Systems Software and Libraries for Sparse Computational Kernels in PIM Architectures

Christina Giannoula

Tutorial on Memory-Centric Computing Systems
MICRO 2024

Sparse Data is Everywhere

Sparse Data is Everywhere

Sparse Tables:

Recommendation Data

Sparse Data Processing Applications

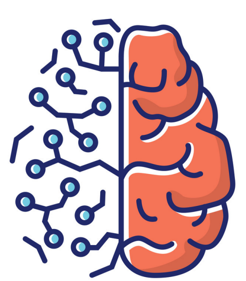
Graph Analytics

Databases

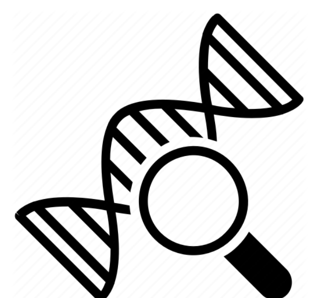
Medical Imaging

How can we accelerate the sparse kernels?

Neural Networks



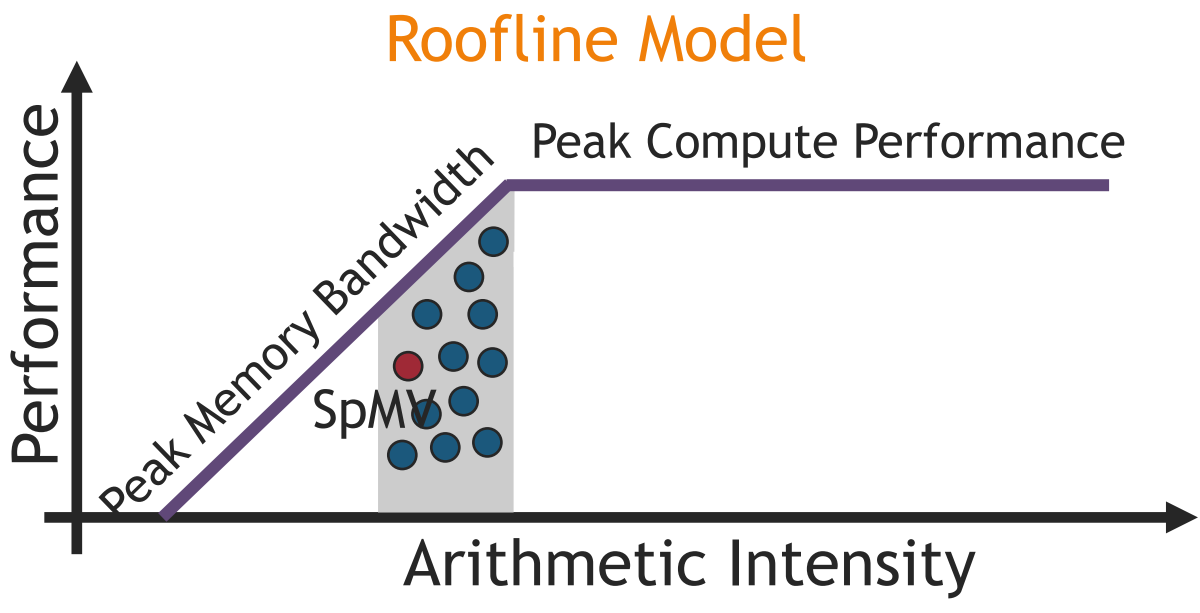
Bioinformatics



Economic Modeling

Sparse Computational Kernels

- E.g., Sparse Matrix Vector/Matrix Multiplication
- Characteristics:
  - Random memory accesses
    - Not sequential/strided
    - Input-driven
  - Low arithmetic intensity
- Highly memory-bound kernels in CPUs/GPUs

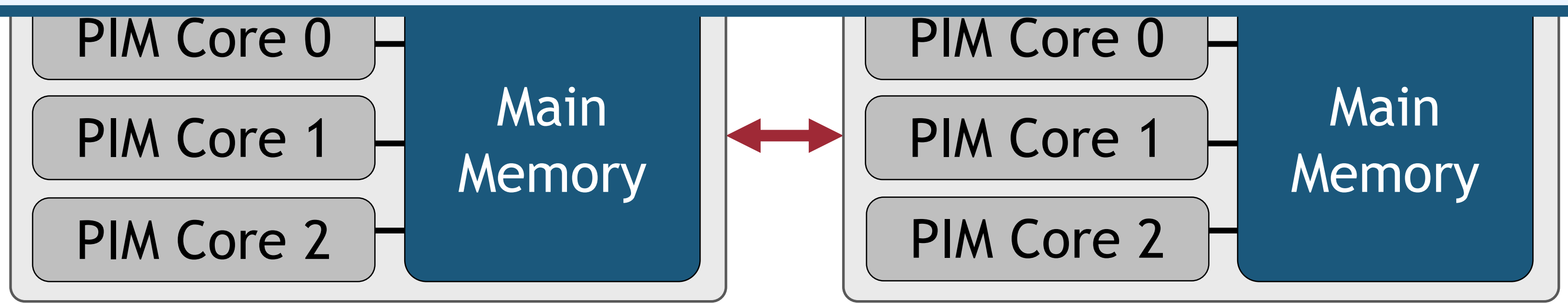


Processing-In-Memory Systems

Processing-In-Memory (PIM) Systems:

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

PIM constitutes a promising paradigm for accelerating sparse kernels



The Challenge

Real Processing-In-Memory (PIM) Systems:

- Different architectures
- Software stacks are still in early stage
- Specialized low-level programming interfaces

Programming a real PIM architecture for a high-level application is a hard task

Instruction-level API

Kwon+, [ISSCC 2021]

C-like API

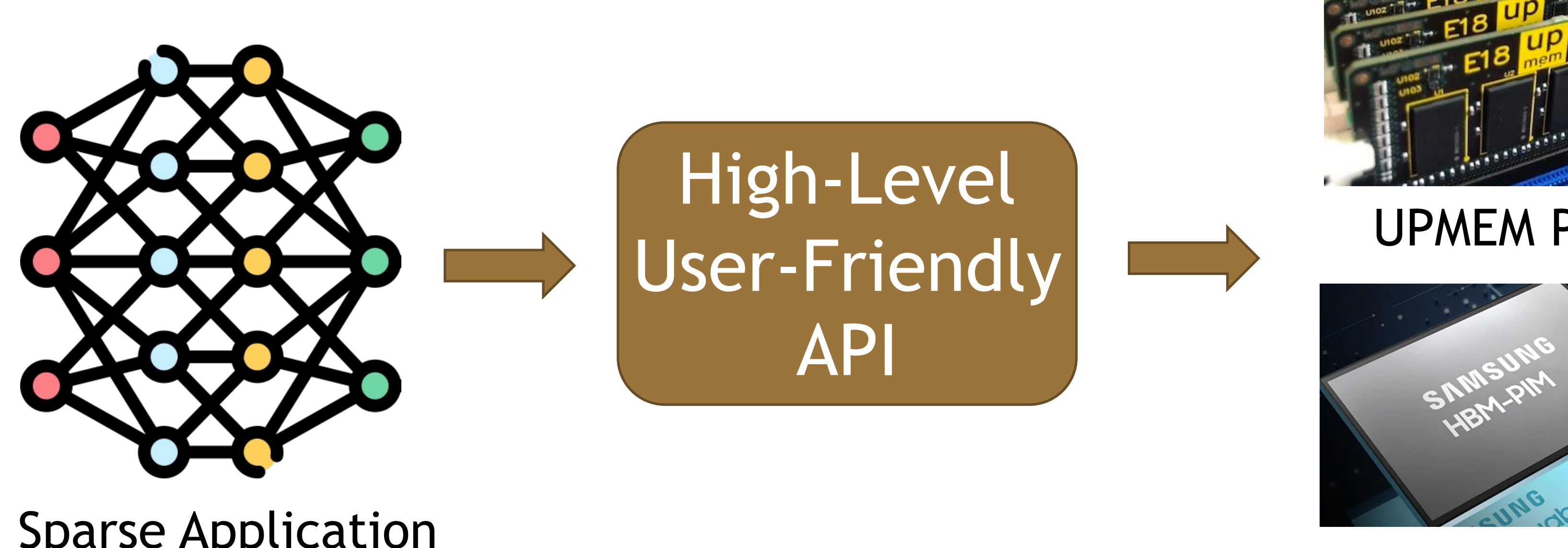
https://www.upmem.com

Instruction-level API

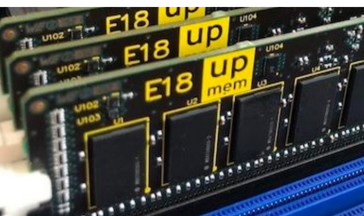
Lee+, [ISSCC 2022]

Our Goal

Bridge the programming gap
between software engineers/researchers
and real-world PIM architectures



Sparse Application

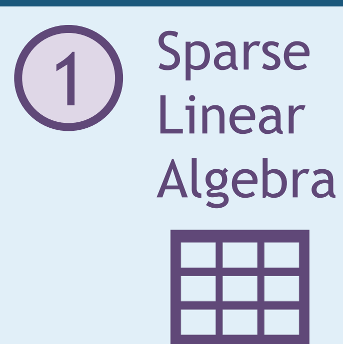


UPMEM PIM

HBM PIM

Outline

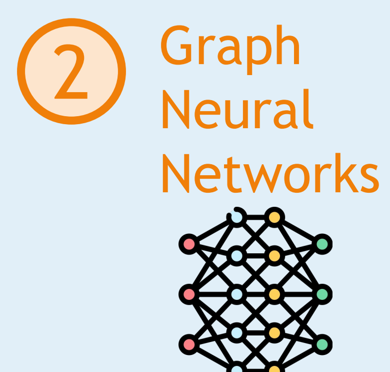
1 Sparse Linear Algebra



SparseP (Sigmetrics'22)

A Library of Efficient Sparse Matrix Vector Multiplication Kernels for Real PIM Systems

2 Graph Neural Networks



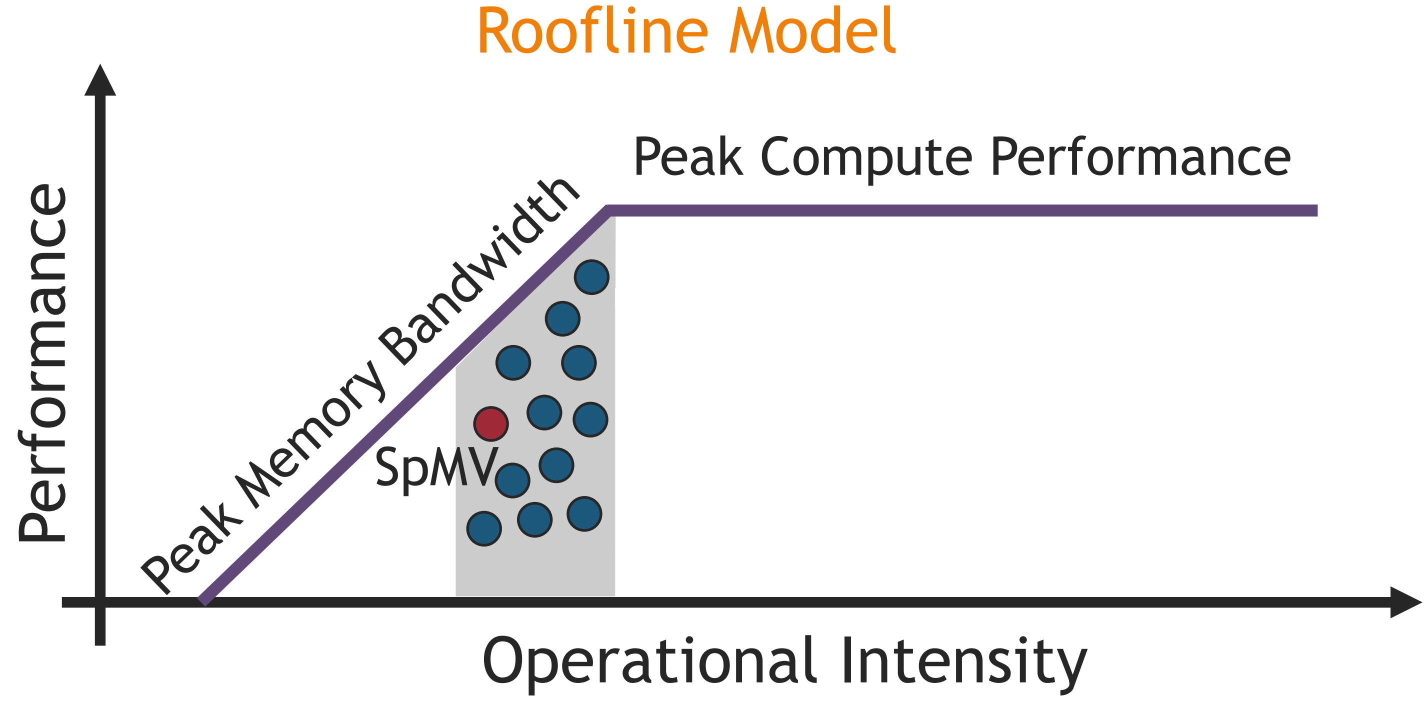
PyGim (Sigmetrics'25)

An Efficient Graph Neural Network Framework for Real PIM Systems

Sparse Matrix Vector Multiplication

Sparse Matrix Vector Multiplication (SpMV):

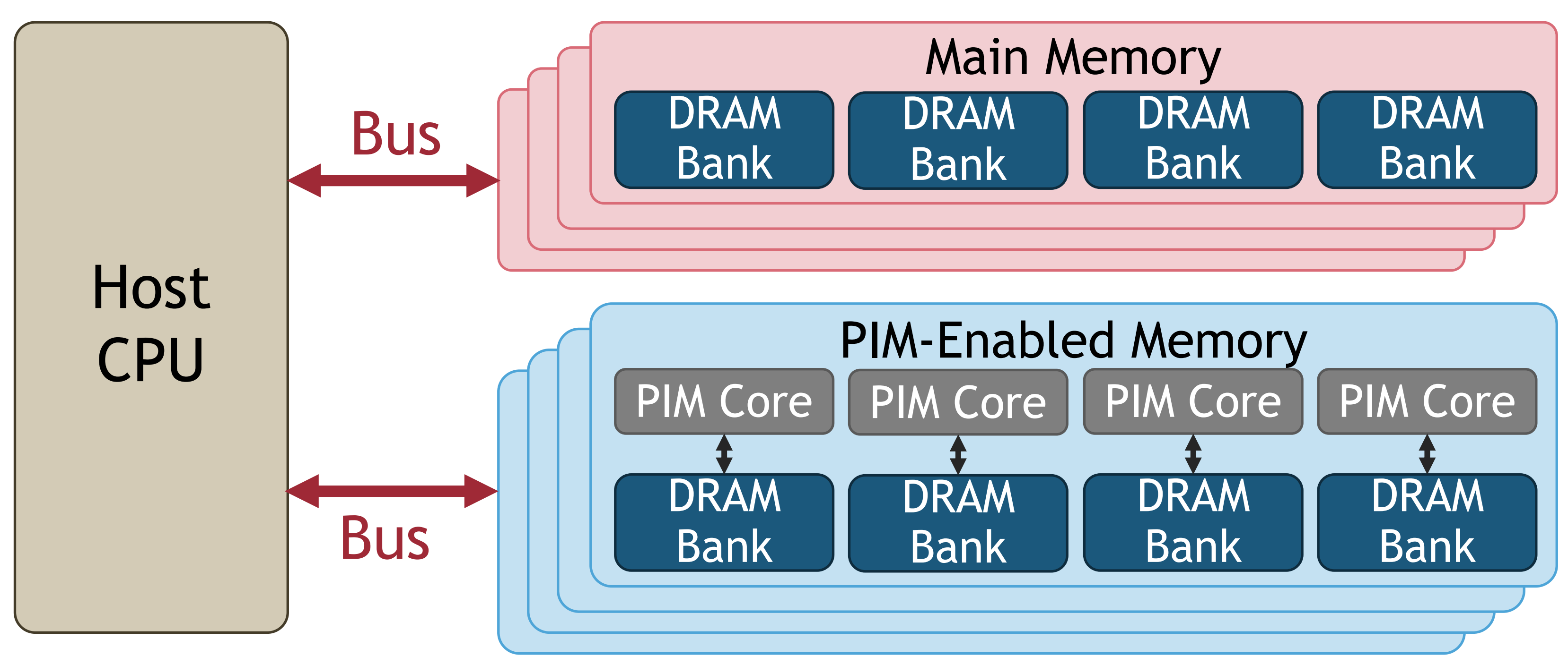
- Widely-used kernel in graph processing, machine learning, scientific computing ...
- A highly memory-bound kernel



Real Near-Bank PIM Systems

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

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



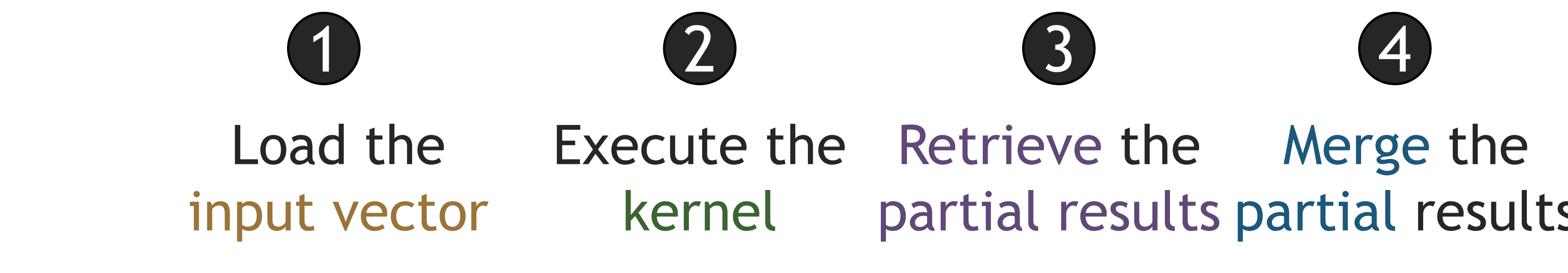
SparseP: SpMV Library for Real PIMs

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 upmem
   - 26 sparse matrices
   - Comparisons to state-of-the-art CPU and GPU systems
   - Recommendations for software, system and hardware designers

SpMV Execution on a PIM System



Main Memory

DRAM Bank

DRAM Bank

bus

PIM-Enabled Memory

PIM Core

PIM Core

PIM Core

DRAM Bank

DRAM Bank

DRAM Bank

bus

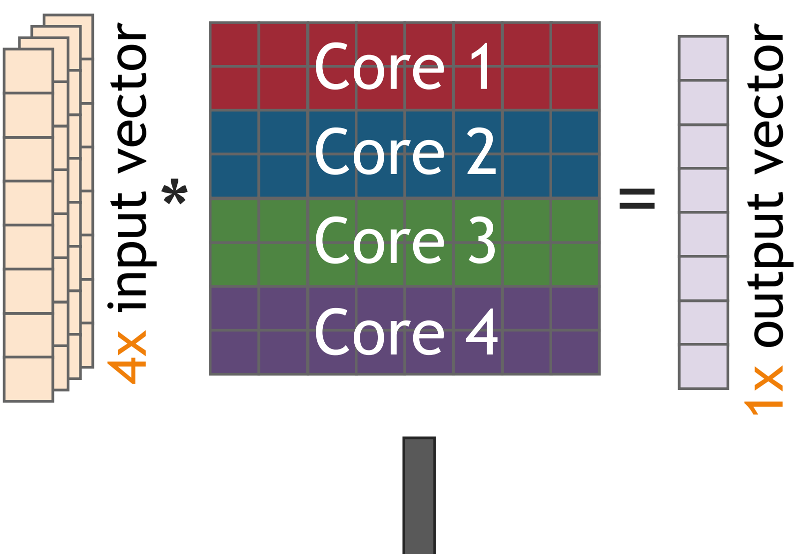
Host CPU

+

Data Partitioning Techniques

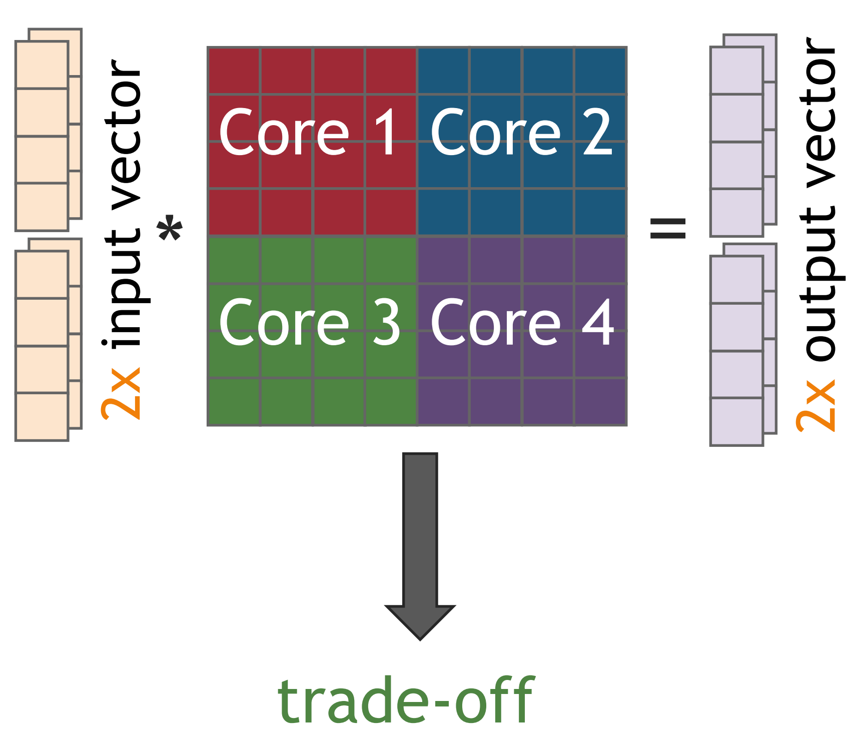
SparseP supports two types of data partitioning techniques:

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:

- #Rows or #NNZs
- CSR (row-granularity), COO

CSR

row-granularity

Core 1

Core 2

Core 3

COO

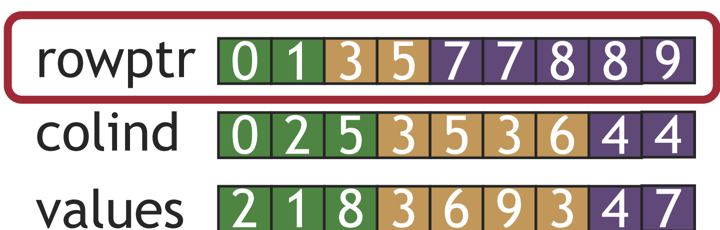
nnz-granularity

Core 1

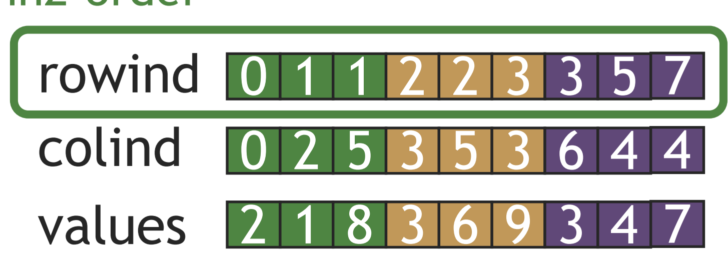
Core 2

Core 3

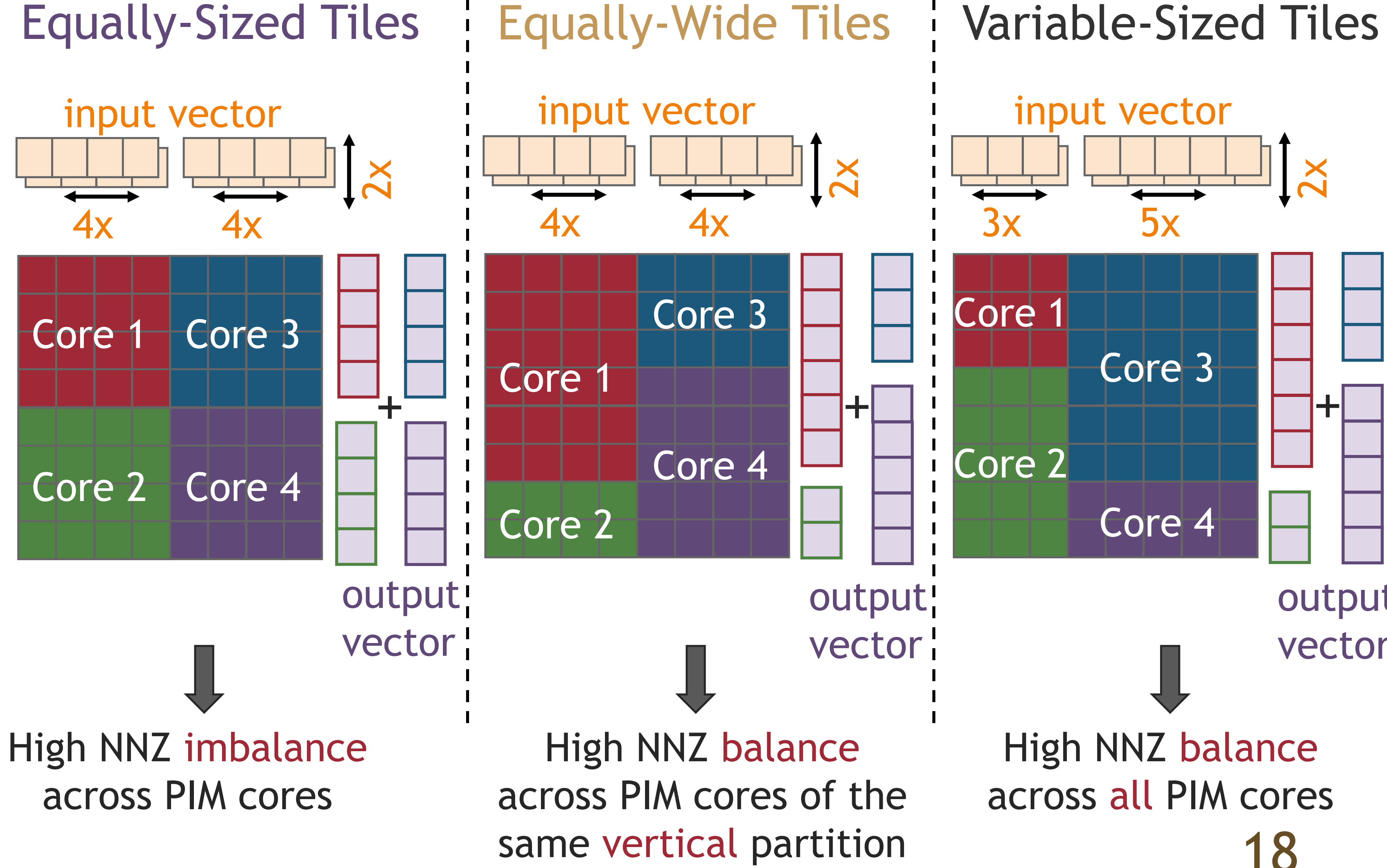
row-order



nnz-order

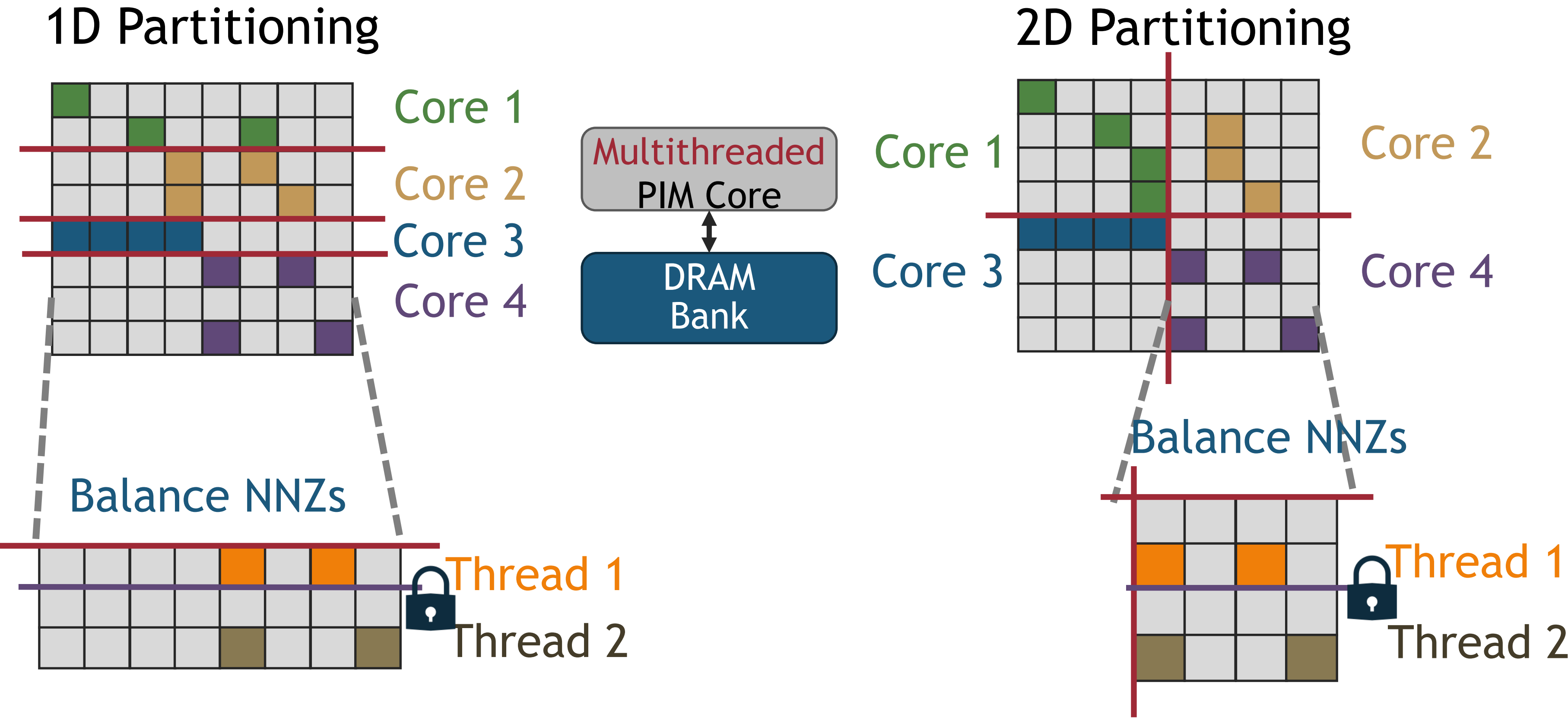


2D Partitioning Technique



Parallelization across Threads

Multithreaded PIM Cores:



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

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

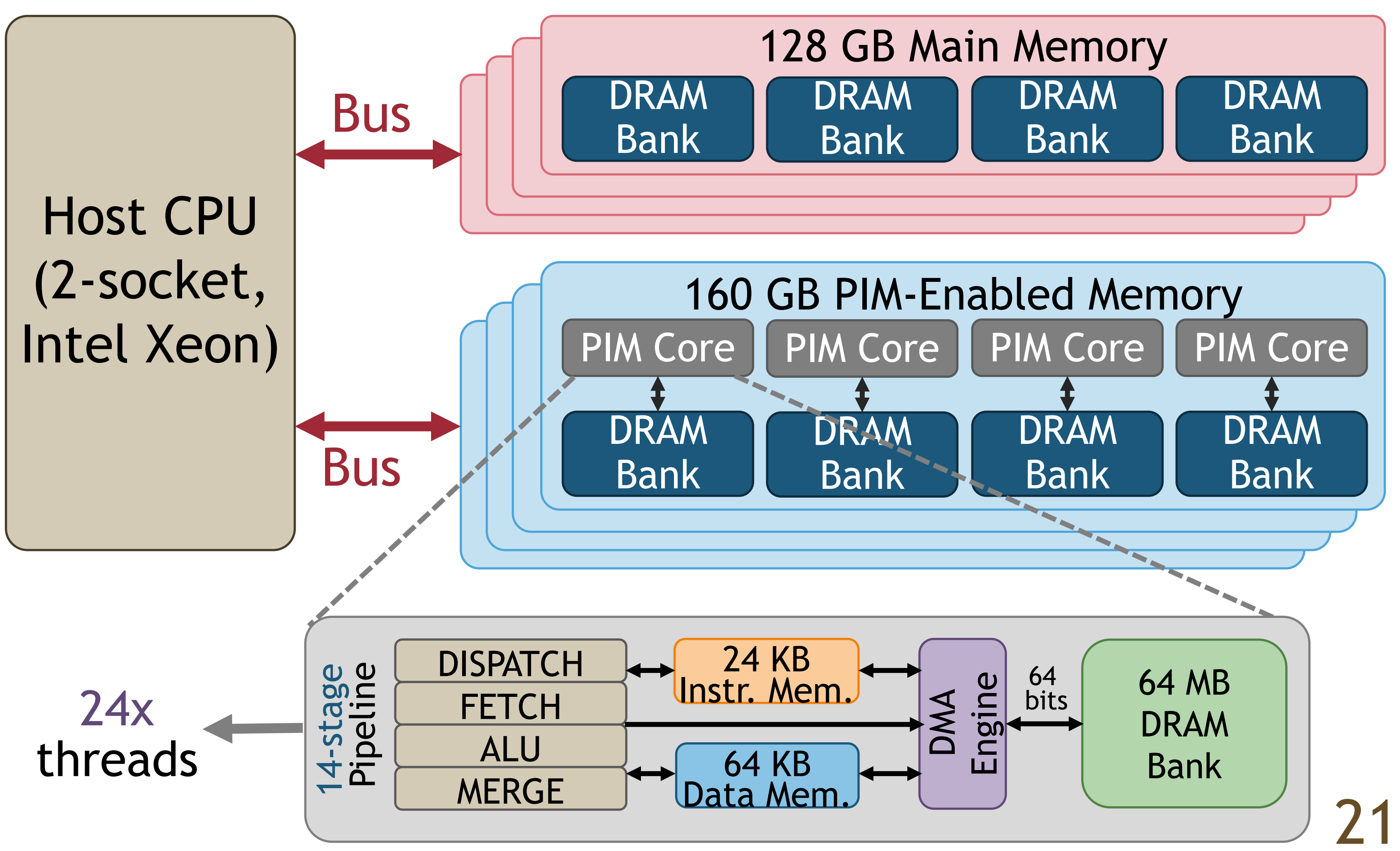
Load-balance
across PIM cores/threads:
* row-granularity (CSR)
^ block-row-granularity (BCSR)

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

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

UPMEM-based PIM System

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

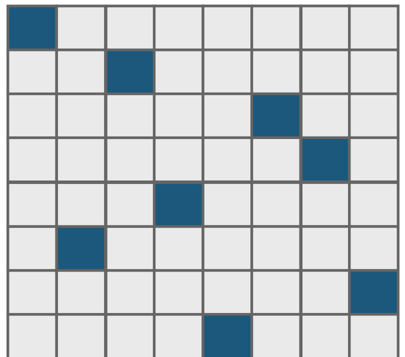


Sparse Matrix Data Set

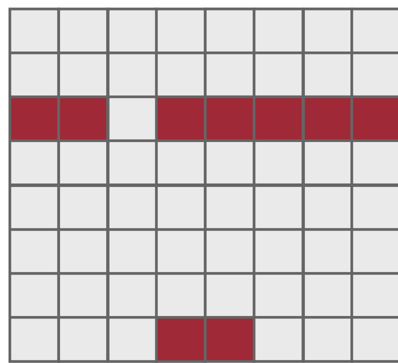
26 sparse matrices*:

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

Regular Matrix

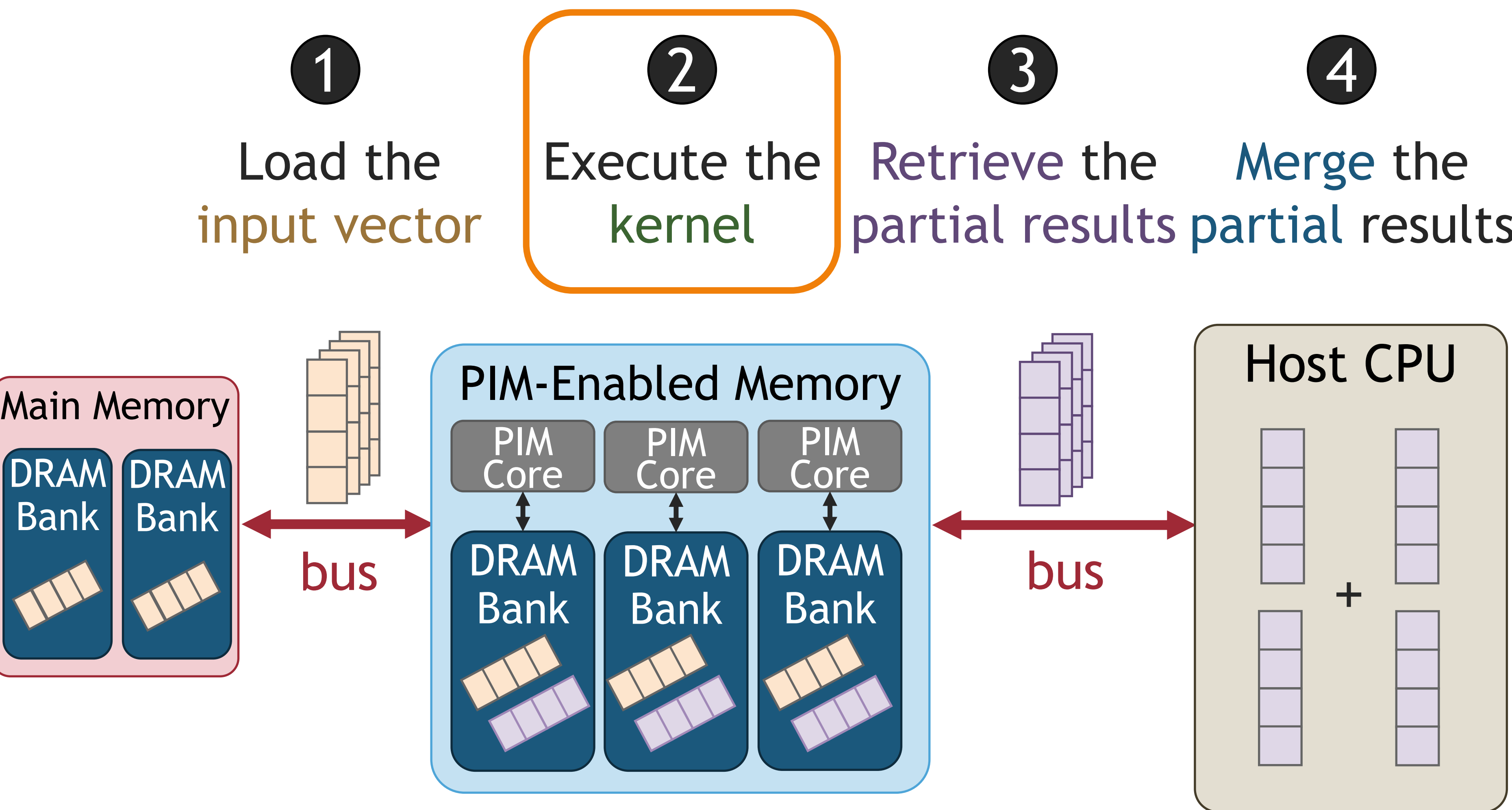


Scale-Free Matrix



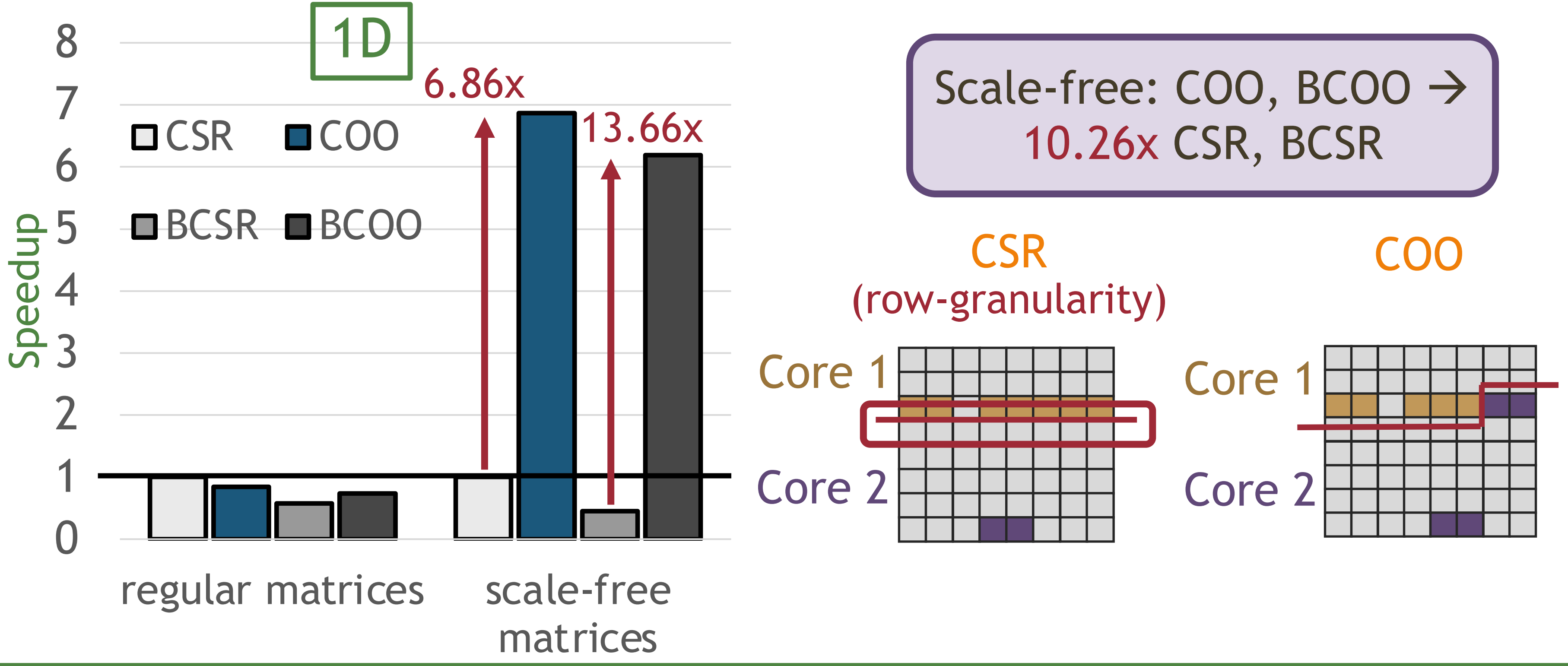
* Suite Sparse Matrix Collection: https://sparse.tamu.edu/

Kernel Execution on PIM Cores



Comparison of Compressed Formats

2048 PIM Cores, 32-bit integer



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

1D

2D Equally-Sized

Speedup

0

regular matrices

scale-free matrices

0

regular matrices

scale-free matrices

2D Equally-Wide

2D Variable-Sized

50

40

30

20

10

0

Speedup

0

regular matrices

scale-free matrices

regular matrices

scale-free matrices

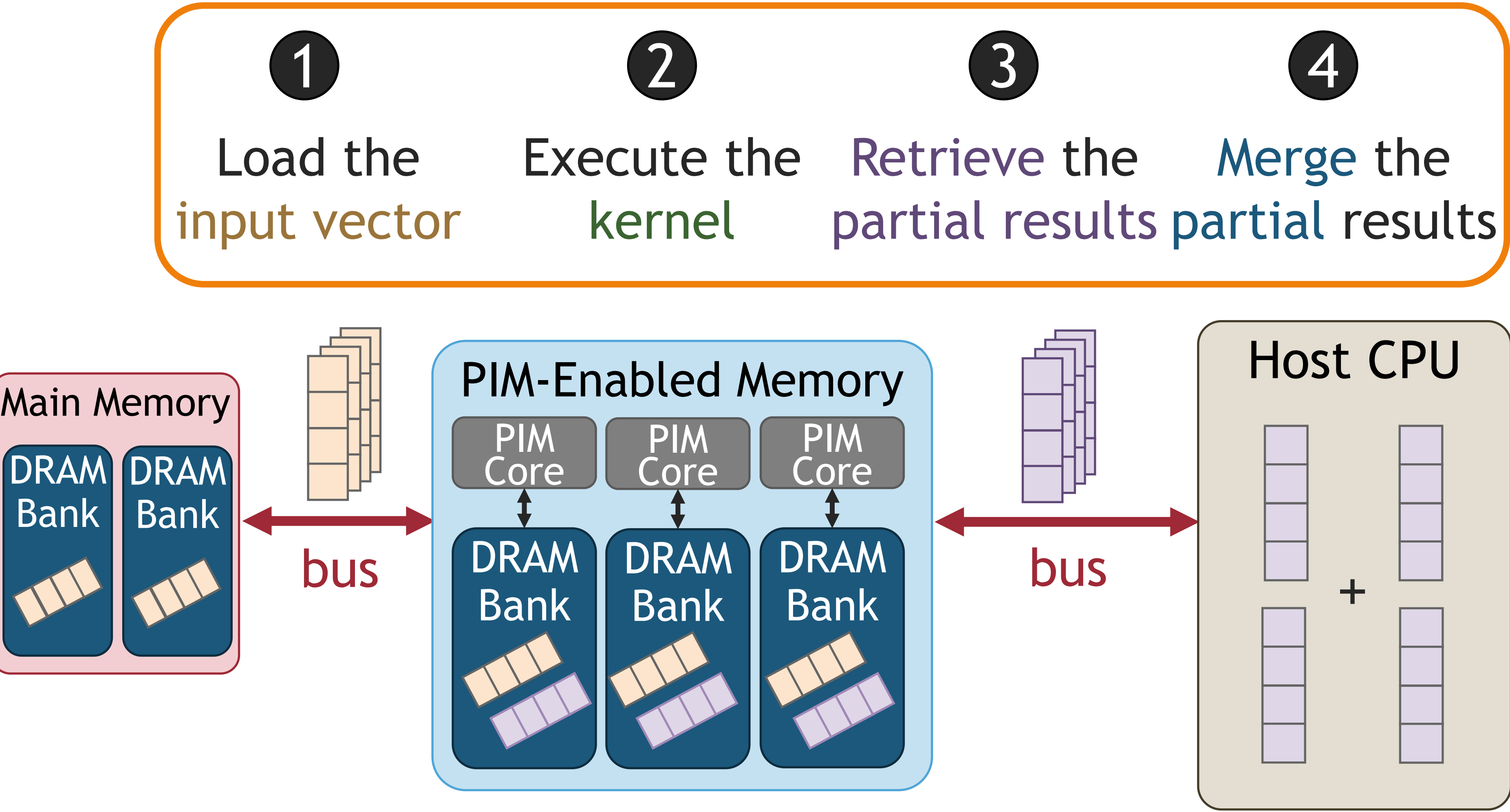
Key Takeaway 1

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.

Recommendation 1

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).

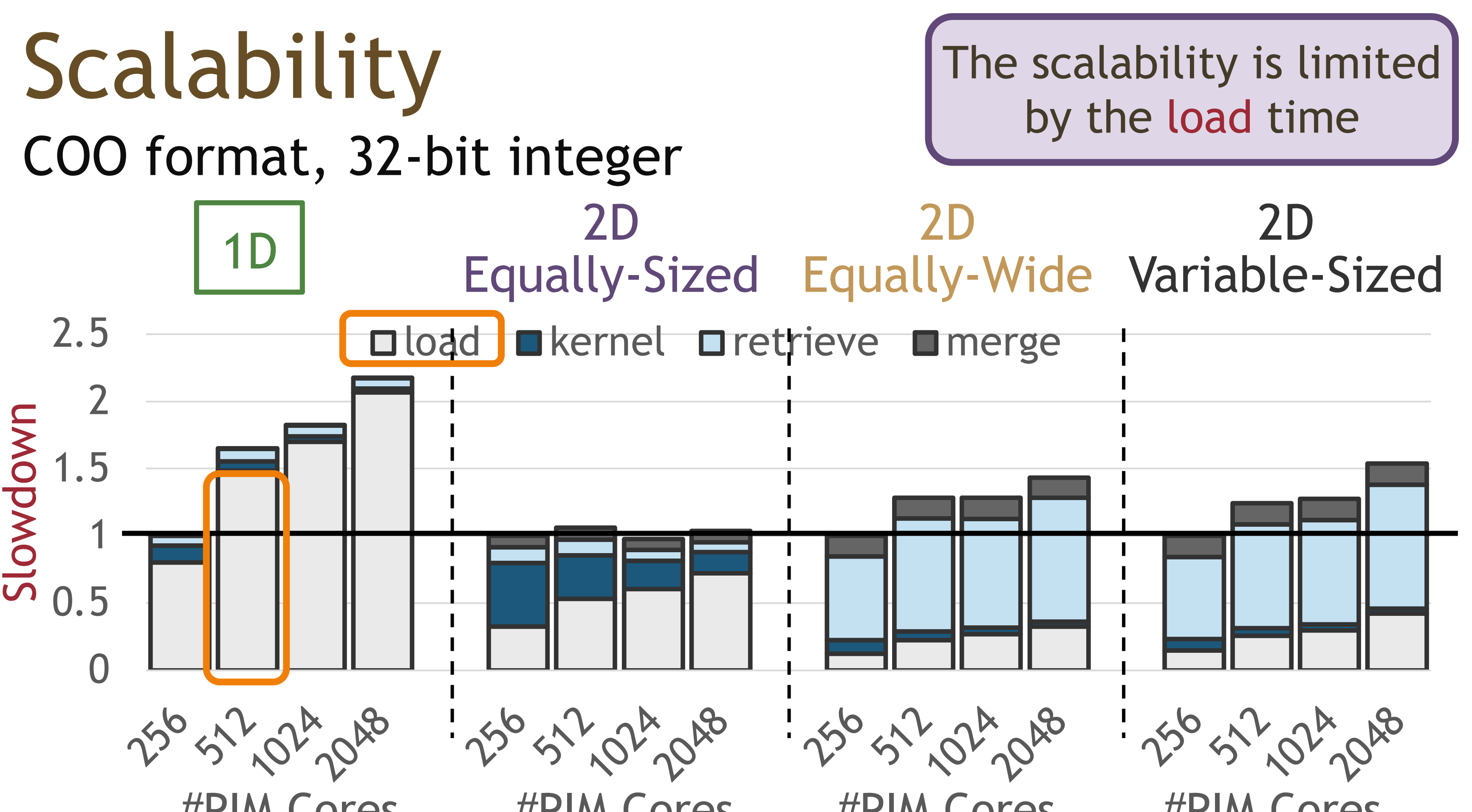
End-to-End Performance



Scalability

COO format, 32-bit integer

The scalability is limited by the load time



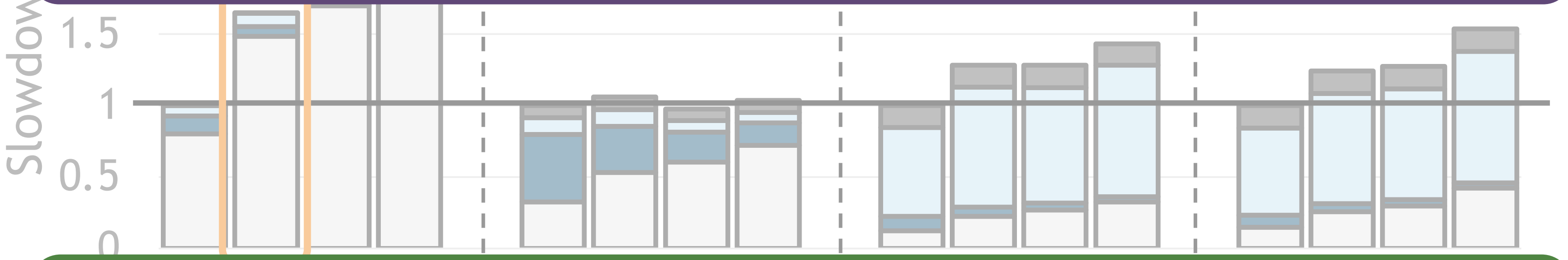
1D: #bytes to load the input vector grows linearly to #PIM cores

Scalability

COO format, 32-bit integer

Key Takeaway 2

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.



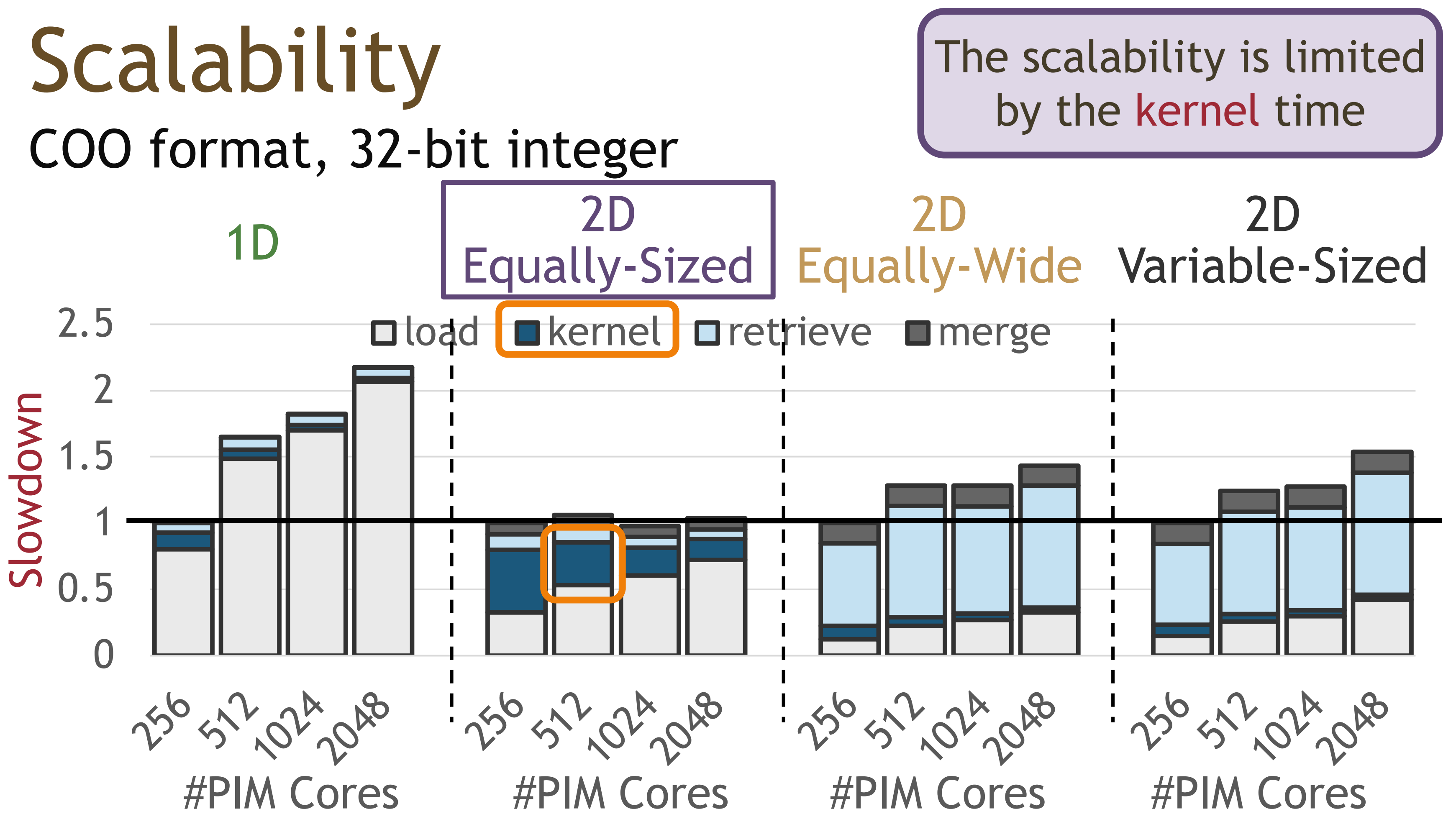
Recommendation 2

Optimize the broadcast collective operation 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

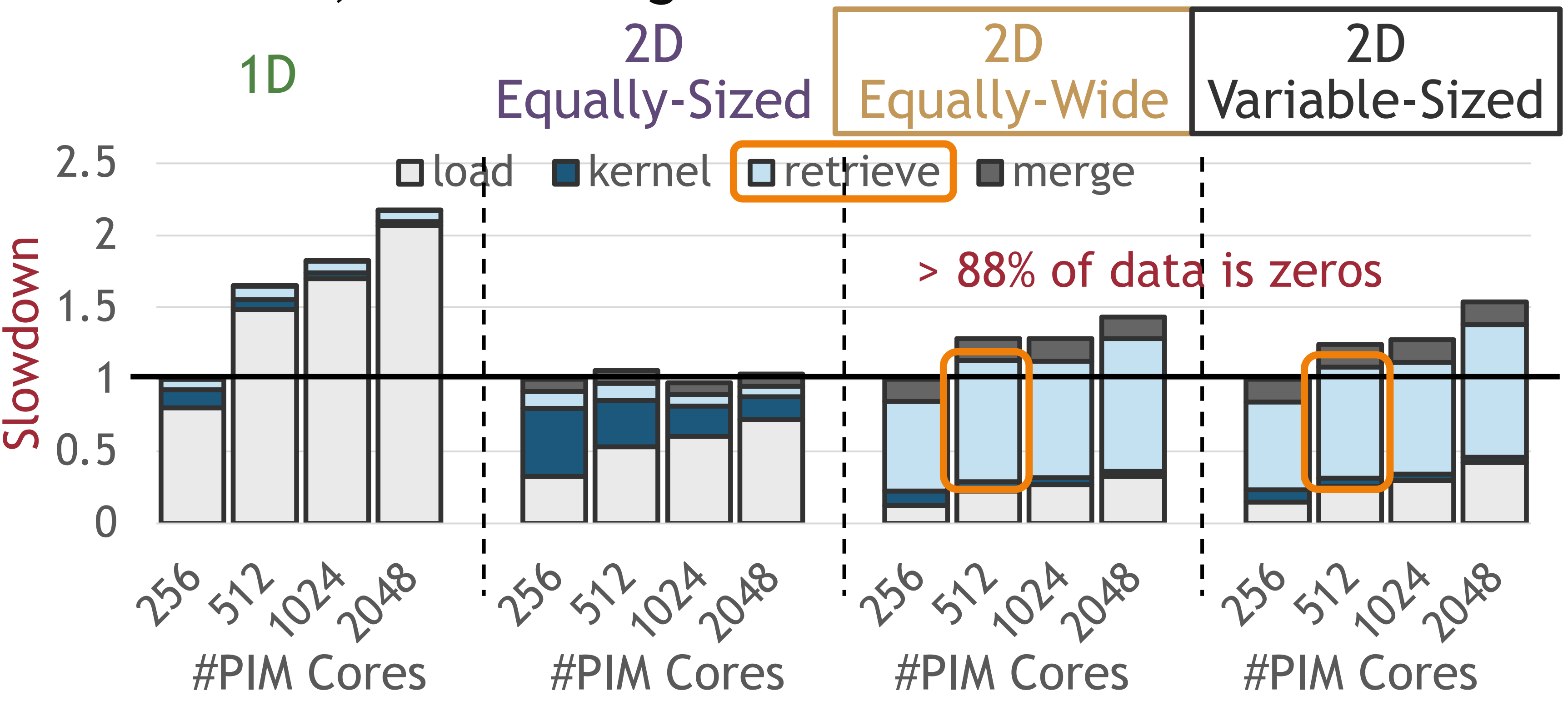


2D Equally-Sized: kernel time is limited by only a few PIM cores assigned to the 2D tiles with the largest #NNZs

Scalability

COO format, 32-bit integer

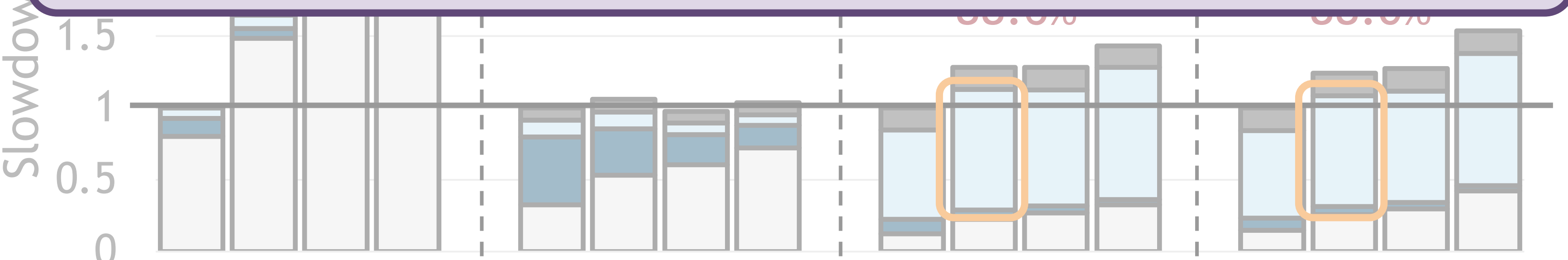
The scalability is limited by the retrieve time



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 3

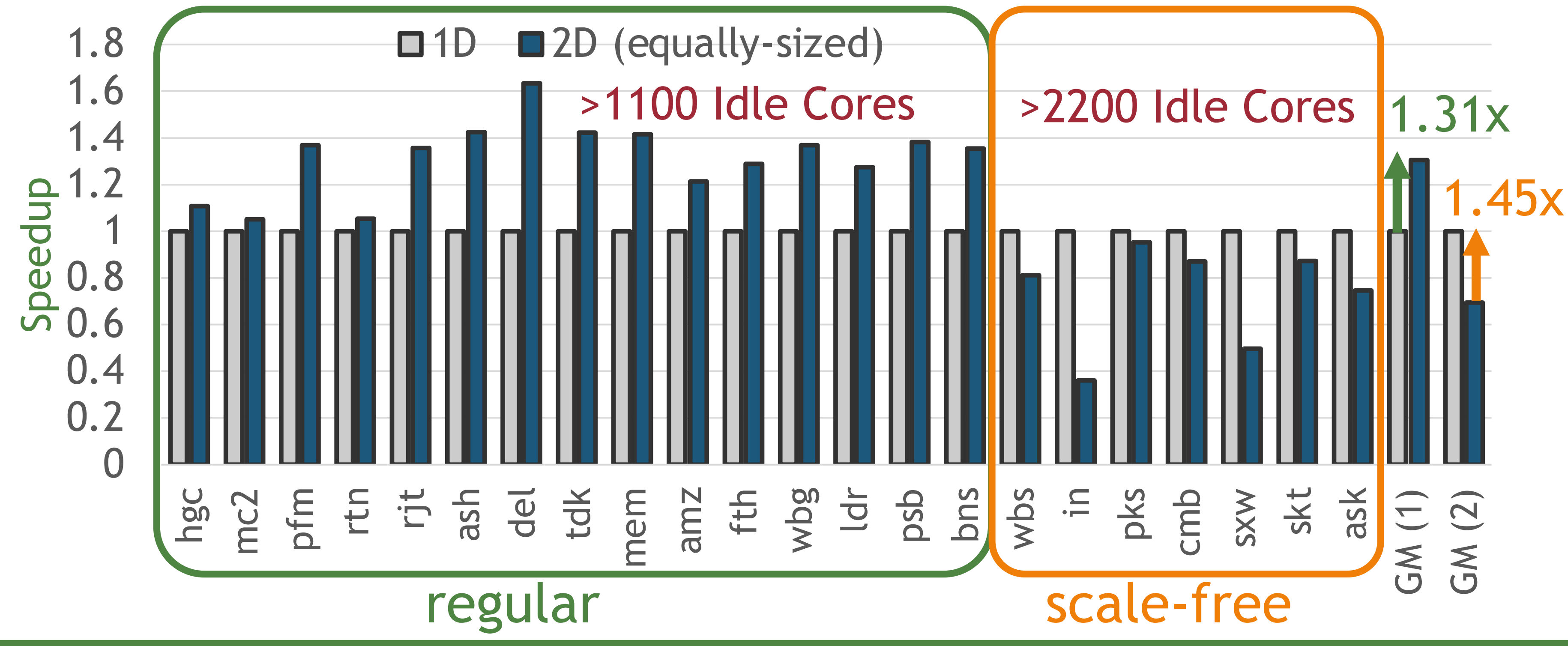
The 2D equally-wide and variable-sized kernels need fine-grained parallel data transfers at DRAM bank granularity (zero padding) to be supported by the PIM system to achieve high performance.

Recommendation 3

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.

1D vs 2D

Up to 2528 PIM Cores, 32-bit float

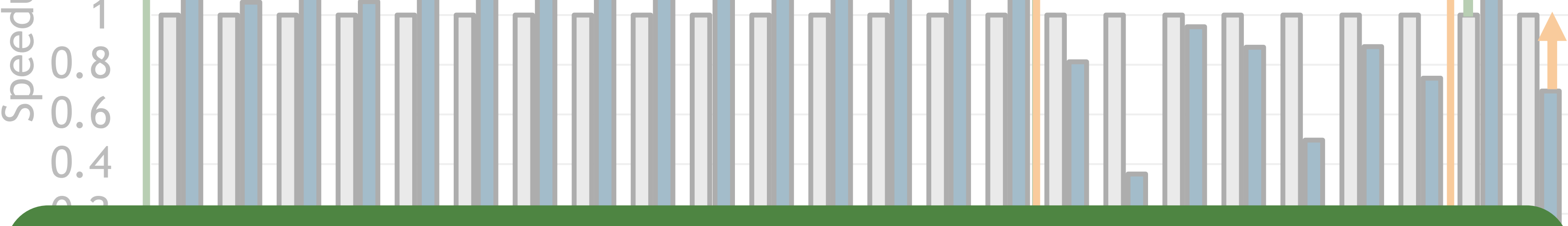


Best-performing SpMV execution:
trades off computation with lower data transfer costs

1D vs 2D

Key Takeaway 4

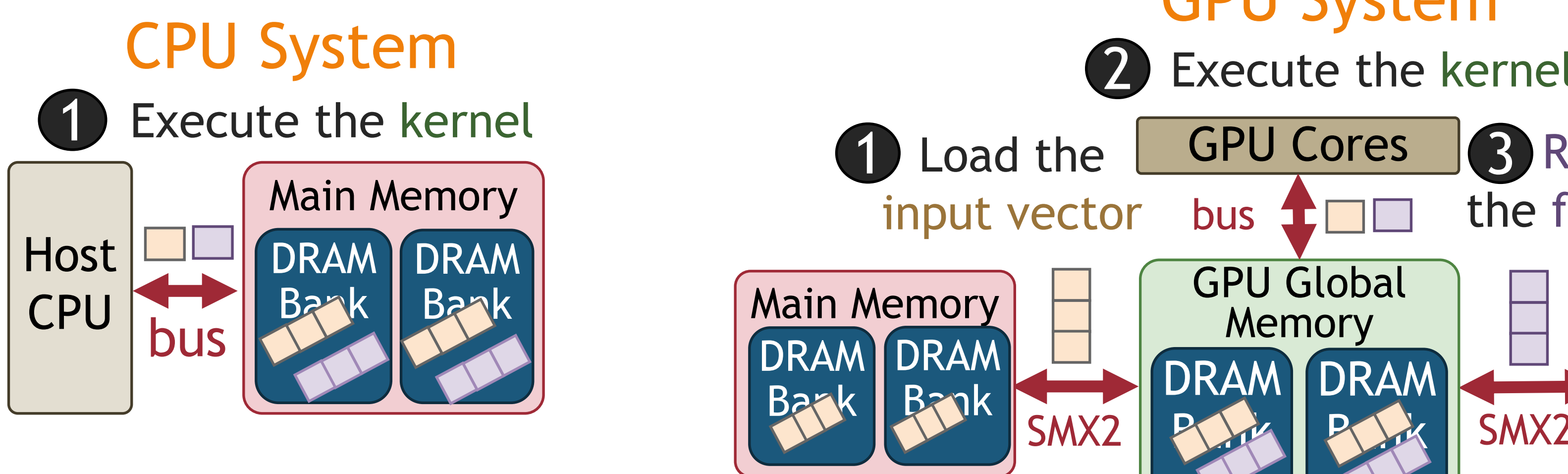
Expensive data transfers to/from PIM-enabled memory performed via the narrow memory bus impose significant performance overhead to end-to-end SpMV execution. Thus, it is hard to fully exploit all available PIM cores of the system.

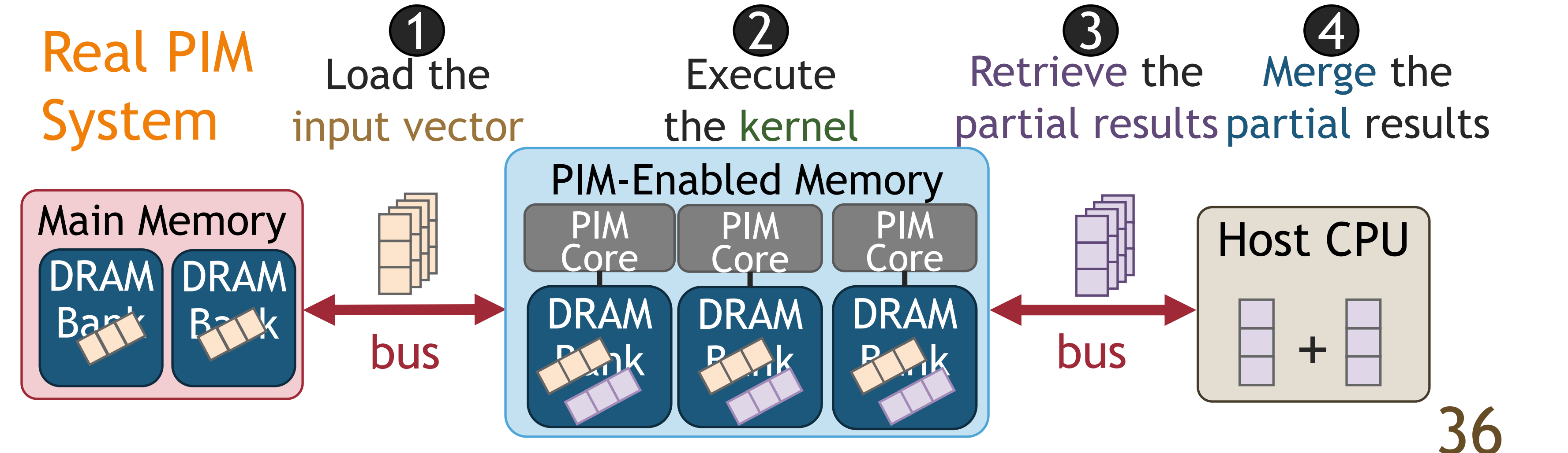


Recommendation 4

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

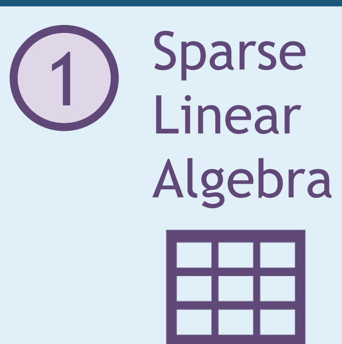
CPU/GPU Comparisons

- Kernel-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

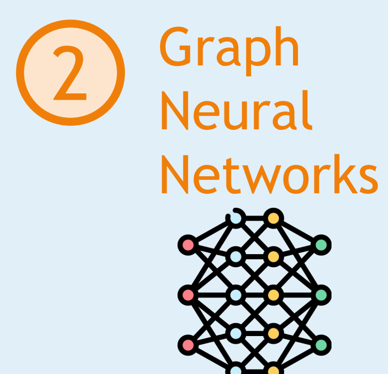
Many more results in the full paper:
https://arxiv.org/pdf/2201.05072.pdf

Outline



SparseP (Sigmetrics'22)

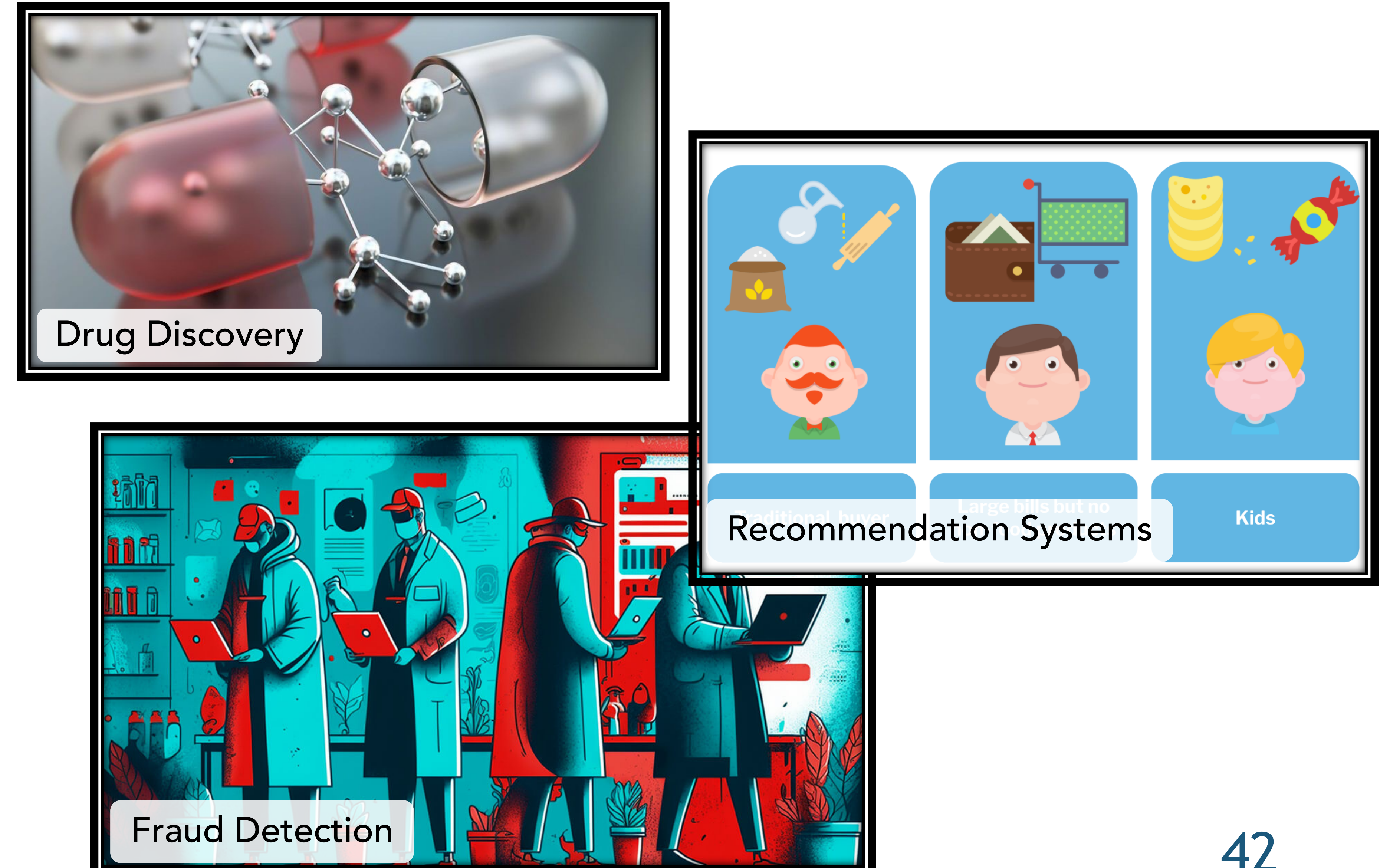
A Library of Efficient Sparse Matrix Vector Multiplication Kernels for Real PIM Systems



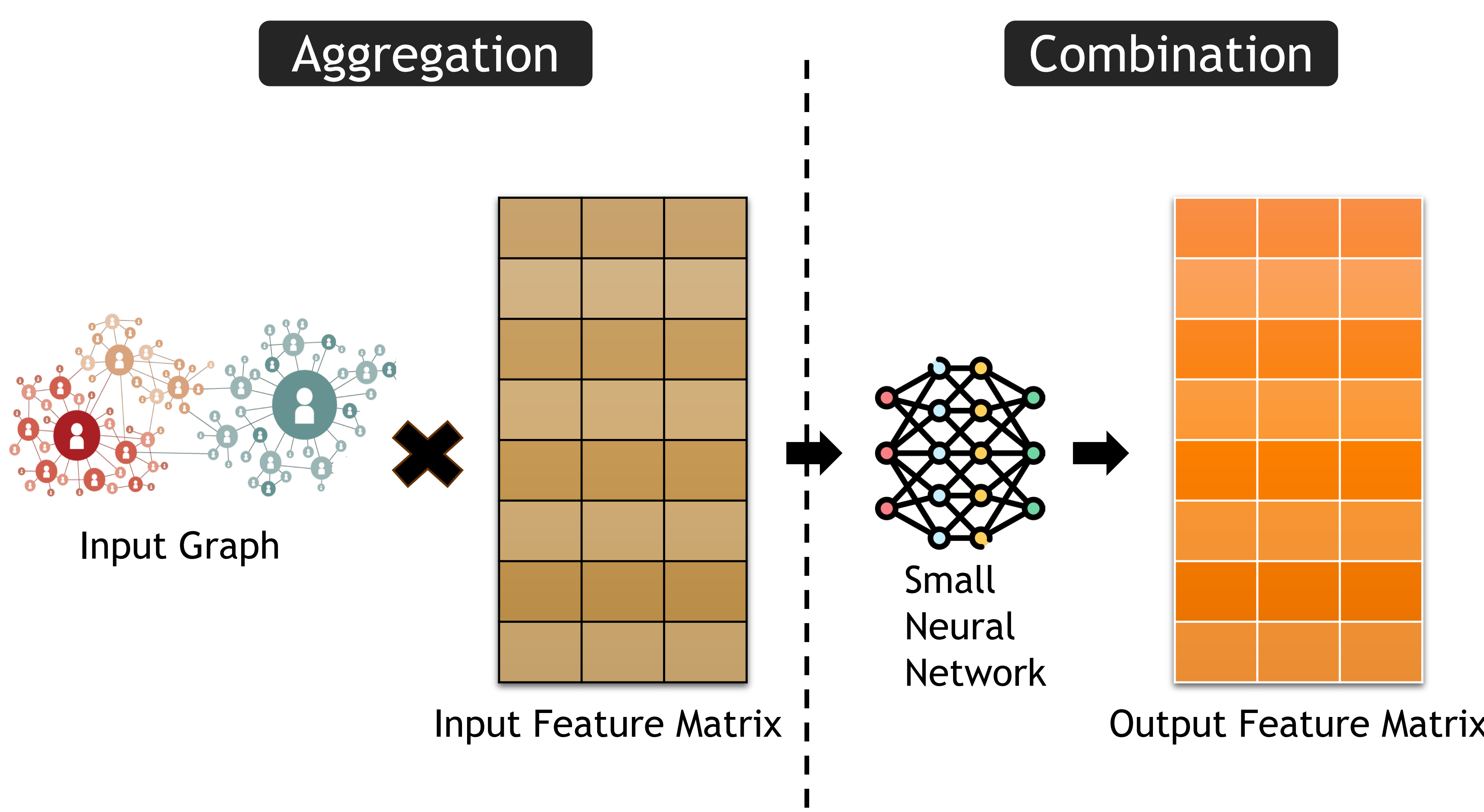
PyGim (Sigmetrics'25)

An Efficient Graph Neural Network Framework for Real PIM Systems

Applications of Graph Neural Networks (GNNs)

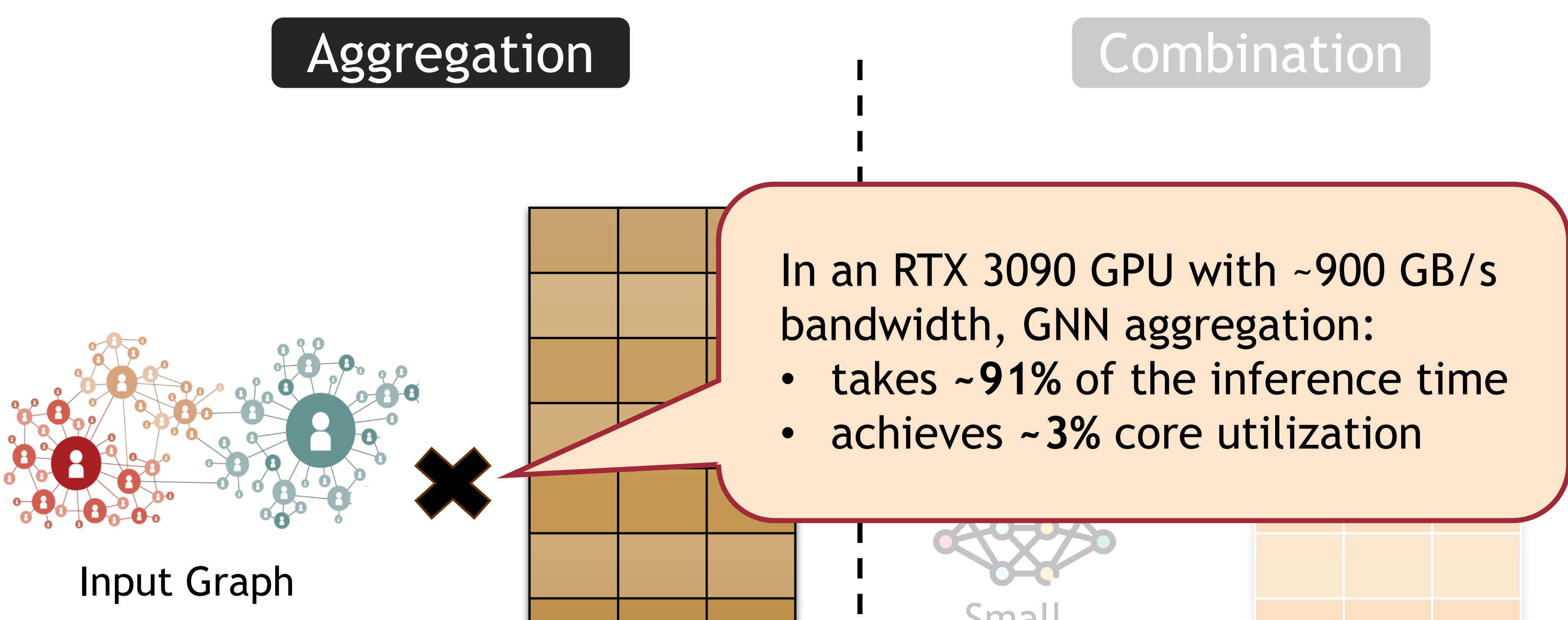


A GNN Layer



A GNN Layer

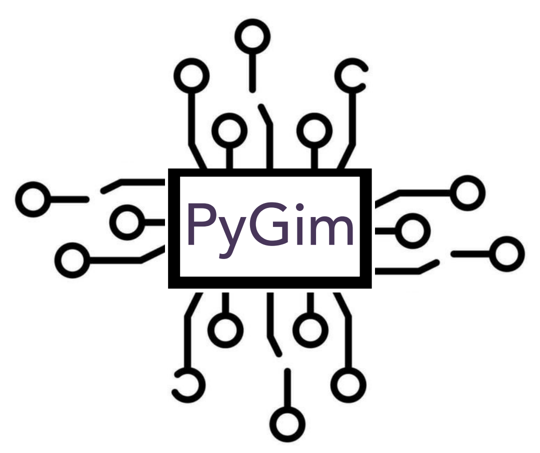
GNN Execution is Bandwidth-Bound



GNN execution is significantly limited by memory bandwidth in processor-centric systems

A GNN Layer

The PyGim Framework: Overview

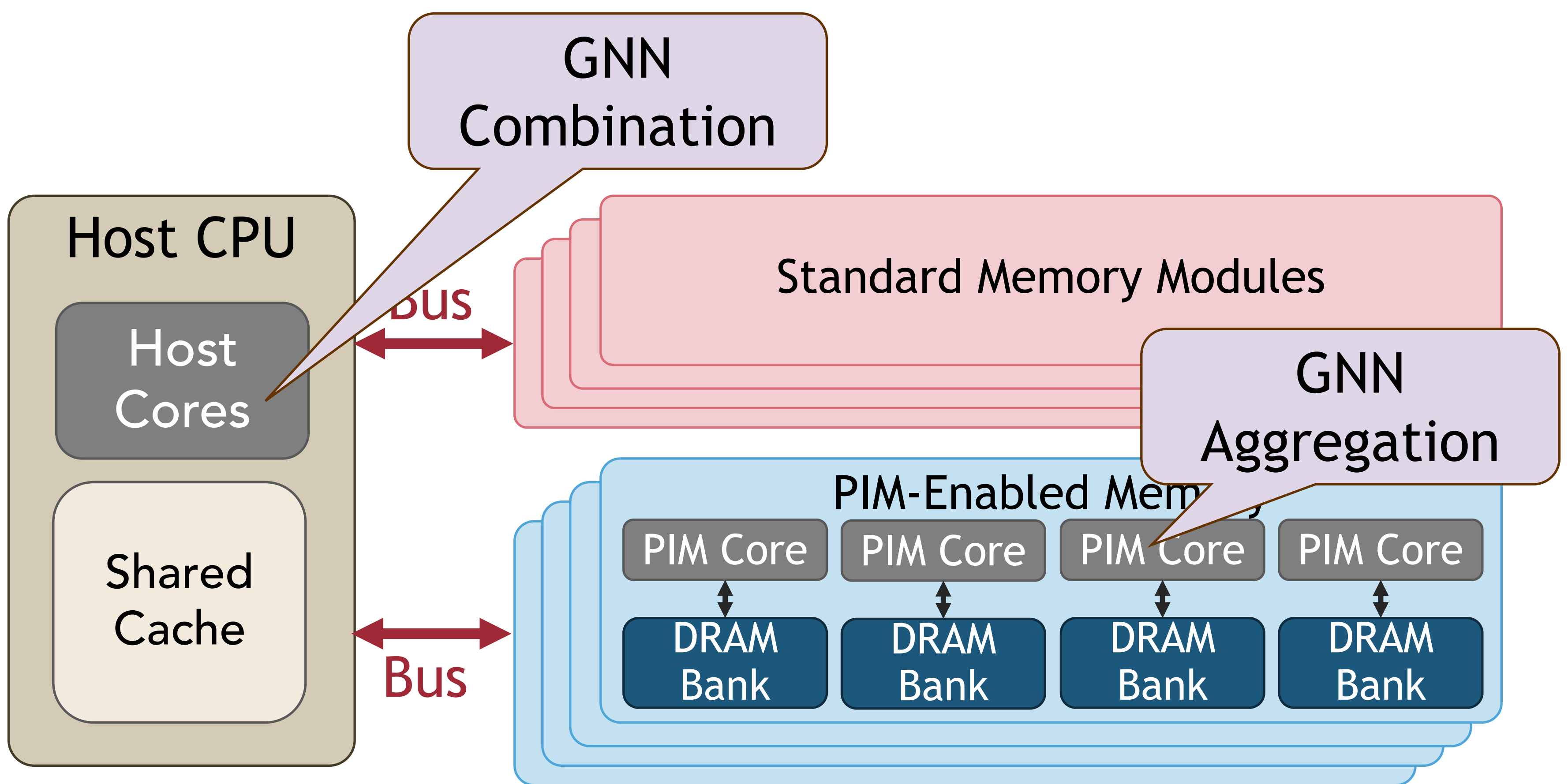


PyGim:

- An efficient GNN framework for real PIM systems
- Bridges the gap between ML engineers and real PIM hardware for GNNs
- Incorporates 4 key techniques:
  1. Cooperative Acceleration (CoA)
  2. Parallelism Fusion (PaF)
  3. Lightweight Tuner
  4. Python-like Programming Interface

PyGim Cooperative Acceleration (CoA)

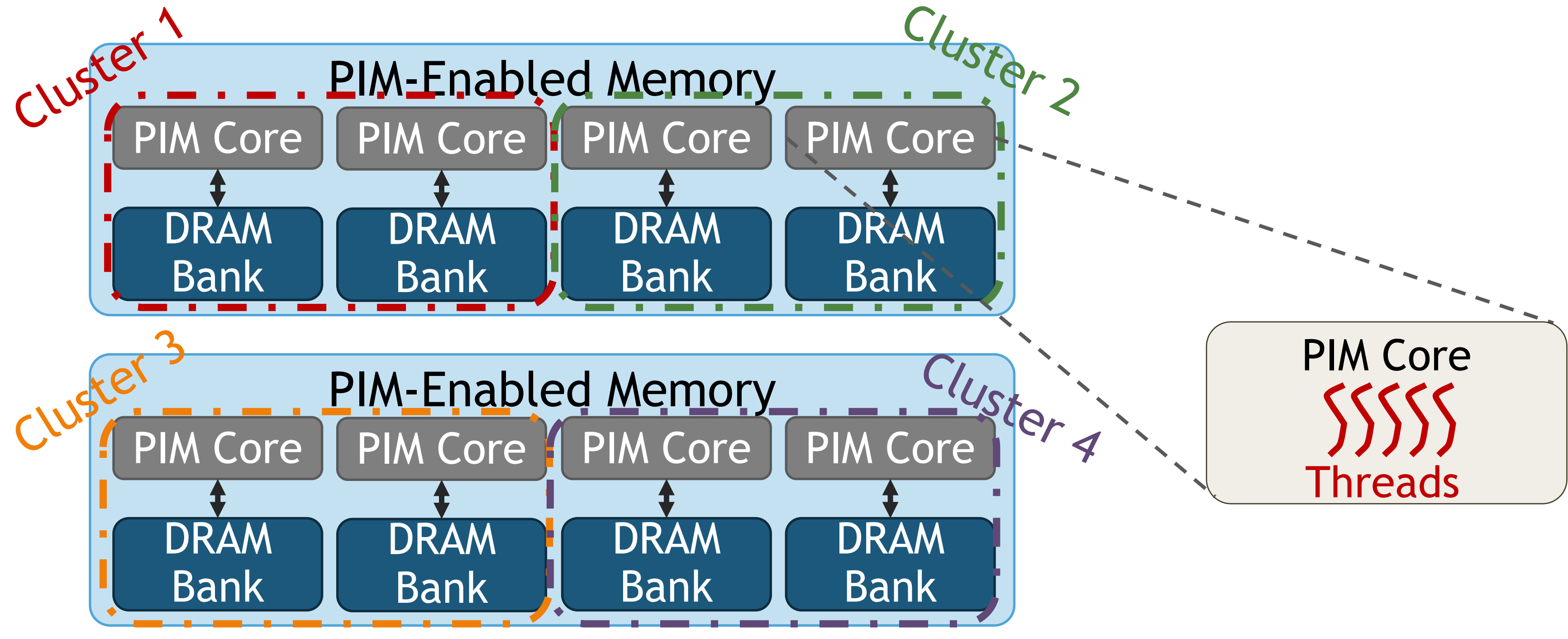
- Combination runs on Host cores
- Aggregation runs on PIM cores



PyGim Parallelism Fusion (PaF)

3 parallelization levels with different strategy at each level

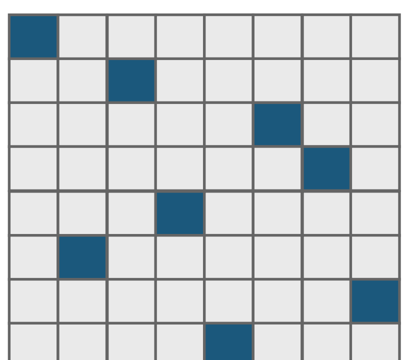
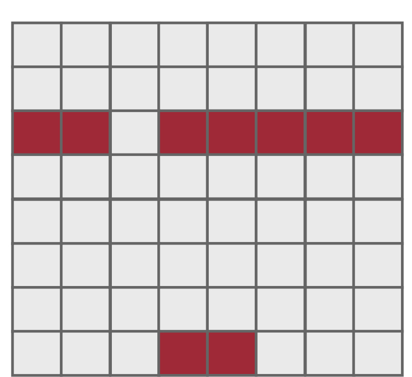
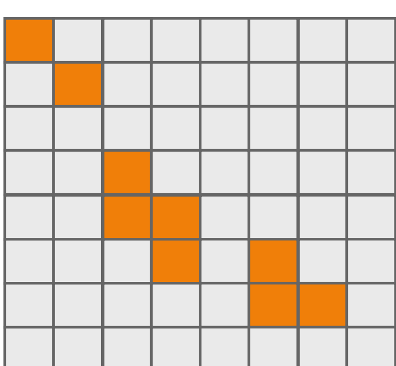
1. Across PIM Clusters: Feature- + Edge-level Parallelism
2. Within a PIM Cluster: Vertex-/Edge-level Parallelism
3. Within a PIM Core: Vertex-/Edge-level Parallelism



PyGim Parallelism Fusion (PaF)

- Provides various parallelization and load balancing strategies across, within PIM clusters and within a PIM core

- Strives a balance between computation and data transfer costs for various real-world graphs

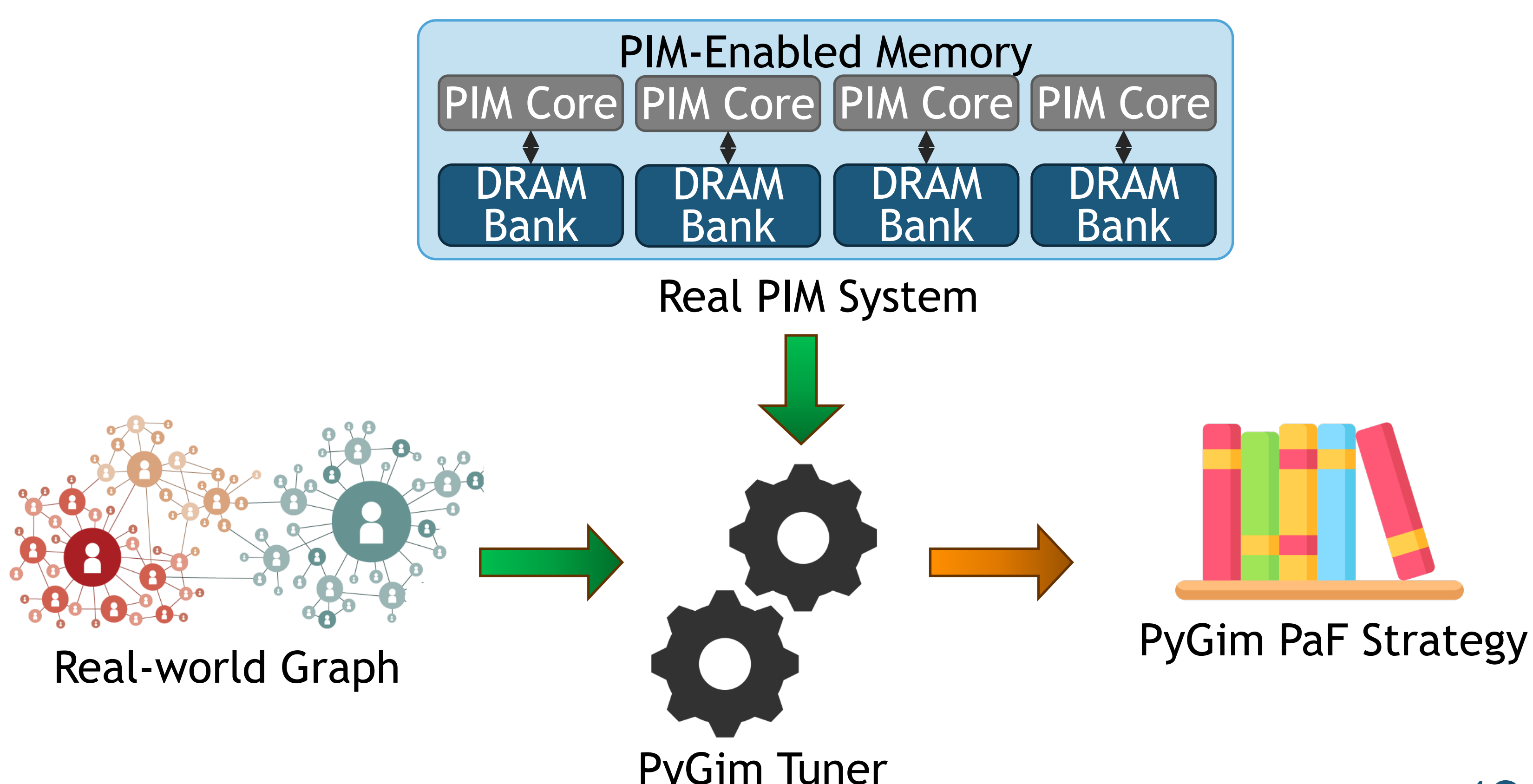




Real-world graphs have different characteristics

PyGim Tuner

Tuner selects the best-performing parallelization strategy based on:

- Real-world graph characteristics
- PIM hardware characteristics

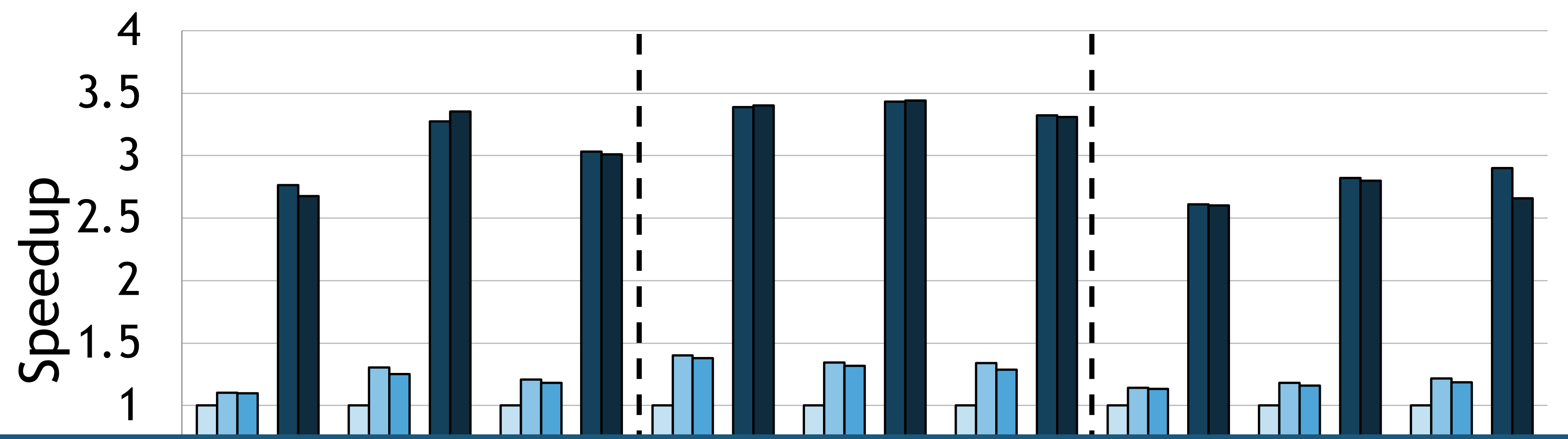


PyGim Interface

A handy Python interface (currently integrated with PyTorch)

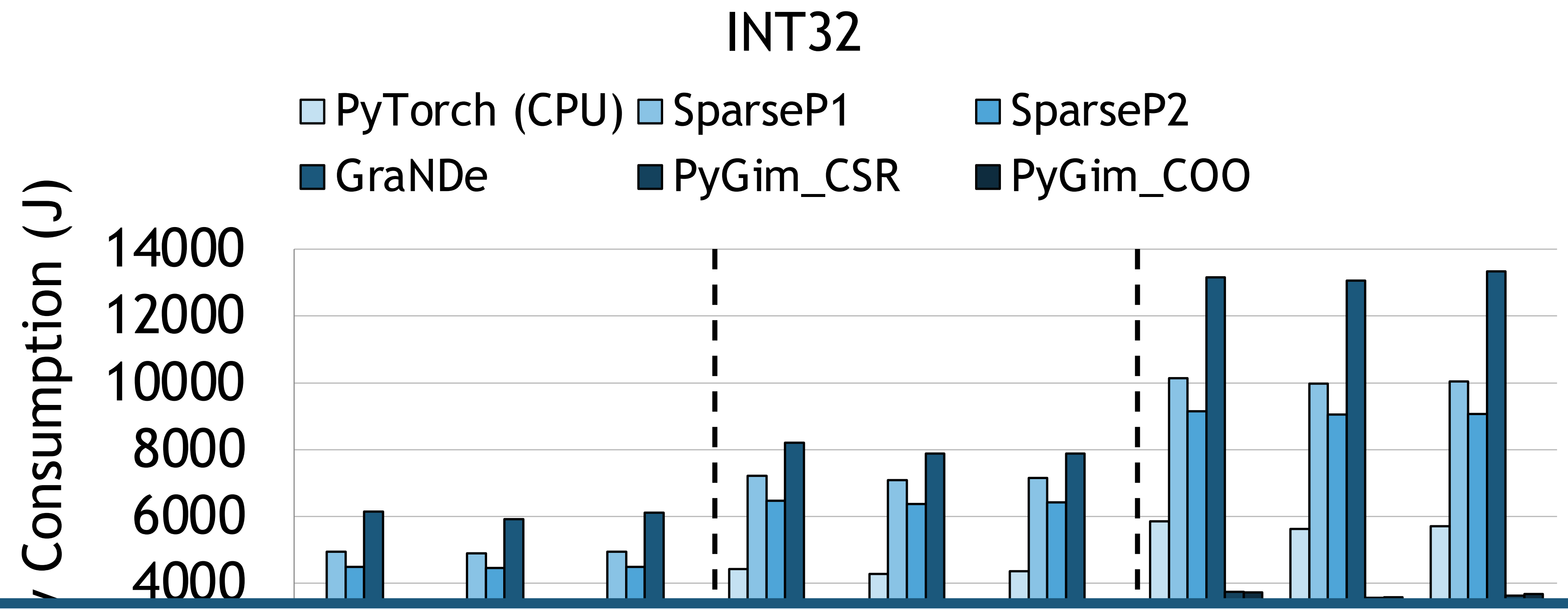
```
1  import torch, pygim as gyn
2  class GCNConv(torch.nn.Module):
3    def __init__(self, hidden_size):
4      self.linear = torch.nn.Linear(hidden_size, hidden_size)
5
6    def forward(self, graph_pim, in_dense):
7      # Execute Aggregation in PIM
8      dense_parts = col_split(in_dense)
9      out_dense = gyn.pim_run_aggr(graph_pim, dense_parts)
10     # Execute Combination in Host
11     out = self.linear(out_dense)
12     return out
13
14 gyn.pim_init_devices(num_pim_devices, groups_per_device) # Allocate PIM Devices
15 data = load_dataset() # Load graph in PIM devices
16 graph_parts, config = gyn.tune(data.graph, hidden_size, device_info)
17 graph_pim = gyn.load_graph_pim(graph_parts)
18 # Create GNN model
19 model = torch.nn.Sequential([Linear(in_channels, hidden_size),
20   GCNConv(hidden_size),
21   GCNConv(hidden_size),
22   GCNConv(hidden_size),
23   Linear(hidden_size, out_channels) ])
24 model.forward(graph_pim, data.features)
```

Performance Evaluation



PyGim outperforms PyTorch CPU and prior PIM-based schemes by 3.1x and 4.4x, respectively

Energy Efficiency Evaluation



PyGim is 2.7x and 3.3x more energy efficient than PyTorch CPU and prior PIM-based schemes

Conclusion

- Sparse computational kernels form the backbone of many important applications (HPC, machine learning, graph analytics…)
- Sparse kernels are typically 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)
- Real PIM systems typically provide specialized low-level programming interfaces and need high expertise of PIM hardware
- Our Contributions:
  - *SparseP* : first open-source SpMV library for real PIM systems
  - *PyGim* : first open-source GNN framework for real PIM systems
  - Recommendations for future PIM hardware and software

Our Work

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

PyGim Code: https://github.com/CMU-SAFARI/PyGim
PyGim Paper: https://arxiv.org/pdf/2402.16731.pdf

Systems Software and Libraries for Sparse Computational Kernels in PIM Architectures

Christina Giannoula

Tutorial on Memory-Centric Computing Systems
MICRO 2024