DESIGN, AUTOMATION & TEST IN EUROPE

14 – 18 March, 2016 · ICC · Dresden · Germany

The European Event for Electronic
System Design & Test

# Sparsity-Oriented Sparse Solver Design for Circuit Simulation


**Xiaoming Chen, Lixue Xia, Yu Wang, Huazhong Yang**

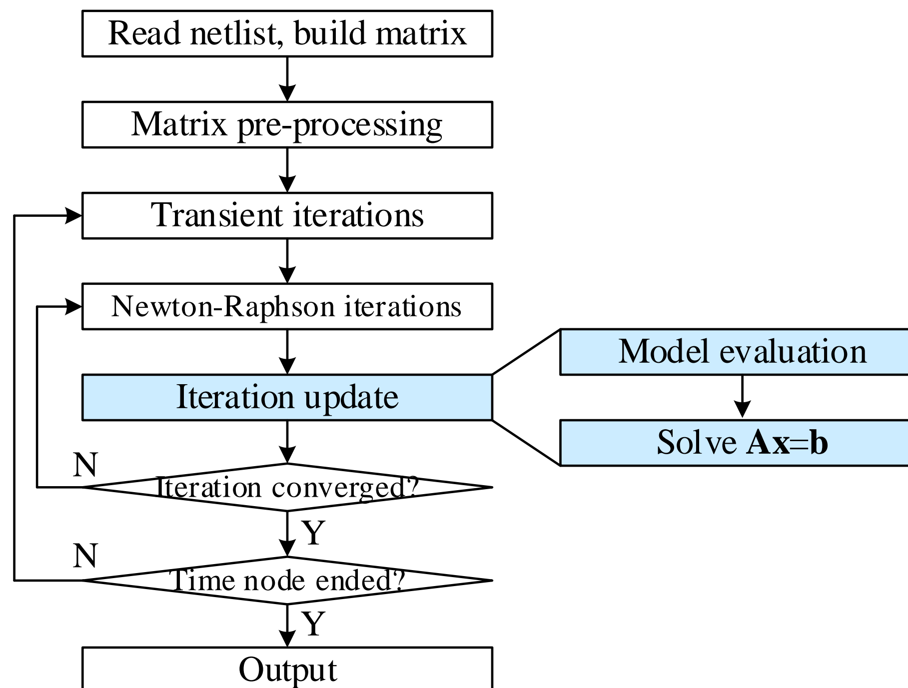
Department of Electronic Engineering,

Tsinghua National Laboratory for Information Science and Technology,

Tsinghua University, Beijing 100084, China

# SPICE

- **SPICE: most widely used circuit simulator for transistor-level simulation**
- **Performance challenges**
  - **Increasing circuit size and device model complexity**
  - **Days to weeks to do post-layout simulation**



# Bottlenecks of SPICE

- Model evaluation
  - Dominate in pre-layout simulation
  - Many approaches to accelerate
    - Parallel evaluation
    - Simplified model
    - Lookup table
- Sparse linear solver (Ax=b)
  - 50% to 75% of total time in post-layout simulation
  - Difficult to accelerate or parallelize
    - Strong data dependence in sparse LU factorization: low parallelism
    - Highly irregular pattern of nonzero elements: high cache miss
    - Low computation-to-memory ratio: bandwidth constrained

# Existing Sparse Solvers

- **SPARSE** [Kundert, 1986]
  - **Circuit simulation, sequential, old for modern CPUs**
- **KLU** [Davis, TOMS 2010]
  - **Circuit simulation, sequential**
- **NICSLU (our previous work)** [Chen, TCAD 2013]
  - **Circuit simulation, parallel**
- **PARDISO** [Schenk, FGCS 2004], **SuperLU** [Demmel, SIMAX 1999], **MUMPS** [Amestoy, SIMAX 2001], **UMFPACK** [Davis, TOMS 2004] **……**
  - **High-performance for general-purpose, not good in circuit simulation**

# Outline

- Introduction
- Background
  - G/P Algorithm
  - Scatter-Gather in G/P Algorithm
  - Motivation
- Proposed Methods
- Results
- Conclusions

# Gilbert-Peierls (G/P) Algorithm

- **Basic algorithm adopted by KLU and this work**
- **Sparse “left-looking”**
  - **Factorize a matrix column by column, from left to right**
  - **When factorizing a column, use dependent columns on the left side to update it**

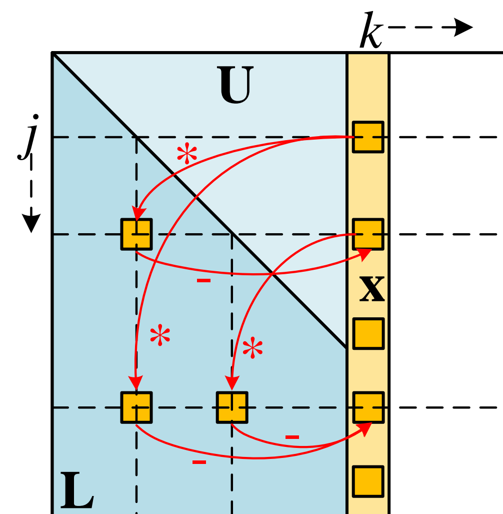
---

**Algorithm 1** G/P Algorithm (no pivoting) [15].

---

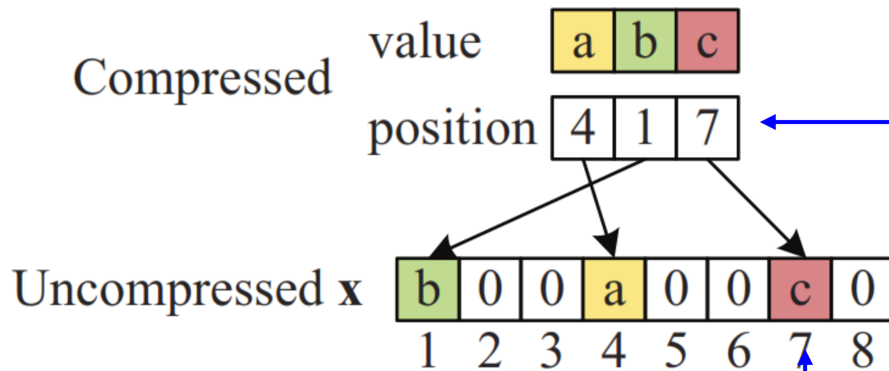
1: $\mathbf{L} = \mathbf{I}$; //I is the identity matrix
2: **for** $k = 1 : N$ **do**
3: $\quad \mathbf{x} = \mathbf{A}(:, k)$; //x is an uncompressed vector of length $N$
4: $\quad$ **for** $j = 1 : k-1$ where $\mathbf{U}(j,k) \neq 0$ **do**
5: $\quad\quad \mathbf{x}(j+1:N) -= \mathbf{x}(j) \cdot \mathbf{L}(j+1:N, j)$;
6: $\quad$ **end for**
7: $\quad \mathbf{U}(1:k, k) = \mathbf{x}(1:k)$;
8: $\quad \mathbf{L}(k:N, k) = \dfrac{\mathbf{x}(k:N)}{\mathbf{x}(k)}$;
9: **end for**

---



# Sparse Matrix Storage

- Sparse matrix is stored in a compressed format
  - Only values and indexes of nonzero elements are stored
- Accessing an element in a compressed format cannot be done in O(1) time complexity
  - Its position in compressed format is unknown, unless traversing

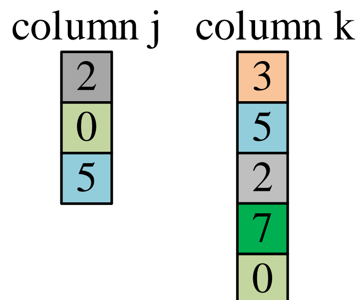



Accessing x[7] in compressed x: traverse the vector and find it at index [2]: O(n)

Accessing x[7] in uncompressed x: directly visit x[7]: O(1)

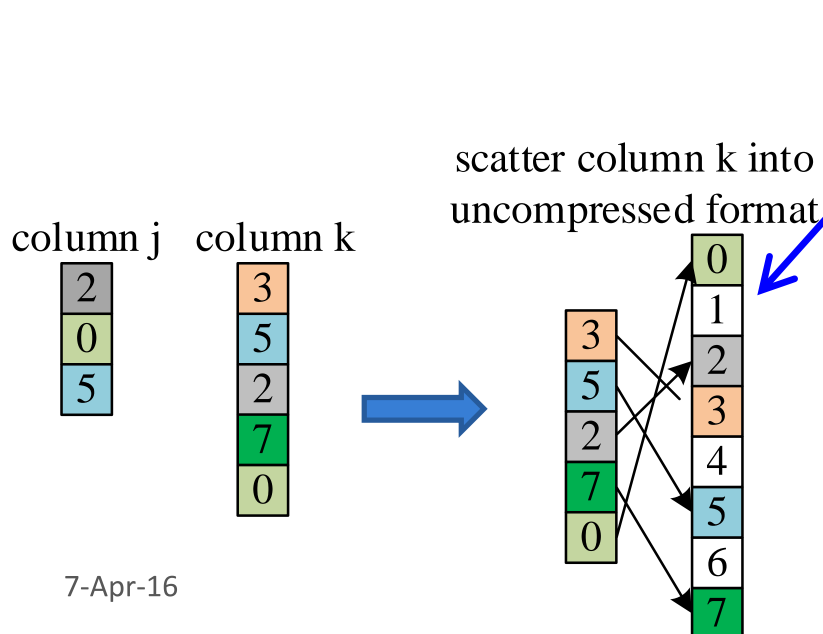
# Scatter-Gather in G/P Algorithm

- **When using column j to update column k (j<k)**
  - **For each nonzero in column j, its corresponding position in column k is unknown**
- **Solution in G/P algorithm: scatter-gather**



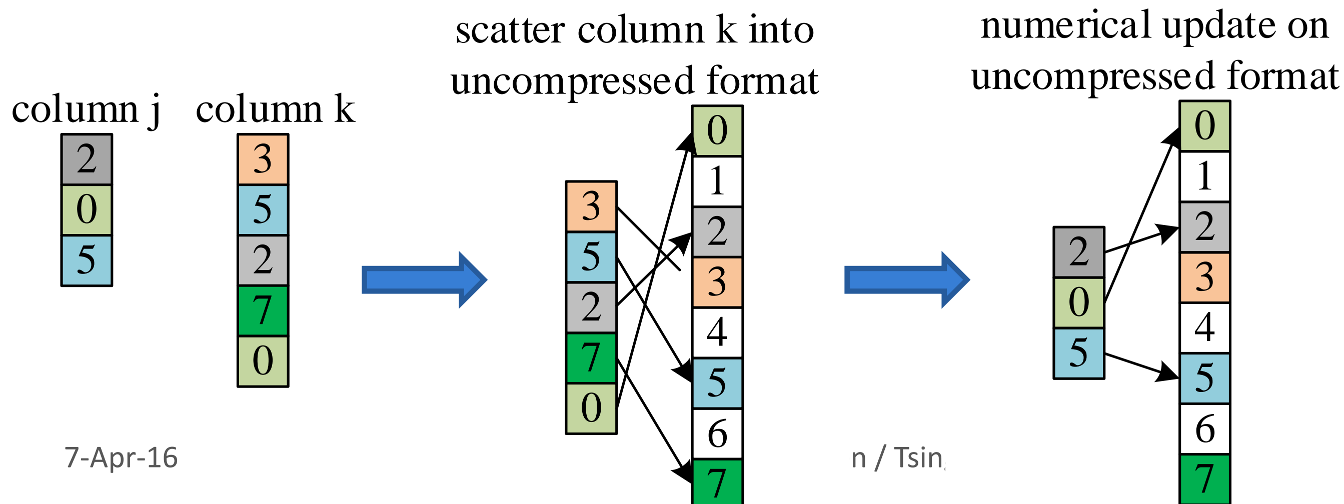
# Scatter-Gather in G/P Algorithm

- When using column j to update column k (j<k)
  - For each nonzero in column j, its position in column k is unknown, and vice versa
- Solution in G/P algorithm: scatter-gather
  - Scatter compressed format into uncompressed format
    - An additional vector of length N is required



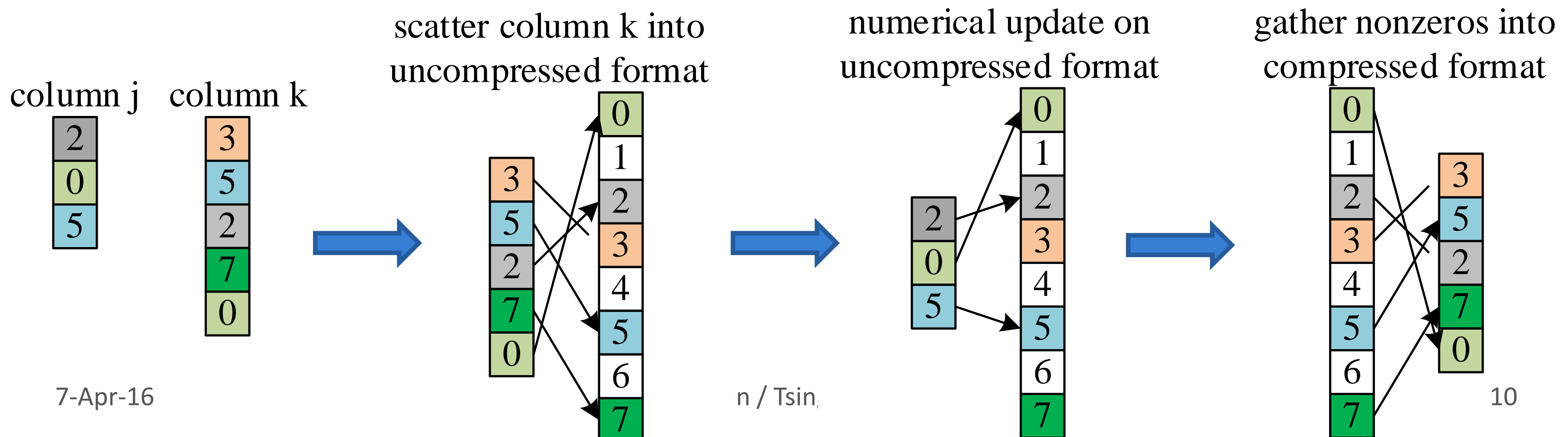
# Scatter-Gather in G/P Algorithm

- When using column j to update column k (j<k)
  - For each nonzero in column j, its position in column k is unknown, and vice versa
- Solution in G/P algorithm: scatter-gather
  - Scatter compressed format into uncompressed format
  - Accumulate numerical update on uncompressed format



# Scatter-Gather in G/P Algorithm

- **When using column j to update column k (j<k)**
  - **For each nonzero in column j, its position in column k is unknown, and vice versa**
- **Solution in G/P algorithm: scatter-gather**
  - **Scatter compressed format into uncompressed format**
  - **Accumulate numerical update on uncompressed format**
  - **Gather nonzeros into compressed format**



# Motivation

- If matrix is very sparse, scatter-gather may take too much time
  - Too few numerical computations
  - Uncompressed format is too sparse → high cache miss
  - Solution: avoid scatter-gather by recording positions of nonzeros in compressed format
- If matrix is dense, column-based G/P algorithm may lead to low performance
  - Indexes are nearly continuous → too many unnecessary indirect accesses
  - Solution: gather nonzeros into dense sub-blocks and use dense kernels

# Outline

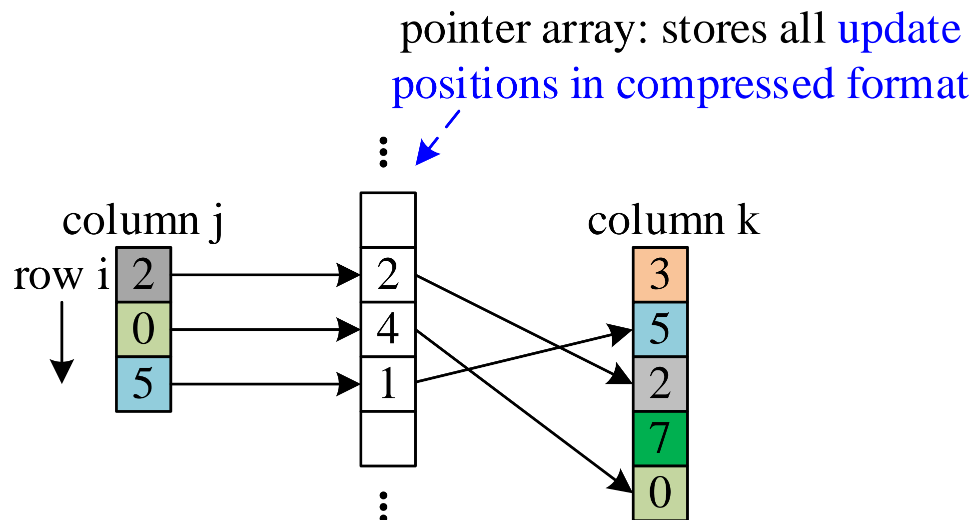
- Introduction
- Background
- Proposed Methods
  - Map Algorithm
  - Supernode Algorithm
  - Overall Flow
- Results
- Conclusions

# Map Algorithm

- **Intention: avoid scatter-gather by recording positions of nonzeros in compressed format**
- **Ideal case: substitute data and compile LU factorization process into a sequence of assembly instructions ---- running instructions in sequence finishes factorization**
  - **We need a compiler!**
  - **Data-dependent: compile for each matrix**
  - **We need a universal and easy approach**

# Map Algorithm

- **Map:** **a pointer array (*ptr*) to record all update positions (in compressed format) in sequence**
- **For each numerical update**
  - **Get update position *ptr***
  - **Perform numerical update *ptr-=U(j,k)*L(i,j)**
  - **++*ptr* (to next update position)**

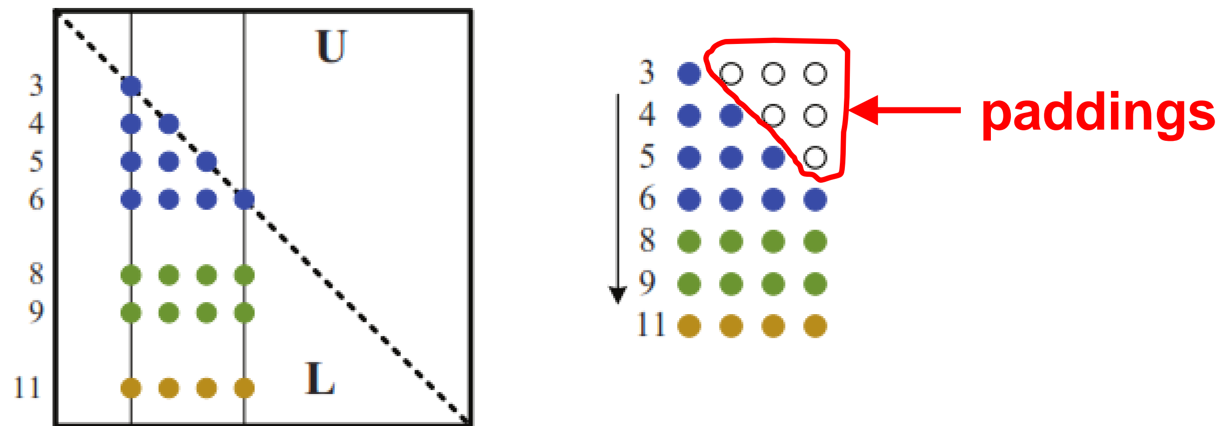


# Map Algorithm

- Scatter-gather is avoided
- Only suitable for very sparse matrix
  - Scatter-gather cost can be ignored for dense matrix
  - Length of map = # of floating-point operations (flops), too long map for dense matrix
- How to build map?
  - Go through LU factorization process and record all update positions in a sequence
  - One-time work

# Supernode Algorithm

- **What is supernode? Not a new concept**
  - **A set of successive columns of L with triangular diagonal block full and same structure below the diagonal block**
    **[Demmel, SIMAX 1999]**
  - **Stored by a dense matrix by padding upper triangular block**




(a) Definition of supernode. (b) Storage of a supernode.

# G/P Algorithm with Supernode

- **Conventional G/P: use columns on the left side to update a column (column-column)**
- **Supernode G/P: use supernodes on the left side to update a column (supernode-column)**

**Algorithm 2** Supernode-column algorithm.

1: $\mathbf{L} = \mathbf{I}$; //I is the identity matrix
2: **for** $k = 1 : N$ **do**
3: $\quad \mathbf{x} = \mathbf{A}(:, k)$; //x is an uncompressed vector of length $N$
4: $\quad$ **for** $j = 1 : k - 1$ where $\mathbf{U}(j, k) \neq 0$ **do**
5: $\quad\quad$ **if** column $j$ has not been used for update column $k$ **then**
6: $\quad\quad\quad$ **if** column $j$ belongs to a supernode that ends at column $s$ **then**
$\quad\quad\quad$ //perform supernode-column update
7: $\quad\quad\quad\quad \mathbf{x}(j : s) = \mathbf{L}(j : s, j : s)^{-1} \cdot \mathbf{x}(j : s)$;
8: $\quad\quad\quad\quad \mathbf{x}(s + 1 : N) -= \mathbf{L}(s + 1 : N, j : s) \cdot \mathbf{x}(j : s)$;
9: $\quad\quad\quad$ **else** //perform column-column update
10: $\quad\quad\quad\quad \mathbf{x}(j + 1 : N) -= \mathbf{x}(j) \cdot \mathbf{L}(j + 1 : N, j)$;
11: $\quad\quad\quad$ **end if**
12: $\quad\quad$ **end if**
13: $\quad$ **end for**
14: $\quad \mathbf{U}(1 : k, k) = \mathbf{x}(1 : k)$;
15: $\quad \mathbf{L}(k : N, k) = \dfrac{\mathbf{x}(k : N)}{\mathbf{x}(k)}$;
16: **end for**

Basic Linear Algebra Subprograms

**Implemented by two BLAS routines dtrsv and dgemv**

# Supernode Algorithm

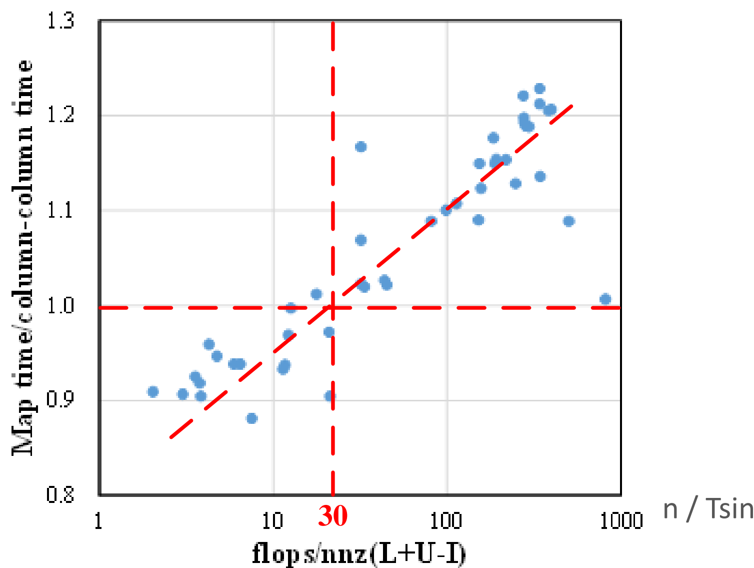
- **Suitable for dense matrix**
  - **Indirect memory accesses are avoided**
  - **Improved cache performance due to dense sub-matrix storage**
  - **High computational performance by utilizing BLAS**
- **How to detect supernodes?**
  - **For column k, check with column k-1**
    - **# of nonzeros in L(:, k) == # of nonzeros in L(:, k-1) - 1 ?**
    - **Symbolic pattern of L(:, k) ∈ symbolic pattern of L(:, k-1) ?**
- **Difference from SuperLU, PARDISO?**
  - **Supernode-supernode, suitable for highly dense matrix**

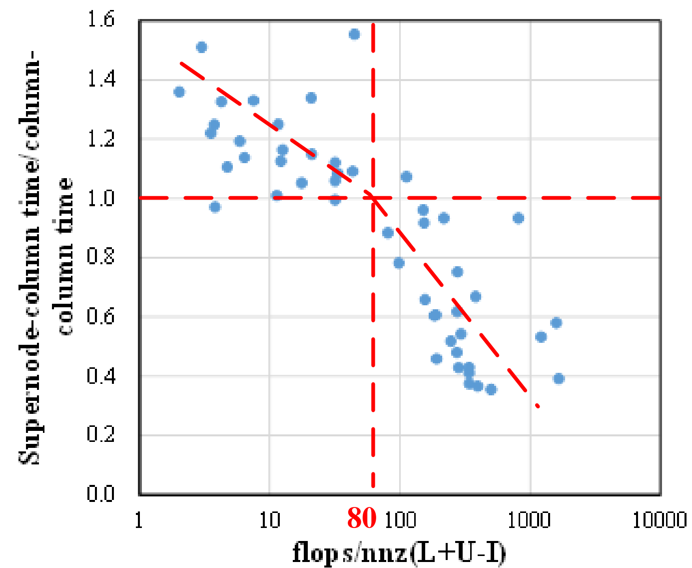
# Algorithm Selection

- How to evaluate sparsity? [Davis, TOMS 2008]

  Average flops per nonzero = flops / NNZ(L+U-I)

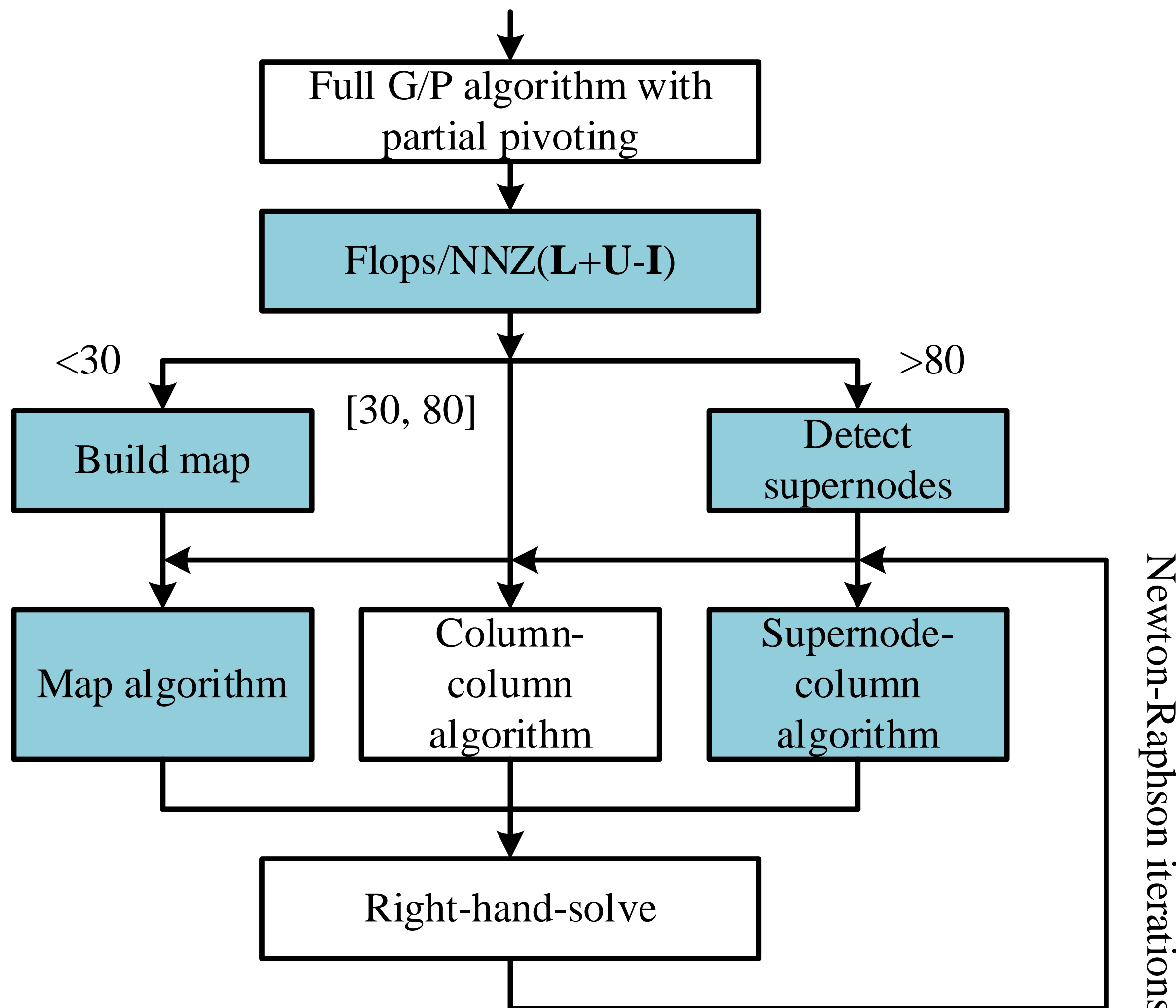
- How to determine optimal method?
  - According to the sparsity ratio
  - Flops/NNZ(L+U-I)<30, use map algorithm
  - Flops/NNZ(L+U-I)>80, use supernode-column algorithm



# Overall Flow

- **How to combine all these together?**



# Outline

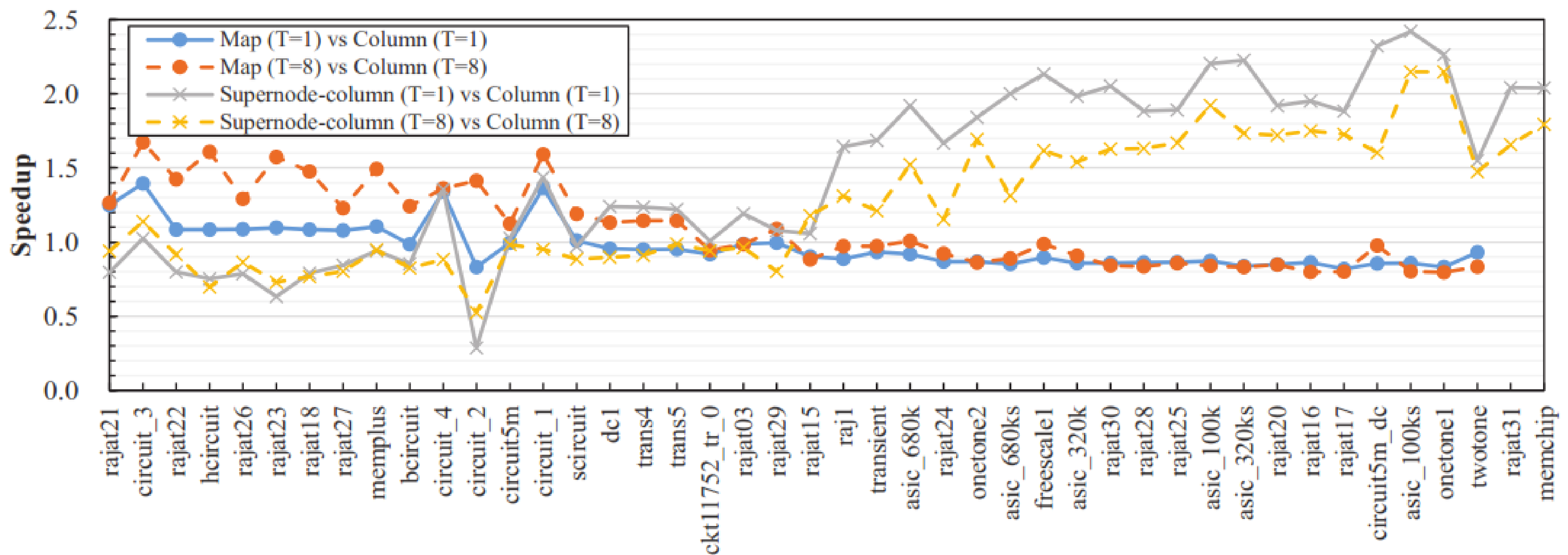
- Introduction
- Background
- Proposed Methods
- **Results**
- **Conclusions**

# Experimental Setup

- **Intel Xeon E5-2690 CPU**
- **64GB memory**
- **Linux server with Intel C compiler**
- **40+ circuit matrices with different sparsity from the University of Florida Sparse Matrix Collection**

# Speedup

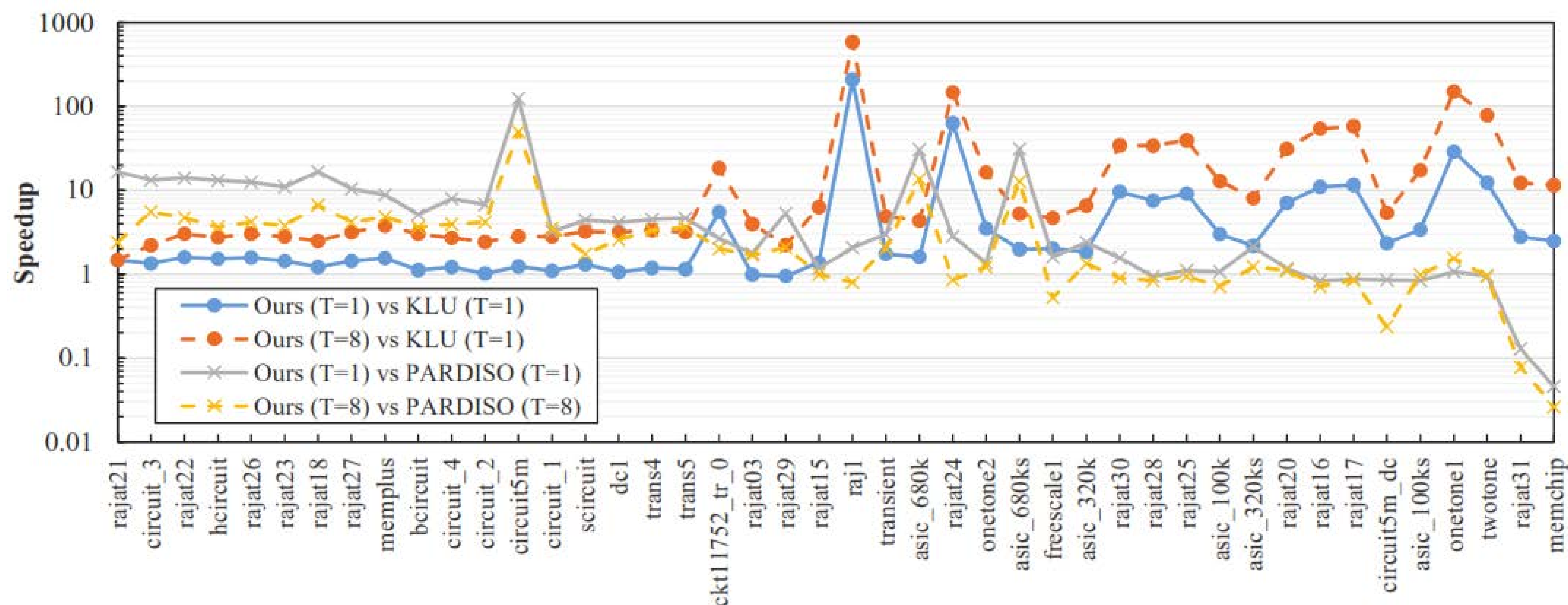
- **Combined solver (map, column-column & supernode-column) achieves 1.5X speedup compared with pure G/P algorithm (column-column)**




Sorted by flops/NNZ(L+U-I)

# Benchmark Results

- **Our solver vs. KLU (sequential) and PARDISO (parallel)**




| | Geometric mean |
|---|---|
| Ours (T=1) vs KLU (T=1) | 2.80 |
| Ours (T=8) vs KLU (T=1) | 8.36 |
| Ours (T=1) vs PARDISO (T=1) | 3.17 |
| Ours (T=8) vs PARDISO (T=8) | 1.78 |

# NGSpice Results

- **NGSpice total TRAN simulation time**
  - **Post-layout & power grid cases**
  - **NGSpice with KLU vs. NGSpice with our solver**

| Benchmark | Dimen. | Transis. | KLU | Ours (T=1) | Ours (T=4) |
|---|---|---|---|---|---|
| ckt1 | 21110 | 200 | 144.6 | 94.2 (1.53×) | 47.4 (3.05×) |
| ckt2 | 82210 | 400 | 1722.4 | 633.1 (2.72×) | 297.5 (5.79×) |
| ckt3 | 183309 | 600 | 5862.6 | 2377.6 (2.47×) | 1366.1 (4.29×) |
| ibmpg1t mod. | 55356 | 242 | 46.2 | 45.4 (1.02 ×) | 36.6 (1.26×) |
| ibmpg2t mod. | 167779 | 768 | 963.9 | 516.3 (1.87 ×) | 308.2 (3.13 ×) |
| **Average** | | | | **1.92**× | **3.50**× |

- **50X-150X faster than default solver of NGSpice (SPARSE)**

# Conclusions

- Pure G/P algorithm (column-column) is not always best for circuit matrix
- We propose two techniques to improve performance of G/P algorithm
  - Map algorithm for very sparse matrix
  - Supernode algorithm for dense matrix
- Selection of best algorithm based on sparsity

# Our Solver

- **NICSLU: a parallel sparse direct solver for circuit simulation**
  - **nics.ee.tsinghua.edu.cn/people/chenxm/nicslu.htm**
- **High performance has been proved by several EDA companies**
  - **Synopsys, Cadence, Tanner EDA, etc.**

NICSLU

High-performance Parallel Sparse Solver for Circuit Simulation

## License

NICSLU, Copyright (c) 2011-2016 Tsinghua University. All Rights Reserved.

**For non-commercial purpose:**

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

**For commercial purpose:**

Please contact the authors for commercial licenses.

# Thanks for your attention!

# Q & A