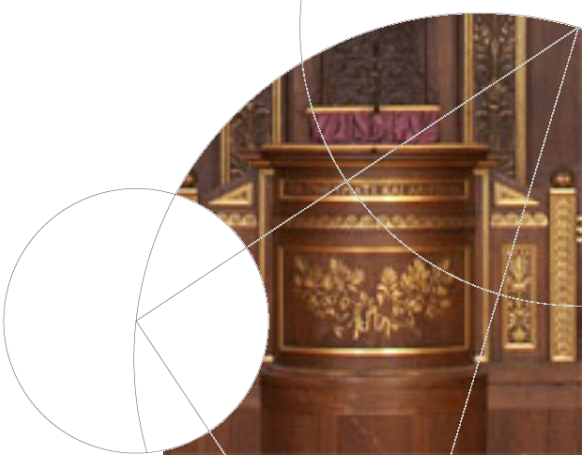
A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves

Weifeng Liu (University of Copenhagen, Denmark, &
STFC Rutherford Appleton Laboratory, UK)
Ang Li (Eindhoven University of Technology, Netherlands)
Jonathan D. Hogg (STFC Rutherford Appleton Laboratory, UK)
Iain S. Duff (STFC Rutherford Appleton Laboratory, UK)
Brian Vinter (University of Copenhagen, Denmark)



Euro-Par `16, Grenoble, France, 25 August 2016

Sparse Matrix
&
Sparse Triangular Solve (SpTRSV)

Sparse Matrix (in the CSR format)

- If the majority of entries in a matrix are zeros, we can store the matrix using a sparse representation that only records nonzero entries.
- The **CSR** format stores a matrix **row**-wise

0	0	1	0
2	3	0	0
0	0	0	0
4	0	5	6

A
(4x4)
dense

		1	
2	3		
4		5	6

A
(4x4)
nnzA = 6

row pointer =	0	1	3	3	6	
column index =	2	0	1	0	2	3
value =	1	2	3	4	5	6

A (in the CSR format)
(4x4)
nnzA = 6

Sparse Matrix (in the CSC format)

- If the majority of entries in a matrix are zeros, we can store the matrix using a sparse representation that only records nonzero entries.
- The **CSC** format stores a matrix **column**-wise.

0	0	1	0
2	3	0	0
0	0	0	0
4	0	5	6

A (4x4) *dense*

		1	
2	3		
4		5	6

A (4x4) *nnzA* = 6

column pointer = 0 2 3 5 6

row index = 1 3 1 0 3 3

value = 2 4 3 1 5 6

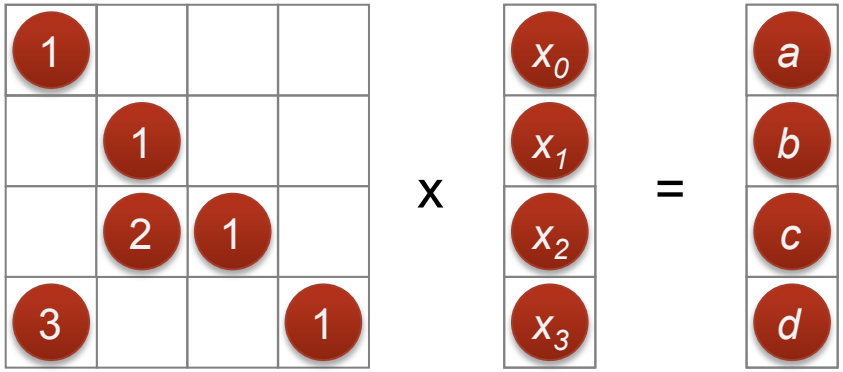
A (in the CSC format) (4x4) *nnzA* = 6

Hao Wang, Weifeng Liu, Kaixi Hou, and Wu-Chun Feng. **Parallel Transposition of Sparse Data Structures**. *ACM International Conference on Supercomputing '16*.

Sparse Triangular Solve

- Example: $Lx = b$

  Compute a dense solution vector x from the sparse linear system, where L is a square lower triangular sparse matrix, and b is a dense vector.



L	x	b
(4x4)	(4x1)	(4x1)
nnzL = 6	*dense*	*dense*
known	*unknown*	*known*

system:

$$\begin{aligned} 1*x_0 &= a \\ 1*x_1 &= b \\ 2*x_1+1*x_2 &= c \\ 3*x_0+1*x_3 &= d \end{aligned}$$

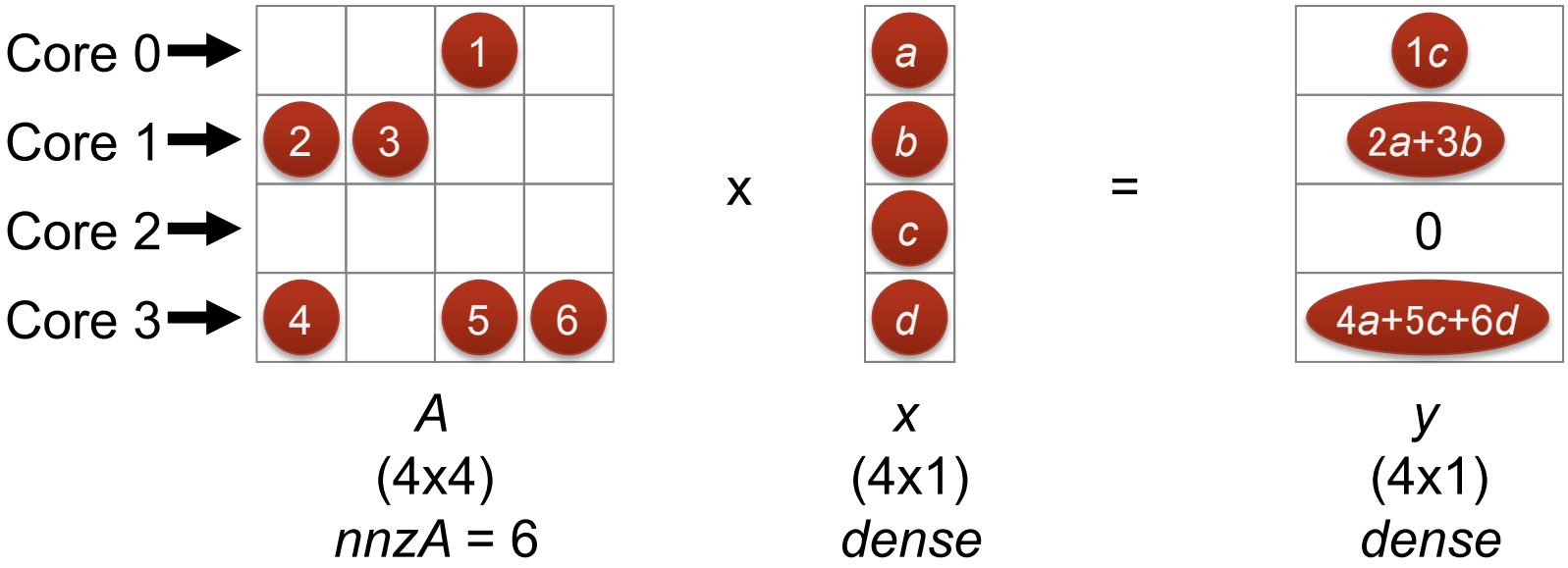
solution:

$$\begin{aligned} x_0 &= a \\ x_1 &= b \\ x_2 &= c - 2b \\ x_3 &= d - 3a \end{aligned}$$

Why parallelizing SpTRSV is not trivial?

Sparse Matrix-Vector Multiplication

- Multiply a sparse matrix *A* by a dense vector *x*, obtain a dense vector *y*.

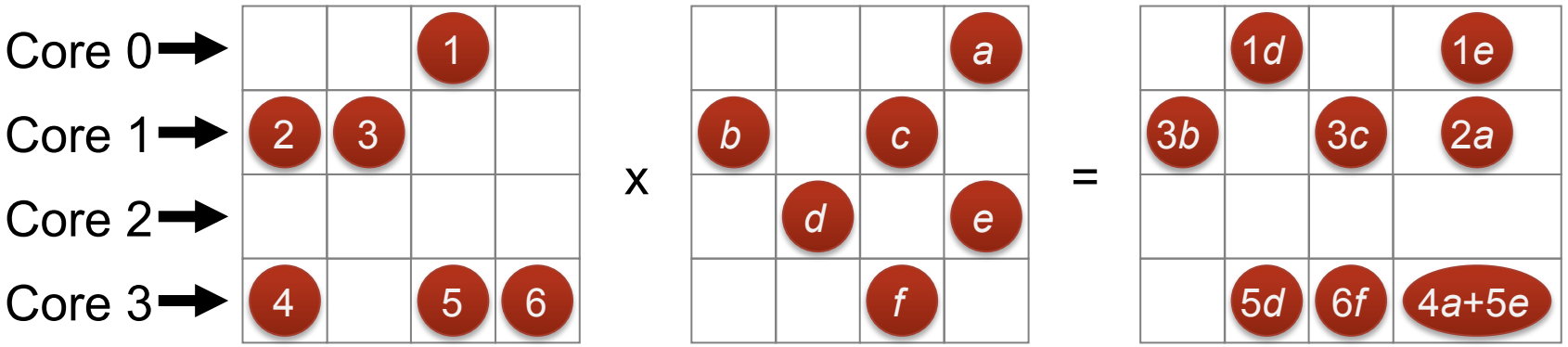


Weifeng Liu and Brian Vinter. **CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication**. *ACM International Conference on Supercomputing '15*.

Sparse Matrix-Matrix Multiplication

- Example: $C = AB$

Multiply a sparse matrix A by a sparse matrix B, obtain a sparse matrix C.



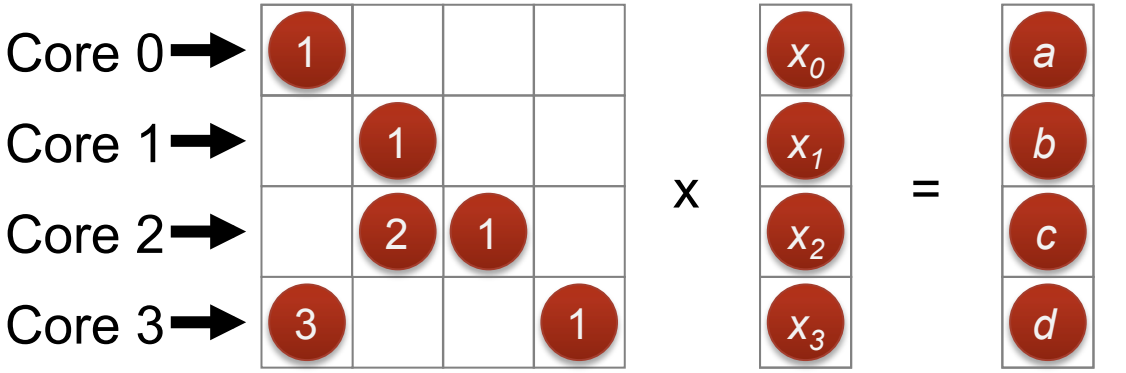
A	B	C
(4x4)	(4x4)	(4x4)
$nnzA = 6$	$nnzB = 6$	$nnzC = 8$

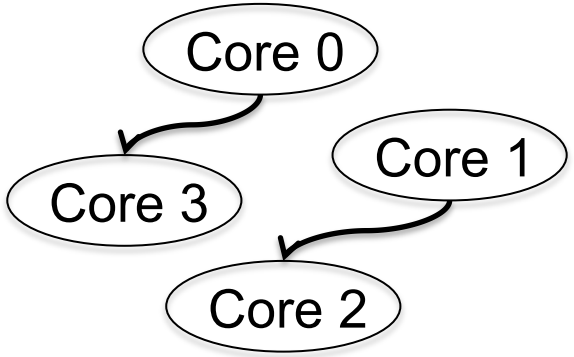
Weifeng Liu and Brian Vinter. **A Framework for General Sparse Matrix -Matrix Multiplication on GPUs and Heterogeneous Processors**. *Journal of Parallel and Distributed Computing '15*.

Sparse Triangular Solve

- Example: $Lx = b$

SpTRSV is inherently sequential. In this case, Core 2 has to wait for Core 1 to finish, and Core 3 has to wait for Core 0, meaning that the all four cores cannot work in parallel.





L	*x*	*b*
(4x4)	(4x1)	(4x1)
nnzL = 6	*dense*	*dense*
known	*unknown*	*known*

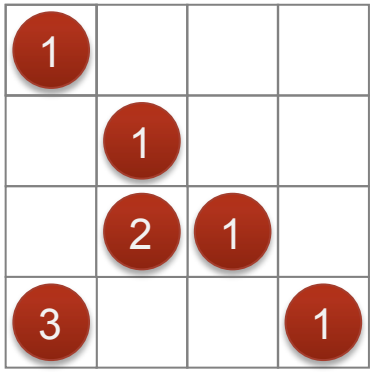
$$1*x_0 = a$$
$$1*x_1 = b$$
$$2*x_1+1*x_2 = c$$
$$3*x_0+1*x_3 = d$$

The level-set method (related work)

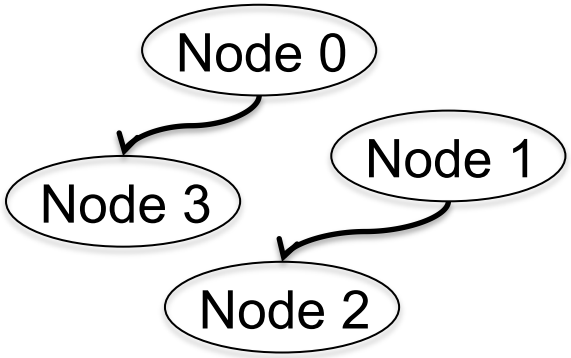
Anderson and Saad [1] and Saltz [23]

Level-set method for parallel SpTRSV

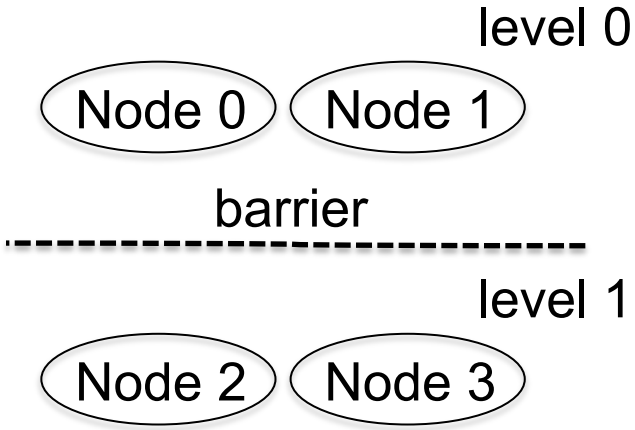
- The level-set method has two phases: (1) grouping nodes (rows or columns) that can be consumed in parallel, and (2) solving nodes group by group with barriers between.



matrix form



graph form

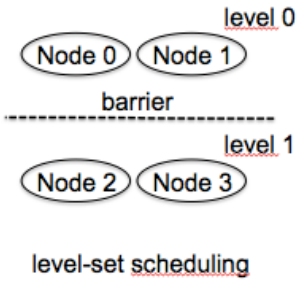


level-set scheduling

Level-set method for parallel SpTRSV

- However, the level-set method has high overhead for preprocessing (grouping nodes) and explicit barrier synchronization at runtime.

Matrix name	Preprocessing cost (ms)	SpTRSV cost (ms)	SpTRSV cost breakdown (ms)		#Level-sets
			Synchronization	Compute	
FEM/ship_003	92.46	12.95	10.96	1.99	4367
FEM/Cantilever	47.89	9.60	5.62	3.98	2397
chipcool0	8.74	1.99	1.15	0.84	534
nlpkkt160	484.67	38.30	0.01	38.29	2



Preprocessing overhead is equal to cost for a few SpTRSV operations

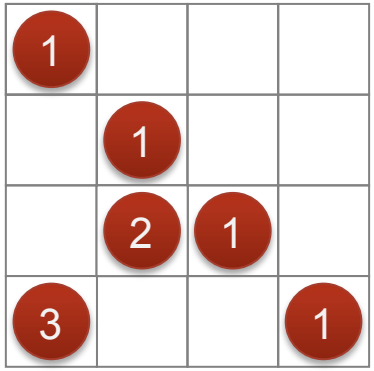
Synchronization dominates SpTRSV when #level-sets is high

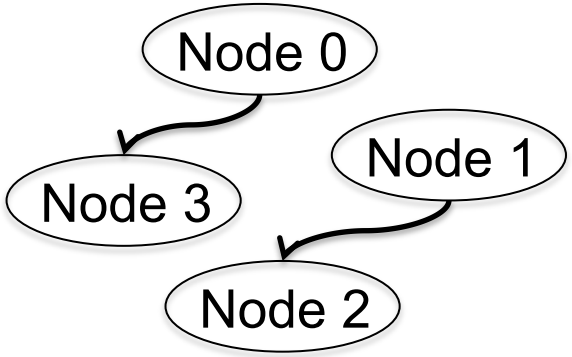
Synchonization-Free SpTRSV (this work)

Basic idea: in-degree and spin-lock

- All cores (one for each column as a node) are activiated: some of them compute solution, the others busy-wait for spin-locks unlocked to start.

$$
\begin{cases}
1*x_0 = a \\
1*x_1 = b \\
2*x_1+1*x_2 = c \\
3*x_0+1*x_3 = d
\end{cases}
$$





matrix form

graph form

CSR row_ptr:	0	1	2	4	6

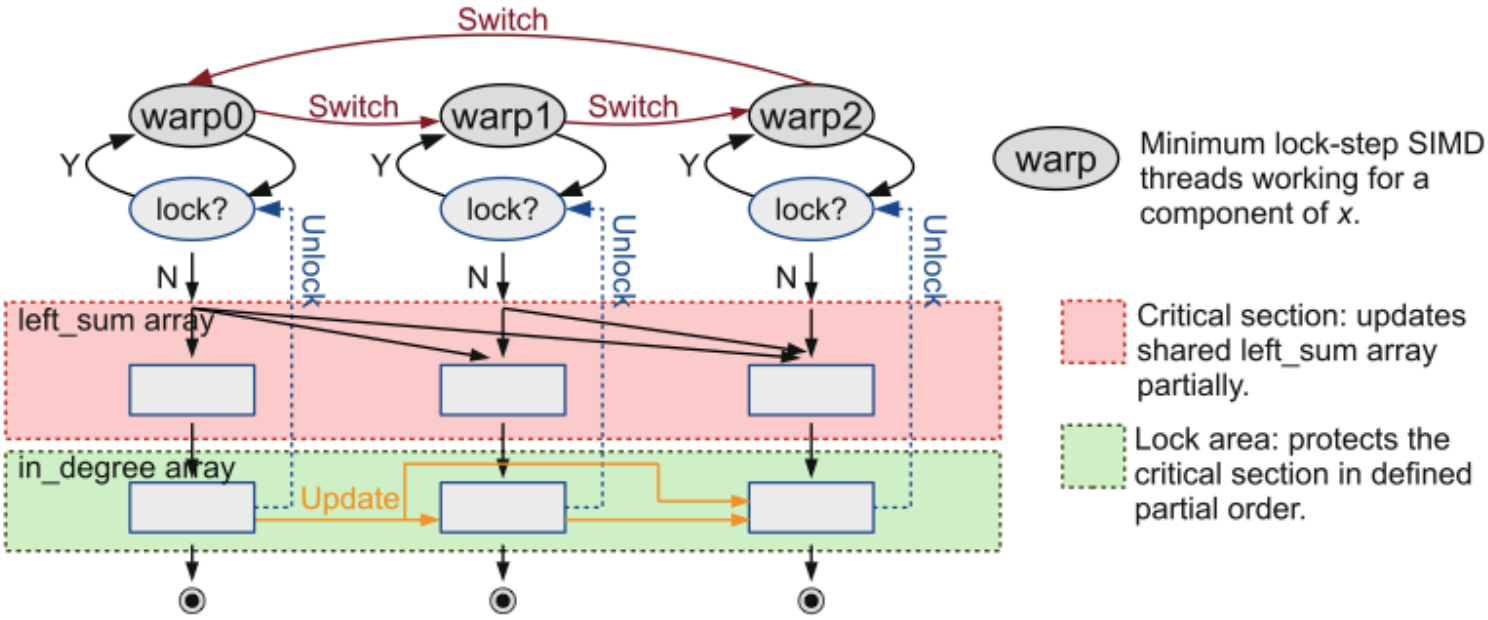
In-degree:	1	1	2	2

CSC column_ptr:	0	2	4	5	6

Out-degree:	2	2	1	1

Basic idea: warp/wavefront switch

- #components of the solution may be much larger than #warps/wavefronts running concurrently. So warps /wavefronts are required to be issued in the ascending order to avoid dead-lock.

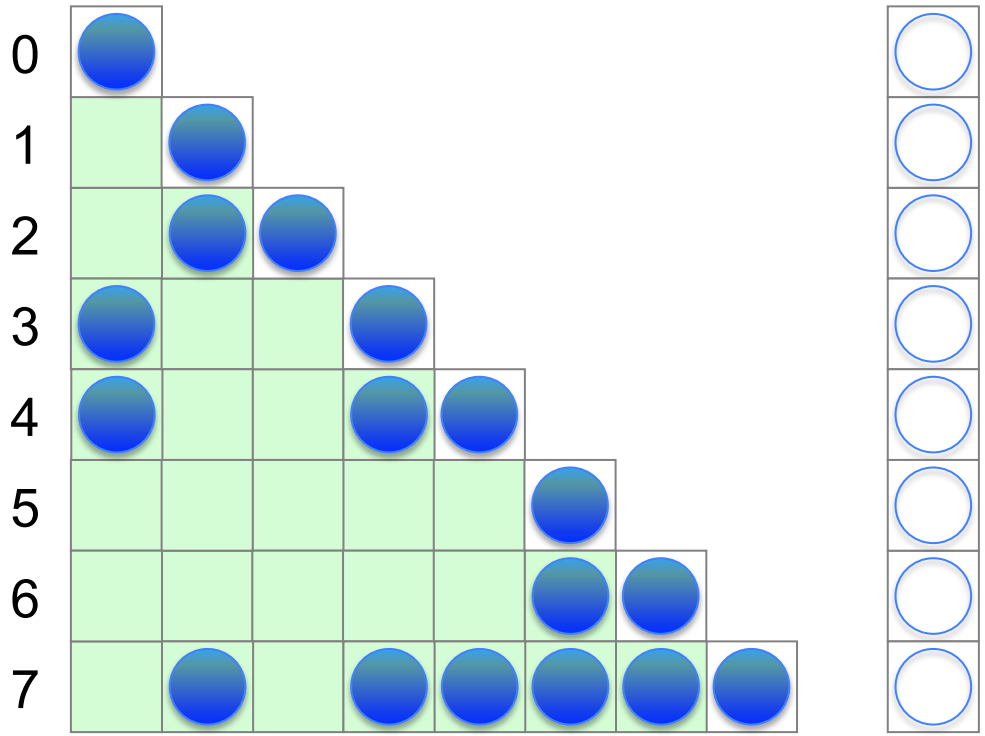


The proposed sync-free algorithm

Algorithm 2. The proposed synchronization-free SpTRSV algorithm.

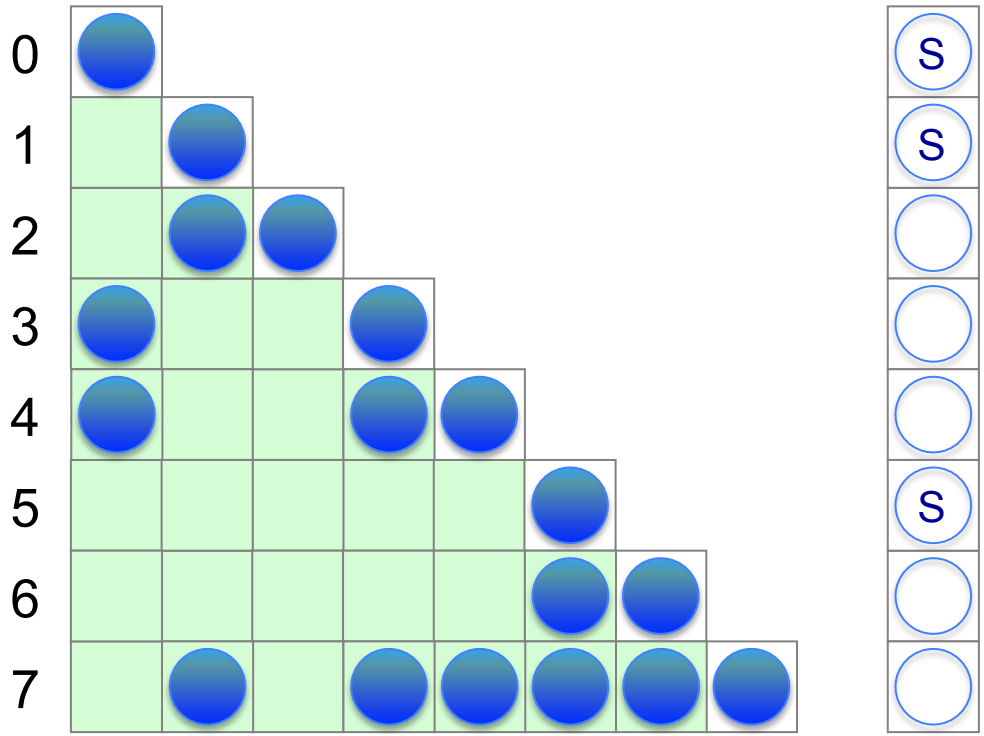
```
1: MALLOC(*d_left_sum, *s_left_sum, *d_in_degree, *s_in_degree, n)
2: MEMSET(*d_left_sum, *s_left_sum, *d_in_degree, *s_in_degree, 0)
3: function PREPROCESSING-STAGE()
4:     for i = 0 to nnz − 1 in parallel do
5:         ATOMIC-ADD(&d_in_degree[row_idx[i]], 1)
6:     end for
7: end function
8: function SOLVING-STAGE()
9:     th ← SET()                                   ▷ size of diagonal block
10:    for i = 0 to n − 1 in parallel do            ▷ One concurrent warp for one component.
11:        while s_in_degree[i]+1 ≠ d_in_degree[i] do
12:            //busy wait
13:        end while
14:        x[i] ← (b[i]-d_left_sum[i]-s_left_sum[i])/val[col_ptr[i]]
15:        for j = col_ptr[i]+1 to col_ptr[i + 1]−1 in parallel do ▷ One thread for one nonzero.
16:            rid ← row_idx[j]
17:            if rid < i + th − i%th then          ▷ Use on-chip scratchpad for red areas in Figure 3.
18:                ATOMIC-ADD(&s_left_sum[rid], val[j] × x[i])
19:                ATOMIC-ADD(&s_in_degree[rid], 1)
20:            else                                 ▷ Use off-chip memory for green area in Figure 3.
21:                ATOMIC-ADD(&d_left_sum[rid], val[j] × x[i])
22:                ATOMIC-SUB(&d_in_degree[rid], 1)
23:            end if
24:        end for
25:    end for
26: end function
27: FREE(*d_left_sum, *s_left_sum, *d_in_degree, *s_in_degree)
```

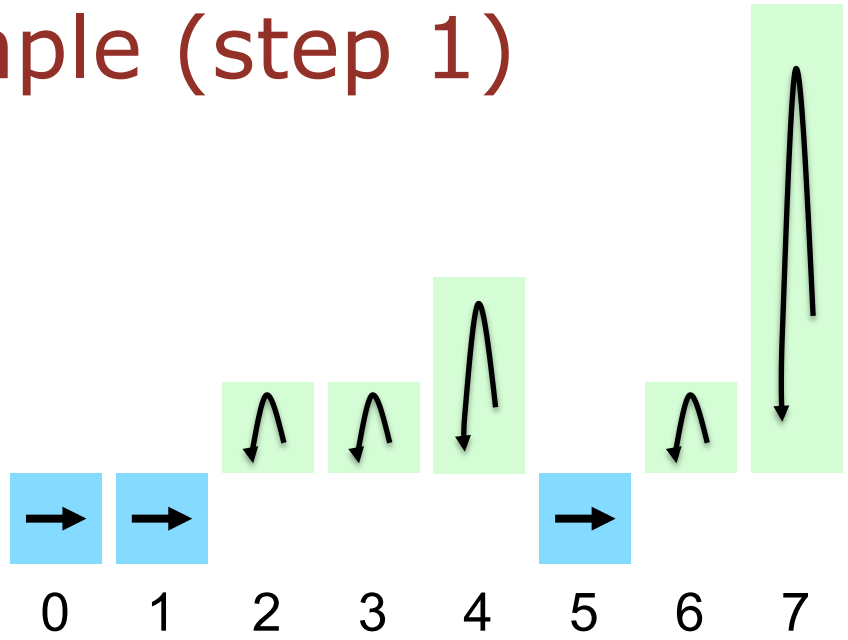
Cartoon of an example (init)



in-degree:	1	1	2	2	3	1	2	6

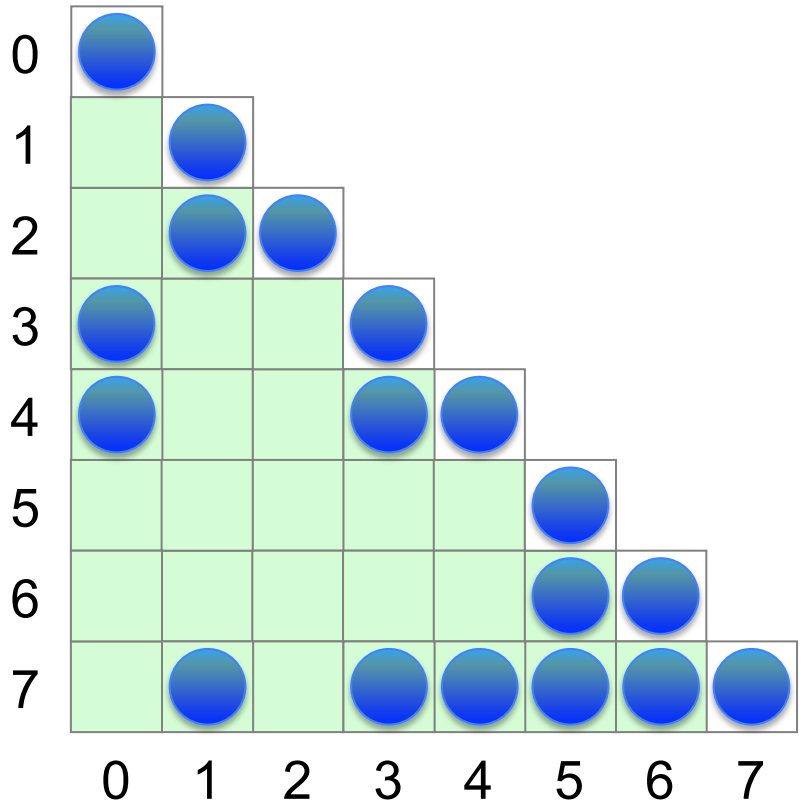
Cartoon of an example (step 1)



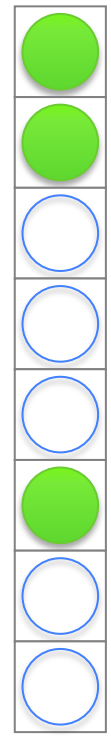


in-degree:	1	1	2	2	3	1	2	6

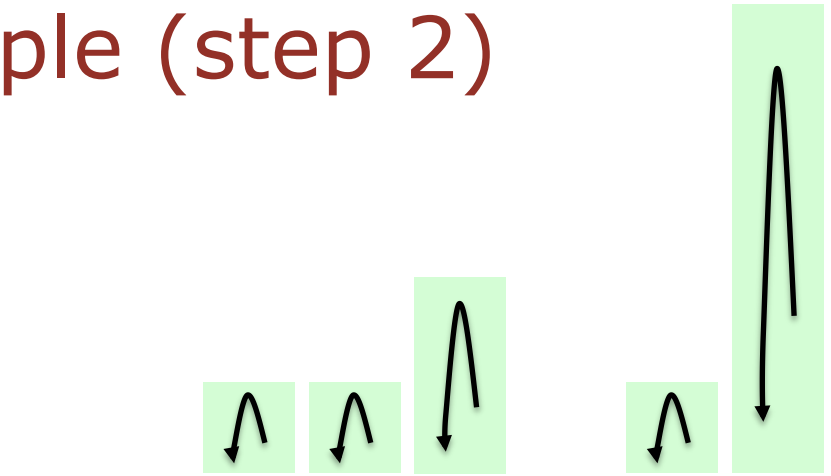
Cartoon of an example (step 2)



matrix



solution

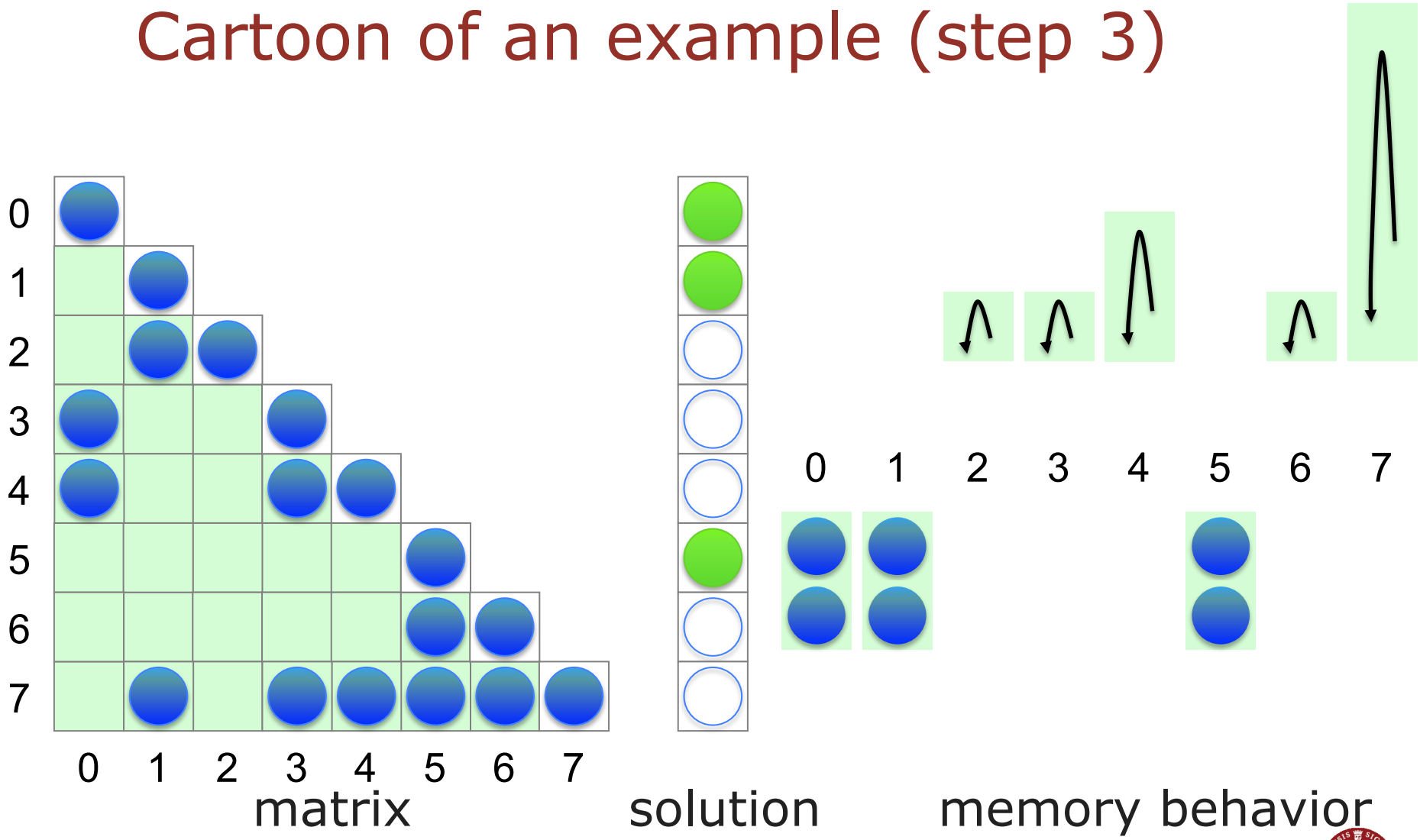


memory behavior

in-degree:

1	1	2	2	3	1	2	6

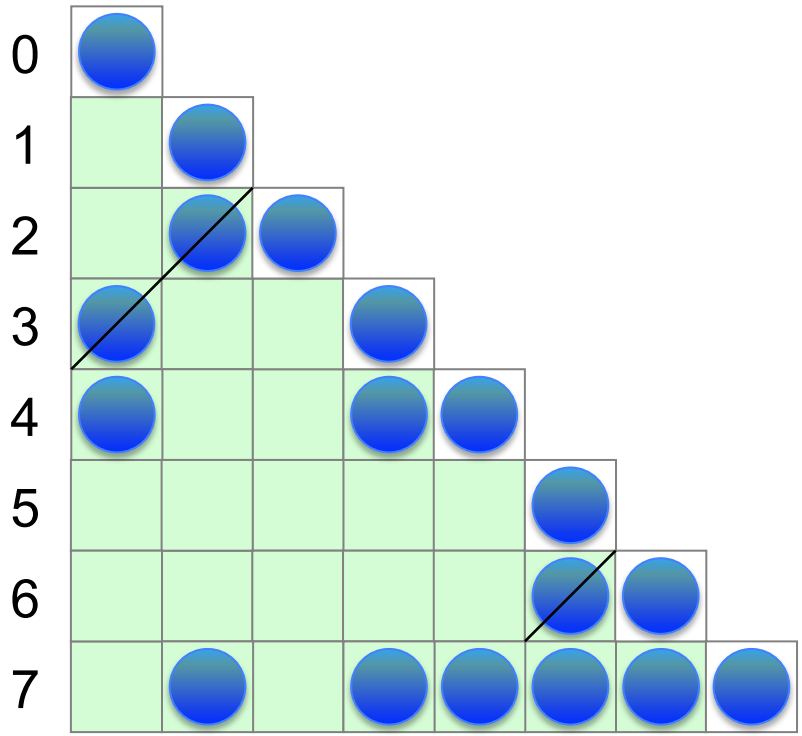
Cartoon of an example (step 3)

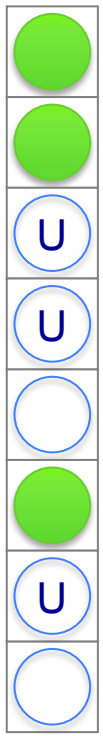


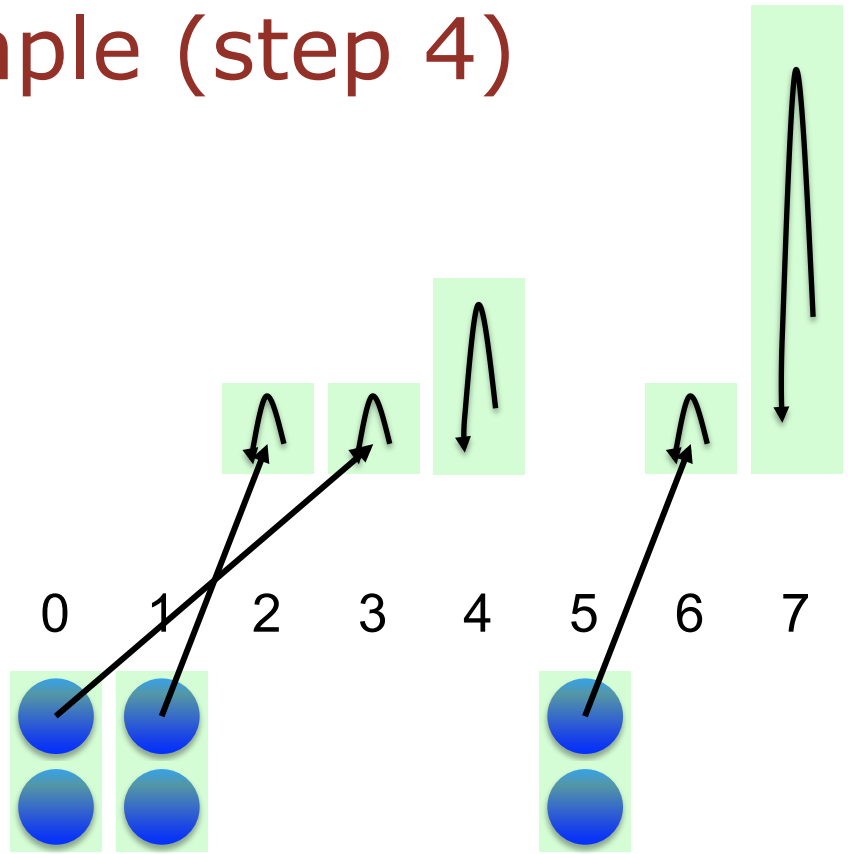
in-degree:

1	1	2	2	3	1	2	6

Cartoon of an example (step 4)



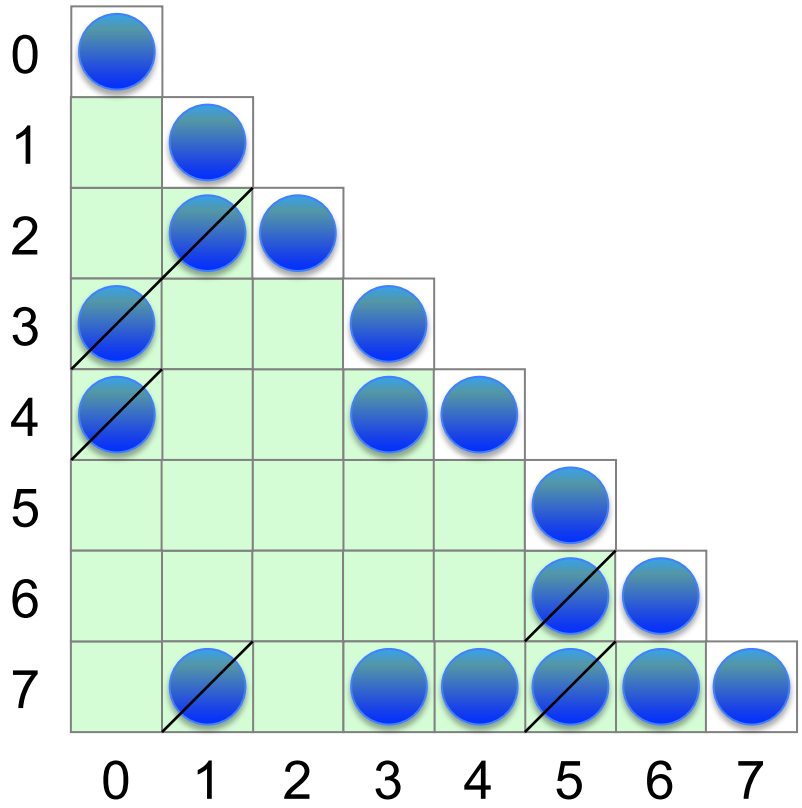


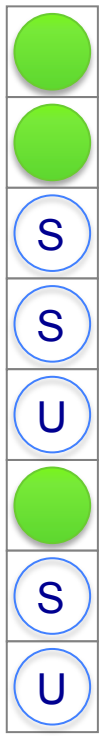


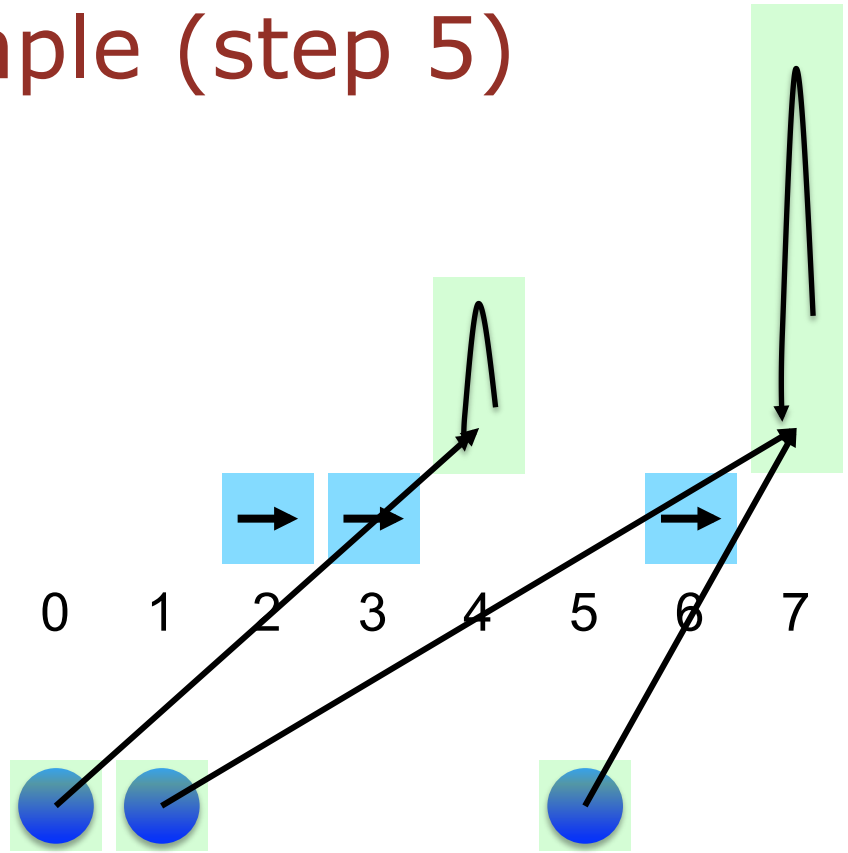
in-degree:

1	1	1	1	3	1	1	6

Cartoon of an example (step 5)



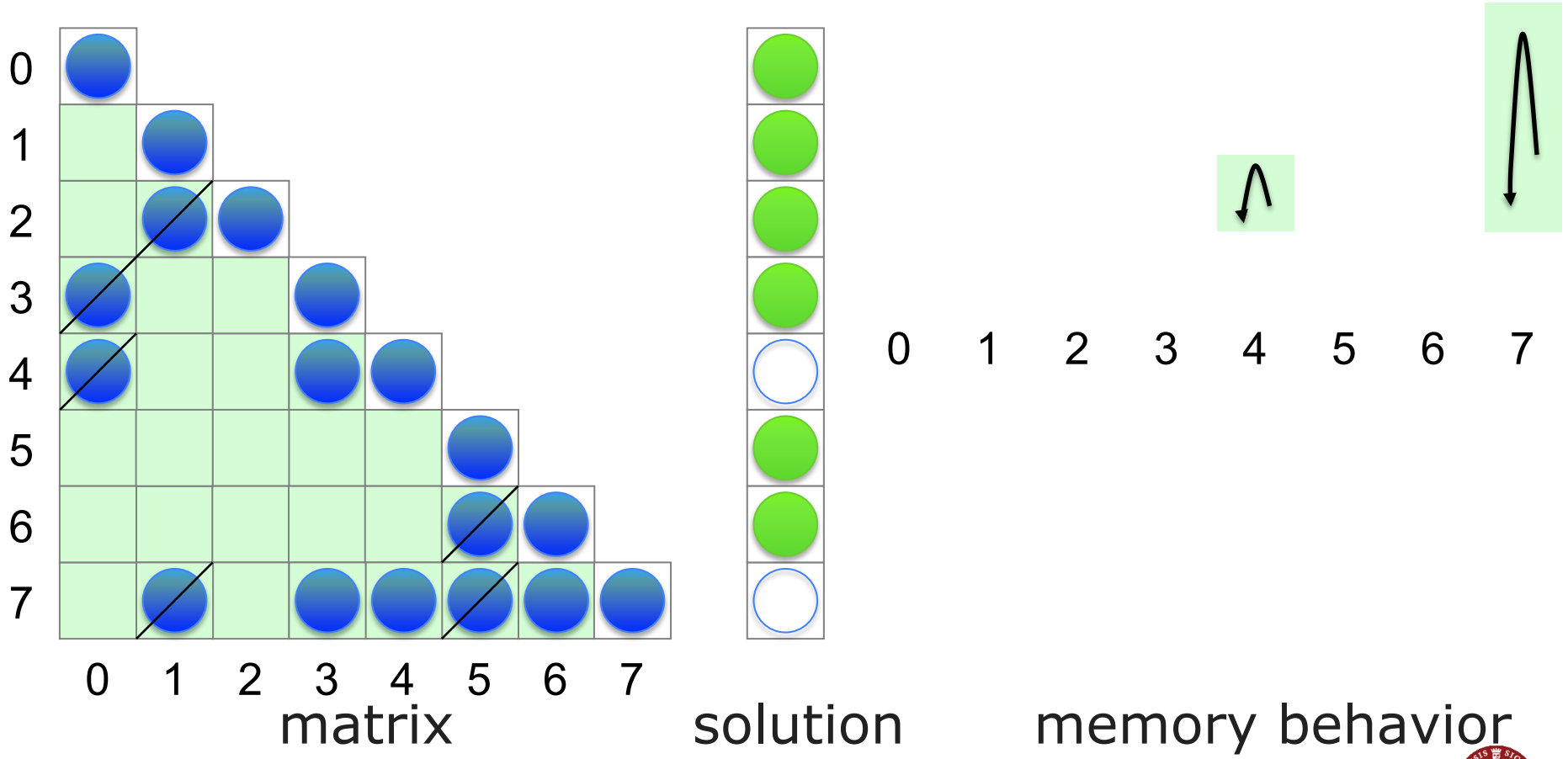




in-degree:

1	1	1	1	2	1	1	4

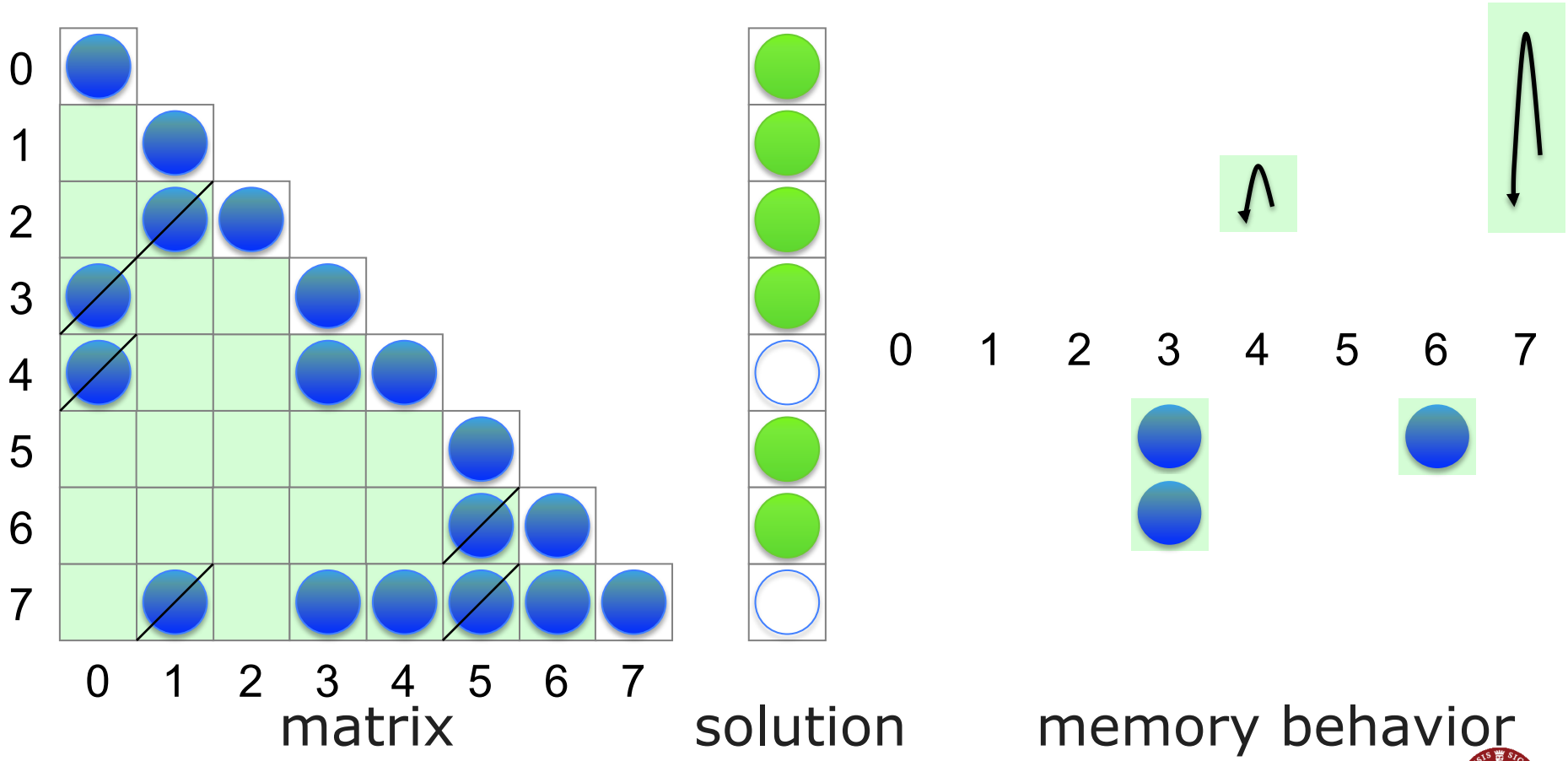
Cartoon of an example (step 6)



in-degree:

1	1	1	1	2	1	1	4

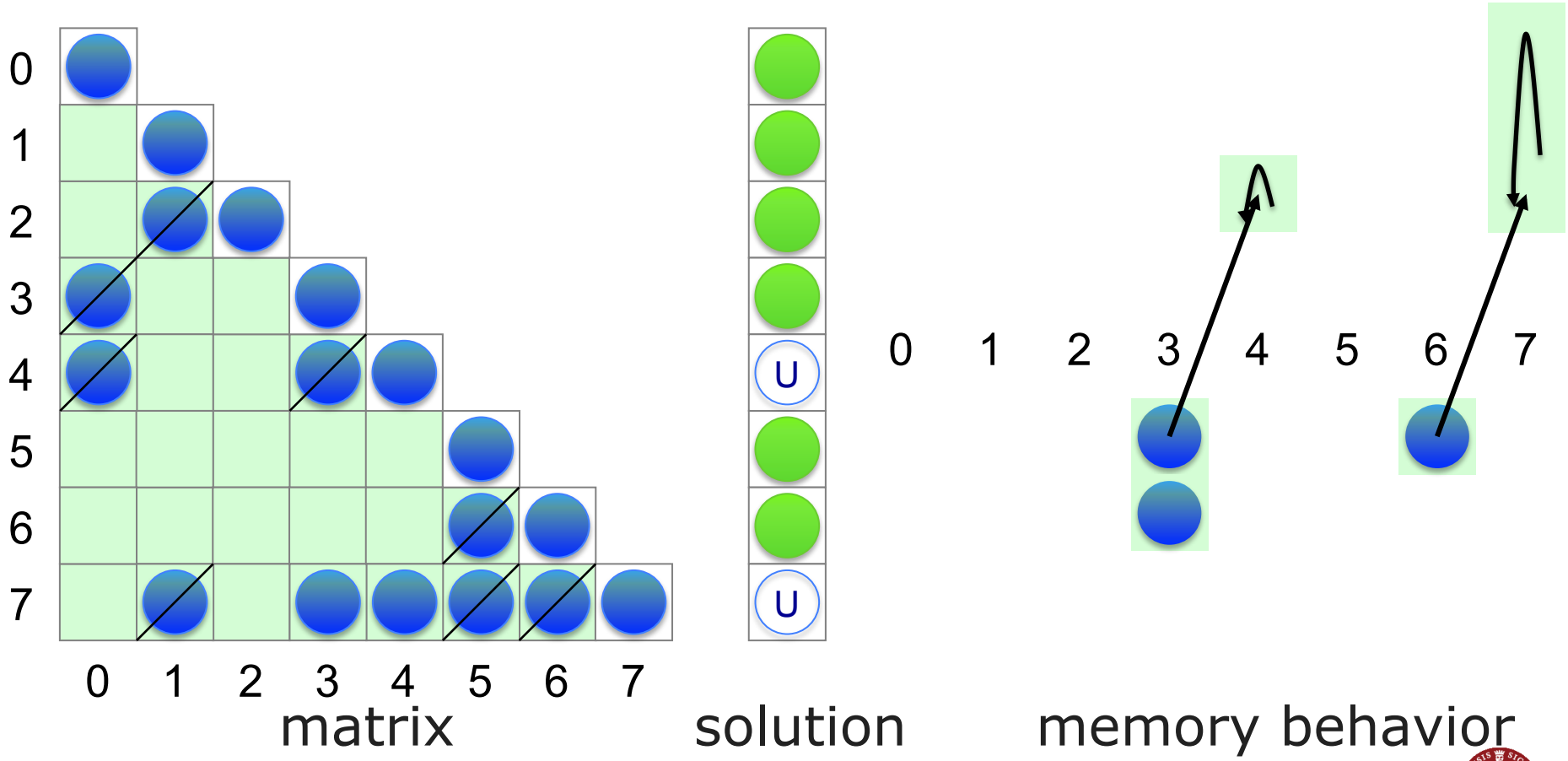
Cartoon of an example (step 7)



in-degree:

1	1	1	1	2	1	1	4

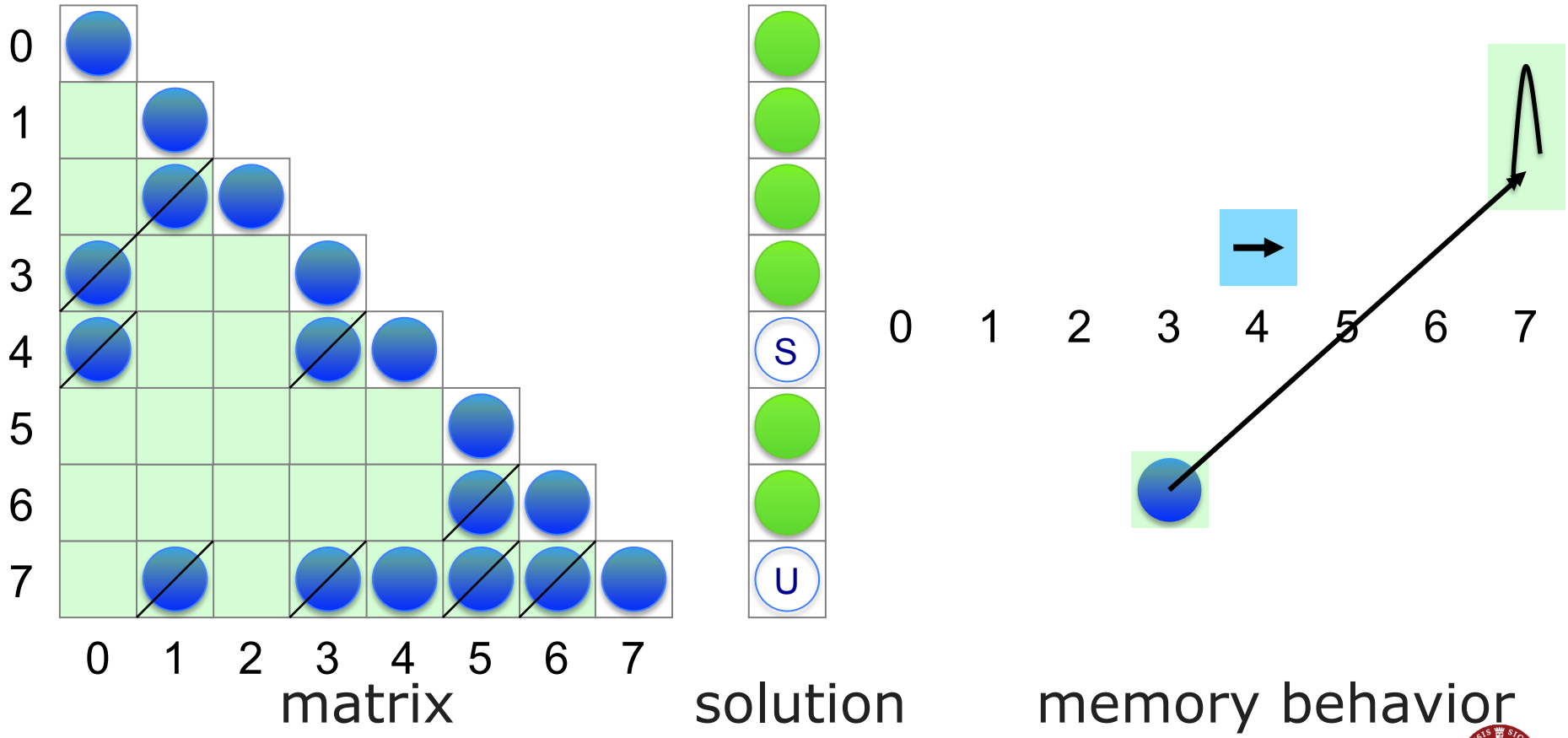
Cartoon of an example (step 8)



in-degree:

1	1	1	1	1	1	1	3

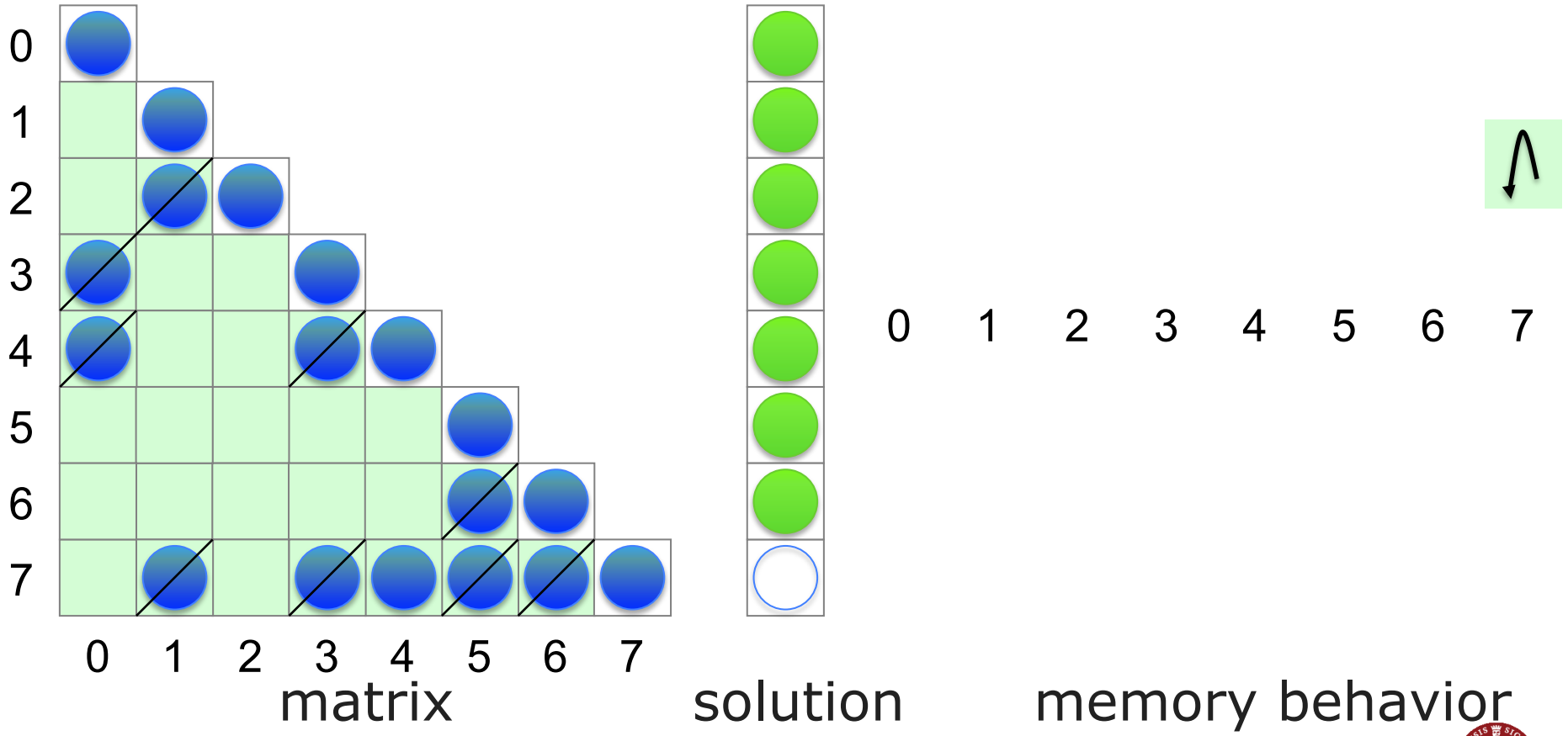
Cartoon of an example (step 9)



in-degree:

1	1	1	1	1	1	1	2

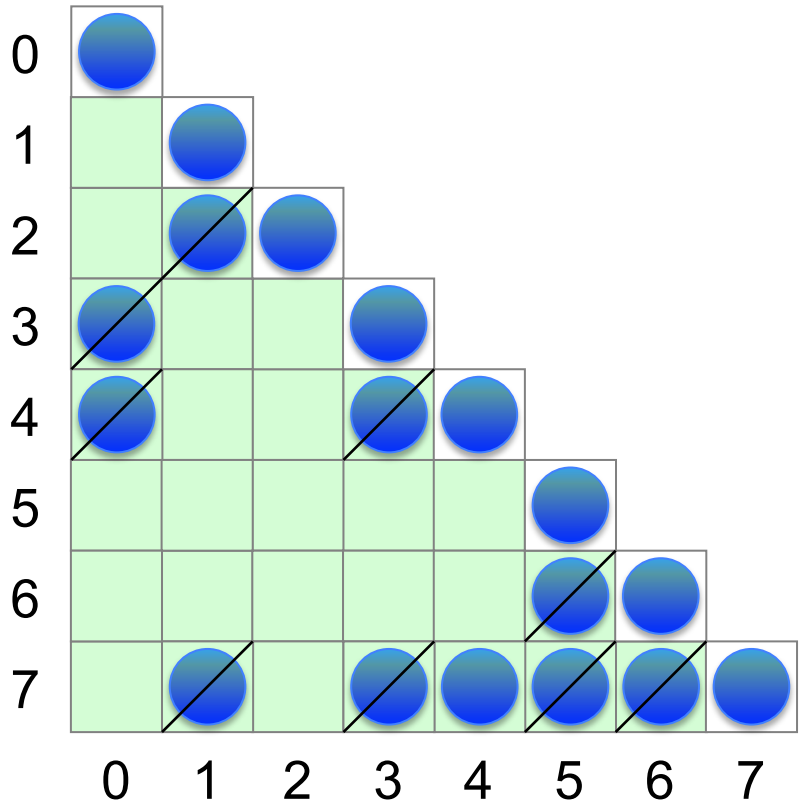
Cartoon of an example (step 10)

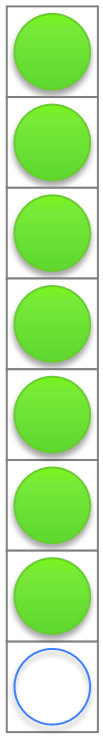


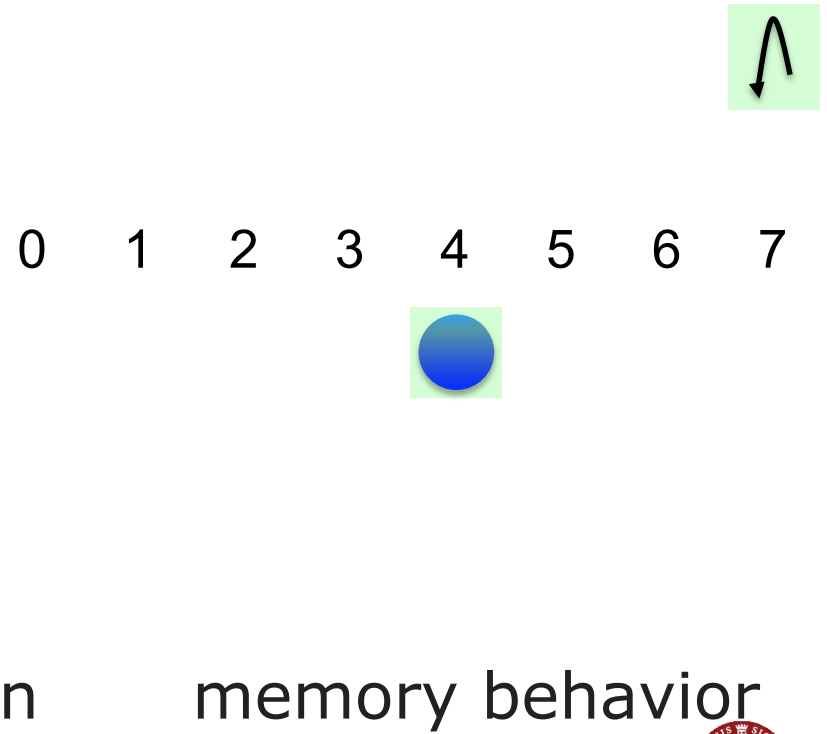
in-degree:

1	1	1	1	1	1	1	2

Cartoon of an example (step 11)



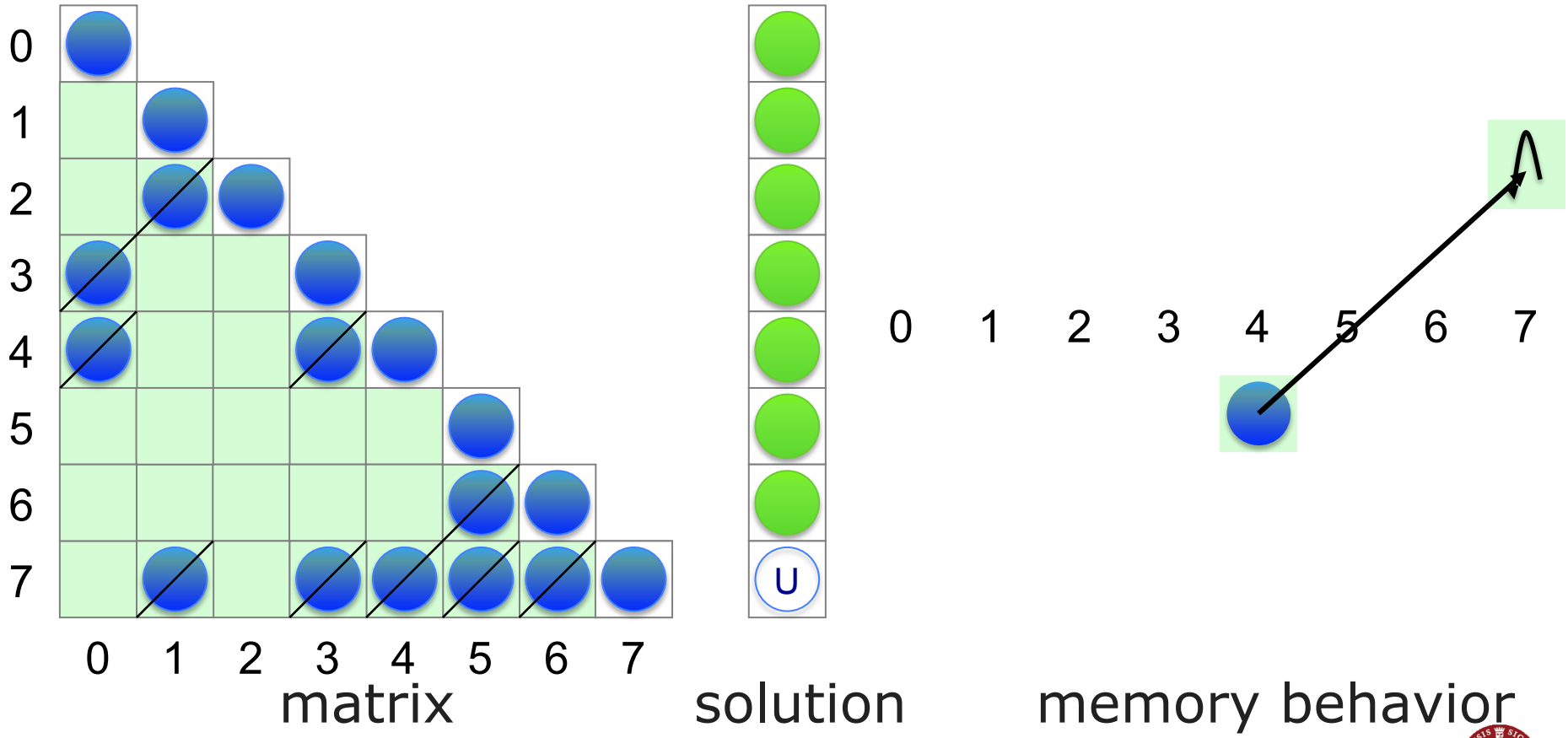




in-degree:

1	1	1	1	1	1	1	2

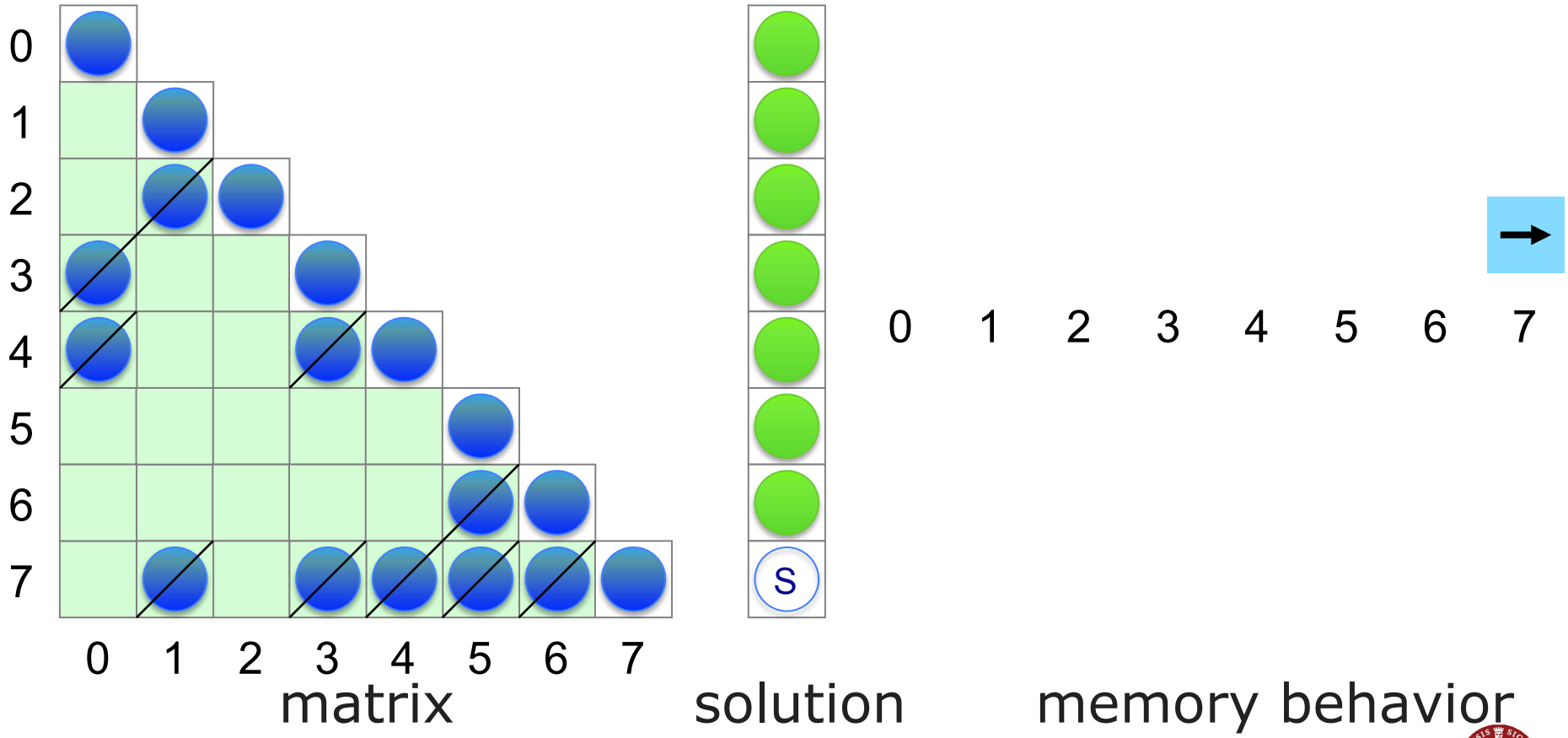
Cartoon of an example (step 12)



in-degree:

1	1	1	1	1	1	1	1

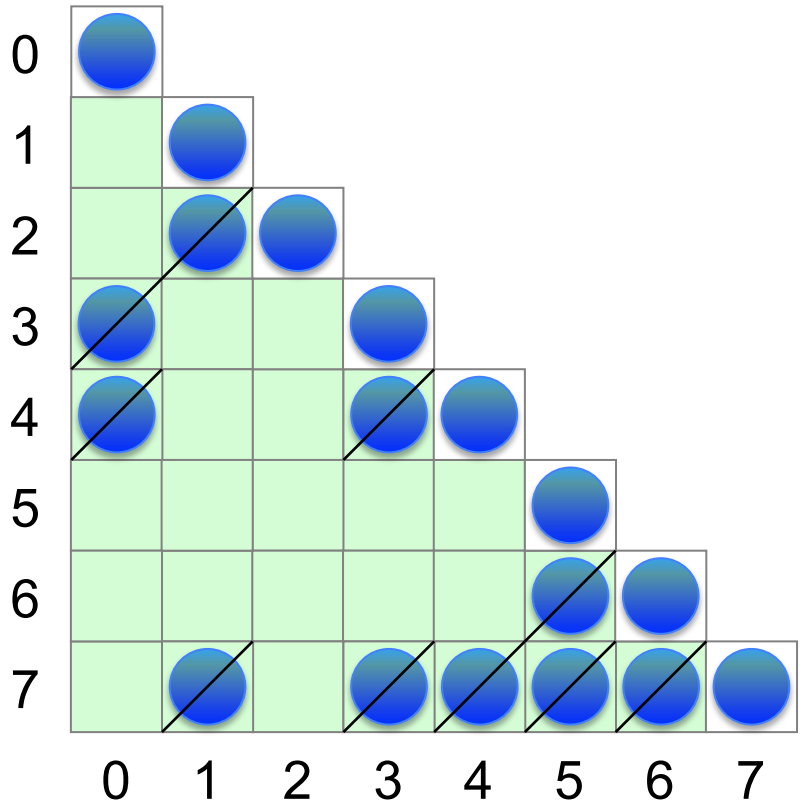
Cartoon of an example (step 13)

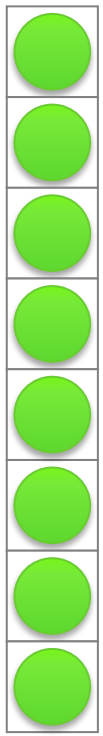


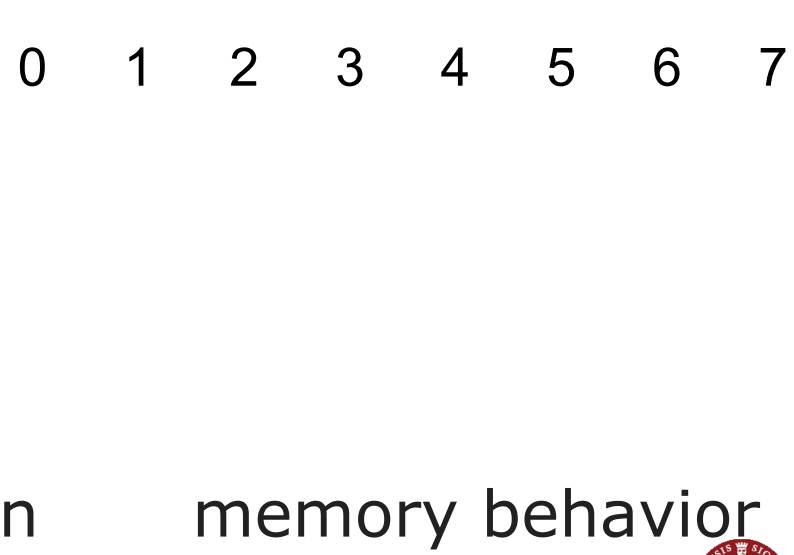
in-degree:

1	1	1	1	1	1	1	1

Cartoon of an example (step 14)







in-degree:

1	1	1	1	1	1	1	1

Advantages of the sync-free SpTRSV

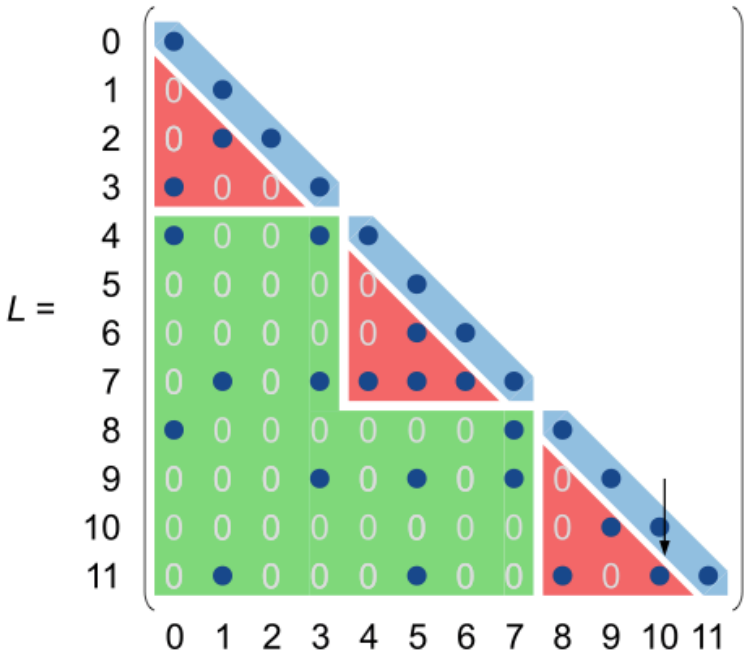
- The preprocessing stage only generates in-degree information. In our case, it's an array

1	1	2	2	3	1	2	6

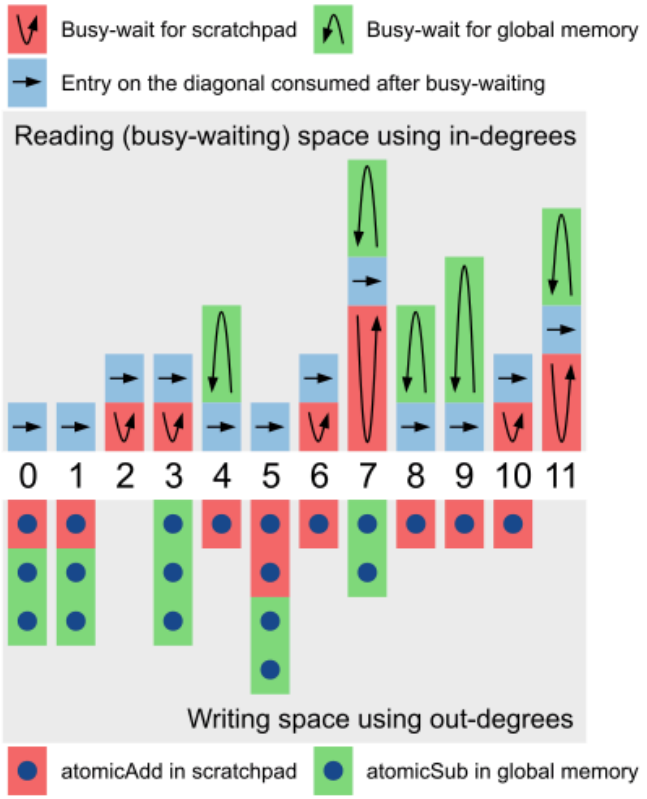
  which takes negligible cost to calculate.

- In the solution stage, no explicit device-level barrier synchronization is required.

See a more complex case in the paper



(a) Matrix.



(b) Read/write behaviours.

Experimental Results

Experiment platforms

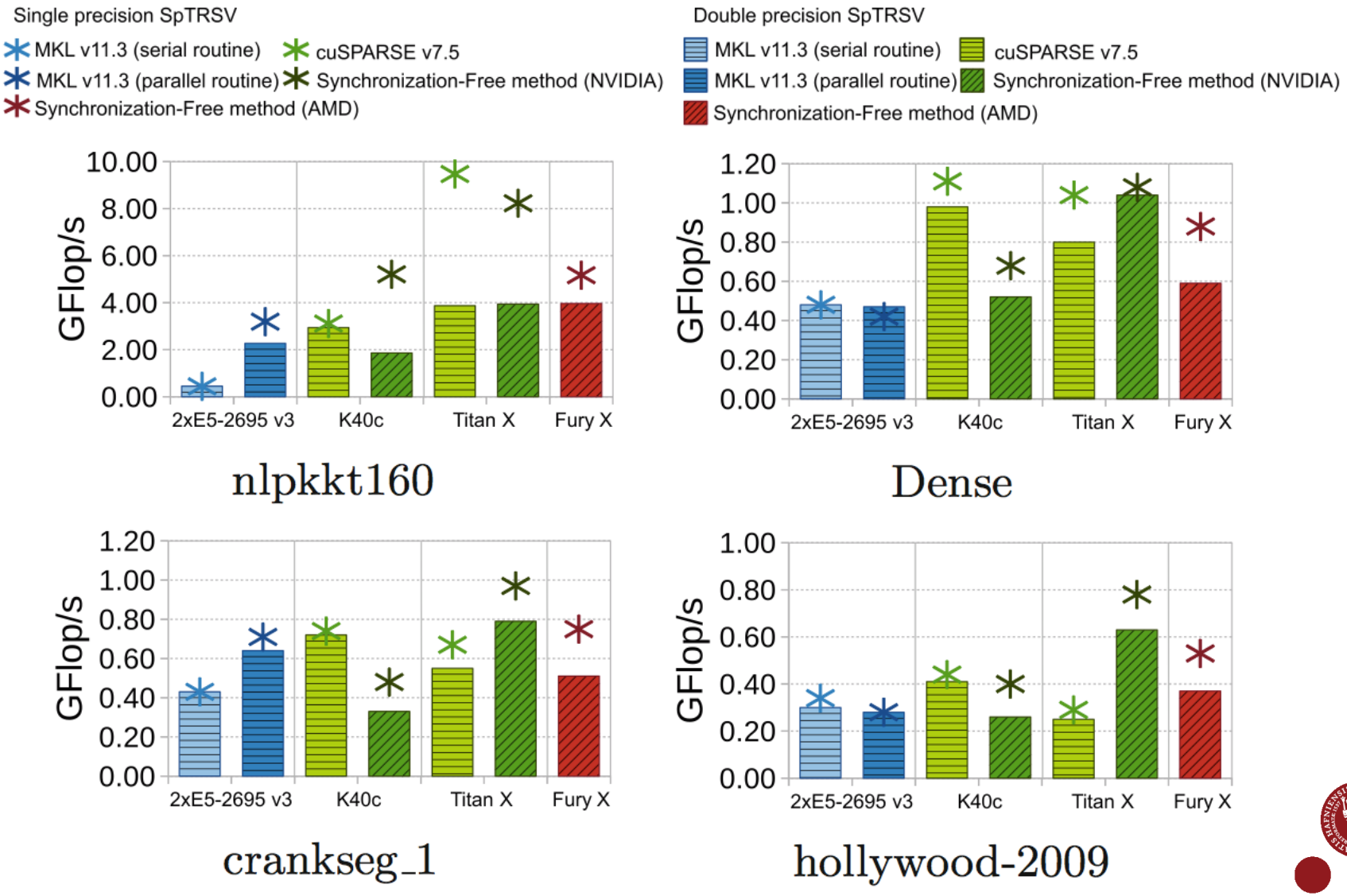
- **Intel Xeon E5-2695 v3 (dual-socket)**, 2x 14 Haswell cores @ 2.3 GHz, 2x 68.3 GB/s Bandwidth. (1) *mkl_?csrtrsv* in MKL v11.3, (2) inspector-executor *mkl_sparse_?_trsv* in MKL v11.3.
- **nVidia Tesla K40c**, 2880 CUDA cores @ 0.75 GHz, 288 GB/s bandwidth. (1) *cusparse?csrsv2_solve* in cuSPARSE v7.5, (2) **Sync-Free SpTRSV (this work)**.
- **nVidia GeForce GTX Titan X**, 3072 CUDA cores @ 1 GHz, 336.5 GB/s bandwidth. (1) *cusparse?csrsv2_solve* in cuSPARSE v7.5, (2) **Sync-Free SpTRSV (this work)**.
- **AMD Radeon R9 Fury X**, 4096 Radeon cores @ 1.05 GHz, 512 GB/s bandwidth. (1) **Sync-Free SpTRSV (this work)**.

Benchmark suite

- 11 matrices downloadable from the University of Florida sparse matrix collection.

Matrix name	#Rows/Columns	#Nonzeros	#Level-sets	Parallelism
nlpkkt160	8,345,600	229,518,112	2	4,172,800
road_central	14,081,816	33,866,826	59	238,675
road_usa	23,947,347	57,708,624	77	311,004
webbase-1M	1,000,005	3,105,536	514	1,946
wiki-Talk	2,394,385	5,021,410	522	4,587
chipcool0	20,082	281,150	534	37
Dense	2,000	4,000,000	2,000	1
FEM/Cantilever	62,451	4,007,383	2,397	26
crankseg_1	52,804	10,614,210	4,056	13
FEM/ship_003	121,728	8,086,034	4,367	28
hollywood-2009	1,139,905	113,891,327	82,735	14

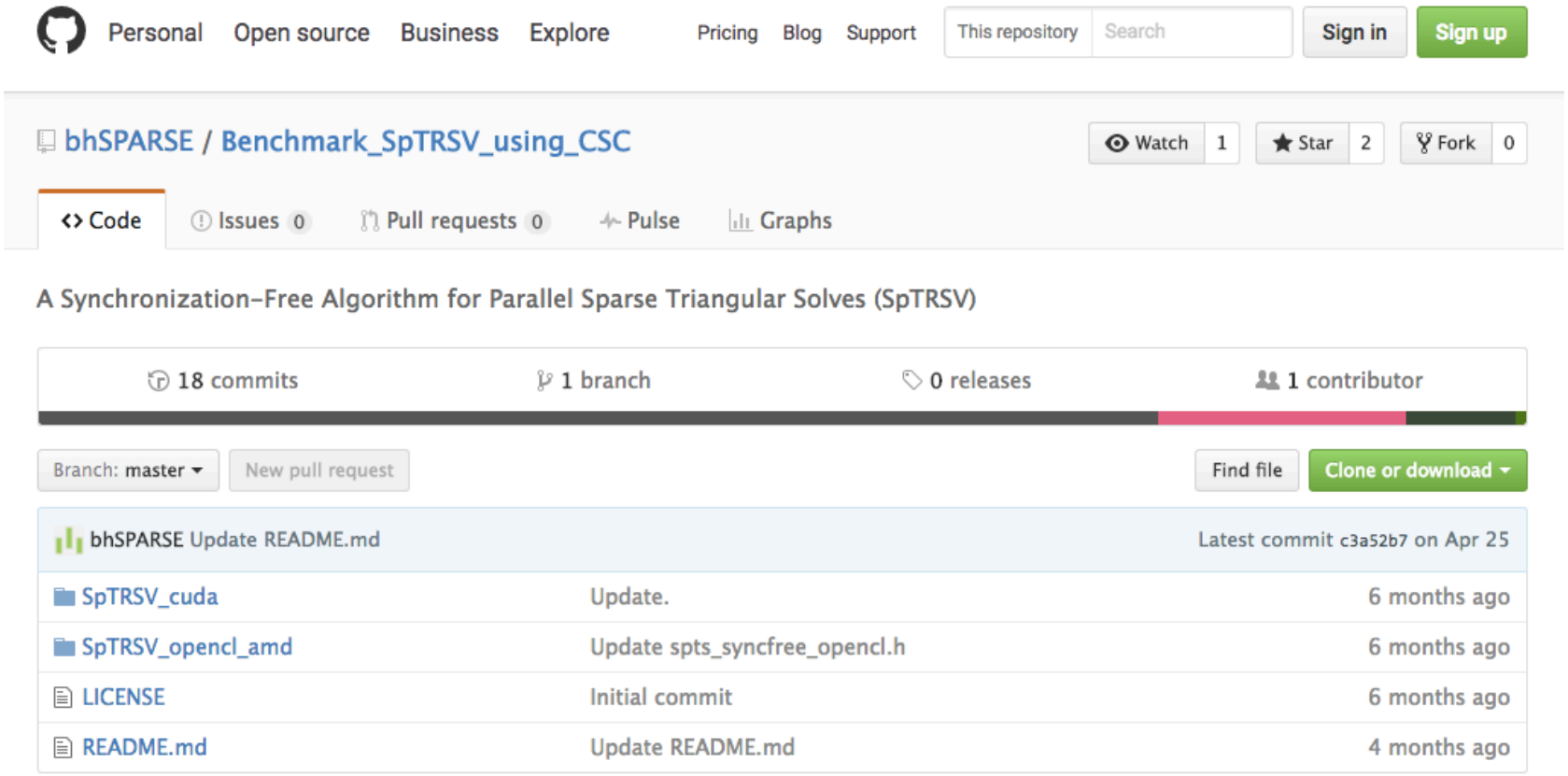
Performance (GFlop/s, FP32 & FP64)



Preprocessing cost (milliseconds)

Matrix name	Intel 2xE5-2695 v3	NVIDIA K40c		NVIDIA Titan X		AMD Fury X
	MKL	cuSPARSE	Sync-Free	cuSPARSE	Sync-Free	Sync-Free
nlpkkt160	64.43	40.58	7.27	19.99	8.91	5.58
road_usa	155.48	160.41	5.06	84.01	3.37	2.31
road_central	92.16	82.01	9.28	42.62	6.98	5.53
wiki-Talk	17.38	16.27	0.33	10.49	0.20	0.16
webbase-1M	7.08	8.53	0.19	5.48	0.13	0.11
chipcool0	1.05	1.48	0.02	1.41	0.02	0.02
FEM/ship_003	9.14	6.41	0.19	4.34	0.26	0.13
FEM/Cantilever	9.52	8.92	0.10	8.28	0.16	0.07
hollywood-2009	223.54	139.98	5.20	204.10	4.82	2.78
crankseg_1	9.30	8.93	0.24	6.14	0.43	0.14
Dense	9.29	3.46	0.08	2.99	0.12	0.05
Harmonic mean	6.80	6.99	0.12	5.71	0.13	0.10

Source Code at Github



https://github.com/bhSPARSE/Benchmark_SpTRSV_using_CSC

Conclusion

The proposed synchronization-free SpTRSV

- removes the cost for finding level-sets and the cost for explicit barrier synchronization at runtime;
- includes optimization for on-chip and off-chip memory operations;
- shows high speedups over level-set methods, in particular when #level-sets is high;
- supports both CUDA for NVIDIA GPUs and OpenCL for AMD GPUs.

T k u !

0	4	8	9

A y Q s n s ?

0	2	4	7	11	12	13