



sparch.mit.edu

SpArch: Efficient Architecture for Sparse Matrix Multiplication

Zhekai Zhang^{*1}, Hanrui Wang^{*1}, Song Han¹, William J. Dally²

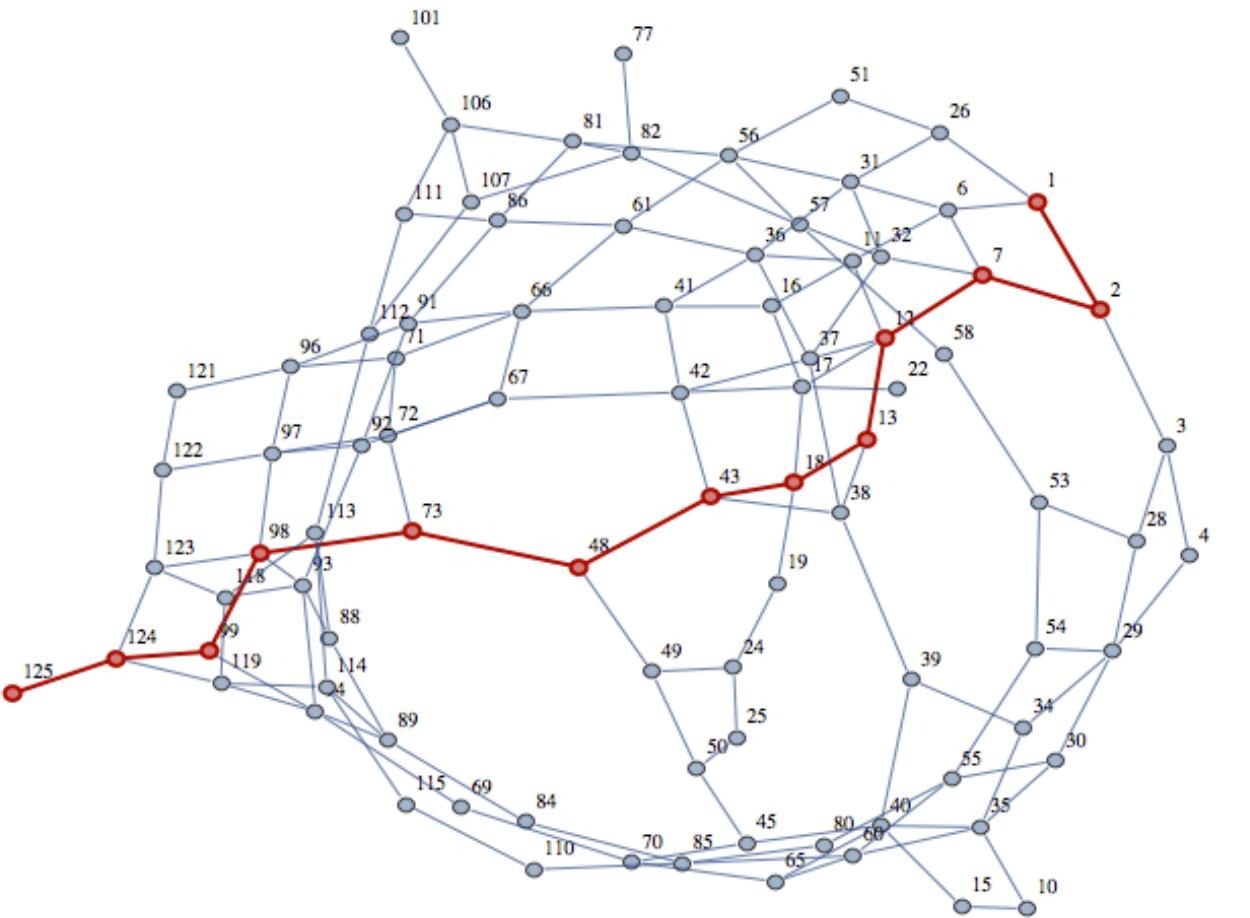
¹Massachusetts Institute of Technology

²Stanford University / Nvidia

*Equal Contributions

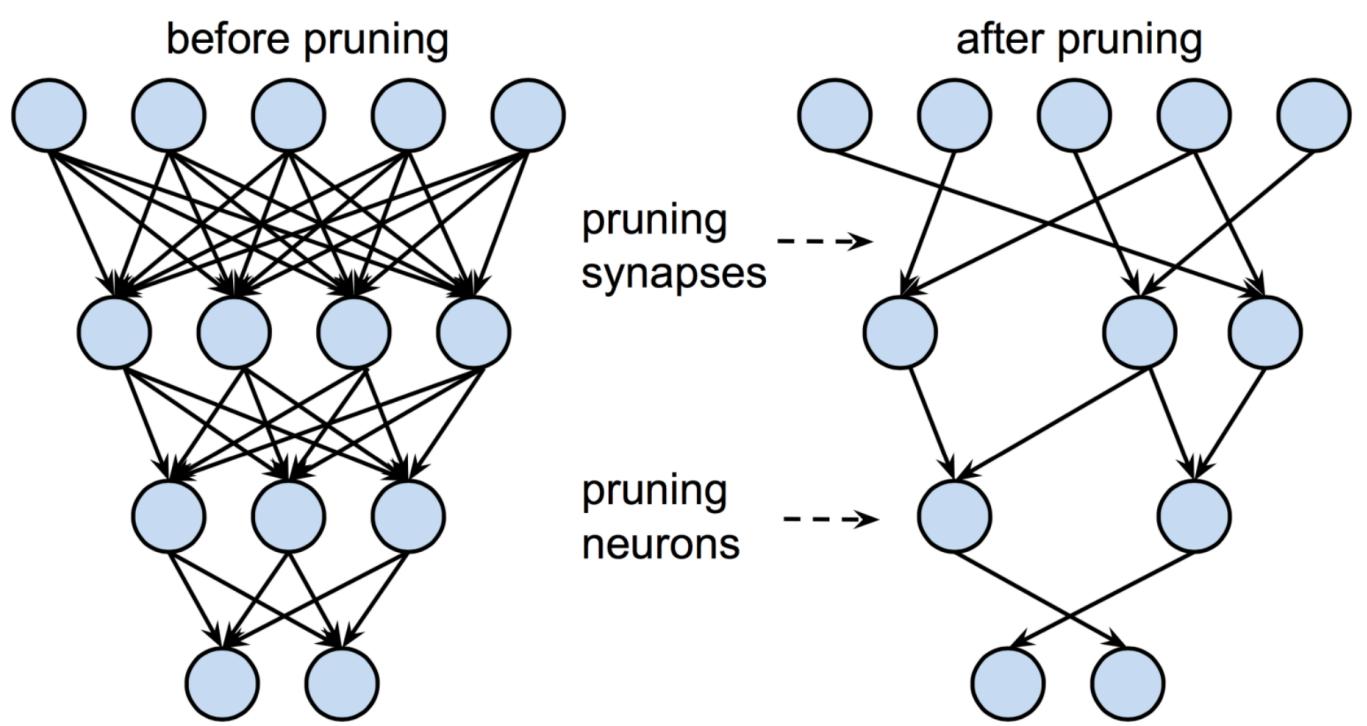
Accelerate Sparse Matrix Multiplication

Graph Computing



Dimension $\approx 4 \times 10^7$
Sparsity $\approx 8 \times 10^{-7}$

Compressed Neural Networks



Dimension $\approx 10^{11}$
Sparsity $\approx 10^{-8}$

CPUs and GPUs are Slow and Under-Utilized for SpMM

Double Precision SpMM	MKL on Intel Core i7-5930K	cuSPARSE on TITAN Xp	CUSP on TITAN Xp	Armadillo on Arm Cortex-A53
Average GFLOPS on 20 benchmarks ^{1,2}	0.560	0.595	0.631	0.00813
Theoretical GFLOPS ^{3,4,5}	289	343	343	5.47
Utilization	0.194%	0.173%	0.184%	0.149%

¹Leskovec, Jure, and Rok Sosič. "Snap: A general-purpose network analysis and graph-mining library." *ACM Transactions on Intelligent Systems and Technology (TIST)* 8.1 (2016): 1.

²Davis, Timothy A., and Yifan Hu. "The University of Florida sparse matrix collection." *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011): 1.

³<https://www.pugetsystems.com/labs/hpc/Linpack-performance-Haswell-E-Core-i7-5960X-and-5930K-594/>

⁴<https://www.techpowerup.com/gpu-specs/titan-x-pascal.c2863>

⁵<http://web.eece.maine.edu/~vweaver/group/machines.html>

Challenges

- Super-large
 - Limited on-chip memory



Dimension $\approx 4 \times 10^7$
Density $\approx 8 \times 10^{-7}$



Dimension $\approx 10^{11}$
Density $\approx 10^{-8}$

Challenges

- Super-large
 - Limited on-chip memory
- Ultra-sparse
 - Low operational intensity
 - Limited memory bandwidth

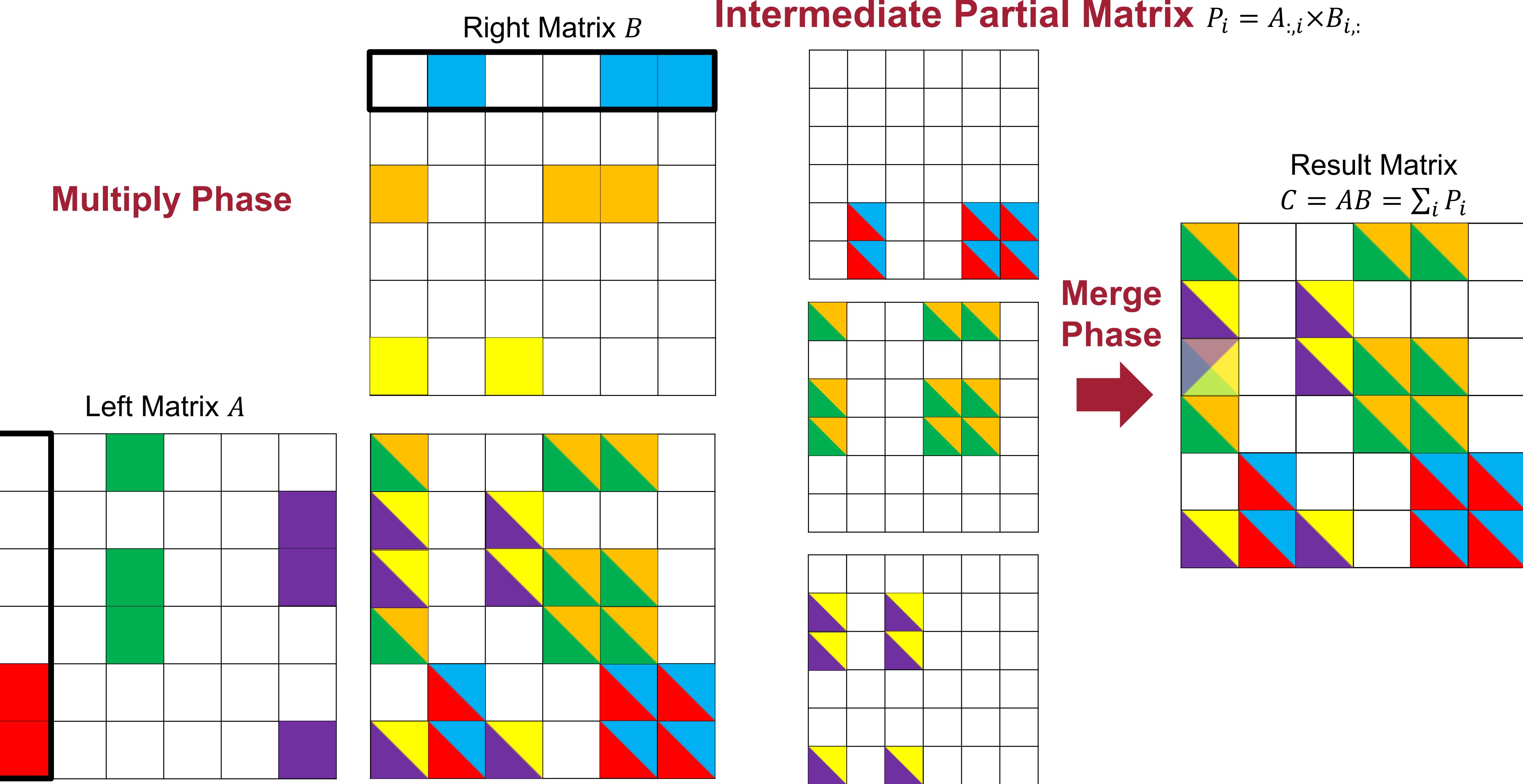


Dimension $\approx 4 \times 10^7$
Density $\approx 8 \times 10^{-7}$



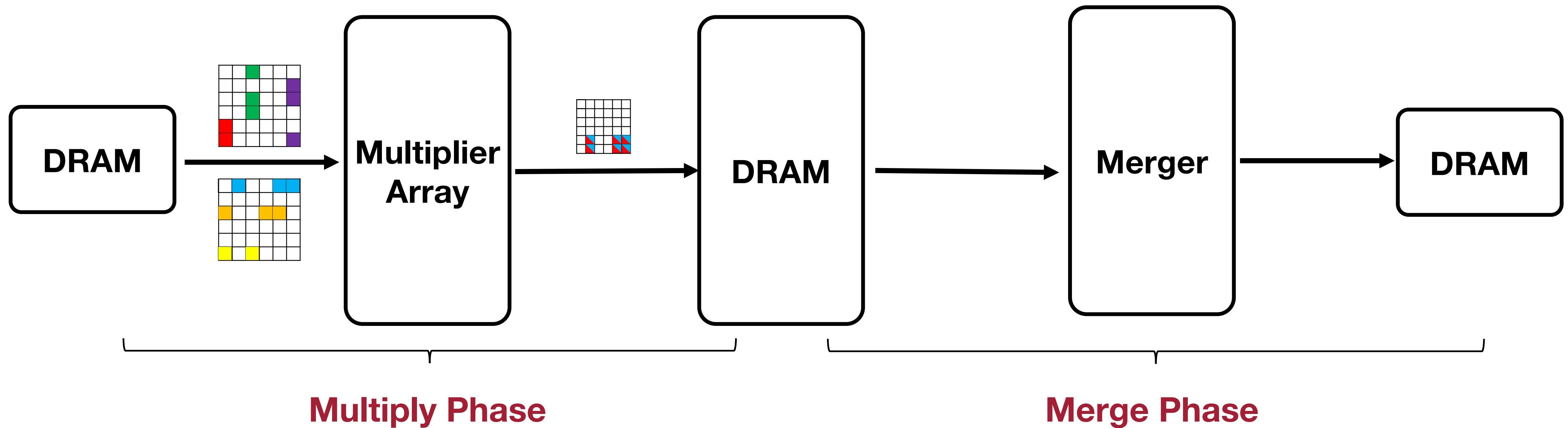
Dimension $\approx 10^{11}$
Density $\approx 10^{-8}$

Background: Outer Product

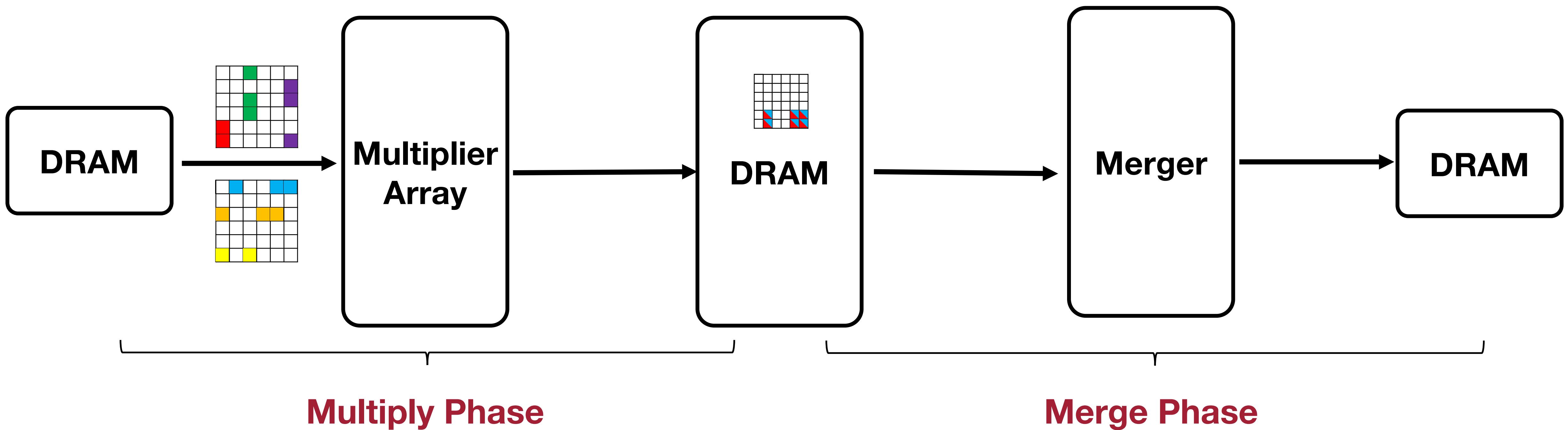


Background: Outer Product

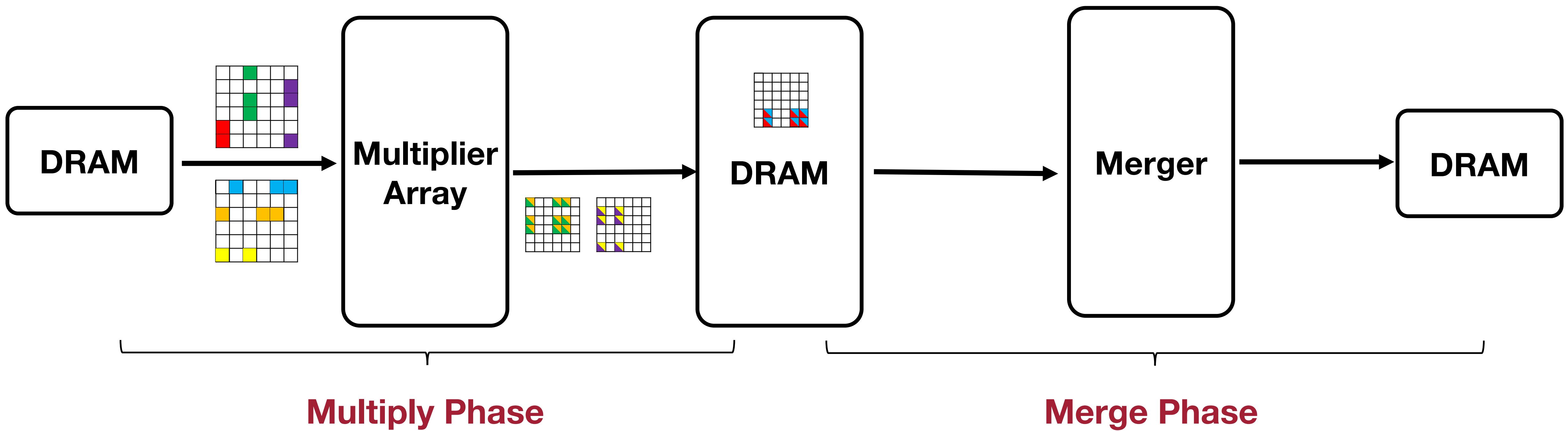
Perfect input reuse:
read input matrix only once



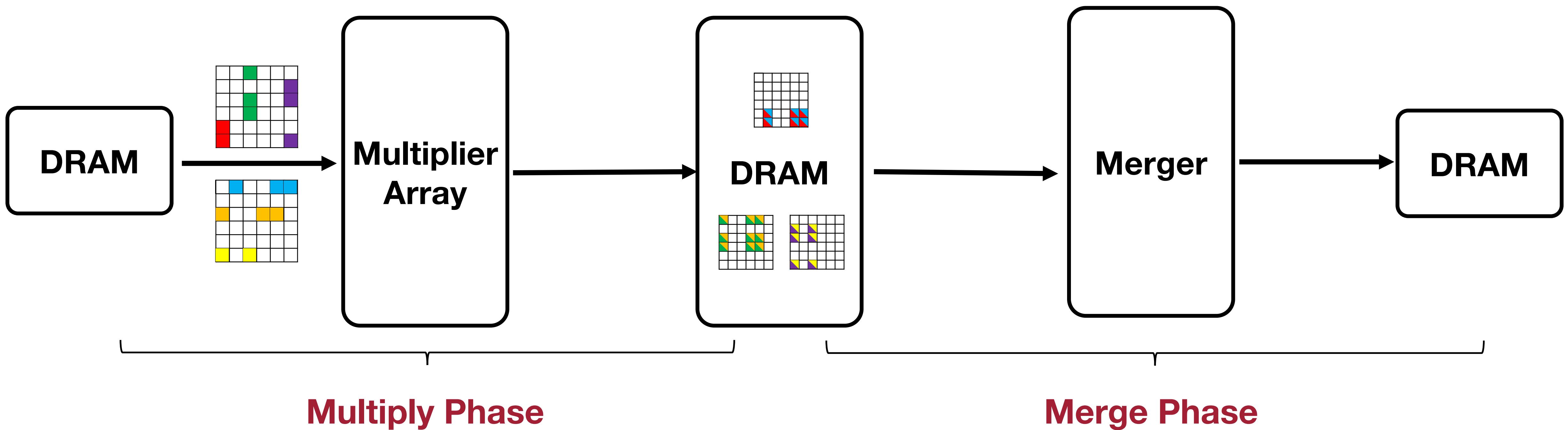
Background: Outer Product



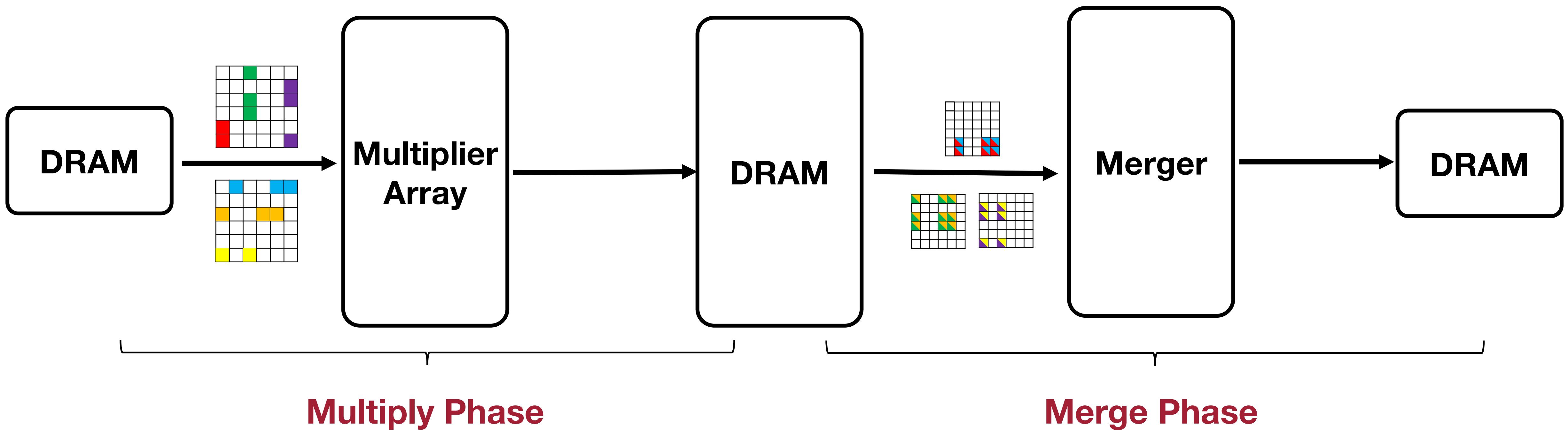
Background: Outer Product



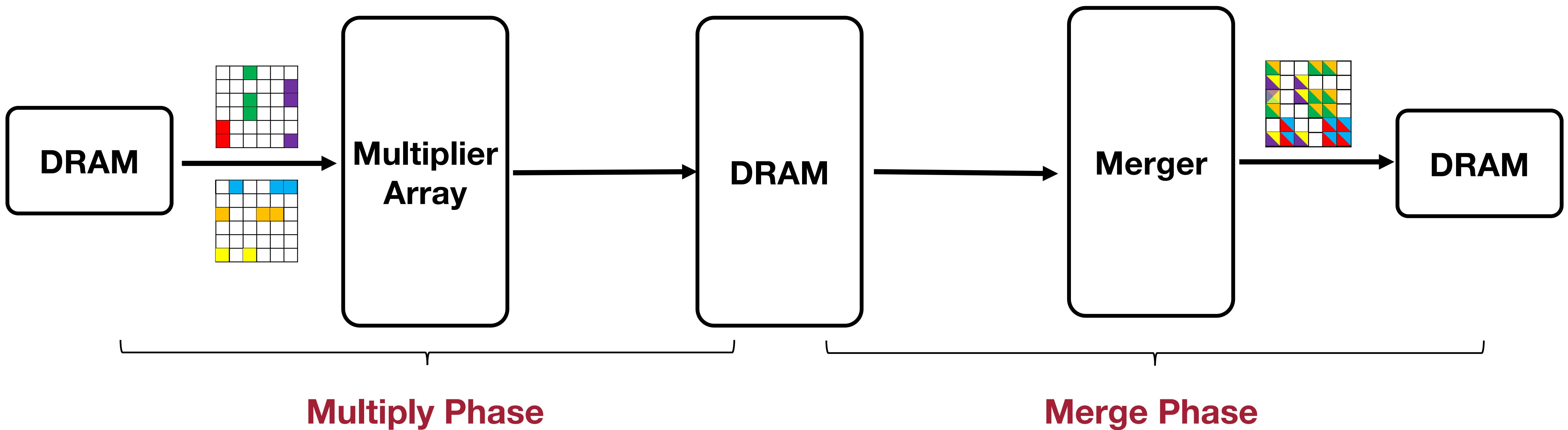
Background: Outer Product



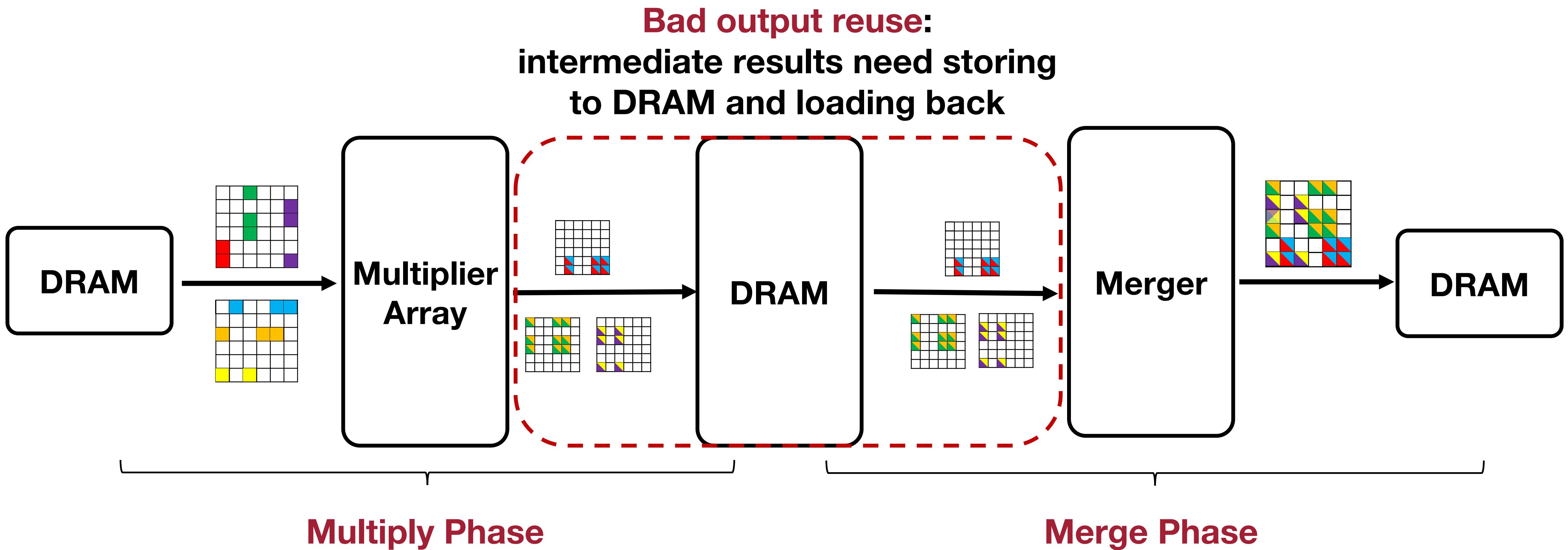
Background: Outer Product



Background: Outer Product

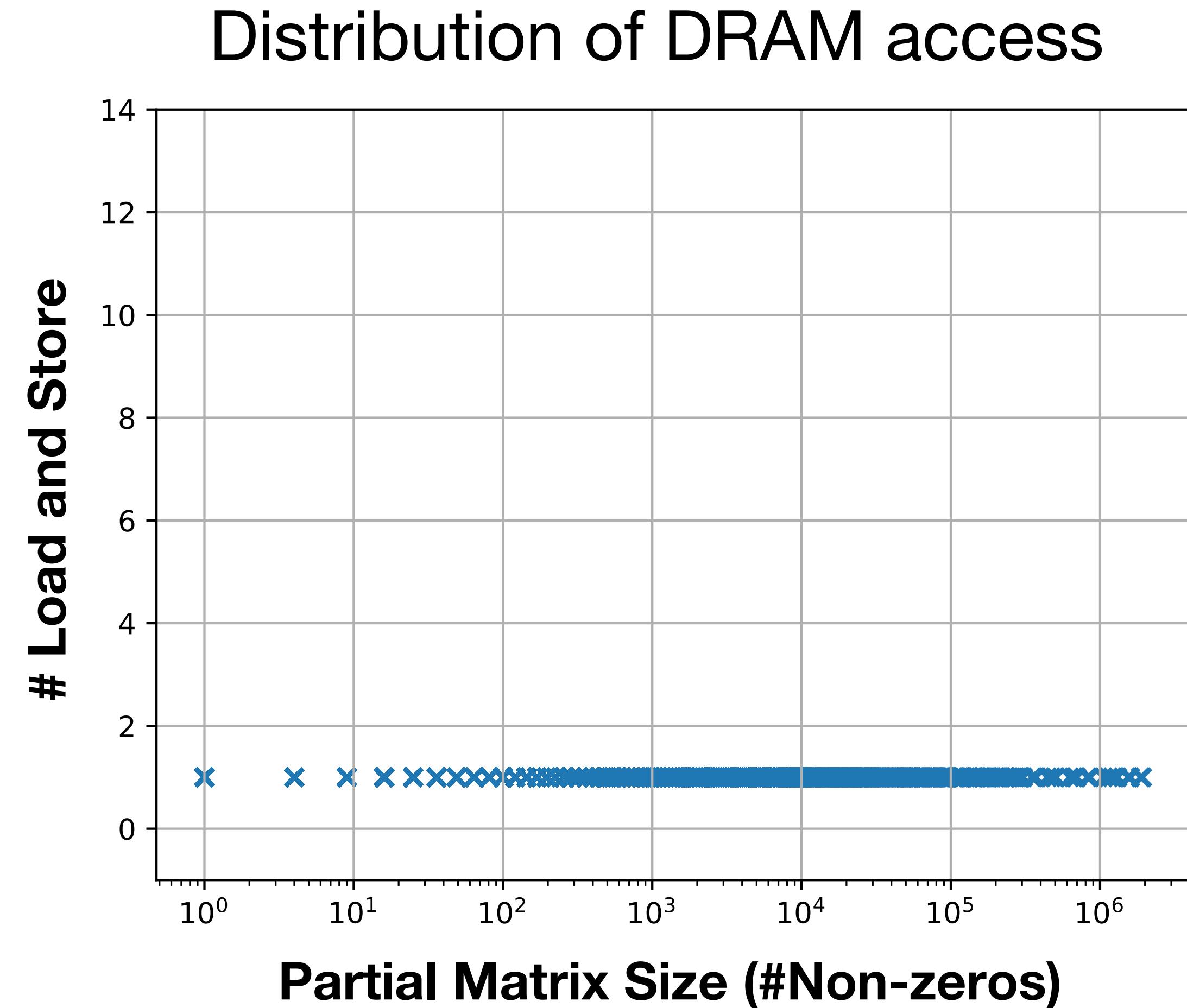


Background: Outer Product

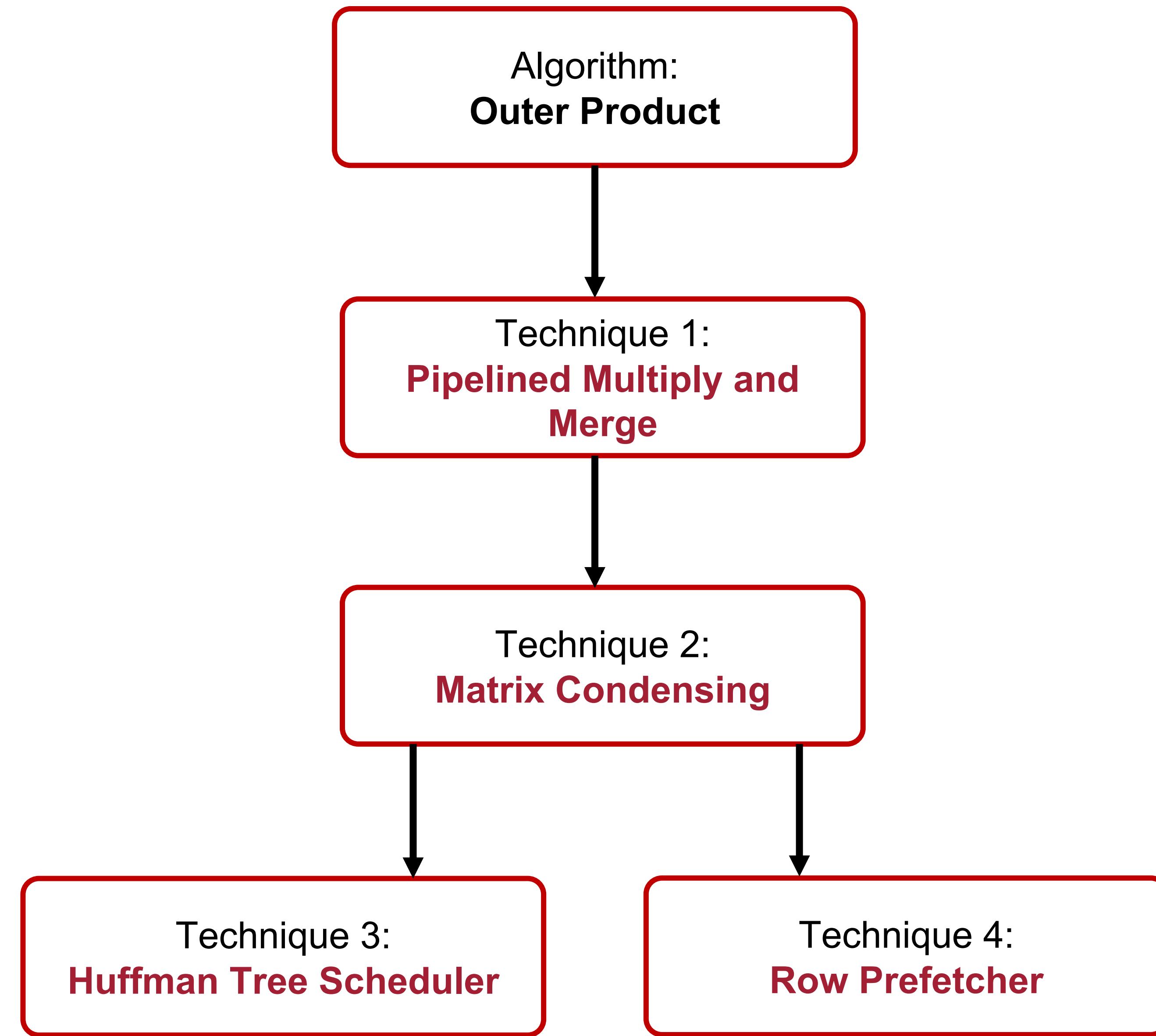


DRAM access of Intermediate Matrix in the baseline implementation

- Baseline Implementation:
OuterSPACE.
- Row-wise Output Stationary
- Each Intermediate Matrix has one-round of Store and Load.



Key idea: reduce both input and partial matrix DRAM access



Multiply and Merge

$$\begin{array}{|c|c|c|c|} \hline & 0.1 & & 0.5 \\ \hline 0.2 & & & 0.3 \\ \hline & & & \\ \hline & 1.2 & & \\ \hline \end{array}
 +
 \begin{array}{|c|c|c|c|} \hline & & & 0.6 \\ \hline & 1.3 & & 1.2 \\ \hline & -0.8 & 2.2 & \\ \hline 1.1 & & & \\ \hline \end{array}
 =
 \begin{array}{|c|c|c|c|} \hline & 0.1 & & 1.1 \\ \hline 0.2 & 1.3 & & 1.5 \\ \hline & -0.8 & 2.2 & \\ \hline 1.1 & 1.2 & & \\ \hline \end{array}$$

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
MatA	0	0.1	0	0.5	0.2	0	0	0.3	0	0	0	0	0	1.2	0	0
MatB	0	0	0	0.6	0	1.3	0	1.2	0	-0.8	2.2	0	1.1	0	0	0
Element-wise Add	0	0.1	0	1.1	0.2	1.3	0	1.5	0	-0.8	2.2	0	1.1	1.2	0	0

Merge Phase

MatA: (1,0.1) (3, 0.5) (4, 0.2) (7, 0.3) (13, 1.2)

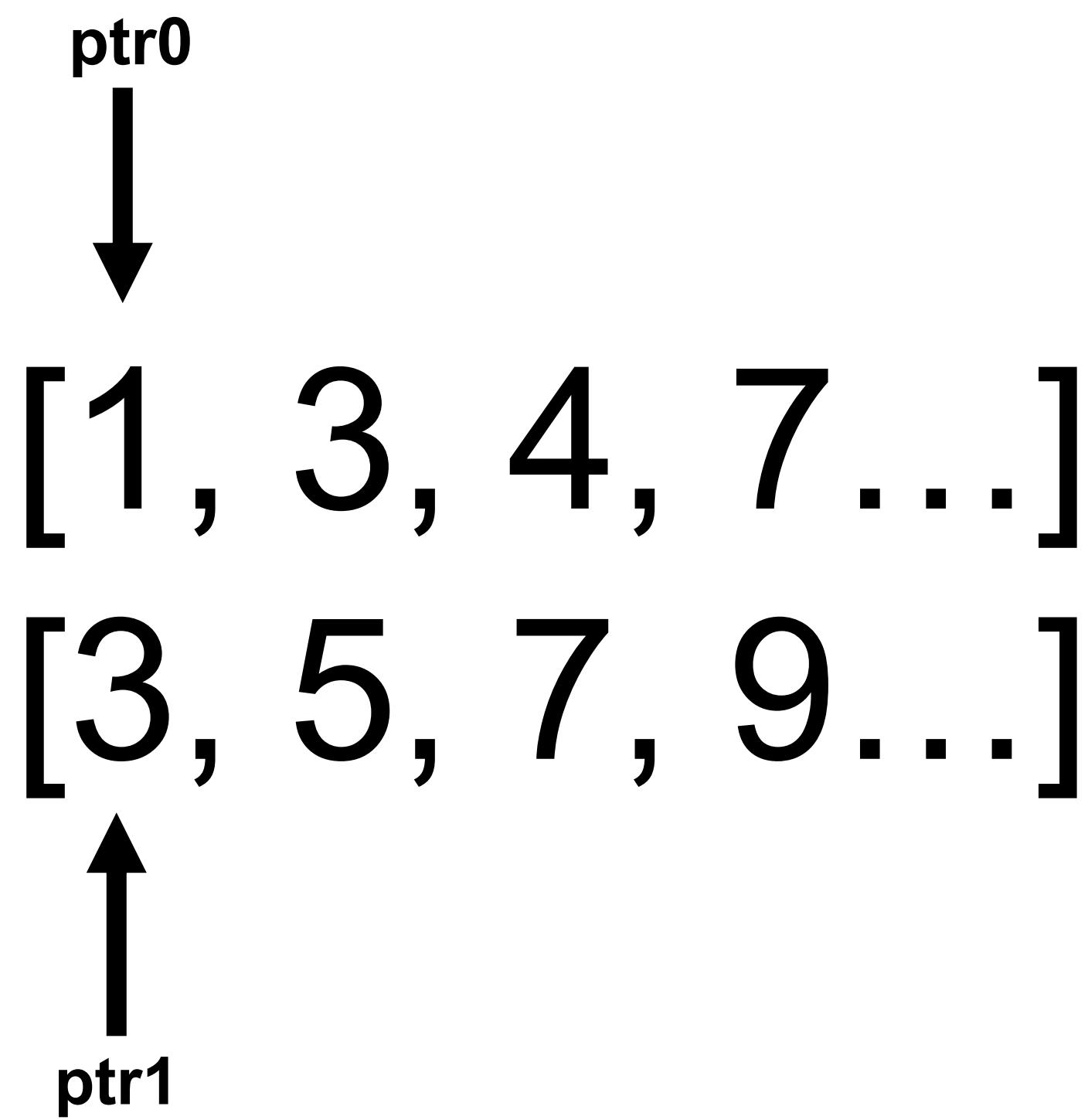
MatB: (3, 0.6) (5,1.3) (7, 1.2) (9, -0.8) (10, 2.2) (12, 1.1)

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
MatA	0	0.1	0	0.5	0.2	0	0	0.3	0	0	0	0	0	1.2	0	0
MatB	0	0	0	0.6	0	1.3	0	1.2	0	-0.8	2.2	0	1.1	0	0	0
Element-wise Add	0	0.1	0	1.1	0.2	1.3	0	1.5	0	-0.8	2.2	0	1.1	1.2	0	0

Merge Phase

MatA: (1,0.1) (3, 0.5) (4, 0.2) (7, 0.3) (13, 1.2)

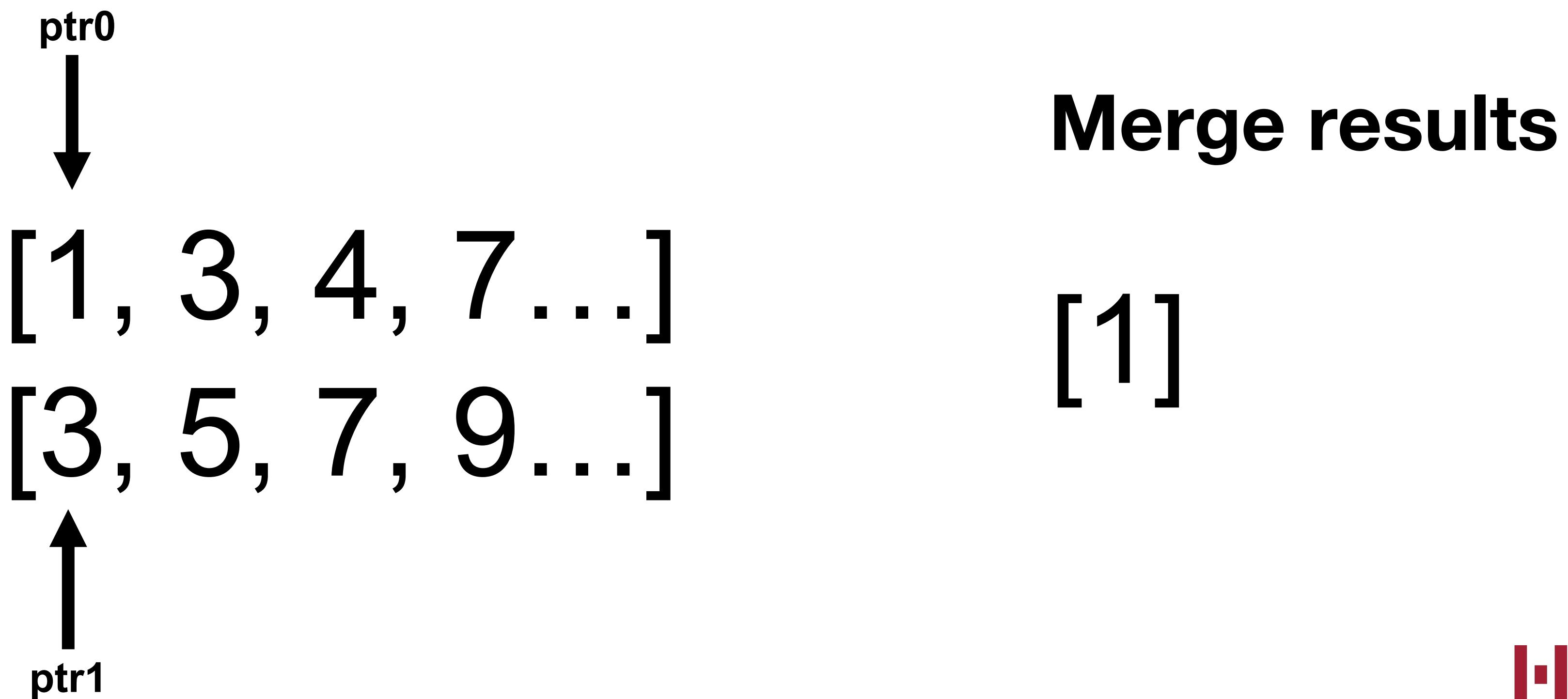
MatB: (3, 0.6) (5, 1.3) (7, 1.2) (9, -0.8) (10, 2.2) (12, 1.1)



Merge Phase

MatA: (1,0.1) (3, 0.5) (4, 0.2) (7, 0.3) (13, 1.2)

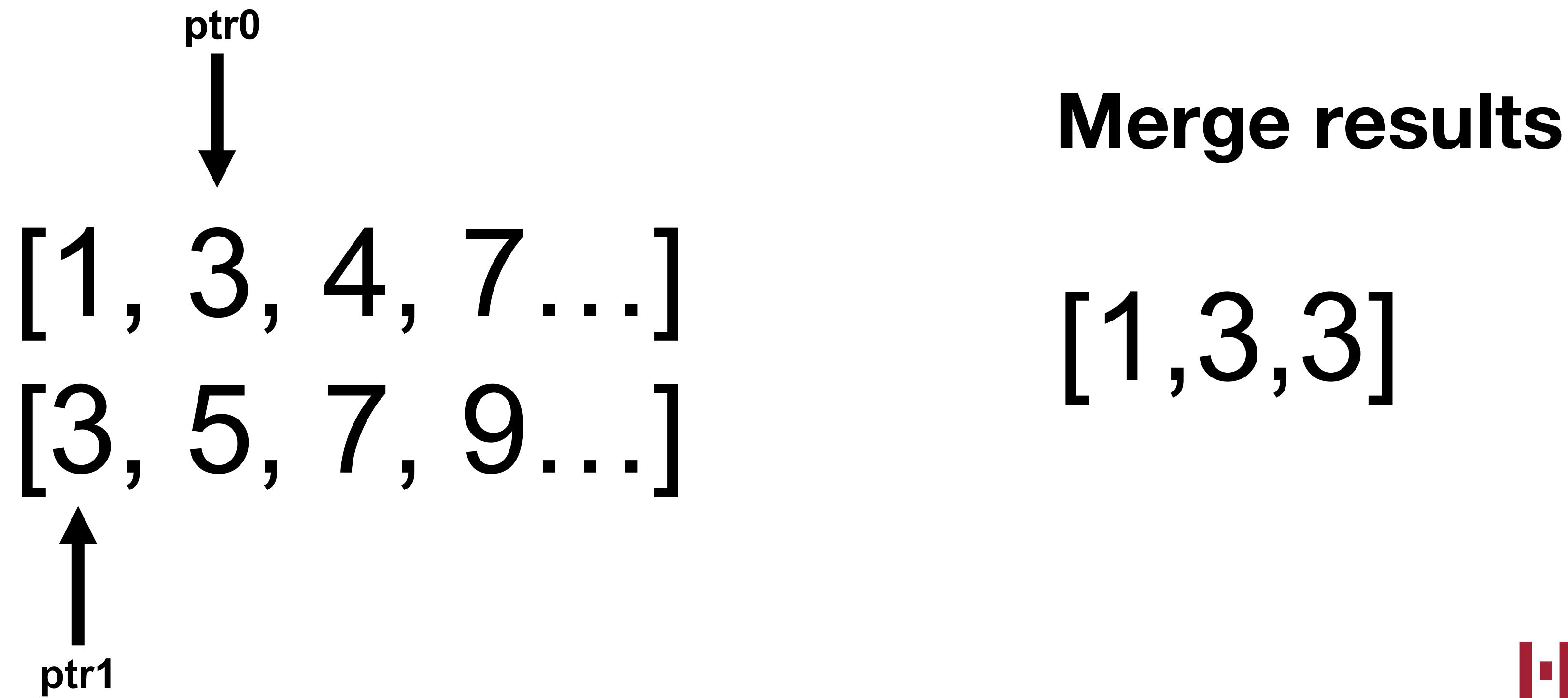
MatB: (3, 0.6) (5, 1.3) (7, 1.2) (9, -0.8) (10, 2.2) (12, 1.1)



Merge Phase

MatA: (1,0.1) (3, 0.5) (4, 0.2) (7, 0.3) (13, 1.2)

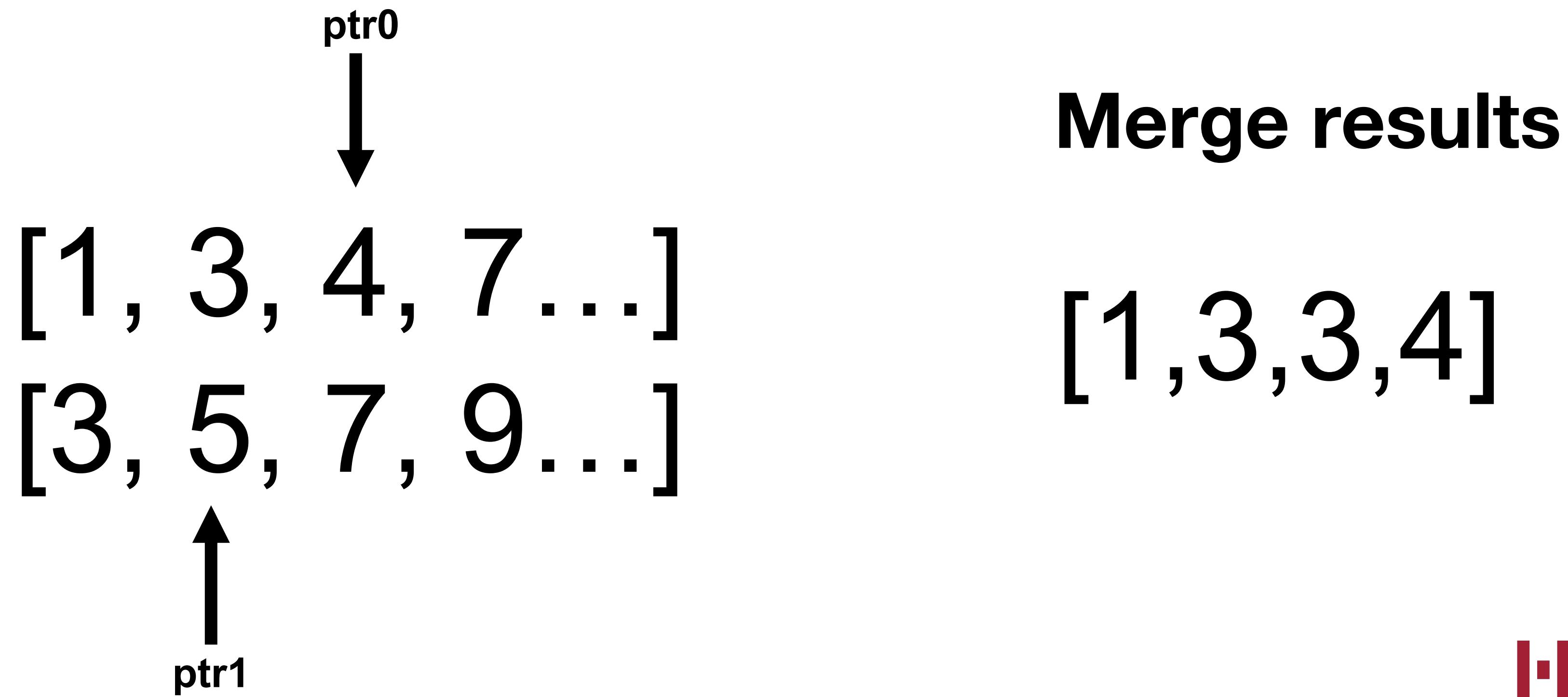
MatB: (3, 0.6) (5,1.3) (7, 1.2) (9, -0.8) (10, 2.2) (12, 1.1)



Merge Phase

MatA: (1,0.1) (3, 0.5) (4, 0.2) (7, 0.3) (13, 1.2)

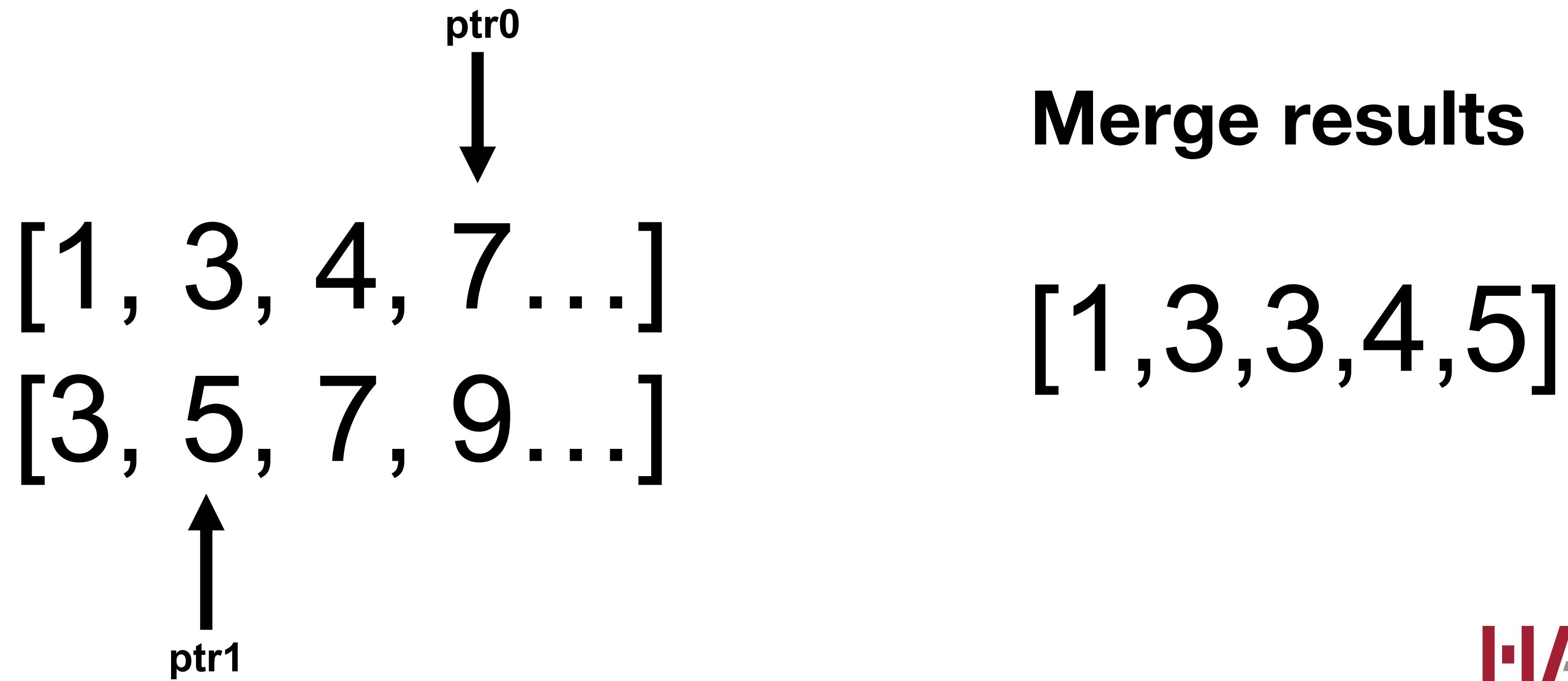
MatB: (3, 0.6) (5,1.3) (7, 1.2) (9, -0.8) (10, 2.2) (12, 1.1)



Merge Phase

MatA: (1,0.1) (3, 0.5) (4, 0.2) (7, 0.3) (13, 1.2)

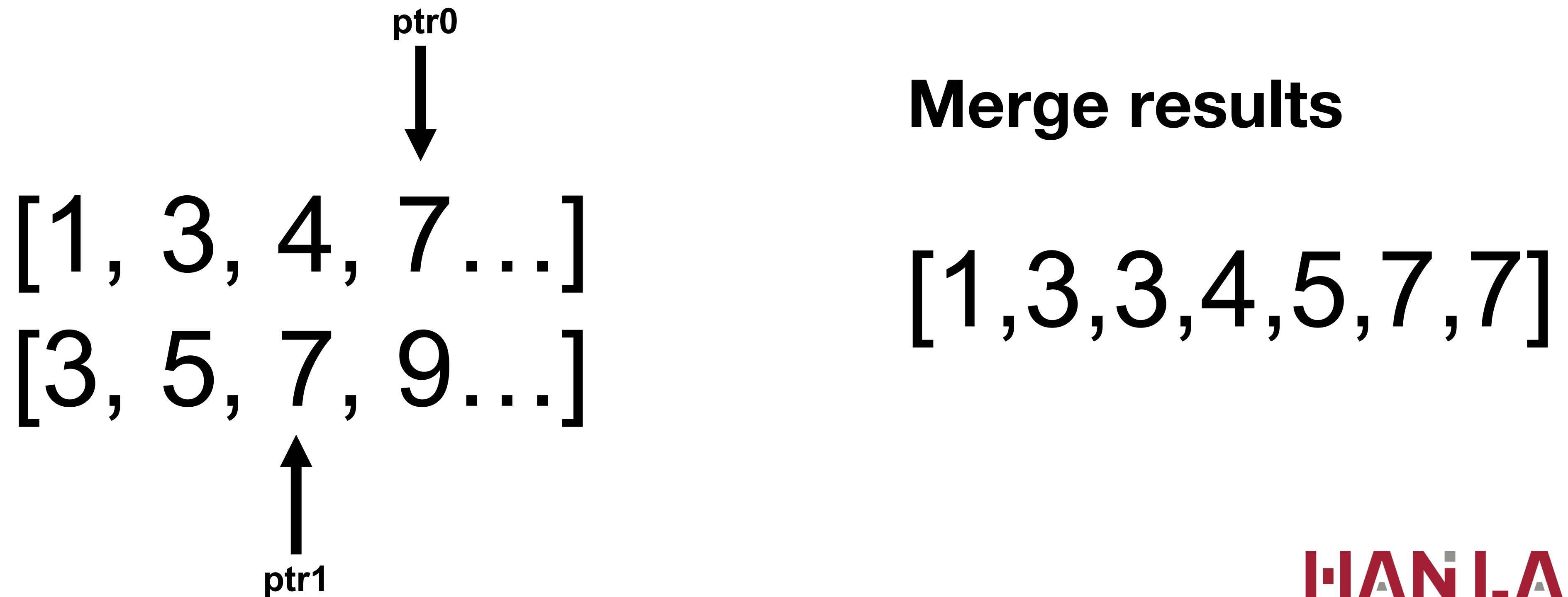
MatB: (3, 0.6) (5,1.3) (7, 1.2) (9, -0.8) (10, 2.2) (12, 1.1)



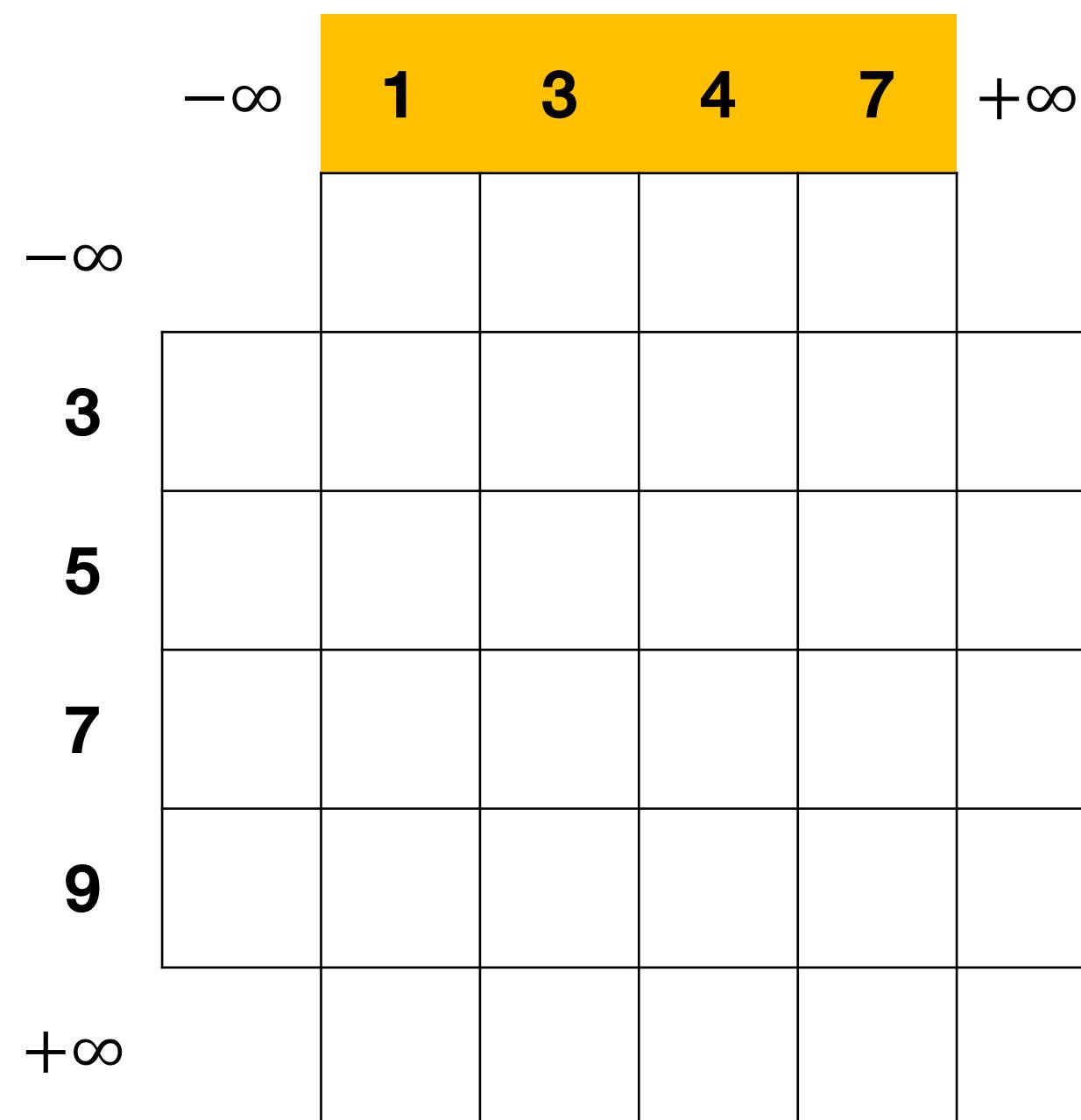
Merge Phase

MatA: (1,0.1) (3, 0.5) (4, 0.2) (7, 0.3) (13, 1.2)

MatB: (3, 0.6) (5,1.3) (7, 1.2) (9, -0.8) (10, 2.2) (12, 1.1)



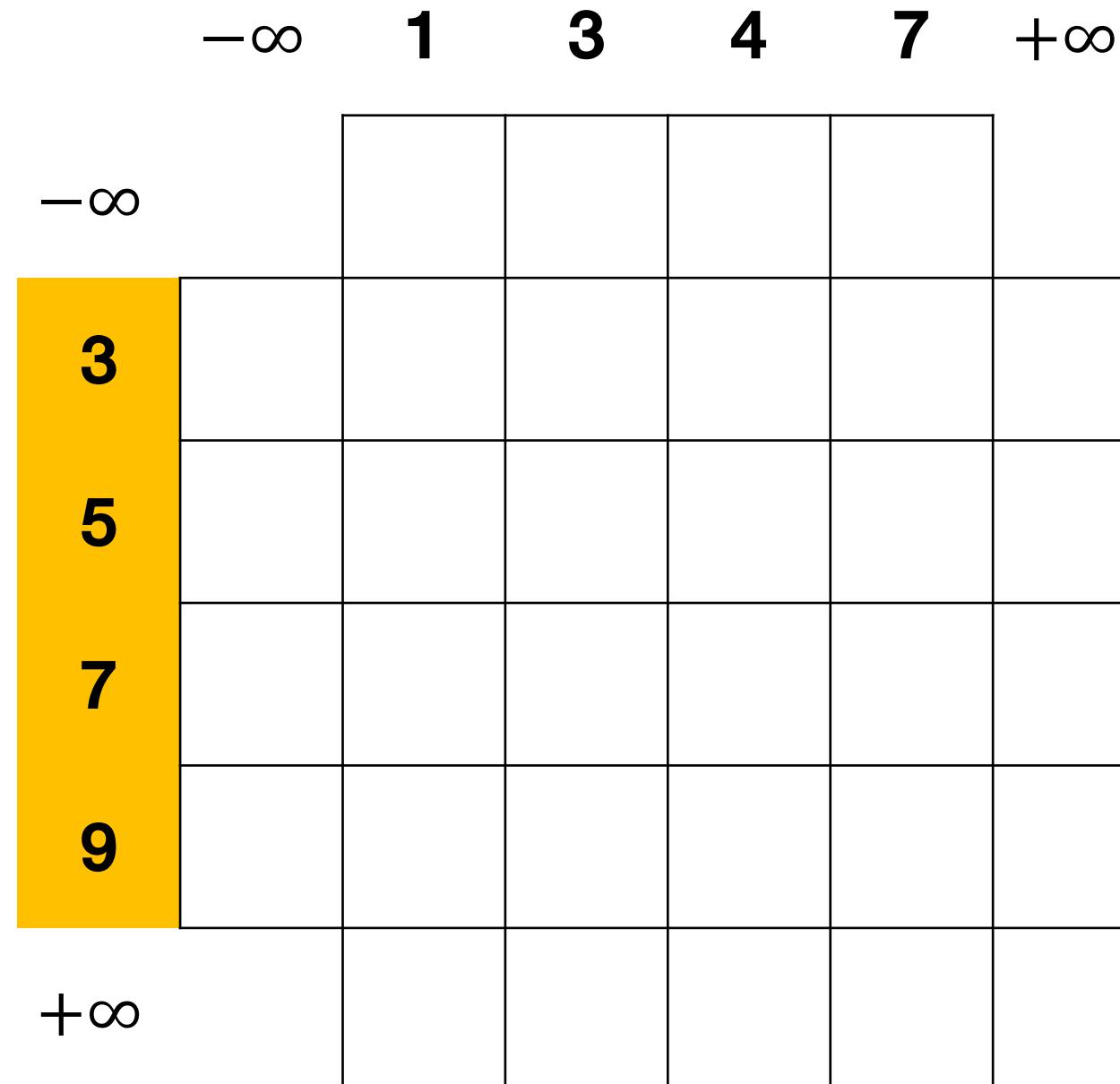
Technique 1: Pipelined Multiply and Merge



Index	0	1	2	3	4	5	6	7	8	9
MatA	0	0.1	0	0.5	0.2	0	0	0.3	0	0
MatB										
Element-wise Add										

Paralleled in space instead of serialized in time

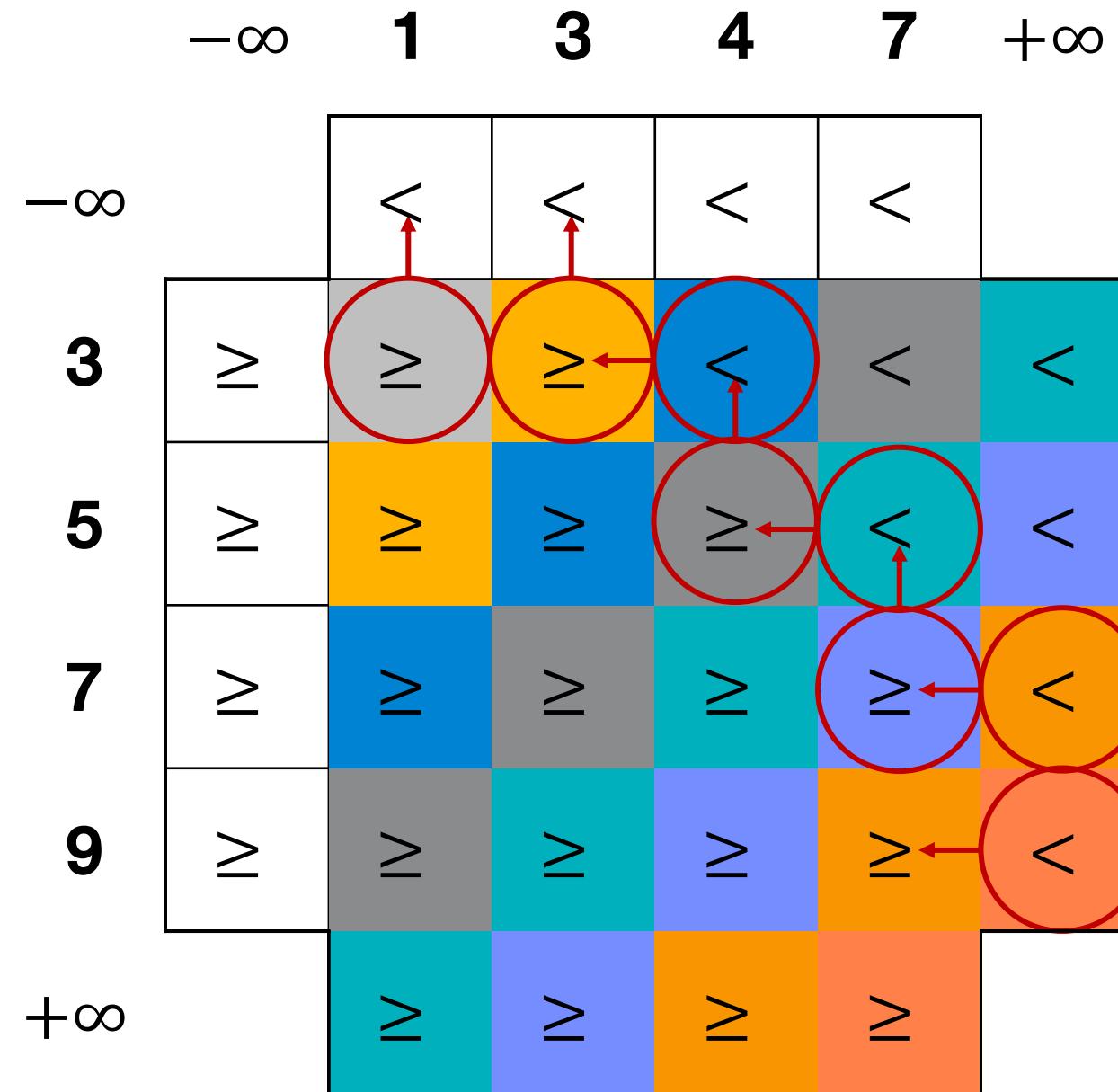
Technique 1: Pipelined Multiply and Merge



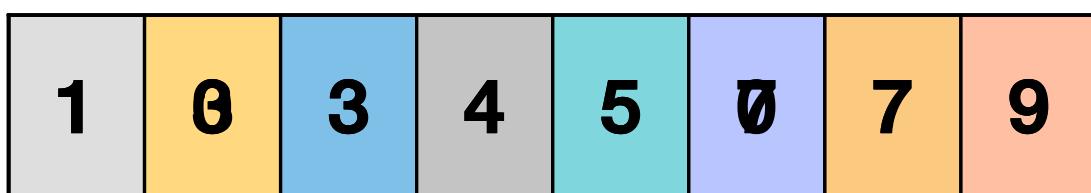
Index	0	1	2	3	4	5	6	7	8	9
MatA										
MatB	0	0	0	0.6	0	1.3	0	1.2	0	-0.8
Element-wise Add										

Paralleled in space instead of serialized in time

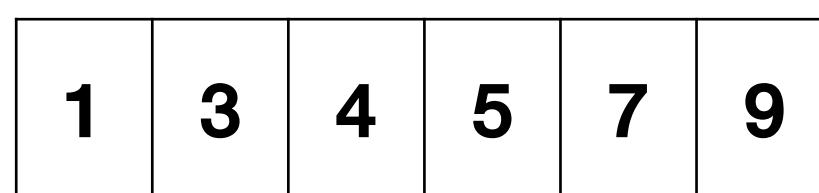
Technique 1: Pipelined Multiply and Merge



Index	0	1	2	3	4	5	6	7	8	9
MatA	0	0.1	0	0.5	0.2	0	0	0.3	0	0
MatB	0	0	0	0.6	0	1.3	0	1.2	0	-0.8
Element-wise Add										

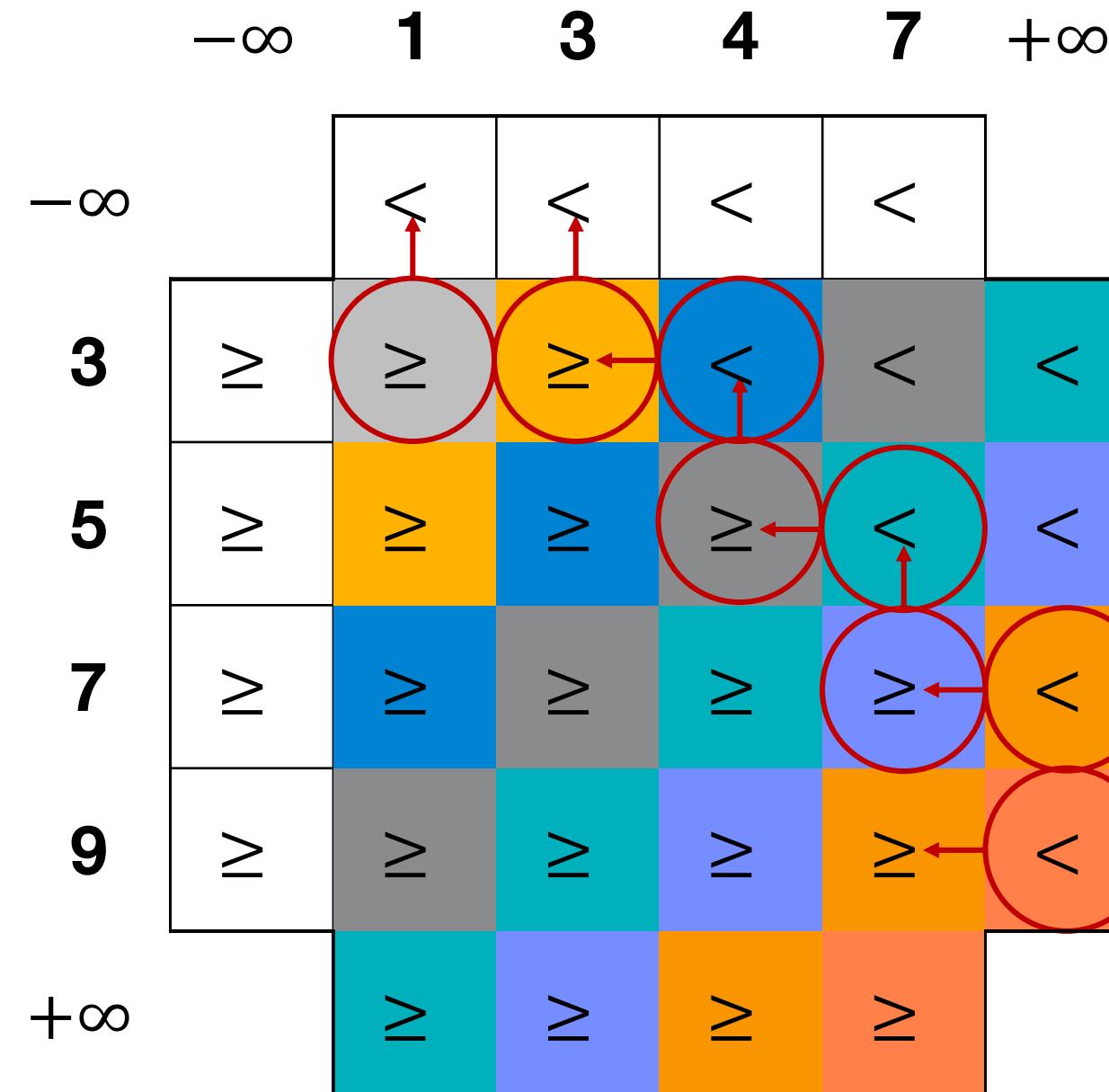


↓ Add values of same indices



0.1 1.1 0.2 1.3 1.5 -0.8

Technique 1: Pipelined Multiply and Merge



Index	0	1	2	3	4	5	6	7	8	9
MatA	0	0.1	0	0.5	0.2	0	0	0.3	0	0
MatB	0	0	0	0.6	0	1.3	0	1.2	0	-0.8
Element-wise Add	0	0.1	0	1.1	0.2	1.3	0	1.5	0	-0.8

1	0	3	4	5	0	7	9
---	---	---	---	---	---	---	---

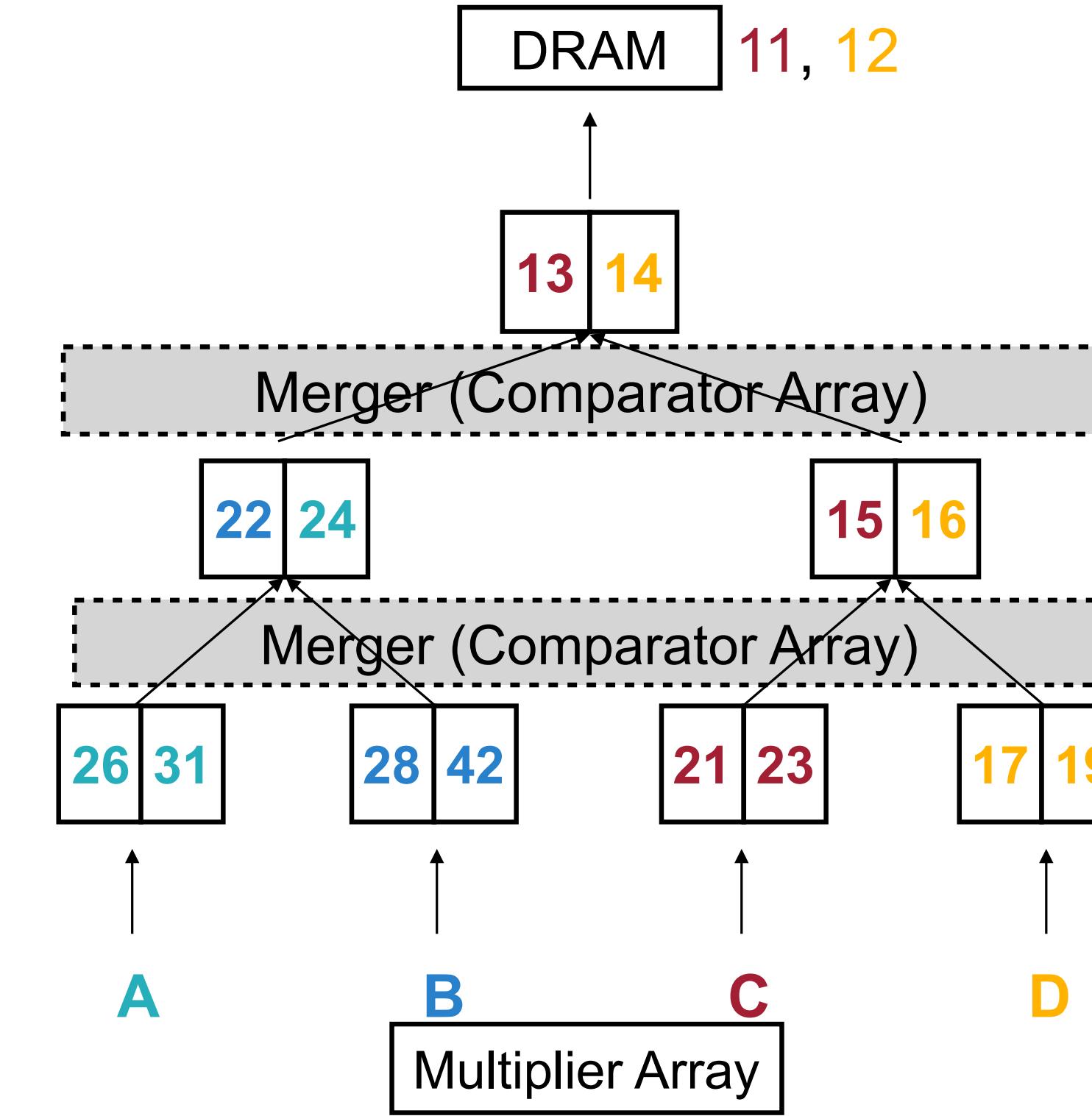
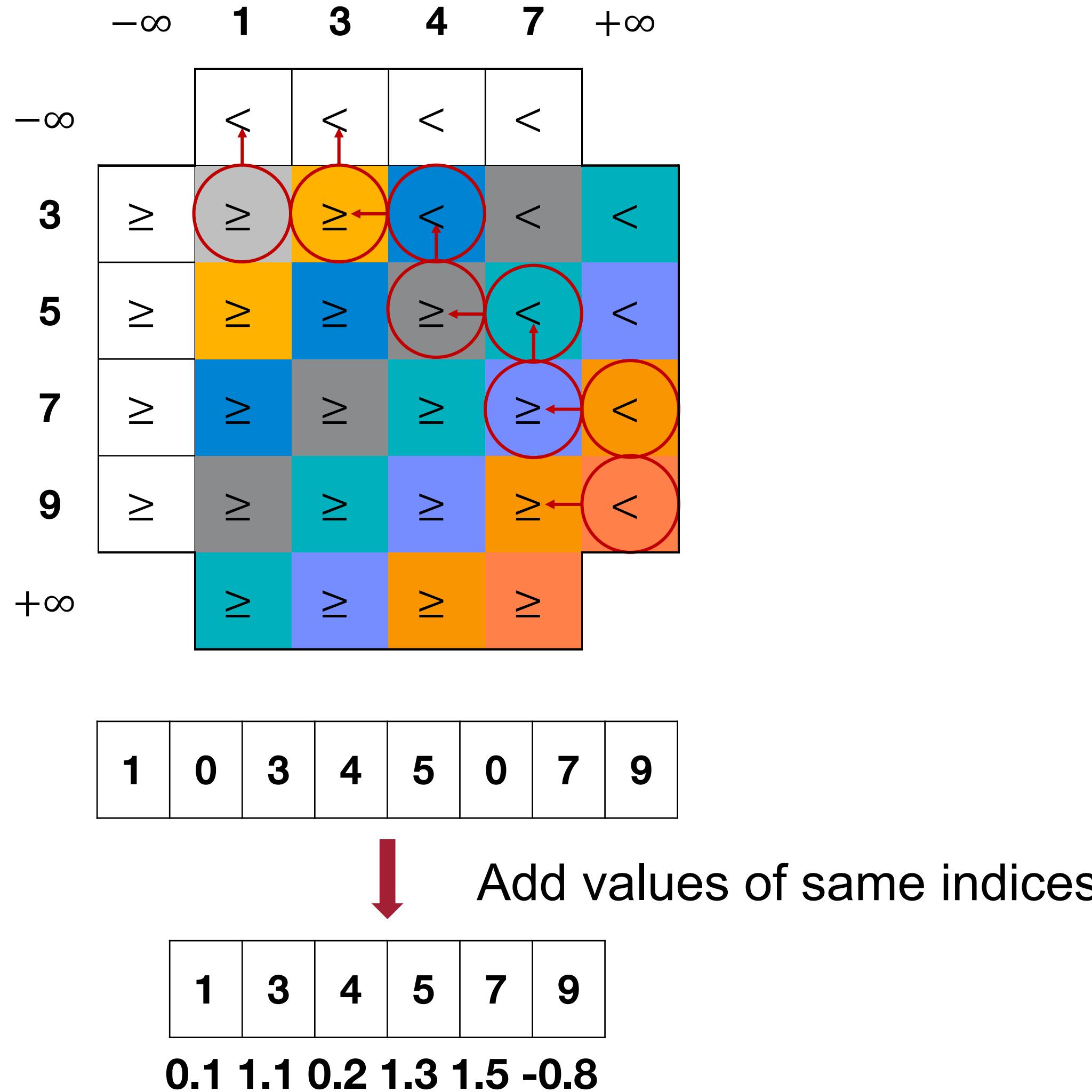
↓ Add values of same indices

1	3	4	5	7	9
---	---	---	---	---	---

0.1 1.1 0.2 1.3 1.5 -0.8

Get the results in one clock cycle

Technique 1: Pipelined Multiply and Merge

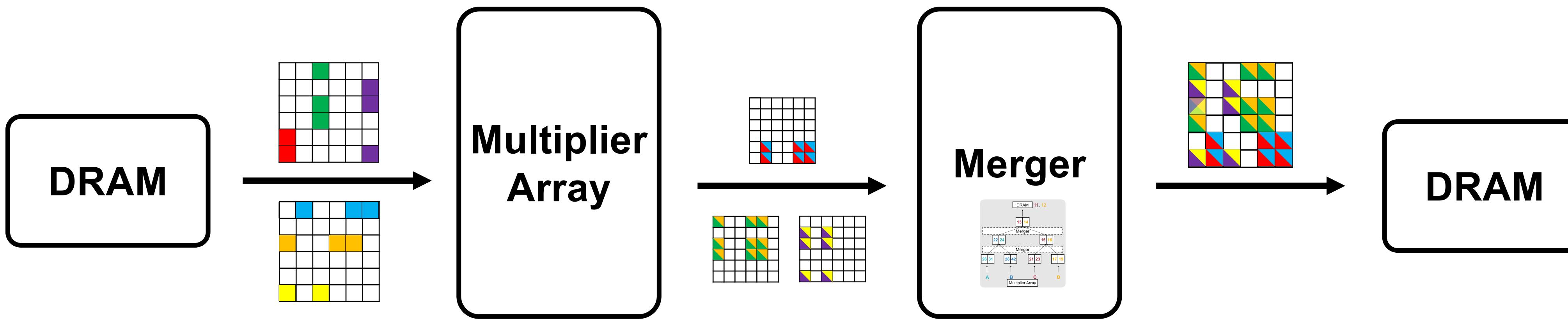


Merge Tree

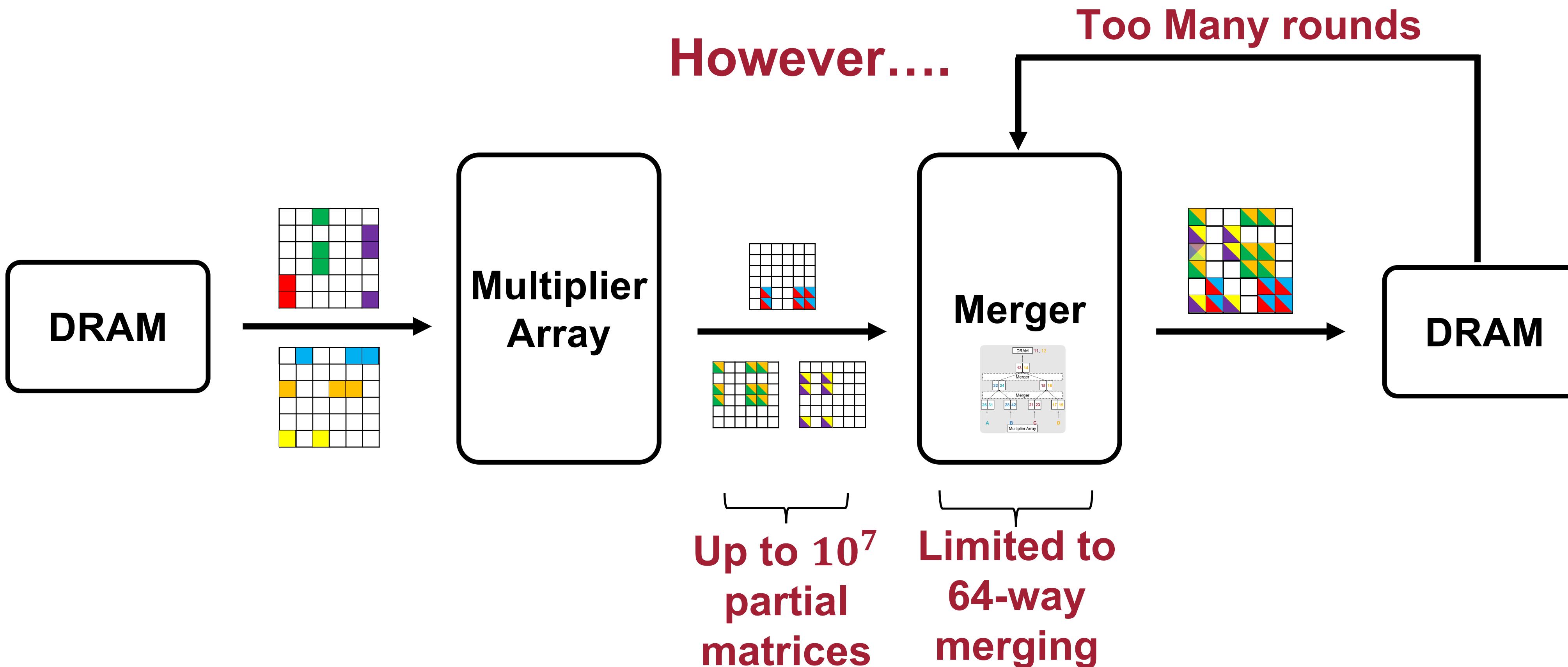
A: (24)(26)(31)(52)(54)(56)(57)(58)(73)(75)
 B: (22)(28)(42)(44)(46)(47)(48)
 C: (11)(13)(15)(21)(23)(25)(41)(43)(45)
 D: (12)(14)(16)(17)(18)(32)(34)(36)(37)(38)(72)

Technique 1: Pipelined Multiply and Merge

Ideally, partial matrices will not be stored on DRAM.

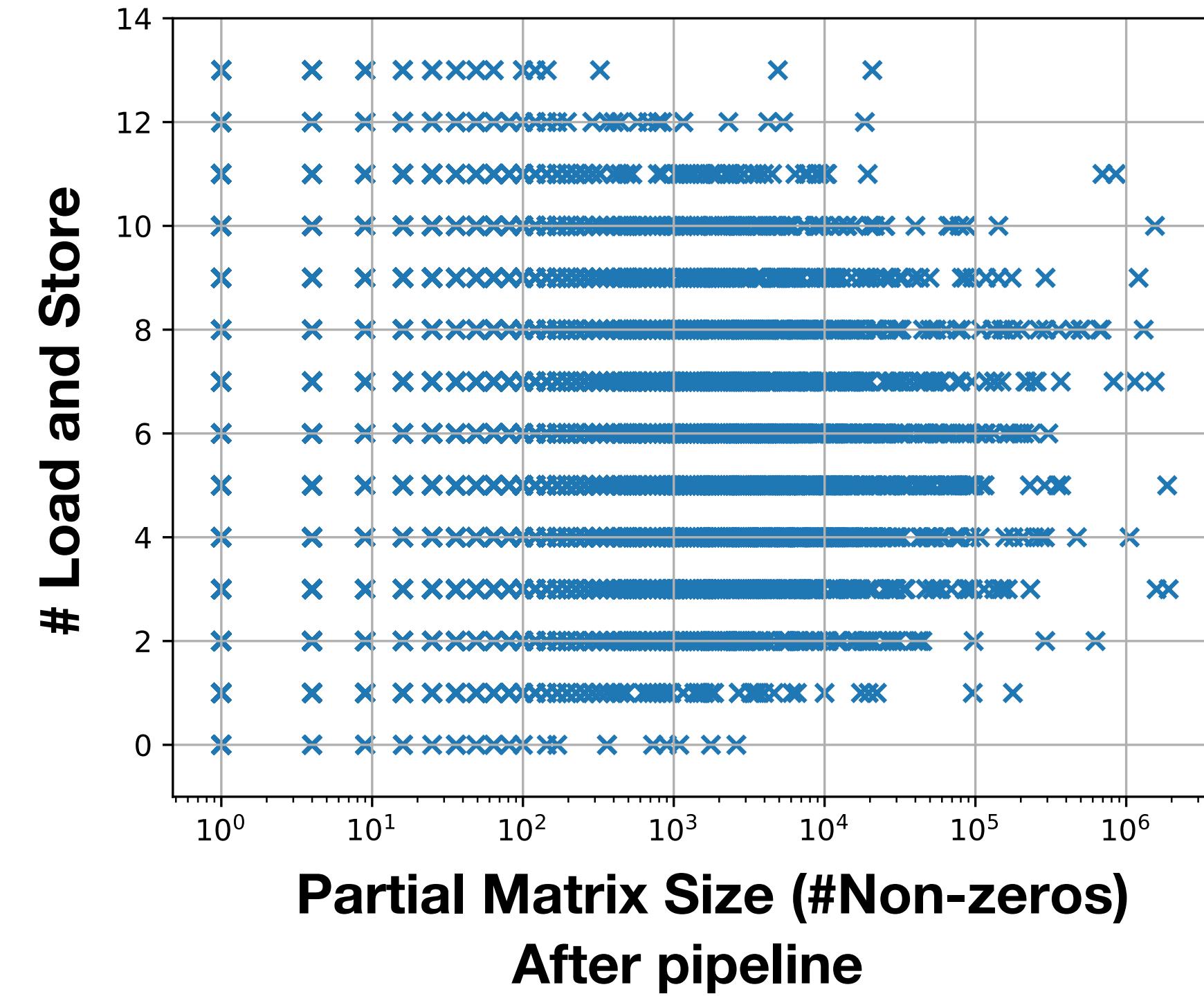
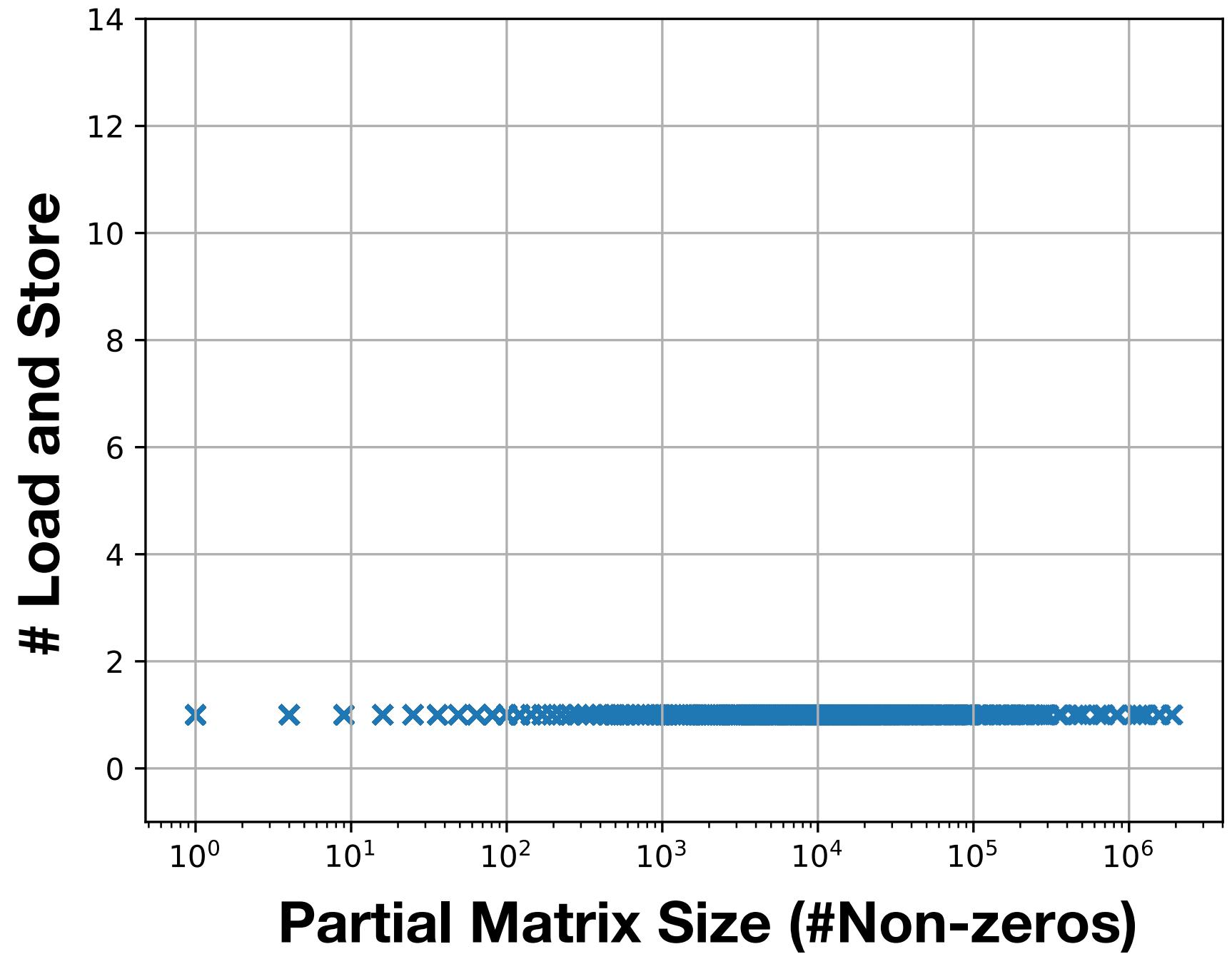


Technique 1: Pipelined Multiply and Merge



Technique 1: Pipelined Multiply and Merge

Distribution of DRAM access

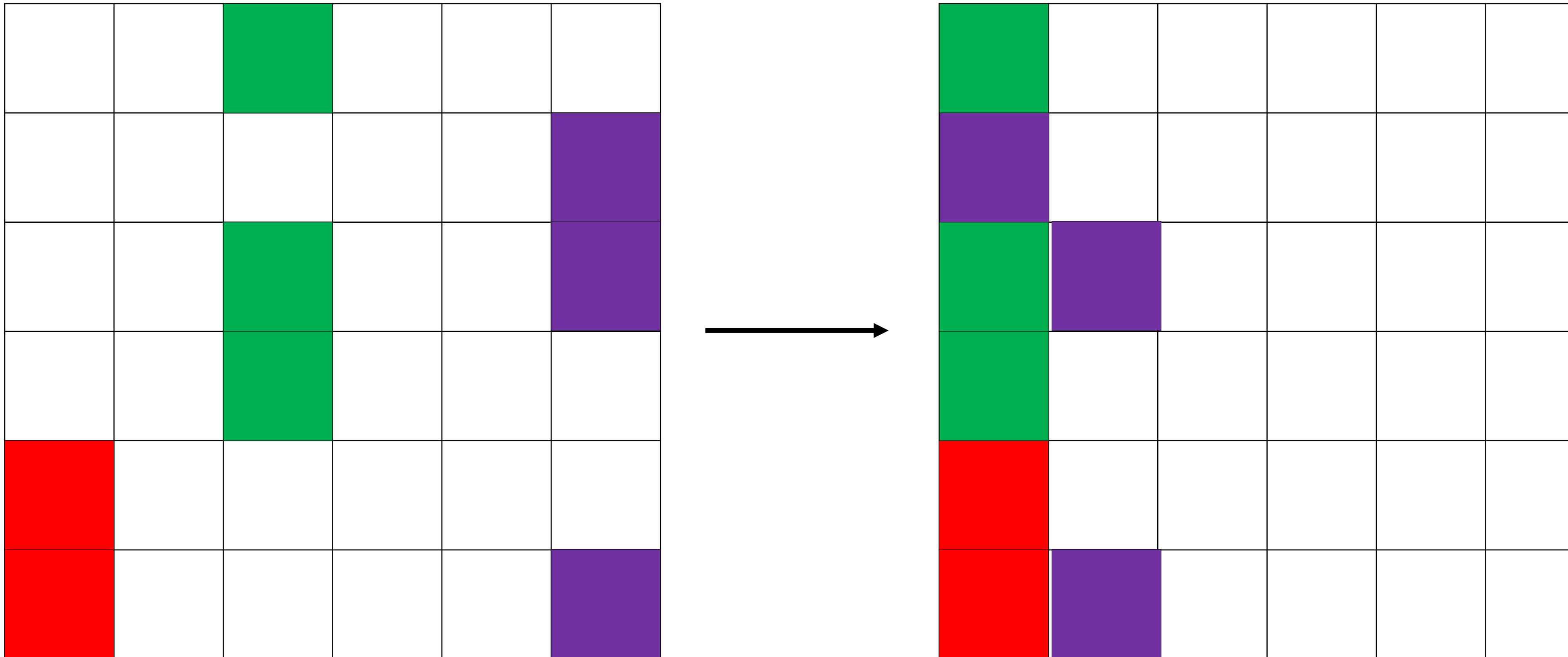


Breakdown of Memory Access



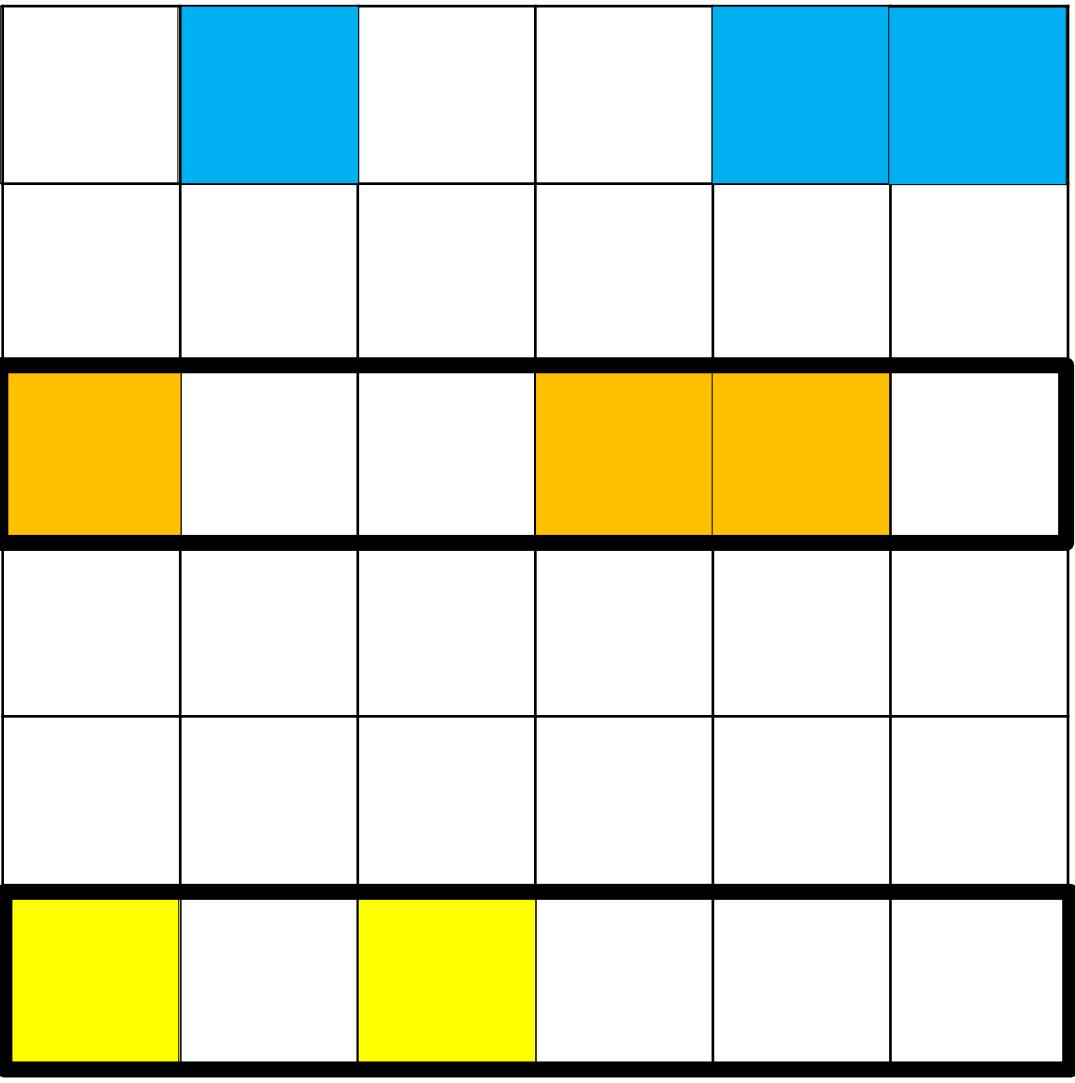
■ Read Left Matrix ■ Read Right Matrix ■ R/W Intermediate Results ■ Write Final Result

Technique 2: Matrix Condensing

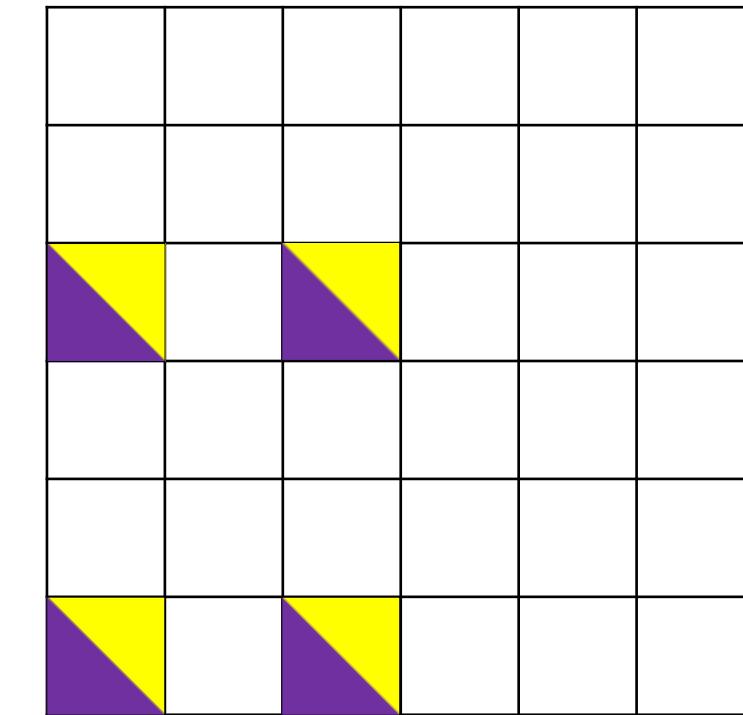
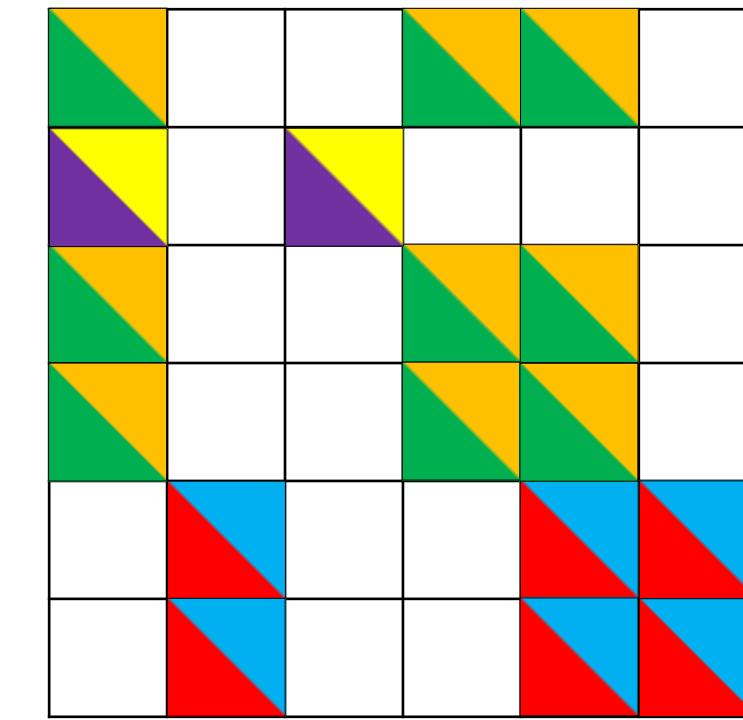
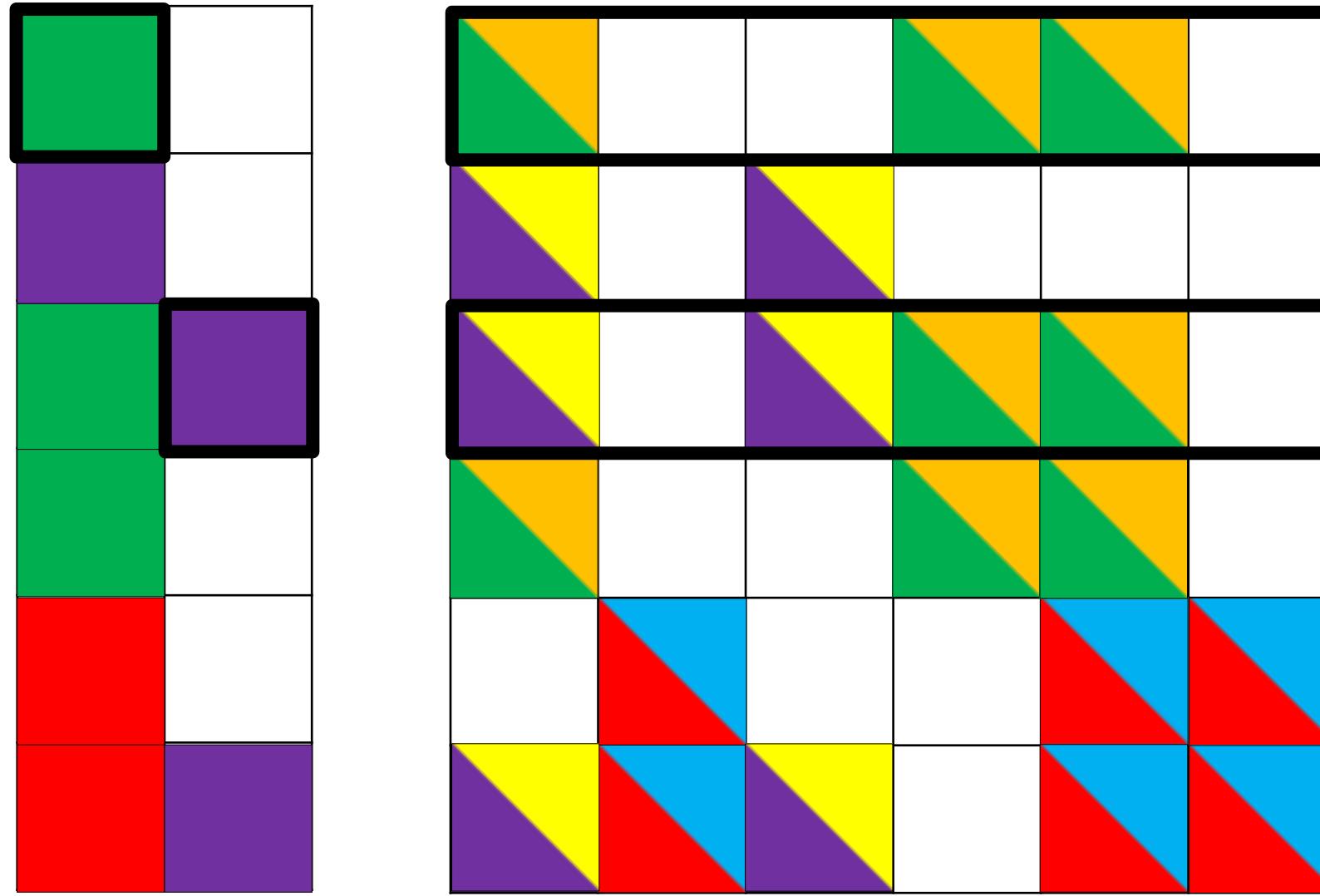


Technique 2: Matrix Condensing

Right Matrix B (CSR)

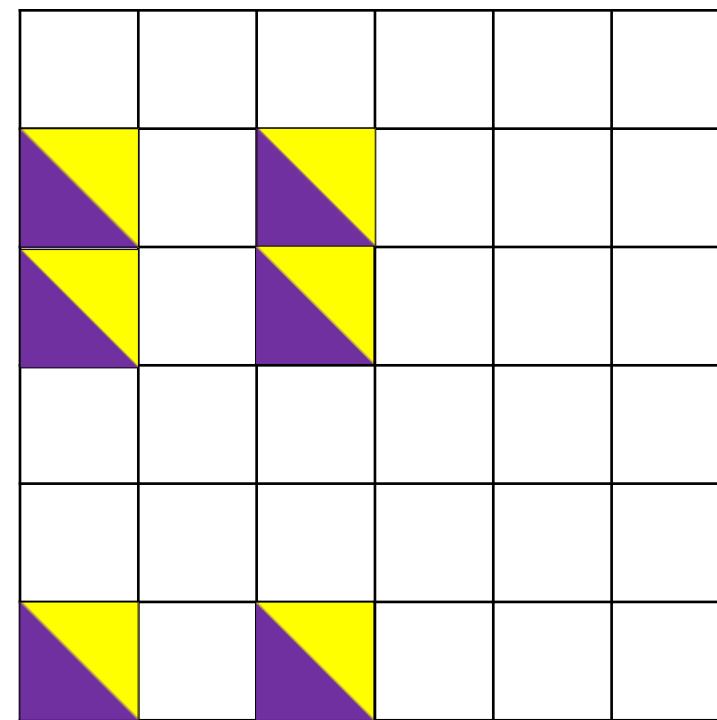
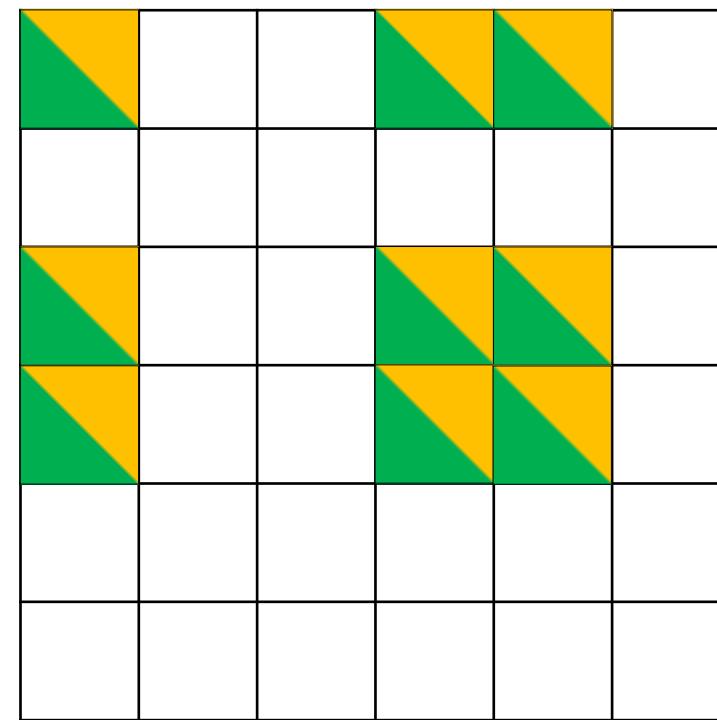
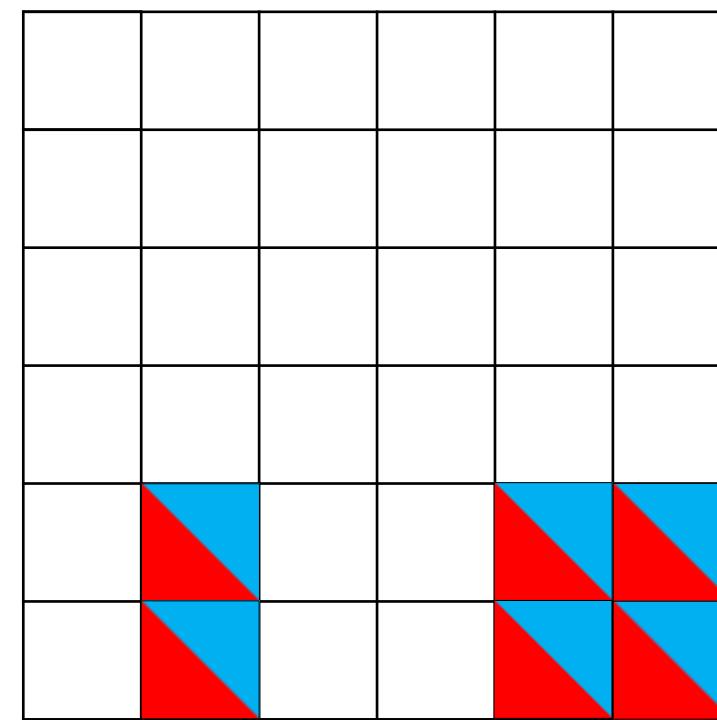


Condensed Matrix A'
(CSR)

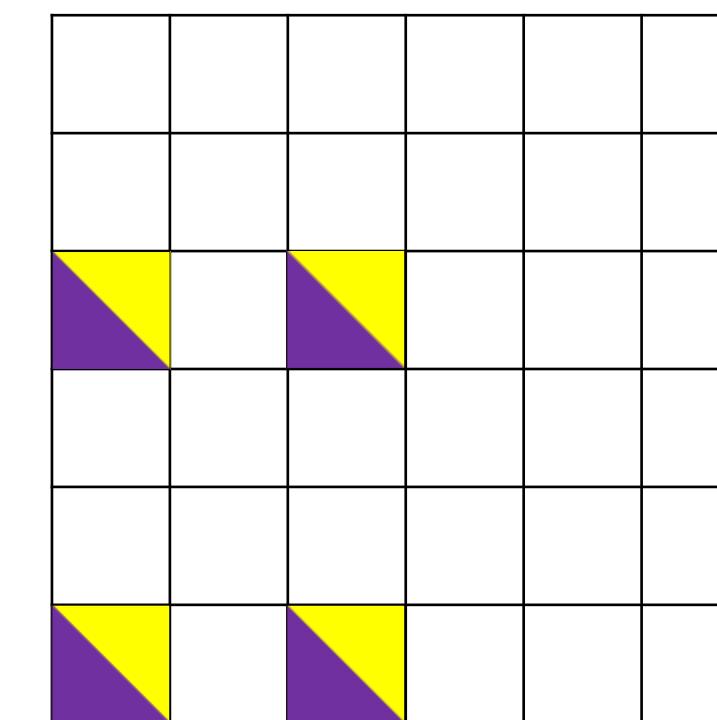
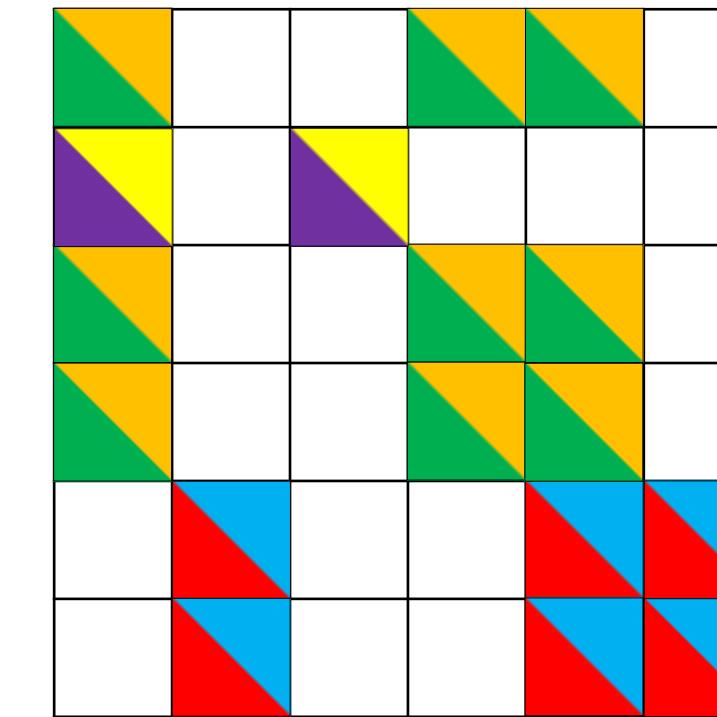


Technique 2: Matrix Condensing

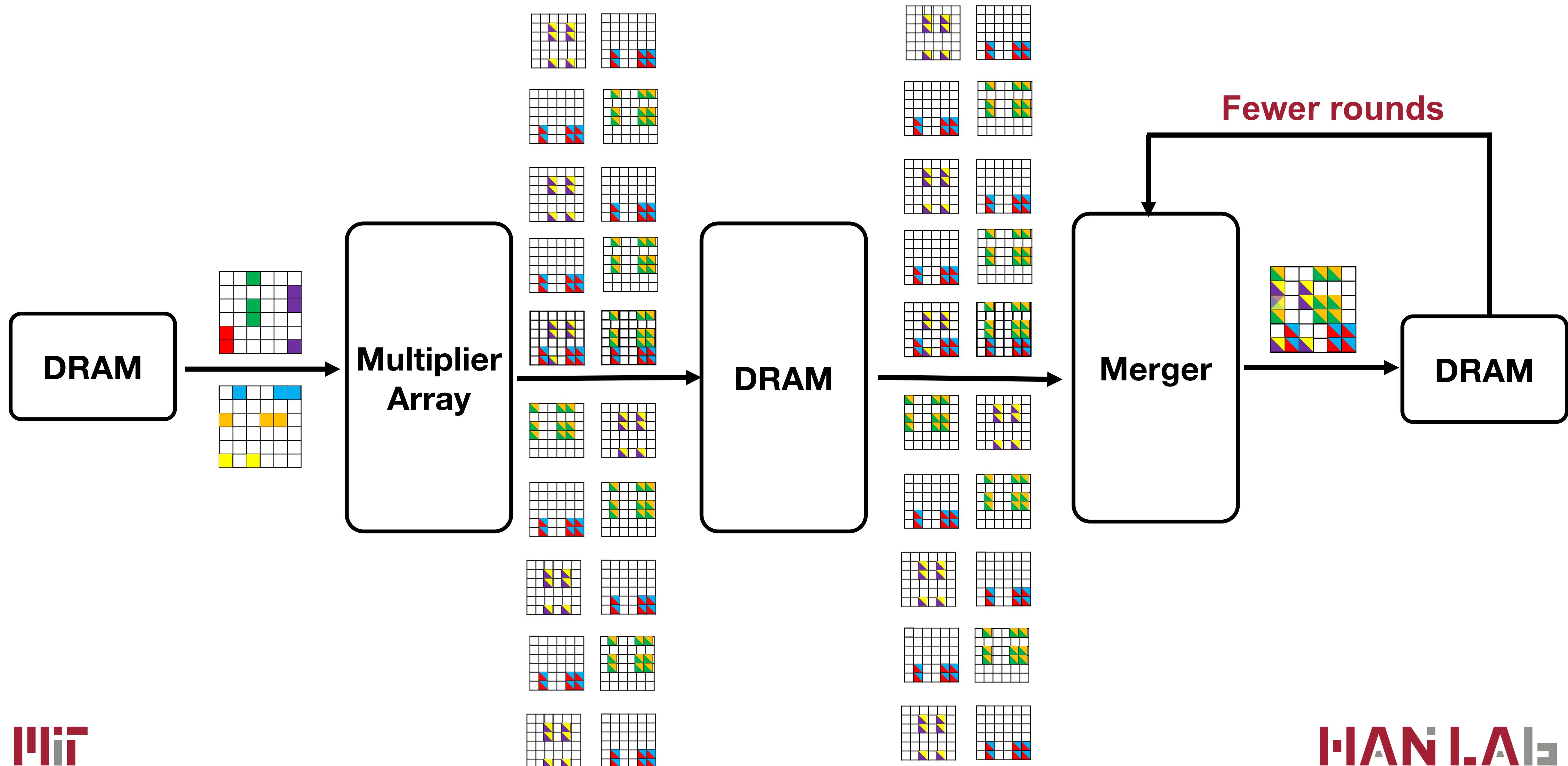
Before Condensing:
Up to 10^7 partial matrices



After Condensing:
Only $10 \sim 10^3$ partial matrices

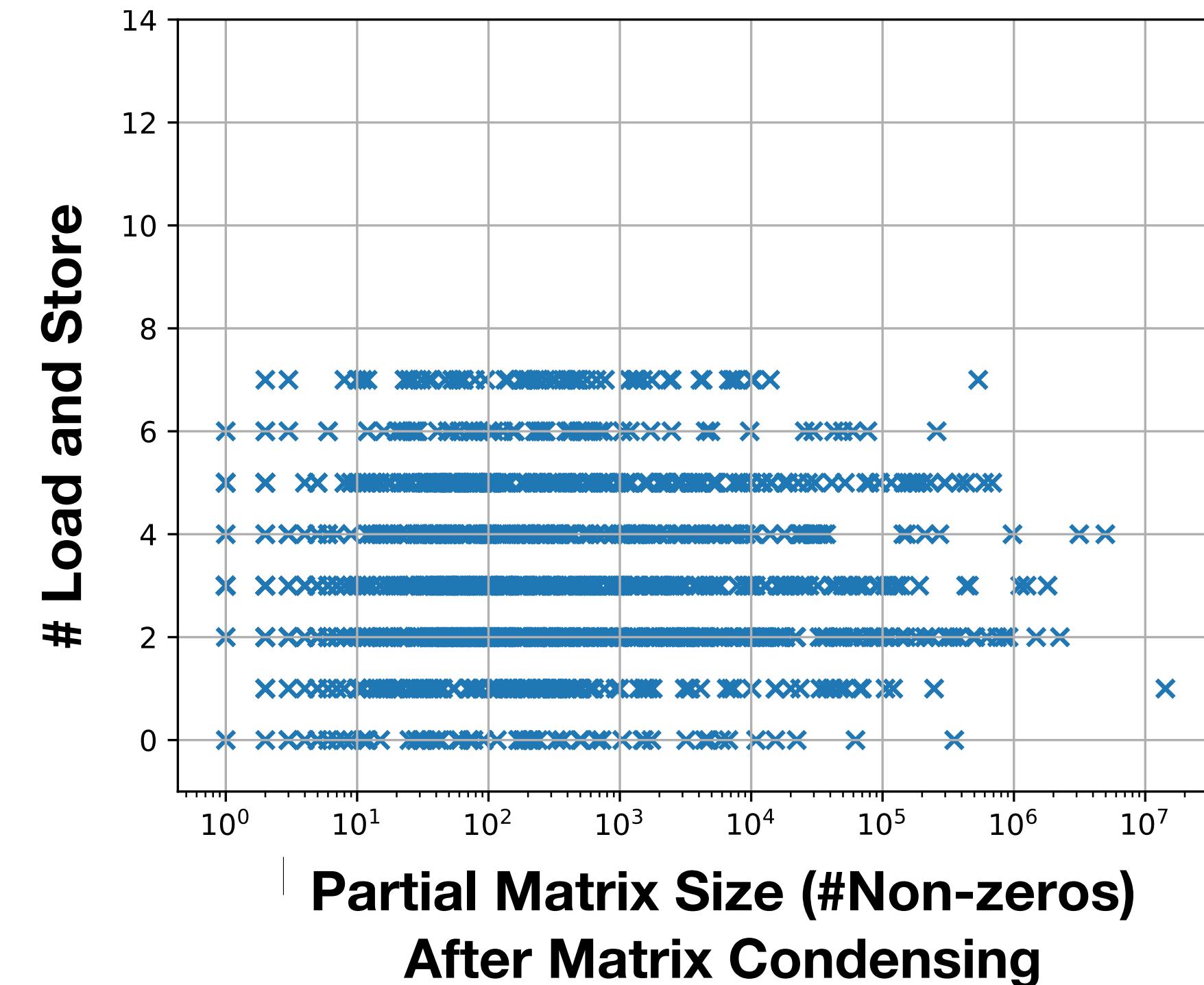
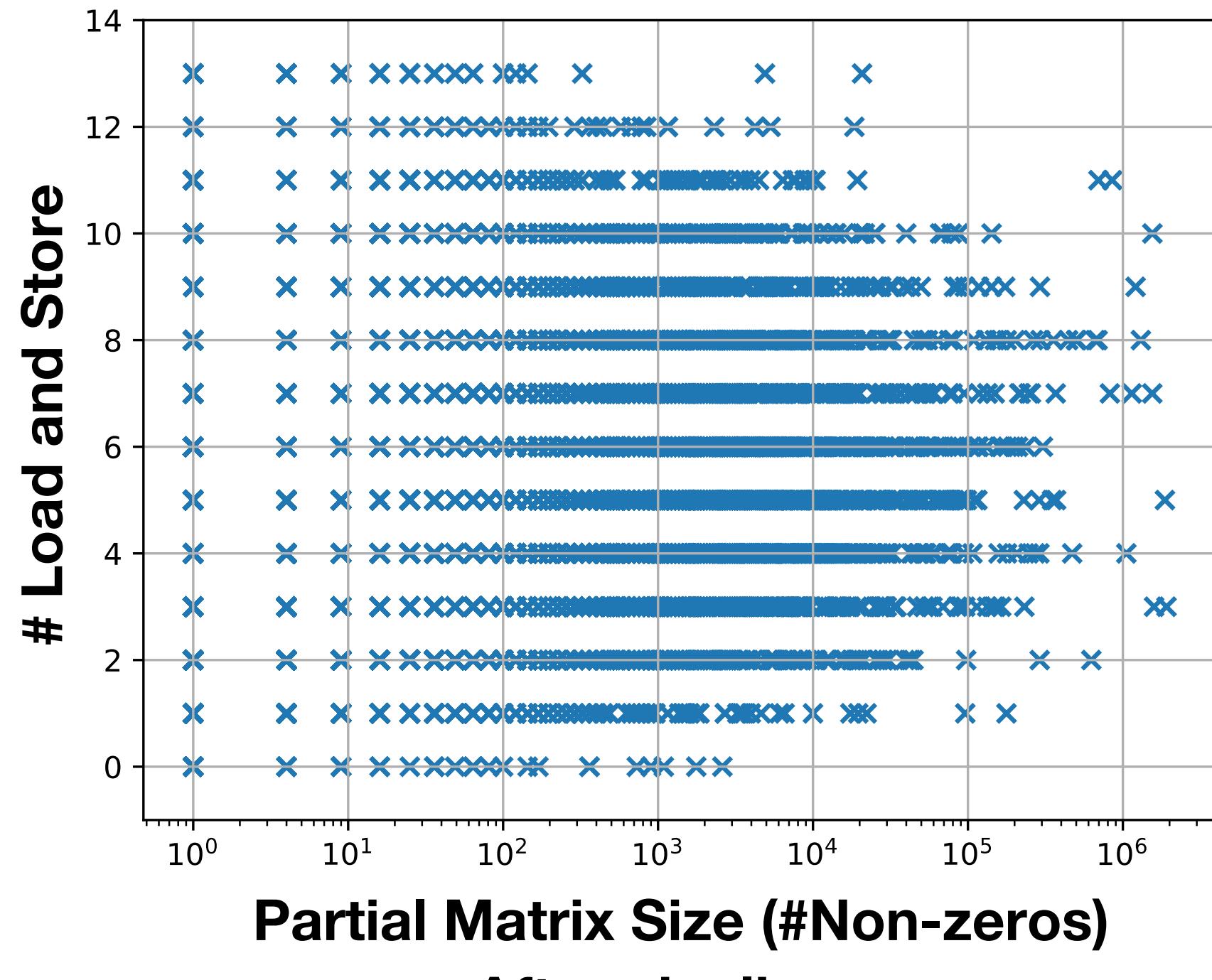


Technique 2: Matrix Condensing

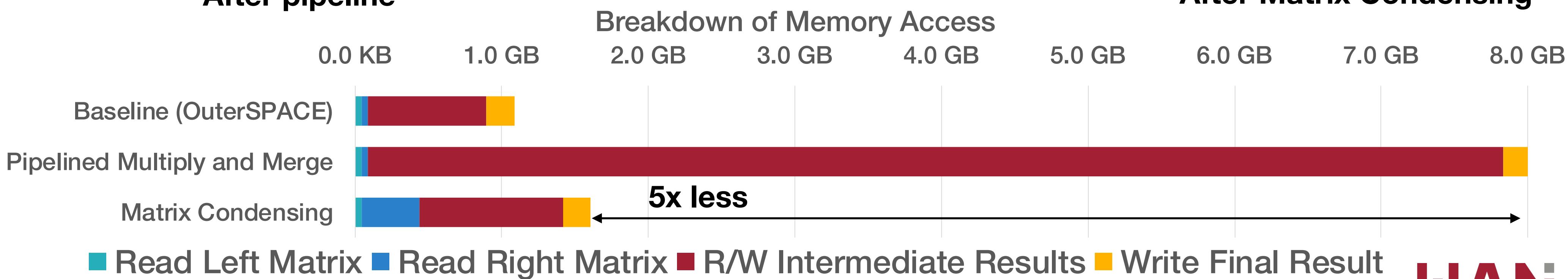


Technique 2: Matrix Condensing

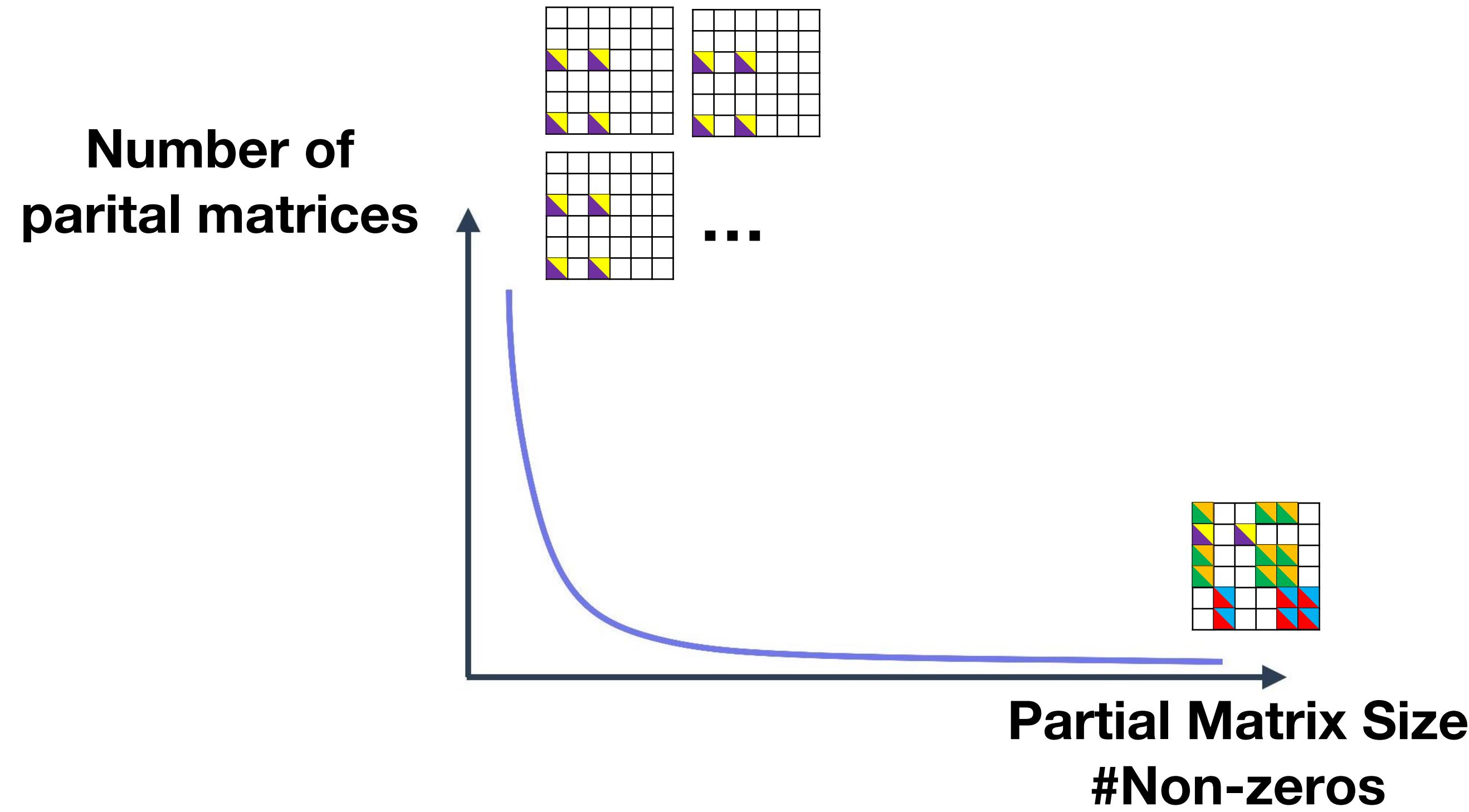
Distribution of DRAM access



Breakdown of Memory Access

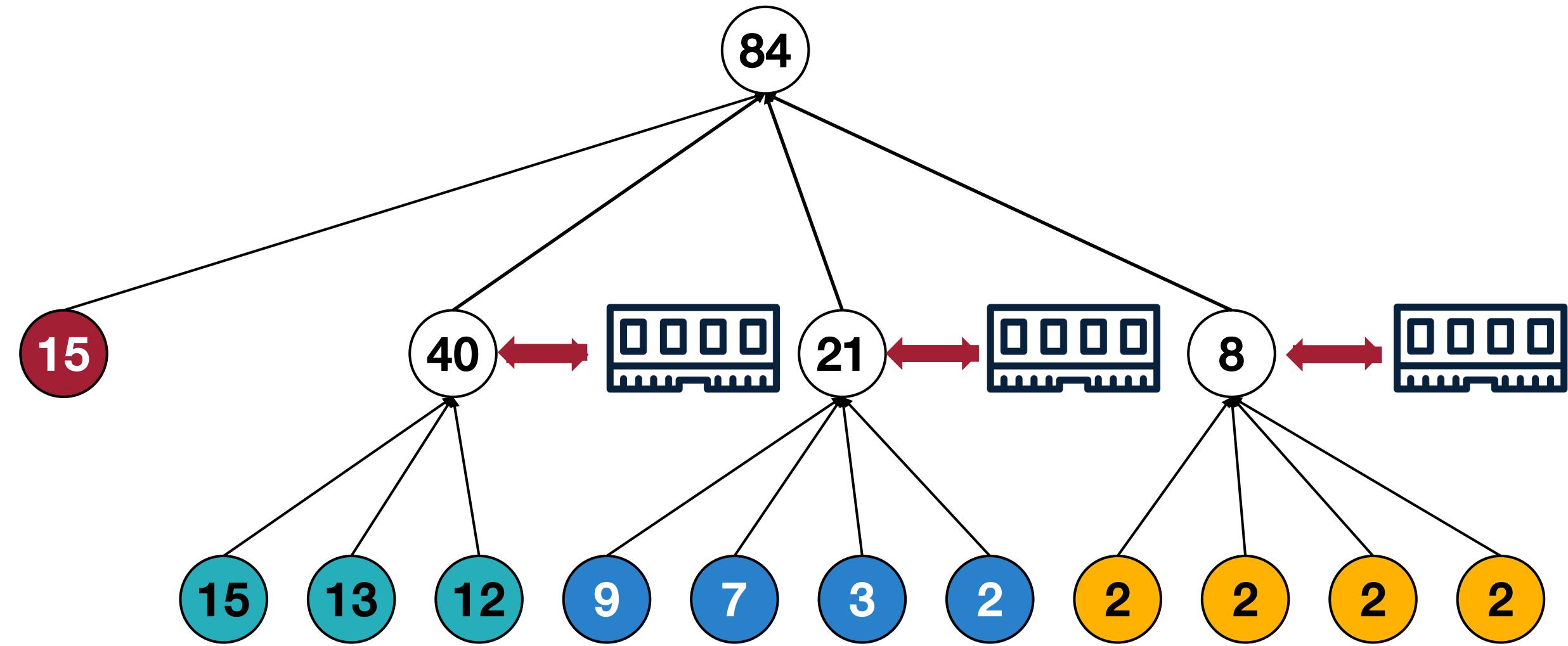
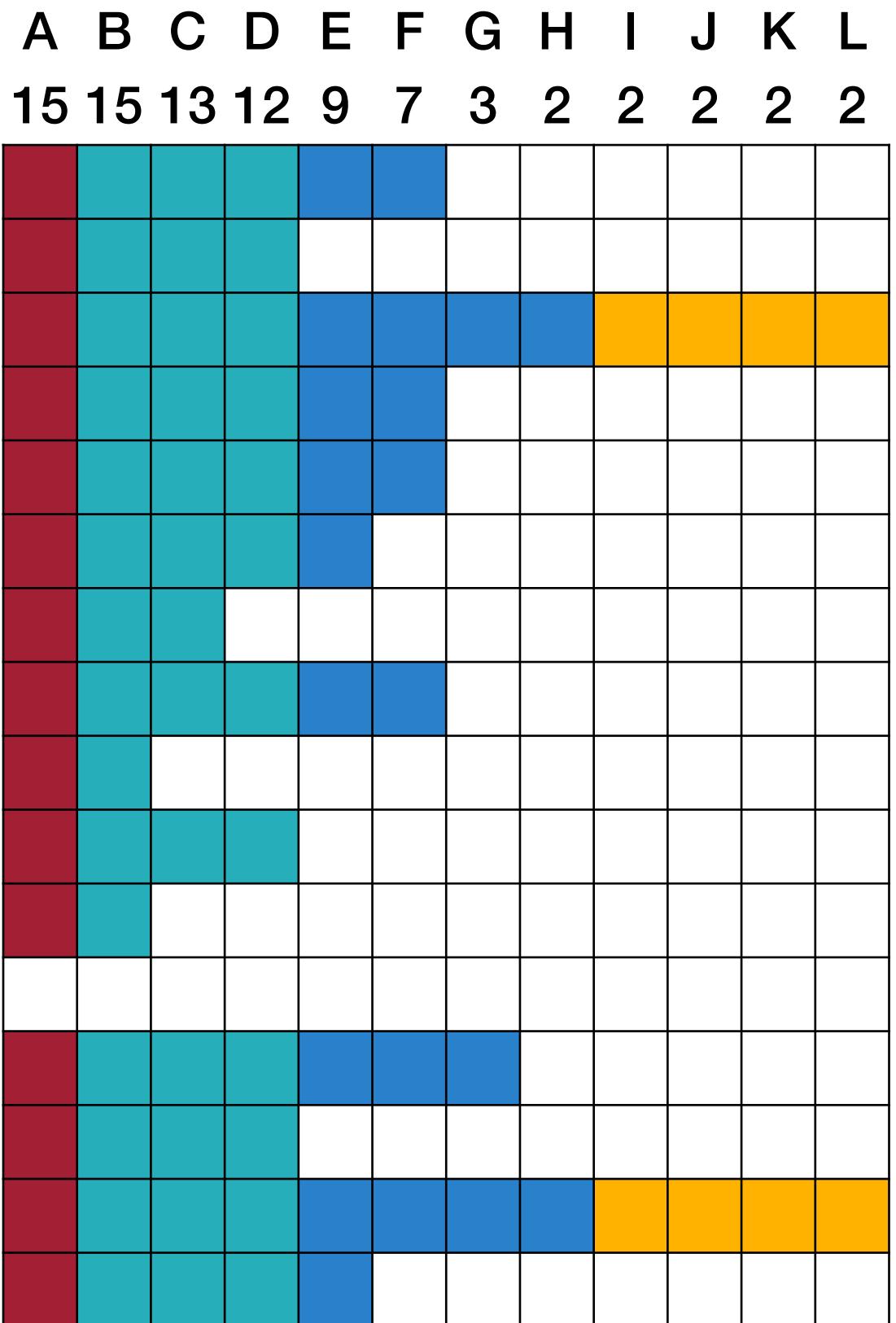


Technique 3: Huffman Tree Scheduler



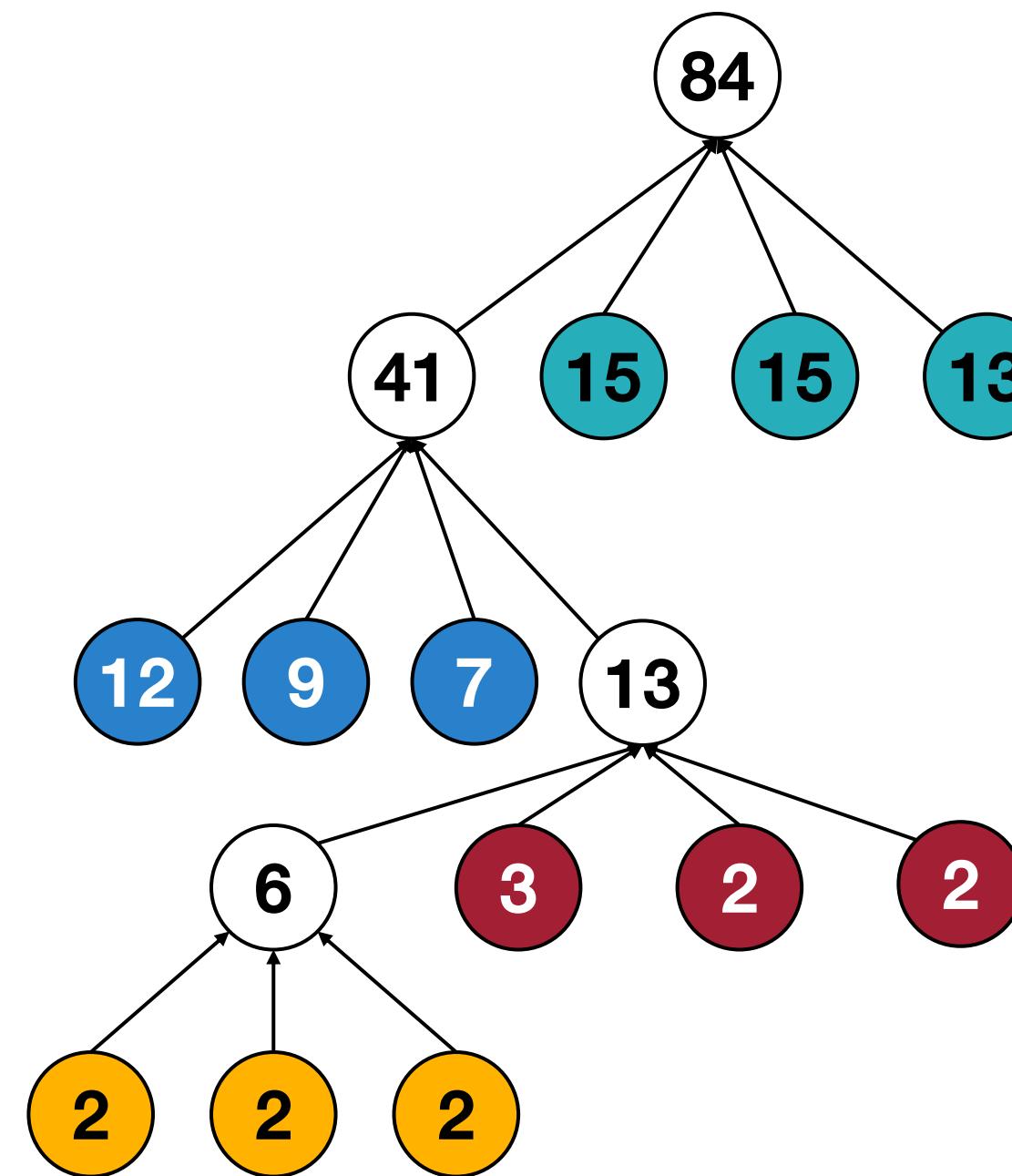
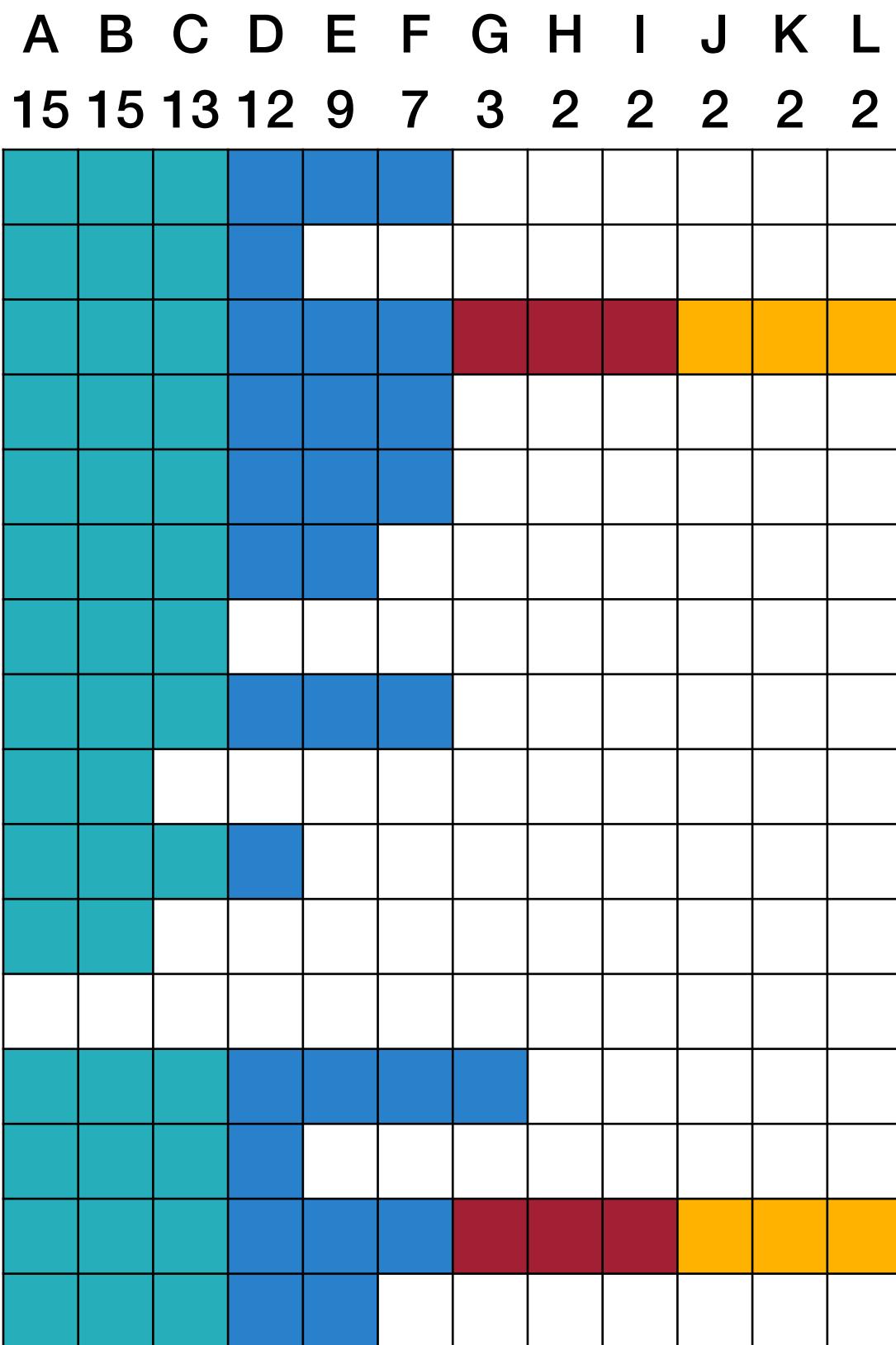
Most partial matrices contains few nonzeros

Technique 3: Huffman Tree Scheduler



$$\begin{aligned} & \# \text{DRAM access of intermediate partial results} \\ &= \sum_{k \text{ is a intermediate node}} \text{weight}[k] \\ &= 40 + 21 + 8 = 69 \end{aligned}$$

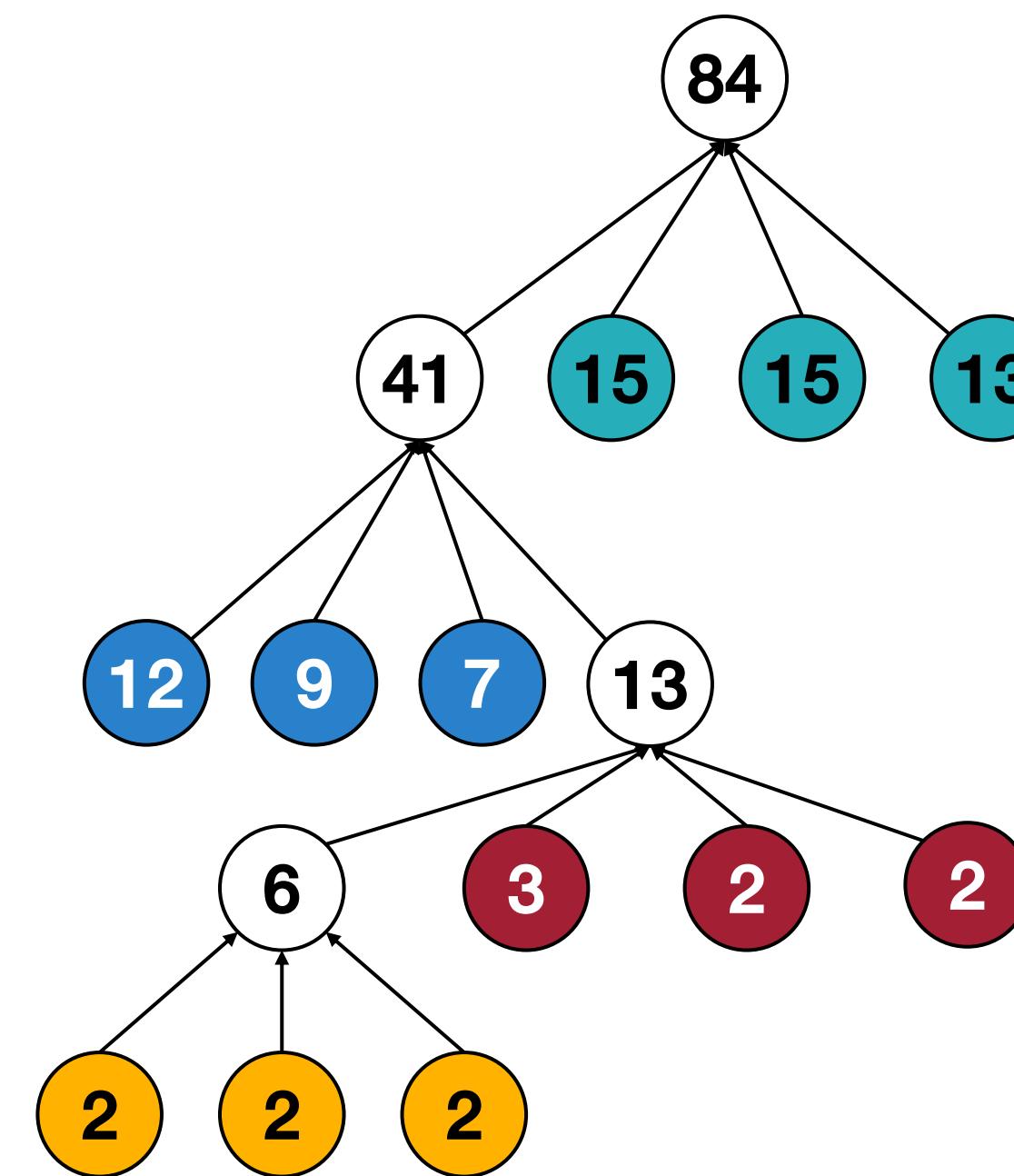
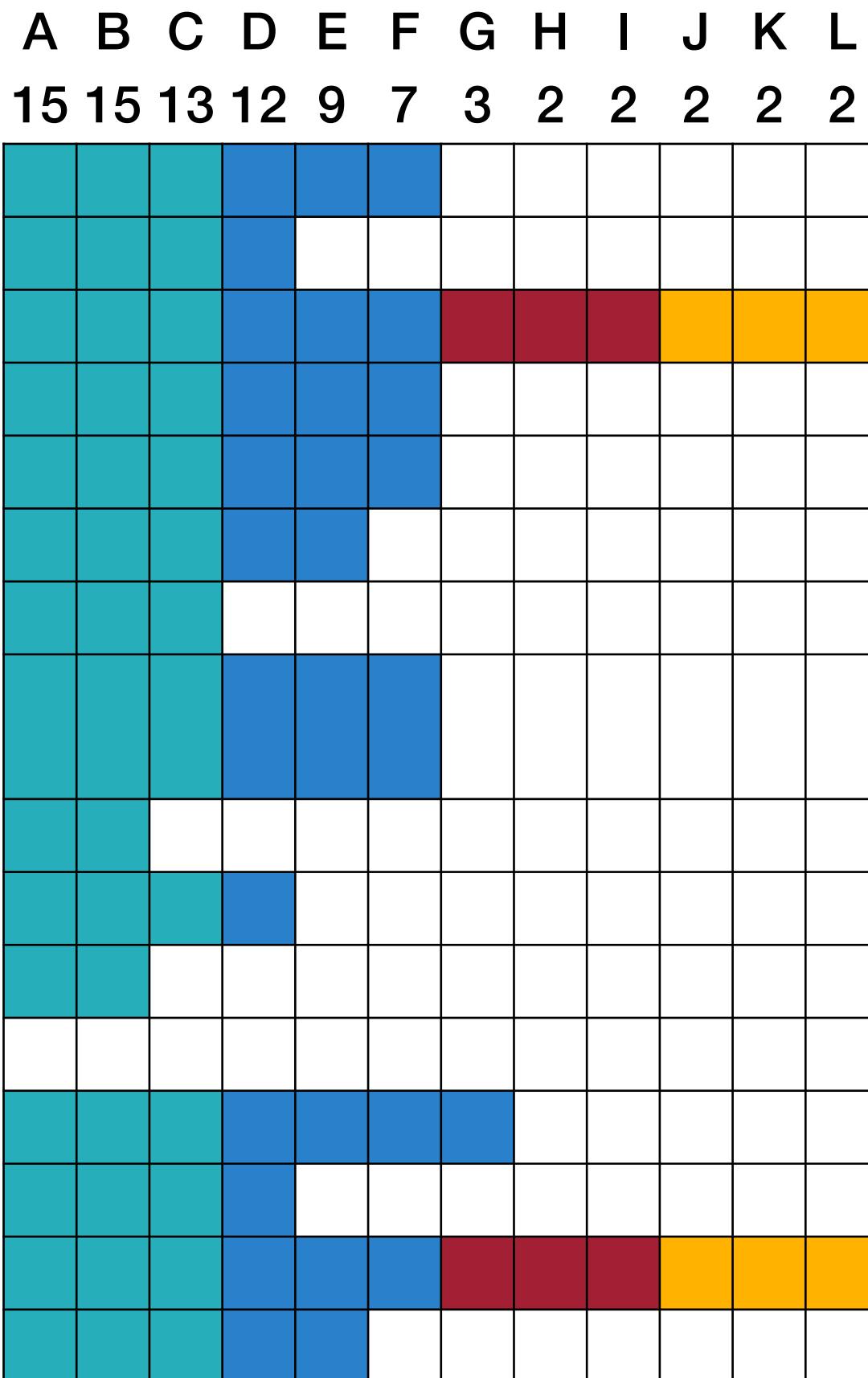
Technique 3: Huffman Tree Scheduler



DRAM access of intermediate partial results

$$= \sum_{k \text{ is a } \textcolor{red}{\text{intermediate}} \text{ node}} \text{weight}[k]$$
$$= 6 + 13 + 41 = 60$$

Technique 3: Huffman Tree Scheduler



DRAM access

$$= \sum_{k \text{ is a } \textcolor{red}{\text{intermediate node}}} \text{weight}[k]$$

$$= \sum_{k \text{ is a node}} \text{weight}[k] - \sum_{k \text{ is a } \textcolor{red}{\text{leaf node}}} \text{weight}[k]$$

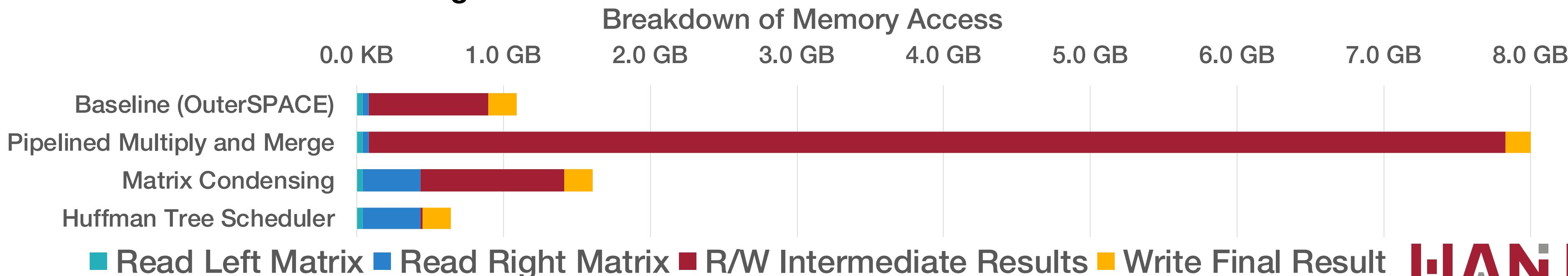
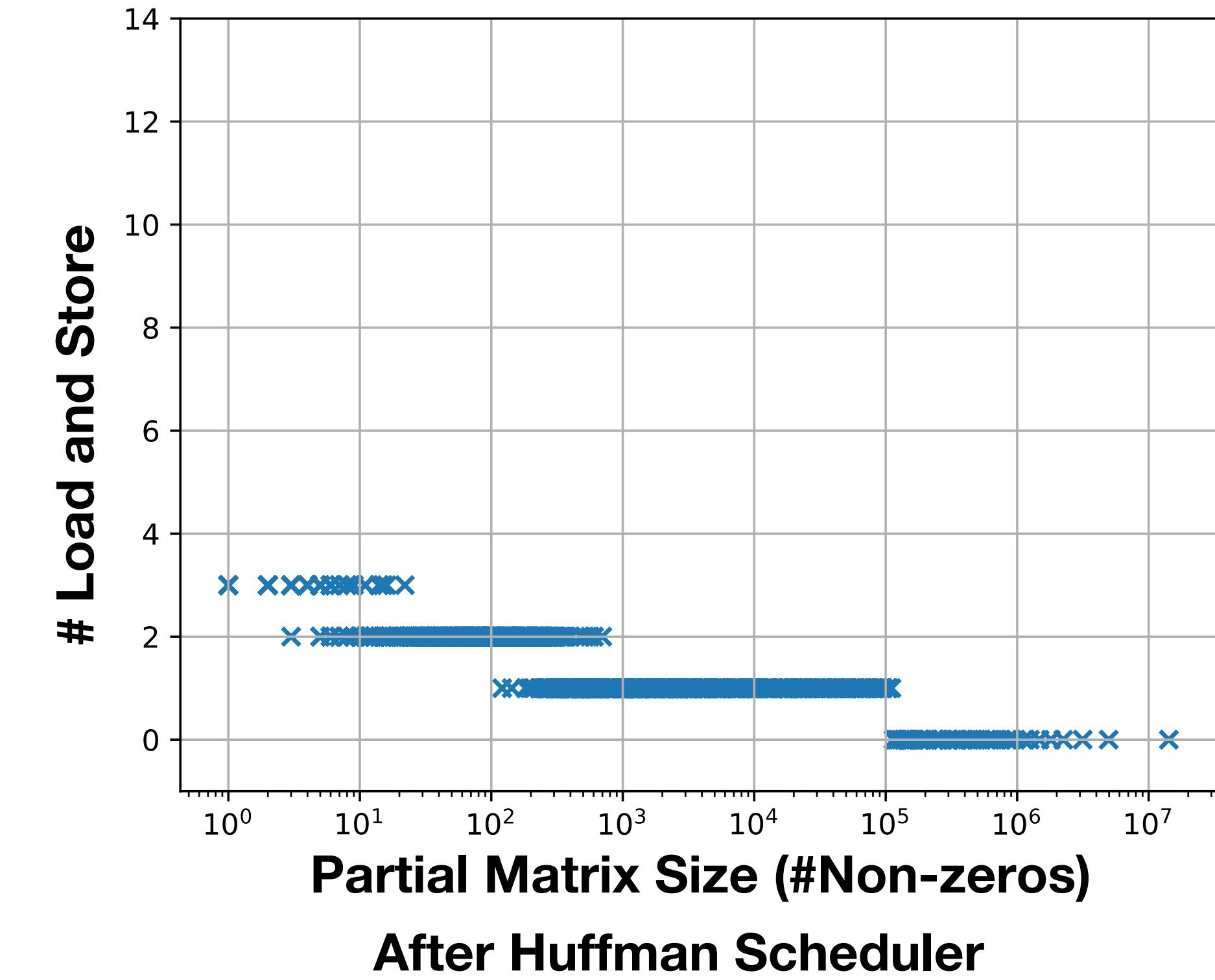
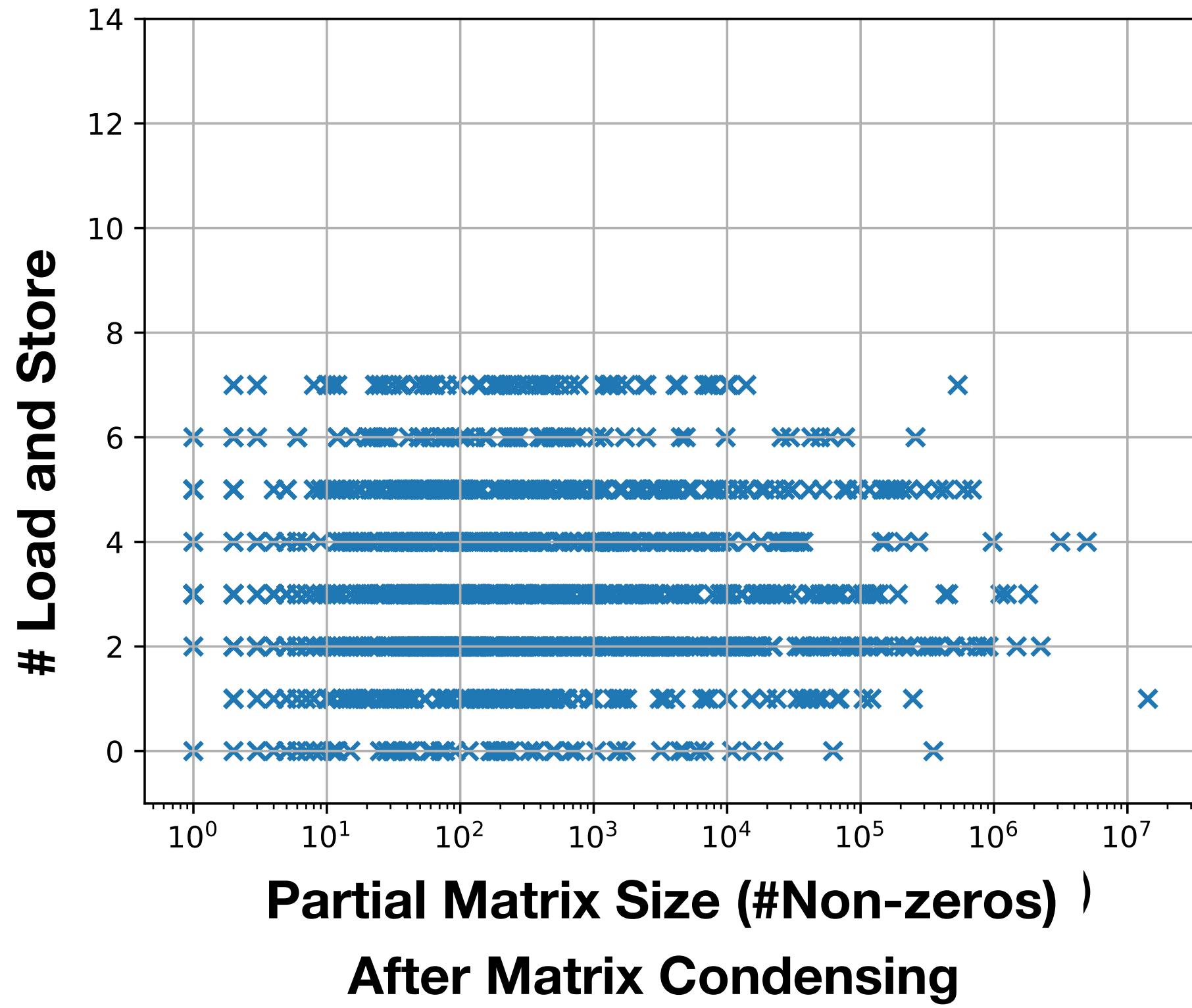
$$= \sum_{k \text{ is a } \textcolor{red}{\text{leaf node}}} \text{weight}[k] * \text{depth}[k] - \text{Constant}$$

Huffman coding minimizes

$$\sum_{k \text{ is a symbol}} \text{weight}[k] * \text{length}[k]$$

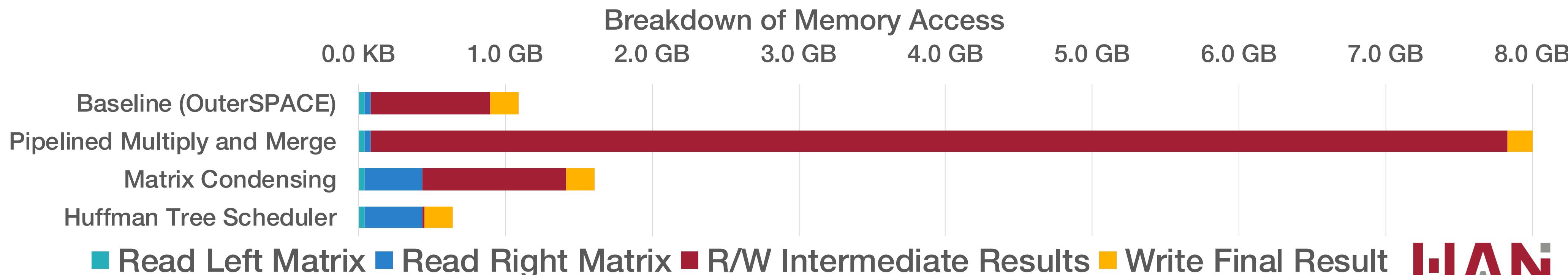
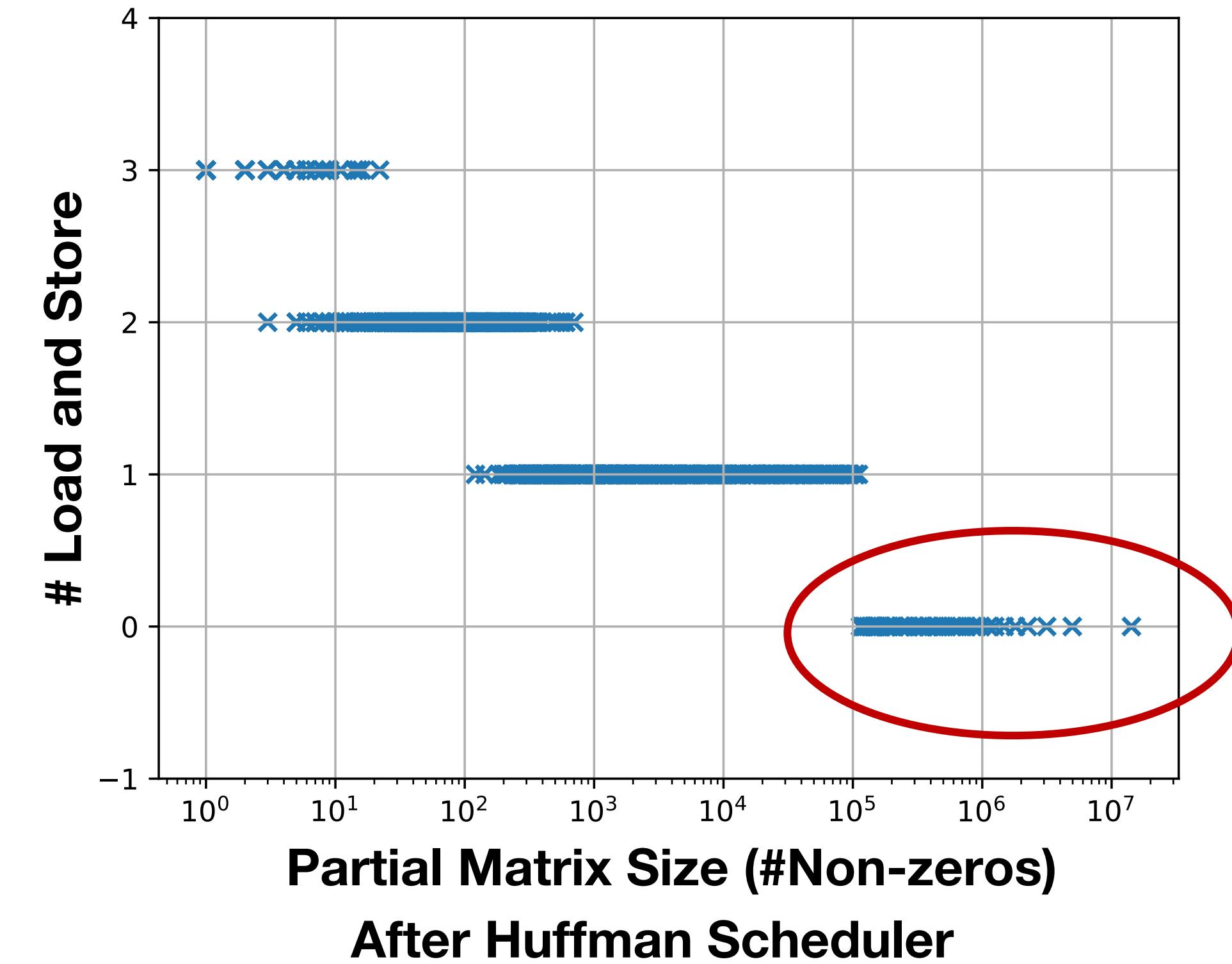
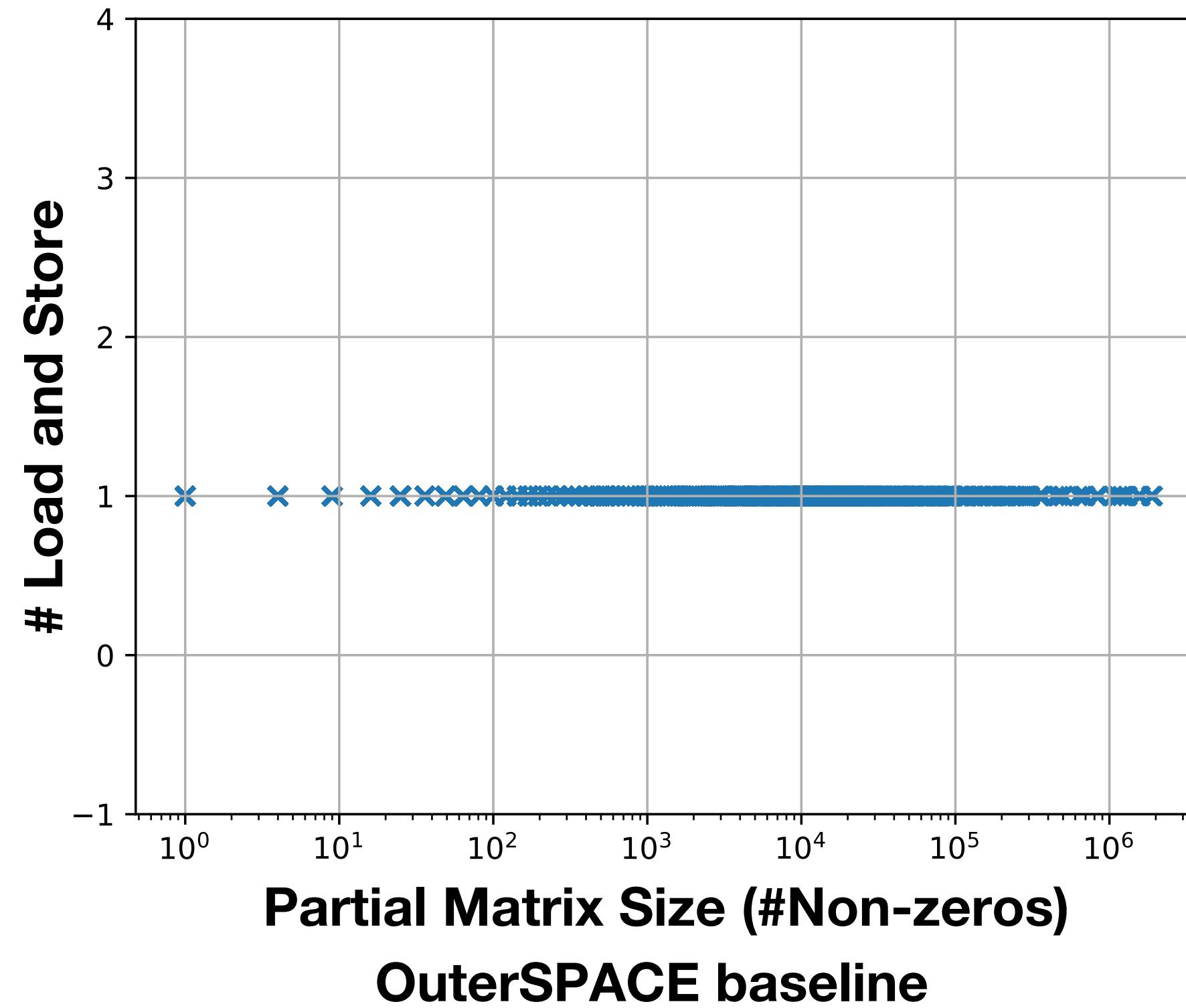
Technique 3: Huffman Tree Scheduler

Distribution of DRAM access

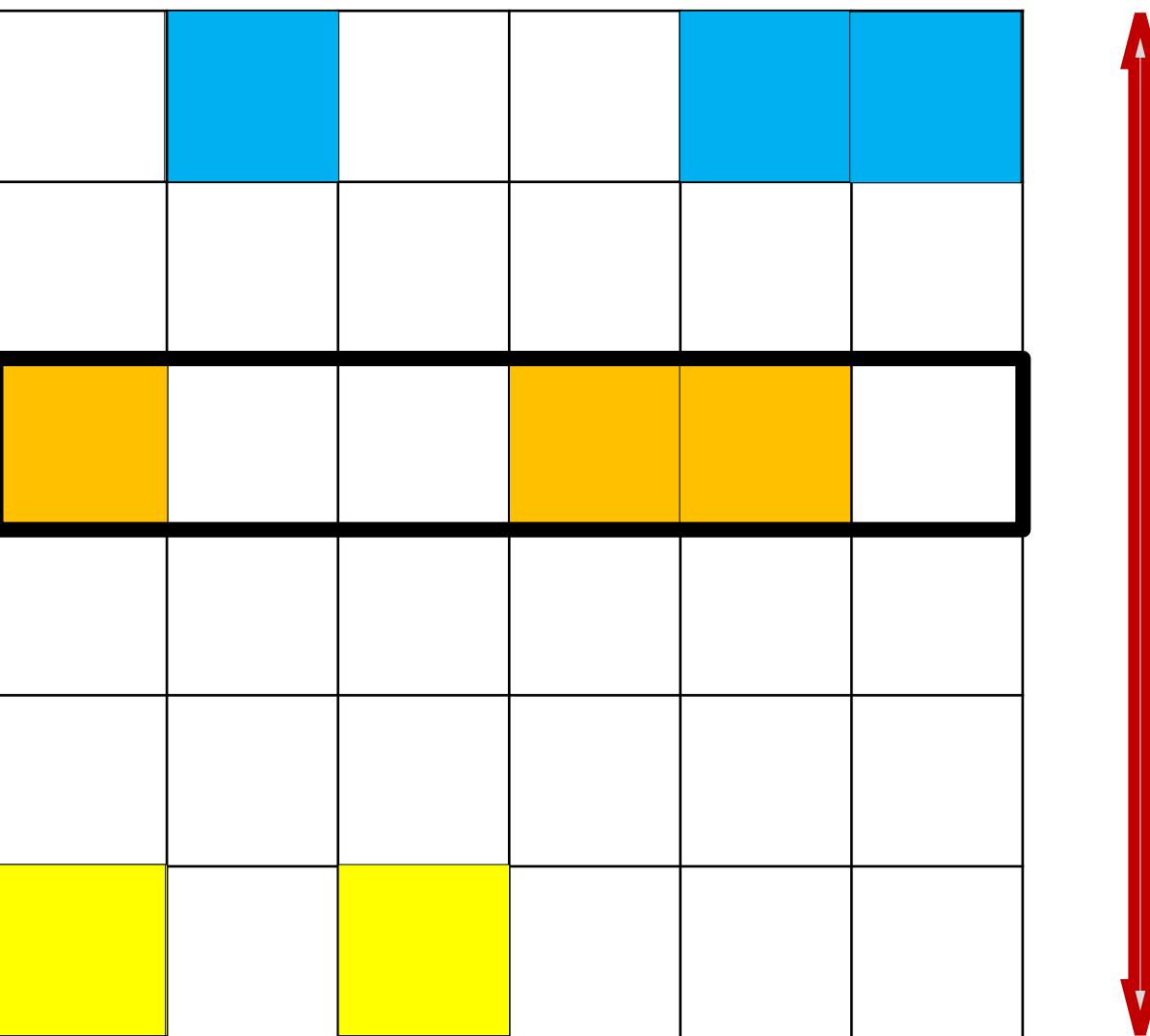


Technique 3: Huffman Tree Scheduler

Distribution of DRAM access



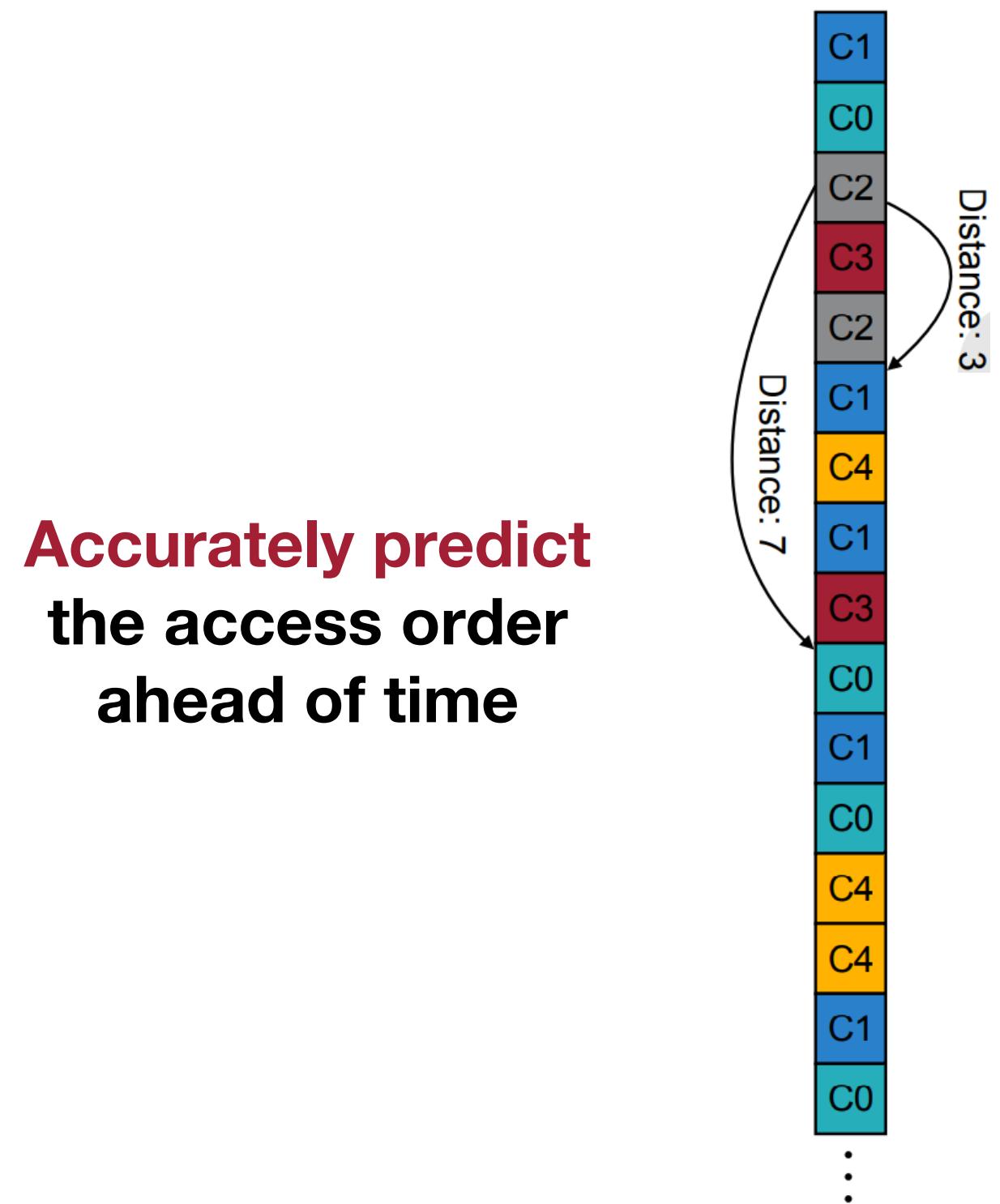
Technique 4: Row Prefetcher



After Condensing:
Access to right matrix becomes irregular
A cache to the rescue!

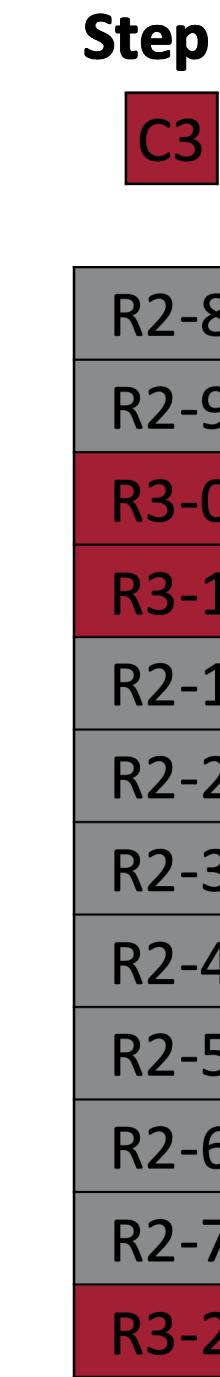
Technique 4: Row Prefetcher

Predicted Order using Matrix A

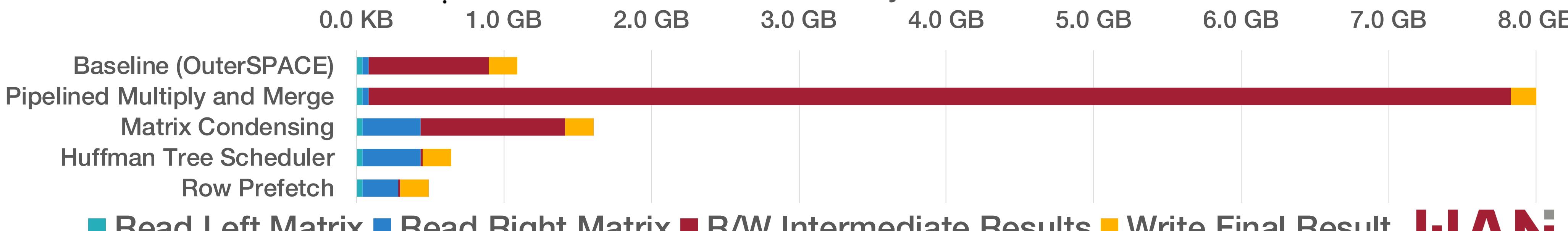


**Accurately predict
the access order
ahead of time**

On-chip Buffer



Breakdown of Memory Access



Rows of second input Matrix B

R0	R0-0	R0-1	R0-2	R0-3	R0-4	R0-5	R0-6
R1	R1-0	R1-1	R1-2	R1-3			
R2	R2-0	R2-1	R2-2	R2-3	R2-4	R2-5	R2-6
R3	R3-0	R3-1	R3-2				
R4	R4-0	R4-1	R4-2	R4-3	R4-4	R4-5	

Prefetch the rows before used

Evaluation Setup

- Setups

	OuterSPACE	Ours
Technology	TSMC 32nm	TSMC 40nm
Area	87 mm ²	28.49 mm ²
Power	12.39 W	9.26 W
DRAM	HBM@128GB/s	HBM@128GB/s

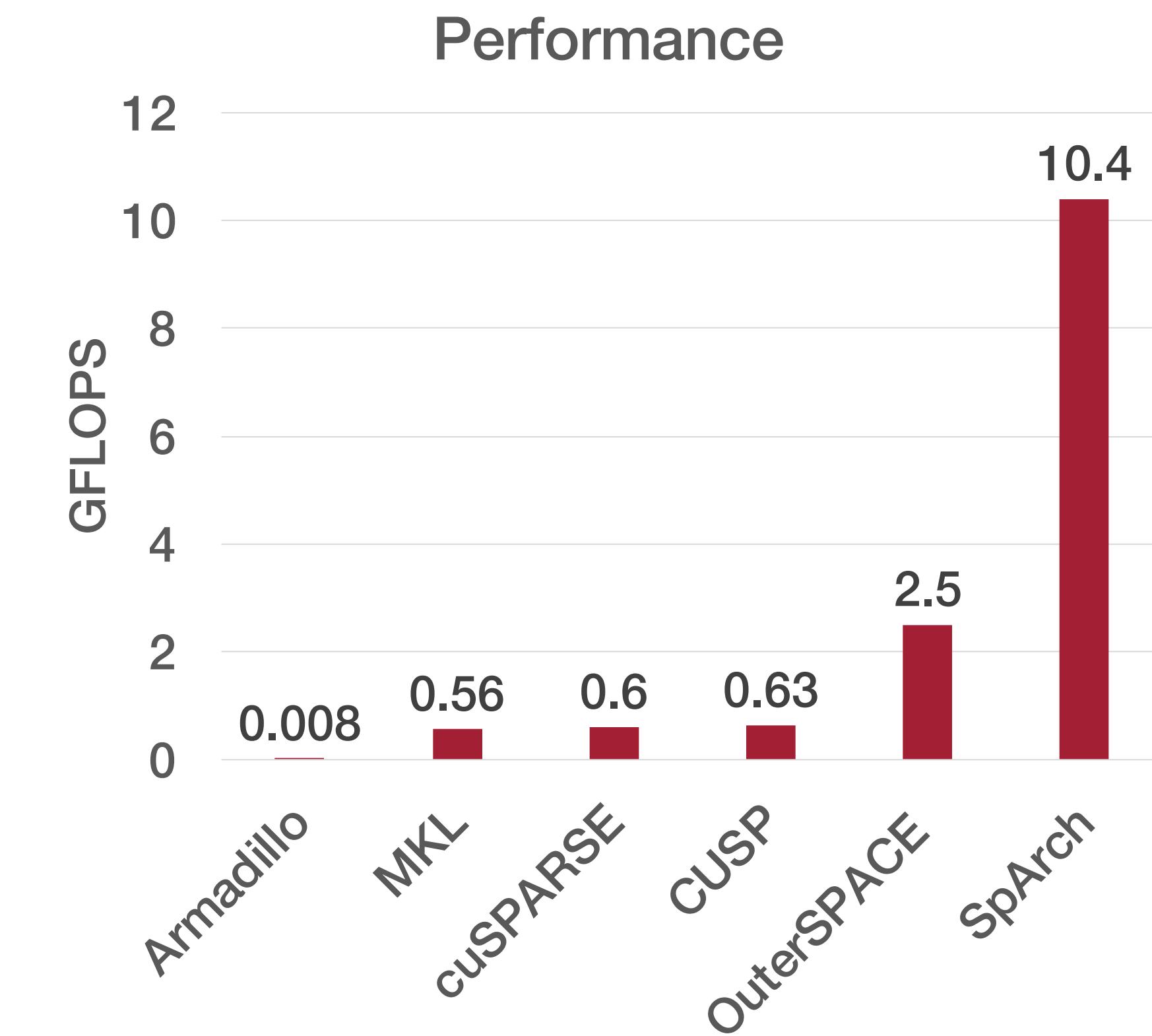
- Benchmarks
 - SuiteSparse Matrix Collection
 - Stanford Network Analysis Project (SNAP) dataset

Evaluation

On SuiteSparse & SNAP:

- **4.2x** speedup and **6x** energy efficiency over prior SOTA (OuterSPACE)

Platform	Performance (GFLOPS)
Armadillo (ARM)	0.008 (1285x)
MKL (CPU)	0.56 (19x)
cuSPARSE (GPU)	0.60 (18x)
CUSP (GPU)	0.63 (17x)
OuterSPACE (ASIC)	2.5 (4x)
SpArch (ASIC)	10.4

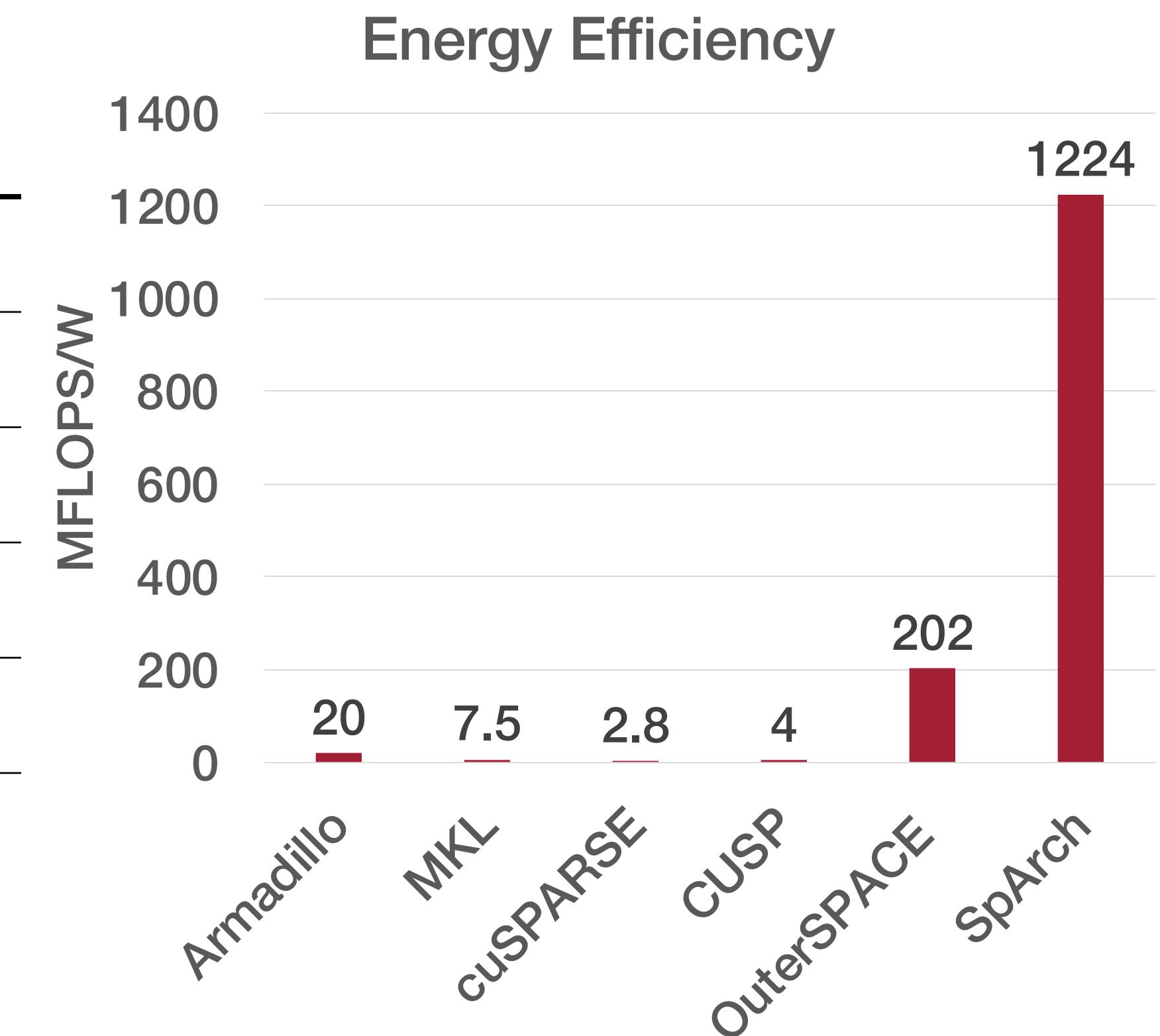


Evaluation

On SuiteSparse & SNAP:

- **4.2x** speedup and **6x** energy efficiency over prior SOTA (OuterSPACE)

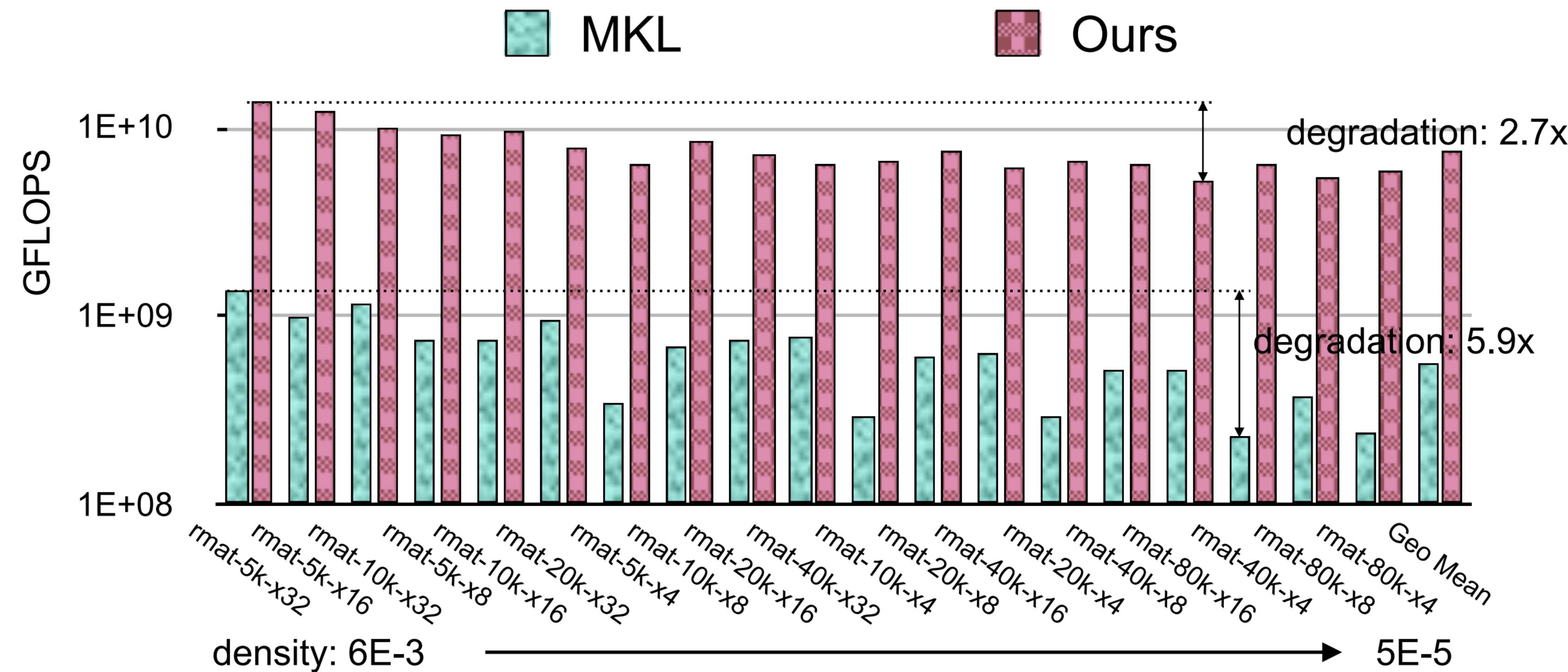
Platform	Power (W)	Energy Efficiency (MFLOPS/W)
Armadillo (ARM)	0.4	20 (62x)
MKL (CPU)	74	7.5 (164x)
cuSPARSE (GPU)	210	2.8 (435x)
CUSP (GPU)	157	4.0 (307x)
OuterSPACE (ASIC)	12.4	202 (6x)
SpArch (ASIC)	8.5	1224



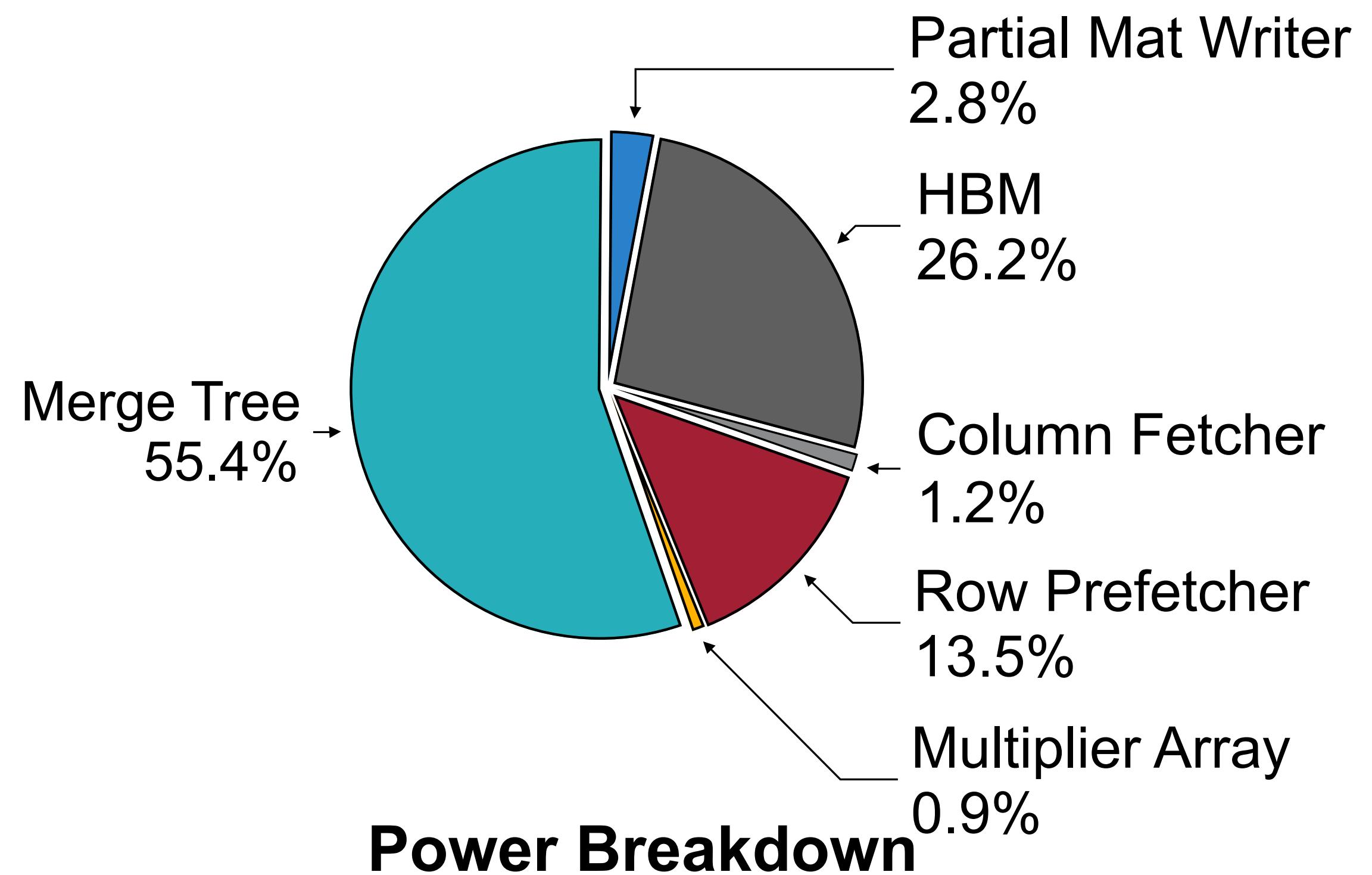
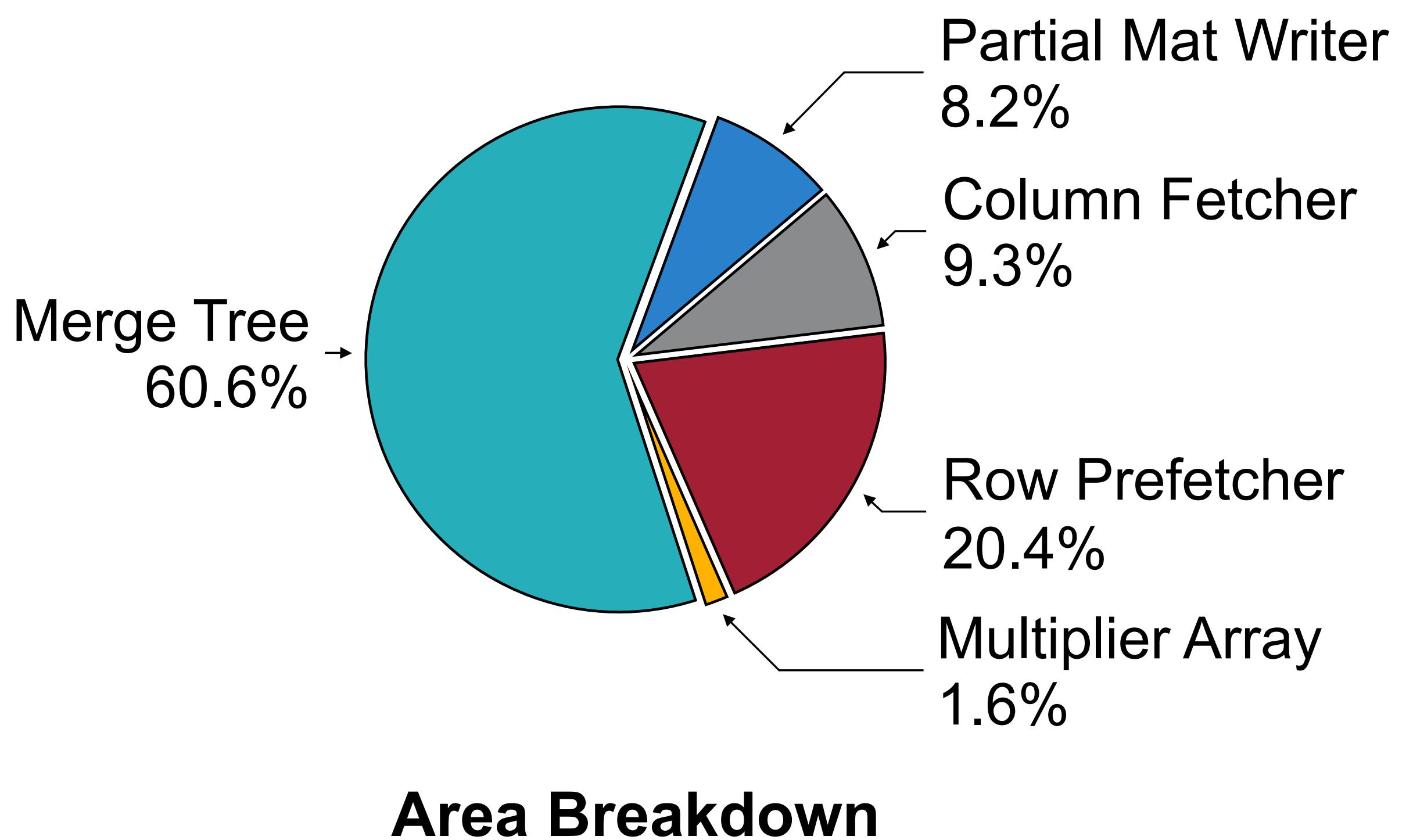
Evaluation

On RMAT matrices:

- 10x~30x faster than Intel CPU (MKL).
- Higher scalability on ultra-sparse matrices.
 - MKL performance degradation: 5.9x, Ours: 2.7x



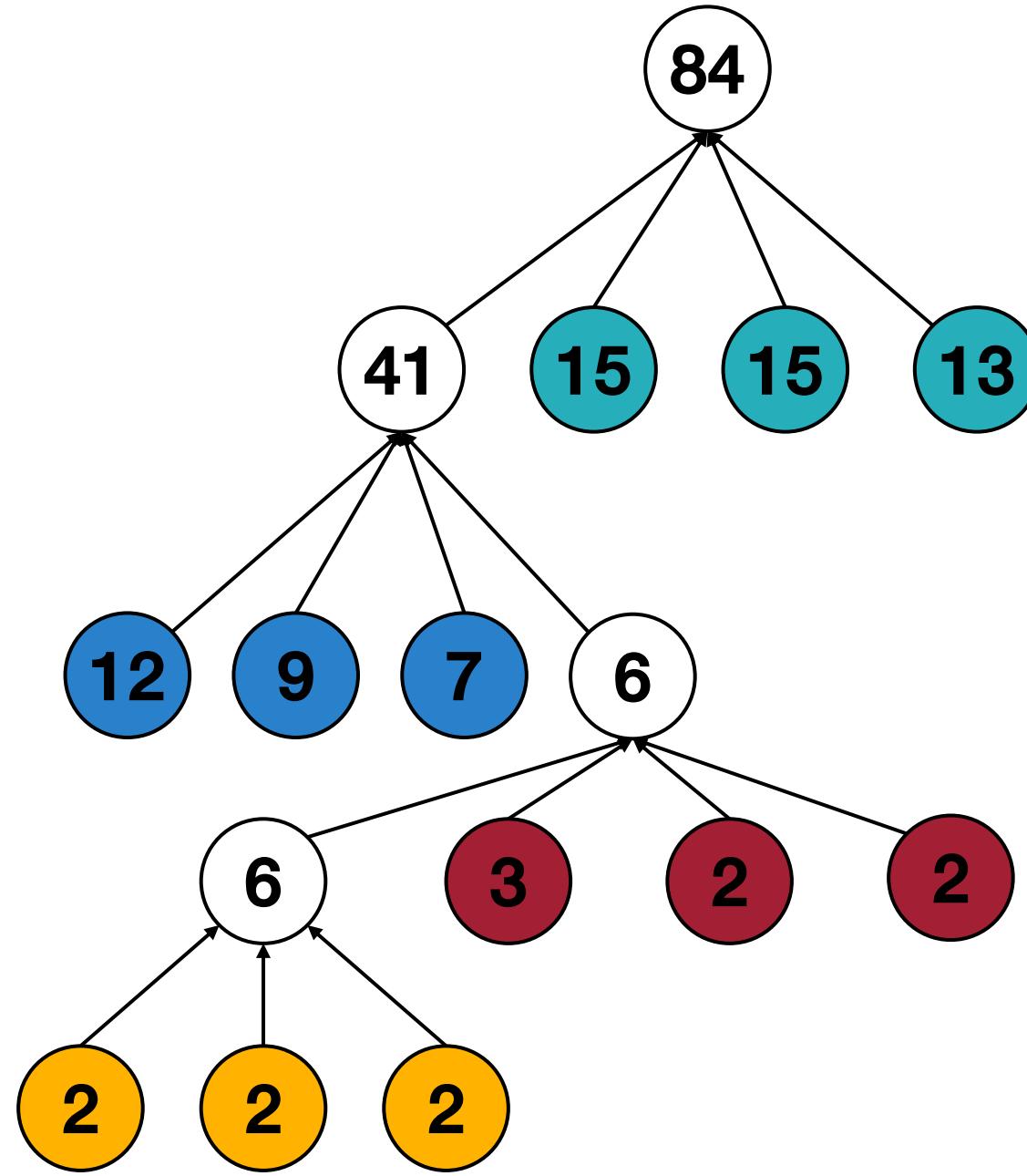
Evaluation





sparch.mit.edu

Conclusion



- SpMM is an important primitive (graphs, sparse neural networks)
- SpArch accelerates SpMM by using a **Spatial Merge Array** and **Huffman Tree Scheduler** to reduce the DRAM access of partial matrices.
- Achieve **4.2x** speedup and **6x** energy efficiency over prior SOTA (OuterSPACE)

Thank you!