

# PyGim: An Efficient Graph Neural Network Library for Real Processing-In-Memory Architectures



Christina Giannoula,

Peiming Yang, Ivan Fernandez, Jiacheng Yang, Sankeerth Durvasula, Yu Xin Li, Mohammad Sadrosadati, Juan Gomez Luna, Onur Mutlu, Gennady Pekhimenko



# Executive Summary

**Problem:** The *memory-intensive* kernels of Graph Neural Networks (GNNs) dominate execution time (~91%) and are significantly *bottlenecked by memory bandwidth* in processor-centric systems (CPUs/GPUs)

**Motivation:** PIM provide significantly *high memory bandwidth* by enabling computation to be performed close to the application data

**PyGim:** An *efficient* and *easy-to-use* GNN library for real Processing-In-Memory (PIM) systems

## Key Ideas & Benefits:

- **Cost Effectiveness:** *Heterogenous* GNN kernels are executed in the *best-fit hardware*
- **High Performance:** (i) Enabling *three levels of parallelism with various strategies* in the PIM side and (ii) *adapting* best-performing parallelization strategy to the graph's *unique characteristics*
- **High Programming Ease:** (i) Providing a *handy Python API* and (ii) *automatically tuning* the best-fit parallelization strategy without programmer intervention

**Key Results:** PyGim improves (i) *performance* and *energy efficiency* by **3.7×** and **2.3×** over state-of-the-art schemes, and (ii) *core utilization* on PIM system by **11.6×** over PyTorch on GPUs

# Talk Outline

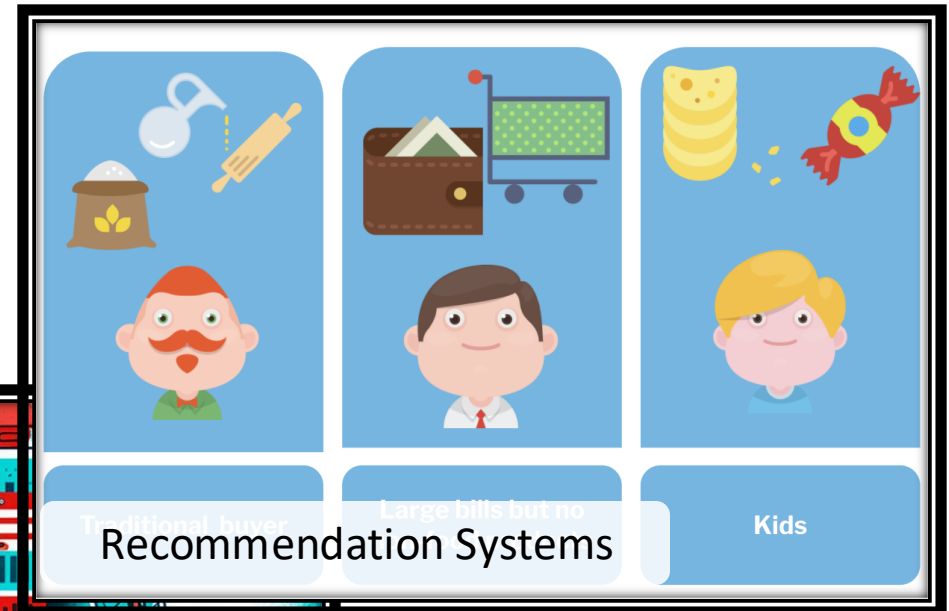
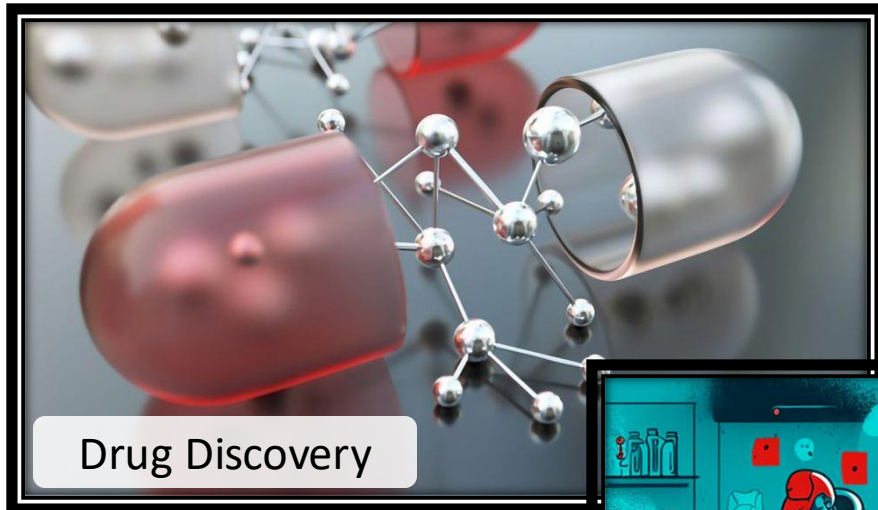
Background & Motivation

PyGim Design

Evaluation

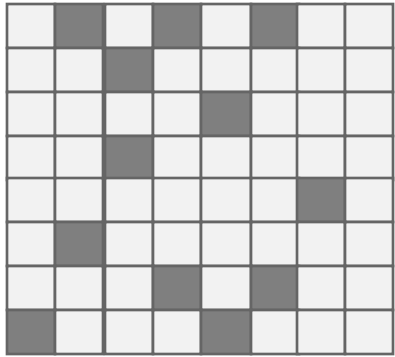
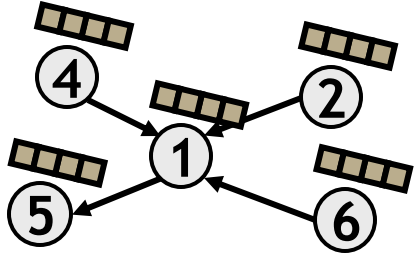
# GNNs Are Widely Used in Real-World Applications

- GNNs are state-of-the-art ML models for analyzing graph-structure data
- GNNs have a lot of applications:

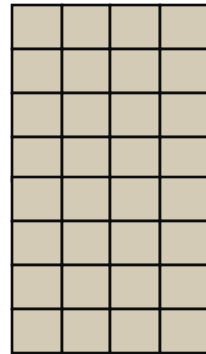


# Execution Steps of GNN Layers

- GNNs comprise a few layers (e.g., 3-5 layers)



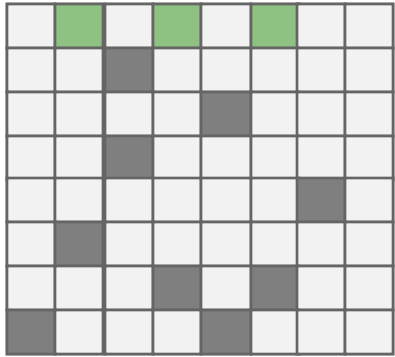
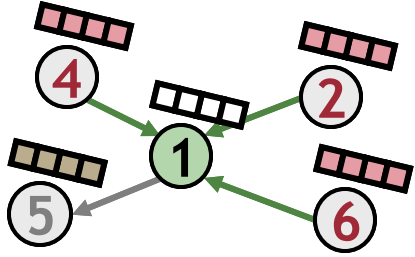
Adjacency (Sparse) Matrix



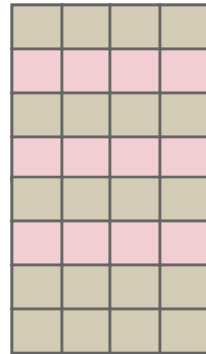
Input Feature

# Execution Steps of GNN Layers

- GNNs comprise a few layers (e.g., 3-5 layers)



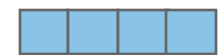
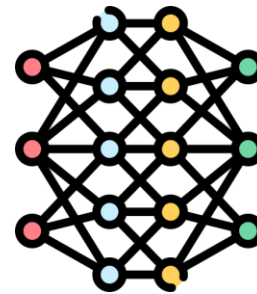
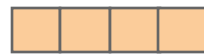
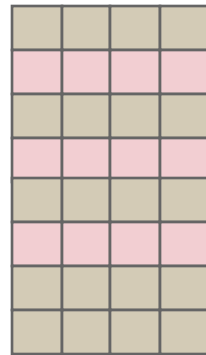
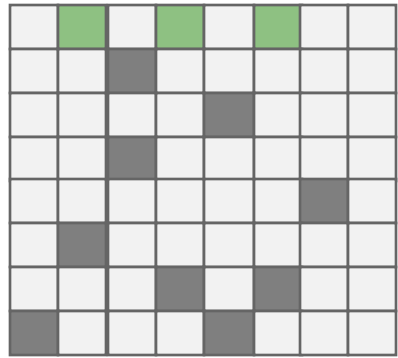
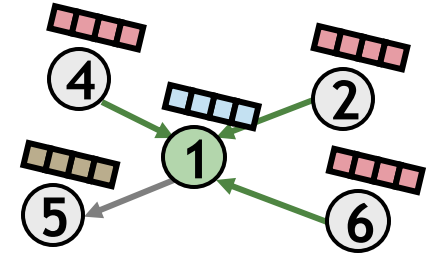
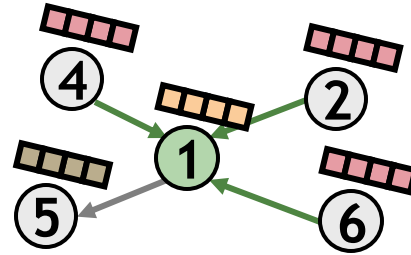
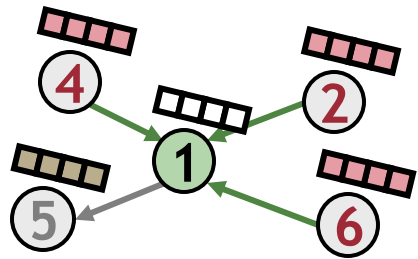
Adjacency (Sparse) Matrix



Input Feature

# Execution Steps of GNN Layers

- GNNs comprise a few layers (e.g., 3-5 layers)



Adjacency (Sparse) Matrix

Input Feature

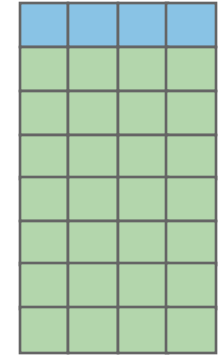
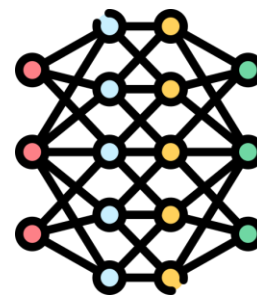
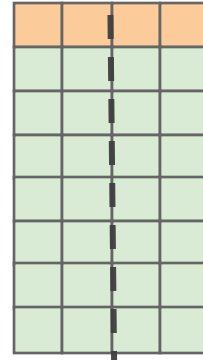
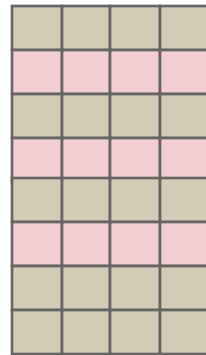
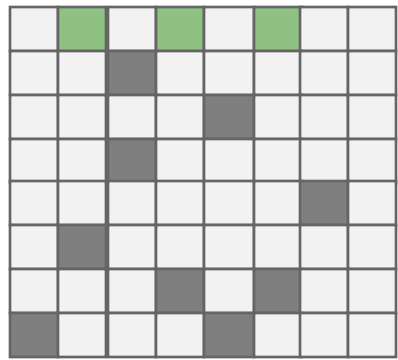
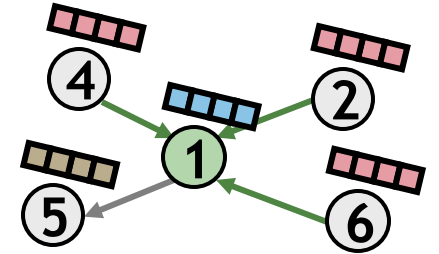
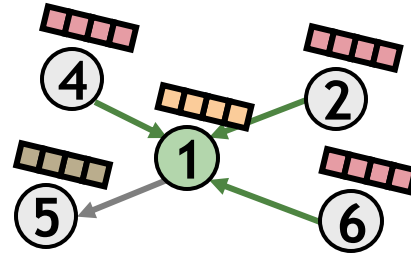
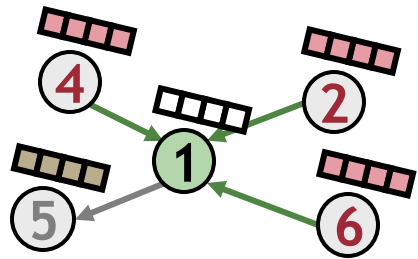
Aggregation Result

Neural Network (NN)

Output Feature

# Execution Steps of GNN Layers

- GNNs comprise a few layers (e.g., 3-5 layers)
- Each GNN layer has *two* execution steps:



Adjacency (Sparse) Matrix

Input Feature

Aggregation Result

Neural Network (NN)

Output Feature

Aggregation

Combination

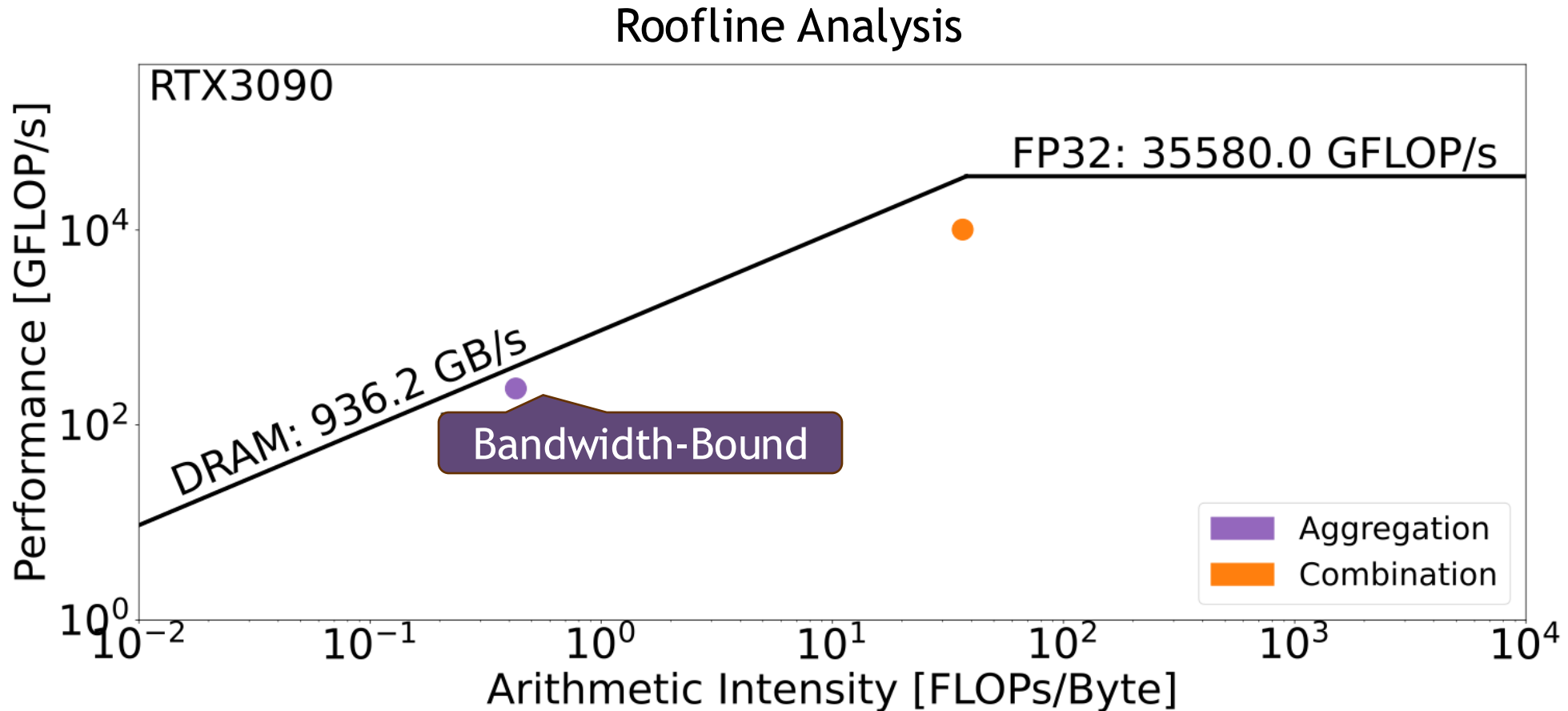
Aggregate neighbor's feature, corresponds to Sparse Matrix Matrix Multiplication (*SpMM*)

Combine features by NN. Typically comprises compute-intensive kernels (e.g., GEMMs)

# GNN Aggregation Is Memory-Bandwidth-Bound In GPUs

Using a RTX 3090 GPU with ~900 GB/s bandwidth, we find that GNN Aggregation

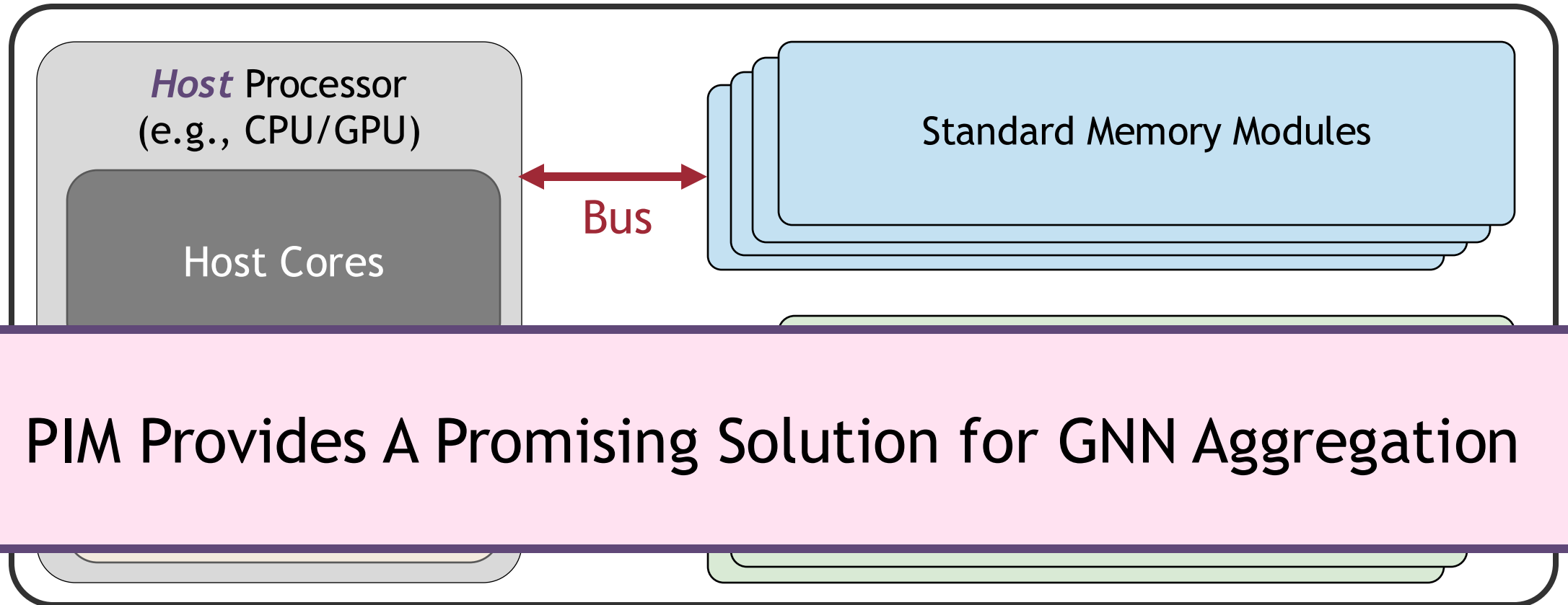
- takes **~91%** of the inference time
- achieves **less than 2%** core utilization



# PIM Alleviates The Data Movement Bottleneck

- Near-bank PIM: each PIM core is tightly coupled with one (or a few) DRAM banks
- Near-bank PIM cores have **significantly higher memory bandwidth** than Host cores

A Near-Bank PIM System



# Talk Outline

Background & Motivation

PyGim Design

Evaluation

# PyGim Overview

- An **efficient** and **easy-to-use** GNN library for real PIM systems
- PyGim incorporates 4 **key** components:

Cooperative Acceleration (CoA)

Run *heterogeneous* kernels in the best-fit hardware

Parallelism Fusion (PaF)

Strives a balance between *computation* and *data transfer*

Lightweight Tuning

*Automatically* tunes the *best-performing* PaF strategy

Handy Programming Interface

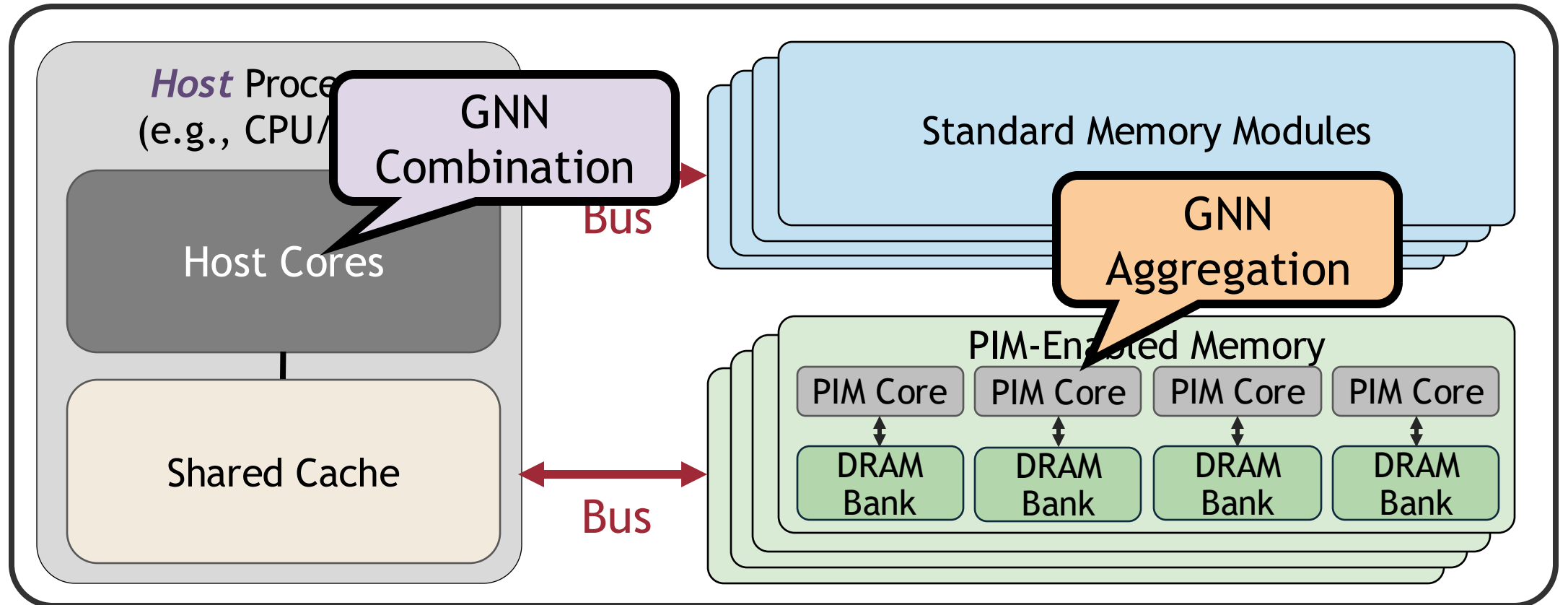
Integrates a *handy* Python (PyTorch) API

# 1. Cooperative Acceleration (CoA)

Heterogeneous kernels are running in the best-fit underlying hardware

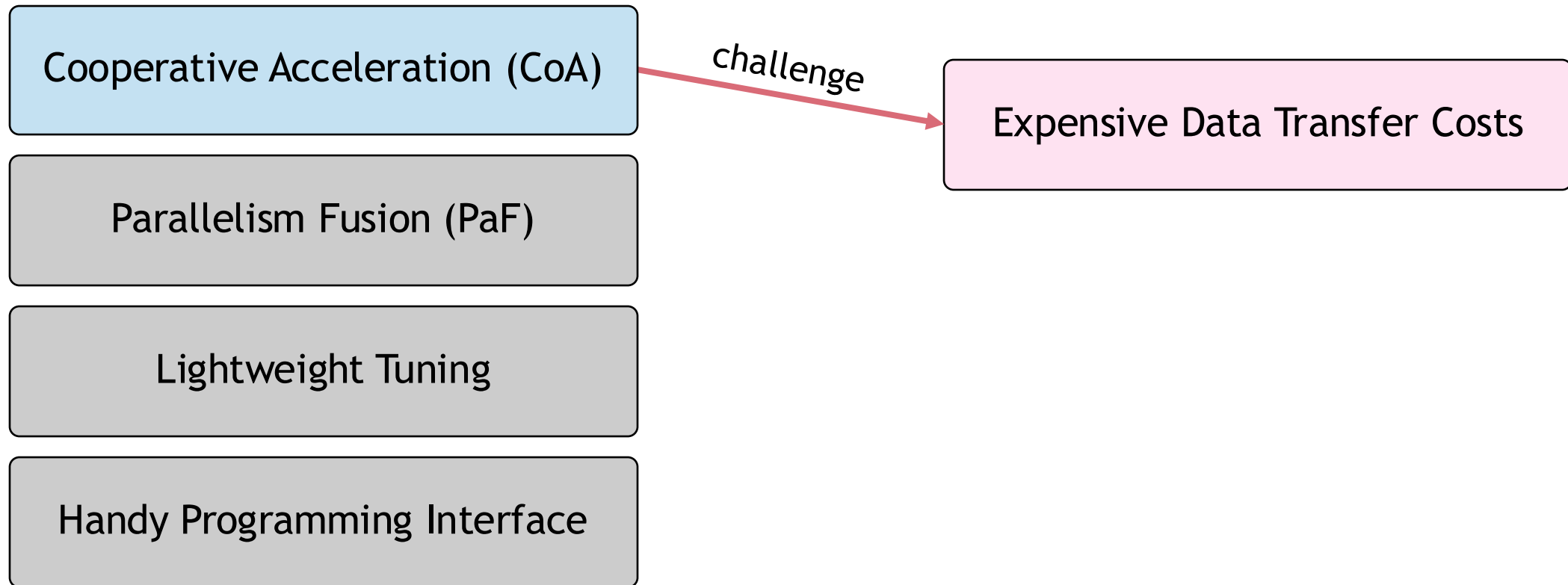
- **Combination** runs on **Host** cores
- **Aggregation** runs on **PIM** cores

A Near-Bank PIM System



# PyGim Overview

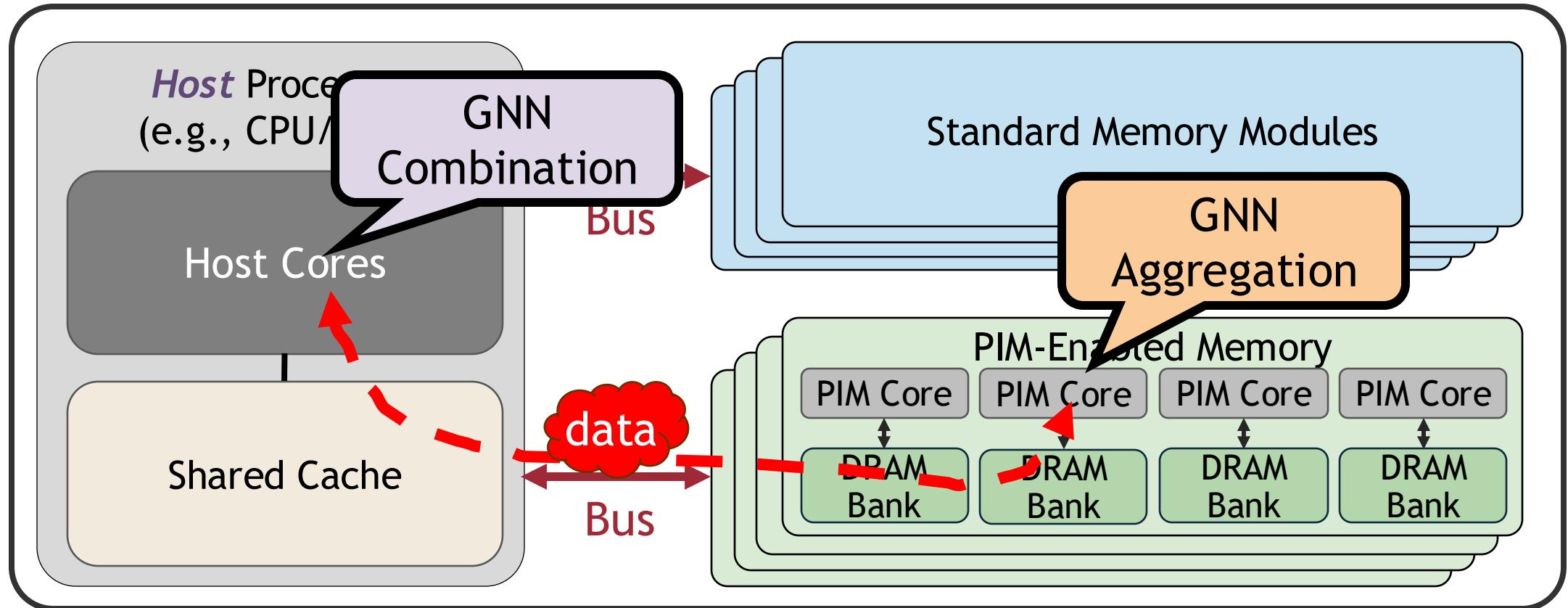
- An **efficient** and **easy-to-use** GNN library for real PIM systems
- PyGim incorporates 4 **key** components:



# Challenge 1: Expensive Data Transfer Costs

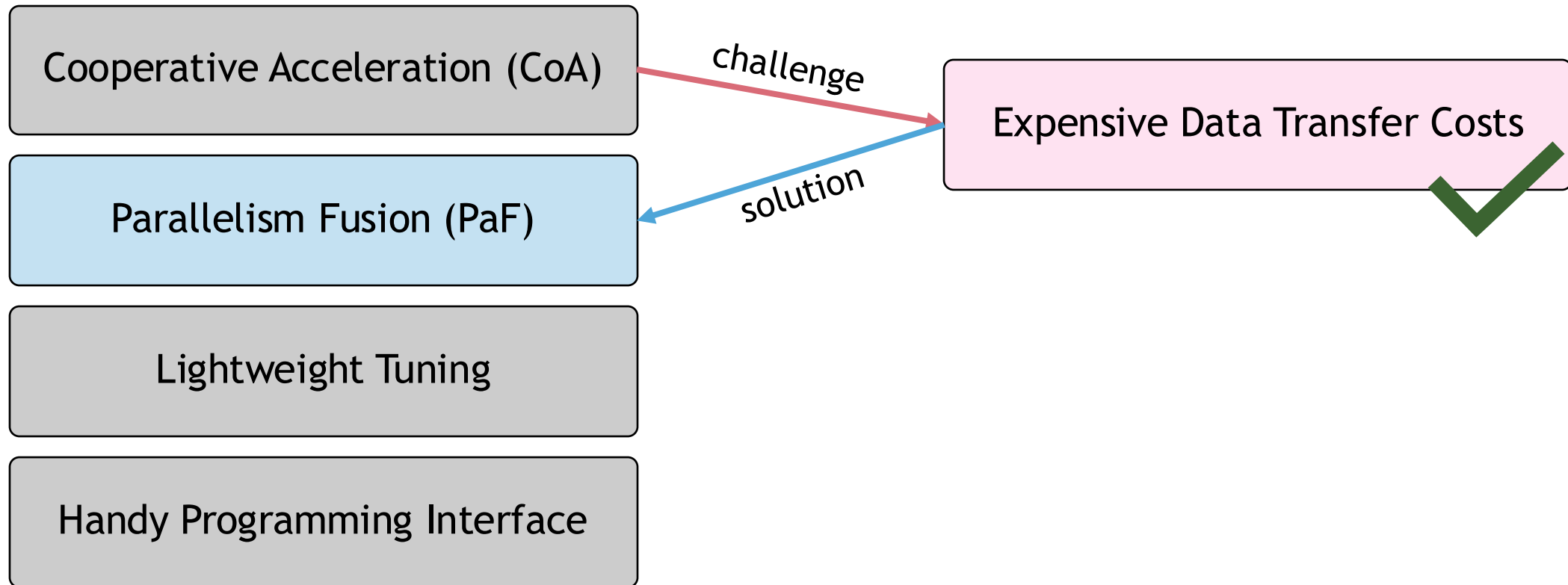
- Alleviate the overheads of passing the output data of the one step as input data to the next step

A Near-Bank PIM System



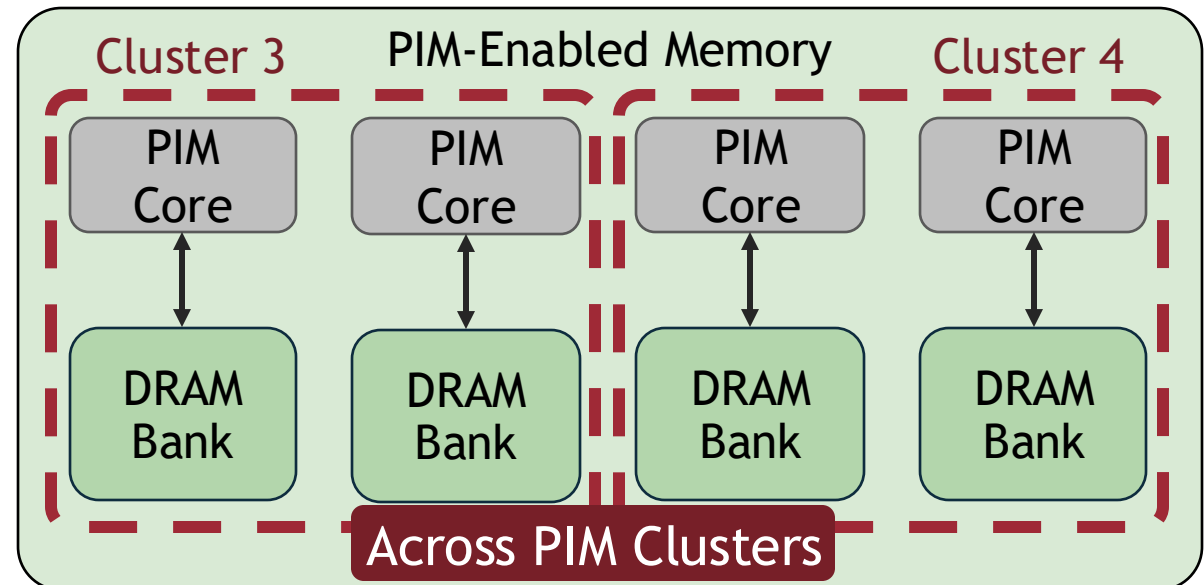
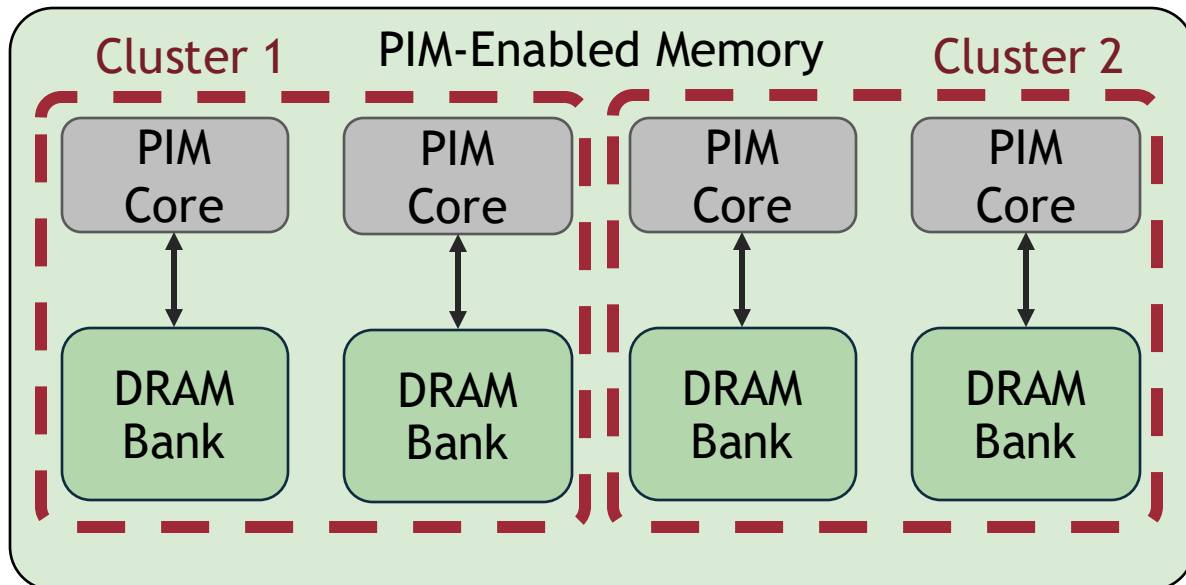
# PyGim Overview

- An **efficient** and **easy-to-use** GNN library for real PIM systems
- PyGim incorporates 4 **key** components:



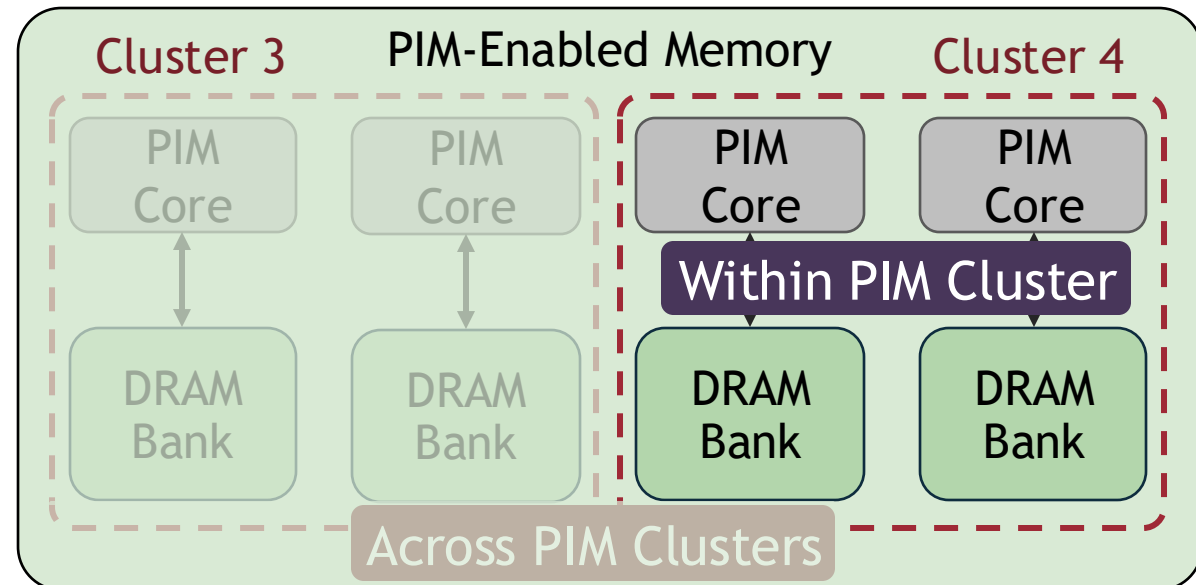
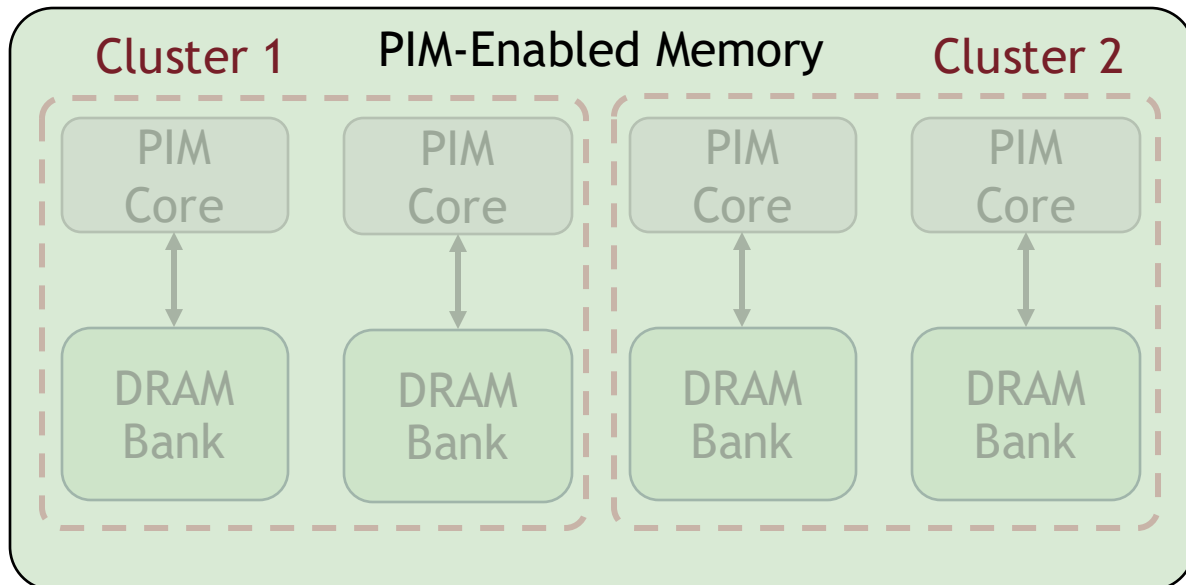
## 2. Parallelism Fusion (PaF)

- PaF (i) strives a balance between *computation* and *data transfer* costs and (ii) efficiently covers various real-world *graphs* that exhibit *diverse characteristics*
- PaF enablers 3 levels of parallelism:
  1. *Across PIM Clusters*: Edge- + Feature-level parallelism



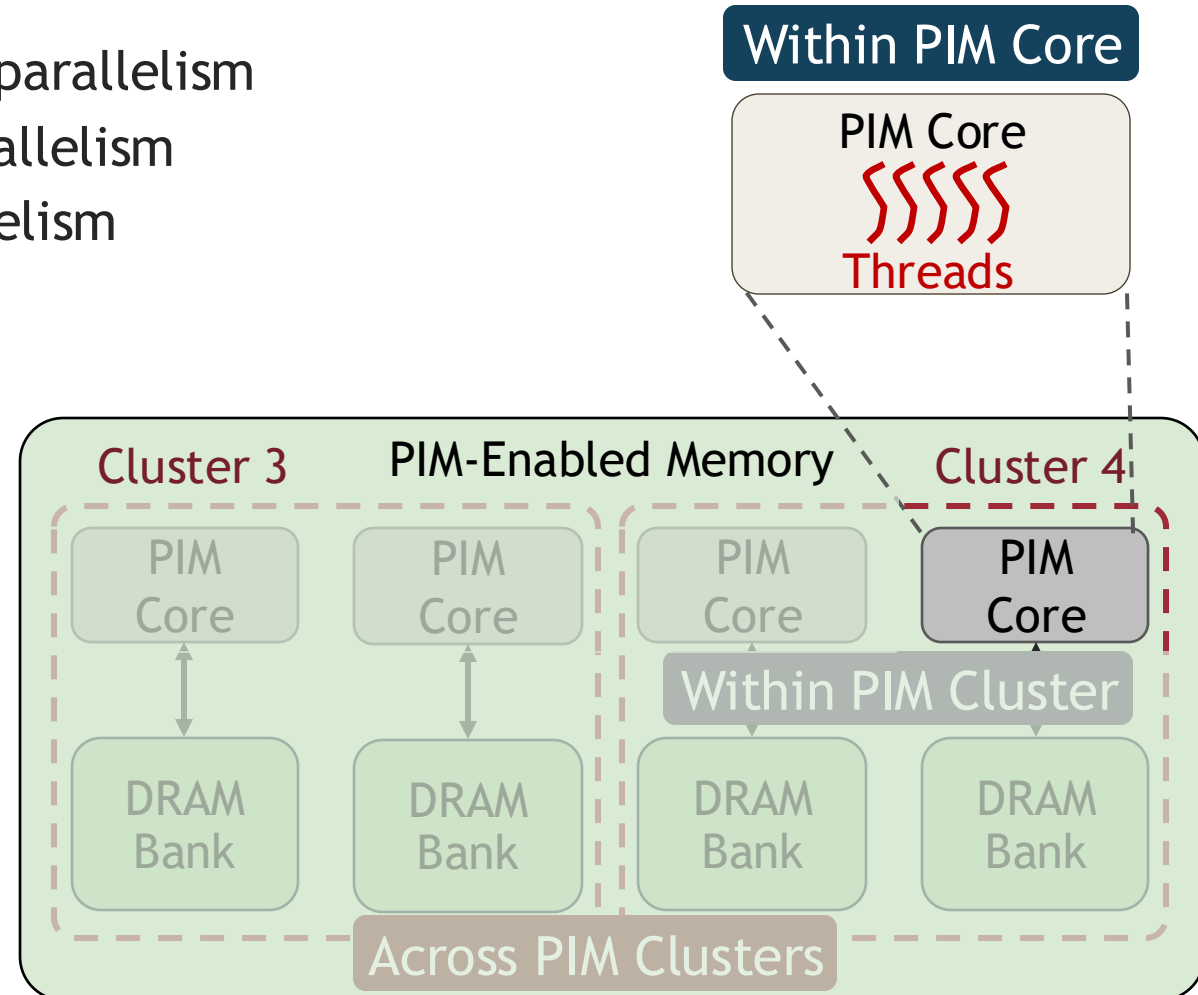
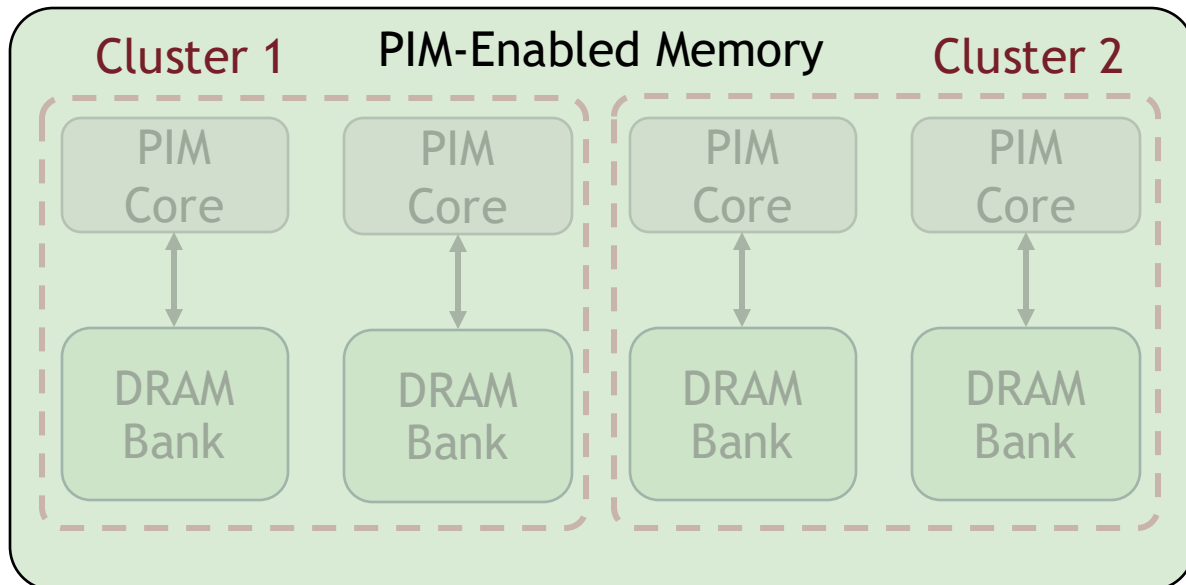
## 2. Parallelism Fusion (PaF)

- PaF (i) strives a balance between *computation* and *data transfer* costs and (ii) efficiently covers various real-world *graphs* that exhibit *diverse characteristics*
- PaF enablers **3 levels** of parallelism:
  1. **Across PIM Clusters**: Edge- + Feature-level parallelism
  2. **Within PIM Cluster**: Vertex-/Edge-level parallelism



## 2. Parallelism Fusion (PaF)

- PaF (i) strives a balance between *computation* and *data transfer* costs and (ii) efficiently covers various real-world *graphs* that exhibit *diverse characteristics*
- PaF enablers 3 levels of parallelism:
  1. **Across PIM Clusters**: Edge- + Feature-level parallelism
  2. **Within PIM Cluster**: Vertex-/Edge-level parallelism
  3. **Within PIM Core**: Vertex-/Edge-level parallelism



## 2. Parallelism Fusion (PaF)

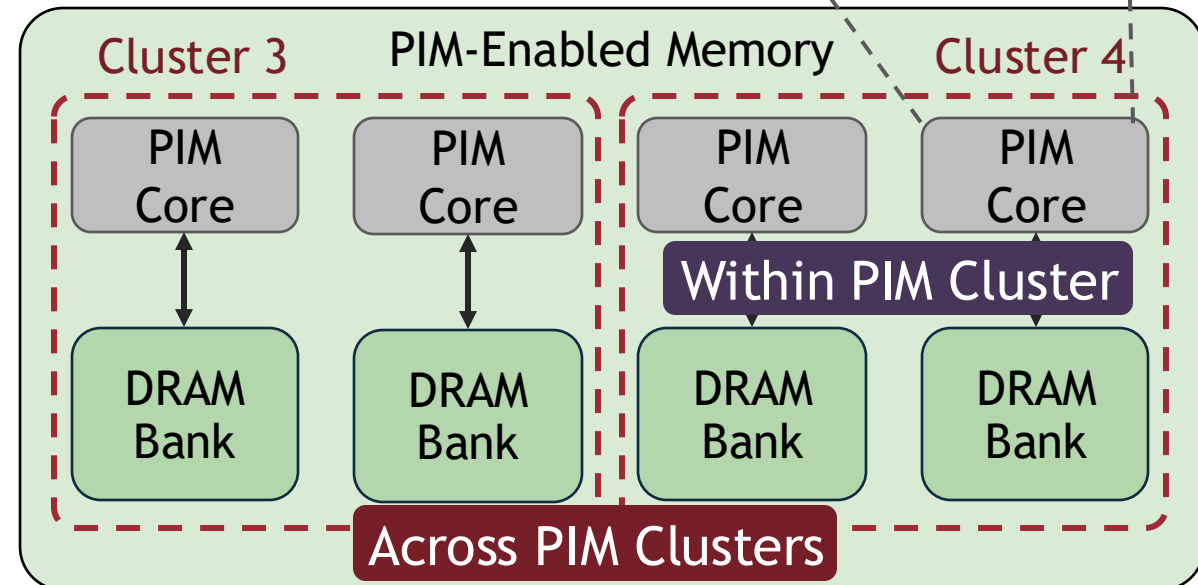
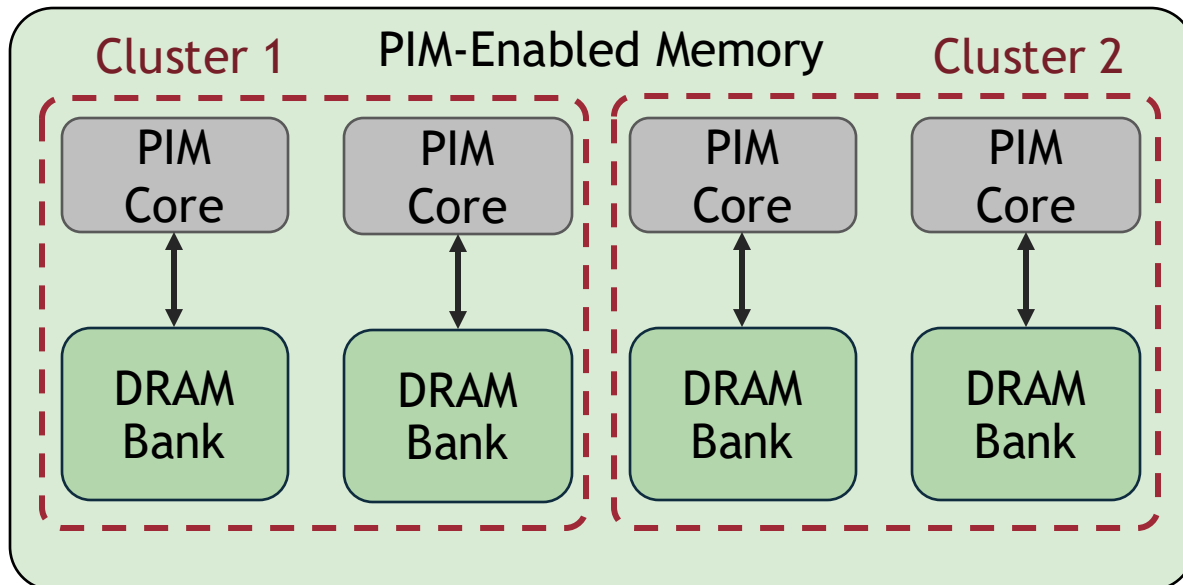
- PaF (i) strives a balance between *computation* and *data transfer* costs and (ii) efficiently covers various real-world *graphs* that exhibit *diverse characteristics*
- PaF enablers **3 levels** of parallelism: Reduces data transfer costs

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

Reduce computation costs

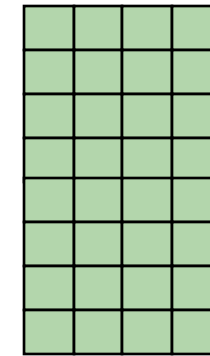
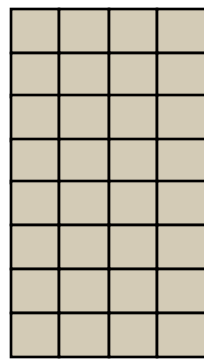
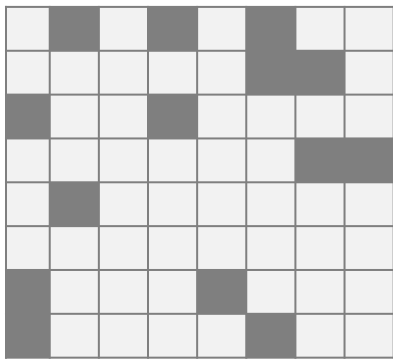
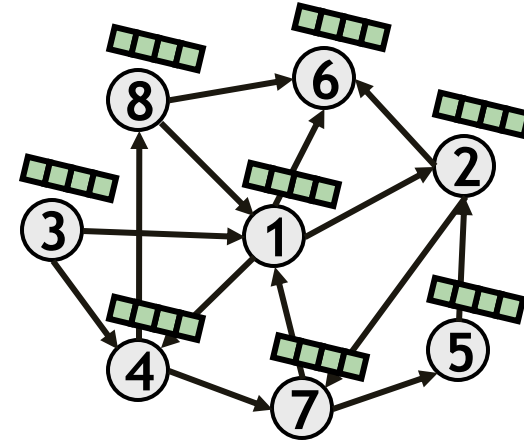
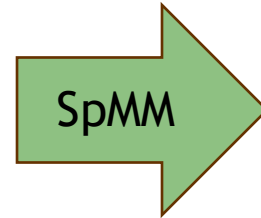
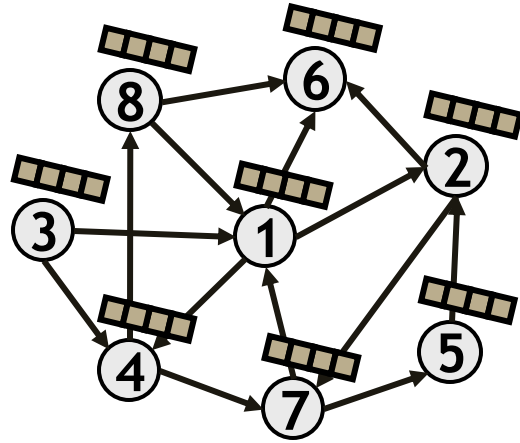
Within PIM Core

PIM Core  
  
Threads



# An Aggregation Example

- E.g., a graph with 8 vertices and 14 edges
  - SpMM is used for aggregation



Adjacency (Sparse) Matrix

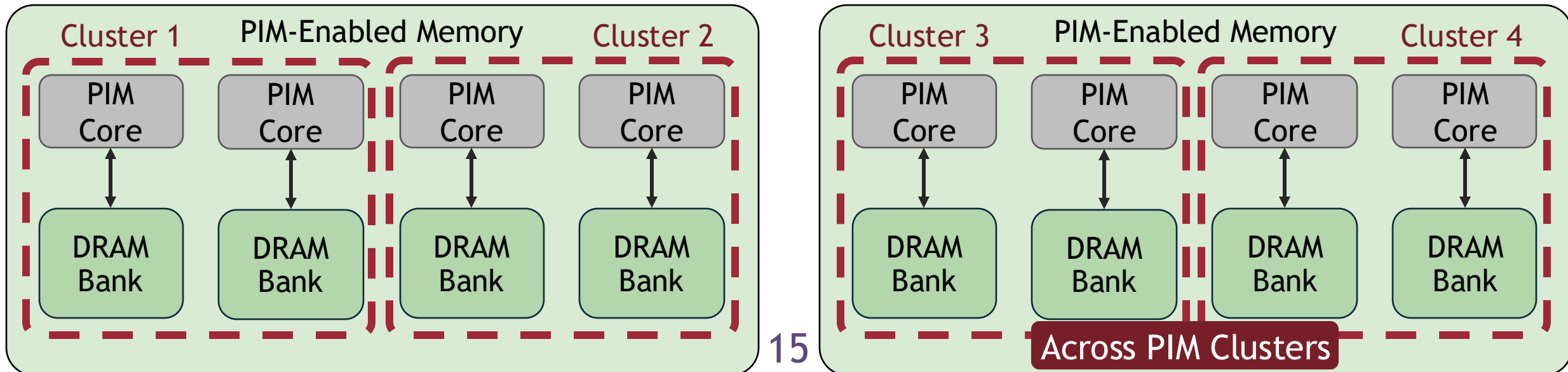
Input Feature Matrix

Aggregation Results

## 2.1 PaF Parallelism Across PIM Clusters

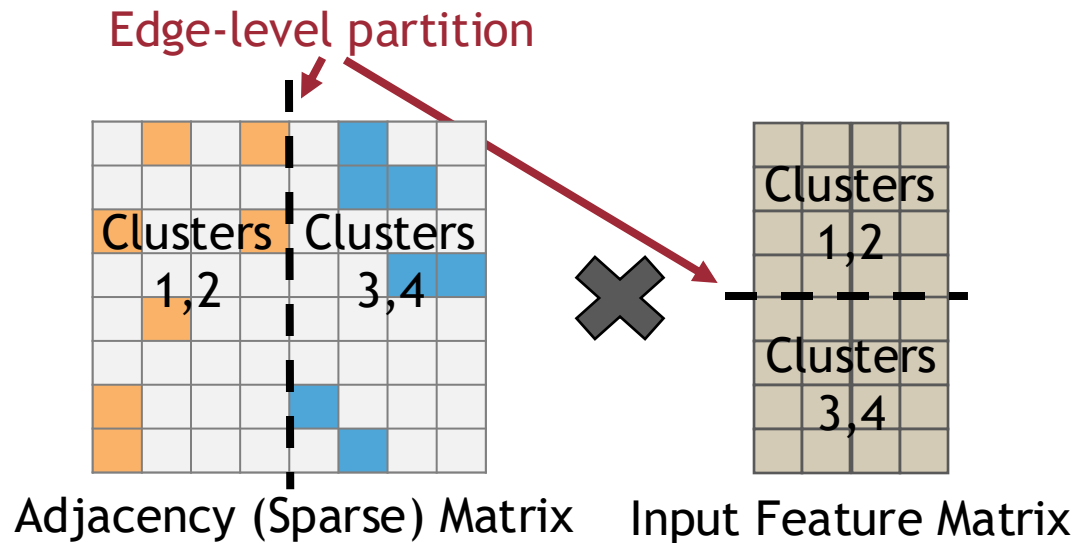
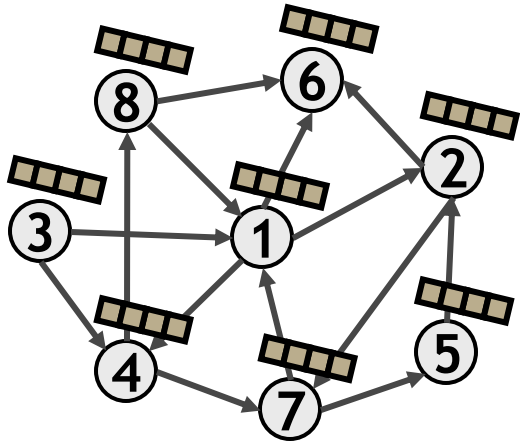
- PaF (i) strives a balance between *computation* and *data transfer* costs and (ii) covers various *graphs* with *diverse characteristics*
- PaF enables 3 levels of parallelism:
  1. Across PIM Clusters: Edge- and Feature-level parallelism
  2. Within PIM Cluster: Vertex- or Edge-level parallelism
  3. Within PIM Core: Vertex- or Edge-level parallelism

### 4 PIM Clusters



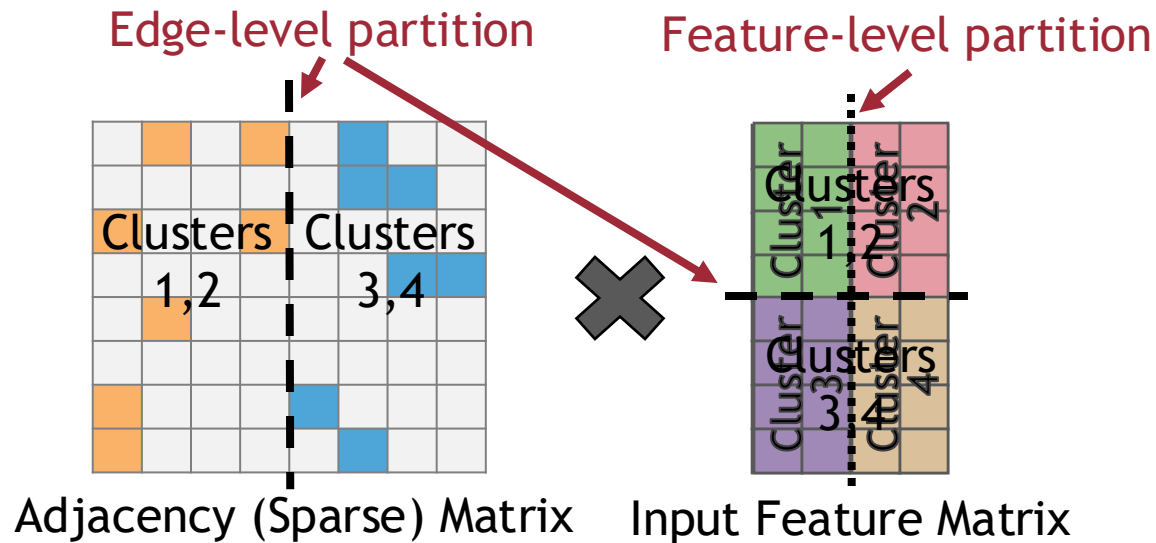
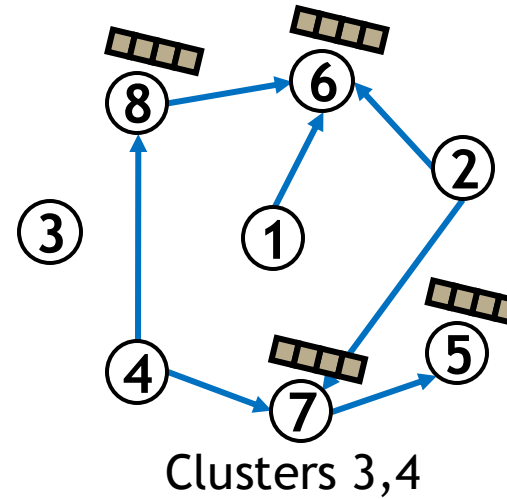
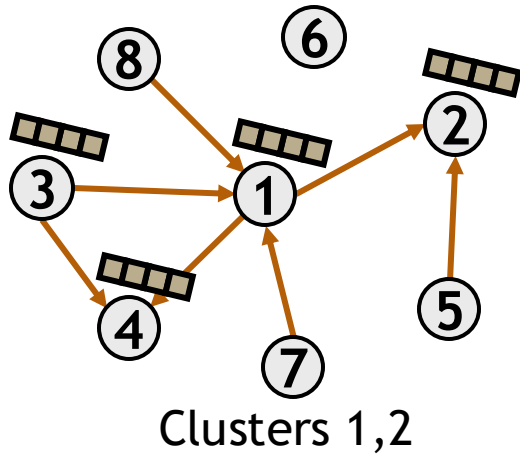
# Across PIM Clusters: Edge- & Feature-Level Parallelism

- E.g., creating 4 PIM clusters with 2 Edge partitions and 2 Feature partitions



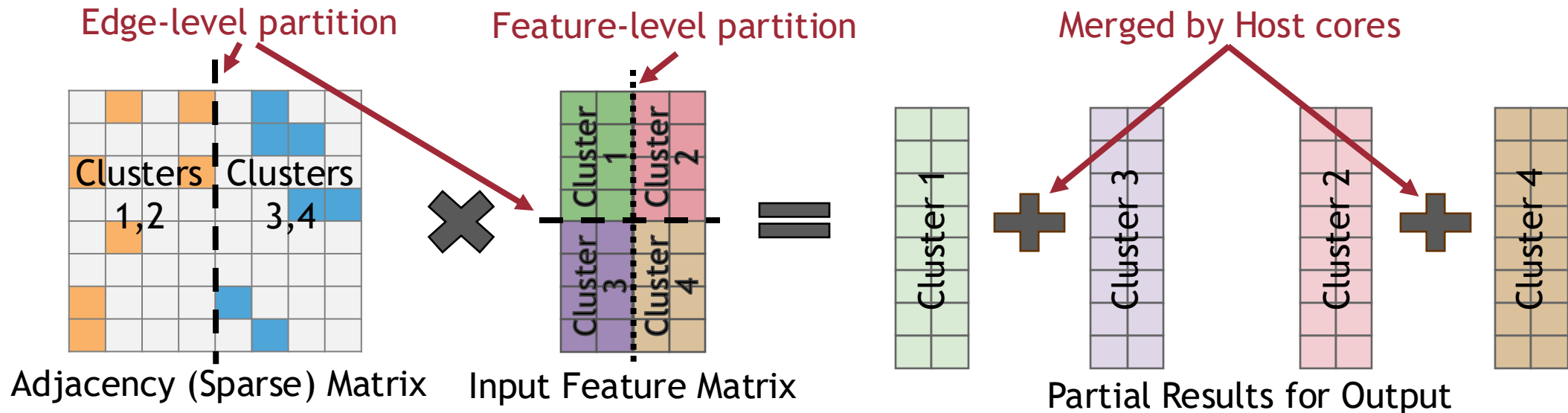
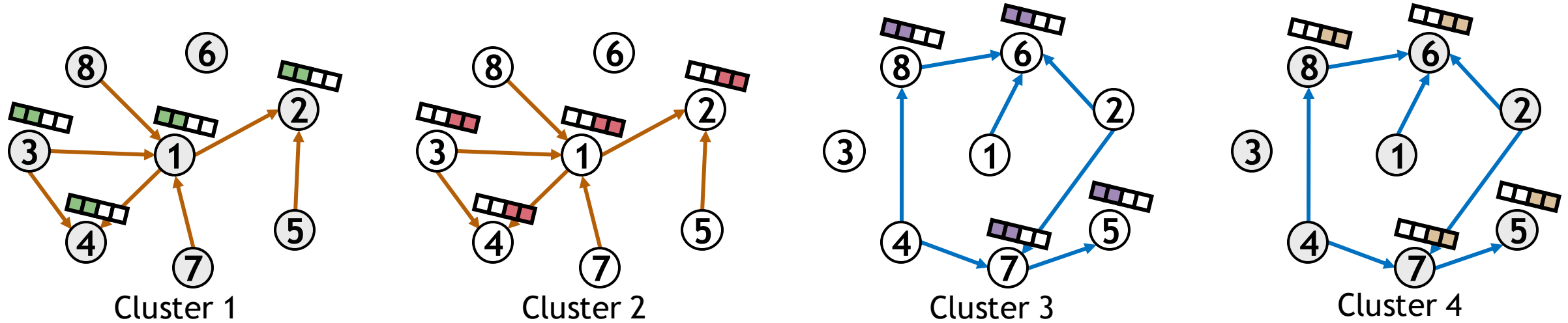
# Across PIM Clusters: Edge- & Feature-Level Parallelism

- E.g., creating 4 PIM clusters with 2 Edge partitions and 2 Feature partitions



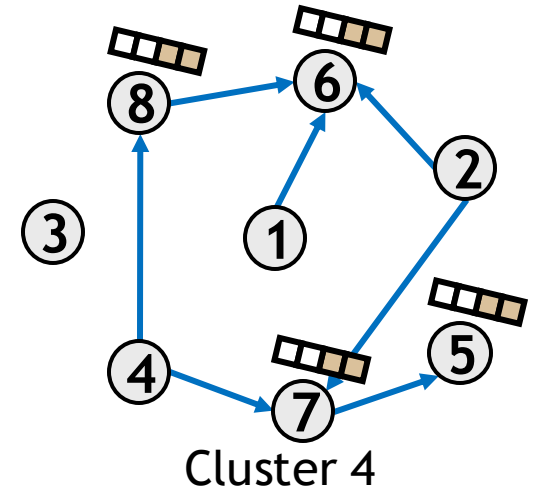
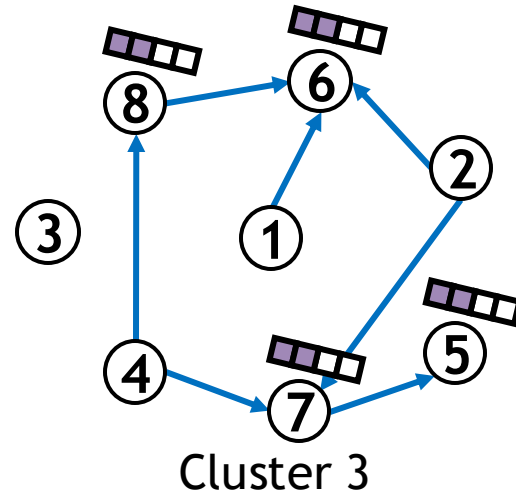
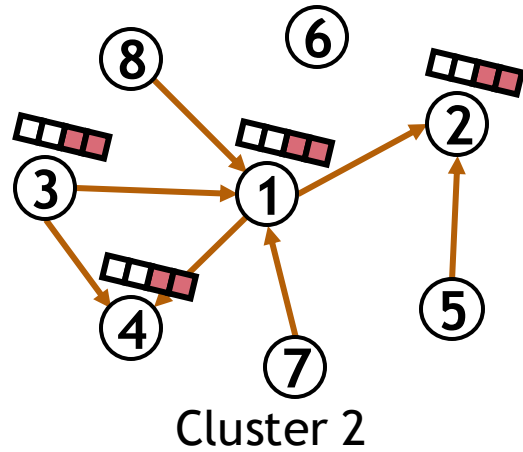
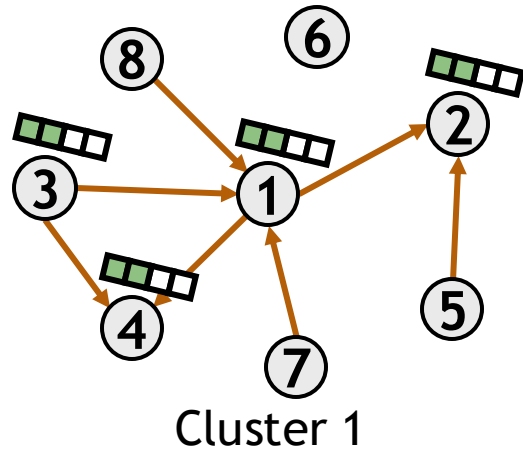
# Across PIM Clusters: Edge- & Feature-Level Parallelism

- E.g., creating 4 PIM clusters with 2 Edge partitions and 2 Feature partitions



# Across PIM Clusters: Edge- & Feature-Level Parallelism

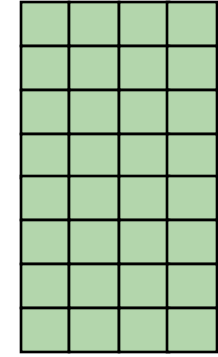
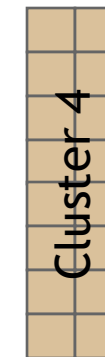
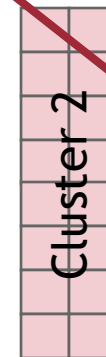
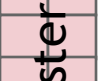
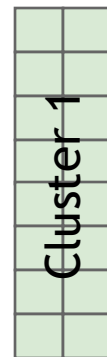
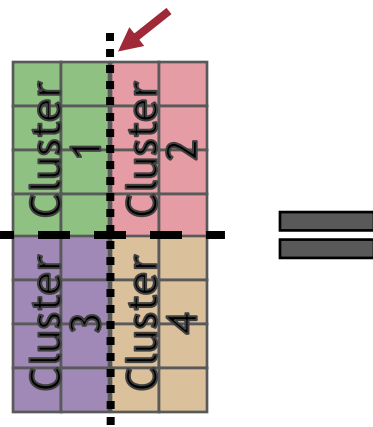
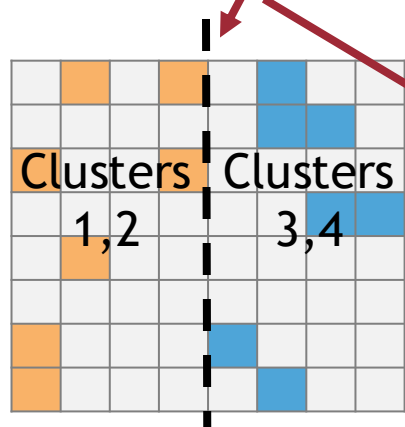
- E.g., creating 4 PIM clusters with 2 Edge partitions and 2 Feature partitions



Edge-level partition

Feature-level partition

Merged by Host cores



Adjacency (Sparse) Matrix

Input Feature Matrix

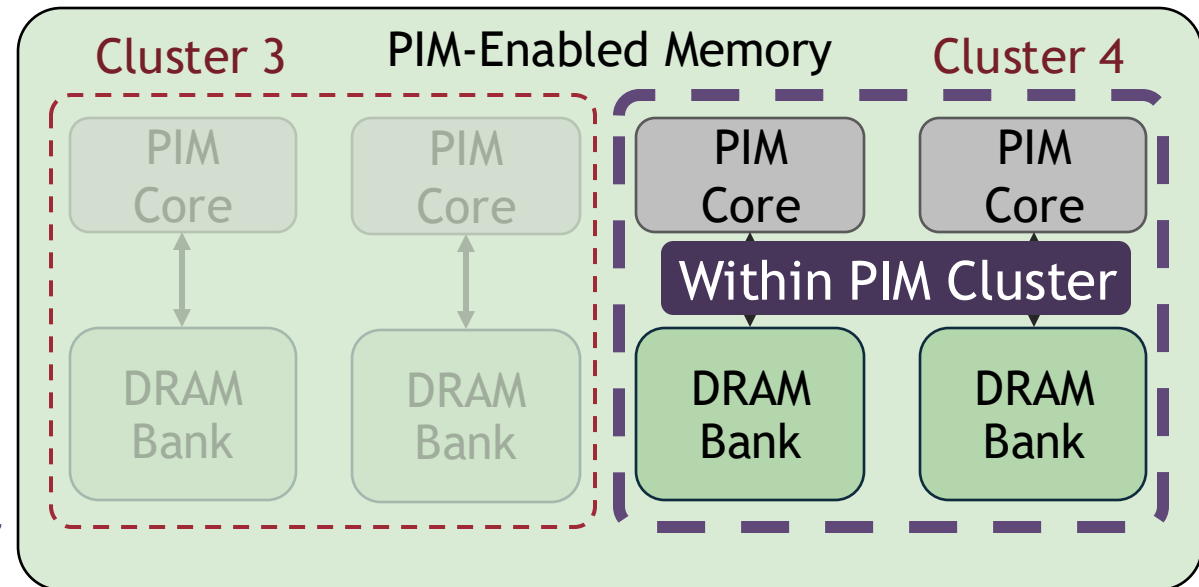
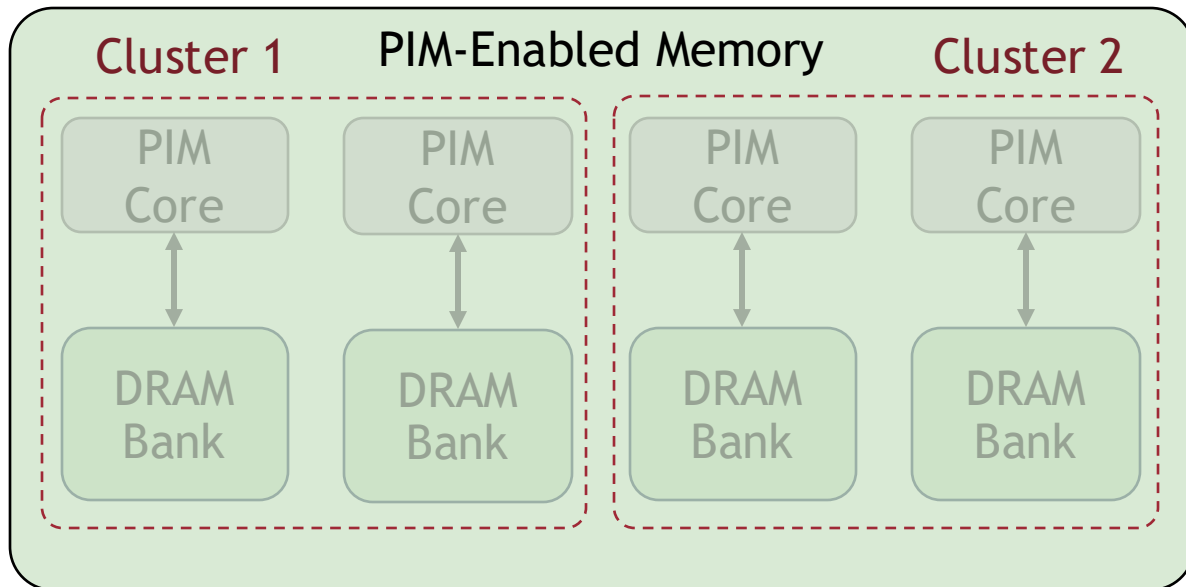
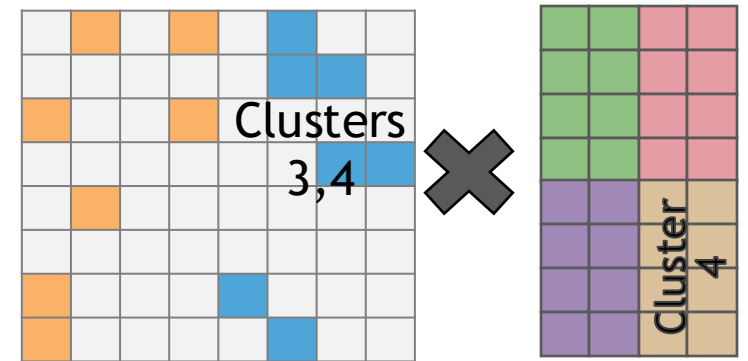
Partial Results for Output

Aggregation Results 16

## 2.2 PaF Parallelism Within PIM Cluster

- PaF (i) strives a balance between *computation* and *data transfer* costs and (ii) covers various *graphs* with *diverse characteristics*
- PaF enables 3 levels of parallelism:

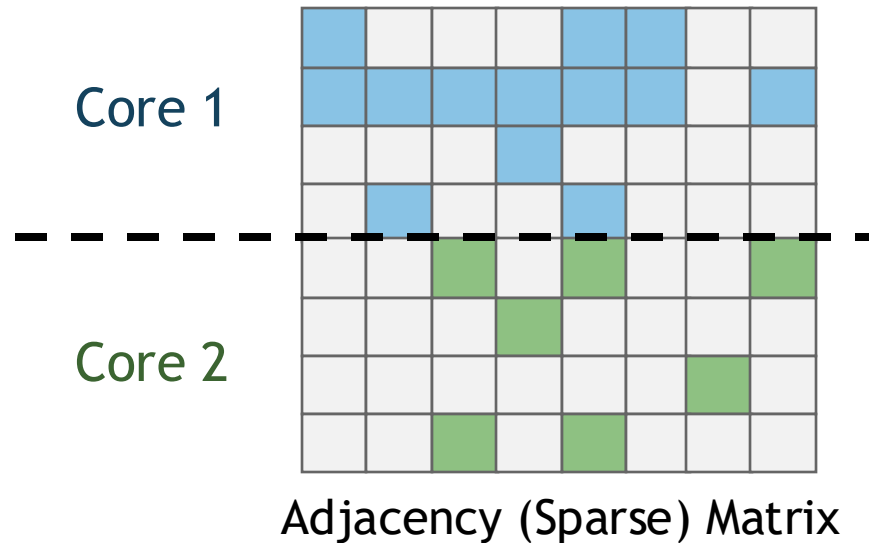
1. Across PIM Clusters: Edge- and Feature-level parallelism
2. Within PIM Cluster: Vertex- or Edge-level parallelism
3. Within PIM Core: Vertex- or Edge-level parallelism



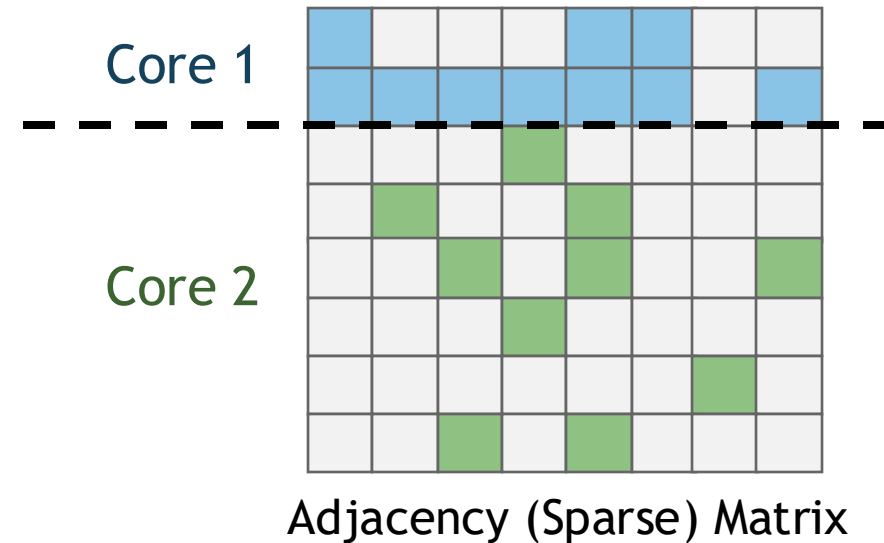
# Within a PIM Cluster: Vertex- or Edge-Level Parallelism

- E.g., balancing vertices or balancing edges across PIM cores within the cluster

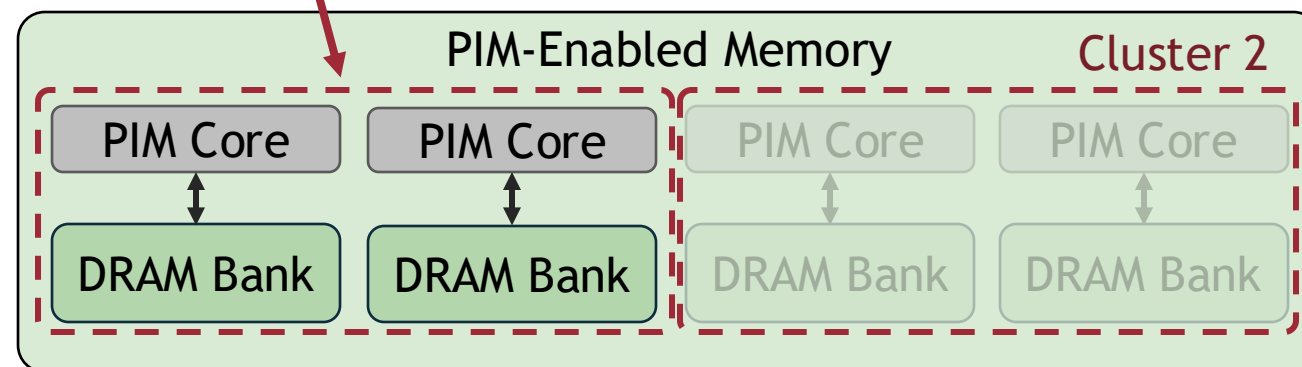
Balance Vertices Across PIM Cores



Balance Edges Across PIM Cores



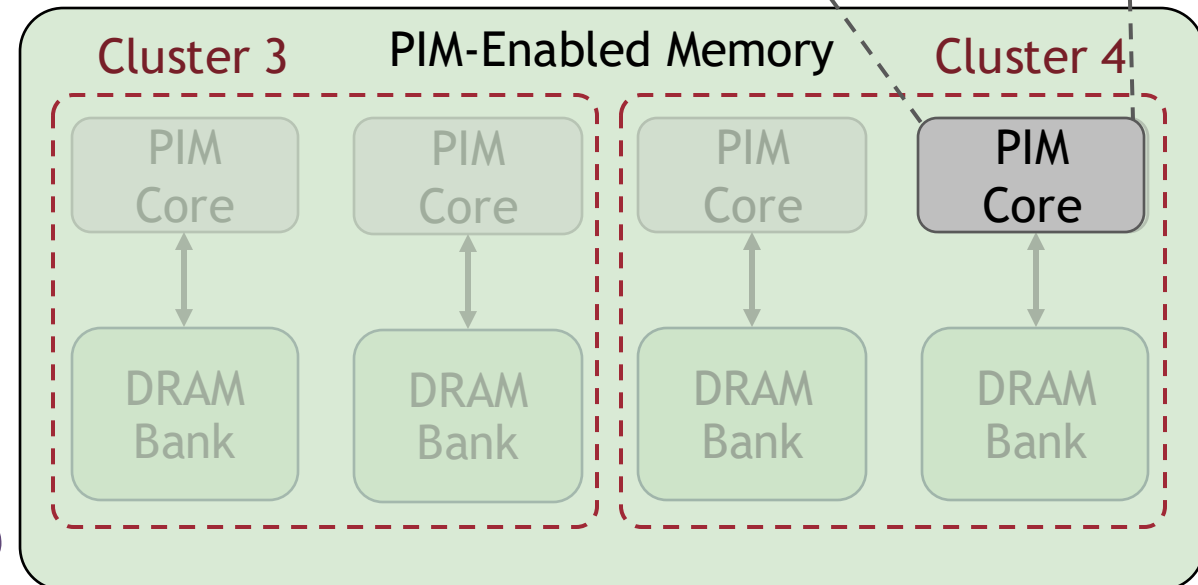
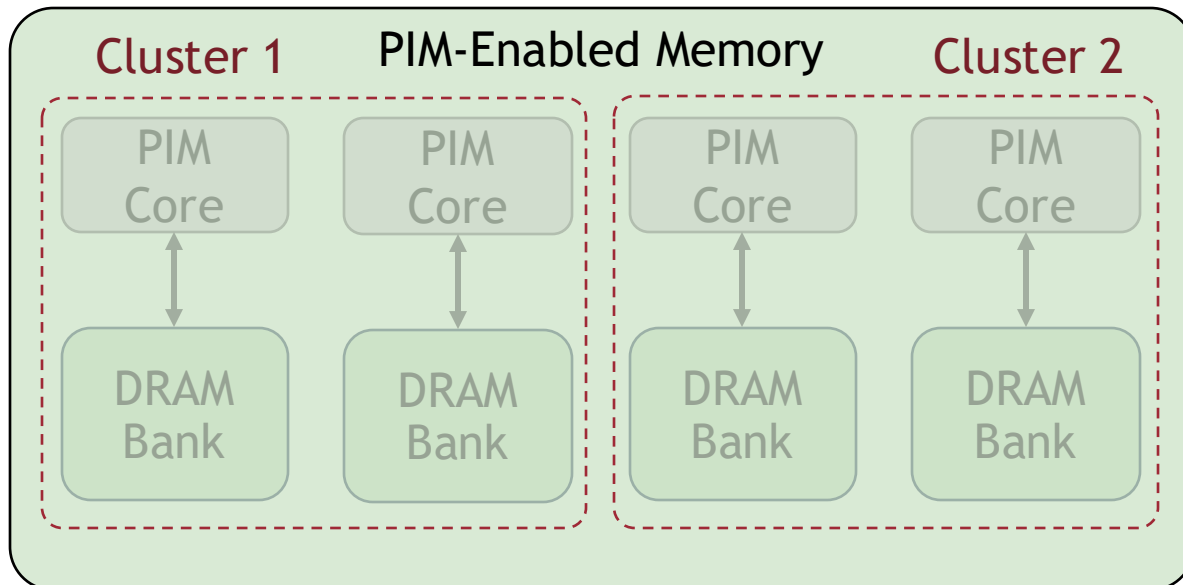
Cluster 1 has 2 PIM Cores



## 2.3 PaF Parallelism Within PIM Core

- PaF (i) strives a balance between *computation* and *data transfer* costs and (ii) covers various *graphs* with *diverse characteristics*
- PaF enables 3 levels of parallelism:

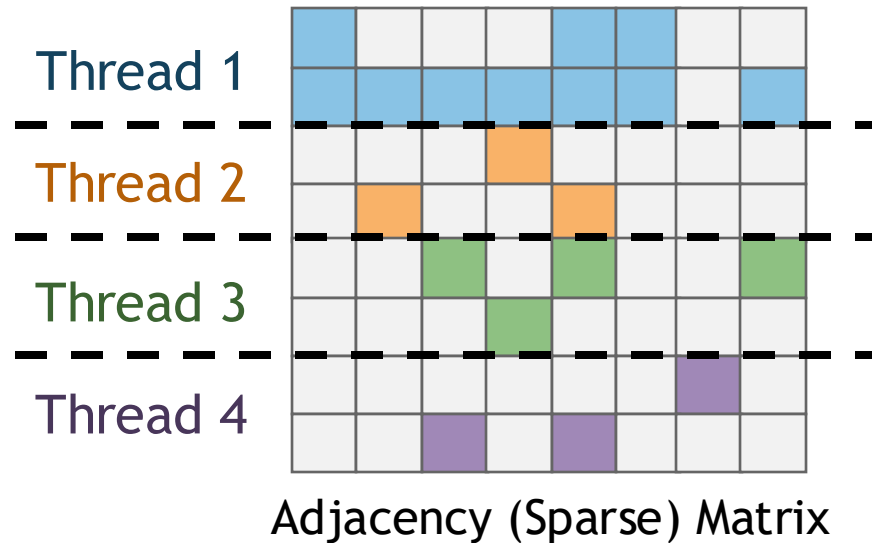
1. Across PIM Clusters: Edge- and Feature-level parallelism
2. Within PIM Cluster: Vertex- or Edge-level parallelism
3. Within PIM Core: Vertex- or Edge-level parallelism



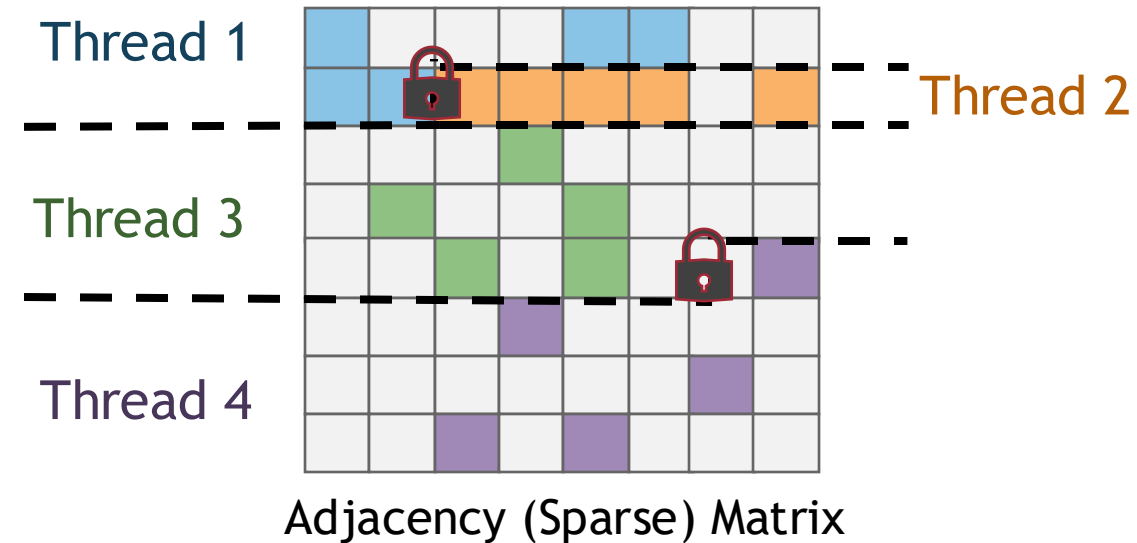
# Within a PIM Core: Vertex- or Edge-Level Parallelism

- E.g., balancing vertices or balancing edges across threads within a PIM core

Balance Vertices Across Threads



Balance Edges Across Threads



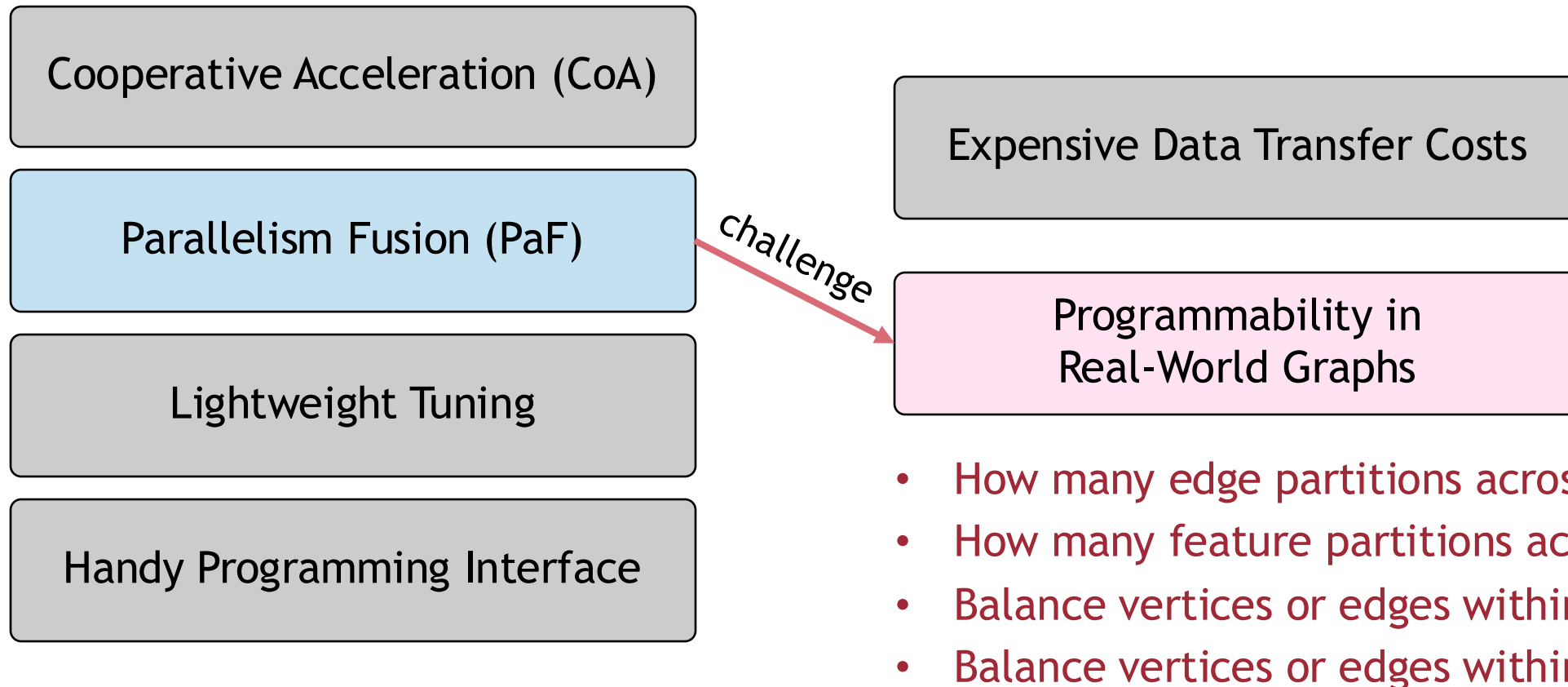
PIM Core supports  
4 threads



Synchronization is implement  
with lock-free or fine-grained locking schemes

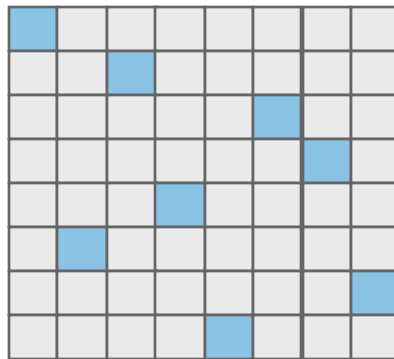
# PyGim Overview

- An **efficient** and **easy-to-use** GNN library for real PIM systems
- PyGim incorporates 4 **key** components:

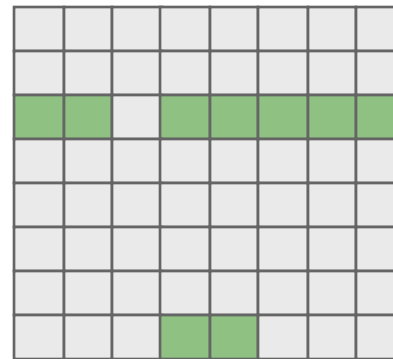


# Challenge 2: Programmability in Real-World Graphs

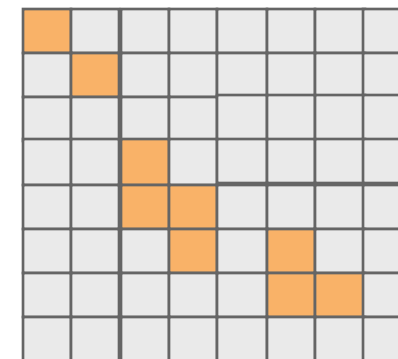
- PaF supports a wide variety of parallelization strategies:
  - Typically there is **no one-size-fits-all** solution
- Challenge = **manually tuning** the parallelization **strategy** poses **significant burdens** for developers
  - *Unique* graph's characteristics need different tuning



regular graph



power-law graph

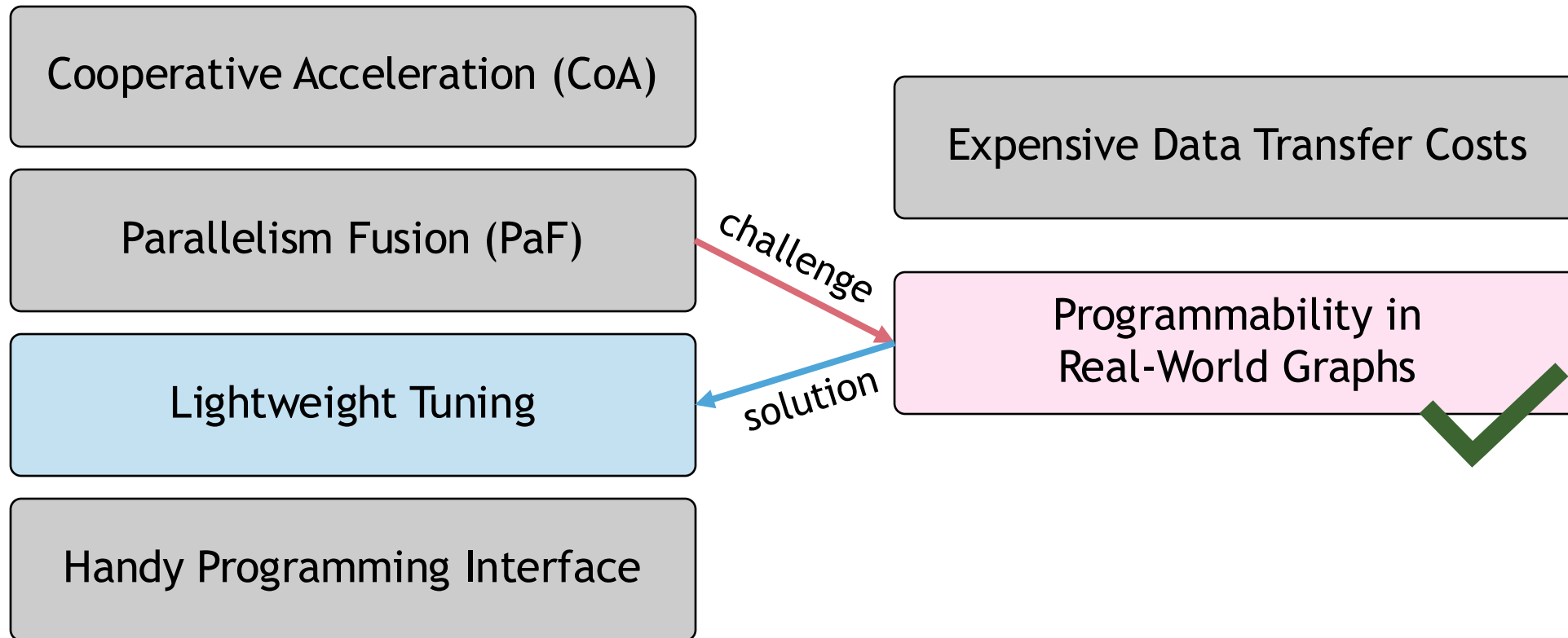


diagonal graph

Real-world graphs exhibit diverse characteristics

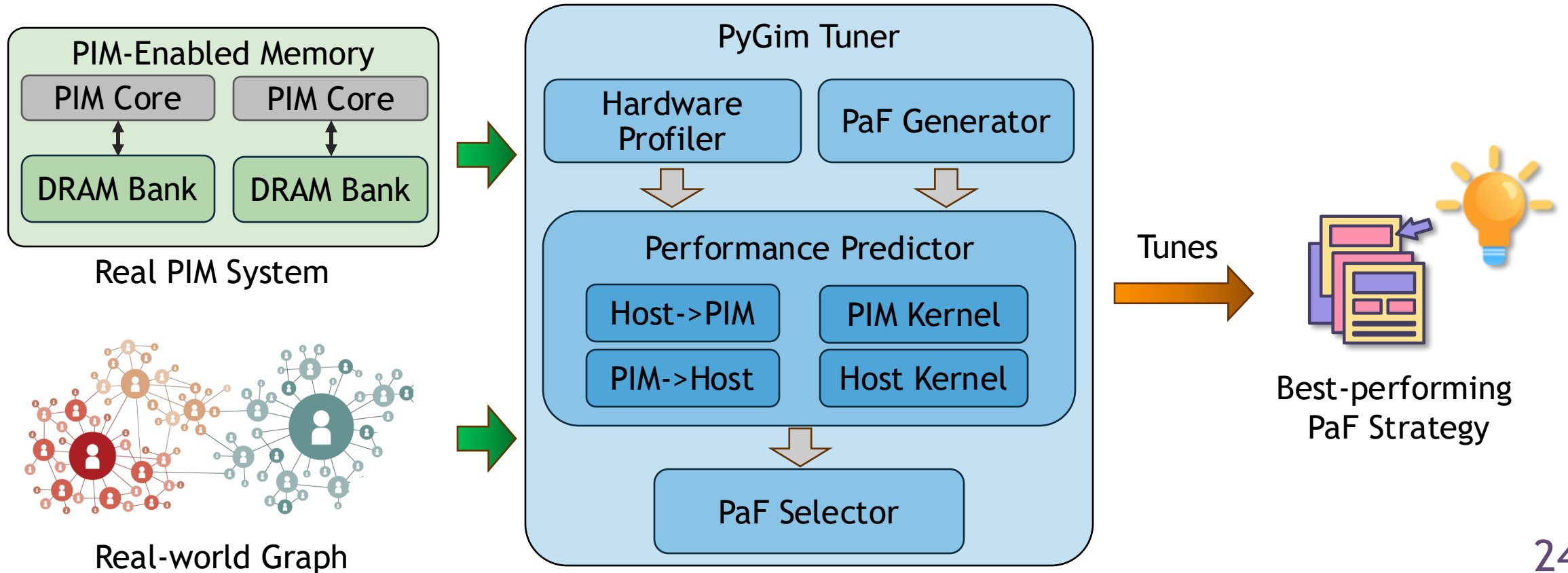
# PyGim Overview

- An **efficient** and **easy-to-use** GNN library for real PIM systems
- PyGim incorporates 4 **key** components:



# 3. Lightweight Tuning

- PyGim Tuner predicts the *best-performing PaF strategy* without manual programmer intervention
  - **Hardware profiler** generate a group of performance measurements
  - **Performance predictor** predict the execution of potential PaF strategy
  - **PaF selector** apply the best-performing PaF strategy



# PyGim Overview

- An **efficient** and **easy-to-use** GNN library for real PIM systems
- PyGim incorporates 4 **key** components:

Cooperative Acceleration (CoA)

Parallelism Fusion (PaF)

Lightweight Tuning

Handy Programming Interface

# 4. Handy Programming Interface

- PyGim integrates a *handy* Python API (currently integrated with PyTorch)

```
1 import ... pygim as gyn
2 class GCNConv(torch.nn.Module):
3     def __init__(self, hidden_size):
4         self.linear = torch.nn.Linear(feature_size, features_size)
5
6     def forward(self, graph_pim, in_dense):
7         # Execute memory-intensive kernel in real PIM devices
8         dense_parts = col_split(in_dense)
9         out_dense = gyn.pim_run_aggr(graph_pim, dense_parts)
10        # Execute compute-intensive operator in Host (e.g., CPU/GPU)
11        out = self.linear(out_dense)
12        return out
13
14 gyn.pim_init_devices(num_pim_devices) # Initialize PIM devices
15 data = load_dataset() # Load graph
16 # Tune the PaF strategy
17 graph_pim= gyn.tune(data.graph, feature_size, device_info)
18 graph_pim = gyn.load_graph_pim(graph_parts) # Partition graph to PIM
19 # Create GNN model
20 model=torch.nn.Sequential([Linear(in_channels,feature_size),
21                             GCNConv(feature_size),
22                             GCNConv(feature_size),
23                             GCNConv(feature_size),
24                             Linear(feature_size, out_channels)])
25 model.forward(graph_pim, data.features) # GCN inference
```

# Deploy Your GNNs Effortlessly with PyGim and Enjoy the PIM Benefits!

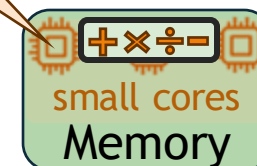
```
1 import ... pygim as gyn
2 class GCNConv(torch.nn.Module):
3 def __init__(self, hidden_size):
4     self.linear = torch.nn.Linear(feature_size, features_size)
5
6 def forward(self, graph_pim, in_dense):
7     # Execute memory-intensive kernel in real PIM devices
8     dense_parts = col_split(in_dense)
9     out_dense = gyn.pim_run_aggr(graph_pim, dense_parts)
10    # Execute compute-intensive operator in Host (e.g., CPU/GPU)
11    out = self.linear(out_dense)
12    return out
13
14 gyn.pim_init_devices(num_pim_devices)
15 data = load_dataset() # Load graph
16 # Tune the PaF strategy
17 graph_pim= gyn.tune(data.graph, feature_size, device_info)
18 graph_pim = gyn.load_graph_pim(graph_parts) # Partition graph
19 # Create GNN model
20 model=torch.nn.Sequential([Linear(in_channels,feature_size),
21     GCNConv(feature_size),
22     GCNConv(feature_size),
23     GCNConv(feature_size),
24     Linear(feature_size, out_channels)])
25 model.forward(graph_pim, data.features) # GCN inference
```

Computation is performed inside real PIM devices!

```
Loading kernel from: /home/upmem0013/
m_mul_coo_dpu
1000 DPUs are allocated in 16 ranks
Allocated 16 TASKLET(s) per DPU
BLNC = BLNC_NNZ
SYNC = True
BLNC_TSKLT = BLNC_TSKLT_NNZ
LOCK = LOCKFREEV2
MERGE = BLOCK
PIM_SEQREAD_CACHE_SIZE=32
val_dt = INT32
sppm_coo_to_device_group
prepare_pim finished
```

```
Iteration 0000: Time: 7127.9930 ms.
Iteration 0001: Time: 7191.6390 ms.
Iteration 0002: Time: 7102.1040 ms.
Iteration 0003: Time: 6888.6810 ms.
Iteration 0004: Time: 7075.0290 ms.
Iteration 0005: Time: 6844.8220 ms.
```

fast-forwarded



# Talk Outline

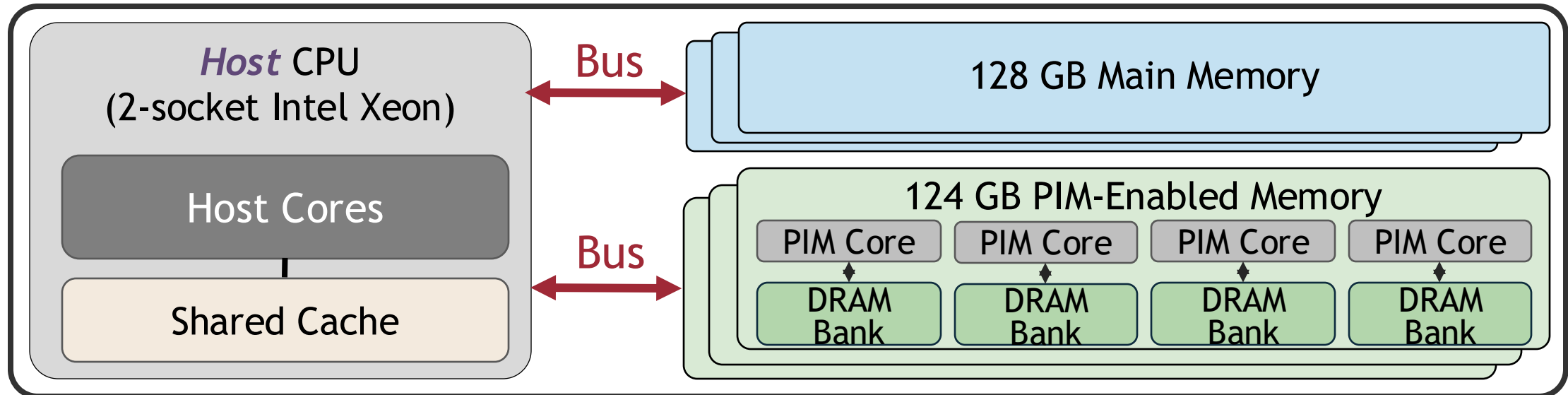
Background & Motivation

PyGim Design

Evaluation

# Evaluation Methodology

- UPMEM PIM server: 16 PIM DIMMs with **1992** PIM Cores (24 threads per core) in total
- GNN models: GCN, GIN, SAGE
- Datasets: ogbn-proteins, reddit, amazonProducts
- Comparison points:
  - **PyTorch** running on host CPU
  - **SparseP** [Sigmetrics'22] (**2×**) running SpMM as multiple SpMV kernels on PIM cores
  - **GraNDe** [IEEE Trans. Comput.'23]: optimizes GNN aggregation on near-rank PIM systems

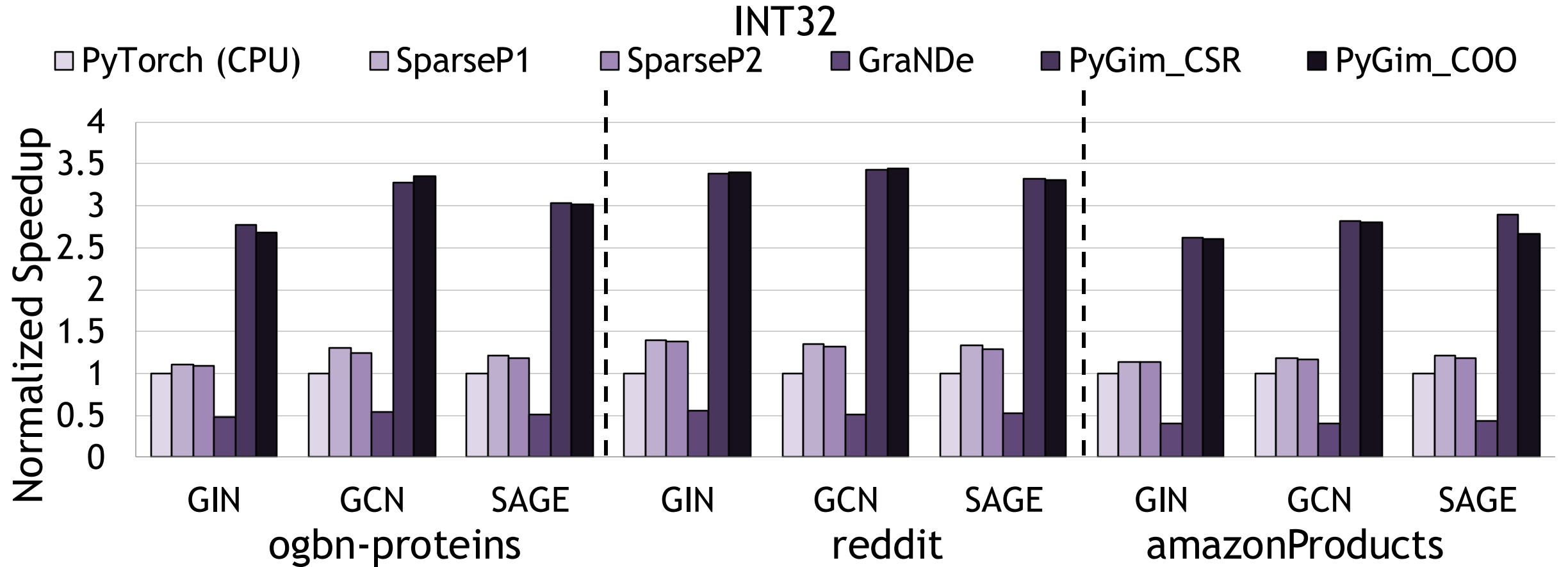


UPMEM PIM System

# Performance Evaluation in GNN Inference

## INT32

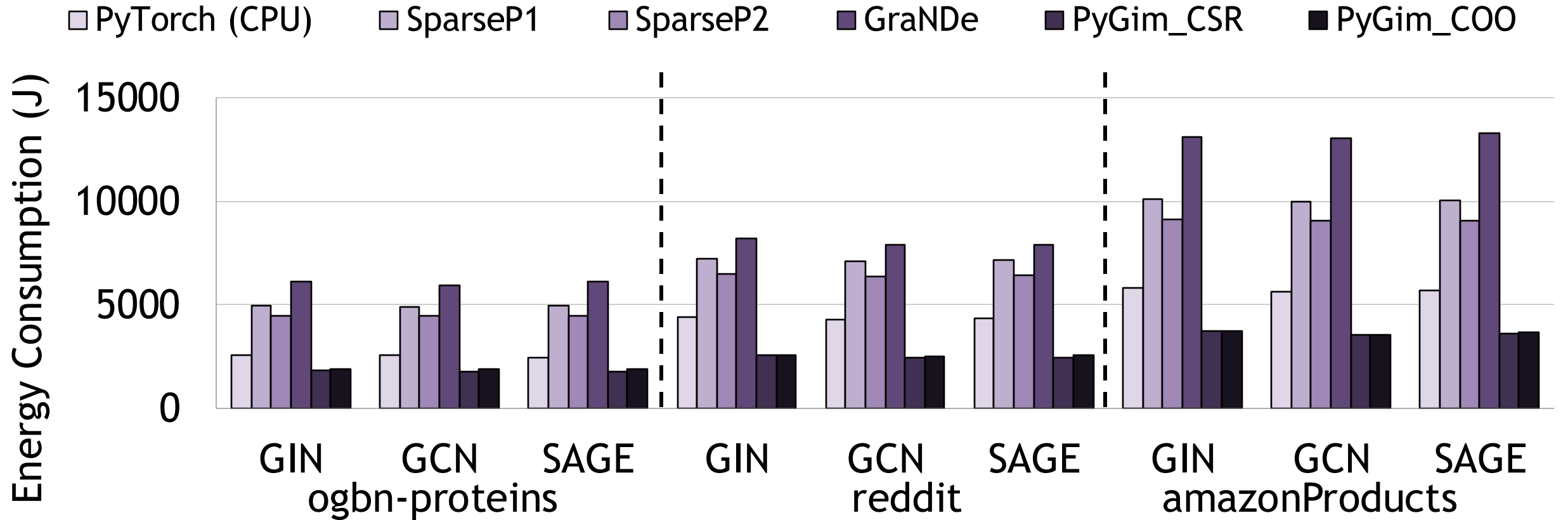
# Performance Evaluation in GNN Inference



PyGim significantly outperforms **PyTorch (CPU)** and prior **PIM-based schemes** by **3.1×** and **4.4×**, respectively

# Energy Efficiency Evaluation in GNN Inference

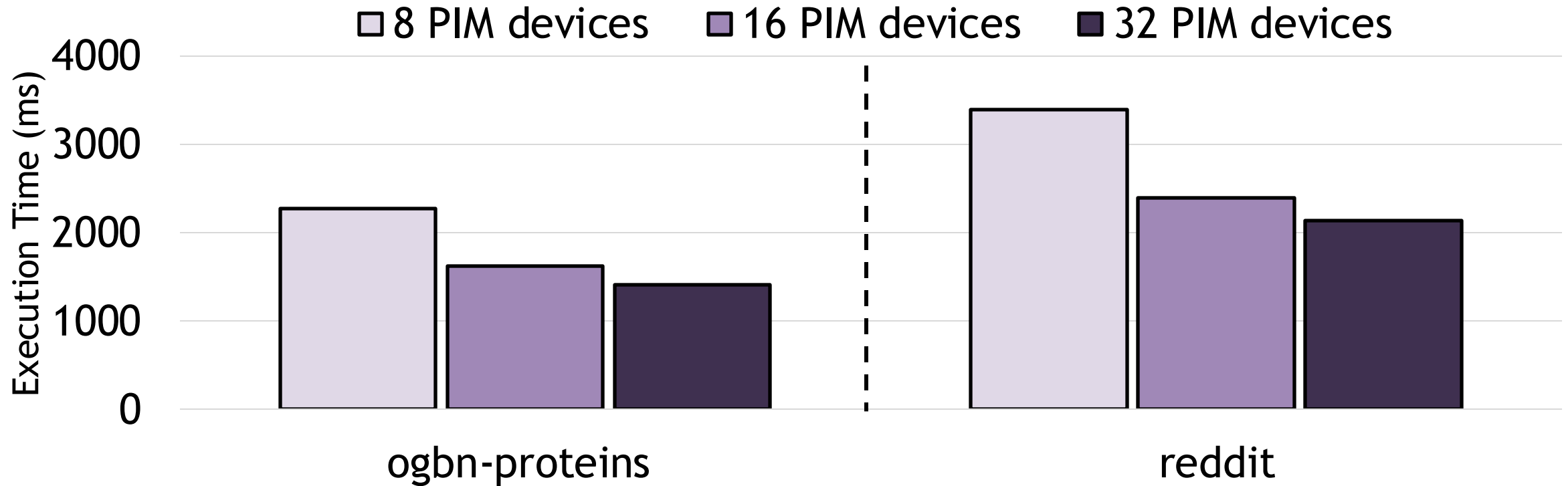
INT32



PyGim improves energy efficiency by **2.7×** and **3.3×** compared to **PyTorch (CPU)** and prior **PIM-based schemes**, respectively

# Scalability Analysis of PyGim in the Aggregation Kernel

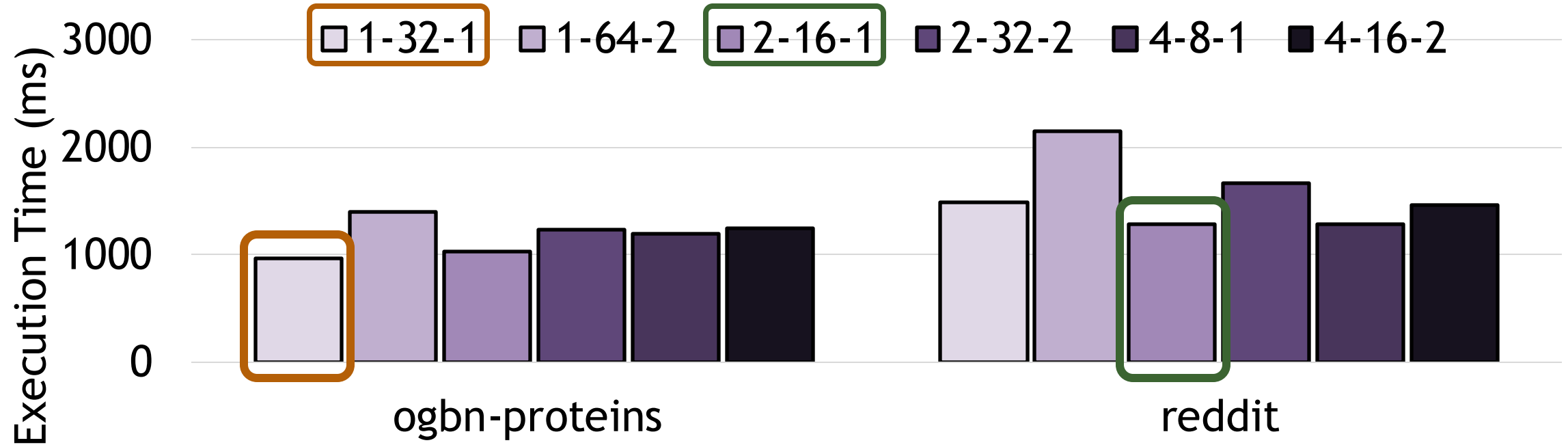
INT32 (fixed PaF configuration)



PyGim is **scalable**: when we double the number of PIM devices, performance improves by **1.4x** on average

# Performance of PaF Configurations in Aggregation for Various Graphs

INT32 (fixed 32 PIM devices)



Triple of values = PaF configuration for #edge partitions, #feature partitions and #PIM clusters per device

**No one-size-fits-all:** the graph's characteristics affect the **best-performing** parallelization strategy

# Performance of PaF Configurations in Aggregation for Various Graphs

INT32 (fixed 32 PIM devices)

## Recommendation 1

Commodity DRAM **has multiple hierarchy levels** (e.g., DIMM-, rank-, bank-group-, bank-level), each level has different characteristics. PIM **architects** can enable different **hardware optimizations at each level of hierarchy**. Then, **software** engineers can design different **software optimization techniques for each different level** of hardware hierarchy to enable high performance in PIM executions via hardware software co-design.

## Recommendation 2

The system performance of **sparse workloads** highly depends on the particular patterns of **each input**. Systems software engineers can design **various optimizations strategies** to effectively support **diverse inputs**. They can then develop **tuning mechanisms** (e.g., intelligent heuristics, prediction models, compilers, automation tools) to automatically adjust these strategies based on the input, with **minimal programming effort**, thus achieving the highest possible performance in PIM systems.

# Characteristics of CPU, PIM and GPU Systems

System	INT32 Peak Performance	FP32 Peak Performance	Total Bandwidth	Technology Node
CPU Xeon 4215	0.64 TOPS	1.28 TFLOPS	23.1 GB/s	14nm
UPMEM PIM	0.12 TOPS	0.025 TFLOPS	1390 GB/s	at least 20nm
GPU GTX 1080 Ti	13.25 TOPS	13.25 TFLOPS	359.9 GB/s	16nm
GPU RTX 2080 Ti	16.94 TOPS	16.94 TFLOPS	558.1 GB/s	12nm
GPU RTX 3090	17.79 TOPS	35.58 TFLOPS	936.2 GB/s	8nm

Across last GPU generations:

- *memory bandwidth* has tripled (~3x)
- (last two generations) *compute throughput* has been doubled (~2x)

# Core Utilization in GNN Aggregation

Dataset & data type/ Software library	Reddit INT32	Reddit FP32
Intel MKL (CPU Intel Xeon 4215)	0.63%	0.22%
CUDA (GPU GTX 1080 Ti)	0.62%	0.62%
CUDA (GPU RTX 2080 Ti)	0.68%	0.67%
CUDA (GPU RTX 3090)	1.56%	0.78%
PyGim (UPMEM PIM)	13.86%	9.13%

Core utilization in GNN aggregation remains *similarly low* across GPU generations

PyGim running on a real PIM system achieves significantly higher core utilization (11.6x on average) than the PyTorch on GPUs

# More in the Paper

- Analysis within a PIM core
- Analysis within a PIM cluster
- Analysis across PIM clusters
- PyGim tuning efficiency
- Scalability analysis
- Analysis on different data types
- Analysis on different compression formats
- Performance evaluation in GNN training
- Recommendations

## PyGim: An Efficient Graph Neural Network Library for Real Processing-In-Memory Architectures

CHRISTINA GIANNOULA, University of Toronto, Canada, ETH Zürich, Switzerland, Vector Institute, Canada, and CentML, Canada

PEIMING YANG, University of Toronto, Canada

IVAN FERNANDEZ, Barcelona Supercomputing Center, Spain, Universitat Politècnica de Catalunya, Spain, and ETH Zürich, Switzerland

JIACHENG YANG, University of Toronto, Canada and Vector Institute, Canada

SANKEERTH DURVASULA, University of Toronto, Canada and Vector Institute, Canada

YU XIN LI, University of Toronto, Canada

MOHAMMAD SADROSADATI, ETH Zürich, Switzerland

JUAN GOMEZ LUNA, NVIDIA, Switzerland

ONUR MUTLU, ETH Zürich, Switzerland

GENNADY PEKHIMENKO, University of Toronto, Canada, Vector Institute, Canada, and CentML, Canada

Graph Neural Networks (GNNs) are emerging models to analyze graph-structure data. The GNN execution involves both compute-intensive and memory-intensive kernels. The memory-intensive kernels dominate execution time, because they are significantly bottlenecked by data movement between memory and processors. Processing-In-Memory (PIM) systems can alleviate this data movement bottleneck by placing simple processors near or inside memory arrays. To this end, we investigate the potential of PIM systems to alleviate the data movement bottleneck in GNNs, and introduce PyGim, an efficient and easy-to-use GNN library for real PIM systems. We propose intelligent parallelization techniques for memory-intensive kernels of GNNs tailored for real PIM systems, and develop an easy-to-use Python API for them. PyGim employs a cooperative GNN execution, in which the compute- and memory-intensive kernels are executed in processor-centric and memory-centric computing systems, respectively, to fully exploit the hardware capabilities. PyGim integrates a lightweight tuner that configures the parallelization strategy of the memory-intensive kernel of GNNs to provide high system performance, while also enabling high programming ease. We extensively evaluate PyGim on a real-world PIM system that has 16 PIM DIMMs with 1992 PIM cores connected to a Host CPU. In GNN inference, we demonstrate that it outperforms prior state-of-the-art PIM works by on average 4.38× (up to 7.20×), and the state-of-the-art PyTorch implementation running on Host (on Intel Xeon CPU) by on average 3.04× (up to 3.44×). PyGim improves energy efficiency by 2.86× (up to 3.68×) and 1.55× (up to 1.75×) over prior PIM and PyTorch Host schemes, respectively. In memory-intensive kernel of GNNs, PyGim provides 11.6× higher resource utilization in PIM system than that of PyTorch library (optimized CUDA implementation) in GPU systems. Our work provides useful recommendations for software, system and hardware designers. PyGim is publicly and freely available at <https://github.com/CMU-SAFARI/PyGim> to facilitate the widespread use of PIM systems in GNNs.

**Key Words:** machine learning, graph neural networks, sparse matrix-matrix multiplication, library, multicore, processing-in-memory, near-data processing, memory systems, data movement bottleneck, DRAM, benchmarking, real-system characterization, workload characterization

<https://arxiv.org/pdf/2402.16731>

# PyGim is Open Source



CMU-SAFARI / PyGim

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

PyGim Public

main 1 Branch 0 Tags

Go to file Add file Code

cgianoula Update README.md	7612ee9 · 4 months ago	7 Commits
Libs	first commit	4 months ago
backend_pim	first commit	4 months ago
utils	first commit	4 months ago
README.md	Update README.md	4 months ago
build.sh	first commit	4 months ago
inference.py	first commit	4 months ago
spmm_test.py	first commit	4 months ago

README

## PyGim: An Efficient Graph Neural Network Library for Real Processing-In-Memory Architectures

**About**

PyGim is the first runtime framework to efficiently execute Graph Neural Networks (GNNs) on real Processing-in-Memory systems. It provides a high-level Python interface, currently integrated with PyTorch, and supports various GNN models and real-world input graphs. Described by SIGMETRICS'25 by Giannoula et al. (<https://arxiv.org/pdf/2402.16731>)

[arxiv.org/pdf/2402.16731](https://arxiv.org/pdf/2402.16731)

- Readme
- Activity
- Custom properties
- 19 stars
- 5 watching
- 1 fork

Report repository

[github.com/CMU-SAFARI/PyGim](https://github.com/CMU-SAFARI/PyGim)

# Conclusion

We present PyGim, a handy ML library that significantly improves *performance*, *energy efficiency* and *cost effectiveness* in GNNs through real PIM devices

## Key Ideas & Benefits:

- *balances computation* and *data transfer* costs via configurable parallelization strategies for *diverse* real-world graphs
- *automatically* tunes the best-fit strategy *without programmer intervention*

## Key Results:

- *performance* and *energy efficiency* by **3.7×** and **2.3×** over SOTA schemes
- *core utilization* on PIM system by **11.6×** over PyTorch on GPUs



PyGim



Github



Paper

# PyGim: An Efficient Graph Neural Network Library for Real Processing-In-Memory Architectures



Christina Giannoula,

Peiming Yang, Ivan Fernandez, Jiacheng Yang, Sankeerth Durvasula, Yu Xin Li, Mohammad Sadrosadati, Juan Gomez Luna, Onur Mutlu, Gennady Pekhimenko

