Machine Learning and GPU Accelerated Sparse Linear Solvers for Transistor-Level Circuit Simulation: A Perspective Survey

ASP-DAC 2024

Zhou Jin, Wenhao Li, Yinuo Bai, Tengcheng Wang, Yicheng Lu, Weifeng Liu

Super Scientific Software Laboratory,

China University of Petroleum-Beijing, China

Email: jinzhou@cup.edu.cn

Contents

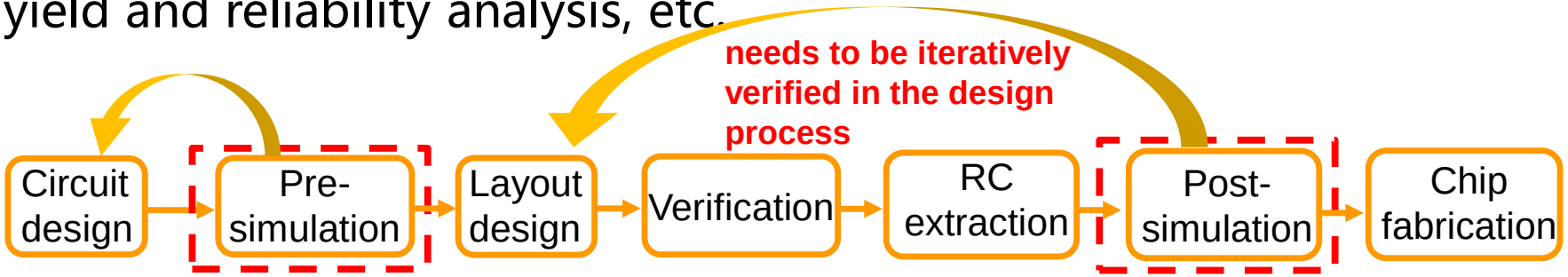
- Background
- Sparse Linear Solvers for Circuit Simulation
  - SFLU: Synchronization-Free Sparse LU Factorization for Fast Circuit Simulation on GPUs
  - Accelerating Sparse LU Factorization with Density-Aware Adaptive Matrix Multiplication for Circuit Simulation
  - PanguLU: A Scalable Regular Two-Dimensional Block-Cyclic Sparse Direct Solver on Distributed Heterogeneous Systems
- Conclusions

Contents

- **Background**
- Sparse Linear Solvers for Circuit Simulation
  - SFLU: Synchronization-Free Sparse LU Factorization for Fast Circuit Simulation on GPUs
  - Accelerating Sparse LU Factorization with Density-Aware Adaptive Matrix Multiplication for Circuit Simulation
  - PanguLU: A Scalable Regular Two-Dimensional Block-Cyclic Sparse Direct Solver on Distributed Heterogeneous Systems
- Conclusions

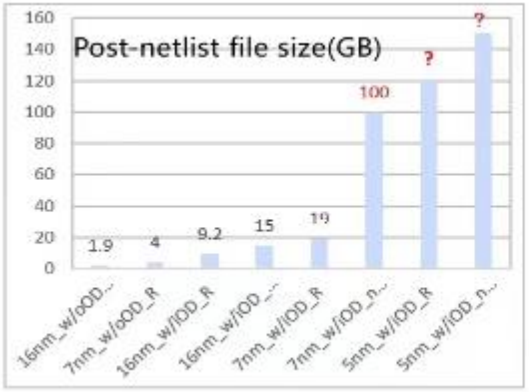
Transistor-Level Circuit Simulation

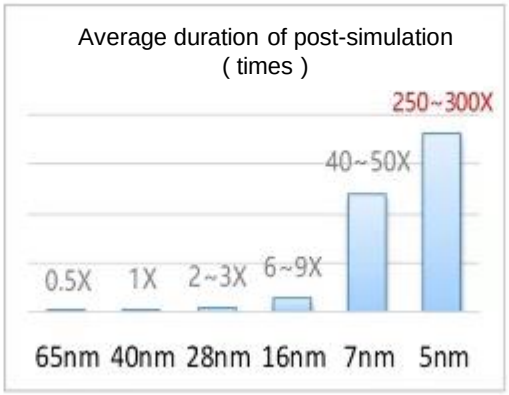
- **Transistor-level circuit simulation (SPICE simulation)** plays a crucial role in verifying circuit performance, and serving as the basis of timing, yield and reliability analysis, etc.



Design process of analog circuits

- **Challenges in SPICE simulation: the continuous expansion of scale, making it extremely time-consuming during the IC design process.**



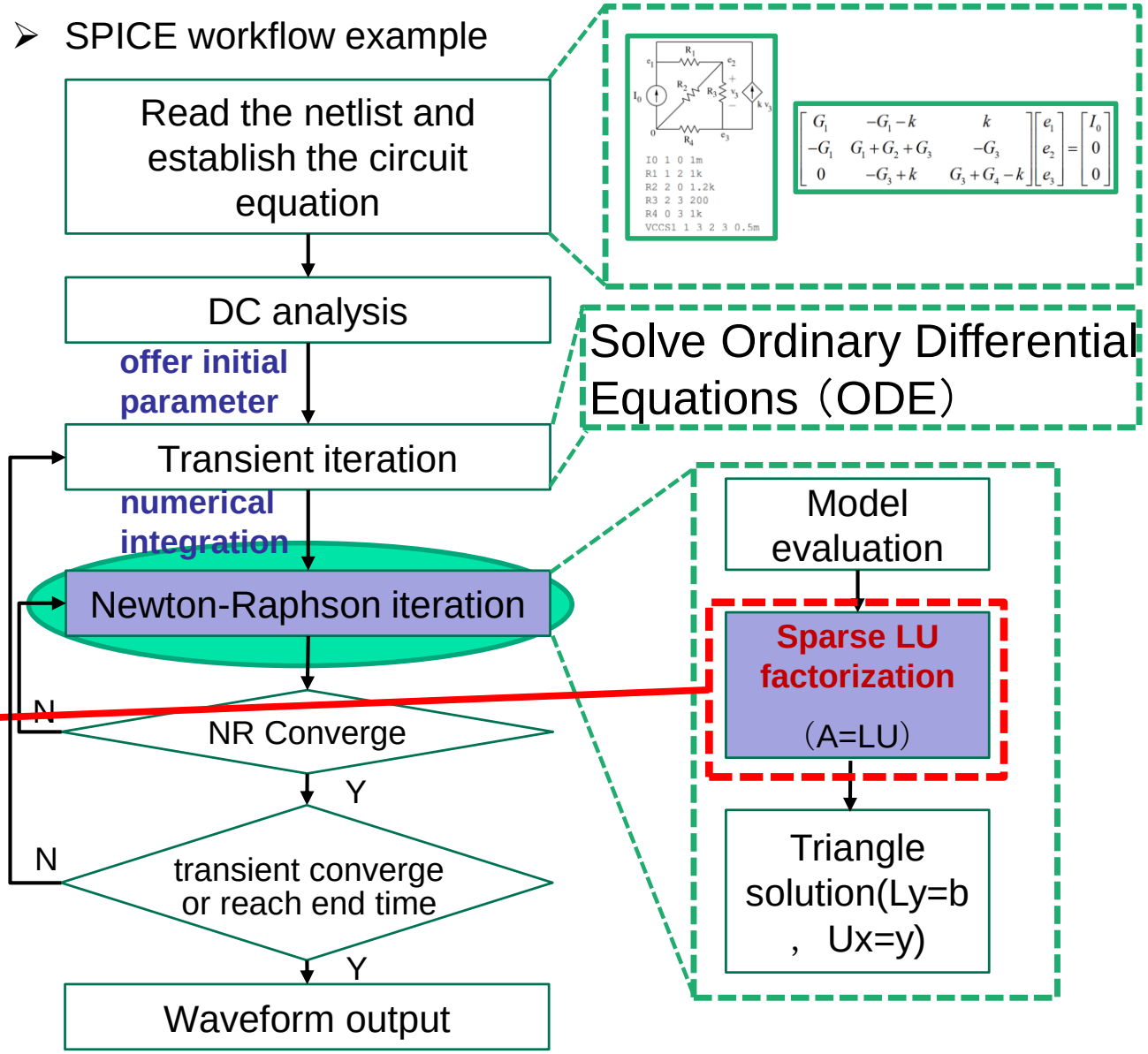


Field	Technology	Maximum Simulation Case	Simulation Time
serdes	7nm	Random Jitter /Top-level MC post-simulation	40~50 days
HBY PHY	7nm	Sub-top post-simulation	40~50 days
AFE	7nm	RXADC post-simulation	40~50 days
AFE	16nm	adc top post-simulation	About 20 days

Transistor-Level Circuit Simulation

- The **most time-consuming step** in SPICE simulation is solving linear equations Ax = b.
- In **post-layout simulation** that takes parasitic effects into account, the linear direct solution typically takes **60-90%** of the time.

- SPICE workflow example



LU Factorization

LU factorization is part of the direct method to solve the linear equations and is also one of the most important tasks in many scientific computing applications. The following figure shows an example of LU factorization:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}$$

The elements of *L* and *U* are calculated from the following two equations:

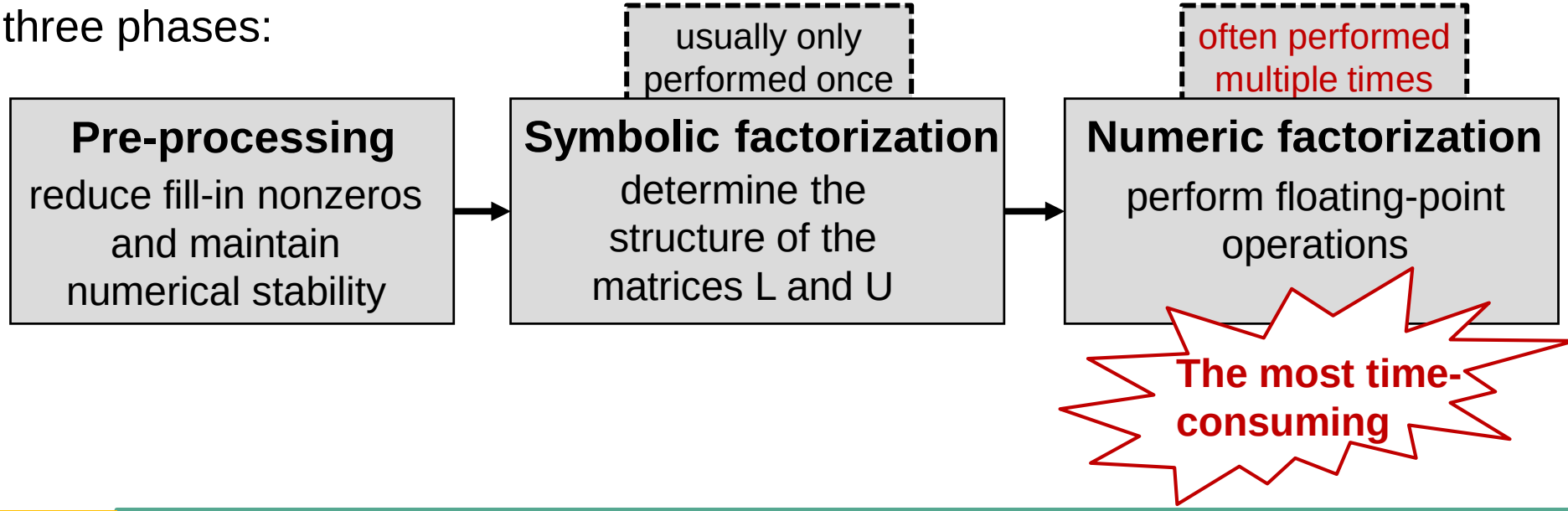
$$u_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik}\, u_{kj} \quad \begin{cases} i = 1, 2, \ldots, N \\ j = i, i+1, \ldots, N \end{cases} \qquad l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}\, u_{kj} \right) \begin{cases} i = 1, 2, \ldots, N \\ j = 1, 2, \ldots, i-1 \end{cases}$$

Sparse LU Factorization

When the input matrix is sparse, the sparse LU factorization can be used to reduce unnecessary computation and storage, which is called sparse LU factorization.

$$\begin{bmatrix} a_{11} & & & a_{14} \\ a_{21} & a_{22} & & a_{24} \\ & & a_{33} & \\ a_{41} & a_{42} & & a_{44} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ & & 1 & 0 \\ l_{41} & l_{42} & & 1 \end{bmatrix} \begin{bmatrix} u_{11} & & & u_{14} \\ 0 & u_{22} & & u_{24} \\ 0 & 0 & u_{33} & \\ 0 & 0 & 0 & u_{44} \end{bmatrix}$$

Unlike LU factorization, sparse LU factorization is divided into the following three phases:



Development of Direct Sparse Solver

MUMPS is a distributed direct solver using the multifrontal method. It was developed by **Patrick** and **Duff et al**. (PARA00)

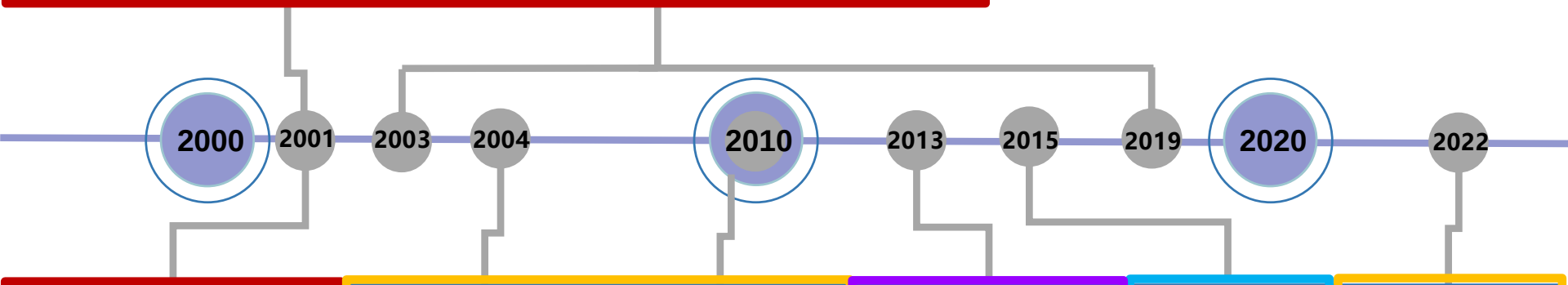
SuperLU using supernodal method is developed by **Li** and **Demmel et al.** SuperLU_DIST is a distributed version. (TOMS03) SuperLU_DIST has been continuously updated and added versions for GPU.(JPDC19)

Only on share memory CPUs

On Single GPU

On share memory CPUs and single GPU

On distributed systems



PARDISO is developed by **Schenk et al**. It uses a combination of left- and right-looking Level-3 BLAS to exploit parallelism. (FGCS01)

Davis developed UMFPACK which combines pre-ordering and multifrontal method**.** (TOMS04)

Davis et al also developed KLU, which is a sparse solver for circuit simulation. (TOMS10)

NICSLU is a sparse solver using level-set and supernode. Developed by **Chen** and **Wang et al**. (TCAD13)

GLU is a sparse direct solver on GPU developed by **Tan et al**. (D&T20)

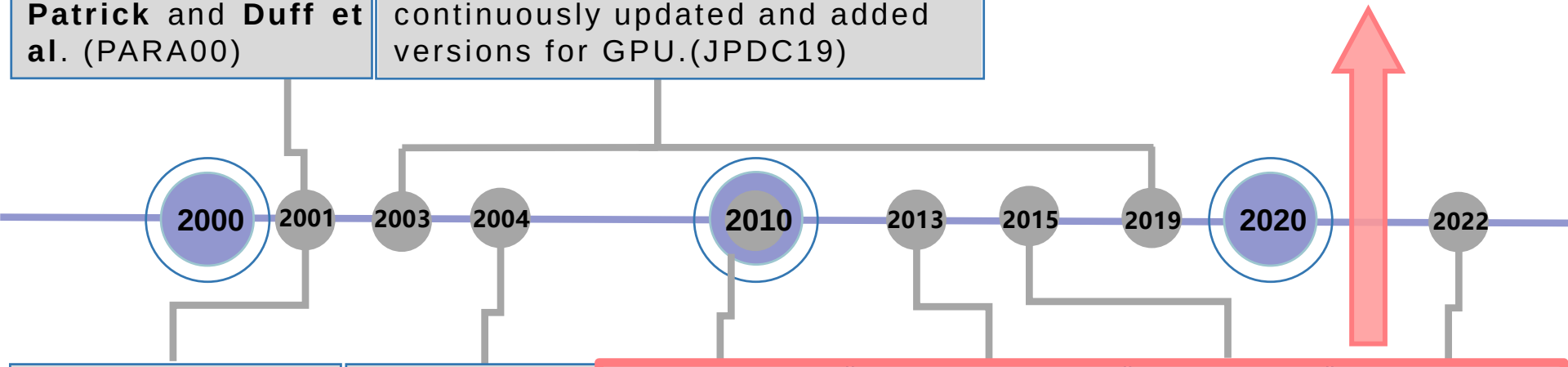
FLU is developed by **Yan et al.** It introduces a register-level method. (DATE22)

Development of Direct Sparse Solver

MUMPS is a distributed direct solver using the multifrontal method. It was developed by **Patrick** and **Duff et al**. (PARA00)

SuperLU using supernodal method is developed by **Li** and **Demmel et al.** SuperLU_DIST is a distributed version. (TOMS03) SuperLU_DIST has been continuously updated and added versions for GPU.(JPDC19)

Dedicated to circuit matrix



PARDISO is developed by **Schenk et al**. It uses a combination of left- and right-looking Level-3 BLAS to exploit parallelism. (FGCS01)

Davis developed UMFPACK which combines pre-ordering and multifrontal method**.** (TOMS04)

Davis et al also developed KLU, which is a sparse solver for circuit simulation. (TOMS10)

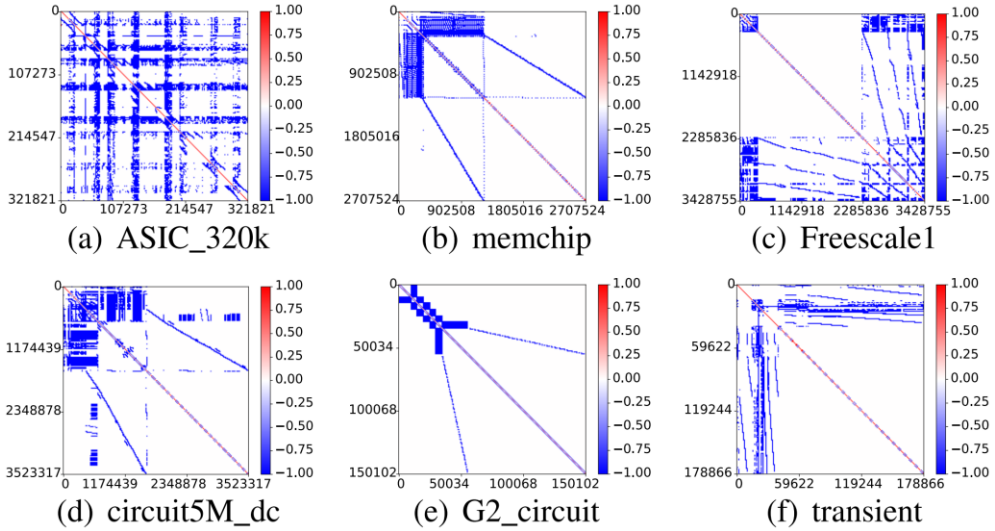
NICSLU is a sparse solver using level-set and supernode. Developed by **Chen** and **Wang et al**. (TCAD13)

GLU is a sparse direct solver on GPU developed by **Tan et al**. (D&T20)

FLU is developed by **Yan et al.** It introduces a register-level method. (DATE22)

Circuit Matrix Properties

The structure of some circuit sparse matrices.



In circuit simulation, the matrix usually has the following properties:

Circuit sparse matrix properties.

Circuit Matrix	N	Entries per row				Symmetry
		max	min	average	variation	
ASIC_320k	321,821	203,800	1	8.2	502.95	100.00%
memchip	2,707,524	27	2	5.5	2.06	0.32%
Freescale1	3,428,755	27	1	5.5	2.07	7.67%
circuit5M	3,523,317	27	1	10.7	1356.61	55.99%
G2_circuit	150,102	4	1	2.9	0.52	0.0005%
transient	178,866	60423	1	5.4	147.2	68.99%

Development of Direct Sparse Solver

On single GPU

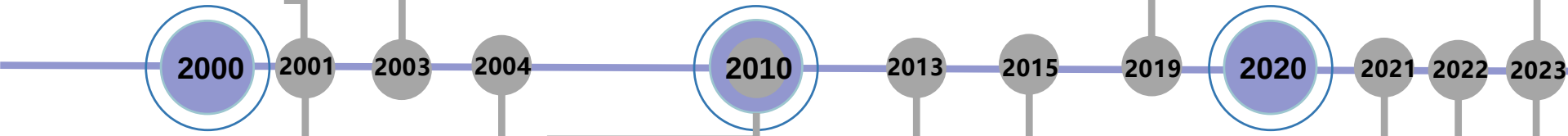
On CPU with AI assisted

On distributed heterogeneous system

MUMPS is a distributed direct solver using the multifrontal method. It was developed by **Patrick and Buffet et al.** (PARA00)

SuperLU using supernodal method is developed by **Li** and **Demmel et al.** SuperLU_DIST is a distributed version. (TOMS03) SuperLU_DIST has been continuously updated and added versions for GPU.(JPDC19)

PanguLU is developed by **SSSLab**. It uses strategies such as regular two-dimensional blocking, synchronization-free and sparse kernels on distributed heterogeneous platform. (SC23 Best paper)



PARDISO is developed by **Schenk et al.** It uses a combination of left- and right-looking Level-3 BLAS to exploit parallelism. (FGCS01)

Davis developed UMFPACK which combines pre-ordering and multifrontal method. (TOMS04)

Davis et al also developed KLU, which is a sparse solver for circuit simulation. (TOMS10)

NICSLU is a sparse solver using level-set and supernode. Developed by **Chen** and **Wang et al.** (TCAD13)

GLU is a sparse direct solver on GPU developed by **Tan et al.** (D&T20)

SFLU is developed by **SSSLab**. It uses synchronization-free strategy on GPU. (DAC21)

FLU is developed by **Yan et al.** It introduces a register-level method. (DATE22)

DALU is developed by **SSSLab**. It proposed a density-aware adaptive method. (DAC23)

Contents

- Background
- **Sparse Linear Solvers for Circuit Simulation**
  - SFLU: Synchronization-Free Sparse LU Factorization for Fast Circuit Simulation on GPUs
  - Accelerating Sparse LU Factorization with Density-Aware Adaptive Matrix Multiplication for Circuit Simulation
  - PanguLU: A Scalable Regular Two-Dimensional Block-Cyclic Sparse Direct Solver on Distributed Heterogeneous Systems
- Conclusions

SFLU: Synchronization-Free Sparse LU Factorization on GPUs (DAC21)

Using GPU for general computing has become a research hotspot.

TABLE I: There are two ways to utilize GPUs for sparse LU.

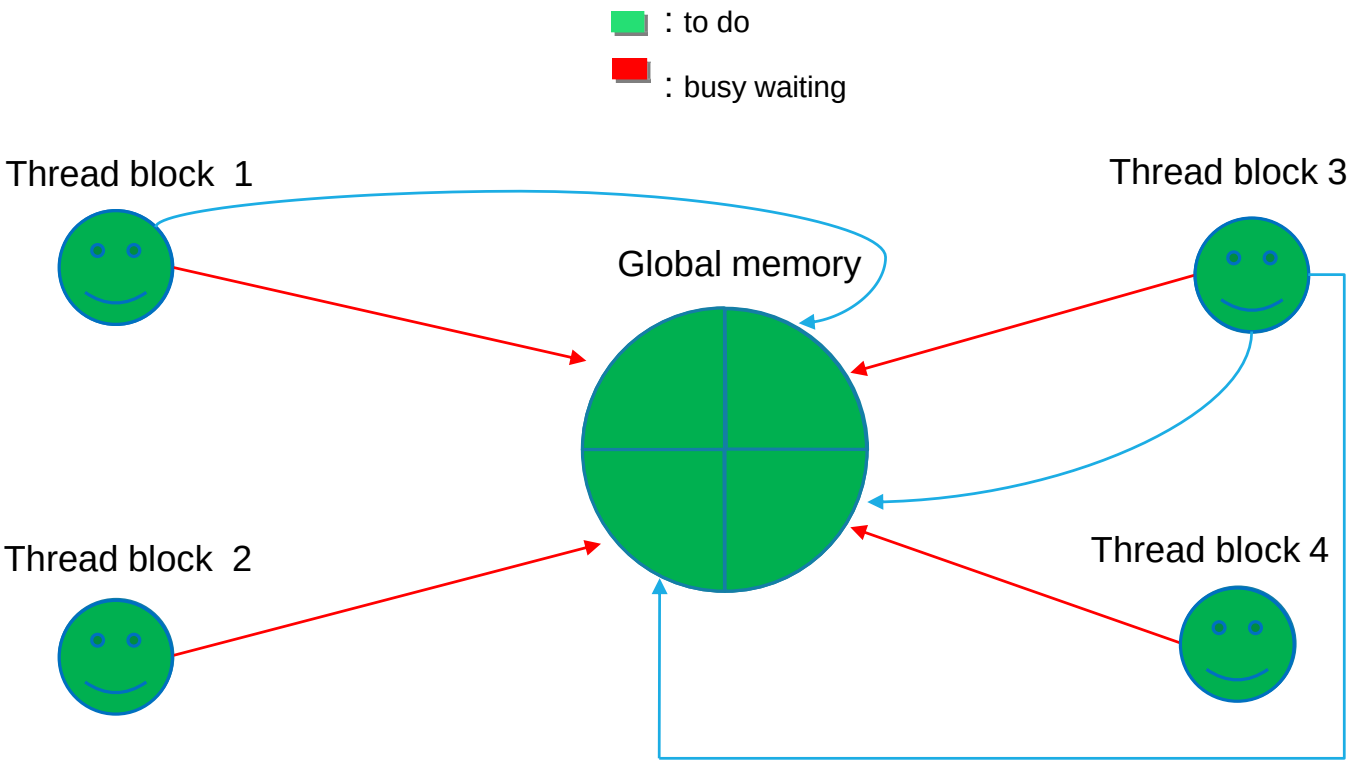
Name	Strategy	Characteristic
SuperLU[1]	Running GPU-version dense matrix-matrix multiplication on (often small) matrices generated from supernodal pattern.	Can not use GPU for scheduling to save the cost for kernel relaunching and cache data flushing.
GLU[2]	Finding dependencies between columns, generating levels, and using level scheduling method for running multiple columns in the same level on GPUs concurrently.	
?	?	Can use GPU for scheduling.

We need to design a method that can use GPU for scheduling.

[1] Demmel, James W , Eisenstat, et al, A supernodal approach to sparse partial pivoting. SIAM Journal on Matrix Analysis & Applications, 1999.

[2] Wai-Kong L , Ramachandra A , Nakhla M S, Dynamic GPU Parallel Sparse LU Factorization for Fast Circuit Simulation, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2018

SFLU: Synchronization-Free Sparse LU Factorization on GPUs (DAC21)

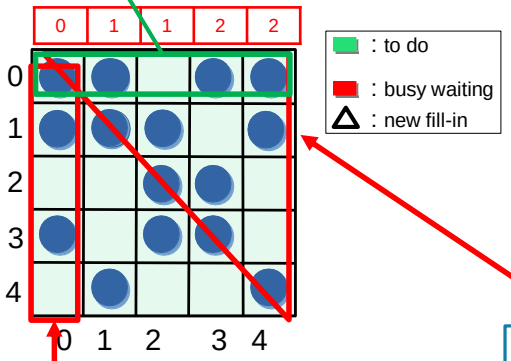


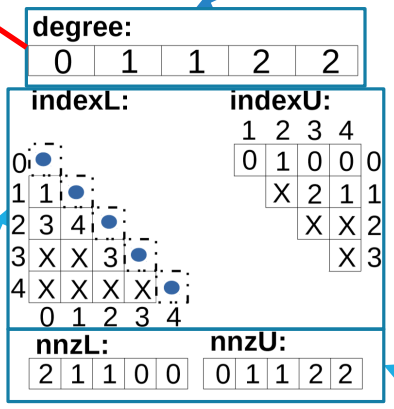
Due to the logic between symbolic and numeric factorization are similar. we only use the symbolic factorization to introduce our algorithm.

Specifically:

Elements above the diagonal line of the row k can be eliminated.

Degree represents the number of uneliminated elements in each column.





All the elements above the diagonal of column k are eliminated.

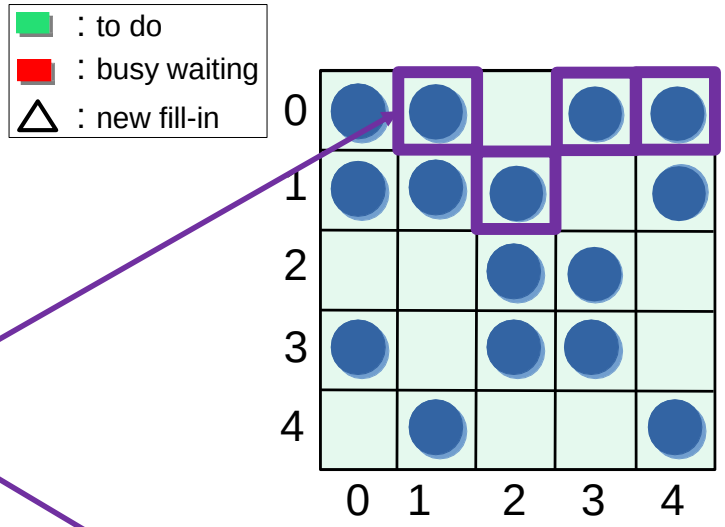
The index represents the row index of each element.

NnzL and nnzU represents the number of elements in each column of L and U.

SFLU: Synchronization-Free Sparse LU Factorization on GPUs (DAC21)

Algorithm 1 Symbolic factorization of SFLU

```
1: /*All columns are executed in parallel*/
2: for  all columns in parallel do
3:     /*Set the column be k*/
4:     for  i = 0 to k where the A[i,k] != 0 do
5:         /*Eliminate elements above the diagonal*/
6:         while degree[i] != 0 do
7:             //Busy wait
8:         end while
9:         Eliminate A[i,k] by using ith column
10:        /*After elimination, update the value of degree*/
11:        atomicSubtract(degree[k], 1)
12:    end for
13: end for
```



Three warps are working

degree:

0	1	1	2	2

indexL:

	0	1	2	3	4
0	●				
1	1	●			
2	3	4	●		
3	X	X	3	●	
4	X	X	X	X	●

indexU:

	1	2	3	4	
0	1	0	0	0	0
	X	2	1	1	1
		X	X	2	2
			X	3	3

nnzL:

2	1	1	0	0

nnzU:

0	1	1	2	2

We use the storage order of indexU to determine the decomposition object of each thread block.

Each unfinished thread block should have an object.

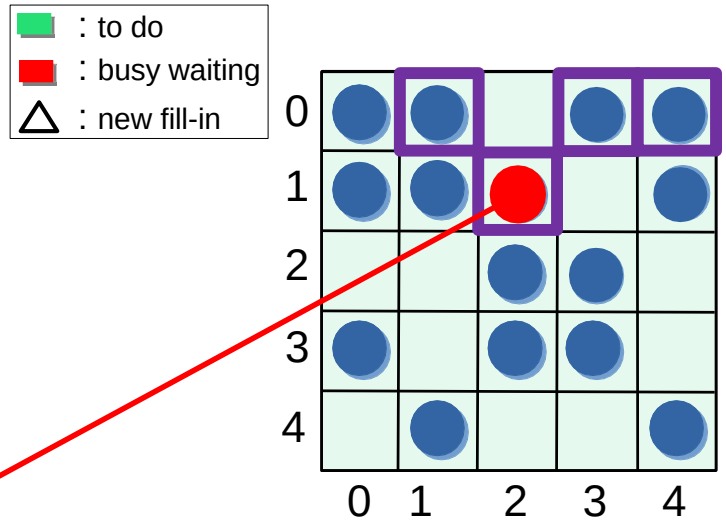
Four objects.

SFLU: Synchronization-Free Sparse LU Factorization on GPUs (DAC21)

Algorithm 1 Symbolic factorization of SFLU

```
1: /*All columns are executed in parallel*/
2: for all columns in parallel do
3:     /*Set the column be k*/
4:     for i = 0 to k where the A[i,k] != 0 do
5:         /*Eliminate elements above the diagonal*/
6:         while degree[i] != 0 do
7:             //Busy wait
8:         end while
9:         Eliminate A[i,k] by using ith column
10:        /*After elimination, update the value of degree*/
11:        atomicSubtract(degree[k], 1)
12:    end for
13: end for
```

One element will be busy waiting.



Three warps are working

degree:

0	1	1	2	2

indexL:

	0	1	2	3	4
0	●				
1	1	●			
2	3	4	●		
3	X	X	3	●	
4	X	X	X	X	●

indexU:

	1	2	3	4	
0	1	0	0	0	0
	X	2	1	1	1
		X	X	2	2
			X	3	3

nnzL:

2	1	1	0	0

nnzU:

0	1	1	2	2

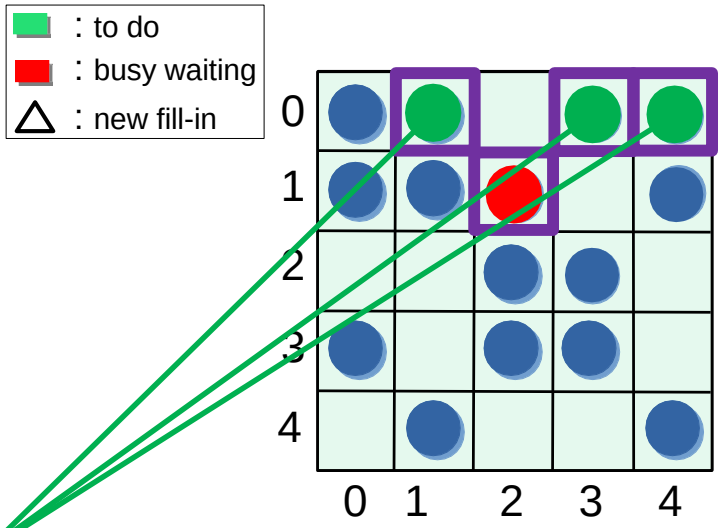
SFLU: Synchronization-Free Sparse LU Factorization on GPUs (DAC21)

Algorithm 1 Symbolic factorization of SFLU

```
1: /*All columns are executed in parallel*/
2: for all columns in parallel do
3:     /*Set the column be k*/
4:     for i = 0 to k where the A[i,k] != 0 do
5:         /*Eliminate elements above the diagonal*/
6:         while degree[i] != 0 do
7:             //Busy wait
8:         end while
9:         Eliminate A[i,k] by using ith column
10:        /*After elimination, update the value of degree*/
11:        atomicSubtract(degree[k], 1)
12:    end for
13: end for
```

One element will be busy waiting.

Three elements will be eliminated.



Three warps are working

degree:

0	1	1	2	2

indexL:

	0	1	2	3	4
0	●				
1	1	●			
2	3	4	●		
3	X	X	3	●	
4	X	X	X	X	●

indexU:

	1	2	3	4	
0	1	0	0	0	0
	X	2	1	1	1
		X	X	2	2
			X	3	3

nnzL:

2	1	1	0	0

nnzU:

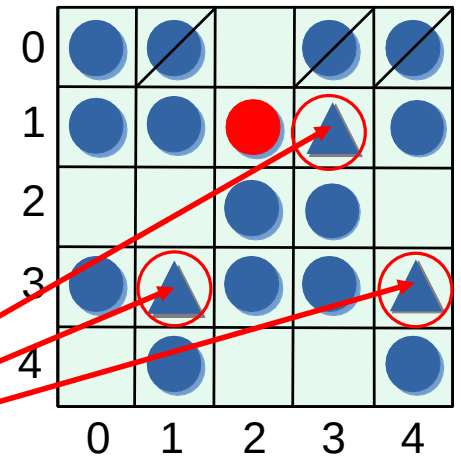
0	1	1	2	2

SFLU: Synchronization-Free Sparse LU Factorization on GPUs (DAC21)

Algorithm 1 Symbolic factorization of SFLU

```
1: /*All columns are executed in parallel*/
2: for all columns in parallel do
3:     /*Set the column be k*/
4:     for i = 0 to k where the A[i,k] != 0 do
5:         /*Eliminate elements above the diagonal*/
6:         while degree[i] != 0 do
7:             //Busy wait
8:         end while
9:         Eliminate A[i,k] by using ith column
10:        /*After elimination, update the value of degree*/
11:        atomicSubtract(degree[k], 1)
12:    end for
13: end for
```

: to do
: busy waiting
: new fill-in



degree:

0	0	1	2	2

indexL:

	0	1	2	3	4
0	●				
1	1	●			
2	3	4	●		
3	X	3	3	●	
4	X	X	X	X	●

indexU:

	1	2	3	4	
	~~0~~	1	~~0~~	~~0~~	0
		X	2	1	1
			1	3	2
				X	3

nnzL:

2	2	1	0	0

nnzU:

0	1	1	3	3

Three nonzero elements were filled in.

Degree Changed.

Experimental Result

We test 1309 matrices from the SuiteSparse matrix set on Titan RTX (Turing) GPUs, Intel 20-core CPUs, comparing the performance of SFLU with SuperLU and GLU.

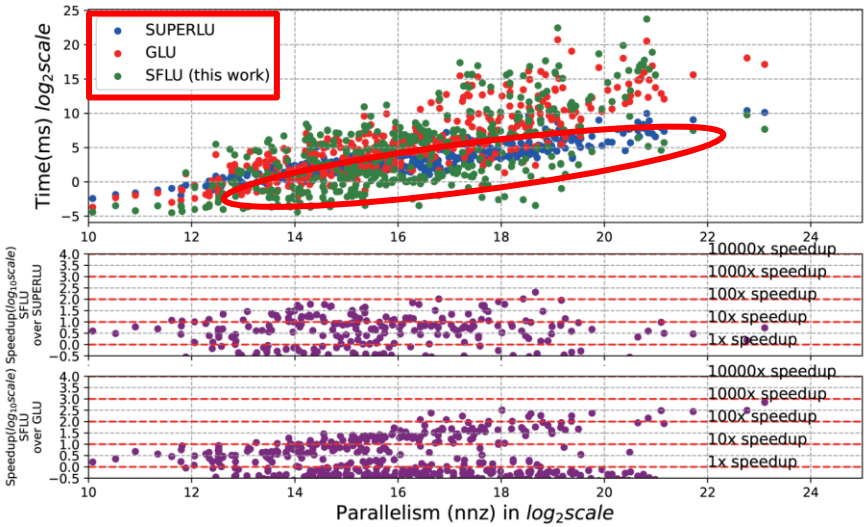


Figure 1. Symbolic factorization comparison.

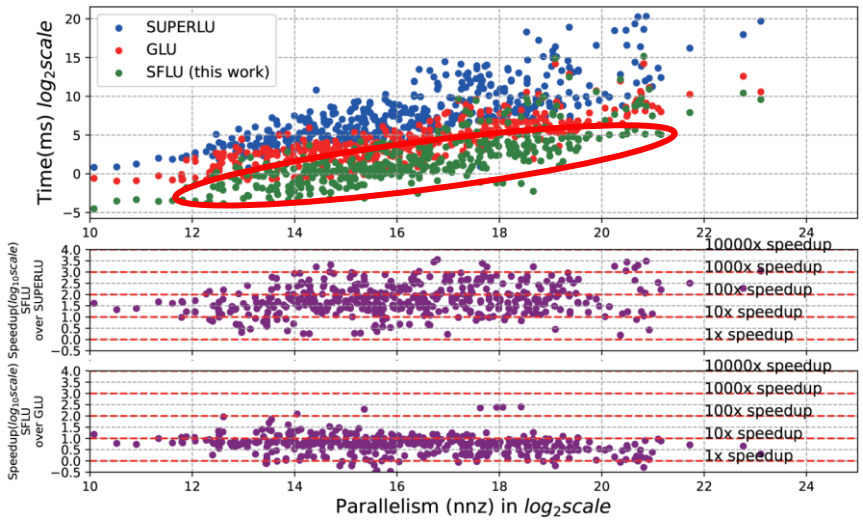


Figure 2. Numeric factorization comparison.

In the symbolic factorization, the acceleration is up to 205.14x and 701x compared to SuperLU and GLU, respectively.

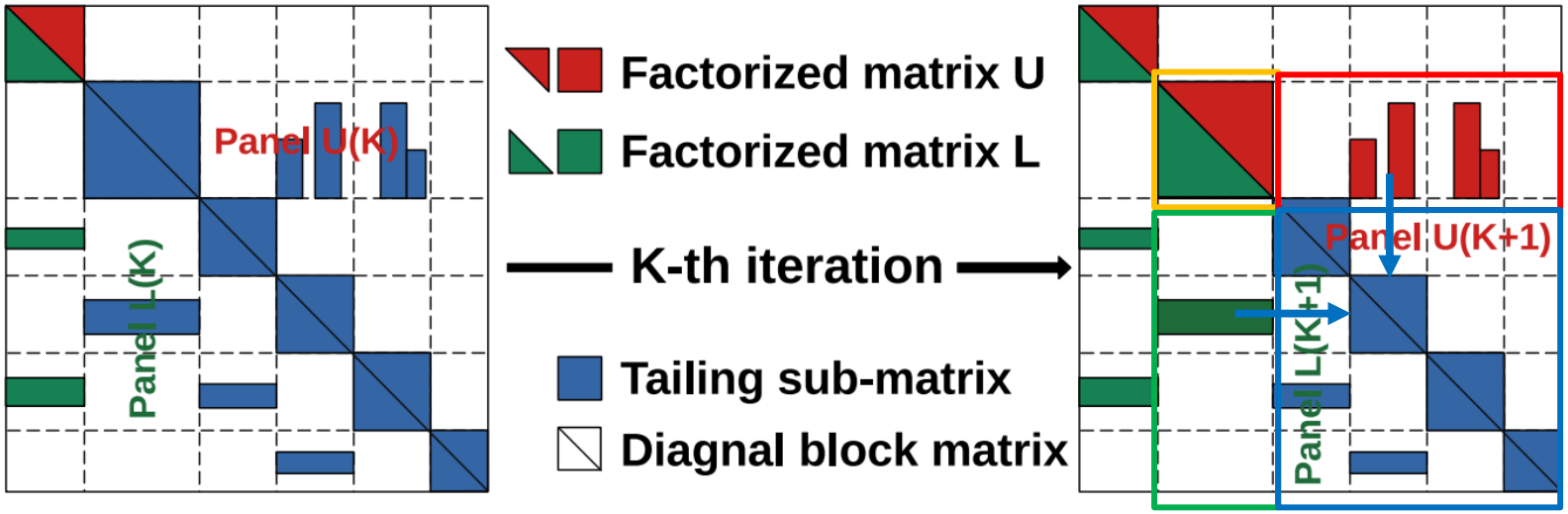
In the numeric factorization, the acceleration is up to 3585.2x and 252.2x compared to SuperLU and GLU, respectively.

Contents

- Background
- **Sparse Linear Solvers for Circuit Simulation**
  - SFLU: Synchronization-Free Sparse LU Factorization for Fast Circuit Simulation on GPUs
  - Accelerating Sparse LU Factorization with Density-Aware Adaptive Matrix Multiplication for Circuit Simulation
  - PanguLU: A Scalable Regular Two-Dimensional Block-Cyclic Sparse Direct Solver on Distributed Heterogeneous Systems
- Conclusions

Sparse LU with Density-Aware Adaptive Matrix Multiplication (DAC23)

The numeric factorization in supernodal LU factorization follows four steps, where K signifies the K-th iteration and N denotes the number of matrix blocks on the diagonal.

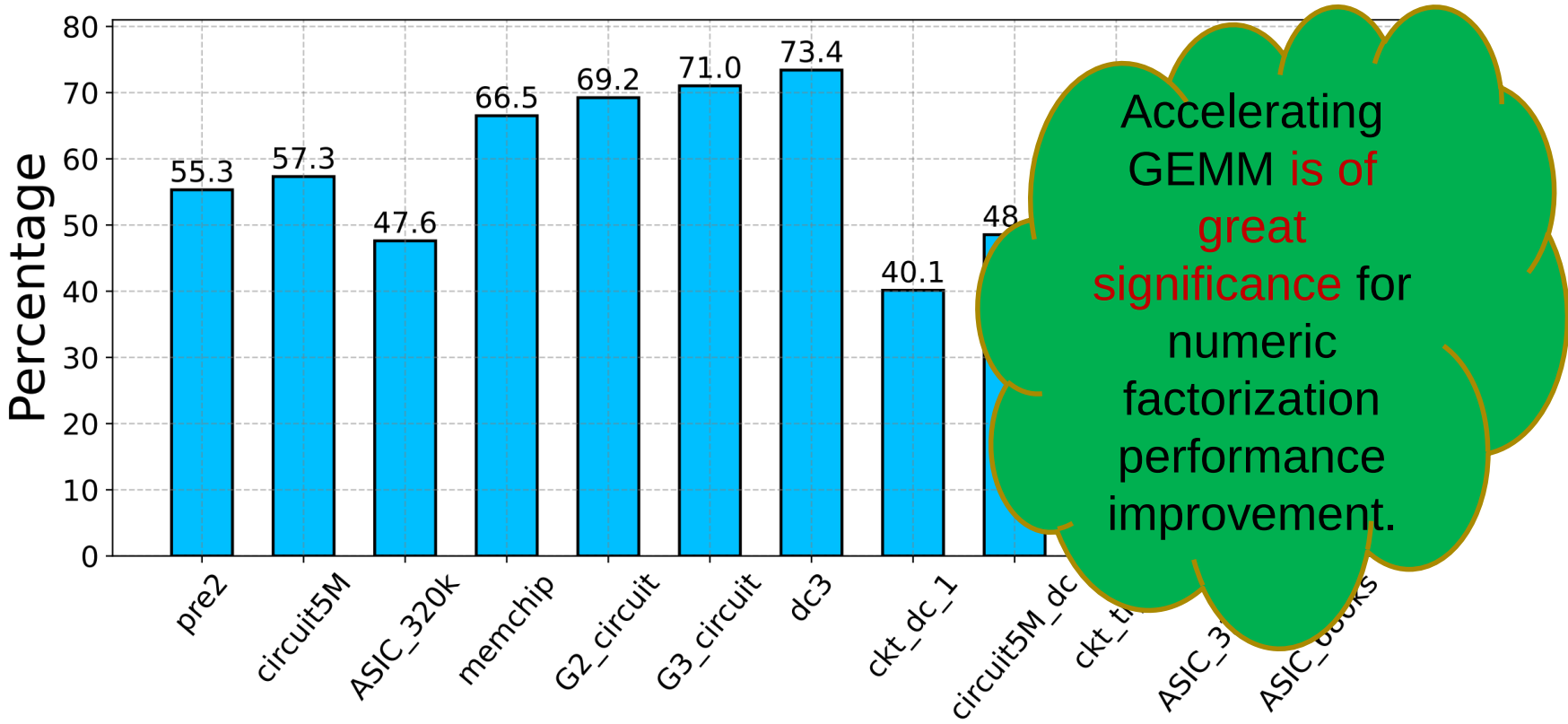


The Schur-complement phase contains a large number of GEMM operations.

- Factorize the diagonal block.
- Factorize the sub-matrices in L panel: L(K : N, K)
- Factorize the sub-matrices in U panel: U(K, K +1 : N)
- Perform the Schur-complement for all the tailing sub-matrices by using $A = A - L \times U$.

Sparse LU with Density-Aware Adaptive Matrix Multiplication (DAC23)

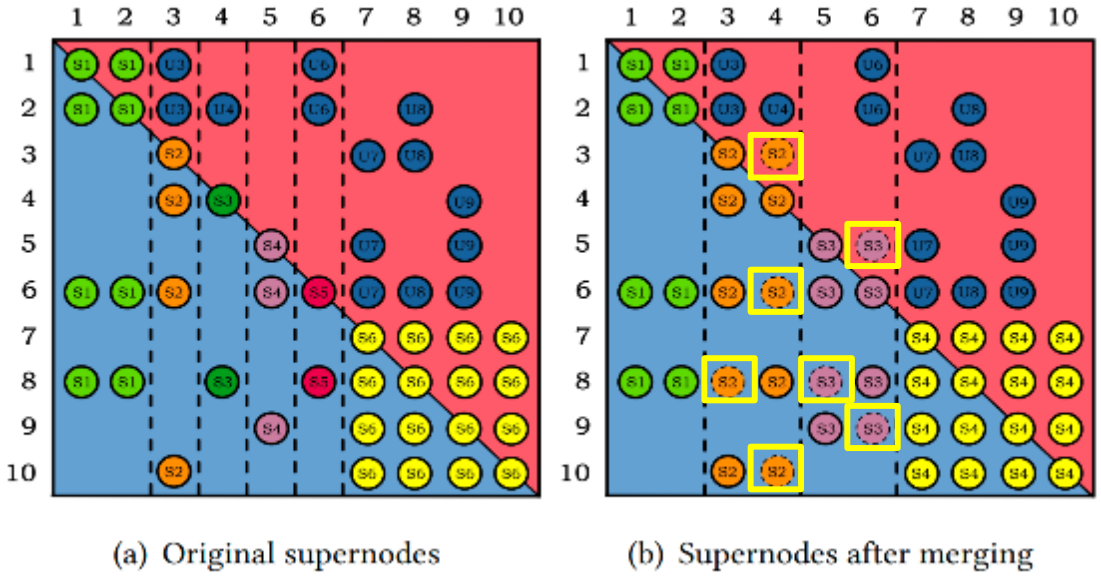
GEMM takes up much of the time for numeric factoriztaion.



We tested some circuit matrices, such as the G3_circuit matrix in the figure, which accounts for as much as 73.4% of the time, and most of the other matrices tend to account for 40%-60% of the GEMM time.

Sparse LU with Density-Aware Adaptive Matrix Multiplication (DAC23)

In order to better **utilize the dense GEMM of Level-3 BLAS**, the traditional approach is to form dense blocks using supernodal (shown in Figure 1) or multifrontal methods. However, for matrices without a regular sparse structure, **supernodes** formed through relaxation **may not always be dense** (shown in Figure 2).



(a) Original supernodes (b) Supernodes after merging

Figure 1. How to form supernodes.

Schur-complement

$$A(I,J) \leftarrow A(I,J) - L(I,K) \times U(K,J);$$

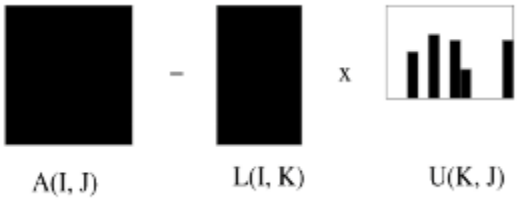


Figure 2. Example of Schur-complement computation.

Sparse LU with Density-Aware Adaptive Matrix Multiplication (DAC23)

We select three representative circuit matrices and can see that the input matrix (L and U blocks) involved in GEMM are not always dense.

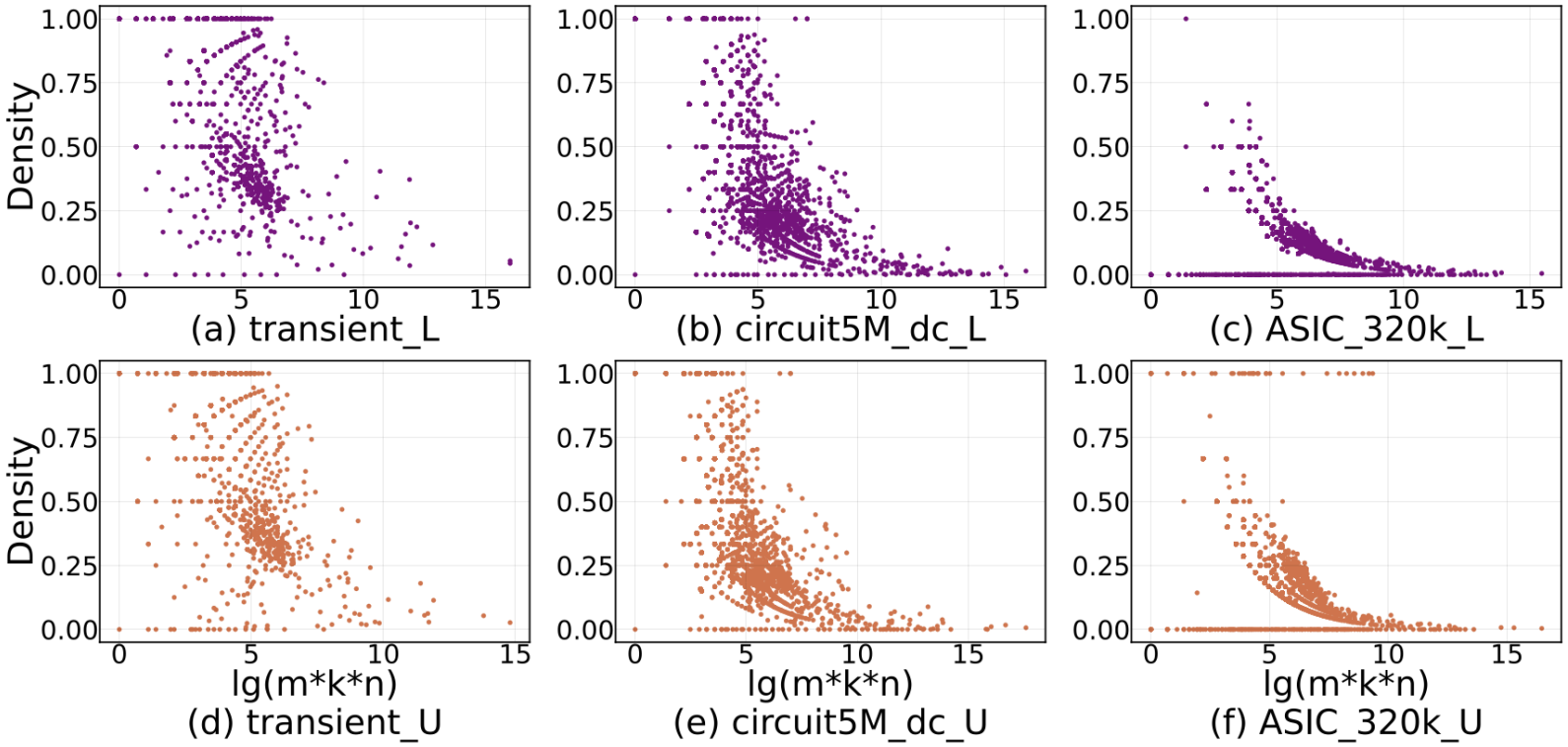


Figure 1. Density distribution of matrix factors (L and U blocks) participating in GEMM.

The introduction of sparse matrix multiplication (SpMM) shows great potential for further accelerating the LU factorization.

Sparse LU with Density-Aware Adaptive Matrix Multiplication (DAC23)

After introducing SpMM, we test the data in Table 1. The "Oracle" in the table indicates the optimal total time computed by picking the optimal kernel in SpMM and GEMM for each matrix multiplication in the ideal case.

TABLE I: Analysis of the performance improvement space of matrix multiplication.

Circuit matrix	nnz (A)	GEMM (s)	SpMM (s)	Oracle (s)	Speedup1	Speedup2
ASIC_320k	1,931,828	3.5809	0.4543	0.3653	9.80x	1.24x
Freescale1	17,052,626	8.8363	35.9429	8.5388	1.03x	4.21x
ckt11752_dc_1	333,029	0.0279	0.0433	0.0225	1.24x	1.92x
pre2	5,834,044	57.2954	10.9046	7.2531	7.90x	1.50x
meg4	58,142	0.0037	0.0027	0.0022	1.68x	1.23x
G2_circuit	726,674			415	1.16x	**4.25x**
Freescale2	14,313,				1.19x	1.95x
FullChip	26,6				1.48x	1.40x
ASIC_320ks	1,				**10.24x**	1.13x
ASIC_680ks	1,6				2.81x	1.44x
circuit5M_dc	14,8			0230	6.04x	1.21x
transient	961,368			0.2100	2.69x	1.29x

Which method is best? GEMM? SpMM?

Compared with GEMM and SpMM, the performance improvement potential of "Oracle" is 1.03x-10.24x and 1.13x-4.25x, respectively.

We need to pick the optimal kernel for each execution of matrix multiplication!

Sparse LU with Density-Aware Adaptive Matrix Multiplication (DAC23)

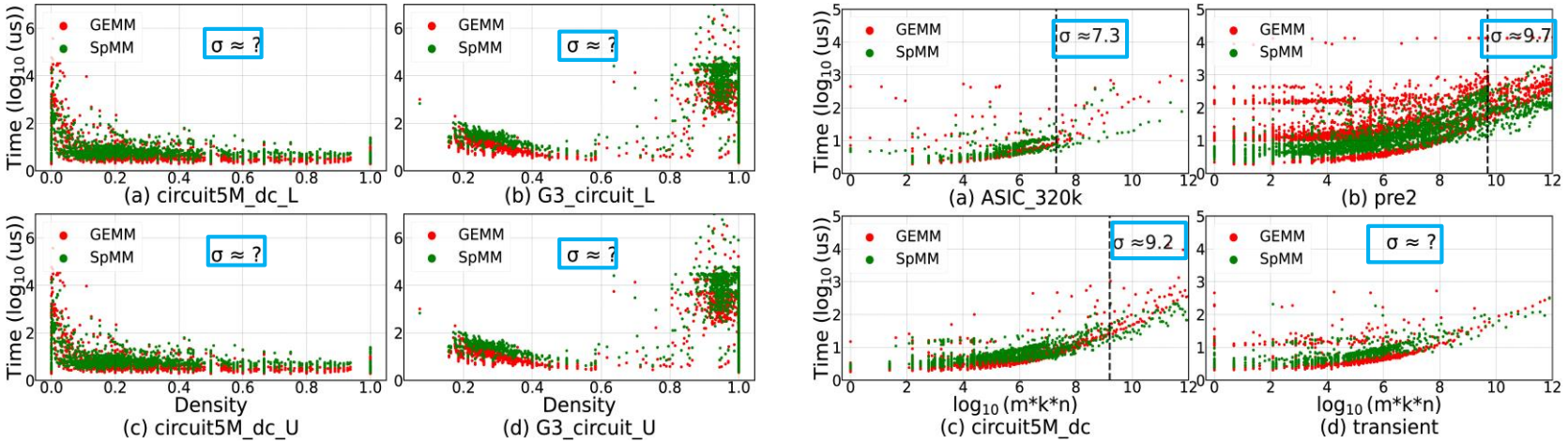


Figure 1. Performance comparison based on density and matrix size.

It is difficult to define thresholds for either density or matrix size. The use of **density-aware adaptive** matrix multiplication equipped with random forest[1] provides an effective solution to the issues mentioned above.

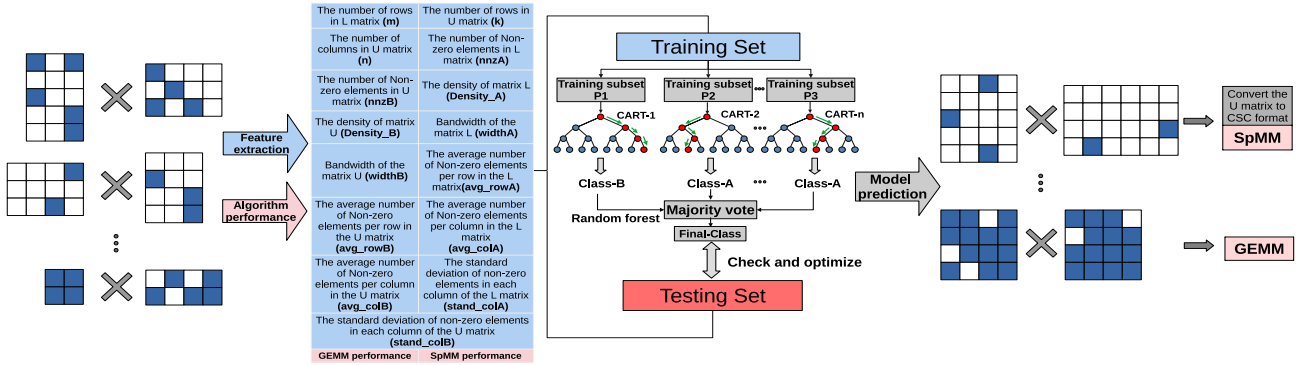
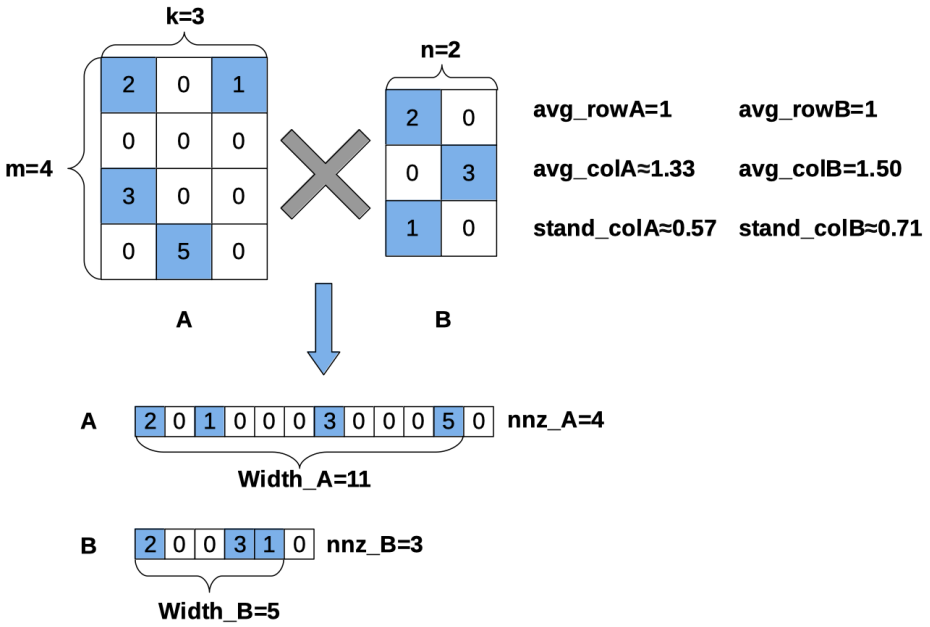


Figure 2. Density-aware LU factorization.

Experimental Result

- **Dataset Source:** The large-scale circuit matrices in SuiteSparse and exported by SuperLU_DIST 8.0.0.
- **Samples in the dataset:** Each sample contains **15 matrix features** (F1-F15) and a label P, where P=1 indicates that using GEMM is better than that of SpMM, and P=0 is vice versa.
- **Data preprocessing:**
  - **Z-score normalization**
  - **Sample equalization**
- **Number of samples in the dataset :**

Dataset	GEMM	SpMM
Total sample set(D1)	456,000	383,000
Training set(D2)	150,000	150,000
Testing set(D3)	306000	233000



An example of 15 matrix features (F1-F15).

Experimental Result

We test several matrix in a 2 * Intel Xeon Silver 4210 CPU @ 2.20GHz, 512GB DDR4 platform configuration. We test 11 test matrix from the SuiteSparse, including 6 circuit matrix and 5 non-circuit matrix.

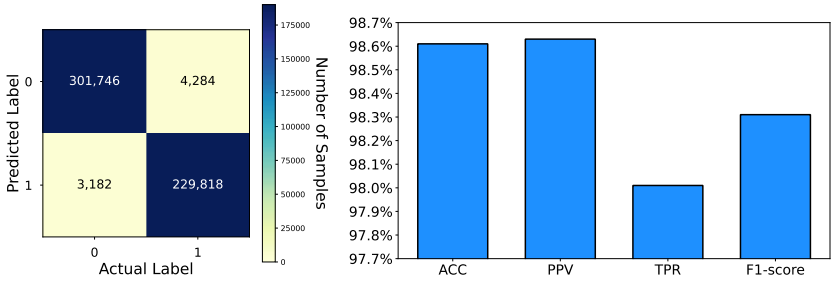


Figure 1. Model accuracy

For circuit matrix, up to **9.35x** acceleration for matrix mutiplication and **1.76x** for LU factorization;

For other irregular matrix, up to **4.32x** and **2.08x**, respectively.

Performance evaluation on circuit matrices.

circuit matrix	nnz	Matrix multiplication time (s)						Numeric factorization time (s)		
		GEMM	SpMM	Oracle	AI	Speedup (AI vs GEMM)	Speedup (AI vs SpMM)	SuperLU	Our work	Speedup
ASIC_320k	1,931,828	3.5809	0.4543	0.3653	0.5045	7.10x	0.90x	7.5201	4.5770	1.64x
ASIC_320ks	1,316,085	2.9155	0.3223	0.2846	0.3117	**9.35x**	1.03x	6.0602	3.5947	1.69x
ASIC_680ks	1,693,767	2.6196	1.3393	0.9323	1.0085	2.60x	1.33x	5.3502	3.8139	1.40x
circuit5M_dc	14,865,409	7.5022	1.2418	1.0231	2.0084	6.04x	0.62x	19.2301	14.5605	1.32x
pre2	5,834,044	57.2954	10.9046	7.2531	8.1021	7.07x	**1.35x**	103.5721	58.7290	**1.76x**
transient	961,368	0.5721	0.2709	0.2121	0.2532	2.26x	1.07x	1.7202	1.4517	1.18x
Average	-	-	-	-	-	5.35x	1.05x	-	-	1.50x

Performance evaluation on non-circuit matrices.

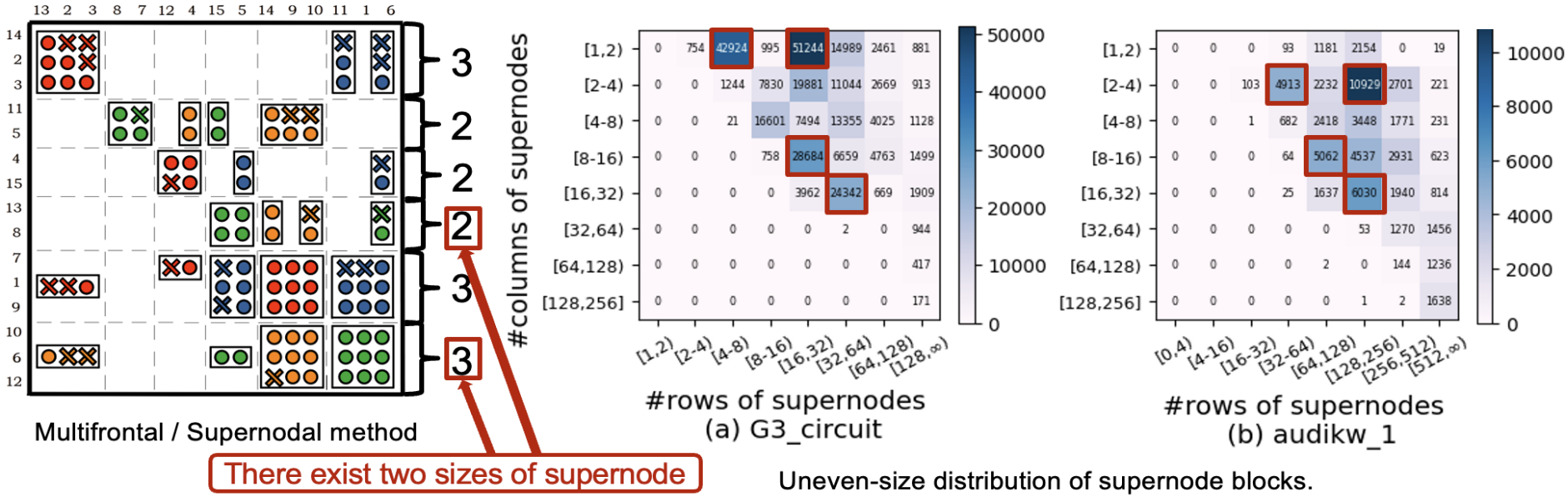
non-circuit matrix	nnz	Matrix multiplication time (s)						Numeric factorization time (s)		
		GEMM	SpMM	Oracle	AI	Speedup (AI vs GEMM)	Speedup (AI vs SpMM)	SuperLU	Our work	Speedup
sinc12	283,992	9.7211	2.7060	2.0120	2.2491	**4.32x**	1.20x	14.7541	7.4277	2.04x
psmigr_3	543,160	16.3840	7.8951	5.8241	6.2611	2.62x	1.26x	28.4921	19.8387	1.54x
psmigr_2	540,022	30.8151	12.0731	8.6251	9.1721	3.36x	1.32x	45.9061	24.5057	**2.08x**
epb2	175,027	0.1291	0.10021	0.05631	0.07621	1.69x	1.31x	0.4351	0.3937	1.08x
benzene	242,669	8.1778	9.6211	6.2315	7.2724	1.13x	**1.32x**	17.6351	16.9551	1.04x
Average	-	-	-	-	-	2.62x	1.28x	-	-	1.55x

Contents

- Background
- **Sparse Linear Solvers for Circuit Simulation**
  - SFLU: Synchronization-Free Sparse LU Factorization for Fast Circuit Simulation on GPUs
  - Accelerating Sparse LU Factorization with Density-Aware Adaptive Matrix Multiplication for Circuit Simulation
  - PanguLU: A Scalable Regular Two-Dimensional Block-Cyclic Sparse Direct Solver on Distributed Heterogeneous Systems
- Conclusions

Motivation 1: Uneven Block Sizes

When the supernodal method aggregates columns of the same structure, there are different sizes of supernodes.



Uneven-size distribution of supernode blocks.

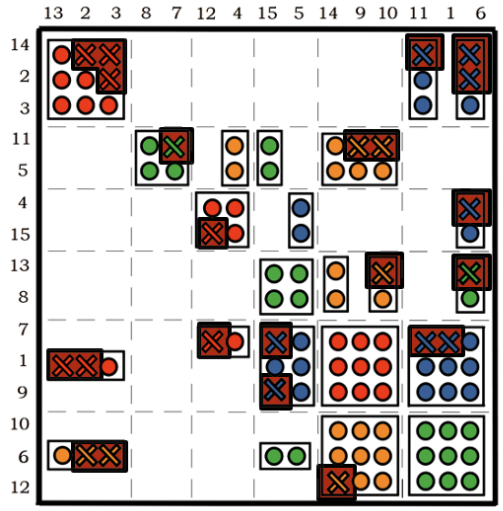
It can be seen that the matrix blocks generated by the supernodes can be very irregular, which often affects the computational efficiency, and their irregular structure makes it difficult to optimise the performance at the kernel level.

Motivation 2: Redundant Zero Fill-ins

The supernodal method divides the matrix into uneven dense blocks to be computed with dense BLAS, which may cause two questions.

1. Redundant zero fill-ins can occur when forming dense blocks that add extra floating-point operations.

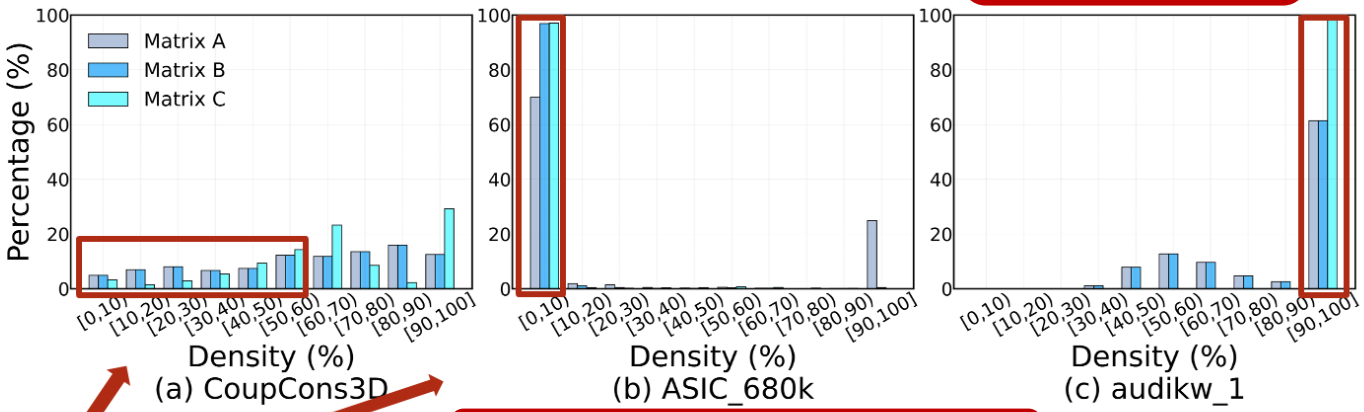
2. The local sparsity of the matrix cannot be exploited during GEMM calculations.



Multifrontal / Supernodal method

This matrix has an overall even distribution but a significant portion is less than 50%

concentrated at [90,100)%



concentrated at [0,10)%

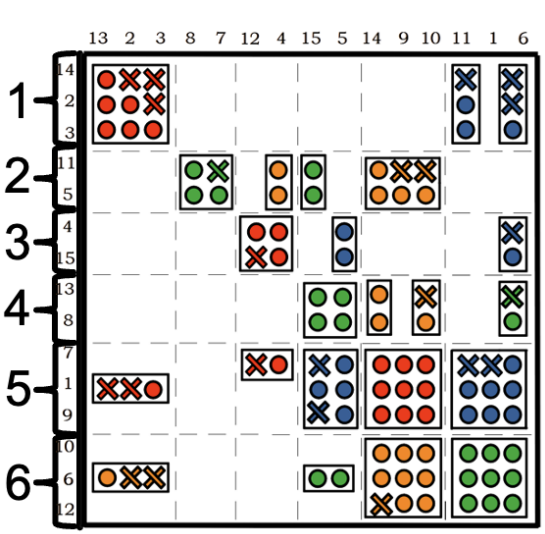
Density distribution of matrices involved in dense BLAS in SuperLU_DIST

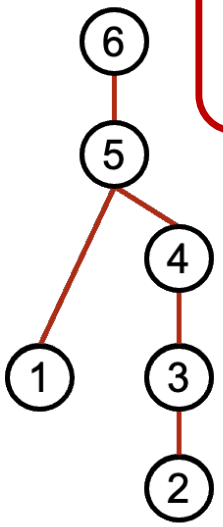
As a result, for the two matrices there is a chance that sparse BLAS is faster compared with dense BLAS.

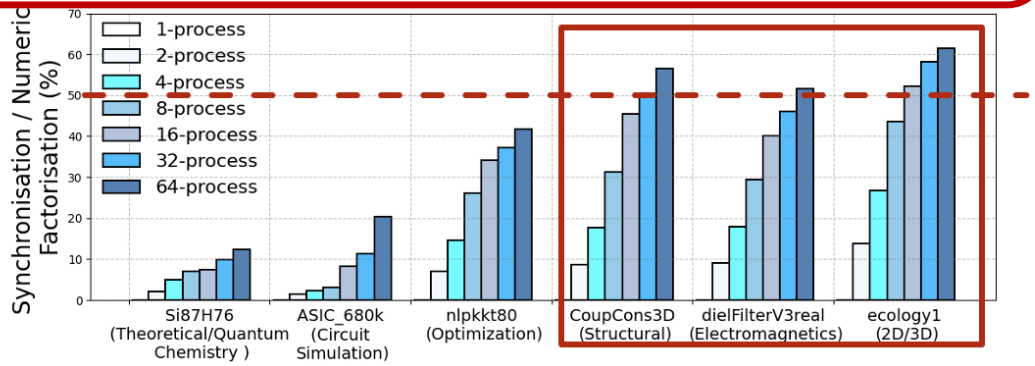
Motivation 3: High Synchronisation Costs

SuperLU_DIST uses the level-set method to generate an elimination tree with tree nodes as the minimum scheduling unit in distributed systems, which can lead to high synchronization costs.

Tree nodes consist of multiple dense BLAS and there are dependencies between each tree node that need to be synchronized resulting in additional synchronization overhead







SuperLU_DIST on 64 NVIDIA A100 GPUs, from 1 to 64 processes

We test six real-world matrices from SuiteSparse Matrix Collection on 64 NVIDIA A100 GPUs.

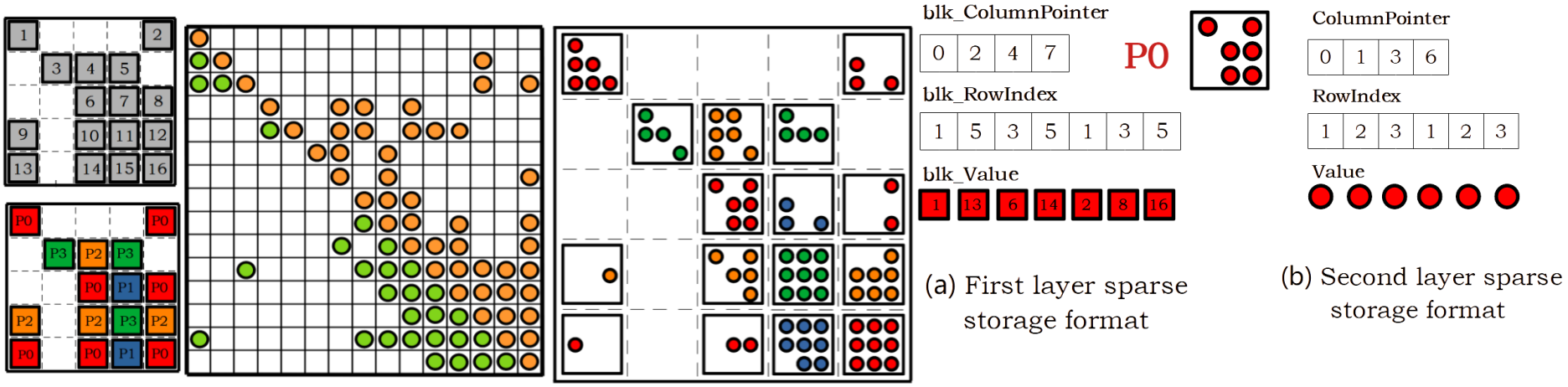
The synchronization cost would be more than 50% at 64 processes on irregular matrices.

We need a new method to reduce synchronization costs.

Data Layout of PanguLU

Our approach is completely different from the classic supernodal method.

PanguLU splits the matrix into multiple blocks of equal size and uses sparse kernel to calculate.



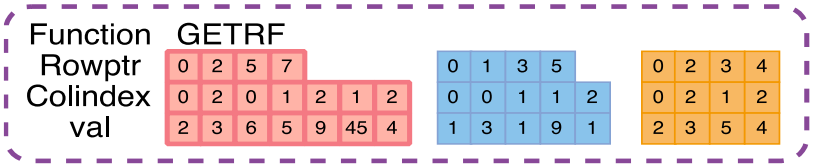
Each process uses a two-layer sparse structure to store the matrix.

First layer sparse structure stores the positional information of sparse matrix blocks. We will get the position of sub-matrix in this layer sparse format.

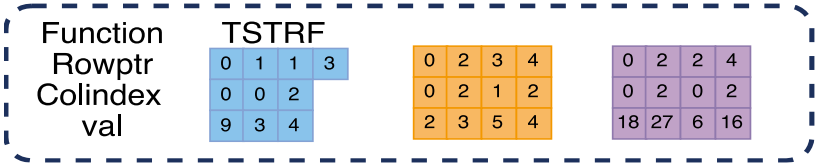
Second layer sparse structure stores the non-zero elements of sub-matrix block with sparse format. We will get internal information about the block, including rows, non-zeros, and sub-matrix structure.

Sparse Kernels of PanguLU

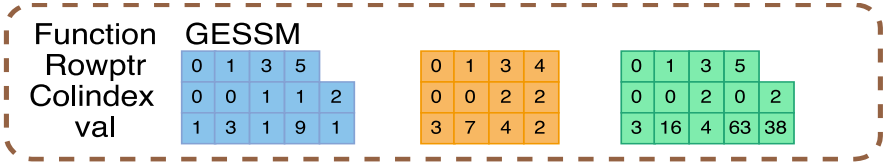
In the numeric factorization of PanguLU, since the matrix blocks are sparse, we develop four dedicated sparse kernels: sparse general triangular factorization (GETRF), sparse upper triangular solve (TSTRF), sparse lower triangular solve (GESSM), and Schur complement with sparse-sparse matrix multiplication (SSSSM).



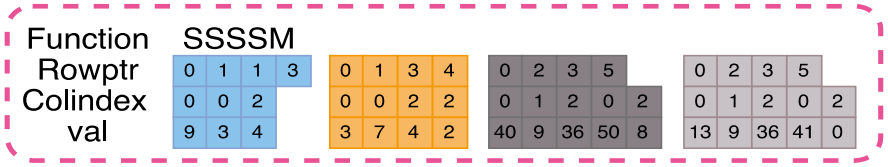
$A = L * U$



$X * U = A$



$L * X = A$



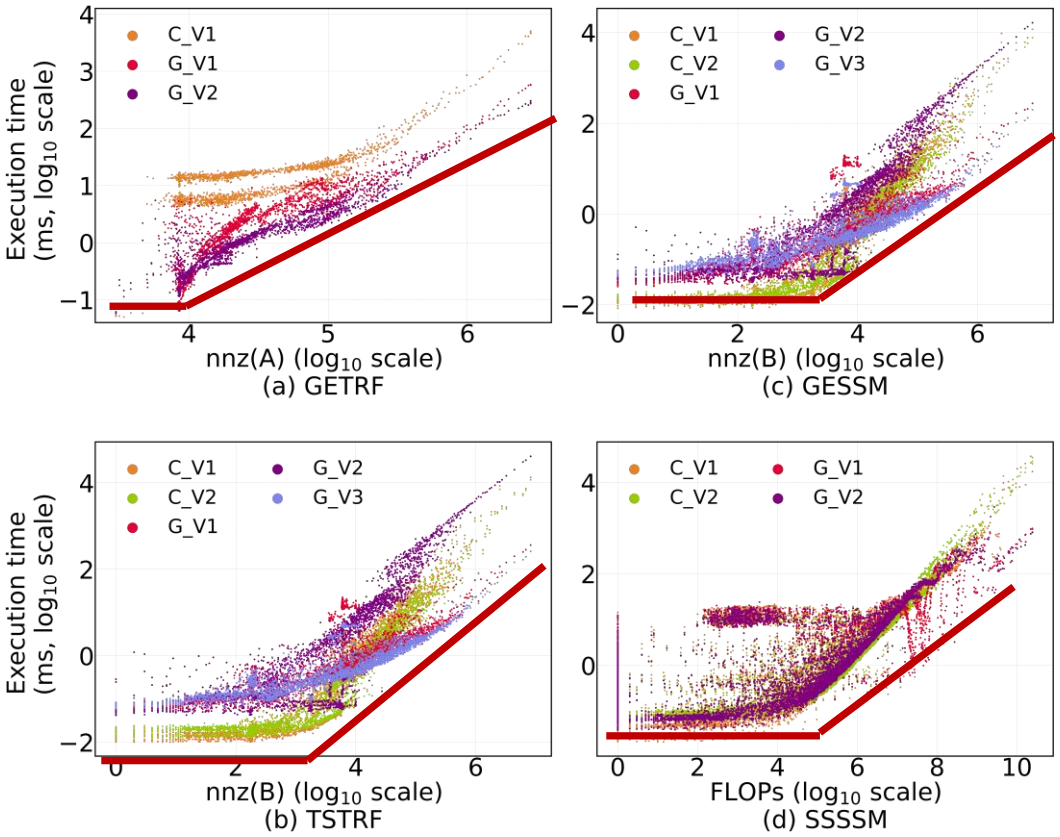
$- L * U + C = C$

All four sparse kernels use sparse sub-matrix blocks as inputs and outputs.

Sparse Kernels of PanguLU

There are several critical factors for the performance of numeric factorzation, such as density, structure and size of the matrix. To exploit the best performance, we implement 17 sparse kernels in PanguLU.

Kernel	Version	Addressing Method	Paralleling Method	Dense Mapping
GETRF	C_V1	Direct	Row	✓
	G_V1	Bin-search	Un-sync SFLU	-
	G_V2	Direct	Un-sync SFLU	✓
TSTRF/ GESSM	C_V1	Merge	Column	-
	C_V2	Direct	Column	✓
	G_V1	Bin-search	Warp-level column	-
	G_V2	Bin-search	Un-sync warp-level row	-
	G_V3	Direct	Warp-level column	✓
SSSSM	C_V1	Direct	Approximate equal load column block	✓
	C_V2	Bin-search	Adaptive split-bin type	-
	G_V1	Bin-search	Adaptive multi-level	-
	G_V2	Direct	Warp-level column	✓



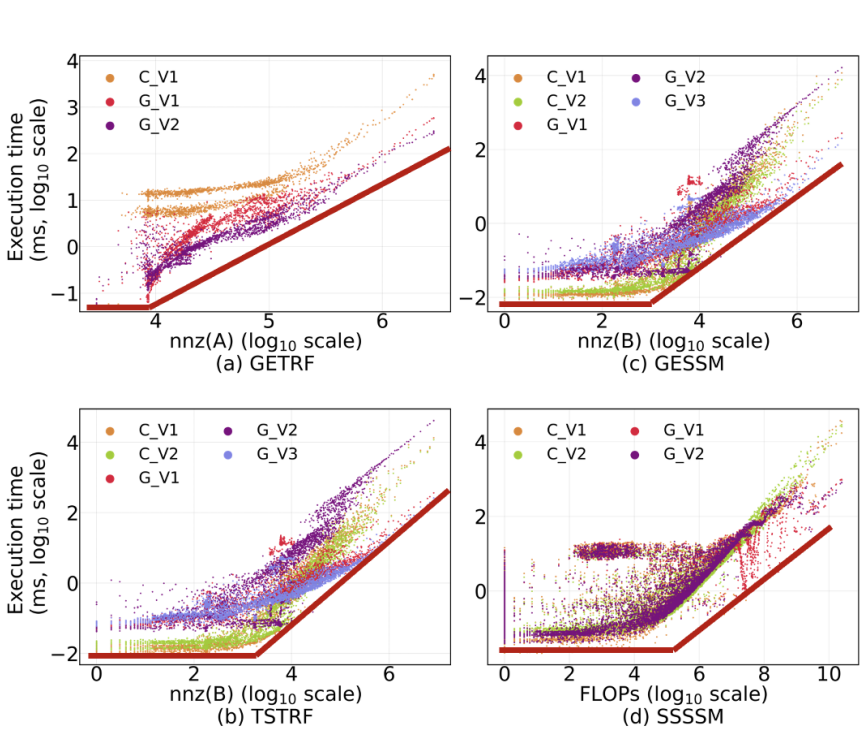
Different sparse kernels have different

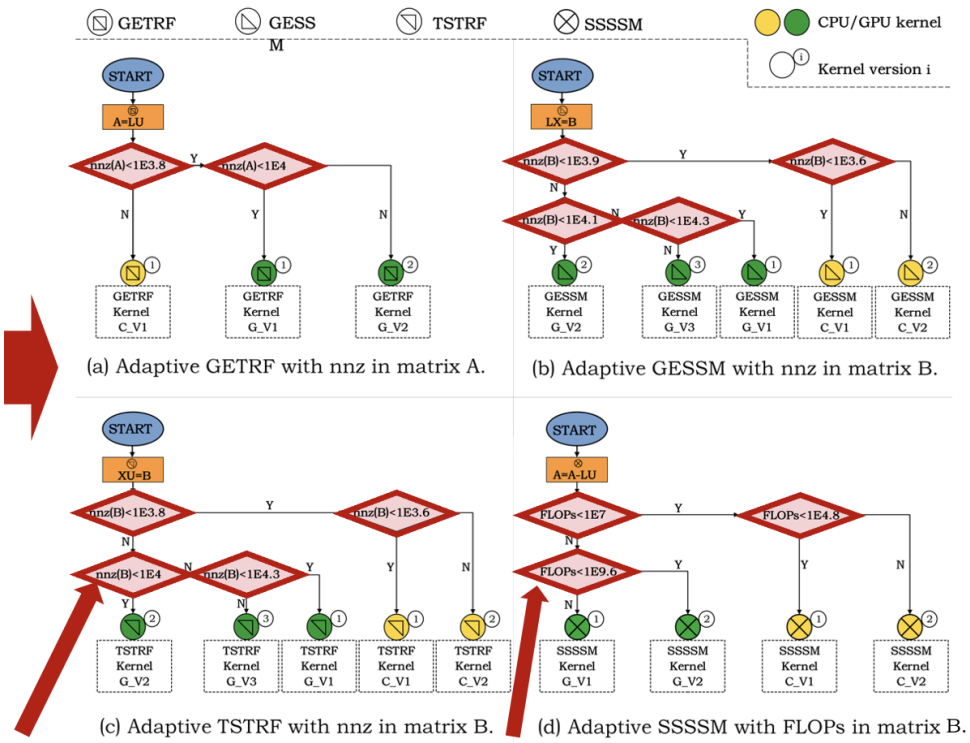
How to select one of these sparse kernels at runtime?

The “Addressing Method” indicates how the calculation value is located and updated. In addition, the “Dense Mapping” represents that the sparse structure is mapped to a dense space.

Sparse Kernels of PanguLU

We propose an algorithm selection strategy for constructing sparse kernels based on large amounts of performance data to select better sparse kernels.

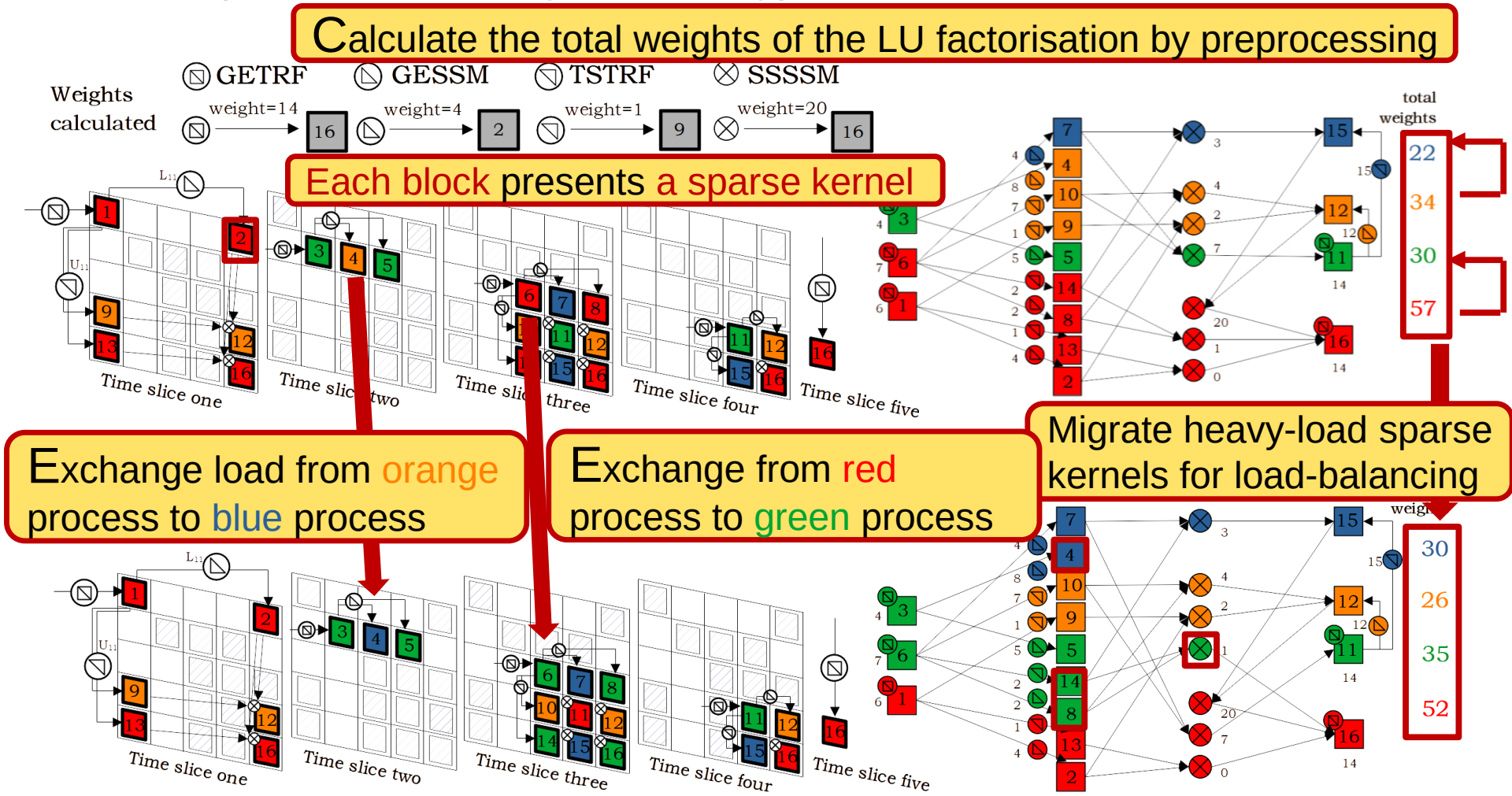




We develop four decision trees according to #non-zeros of matrix A/B and the FLOPs involved in the computation from the performance of 17 sparse kernels to guide our algorithm selection.

Mapping of PanguLU

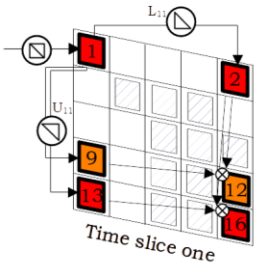
PanguLU develops a mapping method and redistributes the computational load between processes. We map sparse block of heavy-load process to less-load process according to kernel types and time slices.

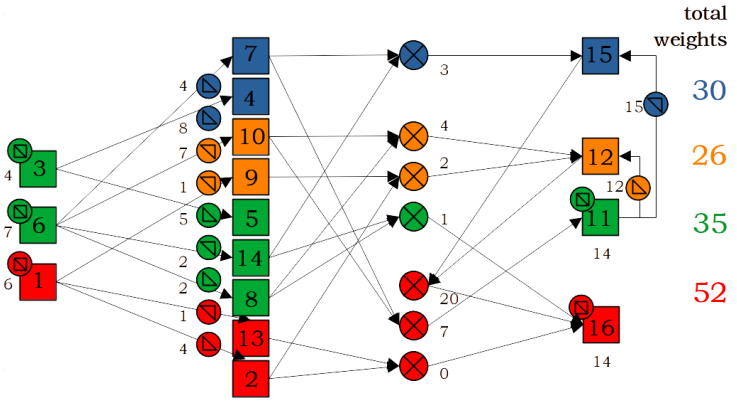


Synchronisation-Free Scheduling

PanguLU are computed sequentially in the order of time slices, to maintain the correctness of the sparse LU factorisation as show in figure.

Executive direction

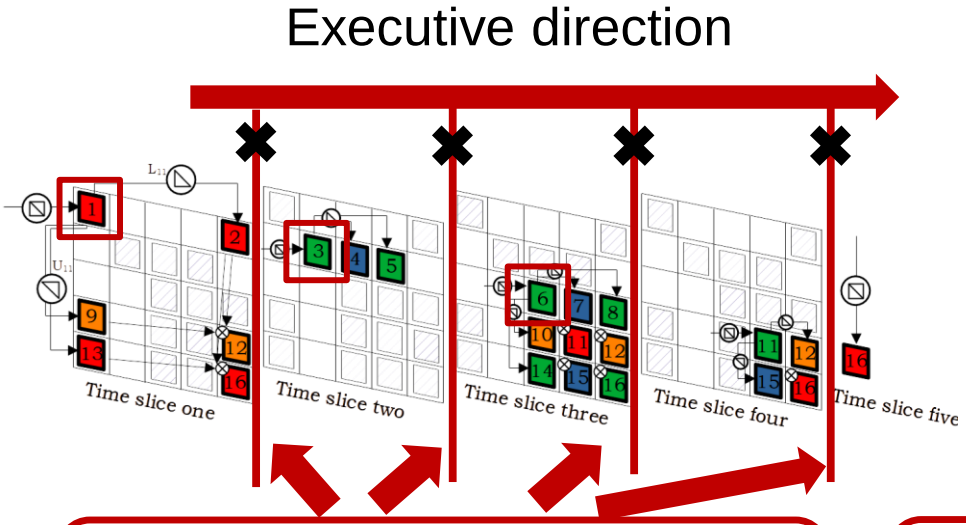


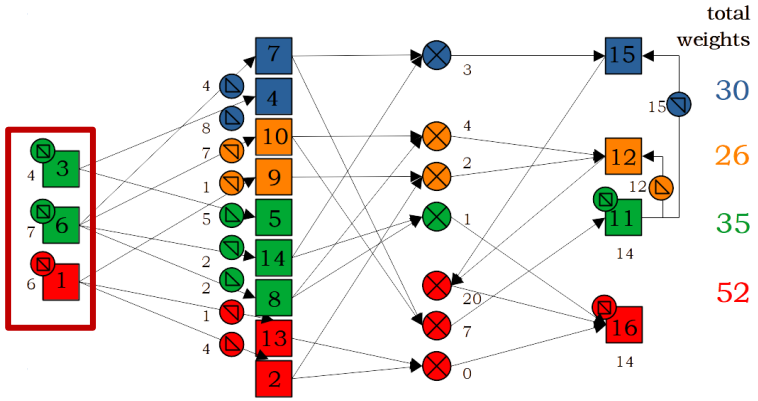


Synchronisation-Free Scheduling

PanguLU are computed sequentially in the order of time slices, to maintain the correctness of the sparse LU factorisation as show in figure.

But obviously these time slices can be computed in parallel on different processes.





Mainly because synchronisation reduces parallelism in distributed systems.

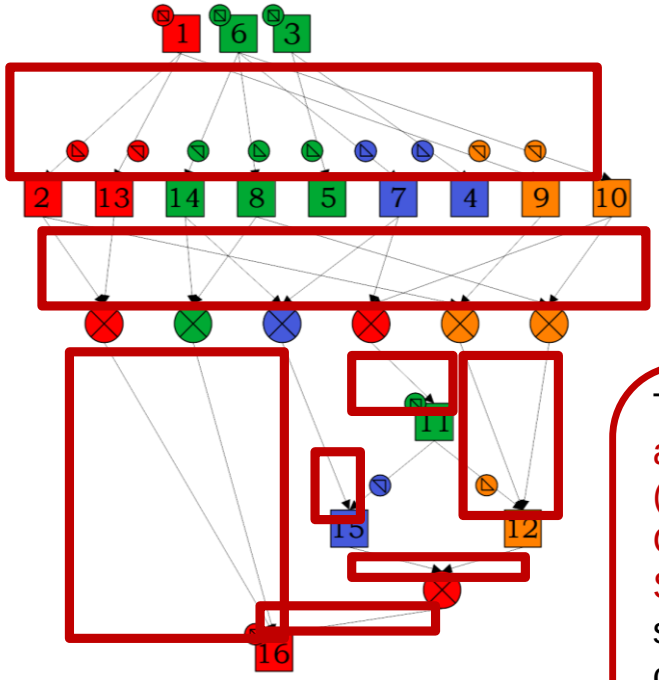
For the blocks 1, 2 and 6, if we do the calculations in time-slice order, these blocks will be executed sequentially.

The synchronisation between each time slice is required to ensure that the correctness of the calculations.

Is there another method of calculation that can guarantee correctness while keeping the processes in a working state as much as possible?

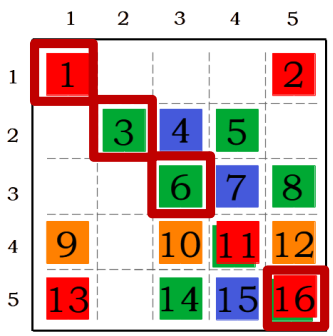
Synchronisation-Free Scheduling

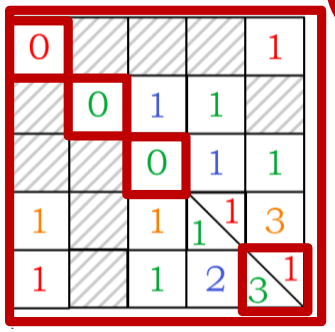
So we researched the order of computation of sparse LU factorisation using sparse kernels.



We find all kernels need to calculate according to the dependencies shown in the figure.

The computation of all sparse kernels (GETRF, TSTRF, GESSM and SSSSM) can only be started after computation of the kernel pointed to by the previous arrow has been completed, which is like the relationship between 0 and non-zero.





Therefore, we can use a synchronisation-free array to keep the number of sparse kernels that still need to be calculated for each block to ensure the correctness of the computation.

For blocks 1, block 2, and block 6, the synchronisation-free array is 0 because there are no arrows.

For block 12, the synchronisation-free array is 3 because there are three arrows.

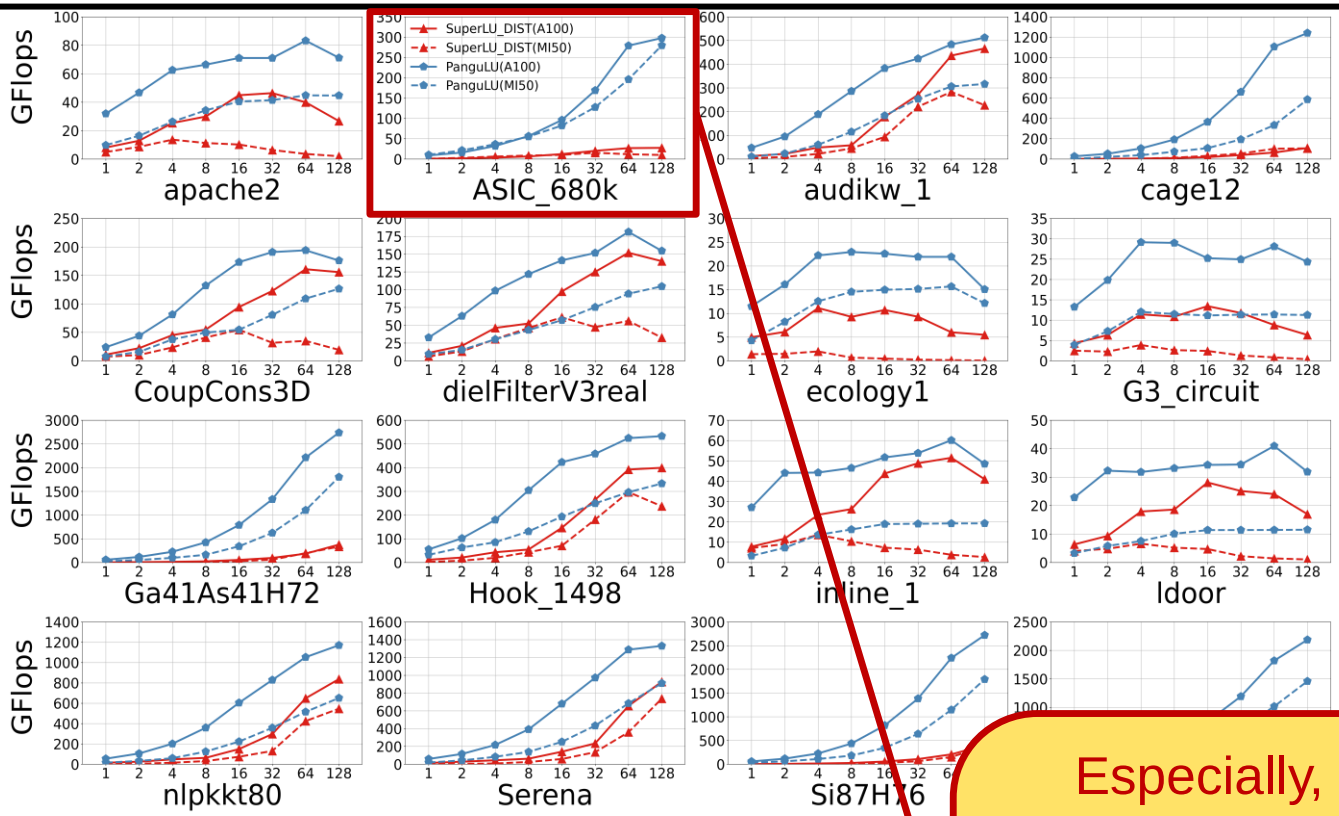
In particular, we do not use any synchronisation in here.

Experimental Result

Experimental Setup

Two experimental platforms	Two heterogeneous distributed Solvers
4 * NVIDIA A100 GPUs (40 GB, B/W 1555GB/s) 2 * Intel Xeon 8180 CPUs @ 2.5 GHz (512GB DDR4)	SuperLU_DIST 8.1.2 PanguLU 3.5
4 * AMD MI50 GPUs (16 GB, B/W 1024GB/s) 1 * AMD Epyc 7601 CPU @ 2.2 GHz (128GB DDR4)	SuperLU_DIST 8.1.2 PanguLU 3.5

Experimental Result



In this figure, the red lines and blue lines the performance of SuperLU_DIST and respectively, while the solid lines and do indicate the performance on the A100 G MI50 GPU platforms, respectively.

Compared with SuperLU_DIST, PanguLU achieves an average speedup of 2.53x and 2.79x on the NVIDIA GPU platform and the AMD GPU platform, with speedups ranging from 1.10x - 11.70x and 1.12x - 17.97x

Especially, PanguLU achieves significant performance advantages for irregular matrices such as ASIC_680k (circuit matrix) with speedups of up to 11.70x and 17.97x on the two GPU platforms, respectively.

Experimental Result

PanguLU achieves better performance by using sparse kernel on a single GPU. In the 16 matrices, PanguLU has a geometric mean of 6.54x compared with SuperLU.

Matrix	Panel Fact Time (s)		Schur Time (s)		All Time (s)		
	SuperLU	PanguLU	SuperLU	PanguLU	SuperLU	PanguLU	Speedup
apache2	9.4	6.32	27.36	3.86	36.76	10.18	3.61x
ASIC_680k	3.39	3.21	982.36	17.8	985.76	21.01	46.92x
audikw_1	103.44	47.94	764.83	159.35	868.27	207.29	4.19x
cage12	15.8	13	3570.76	70.65	3586.56	83.65	42.88x
CoupCons3D	15.69	11.11	55.55	8.5	71.24	19.61	3.63x
dielFilterV3real	41.62	24.63	127.92	25.27	169.54	49.9	3.40x
ecology1	2.26	1.8	2.88	0.57	5.14	2.37	2.17x
G3_circuit	5.69	4.98	14.71	1.35	20.4	6.34	3.22x
Ga41As41H72	979.85	80.91	21893.96	1275.41	22873.81	1356.32	16.86x
Hook_1498	117.77	66.3	1208.79	230.17	1326.56	296.47	4.47x
inline_1	10.15	6.15	19.66	2.04	29.82	8.19	3.64x
ldoor	7.2	4.55	13.87	1.05	21.07	5.59	3.77x
nlpkkt80	173.69	80.48	1816.94	537.19	1990.63	617.68	3.22x
Serena	238.6	115.98	3551.09	899.36	3789.69	1015.35	3.73x
Si87H76	152.21	74.88	17245.73	1034.96	17397.94	1109.84	15.68x
SiO2	77.78	38.57	9762.91	410.6	9840.69	449.17	21.91x
Geometric Mean	-		-		-		6.54x

Kernel time comparison of SuperLU_DIST and PanguLU for 16 test matrices on a signle A100 GPU.

Contents

- Background
- Sparse Linear Solvers for Circuit Simulation
  - SFLU: Synchronization-Free Sparse LU Factorization for Fast Circuit Simulation on GPUs
  - Accelerating Sparse LU Factorization with Density-Aware Adaptive Matrix Multiplication for Circuit Simulation
  - PanguLU: A Scalable Regular Two-Dimensional Block-Cyclic Sparse Direct Solver on Distributed Heterogeneous Systems
- **Conclusions**

Sparse Linear Solvers for Circuit Simulation

Machine Learning and GPU Accelerated Sparse Linear Solver (on CPU, GPU and Distributed Heterogeneous Systems):

- Jianqi Zhao, Yao Wen, Yuchen Luo, Zhou Jin, Weifeng Liu, Zhenya Zhou. SFLU: Synchronization-Free Sparse LU Factorization for Fast Circuit Simulation on GPUs.
  The 58th ACM/IEEE Design Automation Conference (DAC' 21). 2021.
- https://gitee.com/ssslab/sflu

- Tengcheng Wang, Wenhao Li, Haojie Pei, Yuying Sun, Zhou Jin, Weifeng Liu. Accelerating Sparse LU Factorization with Density-
  Aware Adaptive Matrix Multiplication for Circuit Simulation. The 60th ACM/IEEE Design Automation Con ference (DAC' 23). 2023
- https://github.com/SuperScientificSoftwareLaboratory/DALU

- Xu Fu, Bingbin Zhang, Tengcheng Wang, Wenhao Li, Yuechen Lu, Enxin Yi, Jianqi Zhao, Xiaohan Gen g, Fangying Li, Jingwen Zhang, Zhou Jin, Weifeng Liu. PanguLU: A Scalable Regular Two-Dimensional BlockCyclic Sparse Direct Solver on Distributed Heterogeneous Systems. 36th ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC' 23). 2023. (Best paper award)
- https://www.ssslab.cn/software.html

An overview of surveyed sparse direct solver

	Solver	Left/Right looking	Blocking method	Kernel	Parallelism				
					Level	Method	Distributed	Multi-thread	GPU
General	MUMPS	Left	Multifrontal	Level-3 BLAS	Tree/Node	Level-set	√	√	—
	UMFPACK	Left	Multifrontal	Level-3 BLAS	—	—	—	—	—
	SuperLU_DIST	Right	Supernode	Level-3 BLAS	Supernode	Level-set	√	√	√
	PARDISO	Left & Right	Supernode	Level-3 BLAS	Supernode	Pipeline	√	√	—
	PanguLU	Right	Regular 2D block	Adaptive sparse kernel	2D block	Synchronization-free	√	√	√
Dedicated to circuit matrix	KLU	Left	Block diagonal	—	—	—	—	—	—
	NICSLU	Left	Supernode	Level-3 BLAS	Supernode	Level-set	—	√	√
	FLU	Left	Supernode	Level-3 BLAS	Supernode	Register Level	—	√	—
	GLU	Left & Right	—	—	Element	Level-set	—	—	√
	SFLU	Left & Right	—	—	Element	Synchronization-free	—	—	√
	Density-aware LU	Right	Supernode	Adaptive kernel	Supernode	Level-set	√	√	—

Some Observations for Future

The future opportunities in this evolving landscape:

- Taking into account various factors such as the matrix and hardware platform characteristics in algorithm design.
  - How to introduce sparse computations and strike a trade-off between dense and sparse kernels for different computational tasks remains a pivotal challenge.
  - Better task scheduling strategies and computational kernels need to be designed on GPUs to improve the parallel computing performance.

Welcome to collaborate with us!

Thanks!

Zhou Jin

jinzhou@cup.edu.cn