Parallel Transposition of Sparse Data Structures

Hao Wang[†], Weifeng Liu[§‡], Kaixi Hou[†], Wu-chun Feng[†]

[†]Department of Computer Science, Virginia Tech

[§]Niels Bohr Institute, University of Copenhagen

[‡]Scientific Computing Department, STFC Rutherford Appleton Laboratory

VirginiaTech Invent the Future

Science & Technology Facilities Council Rutherford Appleton Laboratory

Sparse Matrix

- If most elements in a matrix are zeros, we can use sparse representations to store the matrix

$$A = \begin{bmatrix} 0 & a & 0 & b & 0 & 0 \\ c & d & e & f & 0 & 0 \\ 0 & 0 & g & h & i & j \\ 0 & k & l & m & n & p \end{bmatrix}$$

Compressed Sparse Row (CSR)

csrRowPtr =	0	2	6	10	15										
csrColIdx =	1	3	0	1	2	3	2	3	4	5	1	2	3	4	5
csrVal =	a	b	c	d	e	f	g	h	i	j	k	l	m	n	p

Compressed Sparse Column (CSC)

cscColPtr =	0	1	4	7	11	13	15								
cscRowIdx =	1	0	1	3	1	2	3	0	1	2	3	2	3	2	3
cscVal =	c	a	d	k	e	g	l	b	f	h	m	i	n	j	p

Spare Matrix Transposition

- Sparse matrix transposition from A to A^T is an indispensable building block for higher-level algorithms

$$A = \begin{bmatrix} 0 & a & 0 & b & 0 & 0 \\ c & d & e & f & 0 & 0 \\ 0 & 0 & g & h & i & j \\ 0 & k & l & m & n & p \end{bmatrix} \Rightarrow A^T = \begin{bmatrix} 0 & c & 0 & 0 \\ a & d & 0 & k \\ 0 & e & g & l \\ b & f & h & m \\ 0 & 0 & i & n \\ 0 & 0 & j & p \end{bmatrix}$$

CSR ⇨ CSC

CSR

csrRowPtr =	0	2	6	10	15						
csrColIdx =	1	3	0	1	2	3	2	3	4	5	…
csrVal =	a	b	c	d	e	f	g	h	i	j	…

CSC

0	1	4	7	11	13	15				
1	0	1	3	1	2	3	0	1	2	…
c	a	d	k	e	g	l	b	f	h	…

Spare Matrix Transposition

- Sparse transposition has not received the attention like other sparse linear algebra, e.g., *SpMV* and *SpGEMM*
  - Transpose A to A^T once and then use A^T multiple times
  - Sparse transposition is fast enough on modern architectures
  - It is not always true!

Driving Cases

- Sparse transposition is inside the main loop
  - K-truss, Simultaneous Localization and Mapping (SLAM)
- Or, may occupy significant percentage of execution time
  - Strongly Connected Components (SCC)

Examples	Description	Class	Build Blocks
K-truss [1]	Detect sub graphs where each edge is part of at least k-2 triangles	Graph algorithm	SpMV, SpGEMM, Transposition
SLAM [2]	Update an information matrix of an autonomous robot trajectory	Motion planning algorithm	SpGEMM, Transposition
SCC [3]	Detect components where every vertex is reachable from every other vertex	Graph algorithm	Transposition, Set operations

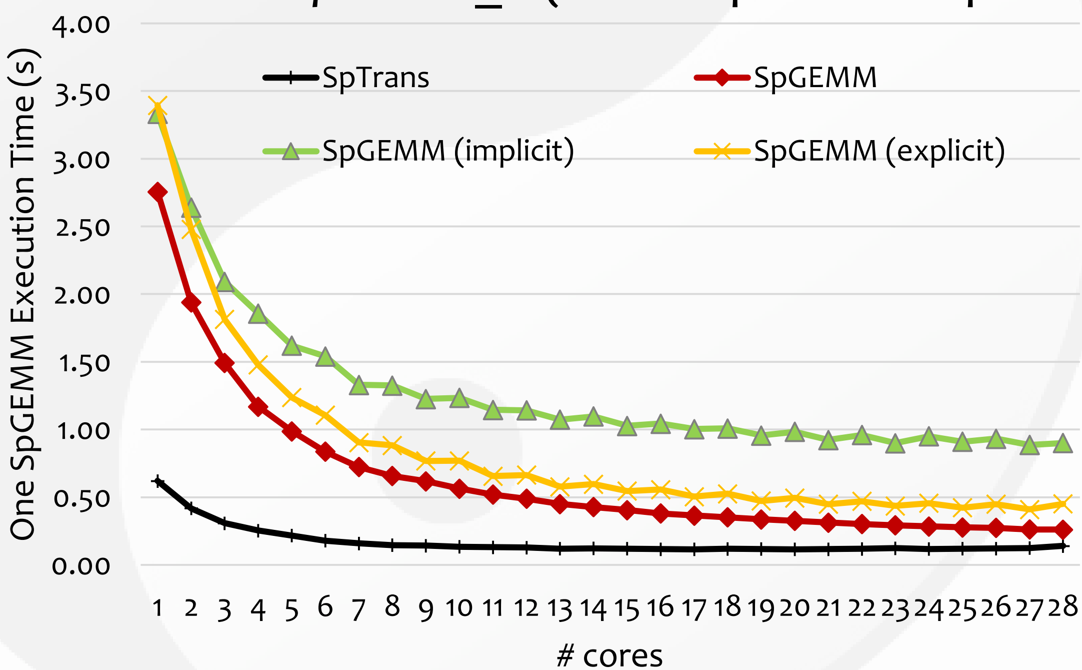
[1] J. Wang and J. Cheng, "Truss Decomposition in Massive Networks", PVLDB 5(9):812-823, 2012

[2] F. Dellaert and M. Kaess, "Square Root SAM: Simultaneous Localization and Mapping via Square Root Information Smoothing ", IJRR 25(12):1181-1203, 2006

[3] S. Hong, etc. "On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-world Graphs", SC'13, 2013

Motivation

- *SpTrans* and *SpGEMM* from Intel MKL Sparse BLAS
  - *SpGEMM* no transposition: *C1 = AA*
  - *SpGEMM_T* (with implicit transposition*): *y2 = trans(A)A*
  - *SpGEMM_T* (with explicit transposition): *B = trans(A) then C2 = BA*



Experiment setup:

1. Sparse matrix is **web-Google**
2. Intel Xeon (Haswell) CPU with 28 cores
3. SpGEMM is iterated only one time

Observations:

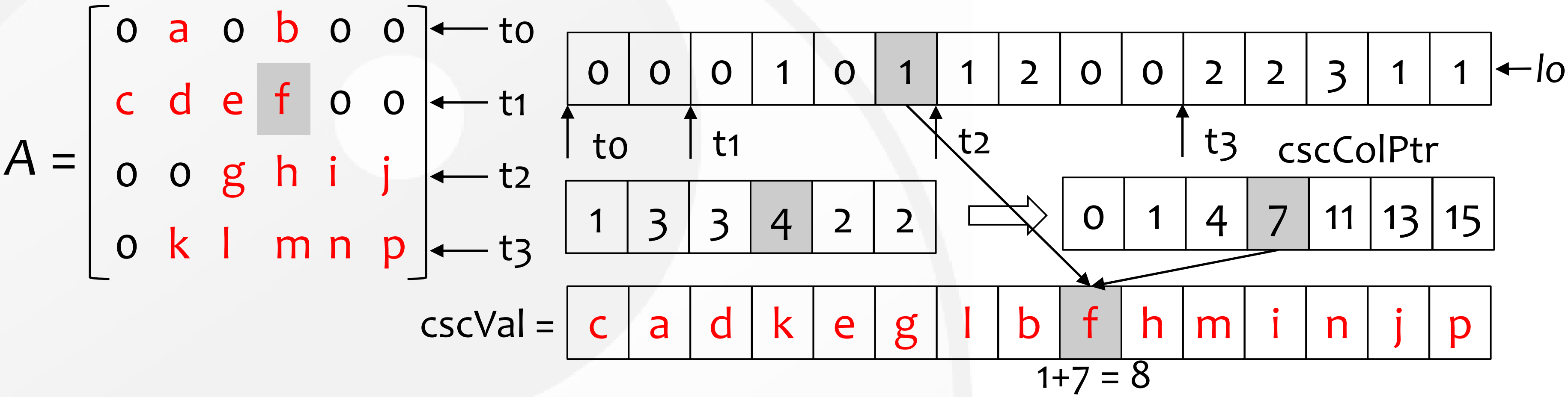
1. *SpTrans* and *SpGEMM* (implicit) did not scale very well
2. Time spending on *SpTrans* was close to *SpGEMM* if multiple cores were used

Implicit transposition can use *A* as an input, but with a hint let higher-level computations operate on A^T. Supported by Intel.

Outlines

- Background
- Motivations
- **Existing Methods**
  - Atomic-based
  - Sorting-based
- Designs
  - ScanTrans
  - MergeTrans
- Experimental Results
- Conclusions

Atomic-based Transposition



1. Calculate the offset of each nonzero element in its column, set offset in auxiliary array *loc*, and count how many nonzero elements in each column
   – Atomic operation fetch_and_add()
2. Use prefix-sum to count the start pointer for each column, i.e., cscColPtr
3. Scan CSR again to get the position of each nonzero element in cscRowIdx and cscVal, and move it
4. An additional step, i.e., segmented sort, may be required to guarantee the order in each column
   – Offset of ‘f’ can be 0, 1, 2, 3, and final position of ‘f’ can be 7, 8, 9, 10

Sorting-based Transposition (First Two Steps)

$$A = \begin{bmatrix} 0 & a & 0 & b & 0 & 0 \\ c & d & e & f & 0 & 0 \\ 0 & 0 & g & h & i & j \\ 0 & k & l & m & n & p \end{bmatrix}$$

Compressed Sparse Row (CSR)

csrRowPtr =	0	2	6	10	15										
csrColIdx =	1	3	0	1	2	3	2	3	4	5	1	2	3	4	5
csrVal =	a	b	c	d	e	f	g	h	i	j	k	l	m	n	p

1. Use key-value sort to sort csrColIdx (key) and auxiliary positions (value)

csrColIdx =	1	3	0	1	2	3	2	3	4	5	1	2	3	4	5
auxPos =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

⇩ key-value sort

csrColIdx =	0	1	1	1	2	2	2	3	3	3	3	4	4	5	5
auxPos =	2	0	3	10	4	6	11	1	5	7	12	8	13	9	14

2. Set cscVal based on auxPos: cscVal[x] = csrVal[auxPos[x]]

cscVal =	c	a	d	k	e	g	l	b	f	h	m	i	n	j	p

For x = 7, cscVal[7] = ?
auxPos[7] = 1
csrVal[1] = b

Constraints in Existing Methods

- Atomic-based sparse transposition
  - Contention from the atomic operation *fetch_and_add()*
  - Additional overhead coming from the segmented sort
- Sorting-based sparse transposition
  - Performance degradation when the number of nonzero elements increases, due to O(*nnz* * *log*(*nnz*)) complexity

Outlines

- Background
- Motivations
- Existing Methods
  - Atomic-based
  - Sorting-based
- **Designs**
  - ScanTrans
  - MergeTrans
- Experimental Results
- Conclusions

Performance Considerations

- Sparsity independent
  - Performance should not be affected by load imbalance, especially for power-law graphs
- Avoid atomic operation
  - Avoid the contention of atomic operation
  - Avoid the additional stage of sorting indices inside a row/column
- Linear complexity
  - The serial method of sparse transposition has the linear complexity $O(m + n + nnz)$
  - Design parallel methods to achieve closer to it

ScanTrans

$$A = \begin{bmatrix} 0 & a & 0 & b & 0 & 0 \\ c & d & e & f & 0 & 0 \\ 0 & 0 & g & h & i & j \\ 0 & k & l & m & n & p \end{bmatrix}$$

csrRowPtr =

0	2	6	10	15

csrColIdx =

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

csrVal =

a	b	c	d	e	f	g	h	i	j	k	l	m	n	p

csrRowIdx =

0	0	1	1	1	1	2	2	2	2	3	3	3	3	3

t_0 t_1 t_2 t_3

① histogram

intra (nnz)

0	0	0	1	0	0	1	1	0	0	0	1	0	0	0

inter ($(p+1) * n$)

	0	0	0	0	0	0
t_0	1	2	0	1	0	0
t_1	0	0	2	2	0	0
t_2	0	1	1	0	1	1
t_3	0	0	0	1	1	1

② vertical_scan

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

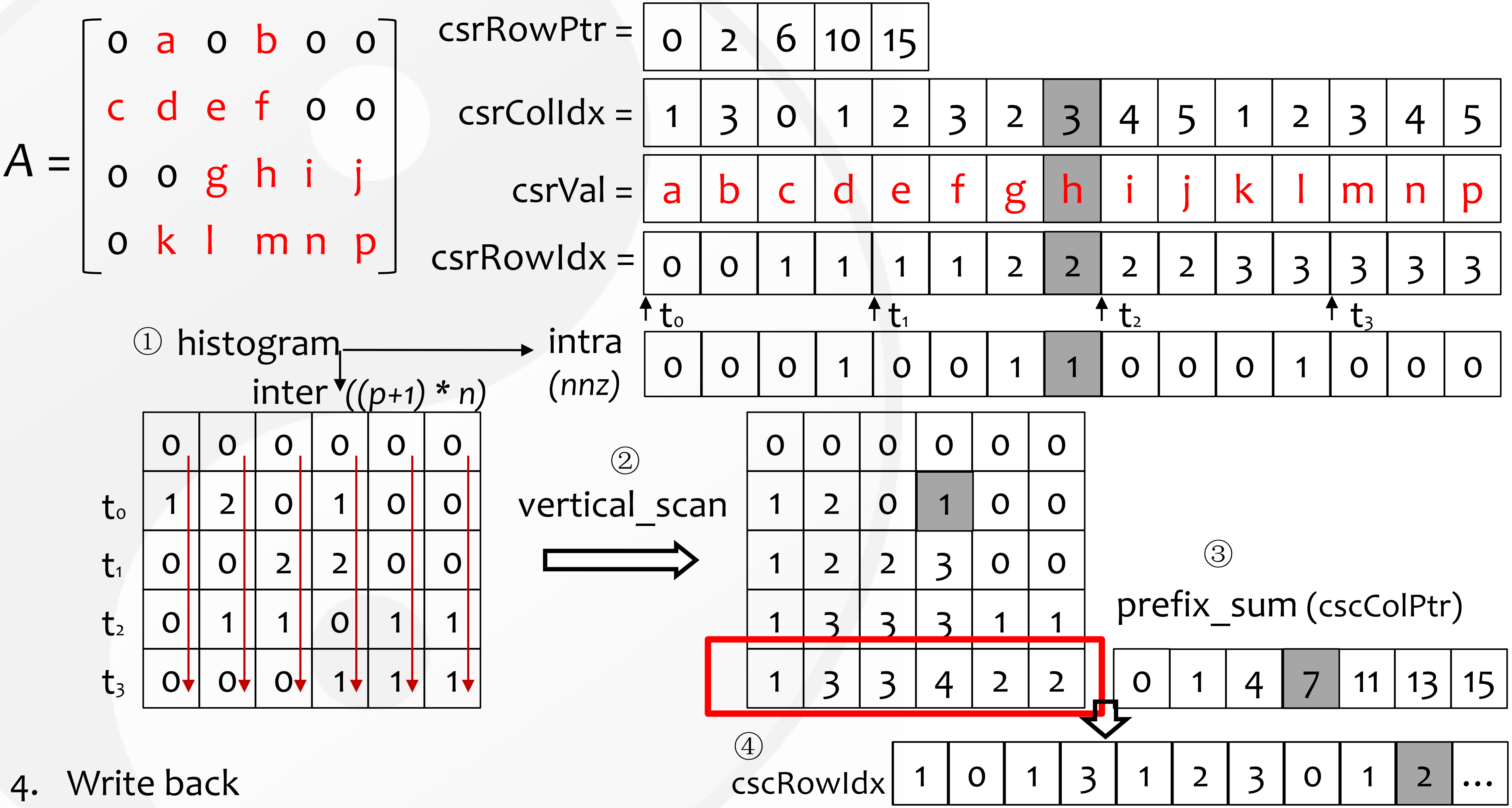
③ prefix_sum (cscColPtr)

0	1	4	7	11	13	15

Preprocess: extend csrRowPtr to csrRowIdx; partition csrVal evenly for threads

1. Histogram: count numbers of column indices per thread independently
2. Vertical scan (on inter) 3. Horizontal scan (on last row of inter)

ScanTrans



4. Write back

*off = cscColPtr[colIdx] + inter[tid*n+colIdx] + intra[pos]*

*off = cscColPtr[3] + inter[1*6+3] + intra[7] = 7+1+1 = 9*

Analysis of ScanTrans

- Pros
  - Two round of Scan on auxiliary arrays to avoid atomic operation
  - Scan operations can be implemented by using SIMD operations
- Cons
  - Write back step has random memory access on cscVal and cscRowIdx

MergeTrans

$$A = \begin{bmatrix} 0 & a & 0 & b & 0 & 0 \\ c & d & e & f & 0 & 0 \\ 0 & 0 & g & h & i & j \\ 0 & k & l & m & n & p \end{bmatrix}$$

csrRowPtr =

0	2	6	10	15

csrColIdx =	1	3	0	1	2	3	2	3	4	5	1	2	3	4	5
csrVal =	a	b	c	d	e	f	g	h	i	j	k	l	m	n	p
csrRowIdx =	0	0	1	1	1	1	2	2	2	2	3	3	3	3	3
	↑ t_0				↑ t_1				↑ t_2				↑ t_3		

① csr2csc_block

t_0:

cscColPtr	0	1	3	3	4	4	4
cscRowIdx	1	0	1	0			
cscVal	c	a	d	b			

t_1:

cscColPtr	0	0	0	2	4	4	4
cscRowIdx	1	2	1	2			
cscVal	e	g	f	h			

t_2:

cscColPtr	0	0	1	2	2	3	4
cscRowIdx	3	3	2	2			
cscVal	k	l	i	j			

t_3:

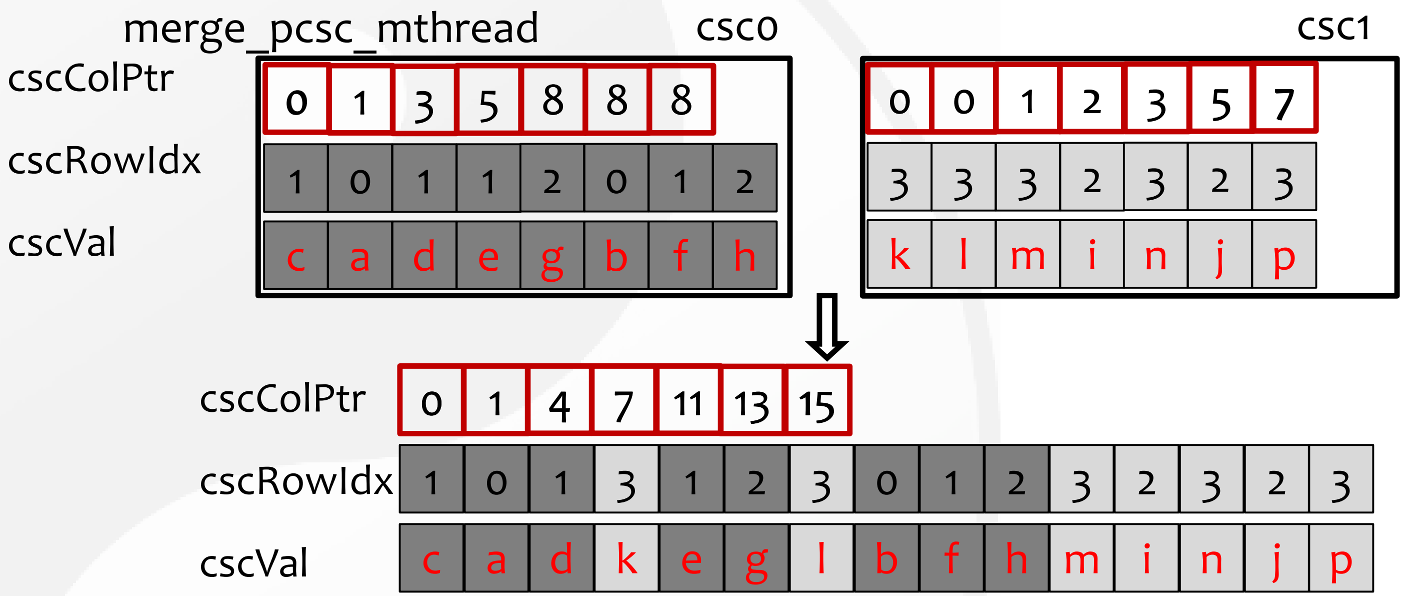
cscColPtr	0	0	0	0	1	2	3
cscRowIdx	3	3	3				
cscVal	m	n	p				

② multiple round merge ⇩

Preprocess: partition nonzero elements to multiple blocks

1. Each thread transpose one or several blocks to CSC format
2. Merge multiple blocks in parallel until one block left

How to Mitigate Random Memory Access



Merge two csc to one csc

1. Add two cscColPtr directly to get the output cscColPtr
2. For each column of output csc, check where the nonzero elements come from; and then move nonzero elements (cscVal and cacRowIdx) from input csc to output csc
   - Opt: only if two successive columns in both input csc change, we move the data

Analysis of MergeTrans

- Pros
  - Successive memory access on both input csc and output csc
- Cons
  - Performance is affected by the number of blocks
  - May have much larger auxiliary data *(2 * nblocks * (n + 1) + nnz)* than ScanTrans and existing methods

Implementations and Optimizations

- SIMD Parallel Prefix-sum
  - Implement prefix-sum on x86-based platforms with Intrinsics
  - Support AVX2, and IMCI/AVX512
  - Apply on atomic-based method and ScanTrans
- SIMD Parallel Sort
  - Implement bitonic sort and mergesort on x86-based platform[4]
  - Support AVX, AVX2, and IMCI/AVX512
  - Apply on sorting-based method
- Dynamic Scheduling
  - Use OpenMP tasking (since OpenMP 3.0)
  - Apply on sorting-based method and MergeTrans

[4] K. Hou, etc. "ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on x86-based Many-core Processors", ICS'15, 2015

Outlines

- Background
- Motivations
- Existing Methods
  - Atomic-based
  - Sorting-based
- Designs
  - ScanTrans
  - MergeTrans
- **Experimental Results**
- Conclusions

Evaluation & Discussion

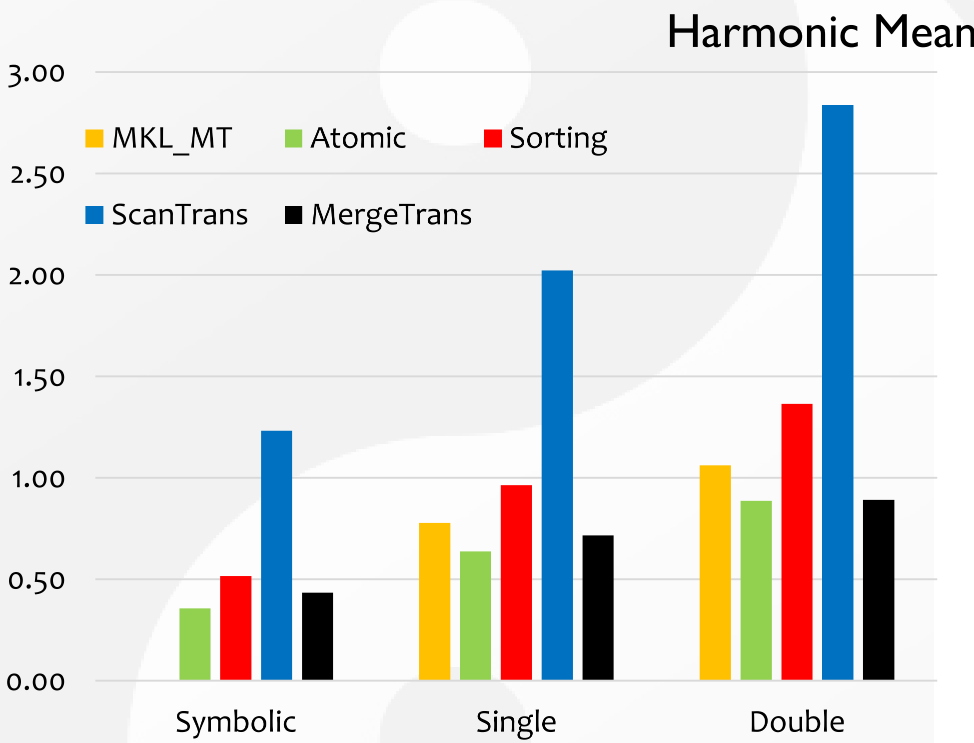
- Experimental Setup (Hardware)

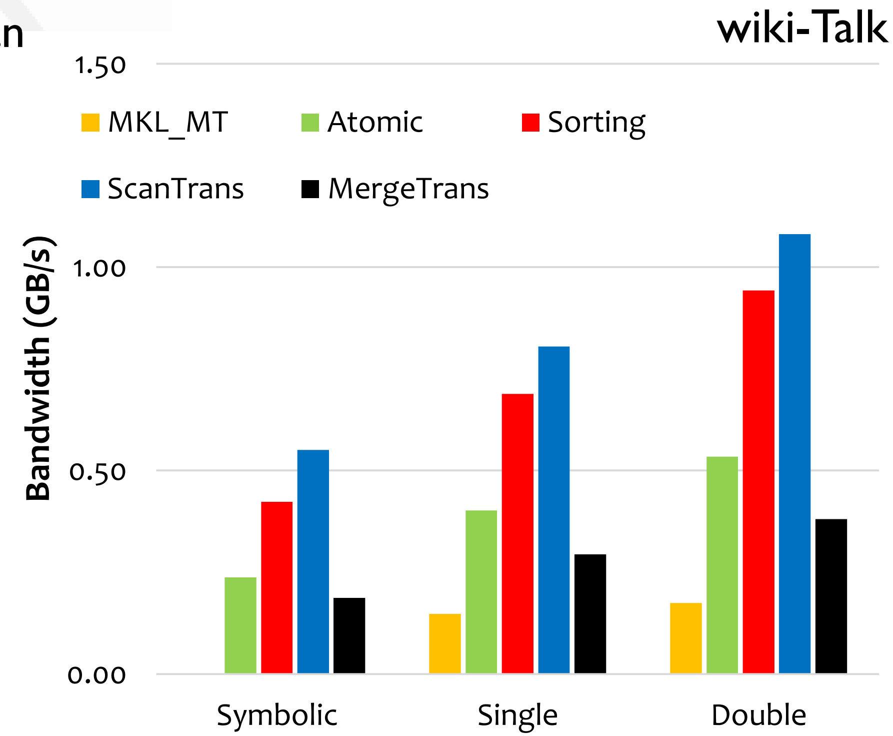
Parameter	CPU	MIC
Product Name	Intel Xeon E5-2695 v3	Intel Xeon Phi 5110P
Code Name	Haswell	Knights Corner
# of Cores	2x14	60
Clock Rate	2.3 GHz	1.05 GHz
L1/L2/L3 Cache	32 KB/ 256 KB/ 35 MB	32 KB/ 512 KB/ -
Memory	128 GB DDR4	8 GB GDDR5
Compiler	icpc 15.3	icpc 15.3
Compiler Options	-xCORE-AVX2 –O3	-mmic -O3
Vector ISA	AVX2	IMCI

Evaluation & Discussion

- Experimental Setup (Methods)
  - Intel MKL 11.3 mkl_sparse_convert_csr()
  - Atomic-based method (from SCC implementation, SC'13[3])
  - Sorting-based method (from bitonic-sort, ICS'15[4])
  - ScanTrans
  - MergeTrans
- Dataset
  - 22 matrices: 21 unsymmetric matrices from University of Florida Sparse Matrix Collection + 1 dense matrix
  - Single precision, Double precision, Symbolic (no value)
- Benchmark Suite
  - Sparse matrix-transpose-matrix addition: $A^T + A$, *SpMV:* $A^T * X$, and SpGEMM: $A^T * A$ (all in explicate mode)
  - Strongly Connected Components: *SCC(A)*

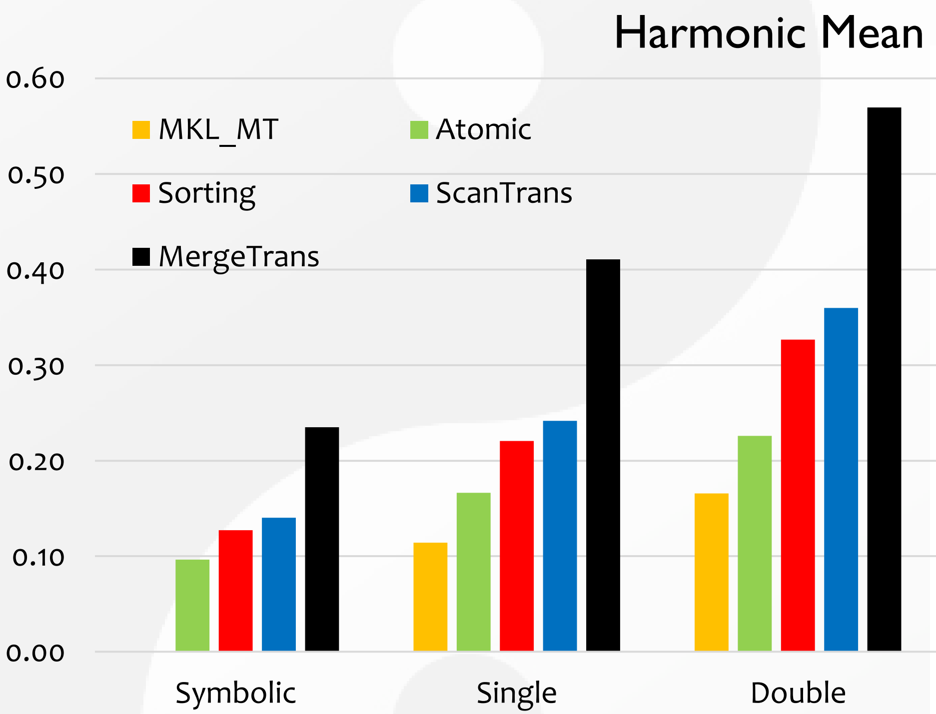
Transposition Performance on Haswell

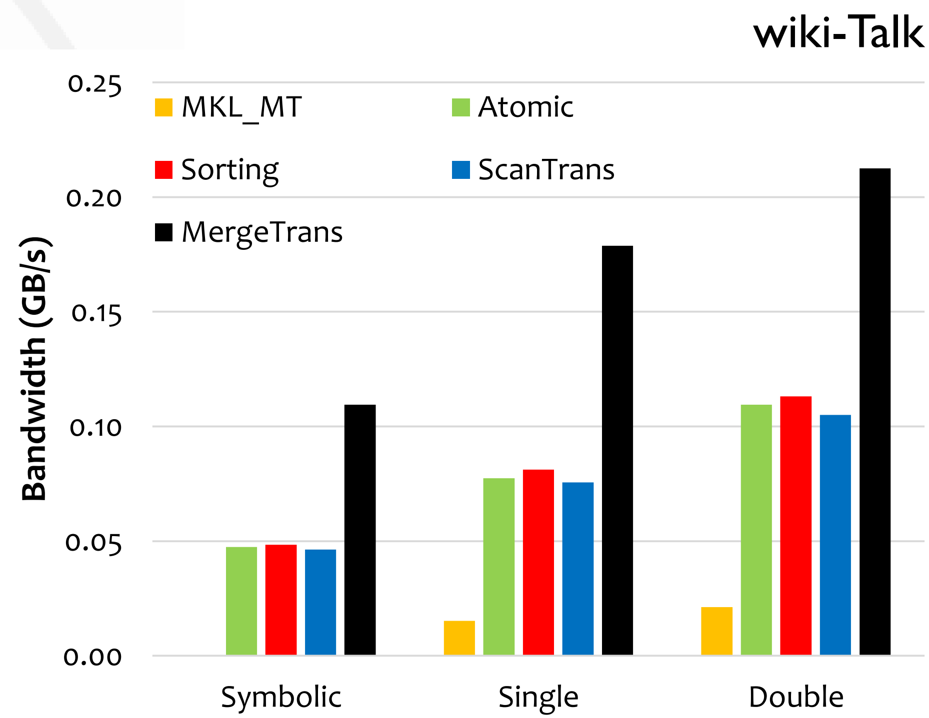




- Compare to Intel MKL method, **ScanTrans** can achieve an average of 2.8x speedup
- On **wiki-Talk**, the speedup can be pushed up to 6.2x for double precision

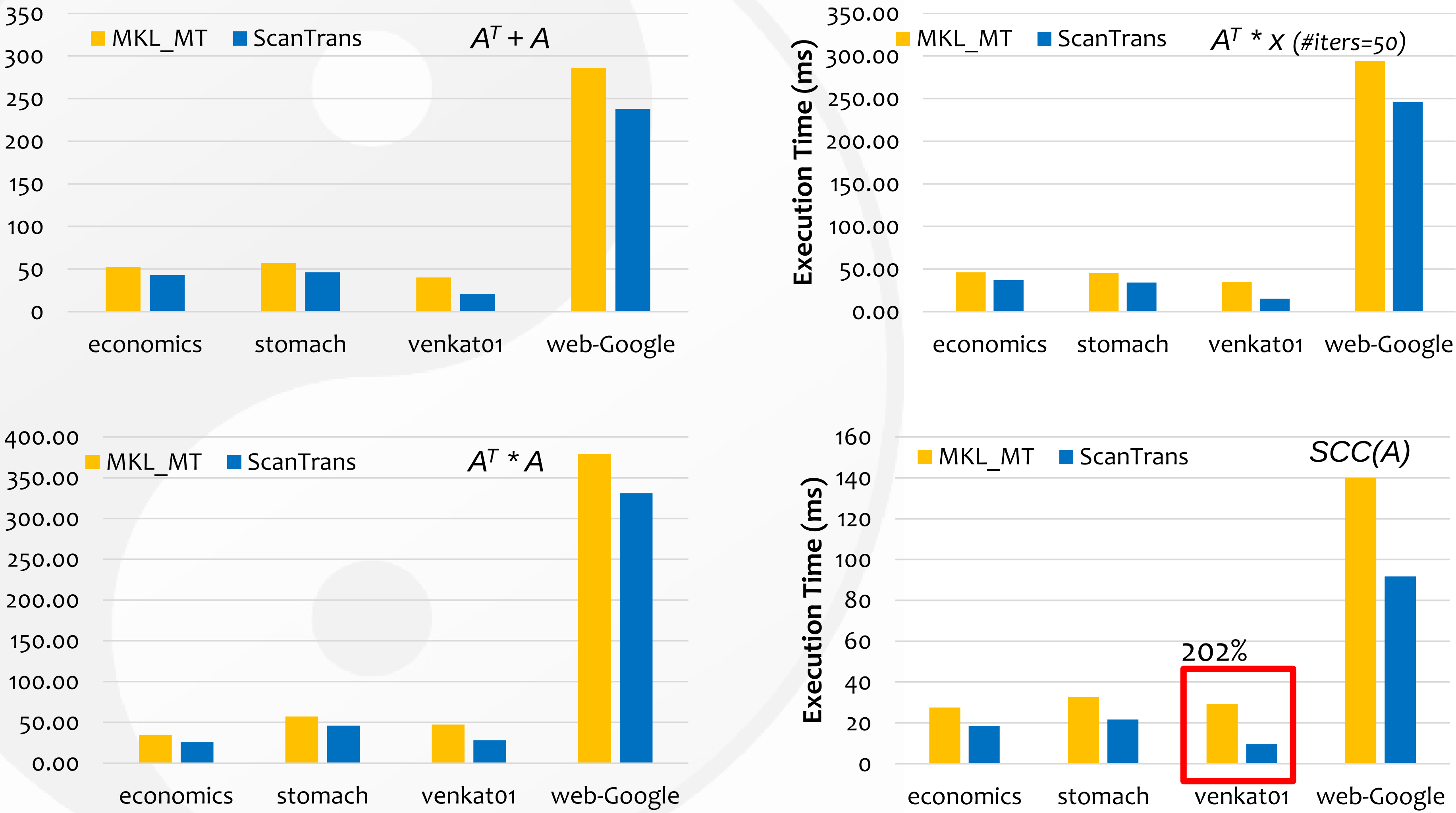
Transposition Performance on MIC





- Compare to Intel MKL method, **MergeTrans** can achieve an average of 3.4x speedup
- On **wiki-Talk**, the speedup can be pushed up to 11.7x for single precision

Higher-level Routines on Haswell



Outlines

- Background
- Motivations
- Existing Methods
  - Atomic-based
  - Sorting-based
- Designs
  - ScanTrans
  - MergeTrans
- Experimental Results
- **Conclusions**

Conclusions

- In this paper
  - We identify the sparse transposition can be the performance bottleneck
  - We propose two sparse transposition methods: ScanTrans and MergeTrans
  - We evaluate the atomic-based, sorting-based, and Intel MKL methods with ScanTrans and MergeTrans on Intel Haswell CPU and Intel MIC
  - Compare to the vendor-supplied library, ScanTrans can achieve an average of 2.8-fold (up to 6.2-fold) speedup on CPU, and MergeTrans can deliver an average of 3.4-fold (up to 11.7-fold) speedup on MIC

Thank you!

Science & Technology Facilities Council
Rutherford Appleton Laboratory