SparseP

Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures

Christina Giannoula

Ivan Fernandez, Juan Gomez-Luna,

Nectarios Koziris, Georgios Goumas, Onur Mutlu

SAFARI ETH zürich

National Technical University of Athens CSLab

UNIVERSIDAD DE MÁLAGA

Our Work

Efficient Algorithmic Designs

The first open-source Sparse Matrix Vector Multiplication (SpMV) software package, SparseP, for real Processing-In-Memory (PIM) systems

SparseP is Open-Source

SparseP: https://github.com/CMU-SAFARI/SparseP

Extensive Characterization

The first comprehensive analysis of SpMV on the first real commercial PIM architecture

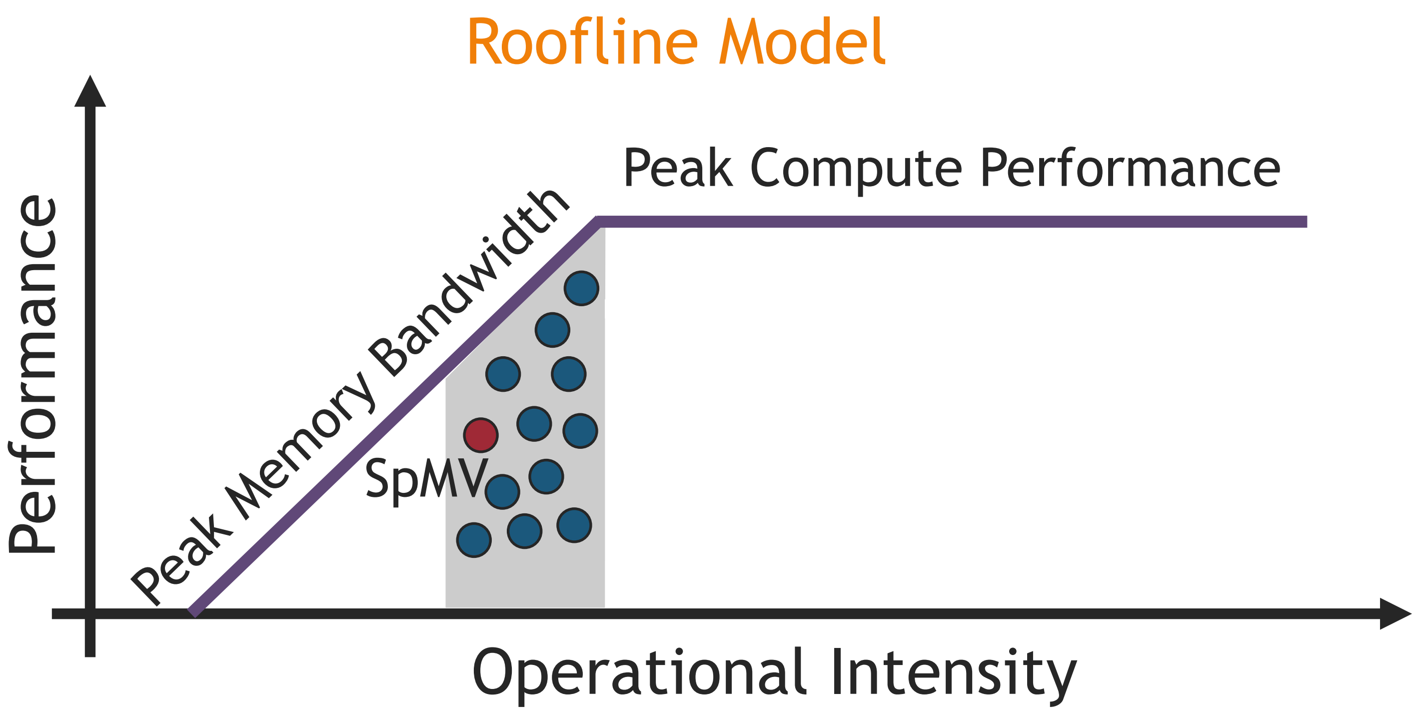
Recommendations for Architects and Programmers

Full Paper: https://arxiv.org/pdf/2201.05072.pdf

Sparse Matrix Vector Multiplication

Sparse Matrix Vector Multiplication (SpMV):

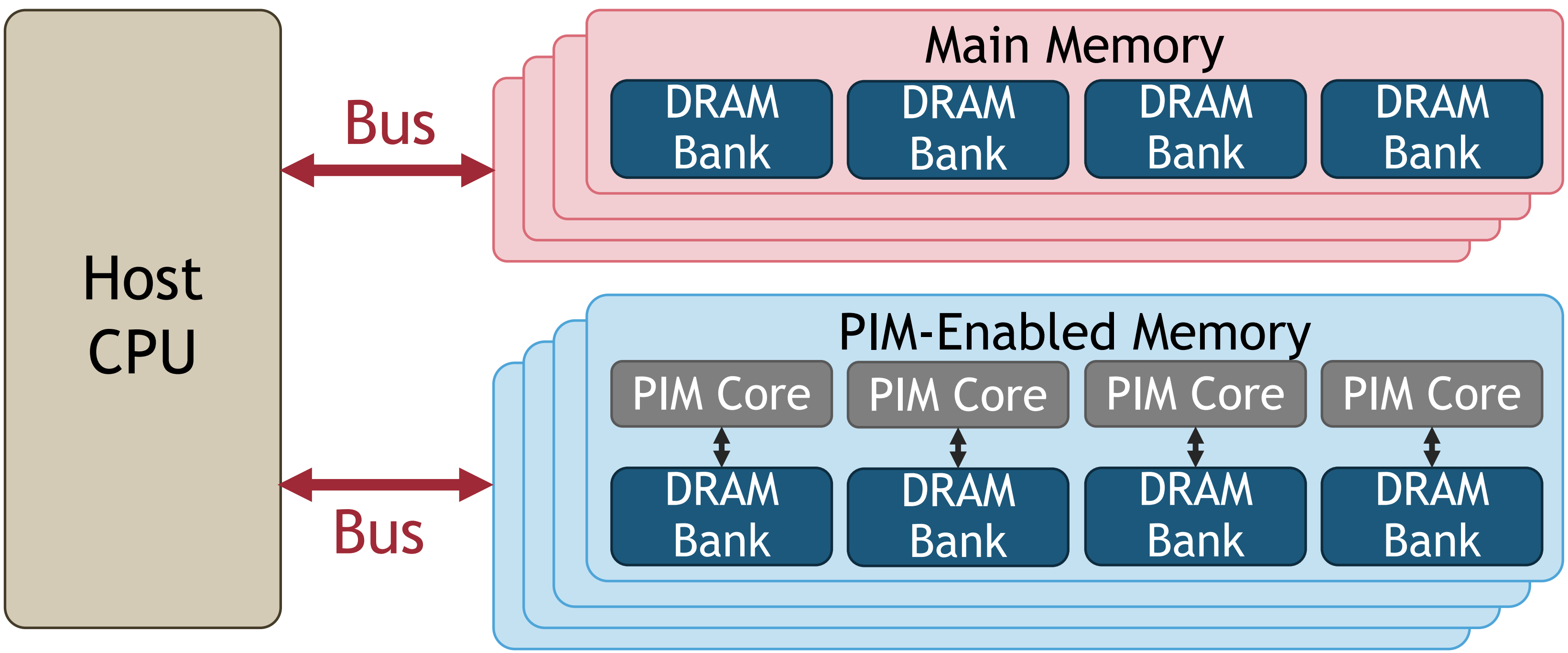
- Widely-used kernel in graph processing, machine learning, scientific computing …
- A highly memory-bound kernel



Real Processing-In-Memory Systems

Real Near-Bank Processing-In-Memory (PIM) Systems:

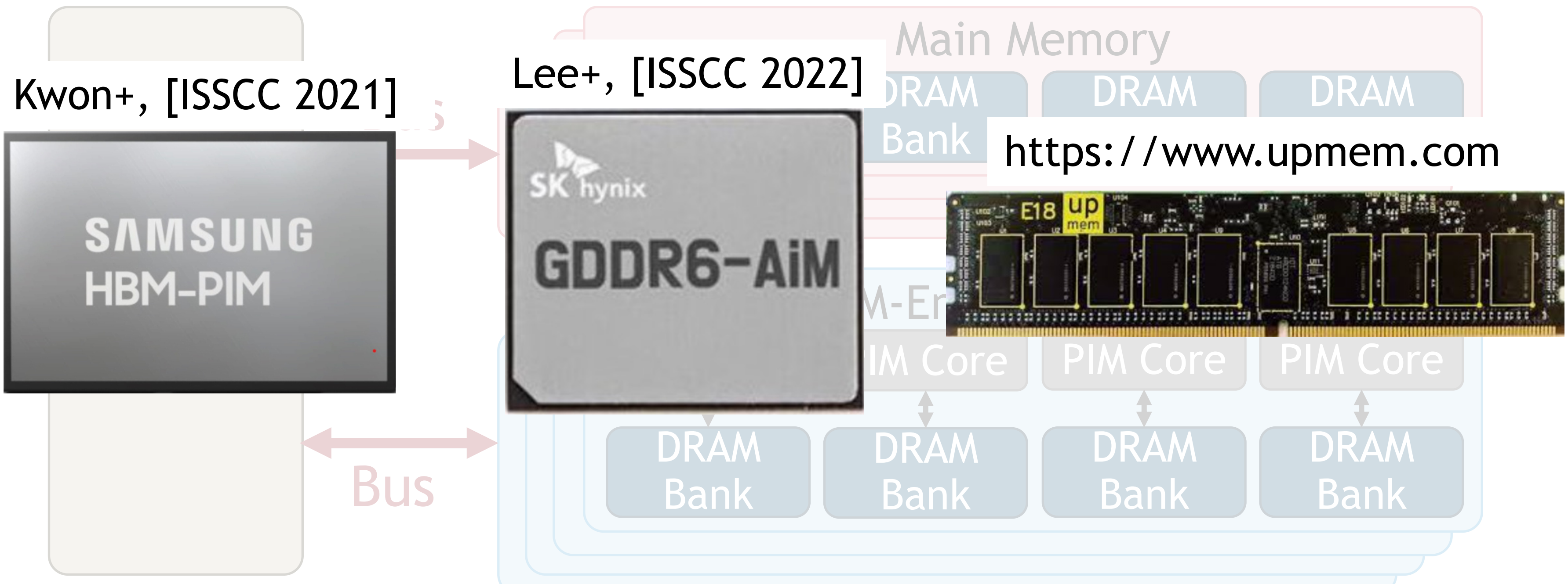
- High levels of parallelism
- Low memory access latency
- Large aggregate memory bandwidth



Real Processing-In-Memory Systems

Real Near-Bank Processing-In-Memory (PIM) Systems:

- High levels of parallelism
- Low memory access latency
- Large aggregate memory bandwidth



SparseP: SpMV Library for Real PIMs

Our Contributions:

1. Design efficient SpMV kernels for current and future PIM systems
   - 25 SpMV kernels
     - 4 compressed matrix formats (CSR, COO, BCSR, BCOO)
     - 6 data types
     - 4 data partitioning techniques
     - Various load balancing schemes among PIM cores/threads
     - 3 synchronization approaches

2. Provide a comprehensive analysis of SpMV on the first commercially-available real PIM system
   - 26 sparse matrices
   - Comparisons to state-of-the-art CPU and GPU systems
   - Recommendations for software, system and hardware designers

Outline

SpMV Kernels for Real PIM Systems

Key Takeaways from Our Study

Conclusion

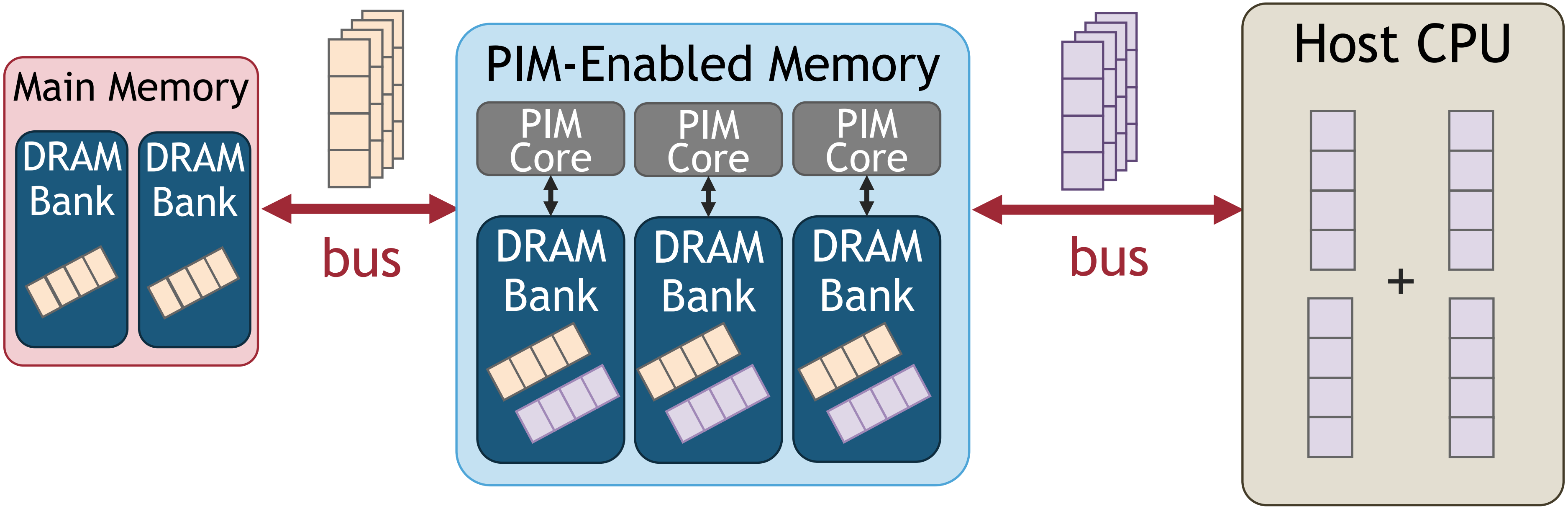
SpMV Execution on a PIM System

1 Load the input vector

2 Execute the kernel

3 Retrieve the partial results

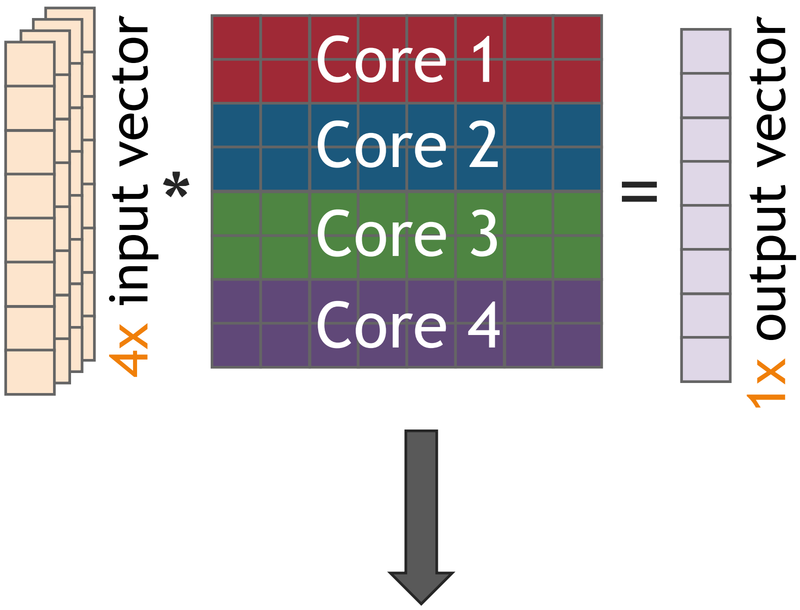
4 Merge the partial results



Data Partitioning Techniques

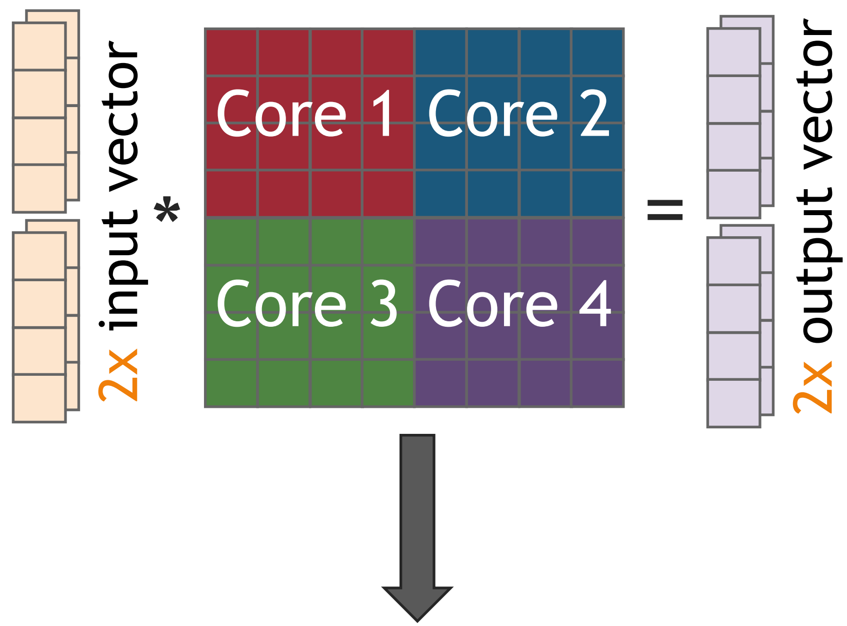
SparseP supports two types of data partitioning techniques:

1D Partitioning



perform the complete SpMV computation only on PIM cores

2D Partitioning



trade-off computation vs data transfer costs

1D Partitioning Technique

Load-Balancing Approaches:

- CSR, COO:
  - Balance Rows
  - Balance NNZs *

- BCSR, BCOO:
  - Balance Blocks ^
  - Balance NNZs ^

* row-granularity for CSR
^ block-row-granularity for BCSR

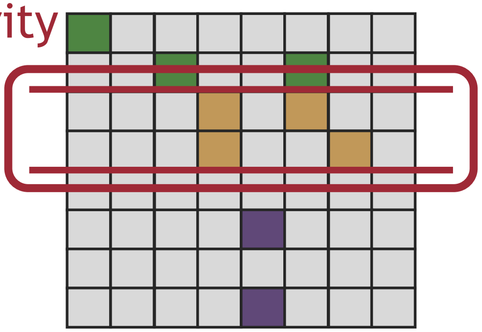
1D Partitioning Technique

Load-Balancing of #NNZs:

- CSR (row-granularity), COO

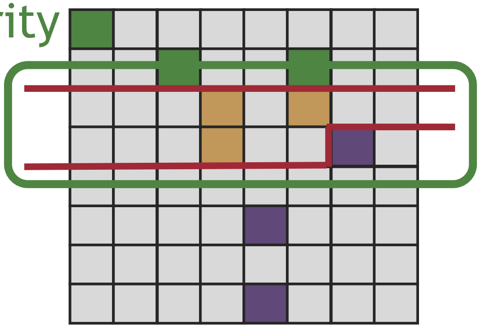
CSR

row-granularity



COO

nnz-granularity



row-order

rowptr	0	1	3	5	7	7	8	8	9
colind	0	2	5	3	5	3	6	4	4
values	2	1	8	3	6	9	3	4	7

nnz-order

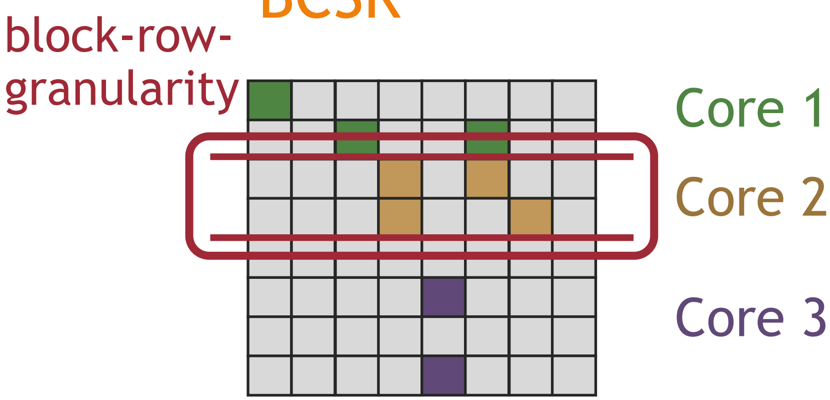
rowind	0	1	1	2	2	3	3	5	7
colind	0	2	5	3	5	3	6	4	4
values	2	1	8	3	6	9	3	4	7

1D Partitioning Technique

Load-Balancing of #NNZs:

- CSR (row-granularity), COO
- BCSR (block-row-granularity), BCOO

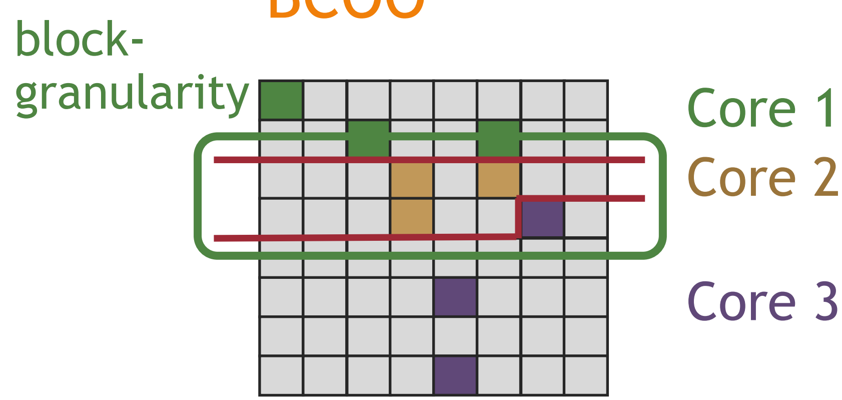
BCSR



block-row-order

rowptr	0	1	3	5	7	7	8	8	9
colind	0	2	5	3	5	3	6	4	4
values	2	1	8	3	6	9	3	4	7

BCOO

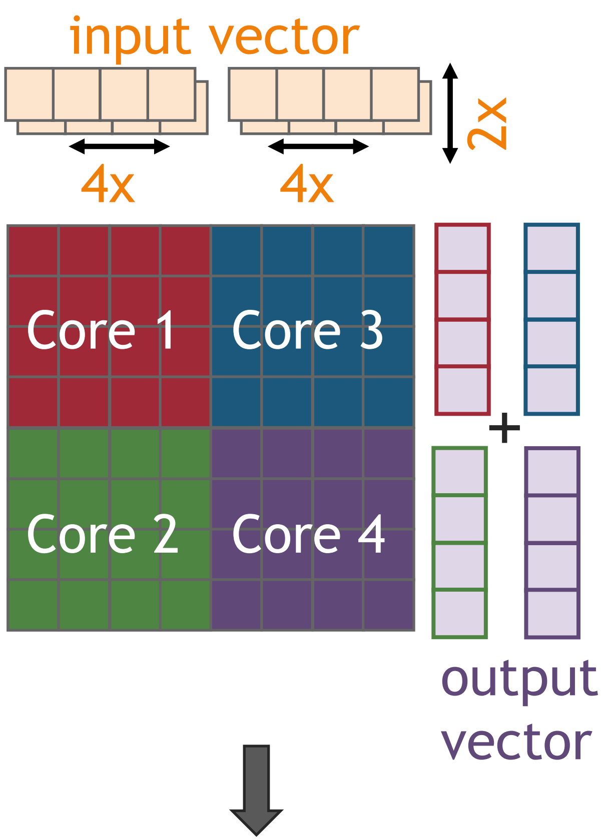


block-order

rowind	0	1	1	2	2	3	3	5	7
colind	0	2	5	3	5	3	6	4	4
values	2	1	8	3	6	9	3	4	7

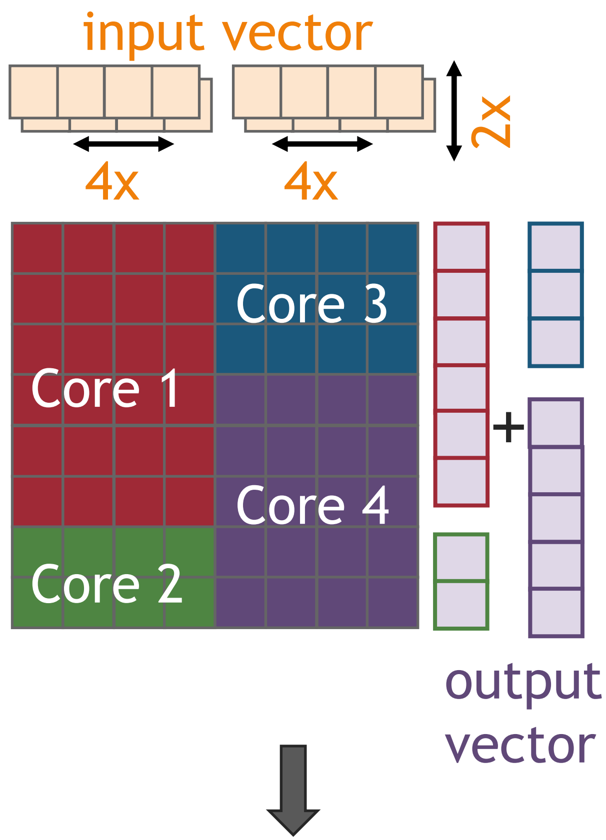
2D Partitioning Technique

Equally-Sized Tiles



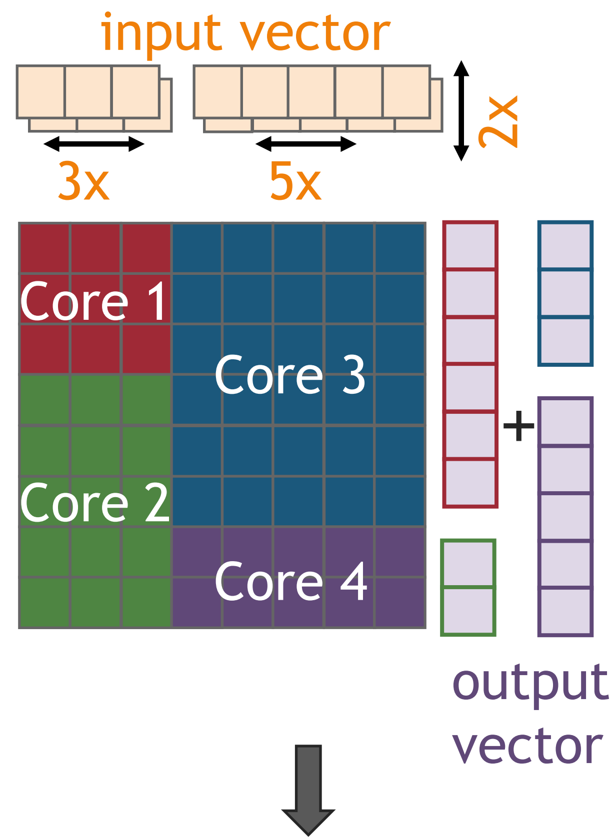
High NNZ imbalance across PIM cores

Equally-Wide Tiles



High NNZ balance across PIM cores of the same vertical partition

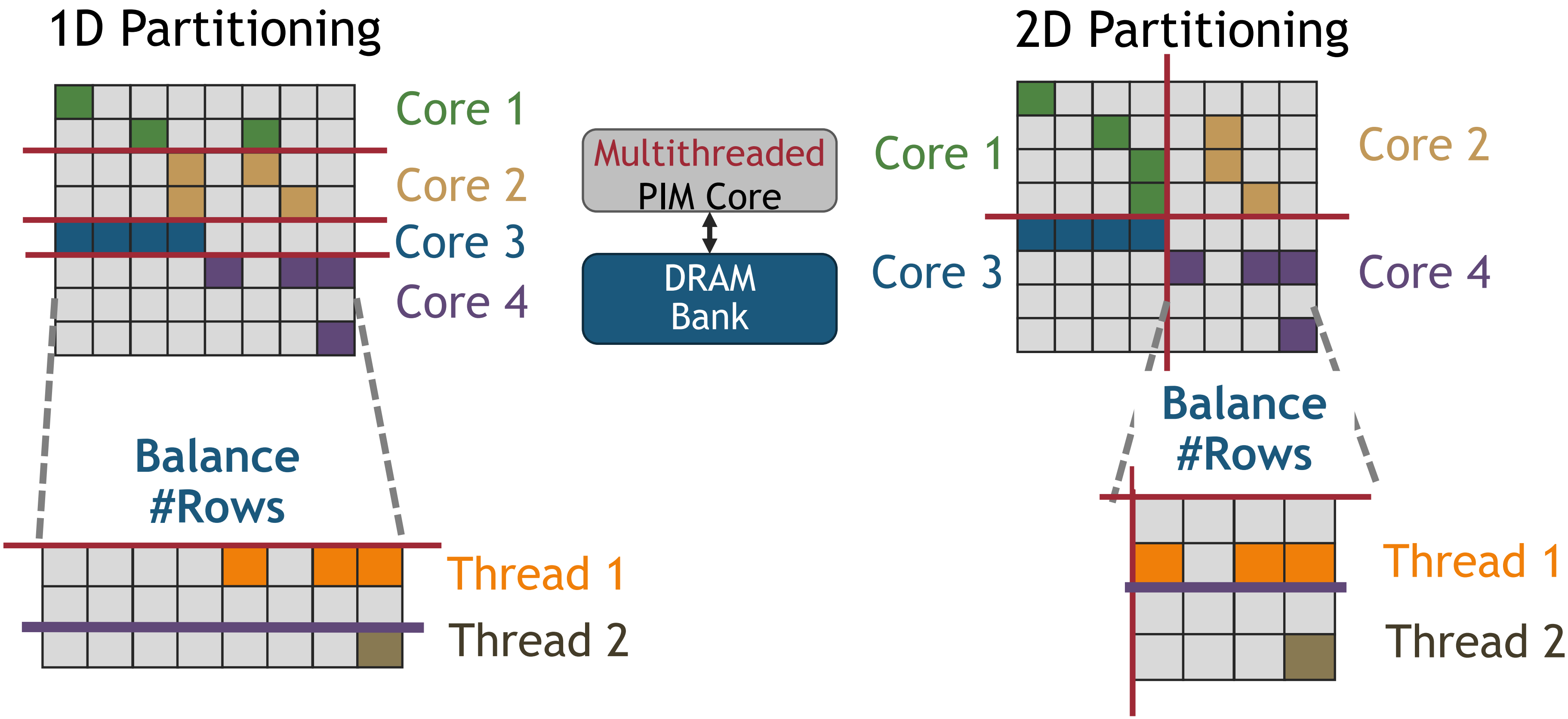
Variable-Sized Tiles



High NNZ balance across all PIM cores

Load-Balance across Threads

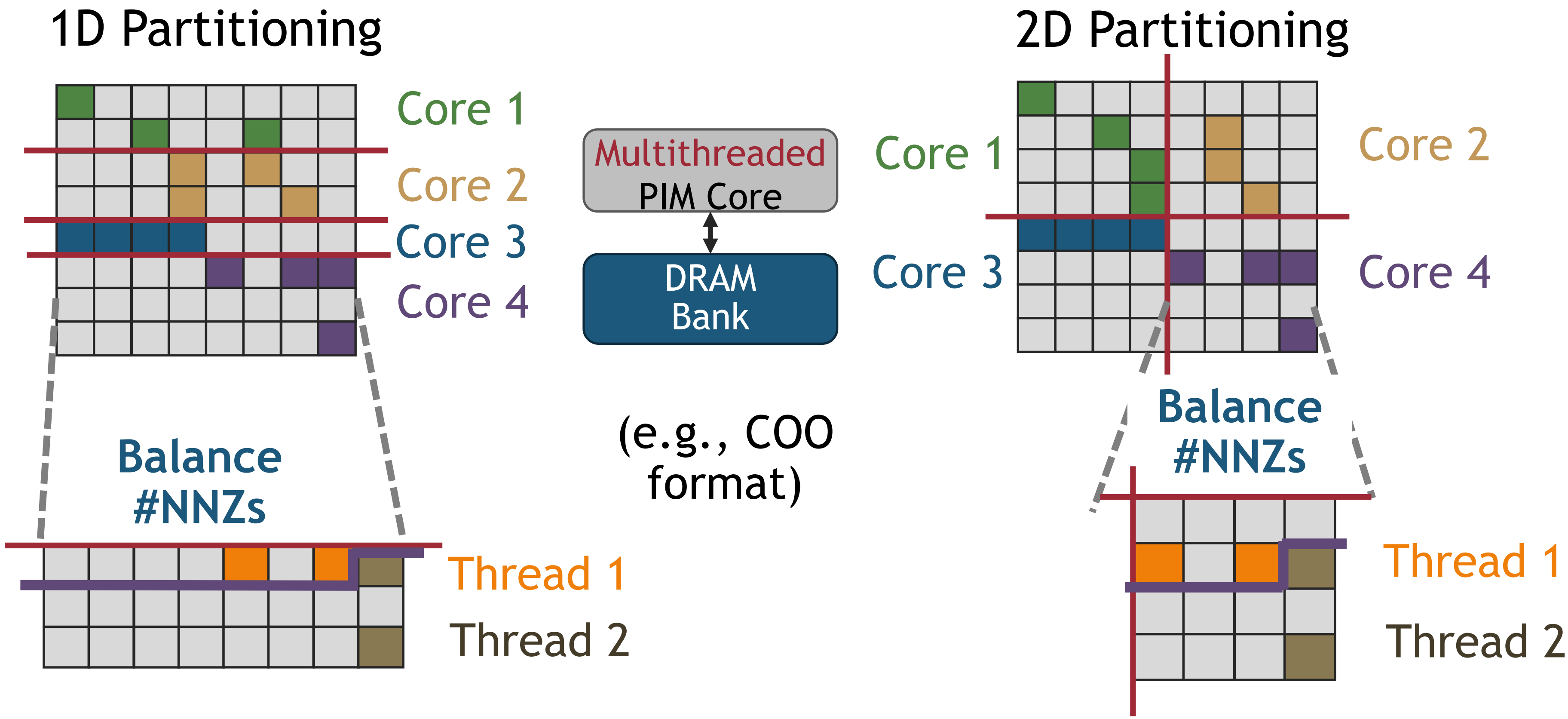
Multithreaded PIM Cores:



- Various load-balance schemes across threads

Load-Balance across Threads

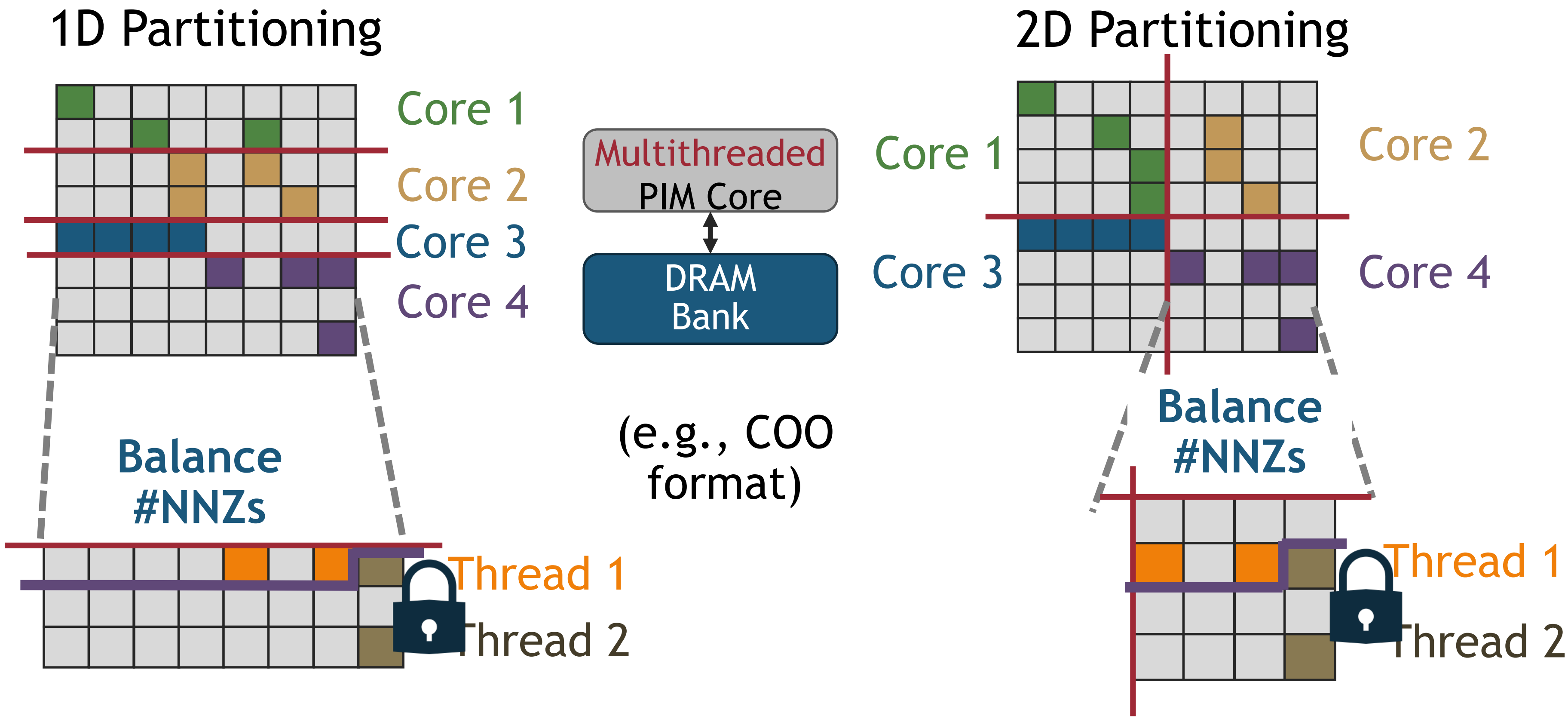
Multithreaded PIM Cores:



- Various load-balance schemes across threads

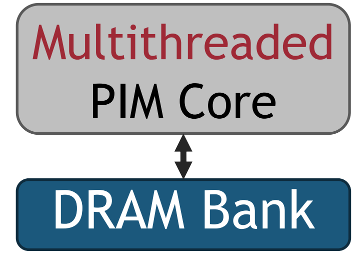
Load-Balance across Threads

Multithreaded PIM Cores:



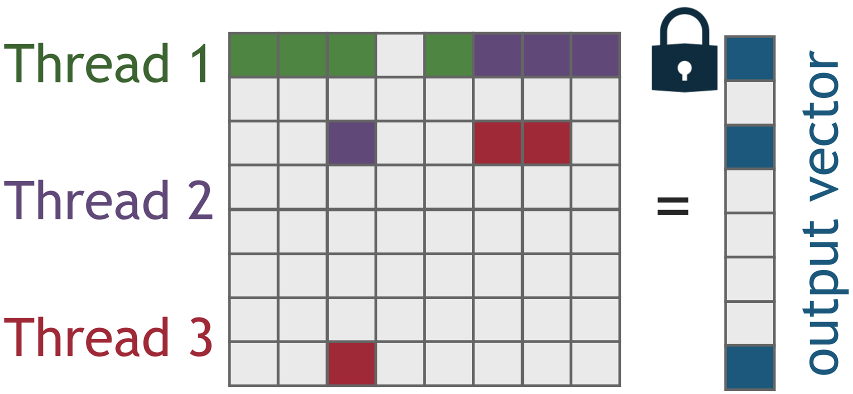
- Various load-balance schemes across threads
- Various synchronization approaches among threads

Synchronization Approaches

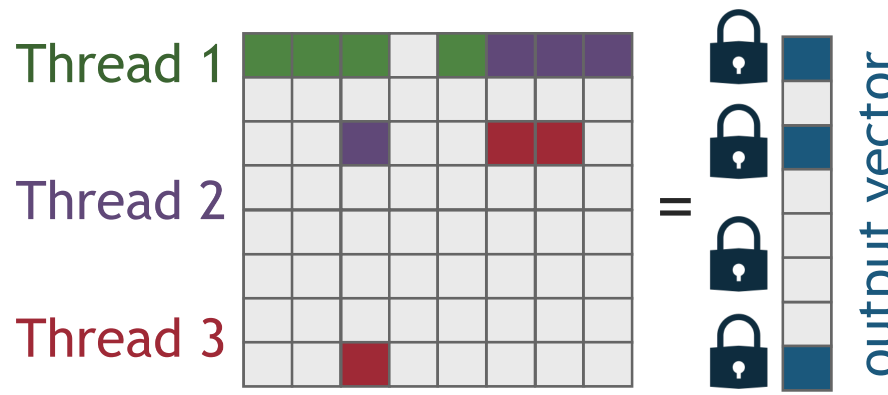


Multithreaded PIM Core:

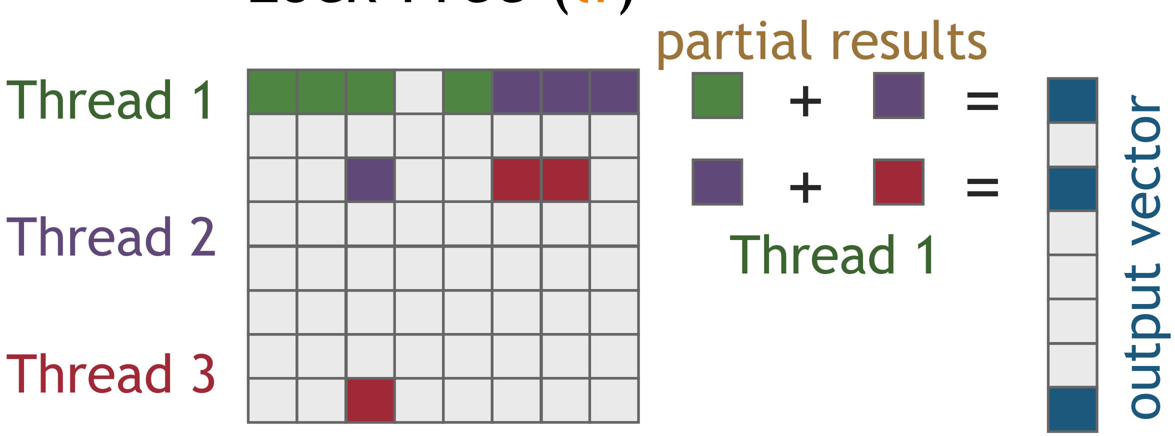
Coarse-Grained (lb-cg)



Fine-Grained (lb-fg)



Lock-Free (lf)



SparseP Software Package

25 SpMV kernels for PIM Systems → https://github.com/CMU-SAFARI/SparseP

Partitioning	Matrix Format	Load-Balancing
9x 1D Kernels	CSR	rows, nnzs *
	COO ▲	rows, nnzs *, nnzs
	BCSR	blocks ^, nnzs ^
	BCOO ▲	blocks, nnzs
4x 2D Equally-Sized Tiles	CSR	--
	COO ▲	--
	BCSR	--
	BCOO ▲	--
6x 2D Equally-Wide Tiles	CSR	nnzs *
	COO ▲	nnzs
	BCSR	blocks ^, nnzs ^
	BCOO ▲	blocks, nnzs
6x 2D Variable-Sized Tiles	CSR	nnzs *
	COO ▲	nnzs
	BCSR	blocks ^, nnzs ^
	BCOO ▲	blocks, nnz

Load-balance
across PIM cores/threads:
* row-granularity (CSR)
^ block-row-granularity (BCSR)

Synchronization
among threads of a PIM core:
▲ lb-cg, lb-fb, lf (COO, BCOO)

Data Types:
- 8-bit integer
- 16-bit integer
- 32-bit integer
- 64-bit integer
- 32-bit float
- 64-bit float

Outline

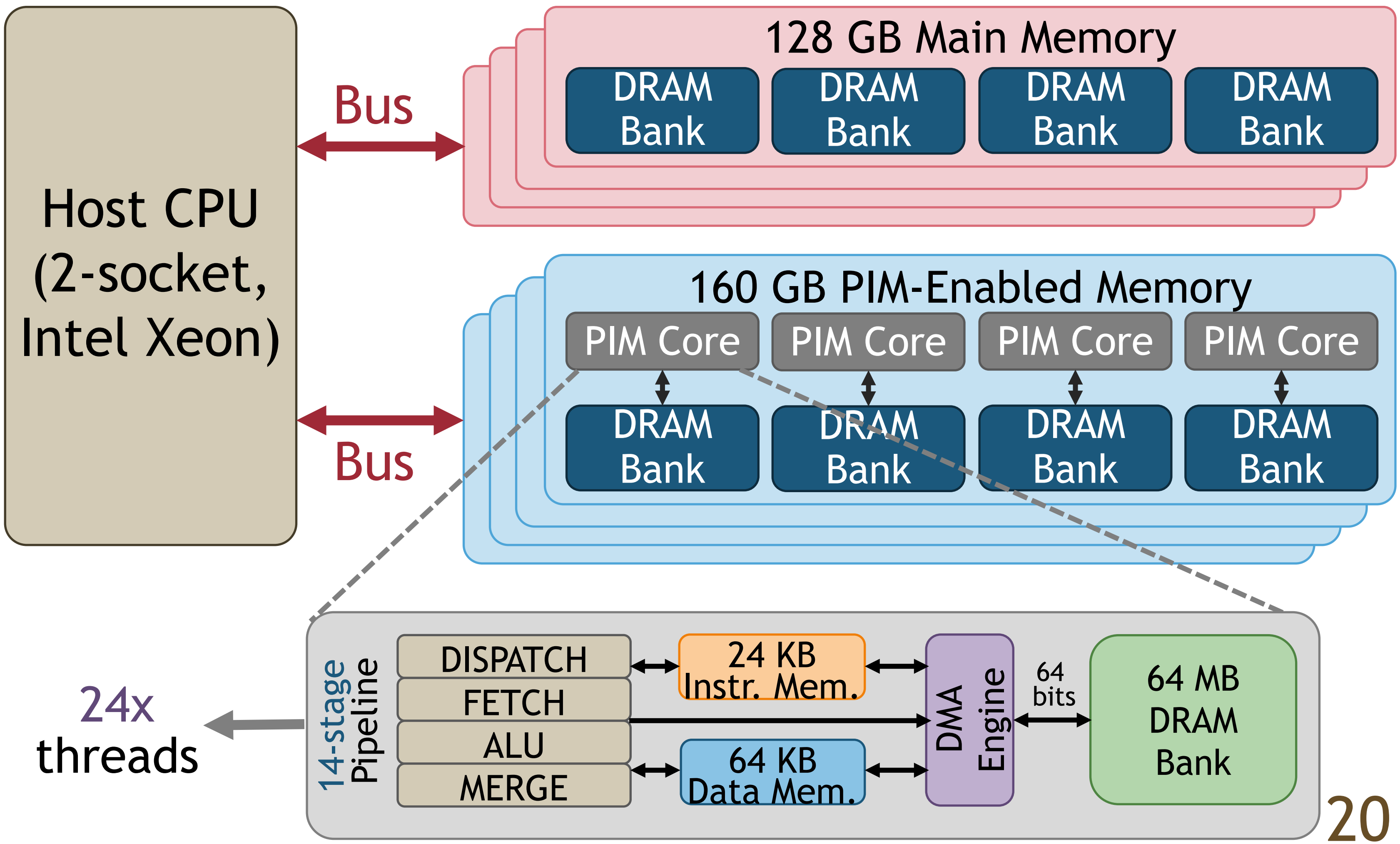
SpMV Kernels for Real PIM Systems

Key Takeaways from Our Study

Conclusion

UPMEM-based PIM System

- 20 UPMEM PIM DIMMs with 2560 PIM cores in total
- Each multithreaded PIM core supports 24 threads

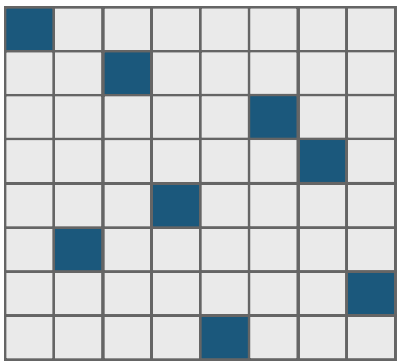


Sparse Matrix Data Set

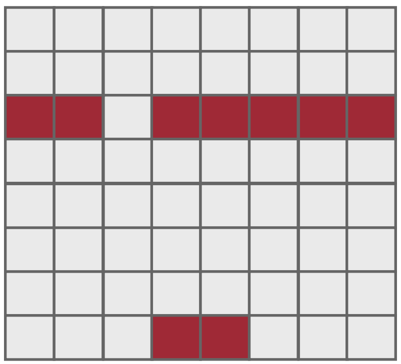
26 sparse matrices*:

- Diverse sparsity patterns
- Variability on irregular patterns
- Variability on block patterns

Regular Matrix



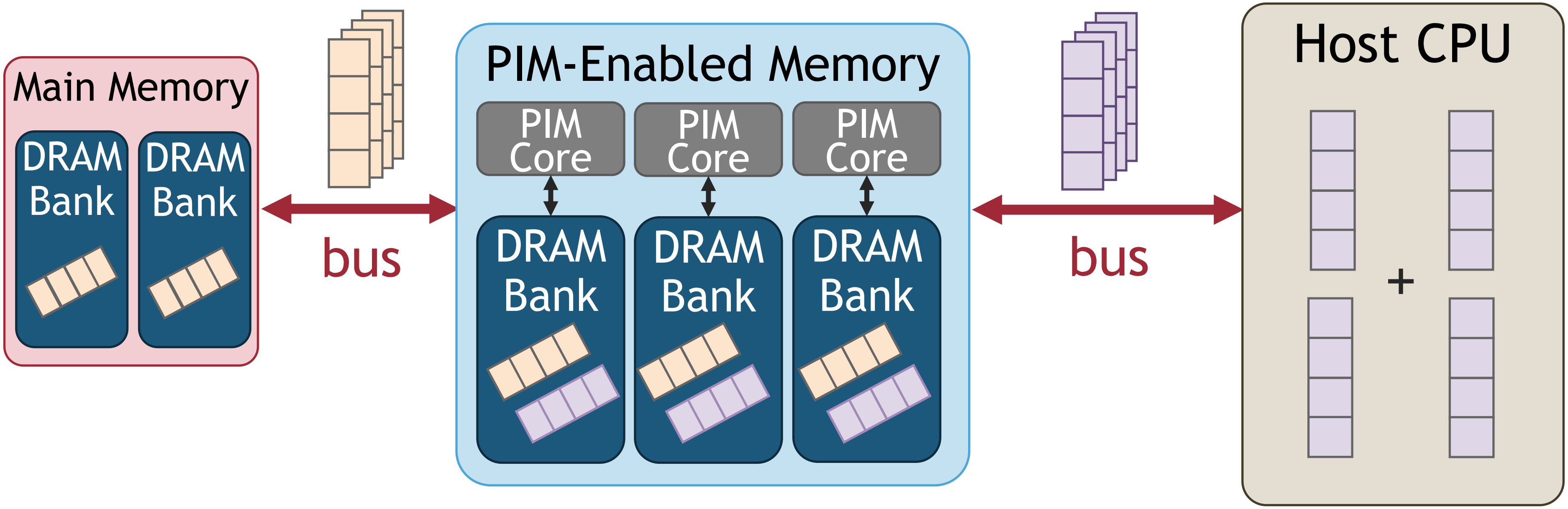
Scale-Free Matrix



* Suite Sparse Matrix Collection: https://sparse.tamu.edu/

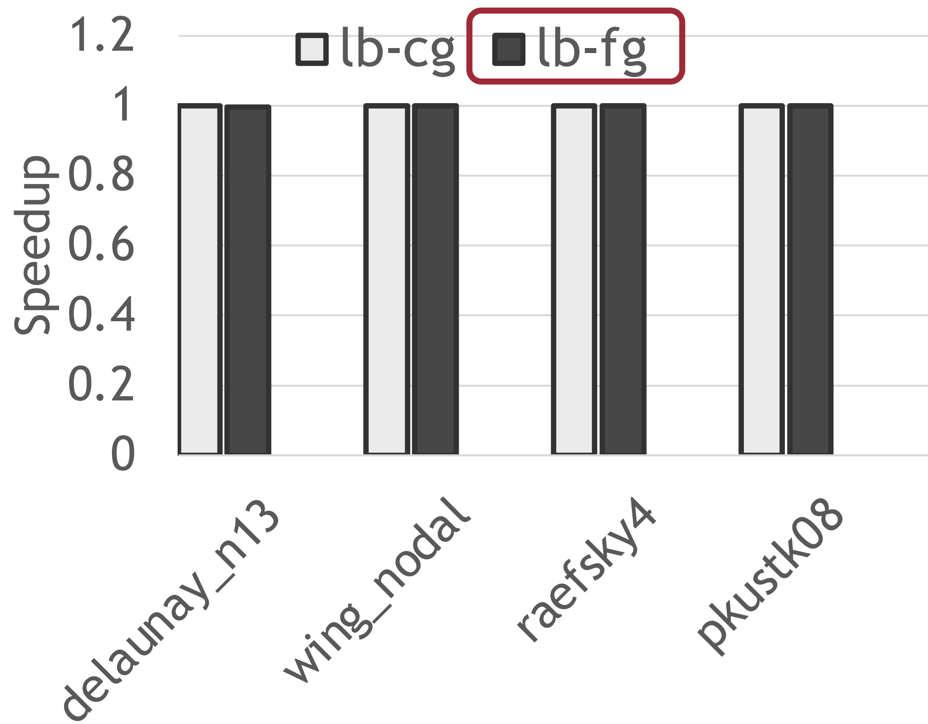
Kernel Execution on One PIM Core

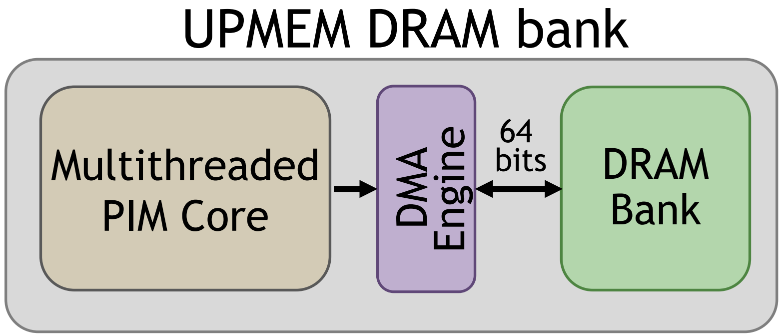
1. Load the input vector
2. Execute the kernel
3. Retrieve the partial results
4. Merge the partial results



Lock-Based Synchronization

16 threads, COO, 32-bit integer



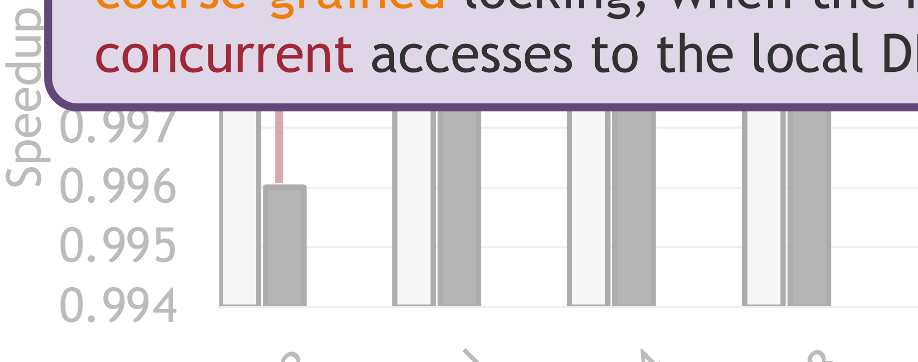


Fine-grained locking (lb-fg) does not improve performance over coarse-grained locking (lb-cg)

Fine-Grained Locking: memory accesses to the local DRAM bank are serialized in the DMA engine of the UPEM PIM hardware.

Lock-Based Synchronization

16 threads, COO, 32-bit integer



Key Takeaway 1

Fine-grained locking approaches cannot improve performance over coarse-grained locking, when the PIM hardware does not support concurrent accesses to the local DRAM bank.

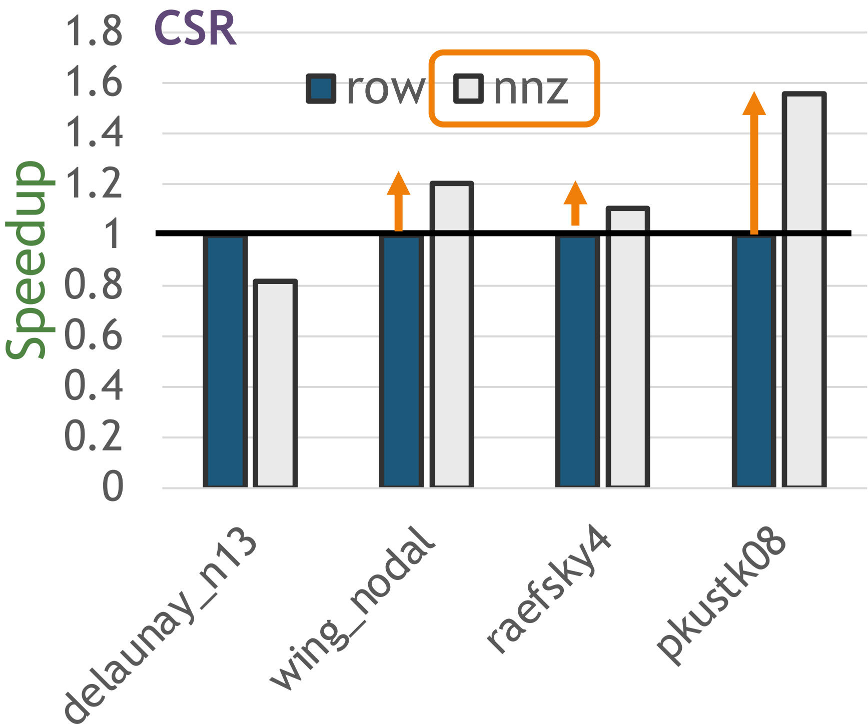
Recommendation 1

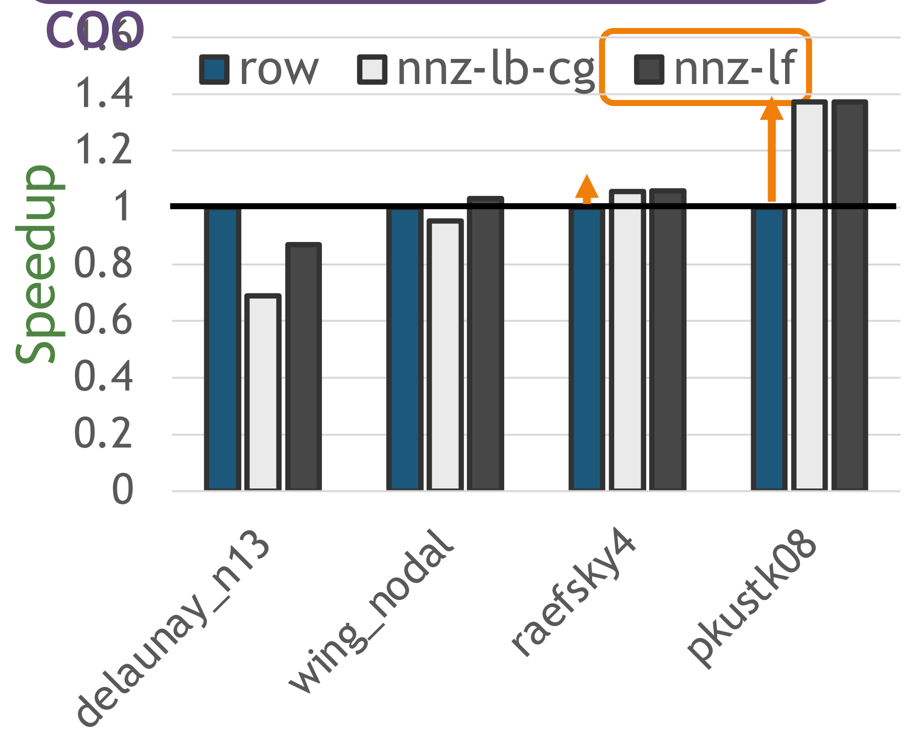
Provide low-cost synchronization support and hardware support to enable concurrent memory accesses to the local DRAM bank, and integrate multiple DRAM banks per PIM core to increase execution parallelism.

Load-Balance within a PIM Core

16 threads, 32-bit integer

Load-balancing #NNZs performs best in most matrices



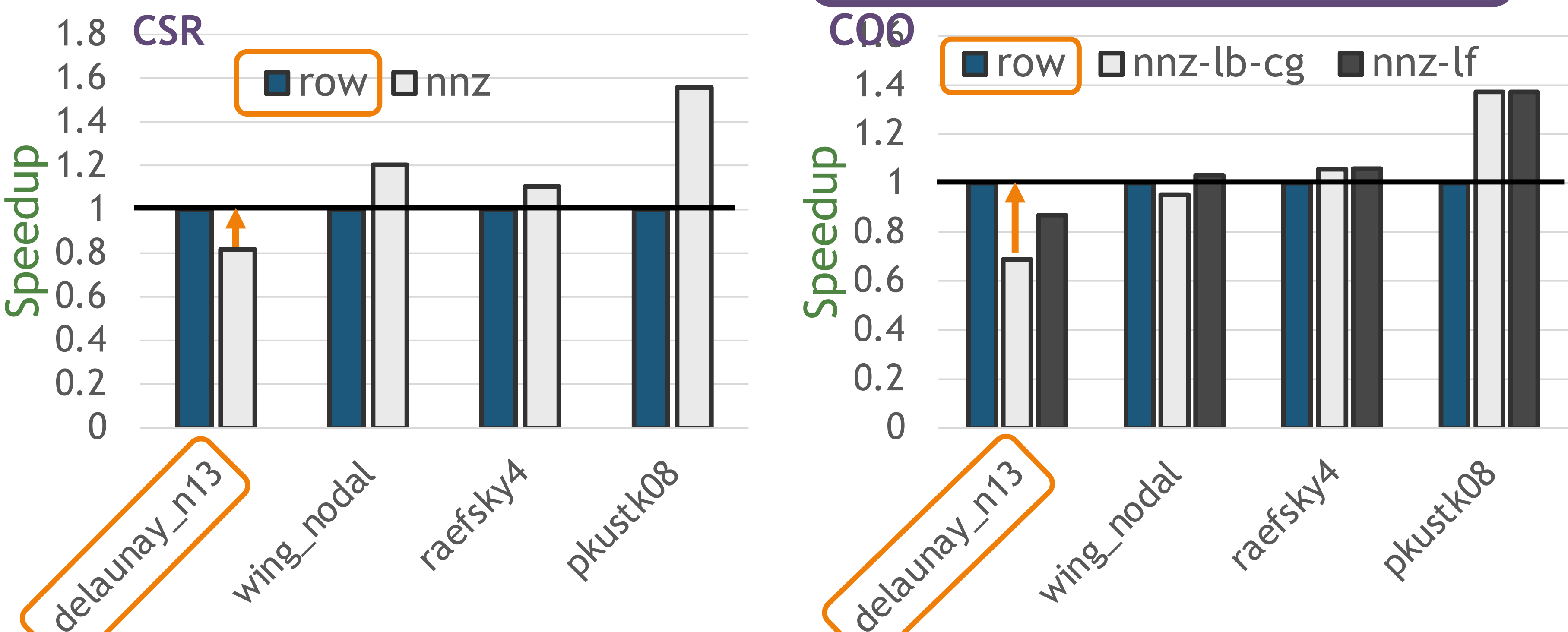


Load-balancing #NNZ typically provides high computation balance across threads of a compute-limited PIM core

Load-Balance within a PIM Core

16 threads, 32-bit integer

Load-balancing #NNZs causes high row imbalance



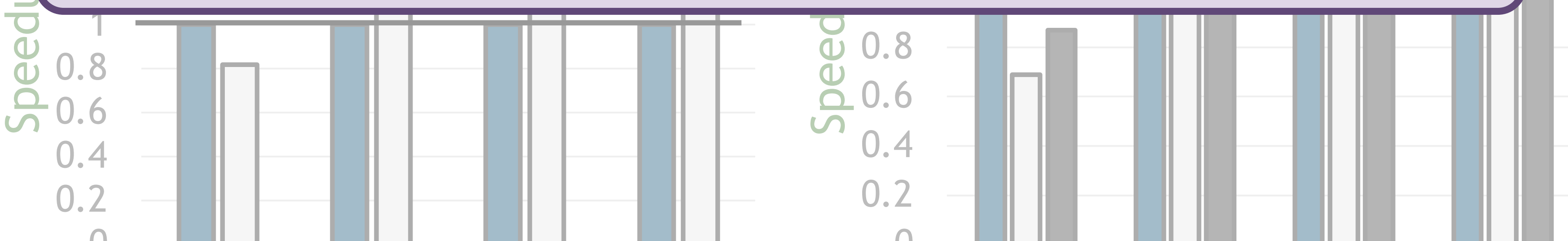
Load-balancing #NNZs: one single thread performs a much higher #memory accesses and #synchronization operations than the rest

Load-Balance within a PIM Core

16 threads, 32-bit integer

Key Takeaway 2

High operation imbalance in computation, synchronization, or memory instructions executed by multiple threads of a PIM core can cause high performance overhead.

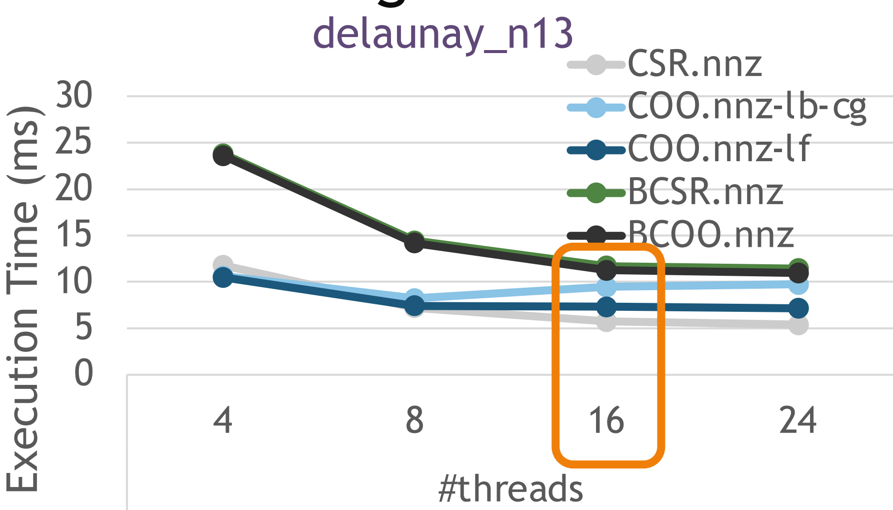


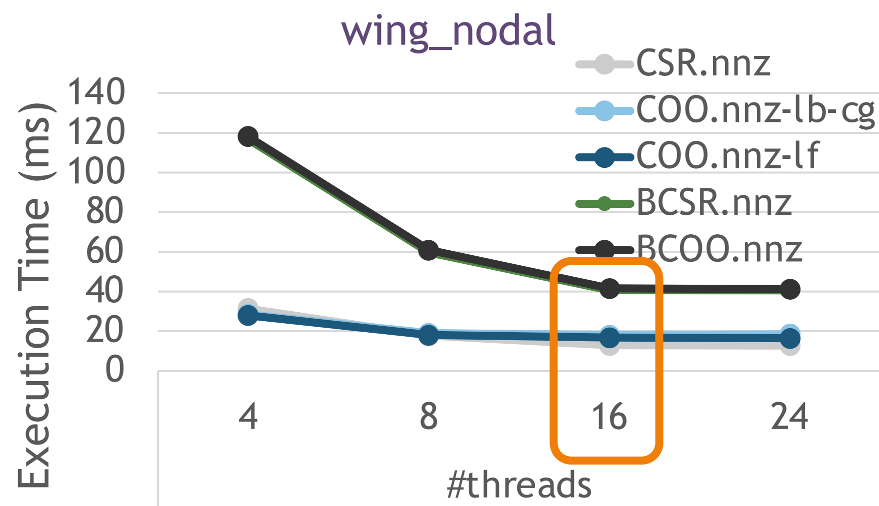
Recommendation 2

Design algorithms that provide high load balance across threads of PIM core in terms of computations, synchronization points and memory accesses.

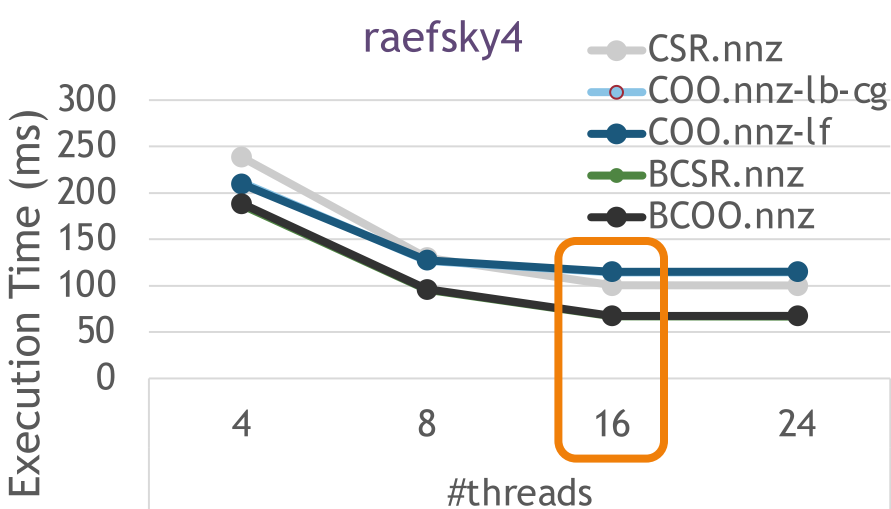
Scalability within a PIM Core

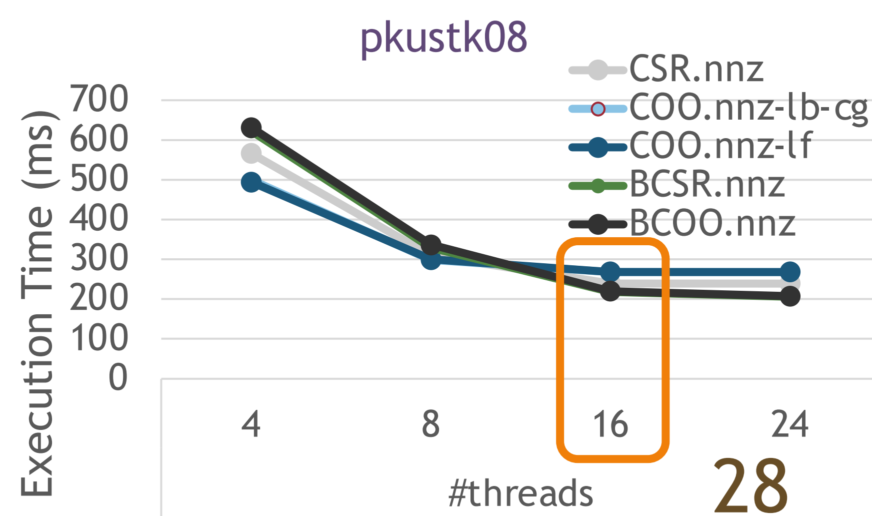
32-bit integer



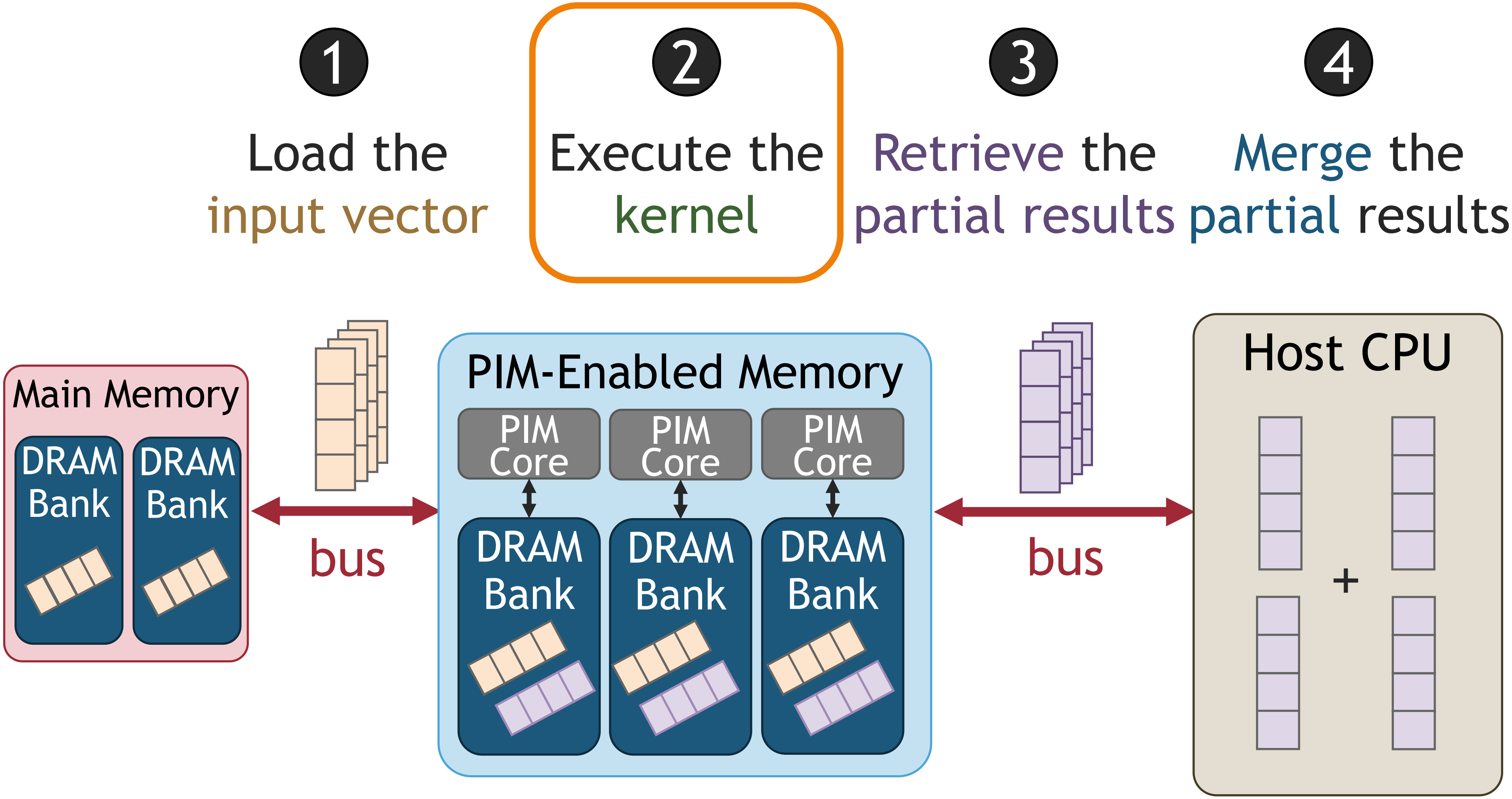


Scalability increases up to 16 threads



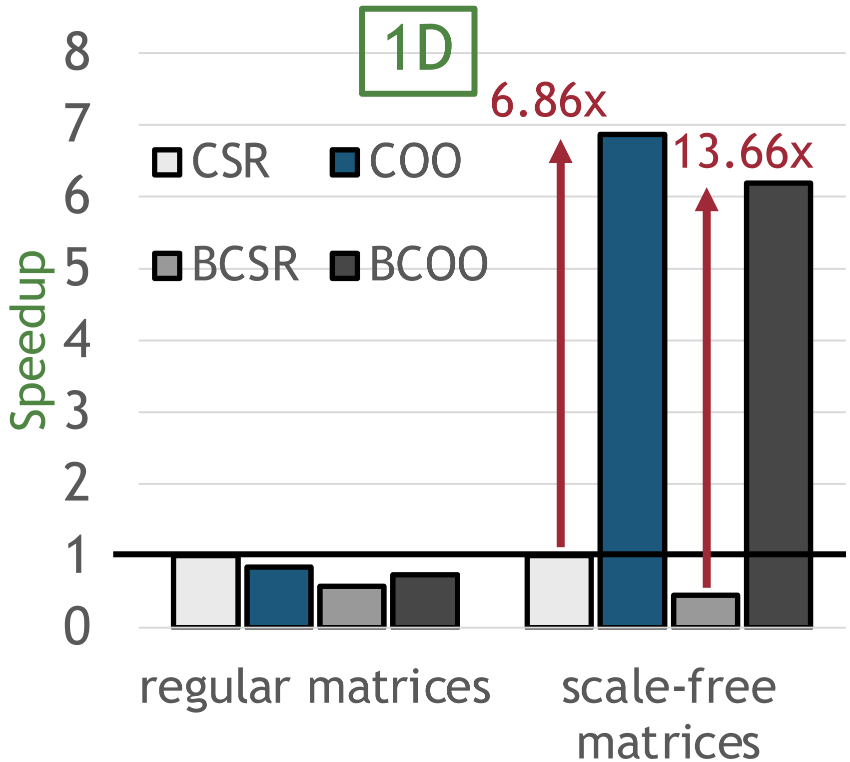


Kernel Execution on Multiple PIM Cores

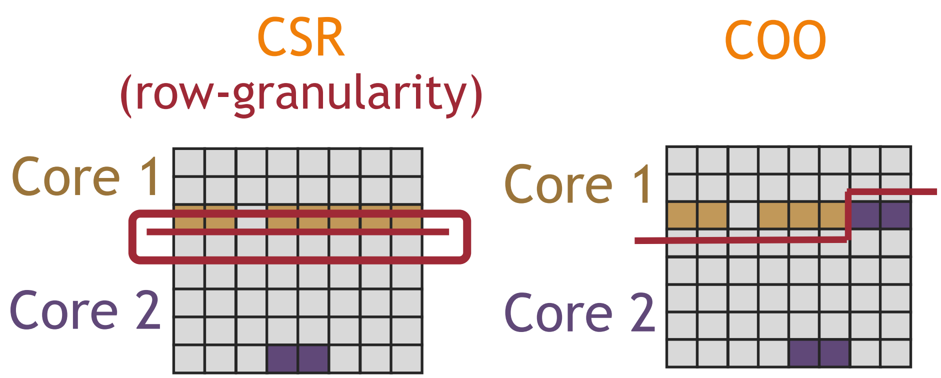


Comparison of Compressed Formats

2048 PIM Cores, 32-bit integer



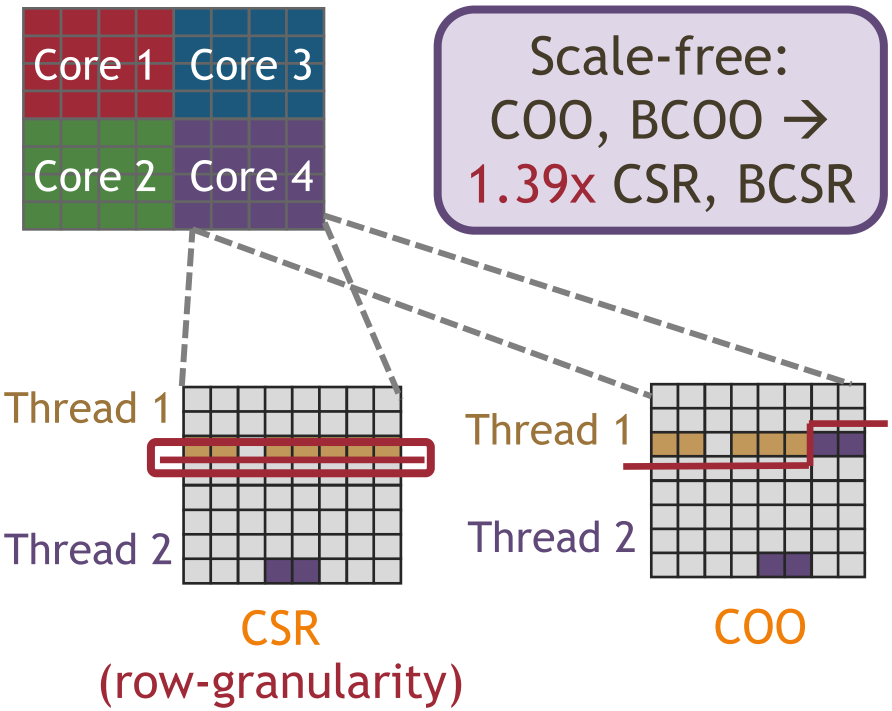
Scale-free: COO, BCOO → 10.26x CSR, BCSR

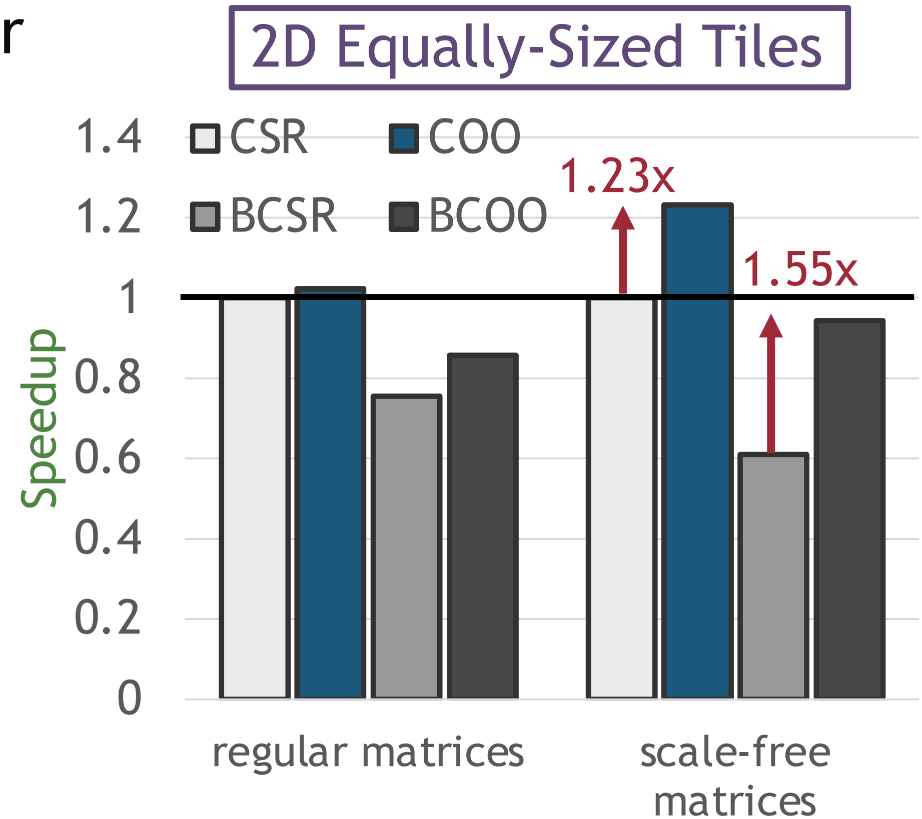


In scale-free matrices, COO + BCOO provide higher non-zero element balance across PIM cores than CSR + BCSR, respectively.

Comparison of Compressed Formats

2048 PIM Cores, 32-bit integer



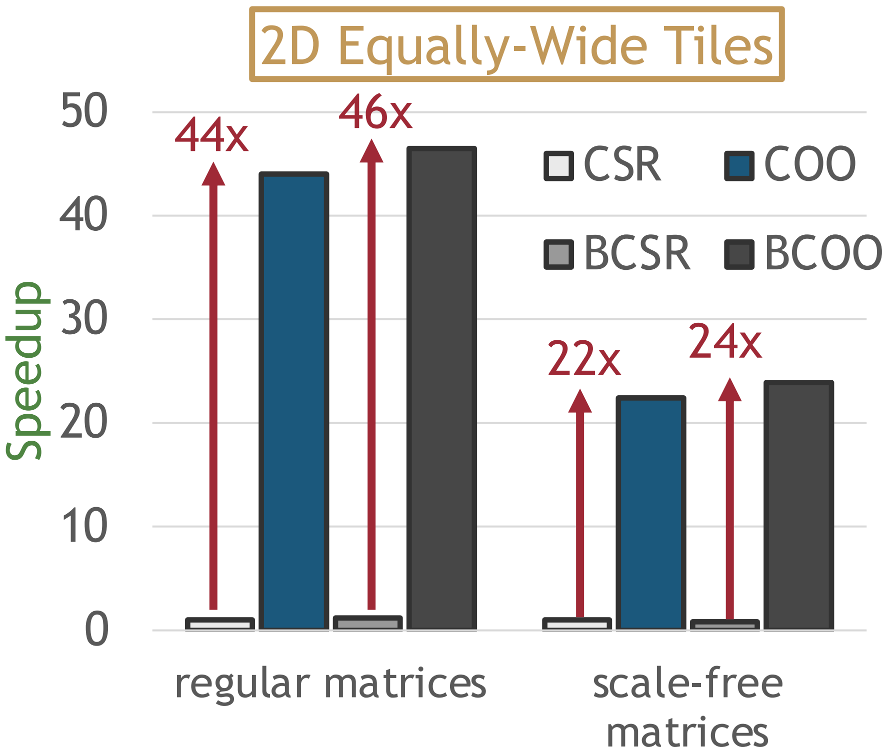


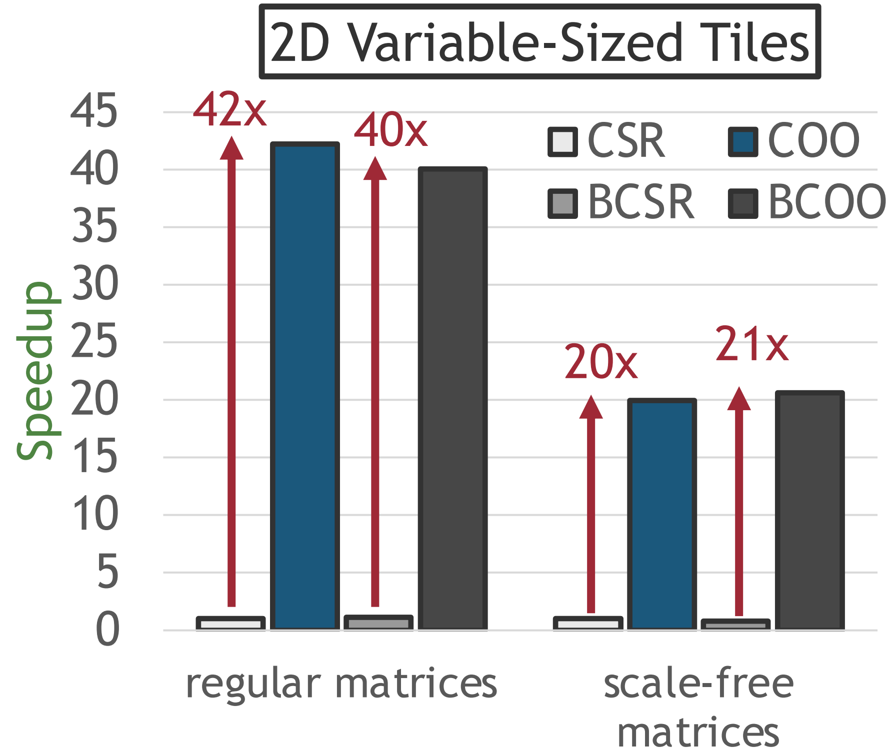
In scale-free matrices, COO + BCOO provide higher non-zero element balance across threads than CSR + BCSR, respectively.

Comparison of Compressed Formats

2048 PIM Cores, 32-bit integer

COO, BCOO → 32.38x CSR, BCSR





COO + BCOO formats provide higher non-zero element balance across PIM cores + threads than CSR + BCSR, respectively.

Comparison of Compressed Formats

2048 PIM Cores, 32-bit integer

1D

2D Equally-Sized

Key Takeaway 3

The compressed matrix format used to store the input matrix determines the data partitioning across DRAM banks of PIM-enabled memory. As a result, it affects the load-balance across PIM cores (and threads of a PIM core) with corresponding performance implications.

2D Equally-Wide

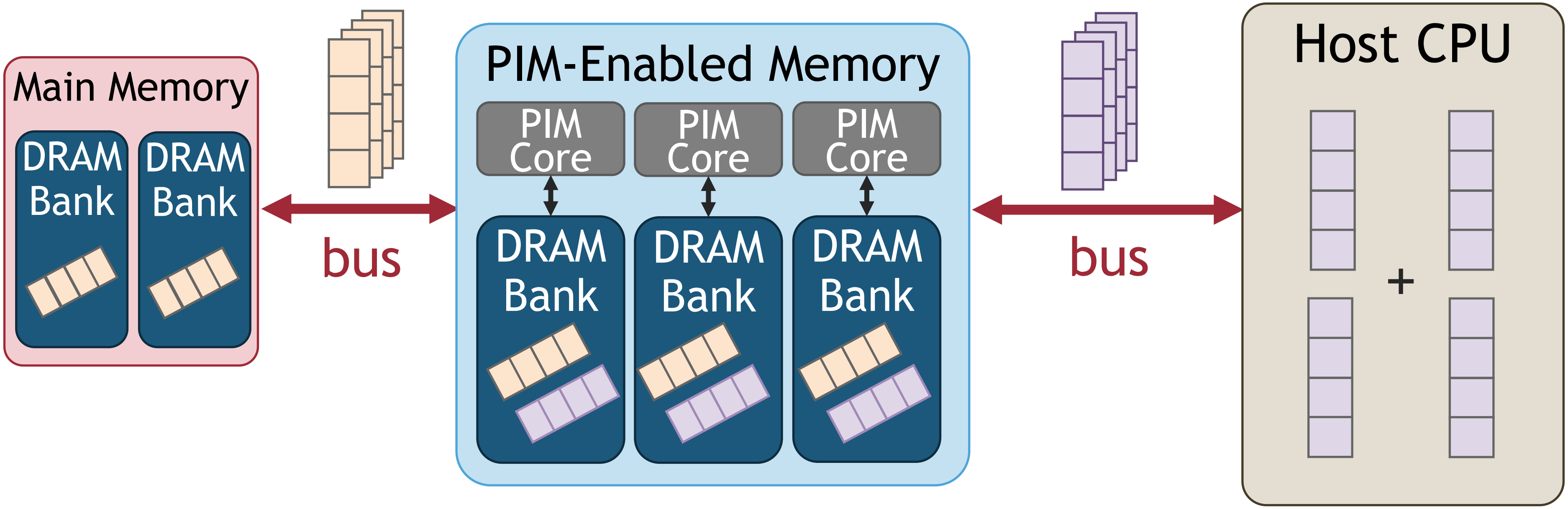
2D Variable-Sized

Recommendation 3

Design compressed data structures that can be effectively partitioned across DRAM banks, with the goal of providing high computation balance across PIM cores (and threads of a PIM core).

End-to-End Performance

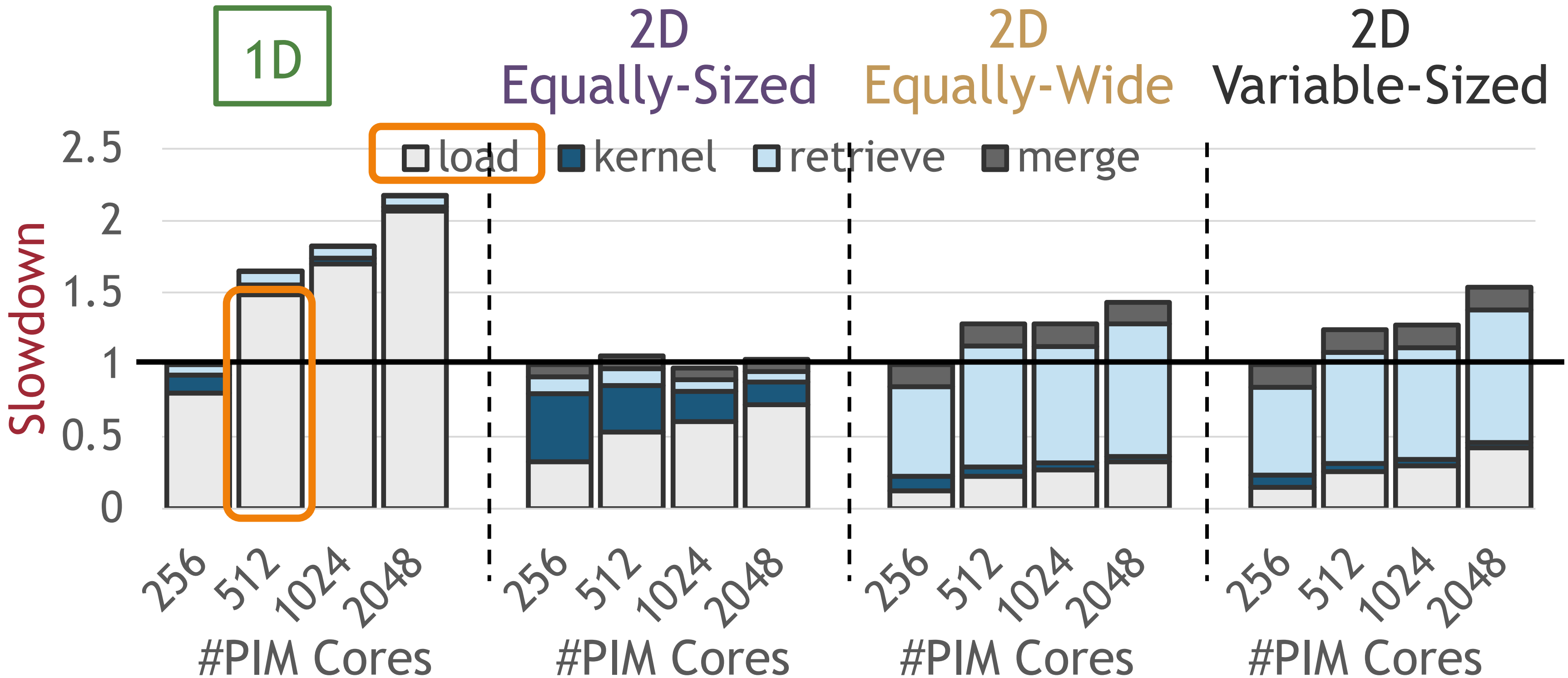
1. Load the input vector
2. Execute the kernel
3. Retrieve the partial results
4. Merge the partial results



Scalability

COO format, 32-bit integer

The scalability is limited by the load time



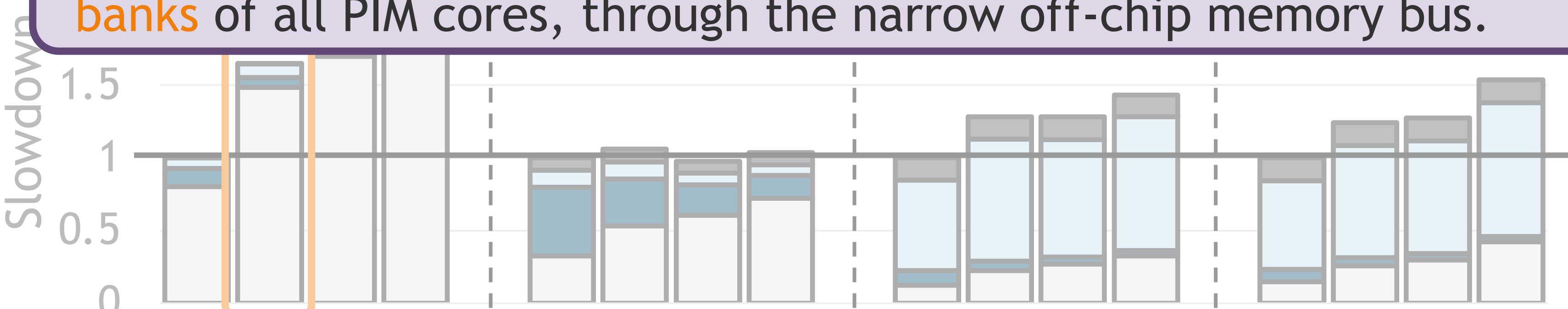
1D: #bytes to load the input vector grows linearly to #PIM cores

Scalability

COO format, 32-bit integer

Key Takeaway 4

The 1D-partitioned kernels are severely bottlenecked by the high data transfer costs to broadcast the whole input vector into DRAM banks of all PIM cores, through the narrow off-chip memory bus.



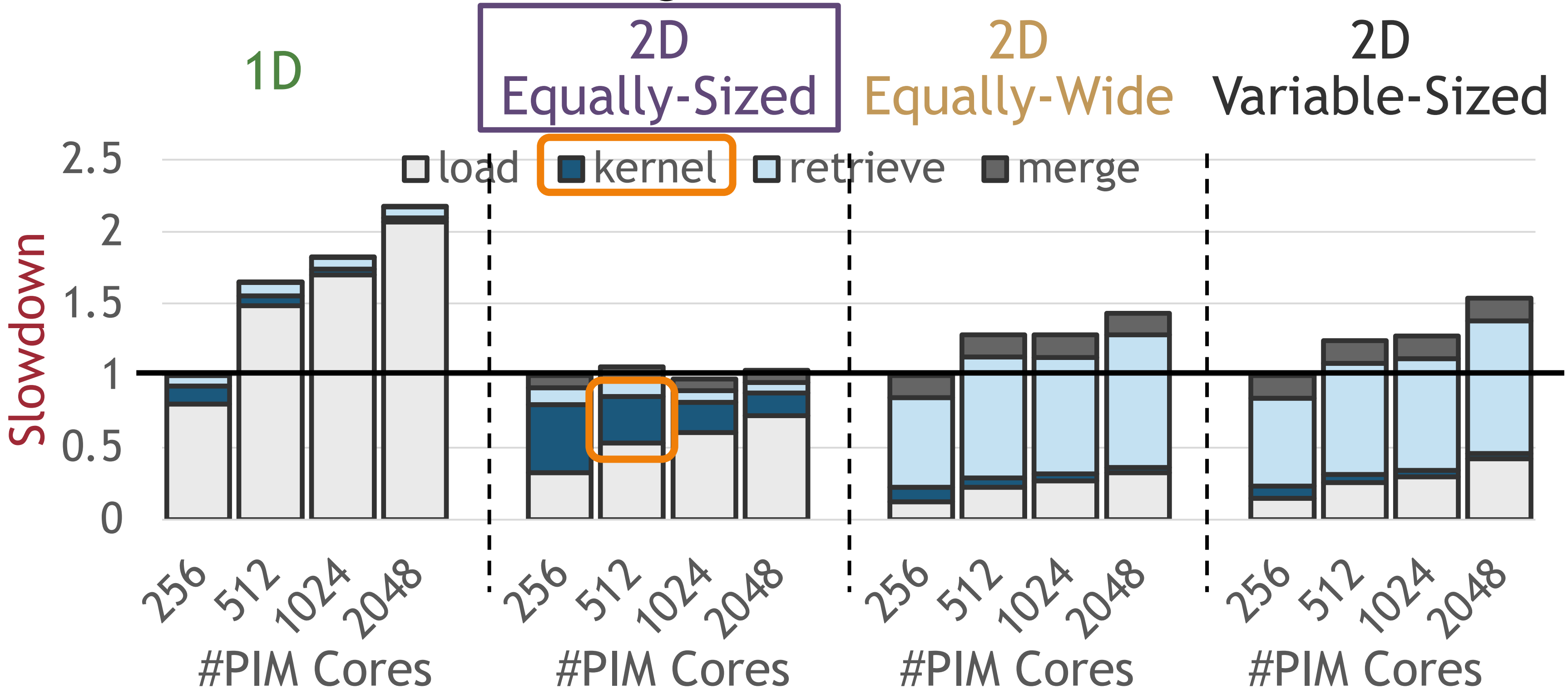
Recommendation 4

Optimize the broadcast collective collective in data transfers to PIM-enabled memory to efficiently copy the input data into DRAM banks in the PIM system.

Scalability

COO format, 32-bit integer

The scalability is limited by the kernel time

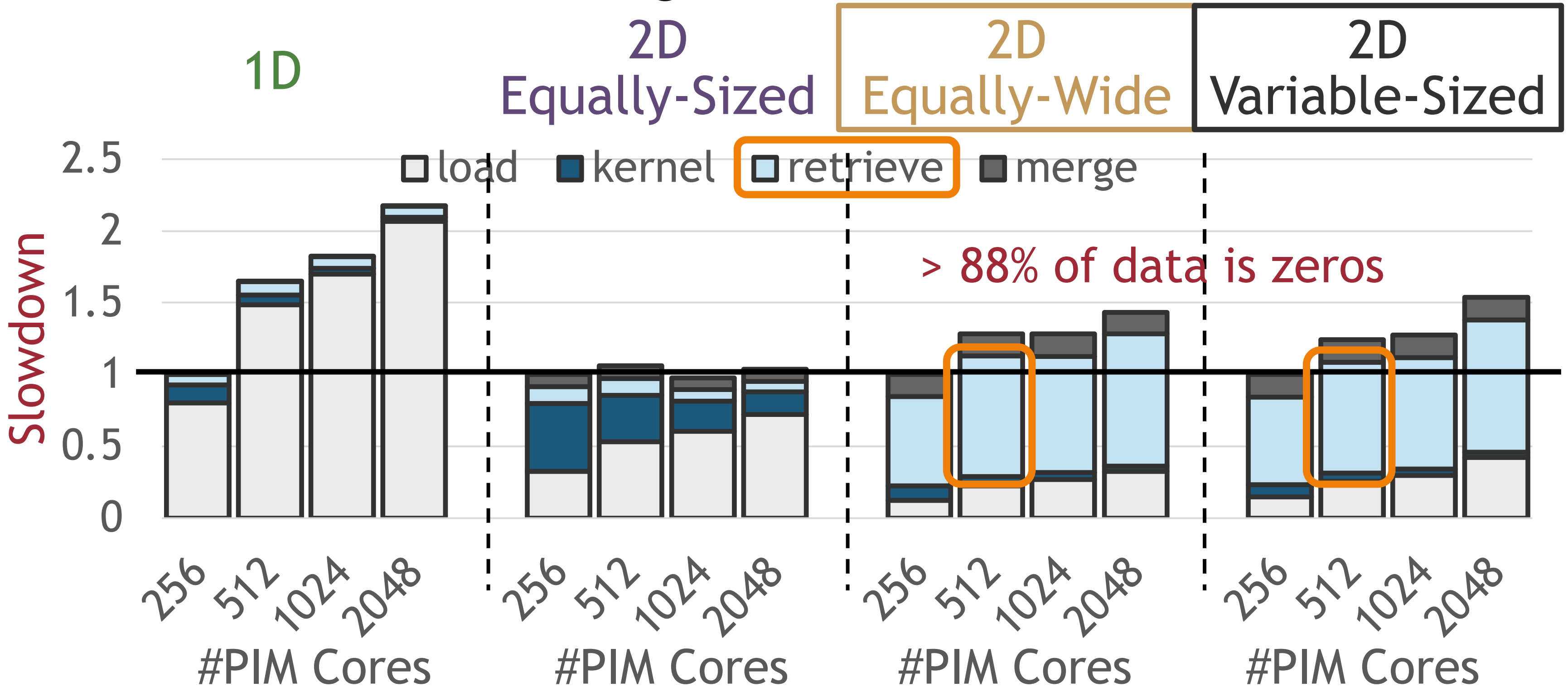


2D Equally-Sized: kernel time is limited by only a few PIM cores assigned to the 2D tiles with the largest #NNZs

Scalability

COO format, 32-bit integer

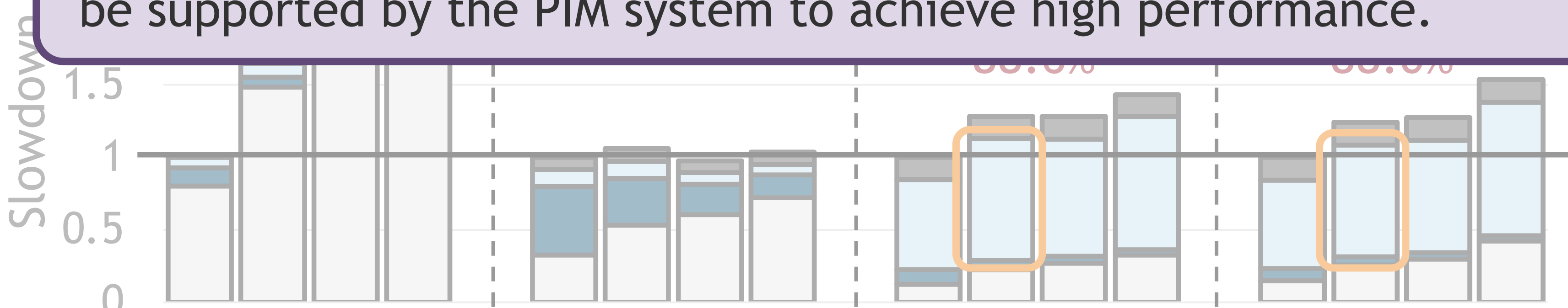
The scalability is limited by the retrieve time



2D Equally-Wide + 2D Variable-Sized:
high amount of zero padding to gather the output vector → parallel transfers supported at rank granularity = 64 PIM cores

Scalability

COO format, 32-bit integer



Key Takeaway 5

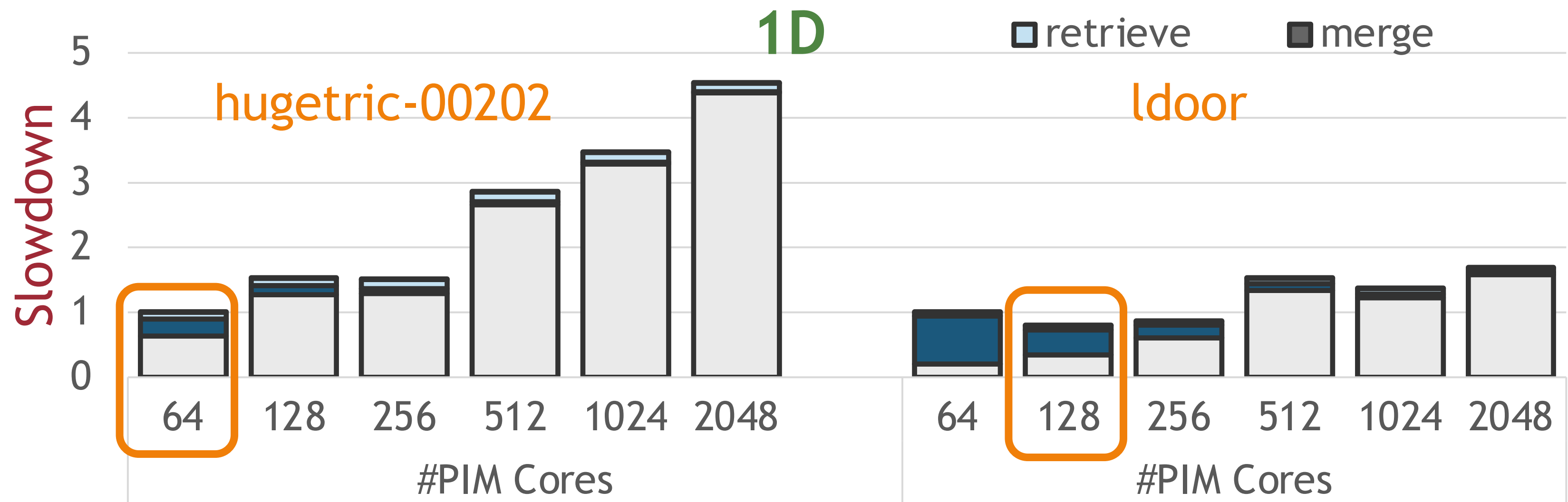
The 2D equally-wide and variable-sized kernels need fine-grained parallel data transfers at DRAM bank granularity (zero padding) to be supported by the PIM system to achieve high performance.

Recommendation 5

Optimize the gather collective operation at DRAM bank granularity in data transfers from PIM-enabled memory to efficiently retrieve the output results to the host CPU.

Comparison of Sparse Matrices

COO format, 32-bit integer



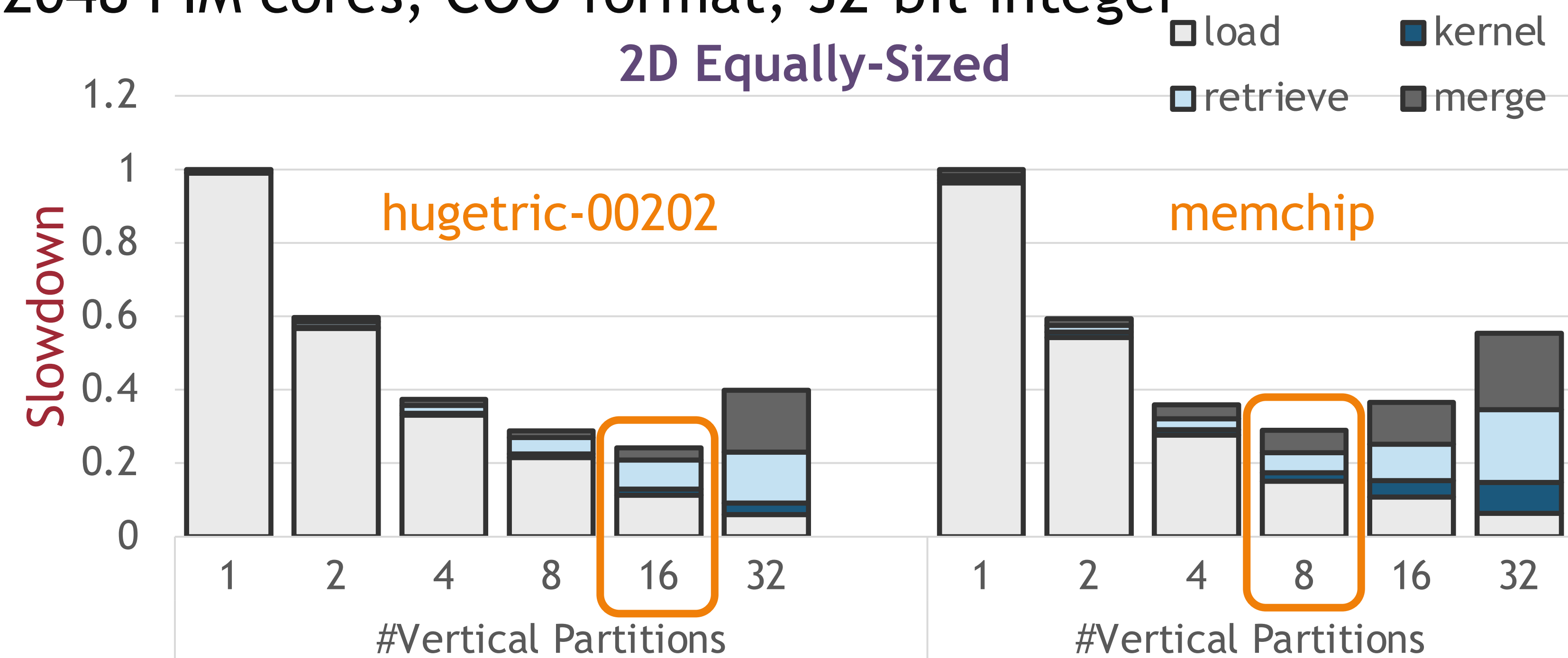
Best-performing = 64 PIM cores

Best-performing = 128 PIM cores

1D: #PIM cores that provides the best performance depends on the sparsity pattern of the input matrix

Comparison of Sparse Matrices

2048 PIM cores, COO format, 32-bit integer



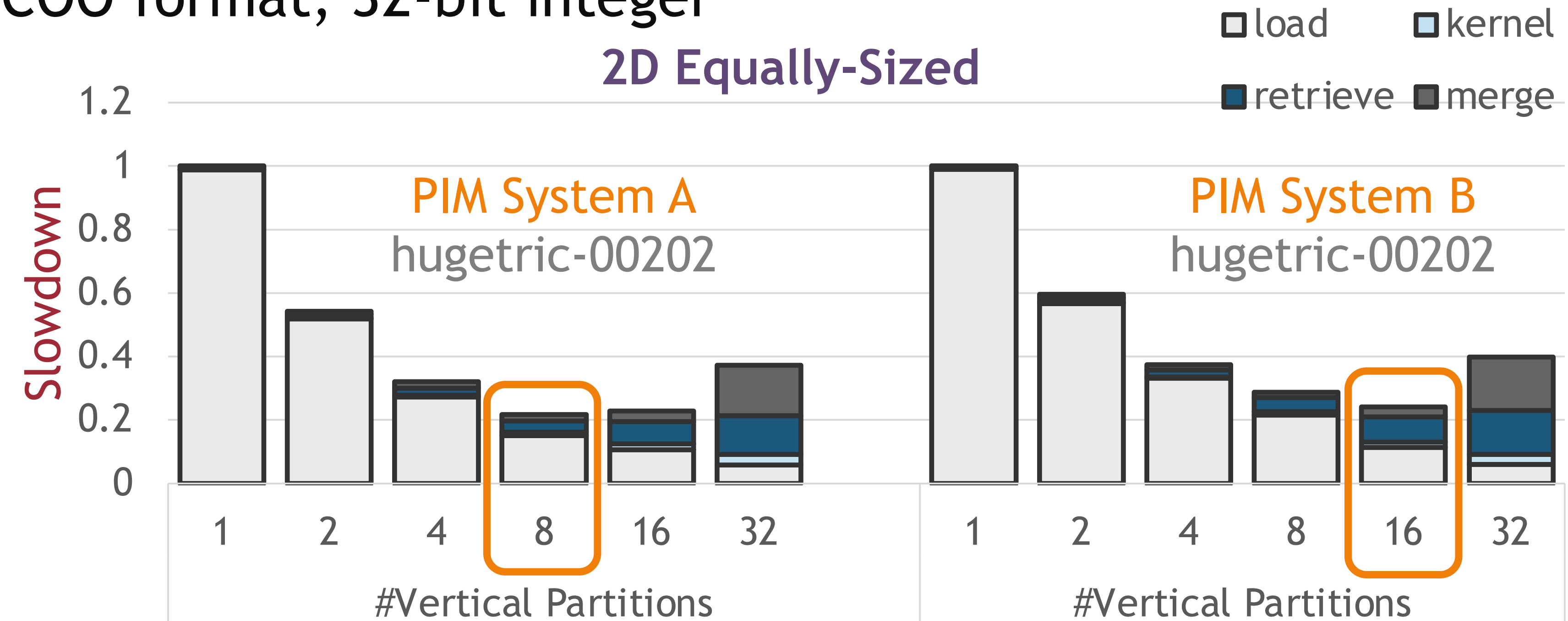
Best-performing = 16 vertical part.

Best-performing = 8 vertical part.

2D: #vertical partitions that provides the best performance depends on the sparsity pattern of the input matrix

Comparison of PIM Systems

COO format, 32-bit integer



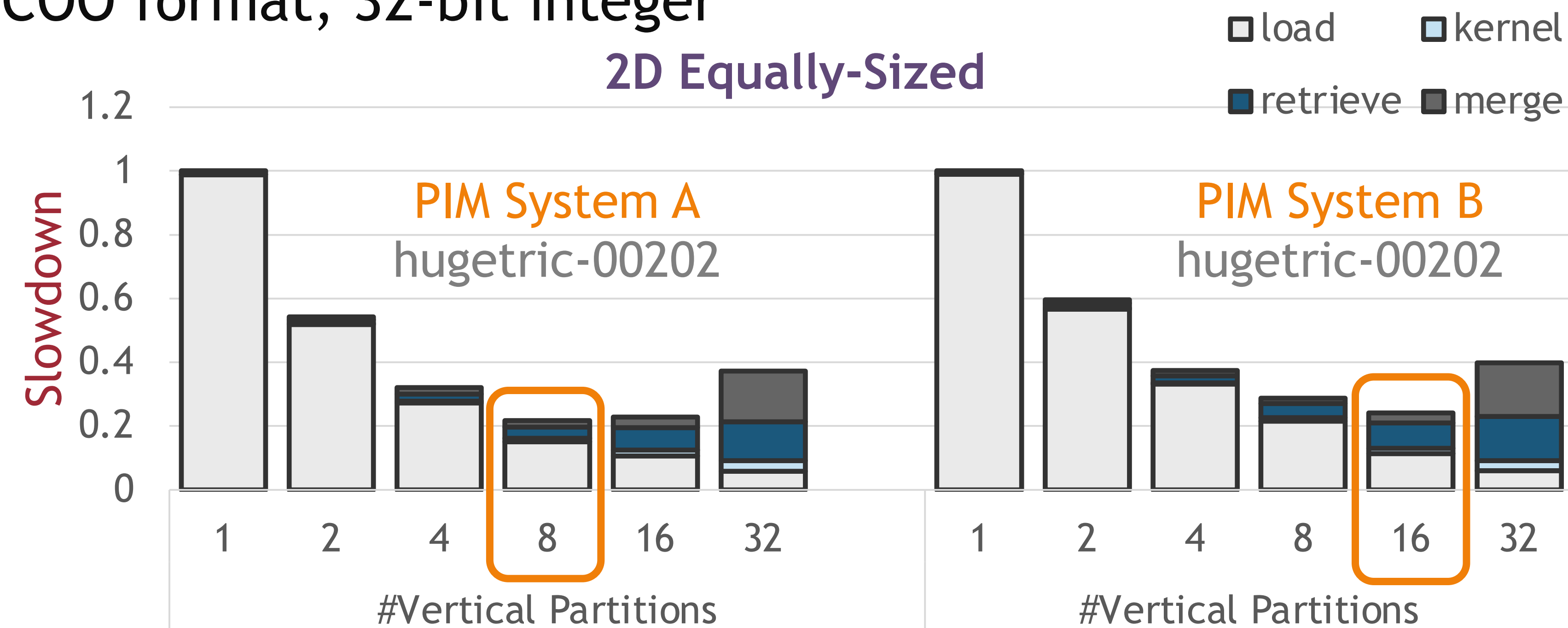
Best-performing = 8 vertical part.

Best-performing = 16 vertical part.

System	PIM Cores	PIM Band.	Host CPU	Bus Band.
PIM A	2048 @350 MHz	1.43 TB/s	Intel Xeon Silver 4110 @2.1 GHz	23.1 GB/s
PIM B	2048 @425 MHz	1.78 TB/s	Intel Xeon Silver 4215 @2.5 GHz	21.8 GB/s

Comparison of PIM Systems

COO format, 32-bit integer



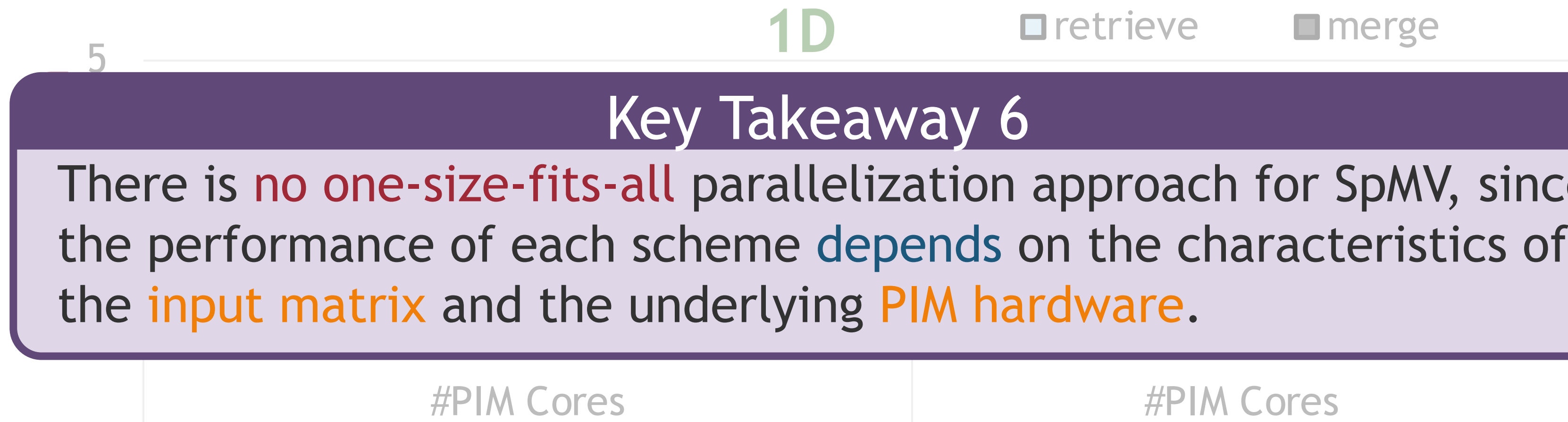
Best-performing = 8 vertical part.

Best-performing = 16 vertical part.

2D: #vertical partitions that provides the best performance depends on the underlying hardware characteristics

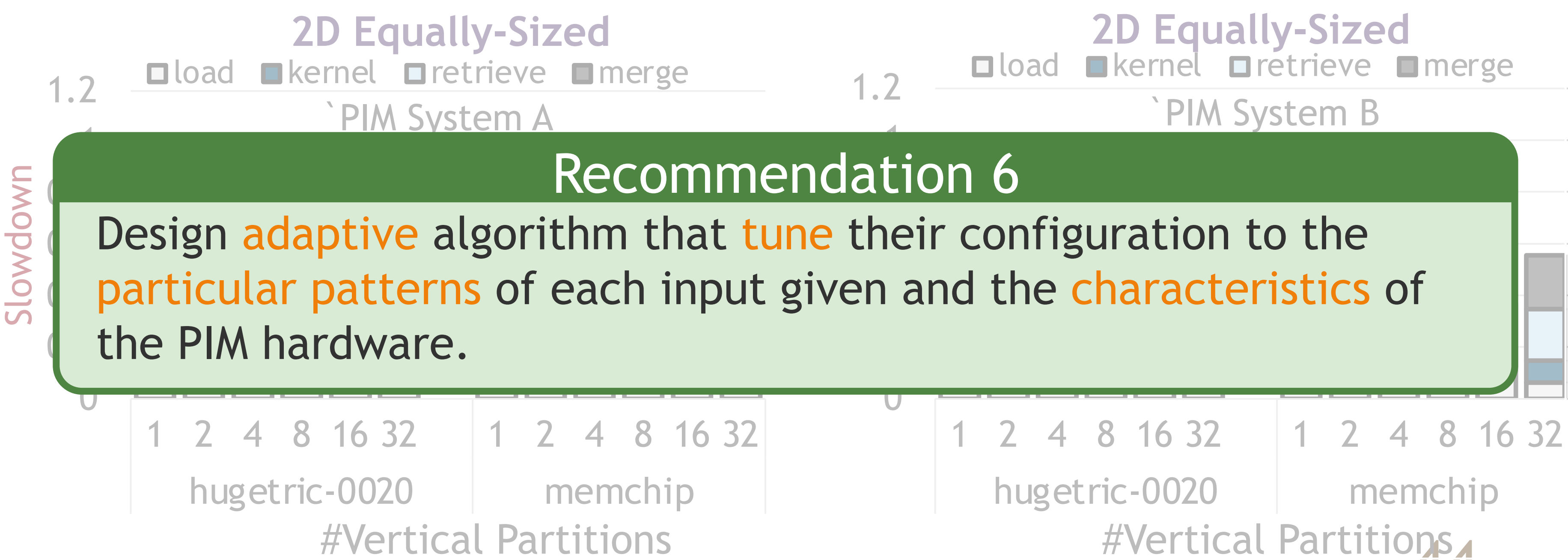
Various Matrices and PIM Systems

COO format, 32-bit integer



Key Takeaway 6

There is no one-size-fits-all parallelization approach for SpMV, since the performance of each scheme depends on the characteristics of the input matrix and the underlying PIM hardware.

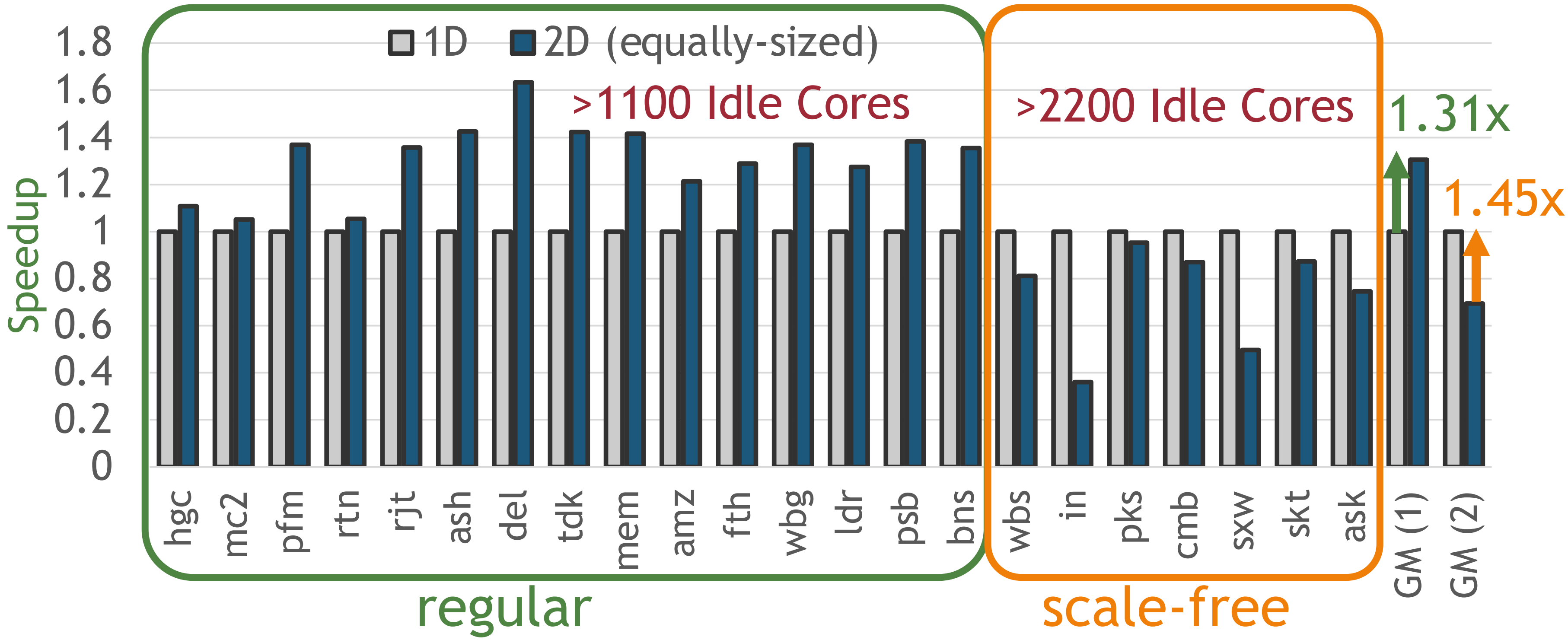


Recommendation 6

Design adaptive algorithm that tune their configuration to the particular patterns of each input given and the characteristics of the PIM hardware.

1D vs 2D

Up to 2528 PIM Cores, 32-bit float

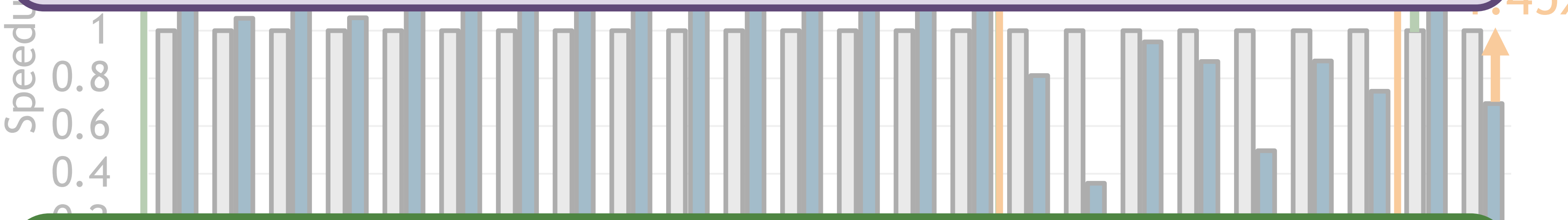


Best-performing SpMV execution:
trades off computation with lower data transfer costs

1D vs 2D

Key Takeaway 7

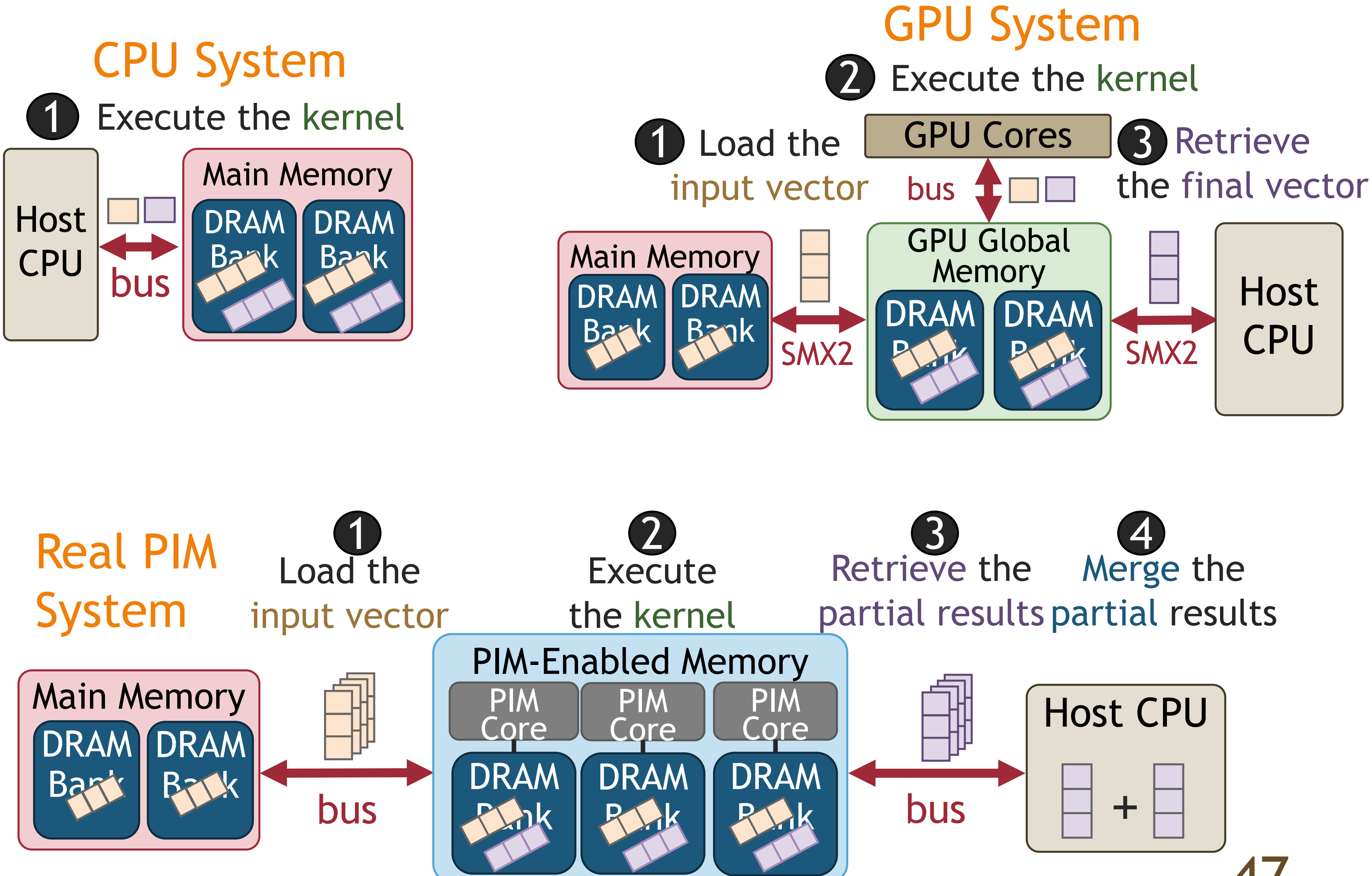
Expensive data transfers to/from PIM-enabled memory performed via the narrow memory bus impose significant performance overhead to end-to-end SpMV execution. Thus, it is hard to fully exploit all available PIM cores of the system.



Recommendation 7

Design high-speed communication channels and optimized libraries in data transfers to/from PIM-enabled memory, provide hardware support to effectively overlap computation with data transfers in the PIM system, and/or integrate PIM-enabled memory as the main memory of the system.

SpMV Execution on Various Systems



CPU/GPU Comparisons

System		Peak Performance	Bandwidth	TDP	
CPU	Intel Xeon Silver 4110	660 GFlops	23.1 GB/s	2x85 W	Processor-Centric
GPU	NVIDIA Tesla V100	14.13 TFlops	897 GB/s	300 W	
PIM	UPMEM 1st Gen.	4.66 GFlops	1.77 TB/s	379 W	Memory-Centric

CPU/GPU Comparisons

- Kernel-Only (COO, 32-bit float):
  - CPU = 0.51% of Peak Perf.
  - GPU = 0.21% of Peak Perf.
  - PIM (1D) = 50.7% of Peak Perf.

System		Peak Performance	Bandwidth	TDP	
CPU	Intel Xeon Silver 4110	660 GFlops	23.1 GB/s	2x85 W	Processor-Centric
GPU	NVIDIA Tesla V100	14.13 TFlops	897 GB/s	300 W	
PIM	UPMEM 1st Gen.	4.66 GFlops	1.77 TB/s	379 W	Memory-Centric

CPU/GPU Comparisons

- Kernel-Only (COO, 32-bit float):
  - CPU = 0.51% of Peak Perf.
  - GPU = 0.21% of Peak Perf.
  - PIM (1D) = 50.7% of Peak Perf.

- End-to-End (COO, 32-bit float):
  - CPU = 4.08 GFlop/s
  - GPU = 1.92 GFlop/s
  - PIM (1D) = 0.11 GFlop/s

System		Peak Performance	Bandwidth	TDP	
CPU	Intel Xeon Silver 4110	660 GFlops	23.1 GB/s	2x85 W	Processor-Centric
GPU	NVIDIA Tesla V100	14.13 TFlops	897 GB/s	300 W	
PIM	UPMEM 1st Gen.	4.66 GFlops	1.77 TB/s	379 W	Memory-Centric

CPU/GPU Comparisons

- Kernel-Energy (COO, 32-bit float):
  - CPU = 0.247 J
  - GPU = 0.051 J
  - PIM (1D) = $\boxed{0.179 \text{ J}}$

PIM: 1.38x higher energy efficiency over CPU

System		Peak Performance	Bandwidth	TDP	
CPU	Intel Xeon Silver 4110	660 GFlops	23.1 GB/s	2x85 W	Processor-Centric
GPU	NVIDIA Tesla V100	14.13 TFlops	897 GB/s	300 W	
PIM	UPMEM 1st Gen.	4.66 GFlops	1.77 TB/s	379 W	Memory-Centric

CPU/GPU Comparisons

- Kernel-Energy (COO, 32-bit float):
  - CPU = 0.247 J
  - GPU = 0.051 J
  - PIM (1D) = 0.179 J

System		Peak Performance	Bandwidth	TDP
	Intel Xeon ... 1st Gen.			

Centric

Many more results in the full paper:
https://arxiv.org/pdf/2201.05072.pdf

Outline

SpMV Kernels for Real PIM Systems

Key Takeaways from Our Study

Conclusion

Conclusion

- SpMV is a fundamental linear algebra kernel for important applications (HPC, machine learning, graph analytics…)
- SpMV is a highly memory-bound kernel in processor-centric systems (e.g., CPU and GPU systems)
- Real near-bank PIM systems can tackle the data movement bottleneck (high parallelism, large aggregate memory bandwidth)
- Key Contributions:
  - *SparseP* : first open-source SpMV library for real PIM systems
  - Comprehensive characterization and analysis of SPMV on the first real PIM system
  - Recommendations to improve multiple aspects of future PIM hardware and software

Our Work

SparseP: https://github.com/CMU-SAFARI/SparseP
Full Paper: https://arxiv.org/pdf/2201.05072.pdf

SparseP

Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures

Christina Giannoula

Ivan Fernandez, Juan Gomez-Luna,

Nectarios Koziris, Georgios Goumas, Onur Mutlu

SAFARI ETH zürich National Technical University of Athens CSLab UNIVERSIDAD DE MÁLAGA