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

Christina Giannoula

https://cgiannoula.github.io/

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

1st Workshop on Memory-Centric Computing Systems (MCCSys)
March 2025

Executive Summary

Motivation: Graph Neural Networks (GNNs) analyze graph-structure data in important real-world applications such as drug discovery, social network analysis, recommendation systems…

Problem: The memory-intensive kernels of GNNs, which dominate execution time (~91%), are significantly *bottlenecked by memory bandwidth* in procesor-centric systems (CPUs/GPUs)

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

Key Ideas & Benefits:

- Cost Effectiveness: *Heteregenous* 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) *resource utilization* on PIM system by **11.6×** over PyTorch on GPUs

github.com/CMU-SAFARI/PyGim

Talk Outline

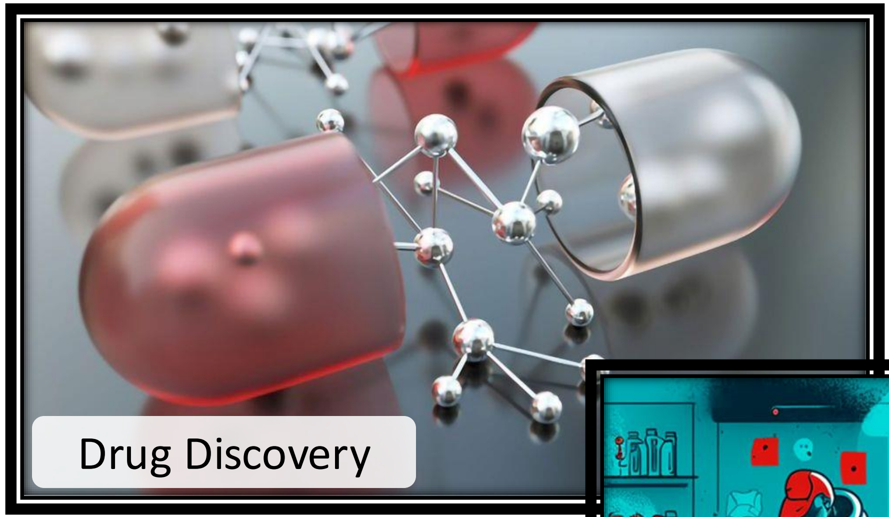
Background & Motivation

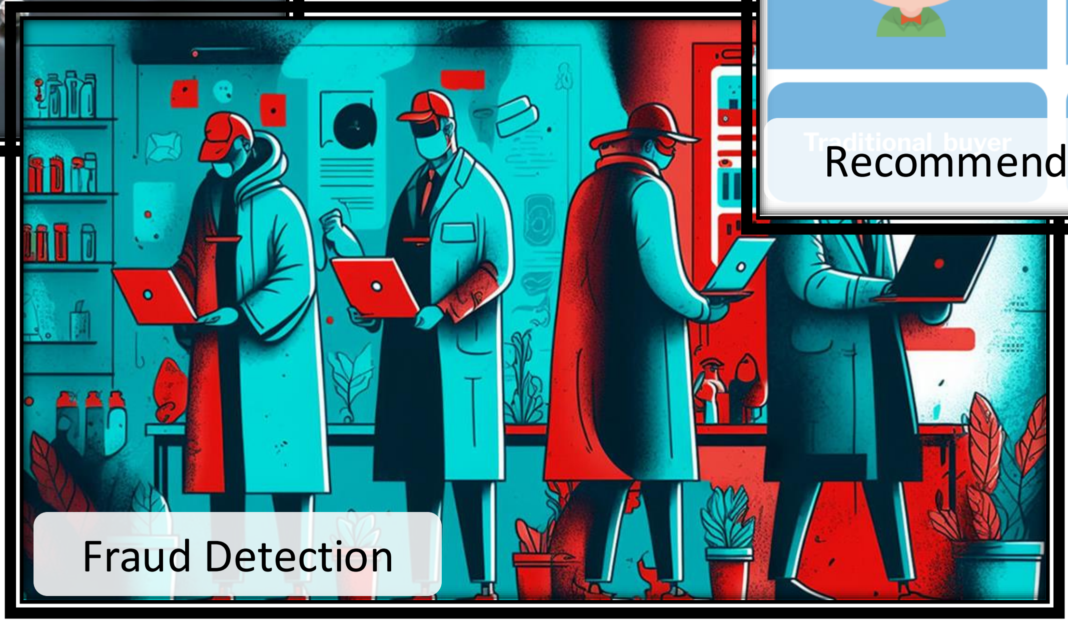
PyGim Design

Evaluation

GNNs Are Widely Used in Real-World Applications

- GNNs are state-of-the-art ML models for analyzing graph-structure data
- Applications of GNNs are:

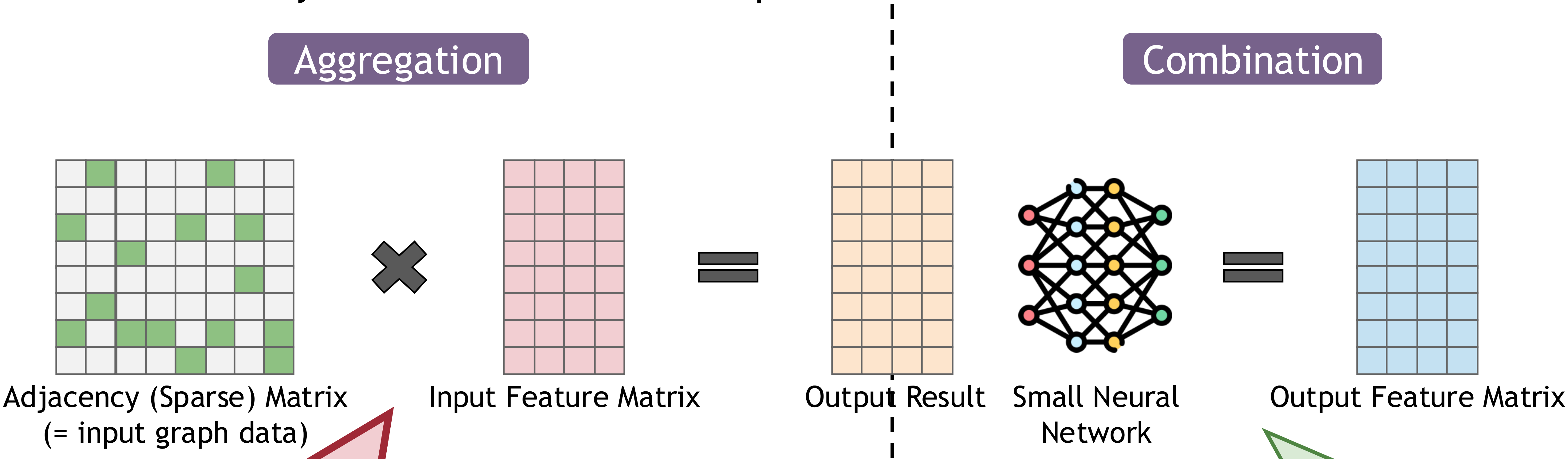

Drug Discovery


Fraud Detection

Recommendation Systems

Execution Steps of GNN Layers

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

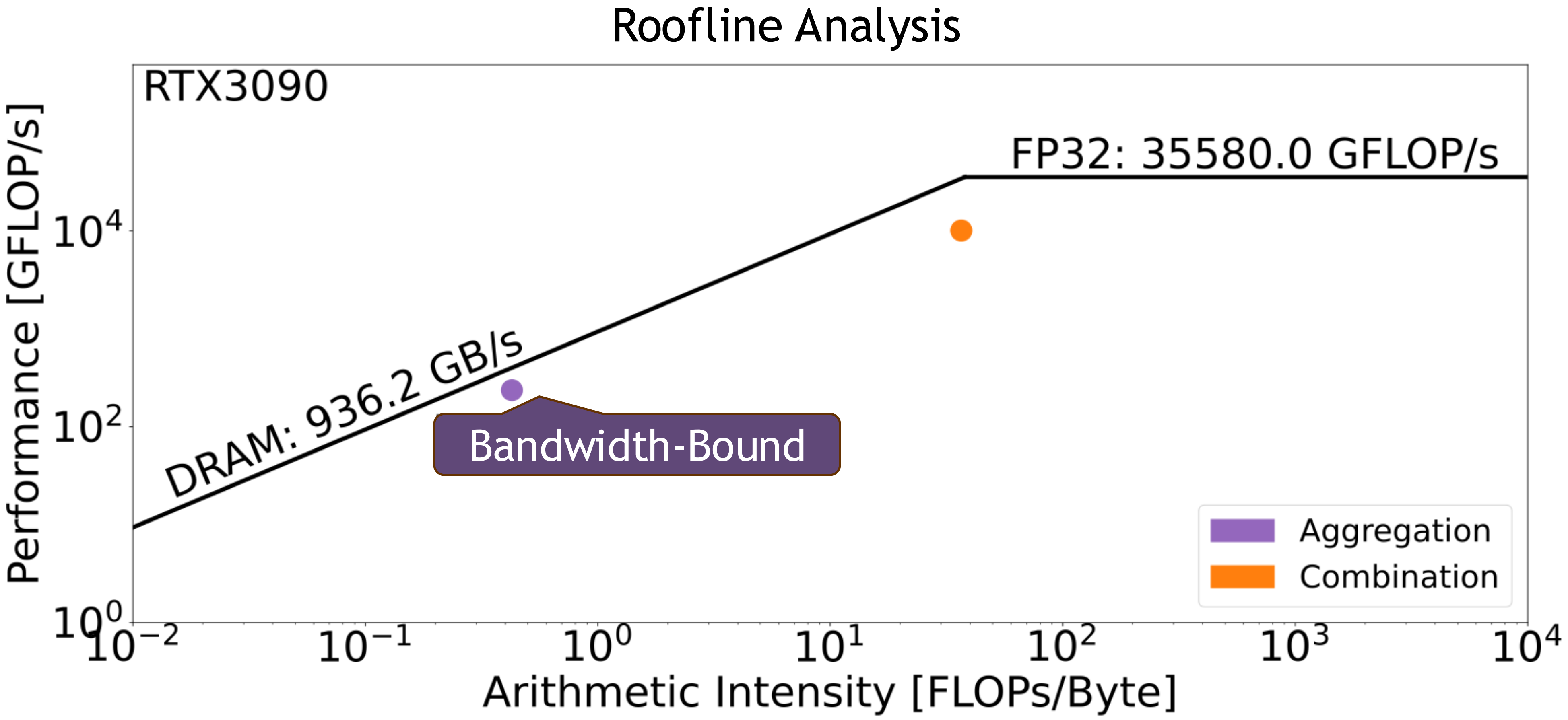


Aggregation corresponds to Sparse Matrix Matrix Multiplication (***SpMM***)

Combination typically comprises computational kernels (e.g., GEMMs)

GNN Aggregation Is Memory-Bandwidth-Bound In GPUs

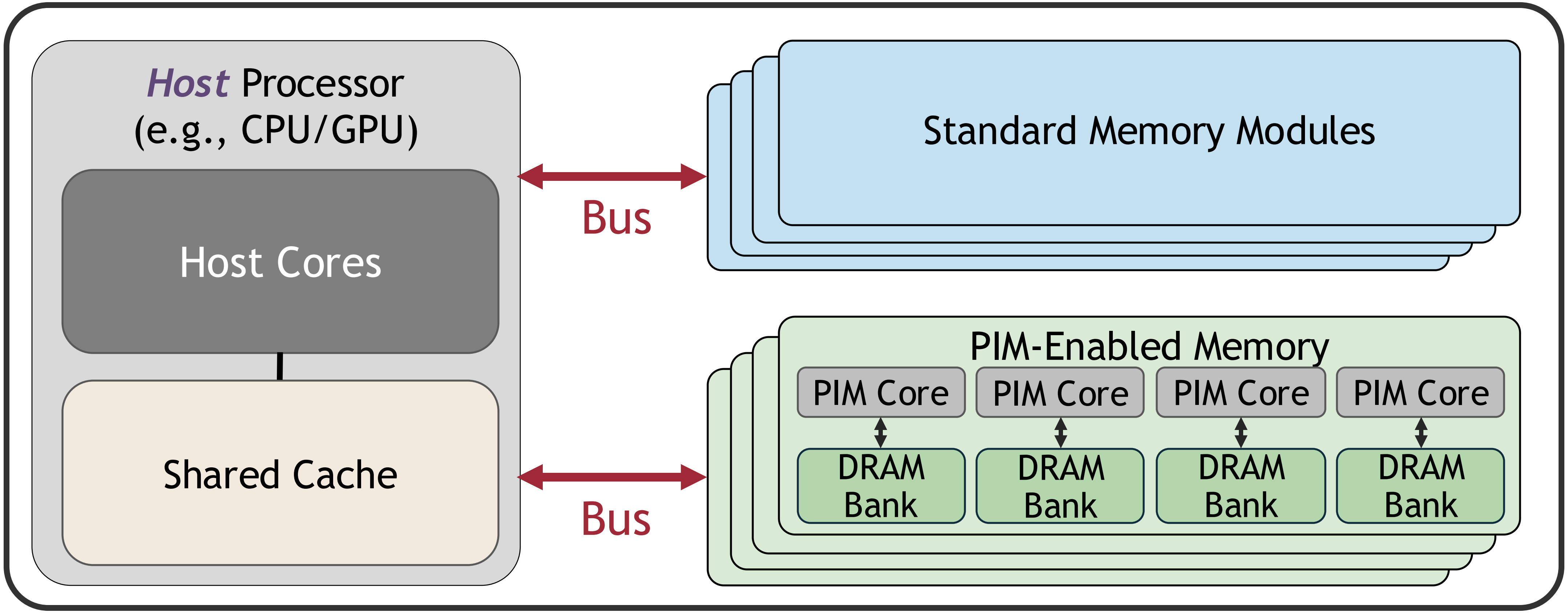
Using a RTX 3090 GPU with ~900 GB/s bandwidth, we find that GNN Aggregation
- takes ***~91%*** of the inference time
- achieves ***less than 2%*** core utilization



PIM Provides A Promising Solution for GNN Aggregation

- 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 that available on 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:
  1. Cooperative Acceleration (CoA)
  2. Parallelism Fusion (PaF)
  3. Lightweight Tuning
  4. Handy Programming Interface
- PyGim is open source:

PyGim: github.com/CMU-SAFARI/PyGim

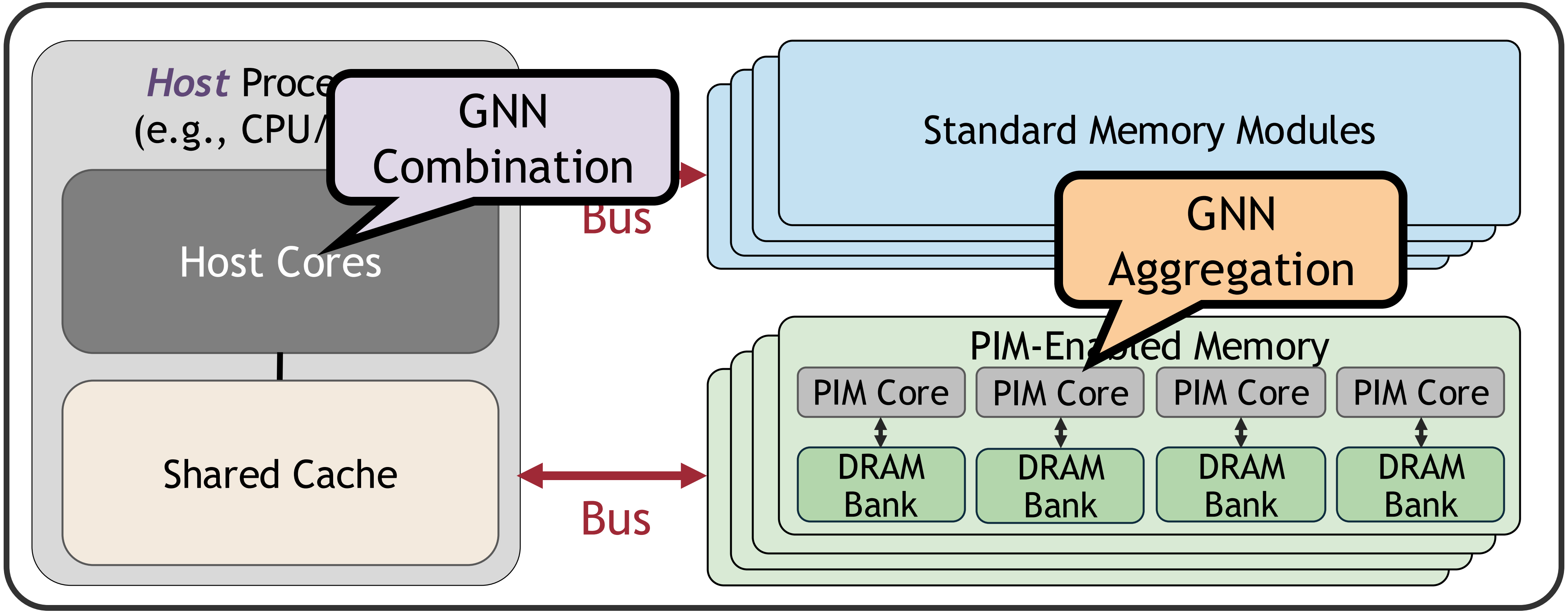
Deploy your GNNs *effortless* and enjoy the PIM benefits!

1. Cooperative Acceleration (CoA)

Heterogeneous kernels are running in the best-fit underlying hardware

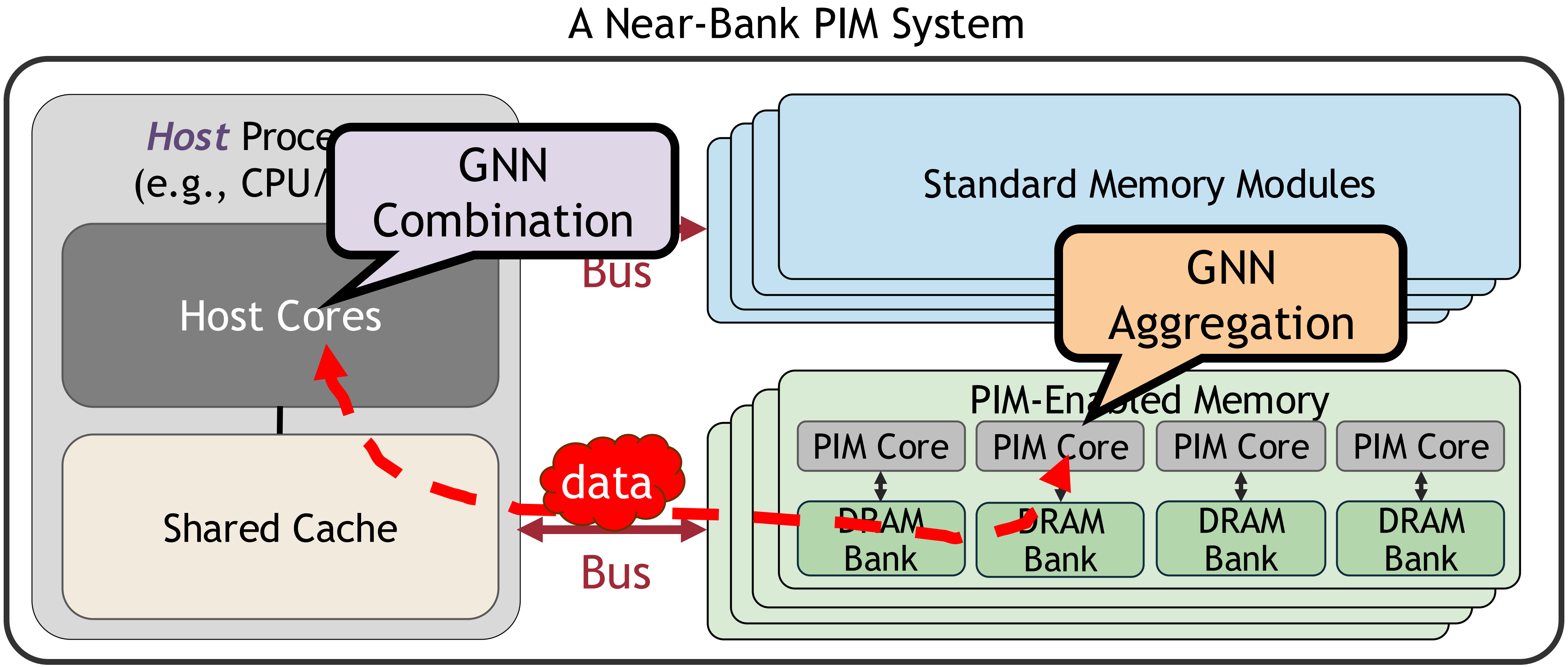
- Combination runs on Host cores
- Aggregation runs on PIM cores

A Near-Bank PIM System



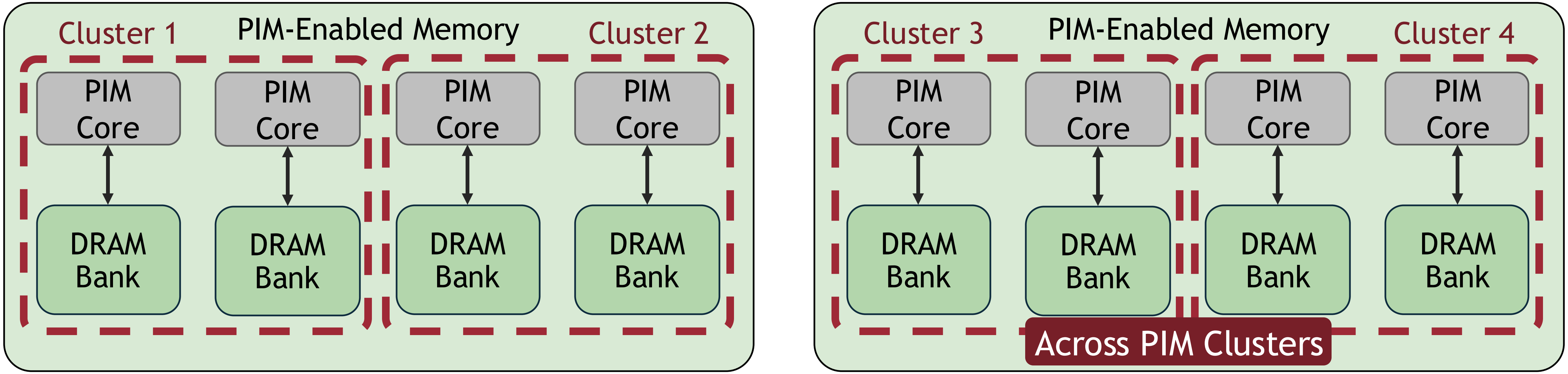
Challenge 1: Data Transfer Costs

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



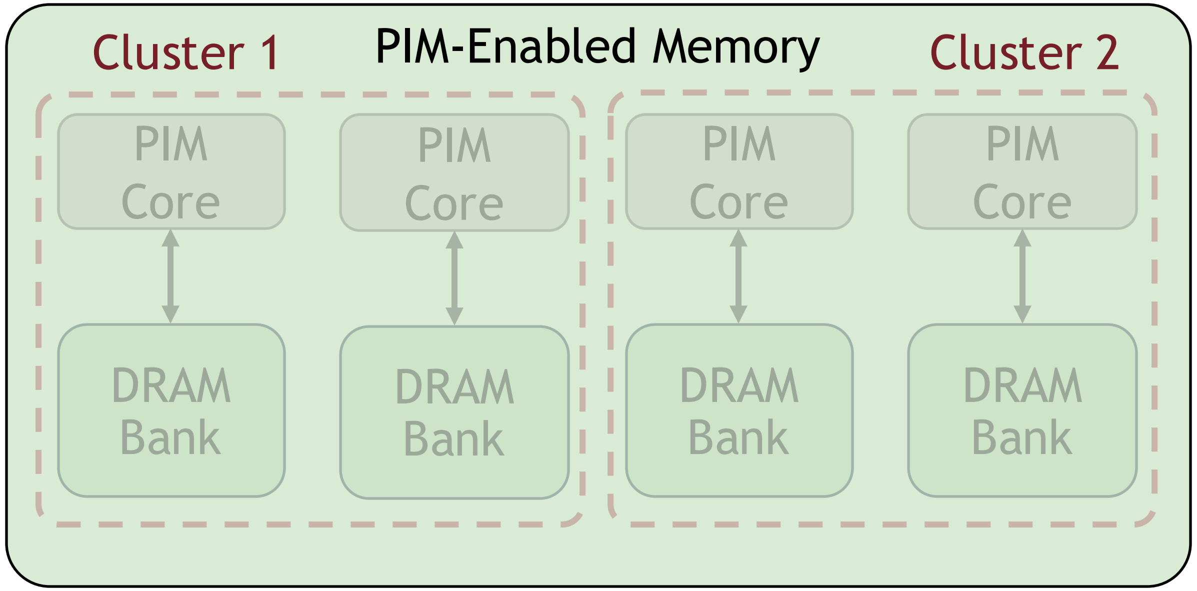
2. Parallelism Fusion (PaF)

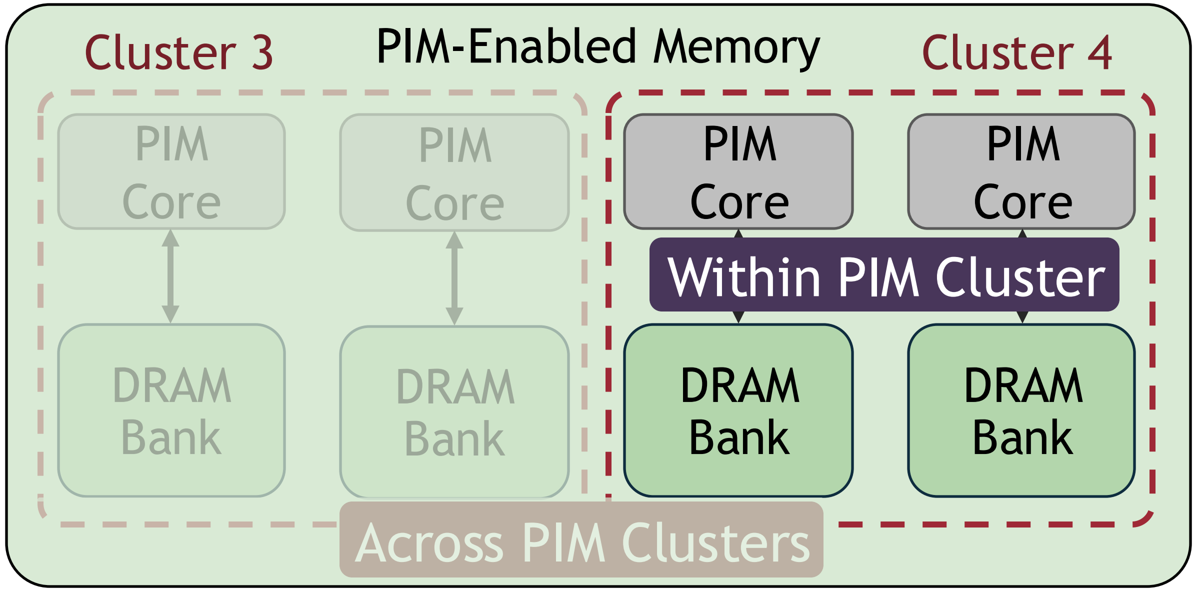
- PaF (i) strives a balance between *computation* and *data transfer* costs and (ii) efficiently covers various real-world *graphs* that exhibit *diverse characteristics*
- PaF enablers 3 levels of parallelism:
    1. Across PIM Clusters: Edge- + Feature-level parallelism



2. Parallelism Fusion (PaF)

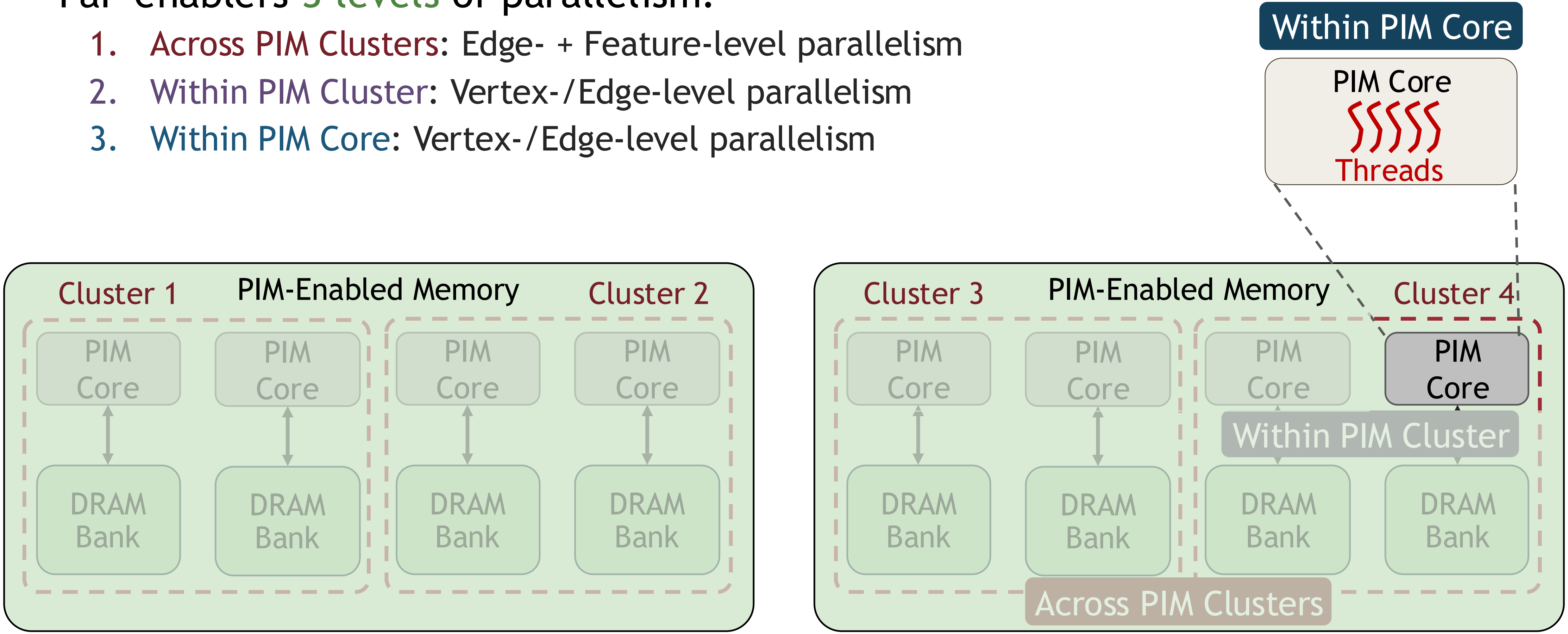
- PaF (i) strives a balance between *computation* and *data transfer* costs and (ii) efficiently covers various real-world *graphs* that exhibit *diverse characteristics*
- PaF enablers 3 levels of parallelism:
  1. Across PIM Clusters: Edge- + Feature-level parallelism
  2. Within PIM Cluster: Vertex-/Edge-level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



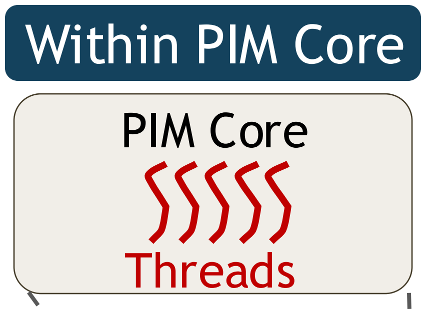
2. Parallelism Fusion (PaF)

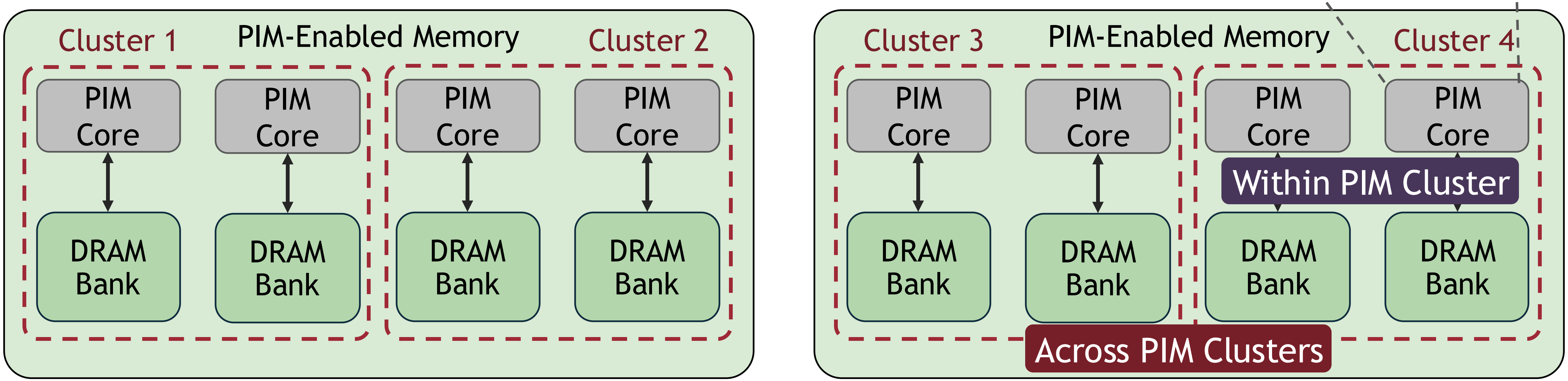
- PaF (i) strives a balance between *computation* and *data transfer* costs and (ii) efficiently covers various real-world *graphs* that exhibit *diverse characteristics*
- PaF enablers 3 levels of parallelism:

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

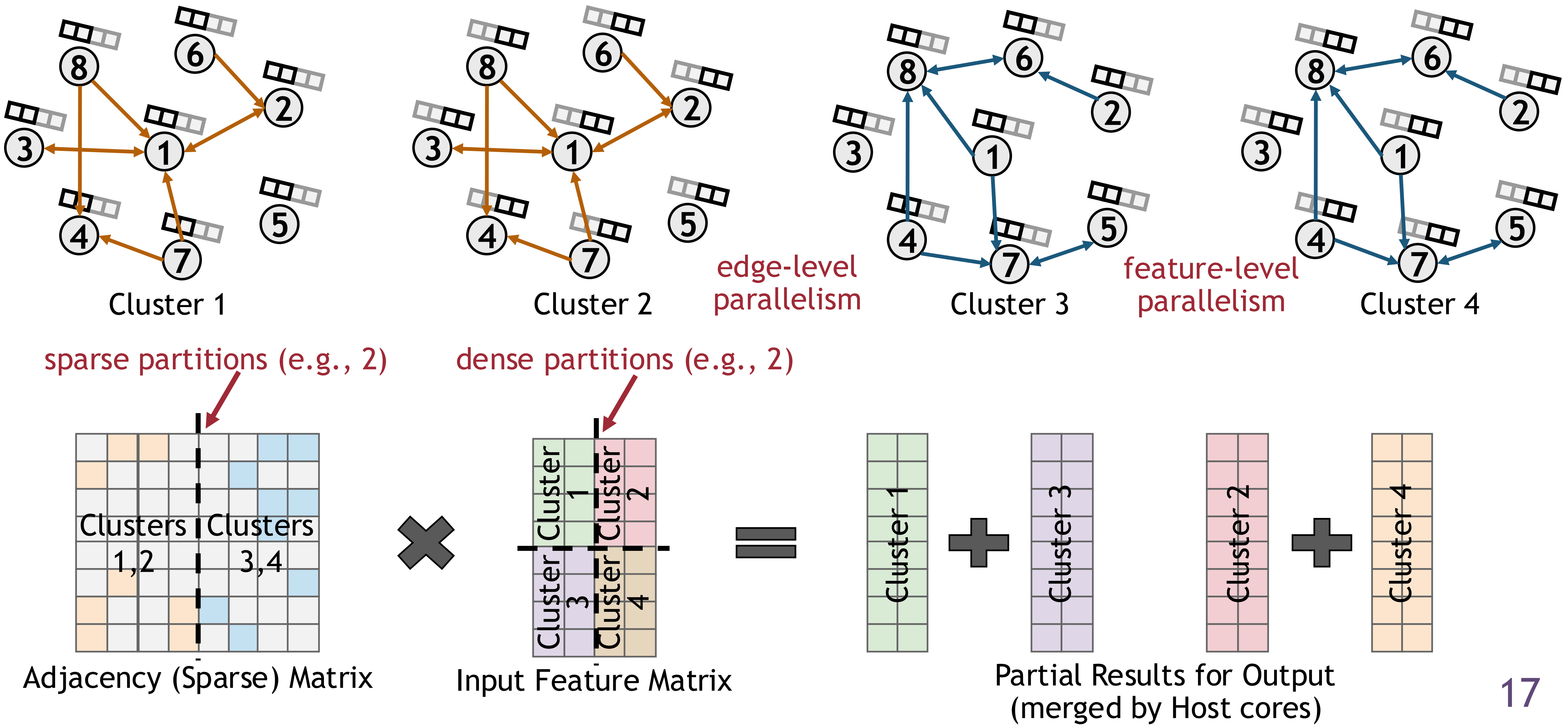
Reduce computation costs





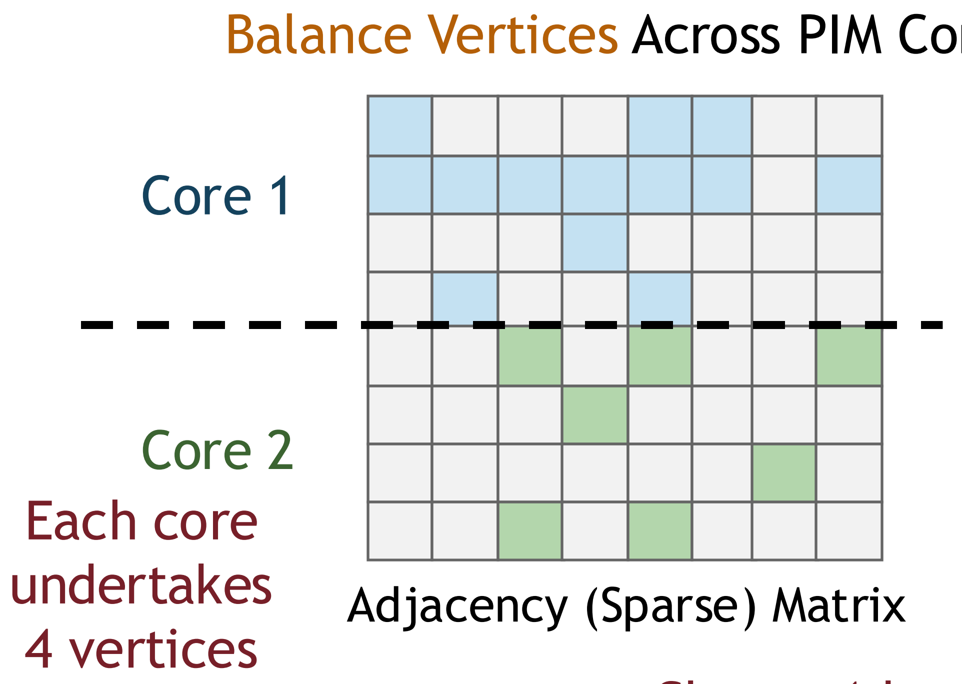
Across PIM Clusters: Edge- + Feature-Level Parallelism

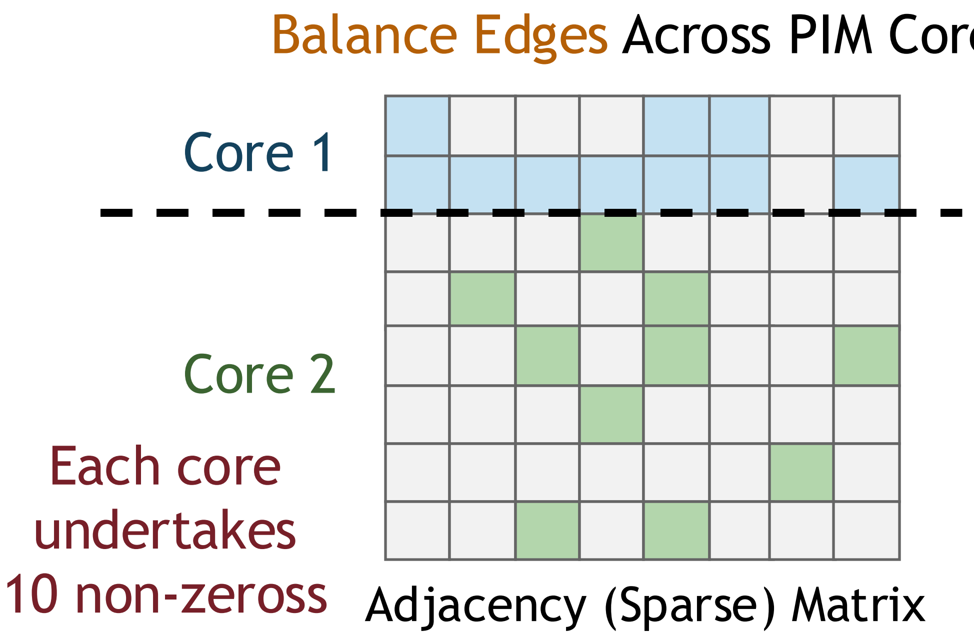
- E.g., creating 4 PIM clusters with 2 sparse partitions and 2 dense partitions

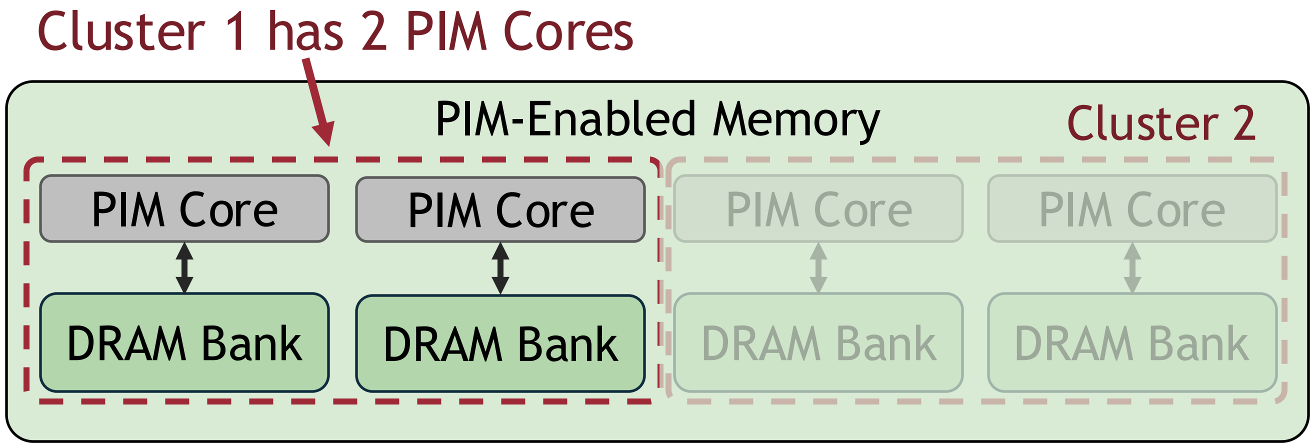


Within a PIM Cluster: Vertex-/Edge-Level Parallelism

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

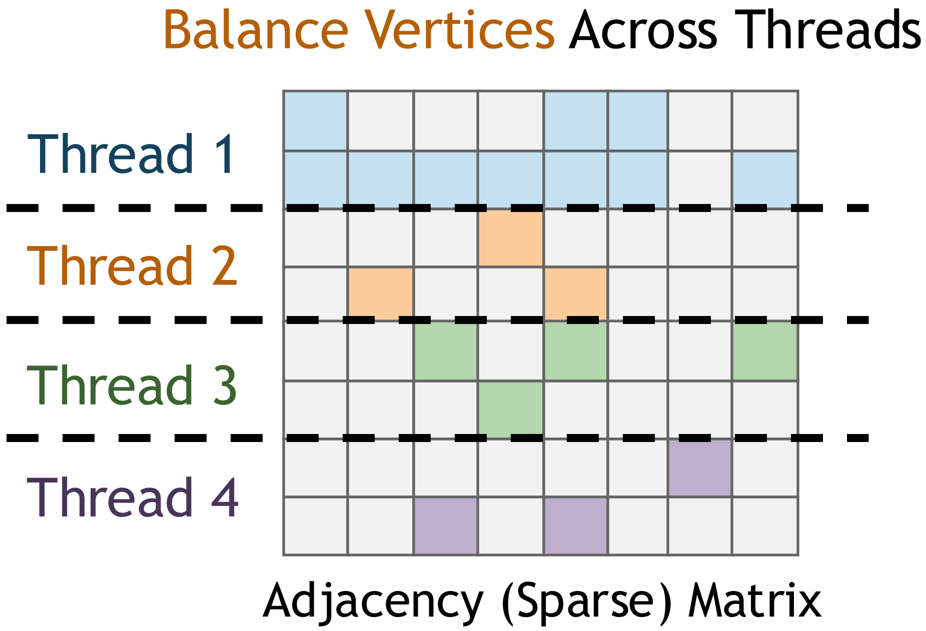




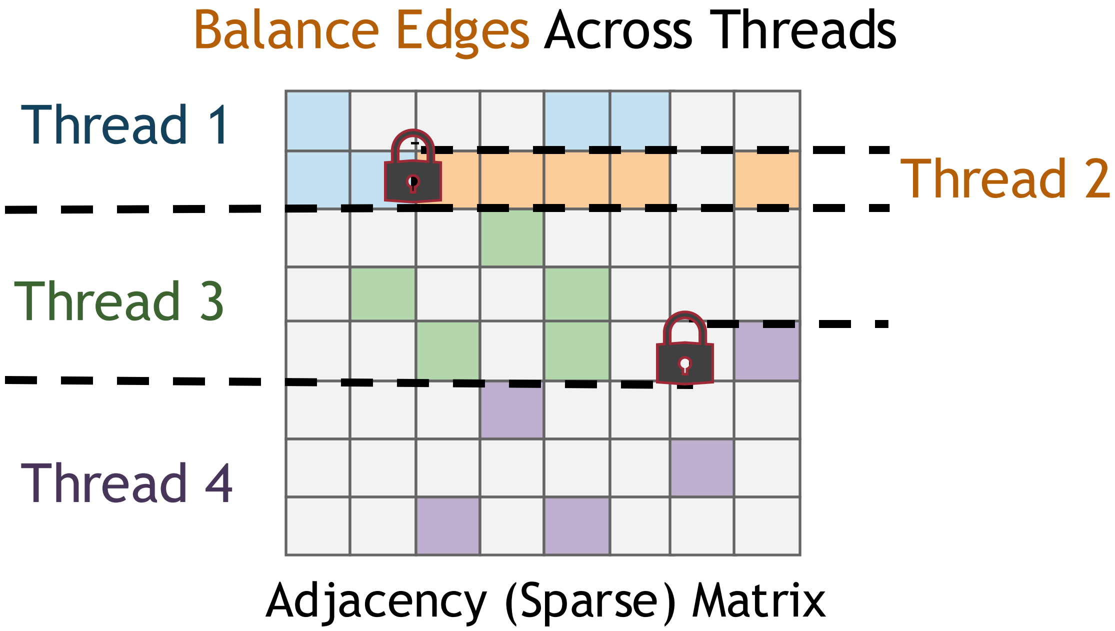


Within a PIM Core: Vertex-/Edge-Level Parallelism

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

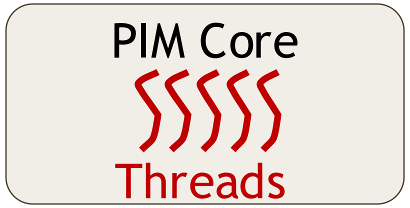


Each thread undertakes 2 vertices



Each thread undertakes 5 non-zeros

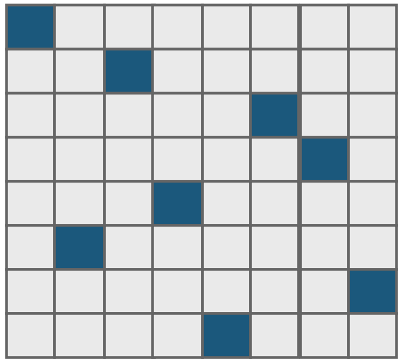
PIM Core supports 4 threads



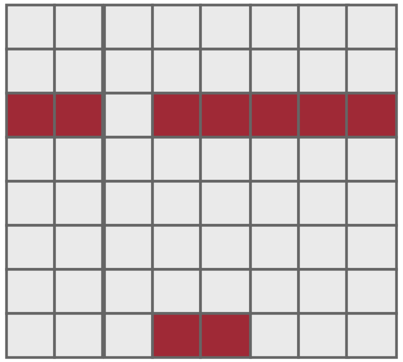
Synchronization is implement with coarse-grained and fine-grained locking schemes

Challenge 2: Programmability in Real-World Graphs

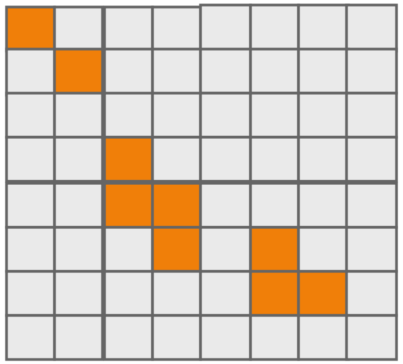
- Real-world graphs exhibit diverse (non-zero point) characteristics:
  - Min, max or average vertex neighboring degree, graph's diameter…
- Typically there is no one-size-fits-all solution:
  → PaF supports a wide variety of parallelization strategies for diverse real-world graphs
- Key challenge = ***manually tuning*** the best-performing parallelization strategy for each *unique* graph's characteristics poses significant challenges for developers



regular graph



power-law graph

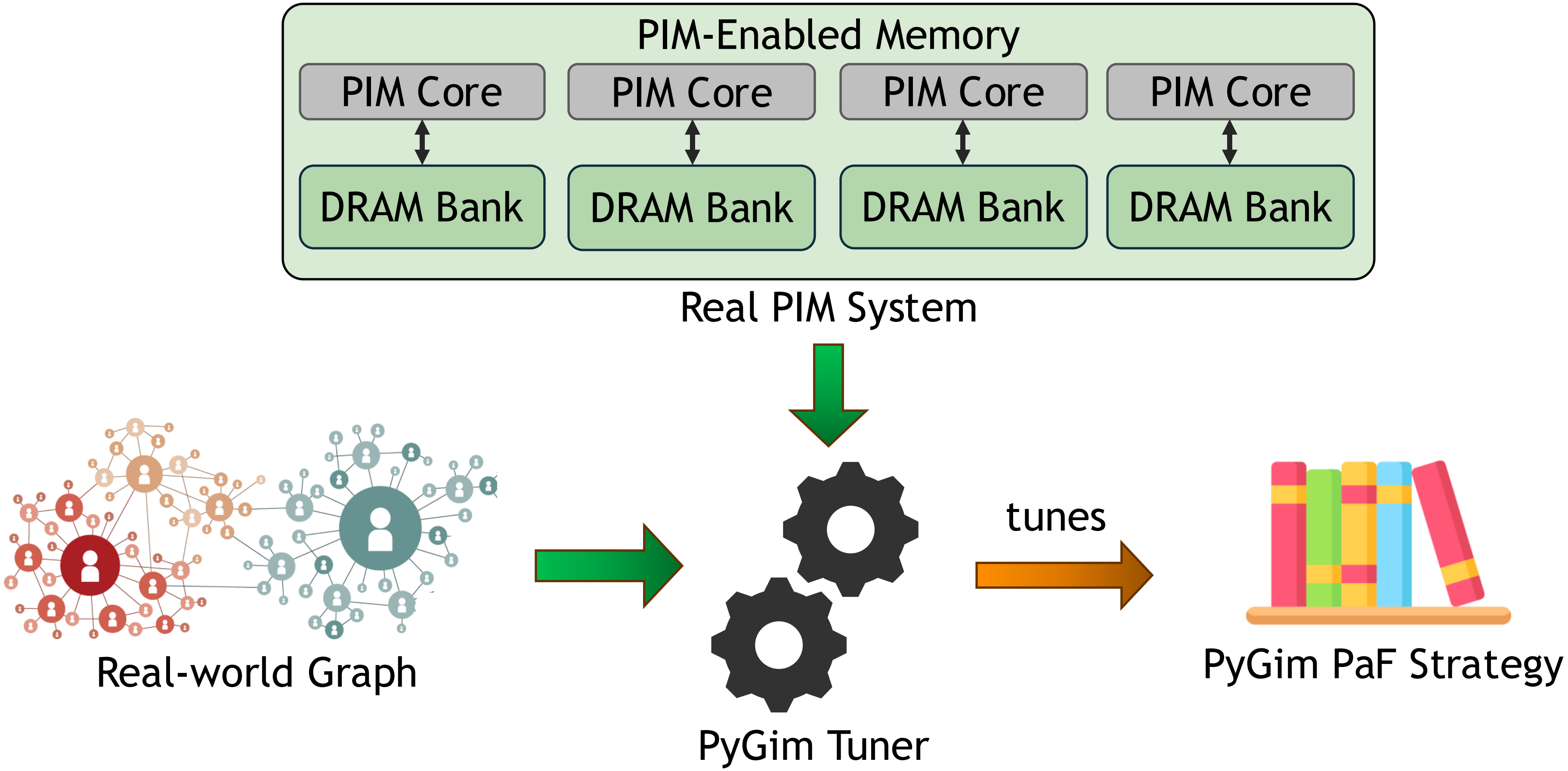


diagonal graph

real-world graphs with diverse characteristics

3. Lightweight Tuning

- PyGim Tuner predicts and ***automatically*** tunes the *best-performing* PaF *strategy* without the need for manual programmer intervention based on the:
  - Graph's characteristics (i.e., non-zero patterns)
  - PIM system's characteristics (i.e., compute capabilities, available memory bandwidth…)



4. Handy Programming Interface

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

```
import …    pygim as gyn
class GCNConv(torch.nn.Module):
def __init__(self, hidden_size):
  self.linear = torch.nn.Linear(feature_size, features_size)

def forward(self, graph_pim, in_dense):
 # Execute memory-intensive kernel in real PIM devices
  dense_parts = col_split(in_dense)
 out_dense = gyn.pim_run_aggr(graph_pim, dense_parts)
  # Execute compute-intensive operator in Host (e.g., CPU/GPU)
  out = self.linear(out_dense)
  return out

gyn.pim_init_devices(num_pim_devices) # Initialize PIM devices
data = load_dataset() # Load graph
 # Tune the PaF strategy
graph_pim= gyn.tune(data.graph, feature_size, device_info)
graph_pim = gyn.load_graph_pim(graph_parts) # Partition graph to PIM
 # Create GNN model
 model=torch.nn.Sequential([Linear(in_channels,feature_size),
     GCNConv(feature_size),
     GCNConv(feature_size),
     GCNConv(feature_size),
     Linear(feature_size, out_channels)])
model.forward(graph_pim, data.features) # GCN inference
```

4. Handy Programming Interface

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

```
import …    pygim as gyn
class GCNConv(torch.nn.Module):
def __init__(self, hidden_size):
  self.linear = torch.nn.Linear(feature_size, features_size)

def forward(self, graph_pim, in_dense):
 # Execute memory-intensive kernel in real PIM devices
  dense_parts = col_split(in_dense)
out_dense = gyn.pim_run_aggr(gr
  # Execute compute-intensive operator
  out = self.linear(out_dense)
  return out

# Create GNN model
model=torch.nn.Sequential([Linear(in_channels,feature_size),
    GCNConv(feature_size),
    GCNConv(feature_size),
    GCNConv(feature_size),
    Linear(feature_size, out_channels)])
model.forward(graph_pim, data.features) # GCN inference
```

Computation is performed inside **real** PIM devices!

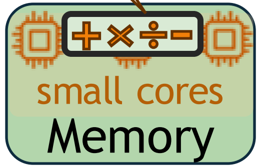
PyGim: github.com/CMU-SAFARI/PyGim

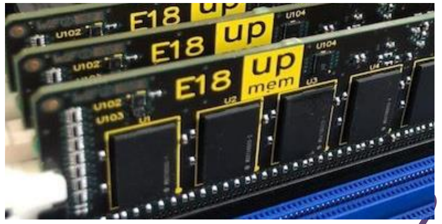
Deploy your GNNs effortless and enjoy the PIM benefits!

```
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
spmm_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





UPMEM PIM

Talk Outline

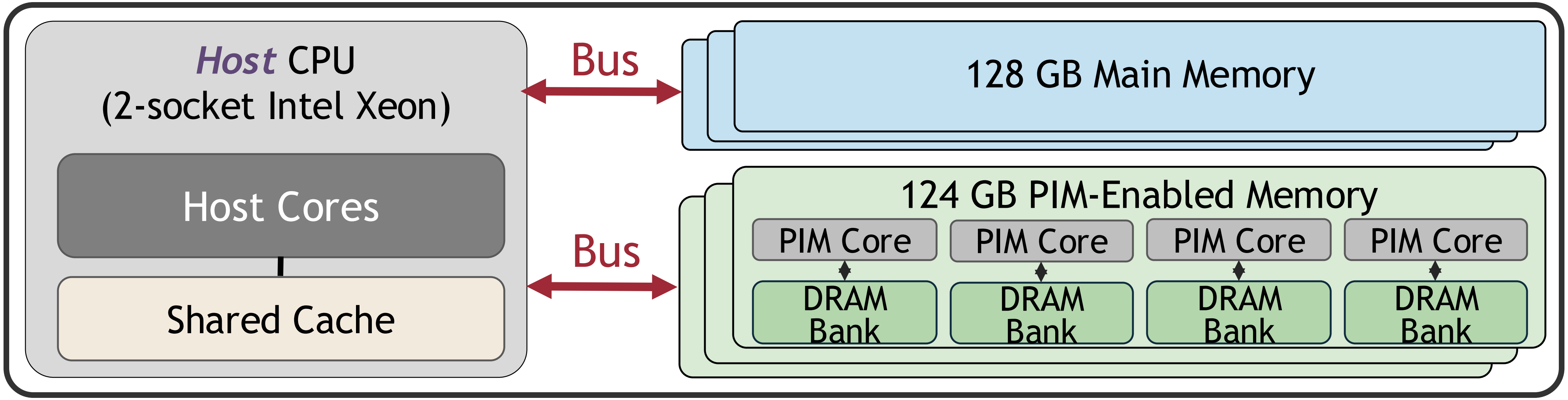
Background & Motivation

PyGim Design

Evaluation

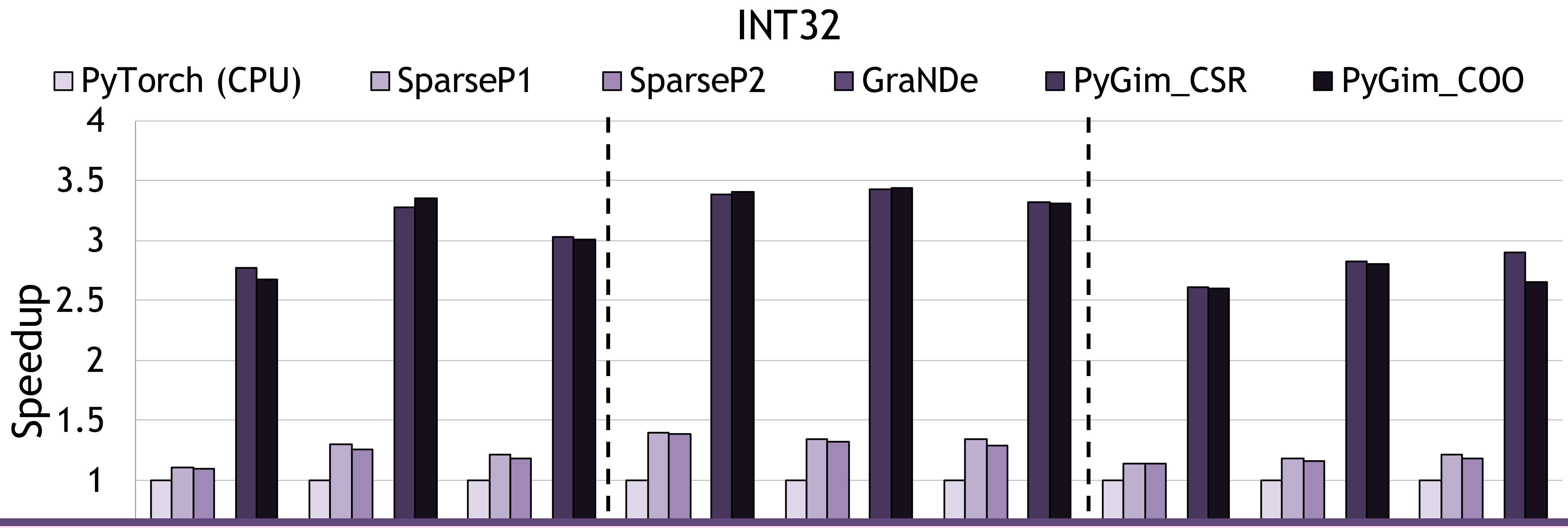
Evaluation Methodology

- UPMEM PIM server: 16 PIM DIMMs with *1992* PIM Cores (24 threads per core) in total
- Graph models: GCN, GIN SAGE
- Datasets: obn-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



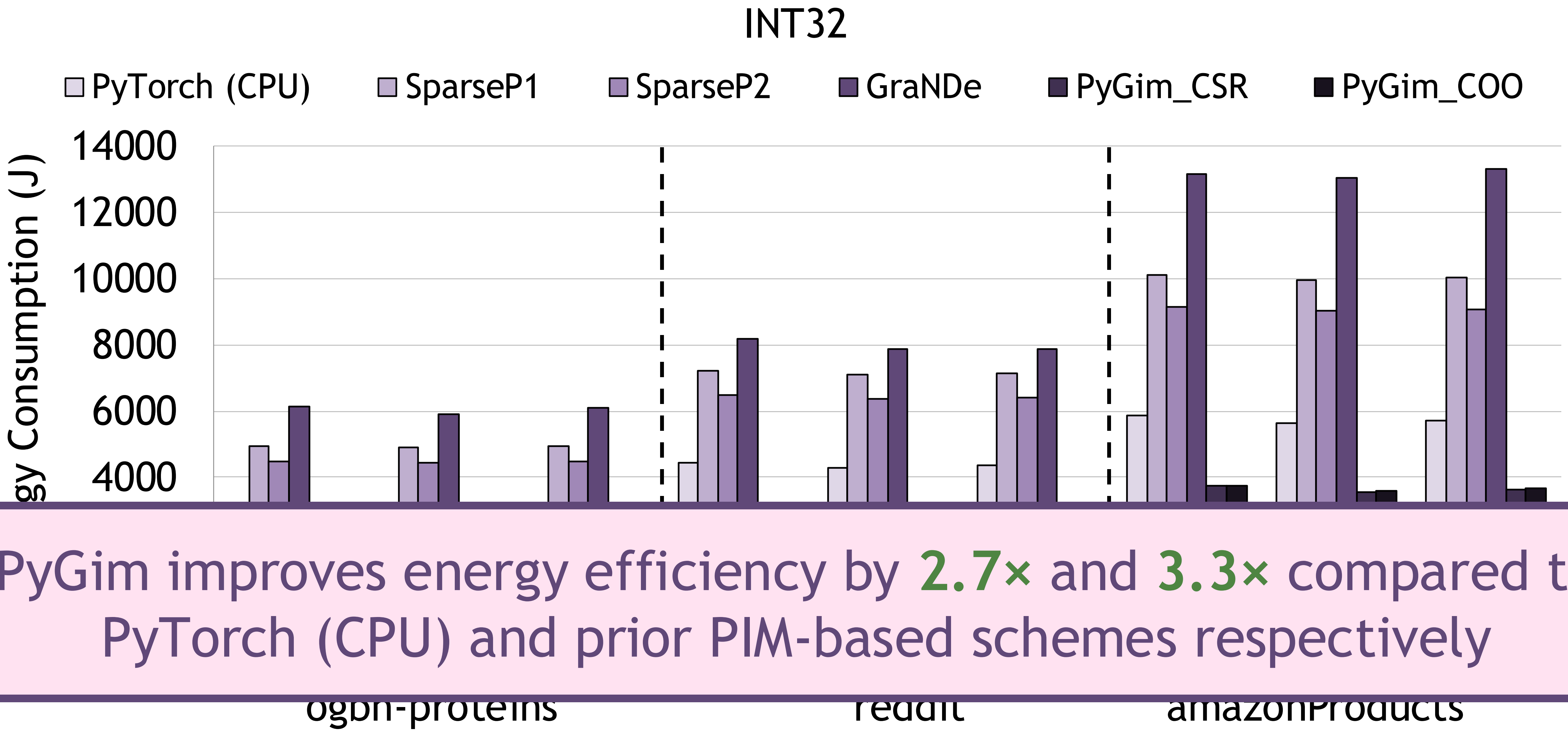
UPMEM PIM System

Performance Evaluation in GNN Inference



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

Energy Efficiency Evaluation in GNN Inference



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

Characteristics of CPU, PIM and GPU Systems

System	Total Cores	Freq.	INT32 Peak Performance	FP32 Peak Performance	Memory Capacity	Total Bandwidth	Technology Node
CPU Intel Xeon 4215	2×8 x86 cores	2.5 GHz	0.64 TOPS	1.28 TFLOPS	128 GB	23.1 GB/s	14nm
UPMEM PIM	1992 PIM cores	350 MHz	115.93 GOPS	24.85 GFLOPS	124.5 GB	1.39 TB/s	at least 20nm
GPU GTX 1080 Ti	3584 CUDA cores	1.48 GHz	13.25 TOPS	13.25 TFLOPS	11 GB	359.9 GB/s	16nm
GPU RTX 2080 Ti	4352 CUDA cores	1.35 GHz	16.94 TOPS	16.94 TFLOPS	11 GB	558.1 GB/s	12nm
GPU RTX 3090	10496 CUDA cores	1.40 GHz	17.79 TOPS	35.58 TFLOPS	24 GB	936.2 GB/s	8nm

Across last GPU generations:

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

Comparing latest GPU vs PIM:

- *GPU RTX 3090* provides **~150×** greater *compute throughput*
- *PIMs* provide only **~1.5×** larger *memory bandwidth*

Resource Utilization in GNN Aggregation

Dataset & data type/ Software library	OGBN INT32	RDT INT32	AMZ INT32	OGBN FP32	RDT FP32	AMZ FP32
pytorch_sparse - Intel MKL (CPU Intel Xeon 4215)	0.74%	0.63%	0.67%	0.26%	0.22%	0.20%
pytorch_sparse - CUDA (GPU GTX 1080 Ti)	2.15%	0.62%	0.71%	2.02%	0.62%	0.71%
pytorch_sparse - CUDA (GPU RTX 2080 Ti)	1.45%	0.68%	0.71%	1.45%	0.67%	0.71%
pytorch_sparse - CUDA (GPU RTX 3090)	3.03%	1.56%	1.32%	1.58%	0.78%	0.67%
PyGim (UPMEM PIM)	**14.09%**	**13.86%**	**12.32%**	**8.21%**	**9.13%**	**8.84%**

Although memory bandwidth and compute throughput have improved across GPU generations, resource utilization in GNN aggregation remains *similarly low* (less than **~3%**)

PyGim running on a real PIM system achieves significantly higher resource utilization than the state-of-the-art PyTorch library running on high-end GPUs (***at least a 9× increase***)

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

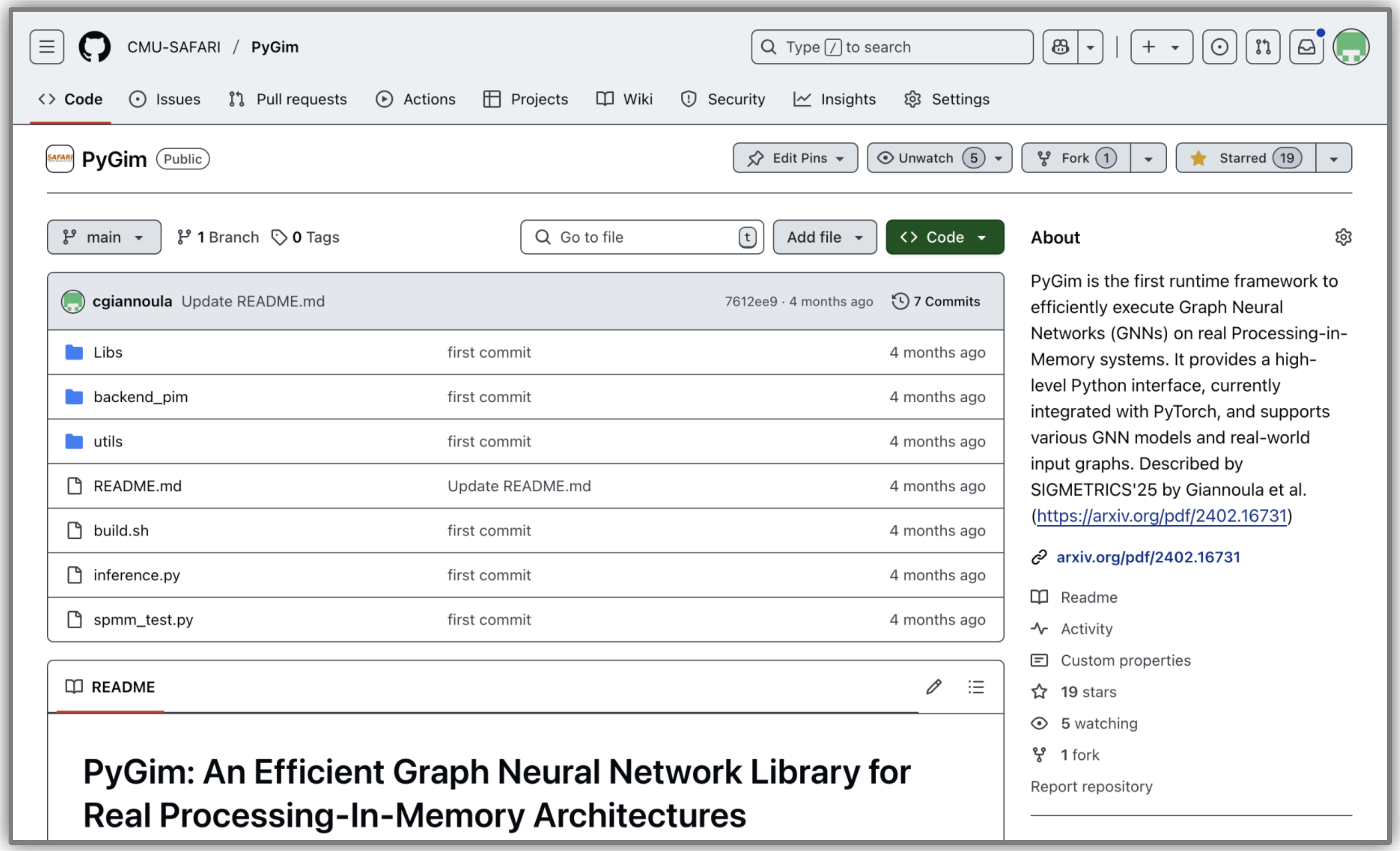
Graph Neural Networks (GNNs) are emerging models to analyze graph-structure data. The GNN execution involves both compute-intensive and memory-intensive kernels. The memory-intensive kernels dominate execution time, because they are significantly bottlenecked by data movement between memory and processors. 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

PyGim



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: PyGim runs *heterogeneous* kernels in *the best-fit* underlying *hardware* and *balances computation* and *data transfer* costs via configurable parallelization strategies for *diverse* real-world graphs. PyGim *automatically* tunes the best-fit strategy, enhancing both efficiency and ease of use *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) *resource utilization* on PIM system by **11.6×** over PyTorch on GPUs

PyGim github.com/CMU-SAFARI/PyGim

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

Christina Giannoula
https://cgiannoula.github.io/

Thank you!