Hybrid-Right-Looking Sparse LU Solver on GPU - GLU V 3.0

Prof. Sheldon Tan

Shaoyi Peng

stan@ece.ucr.edu

Department of Electrical Computer Engineering

University of California, Riverside, CA

Outline

- Introduction
- LU Factorization introduction
- GPU based hybrid right-looking LU solver (GLU 1.0)
- New double-U column dependency and GLU 2.0
- GLU 3.0
  - New double-U dependency detection algorithm
  - New GPU kernel with dynamic resource allocation
- Numerical results
- Summary

Review of LU factorization

- Right-looking algorithm
- Left-looking algorithm
- GPU-based Hybrid column-based right-looking LU (GLU 1.0)

Right-looking LU factorization

- Solve 1 row of $\boldsymbol{U}$ and 1 column of $\boldsymbol{L}$ at each step
  - Recursive procedure, **no parallelism**

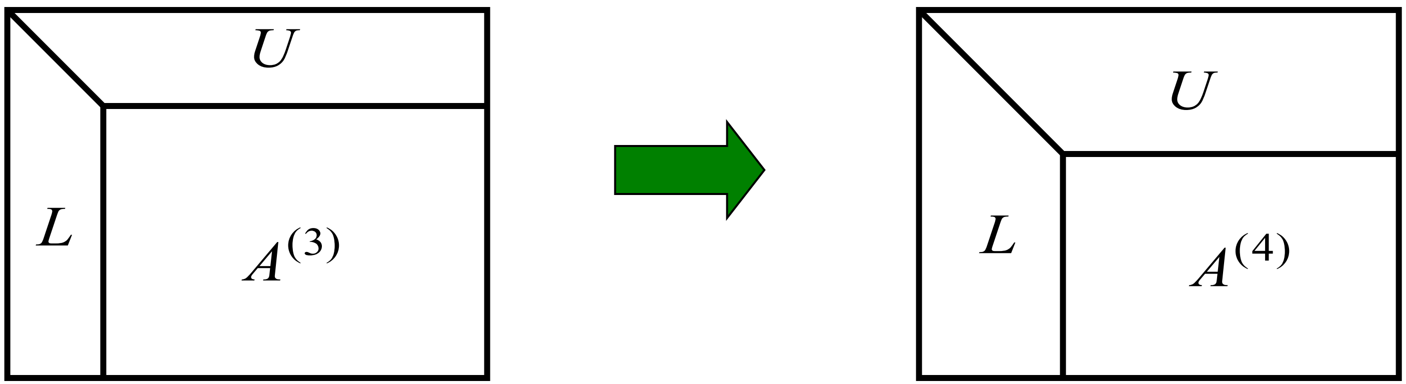
- $$\begin{bmatrix} a_{11} & \boldsymbol{a_{12}} \\ \boldsymbol{a_{21}} & \boldsymbol{A}_{22} \end{bmatrix} = \begin{bmatrix} l_{11} & \\ \boldsymbol{l_{21}} & \boldsymbol{L}_{22} \end{bmatrix} \begin{bmatrix} u_{11} & \boldsymbol{u_{12}} \\ & \boldsymbol{U}_{22} \end{bmatrix}$$

where $l_{11} = 1$

$$\left\{ \begin{aligned} u_{11} &= a_{11} \\ \boldsymbol{u_{12}} &= \boldsymbol{a_{12}} \\ \boldsymbol{l_{21}} u_{11} &= \boldsymbol{a}_{21} \\ \boldsymbol{l}_{21} \boldsymbol{u}_{12} + \boldsymbol{L}_{22} \boldsymbol{U}_{22} &= \boldsymbol{A}_{22} \end{aligned} \right.$$

Storage of LU Factorization

Using only one 2-dimensional array !



- In sparse matrix implementation, this type of storage requires increasing memory space because of fill-ins during the factorization.
- The Doolittle and Crout methods are called right-looking method we looking for the right direction as the algorithm progresses.

Left-looking LU method

$$\begin{bmatrix} L_{11} & & \\ l_{21} & 1 & \\ L_{31} & l_{32} & L_{33} \end{bmatrix} \begin{bmatrix} U_{11} & u_{12} & U_{13} \\ & u_{22} & u_{23} \\ & & U_{33} \end{bmatrix} = \begin{bmatrix} A_{11} & a_{12} & A_{13} \\ a_{21} & a_{22} & a_{23} \\ A_{31} & a_{32} & A_{33} \end{bmatrix}$$

If the k-1 columns of L and U are known, then we can compute kth columns of L an U:

$$L_{11}u_{12} = a_{12} \Rightarrow u_{12}$$

$$l_{21}u_{12} + u_{22} = a_{22} \Rightarrow u_{22}$$

$$L_{31}u_{12} + l_{32}u_{22} = a_{32} \Rightarrow l_{32}$$

Column Vector

Row vector

Unknown variables

Left-looking LU method (cont'd)

We can solve following triangular equation:

$$L_{11}u_{12} = a_{12} \Rightarrow u_{12}$$
$$l_{21}u_{12} + u_{22} = a_{22} \Rightarrow u_{22}$$
$$L_{31}u_{12} + l_{32}u_{22} = a_{32} \Rightarrow l_{32}$$

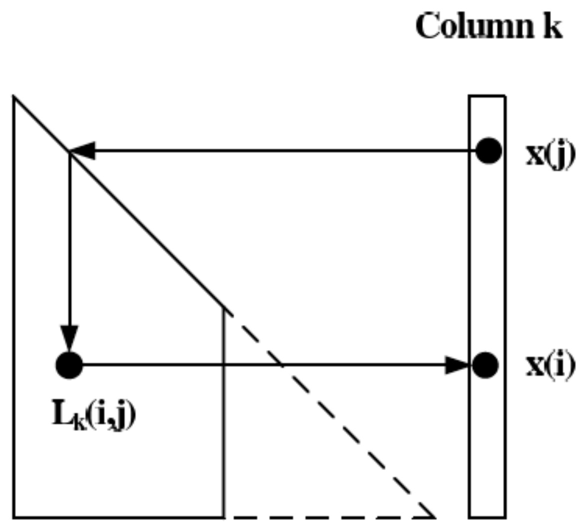
$$\begin{bmatrix} L_{11} & & \\ \boxed{l_{21}} & 1 & \\ L_{31} & 0 & I \end{bmatrix} \begin{bmatrix} \boxed{x_1} \\ x_2 \\ \boxed{x_3} \end{bmatrix} = \begin{bmatrix} a_{12} \\ a_{22} \\ a_{32} \end{bmatrix}$$

After this, we can have: (in matlab: lu_left function)

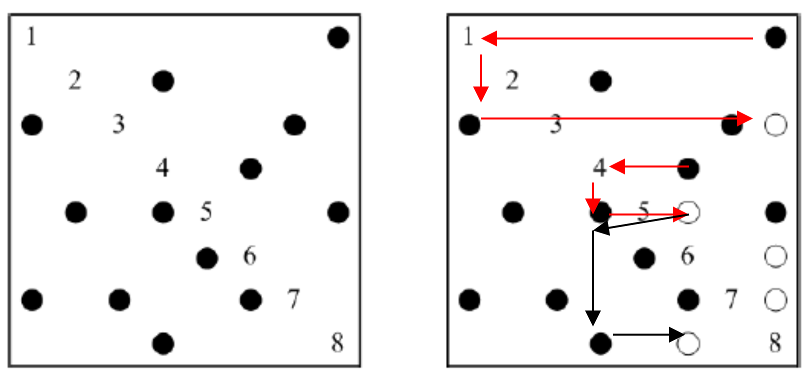
$$u_{12} = x_1$$
$$u_{22} = x_2$$
$$l_{32} = x_3 / u_{22}$$

Nonzero pattern for a sparse triangular matrix and symbolic analysis

If x(j) (or U(j,k)) is not zero, L(i,j) is not zero, then *i*th element in *k*th column in LU will not zero.



$$x(i) = x(i) - x(j)L(i,j)$$



How about those two nonzero fill-ins?

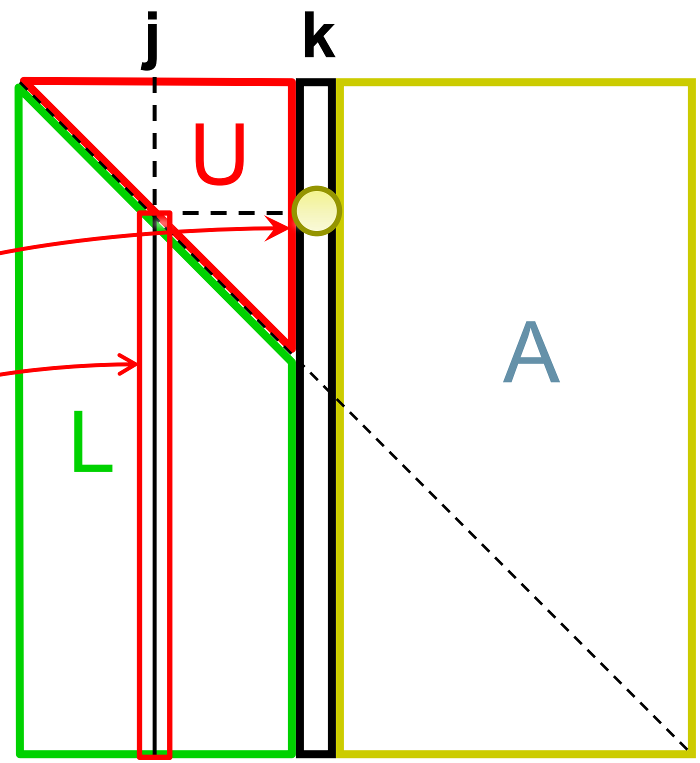
Fig. 6. The original matrix A (left) and the matrix A (right) after symbolic analysis with predicted nonzero pattern of LU factors of A

Sparse left-looking LU method illustration

- LU factorization implementation example

Algorithm 1 Sequential G/P left-looking algorithm

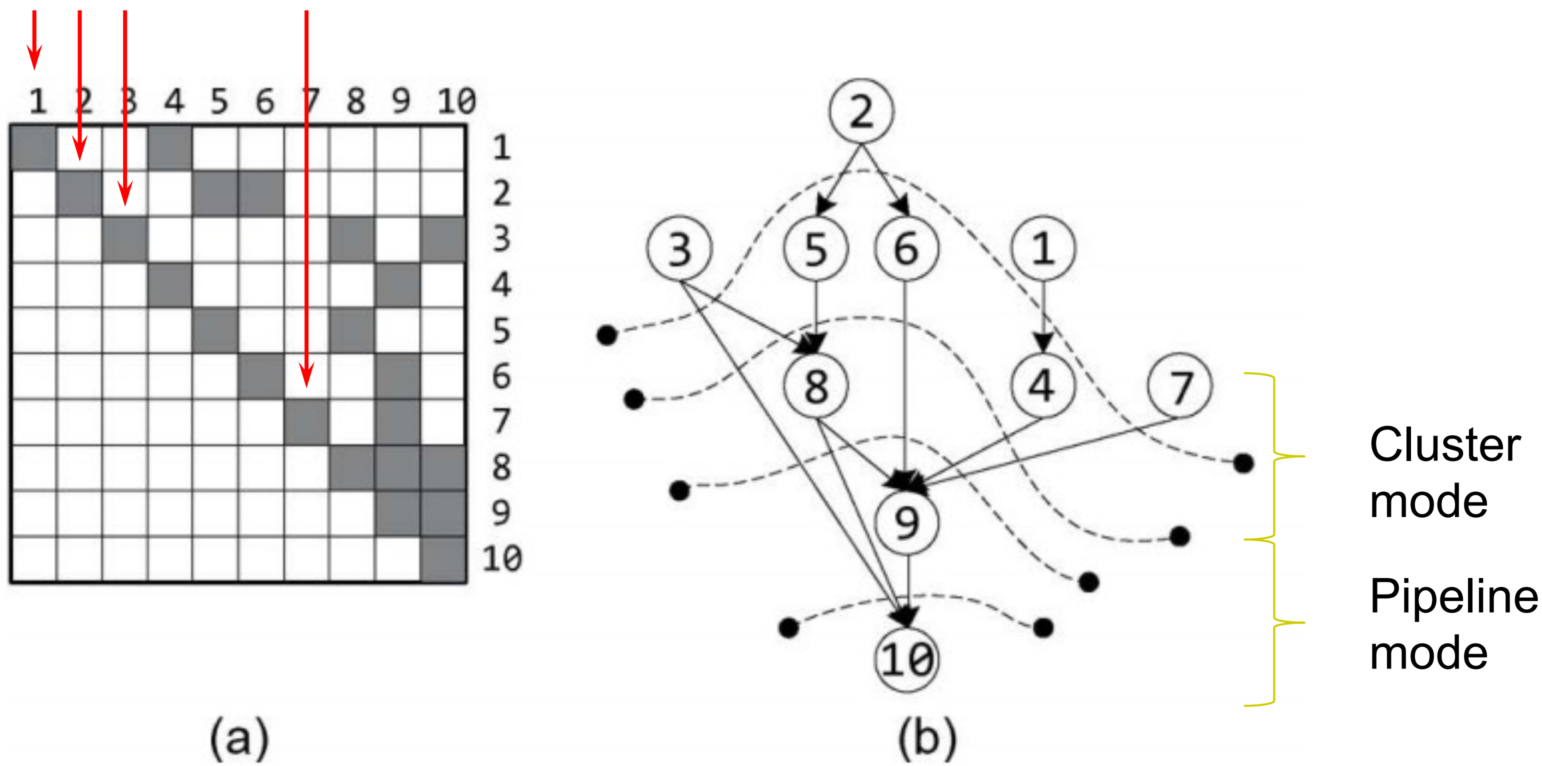
$\mathbf{L} = \mathbf{I}$
for $k = 1 : n$ **do**
  // *solving* $Lx = b$ $b = A(:, k)$ *the kth column of* $\mathbf{A}$
  $x = b$;
  for $j = 1 : k - 1$ where $U(j, k) \neq 0$ **do**
    // *Vector MAD*
    $x(j + 1 : n) = x(j + 1 : n) - L(j + 1 : n, j) \cdot x(j)$;
  end for
  $U(1 : k, k) = x(1 : k)$;
  $L(k : n, k) = x(k : n)/U(k, k)$;
end for



Column k depends on column j, if $U(j, k) \neq 0$ $(j < k)$, note that L and U matrices have nonzeron fills after symbolic analysis

Sparse left-looking LU data dependence

- parallel solving of each column and column dependence analysis



Example of an upper triangular matrix U and its scheduling graph

The G/P left-looking algorithm

The left-looking algorithm: the key operation is solving triangular matrix.

Algorithm 1 G/P Left-Looking Algorithm

for $j = 1$ to n **do**
  /*Triangular matrix solving*/
  for $k = 1$ to $j - 1$ where $A_s(k,j) \neq 0$ **do**
    /*Vector multiple-and-add*/
    for $i = k + 1$ to n where $A_s(i,k) \neq 0$ **do**
      $A_s(i,j) = A_s(i,j) - A_s(i,k) * A_s(k,j)$
    end for
  end for
  /*Compute column j for L matrix*/
  for $i = j + 1$ to n where $A_s(i,j) \neq 0$ **do**
    $A_s(i,j) = A_s(i,j)/A_s(j,j)$
  end for
end for

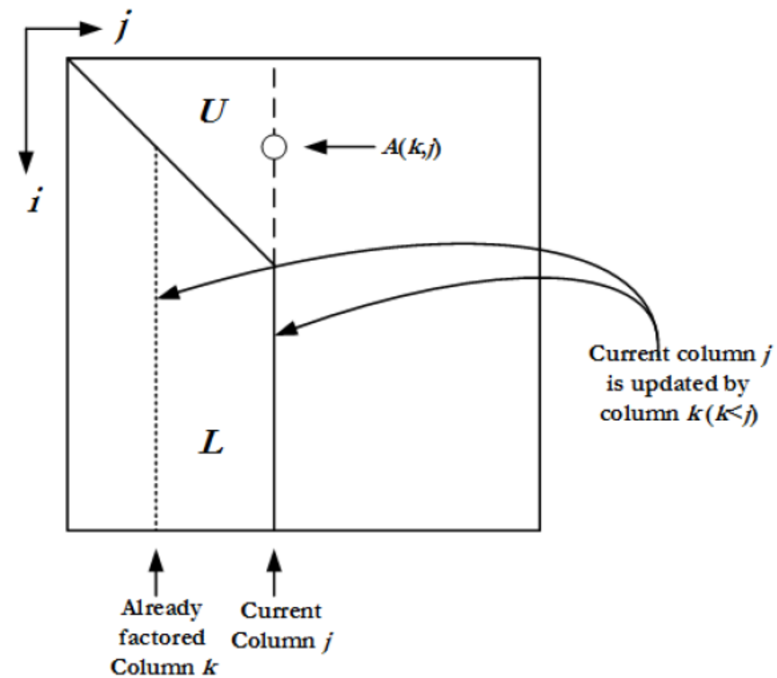
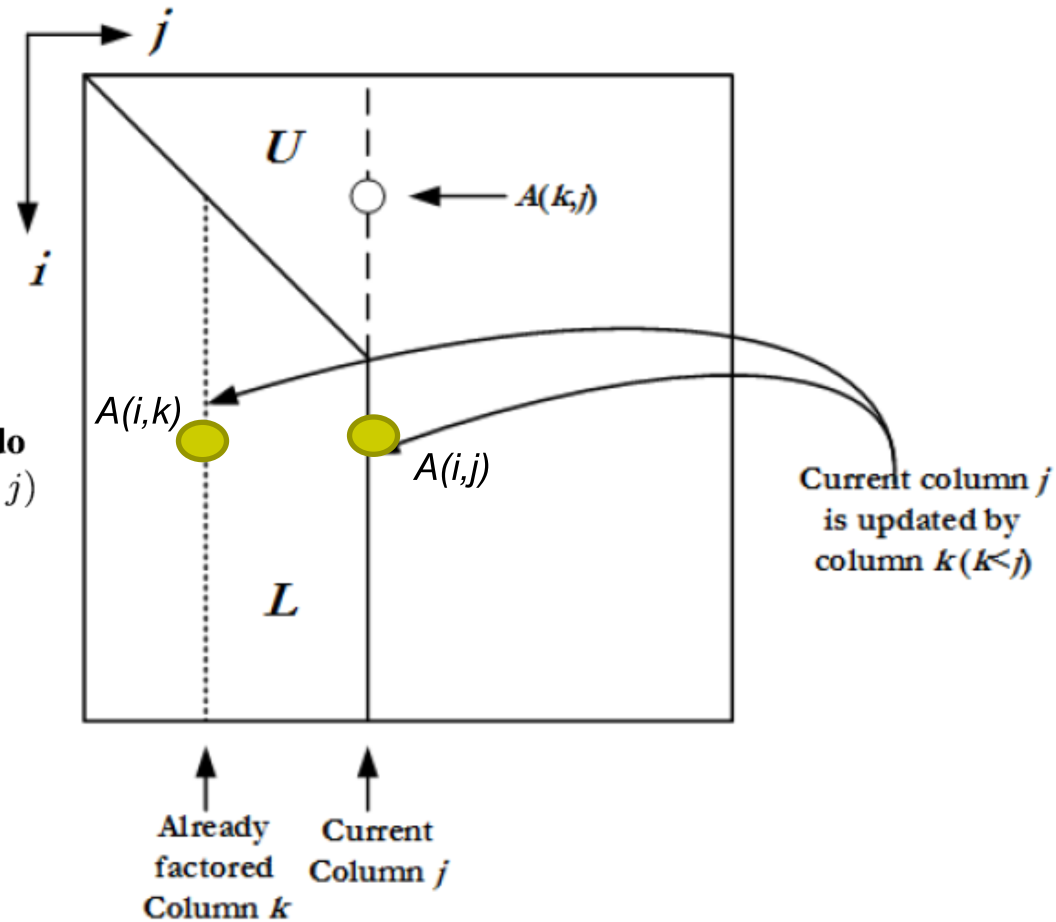


Fig. 1. Left-looking update for column j.

The triangle matrix solving in G/P LL method

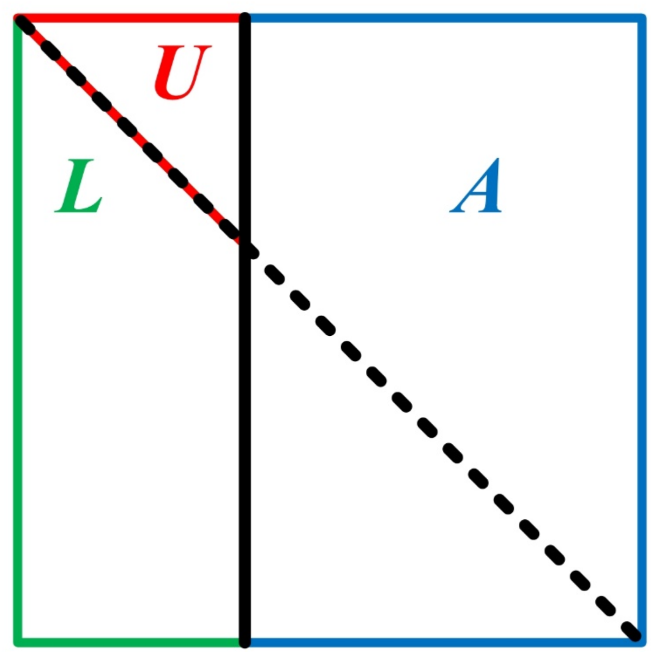
Triangle matrix solving

for $k = 1$ to $j - 1$ where $A_s(k, j) \neq 0$ **do**
    /*Vector multiple-and-add*/
    for $i = k + 1$ to n where $A_s(i, k) \neq 0$ **do**
        $A_s(i, j) = A_s(i, j) - A_s(i, k) * A_s(k, j)$
    end for
end for



Left-looking LU factorization (cont'd)

- Two loops can be parallelized
  - Look j: index the current column
  - Loop i: Element-wide multiply and add (MAD) operation



Current column j

GLU 1.0 (2016)

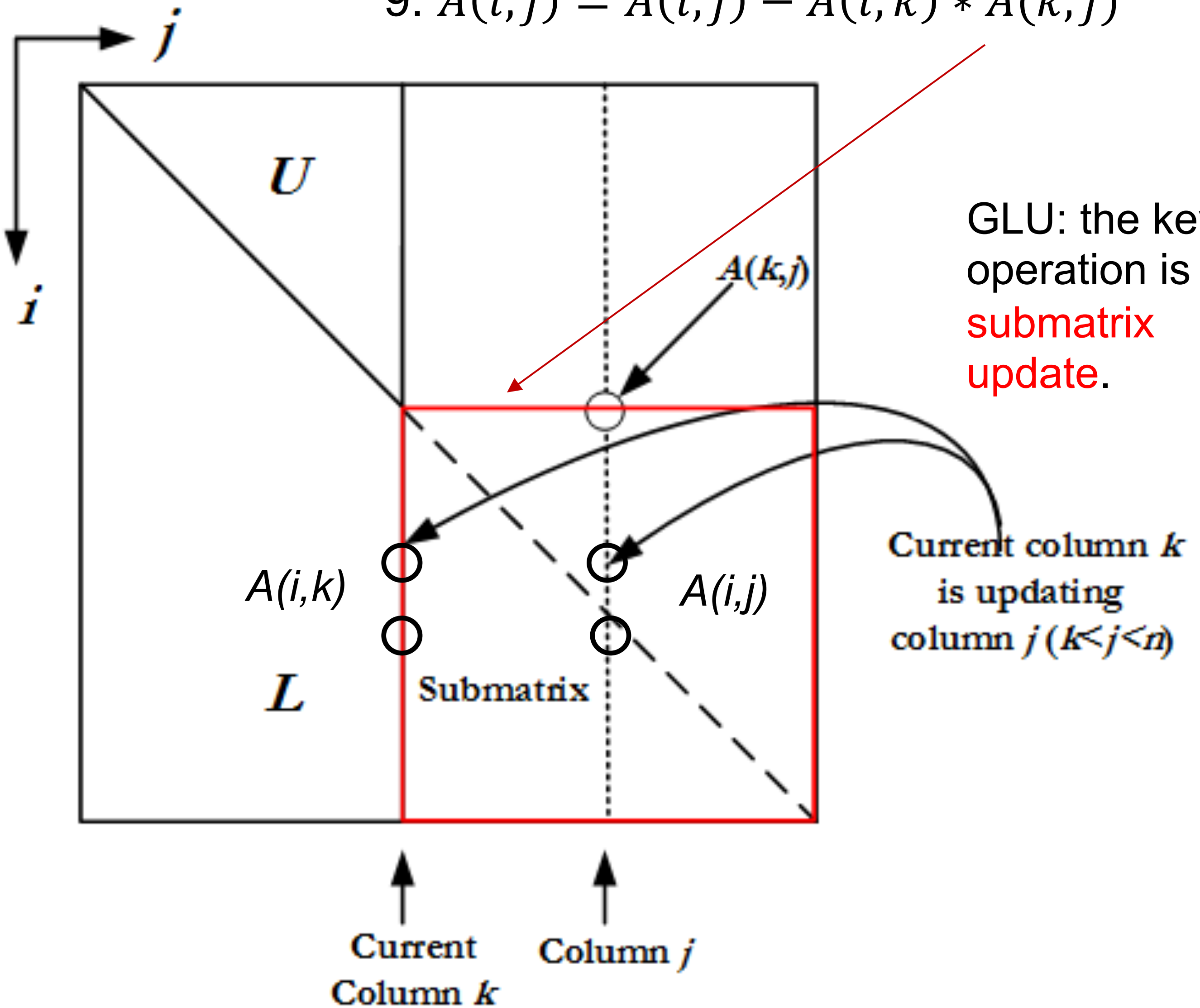
Hybrid right-looking LU solver-GLU 1.0 [He, TVLSI16]

Algorithm 2 Hybrid Column-Based Right-Looking Algorithm

for $k = 1$ to n **do**
    /*Compute column k of L matrix*/
    for $i = k + 1$ to n where $A_s(i, k) \neq 0$ **do**
        $A_s(i, k) = A_s(i, k)/A_s(k, k)$
    end for
    /*Update the submatrix for next iteration*/
    for $j = k + 1$ to n where $A_s(k, j) \neq 0$ **do**
        for $i = k + 1$ to n where $A_s(i, k) \neq 0$ **do**
            $A_s(i, j) = A_s(i, j) - A_s(i, k) * A_s(k, j)$
        end for
    end for
end for

Hybrid column based right-looking update in GLU

9: $A(i,j) = A(i,j) - A(i,k) * A(k,j)$



GLU: the key operation is submatrix update.

The submatrix operation in matrix format in GLU

- Given k is the current column, the submatrix operation has two operations:

  (1) Two vector tensor production

  (2) Two matrix addition after (1)

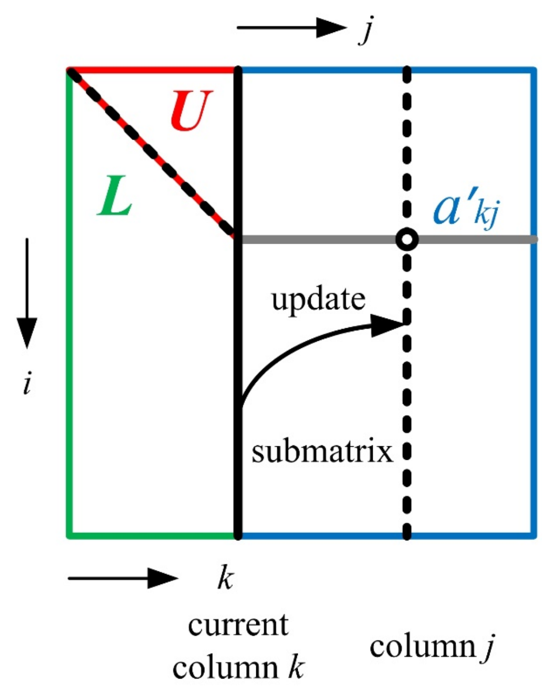
```
/*Update the submatrix for next iteration*/
for j = k + 1 to n where A_s(k, j) ≠ 0 do
    for i = k + 1 to n where A_s(i, k) ≠ 0 do
        A_s(i, j) = A_s(i, j) − A_s(i, k) ∗ A_s(k, j)
    end for
end for
```

$$-\begin{bmatrix} A(k+1,k) \\ A(k+2,k) \\ . \\ . \\ A(n,k) \end{bmatrix} [A(k,k+1), A(k,k+2), .., A(k,n)] + \begin{bmatrix} A(k+1,k+1) & \cdots & A((k+1,n) \\ \vdots & \ddots & \vdots \\ A(n,k+1) & \cdots & A(n,n) \end{bmatrix}$$

The size of the matrix is NxN, where N = n-k, the size of the two vectors is Nx1, and 1xN. Both two vectors and NxN matrix are sparse matrices. As a result, we can easily parallelize the vector and matrix operations

Hybrid column-based right-looking LU (GLU)

- Three loops can be parallelized
  - Loop k: index for current column
  - Loop j: update submatrix, compute partial column j
  - Loop i: in the partial column j, do MAD



GLU 2.0 (2017)

GLU V2.0 flowchart

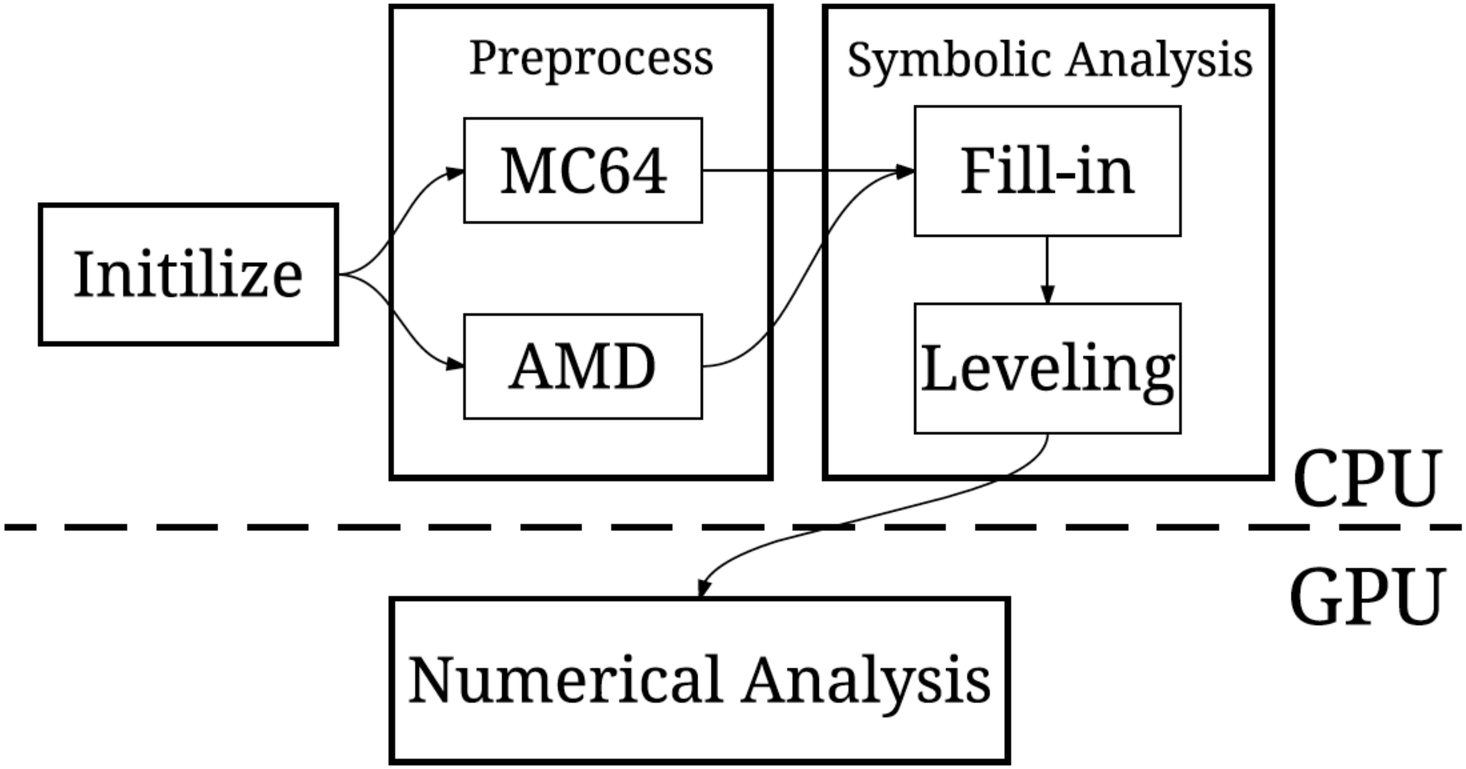


Figure 2.1: The flowchart of GLU V2.0

GLU flowchart (cont'd)

- Step 1: preprocessor
  - Scaling and permuting by MC64 and AMD
- Step 2: symbolic factorization
  - Fill-in prediction
  - CSR prediction
  - Column dependency prediction
- Step 3: numerical factorization
  - Compute L and U level by level

CPU
(only once)

GPU

HSL_MC64 algorithm

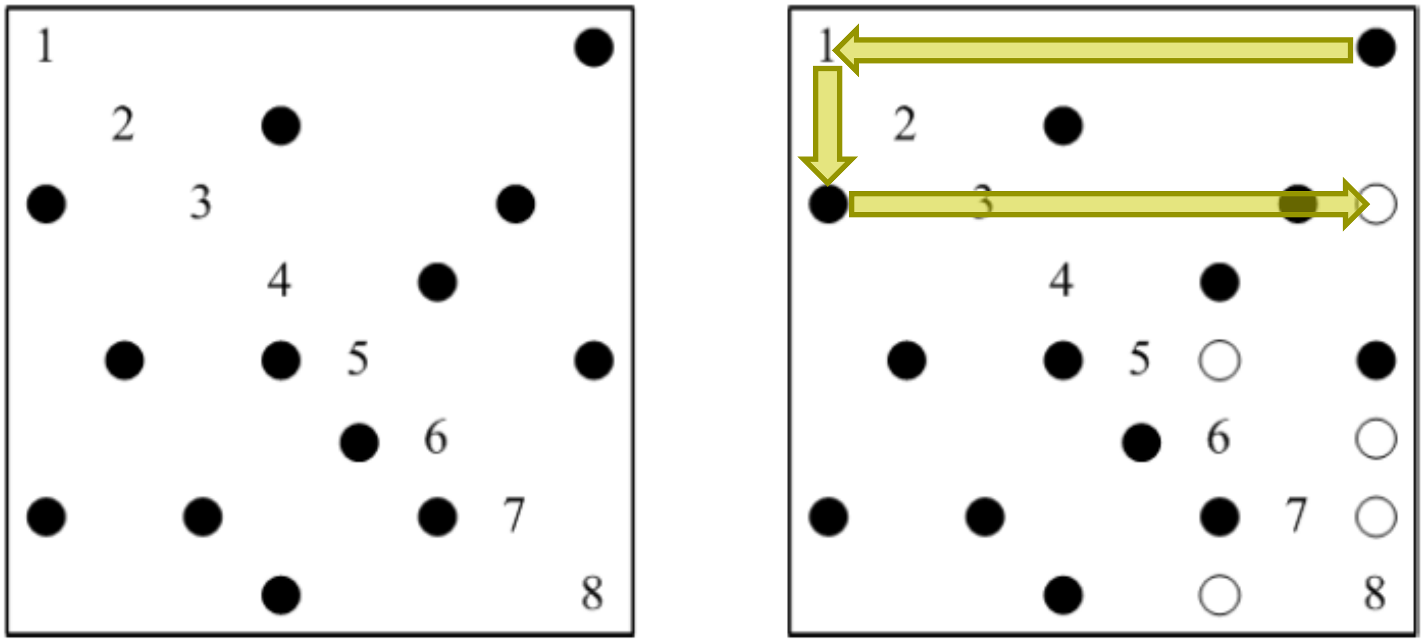
- Permute and scale an asymmetric matrix to put large entries on the diagonal

$$A_p = P D_r A D_c Q$$

where A is the original matrix, D_r and D_c are two diagonal matrices to scale A to enhance numerical stability; P and Q are row and column permutation matrices, which are used to maintain sparsity (i.e. reduce fill-ins)

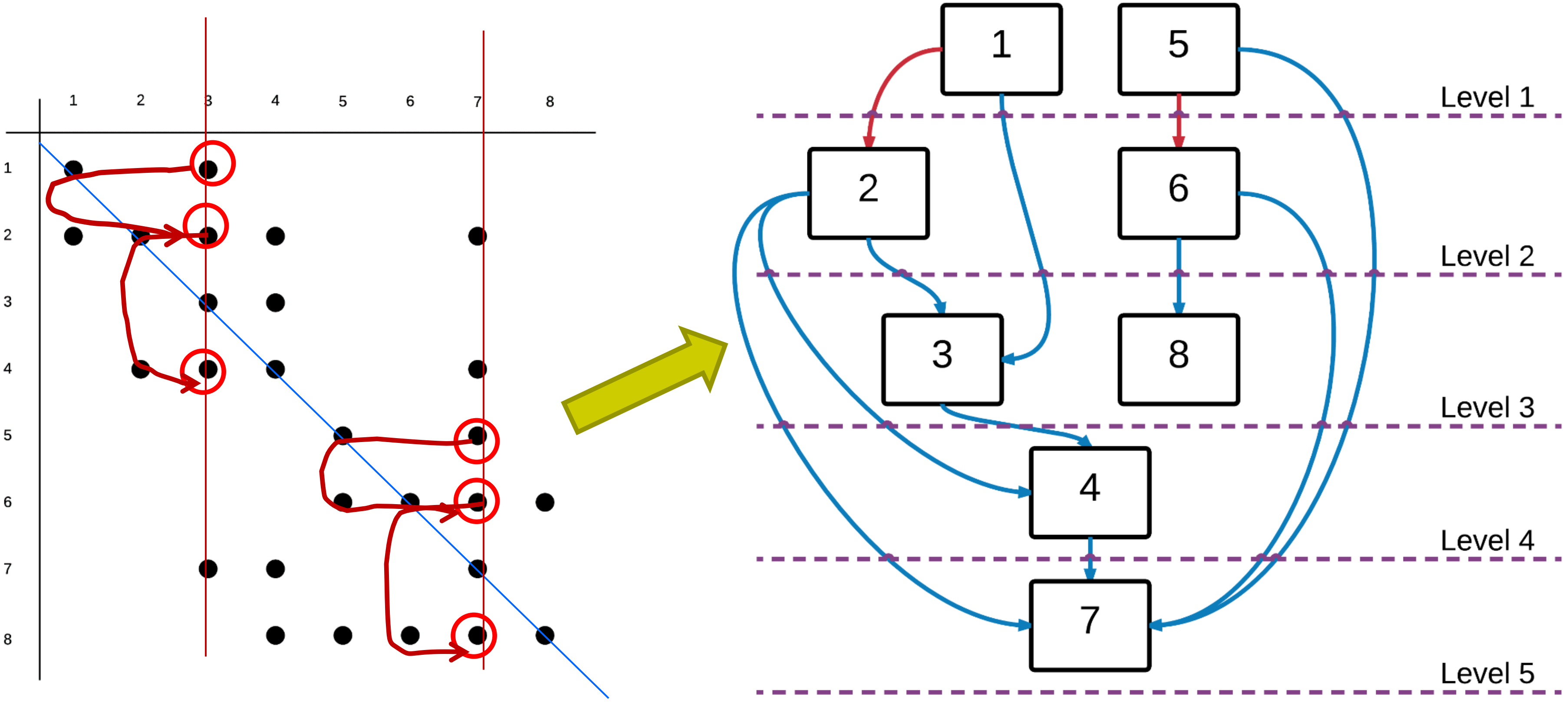
- After HSL_MC64 algorithm, most of elements on the diagonal have the largest absolute value (1 or -1)

Fill-in prediction and U-relationship



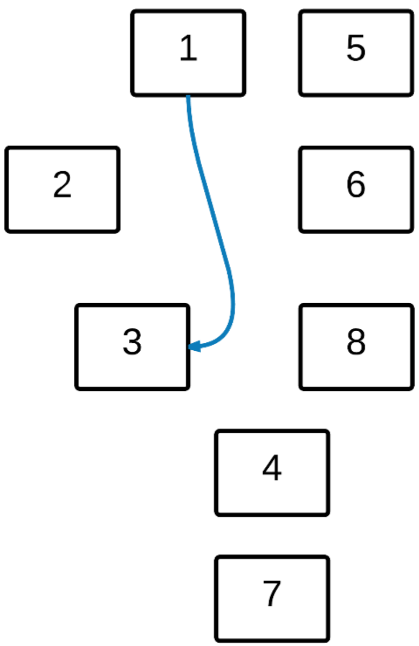
Dependency prediction

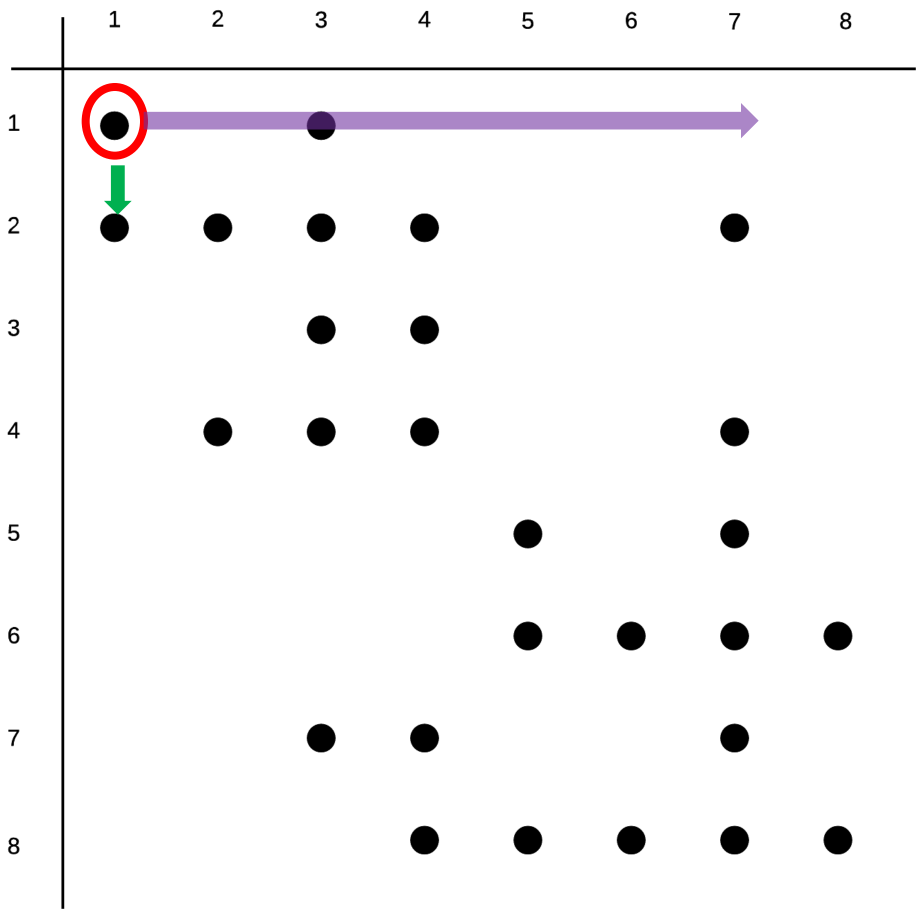
- “L dependency”
- “Double-U dependency”



Dependency prediction - "L dependency"

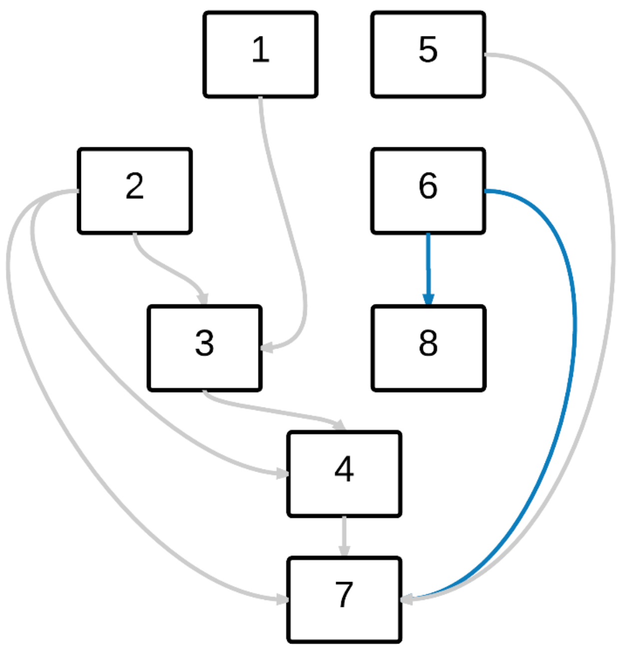
- Check column 1
  - Check L is non-empty?
    - If yes, do
      - Check U elements

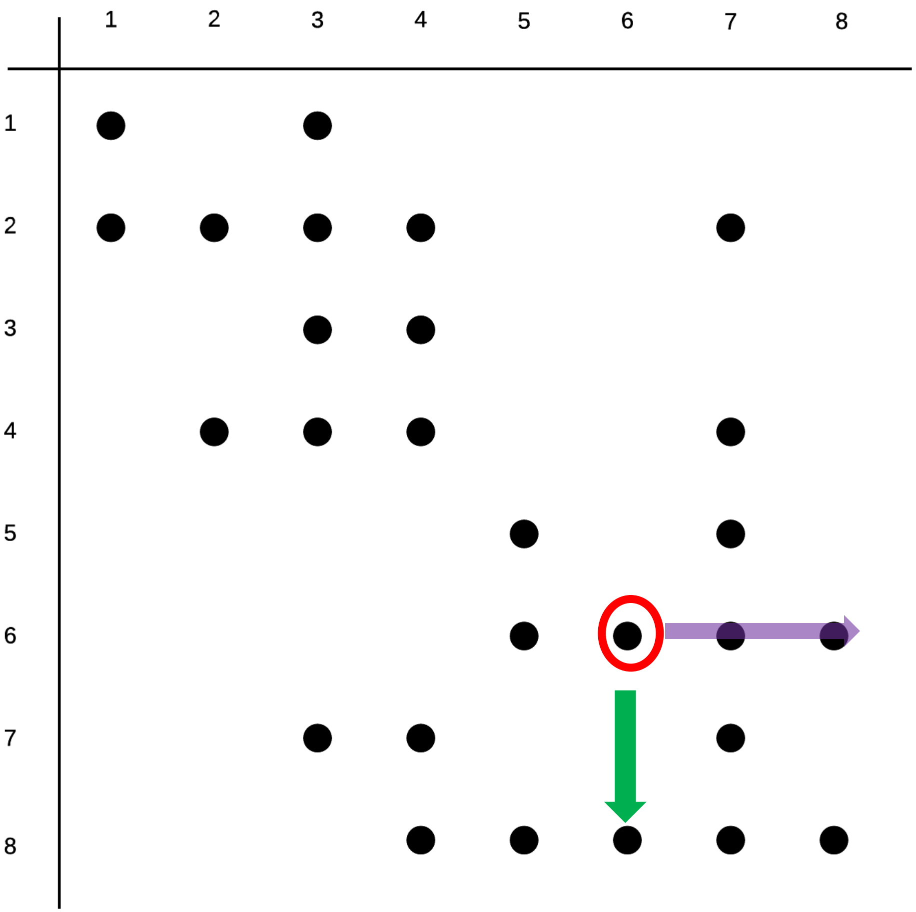




Dependency prediction - "L dependency"

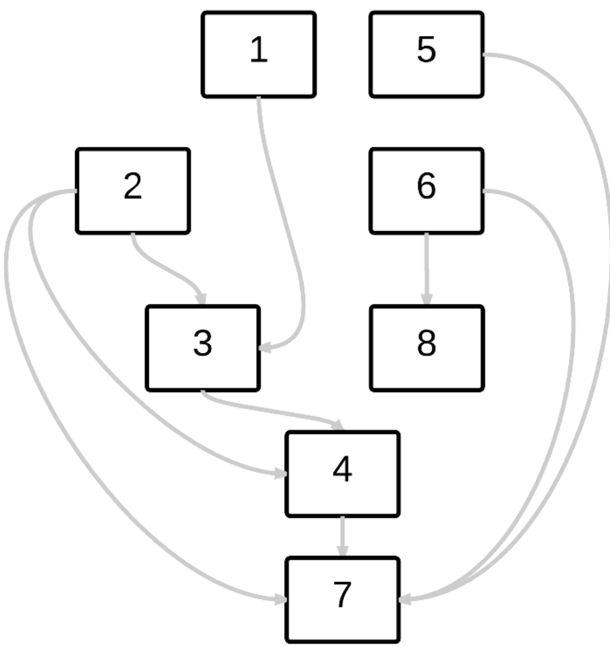
- Check column 6
  - Check L is non-empty?
    - If yes, do
      - Check U elements





Dependency prediction - “L dependency”

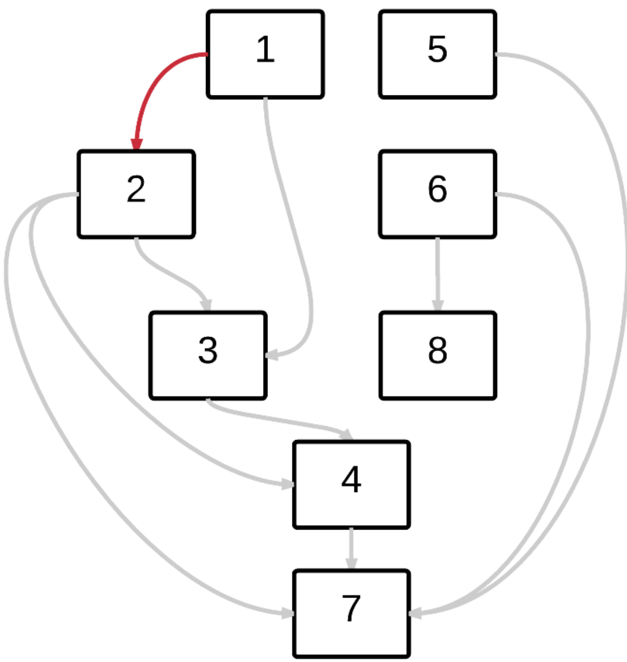
- Check column 1
  - Check L is non-empty?
    - If yes, do
      - Check U elements

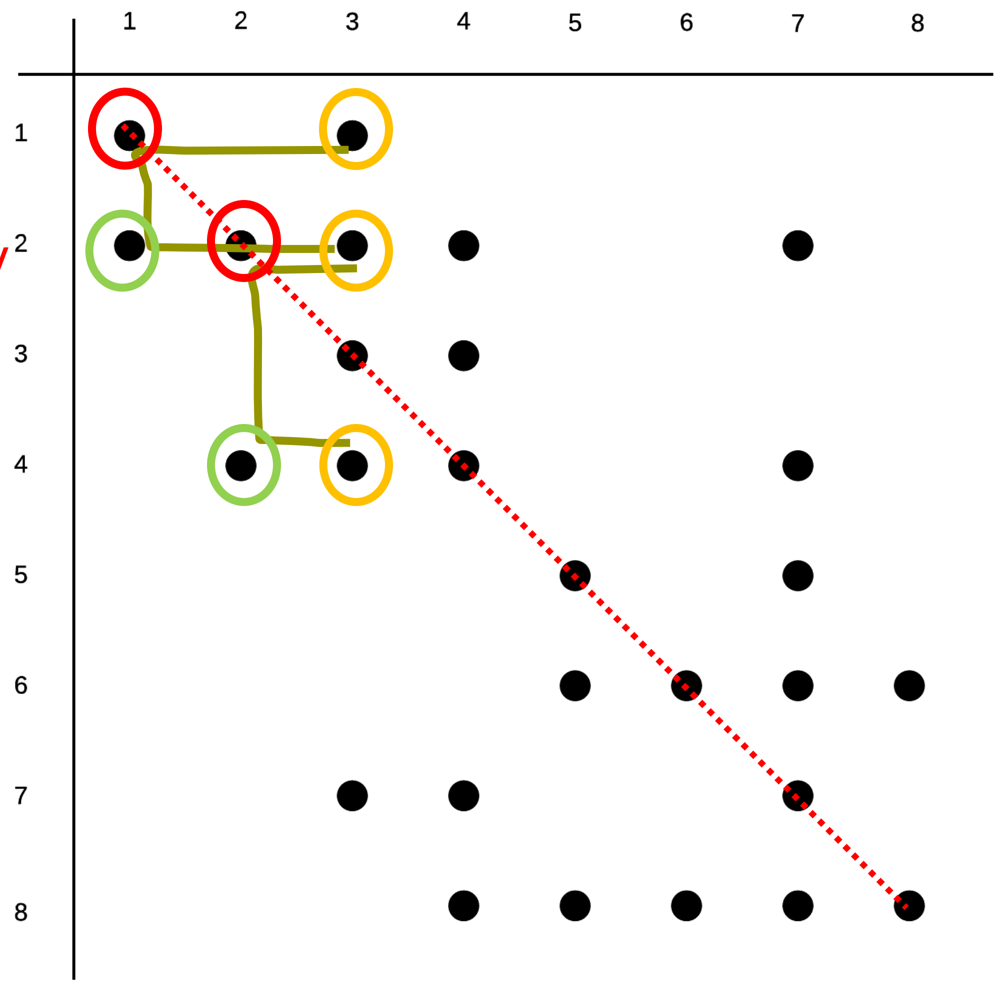


	1	2	3	4	5	6	7	8
1	●		●					
2	●	●	●	●			●	
3			●	●				
4		●	●	●			●	
5					●		●	
6					●	●	●	●
7			●	●			●	
8				●	●	●	●	●

Dependency prediction - “Double-U dependency”

- Check column 3
  - Two-U shapes?
    - If yes, do
      - Add another dependency





Another example from [Lee,TVLSI 18]

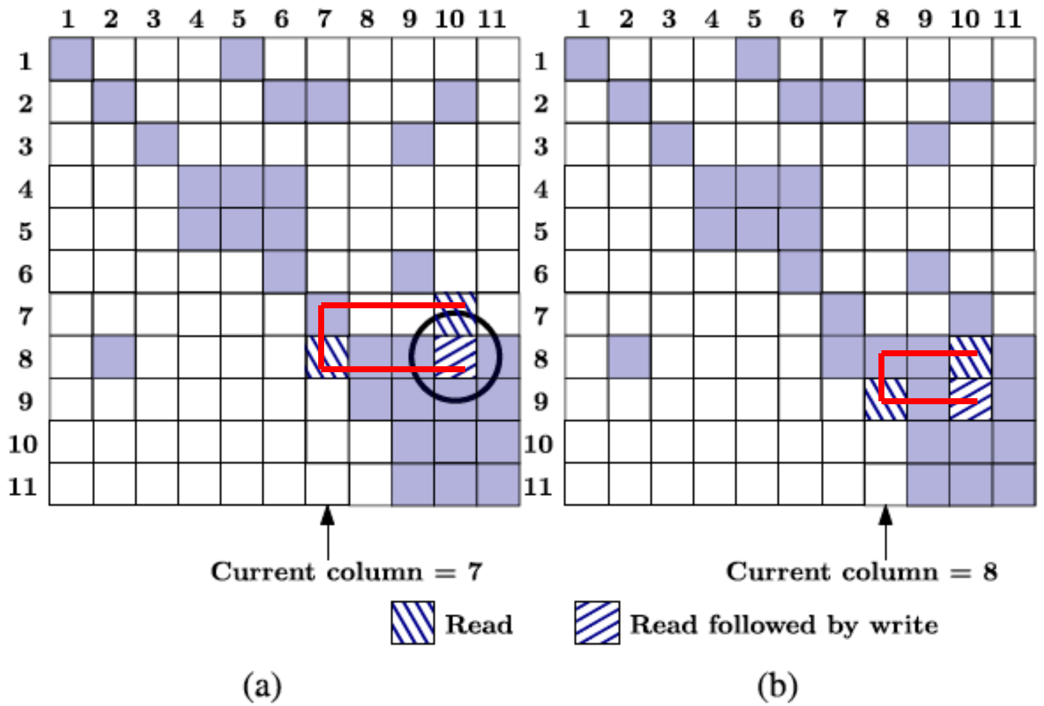
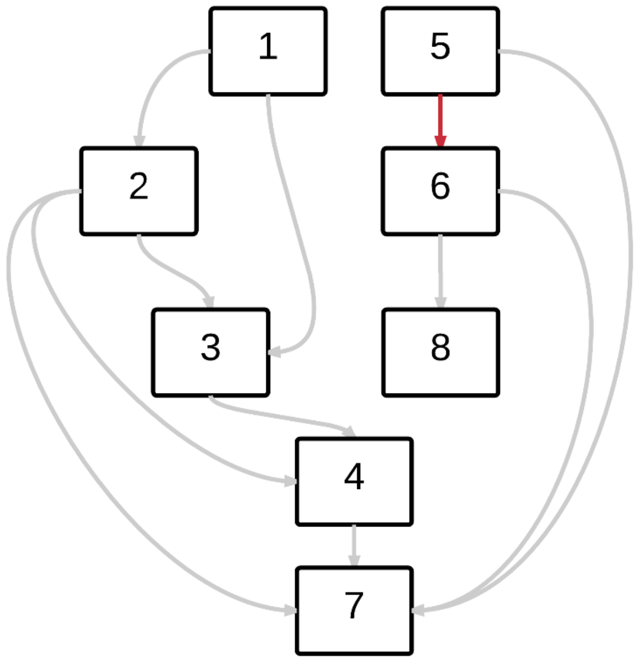


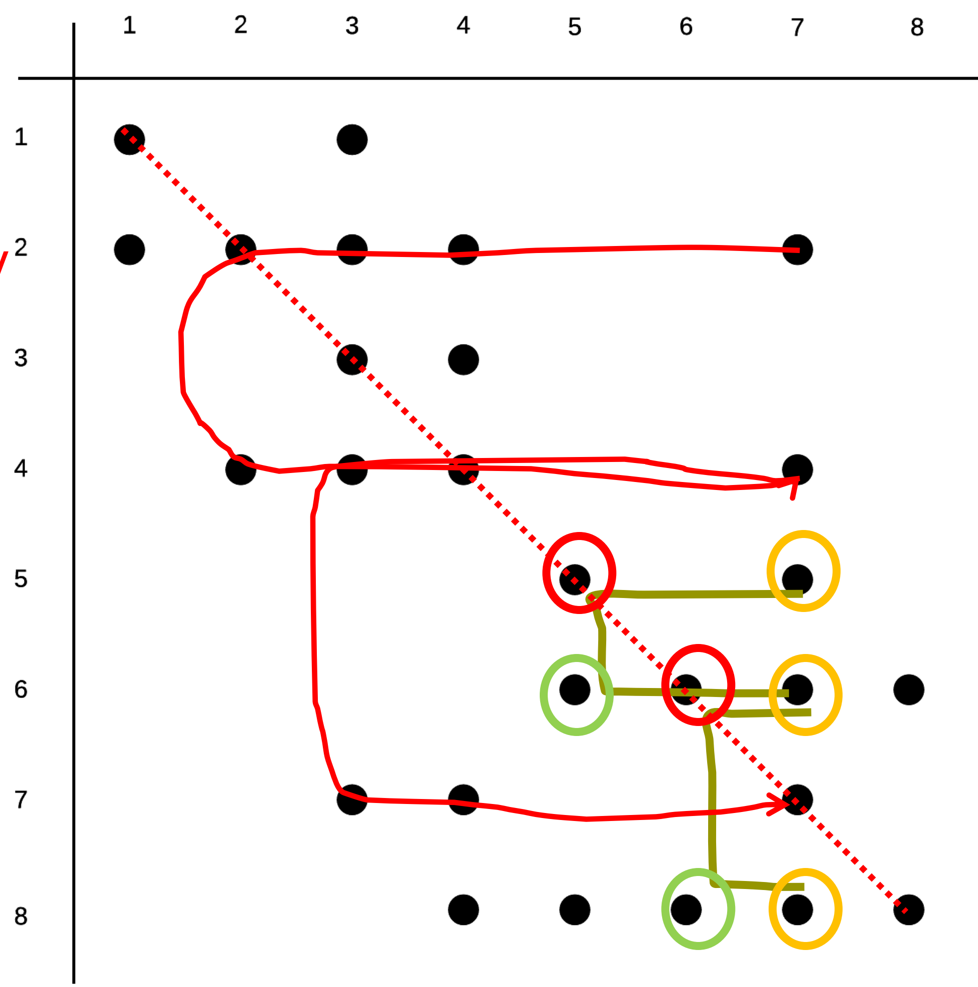
Fig. 6. Additional dependence for hybrid column-based RLA.

Wai-Kong Lee, Ramachandra Achar, and Michel S Nakhla, “Dynamic GPU parallel sparse LU factorization for fast circuit simulation”, *IEEE Transactions on Very Large Scale Integration* (VLSI) Systems, (99):1–12, 2018.

Dependency prediction - "Double-U dependency"

- Check column 3
  - Two-U shapes?
    - If yes, do
      - Add another dependency





“Double-U dependency” example

Serial implementation

$$\begin{bmatrix} 1 & & 1 \\ 1 & 1 & 1 \\ & 1 & 1 \end{bmatrix} \begin{array}{l} (1) \\ (2) \\ (3) \end{array}$$

1. -1*(1)+(2)

$$\begin{bmatrix} 1 & & 1 \\ 0 & 1 & 0 \\ & 1 & 1 \end{bmatrix} \begin{array}{l} (1) \\ (2) \\ (3) \end{array}$$

2. -1*(2)+(3)

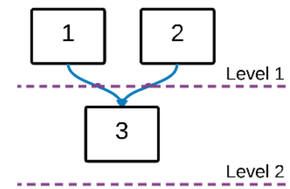
$$\begin{bmatrix} 1 & & 1 \\ 0 & 1 & 0 \\ & 0 & 1 \end{bmatrix} \begin{array}{l} (1) \\ (2) \\ (3) \end{array}$$

3. Get L and U

$$L = \begin{bmatrix} 1 & & \\ 1 & 1 & \\ & 1 & 1 \end{bmatrix}, U = \begin{bmatrix} 1 & & 1 \\ & 1 & \\ & & 1 \end{bmatrix}$$

GLU implementation only with “L dependency”

$$\begin{bmatrix} 1 & & 1 \\ 1 & 1 & 1 \\ & 1 & 1 \end{bmatrix} \begin{array}{l} (1) \\ (2) \\ (3) \end{array}$$



1. Level prediction:

Level 1: column 1 to column 2
Level 2: column 3

More dependency

Level 1: column 1
Level 2: column 2
Level 3: column 3

2. -1*(1)+(2) and -1*(2)+(3) <u>in parallel</u>

$$\begin{bmatrix} 1 & & 1 \\ 0 & 1 & 0 \\ & 0 & 0 \end{bmatrix} \begin{array}{l} (1) \\ (2) \\ (3) \end{array}$$

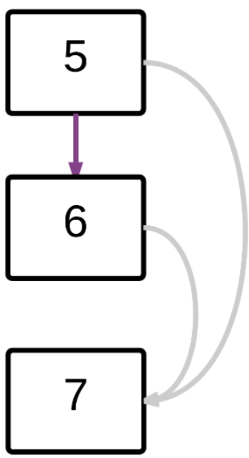
3. Get L and U

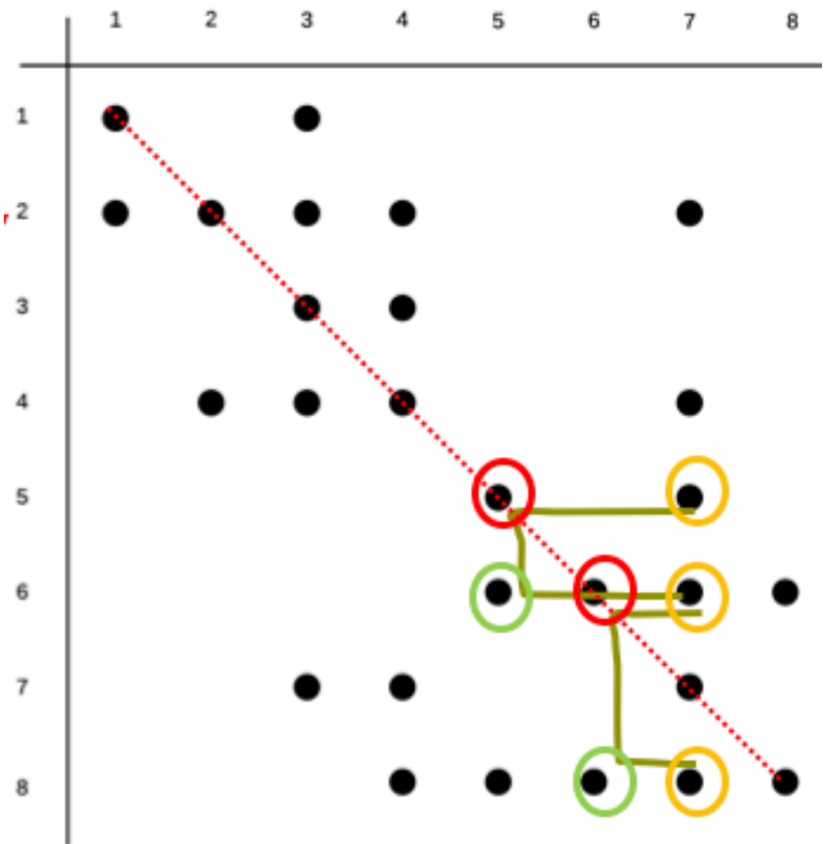
$$L = \begin{bmatrix} 1 & & \\ 1 & 1 & \\ & 1 & 1 \end{bmatrix}, U = \begin{bmatrix} 1 & & 1 \\ & 1 & \\ & & \end{bmatrix}$$

“Double-U dependency” pattern

- Check 3 elements in the column
- Check 2 elements in the diagonal
- Check 2 elements in the L matrix

✓ Add one more dependency

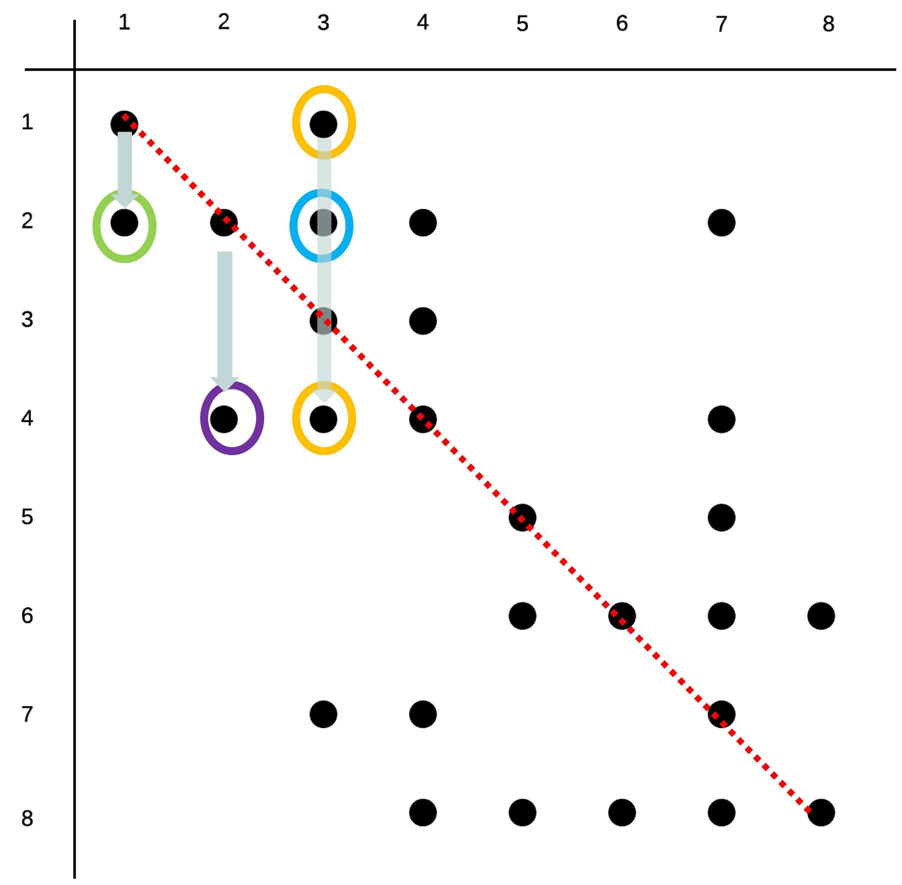




How to find "Double-U dependency"

- Find first non-empty L elements
- Find second non-empty L elements
- Find 2 elements in the same column

- Remember: we only have CSC and CSR for symbolic matrix -- time consuming with a lot of loops!!!

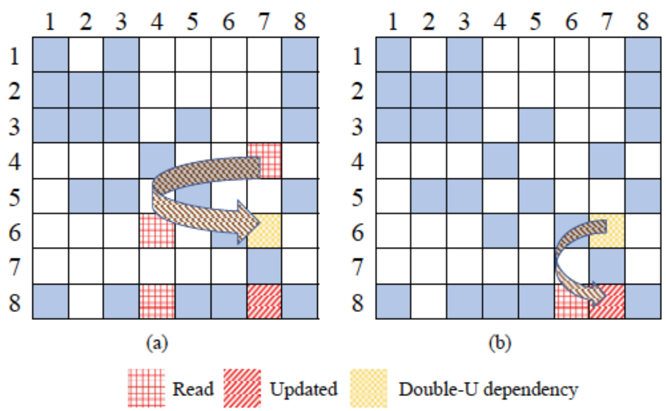


GLU 3.0 (2019)

Main improvements over GLU 2.0

- New double-U column dependency detection method (relaxed dependency detection)
  - Order of magnitude faster than the GLU 2.0
- New GPU kernel for submatrix update
  - Using dynamic resource allocation with three kernel computing modes.

Double-U dependency and detection in GLU 2.0



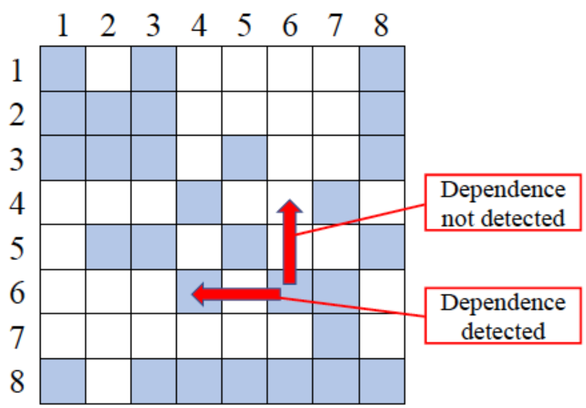
Element (6,7) has the double-U dependency

Algorithm 3 Read-write dependency detection algorithm used in GLU2.0

```
for i = 1 to n do
    Store all non-zero indices of row i in I_i
    for t = i to n where A_s(t, i) ≠ 0 do
        for j = t to n where A_s(j, t) ≠ 0 do
            Store all non-zero indices of row j in I_j
            if ∃k, k ∈ I_i, k ∈ I_j, k > t then
                Add i to t's dependency list
            end if
        end for
    end for
end for
```

The double-U dependency detection algorithm in GLU 2.0
It has three loops, as a result, it is very expensive search operation.

New column dependency detection algorithm in GLU 3.0



Comparison of left looking and up looking, left looking is able to detect double-U dependency.

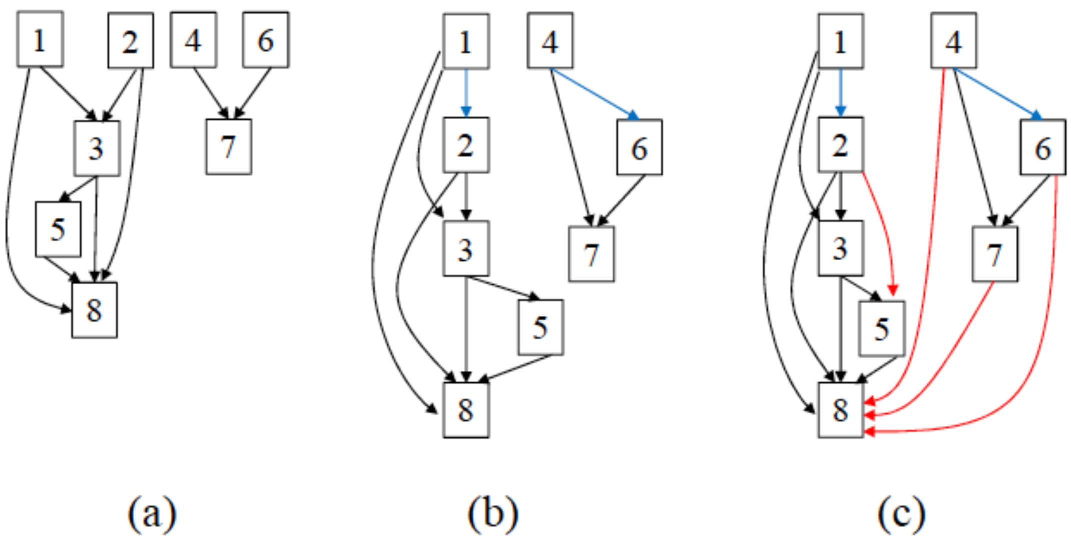
Key observation: So we can just look at U and L separately to determine the double-U dependency, which is the sufficient condition for U-dependency.

Algorithm 4 The proposed relaxed column dependency detection method

for $k = 1$ to n **do**
  /* Look up for all nonzeros in column k of U */
  for $i = 1$ to $k - 1$ where $A_s(i, k) \neq 0$ **do**
    if Column i of L is not empty **then**
      Add i to k's dependency list
    end if
  end for
  /* Look left for all nonzeros in row k of L */
  for $i = 1$ to $k - 1$ where $A_s(k, i) \neq 0$ **do**
    Add i to k's dependency list
  end for
end for

The proposed relaxed column dependency detection method in GLU 3.0.
Only two loops are used in the new algorithm

New column dependency detection algorithm in GLU 3.0



Dependency graph generated from 3 methods: (a) GLU1.0: incorrect result (b) GLU2.0: correct result (c) This work: the relaxed redundant dependency

Comparison results

- Numerical results and comparison with GLU 2.0 for the column dependency detection algorithms

TABLE II

EXPERIMENTAL RESULTS OF LEVELIZATION ON TEST MATRICES, WHERE *nz* STANDS FOR NUMBER OF NONZEROS BEFORE FILL-IN, AND *nnz* STANDS FOR NUMBER OF NONZEROS AFTER FILL-IN

Matrix	Number of levels		Levelization Time (ms)		
	GLU2.0	this work	GLU2.0	this work	speed-up
rajat12	37	39	3.048	0.035	87.1
circuit_2	101	102	17.187	0.074	232.3
memplus	147	147	345.568	0.234	1476.8
rajat27	123	125	272.216	0.32	850.7
onetone2	1213	1213	4009.51	1.589	2523.3
rajat15	968	968	3680.02	2.224	1654.7
rajat26	157	158	1703.92	0.711	2396.5
circuit_4	228	229	5053.39	0.944	5353.2
rajat20	1216	1219	15931.2	3.389	4700.9
ASIC_100ks	1626	1626	36388.8	5.301	6864.5
hcircuit	144	145	6122.57	1.206	5076.8
Raj1	1594	1595	56580.9	11.102	5096.5
ASIC_320ks	1669	1669	168979	8.573	19710.6
ASIC_680ks	1450	1450	530478	10.642	49847.6
G3_circuit	652	688	1741860	66.508	26190.2
				Arithmetic mean	8804.1
				Geometric mean	3145.8

GLU 3.0 algorithm revisited

For GLU 3.0, we use different loop indices than GLU 1.0/2.0.
So we present the whole algorithm here again

Algorithm 2 The hybrid column-based right-looking algorithm for GLU1.0/2.0

/* Scan each column from left to right */
for $j = 1$ to n **do**
    /*Compute column j of L matrix*/
    for $k = j + 1$ to n where $A_s(k, j) \neq 0$ **do**
        $A_s(k, j) = A_s(k, j)/A_s(j, j)$
    end for
    /*Update the submatrix for next iteration*/
    for $k = j + 1$ to n where $A_s(j, k) \neq 0$ **do**
        for $i = j + 1$ to n where $A_s(i, j) \neq 0$ **do**
            $A_s(i, k) = A_s(i, k) - A_s(i, j) * A_s(j, k)$
        end for
    end for
end for

The submatrix operation in matrix format in GLU in GLU 3.0

- Given j is the current column, the submatrix operation has two operations:

  (1) Two vector tensor production

  (2) Two matrix addition after (1)

Algorithm 5 The submatrix update in the GLU

```
/*Update the submatrix for next iteration*/
for k = j + 1 to n where A_s(j, k) ≠ 0 do
    for i = j + 1 to n where A_s(i, j) ≠ 0 do
        A_s(i, k) = A_s(i, k) − A_s(i, j) * A_s(j, k)
    end for
end for
```

$$A_{sub} = \begin{bmatrix} A_s(j+1, j+1) & \cdots & A_s(j+1, n) \\ \vdots & \ddots & \vdots \\ A_s(n, j+1) & \cdots & A_s(n, n) \end{bmatrix}$$

$$A_{sub} \leftarrow A_{sub} - \begin{bmatrix} A_s(j+1, j) \\ \vdots \\ A_s(n, j) \end{bmatrix} \cdot [A_s(j, j+1), \cdots, A_s(j, n)]$$

The size of the matrix is NxN, where N = n-k, the size of the two vectors is Nx1, and 1xN. Both two vectors and NxN matrix are sparse matrices. As a result, we can easily parallelize the vector and matrix operations

Submatrix update, subcolumn update

$$A_{sub} - \begin{bmatrix} A_s(j+1,j) \\ \vdots \\ A_s(n,j) \end{bmatrix} \cdot [A_s(j,j+1),\cdots,A_s(j,n)]$$

The submatrix update is done in the column-wise way:

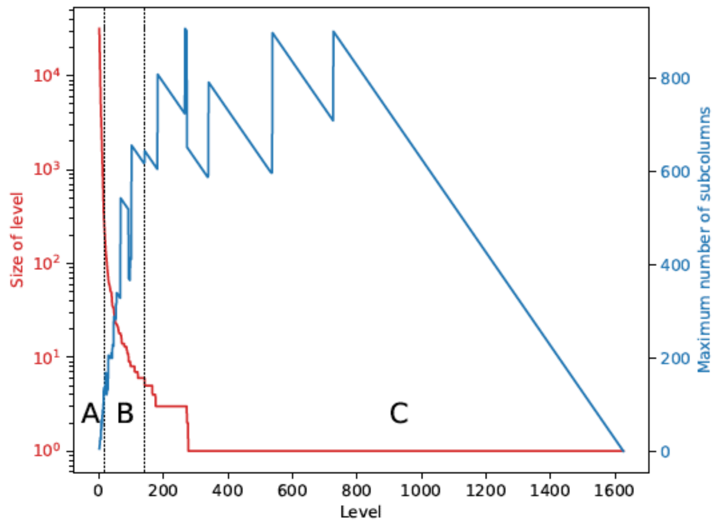
$$\vec{A}_s(j+1:n,i) - \vec{A}_s(j+1:n,j) \cdot A_s(j,i),$$
$$\text{for } i = [j+1,\cdots,n]$$

Where

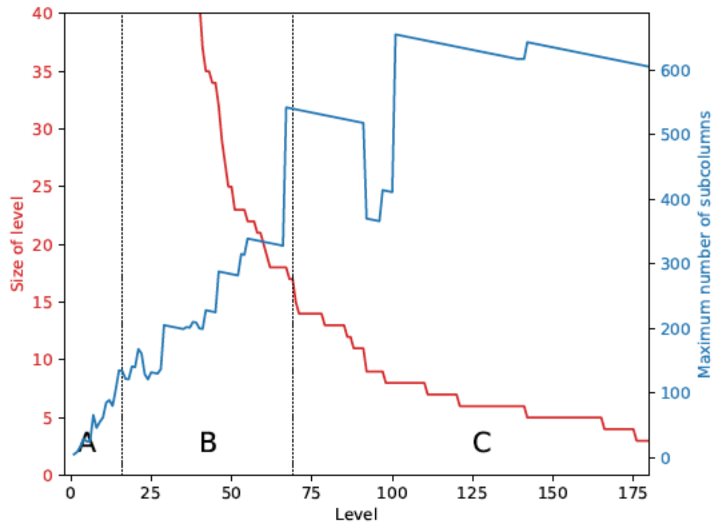
$$\vec{A}_s(j+1:n,i) = [A_s(j+1,i),\ldots,A_s(n,i)]^T$$
$$\vec{A}_s(j+1:n,j) = [A_s(j+1,j),\ldots,A_s(n,j)]^T.$$

The submatrix update consists of vector operations or *subcolumn update*. Each time, we can update *one subcolumn i .* This can be parallelized in GPU where each resulting element can be computed by one thread, (multiply-accumulate (MAC) operation). There are two levels of parallelism: namely (a) the vector operations (or subcolumn updates) for different vectors and (b) element-wise MAC operations in each vector or subcolumn.

Subcolumn count versus level



(a) Level versus its size and the maximum number of subcolumns



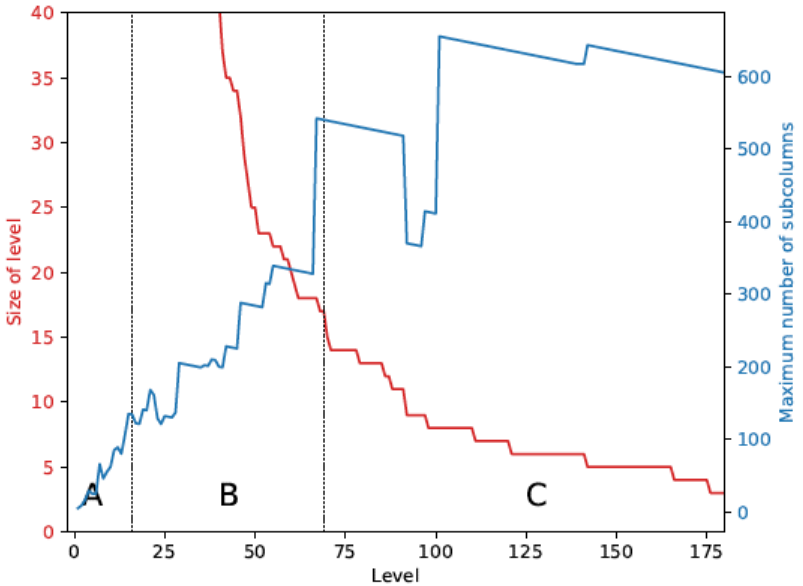
(b) Zoomed in view

Number of columns and subcolumns across different levels. Maximum number of subcolumns is used for each level (ASIC100ks).

Key observation: The number of columns and associated subcolumns are inversely correlated as a function of levels.

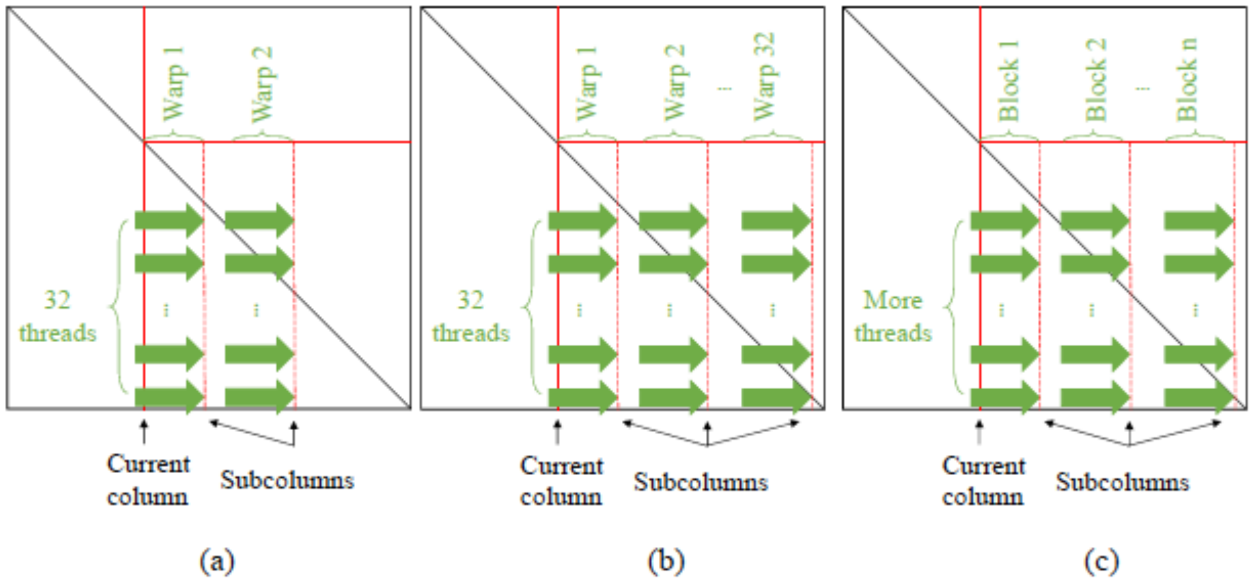
New GLU GPU kernel

- Three kernel computing modes
  - Small block mode (A mode):
    - One warp for each sub-column
    - A few warps are assigned to a block
    - More blocks are assigned
  - Large block mode (B mode):
    - Still one warp for each sub-column
    - 32 warps assigned into a block
    - Less blocks are assigned
  - Stream mode (C mode):
    - One block is assigned to a sub-column
    - One kernel launch for each kernel
    - Multiple kernel calls (stream) is multiple columns



(b) Zoomed in view

New GLU GPU kernel



Comparison of the concurrency layout for one column in different kernels:
(a) Small block mode (b) Large block mode (c) Stream mode

Numerical results and comparison

TABLE I
GPU KERNEL RUNTIMES OF GLU3.0 VS PREVIOUS WORKS

Matrix	Number of rows	nz	nnz	GPU time (ms)			
				GLU2.0 [21]	GLU3.0 (this work)	speed-up over [21]	speed-up over [22]
rajat12	1879	12926	13948	2.44883	2.237	1.1	1.0
circuit_2	4510	21199	32671	8.36301	4.144	2.0	1.9
memplus	17758	126150	126152	6.90432	6.672	1.0	0.9
rajat27	20640	99777	143438	23.8673	10.539	2.3	2.0
onetone2	36057	227628	1306245	550.598	60.964	9.0	8.3
rajat15	37261	443573	1697198	458.611	71.135	6.4	6.1
rajat26	51032	249302	343497	104.12	32.366	3.2	4.2
circuit_4	80209	307604	438628	394.995	68.944	5.7	9.1
rajat20	86916	605045	2204552	2538.24	241.822	10.5	8.8
ASIC_100ks	99190	578890	3638758	2652.79	215.493	12.3	14.1
hcircuit	105676	513072	630666	243.846	46.996	5.2	9.5
Raj1	263743	1302464	7287722	7969.05	845.189	9.4	8.7
ASIC_320ks	321671	1827807	4838825	5632.8	216.517	26.0	21.3
ASIC_680ks	682712	2329176	4957172	11771.7	210.697	55.9	18.4
G3_circuit	1585478	4623152	36699336	38780.9	878.153	44.2	8.2
					Arithmetic mean	13.0	7.1
					Geometric mean	6.7	4.8

GLU3.0 achieve 13.0X (arithmetic mean) and 6.7X (geometric mean) speedup over GLU 2.0 and 7.1X (arithmetic mean) and 4.8X (geometric mean) over recent proposed enhanced GLU2.0 sparse LU solver on the same set of circuit matrices.

Summary

- We proposed a new version of GPU-based sparse LU factorization solver, GLU 3.0
- GLU 3.0 feature two major improvement over GLU 2.0/1.0
  - More efficient column dependency detection algorithm
  - More efficient GPU kernel for GLU solver
- GLU 3.0 archive 6.7X over GLU 2.0 and 4.8X over recently proposed enhanced GLU 2.0 (Lee’s work).