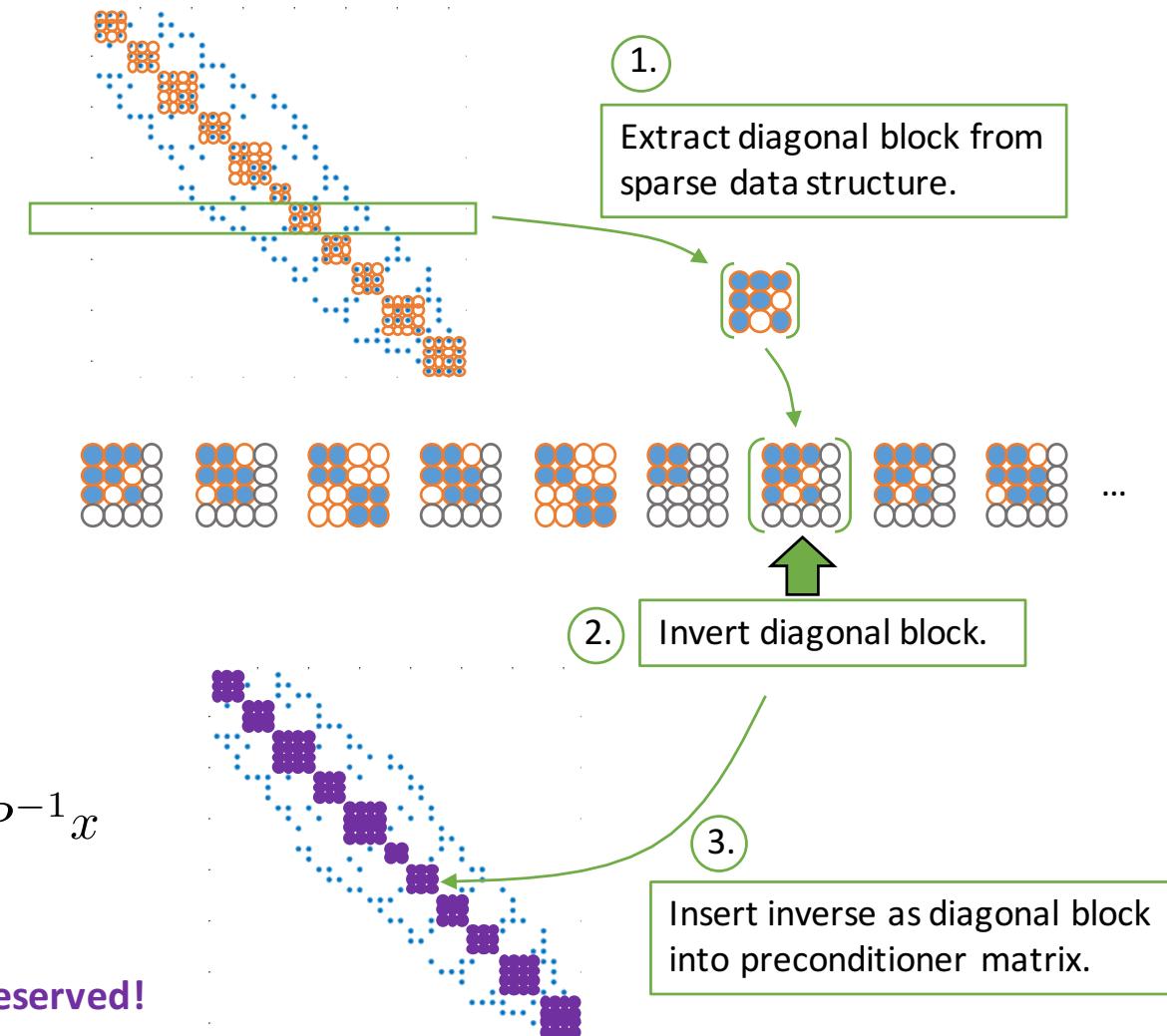


<https://science.nasa.gov/earth-science/focus-areas/earth-weather>

# Spotlight Example: Block-Jacobi Preconditioning

## Preconditioner Setup:

- Identify the diagonal blocks  $P = \text{diag}_B(A)$
- Form the block-Inverse  $P^{-1} \approx A^{-1}$



## Preconditioner Application:

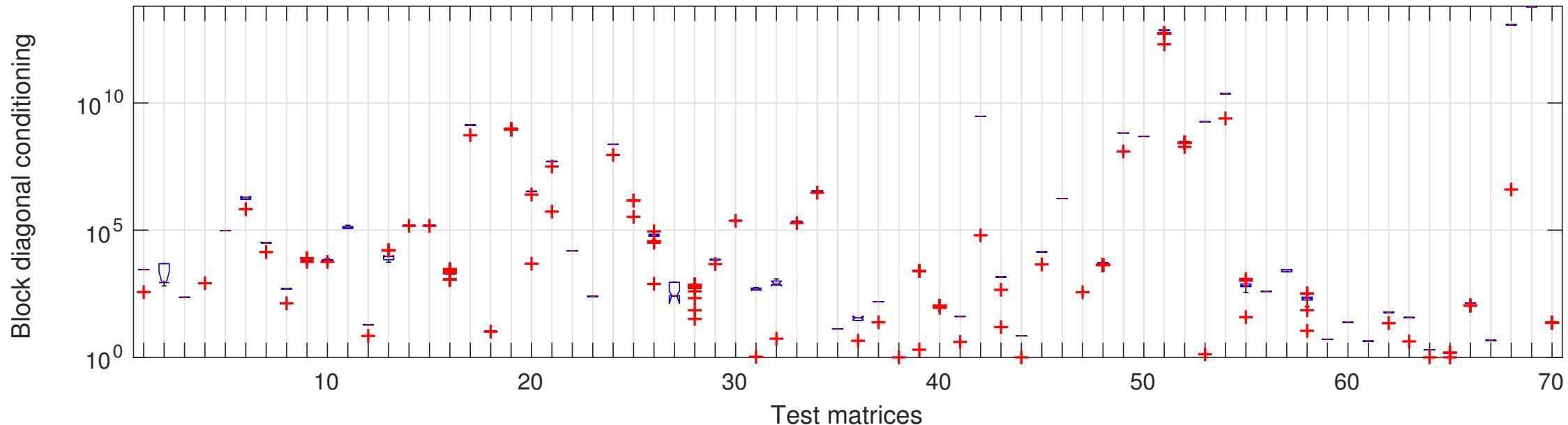
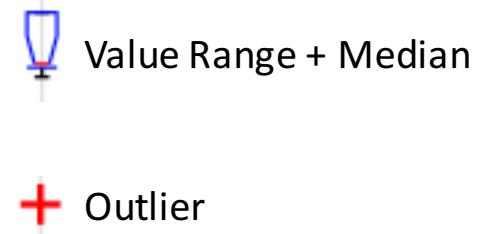
- Apply the preconditioner in every solver iteration via:

$$y := P^{-1}x$$

We can store diagonal blocks in lower precision, if regularity is preserved!

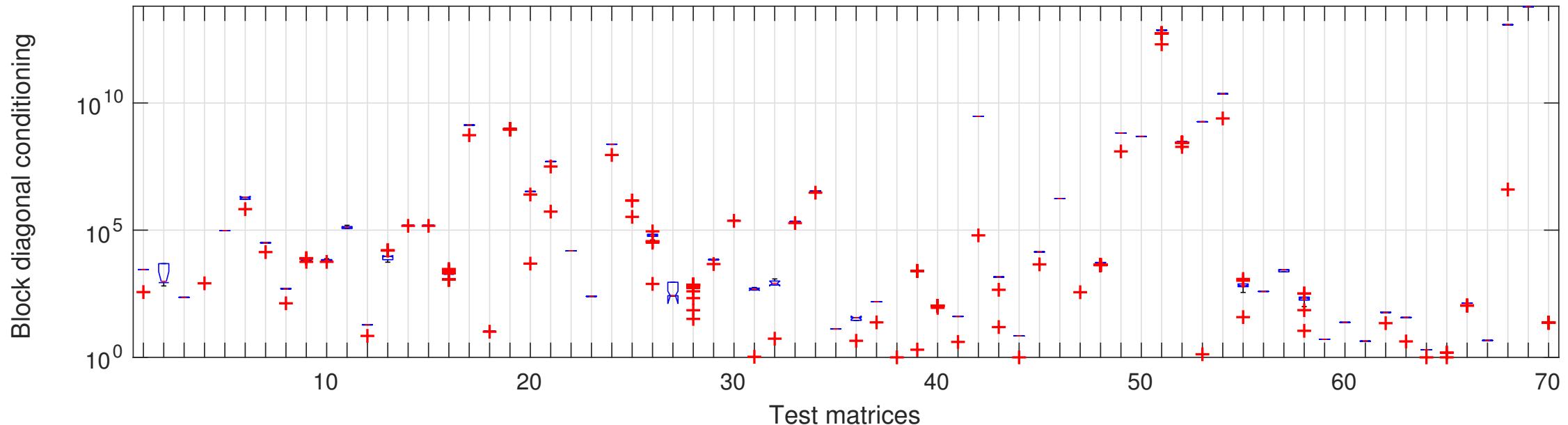
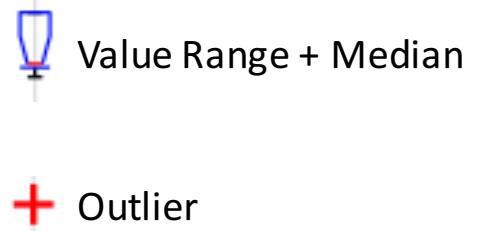
# Adaptive Precision Block-Jacobi Preconditioning

- 70 matrices from the SuiteSparse Matrix Collection
- Use block-size 24 with Super-Variable agglomeration (24 is upper bound for size of blocks)
- Report conditioning of all arising diagonal blocks

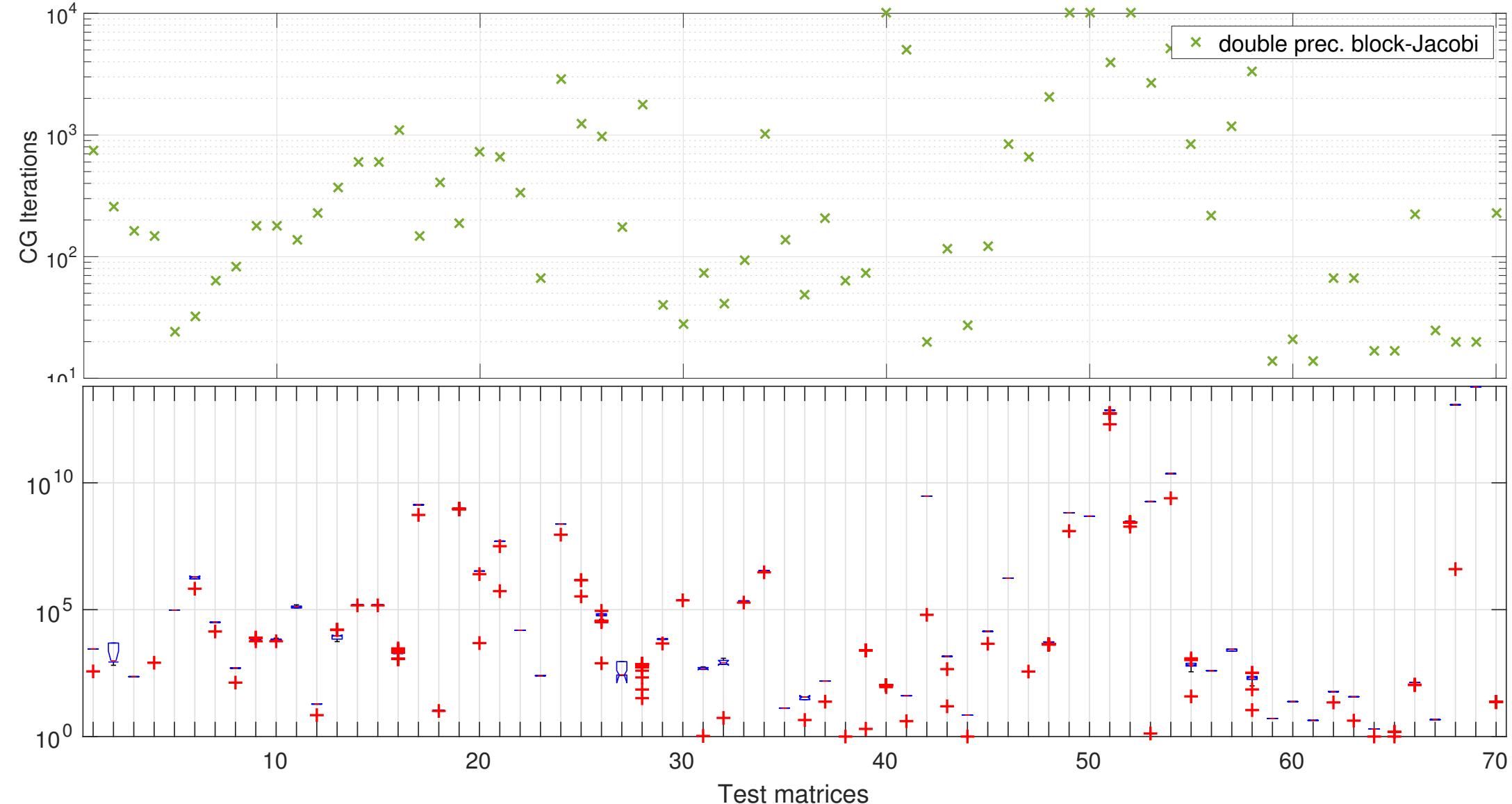


# Adaptive Precision Block-Jacobi Preconditioning

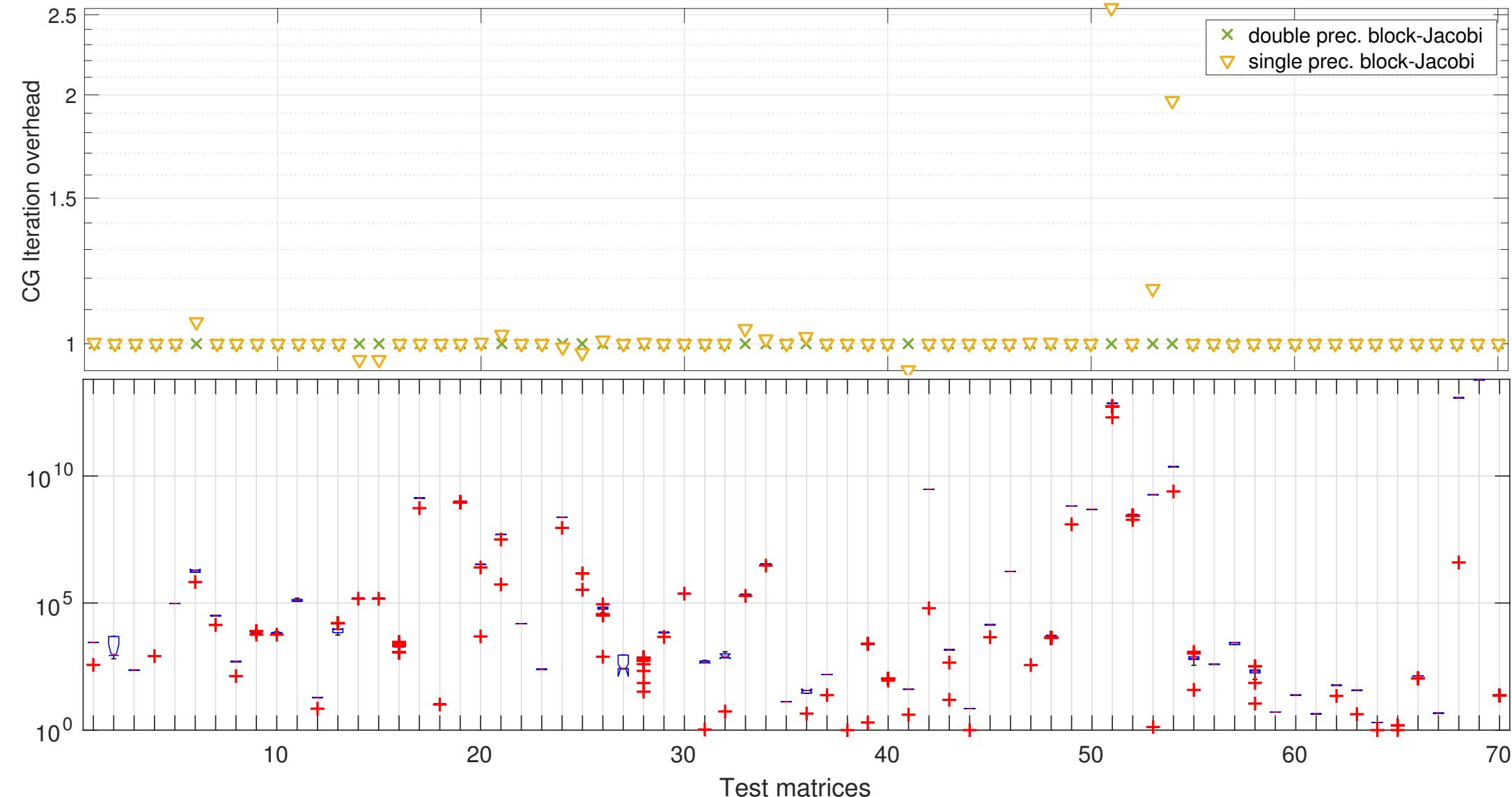
- 70 matrices from the SuiteSparse Matrix Collection
- Use block-size 24 with Super-Variable agglomeration (24 is upper bound for size of blocks)
- Report conditioning of all arising diagonal blocks
- Analyze the impact of storing block-Jacobi in lower precision a top-level Conjugate Gradient solver (CG)



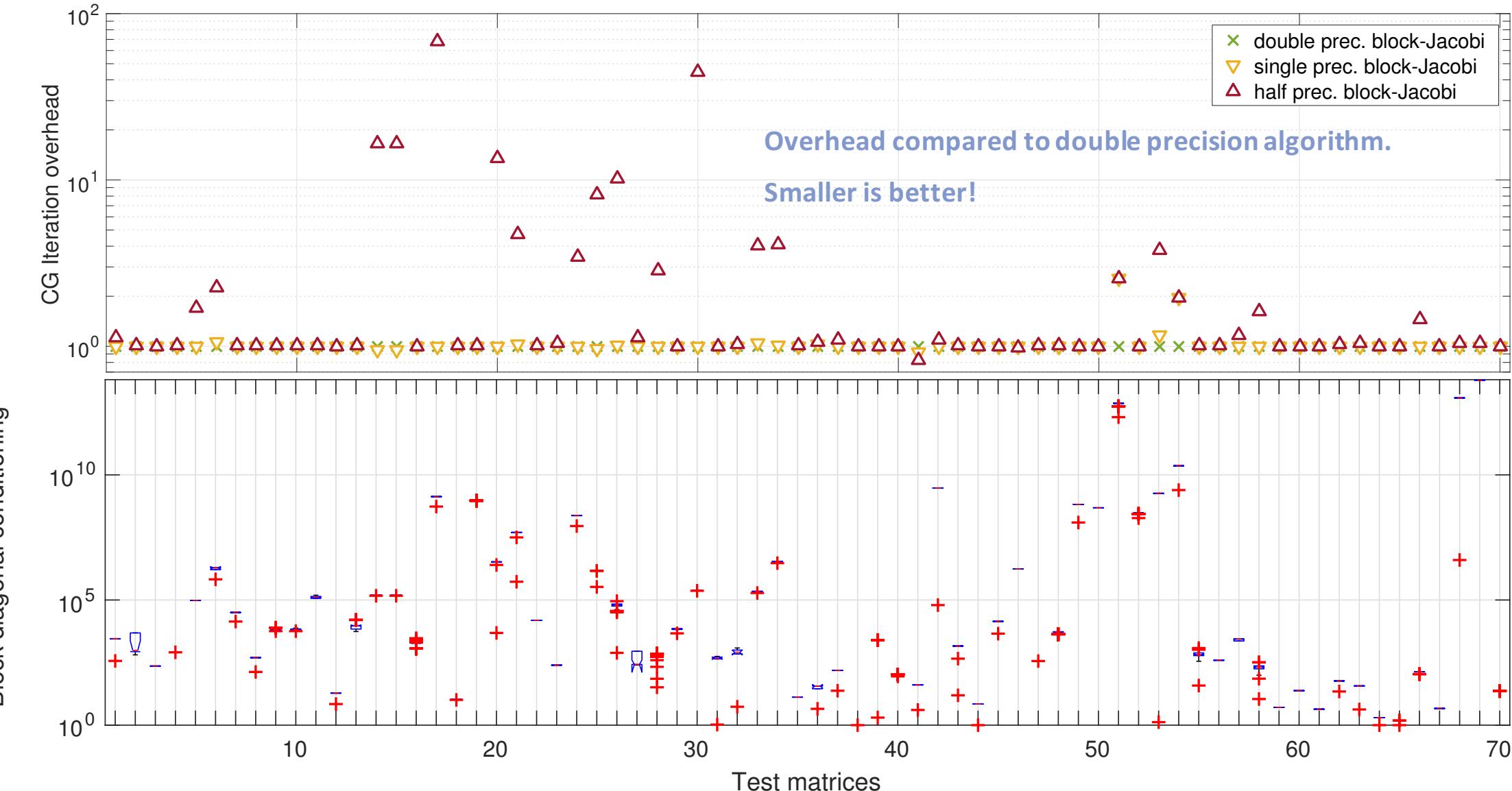
# Adaptive Precision Block-Jacobi Preconditioning



# Adaptive Precision Block-Jacobi Preconditioning



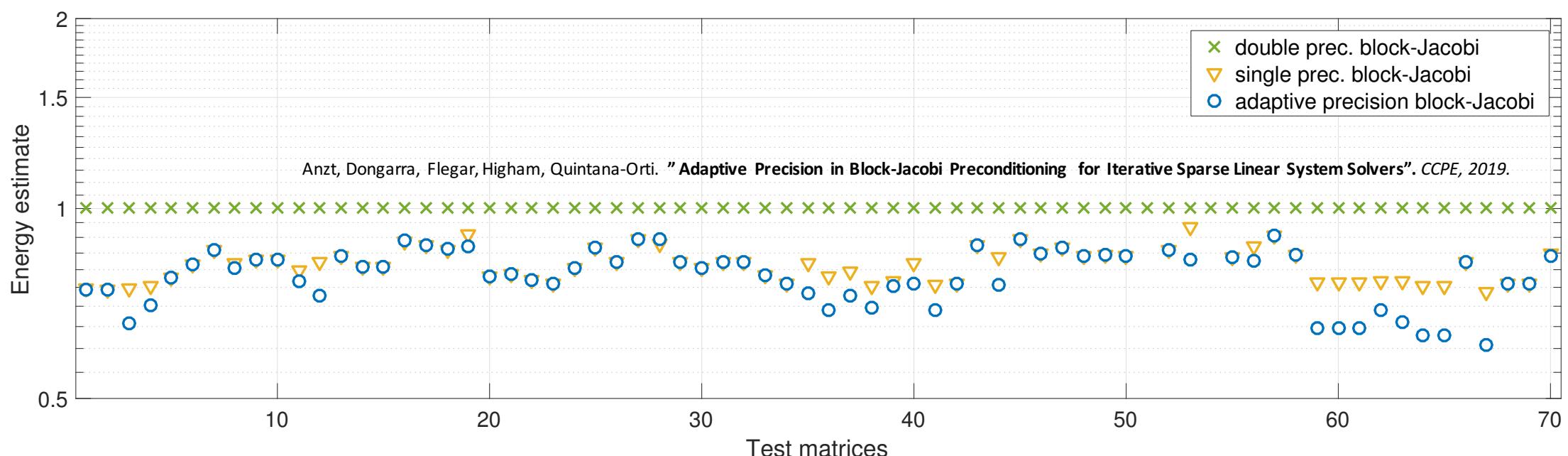
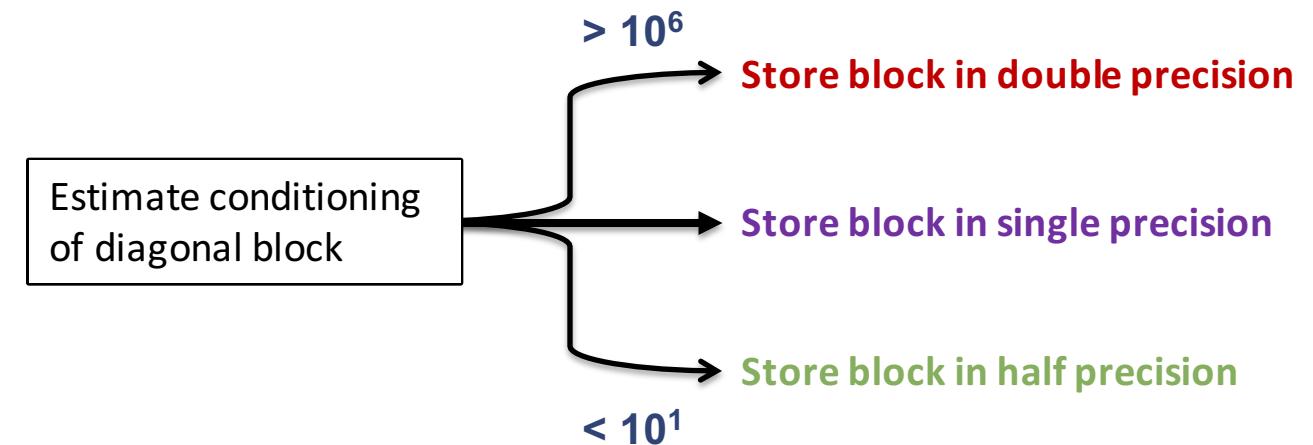
# Adaptive Precision Block-Jacobi Preconditioning



# Adaptive Precision Block-Jacobi Preconditioning

## Multi-Precision Idea:

- All computations use double precision!
- Store distinct blocks in different formats
- Use single precision as standard storage format
- Where necessary: switch to double
- For well-conditioned blocks use half precision



# Adaptive Precision Block-Jacobi Preconditioning

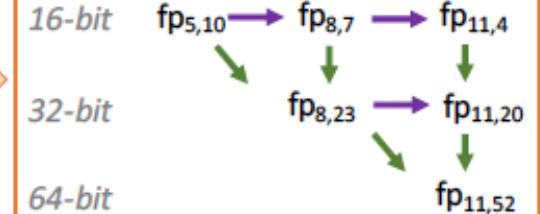
## Multi-Precision Idea:

- All computations use double precision!
- Depart from the rigid IEEE precision formats!
- Preserve either 1 or 2 digits accuracy of the inverted diagonal blocks.

Invert the diagonal block using Gauss-Jordan elimination.

Compute condition number and exponent range.

Select storage format:



Flegar, Anzt, Quintana-Orti. "Customized-Precision Block-Jacobi Preconditioning for Krylov Iterative Solvers on Data-Parallel Manycore Processors". TOMS, submitted.

# Adaptive Precision Block-Jacobi Preconditioning

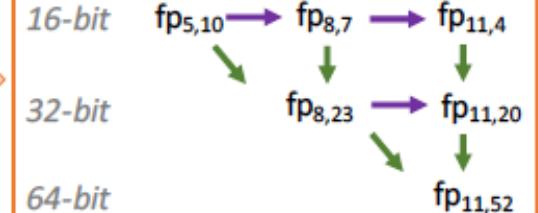
## Multi-Precision Idea:

- All computations use double precision!
- Depart from the rigid IEEE precision formats!
- Preserve either 1 or 2 digits accuracy of the inverted diagonal blocks.

Invert the diagonal block using Gauss-Jordan elimination.

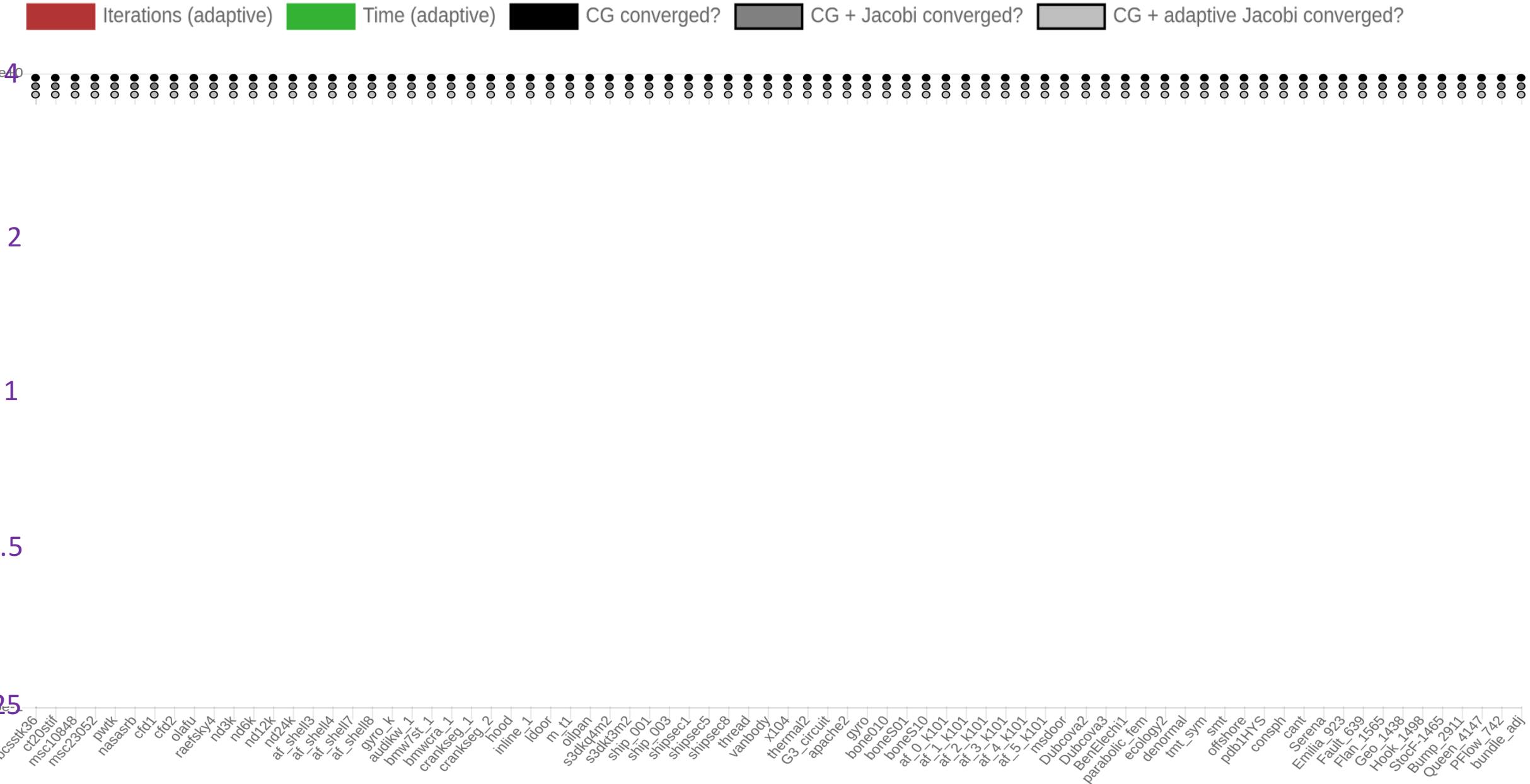
Compute condition number and exponent range.

Select storage format:

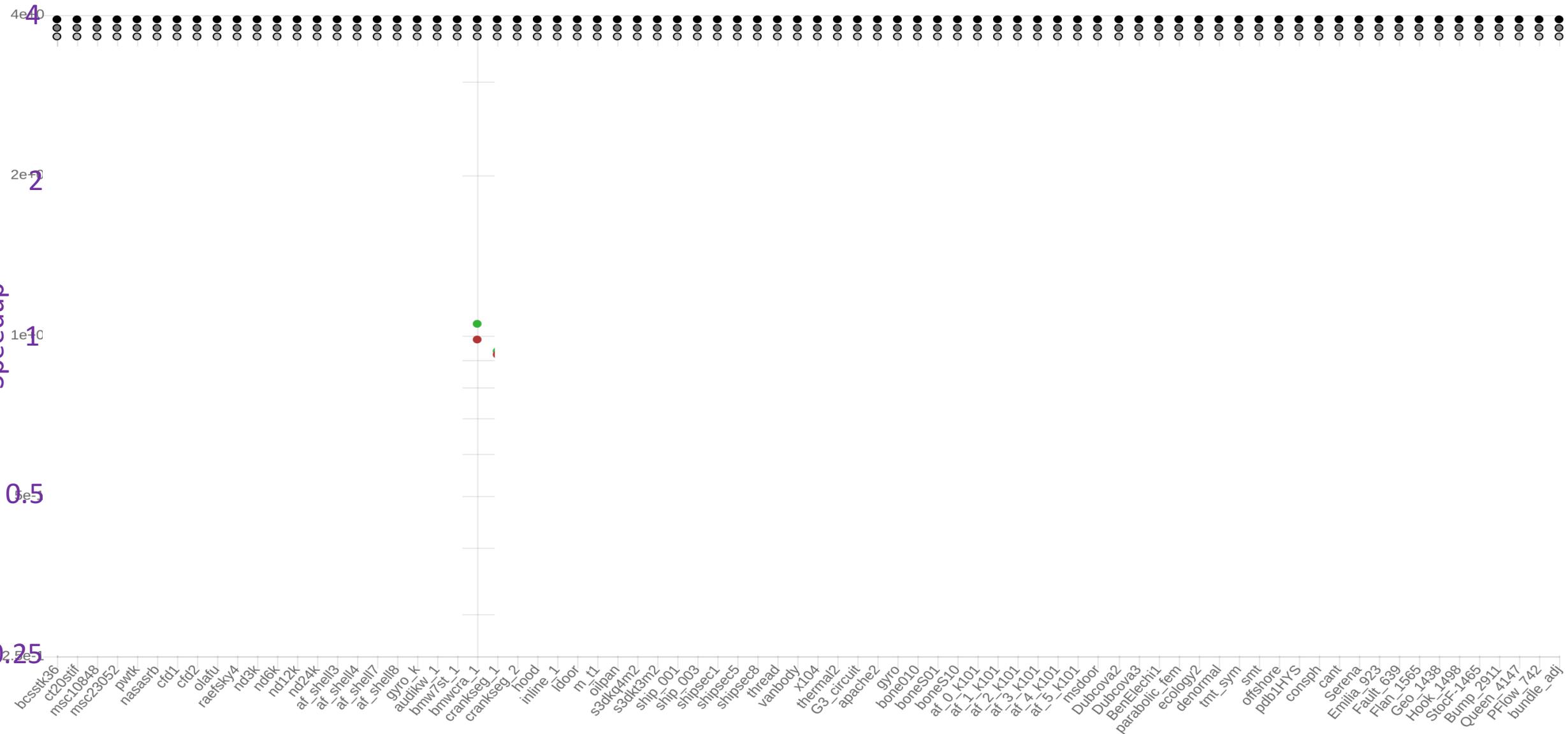


- ✓ Regularity preserved;
- ✓ No flexible Krylov solver needed  
(Preconditioner constant operator);
- ✓ Can handle non-spd problems  
(inversion features pivoting);
- ✓ Preconditioner for any iterative preconditionable solver;

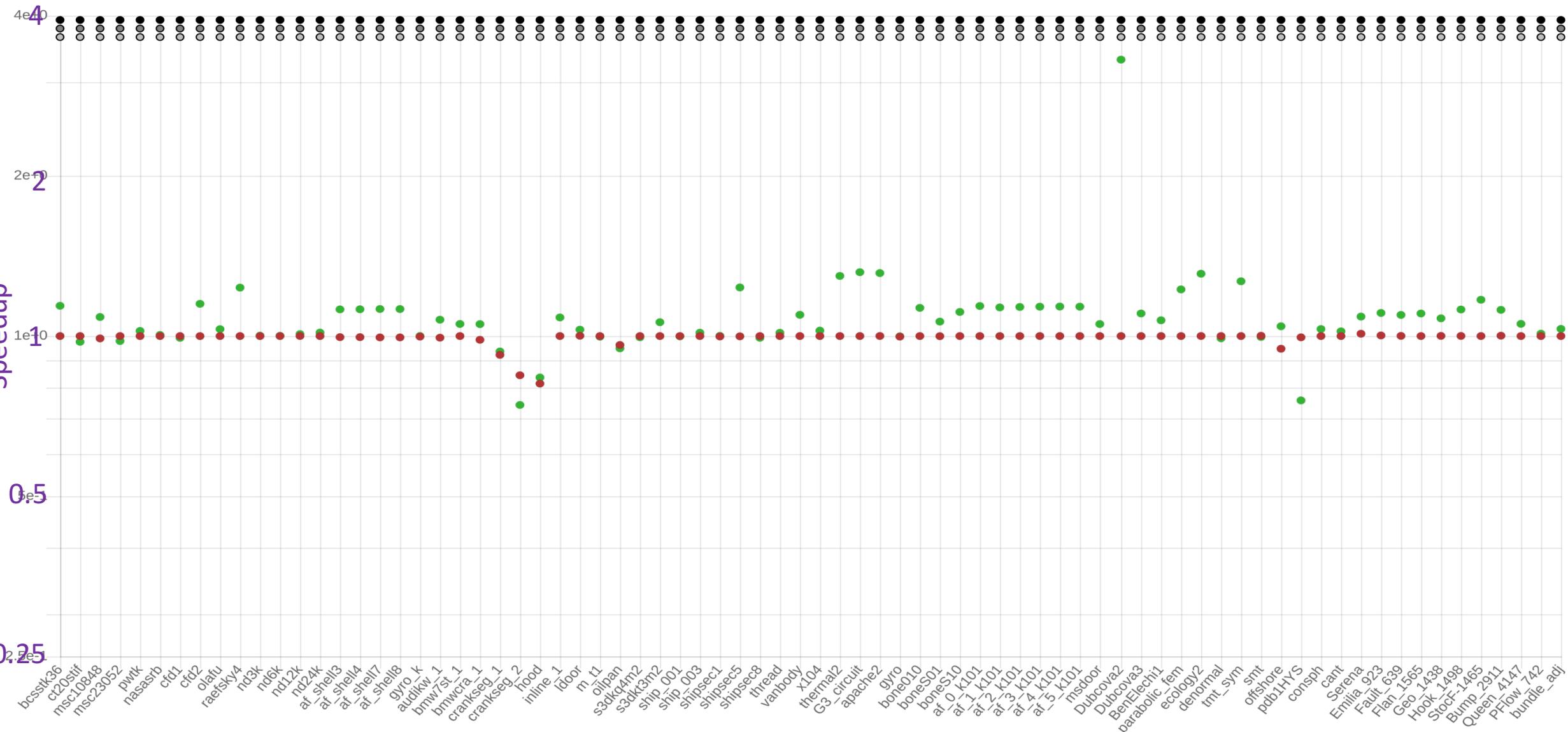
- Overhead of the precision detection  
(condition number calculation);
- Overhead from storing precision information  
(need to additionally store/retrieve flag);
- Speedups / preconditioner quality problem-dependent;



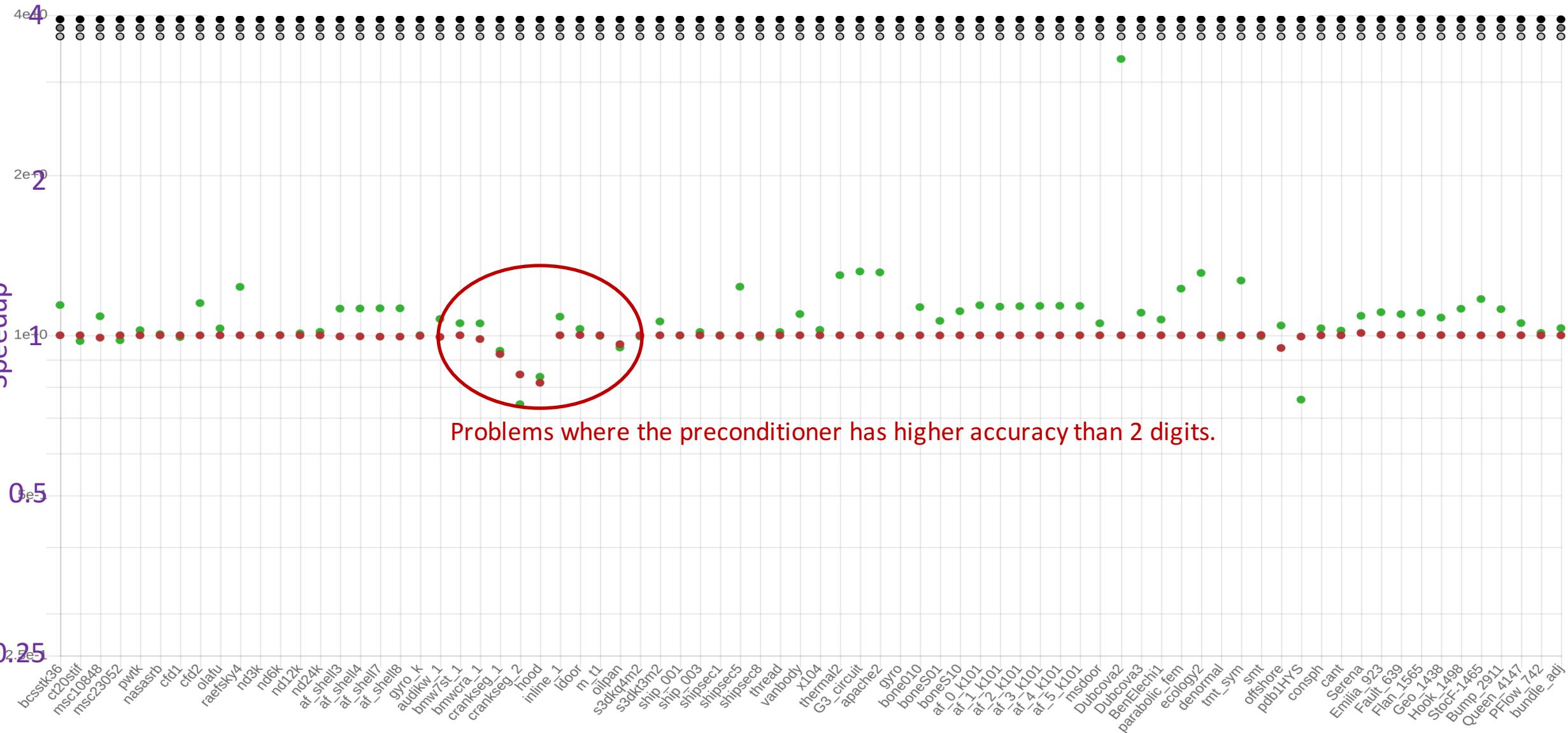
Iterations (adaptive) Time (adaptive) CG converged? CG + Jacobi converged? CG + adaptive Jacobi converged?

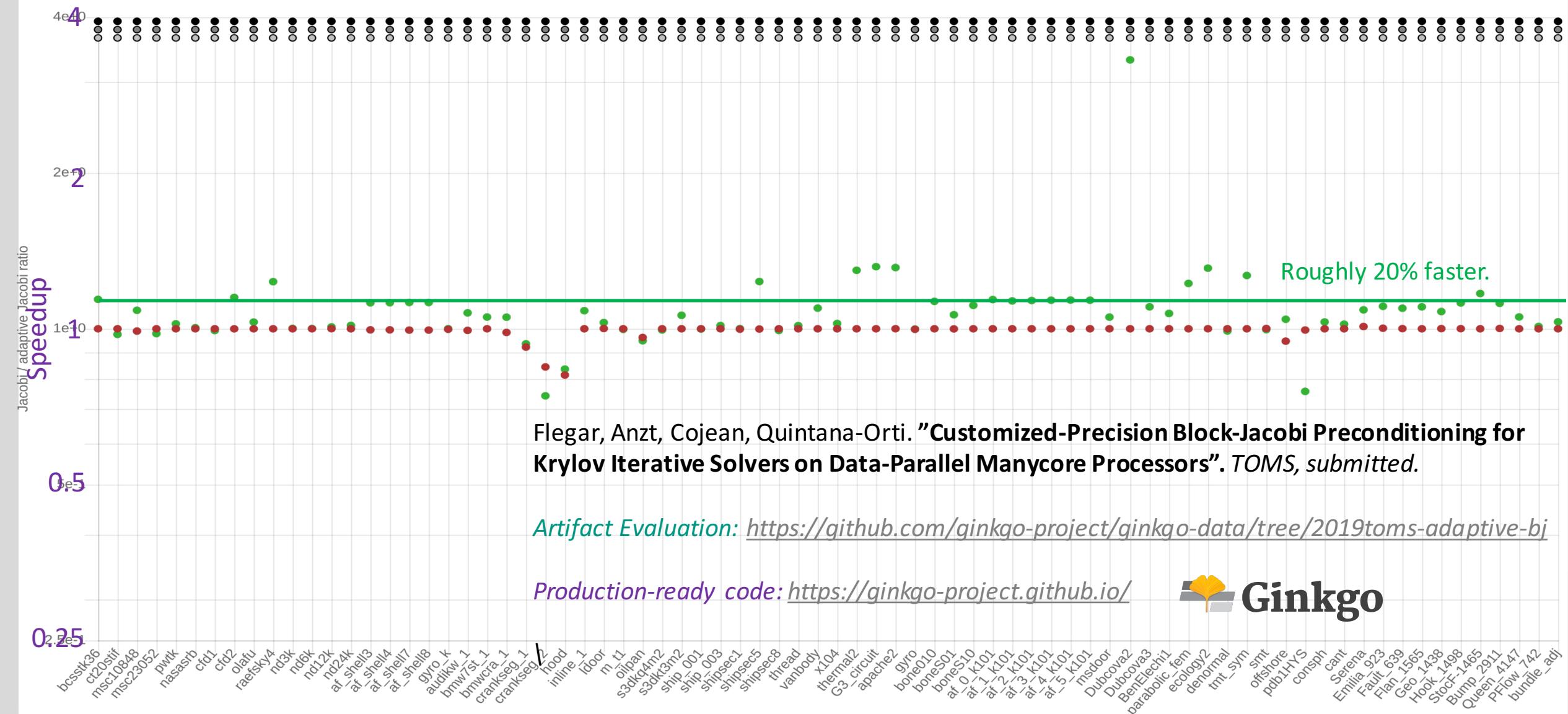


Iterations (adaptive) Time (adaptive) CG converged? CG + Jacobi converged? CG + adaptive Jacobi converged?



Iterations (adaptive) Time (adaptive) CG converged? CG + Jacobi converged? CG + adaptive Jacobi converged?

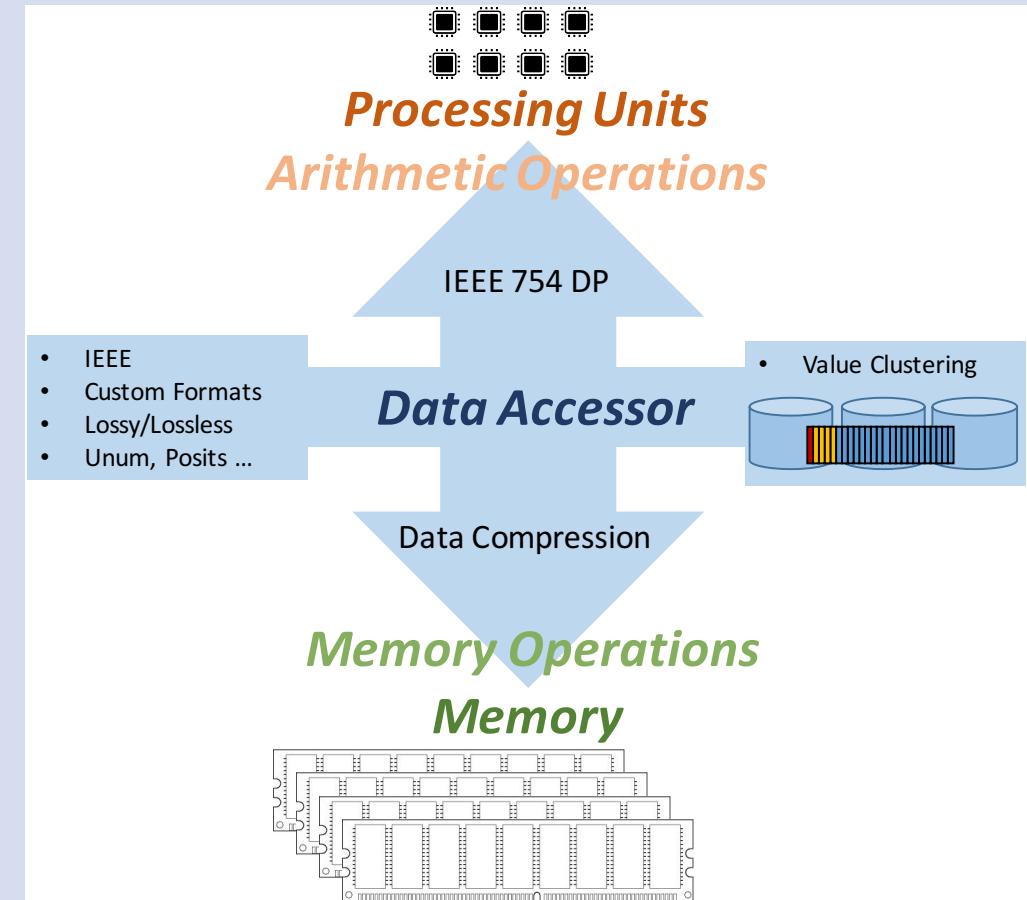




# The Communication Bottleneck

Radically decouple storage format from arithmetic format.

- The arithmetic operations should use high precision formats natively supported by hardware.
- Data access should be as cheap as possible, reduced precision.
- Consider a wide range of memory formats:
  - IEEE standard precision formats
  - Customized formats (configuring mantissa/exponent)
  - Lossy compression
  - ...



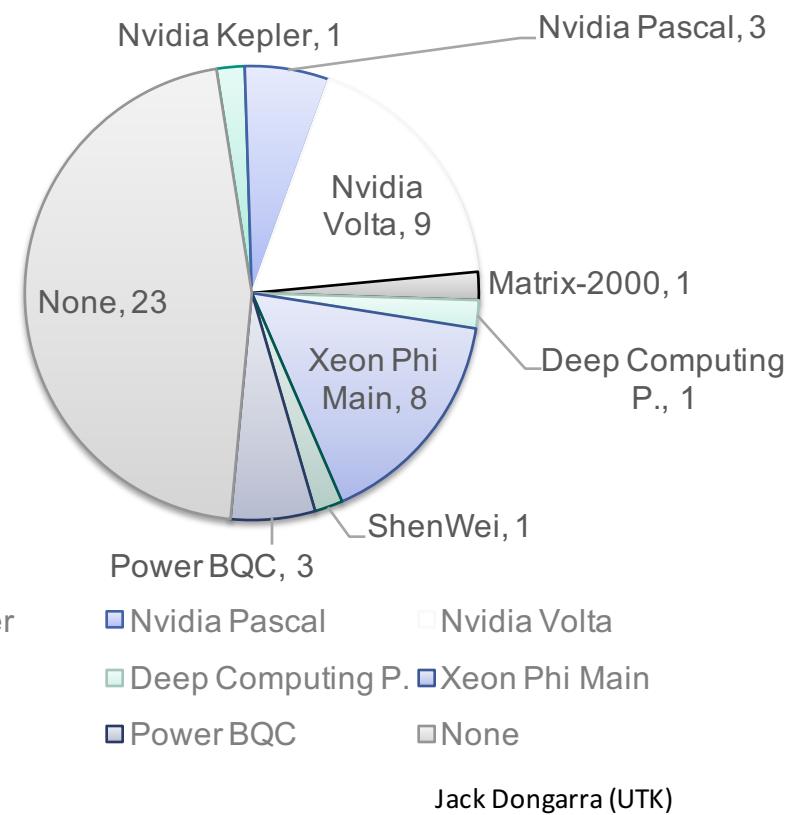
Copyright@ORNL

<sup>1</sup>SuiteSparse Matrix Collection: <https://sparse.tamu.edu/>

# How to deal with the Manycore Parallelism?

- Increasing adoption of manycore accelerators
  - partly motivated by the Machine Learning excitement;
- Integration of low-precision tensor units;
- The GPU streaming model is dominating;
- Algorithms need **fine-grained parallelism**
  - **thousands of SIMT threads!**
- Global **synchronizations** are killing performance;
- Runtime scheduling of thread blocks **virtually impossible**;
- Memory access pattern central (coalesced data access);
- Asynchronous algorithms needed;
- Reformulation as fixed-point iteration;

Accelerator share in the TOP50 systems [Jun 2019]



## Spotlight Example: Incomplete Sparse Factorizations

We are looking for a factorization-based preconditioner such that  $A \approx \textcolor{blue}{L} \cdot \textcolor{magenta}{U}$ .  
is a good approximation with moderate nonzero count (e.g.  $\text{nnz}(L + U) = \text{nnz}(A)$  ).

$$\begin{pmatrix} \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} \\ \textcolor{blue}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} \\ \textcolor{blue}{x} & \textcolor{blue}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} \\ \textcolor{blue}{x} & \textcolor{blue}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} \\ \textcolor{blue}{x} & \textcolor{blue}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} \\ \textcolor{blue}{x} & \textcolor{blue}{x} & \textcolor{blue}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} \\ \textcolor{blue}{x} & \textcolor{blue}{x} & \textcolor{blue}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} \\ \textcolor{blue}{x} & \textcolor{blue}{x} & \textcolor{blue}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} \end{pmatrix}$$

## Spotlight Example: Incomplete Sparse Factorizations

We are looking for a factorization-based preconditioner such that  $A \approx L \cdot U$ .  
is a good approximation with moderate nonzero count (e.g.  $nnz(L + U) = nnz(A)$  ).

- *Where should these nonzero elements be located?*
- *How can we compute the preconditioner in a highly parallel fashion?*

$$\begin{pmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & & & \\ \times & \times & \times & \times & & \\ \times & & \times & \times & \times & \\ \times & & \times & \times & \times & \times \\ \times & & \times & \times & \times & \times \\ \times & & \times & \times & \times & \times \\ \times & & & \times & \times & \times \end{pmatrix}$$

$$\mathcal{S}(A) = \{(i, j) \in \mathbb{N}^2 : A_{ij} \neq 0\}$$

# Spotlight Example: Incomplete Sparse Factorizations

We are looking for a factorization-based preconditioner such that  $A \approx L \cdot U$ .  
is a good approximation with moderate nonzero count (e.g.  $nnz(L + U) = nnz(A)$  ).

- *Where should these nonzero elements be located?*
- *How can we compute the preconditioner in a highly parallel fashion?*

$$\begin{pmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & & & \\ \times & \times & \times & \times & & \\ \times & \times & \times & \times & \times & \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \end{pmatrix}$$

$$\mathcal{S}(A) = \{(i, j) \in \mathbb{N}^2 : A_{ij} \neq 0\}$$

## Exact LU Factorization

- Decompose system matrix into product  $A = L \cdot U$ .
- Based on Gaussian elimination.
- Triangular solves to solve a system  $Ax = b$ :

$$Ly = b \Rightarrow y \quad \Rightarrow \quad Ux = y \Rightarrow x$$

- De-Facto standard for solving dense problems.
- *What about sparse? Often significant fill-in...*

# Spotlight Example: Incomplete Sparse Factorizations

We are looking for a factorization-based preconditioner such that  $A \approx L \cdot U$ .  
is a good approximation with moderate nonzero count (e.g.  $nnz(L + U) = nnz(A)$  ).

- *Where should these nonzero elements be located?*
- *How can we compute the preconditioner in a highly parallel fashion?*

$$\begin{pmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & & & \\ \times & \times & \times & \times & & \\ \times & \times & \times & \times & \times & \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \end{pmatrix}$$

## Exact LU Factorization

- Decompose system matrix into product  $A = L \cdot U$ .
- Based on Gaussian elimination.
- Triangular solves to solve a system  $Ax = b$ :

$$Ly = b \Rightarrow y \quad \Rightarrow \quad Ux = y \Rightarrow x$$

- De-Facto standard for solving dense problems.
- *What about sparse? Often significant fill-in...*

## Incomplete LU Factorization (ILU)

- **Focused on restricting fill-in to a specific sparsity pattern  $\mathcal{S}$ .**

# Spotlight Example: Incomplete Sparse Factorizations

We are looking for a factorization-based preconditioner such that  $A \approx L \cdot U$ .  
is a good approximation with moderate nonzero count (e.g.  $nnz(L + U) = nnz(A)$  ).

- *Where should these nonzero elements be located?*
- *How can we compute the preconditioner in a highly parallel fashion?*

$$\begin{pmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & & & \\ \times & \times & \times & \times & & \\ \times & & \times & \times & \times & \\ \times & & \times & \times & \times & \times \\ \times & & & \times & \times & \times \\ \times & & & \times & \times & \times \\ \times & & & & \times & \times \\ \times & & & & & \times \end{pmatrix}$$

## Exact LU Factorization

- Decompose system matrix into product  $A = L \cdot U$ .
- Based on Gaussian elimination.
- Triangular solves to solve a system  $Ax = b$ :

$$Ly = b \Rightarrow y \quad \Rightarrow \quad Ux = y \Rightarrow x$$

- De-Facto standard for solving dense problems.
- *What about sparse? Often significant fill-in...*

## Incomplete LU Factorization (ILU)

- **Focused on restricting fill-in** to a specific sparsity pattern  $\mathcal{S}$ .
- For **ILU(0),  $\mathcal{S}$**  is the sparsity pattern of  $A$ .
  - Works well for many problems.
  - *Is this the best we can get for nonzero count?*

# Spotlight Example: Incomplete Sparse Factorizations

We are looking for a factorization-based preconditioner such that  $A \approx L \cdot U$ .  
is a good approximation with moderate nonzero count (e.g.  $nnz(L + U) = nnz(A)$  ).

- *Where should these nonzero elements be located?*
- *How can we compute the preconditioner in a highly parallel fashion?*

$$\begin{pmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & & & \\ \times & \times & \times & \times & & \\ \times & \times & \times & \times & \times & \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \end{pmatrix}$$

## Exact LU Factorization

- Decompose system matrix into product  $A = L \cdot U$ .
- Based on Gaussian elimination.
- Triangular solves to solve a system  $Ax = b$ :

$$Ly = b \Rightarrow y \quad \Rightarrow \quad Ux = y \Rightarrow x$$

- De-Facto standard for solving dense problems.
- *What about sparse? Often significant fill-in...*

## Incomplete LU Factorization (ILU)

- **Focused on restricting fill-in** to a specific sparsity pattern  $\mathcal{S}$ .
- For **ILU(0)**,  $\mathcal{S}$  is the sparsity pattern of  $A$ .
  - Works well for many problems.
  - *Is this the best we can get for nonzero count?*
- Fill-in threshold ILU (**ILUT**) bases  $\mathcal{S}$  on the significance of elements (e.g. magnitude).
  - Often **better preconditioners** than level-based ILU.
  - Difficult to parallelize.

# Spotlight Example: Incomplete Sparse Factorizations

We are looking for a factorization-based preconditioner such that  $A \approx L \cdot U$ .  
is a good approximation with moderate nonzero count (e.g.  $nnz(L + U) = nnz(A)$  ).

- *Where should these nonzero elements be located?*
- *How can we compute the preconditioner in a highly parallel fashion?*

$$\begin{pmatrix} \times & \times & \times & \times & & \times & \times \\ \times & \times & \times & & & & \\ \times & \times & \times & \times & & & \\ \times & & \times & \times & \times & & \\ \times & & \times & \times & \times & & \\ \times & & \times & \times & \times & & \\ \times & & \times & \times & \times & & \\ \times & & & \times & \times & & \\ \end{pmatrix}$$

## Rethink the overall strategy!

- Use a parallel iterative process to generate factors.
- The preconditioner should have a moderate number of nonzero elements,  
*but we don't care too much about intermediate data.*

# Spotlight Example: Incomplete Sparse Factorizations

We are looking for a factorization-based preconditioner such that  $A \approx L \cdot U$ .  
is a good approximation with moderate nonzero count (e.g.  $nnz(L + U) = nnz(A)$  ).

- *Where should these nonzero elements be located?*
- *How can we compute the preconditioner in a highly parallel fashion?*

$$\begin{pmatrix} \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} \\ \textcolor{blue}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} \\ \textcolor{blue}{x} & \textcolor{blue}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} \\ \textcolor{blue}{x} & \textcolor{blue}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} \\ \textcolor{blue}{x} & \textcolor{blue}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} \\ \textcolor{blue}{x} & \textcolor{blue}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} \\ \textcolor{blue}{x} & \textcolor{blue}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} \\ \textcolor{blue}{x} & \textcolor{blue}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} & \textcolor{magenta}{x} \end{pmatrix}$$

## Rethink the overall strategy!

- Use a parallel iterative process to generate factors.
- The preconditioner should have a moderate number of nonzero elements,  
*but we don't care too much about intermediate data.*

1. *Select a set of nonzero locations.*
2. *Compute values in those locations such that  $A \approx L \cdot U$  is a “good” approximation.*
3. *Maybe change some locations in favor of locations that result in a better preconditioner.*
4. *Repeat until the preconditioner quality does no longer improve for the nonzero count.*

# Considerations

1. Select a set of nonzero locations.
2. Compute values in those locations such that  $A \approx L \cdot U$  is a “good” approximation.
3. Maybe change some locations in favor of locations that result in a better preconditioner.
4. Repeat until the preconditioner quality stagnates.

- This is an optimization problem...

**ILU residual**  $R = A - L \times U$

$$\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & & \\ * & & * & & \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} - \begin{pmatrix} * & & & & \\ * & * & & & \\ * & * & * & & \\ * & & * & & \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} \times \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix}$$

# Considerations

1. Select a set of nonzero locations.
2. Compute values in those locations such that  $A \approx L \cdot U$  is a “good” approximation.
3. Maybe change some locations in favor of locations that result in a better preconditioner.
4. Repeat until the preconditioner quality stagnates.

- This is an optimization problem...

**ILU residual**  $R = A - L \times U$

$$R = \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & & \\ * & & * & & \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} - \begin{pmatrix} * & & & & \\ * & * & & & \\ * & * & * & & \\ * & & & * & \\ * & * & & * & \\ * & * & * & * & * \end{pmatrix} \times \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix}$$

The matrix  $L$  has a red box highlighting its second column. The matrix  $U$  has a red box highlighting its second row. The resulting matrix  $R$  has a red box highlighting its second column, with a red star in the fourth row, fourth column position.

# Considerations

1. Select a set of nonzero locations.
2. Compute values in those locations such that  $A \approx L \cdot U$  is a “good” approximation.
3. Maybe change some locations in favor of locations that result in a better preconditioner.
4. Repeat until the preconditioner quality stagnates.

- This is an optimization problem...

**ILU residual**  $R = A - L \times U$

$$R = \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & & \\ * & & * & & \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} - \begin{pmatrix} * & & & & \\ * & * & & & \\ * & * & * & & \\ * & * & * & * & \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} \times \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix}$$

The diagram shows the ILU factorization equation  $R = A - L \times U$ . The matrix  $A$  is represented by a grid of asterisks (\*). The matrices  $L$  and  $U$  are shown as triangular matrices with asterisks. A red rectangle highlights a 2x3 submatrix in the  $L$  matrix, and a red box highlights a 2x2 submatrix in the  $U$  matrix. A red star is placed at the top-right corner of the  $U$  matrix's highlighted submatrix.

$$\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & & \\ * & & * & & \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} - \begin{pmatrix} * & & & & \\ * & * & & & \\ * & * & * & & \\ * & * & * & * & \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} \times \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix}$$
$$\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & & * & & \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix}$$

# Considerations

1. Select a set of nonzero locations.
2. Compute values in those locations such that  $A \approx L \cdot U$  is a “good” approximation.
3. Maybe change some locations in favor of locations that result in a better preconditioner.
4. Repeat until the preconditioner quality stagnates.

- This is an optimization problem...

**ILU residual**  $R = A - L \times U$

$$\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} - \begin{pmatrix} * & & & & \\ * & * & & & \\ * & * & * & & \\ * & * & * & * & \\ * & * & * & * & * \end{pmatrix} \times \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix}$$
$$\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix}$$

# Considerations

1. Select a set of nonzero locations.
2. Compute values in those locations such that  $A \approx L \cdot U$  is a “good” approximation.
3. Maybe change some locations in favor of locations that result in a better preconditioner.
4. Repeat until the preconditioner quality stagnates.

- This is an optimization problem...

**ILU residual**     $R = A - L \times U$

$$\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} - \begin{pmatrix} * & & & & \\ * & * & & & \\ * & * & * & & \\ * & * & * & * & \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} \times \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix}$$

**ILU residual matrix pattern**

$$\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} = \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} - \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix}$$

# Considerations

1. Select a set of nonzero locations.
2. Compute values in those locations such that  $A \approx L \cdot U$  is a “good” approximation.
3. Maybe change some locations in favor of locations that result in a better preconditioner.
4. Repeat until the preconditioner quality stagnates.

- This is an optimization problem with  $\text{nnz}(A - L \cdot U)$  equations and  $\text{nnz}(L + U)$  variables.

$$\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & & * \\ * & & * & & * \\ * & * & & * & * \\ * & * & & * & * \end{pmatrix} - \begin{pmatrix} * & & & & \\ * & * & & & \\ * & * & * & & \\ * & & & * & \\ * & * & & * & * \\ * & * & & * & * \end{pmatrix} \times \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & & * & * & * \\ * & * & * & * & * \end{pmatrix}$$
$$\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} = \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & & * \\ * & & * & & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} - \begin{pmatrix} * & & & & \\ * & * & & & \\ * & * & * & & \\ * & & & * & \\ * & * & & * & * \\ * & * & & * & * \end{pmatrix} \times \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & & * & * & * \\ * & * & * & * & * \end{pmatrix}$$

Sparsity pattern  $\mathcal{S}$

# Considerations

1. Select a set of nonzero locations.
2. Compute values in those locations such that  $A \approx L \cdot U$  is a “good” approximation.
3. Maybe change some locations in favor of locations that result in a better preconditioner.
4. Repeat until the preconditioner quality stagnates.

- This is an optimization problem with  $\text{nnz}(A - L \cdot U)$  equations and  $\text{nnz}(L + U)$  variables.
- We may want to compute the values in  $L, U$  such that  $R = A - L \cdot U = 0|_{\mathcal{S}}$ , the approximation being exact in the locations included in  $\mathcal{S}$ , but not outside!

$\text{nnz}(L + U)$  equations  
 $\text{nnz}(L + U)$  variables

$$\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} = \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} - \begin{pmatrix} * & & & & \\ * & * & & & \\ * & * & * & & \\ * & * & * & * & \\ * & * & * & * & * \end{pmatrix} \times \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} \quad \text{Sparsity pattern } \mathcal{S}$$

# Considerations

1. Select a set of nonzero locations.
2. Compute values in those locations such that  $A \approx L \cdot U$  is a “good” approximation.
3. Maybe change some locations in favor of locations that result in a better preconditioner.
4. Repeat until the preconditioner quality stagnates.

- This is an optimization problem with  $\text{nnz}(A - L \cdot U)$  equations and  $\text{nnz}(L + U)$  variables.
- We may want to compute the values in  $L, U$  such that  $R = A - L \cdot U = 0|_{\mathcal{S}}$ , the approximation being exact in the locations included in  $\mathcal{S}$ , but not outside!
- This is the underlying idea of Edmond Chow’s parallel ILU algorithm<sup>1</sup>:

$$L \cdot U = A|_{\mathcal{S}} \quad \Rightarrow \quad F(l_{ij}, u_{ij}) = \begin{cases} \frac{1}{u_{jj}} \left( a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right), & i > j \\ a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}, & i \leq j \end{cases}$$

<sup>1</sup>Chow and Patel. “Fine-grained Parallel Incomplete LU Factorization”. In: SIAM J. on Sci. Comp. (2015).

# Considerations

1. Select a set of nonzero locations.
2. Compute values in those locations such that  $A \approx L \cdot U$  is a “good” approximation.
3. Maybe change some locations in favor of locations that result in a better preconditioner.
4. Repeat until the preconditioner quality stagnates.

- This is an optimization problem with  $\text{nnz}(A - L \cdot U)$  equations and  $\text{nnz}(L + U)$  variables.
- We may want to compute the values in  $L, U$  such that  $R = A - L \cdot U = 0|_{\mathcal{S}}$ , the approximation being exact in the locations included in  $\mathcal{S}$ , but not outside!
- This is the underlying idea of Edmond Chow’s parallel ILU algorithm<sup>1</sup>:

$$L \cdot U = A|_{\mathcal{S}} \quad \Rightarrow \quad F(l_{ij}, u_{ij}) = \begin{cases} \frac{1}{u_{jj}} \left( a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right), & i > j \\ a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}, & i \leq j \end{cases}$$

- Converges in the asymptotic sense towards incomplete factors  $L, U$  such that  $R = A - L \cdot U = 0|_{\mathcal{S}}$

<sup>1</sup>Chow and Patel. “Fine-grained Parallel Incomplete LU Factorization”. In: SIAM J. on Sci. Comp. (2015).

# Considerations

1. Select a set of nonzero locations.
2. Compute values in those locations such that  $A \approx L \cdot U$  is a “good” approximation.
3. Maybe change some locations in favor of locations that result in a better preconditioner.
4. Repeat until the preconditioner quality stagnates.

- This is an optimization problem with  $\text{nnz}(A - L \cdot U)$  equations and  $\text{nnz}(L + U)$  variables.
- We may want to compute the values in  $L, U$  such that  $R = A - L \cdot U = 0|_{\mathcal{S}}$ , the approximation being exact in the locations included in  $\mathcal{S}$ , but not outside!
- This is the underlying idea of Edmond Chow’s parallel ILU algorithm<sup>1</sup>:

$$L \cdot U = A|_{\mathcal{S}} \quad \Rightarrow \quad F(l_{ij}, u_{ij}) = \begin{cases} \frac{1}{u_{jj}} \left( a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right), & i > j \\ a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}, & i \leq j \end{cases}$$

- Converges in the asymptotic sense towards incomplete factors  $L, U$  such that  $R = A - L \cdot U = 0|_{\mathcal{S}}$

## ParILU Algorithm

- Fixed-Point based algorithm for computing ILU;
- Fine-grained parallelism and asynchronous execution;
- Faster than Level-Scheduling
- Outperforms NVIDIA’s cuSPARSE ILU

| Matrix<br>(UFMC) | NVIDIA<br>cuSPARSE | ParILU  | Speedup     |
|------------------|--------------------|---------|-------------|
| APA              | 61. ms             | 8.8 ms  | <b>6.9</b>  |
| ECO              | 107. ms            | 6.7 ms  | <b>16.0</b> |
| G3               | 110. ms            | 12.1 ms | <b>9.1</b>  |
| OFF              | 219. ms            | 25.1 ms | <b>8.7</b>  |
| PAR              | 131. ms            | 6.1 ms  | <b>21.6</b> |
| THM              | 454. ms            | 15.7 ms | <b>28.9</b> |
| L2D              | 112. ms            | 7.4 ms  | <b>15.2</b> |
| L3D              | 94. ms             | 47.5 ms | <b>2.0</b>  |

Chow, Anzt, Dongarra, ISC 2015

<sup>1</sup>Chow and Patel. “Fine-grained Parallel Incomplete LU Factorization”. In: SIAM J. on Sci. Comp. (2015).

# Considerations

1. Select a set of nonzero locations.
2. Compute values in those locations such that  $A \approx L \cdot U$  is a “good” approximation.
3. Maybe change some locations in favor of locations that result in a better preconditioner.
4. Repeat until the preconditioner quality stagnates.

- This is an optimization problem with  $\text{nnz}(A - L \cdot U)$  equations and  $\text{nnz}(L + U)$  variables.
- We may want to compute the values in  $L, U$  such that  $R = A - L \cdot U = 0|_{\mathcal{S}}$ , the approximation being exact in the locations included in  $\mathcal{S}$ , but not outside!
- This is the underlying idea of Edmond Chow’s parallel ILU algorithm<sup>1</sup>:

$$L \cdot U = A|_{\mathcal{S}} \quad \Rightarrow \quad F(l_{ij}, u_{ij}) = \begin{cases} \frac{1}{u_{jj}} \left( a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right), & i > j \\ a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}, & i \leq j \end{cases}$$

- We may not need high accuracy here, because we may change the pattern again... One single fixed-point sweep.

Fixed-point sweep approximates incomplete factors.

<sup>1</sup>Chow and Patel. “Fine-grained Parallel Incomplete LU Factorization”. In: SIAM J. on Sci. Comp. (2015).

# Considerations

1. Select a set of nonzero locations.
2. Compute values in those locations such that  $A \approx L \cdot U$  is a “good” approximation.
3. Maybe change some locations in favor of locations that result in a better preconditioner.
4. Repeat until the preconditioner quality stagnates.

Compute ILU residual & check convergence.

- Maybe use the ILU residual norm as quality metric.

**ILU residual**     $R = A - L \times U$

$$\begin{pmatrix} * & * & * & * & * \\ * & * & * & \color{red}{*} & * \\ * & * & * & \color{red}{*} & \color{red}{*} \\ * & \color{red}{*} & \color{red}{*} & * & \color{red}{*} \\ * & \color{red}{*} & * & * & * \\ * & * & \color{red}{*} & * & * \\ * & * & \color{red}{*} & \color{red}{*} & * \end{pmatrix} = \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} - \begin{pmatrix} * & & & & \\ \color{blue}{*} & * & & & \\ \color{blue}{*} & \color{blue}{*} & * & & \\ \color{blue}{*} & \color{blue}{*} & \color{blue}{*} & * & \\ \color{blue}{*} & \color{blue}{*} & \color{blue}{*} & \color{blue}{*} & * \end{pmatrix} \times \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix}$$

Fixed-point sweep approximates incomplete factors.

# Considerations

1. *Select a set of nonzero locations.*
2. *Compute values in those locations such that  $A \approx L \cdot U$  is a “good” approximation.*
3. *Maybe change some locations in favor of locations that result in a better preconditioner.*
4. *Repeat until the preconditioner quality stagnates.*

- The sparsity pattern of  $A$  might be a **good initial start** for nonzero locations.

Compute ILU residual & check convergence.

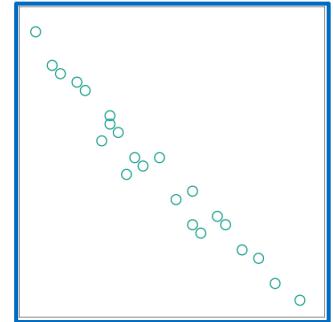
Fixed-point sweep approximates incomplete factors.

# Considerations

1. Select a set of nonzero locations.
2. Compute values in those locations such that  $A \approx L \cdot U$  is a “good” approximation.
3. **Maybe change some locations in favor of locations that result in a better preconditioner.**
4. Repeat until the preconditioner quality stagnates.

Identify locations with nonzero ILU residual.

Compute ILU residual & check convergence.



- The sparsity pattern of  $A$  might be a **good initial start** for nonzero locations.
- Then, the approximation will be exact for all locations  $\mathcal{S}_0 = \mathcal{S}(L_0 + U_0)$  and nonzero in locations  $\mathcal{S}_1 = (\mathcal{S}(A) \cup \mathcal{S}(L_0 \cdot U_0)) \setminus \mathcal{S}(L_0 + U_0)$ <sup>1</sup>.

$$\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} - \begin{pmatrix} * & & & & \\ * & * & & & \\ * & * & * & & \\ * & * & * & * & \\ * & * & * & * & * \end{pmatrix} \times \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} = \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix}$$

Fixed-point sweep approximates incomplete factors.

<sup>1</sup>Saad. “Iterative Methods for Sparse Linear Systems, 2<sup>nd</sup> Edition”. (2003).

# Considerations

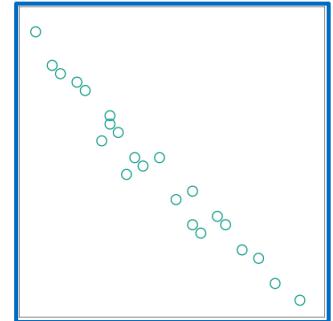
1. Select a set of nonzero locations.
2. Compute values in those locations such that  $A \approx L \cdot U$  is a “good” approximation.
3. **Maybe change some locations in favor of locations that result in a better preconditioner.**
4. Repeat until the preconditioner quality stagnates.

- The sparsity pattern of  $A$  might be a **good initial start** for nonzero locations.
- Then, the approximation will be exact for all locations  $\mathcal{S}_0 = \mathcal{S}(L_0 + U_0)$  and nonzero in locations  $\mathcal{S}_1 = (\mathcal{S}(A) \cup \mathcal{S}(L_0 \cdot U_0)) \setminus \mathcal{S}(L_0 + U_0)$ <sup>1</sup>.
- Adding all these locations (**level-fill!**) might be good idea...

$$\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} - \begin{pmatrix} * & & & & \\ * & * & & & \\ * & * & * & & \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} \times \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix}$$

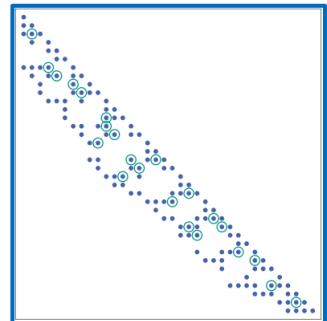
Identify locations with nonzero ILU residual.

Compute ILU residual & check convergence.



Add locations to sparsity pattern of incomplete factors.

Fixed-point sweep approximates incomplete factors.



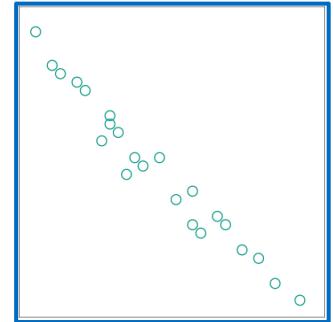
# Considerations

1. Select a set of nonzero locations.
2. Compute values in those locations such that  $A \approx L \cdot U$  is a “good” approximation.
3. **Maybe change some locations in favor of locations that result in a better preconditioner.**
4. Repeat until the preconditioner quality stagnates.

- The sparsity pattern of  $A$  might be a **good initial start** for nonzero locations.
- Then, the approximation will be exact for all locations  $\mathcal{S}_0 = \mathcal{S}(L_0 + U_0)$  and nonzero in locations  $\mathcal{S}_1 = (\mathcal{S}(A) \cup \mathcal{S}(L_0 \cdot U_0)) \setminus \mathcal{S}(L_0 + U_0)$ <sup>1</sup>.
- Adding all these locations (**level-fill!**) might be good idea, **but adding these will again generate new nonzero residuals**  $\mathcal{S}_2 = (\mathcal{S}(A) \cup \mathcal{S}(L_1 \cdot U_1)) \setminus \mathcal{S}(L_1 + U_1)$

$$\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} - \begin{pmatrix} * & & & & \\ * & * & & & \\ * & * & * & & \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} \times \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix} = \begin{pmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \end{pmatrix}$$

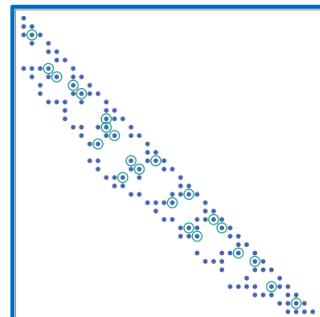
Fixed-point sweep approximates incomplete factors.



Identify locations with nonzero ILU residual.

Compute ILU residual & check convergence.

Add locations to sparsity pattern of incomplete factors.

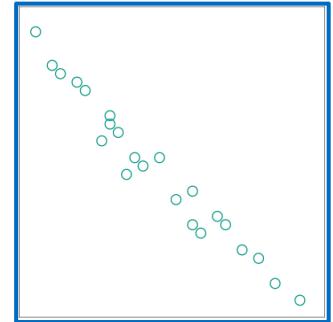


# Considerations

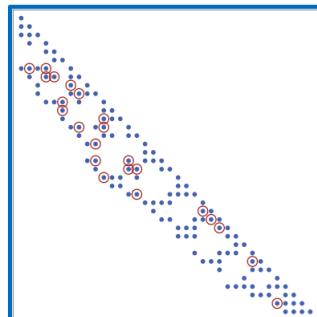
1. Select a set of nonzero locations.
  2. Compute values in those locations such that  $A \approx L \cdot U$  is a “good” approximation.
  3. **Maybe change some locations in favor of locations that result in a better preconditioner.**
  4. Repeat until the preconditioner quality stagnates.
- At some point we should remove some locations again, e.g. the smallest elements, and start over looking at locations  $R = A - L_k \cdot U_k \dots$

Identify locations with nonzero ILU residual.

Compute ILU residual & check convergence.



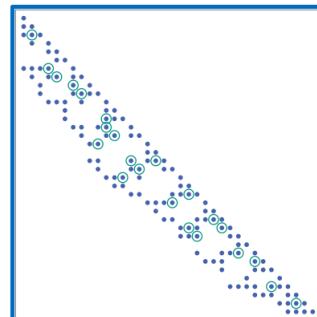
Remove smallest elements from incomplete factors.



Select a threshold separating smallest elements.

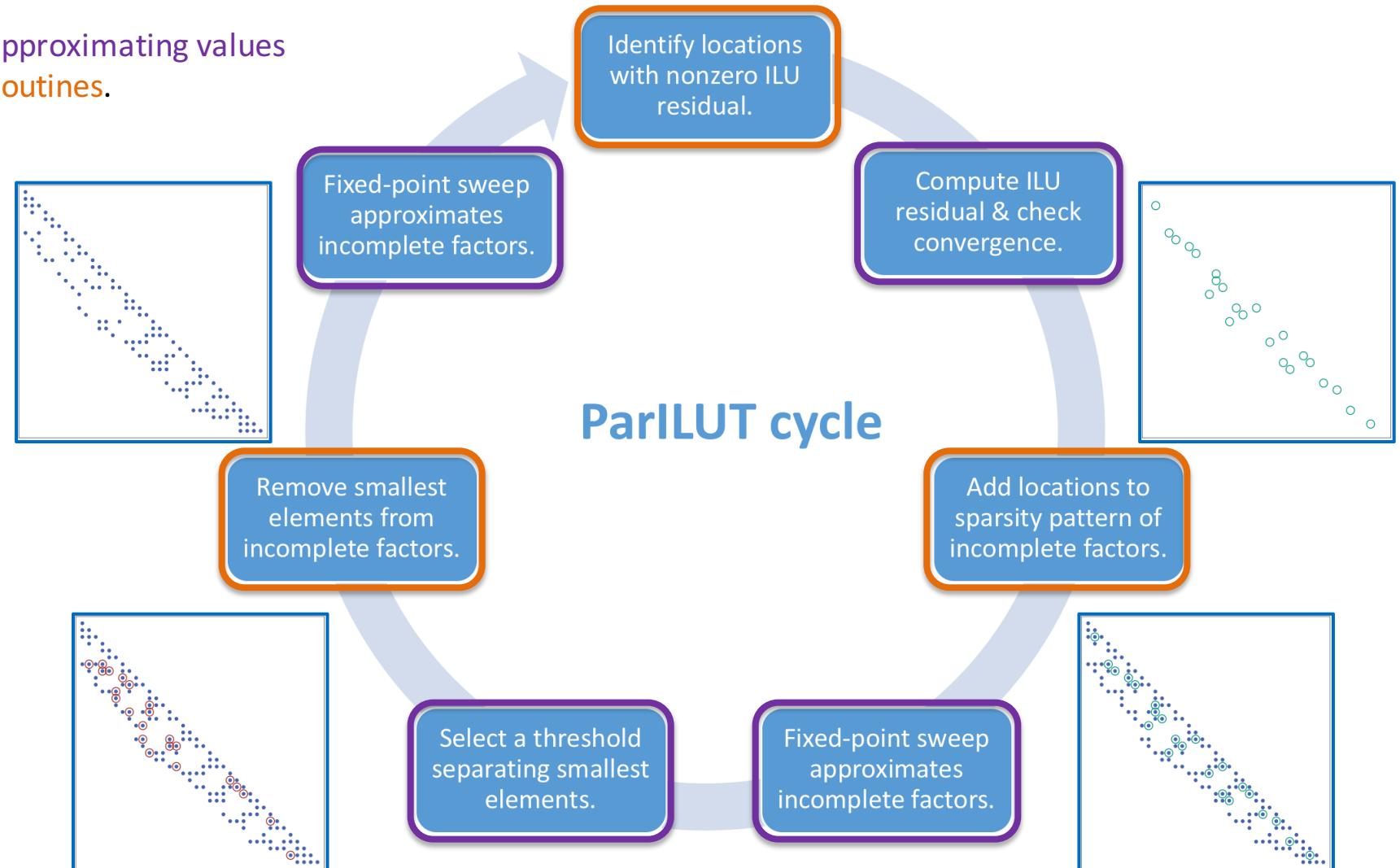
Fixed-point sweep approximates incomplete factors.

Add locations to sparsity pattern of incomplete factors.



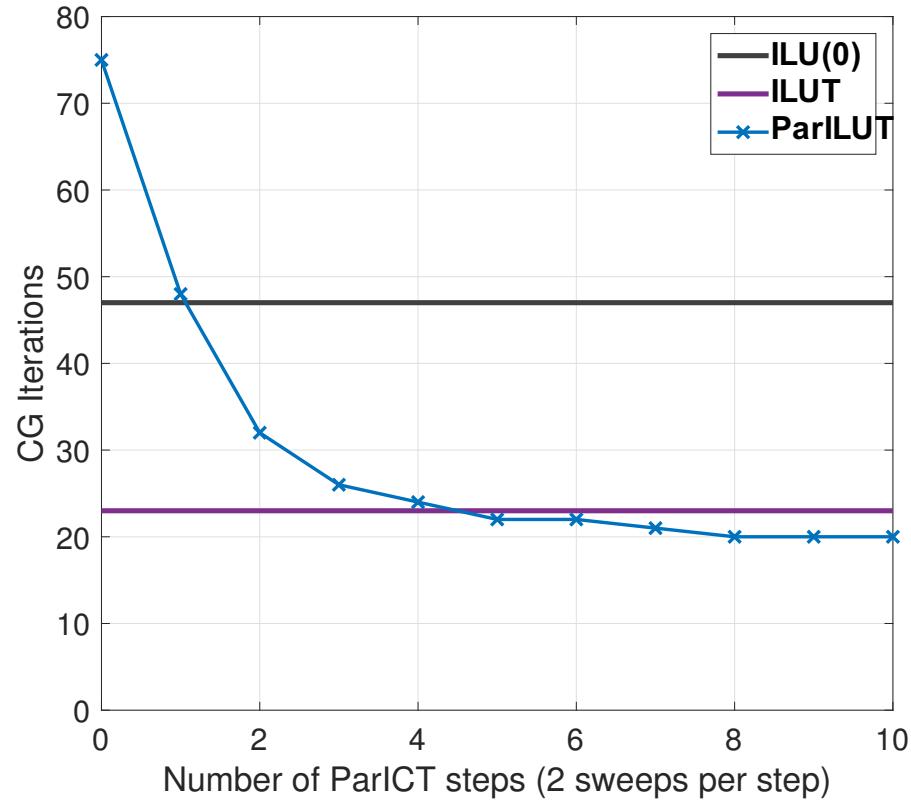
# ParILUT Algorithm

Interleaving fixed-point sweeps approximating values with pattern-changing symbolic routines.



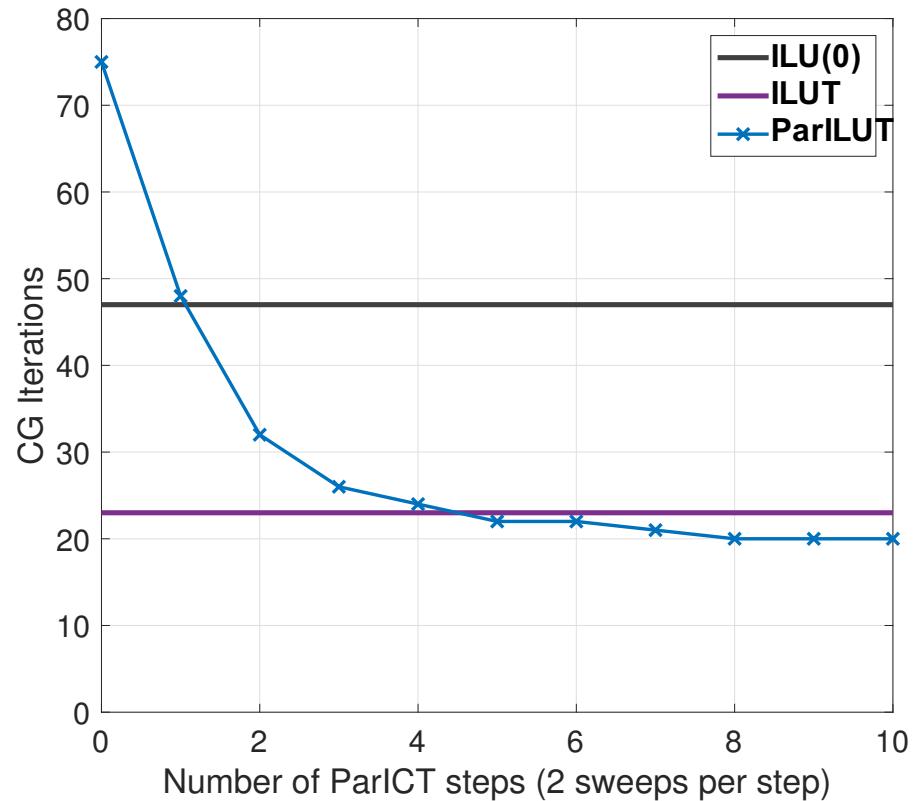
# ParILUT Quality

Anisotropic diffusion problem  
n: 741, nz: 4,951



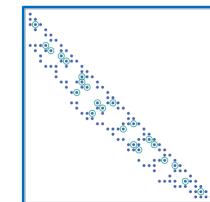
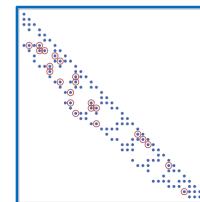
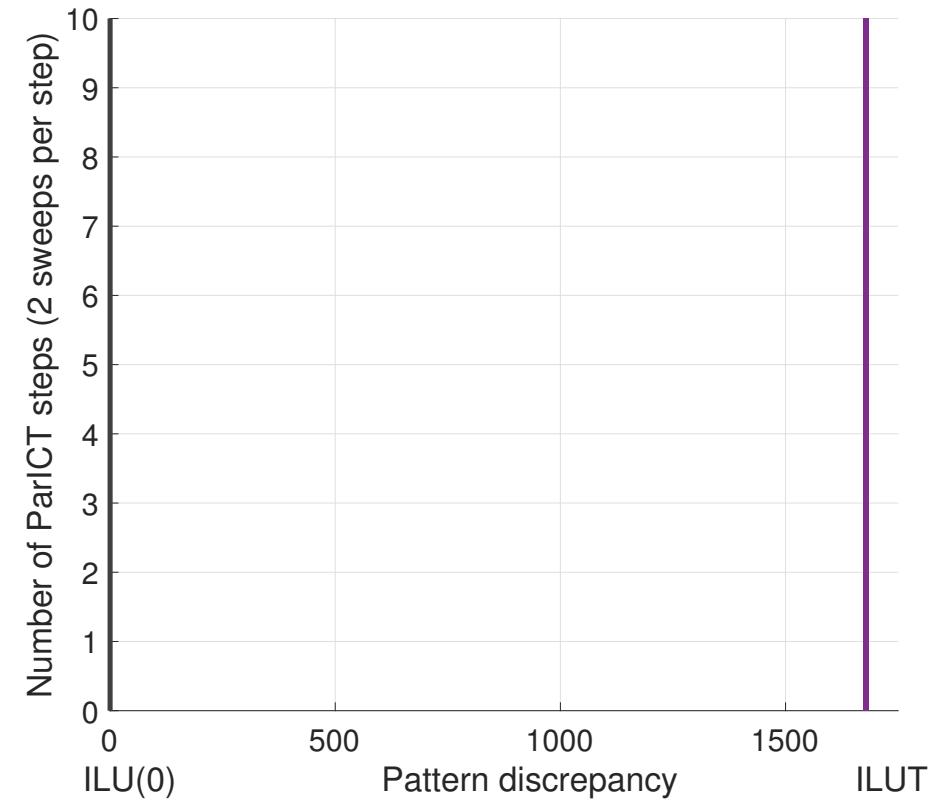
- Top-level solver iterations as quality metric.
- Few sweeps give a “better” preconditioner than ILU(0).
- Better than conventional ILUT?

# ParILUT Quality

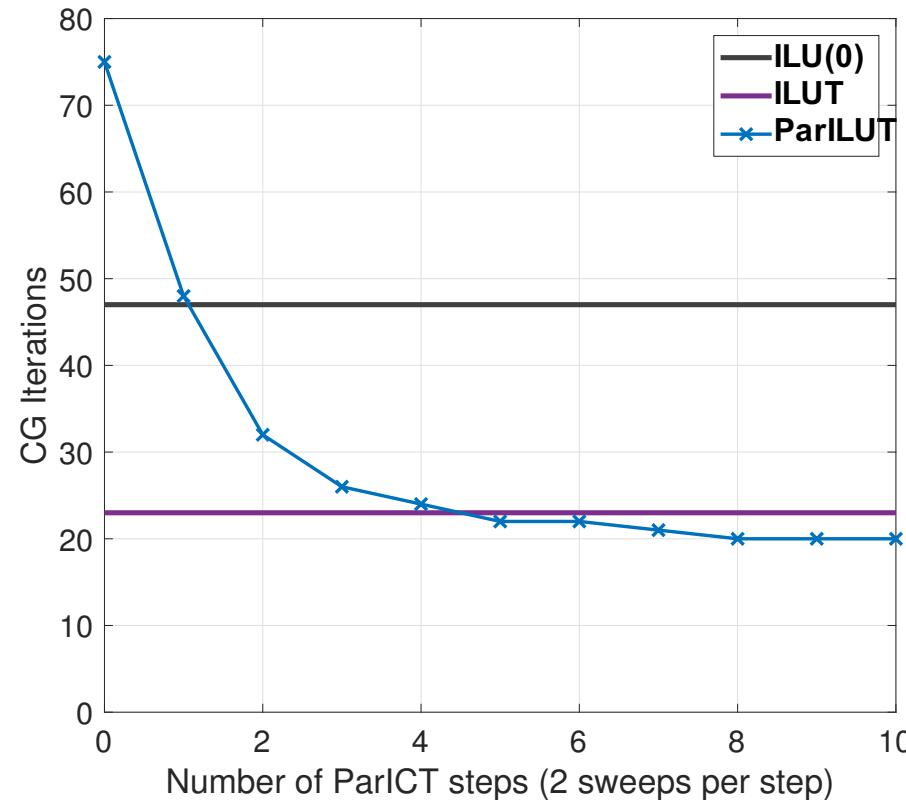


- Top-level solver iterations as quality metric.
- Few sweeps give a “better” preconditioner than ILU(0).
- Better than conventional ILUT?

Anisotropic diffusion problem  
n: 741, nz: 4,951

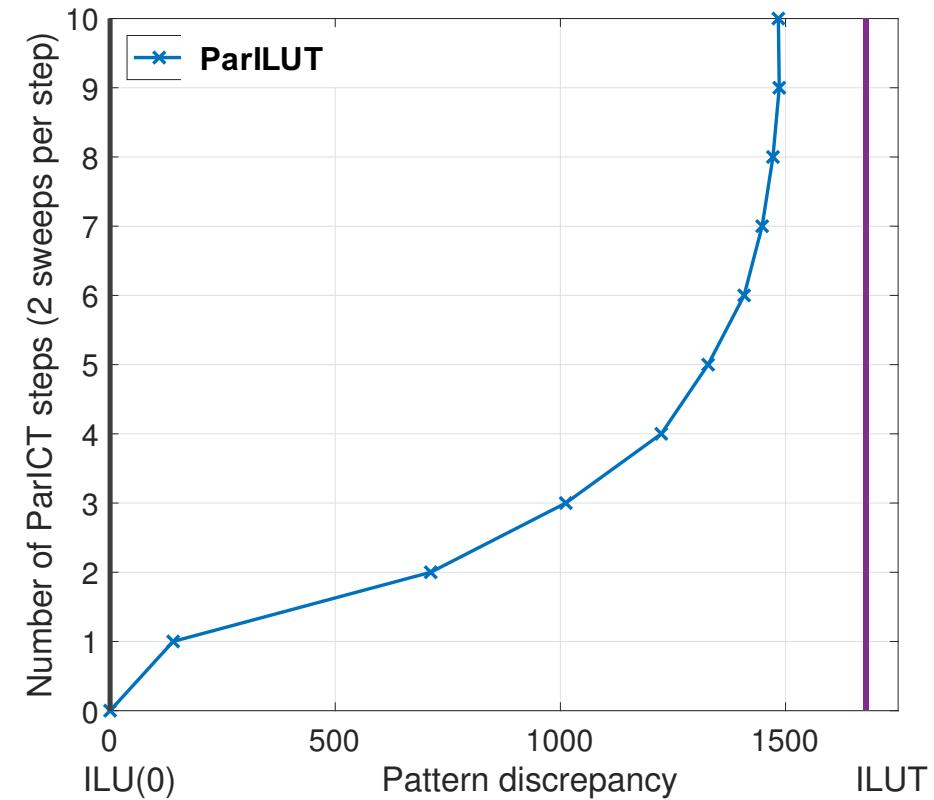


# ParILUT Quality



- Top-level solver iterations as quality metric.
- Few sweeps give a “better” preconditioner than ILU(0).
- Better than conventional ILUT?

Anisotropic diffusion problem  
n: 741, nz: 4,951

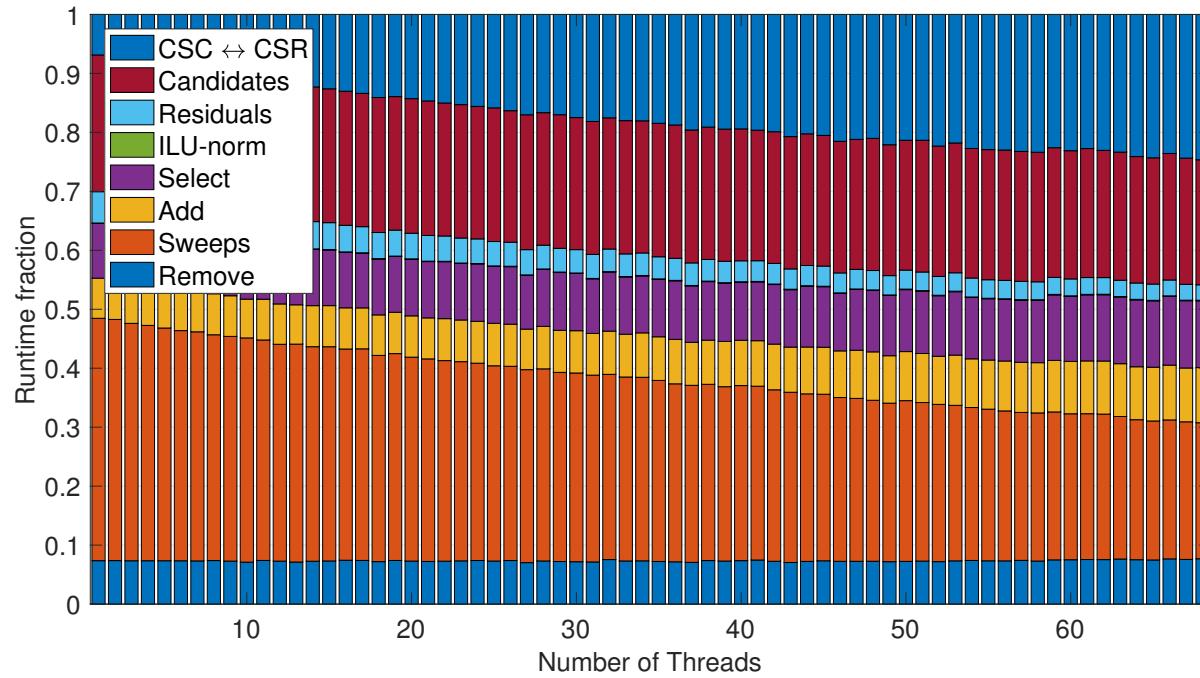
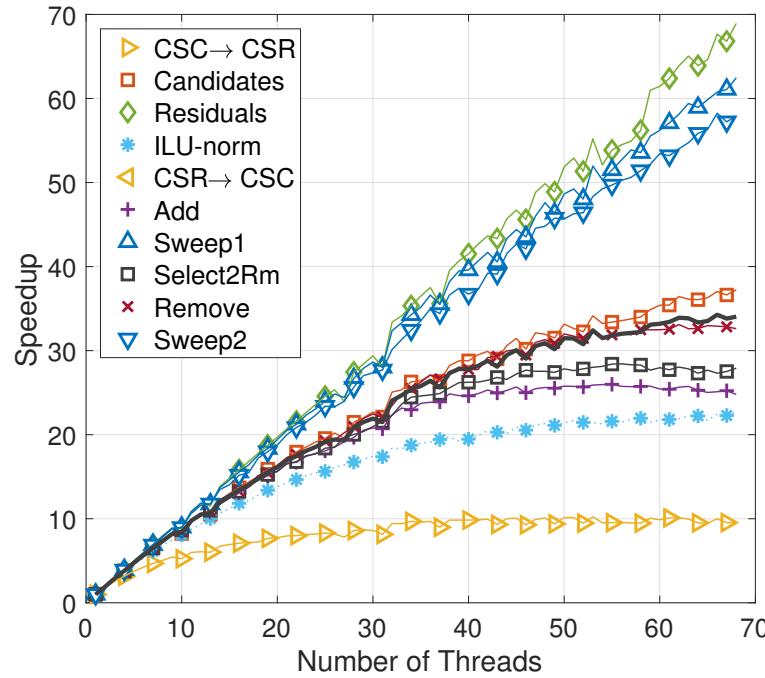


- Pattern converges after few sweeps.
- Pattern “more like” ILUT than ILU(0).

# ParILUT Scalability

thermal2 matrix from SuiteSparse, RCM ordering, 8 el/row.

Intel Xeon Phi 7250 "Knights Landing"  
68 cores @1.40 GHz,  
16GB MCDRAM @490 GB/s

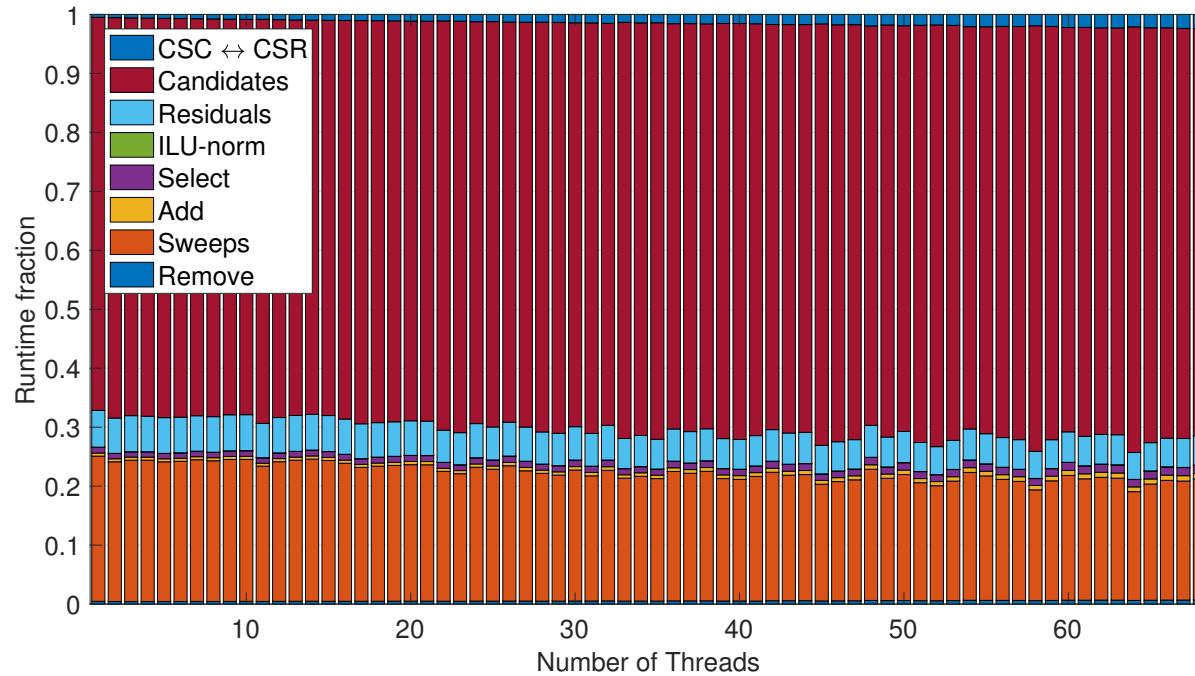
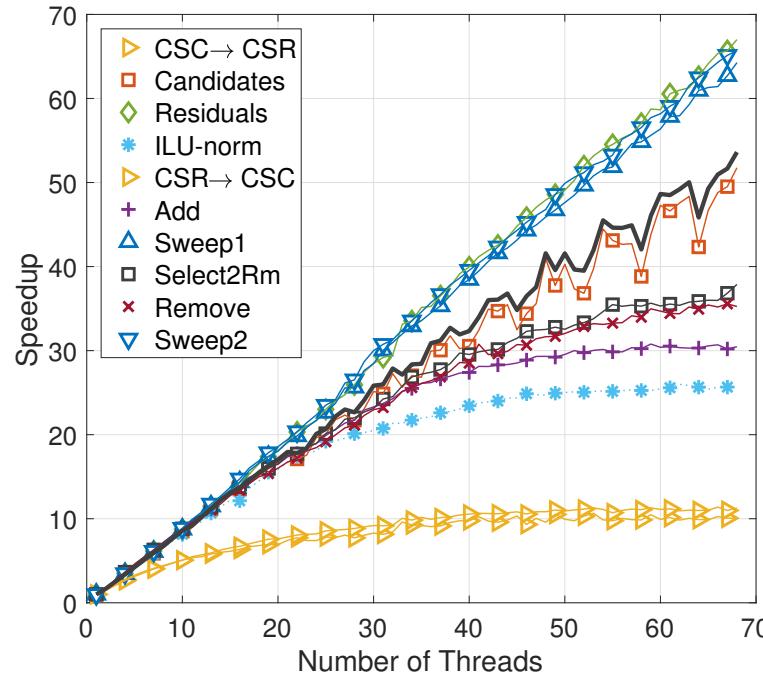


- Building blocks scale with 15% - 100% parallel efficiency.
- Transposition and sort are the bottlenecks.
- Overall speedup ~35x when using 68 KNL cores.

# ParILUT Scalability

topopt120 matrix from topology optimization, 67 el/row.

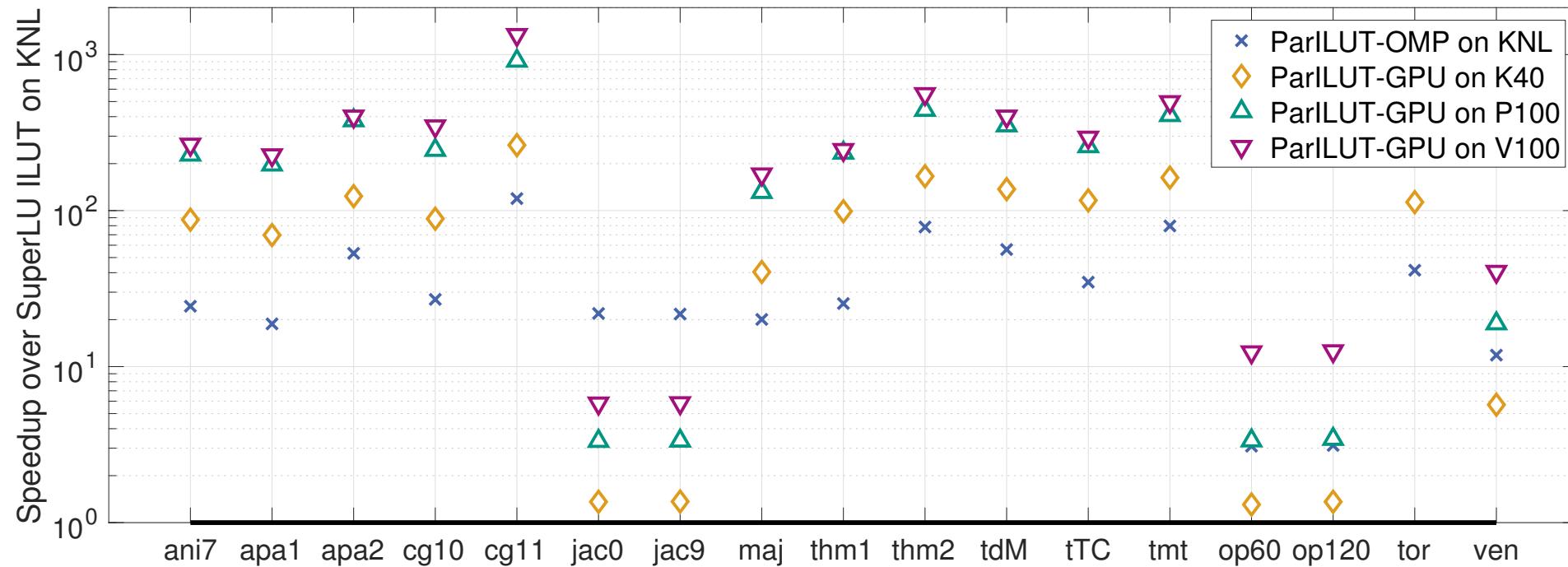
Intel Xeon Phi 7250 "Knights Landing"  
68 cores @1.40 GHz,  
16GB MCDRAM @490 GB/s



- Building blocks scale with 15% - 100% parallel efficiency.
- Dominated by candidate search.
- Overall speedup ~52x when using 68 KNL cores.

# ParILUT Performance across Manycore architectures

We compare against ILUT in SuperLU from LBNL – and thank *Sherry Li* for help and support in doing this comparison.  
The SuperLU ILUT is a sequential implementation – **ParILUT is the first parallel ILUT algorithm.**



Bibliography: <sup>1</sup>Chow et al. “Asynchronous Iterative Algorithm for Computing Incomplete Factorizations on GPUs”. In ISC 2015.

<sup>2</sup>Anzt et al. “ParILUT – A new parallel threshold ILU”. In: SIAM Journal on Scientific Comp. (2018).

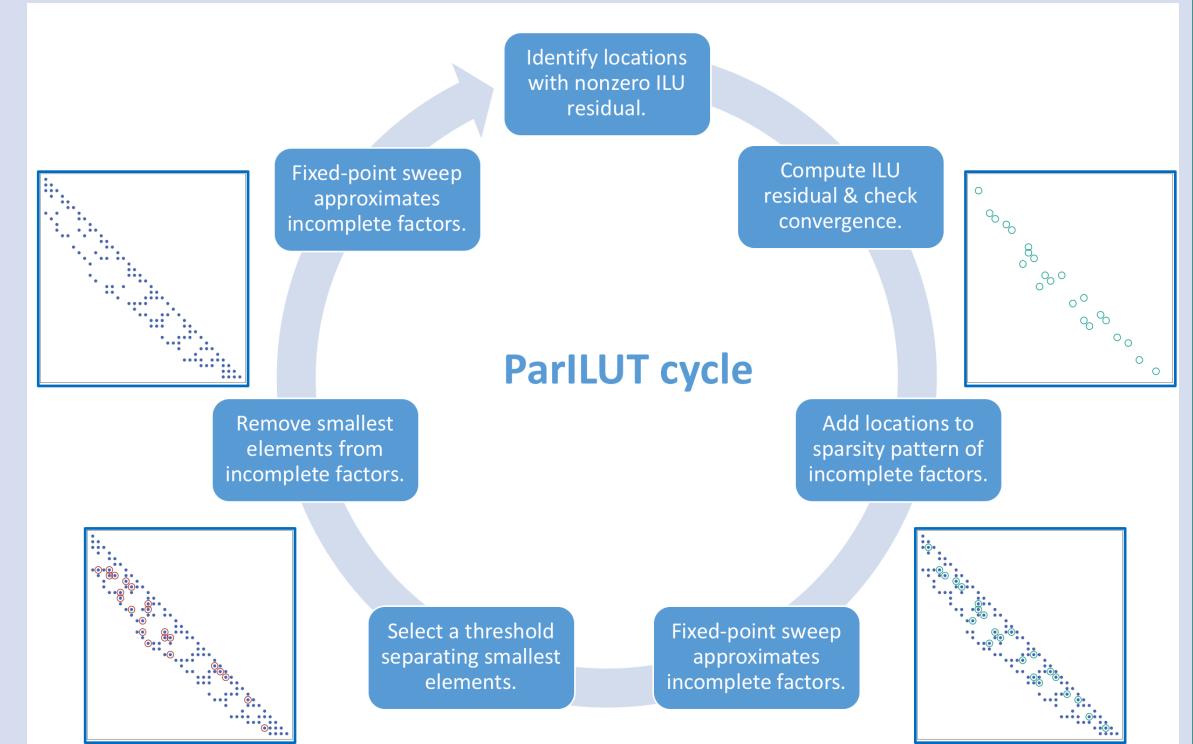
<sup>3</sup>Ribizel et al. “Approximate and Exact Selection on GPUs”. In AsHES workshop, 2019.

<sup>4</sup>Anzt et al. “ParILUT – A parallel threshold ILU for GPUs”. In IPDPS conference, 2019.

# The Manycore Challenge

## Reformulate algorithms as element-parallel fixed-point Iterations

- Algorithms need **fine-grained parallelism**  
-- thousands of SIMT threads!
- Global **synchronizations** are killing performance;
- Runtime scheduling of thread blocks **virtually impossible**;
- Memory access pattern central (coalesced data access);
- Asynchronous algorithms needed;
- Reformulation as fixed-point iteration;



Copyright@ORNL

# The Software Challenge

---

- **Software is an central component in Exascale Computing!**
  - We should focus more on sustainable software than on hardware development.
  - Software often lives longer than a HPC system.
- **Close collaboration with hardware developers and Universities is key to prepare for future hardware!**
- **We still lack the acceptance of scientific software engineers!**
  - The standard perception is: we buy new hardware, your core runs faster....
  - We need the **academic acceptance of scientific software engineers!**
- **We are running an inefficient, publication-driven system ignoring the importance of production code!**

# Creating a Sustainable HPC Landscape

## The Typical Publication in HPC Conferences / Journals

- An article describing a **new algorithm / implementation** outperforming existing solutions.
- **Performance benchmarks** on high-end HPC resources (not even archived)
- **Internal prototype code** (not publicly accessible)

*How does the community benefit from reading this?*

- + **New ideas** presented;
- + **Performance evaluations** presented;
- Performance evaluations are typically “**selective**”;
- Users / Application Scientists need to **re-implement code**;
- Difficult if **few details** are provided;
- **Not integrated into community packages**;

# Creating a Sustainable HPC Landscape

## The Typical Publication in HPC Conferences / Journals

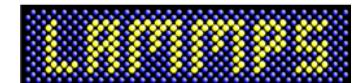
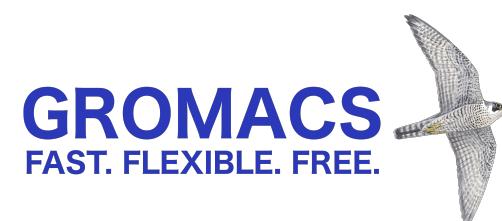
- An article describing a **new algorithm / implementation** outperforming existing solutions.
- **Performance benchmarks** on high-end HPC resources (not even archived)
- **Internal prototype code** (not publicly accessible)

*How does the community benefit from reading this?*

- + **New ideas** presented;
- + **Performance evaluations** presented;
- Performance evaluations are typically “**selective**”;
- Users / Application Scientists need to **re-implement code**;
- Difficult if **few details** are provided;
- **Not integrated into community packages**;

## Established community software packages

- are the **powertrain** behind many **scientific simulation codes**;
- often **fall short** in providing production-ready implementations of **novel algorithms**;
- often accept merge requests that **lack comprehensive documentation** and rigorous **performance assessment**;



# Creating a Sustainable HPC Landscape

## The Typical Publication in HPC Conferences / Journals

- An article describing a **new algorithm / implementation** outperforming existing solutions.
- **Performance benchmarks** on high-end HPC resources (not even archived)
- **Internal prototype code** (not publicly accessible)

*How does the community benefit from reading this?*

- + **New ideas** presented;
- + **Performance evaluations** presented;
- Performance evaluations are typically “**selective**”;
- Users / Application Scientists need to **re-implement code**;
- Difficult if **few details** are provided;
- **Not integrated into community packages**;

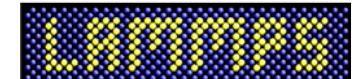


In a **perfect world**, new algorithms, implementations & performance results are

- fully reproducible;
- publicly accessible;
- ready to be used by the community / domain scientists;
- integrated into community packages;

## Established community software packages

- are the **powertrain** behind many **scientific simulation codes**;
- often **fall short** in providing production-ready implementations of **novel algorithms**;
- often accept merge requests that **lack comprehensive documentation** and rigorous **performance assessment**;



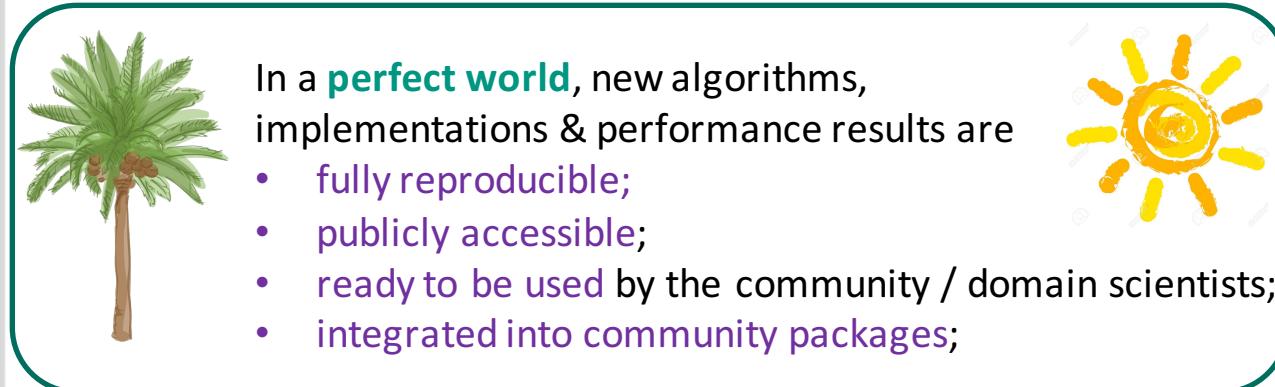
# Creating a Sustainable HPC Landscape

## The Typical Publication in HPC Conferences / Journals

- An article describing a **new algorithm / implementation** outperforming existing solutions.
- **Performance benchmarks** on high-end HPC resources (not even archived)
- **Internal prototype code** (not publicly accessible)

*How does the community benefit from reading this?*

- + **New ideas** presented;
- + **Performance evaluations** presented;
- Performance evaluations are typically “**selective**”;
- Users / Application Scientists need to **re-implement code**;
- Difficult if **few details** are provided;
- **Not integrated into community packages**;



## Established community software packages

- are the **powertrain** behind many **scientific simulation codes**;
- often **fall short** in providing production-ready implementations of **novel algorithms**;
- often accept merge requests that **lack comprehensive documentation** and rigorous **performance assessment**;

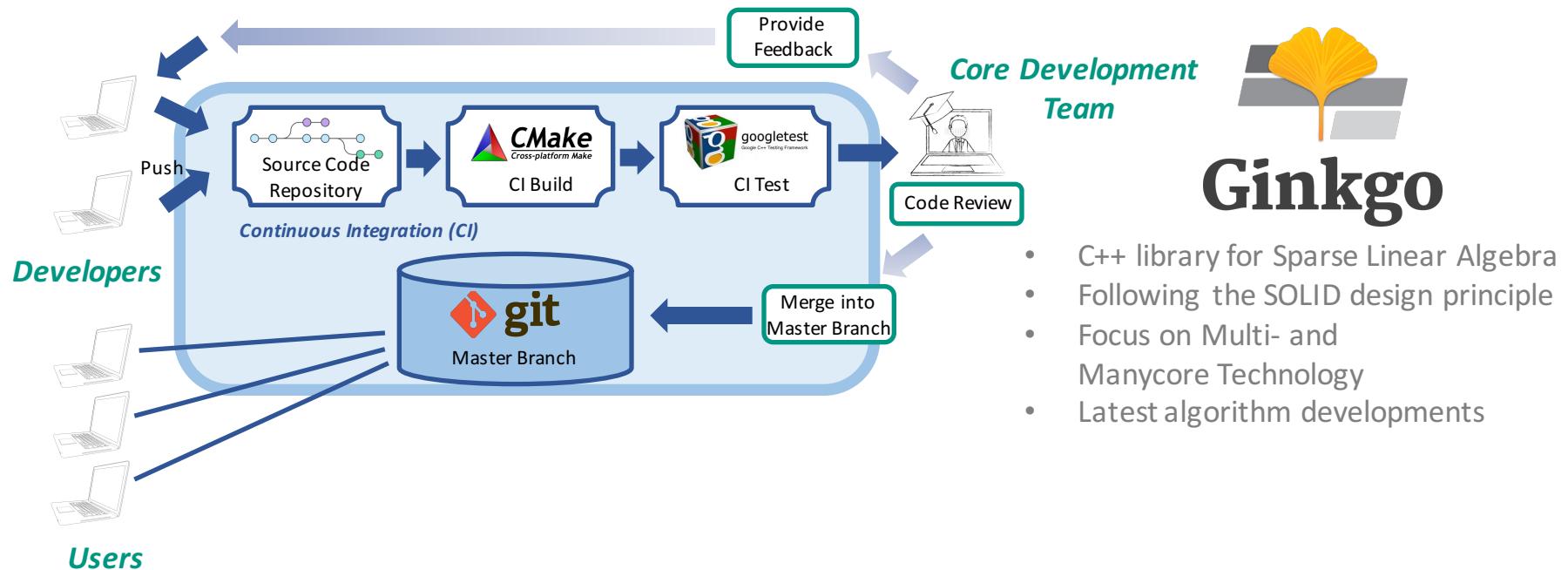
## *Why are we not changing the system?*

- $\text{effort( Prototype Code )} \ll \text{effort( Production Code )}$  ;
- Little academic reward for **sustainable software development**;
- Promotion and appointability based on **scientific papers**;

***Status Quo Extremely inefficient and unsatisfying!***

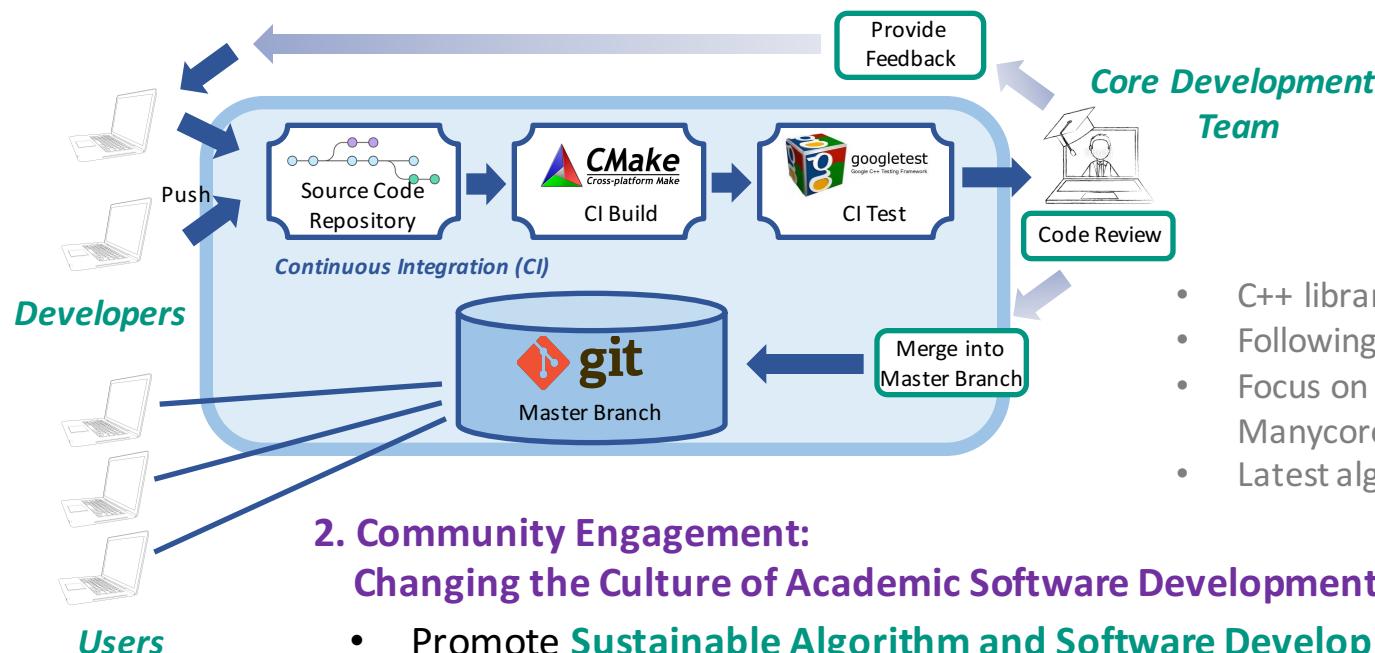
# My Efforts towards a Sustainable HPC Landscape

## 1. Sustainable Software Development in the HYIG FiNE



# My Efforts towards a Sustainable HPC Landscape

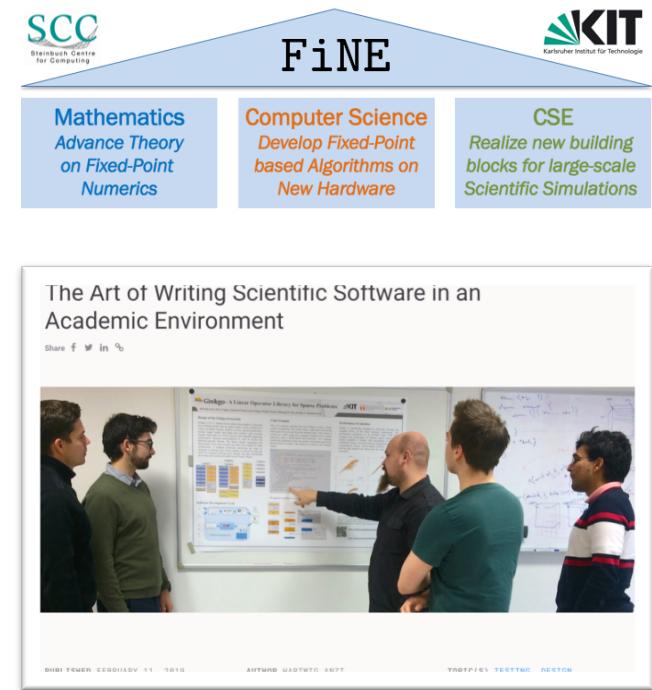
## 1. Sustainable Software Development in the HYIG FiNE



- C++ library for Sparse Linear Algebra
- Following the SOLID design principle
- Focus on Multi- and Manycore Technology
- Latest algorithm developments

## 2. Community Engagement: Changing the Culture of Academic Software Development

- Promote **Sustainable Algorithm and Software Development** (PASC 2019)  
[www.bit.ly/ContinuousBenchmarking](http://www.bit.ly/ContinuousBenchmarking)
- Address the **Challenges of Academic Software** Development (BSSw Blog Article)  
[www.bit.ly/AcademicResearchSoftware](http://www.bit.ly/AcademicResearchSoftware)
- Argue for accepting **Software Patches as Full Conference Contributions** (PDSEC 2019)  
[www.bit.ly/AreWeDoingTheRightThing](http://www.bit.ly/AreWeDoingTheRightThing)
- Welcome Software Patches as Conference Contributions at  
**Workshop on Scalable Data Analytics in Scientific Computing (SDASC 2020)** in conjunction with ISC'20 in Frankfurt



# Core Concept: Separate Algorithm from Kernels



Library core contains architecture-agnostic algorithm implementation;

Architecture-specific kernels execute the algorithm on target architecture;

## Kernels

- Accessor
- SpMV
- Solver kernels
- Precond kernels
- ...

## Core

Library Infrastructure  
Algorithm Implementations

- Iterative Solvers
- Preconditioners
- ...

# Core Concept: Separate Algorithm from Kernels



Library core contains architecture-agnostic algorithm implementation;

Architecture-specific kernels execute the algorithm on target architecture;

## Kernels

### Reference

Reference kernels

- Accessor
- SpMV
- Solver kernels
- Precond kernels
- ...

Reference are sequential kernels to check correctness of algorithm design and optimized kernels;

### Core

Library Infrastructure  
Algorithm Implementations

- Iterative Solvers
- Preconditioners
- ...

# Core Concept: Separate Algorithm from Kernels



Library core contains architecture-agnostic algorithm implementation;

Runtime polymorphism selects the right kernel depending on the target architecture;

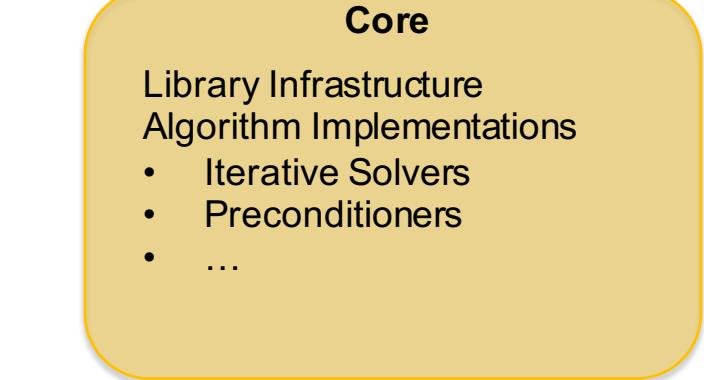
Architecture-specific kernels execute the algorithm on target architecture;

## Kernels

### Reference

- Reference kernels
  - Accessor
  - SpMV
  - Solver kernels
  - Precond kernels
  - ...

Reference are sequential kernels to check correctness of algorithm design and optimized kernels;



### CUDA

- NVIDIA-GPU kernels
  - Accessor
  - SpMV
  - Solver kernels
  - Precond kernels
  - ...

Optimized architecture-specific kernels;

### OpenMP

- OpenMP-kernels
  - Accessor
  - SpMV
  - Solver kernels
  - Precond kernels
  - ...

# Core Concept: Separate Algorithm from Kernels



Library core contains architecture-agnostic algorithm implementation;

Runtime polymorphism selects the right kernel depending on the target architecture;

Architecture-specific kernels execute the algorithm on target architecture;

## Kernels

### Reference

- Reference kernels
- Accessor
  - SpMV
  - Solver kernels
  - Precond kernels
  - ...

### CUDA

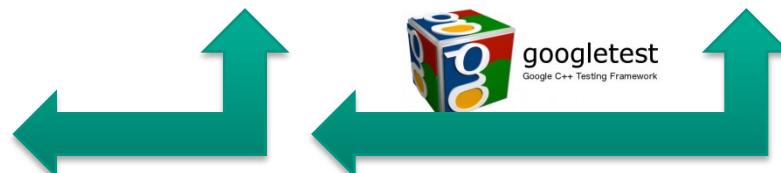
- NVIDIA-GPU kernels
- Accessor
  - SpMV
  - Solver kernels
  - Precond kernels
  - ...

### OpenMP

- OpenMP-kernels
- Accessor
  - SpMV
  - Solver kernels
  - Precond kernels
  - ...

Reference are sequential kernels to check correctness of algorithm design and optimized kernels;

Optimized architecture-specific kernels;



# Core Concept: Separate Algorithm from Kernels



Library core contains architecture-agnostic algorithm implementation;

Runtime polymorphism selects the right kernel depending on the target architecture;

Architecture-specific kernels execute the algorithm on target architecture;

## Kernels

### Reference

- Reference kernels
  - Accessor
  - SpMV
  - Solver kernels
  - Precond kernels
  - ...

Reference are sequential kernels to check correctness of algorithm design and optimized kernels;

### CUDA

- NVIDIA-GPU kernels
  - Accessor
  - SpMV
  - Solver kernels
  - Precond kernels
  - ...

### OpenMP

- OpenMP-kernels
  - Accessor
  - SpMV
  - Solver kernels
  - Precond kernels
  - ...

### HIP

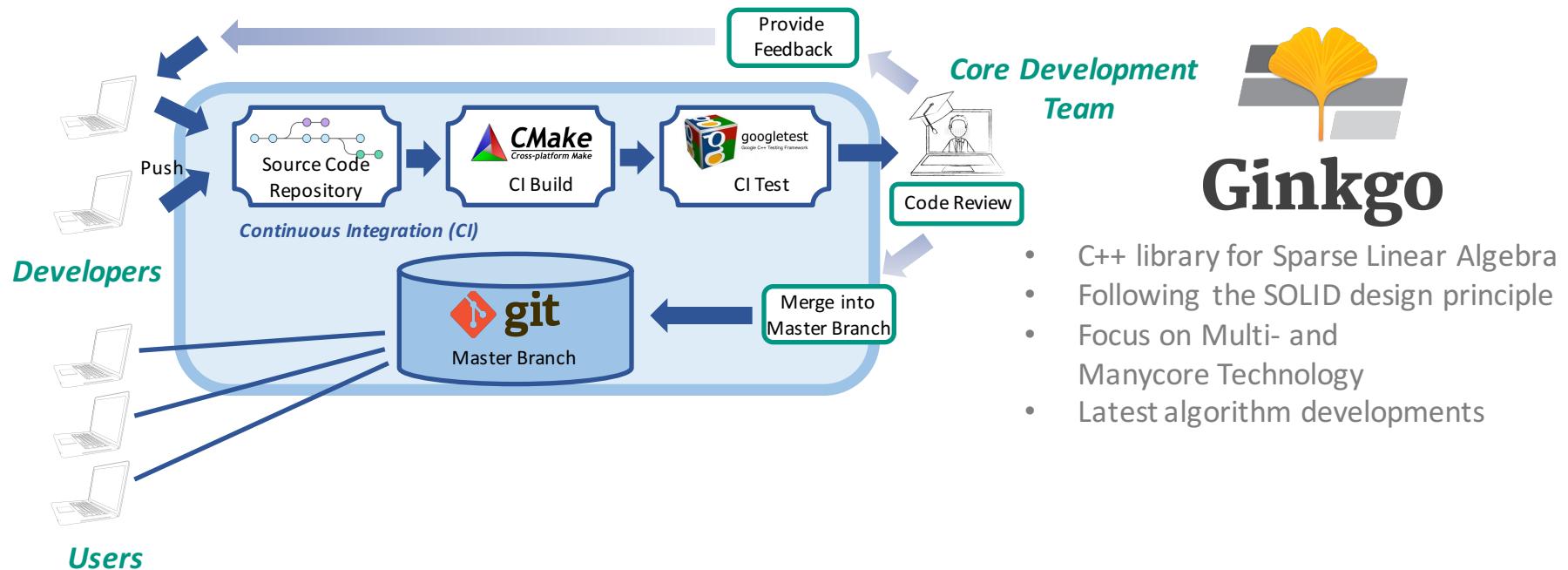
- AMD-GPU kernels
  - Accessor
  - SpMV
  - ...
- Multi-GPU
  - NVIDIA-GPU kernels
    - Accessor
    - SpMV
    - Solver kernels
    - Precond kernels
    - ...

Optimized architecture-specific kernels;



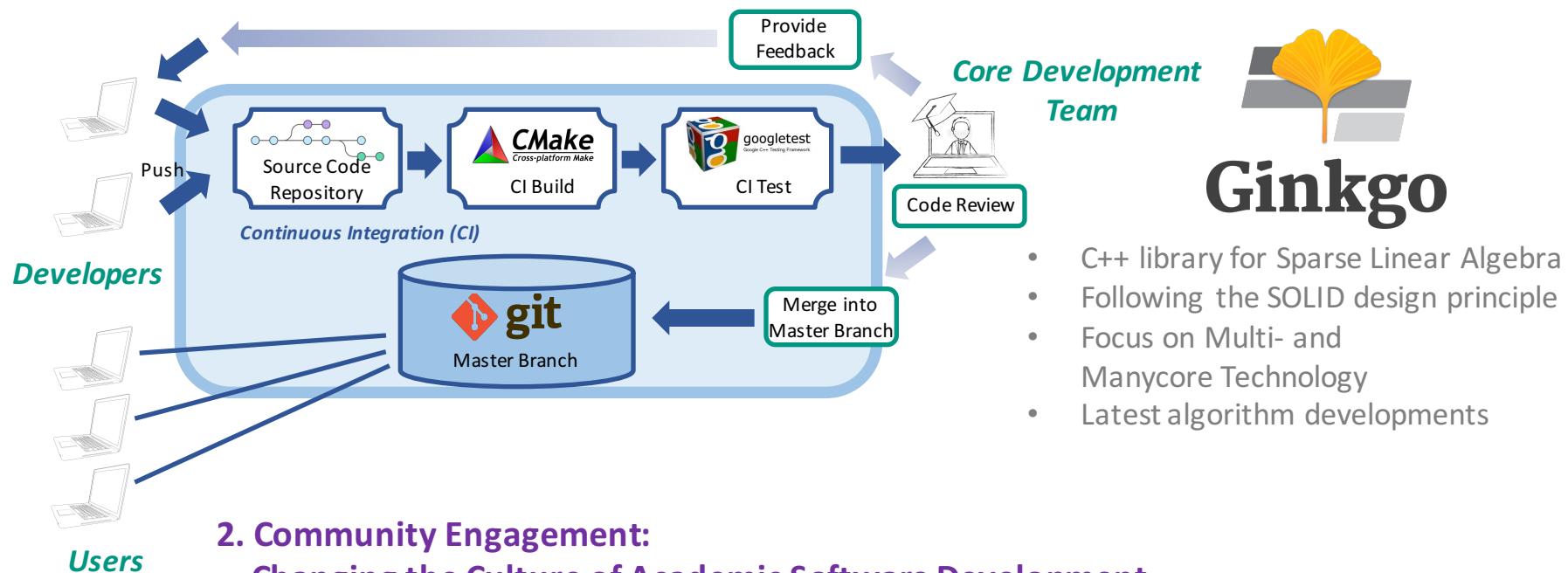
# My Efforts towards a Sustainable HPC Landscape

## 1. Sustainable Software Development in the HYIG FiNE



# My Efforts towards a Sustainable HPC Landscape

## 1. Sustainable Software Development in the HYIG FiNE



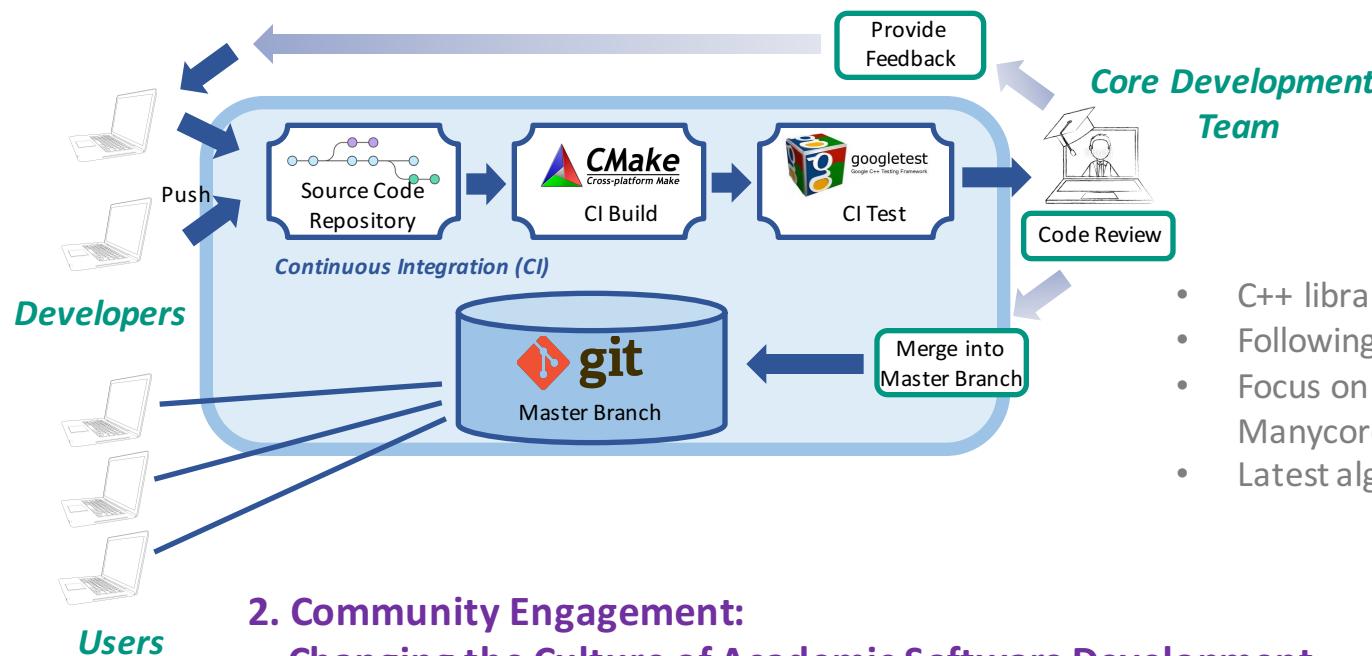
## 2. Community Engagement: Changing the Culture of Academic Software Development

- Promote **Sustainable Algorithm and Software Development** (PASC 2019)  
[www.bit.ly/ContinuousBenchmarking](http://www.bit.ly/ContinuousBenchmarking)
- Address the **Challenges of Academic Software** Development (BSSw Blog Article)  
[www.bit.ly/AcademicResearchSoftware](http://www.bit.ly/AcademicResearchSoftware)
- Argue for accepting **Software Patches as Full Conference Contributions** (PDSEC 2019)  
[www.bit.ly/AreWeDoingTheRightThing](http://www.bit.ly/AreWeDoingTheRightThing)



# My Efforts towards a Sustainable HPC Landscape

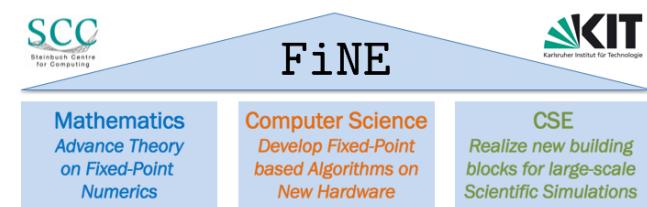
## 1. Sustainable Software Development in the HYIG FiNE



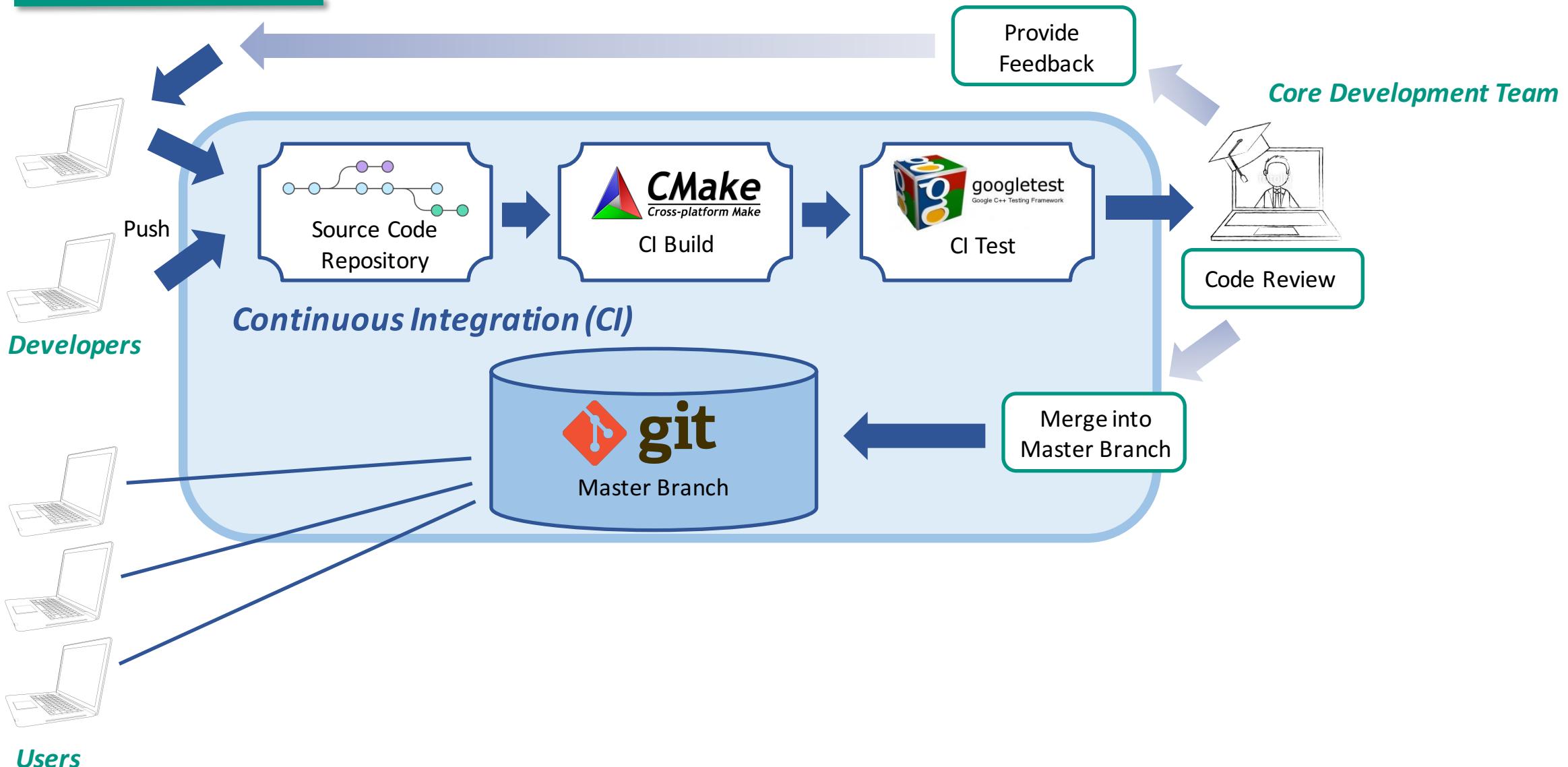
- C++ library for Sparse Linear Algebra
- Following the SOLID design principle
- Focus on Multi- and Manycore Technology
- Latest algorithm developments

## 2. Community Engagement: Changing the Culture of Academic Software Development

- Promote **Sustainable Algorithm and Software Development** (PASC 2019)  
[www.bit.ly/ContinuousBenchmarking](http://www.bit.ly/ContinuousBenchmarking)
- Address the **Challenges of Academic Software** Development (BSSw Blog Article)  
[www.bit.ly/AcademicResearchSoftware](http://www.bit.ly/AcademicResearchSoftware)
- Argue for accepting **Software Patches as Full Conference Contributions** (PDSEC 2019)  
[www.bit.ly/AreWeDoingTheRightThing](http://www.bit.ly/AreWeDoingTheRightThing)



# A Healthy Software Development Cycle



# Software Patches

- Software patches usually submitted as *merge-/push-request* in the *software versioning system* (e.g. Git).
- The patches are accompanied by detailed documentation explaining code functionality and feature usage.

ginkgo-project / ginkgo

Code Issues 44 Pull requests 7 Projects 0 Wiki Insights

Merged gflegar merged 3 commits into develop from interleaved\_block\_jacobi on Nov 26, 2018

Conversation 10 Commits 3 Checks 0 Files changed 9 +481 -169

gflegar commented on Oct 31, 2018 • edited Member

This PR further improves the performance of the block-Jacobi preconditioner for smaller block sizes by redesigning the way blocks are stored in memory. In addition to column-major storage introduced in #158, this PR interleaves the blocks to maximize coalescence when a single warp handles multiple problems.

The idea is shown in the following figure, where the maximum block size allows to interleave 2 blocks to fill the cache line:

Option 1:

Legend:  
- Jacobi block  
- leading dimension  
- padding

cache line size

Option 2:

Legend:  
- Jacobi block  
- leading dimension  
- padding

cache line size

There's trade-off in both approaches depicted in the figure. Option 1 always results in aligned data access, but consumes more memory in total. Option 2 consumes less memory, but data accesses are not always aligned.

I'm currently running benchmarks for both options on PizDaint, but the results on an initial implementation of this I got before suggest that option 2 is faster.

Reviewers  
pratikvn hartwiganzt tcojean

Assignees  
gflegar

Labels  
CUDA Core Enhancement Reference

Projects  
None yet

Milestone  
No milestone

Notifications  
Unsubscribe

You're receiving notifications because your review was requested.

4 participants

# Software Patches

- Software patches usually submitted as *merge-/push-request* in the *software versioning system* (e.g. Git).
- The patches are accompanied by detailed documentation explaining code functionality and feature usage.
- The community can comment and review the code.

The screenshot shows a GitHub pull request page for the 'ginkgo-project / ginkgo' repository. The pull request has been reviewed by 'pratikvn' on Oct 31, 2018. The code changes are shown in a diff view for the file 'core/preconditioner/block\_jacobi.hpp'. The changes are:

```
293 + /**
294 + * Stride between two columns of a block (as number of elements).
295 +
296 + * Should be a multiple of cache line size for best performance.
```

Comments from the review:

- pratikvn** on Oct 31, 2018 Member  
The cache line size varies across machines and different hardware right? Would you not like to have this as a parameter then? Or have I misunderstood something ?
- glegar** on Nov 1, 2018 Member  
You're completely right. This one is an approximation that is actually a (small) multiple of the cache line size, depending on the combination of hardware and size of the elements. (E.g. float on NVIDIA GPUs is exactly 1 cache line, double is 2, on CPUs it's more cache lines).  
However having it as a parameter brings a lot of problems that are not yet solved in Ginkgo:
  1. The user is required to know what is the cache line size of the system, or we somehow have to figure that out by ourselves.
  2. Whatever value is passed cannot be smaller than the maximum block size, otherwise the storage scheme breaks.
  3. Since the cache line size is different on the CPU than on the GPU, copying the object between them would require non-trivial storage transformations (i.e. a simple `memcpy` would not be enough).For that reason, I've just used a compile-time approximation that works correctly (but not optimally) on all systems, until those problems are solved.
- tcojean** on Nov 1, 2018 • edited Member  
I don't know of tools to get that for all architectures (both GPU and CPU), there surely is some, but for the CPU you can at least find the information here on Linux:  
`/sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size`  
Mine says 64 bytes for example. We could either get this information statically through CMake (but you have to compile on the final system) or use some executable/functions to get the information dynamically.

There is also this tool (just a simple function really) for the CPU which has Linux, MacOS and Windows compatibility.  
<https://github.com/NickStrupat/CacheLineSize>

# Software Patches

- Software patches usually submitted as *merge-/push-request* in the *software versioning system* (e.g. Git).
- The patches are accompanied by detailed documentation explaining code functionality and feature usage.
- The community can comment and review the code.
- The submitter can attach a performance analysis to the software patch.

ginkgo-project / ginkgo

pratikvn reviewed on Oct 31, 2018

View changes

Block Member + 88 ...

gflegar commented on Nov 18, 2018

Unlike the V100 version, where both interleaved options were a bit slower, due to some strange spikes in performance for the non-interleaved version, on the V100, interleaved storage (version 2) wins:

V100 performance of 'simple\_apply' step of 'apply' stage

GFlops

Block size

| Block size | column-major (GFlops) | interleaved (GFlops) | padded interleaved (GFlops) |
|------------|-----------------------|----------------------|-----------------------------|
| 1          | ~10                   | ~10                  | ~10                         |
| 2          | ~30                   | ~35                  | ~30                         |
| 3          | ~50                   | ~55                  | ~45                         |
| 4          | ~60                   | ~70                  | ~65                         |
| 5          | ~75                   | ~85                  | ~80                         |
| 6          | ~90                   | ~100                 | ~95                         |
| 7          | ~105                  | ~110                 | ~100                        |
| 8          | ~115                  | ~120                 | ~110                        |
| 9          | ~120                  | ~125                 | ~115                        |
| 10         | ~125                  | ~130                 | ~110                        |
| 11         | ~130                  | ~135                 | ~120                        |
| 12         | ~135                  | ~140                 | ~125                        |
| 13         | ~140                  | ~145                 | ~130                        |
| 14         | ~145                  | ~150                 | ~135                        |
| 15         | ~150                  | ~155                 | ~140                        |
| 16         | ~155                  | ~160                 | ~145                        |
| 17         | ~140                  | ~135                 | ~130                        |
| 18         | ~135                  | ~140                 | ~135                        |
| 19         | ~140                  | ~145                 | ~140                        |
| 20         | ~145                  | ~150                 | ~145                        |
| 21         | ~150                  | ~155                 | ~150                        |
| 22         | ~155                  | ~160                 | ~155                        |
| 23         | ~160                  | ~165                 | ~160                        |
| 24         | ~165                  | ~170                 | ~165                        |
| 25         | ~155                  | ~160                 | ~160                        |
| 26         | ~160                  | ~165                 | ~165                        |
| 27         | ~165                  | ~170                 | ~170                        |
| 28         | ~170                  | ~175                 | ~175                        |
| 29         | ~175                  | ~180                 | ~180                        |
| 30         | ~180                  | ~185                 | ~185                        |
| 31         | ~185                  | ~190                 | ~190                        |
| 32         | ~190                  | ~195                 | ~195                        |

I'll generate more details plots and send them around tomorrow.

tcojean on Nov 1, 2018 • edited Member

I don't know of tools to get that for all architectures (both GPU and CPU), there surely is some, but for the CPU you can at least find the information here on Linux:

`/sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size`

Mine says 64 bytes for example. We could either get this information statically through CMake (but you have to compile on the final system) or use some executable/functions to get the information dynamically.

There is also this tool (just a simple function really) for the CPU which has Linux, MacOS and Windows compatibility.  
<https://github.com/NickStrupat/CacheLineSize>

Unsubscribe

receiving notifications  
use your review was posted.

Participants

# Software Patches

- Software patches usually submitted as *merge-/push-request* in the *software versioning system* (e.g. Git).
- The patches are accompanied by detailed documentation explaining code functionality and feature usage.
- The community can comment and review the code.
- The submitter can attach a performance analysis to the software patch.
- Software patches can either add new functionality...

The screenshot shows a GitHub pull request interface. The top bar indicates the pull request number (178) and the file being viewed (core/preconditioner/block\_jacobi.hpp). The main area displays a code diff between two branches. The left column shows the current state of the code, and the right column shows the changes introduced by the pull request. The changes include comments explaining the purpose of the interleaved block storage scheme, including offsets and group sizes. The right side of the interface includes a vertical toolbar with various icons for repository management, and a sidebar on the right showing commit history and other pull requests.

```
178 core/preconditioner/block_jacobi.hpp
@@ -78,6 +78,106 @@ struct index_type<Op<ValueType, IndexType>> {
78     };
79     // namespace detail
80
81     // TODO: replace this with a custom accessor
82     /**
83      * Defines the parameters of the interleaved block storage scheme used by
84      * block-Jacobi blocks.
85      *
86      * @param IndexType type used for storing indices of the matrix
87      */
88     template <typename IndexType>
89     struct block_interleaved_storage_scheme {
90         /**
91          * The offset between consecutive blocks within the group.
92          */
93         IndexType block_offset;
94         /**
95          * The offset between two block groups.
96          */
97         IndexType group_offset;
98         /**
99          * Then base 2 power of the group.
100         *
101         * I.e. the group contains `1 << group_power` elements.
102         */
103         uint32 group_power;
104
105         /**
106          * Returns the number of elements in the group.
107          *
108          * @return the number of elements in the group
109          */
110         GKO_ATTRIBUTES IndexType get_group_size() const noexcept
111         {
112             return one<IndexType>() << group_power;
113         }
114
115         /**
116          * Computes the storage space required for the requested number of blocks.
117          *
118          * @param num_blocks the total number of blocks that needs to be stored
119          *
120          * @return the total memory (as the number of elements) that need to be
121          *         allocated for the scheme
122          */
123         GKO_ATTRIBUTES IndexType compute_storage_space(IndexType num_blocks) const
124             noexcept
125         {
```

# Software Patches

- Software patches usually submitted as *merge-/push-request* in the *software versioning system* (e.g. Git).
- The patches are accompanied by detailed documentation explaining code functionality and feature usage.
- The community can comment and review the code.
- The submitter can attach a performance analysis to the software patch.
- Software patches can either add new functionality...  
... or change / enhance existing code.

The screenshot shows a GitHub pull request interface for a file named `block_jacobi_kernels.cu`. The code is a CUDA kernel for a block Jacobi preconditioner. The patch highlights changes in lines 48-106. The changes include:

- Adding namespaces `gko`, `kernels`, and `cuda`.
- Adding a block size configuration section with a template function `__global__ void __launch_bounds__(warps_per_block *cuda_config::warp_size)`.
- Modifying the implementation of `__launch_bounds__` to use `Value_type` pointers instead of `ValueType *`.
- Adding a transpose operation using `copy_matrix<max_block_size, and_transpose>`.
- Modifying the storage scheme to use `storage_scheme.get_global_block_offset(block_id)` and `storage_scheme.get_stride()`.
- Adding a transpose parameter to the transpose operation.
- Adding a transpose parameter to the `__global__ void __launch_bounds__(warps_per_block *cuda_config::warp_size)` function.
- Modifying the implementation of `__launch_bounds__` to use `const ValueType * __restrict__ blocks` and `const IndexType * __restrict__ block_ptrs`.
- Adding a transpose parameter to the transpose operation.
- Modifying the storage scheme to use `storage_scheme.get_stride()`.

# Software Patches as Conference Contribution

---

- ✓ Full reproducibility and traceability is ensured;
- ✓ Not only reviewers but the complete community can track the software patch;
- ✓ The versioning systems helps to identify the main contributors of a software contribution, ensuring full recognition;
- ✓ The versioning systems also links to the right person in case of technical questions;
- ✓ Novel algorithms and hardware-optimized implementations are quickly integrated into community packages;
- ✓ The code quality is increased as the community can comment on the patches;
- ✓ Software patches as conference contributions naturally imply an extremely high level of code documentation;
- ✓ Presenting patches at a conference makes the whole community aware of a new feature;
- ✓ Domain scientists can directly interact with software developers;

# Software Patches as Conference Contribution

---

## Envisioned Workflow:

1. The algorithm/implementation **developer submits a software patch to a community package** with
  - detailed description of the functionality and code documentation;
  - comprehensive performance assessment;
  - mark the patch **for a conference contribution**;
2. The **core development team and the community**
  - **comments** on the algorithm, the implementation, and the performance;
  - **reviews** and ultimately merges the patch;
3. The **developer submits the patch as a conference contribution**
  - **linking** to all documentation, performance results, and comments;
  - **acknowledging significant comments** from community;

# Software Patches as Conference Contribution

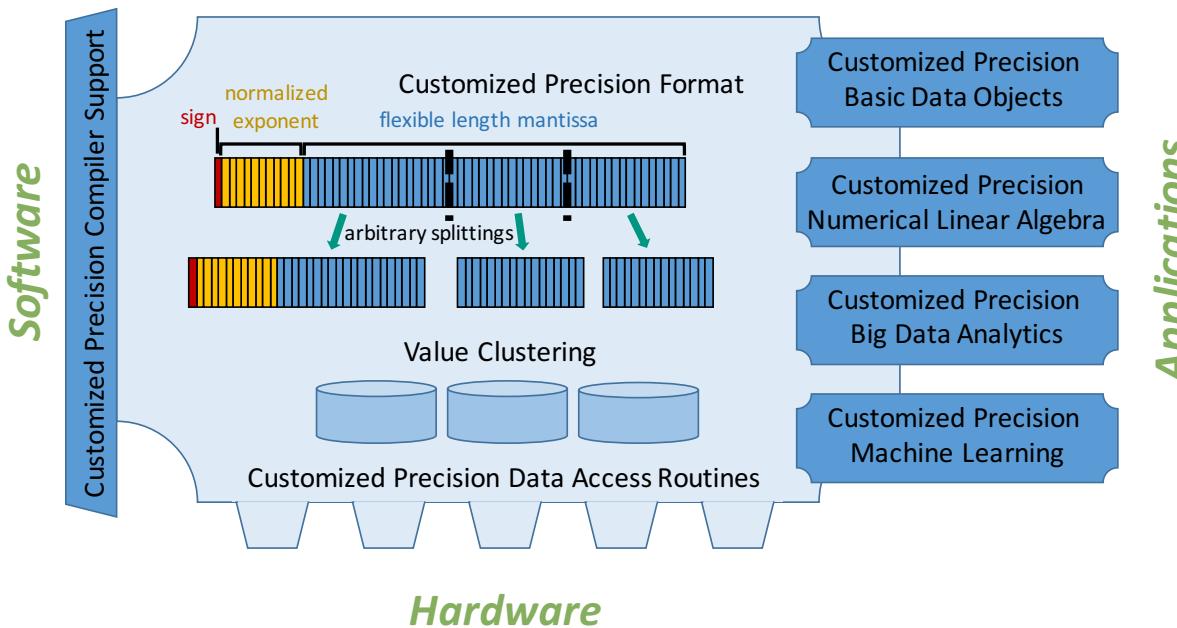
## Envisioned Workflow:

1. The algorithm/implementation **developer submits a software patch to a community package** with
  - detailed description of the functionality and code documentation;
  - comprehensive performance assessment;
  - mark the patch for a conference contribution;
2. The **core development team and the community**
  - **comments** on the algorithm, the implementation, and the performance;
  - **reviews** and ultimately merges the patch;
3. The **developer submits the patch as a conference contribution**
  - linking to all documentation, performance results, and comments;
  - acknowledging significant comments from community;
4. The **conference committee / external reviewers** do a “**light**” review of functionality, documentation, performance.
5. If accepted, the conference contribution is presented along with a **user tutorial or application examples**;
6. The submission is as a **regular paper** included in the conference proceedings
  - potentially featuring a **shorter general introduction**;
  - **including the algorithm description and performance assessment**;
  - potentially including **code segments, digital artifacts**, or a **link** to the merge request;
  - **listing all (significant) code reviewers / commenters**;

# Summary and next steps

- Decouple arithmetic precision from memory precision.
- Using **customized precisions** for memory operations.
- Speedup of up to 1.3x for adaptive precision block-Jacobi preconditioning.
- Creating a **Modular Precision Ecosystem** inside  **Ginkgo**.

<https://github.com/ginkgo-project/ginkgo>



This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration and the Helmholtz Impuls und Vernetzungsfond VH-NG-1241.

**HELMHOLTZ**  
RESEARCH FOR GRAND CHALLENGES

# Parallelism inside the blocks: Fixed-point sweeps

Fixed-point sweep approximates incomplete factors.

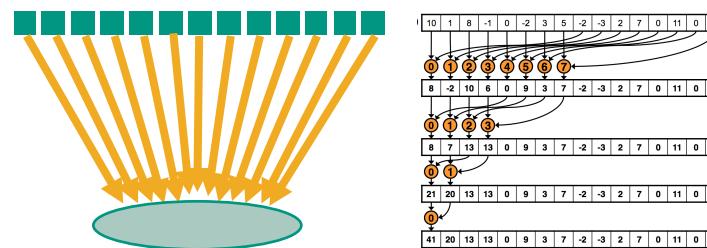
Compute ILU residual & check convergence.

Fixed-point sweeps approximate values in ILU factors and residual<sup>1</sup>:

- Inherently parallel operation.
- Elements can be updated asynchronously.
- *We can expect 100% parallel efficiency if number of cores < number of elements*
- Residual norm is a global reduction.

$$F(l_{ij}, u_{ij}) = \begin{cases} \frac{1}{u_{jj}} \left( a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right), & i > j \\ a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}, & i \leq j \end{cases}$$

bilinear fixed-point iteration can be parallelized by elements



<sup>1</sup>Chow et al. “Asynchronous Iterative Algorithm for Computing Incomplete Factorizations on GPUs”. In ISC 2015.

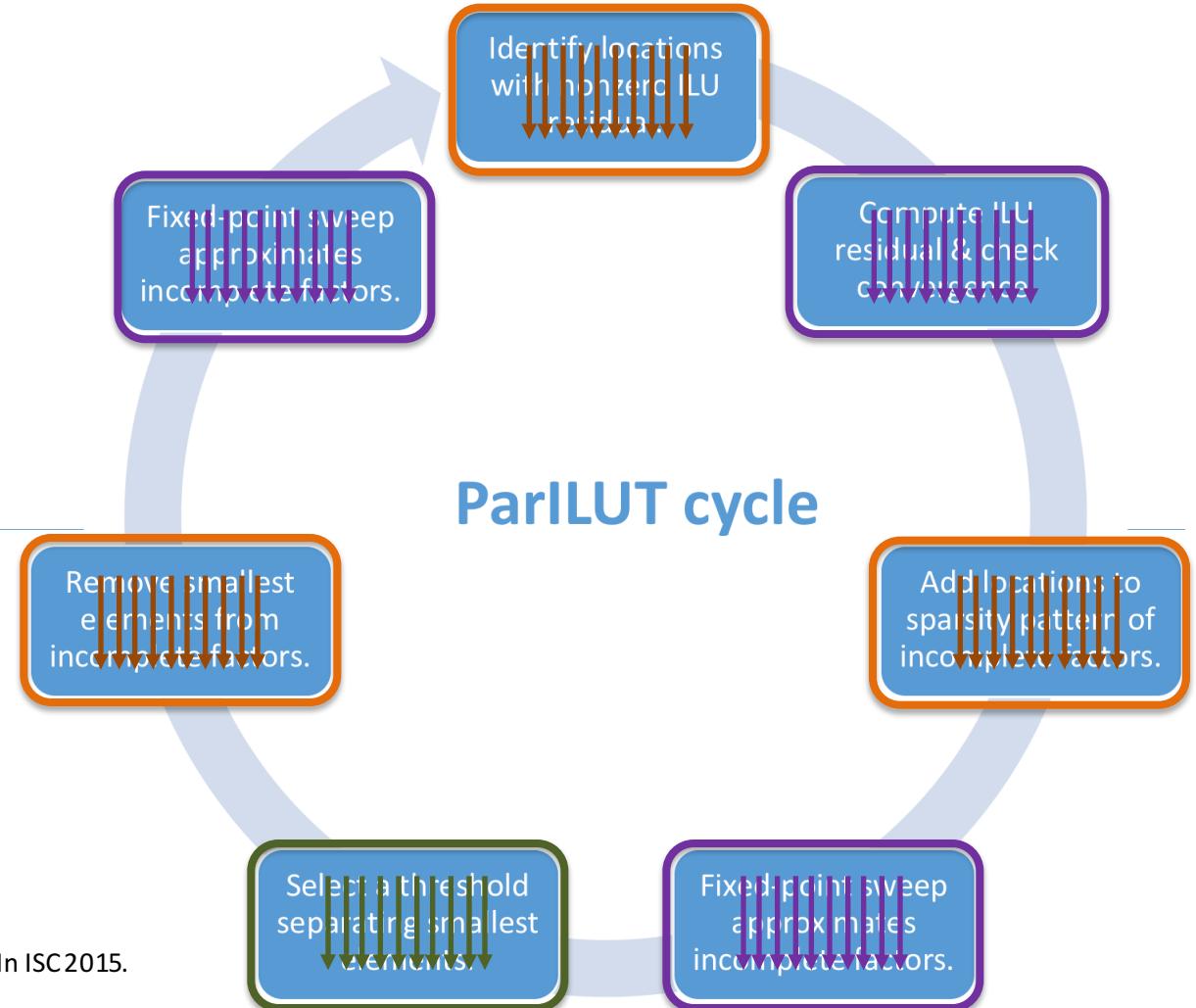
# ParILUT : Parallelism inside the blocks

Interleaving fixed-point sweeps approximating values with pattern-changing symbolic routines.

Parallelism inside the building blocks:

- Fixed-Point Sweeps<sup>1</sup>
- Residuals<sup>1</sup>
- Identify Fill-In Locations<sup>2</sup>
- Add Locations<sup>2</sup>
- Remove Locations<sup>2</sup>
- Select Threshold Separating Smallest Elements

## ParILUT cycle



<sup>1</sup>Chow et al. “Asynchronous Iterative Algorithm for Computing Incomplete Factorizations on GPUs”. In ISC 2015.

<sup>2</sup>Anzt et al. “ParILUT – A new parallel threshold ILU”. In: SIAM J. on Sci. Comp. (2018).

# ParILUT : Parallelism inside the blocks

Interleaving fixed-point sweeps approximating values with pattern-changing symbolic routines.

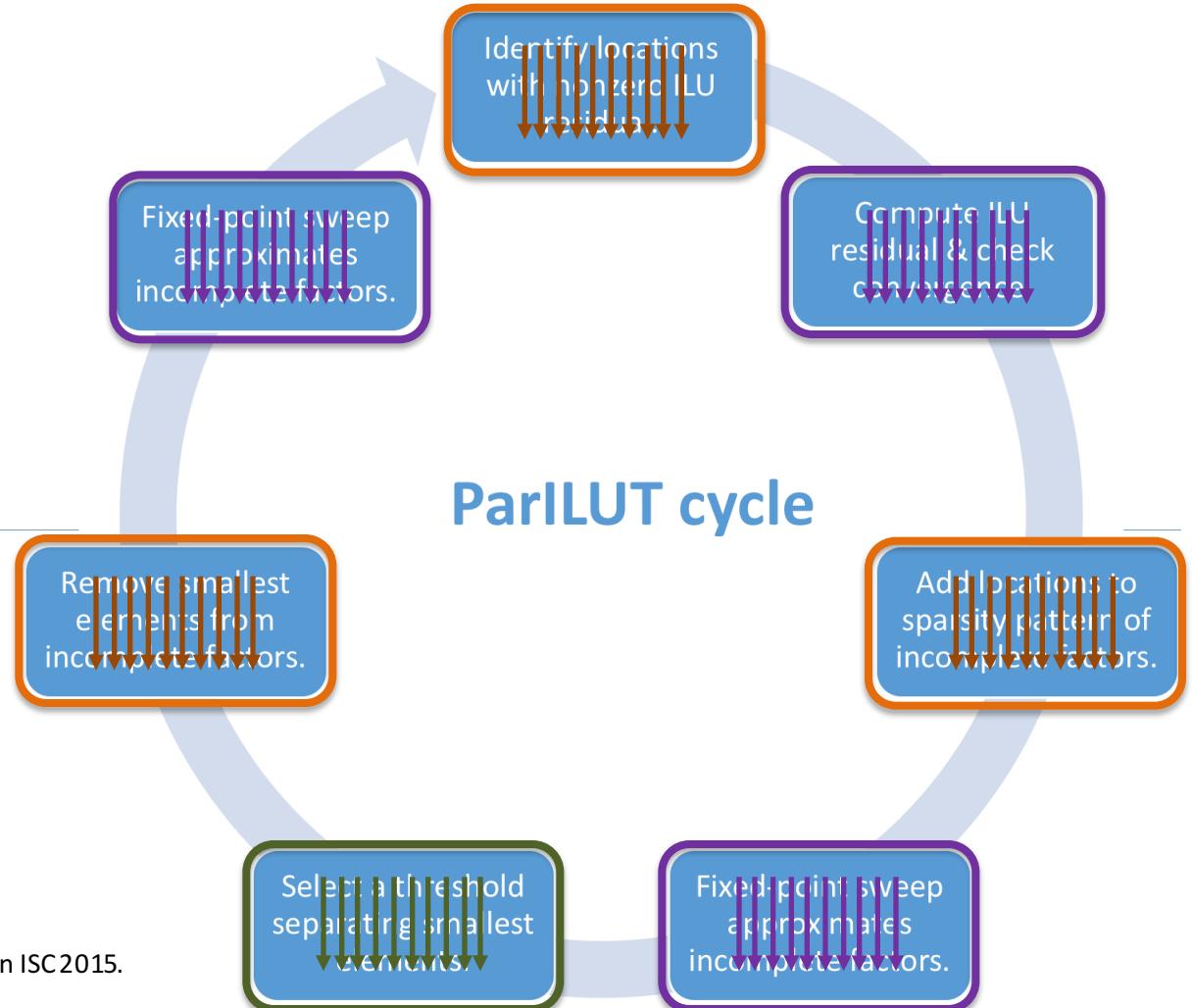
Parallelism inside the building blocks:

- Fixed-Point Sweeps<sup>1</sup> ✓
- Residuals<sup>1</sup> ✓
- Identify Fill-In Locations<sup>2</sup> ✓
- Add Locations<sup>2</sup> ✓
- Remove Locations<sup>2</sup> ✓
- Select Threshold Separating Smallest Elements



<sup>1</sup>Chow et al. "Asynchronous Iterative Algorithm for Computing Incomplete Factorizations on GPUs". In ISC 2015.

<sup>2</sup>Anzt et al. "ParILUT – A new parallel threshold ILU". In: SIAM J. on Sci. Comp. (2018).



# Parallel Threshold Selection on GPUs

*This is equivalent to the Selection Problem!*

Given an unsorted sequence of real numbers  $x_0, x_1, x_2, x_3, \dots, x_{n-1}$ , we want to find the element  $x_{i_k}$  such that in the sorted sequence

$$x_{i_0} \leq x_{i_1} \leq x_{i_2} \leq x_{i_3} \leq \dots \leq x_{i_k} \leq \dots x_{i_{n-1}}$$

$\uparrow$   
 $k$

the element  $x_{i_k}$  is located in position  $k$ .

*We do not necessarily need to sort the complete sequence!*



## Approximate and Exact Selection on GPUs

Tobias Ribizel\*, Hartwig Anzt\*†

\*Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Germany

†Innovative Computing Lab (ICL), University of Tennessee, Knoxville, USA

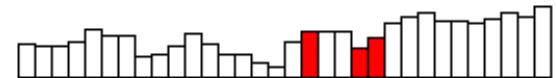
*tobias.ribizel@student.kit.edu, hartwig.anzt@kit.edu*

Tobias Ribizel

<http://bit.ly/SampleSelectGPU>

## SampleSelect Algorithm

Pick splitters



Sort splitters



Group by bucket



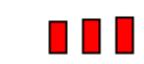
Select bucket



Pick splitters



Sort splitters



Group by bucket



# Parallel Threshold Selection on GPUs

*This is equivalent to the Selection Problem!*

Given an unsorted sequence of real numbers  $x_0, x_1, x_2, x_3, \dots, x_{n-1}$ , we want to find the element  $x_{i_k}$  such that in the sorted sequence

$$x_{i_0} \leq x_{i_1} \leq x_{i_2} \leq x_{i_3} \leq \dots \leq x_{i_k} \leq \dots x_{i_{n-1}}$$

$\uparrow$   
 $k$

the element  $x_{i_k}$  is located in position  $k$ .

*We do not necessarily need to sort the complete sequence!*



## Approximate and Exact Selection on GPUs

Tobias Ribizel\*, Hartwig Anzt\*†

\*Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Germany

†Innovative Computing Lab (ICL), University of Tennessee, Knoxville, USA

*tobias.ribizel@student.kit.edu, hartwig.anzt@kit.edu*

Tobias Ribizel

<http://bit.ly/SampleSelectGPU>

# Parallel Threshold Selection on GPUs

*This is equivalent to the Selection Problem!*

Given an unsorted sequence of real numbers  $x_0, x_1, x_2, x_3, \dots, x_{n-1}$ , we want to find the element  $x_{i_k}$  such that in the sorted sequence

$$x_{i_0} \leq x_{i_1} \leq x_{i_2} \leq x_{i_3} \leq \dots \leq x_{i_k} \leq \dots x_{i_{n-1}}$$

  $k$

the element  $x_{i_k}$  is located in position  $k$ .

*We do not necessarily need to sort the complete sequence!*



## Approximate and Exact Selection on GPUs

Tobias Ribizel\*, Hartwig Anzt\*†

\*Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Germany

†Innovative Computing Lab (ICL), University of Tennessee, Knoxville, USA

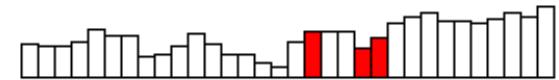
*tobias.ribizel@student.kit.edu, hartwig.anzt@kit.edu*

Tobias Ribizel

<http://bit.ly/SampleSelectGPU>

## SampleSelect Algorithm

Pick splitters



# Parallel Threshold Selection on GPUs

*This is equivalent to the Selection Problem!*

Given an unsorted sequence of real numbers  $x_0, x_1, x_2, x_3, \dots, x_{n-1}$ , we want to find the element  $x_{i_k}$  such that in the sorted sequence

$$x_{i_0} \leq x_{i_1} \leq x_{i_2} \leq x_{i_3} \leq \dots \leq x_{i_k} \leq \dots x_{i_{n-1}}$$



the element  $x_{i_k}$  is located in position  $k$ .

*We do not necessarily need to sort the complete sequence!*



## Approximate and Exact Selection on GPUs

Tobias Ribizel\*, Hartwig Anzt\*†

\*Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Germany

†Innovative Computing Lab (ICL), University of Tennessee, Knoxville, USA

*tobias.ribizel@student.kit.edu, hartwig.anzt@kit.edu*

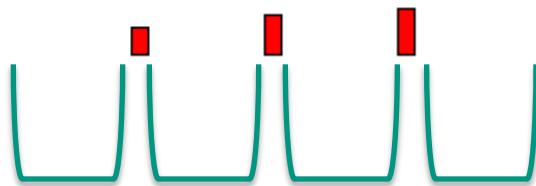
Tobias Ribizel

<http://bit.ly/SampleSelectGPU>

## SampleSelect Algorithm

Pick splitters

Sort splitters



*Splitters separate buckets*

# Parallel Threshold Selection on GPUs

*This is equivalent to the Selection Problem!*

Given an unsorted sequence of real numbers  $x_0, x_1, x_2, x_3, \dots, x_{n-1}$ , we want to find the element  $x_{i_k}$  such that in the sorted sequence

$$x_{i_0} \leq x_{i_1} \leq x_{i_2} \leq x_{i_3} \leq \dots \leq x_{i_k} \leq \dots x_{i_{n-1}}$$

k

the element  $x_{i_k}$  is located in position  $k$ .

*We do not necessarily need to sort the complete sequence!*



## Approximate and Exact Selection on GPUs

Tobias Ribizel\*, Hartwig Anzt\*†

\*Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Germany

†Innovative Computing Lab (ICL), University of Tennessee, Knoxville, USA

*tobias.ribizel@student.kit.edu, hartwig.anzt@kit.edu*

Tobias Ribizel

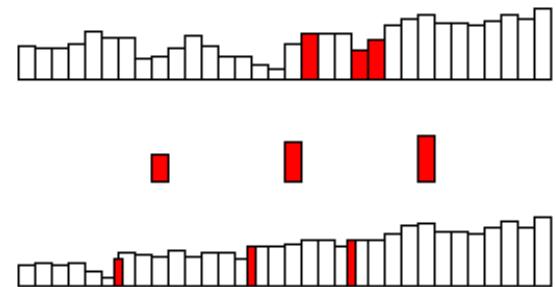
<http://bit.ly/SampleSelectGPU>

## SampleSelect Algorithm

Pick splitters

Sort splitters

Group by bucket



# Parallel Threshold Selection on GPUs

*This is equivalent to the Selection Problem!*

Given an unsorted sequence of real numbers  $x_0, x_1, x_2, x_3, \dots, x_{n-1}$ , we want to find the element  $x_{i_k}$  such that in the sorted sequence

$$x_{i_0} \leq x_{i_1} \leq x_{i_2} \leq x_{i_3} \leq \dots \leq x_{i_k} \leq \dots x_{i_{n-1}}$$

$\uparrow$   
 $k$

the element  $x_{i_k}$  is located in position  $k$ .

*We do not necessarily need to sort the complete sequence!*



## Approximate and Exact Selection on GPUs

Tobias Ribizel\*, Hartwig Anzt\*†

\*Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Germany

†Innovative Computing Lab (ICL), University of Tennessee, Knoxville, USA

*tobias.ribizel@student.kit.edu, hartwig.anzt@kit.edu*

Tobias Ribizel

<http://bit.ly/SampleSelectGPU>

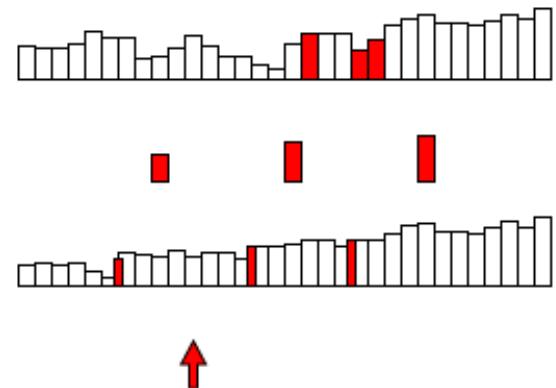
## SampleSelect Algorithm

Pick splitters

Sort splitters

Group by bucket

Select bucket



# Parallel Threshold Selection on GPUs

*This is equivalent to the Selection Problem!*

Given an unsorted sequence of real numbers  $x_0, x_1, x_2, x_3, \dots, x_{n-1}$ , we want to find the element  $x_{i_k}$  such that in the sorted sequence

$$x_{i_0} \leq x_{i_1} \leq x_{i_2} \leq x_{i_3} \leq \dots \leq x_{i_k} \leq \dots x_{i_{n-1}}$$

$\uparrow$   
 $k$

the element  $x_{i_k}$  is located in position  $k$ .

*We do not necessarily need to sort the complete sequence!*



## Approximate and Exact Selection on GPUs

Tobias Ribizel\*, Hartwig Anzt\*†

\*Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Germany

†Innovative Computing Lab (ICL), University of Tennessee, Knoxville, USA

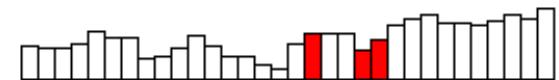
*tobias.ribizel@student.kit.edu, hartwig.anzt@kit.edu*

Tobias Ribizel

<http://bit.ly/SampleSelectGPU>

## SampleSelect Algorithm

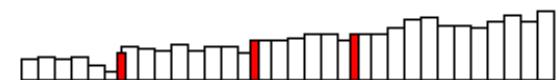
Pick splitters



Sort splitters



Group by bucket



Select bucket



Pick splitters



# Parallel Threshold Selection on GPUs

*This is equivalent to the Selection Problem!*

Given an unsorted sequence of real numbers  $x_0, x_1, x_2, x_3, \dots, x_{n-1}$ , we want to find the element  $x_{i_k}$  such that in the sorted sequence

$$x_{i_0} \leq x_{i_1} \leq x_{i_2} \leq x_{i_3} \leq \dots \leq x_{i_k} \leq \dots x_{i_{n-1}}$$

$\uparrow$   
 $k$

the element  $x_{i_k}$  is located in position  $k$ .

*We do not necessarily need to sort the complete sequence!*



## Approximate and Exact Selection on GPUs

Tobias Ribizel\*, Hartwig Anzt\*†

\*Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Germany

†Innovative Computing Lab (ICL), University of Tennessee, Knoxville, USA

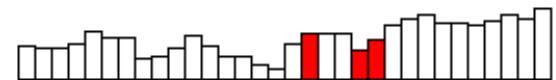
*tobias.ribizel@student.kit.edu, hartwig.anzt@kit.edu*

Tobias Ribizel

<http://bit.ly/SampleSelectGPU>

## SampleSelect Algorithm

Pick splitters



Sort splitters



Group by bucket



Select bucket



Pick splitters



Sort splitters



# Parallel Threshold Selection on GPUs

*This is equivalent to the Selection Problem!*

Given an unsorted sequence of real numbers  $x_0, x_1, x_2, x_3, \dots, x_{n-1}$ , we want to find the element  $x_{i_k}$  such that in the sorted sequence

$$x_{i_0} \leq x_{i_1} \leq x_{i_2} \leq x_{i_3} \leq \dots \leq x_{i_k} \leq \dots x_{i_{n-1}}$$

$\uparrow$   
 $k$

the element  $x_{i_k}$  is located in position  $k$ .

*We do not necessarily need to sort the complete sequence!*



## Approximate and Exact Selection on GPUs

Tobias Ribizel\*, Hartwig Anzt\*†

\*Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Germany

†Innovative Computing Lab (ICL), University of Tennessee, Knoxville, USA

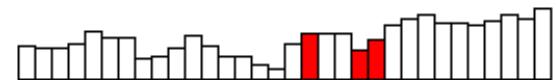
*tobias.ribizel@student.kit.edu, hartwig.anzt@kit.edu*

Tobias Ribizel

<http://bit.ly/SampleSelectGPU>

## SampleSelect Algorithm

Pick splitters



Sort splitters



Group by bucket



Select bucket



Pick splitters



Sort splitters



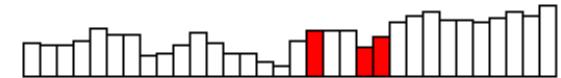
Group by bucket

# Parallel Threshold Selection on GPUs

- We only copy elements of the bucket we are interested in;
- In case of identical splitter elements, they are placed in an *equality bucket*;
- If target rank is in an *equality bucket*, the algorithm can terminate early by returning lower bound;
- For sorting the splitters, small input datasets, and the lowest recursion level a *bitonic sort* in shared memory is used;
- Use a *binary search tree* to sort elements into the buckets;

## SampleSelect Algorithm

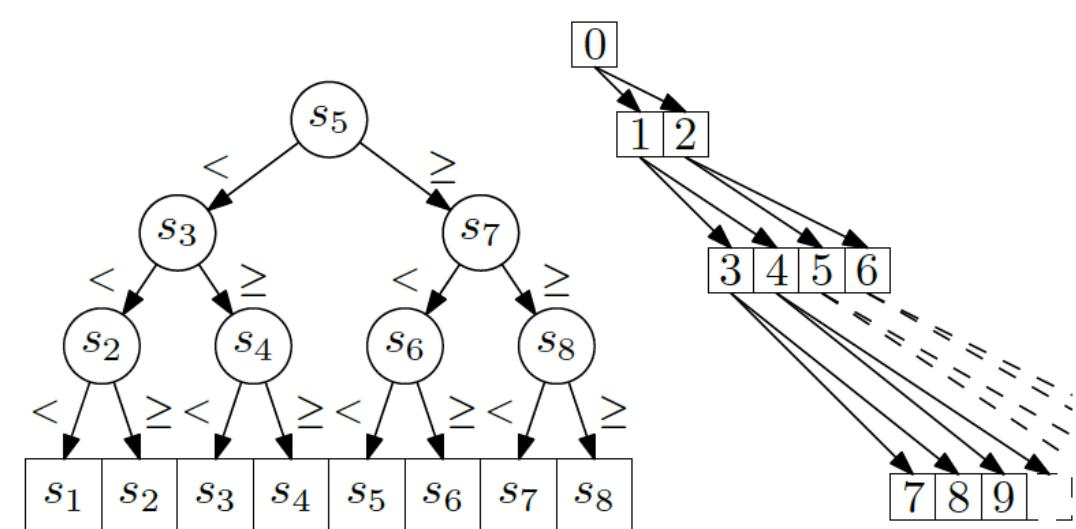
Pick splitters



Sort splitters

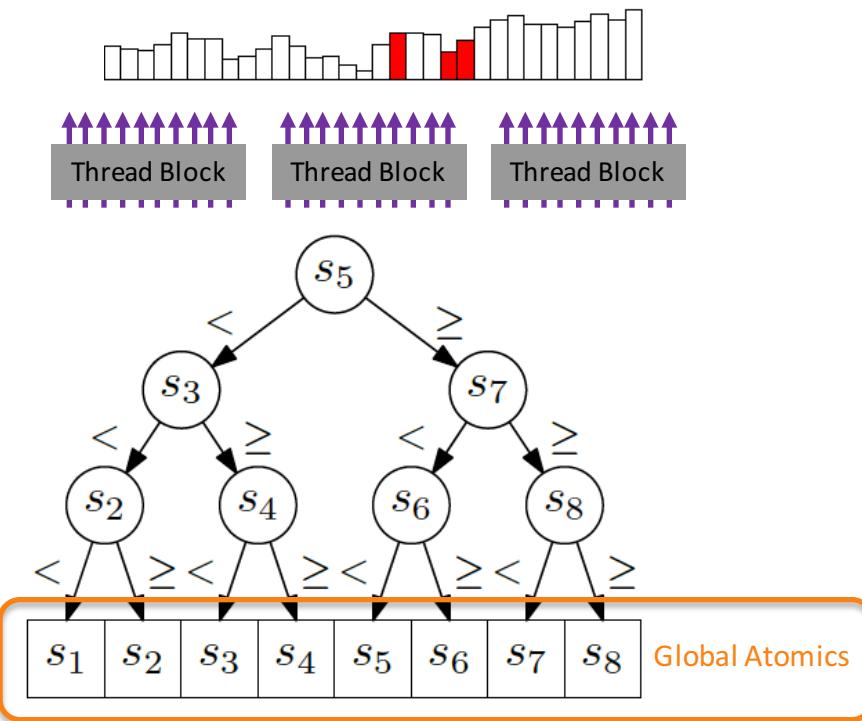


Group by bucket

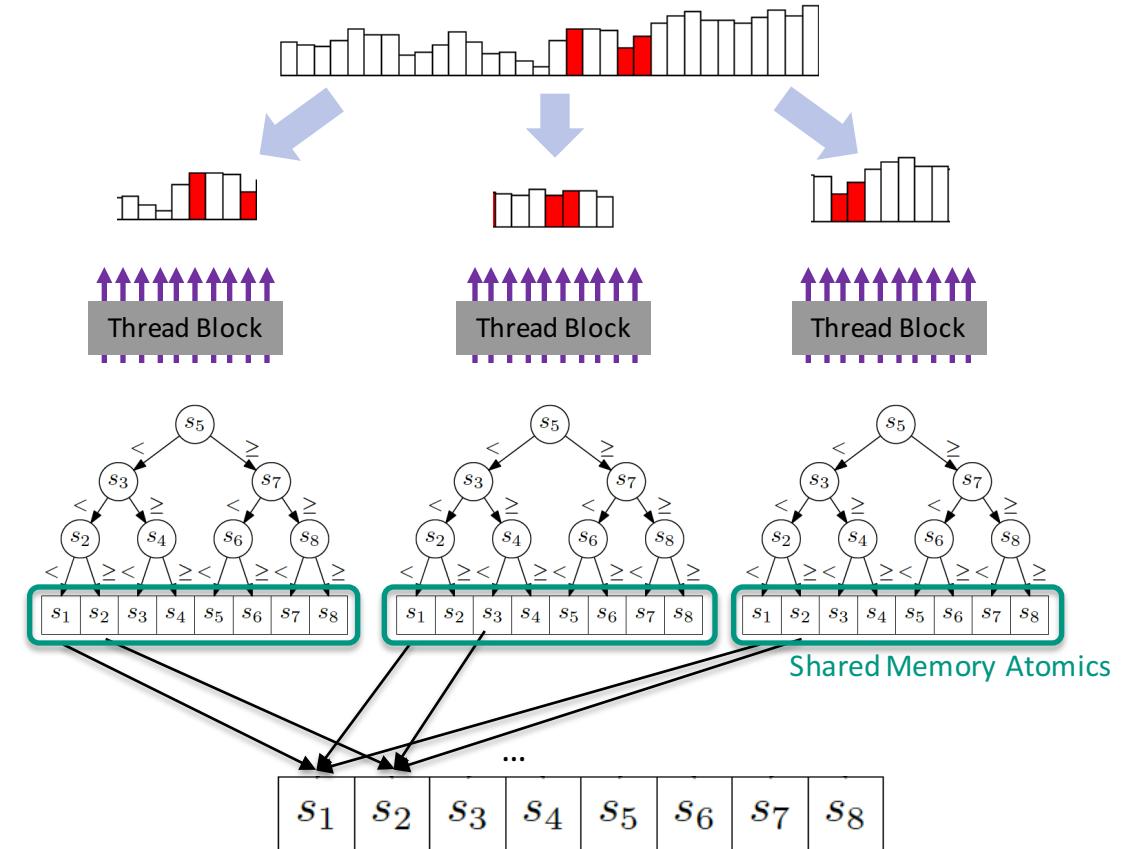


# Parallel Threshold Selection on GPUs

## Global Memory Atomics



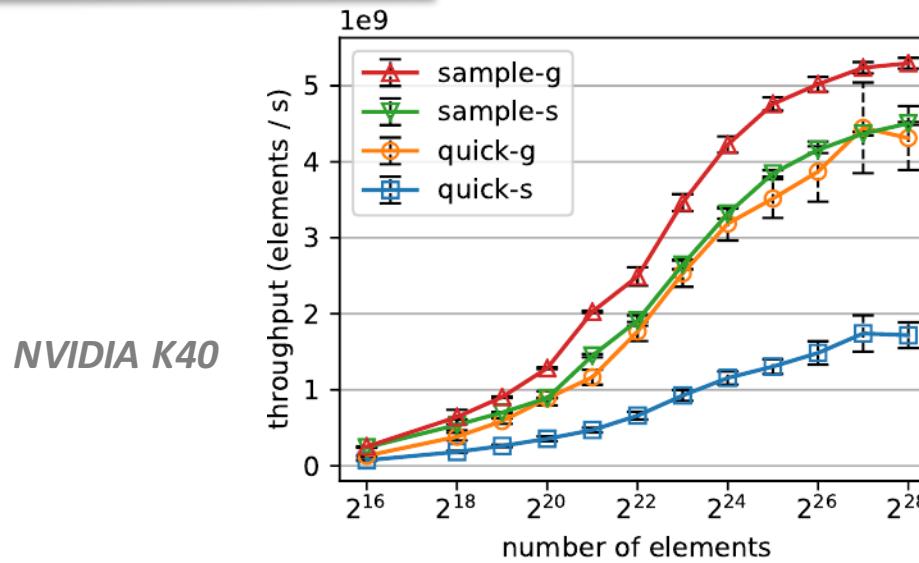
## Shared Memory Atomics



- Run SampleSelect using all resources on complete data set;
- Use global atomics to generate bucket counts;

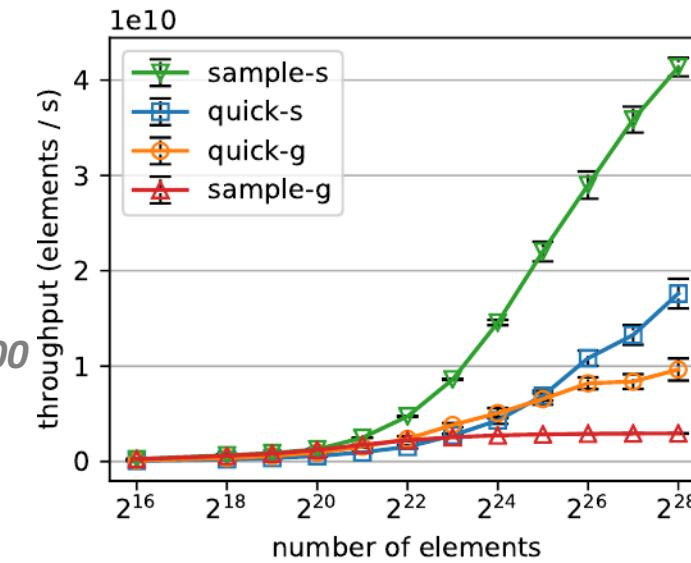
- Split data set into chunks, assign to thread blocks;
- Each thread block runs bucket count on its data;
- Use a global reduction to get global bucket counts;

# Parallel Threshold Selection on GPUs

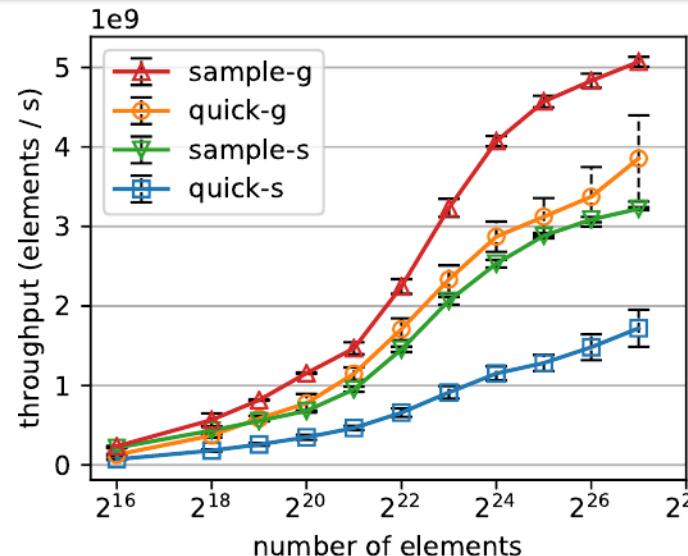


*single precision*

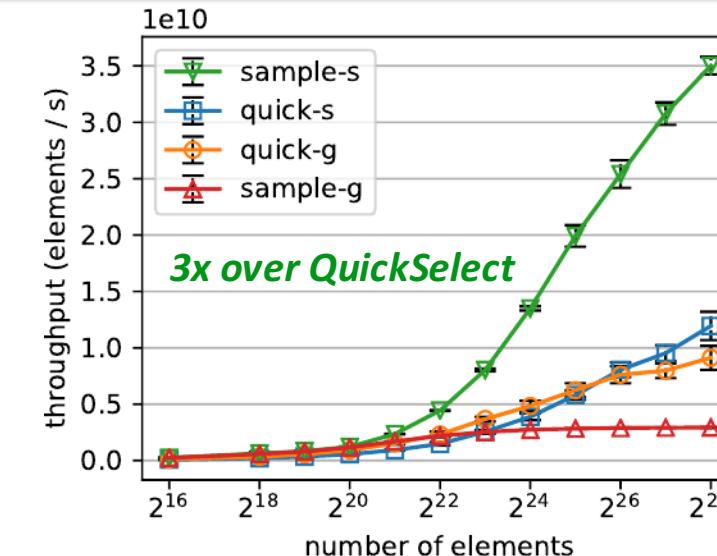
-g : global memory atomics  
-s: shared memory atomics



NVIDIA V100



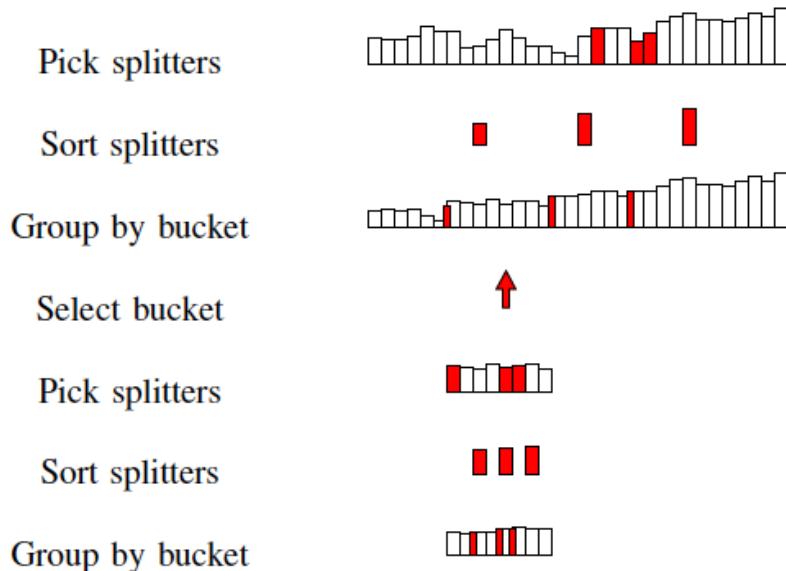
*double precision*



*3x over QuickSelect*

# Approximate Threshold Selection

## SampleSelect Algorithm



## Approximate and Exact Selection on GPUs

Tobias Ribizel\*, Hartwig Anzt\*†

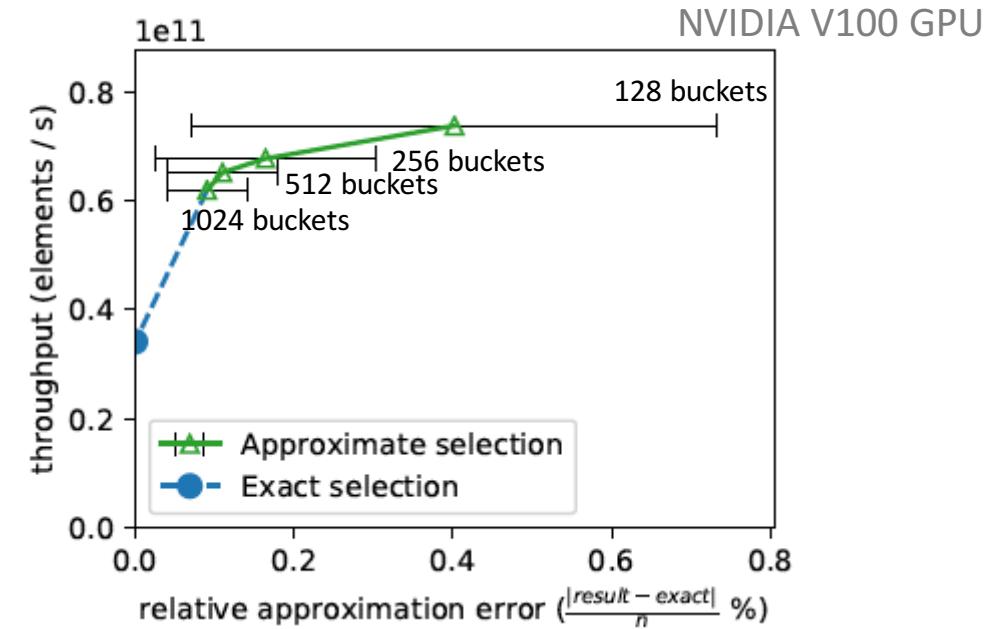
\*Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Germany

†Innovative Computing Lab (ICL), University of Tennessee, Knoxville, USA  
[tobias.ribizel@student.kit.edu](mailto:tobias.ribizel@student.kit.edu), [hartwig.anzt@kit.edu](mailto:hartwig.anzt@kit.edu)

<http://bit.ly/SampleSelectGPU>

We do not descent to the lowest level of the recursion tree if we accept an approximate threshold.

- Accuracy depends on the ratio splitters vs. dataset size;
- Independent of value distribution (works on ranks, only);

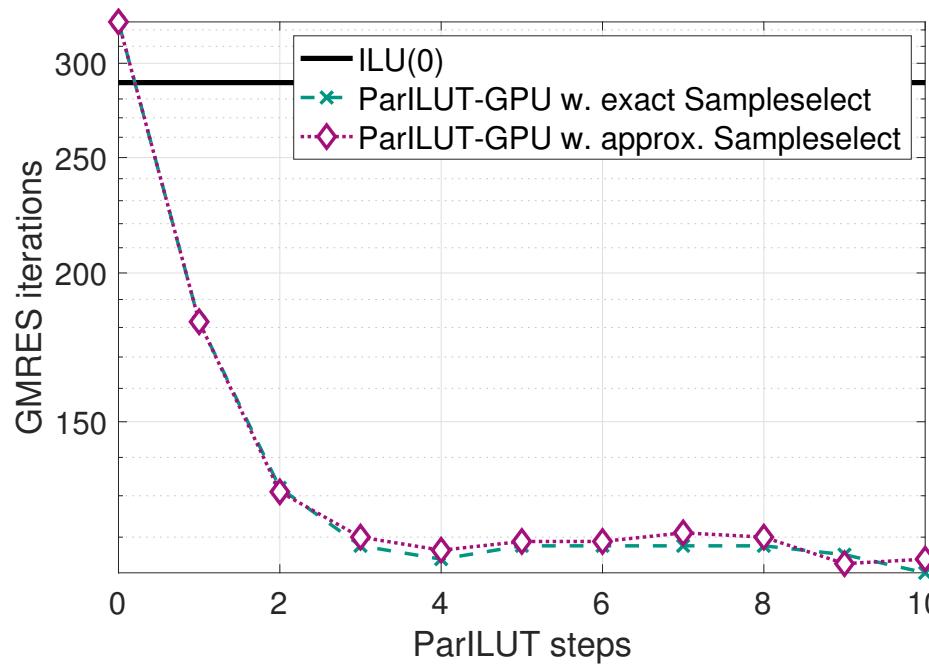


Approximate selection on  $2^{28}$  uniformly distributed single precision values using 1 recursion level, only.

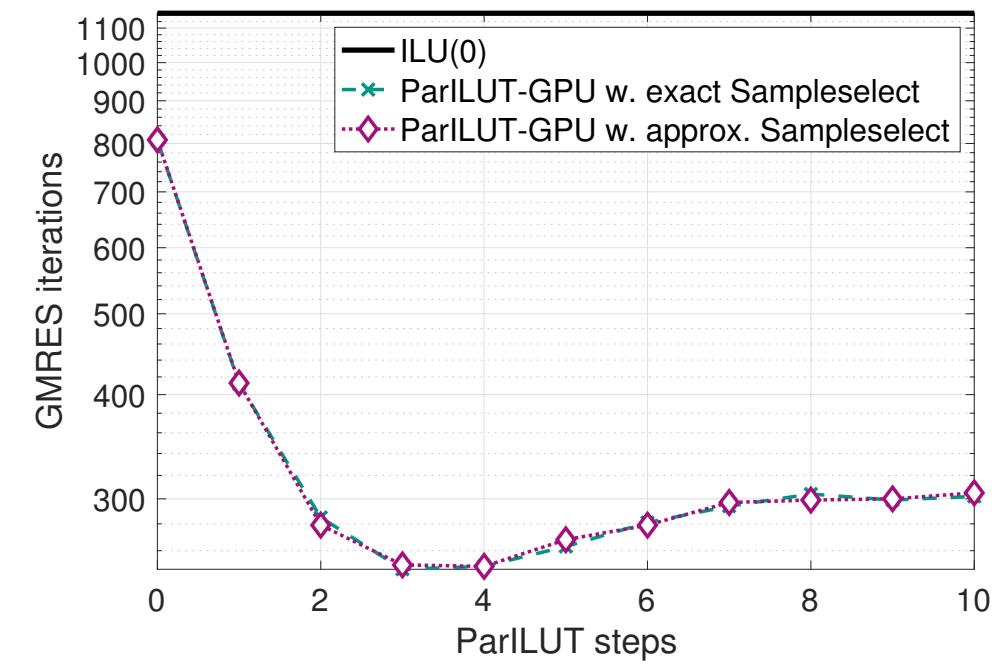
# Approximate Threshold Selection

*Impact of exact/approximate SampleSelect on ParILUT preconditioner quality*

AN15



AN16



# Parallelism inside the blocks: Candidate search

## Identify locations that are symbolically nonzero:

- Combination of sparse matrix product and sparse matrix sums.
- Building blocks available in SparseBLAS.
- Blocks can be combined into one kernel for higher (memory) efficiency.
- Kernel can be parallelized by rows.
- *Cost heavily dependent on sparsity pattern.*
- *Kernel performance bound by memory bandwidth.*
- *Design specialized Kernel*<sup>2</sup>.

Identify locations with nonzero ILU residual.

$$\mathcal{S}^* = (\mathcal{S}(A) \cup \mathcal{S}(L \cdot U)) \setminus \mathcal{S}(L + U)$$

sparse matrix product

sparse matrix sum

sparse matrix sum

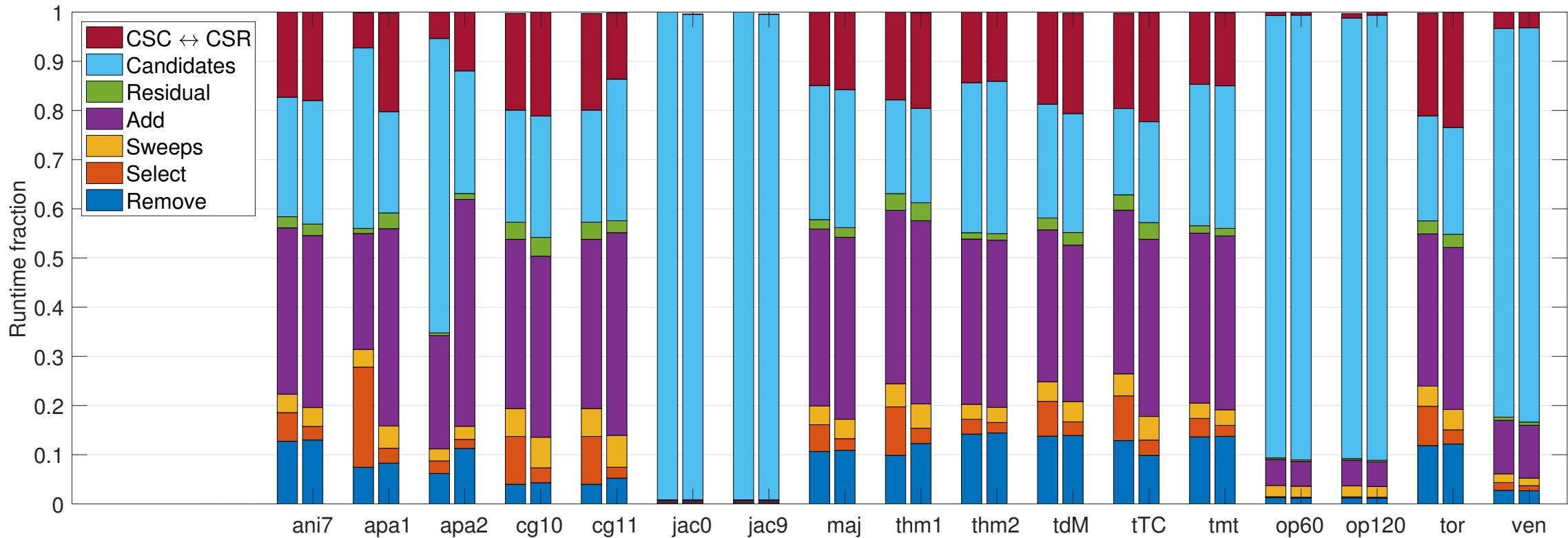
<sup>2</sup>Anzt et al. “ParILUT – A new parallel threshold ILU”. In: SIAM J. on Sci. Comp. (2018).

# ParILUT Performance on GPUs

*Impact of exact(1<sup>st</sup> bar) / approximate (2<sup>nd</sup> bar) SampleSelect on ParILUT runtime breakdown*

NVIDIA V100 GPU.

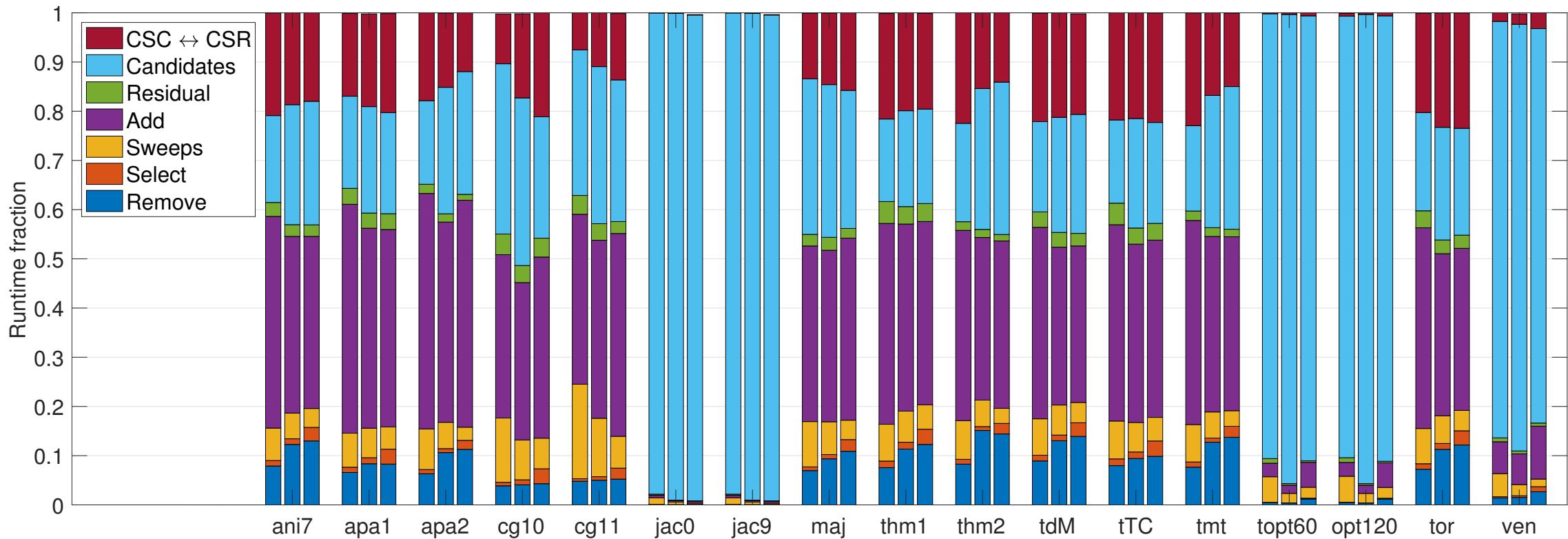
Matrices taken from Suite Sparse Matrix Collection.



# ParILUT Performance on GPUs

*ParILUT performance across different GPU generations:* 1<sup>st</sup> bar: NVIDIA K40  
2<sup>nd</sup> bar: NVIDIA P100  
3<sup>rd</sup> bar: NVIDIA V100

Matrices taken from Suite Sparse Matrix Collection.



# The Challenge: Iterative Solution of a Sparse Linear System

---

We **iteratively solve** a linear system of the form  $Ax = b$

Where  $A \in \mathbb{R}^{n \times n}$  nonsingular and  $b, x \in \mathbb{R}^n$ .

The **convergence rate** typically depends on the **conditioning** of the linear system, which is the ratio between the largest and smallest eigenvalue.

$$\text{cond}_2(A) = \frac{\lambda_{\max}}{\lambda_{\min}} = \frac{\frac{1}{\lambda_{\min}}}{\frac{1}{\lambda_{\max}}} = \text{cond}_2(A^{-1})$$

# The Challenge: Iterative Solution of a Sparse Linear System

---

We **iteratively solve** a linear system of the form  $Ax = b$

Where  $A \in \mathbb{R}^{n \times n}$  nonsingular and  $b, x \in \mathbb{R}^n$ .

The **convergence rate** typically depends on the **conditioning** of the linear system, which is the ratio between the largest and smallest eigenvalue.

$$\text{cond}_2(A) = \frac{\lambda_{\max}}{\lambda_{\min}} = \frac{\frac{1}{\lambda_{\min}}}{\frac{1}{\lambda_{\max}}} = \text{cond}_2(A^{-1})$$

Using a **preconditioner**  $M \approx A^{-1}$ , we can **transform** the linear system into a system with a lower condition number:

$$MAx = Mb \text{ (left preconditioned)}$$

# The Challenge: Iterative Solution of a Sparse Linear System

---

We **iteratively solve** a linear system of the form  $Ax = b$

Where  $A \in \mathbb{R}^{n \times n}$  nonsingular and  $b, x \in \mathbb{R}^n$ .

The **convergence rate** typically depends on the **conditioning** of the linear system, which is the ratio between the largest and smallest eigenvalue.

$$\text{cond}_2(A) = \frac{\lambda_{\max}}{\lambda_{\min}} = \frac{\frac{1}{\lambda_{\min}}}{\frac{1}{\lambda_{\max}}} = \text{cond}_2(A^{-1})$$

Using a **preconditioner**  $M \approx A^{-1}$ , we can **transform** the linear system into a system with a lower condition number:

$$MAx = Mb \text{ (left preconditioned)}$$

If we now apply the iterative solver to the preconditioned System  $MAx = Mb$ , we usually get faster convergence.

Assume  $M = A^{-1}$ , then:  $x = MAx = Mb$ .

*Solution right available, but computing  $M = A^{-1}$  is expensive...*

# The Challenge: Iterative Solution of a Sparse Linear System

We **iteratively solve** a linear system of the form  $Ax = b$   
Where  $A \in \mathbb{R}^{n \times n}$  nonsingular and  $b, x \in \mathbb{R}^n$ .

The **convergence rate** typically depends on the **conditioning** of the linear system, which is the ratio between the largest and smallest eigenvalue.

$$\text{cond}_2(A) = \frac{\lambda_{\max}}{\lambda_{\min}} = \frac{\frac{1}{\lambda_{\min}}}{\frac{1}{\lambda_{\max}}} = \text{cond}_2(A^{-1})$$

Explicitly forming  $MA$  is very expensive. The preconditioner is usually **applied implicitly** in the different iteration steps.

Using a **preconditioner**  $M \approx A^{-1}$ , we can **transform** the linear system into a system with a lower condition number:

$$MAx = Mb \text{ (left preconditioned)}$$

If we now apply the iterative solver to the preconditioned System  $MAx = Mb$ , we usually get faster convergence.

Assume  $M = A^{-1}$ , then:  $x = MAx = Mb$ .

*Solution right available, but computing  $M = A^{-1}$  is expensive...*

# The Challenge: Iterative Solution of a Sparse Linear System

We **iteratively solve** a linear system of the form  $Ax = b$   
Where  $A \in \mathbb{R}^{n \times n}$  nonsingular and  $b, x \in \mathbb{R}^n$ .

The **convergence rate** typically depends on the **conditioning** of the linear system, which is the ratio between the largest and smallest eigenvalue.

$$\text{cond}_2(A) = \frac{\lambda_{\max}}{\lambda_{\min}} = \frac{\frac{1}{\lambda_{\min}}}{\frac{1}{\lambda_{\max}}} = \text{cond}_2(A^{-1})$$

Using a **preconditioner**  $M \approx A^{-1}$ , we can **transform** the linear system into a system with a lower condition number:

$$MAx = Mb \text{ (left preconditioned)}$$

If we now apply the iterative solver to the preconditioned System  $MAx = Mb$ , we usually get faster convergence.

Assume  $M = A^{-1}$ , then:  $x = MAx = Mb$ .  
*Solution right available, but computing  $M = A^{-1}$  is expensive...*

Explicitly forming  $MA$  is very expensive. The preconditioner is usually **applied implicitly** in the different iteration steps.

Instead of forming the preconditioner  $M \approx A^{-1}$  explicitly, **Incomplete Factorization Preconditioners (ILU)** try to find an **approximate factorization**:

$$A \approx L \cdot U$$

In the application phase, the preconditioner is only given implicitly, requiring **two triangular solves**:

$$\begin{aligned} z_{k+1} &= Mr_{k+1} \\ M^{-1}z_{k+1} &= r_{k+1} \\ \underbrace{LU}_{=:y} z_{k+1} &= r_{k+1} \\ \Rightarrow Ly &= r_{k+1}, \quad Uz_{k+1} = y \end{aligned}$$

# Test matrices

| Matrix        | Origin                     | SPD | Num. Rows | Nz        | Nz/Row |
|---------------|----------------------------|-----|-----------|-----------|--------|
| ANI5          | 2D anisotropic diffusion   | yes | 12,561    | 86,227    | 6.86   |
| ANI6          | 2D anisotropic diffusion   | yes | 50,721    | 349,603   | 6.89   |
| ANI7          | 2D anisotropic diffusion   | yes | 203,841   | 1,407,811 | 6.91   |
| APACHE1       | Suite Sparse [10]          | yes | 80,800    | 542,184   | 6.71   |
| APACHE2       | Suite Sparse               | yes | 715,176   | 4,817,870 | 6.74   |
| CAGE10        | Suite Sparse               | no  | 11,397    | 150,645   | 13.22  |
| CAGE11        | Suite Sparse               | no  | 39,082    | 559,722   | 14.32  |
| JACOBIANMAT0  | Fun3D fluid flow [20]      | no  | 90,708    | 5,047,017 | 55.64  |
| JACOBIANMAT9  | Fun3D fluid flow           | no  | 90,708    | 5,047,042 | 55.64  |
| MAJORBASIS    | Suite Sparse               | no  | 160,000   | 1,750,416 | 10.94  |
| TOPOPT010     | Geometry optimization [24] | yes | 132,300   | 8,802,544 | 66.53  |
| TOPOPT060     | Geometry optimization      | yes | 132,300   | 7,824,817 | 59.14  |
| TOPOPT120     | Geometry optimization      | yes | 132,300   | 7,834,644 | 59.22  |
| THERMAL1      | Suite Sparse               | yes | 82,654    | 574,458   | 6.95   |
| THERMAL2      | Suite Sparse               | yes | 1,228,045 | 8,580,313 | 6.99   |
| THERMOMECH_TC | Suite Sparse               | yes | 102,158   | 711,558   | 6.97   |
| THERMOMECH_DM | Suite Sparse               | yes | 204,316   | 1,423,116 | 6.97   |
| TMT_SYM       | Suite Sparse               | yes | 726,713   | 5,080,961 | 6.99   |
| TORSO2        | Suite Sparse               | no  | 115,967   | 1,033,473 | 8.91   |
| VENKAT01      | Suite Sparse               | no  | 62,424    | 1,717,792 | 27.52  |

# Convergence: GMRES iterations

| Matrix       | no prec. | ILU(0) | ILUT | ParILUT |     |     |     |     |     |
|--------------|----------|--------|------|---------|-----|-----|-----|-----|-----|
|              |          |        |      | 0       | 1   | 2   | 3   | 4   | 5   |
| ANI5         | 882      | 172    | 78   | 278     | 161 | 105 | 84  | 74  | 66  |
| ANI6         | 1,751    | 391    | 127  | 547     | 315 | 211 | 168 | 143 | 131 |
| ANI7         | 3,499    | 828    | 290  | 1,083   | 641 | 459 | 370 | 318 | 289 |
| CAGE10       | 20       | 8      | 8    | 9       | 7   | 8   | 8   | 8   | 8   |
| CAGE11       | 21       | 9      | 8    | 9       | 7   | 7   | 7   | 7   | 7   |
| JACOBIANMAT0 | 315      | 40     | 34   | 63      | 36  | 33  | 33  | 33  | 33  |
| JACOBIANMAT9 | 539      | 66     | 65   | 110     | 60  | 55  | 54  | 53  | 53  |
| MAJORBASIS   | 95       | 15     | 9    | 26      | 12  | 11  | 11  | 11  | 11  |
| TOPOPT010    | 2,399    | 565    | 303  | 835     | 492 | 375 | 348 | 340 | 339 |
| TOPOPT060    | 2,852    | 666    | 397  | 963     | 584 | 445 | 417 | 412 | 410 |
| TOPOPT120    | 2,765    | 668    | 396  | 959     | 584 | 445 | 416 | 408 | 408 |
| TORSO2       | 46       | 10     | 7    | 18      | 8   | 6   | 7   | 7   | 7   |
| VENKAT01     | 195      | 22     | 17   | 42      | 18  | 17  | 17  | 17  | 17  |

# Convergence: CG iterations

| Matrix        | no prec. | IC(0) | ICT   | ParICT |       |       |       |       |       |
|---------------|----------|-------|-------|--------|-------|-------|-------|-------|-------|
|               |          |       |       | 0      | 1     | 2     | 3     | 4     | 5     |
| ANI5          | 951      | 226   | —     | 297    | 184   | 136   | 108   | 93    | 86    |
| ANI6          | 1,926    | 621   | —     | 595    | 374   | 275   | 219   | 181   | 172   |
| ANI7          | 3,895    | 1,469 | —     | 1,199  | 753   | 559   | 455   | 405   | 377   |
| APACHE1       | 3,727    | 368   | 331   | 1,480  | 933   | 517   | 321   | 323   | 323   |
| APACHE2       | 4,574    | 1,150 | 785   | 1,890  | 1,197 | 799   | 766   | 760   | 754   |
| THERMAL1      | 1,640    | 453   | 412   | 626    | 447   | 409   | 389   | 385   | 383   |
| THERMAL2      | 6,253    | 1,729 | 1,604 | 2,372  | 1,674 | 1,503 | 1,457 | 1,472 | 1,433 |
| THERMOMECH_DM | 21       | 8     | 8     | 8      | 7     | 7     | 7     | 7     | 7     |
| THERMOMECH_TC | 21       | 8     | 7     | 8      | 7     | 7     | 7     | 7     | 7     |
| TMT_SYM       | 5,481    | 1,453 | 1,185 | 1,963  | 1,234 | 1,071 | 1,012 | 992   | 1,004 |
| TOPOPT010     | 2,613    | 692   | 331   | 845    | 551   | 402   | 342   | 316   | 313   |
| TOPOPT060     | 3,123    | 871   | —     | 988    | 749   | 693   | 1,116 | —     | —     |
| TOPOPT120     | 3,062    | 886   | —     | 991    | 837   | 784   | 2,185 | —     | —     |

# Performance

Intel Xeon Phi 7250 "Knights Landing"

68 cores @1.40 GHz,

16GB MCDRAM @490 GB/s

Runtime of 5 ParILUT / ParICT steps and speedup over SuperLU ILUT\*.

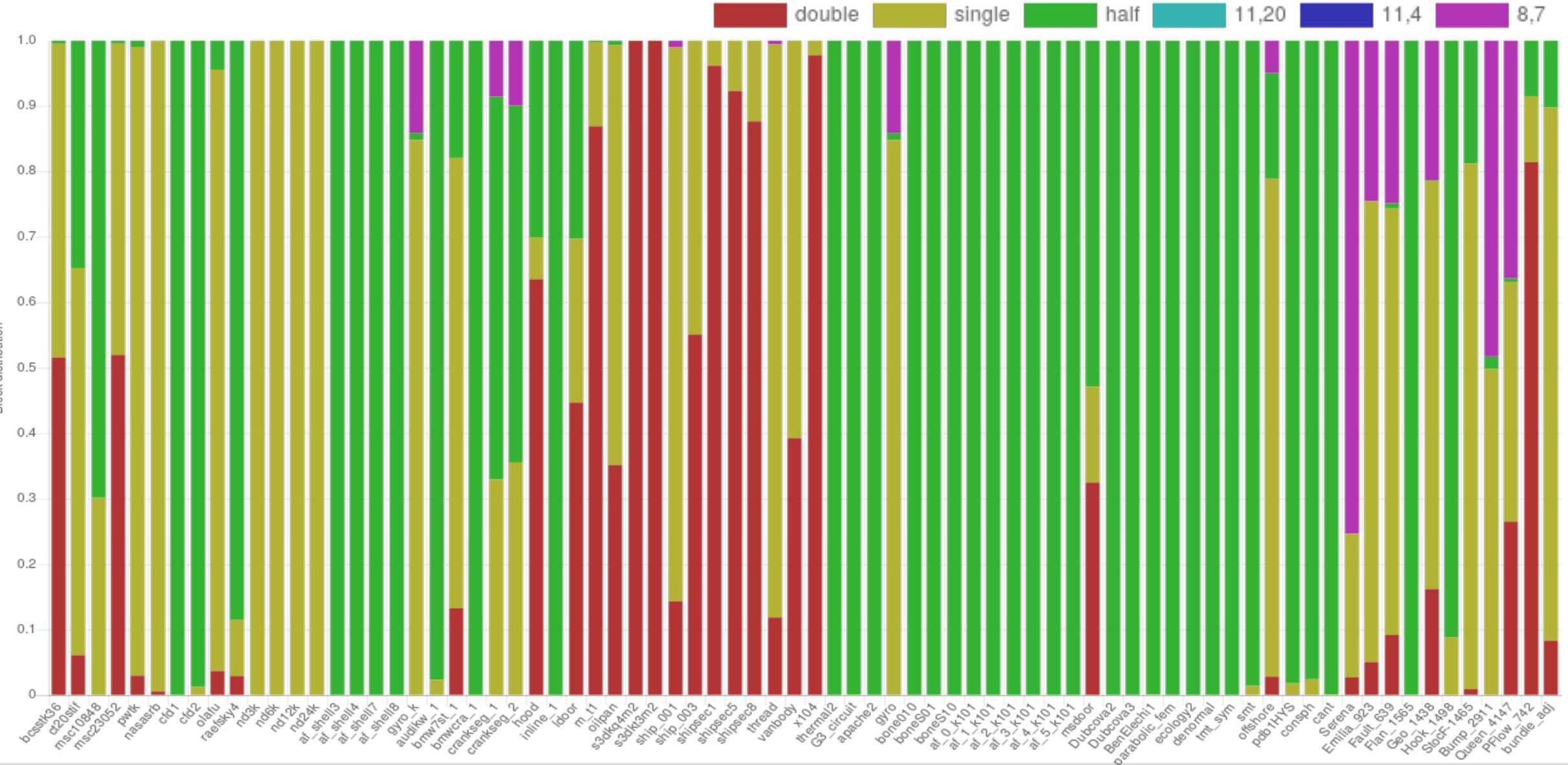
| Matrix       | Origin                       | Rows      | Nonzeros  | Ratio |  |  | SuperLU  | ParILUT |        | ParICT |        |
|--------------|------------------------------|-----------|-----------|-------|--|--|----------|---------|--------|--------|--------|
| ani7         | 2D Anisotropic Diffusion     | 203,841   | 1,407,811 | 6.91  |  |  | 10.48 s  | 0.45 s  | 23.34  | 0.30 s | 35.16  |
| apache2      | Suite Sparse Matrix Collect. | 715,176   | 4,817,870 | 6.74  |  |  | 62.27 s  | 1.24 s  | 50.22  | 0.65 s | 95.37  |
| cage11       | Suite Sparse Matrix Collect. | 39,082    | 559,722   | 14.32 |  |  | 60.89 s  | 0.54 s  | 112.56 | --     | --     |
| jacobianMat9 | Fun3D Fluid Flow Problem     | 90,708    | 5,047,042 | 55.64 |  |  | 153.84 s | 7.26 s  | 21.19  | --     | --     |
| thermal2     | Thermal Problem (Suite Sp.)  | 1,228,045 | 8,580,313 | 6.99  |  |  | 91.83 s  | 1.23 s  | 74.66  | 0.68 s | 134.25 |
| tmt_sym      | Suite Sparse Matrix Collect. | 726,713   | 5,080,961 | 6.97  |  |  | 53.42 s  | 0.70 s  | 76.21  | 0.41 s | 131.25 |
| topopt120    | Geometry Optimization        | 132,300   | 8,802,544 | 66.53 |  |  | 44.22 s  | 14.40 s | 3.07   | 8.24 s | 5.37   |
| torso2       | Suite Sparse Matrix Collect. | 115,967   | 1,033,473 | 8.91  |  |  | 10.78 s  | 0.27 s  | 39.92  | --     | --     |
| venkat01     | Suite Sparse Matrix Collect. | 62,424    | 1,717,792 | 27.52 |  |  | 8.53 s   | 0.74 s  | 11.54  | --     | --     |

\*We thank Sherry Li and Meiyue Shao for technical help in generating the performance numbers.

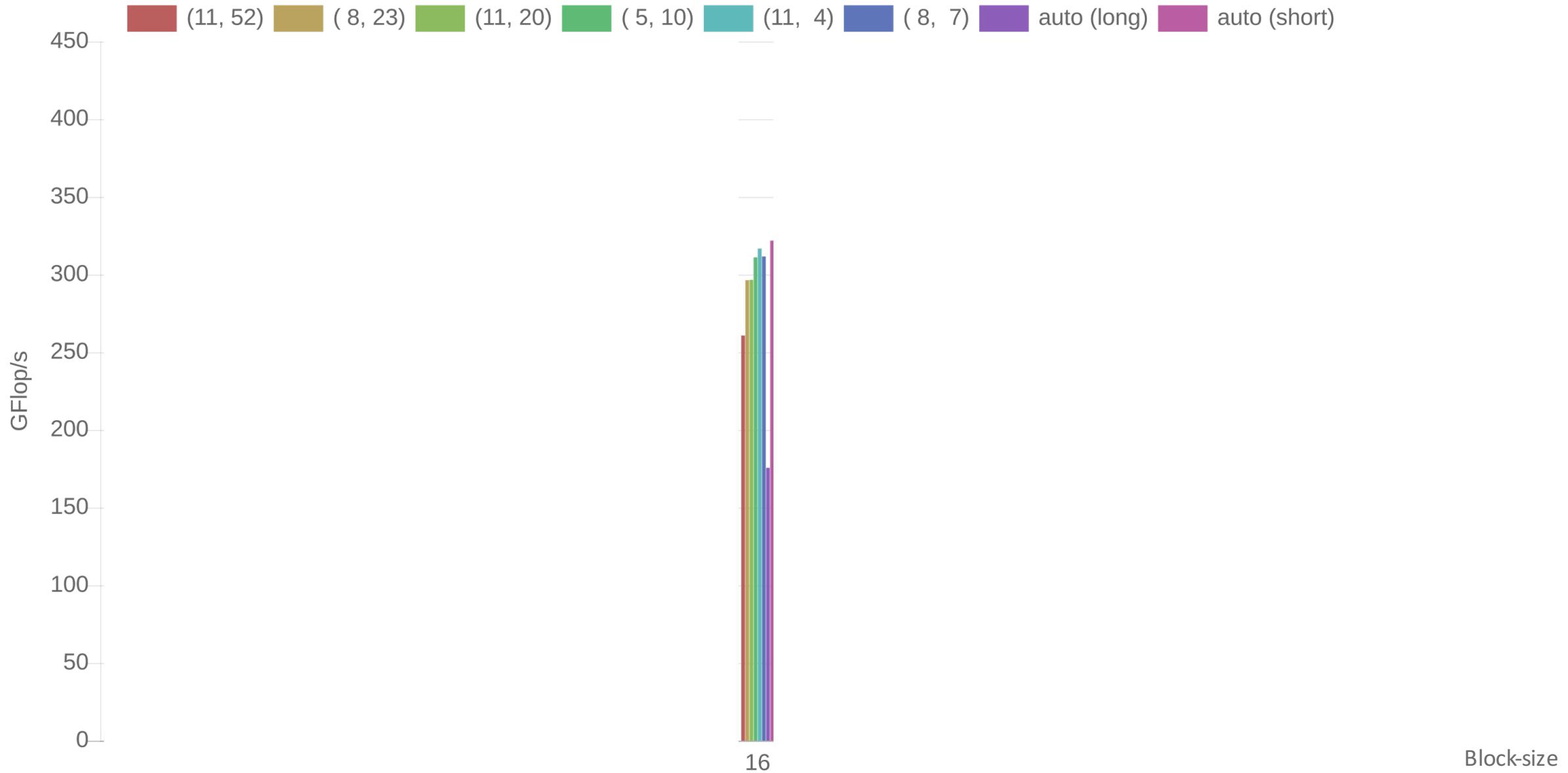
# Precision distribution in Adaptive Block-Jacobi



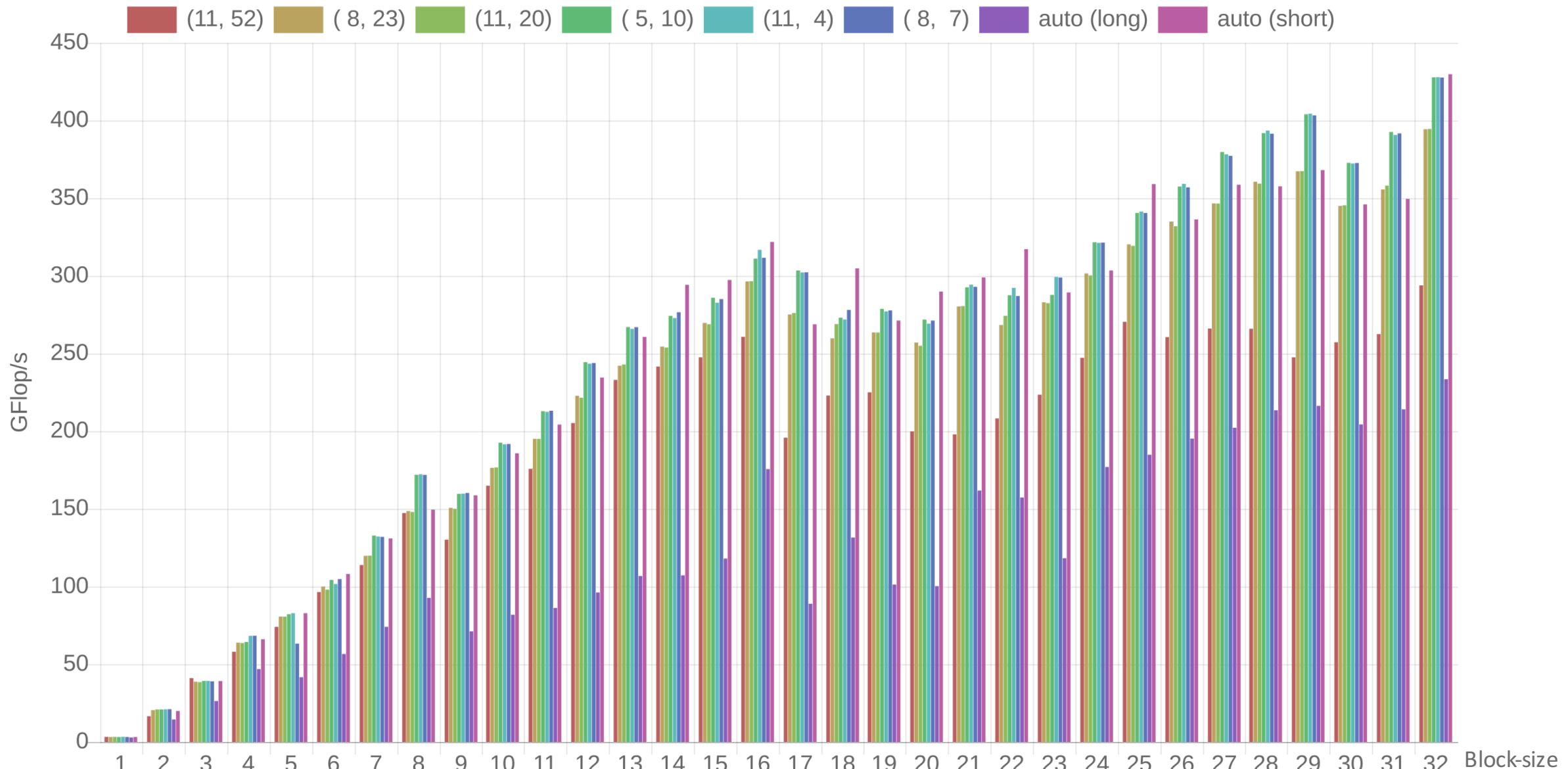
# Precision distribution in Adaptive Block-Jacobi



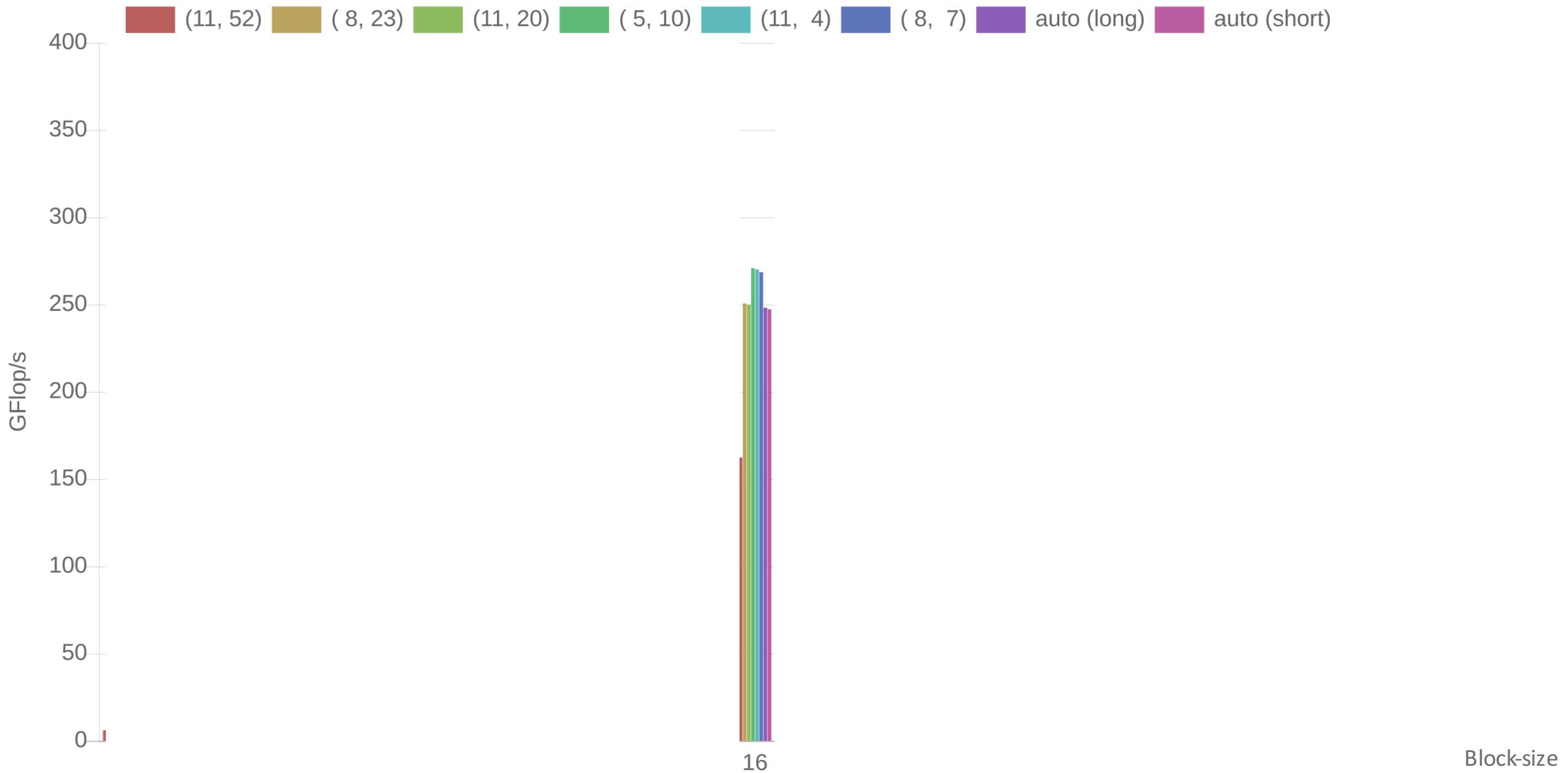
# Adaptive Block-Jacobi Generation



# Adaptive Block-Jacobi Generation



# Adaptive Block-Jacobi Application



# Adaptive Block-Jacobi Application

