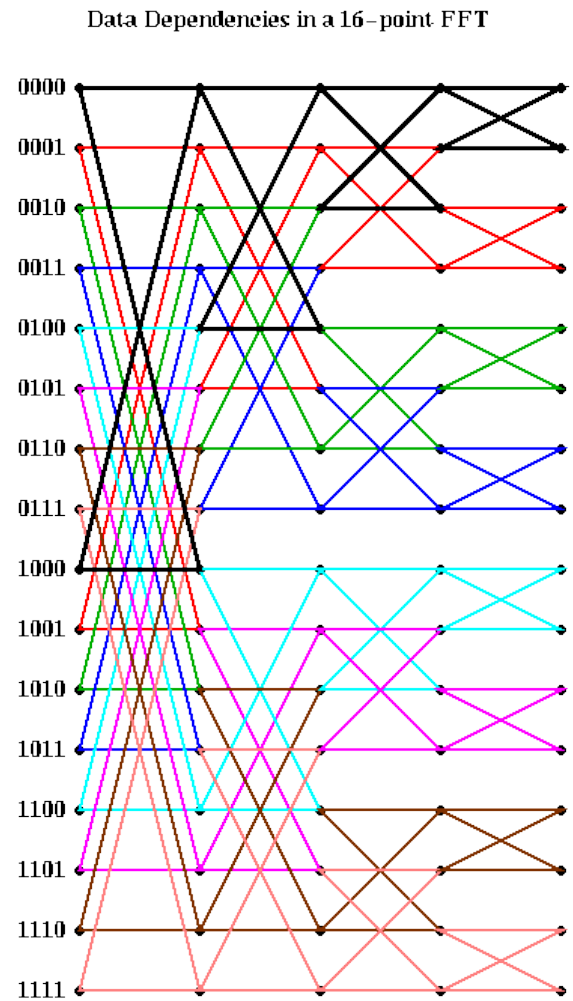
Fast In-place Transposition

I-Jui Sung, University of Illinois
Juan Gómez-Luna, University of Córdoba (Spain)
Wen-Mei Hwu, University of Illinois

Full Transposition

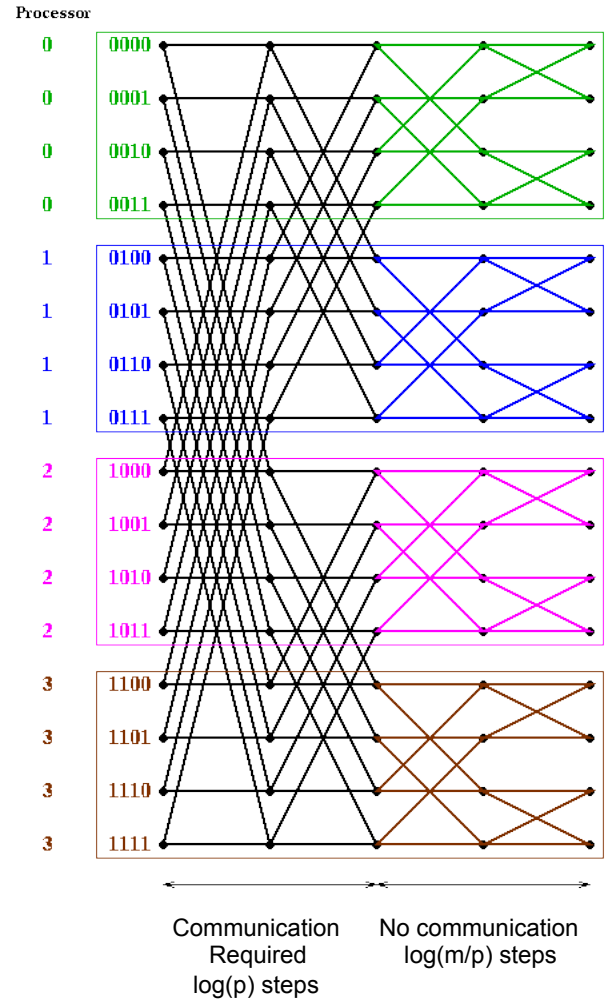
- Full transposition is desired for many algorithms
  - E.g. FFT



Full Transposition

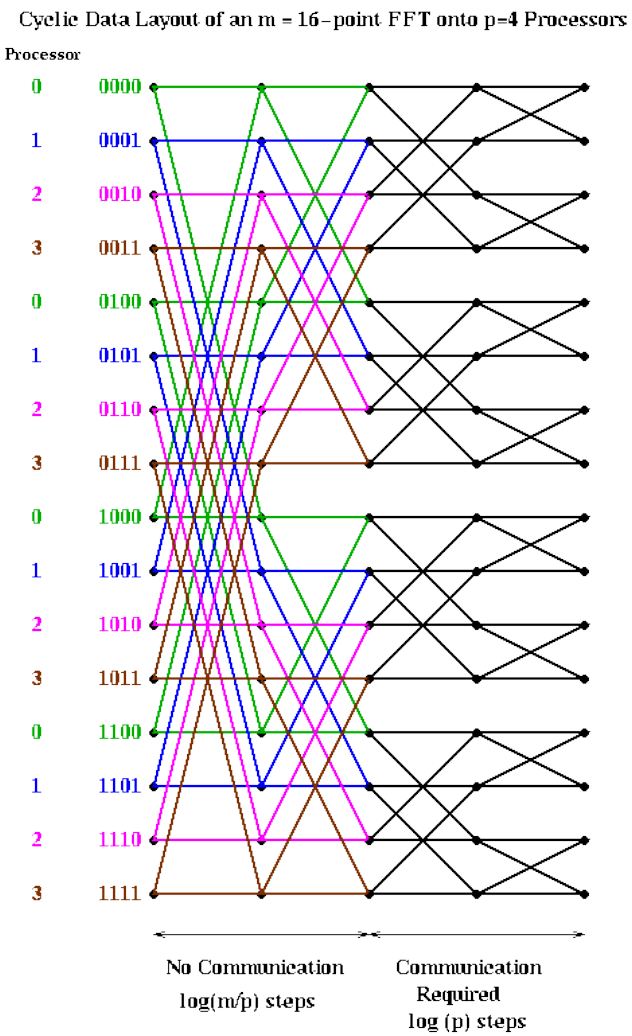
- Full transposition is desired for many algorithms
  - E.g. FFT

Block Data Layout of an m=16-point FFT on p=4 Processors



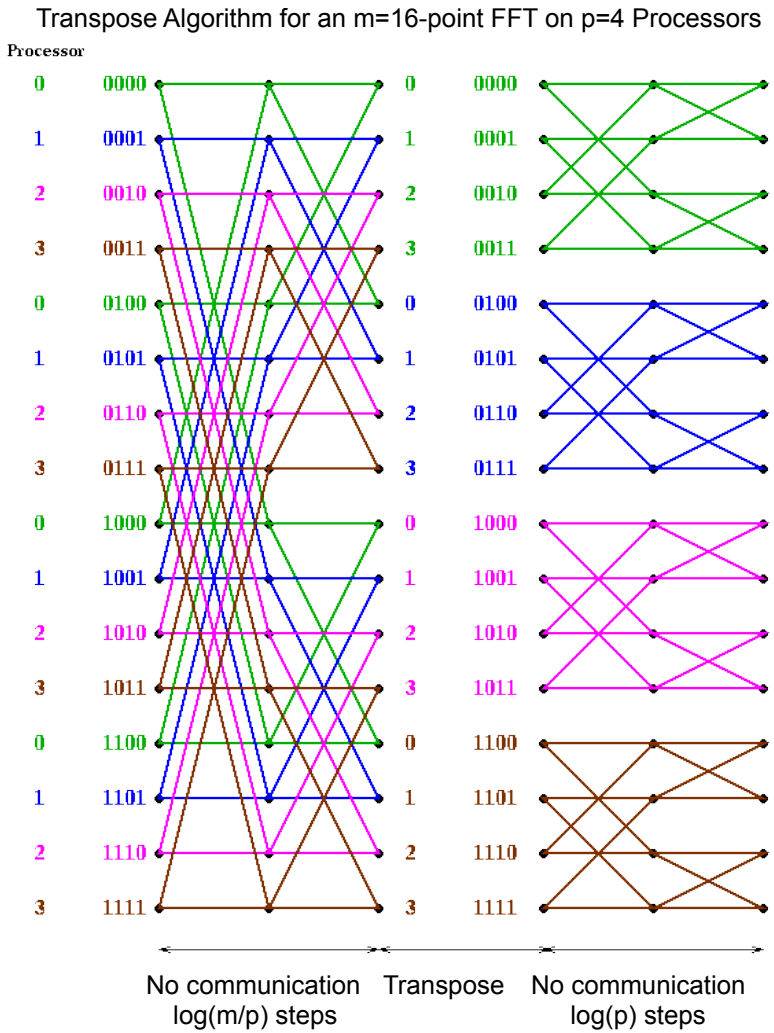
Full Transposition

- Full transposition is desired for many algorithms
  - E.g. FFT



Full Transposition

- Full transposition is desired for many algorithms
  - E.g. FFT



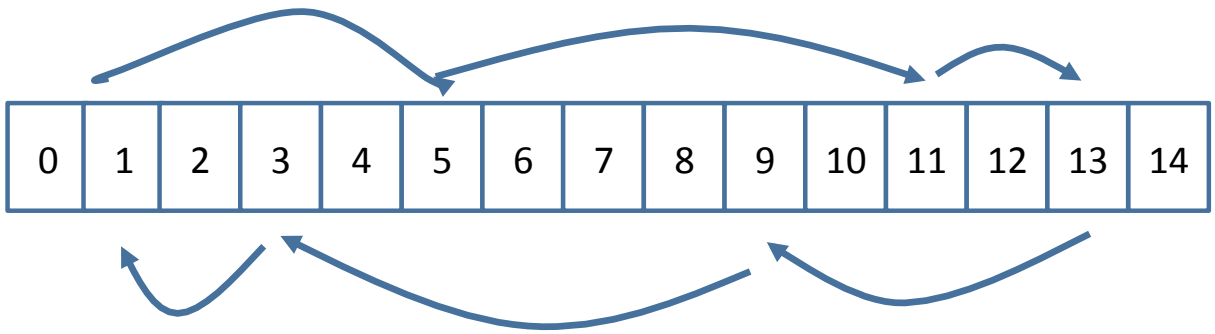
Why Full In-Place Transposition on GPU?

- Transposition is obviously memory bound
  - Sheer raw memory bandwidth on GPU
- Big data sets
  - May not be able to afford the 2X spatial overhead for out-of-place transposition

Why Fast In-Place Transposition is Difficult on GPUs

- "Following-the-cycles"
- No locality

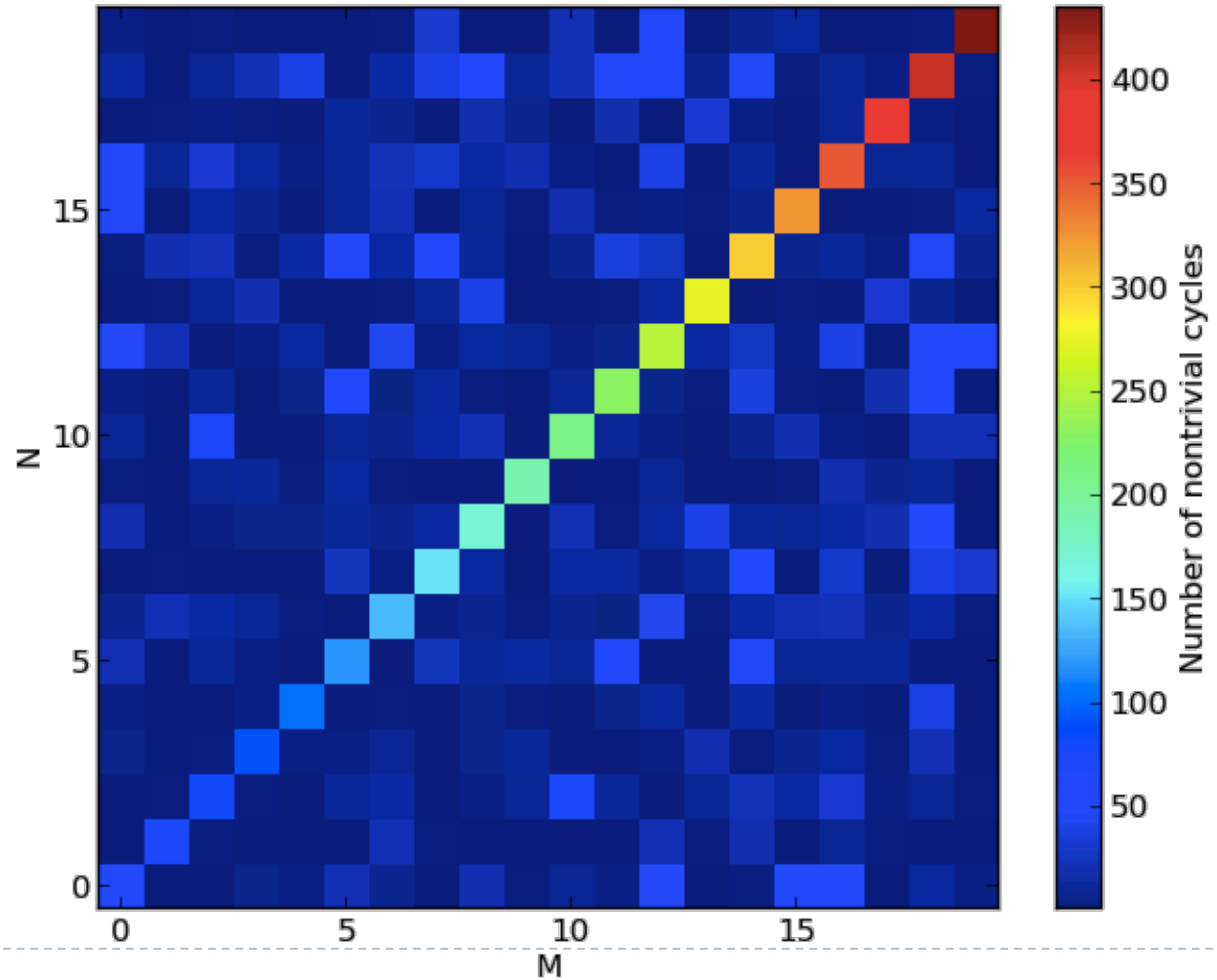
A (0,0)(1,0)(2,0)(0,1)(1,1)(2,1) (0,2)(1,2)(2,2)(0,3)(1,3)(2,3)(0,4)(1,4)(2,4)



A' (0,0)(1,0)(2,0)(3,0)(4,0)(0,1) (1,1)(2,1)(3,1)(4,1)(0,2)(1,2)(2,2)(3,2)(4,2)

Why Fast In-Place Transposition is Difficult on GPUs

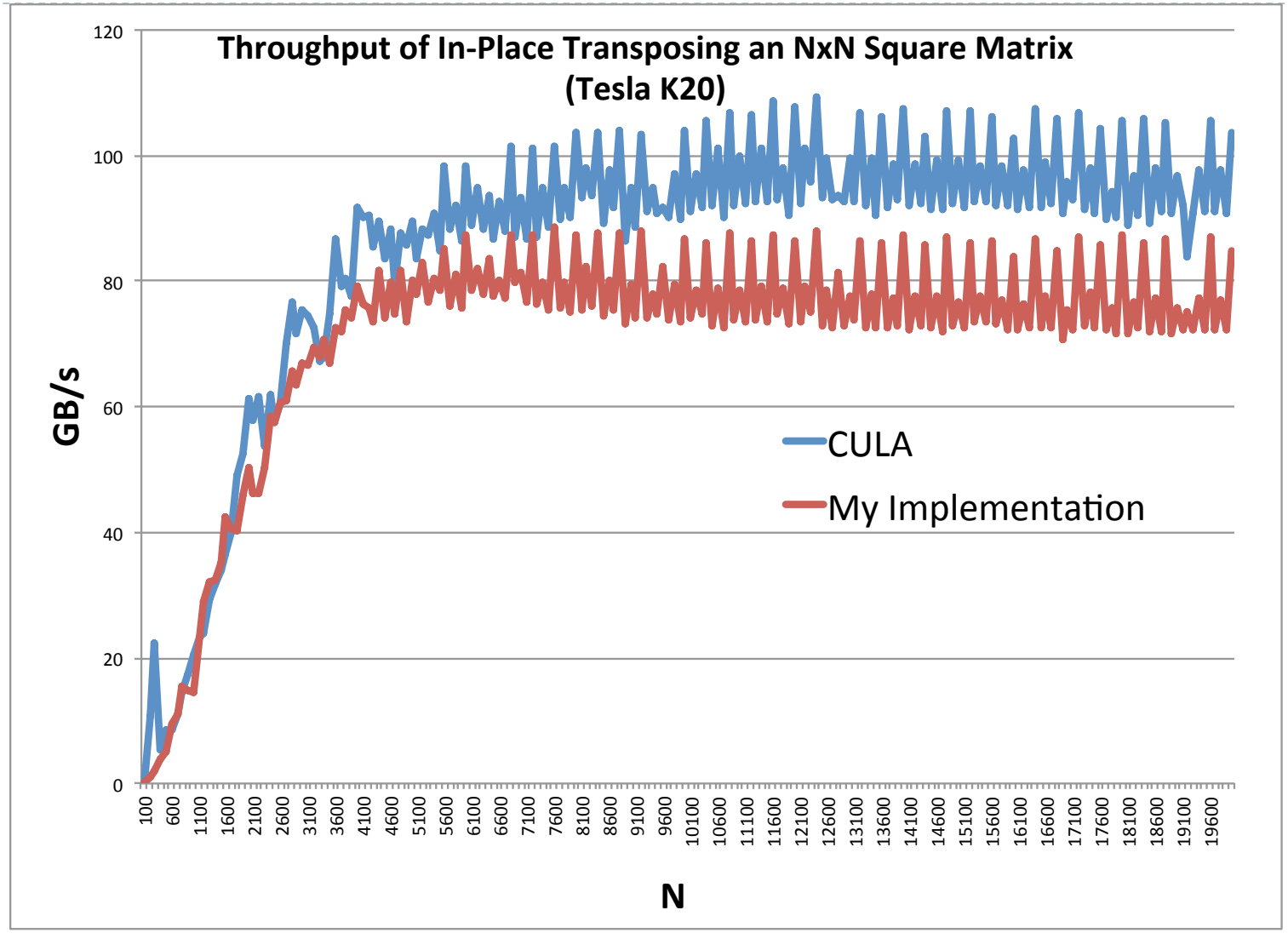
- The amount of cycles is not enough (except square matrices)



Plan

- Let us look at transposing square matrices first
- Then near-square matrices, if we can pad them
- General transposition

In-Place Transposition of Square Matrices

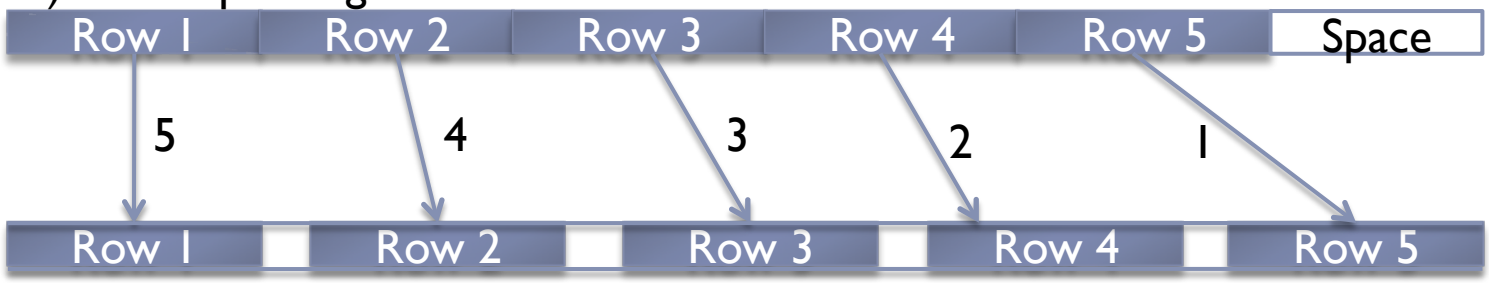


Near Square Matrices

- Case I: adding more rows (trivial for row-major layout)

- Case II: adding more columns
  - Observation: when there are spaces available, multiple rows can be moved in parallel without additional temporary storage

a) Before padding

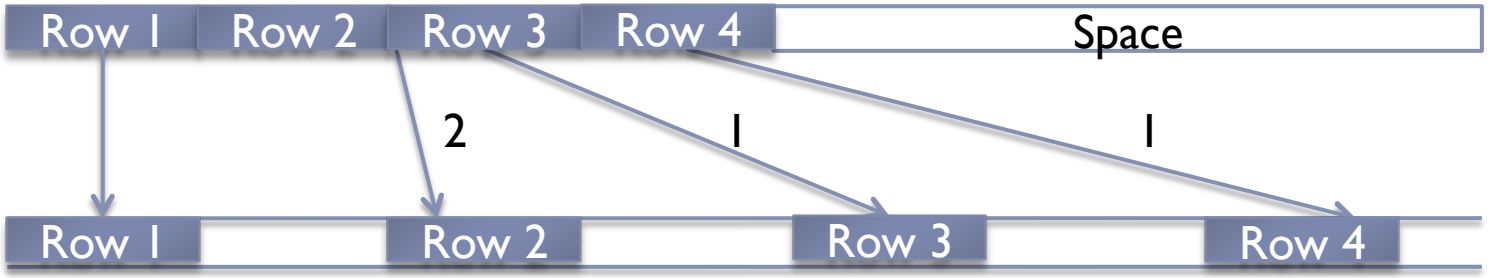


b) After padding

  - Starting from the last few rows, and then move the next few rows, so on so forth

Parallel In-Place Padding

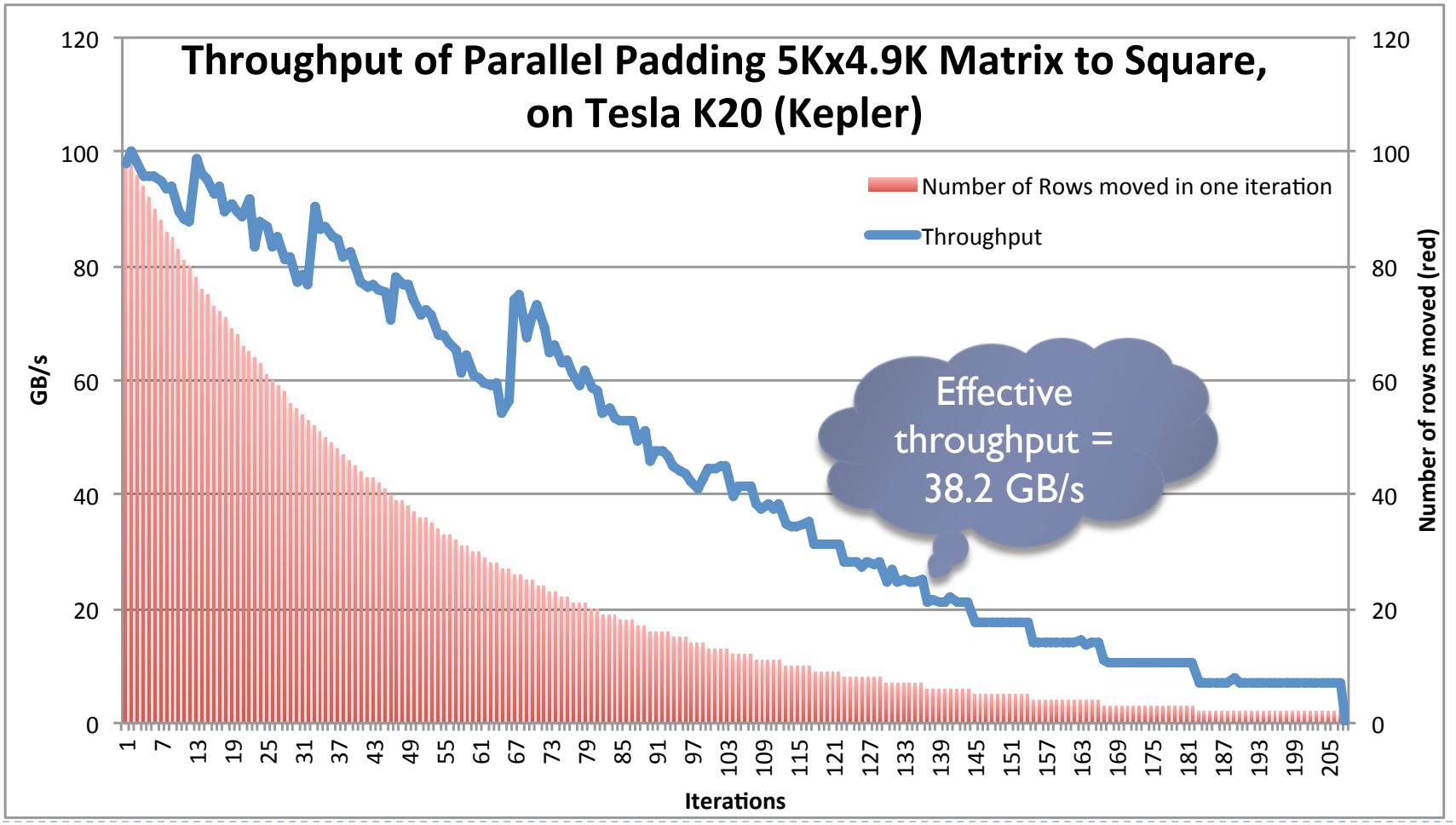
- If space is enough we can move multiple rows w/o overlapping with other rows asynchronously



- In this case, Row 3 and Row 4 can be moved in parallel, but Row 2 needs to be moved in another iteration
  - MovableRows = ((TotalRows − RowsMoved) × C)/RowSize

Parallel Padding for Near-Square Matrices

- Throughput is limited by spaces available for padding



Plan

- Let us look at transposing full square matrices
  - Reached 45~51% peak on Kepler (208GB/s peak mem b/w)
- Maybe near-square matrices if we can pad the matrix
  - Parallelism can be limited.
- General transposition

Improving Locality by Tiled Transposition

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35
36	37	38	39	40	41
42	43	44	45	46	47

~~transpose~~

moving small data pieces over large distances

0	6	12	18	24	30	36	42
1	7	13	19	25	31	37	43
2	8	14	20	26	32	38	44
3	9	15	21	27	33	39	45
4	10	16	22	28	34	40	46
5	11	17	23	29	35	41	47

Improving Locality by Tiled Transposition

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35
36	37	38	39	40	41
42	43	44	45	46	47

0	6	12	18	24	30	36	42
1	7	13	19	25	31	37	43
2	8	14	20	26	32	38	44
3	9	15	21	27	33	39	45
4	10	16	22	28	34	40	46
5	11	17	23	29	35	41	47

Step 1: move large chunks over large distances

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35
36	37	38	39	40	41
42	43	44	45	46	47

transpose

0	1	2	6	7	8	12	13	14	18	19	20	24	25	26	30	31	32	36	37	38	42	43	44
3	4	5	9	10	11	15	16	17	21	22	23	27	28	29	33	34	35	39	40	41	45	46	47

8x6 matrix as 8x2x3 array. Transpose to 2x8x3 array

Improving Locality by Tiled Transposition

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35
36	37	38	39	40	41
42	43	44	45	46	47

0	6	12	18	24	30	36	42
1	7	13	19	25	31	37	43
2	8	14	20	26	32	38	44
3	9	15	21	27	33	39	45
4	10	16	22	28	34	40	46
5	11	17	23	29	35	41	47

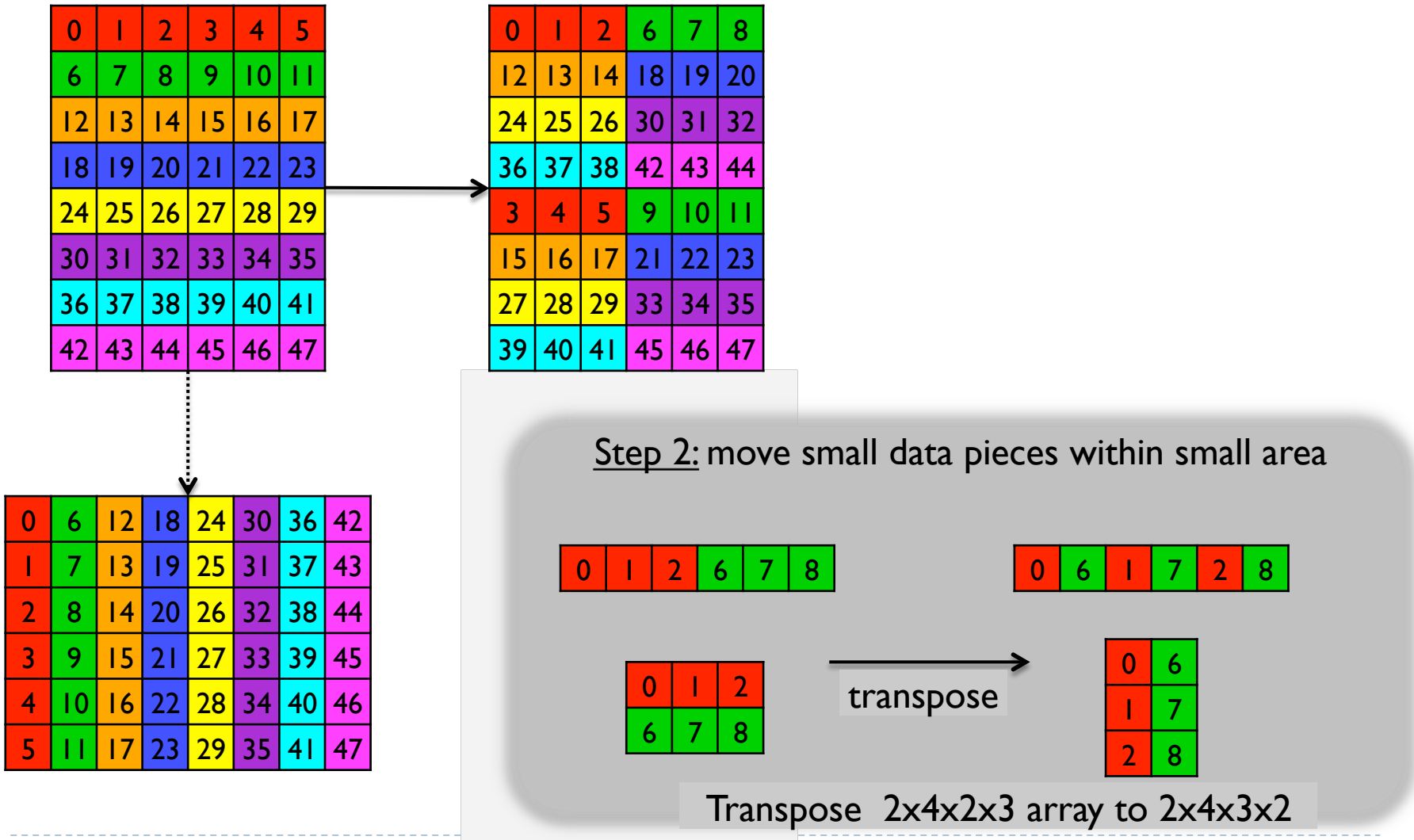
Step 1: move large chunks over large distances

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35
36	37	38	39	40	41
42	43	44	45	46	47

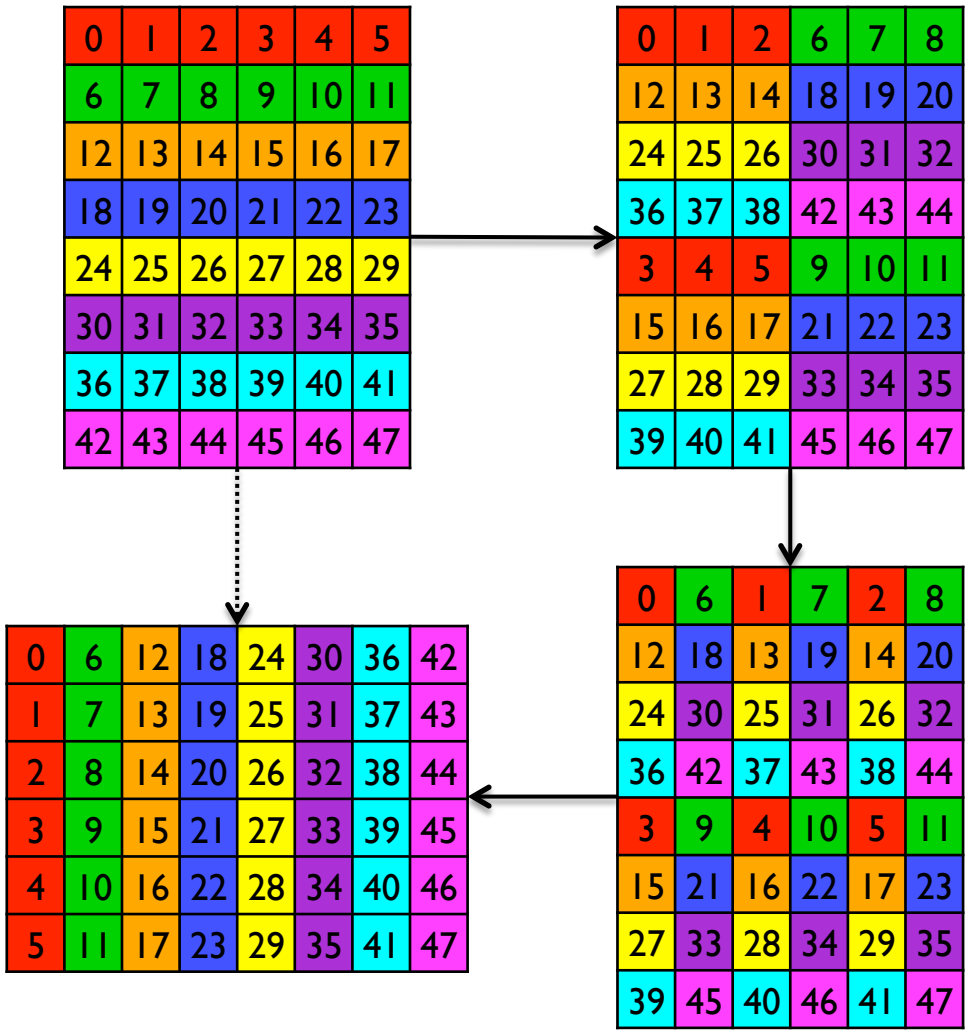
transpose

0	1	2	6	7	8
12	13	14	18	19	20
24	25	26	30	31	32
36	37	38	42	43	44
3	4	5	9	10	11
15	16	17	21	22	23
27	28	29	33	34	35
39	40	41	45	46	47

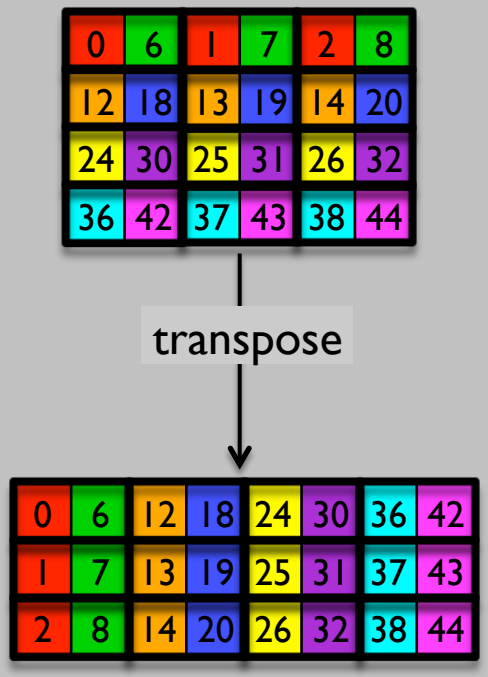
Improving Locality by Tiled Transposition



Improving Locality by Tiled Transposition



<u>Step 3:</u> move large chunks over large distances



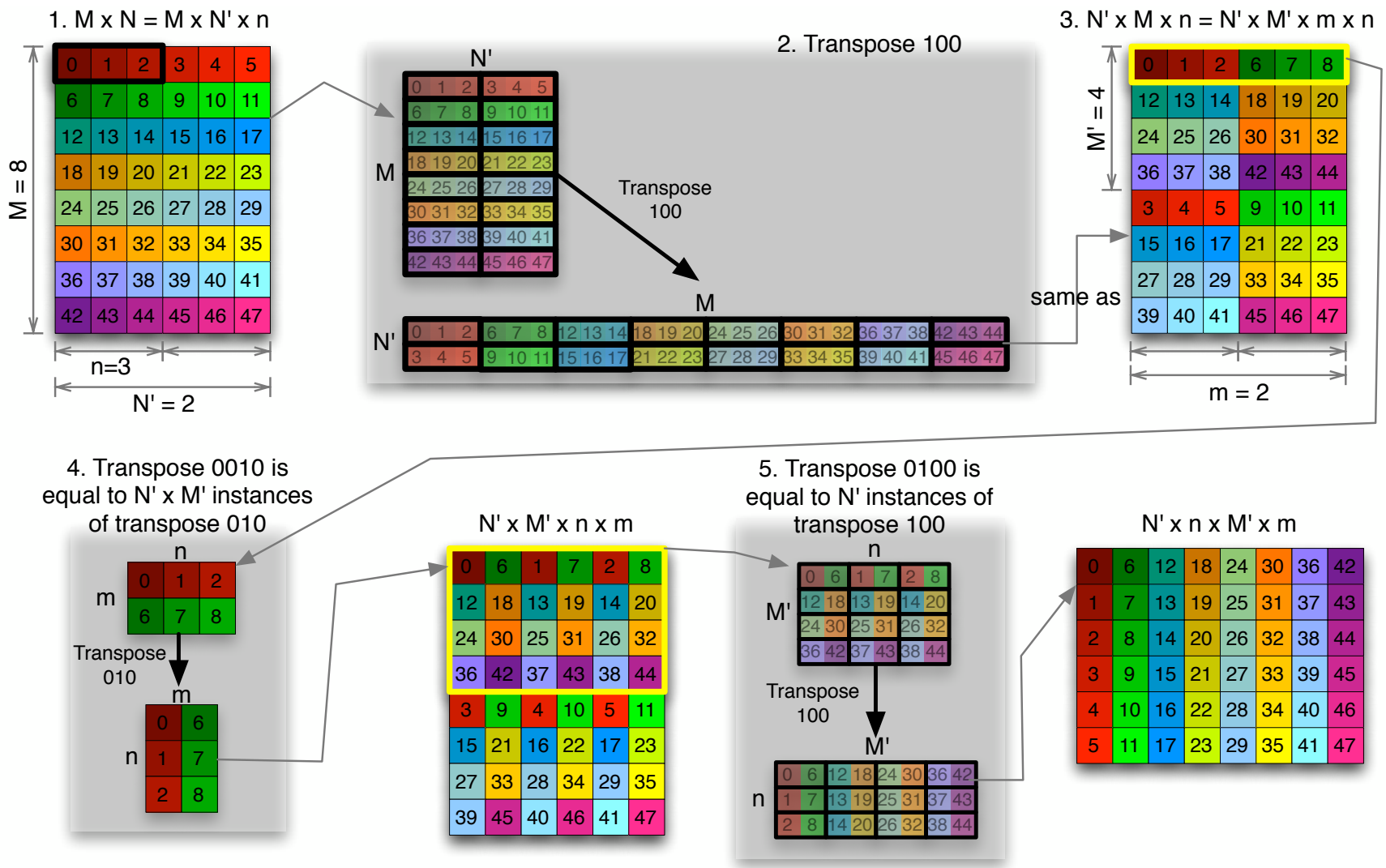
- Are you able to reproduce that sequence of transposition?

- We need a simpler way to refer to these transpositions

- We are essentially permuting dimensions
- Factorial numbering
  - A B C → B A C = $100_!$
    - Second, First, First
  - A B C → A C B = $010_!$
    - First, Second, First

Improving Locality by Tiled Transposition [Submitted to EuroPar 2013]

Assume M = M'm; N = N'n:

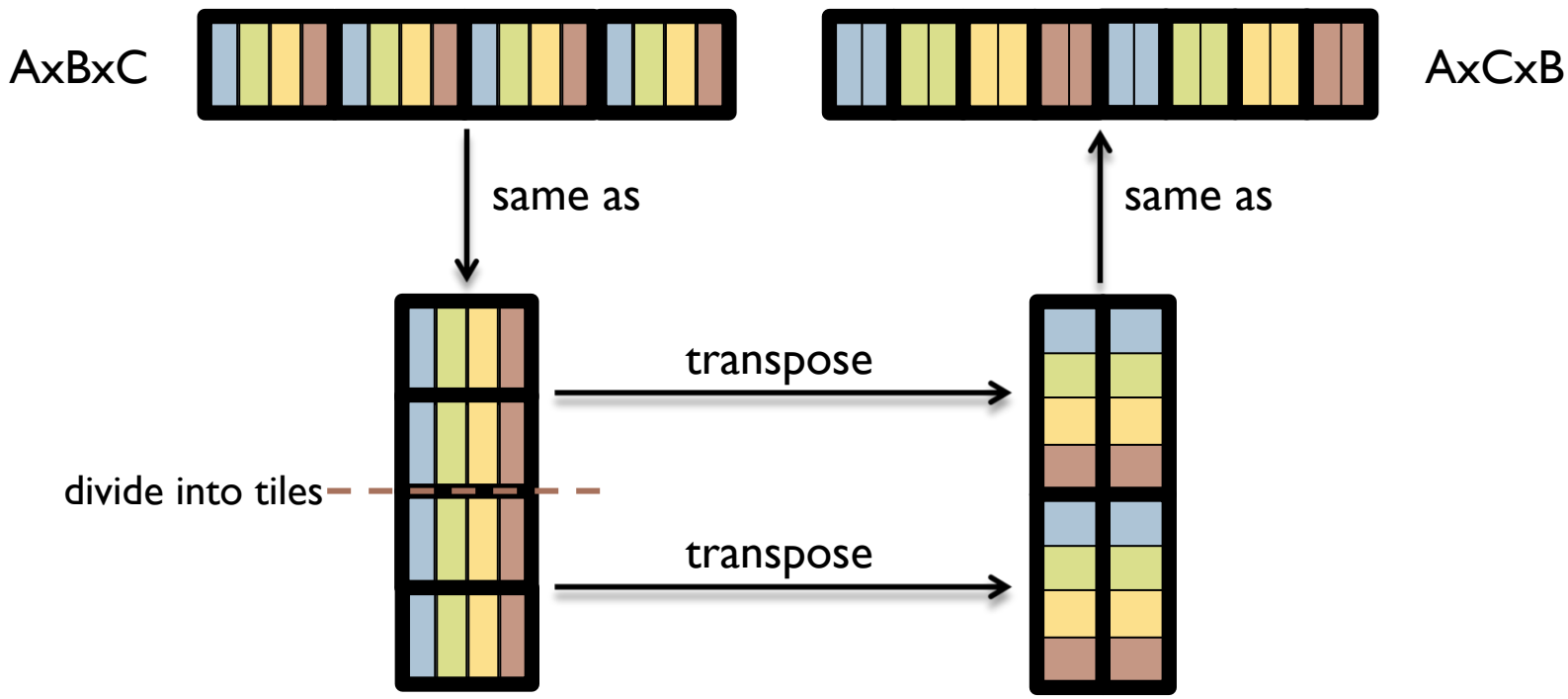


Tiled Transposition Exploits Locality

- 2 basic kind of transposition involved
  - Transposition 010!
    - E.g. Treat 8x2x3 array as 8 instances of 2x3 array
    - Transpose each of the 2x3 array
  - Transposition 100!
    - E.g. Treat 8x6 matrix as 8x2 array of super element of size 3
    - Transpose it to 2x8 array of super element of size 3

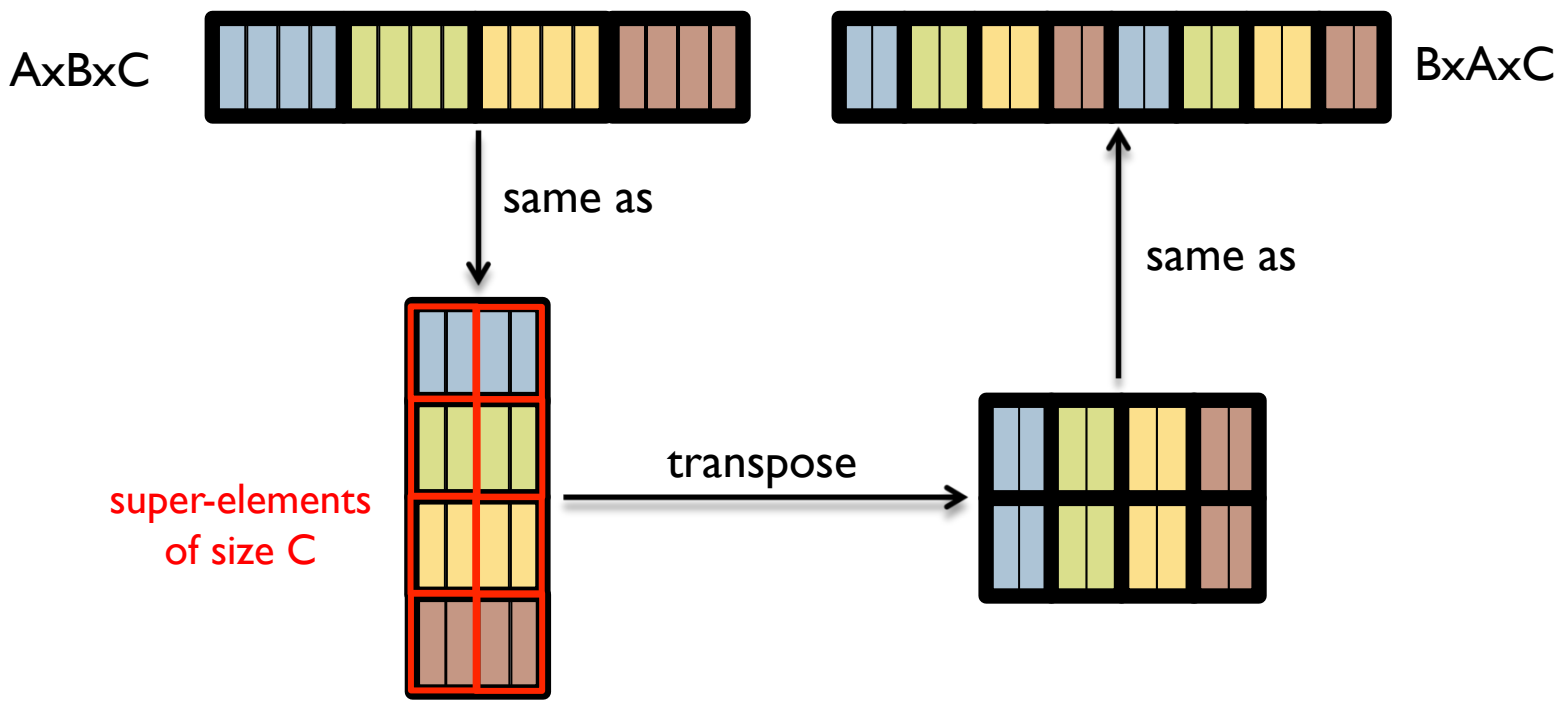
Transposition 010!

- Not too different from transposing a bunch of small tiles



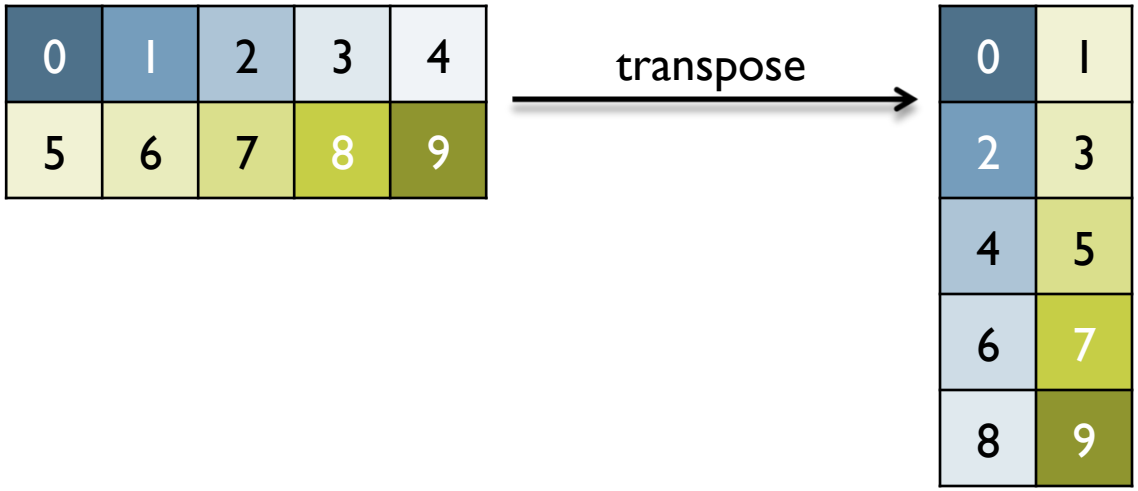
Transposition 100!

- Not too different from transposing a matrix with super-elements



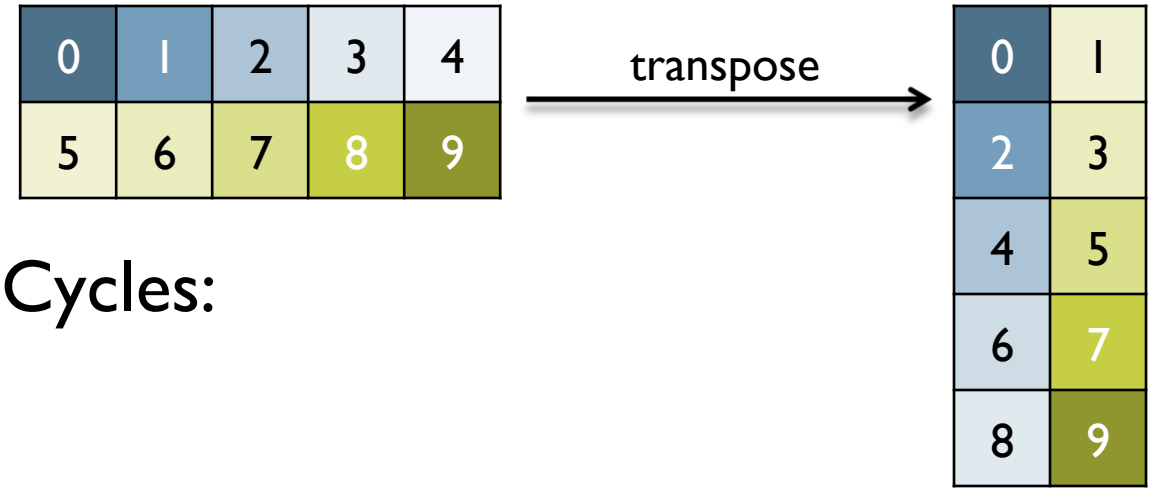
Transposition - Cycles

- Transposition is a permutation
  - A permutation can be decomposed to independent cycles of shifting



Transposition - Cycles

- Transposition is a permutation
  - A permutation can be decomposed to independent cycles of shifting



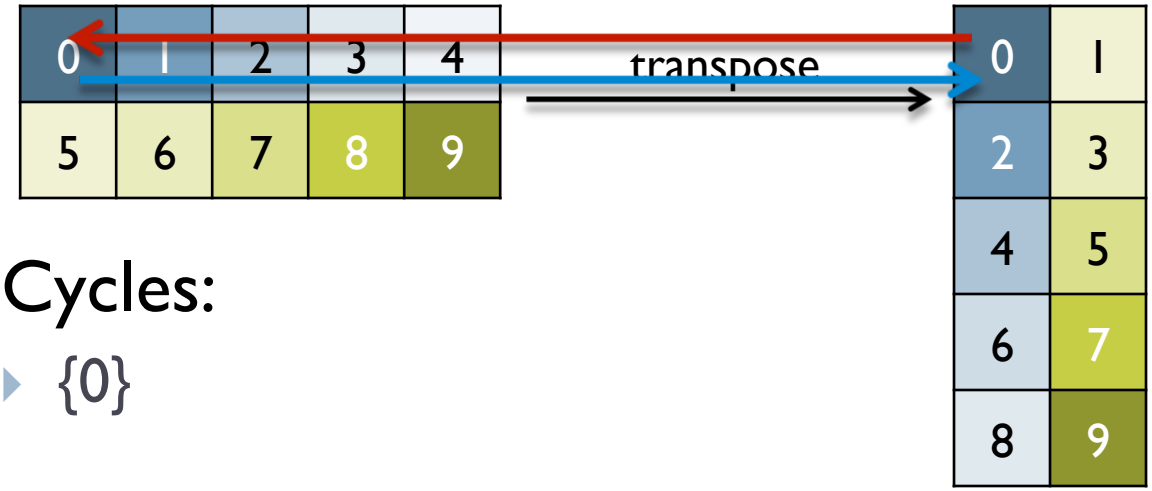
  - Cycles:

curr → (curr % N)*M + curr/N; → next

M = 2, N = 5

Transposition - Cycles

- Transposition is a permutation
  - A permutation can be decomposed to independent cycles of shifting



  - Cycles:
    - {0}

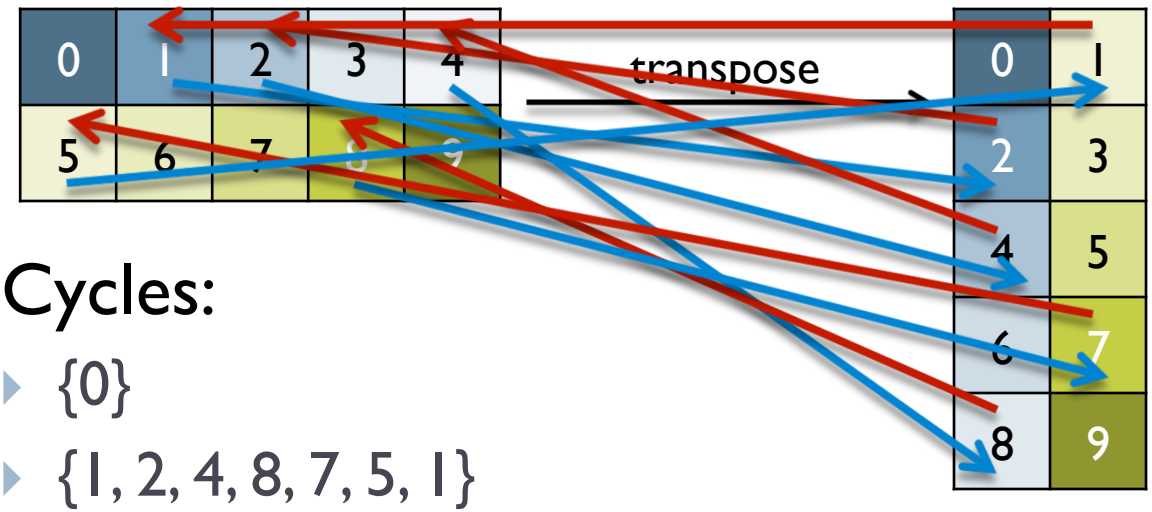
curr → (curr % N)*M + curr/N; → next

0 → 0

M = 2, N = 5

Transposition - Cycles

- Transposition is a permutation
  - A permutation can be decomposed to independent cycles of shifting



  - Cycles:
    - {0}
    - {1, 2, 4, 8, 7, 5, 1}

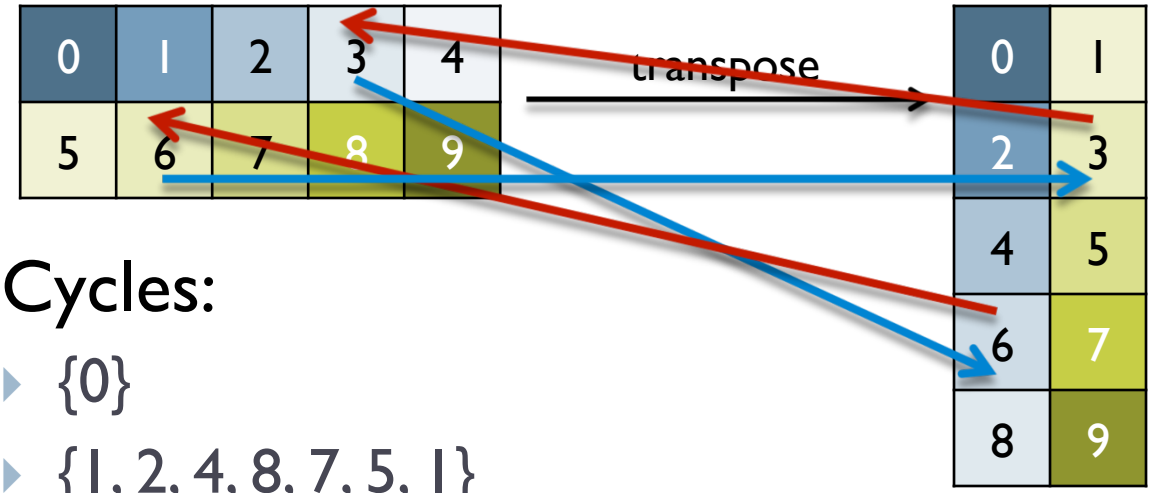
curr → (curr % N)*M + curr/N; → next

1 2

M = 2, N = 5

Transposition - Cycles

- Transposition is a permutation
  - A permutation can be decomposed to independent cycles of shifting



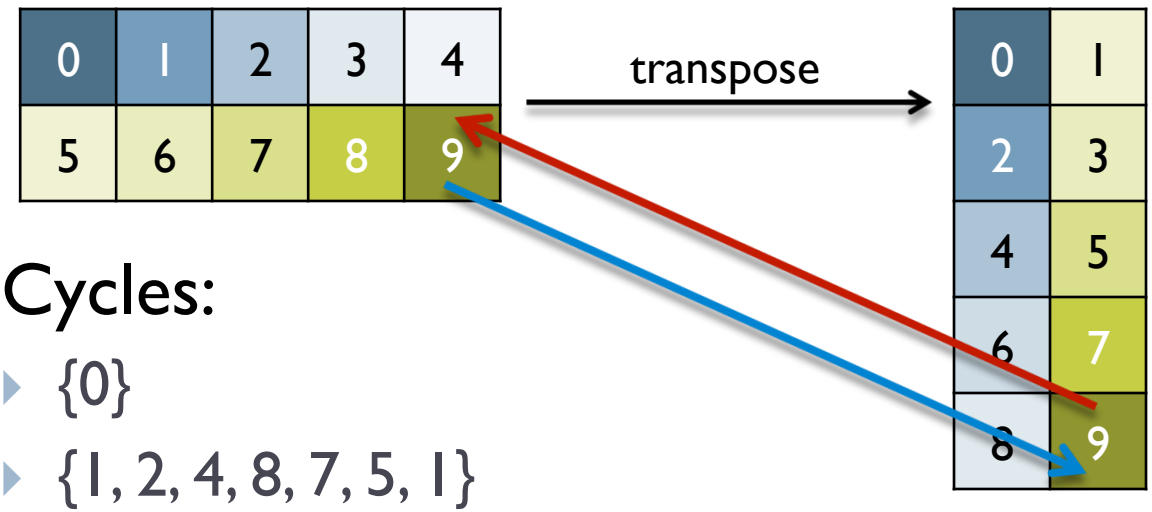
  - Cycles:
    - {0}
    - {1, 2, 4, 8, 7, 5, 1}
    - {3, 6, 3}

curr → (curr % N)*M + curr/N; → next

M = 2, N = 5

Transposition - Cycles

- Transposition is a permutation
  - A permutation can be decomposed to independent cycles of shifting



- Cycles:
  - {0}
  - {1, 2, 4, 8, 7, 5, 1}
  - {3, 6, 3}
  - {9}

curr → (curr % N)*M + curr/N; → next

M = 2, N = 5

Cycle Following – Original

- Cycles:
  - thread 0: {0}
  - thread 1: {1, 2, 4, 8, 7, 5, 1}
  - thread 2: {3, 6, 3}
  - thread 3: {9}

Imbalance

This is equivalent to a straightforward parallelism of the IPT algorithm in Gustavson et al, " In-place transposition of rectangular matrices." in PARA'06.

Cycle Following – Load Balanced

{0} {1, 2, 4, 8, 7, 5, 1} {3, 6, 3} {9}

t0 t1 t2 t3 t4

0	1	2	3	4
5	6	7	8	9

Cycle Following – Load Balanced

{0} {1, 2, 4, 8, 7, 5, 1} {3, 6, 3} {9}

t0 t1 t2 t3 t4

0	1	2	3	4
5	6	7	8	9

Cycle Following – Load Balanced

{0} {1, 2, 4, 8, 7, 5, 1} {3, 6, 3} {9}

t0 t1 t2 t3 t4

0	1	2	3	4
5	6	7	8	9

Cycle Following – Load Balanced

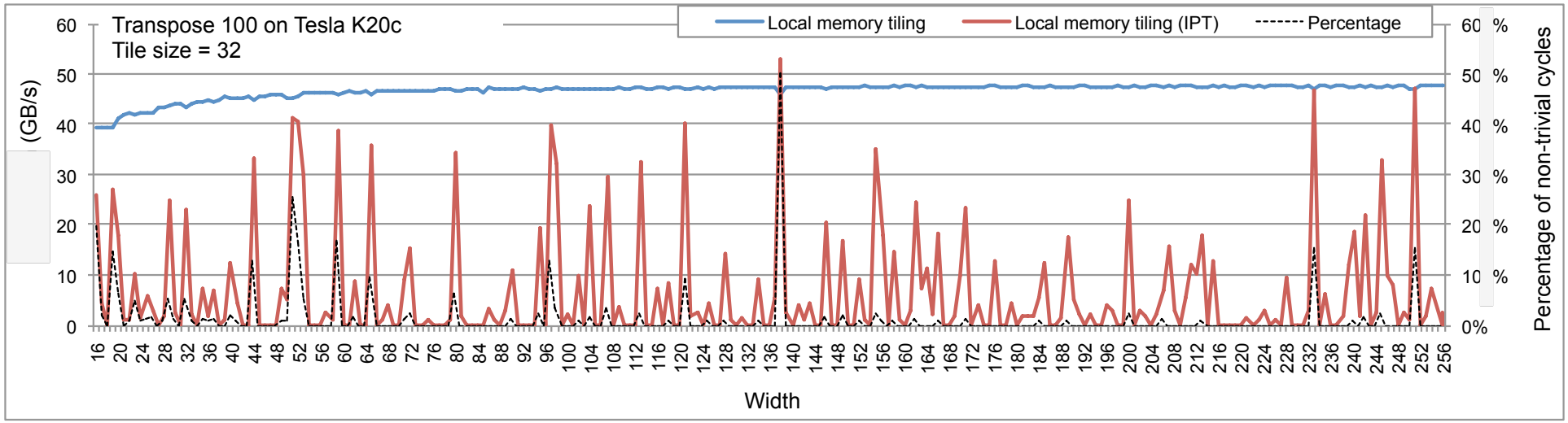
{0} {1, 2, 4, 8, 7, 5, 1} {3, 6, 3} {9}

t0 t1 t2 t3 t4

0	1	2	3	4
5	6	7	8	9

Transposition 100!

- Original vs Load Balanced



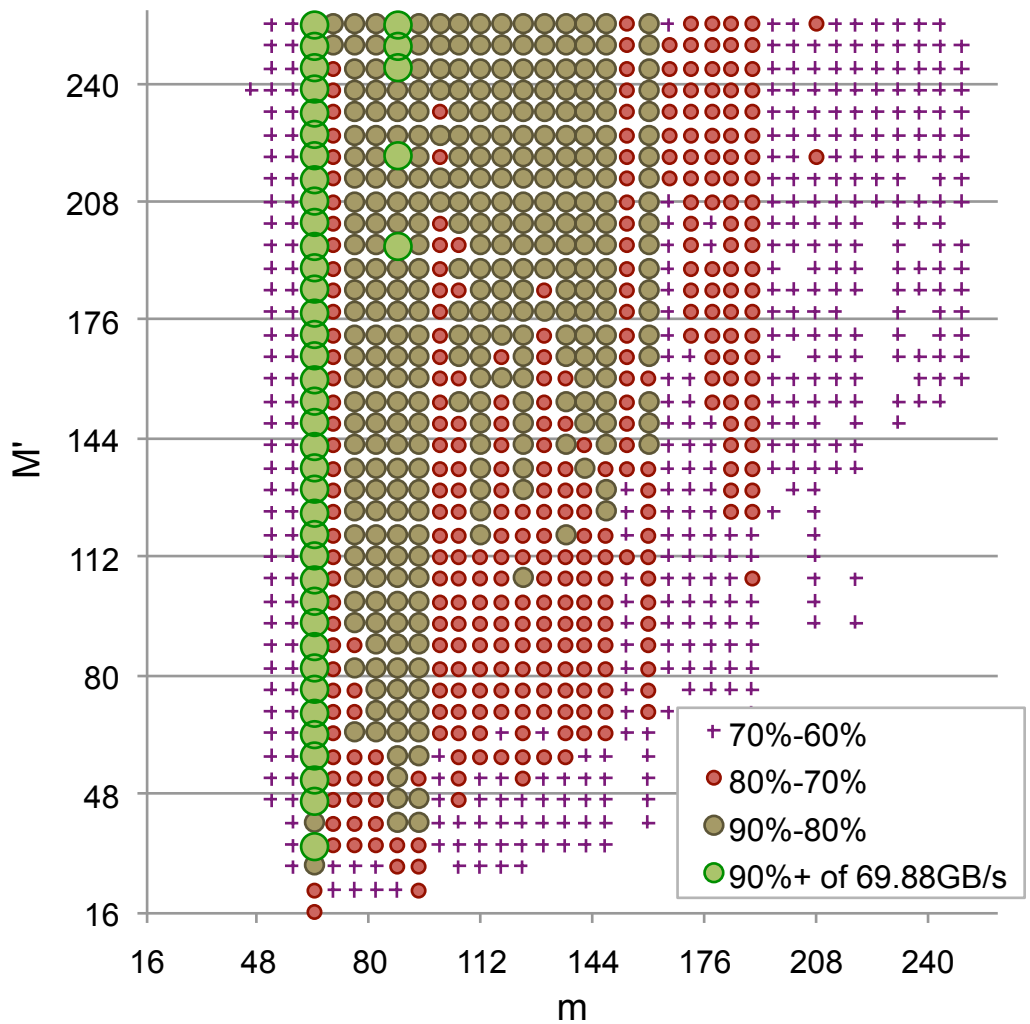
Some Performance Numbers

- Transposing an MxN matrix on an NVIDIA Tesla K20
  - Un-tiled transposition gives 1.5GB/s throughput
  - <2% of peak

- Tiled approach treat the MxN matrix as a 4D array, M' x m x N' x n
  - Applying fast primitives that either
    - Transpose many instances of small matrix involving m and n only
      - ~100GB/s if tile is small
      - We want to somehow apply this only when small matrix = m x n or n x m
    - Transpose one matrix of super-elements either m or n sized
      - ~70GBs, but depending on how big the super-element is
      - We want to somehow apply this only when element size = m or n

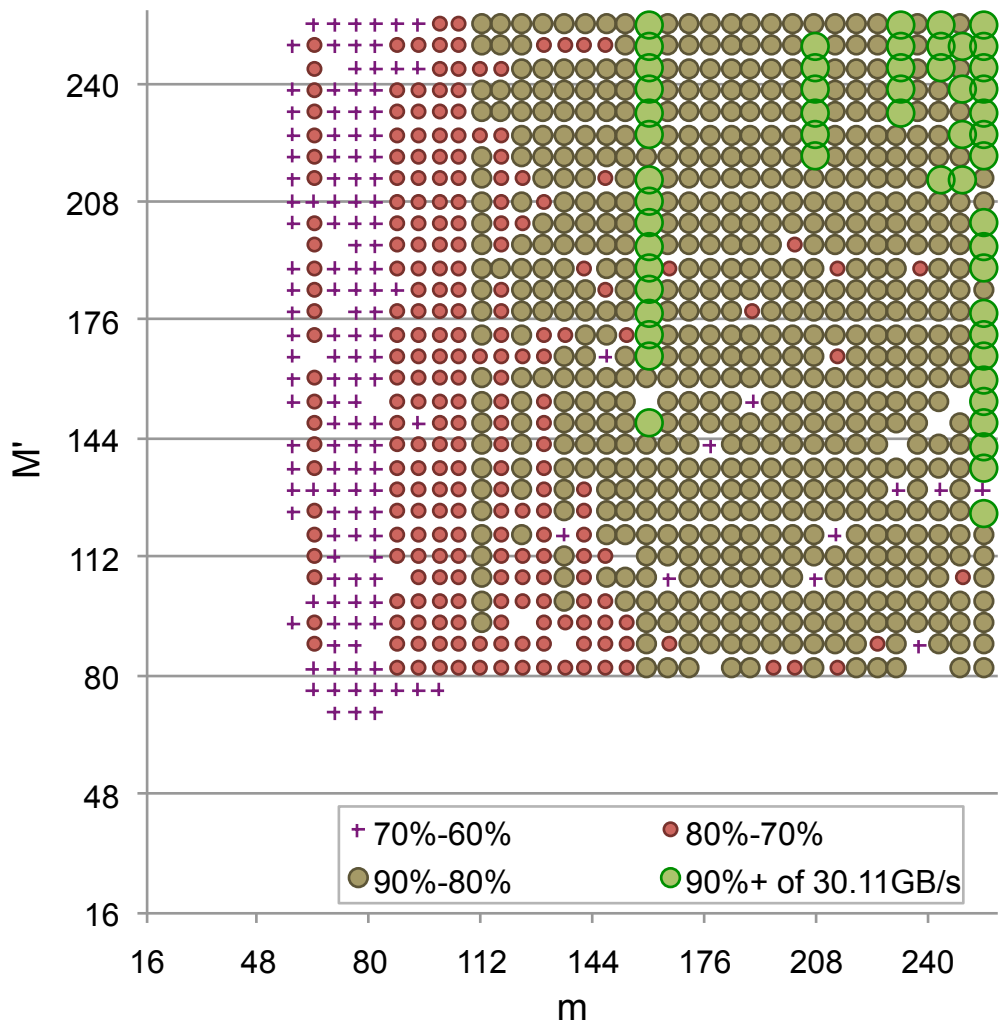
Performance of Transposition 100!

- Transposing an NxM'xm array to M'xNxm array on Tesla K20 (Kepler)
  - Peak = 208GB/s
  - ~33%



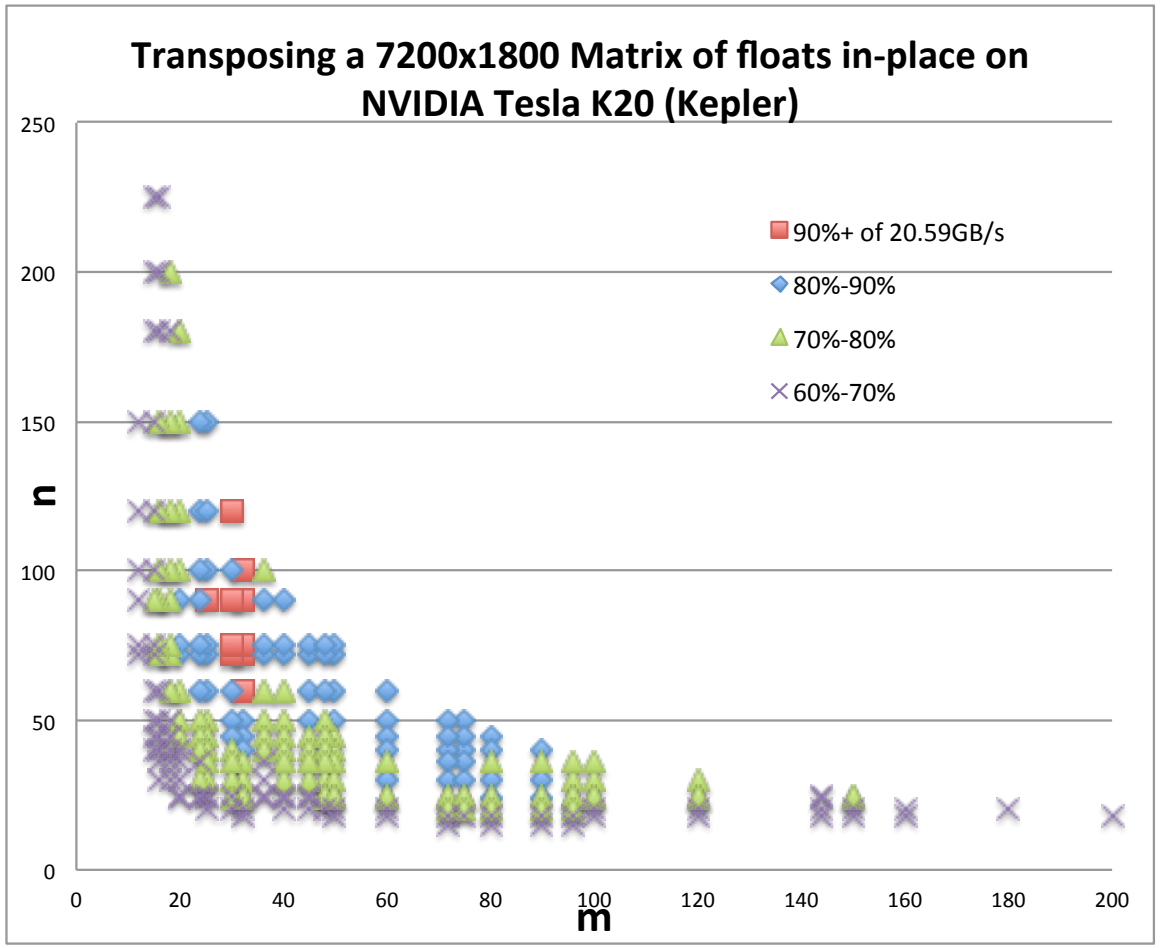
Performance of Transposition 100!

- Transposing an NxM'xm array to M'xNxm array on AMD HD7750 (Cape Verde)
  - Peak = 72GB/s
  - ~46%



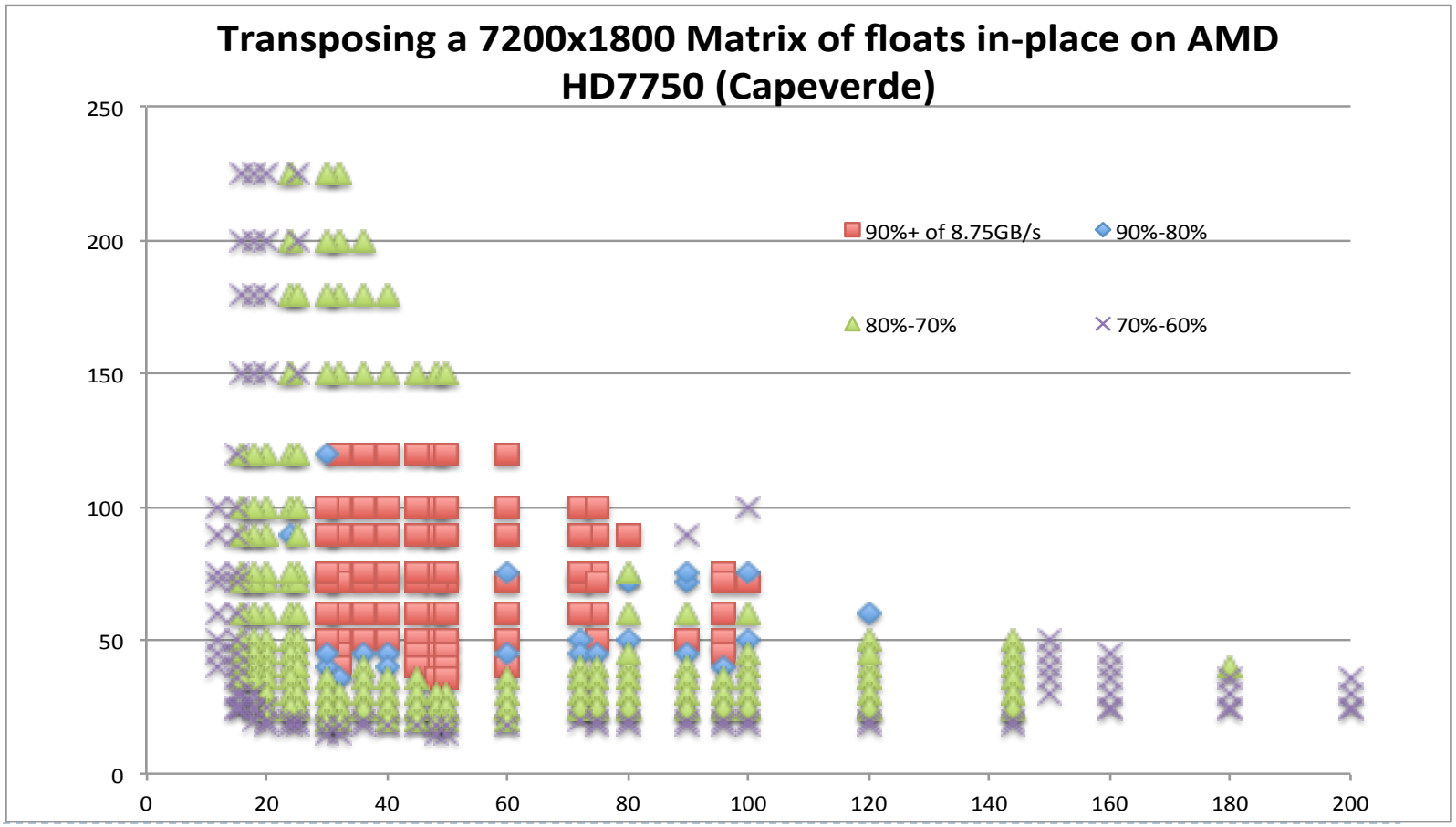
Results: Choosing a Tile Size

- 80%+ performance can be obtained with statically-chosen tile sizes
  - Search space = 1855 candidate tile sizes



Results: Choosing a Tile Size

- 80%+ performance can be obtained with statically-chosen tile sizes



Results: vs. Padding

- Padding/Square Transposition involves:
  - Padding: ~38GB/s each for large padding factor
  - Square transposition. ~80GB/s
  - Packing: ~38GB/s (estimated)
  - Overall throughput ~15GB/s
- 3-Stage approach
  - ~20GB/s with good m and n

Summary

- A full transposition algorithm library for the GPU is presented
  - First known in-place fast rectangular transposition for the GPUs
- Competitive performance even for near square matrices

Library

- We open-source the transposition library so that people don't need to reinvent the wheel
  - OpenCL (most up-to-date), Python (PyOpenCL), CUDA
  - Mathematica version under working
  - https://bitbucket.org/ijsung/libmarshal/wiki/Home

The user API is declared in

```
#include "cl_marshal.h"
```

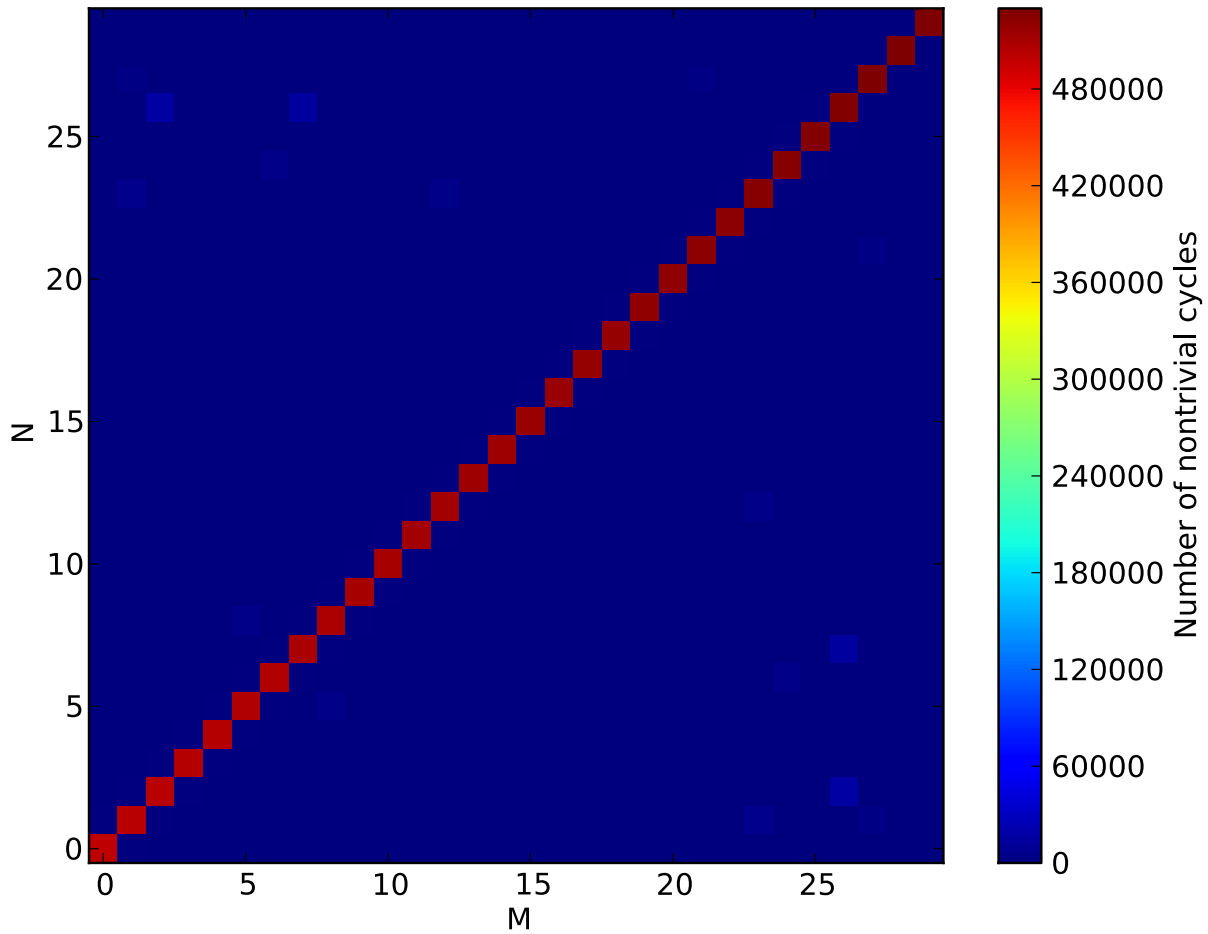
Full In-Place Transposition on GPU

```
extern "C" bool cl_transpose(cl_command_queue queue, cl_mem src, int A, int a, int B, int b);
```

This function implements in-place transposition of a Aa by Bb float matrix of an OpenCL buffer src. Returns false if there is no error.

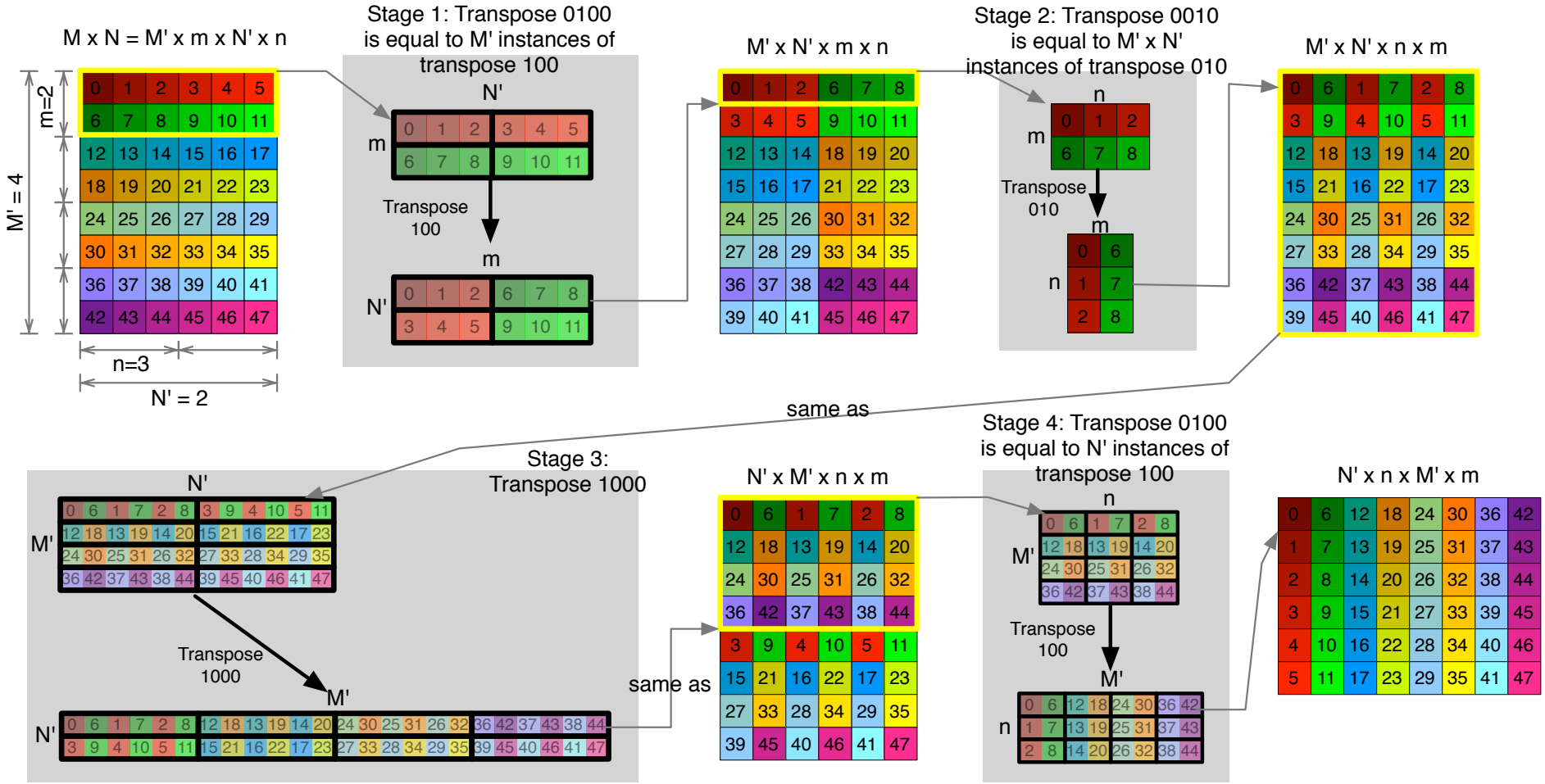
- Questions?

Heat-map of Cycles at (1K+M, 1K+N)



Why is This New?

- Prior art [Gustavson 2009, 2011; Karlsson 2006]



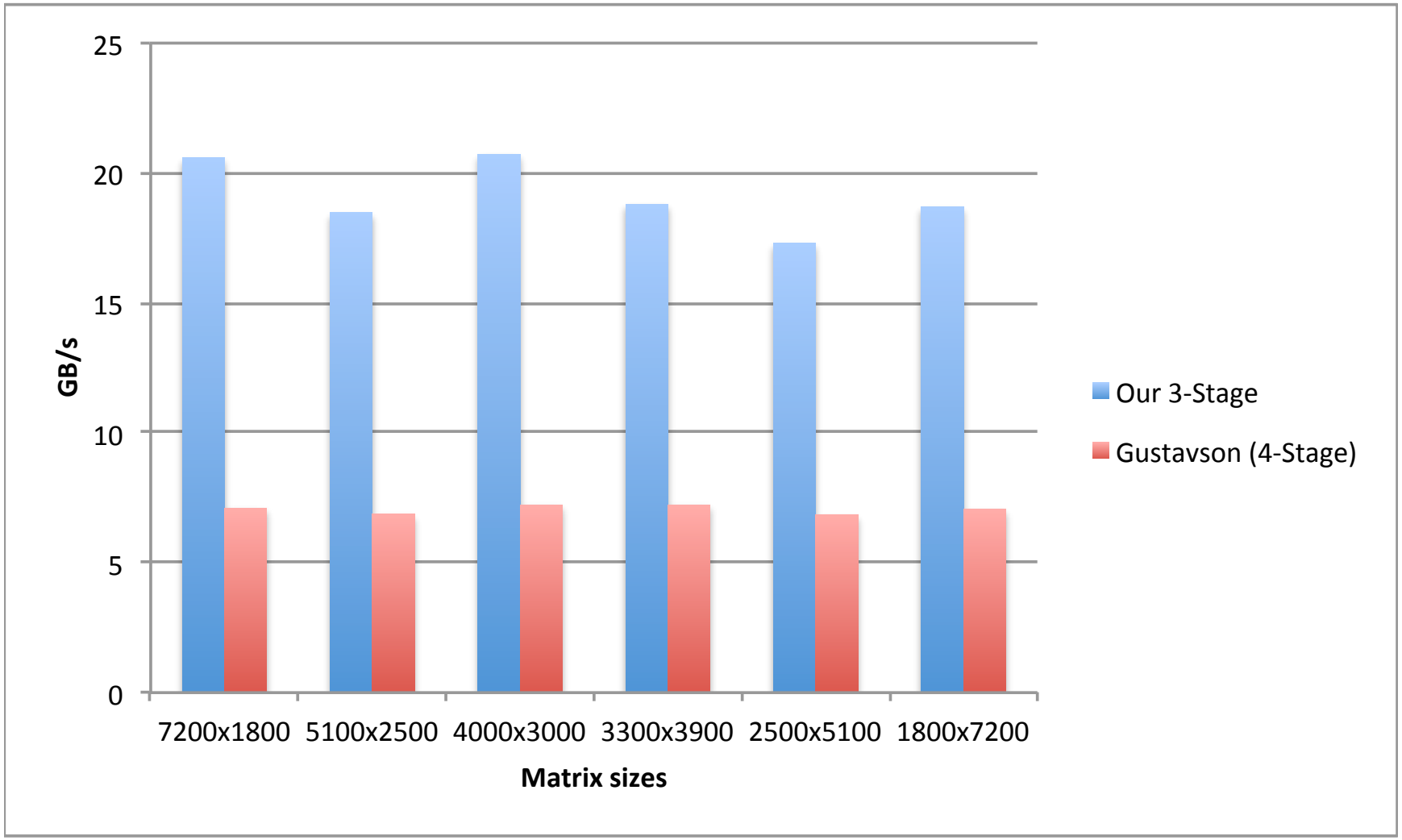
Comparison of Methods, Using a 7200x180 Example

- Prior art [Gustavson2009, 2011; Karlsson 2006] employs a 4-step approach for multicores
  - Tile sizes allowed in each step:
    - For 7200x1800 matrix, best (m, n) = (16,20) on Tesla K20
    - Transposition 0100 (derived from 100!): n (15.81GB/s)
    - Transposition 0010 (derived from 010!): m x n (98.58GB/s)
    - Transposition 1000 (derived from 100!): n x m (49.21GB/s)
    - Transposition 0100 (derived from 100!): m (21.40GB/s)

Size of super-element limited by n x m

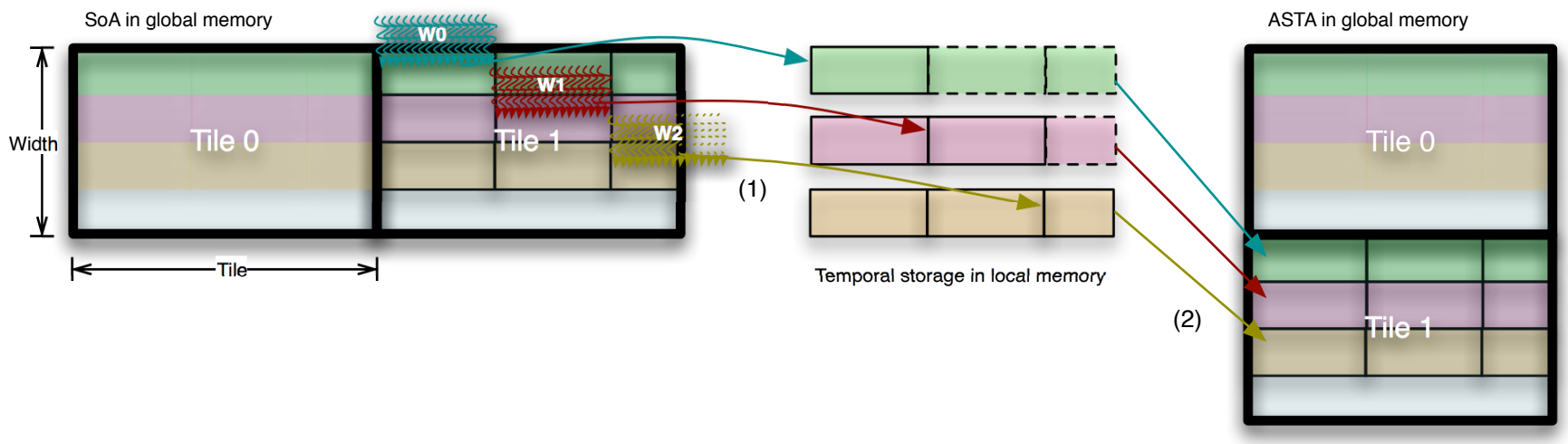
- We proposed a 3-step approach
  - Larger tile sizes allowed in transposition 100 and 0100:
    - For 7200x1800 matrix, 20.59GB/s for (m, n) = (32, 72) on Tesla K20
    - Transposition 100: m (59.55GB/s)
    - Transposition 0010: mxn (113.90GB/s)
    - Transposition 0100 (derived from 100!): n (43.89GB/s)

Results: vs. Prior Art [Gustavson 2009, 2011; Karlsson 2006]



Low-level Optimizations

- Transposition100$_!$: PTTWAC vs P-IPT (one-cycle-per-thread-block)
  - Bottleneck of PTTWAC was on barriers
  - Removed by using a warp to move a super-element instead of a thread-block

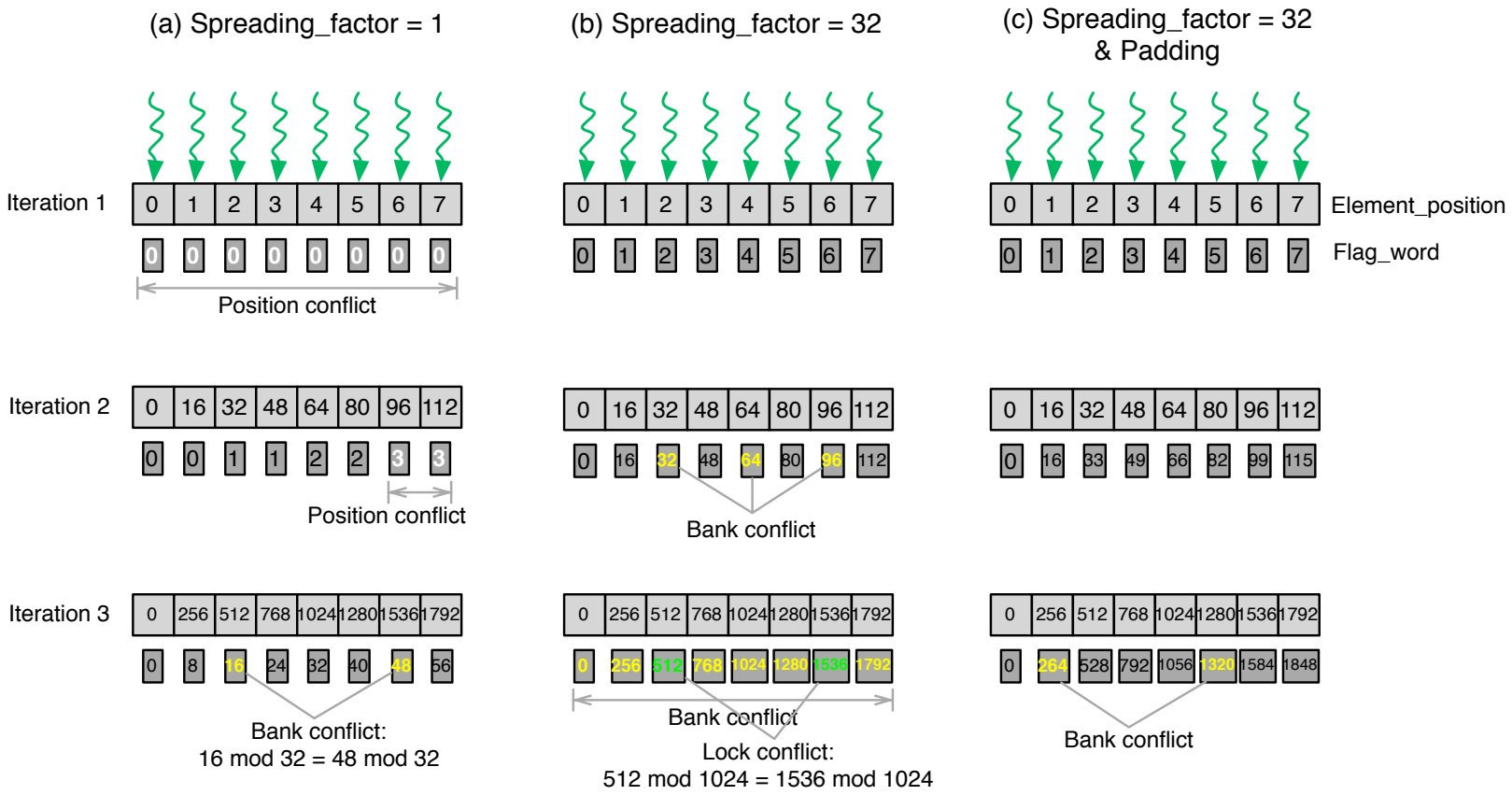


Low-level Optimizations Applied

- Transposition $010_!$: PTTWAC vs P-IPT (one-cycle-per-thread)
  - Bottleneck of PTTWAC was on atomic operations to on-chip memory

Comparing to P-IPT (Cycle Following)

- For transposition $010_!$, re-layout atomic bit flags reduces contention and improves performance



Comparing to P-IPT (Cycle Following)

- Transposition $010_!$ w/ layout adjustment for the atomic flags in on-chip memory