MICRO 2023 Tutorial

Real-world Processing-in-Memory Systems for Modern Workloads

Accelerating Modern Workloads

on a General-purpose PIM System

Dr. Juan Gómez Luna

Professor Onur Mutlu

ETH zürich

SAFARI

Sunday, October 29, 2023

Potential Barriers to Adoption of PIM

1. **Applications** & **software** for PIM

2. Ease of **programming** (interfaces and compiler/HW support)

3. **System** and **security** support: coherence, synchronization, virtual memory, isolation, communication interfaces, …

4. **Runtime** and **compilation** systems for adaptive scheduling, data mapping, access/sharing control, …

5. **Infrastructures** to assess benefits and feasibility

All can be solved with change of mindset

Benchmarking and Workload Suitability

PrIM Benchmarks

- Goal
  - A common set of workloads that can be used to
    - evaluate the UPMEM PIM architecture,
    - compare software improvements and compilers,
    - compare future PIM architectures and hardware

- Two key selection criteria:
  - Selected workloads from different application domains
  - Memory-bound workloads on processor-centric architectures

- 14 different workloads, 16 different benchmarks*

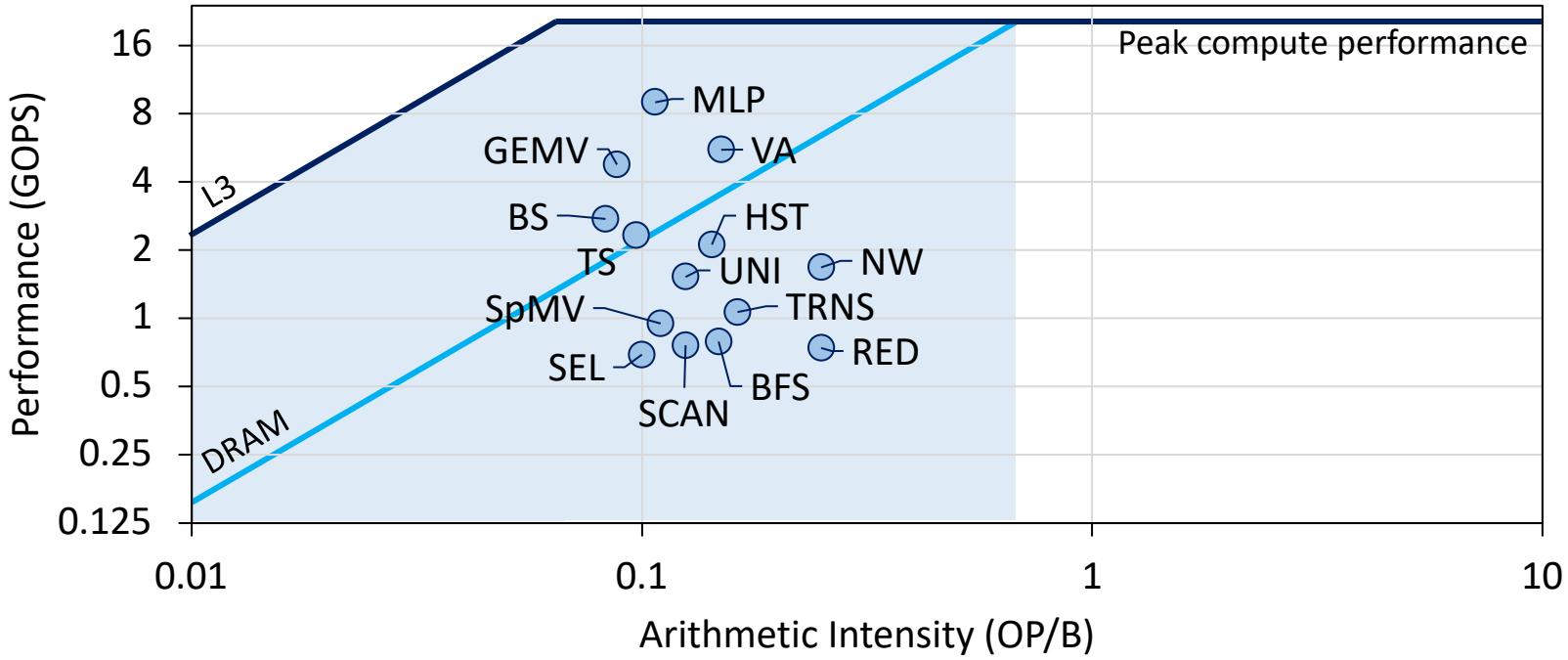
*There are two versions for two of the workloads (HST, SCAN).

PrIM Benchmarks: Application Domains

Domain	Benchmark	Short name
Dense linear algebra	Vector Addition	VA
	Matrix-Vector Multiply	GEMV
Sparse linear algebra	Sparse Matrix-Vector Multiply	SpMV
Databases	Select	SEL
	Unique	UNI
Data analytics	Binary Search	BS
	Time Series Analysis	TS
Graph processing	Breadth-First Search	BFS
Neural networks	Multilayer Perceptron	MLP
Bioinformatics	Needleman-Wunsch	NW
Image processing	Image histogram (short)	HST-S
	Image histogram (large)	HST-L
Parallel primitives	Reduction	RED
	Prefix sum (scan-scan-add)	SCAN-SSA
	Prefix sum (reduce-scan-scan)	SCAN-RSS
	Matrix transposition	TRNS

Roofline Model

- Intel Advisor on an Intel Xeon E3-1225 v6 CPU



All workloads fall in the memory-bound area of the Roofline

PrIM Benchmarks: Diversity

- PrIM benchmarks are diverse:
  - Memory access patterns
  - Operations and datatypes
  - Communication/synchronization

Domain	Benchmark	Short name	Memory access pattern			Computation pattern		Communication/synchronization	
			Sequential	Strided	Random	Operations	Datatype	Intra-DPU	Inter-DPU
Dense linear algebra	Vector Addition	VA	Yes			add	int32_t		
	Matrix-Vector Multiply	GEMV	Yes			add, mul	uint32_t		
Sparse linear algebra	Sparse Matrix-Vector Multiply	SpMV	Yes		Yes	add, mul	float		
Databases	Select	SEL	Yes			add, compare	int64_t	handshake, barrier	Yes
	Unique	UNI	Yes			add, compare	int64_t	handshake, barrier	Yes
Data analytics	Binary Search	BS	Yes		Yes	compare	int64_t		
	Time Series Analysis	TS	Yes			add, sub, mul, div	int32_t		
Graph processing	Breadth-First Search	BFS	Yes		Yes	bitwise logic	uint64_t	barrier, mutex	Yes
Neural networks	Multilayer Perceptron	MLP	Yes			add, mul, compare	int32_t		
Bioinformatics	Needleman-Wunsch	NW	Yes	Yes		add, sub, compare	int32_t	barrier	Yes
Image processing	Image histogram (short)	HST-S	Yes		Yes	add	uint32_t	barrier	Yes
	Image histogram (long)	HST-L	Yes		Yes	add	uint32_t	barrier, mutex	Yes
Parallel primitives	Reduction	RED	Yes	Yes		add	int64_t	barrier	Yes
	Prefix sum (scan-scan-add)	SCAN-SSA	Yes			add	int64_t	handshake, barrier	Yes
	Prefix sum (reduce-scan-scan)	SCAN-RSS	Yes			add	int64_t	handshake, barrier	Yes
	Matrix transposition	TRNS	Yes		Yes	add, sub, mul	int64_t	mutex	

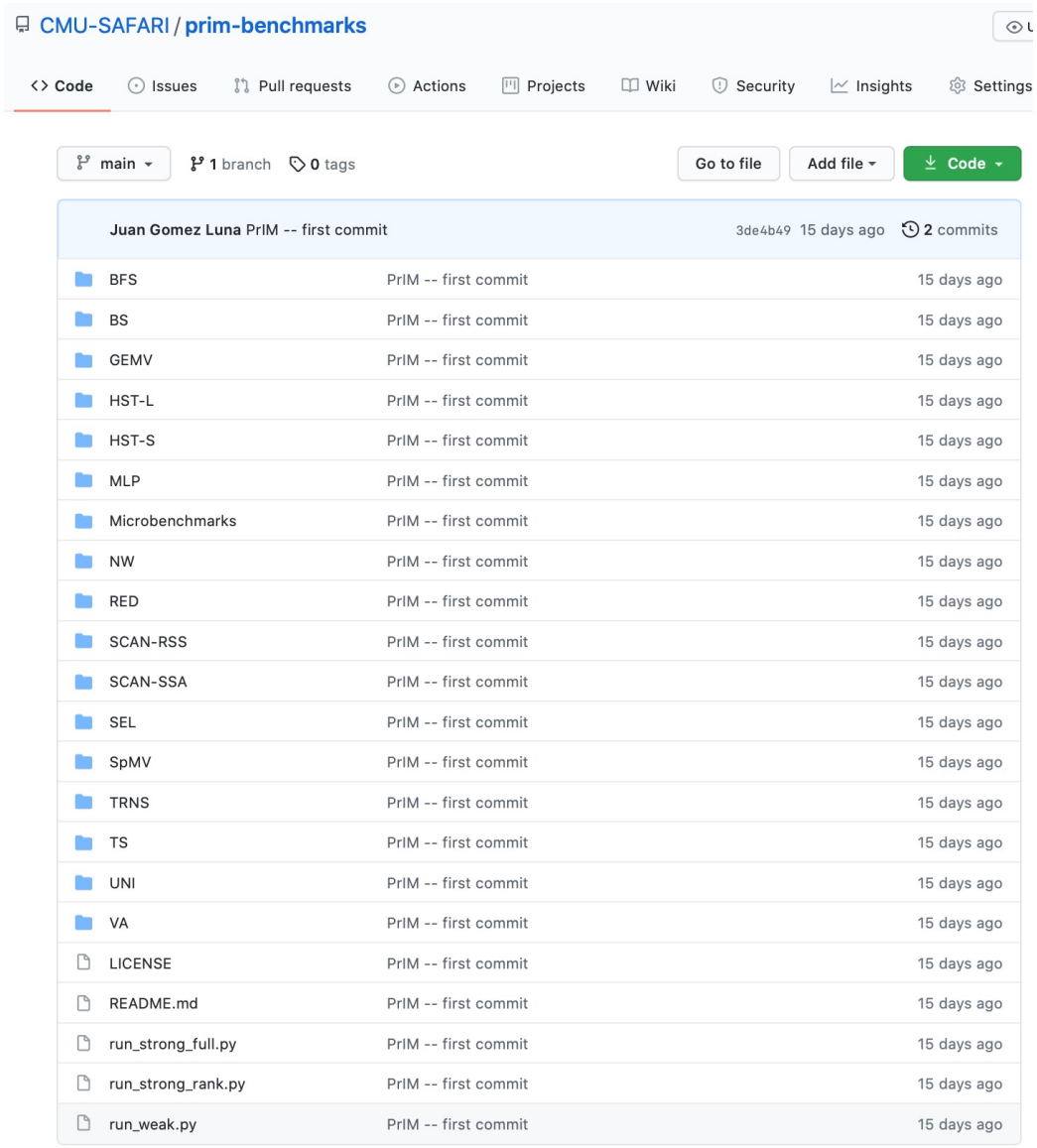
PrIM Benchmarks: Inter-DPU Communication

Domain	Benchmark	Short name	Memory access pattern			Computation pattern		Communication/synchronization	
			Sequential	Strided	Random	Operations	Datatype	Intra-DPU	Inter-DPU
Dense linear algebra	Vector Addition	VA	Yes			add	int32_t		
	Matrix-Vector Multiply	GEMV	Yes			add, mul	uint32_t		
Sparse linear algebra	Sparse Matrix-Vector Multiply	SpMV	Yes		Yes	add, mul	float		
Databases	Select	SEL	Yes			add, compare	int64_t	handshake, barrier	Yes
	Unique	UNI	Yes			add, compare	int64_t	handshake, barrier	Yes
Data analytics	Binary Search	BS	Yes		Yes	compare	int64_t		
	Time Series Analysis	TS	Yes			add, sub, mul, div	int32_t		
Graph processing	Breadth-First Search	BFS	Yes		Yes	bitwise logic	uint64_t	barrier, mutex	Yes
Neural networks	Multilayer Perceptron	MLP	Yes			add, mul, compare	int32_t		
Bioinformatics	Needleman-Wunsch	NW	Yes	Yes		add, sub, compare	int32_t	barrier	Yes
Image processing	Image histogram (short)	HST-S	Yes		Yes	add	uint32_t	barrier	Yes
	Image histogram (long)	HST-L	Yes		Yes	add	uint32_t	barrier, mutex	Yes
Parallel primitives	Reduction	RED	Yes	Yes		add	int64_t	barrier	Yes
	Prefix sum (scan-scan-add)	SCAN-SSA	Yes			add	int64_t	handshake, barrier	Yes
	Prefix sum (reduce-scan-scan)	SCAN-RSS	Yes			add	int64_t	handshake, barrier	Yes
	Matrix transposition	TRNS	Yes		Yes	add, sub, mul	int64_t	mutex	

- Inter-DPU communication
  - Result merging:
    - SEL, UNI, HST-S, HST-L, RED
  - Only DPU-CPU transfers
  - Redistribution of intermediate results:
    - BFS, MLP, NW, SCAN-SSA, SCAN-RSS
  - DPU-CPU and CPU-DPU transfers

PrIM Benchmarks

- 16 benchmarks and scripts for evaluation
- https://github.com/CMU-SAFARI/prim-benchmarks



Outline

- Introduction
  - Accelerator Model
  - UPMEM-based PIM System Overview
- UPMEM PIM Programming
  - Vector Addition
  - CPU-DPU Data Transfers
  - Inter-DPU Communication
  - CPU-DPU/DPU-CPU Transfer Bandwidth
- DRAM Processing Unit
  - Arithmetic Throughput
  - WRAM and MRAM Bandwidth
- PrIM Benchmarks
  - Roofline Model
  - Benchmark Diversity
- Evaluation
  - Strong and Weak Scaling
  - Comparison to CPU and GPU
- Key Takeaways

Evaluation Methodology

- We evaluate the 16 PrIM benchmarks on two UPMEM-based systems:
  - 2,556-DPU system
  - 640-DPU system
- Strong and weak scaling experiments on the 2,556-DPU system
  - 1 DPU with different numbers of tasklets
  - 1 rank (strong and weak)
  - Up to 32 ranks

Strong scaling refers to how the execution time of a program solving a particular problem varies with the number of processors for a fixed problem size

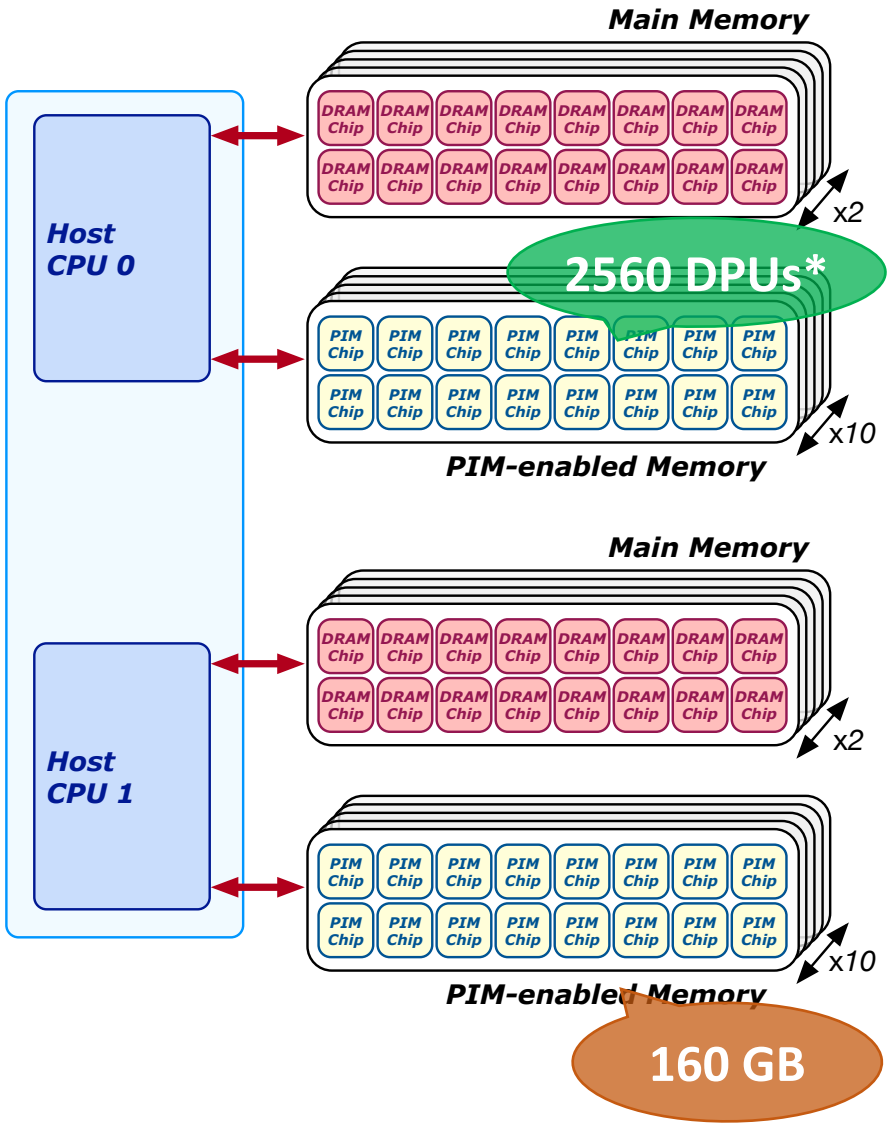
Weak scaling refers to how the execution time of a program solving a particular problem varies with the number of processors for a fixed problem size per processor

Evaluation Methodology

- We evaluate the 16 PrIM benchmarks on two UPMEM-based systems:
  - 2,556-DPU system
  - 640-DPU system
- Strong and weak scaling experiments on the 2,556-DPU system
  - 1 DPU with different numbers of tasklets
  - 1 rank (strong and weak)
  - Up to 32 ranks
- Comparison of both UPMEM-based PIM systems to state-of-the-art CPU and GPU
  - Intel Xeon E3-1240 CPU
  - NVIDIA Titan V GPU

2,560-DPU System

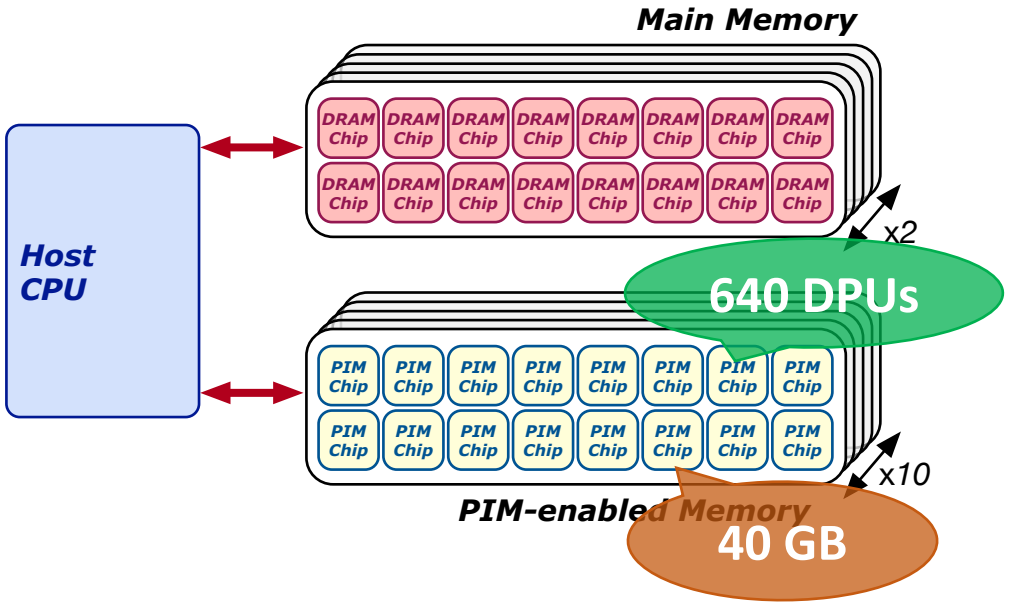
- UPMEM-based PIM system with 20 UPMEM DIMMs of 16 chips each (40 ranks)
  - P21 DIMMs
  - Dual x86 socket
    - UPMEM DIMMs coexist with regular DDR4 DIMMs
    - 2 memory controllers/socket (3 channels each)
    - 2 conventional DDR4 DIMMs on one channel of one controller



* There are 4 faulty DPUs in the system that we use in our experiments. Thus, the maximum number of DPUs we can use is 2,556.

640-DPU System

- UPMEM-based PIM system with 10 UPMEM DIMMs of 8 chips each (10 ranks)
  - E19 DIMMs
  - x86 socket
    - 2 memory controllers (3 channels each)
    - 2 conventional DDR4 DIMMs on one channel of one controller



Datasets

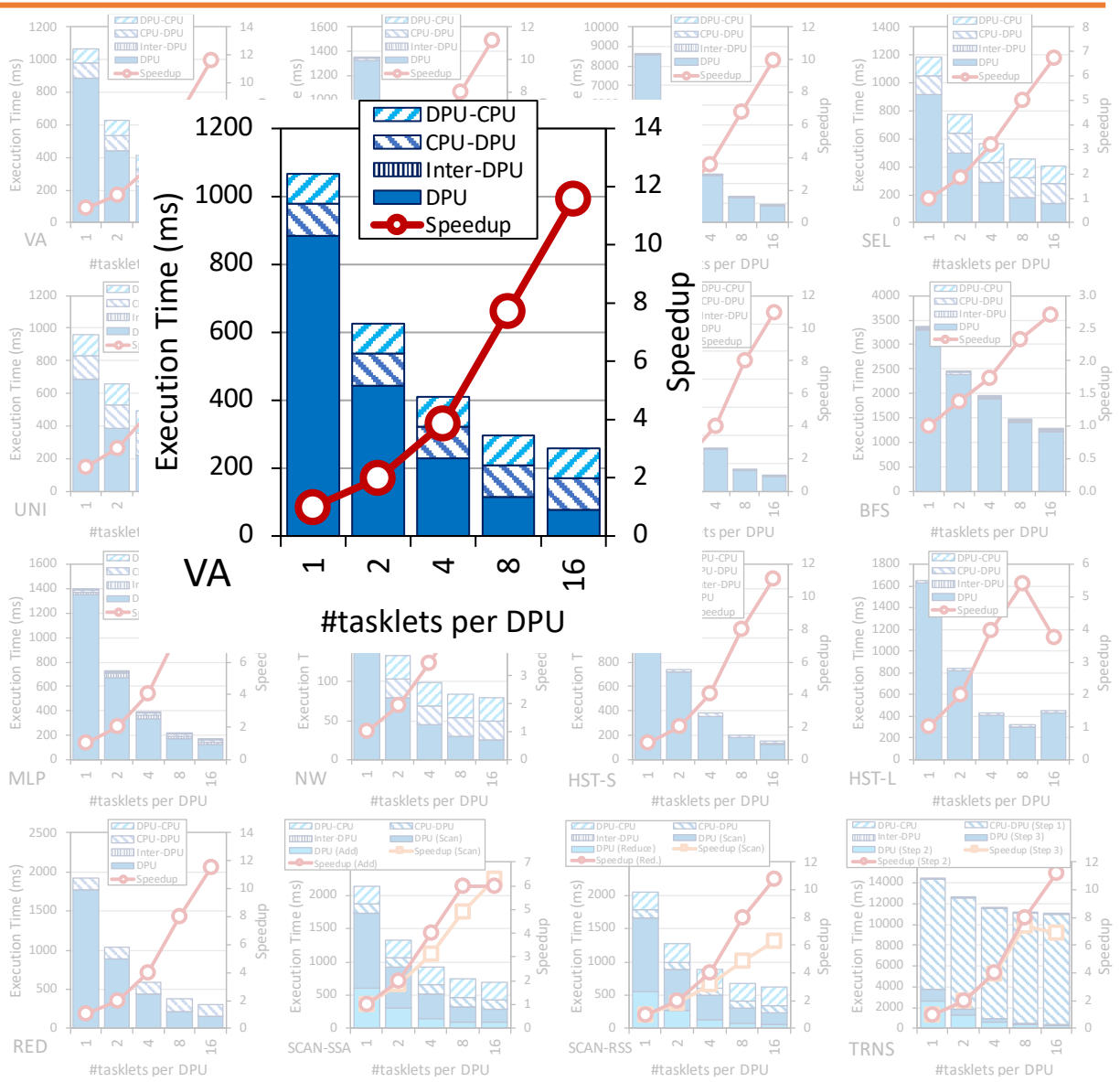
- Strong and weak scaling experiments

Benchmark	Strong Scaling Dataset	Weak Scaling Dataset	MRAM-WRAM Transfer Sizes
VA	1 DPU-1 rank: 2.5M elem. (10 MB) | 32 ranks: 160M elem. (640 MB)	2.5M elem./DPU (10 MB)	1024 bytes
GEMV	1 DPU-1 rank: 8192 × 1024 elem. (32 MB) | 32 ranks: 163840 × 4096 elem. (2.56 GB)	1024 × 2048 elem./DPU (8 MB)	1024 bytes
SpMV	*bcsstk30* [253] (12 MB)	*bcsstk30* [253]	64 bytes
SEL	1 DPU-1 rank: 3.8M elem. (30 MB) | 32 ranks: 240M elem. (1.9 GB)	3.8M elem./DPU (30 MB)	1024 bytes
UNI	1 DPU-1 rank: 3.8M elem. (30 MB) | 32 ranks: 240M elem. (1.9 GB)	3.8M elem./DPU (30 MB)	1024 bytes
BS	2M elem. (16 MB). 1 DPU-1 rank: 256K queries. (2 MB) | 32 ranks: 16M queries. (128 MB)	2M elem. (16 MB). 256K queries./DPU (2 MB).	8 bytes
TS	256 elem. query. 1 DPU-1 rank: 512K elem. (2 MB) | 32 ranks: 32M elem. (128 MB)	512K elem./DPU (2 MB)	256 bytes
BFS	*loc-gowalla* [254] (22 MB)	*rMat* [255] (≈100K vertices and 1.2*M* edges per DPU)	8 bytes
MLP	3 fully-connected layers. 1 DPU-1 rank: 2K neurons (32 MB) | 32 ranks: ≈160K neur. (2.56 GB)	3 fully-connected layers. 1K neur./DPU (4 MB)	1024 bytes
NW	1 DPU-1 rank: 2560 bps (50 MB), large/small sub-block=$\frac{2560}{\#DPUs}/2$ | 32 ranks: 64K bps (32 GB), l./s.=32/2	512 bps/DPU (2MB), l./s.=512/2	8, 16, 32, 40 bytes
HST-S	1 DPU-1 rank: 1536 × 1024 input image [256] (6 MB) | 32 ranks: 64 × input image	1536 × 1024 input image [256]/DPU (6 MB)	1024 bytes
HST-L	1 DPU-1 rank: 1536 × 1024 input image [256] (6 MB) | 32 ranks: 64 × input image	1536 × 1024 input image [256]/DPU (6 MB)	1024 bytes
RED	1 DPU-1 rank: 6.3M elem. (50 MB) | 32 ranks: 400M elem. (3.1 GB)	6.3M elem./DPU (50 MB)	1024 bytes
SCAN-SSA	1 DPU-1 rank: 3.8M elem. (30 MB) | 32 ranks: 240M elem. (1.9 GB)	3.8M elem./DPU (30 MB)	1024 bytes
SCAN-RSS	1 DPU-1 rank: 3.8M elem. (30 MB) | 32 ranks: 240M elem. (1.9 GB)	3.8M elem./DPU (30 MB)	1024 bytes
TRNS	1 DPU-1 rank: 12288 × 16 × 64 × 8 (768 MB) | 32 ranks: 12288 × 16 × 2048 × 8 (24 GB)	12288 × 16 × 1 × 8/DPU (12 MB)	128, 1024 bytes

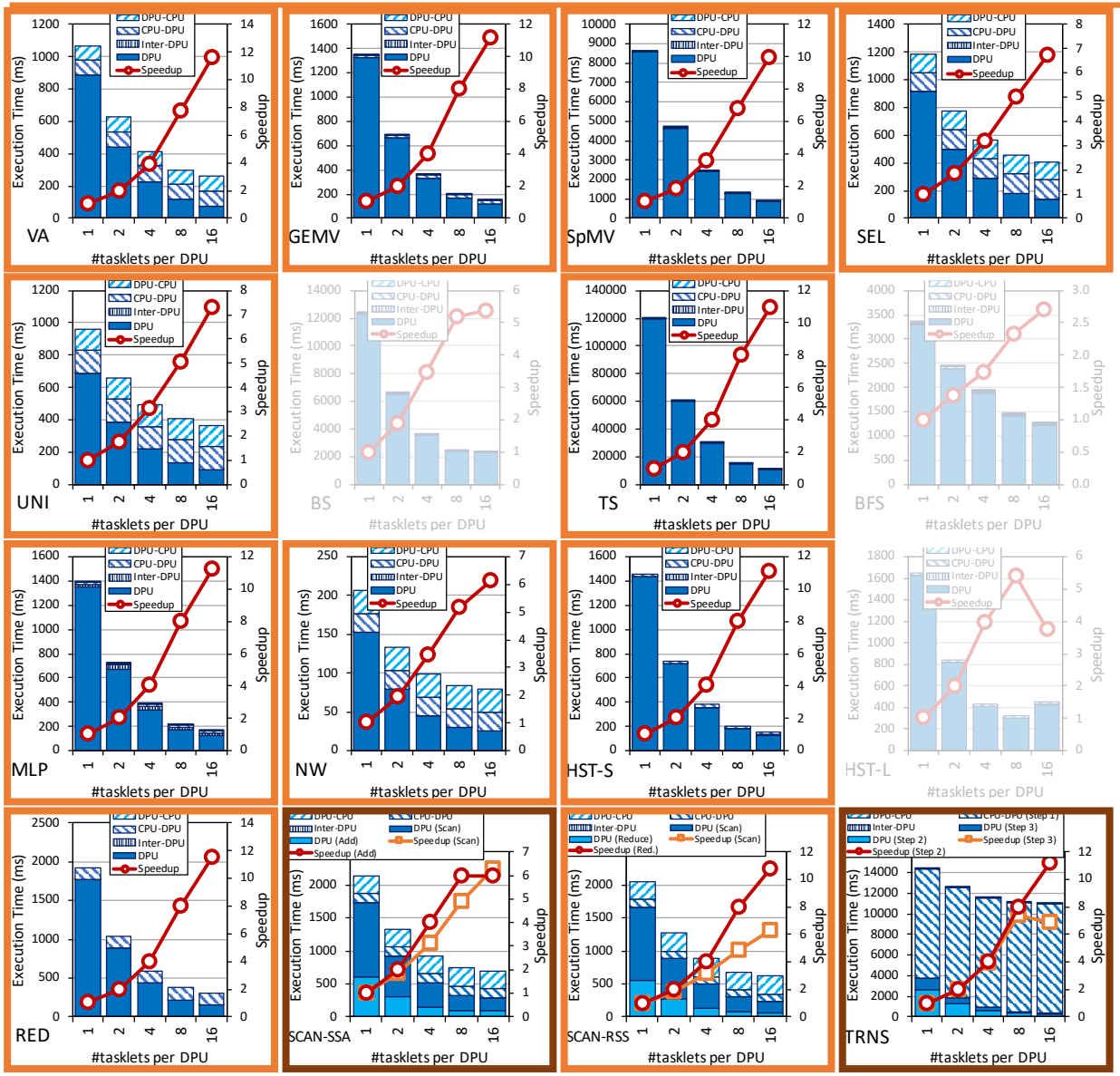
The PrIM benchmarks repository includes
all datasets and scripts used in our evaluation
https://github.com/CMU-SAFARI/prim-benchmarks

Strong Scaling: 1 DPU (I)

- Strong scaling experiments on 1 DPU
  - We set the number of tasklets to 1, 2, 4, 8, and 16
  - We show the breakdown of execution time:
    - DPU: Execution time on the DPU
    - Inter-DPU: Time for inter-DPU communication via the host CPU
    - CPU-DPU: Time for CPU to DPU transfer of input data
    - DPU-CPU: Time for DPU to CPU transfer of final results
  - Speedup over 1 tasklet



Strong Scaling: 1 DPU (II)



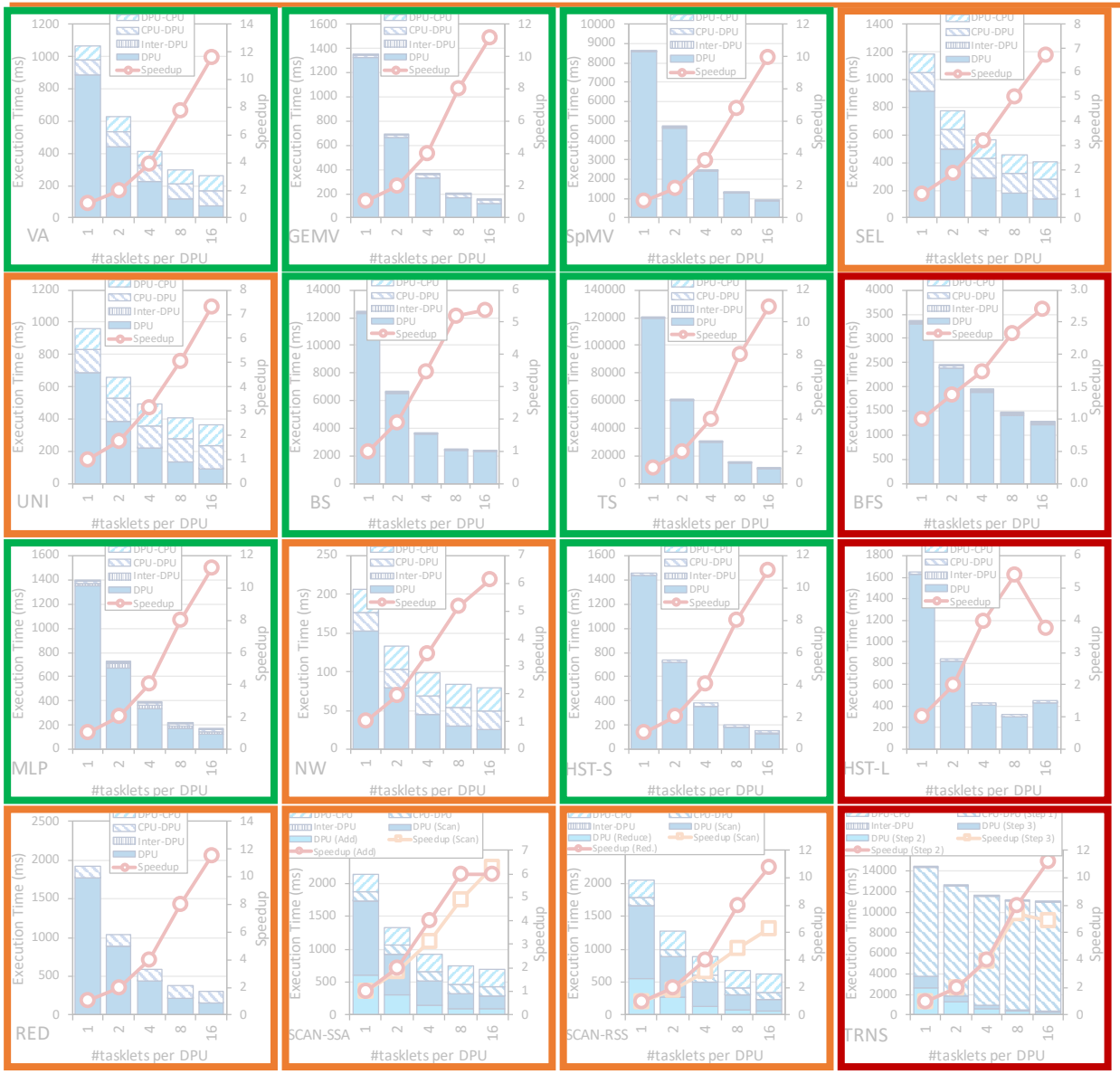
VA, GEMV, SpMV, SEL, UNI, TS, MLP, NW, HST-S, RED, SCAN-SSA (Scan kernel), SCAN-RSS (both kernels), and TRNS (Step 2 kernel), the best performing number of tasklets is 16

Speedups 1.5-2.0x as we double the number of tasklets from 1 to 8. Speedups 1.2-1.5x from 8 to 16, since the pipeline throughput saturates at 11 tasklets

KEY OBSERVATION 10

A number of tasklets greater than 11 is a good choice for most real-world workloads we tested (16 kernels out of 19 kernels from 16 benchmarks), as it fully utilizes the DPU's pipeline.

Strong Scaling: 1 DPU (III)

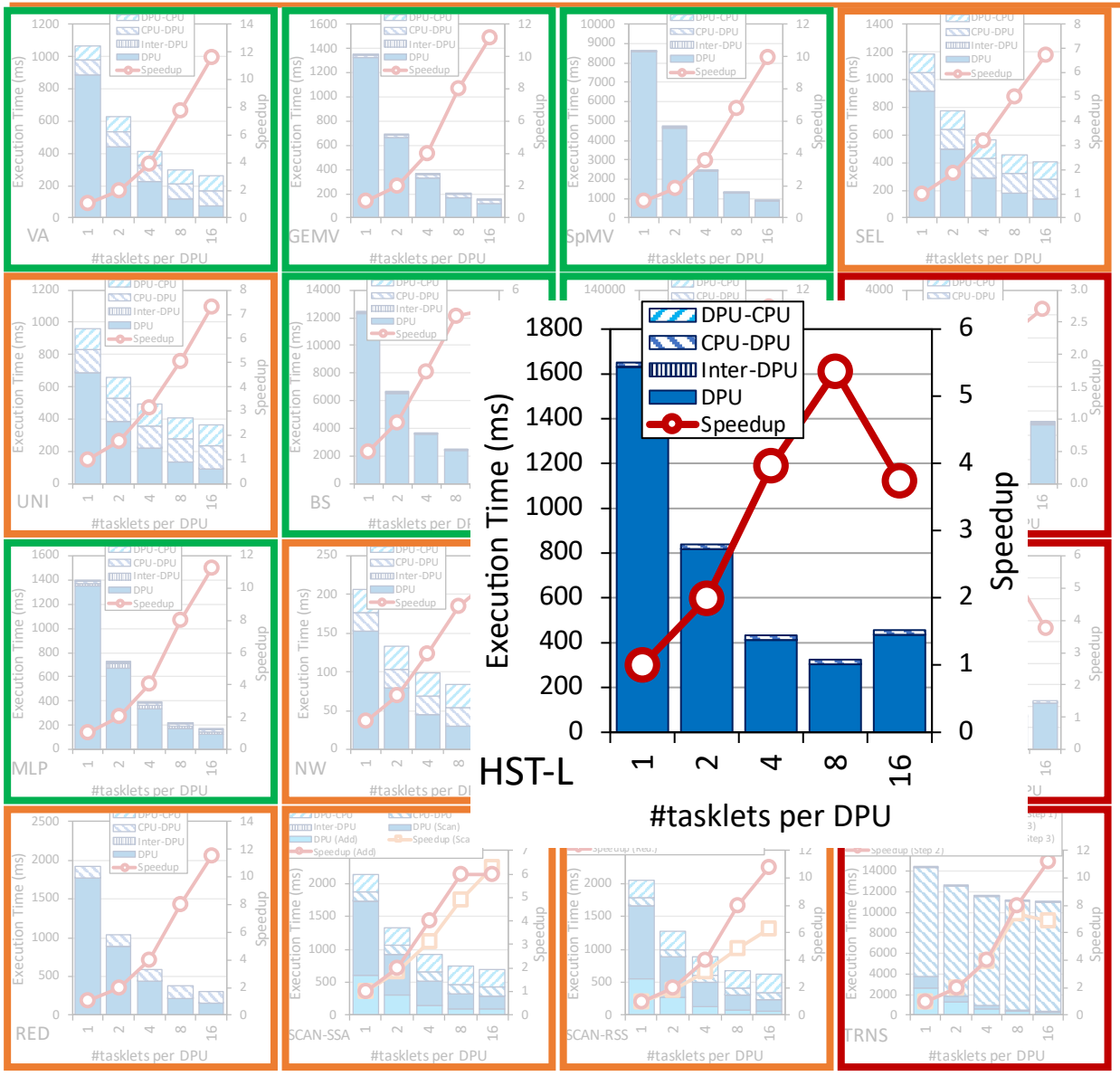


VA, GEMV, SpMV, BS, TS, MLP, HST-S do not use intra-DPU synchronization primitives

In SEL, UNI, NW, RED, SCAN-SSA (Scan kernel), SCAN-RSS (both kernels), synchronization is lightweight

BFS, HST-L, TRNS (Step 3) use mutexes, which cause contention when accessing shared data structures

Strong Scaling: 1 DPU (IV)



VA, GEMV, SpMV, BS, TS, MLP, HST-S do not use intra-DPU synchronization primitives

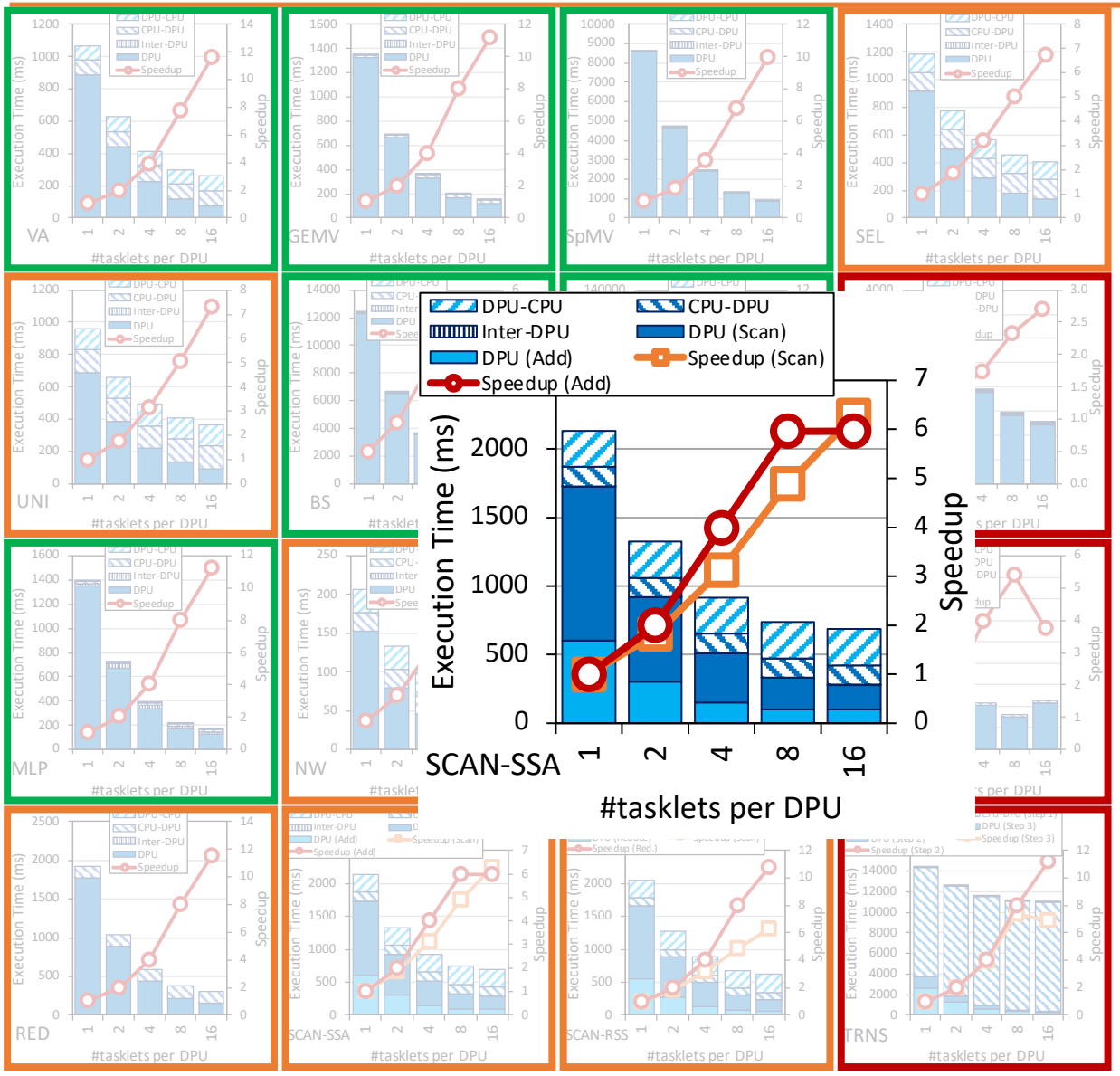
In SEL, UNI, NW, RED, SCAN-SSA (Scan kernel), SCAN-RSS (both kernels), synchronization is lightweight

BFS, HST-L, TRNS (Step 3) use mutexes, which cause contention when accessing shared data structures

KEY OBSERVATION 11

Intensive use of **intra-DPU synchronization across tasklets (e.g., mutexes, barriers, handshakes) may limit scalability**, sometimes causing the best performing number of tasklets to be lower than 11.

Strong Scaling: 1 DPU (V)

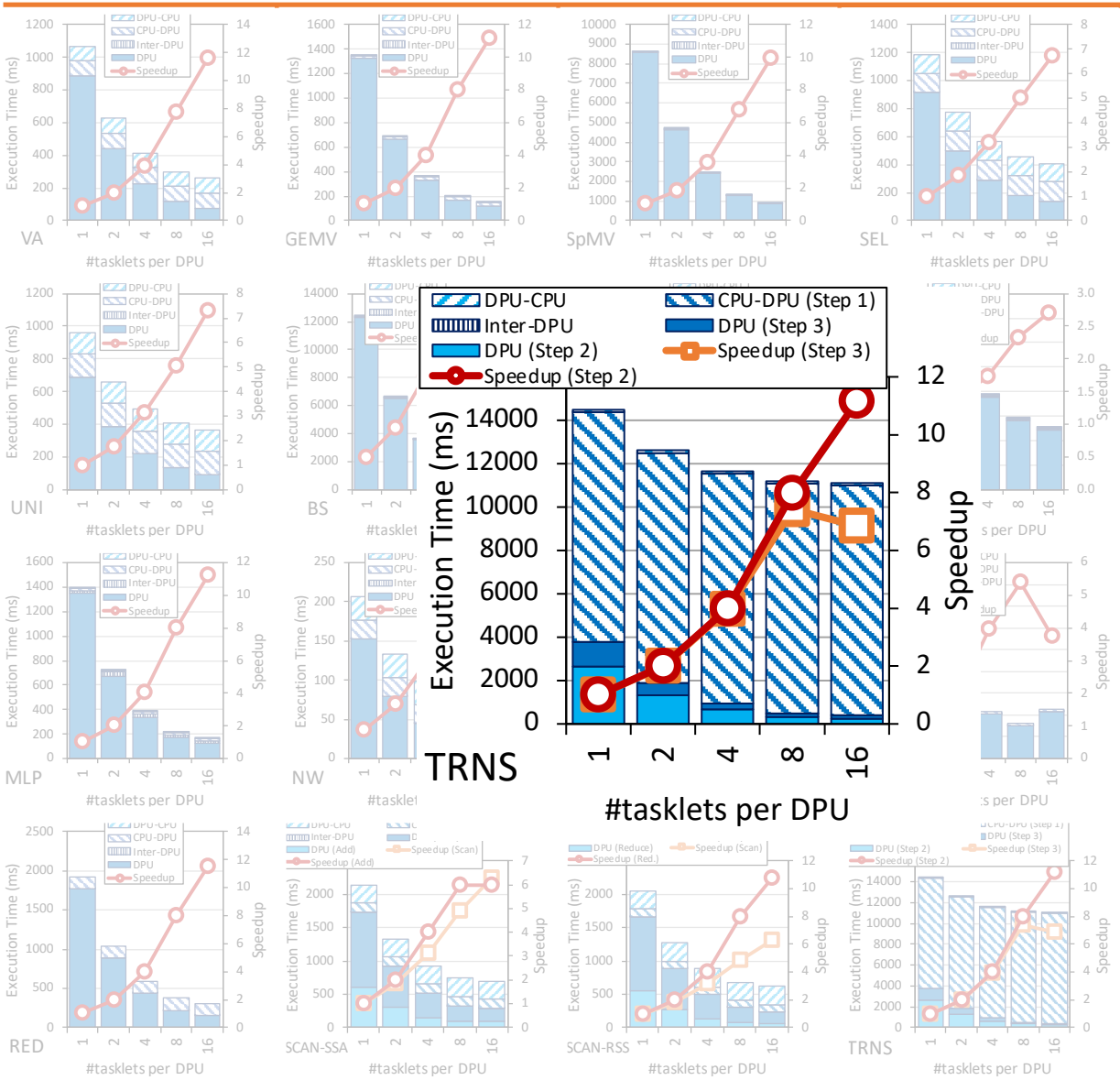


SCAN-SSA (Add kernel) is not compute-intensive. Thus, performance saturates with less that 11 tasklets (recall STREAM ADD).
BS shows similar behavior

KEY OBSERVATION 12

Most real-world workloads are in the compute-bound region of the DPU (all kernels except SCAN-SSA (Add kernel) and BS), i.e., the pipeline latency dominates the MRAM access latency.

Strong Scaling: 1 DPU (VI)



The amount of time spent on CPU-DPU and DPU-CPU transfers is low compared to the time spent on DPU execution

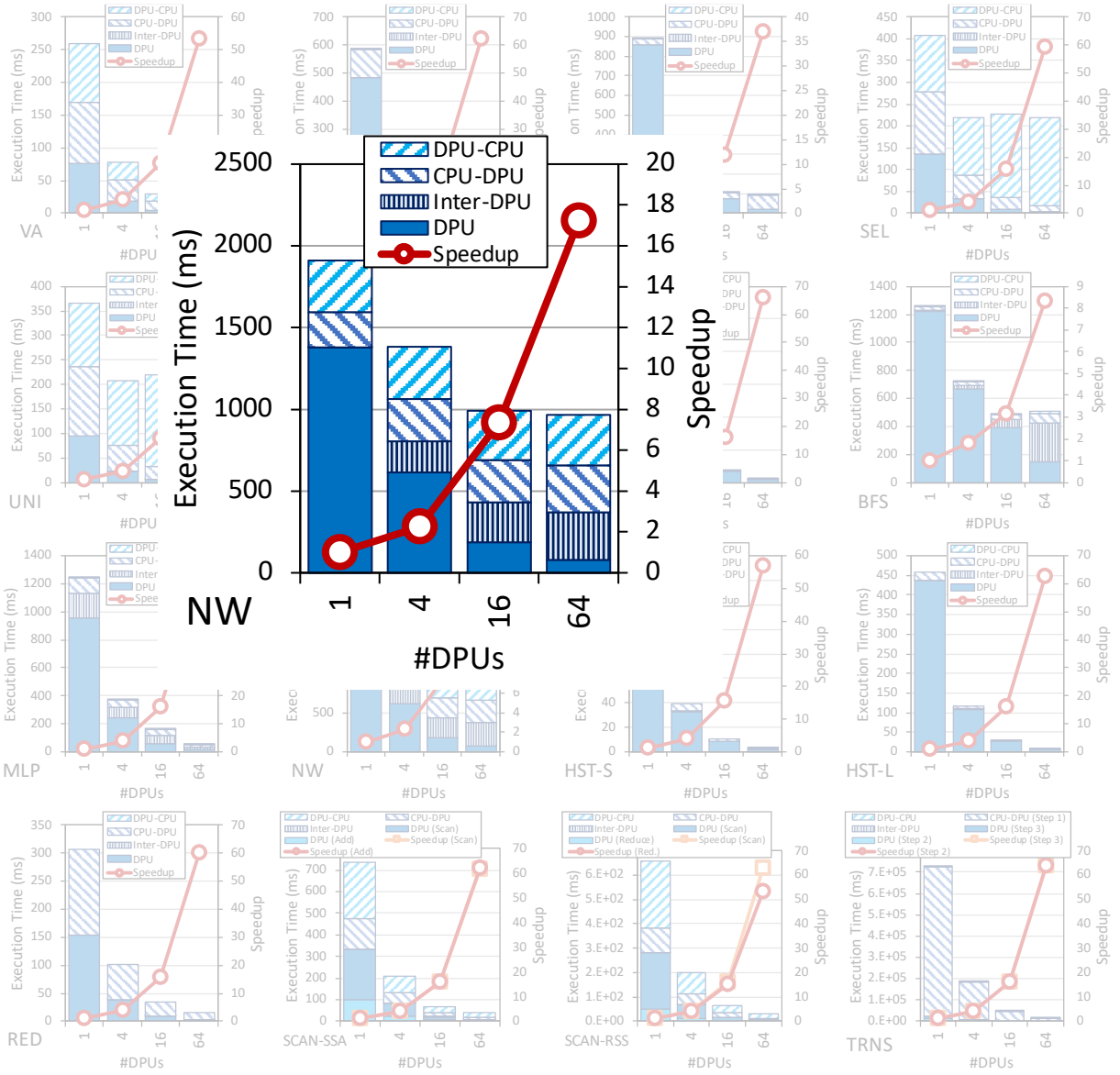
TRNS performs step 1 of the matrix transposition via the CPU-DPU transfer.
Using small transfers (8 elements) does not exploit full CPU-DPU bandwidth

KEY OBSERVATION 13

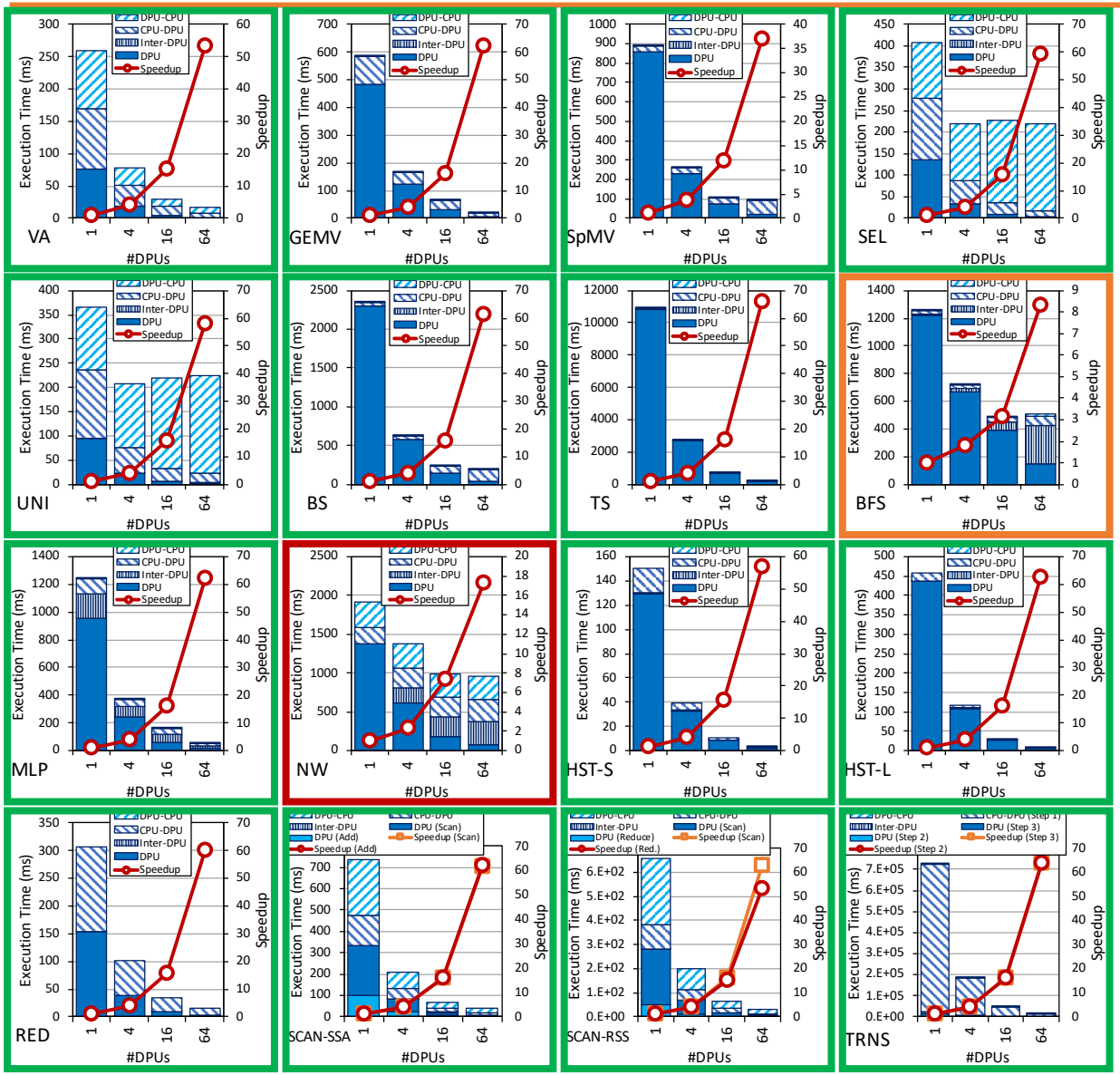
Transferring large data chunks from/to the host CPU is preferred for input data and output results due to higher sustained CPU-DPU/DPU-CPU bandwidths.

Strong Scaling: 1 Rank (I)

- Strong scaling experiments on 1 rank
  - We set the number of tasklets to the best performing one
  - The number of DPUs is 1, 4, 16, 64
  - We show the breakdown of execution time:
    - DPU: Execution time on the DPU
    - Inter-DPU: Time for inter-DPU communication via the host CPU
    - CPU-DPU: Time for CPU to DPU transfer of input data
    - DPU-CPU: Time for DPU to CPU transfer of final results
  - Speedup over 1 DPU



Strong Scaling: 1 Rank (II)



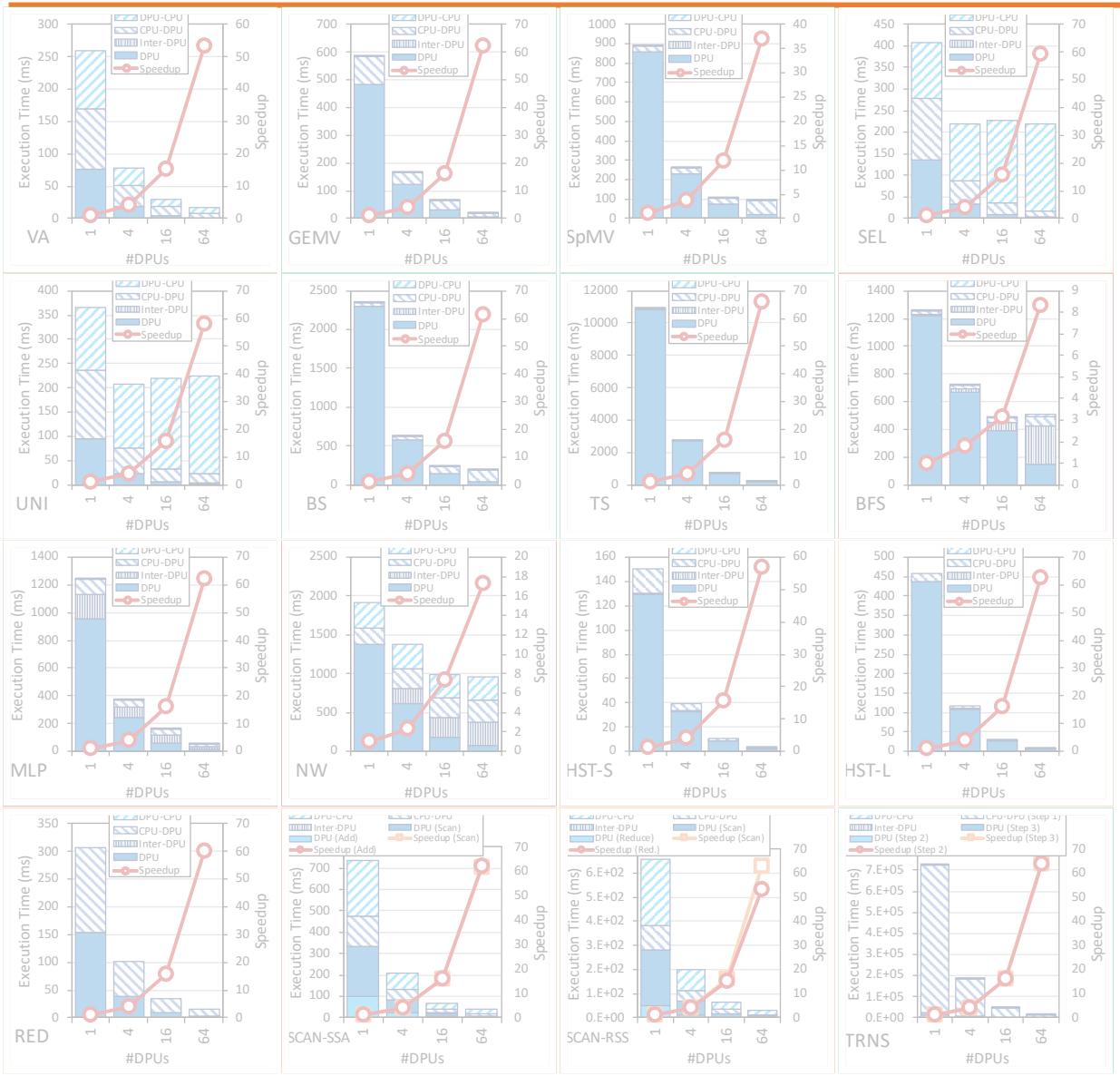
VA, GEMV, SpMV, SEL, UNI, BS, TS, MLP, HST-S, HST-L, RED, SCAN-SSA (both kernel), SCAN-RSS (both kernels), and TRNS (both kernels) scale linearly with the number of DPUs

Scaling is sublinear for BFS and NW

BFS suffers load imbalance due to irregular graph topology

NW computes a diagonal of a 2D matrix in each iteration.
More DPUs does not mean more parallelization in shorter diagonals.

Strong Scaling: 1 Rank (III)

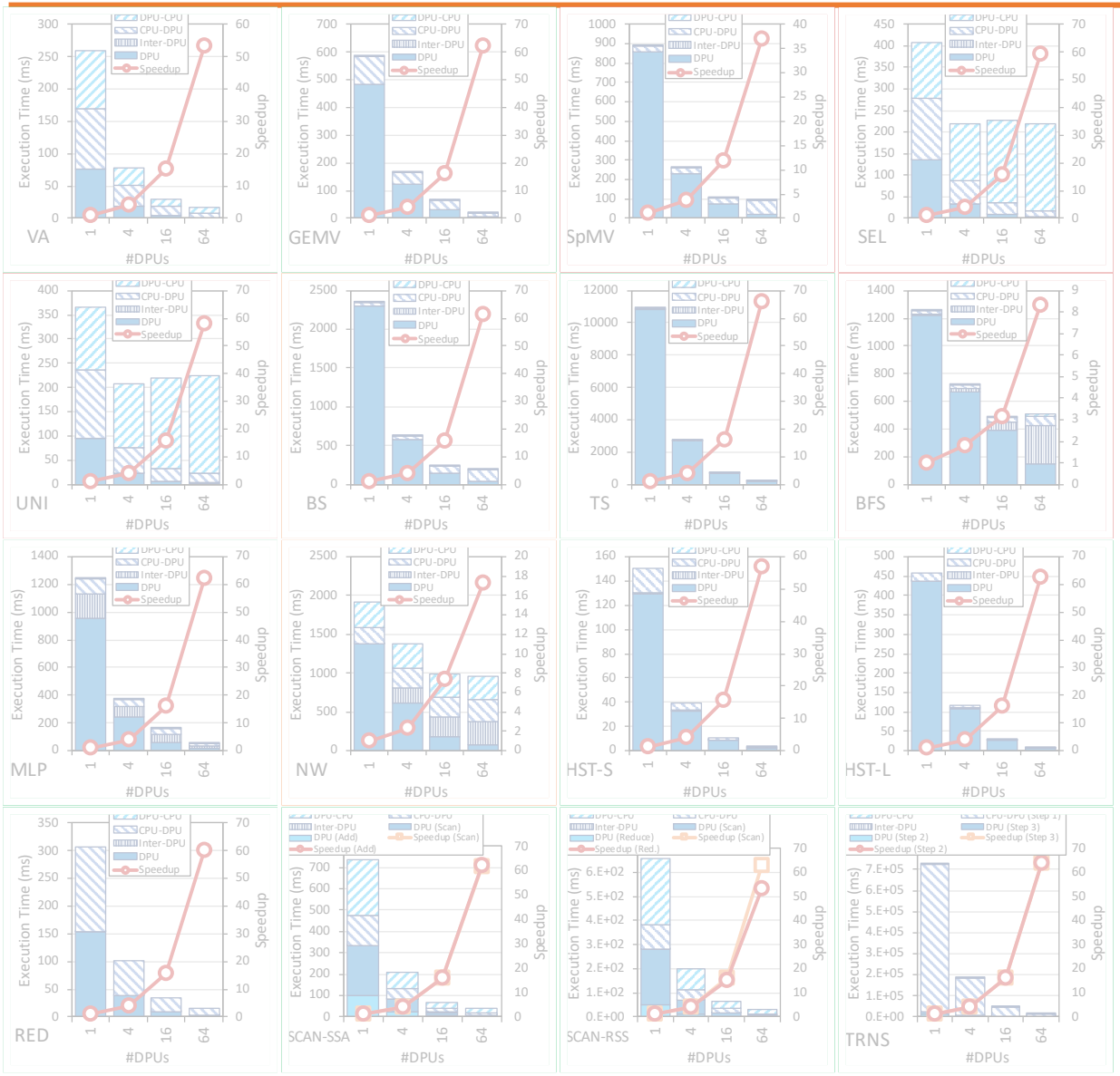


VA, GEMV, SpMV, BS, TS, TRNS do not need inter-DPU synchronization

SEL, UNI, HST-S, HST-L, RED, SCAN-SSA, SCAN-RSS need inter-DPU synchronization but 64 DPUs still obtain the best performance

BFS, MLP, NW require heavy inter-DPU synchronization, involving DPU-CPU and CPU-DPU transfers

Strong Scaling: 1 Rank (IV)



VA, GEMV, TS, MLP, HST-S, HST-L, RED, SCAN-SSA, SCAN-RSS, TRNS use parallel transfers.
CPU-DPU and DPU-CPU transfer times decrease as we increase the number of DPUs

BS, NW use parallel transfers but do not reduce transfer times:

- BS transfers a complete array to all DPUs.
- NW does not use all DPUs in all iterations

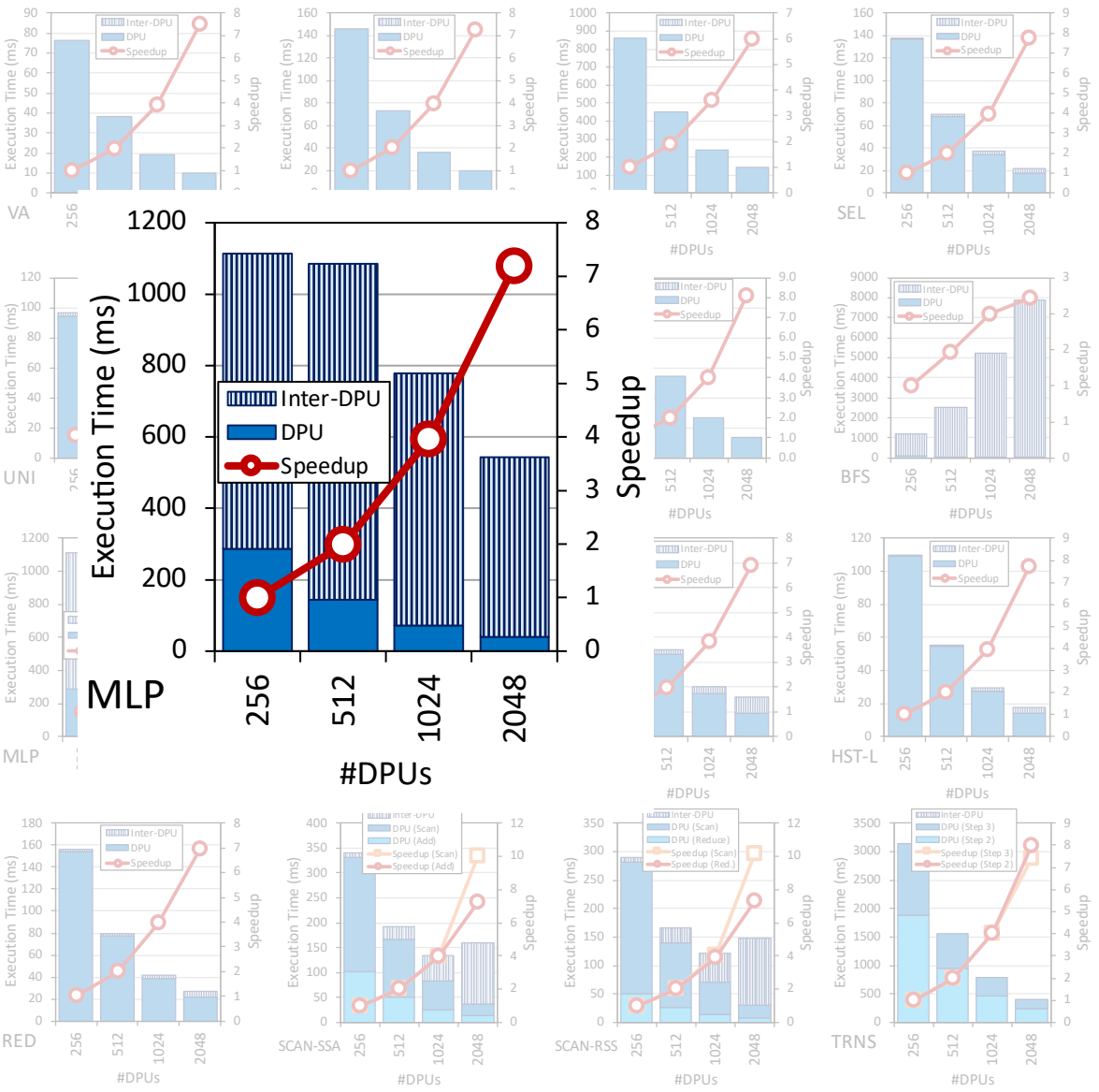
SpMV, SEL, UNI, BFS cannot use parallel transfers, as the transfer size per DPU is not fixed

PROGRAMMING RECOMMENDATION 5

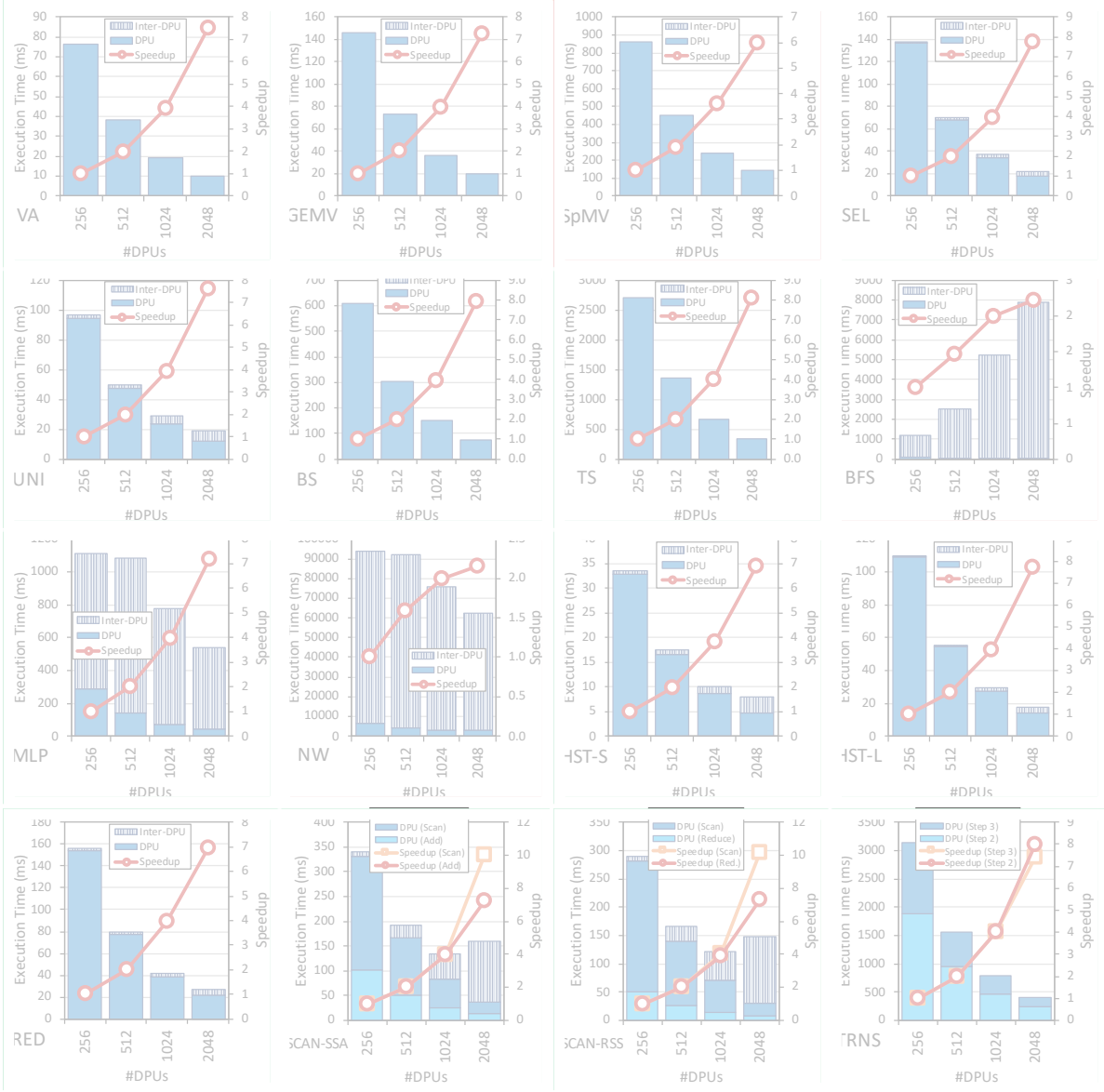
Parallel CPU-DPU/DPU-CPU transfers inside a rank of DPUs are recommended for real-world workloads when all transferred buffers are of the same size.

Strong Scaling: 32 Ranks (I)

- Strong scaling experiments on 32 rank
  - We set the number of tasklets to the best performing one
  - The number of DPUs is 256, 512, 1024, 2048
  - We show the breakdown of execution time:
    - DPU: Execution time on the DPU
    - Inter-DPU: Time for inter-DPU communication via the host CPU
    - We do not show CPU-DPU/DPU-CPU transfer times
  - Speedup over 256 DPUs



Strong Scaling: 32 Ranks (II)



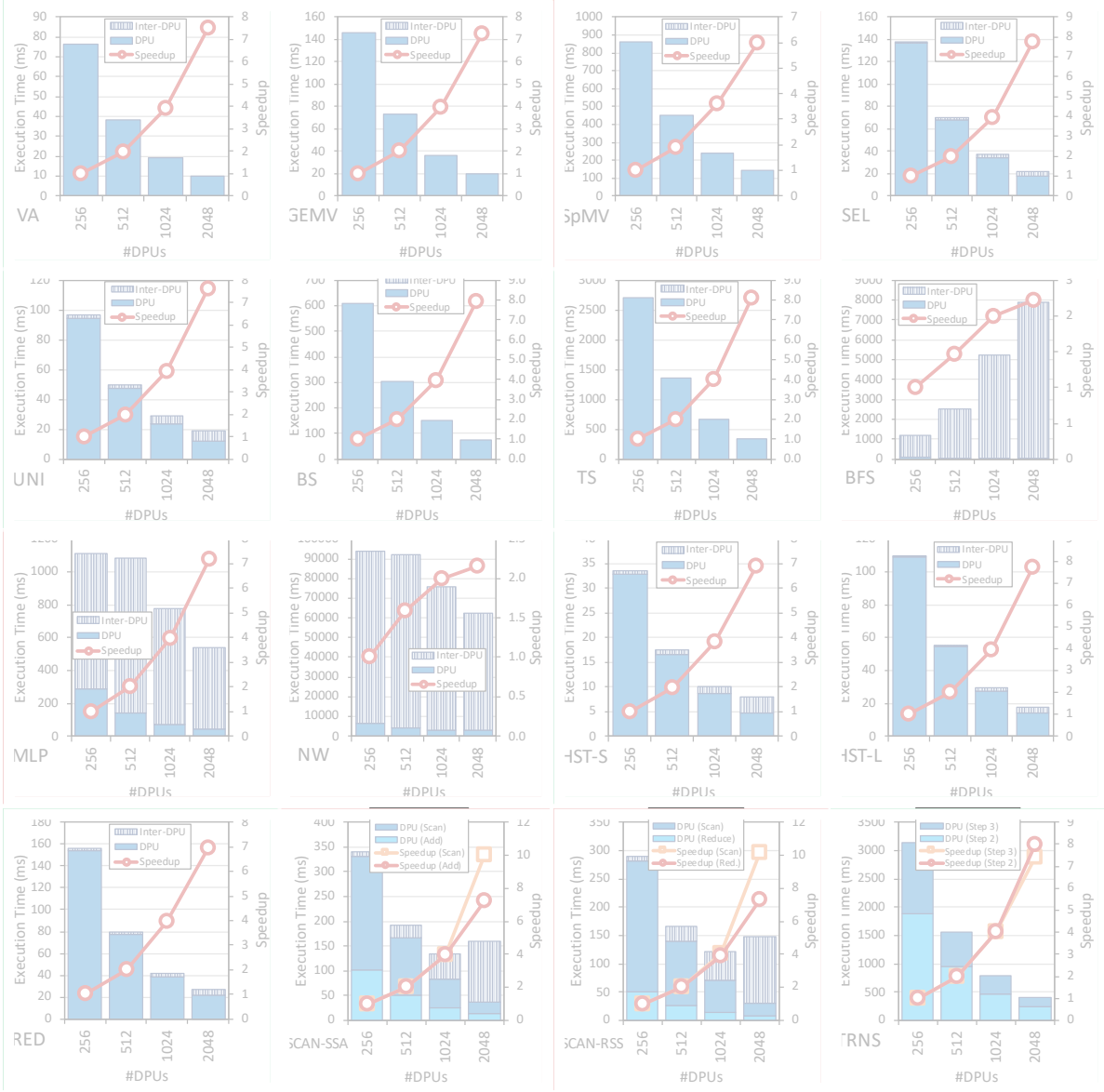
VA, GEMV, SEL, UNI, BS, TS, MLP, HST-S, HSTS-L, RED, SCAN-SSA (both kernel), SCAN-RSS (both kernels), and TRNS (both kernels) scale linearly with the number of DPUs

SpMV, BFS, NW do not scale linearly due to load imbalance

KEY OBSERVATION 14

Load balancing across DPUs ensures linear reduction of the execution time spent on the DPUs for a given problem size, when all available DPUs are used (as observed in strong scaling experiments).

Strong Scaling: 32 Ranks (III)



SEL, UNI, HST-S, HST-L, RED only need to merge final results

KEY OBSERVATION 15

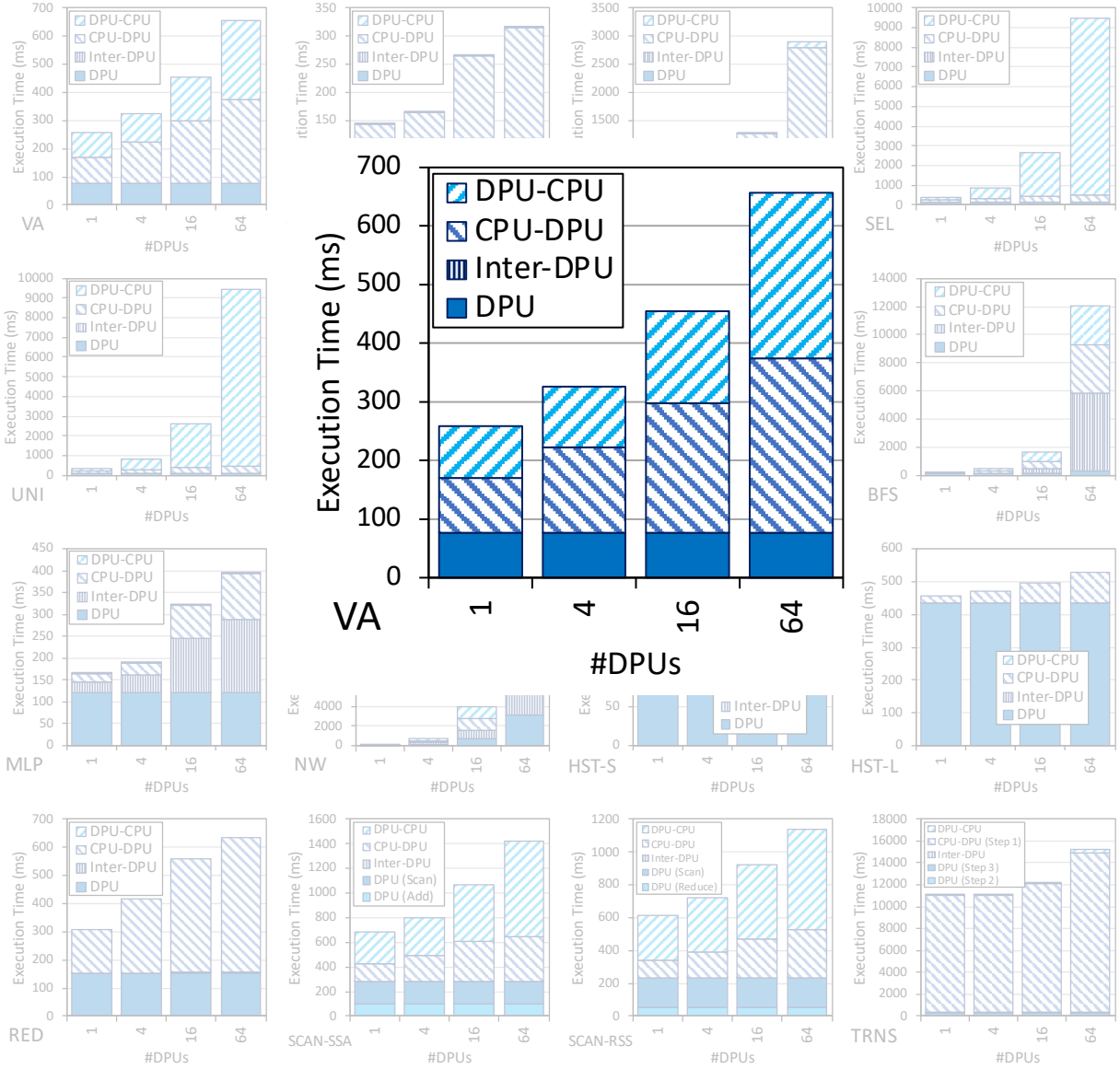
The overhead of merging partial results from DPUs in the host CPU is tolerable across all PrIM benchmarks that need it.

BFS, MLP, NW, SCAN-SSA, SCAN-RSS have more complex communication

KEY OBSERVATION 16

Complex synchronization across DPUs (i.e., inter-DPU synchronization involving two-way communication with the host CPU) imposes significant overhead, which limits scalability to more DPUs.

Weak Scaling: 1 Rank



KEY OBSERVATION 17

Equally-sized problems assigned to different DPUs and little/no inter-DPU synchronization lead to linear weak scaling of the execution time spent on the DPUs (i.e., constant execution time when we increase the number of DPUs and the dataset size accordingly).

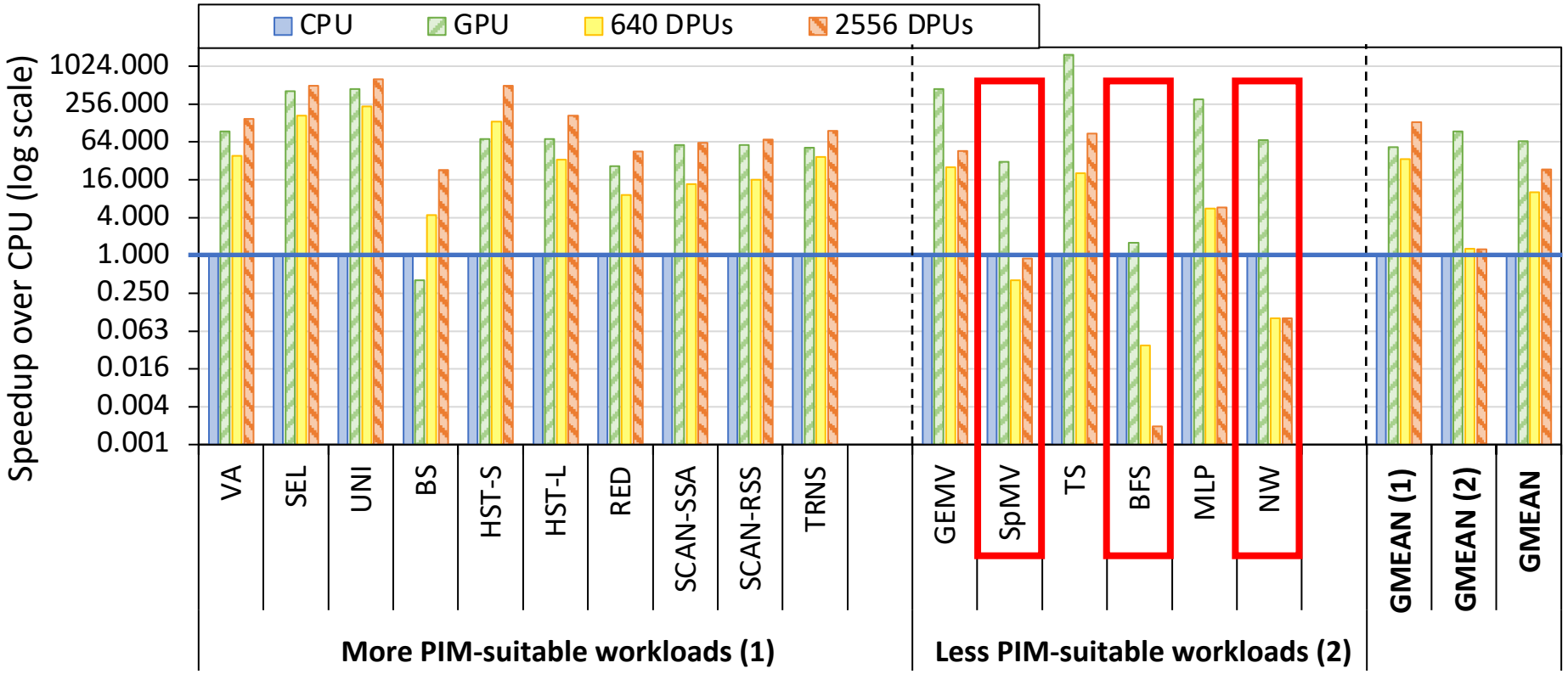
KEY OBSERVATION 18

Sustained bandwidth of parallel CPU-DPU/DPU-CPU transfers inside a rank of DPUs increases sublinearly with the number of DPUs.

CPU/GPU: Evaluation Methodology

- Comparison of both UPMEM-based PIM systems to state-of-the-art CPU and GPU
  - Intel Xeon E3-1240 CPU
  - NVIDIA Titan V GPU
- We use state-of-the-art CPU and GPU counterparts of PrIM benchmarks
  - https://github.com/CMU-SAFARI/prim-benchmarks
- We use the largest dataset that we can fit in the GPU memory
- We show overall execution time, including DPU kernel time and inter DPU communication

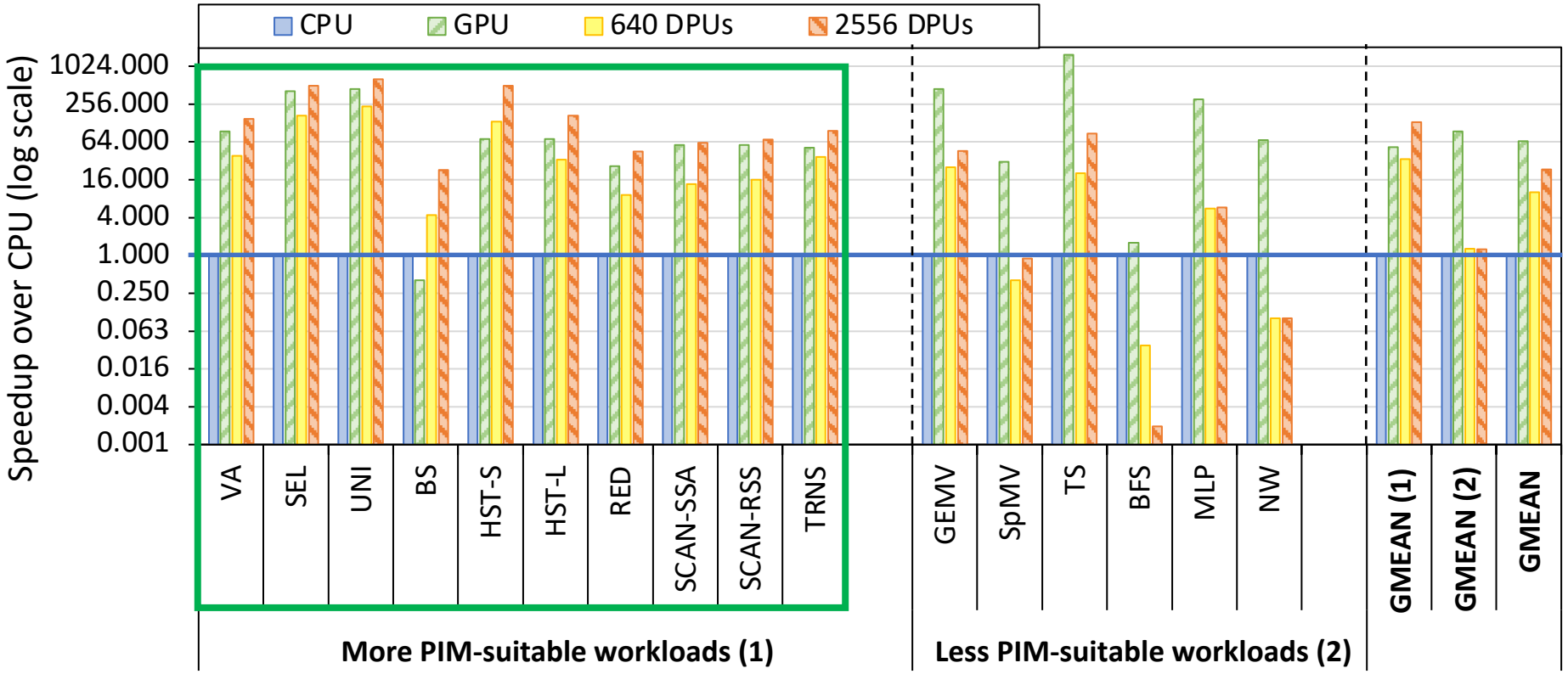
CPU/GPU: Performance Comparison (I)



The 2,556-DPU and the 640-DPU systems outperform the CPU for all benchmarks except SpMV, BFS, and NW

The 2,556-DPU and the 640-DPU are, respectively, 93.0x and 27.9x faster than the CPU for 13 of the PrIM benchmarks

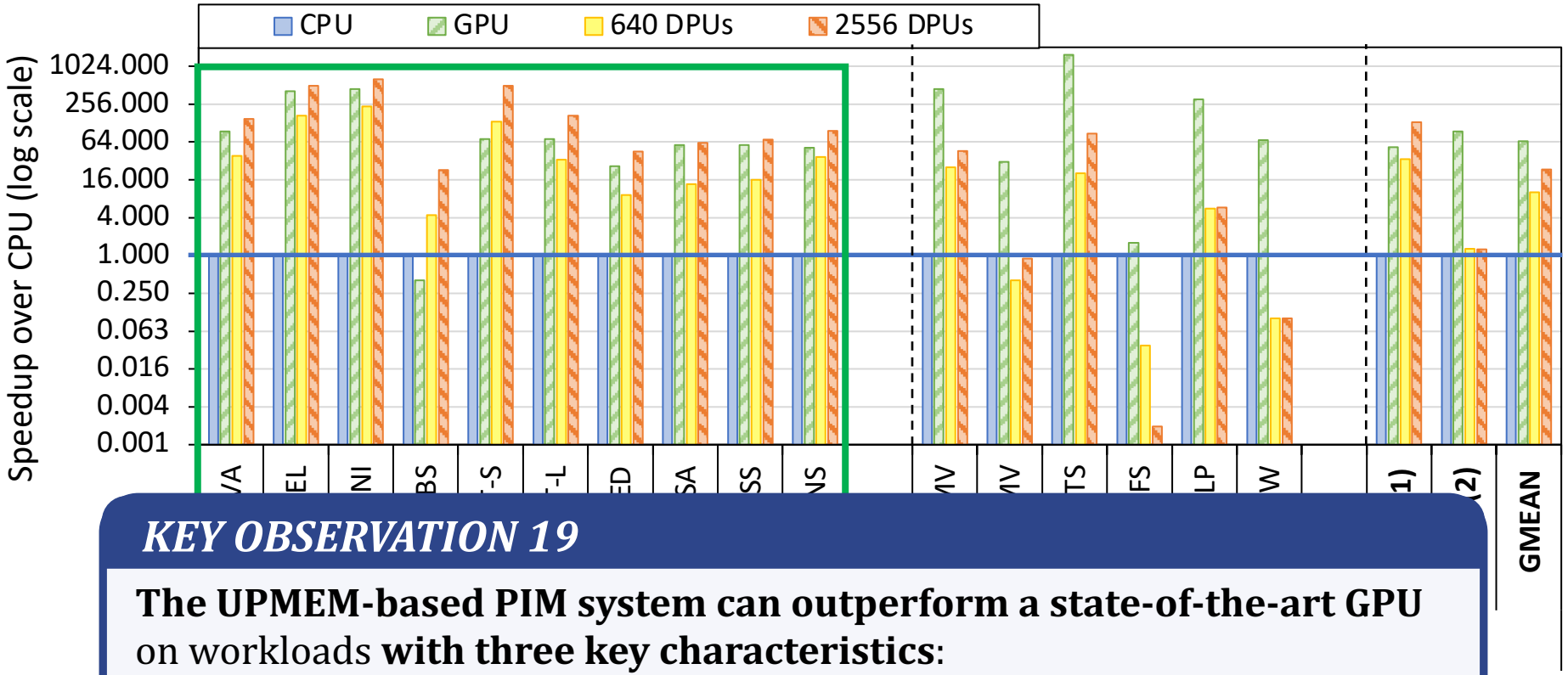
CPU/GPU: Performance Comparison (II)



The 2,556-DPU outperforms the GPU for 10 PrIM benchmarks with an average of 2.54x

The performance of the 640-DPU is within 65% the performance of the GPU for the same 10 PrIM benchmarks

CPU/GPU: Performance Comparison (III)



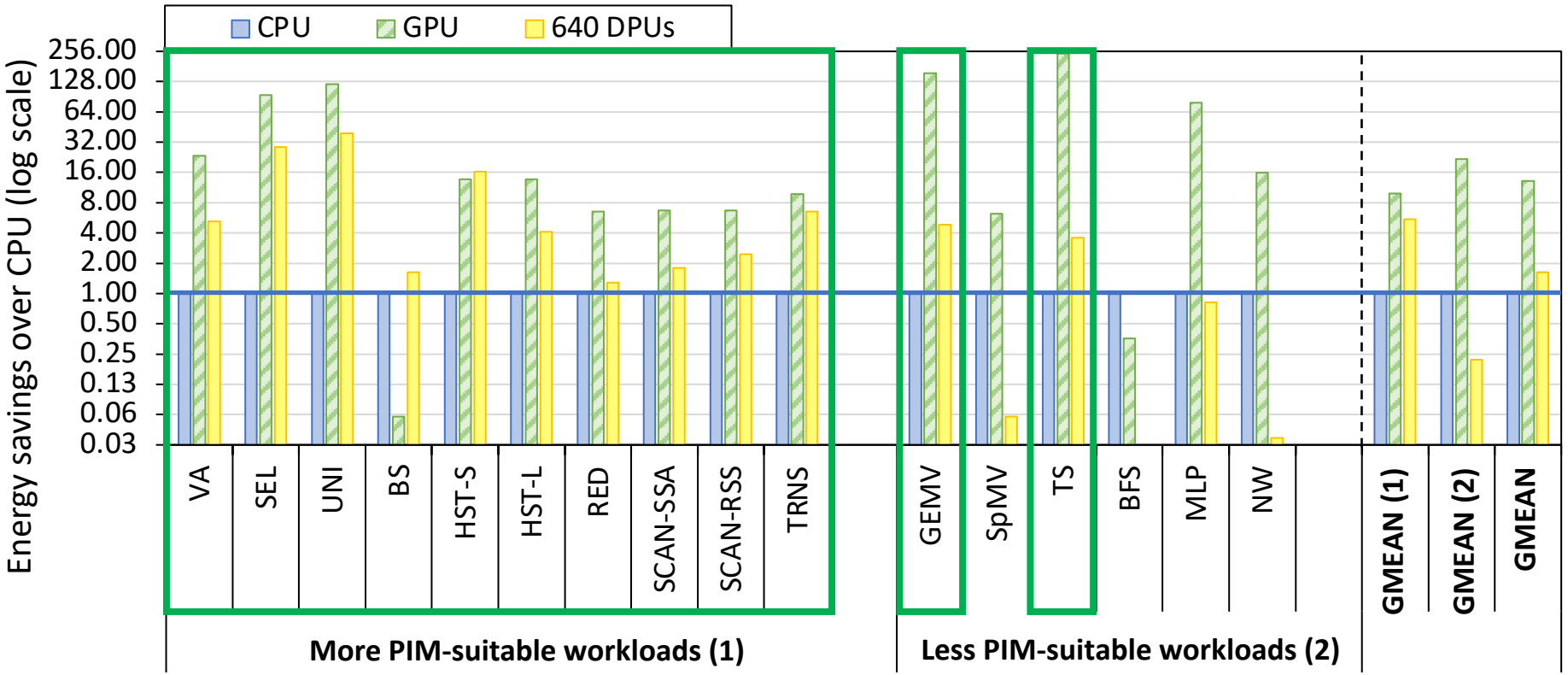
KEY OBSERVATION 19

The UPMEM-based PIM system can outperform a state-of-the-art GPU on workloads **with three key characteristics**:

1. Streaming memory accesses
2. No or little inter-DPU synchronization
3. No or little use of integer multiplication, integer division, or floating point operations

These three key characteristics make a **workload potentially suitable to the UPMEM PIM architecture**.

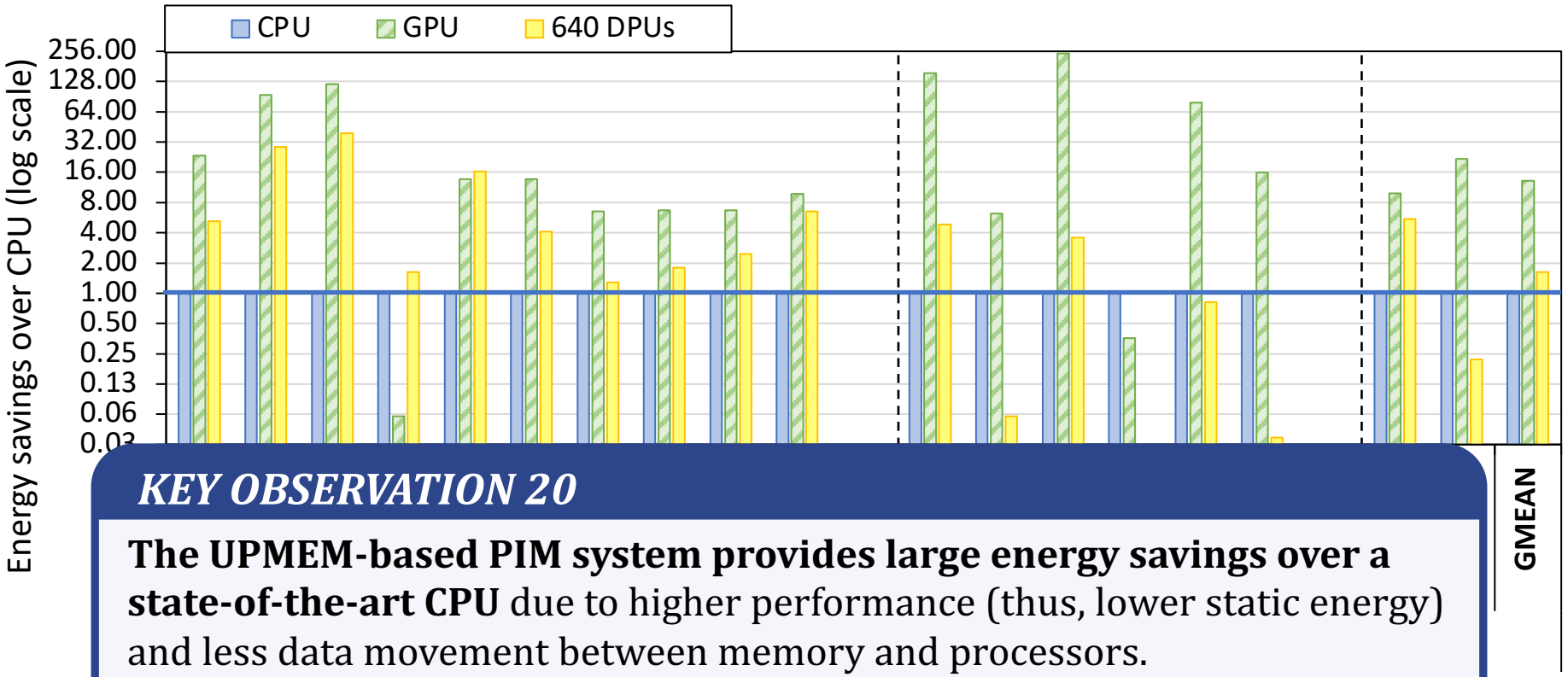
CPU/GPU: Energy Comparison (I)



The 640-DPU system consumes on average 1.64x less energy than the CPU for all 16 PrIM benchmarks

For 12 benchmarks, the 640-DPU system provides energy savings of 5.23x over the CPU

CPU/GPU: Energy Comparison (II)



KEY OBSERVATION 20

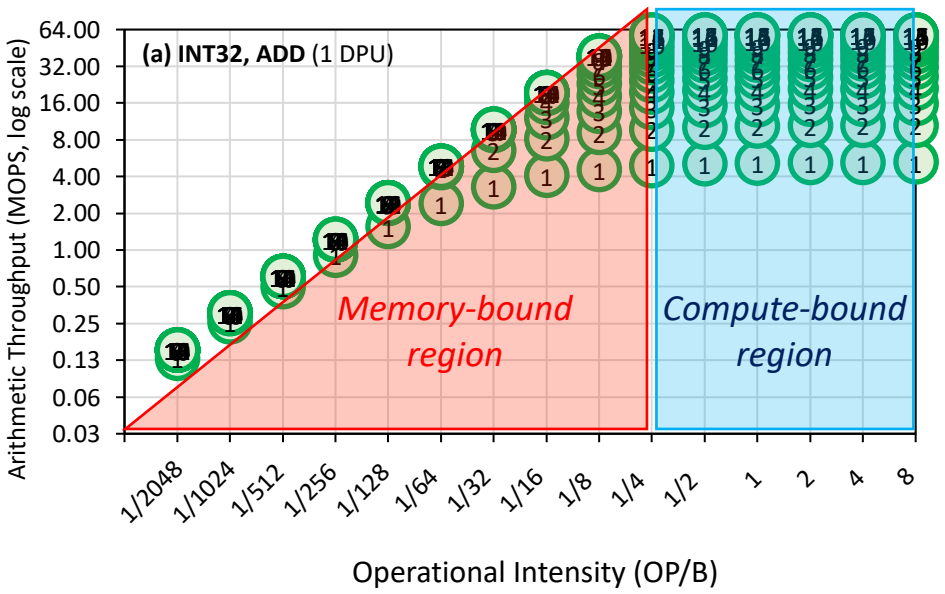
The UPMEM-based PIM system provides large energy savings over a state-of-the-art CPU due to higher performance (thus, lower static energy) and less data movement between memory and processors.

The UPMEM-based PIM system provides energy savings over a state-of-the-art CPU/GPU on workloads where it outperforms the CPU/GPU. This is because the source of both performance improvement and energy savings is the same: **the significant reduction in data movement between the memory and the processor cores**, which the UPMEM-based PIM system can provide for PIM-suitable workloads.

Outline

- Introduction
  - Accelerator Model
  - UPMEM-based PIM System Overview
- UPMEM PIM Programming
  - Vector Addition
  - CPU-DPU Data Transfers
  - Inter-DPU Communication
  - CPU-DPU/DPU-CPU Transfer Bandwidth
- DRAM Processing Unit
  - Arithmetic Throughput
  - WRAM and MRAM Bandwidth
- PrIM Benchmarks
  - Roofline Model
  - Benchmark Diversity
- Evaluation
  - Strong and Weak Scaling
  - Comparison to CPU and GPU
- Key Takeaways

Key Takeaway 1

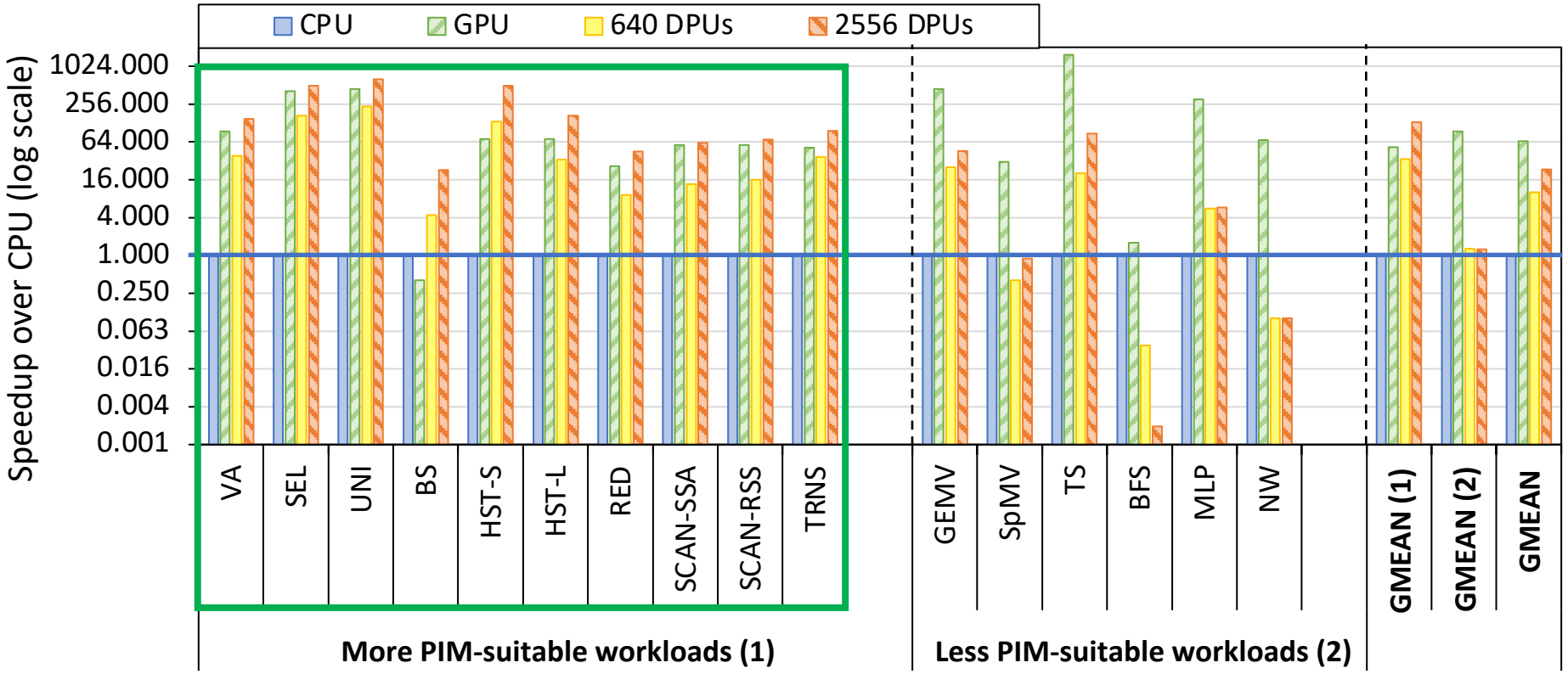


The throughput saturation point is as low as ¼ OP/B, i.e., 1 integer addition per every 32-bit element fetched

KEY TAKEAWAY 1

The UPMEM PIM architecture is fundamentally compute bound.
As a result, **the most suitable workloads are memory-bound.**

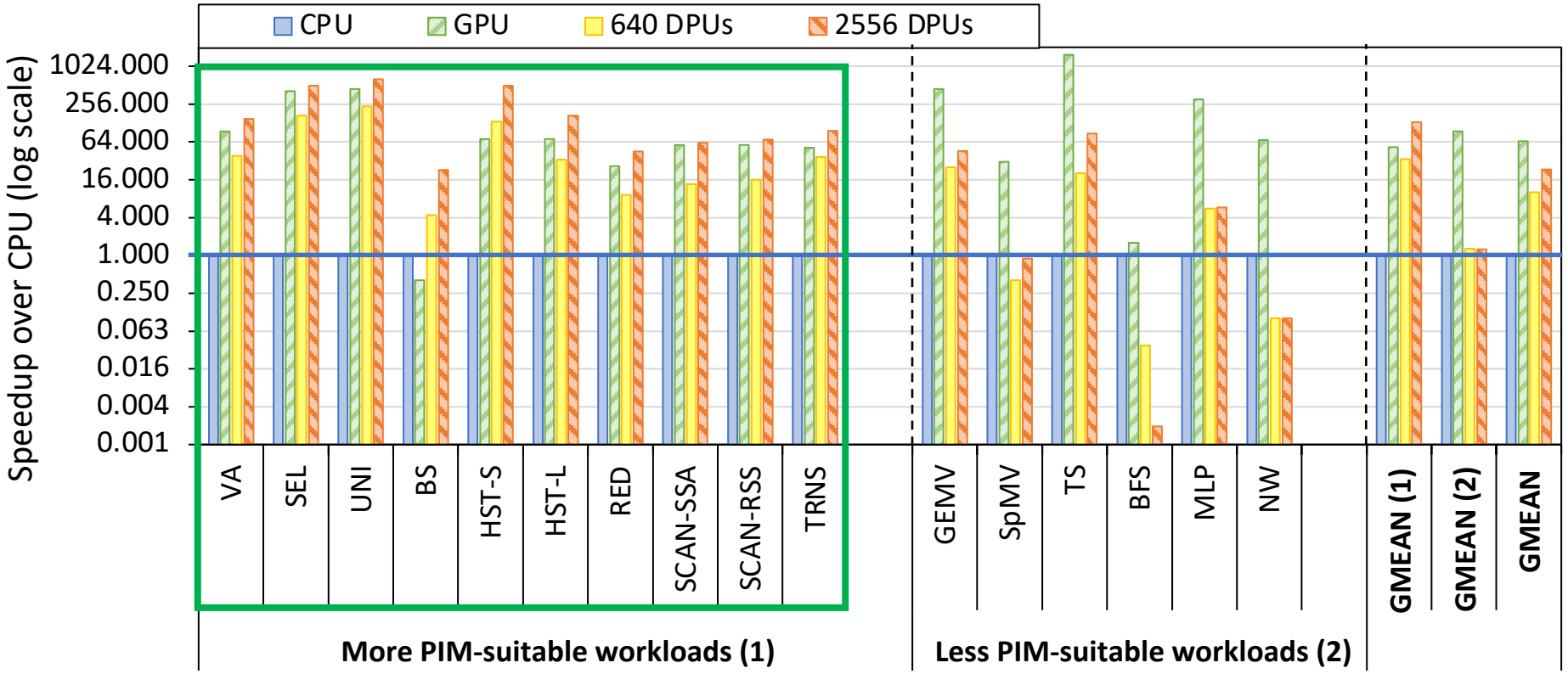
Key Takeaway 2



KEY TAKEAWAY 2

The most well-suited workloads for the UPMEM PIM architecture use no arithmetic operations or use only simple operations (e.g., bitwise operations and integer addition/subtraction).

Key Takeaway 3



KEY TAKEAWAY 3

The most well-suited workloads for the UPMEM PIM architecture require little or no communication across DPUs (inter-DPU communication).

Key Takeaway 4

KEY TAKEAWAY 4

- UPMEM-based PIM systems **outperform state-of-the-art CPUs in terms of performance** (by 23.2× on 2,556 DPUs for 16 PrIM benchmarks) **and energy efficiency on most of PrIM benchmarks.**

- UPMEM-based PIM systems **outperform state-of-the-art GPUs on a majority of PrIM benchmarks** (by 2.54× on 2,556 DPUs for 10 PrIM benchmarks), and the outlook is even more positive for future PIM systems.

- UPMEM-based PIM systems are **more energy-efficient than state-of-the-art CPUs and GPUs on workloads that they provide performance improvements** over the CPUs and the GPUs.

Understanding a Modern PIM Architecture

Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System

JUAN GÓMEZ-LUNA[1], IZZAT EL HAJJ[2], IVAN FERNANDEZ[1,3], CHRISTINA GIANNOULA[1,4], GERALDO F. OLIVEIRA[1], AND ONUR MUTLU[1]

[1]ETH Zürich
[2]American University of Beirut
[3]University of Malaga
[4]National Technical University of Athens

Corresponding author: Juan Gómez-Luna (e-mail: juang@ethz.ch).

https://arxiv.org/pdf/2105.03814.pdf

https://github.com/CMU-SAFARI/prim-benchmarks

Short arXiv Version

Benchmarking Memory-Centric Computing Systems: Analysis of Real Processing-in-Memory Hardware

Juan Gómez-Luna, *ETH Zürich*
Izzat El Hajj, *American University of Beirut*
Ivan Fernandez, *University of Malaga*
Christina Giannoula, *National Technical University of Athens*
Geraldo F. Oliveira, *ETH Zürich*
Onur Mutlu, *ETH Zürich*

https://arxiv.org/pdf/2110.01709.pdf

https://github.com/CMU-SAFARI/prim-benchmarks

Long arXiv Version

Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture

Juan Gómez-Luna[1] Izzat El Hajj[2] Ivan Fernandez[1,3] Christina Giannoula[1,4]
Geraldo F. Oliveira[1] Onur Mutlu[1]

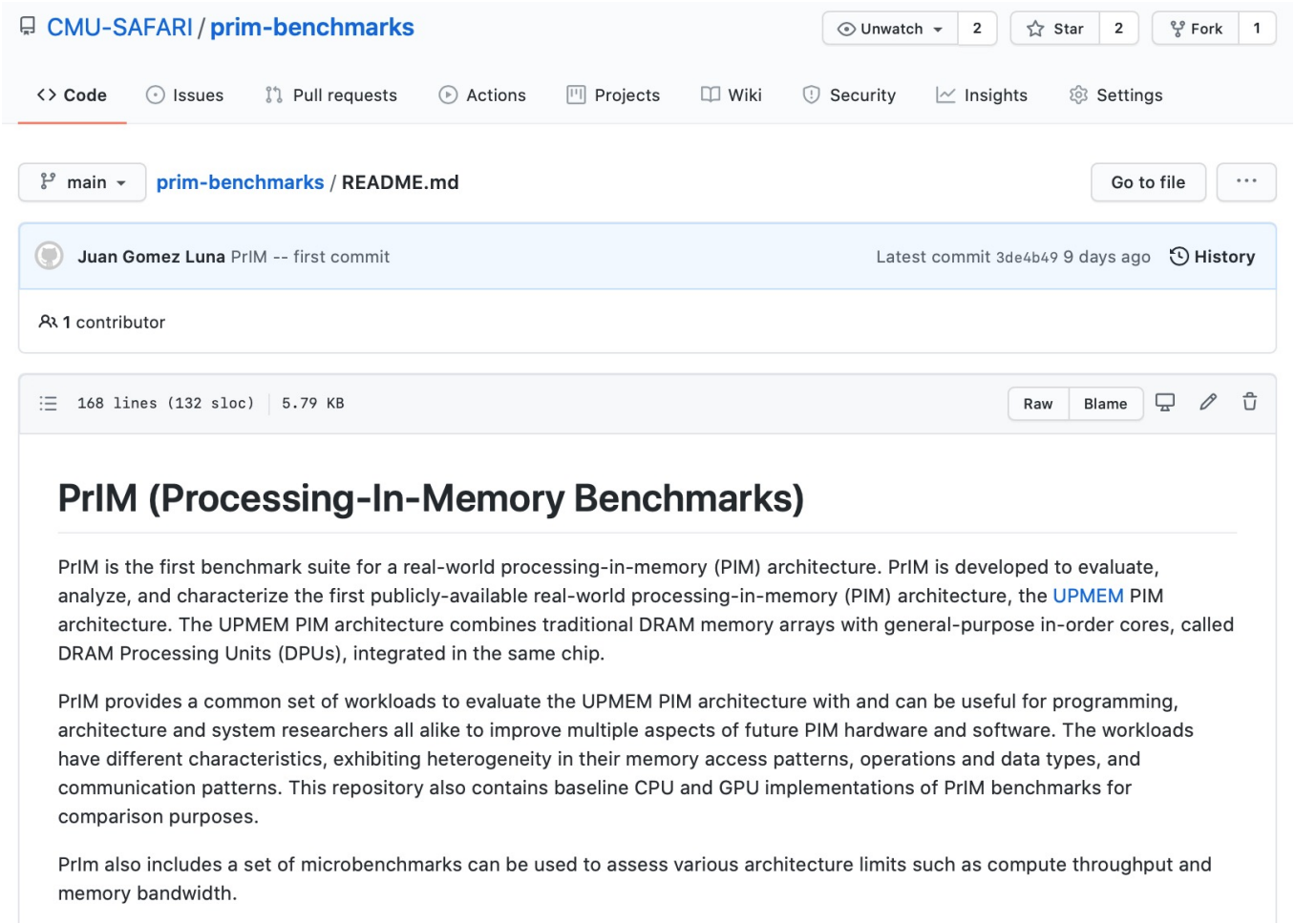
[1]ETH Zürich [2]American University of Beirut [3]University of Malaga [4]National Technical University of Athens

https://arxiv.org/pdf/2105.03814.pdf

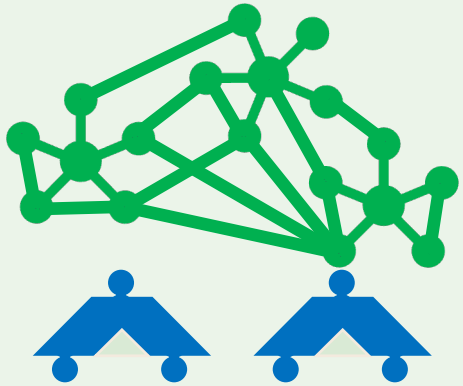
https://github.com/CMU-SAFARI/prim-benchmarks

PrIM Repository

- All microbenchmarks, benchmarks, and scripts
- https://github.com/CMU-SAFARI/prim-benchmarks



Sparse Matrix Vector Multiplication



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

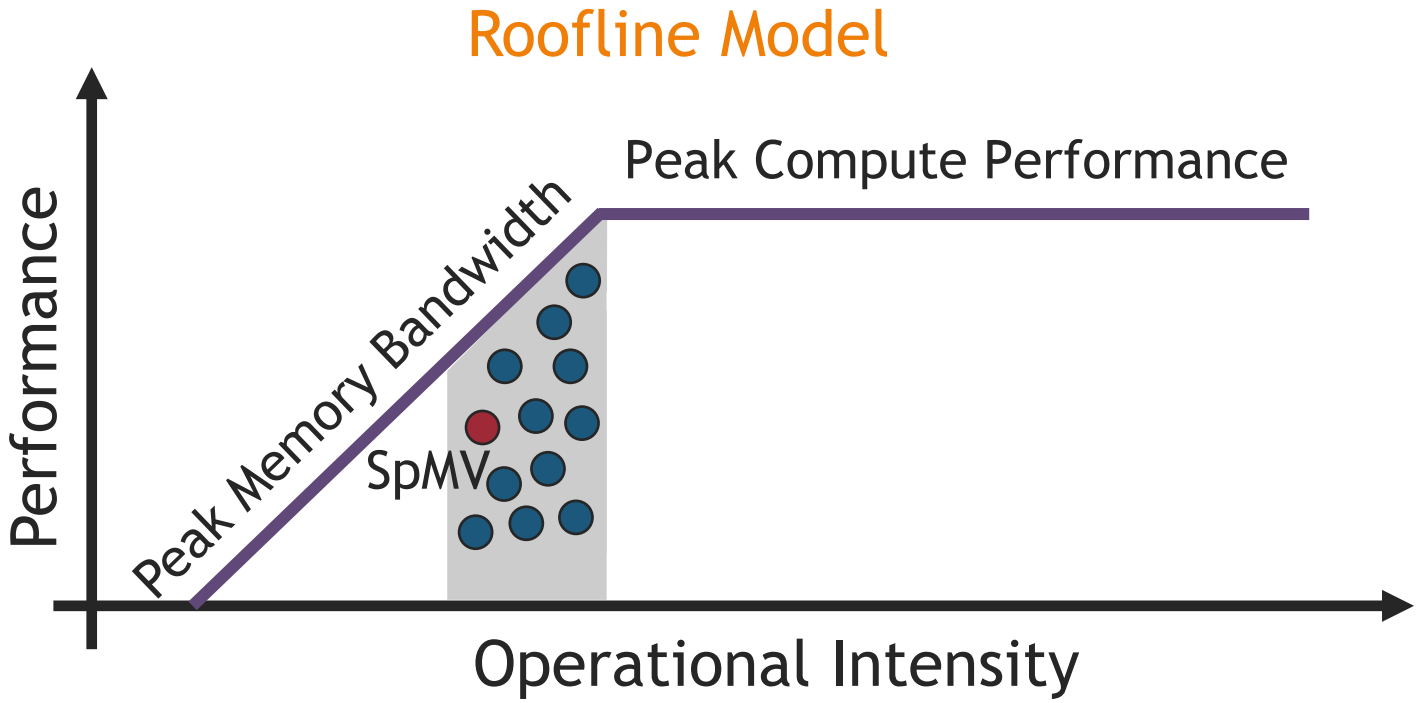
Recommendations for Architects and Programmers

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

Sparse Matrix Vector Multiplication

Sparse Matrix Vector Multiplication (SpMV):

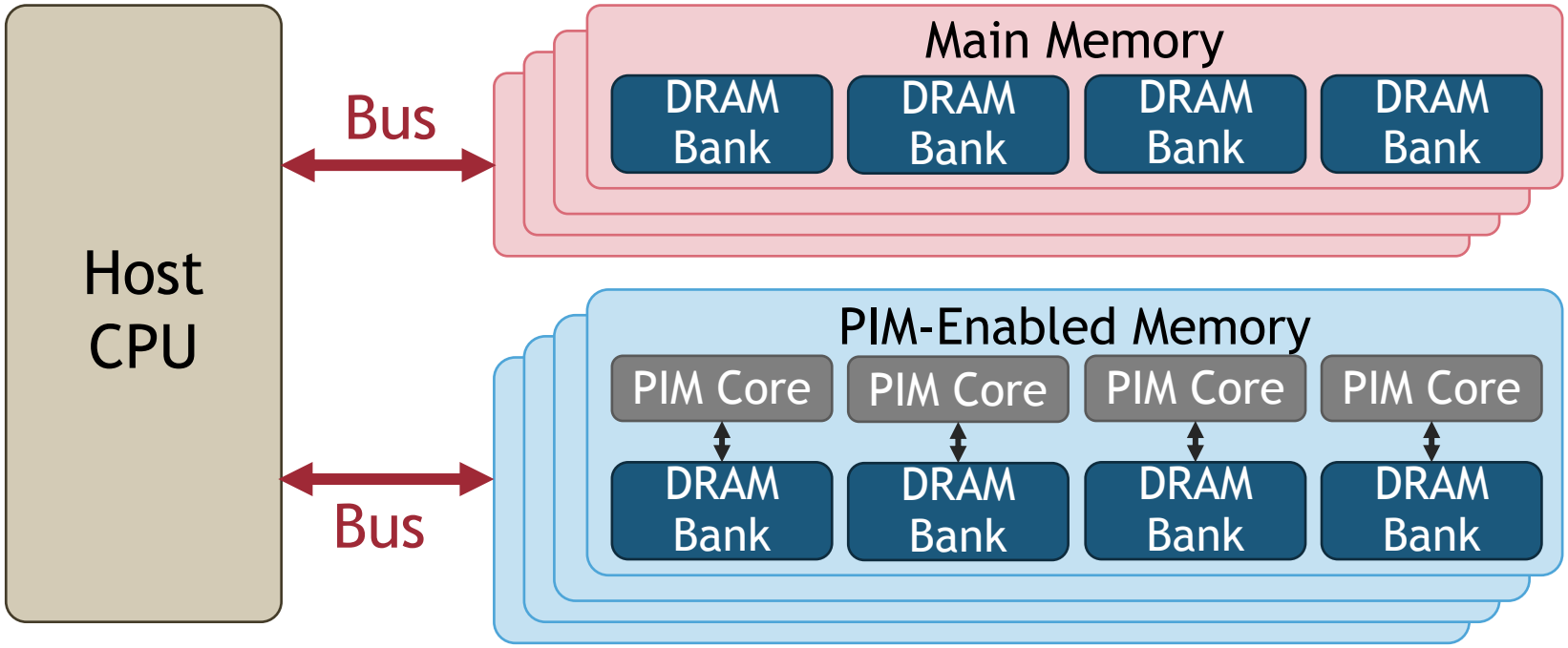
- Widely-used kernel in graph processing, machine learning, scientific computing …
- A highly memory-bound kernel



Real Processing-In-Memory Systems

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

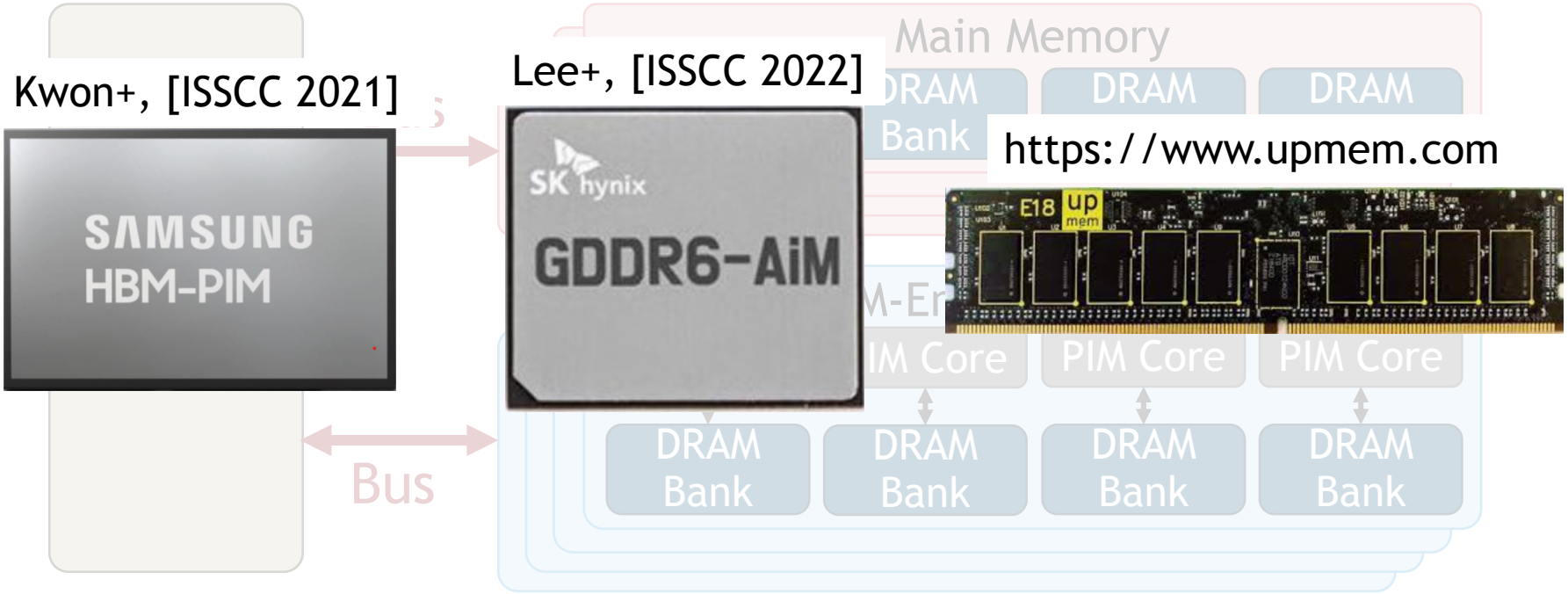
- High levels of parallelism
- Low memory access latency
- Large aggregate memory bandwidth



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

Outline

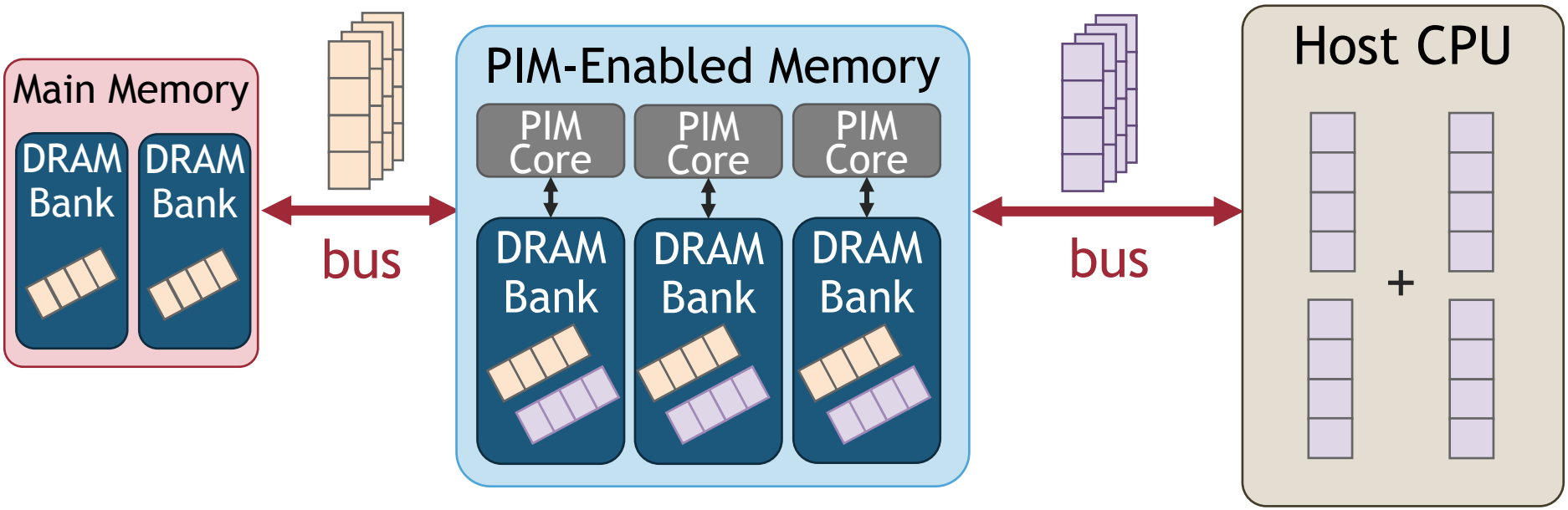
SpMV Kernels for Real PIM Systems

Key Takeaways from Our Study

Conclusion

SpMV Execution on a PIM System

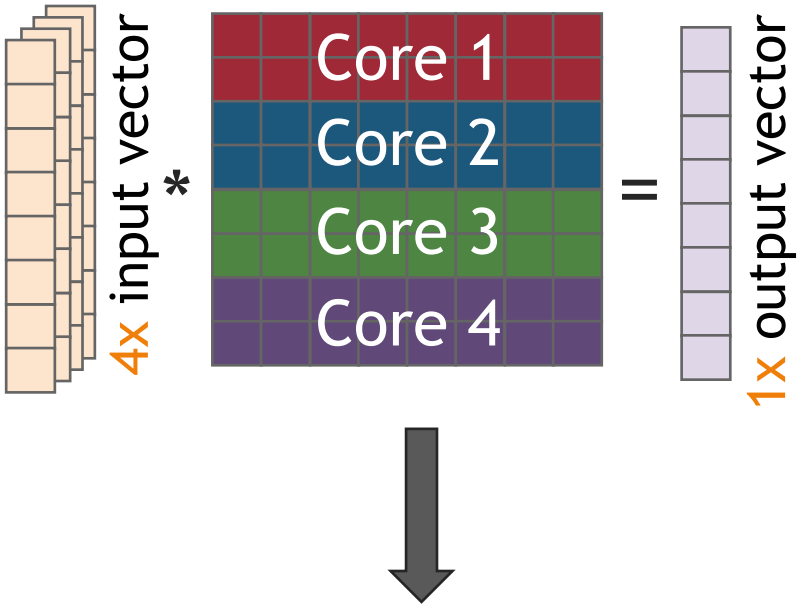
1. Load the input vector
2. Execute the kernel
3. Retrieve the partial results
4. Merge the partial results



Data Partitioning Techniques

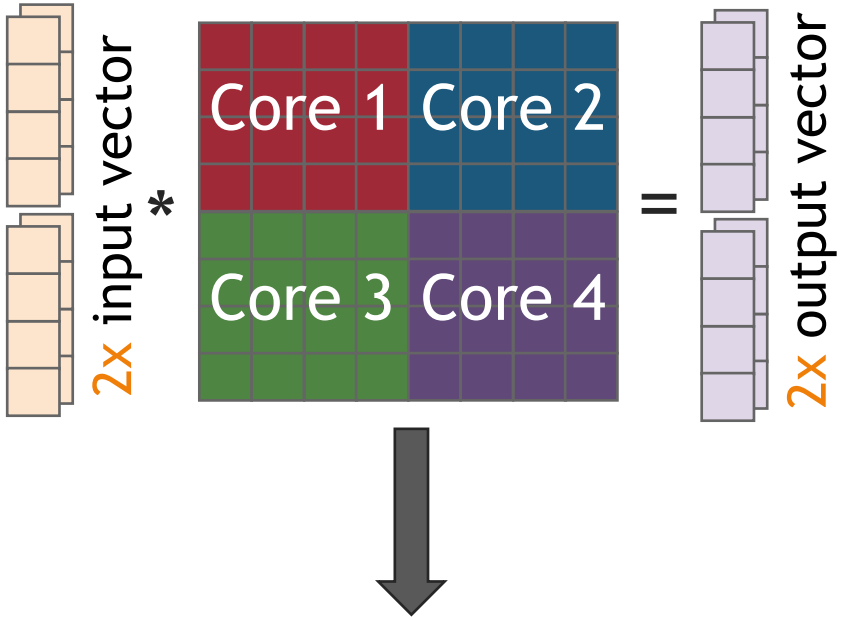
SparseP supports two types of data partitioning techniques:

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 *

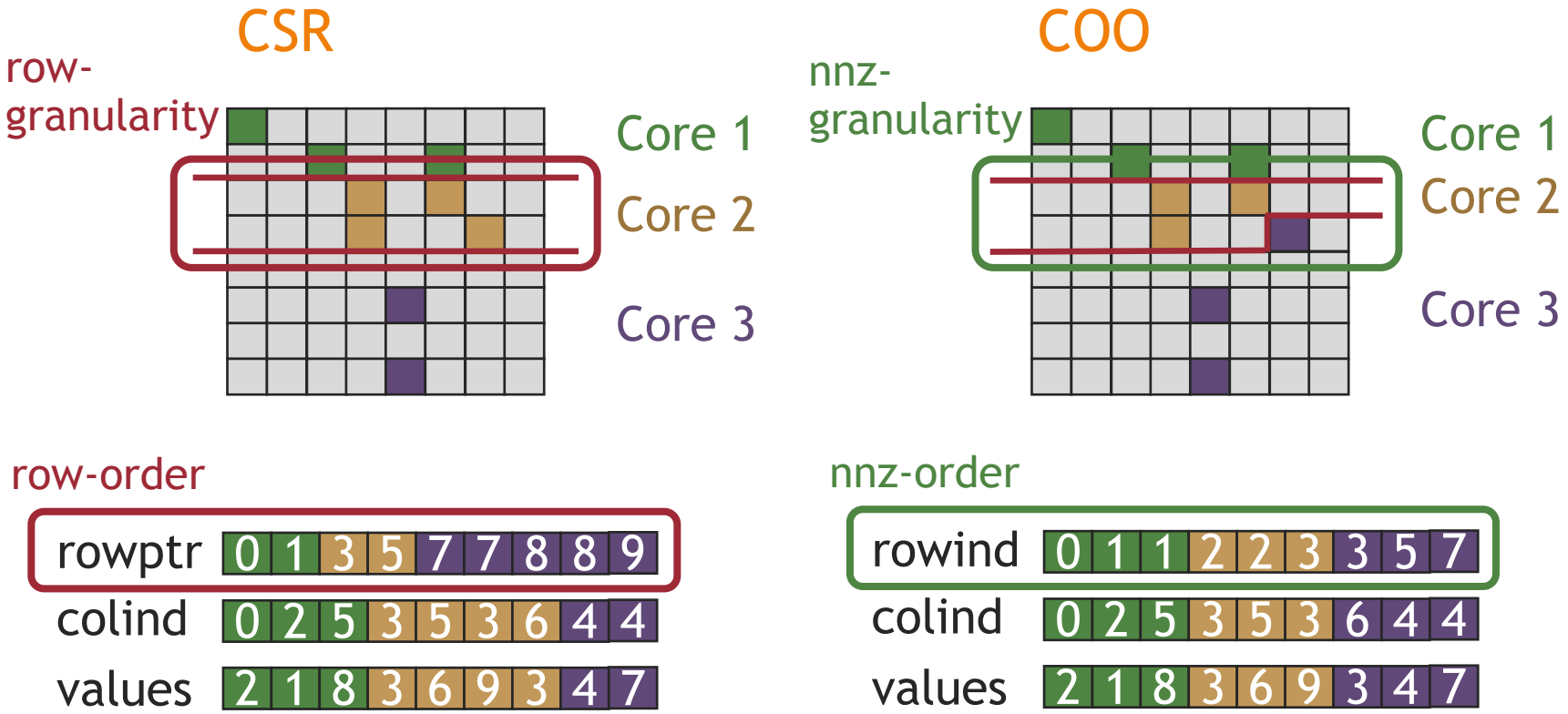
- BCSR, BCOO:
  - Balance Blocks ^
  - Balance NNZs ^

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

1D Partitioning Technique

Load-Balancing of #NNZs:

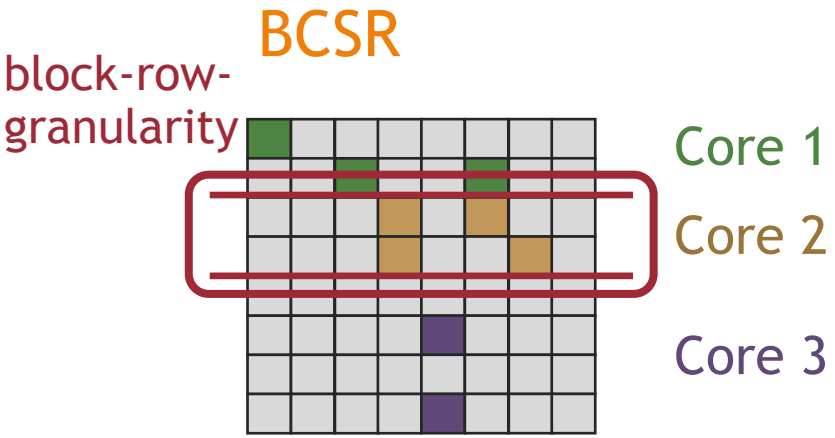
- CSR (row-granularity), COO

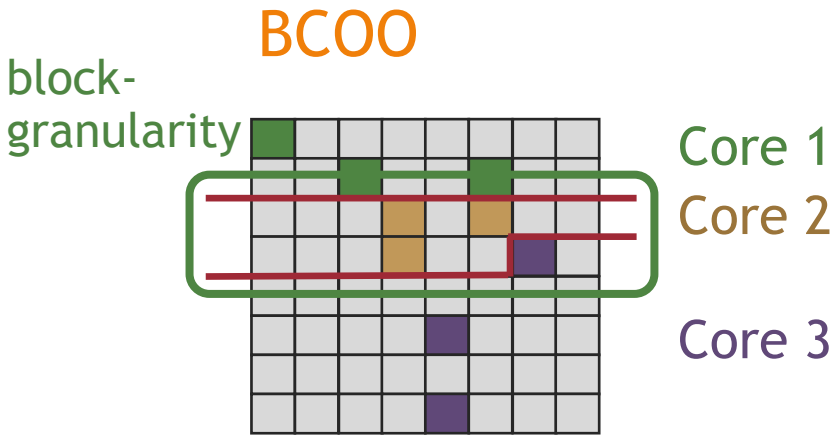


1D Partitioning Technique

Load-Balancing of #NNZs:

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





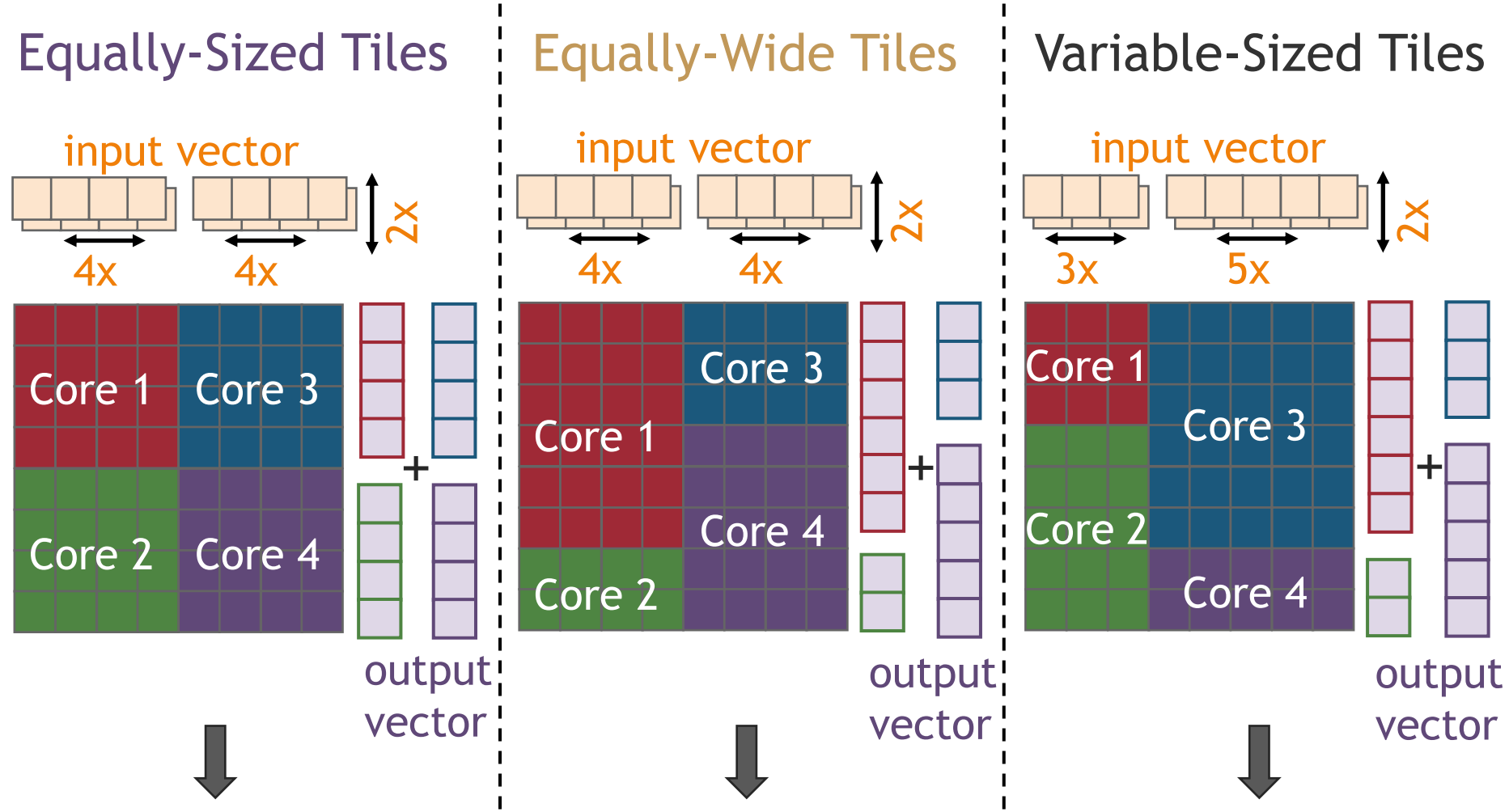
block-row-order

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

block-order

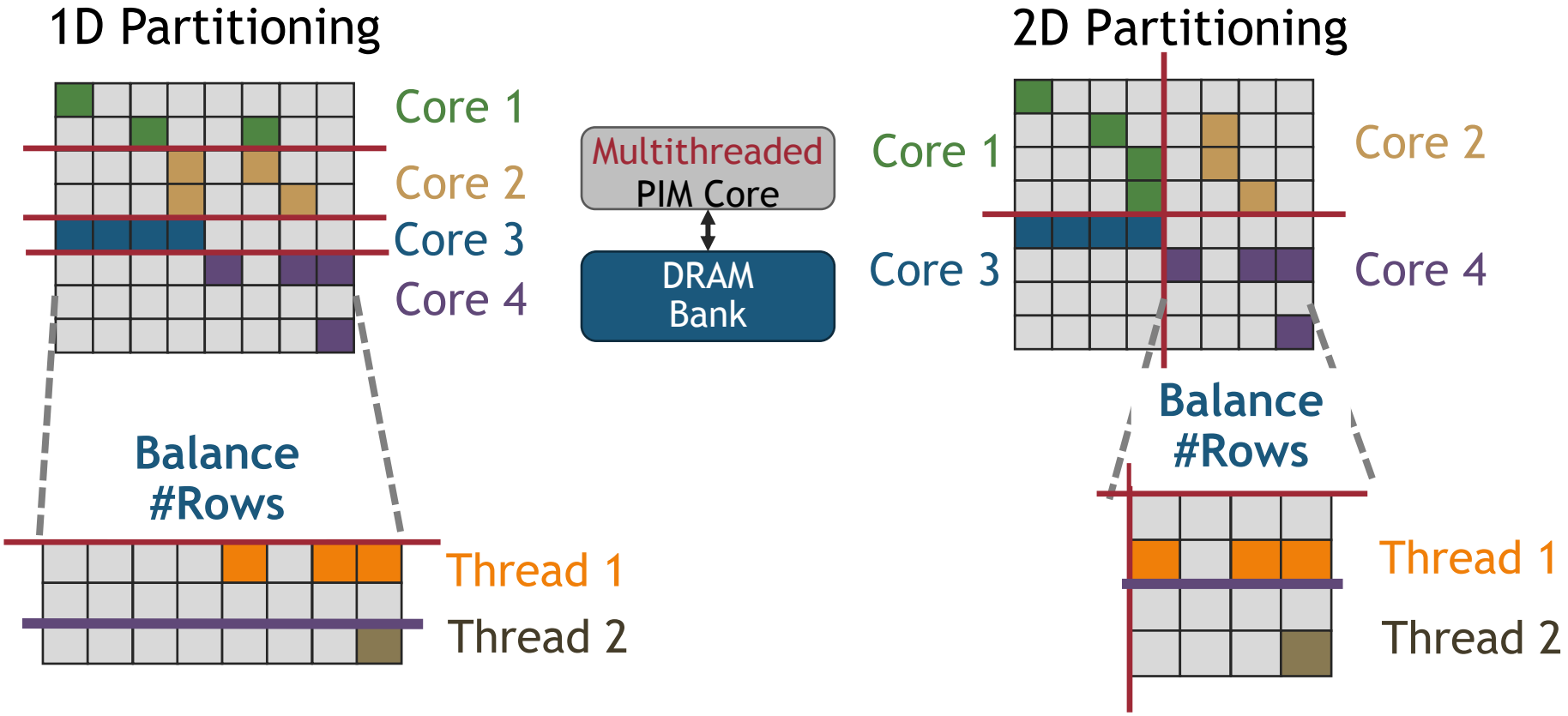
rowind	0	1	1	2	2	3	3	5	7
colind	0	2	5	3	5	3	6	4	4
values	2	1	8	3	6	9	3	4	7

2D Partitioning Technique



Load-Balance across Threads

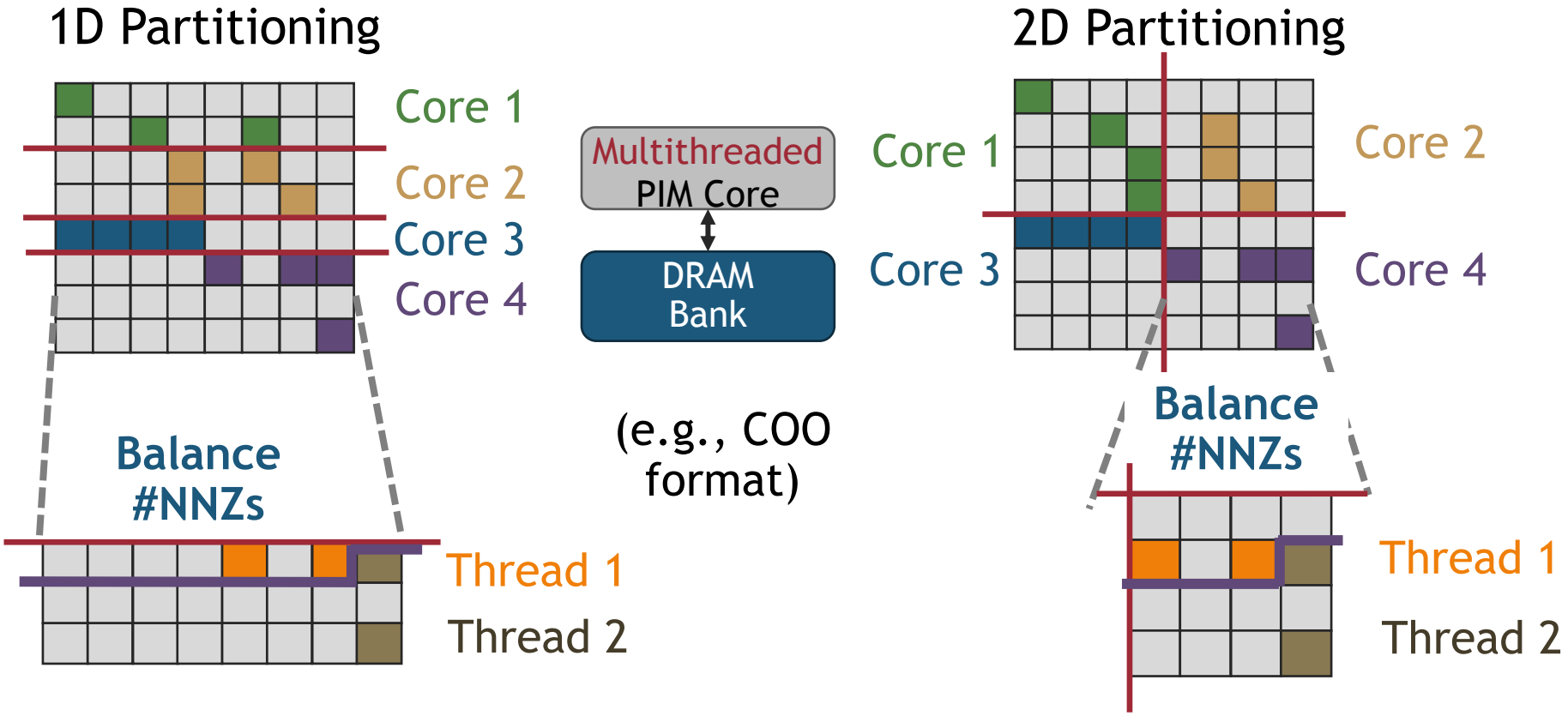
Multithreaded PIM Cores:



- Various load-balance schemes across threads

Load-Balance across Threads

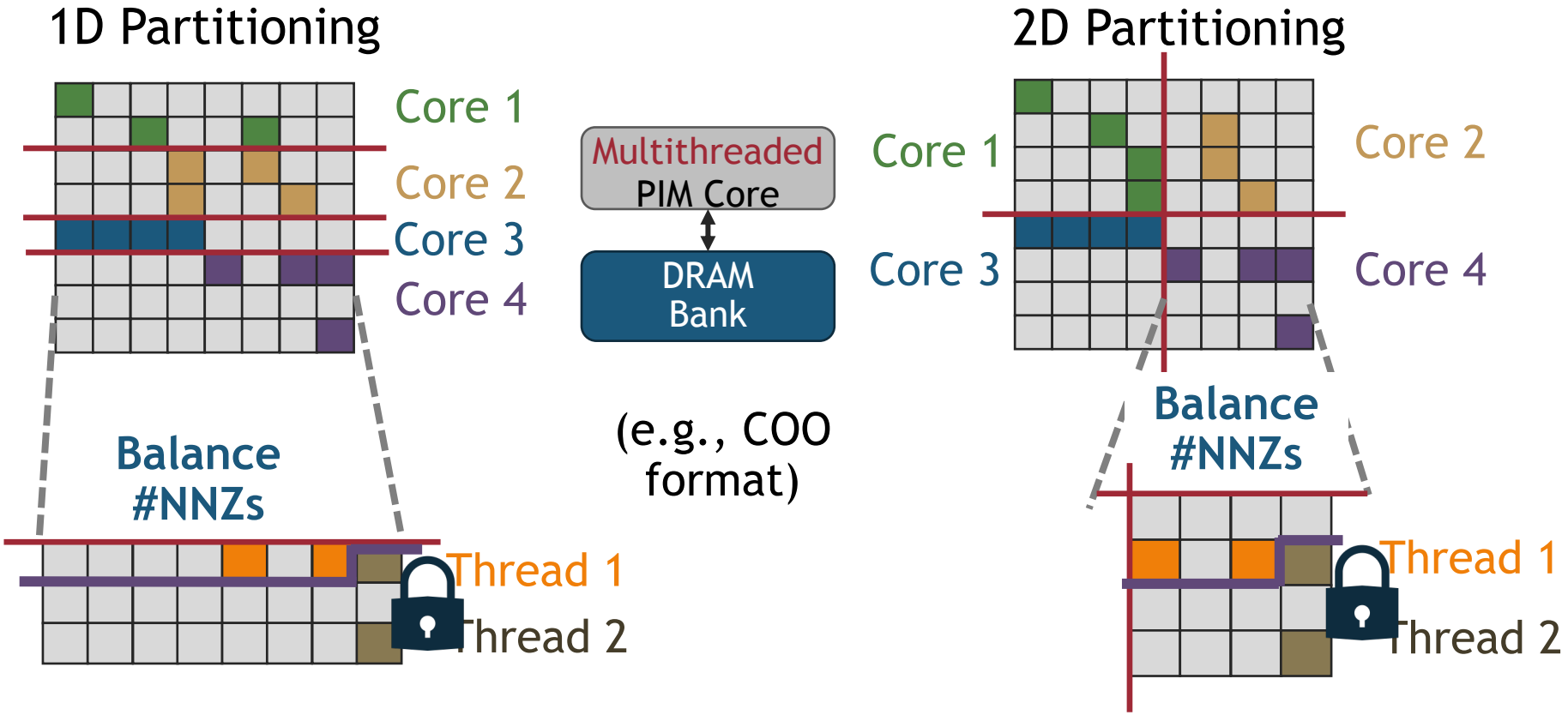
Multithreaded PIM Cores:



- Various load-balance schemes across threads

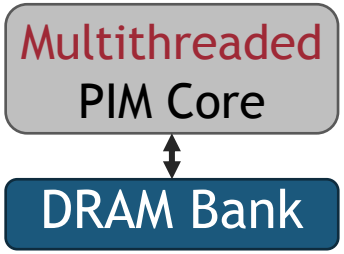
Load-Balance across Threads

Multithreaded PIM Cores:



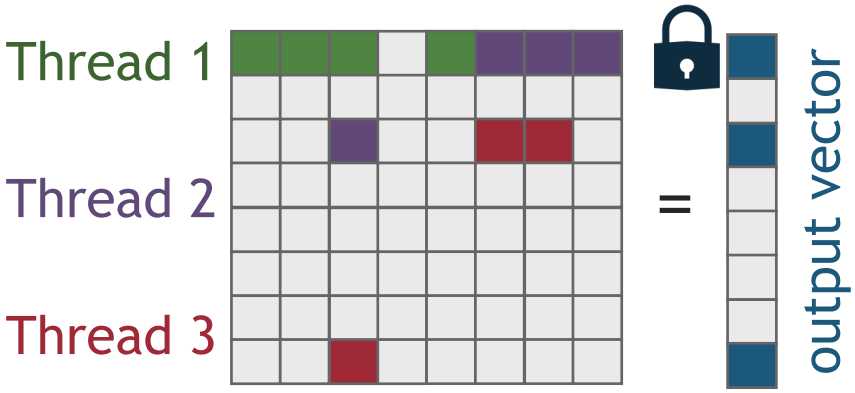
- Various load-balance schemes across threads
- Various synchronization approaches among threads

Synchronization Approaches

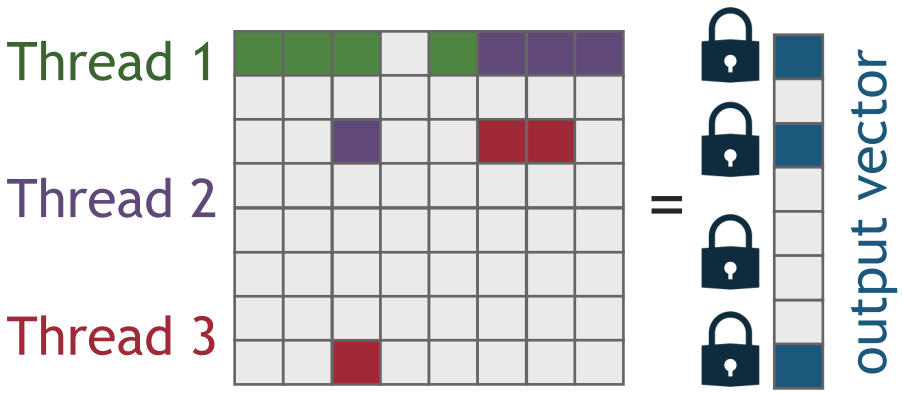


Multithreaded PIM Core:

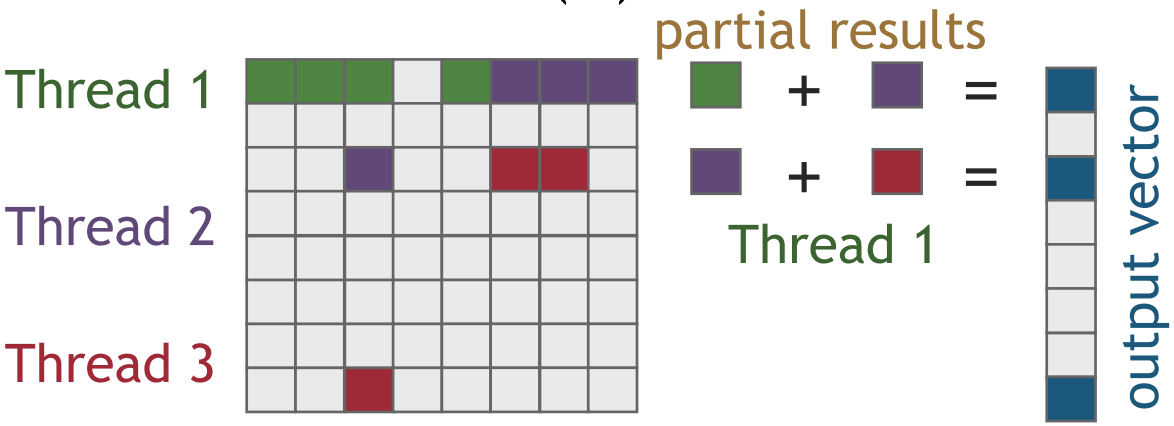
Coarse-Grained (lb-cg)



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

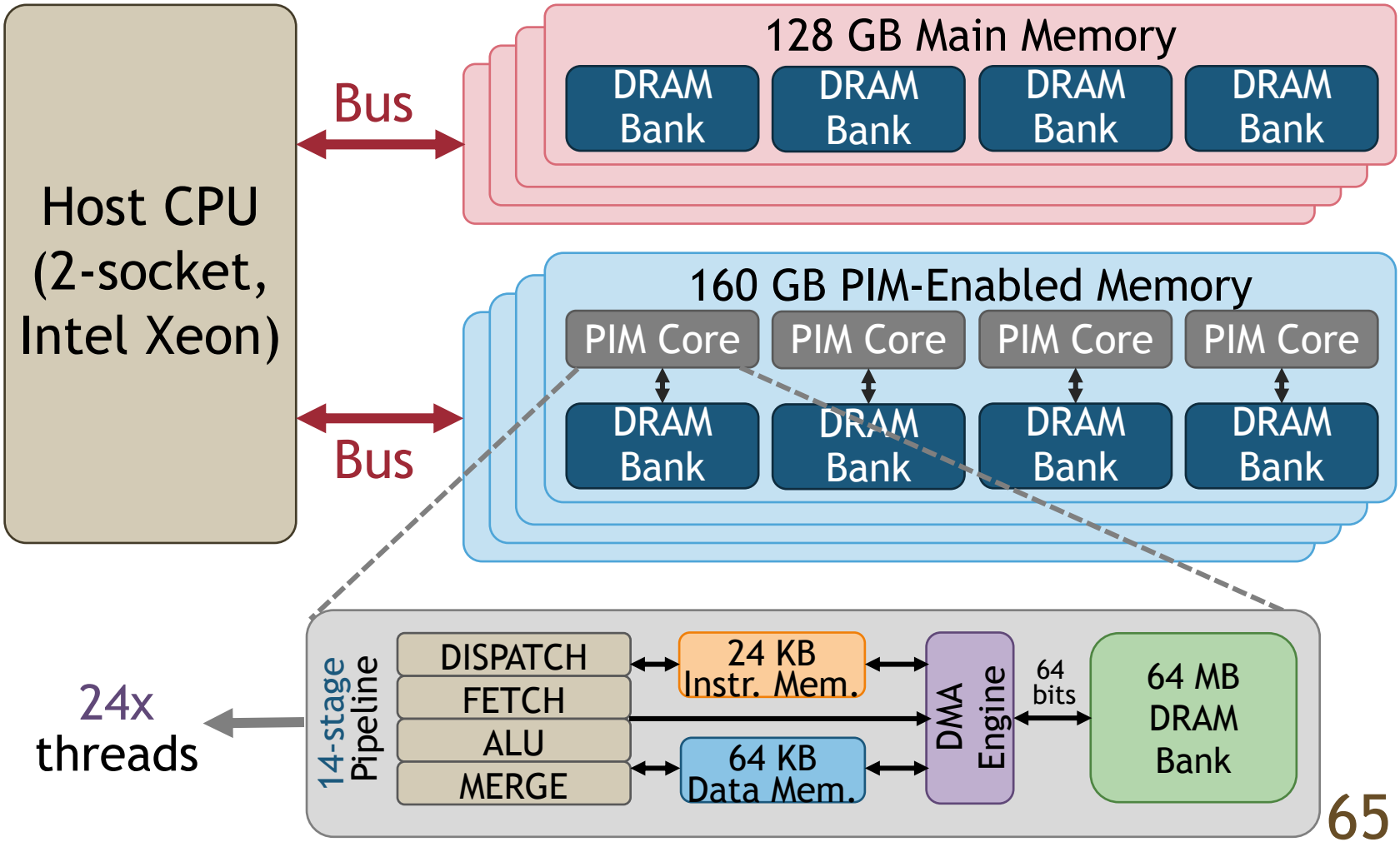
SpMV Kernels for Real PIM Systems

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

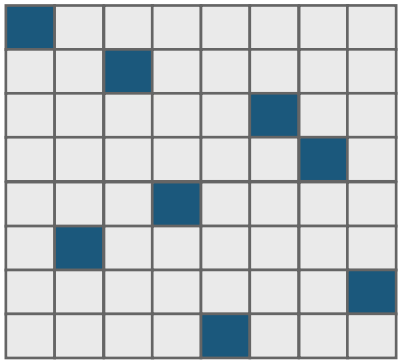


Sparse Matrix Data Set

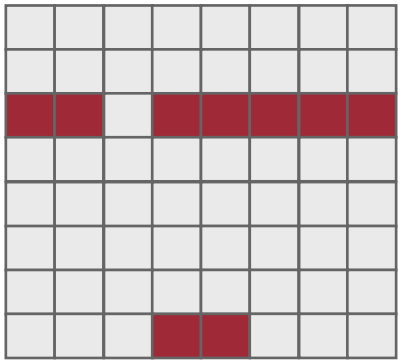
26 sparse matrices*:

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

Regular Matrix



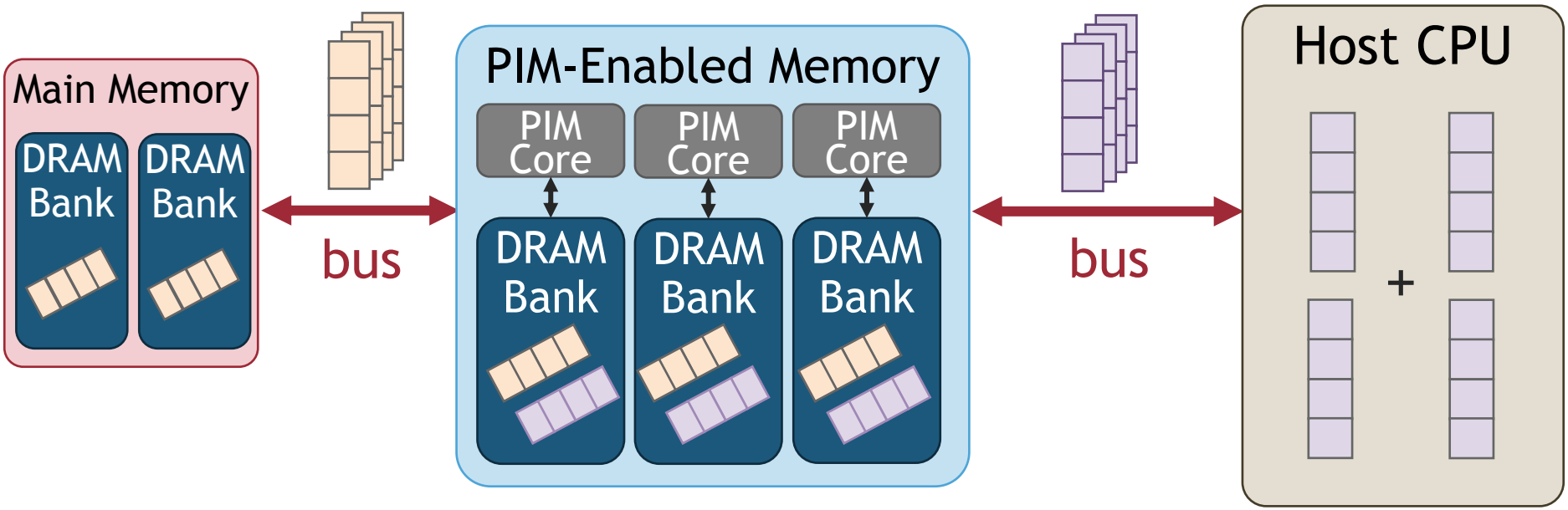
Scale-Free Matrix



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

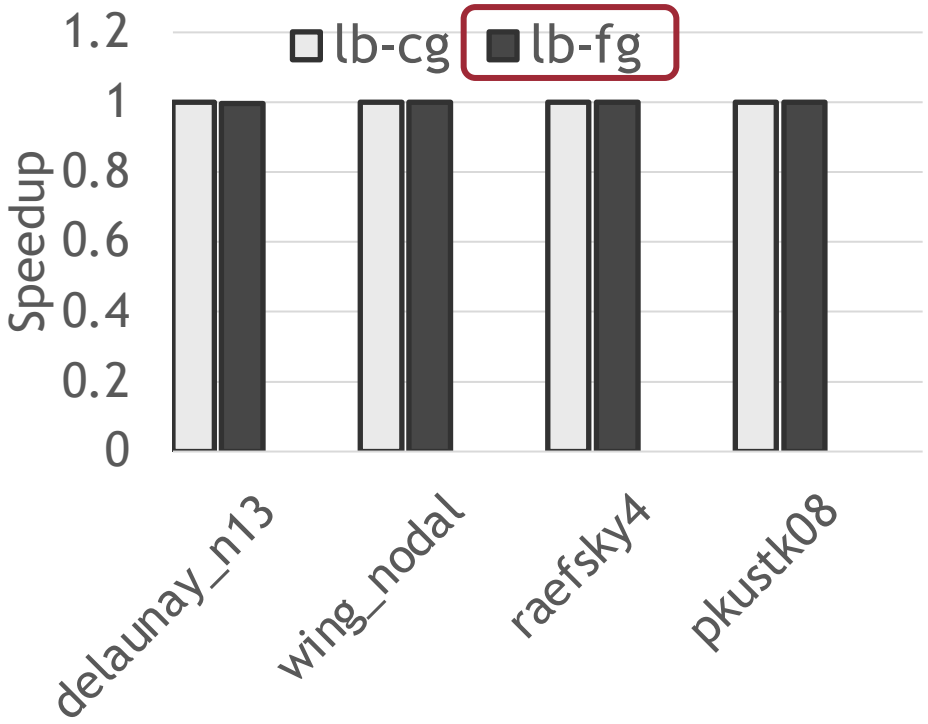
Kernel Execution on One PIM Core

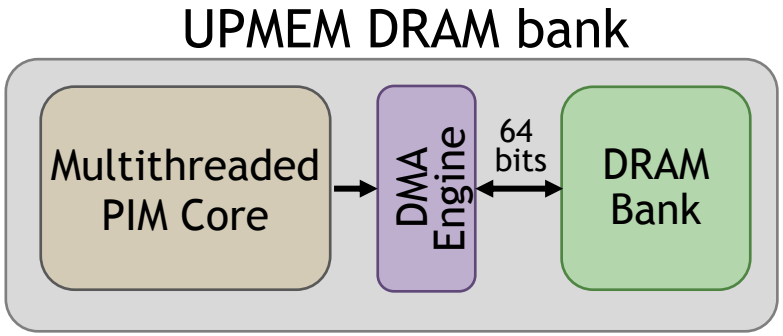
1. Load the input vector
2. Execute the kernel
3. Retrieve the partial results
4. Merge the partial results



Lock-Based Synchronization

16 threads, COO, 32-bit integer



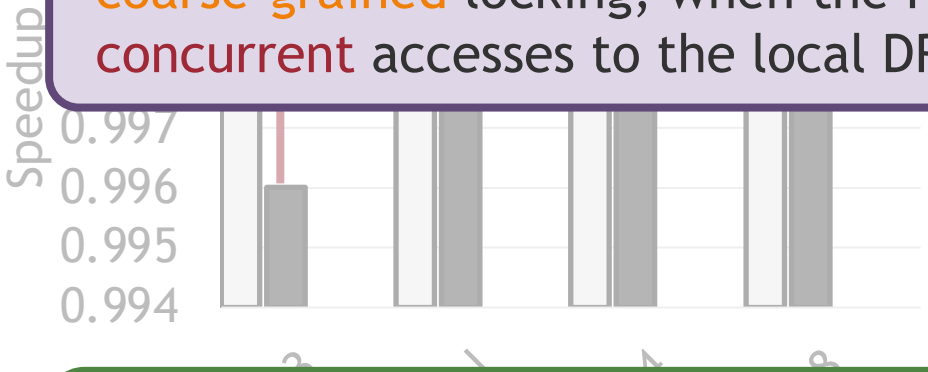


Fine-grained locking (lb-fg) does not improve performance over coarse-grained locking (lb-cg)

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

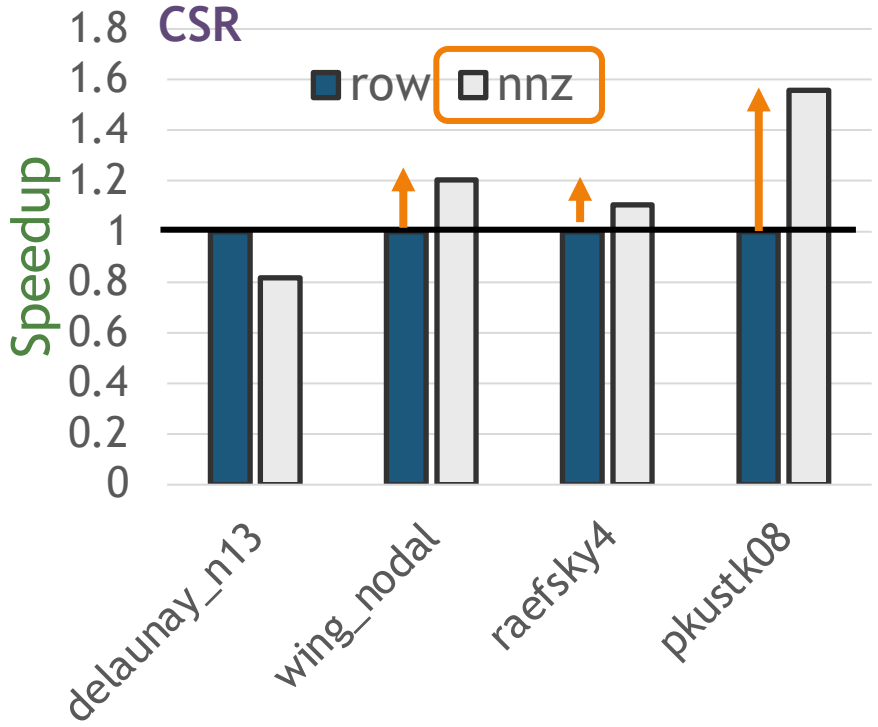
Recommendation 1

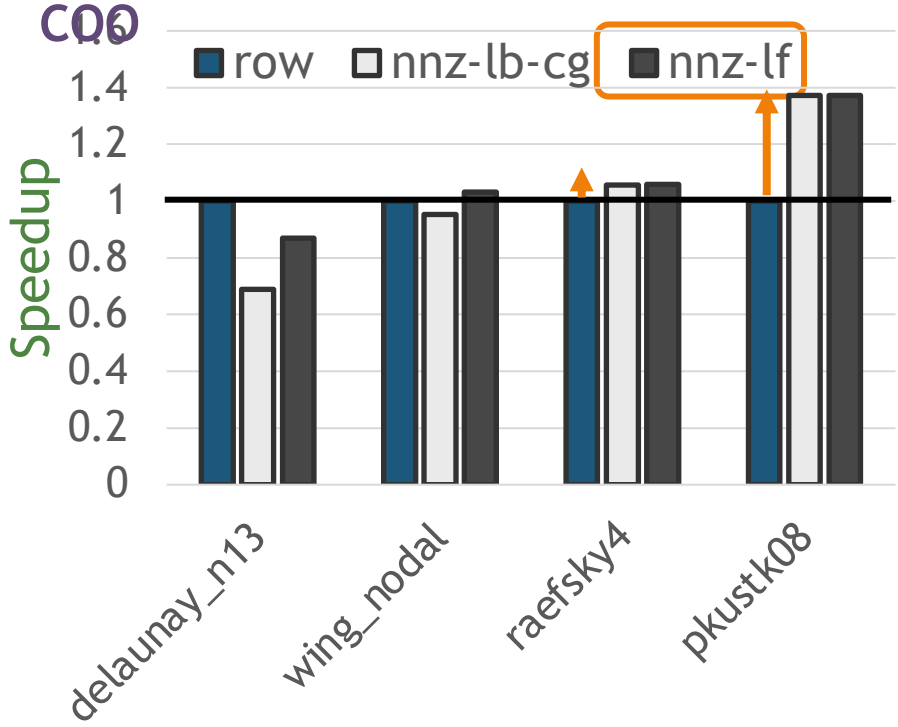
Provide low-cost synchronization support and hardware support to enable concurrent memory accesses to the local DRAM bank, and integrate multiple DRAM banks per PIM core to increase execution parallelism.

Load-Balance within a PIM Core

16 threads, 32-bit integer

Load-balancing #NNZs performs best in most matrices



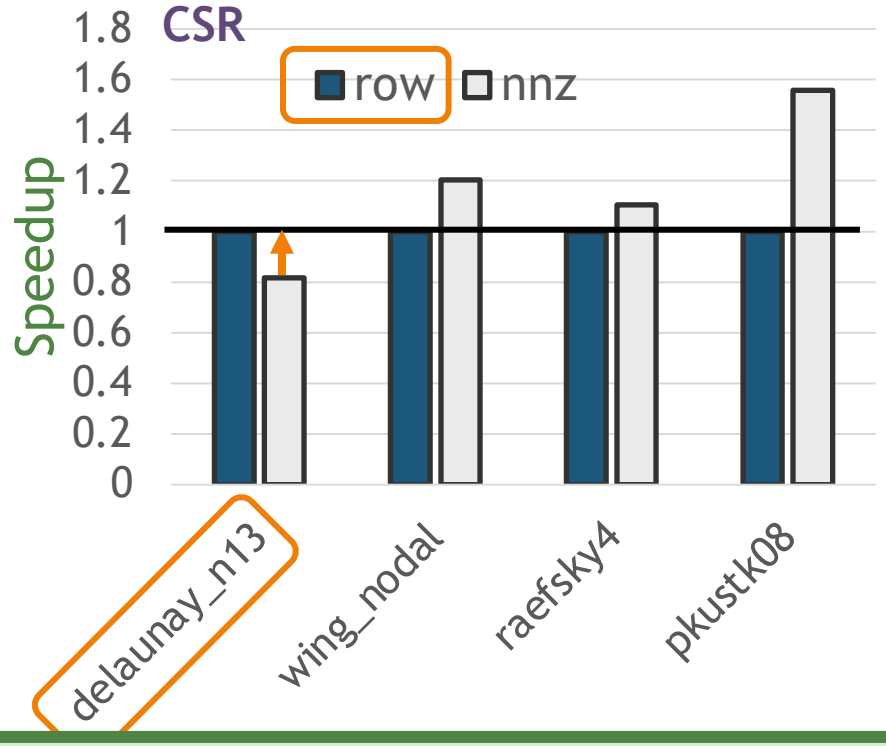


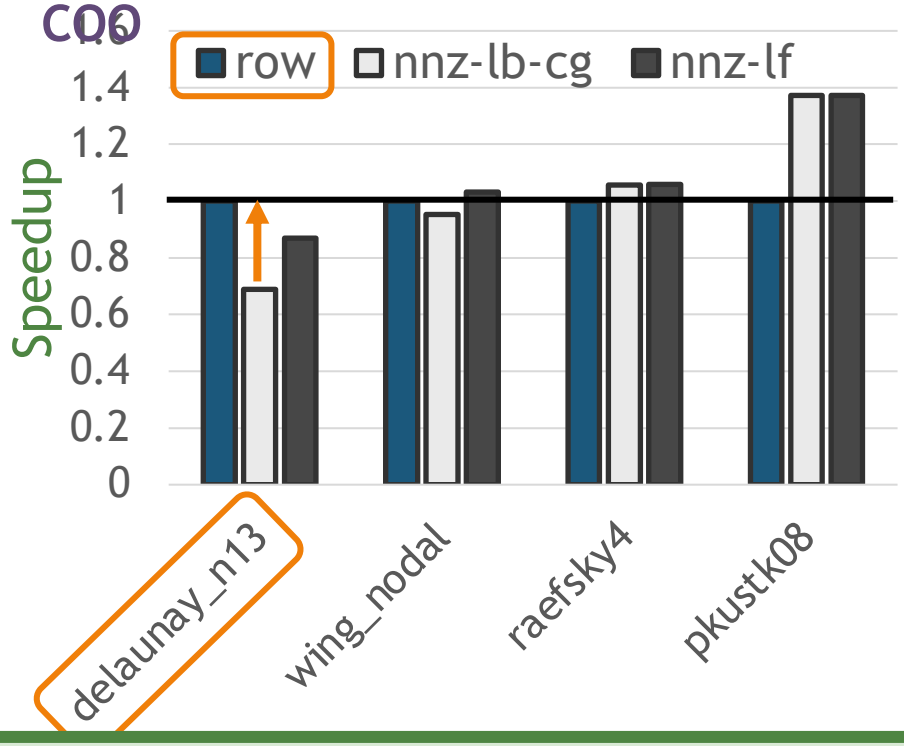
Load-balancing #NNZ typically provides high computation balance across threads of a compute-limited PIM core

Load-Balance within a PIM Core

16 threads, 32-bit integer

Load-balancing #NNZs causes high row imbalance





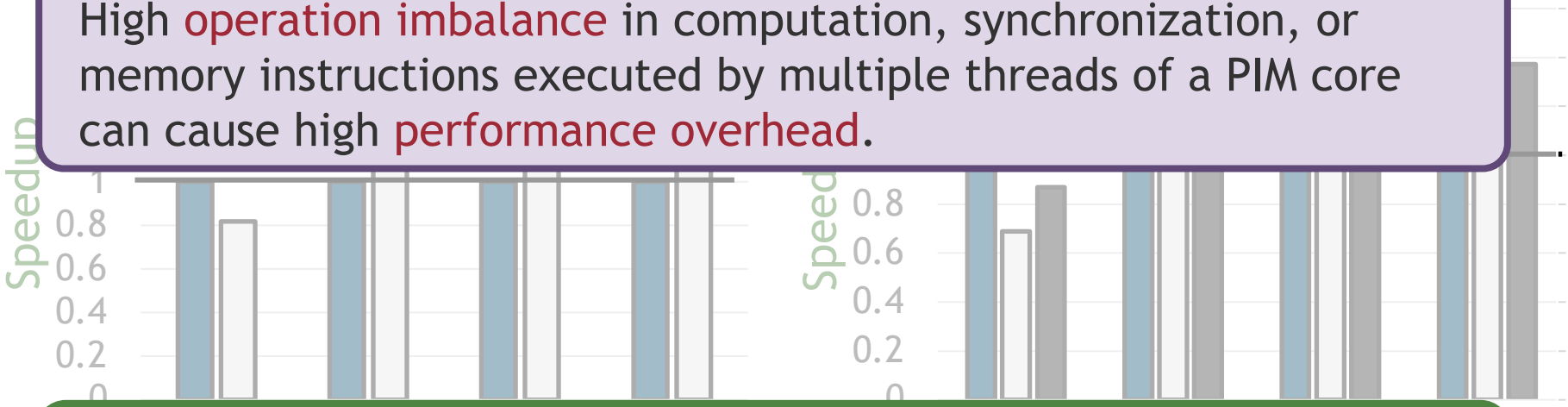
Load-balancing #NNZs: one single thread performs a much higher #memory accesses and #synchronization operations than the rest

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.

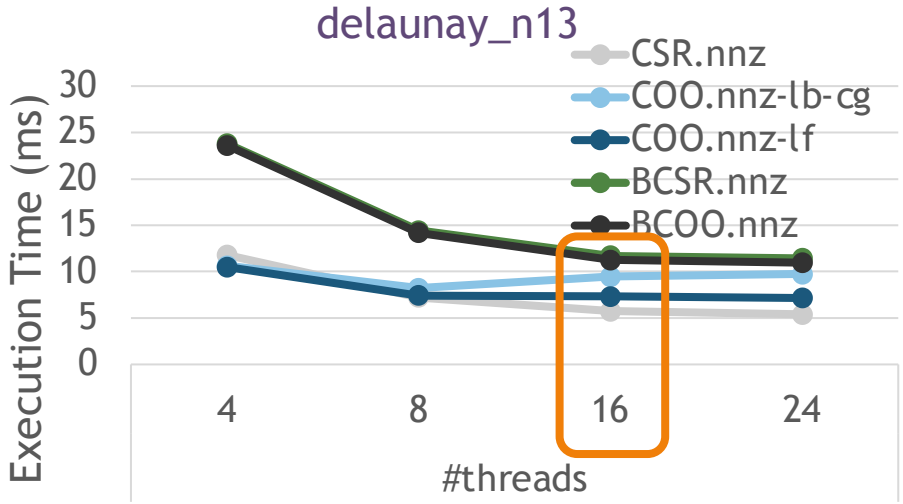


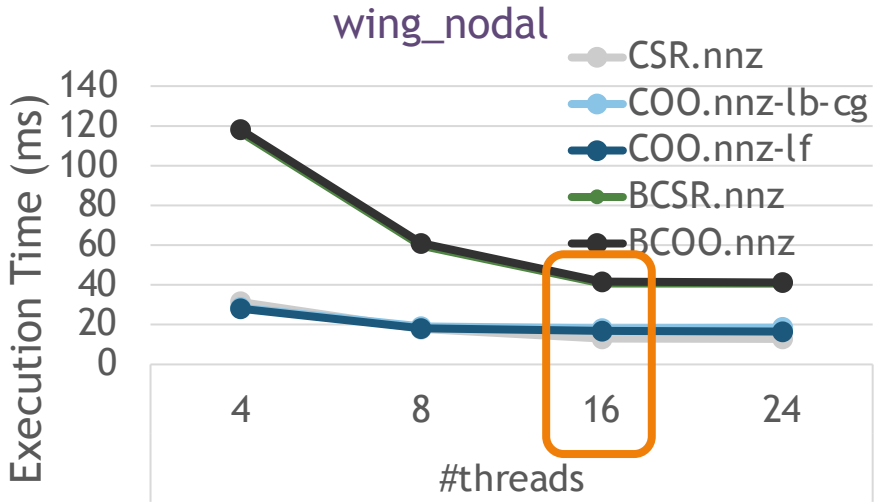
Recommendation 2

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

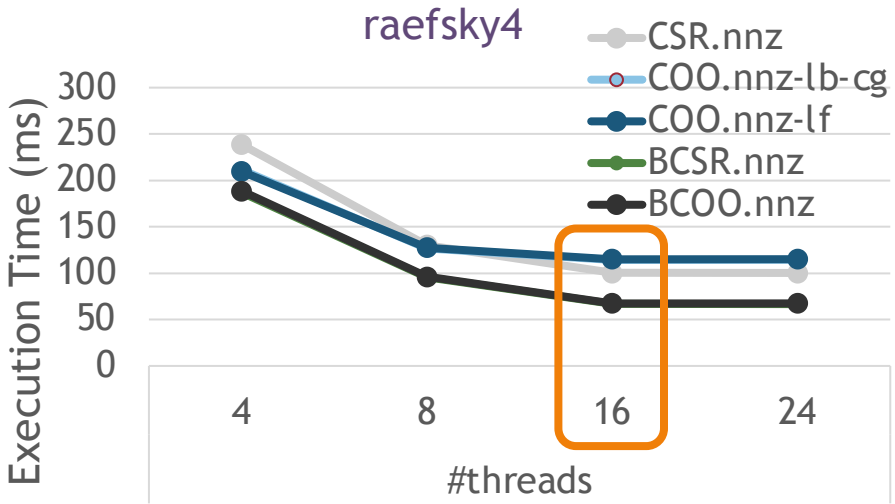
Scalability within a PIM Core

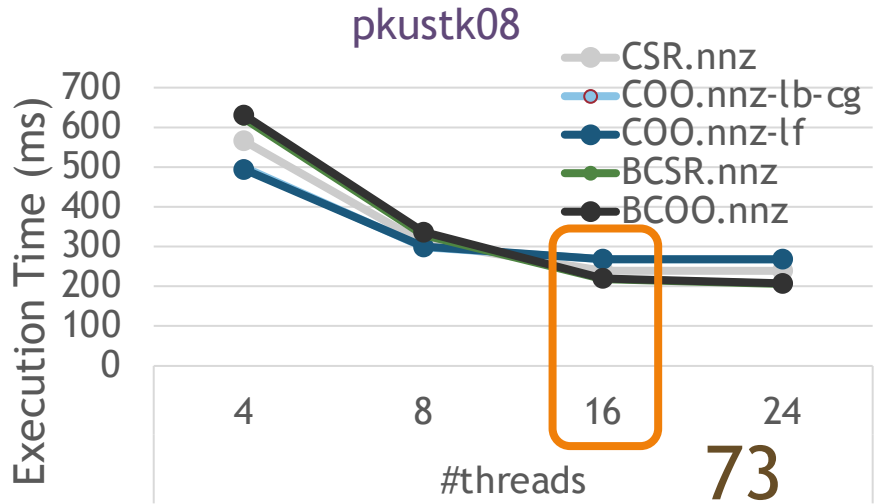
32-bit integer





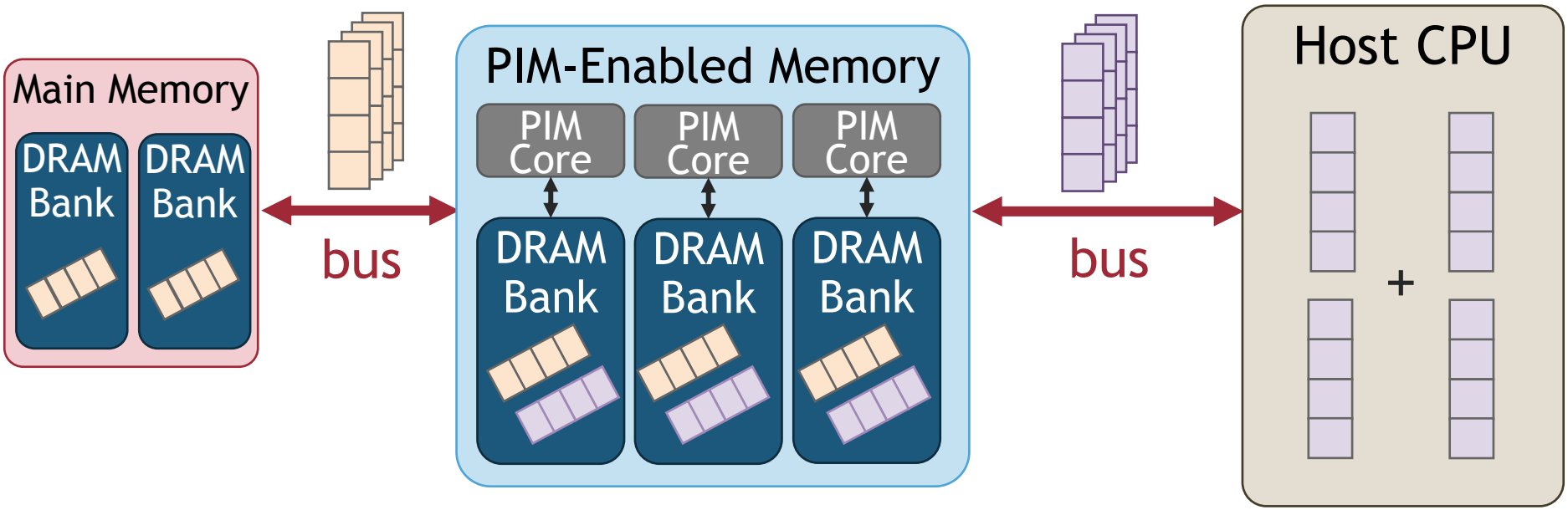
Scalability increases up to 16 threads





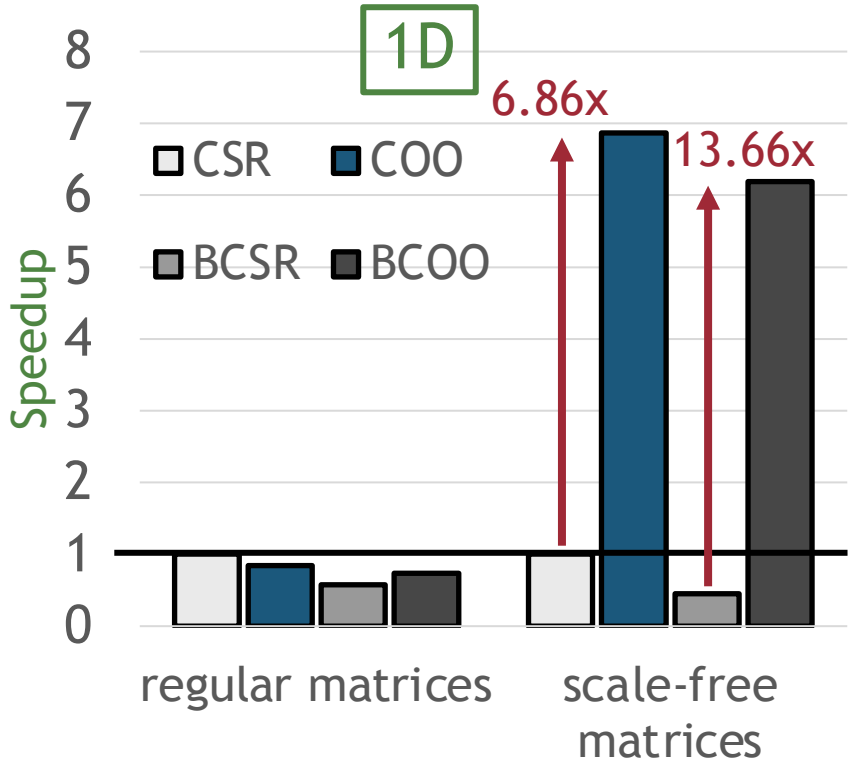
Kernel Execution on Multiple PIM Cores

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

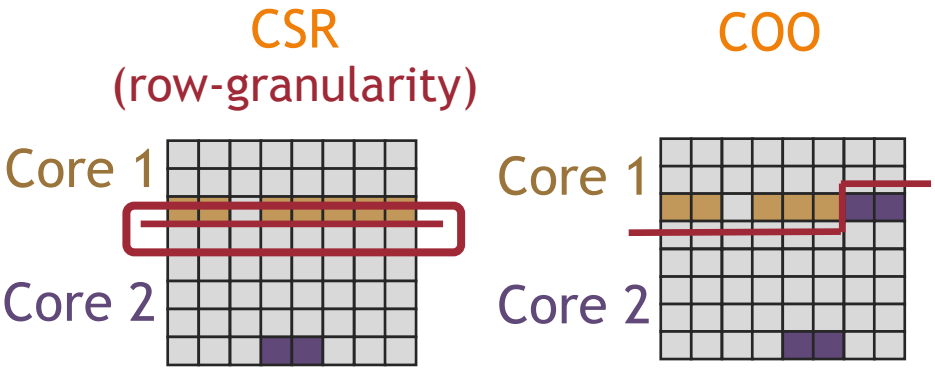


Comparison of Compressed Formats

2048 PIM Cores, 32-bit integer



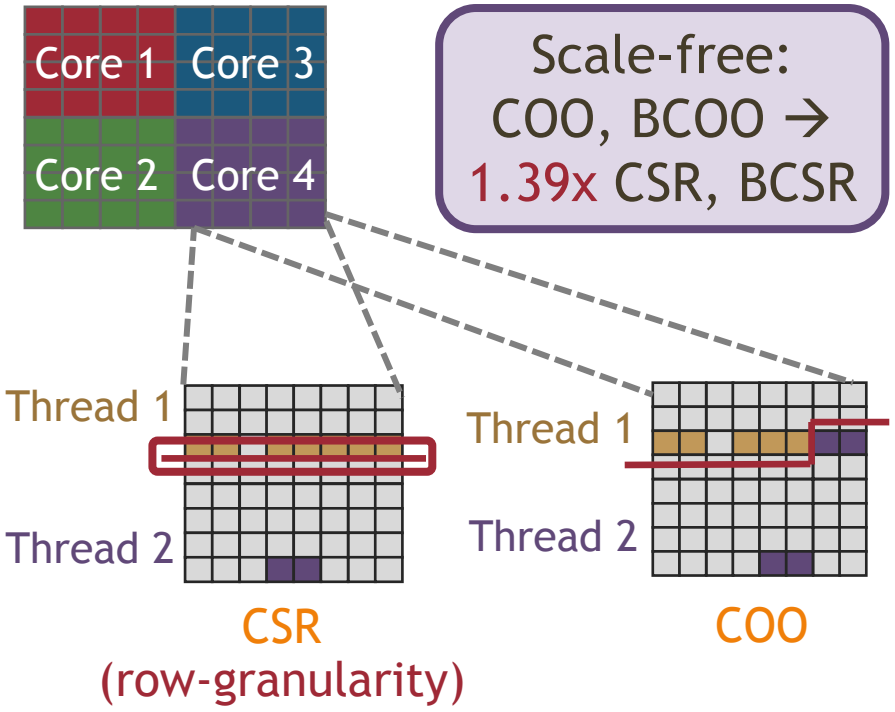
Scale-free: COO, BCOO → 10.26x CSR, BCSR

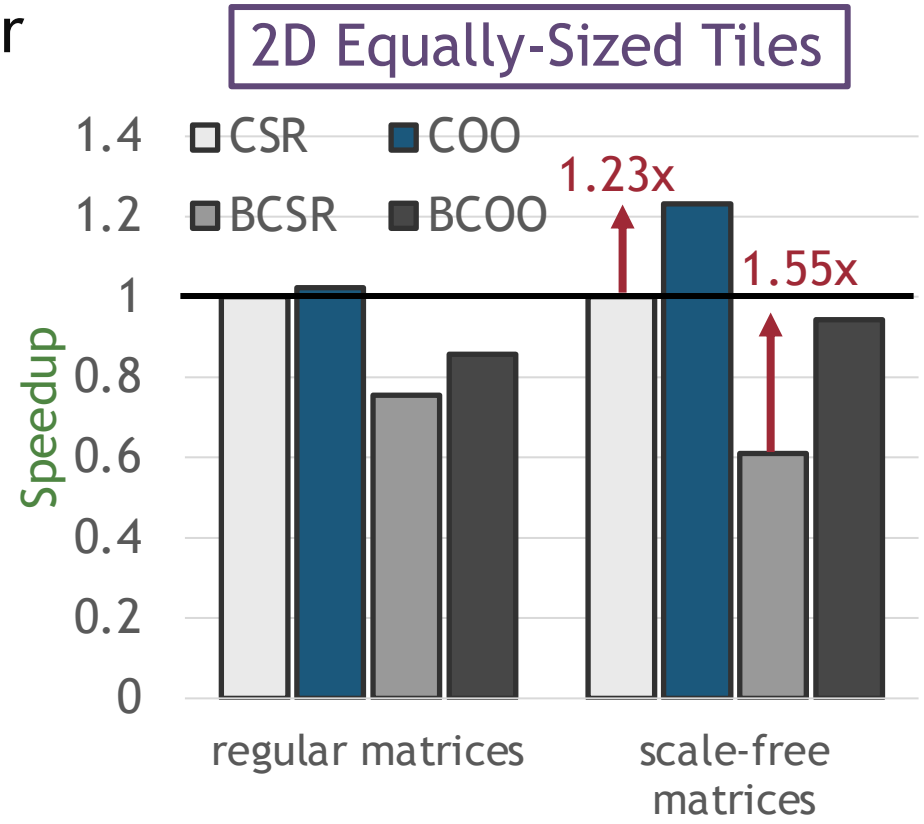


In scale-free matrices, COO + BCOO provide higher non-zero element balance across PIM cores than CSR + BCSR, respectively.

Comparison of Compressed Formats

2048 PIM Cores, 32-bit integer



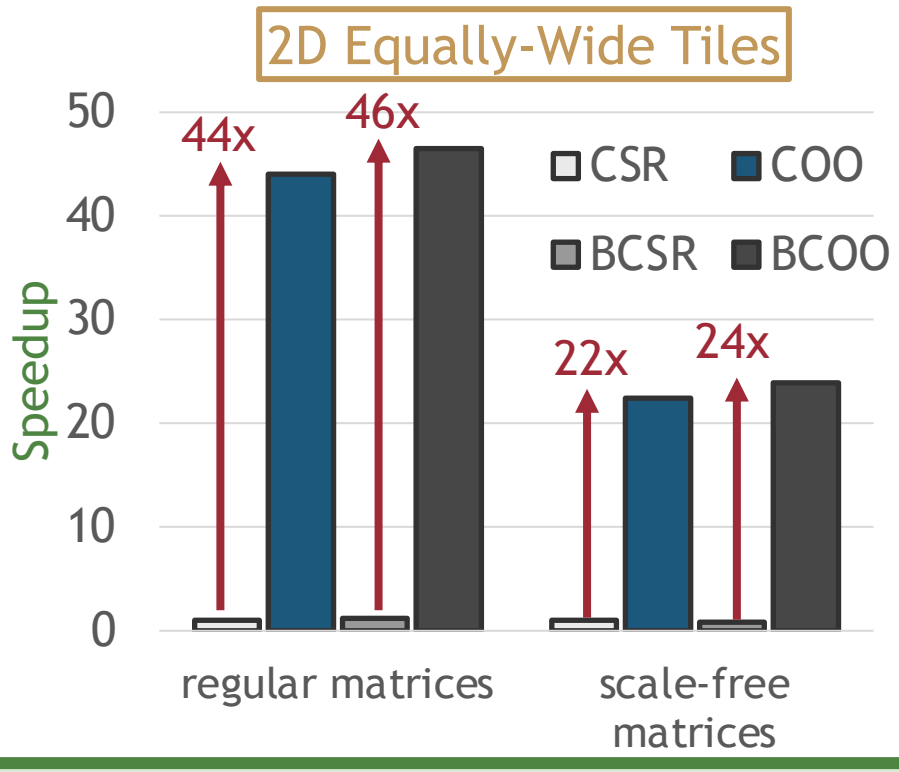


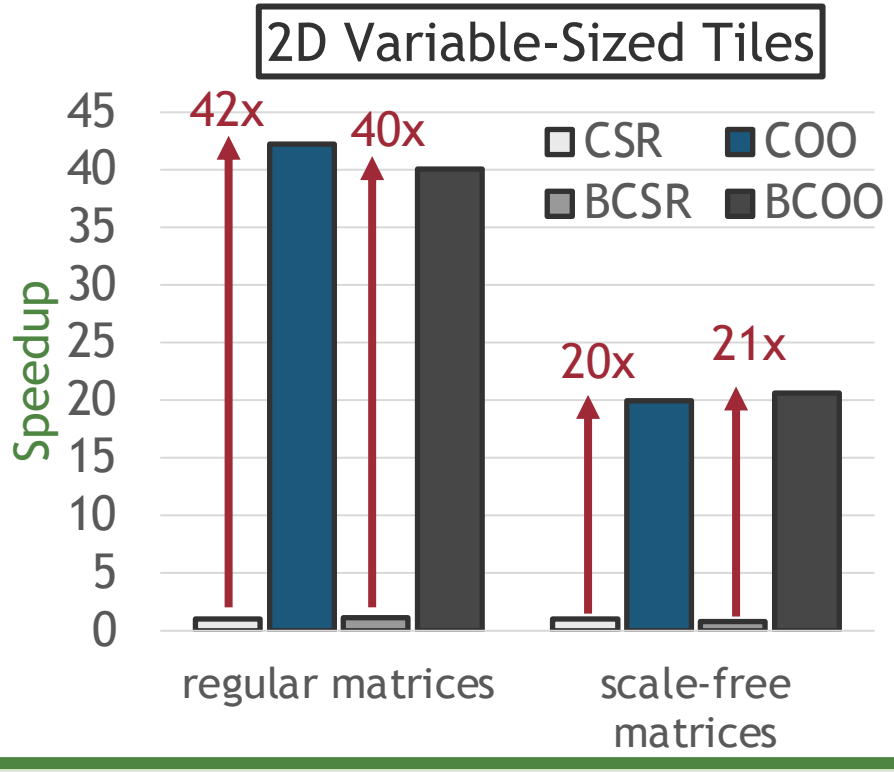
In scale-free matrices, COO + BCOO provide higher non-zero element balance across threads than CSR + BCSR, respectively.

Comparison of Compressed Formats

2048 PIM Cores, 32-bit integer

COO, BCOO → 32.38x CSR, BCSR

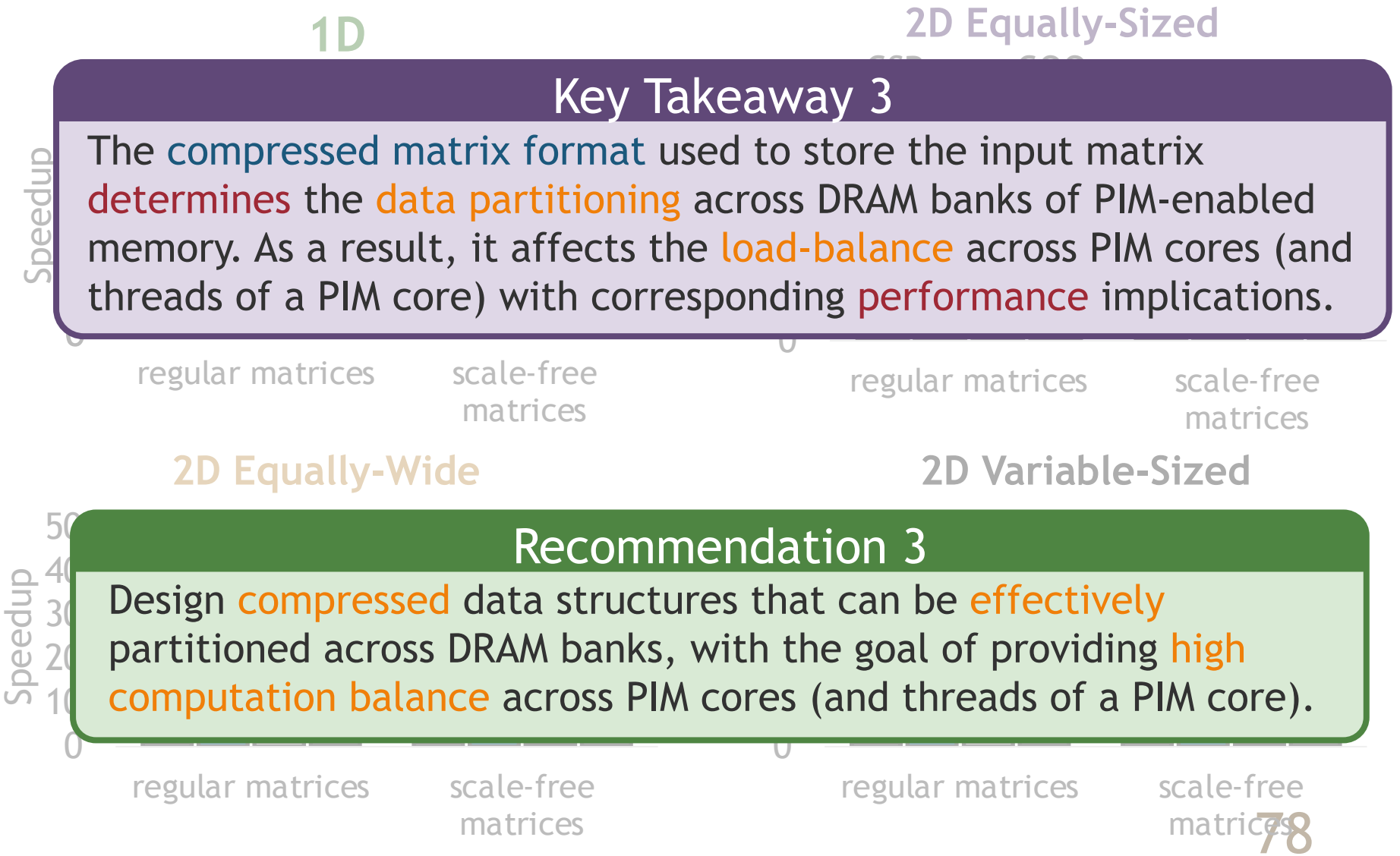




COO + BCOO formats provide higher non-zero element balance across PIM cores + threads than CSR + BCSR, respectively.

Comparison of Compressed Formats

2048 PIM Cores, 32-bit integer



Key Takeaway 3

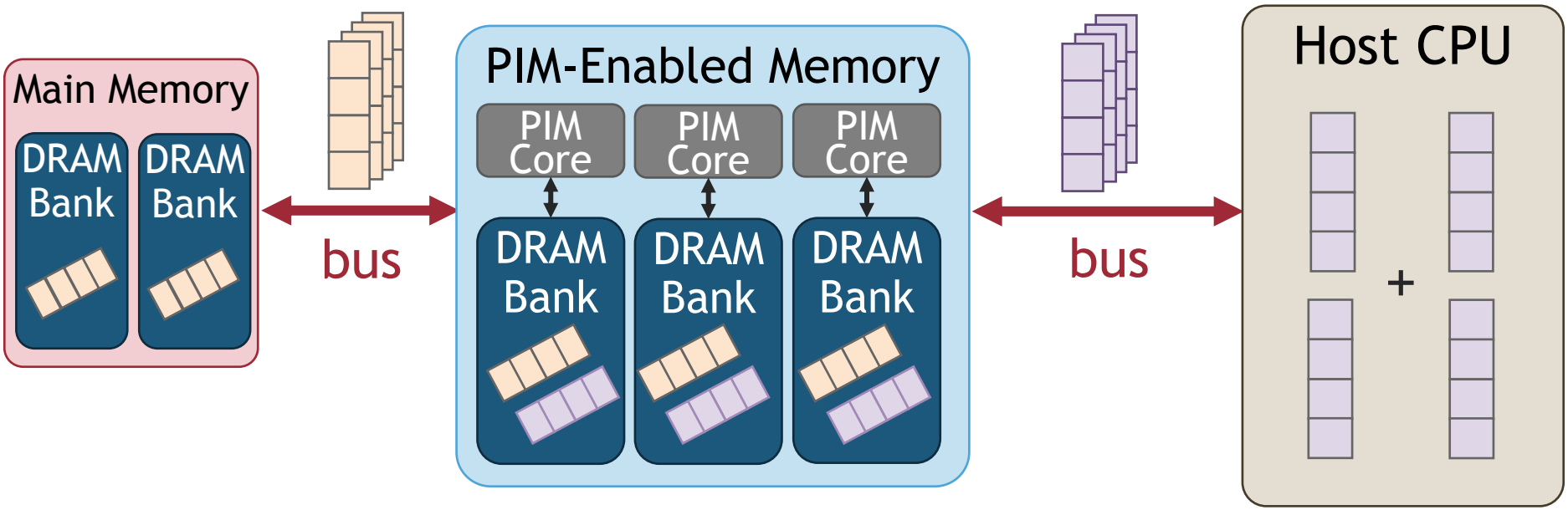
The compressed matrix format used to store the input matrix determines the data partitioning across DRAM banks of PIM-enabled memory. As a result, it affects the load-balance across PIM cores (and threads of a PIM core) with corresponding performance implications.

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

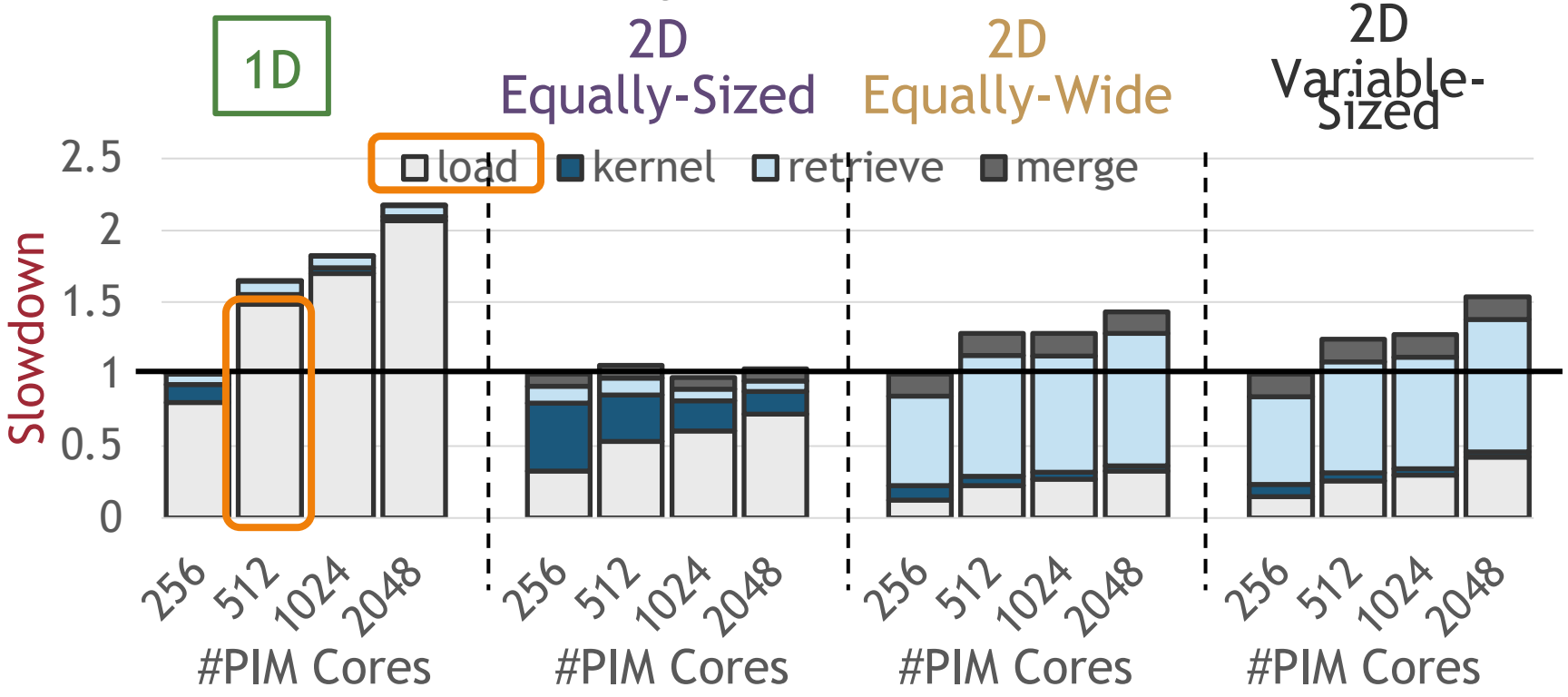
1. Load the input vector
2. Execute the kernel
3. Retrieve the partial results
4. Merge the partial results



Scalability

COO format, 32-bit integer

The scalability is limited by the load time



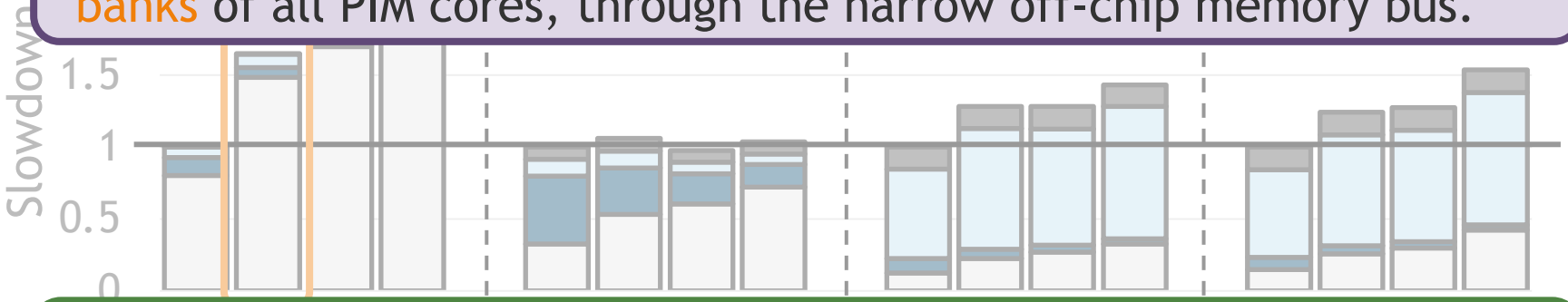
1D: #bytes to load the input vector grows linearly to #PIM cores

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.



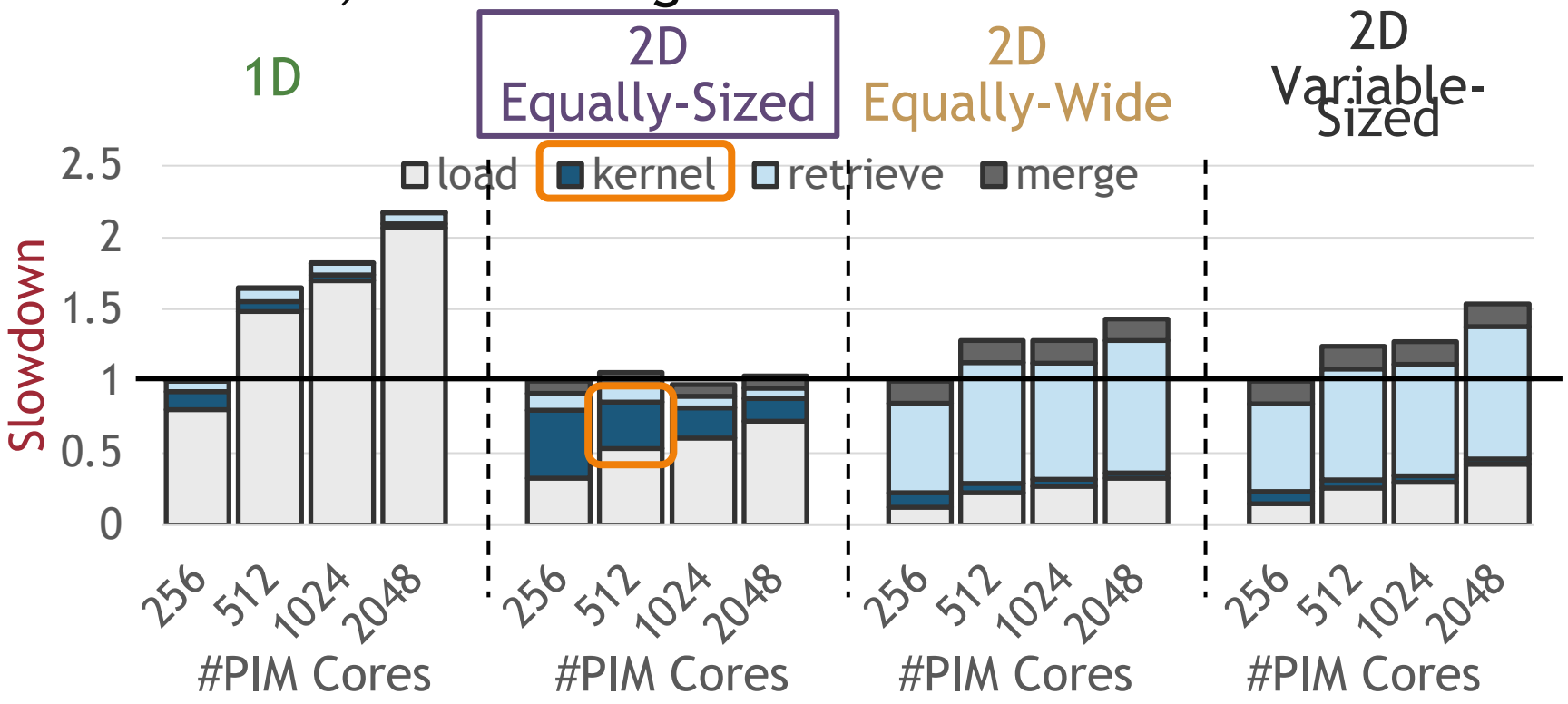
Recommendation 4

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

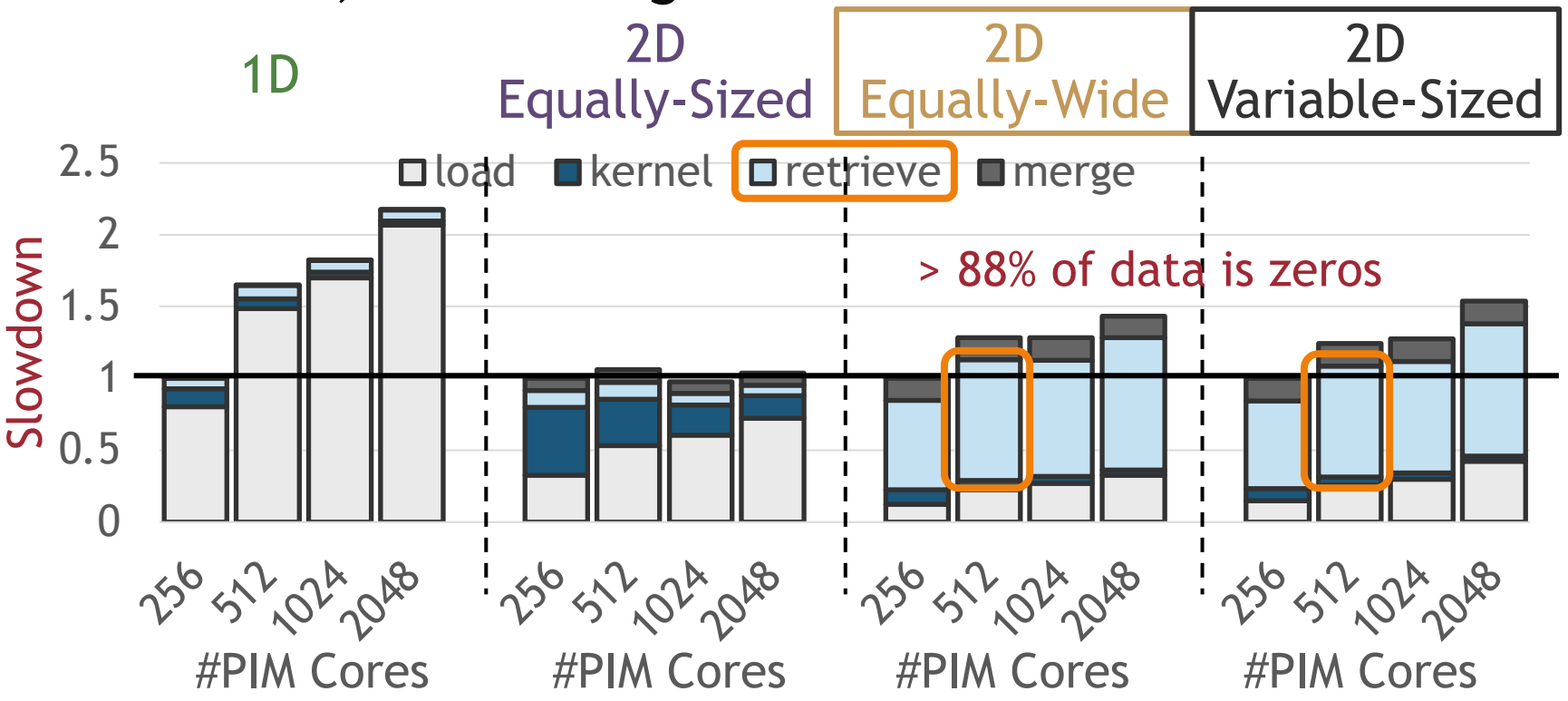


2D Equally-Sized: kernel time is limited by only a few PIM cores assigned to the 2D tiles with the largest #NNZs

Scalability

COO format, 32-bit integer

The scalability is limited by the retrieve time



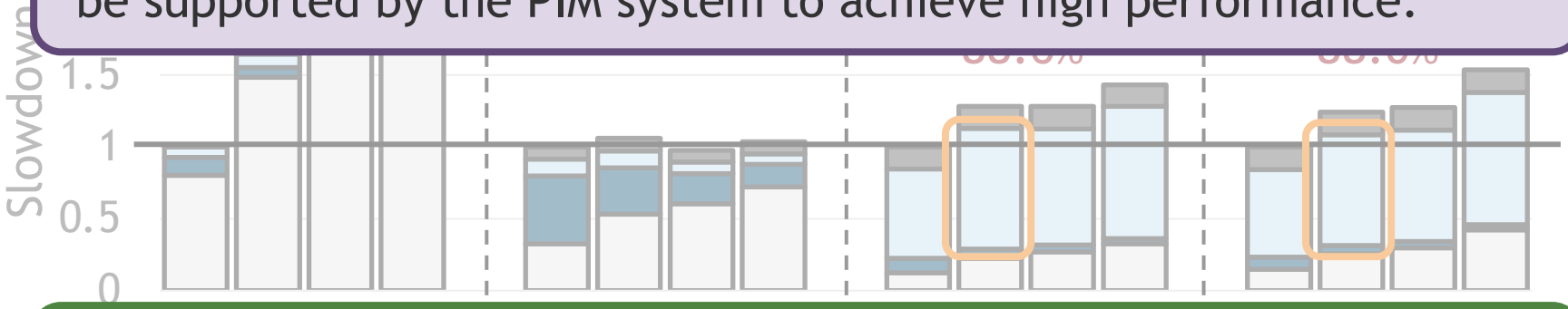
2D Equally-Wide + 2D Variable-Sized:
high amount of zero padding to gather the output vector →
parallel transfers supported at rank granularity = 64 PIM cores

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.

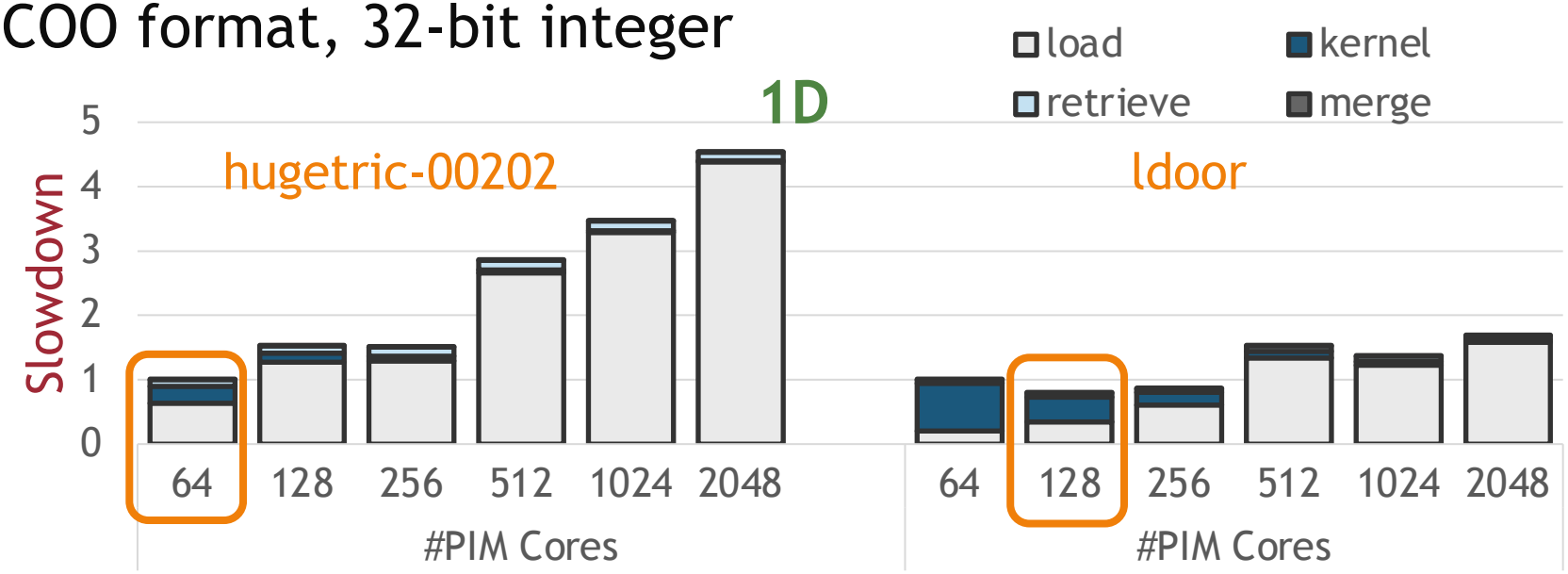


Recommendation 5

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



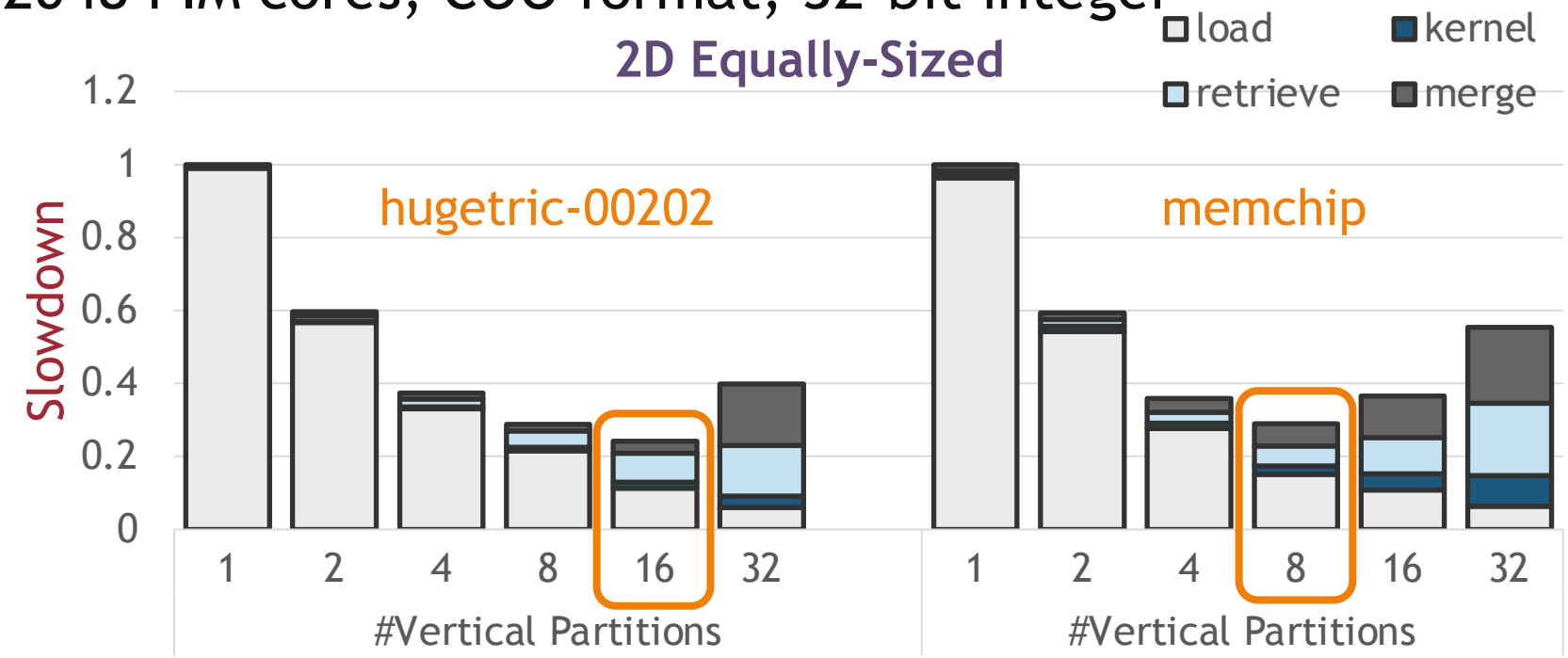
Best-performing = 64 PIM cores

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



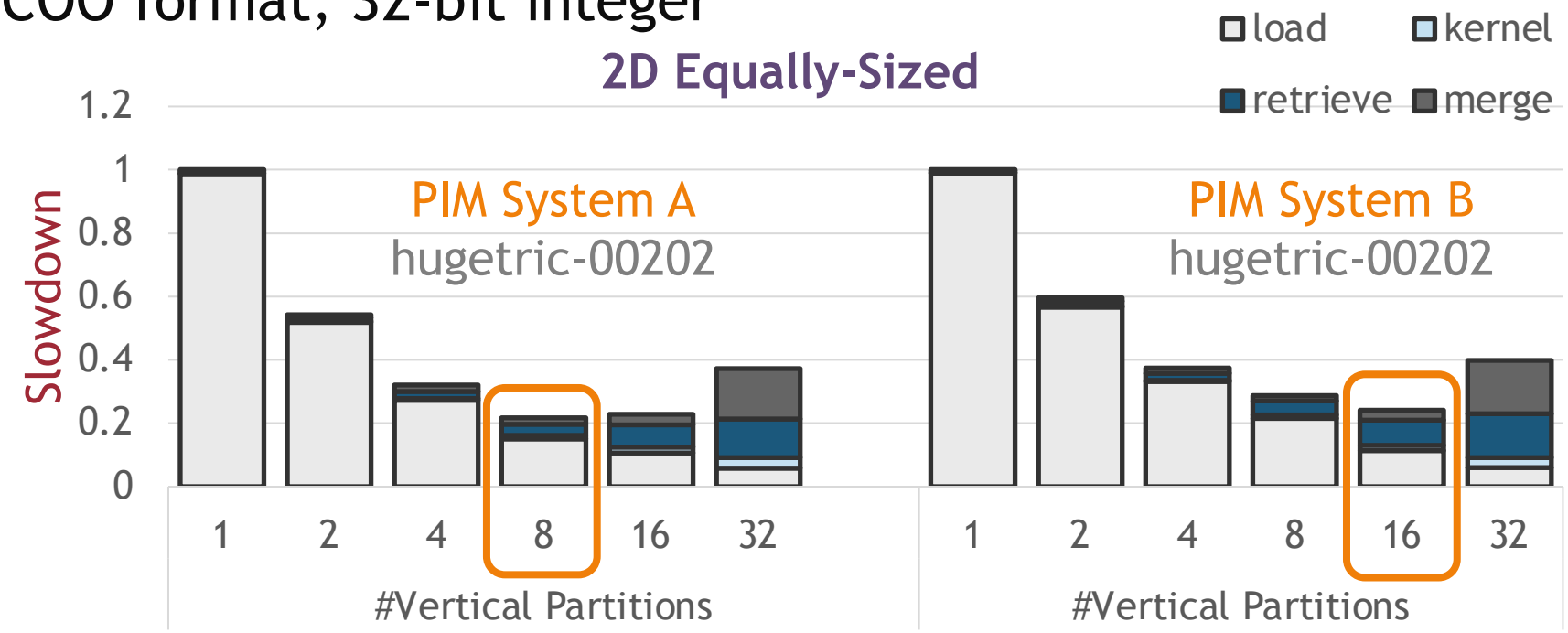
Best-performing = 16 vertical part.

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



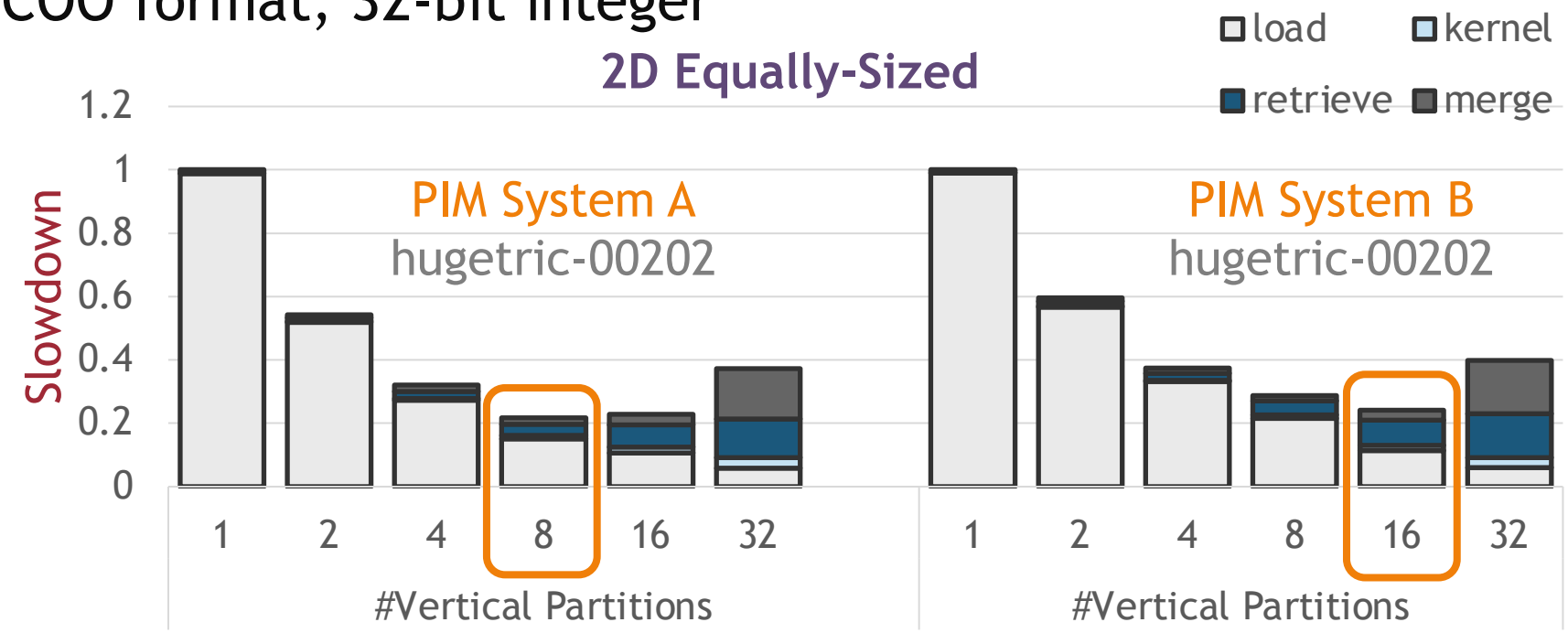
Best-performing = 8 vertical part.

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



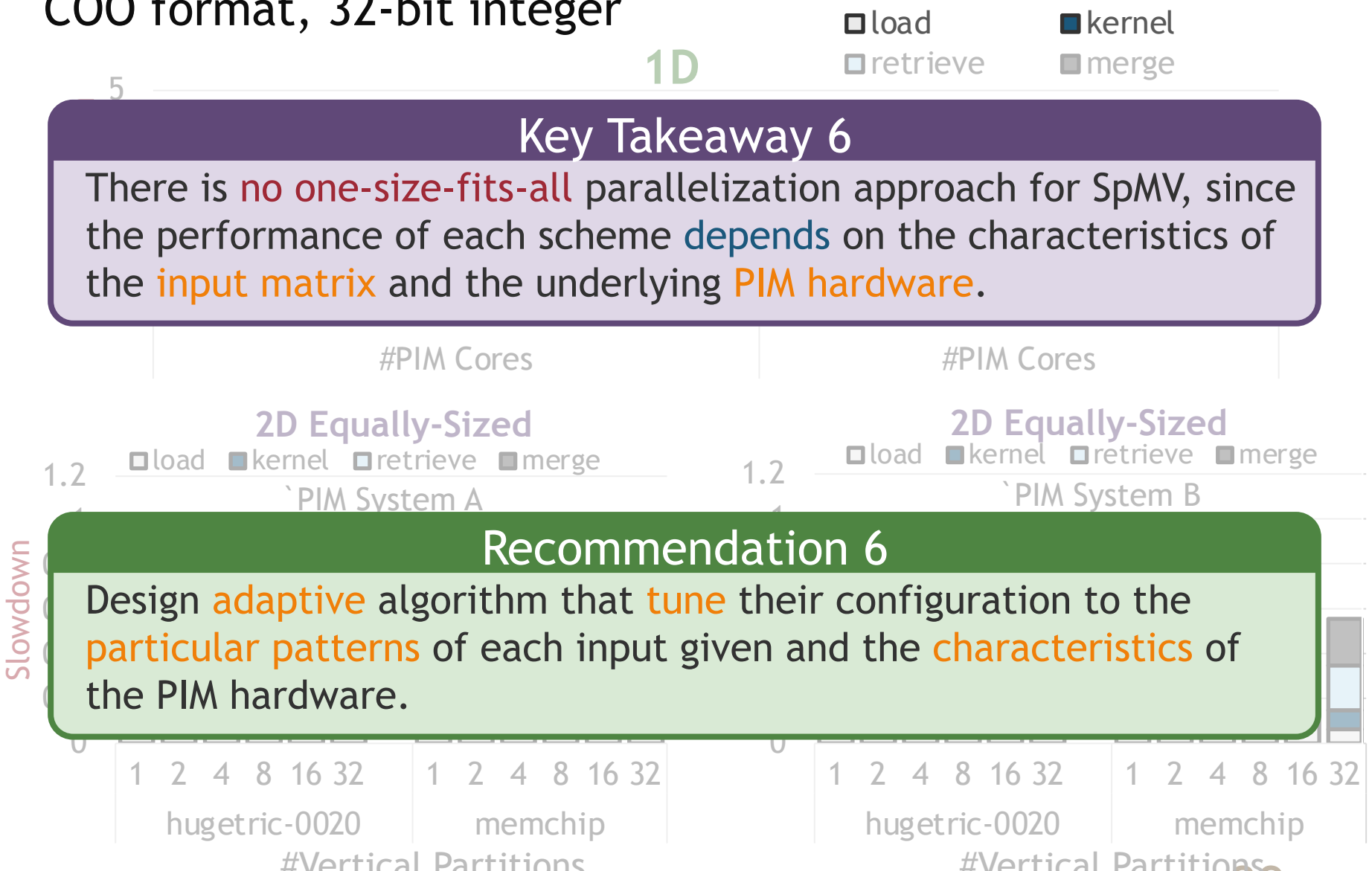
Best-performing = 8 vertical part.

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

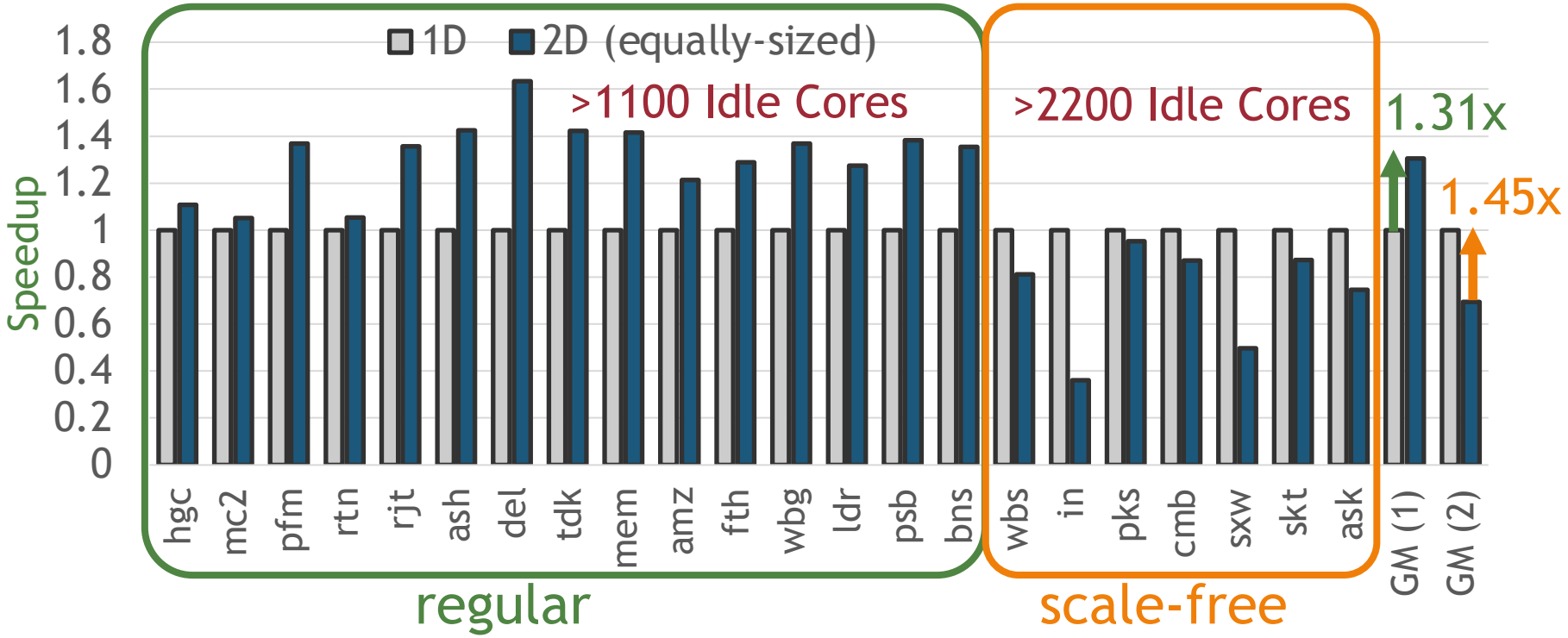
There is no one-size-fits-all parallelization approach for SpMV, since the performance of each scheme depends on the characteristics of the input matrix and the underlying PIM hardware.

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

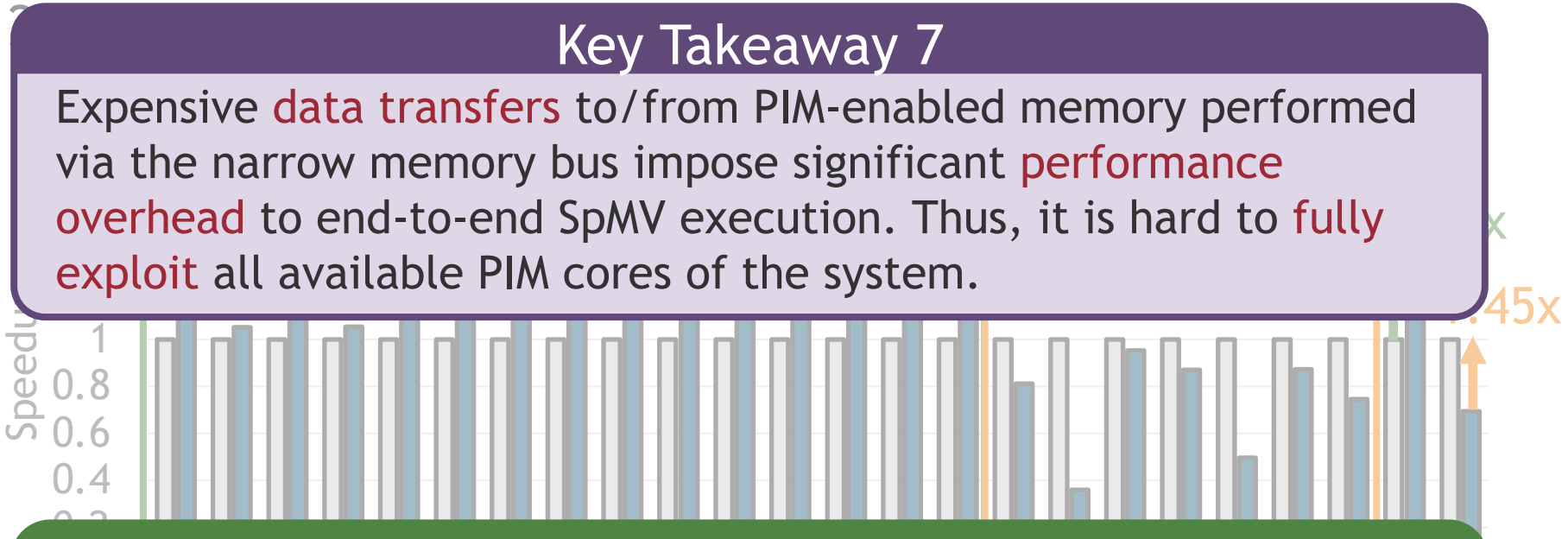


Best-performing SpMV execution:
trades off computation with lower data transfer costs

1D vs 2D

Key Takeaway 7

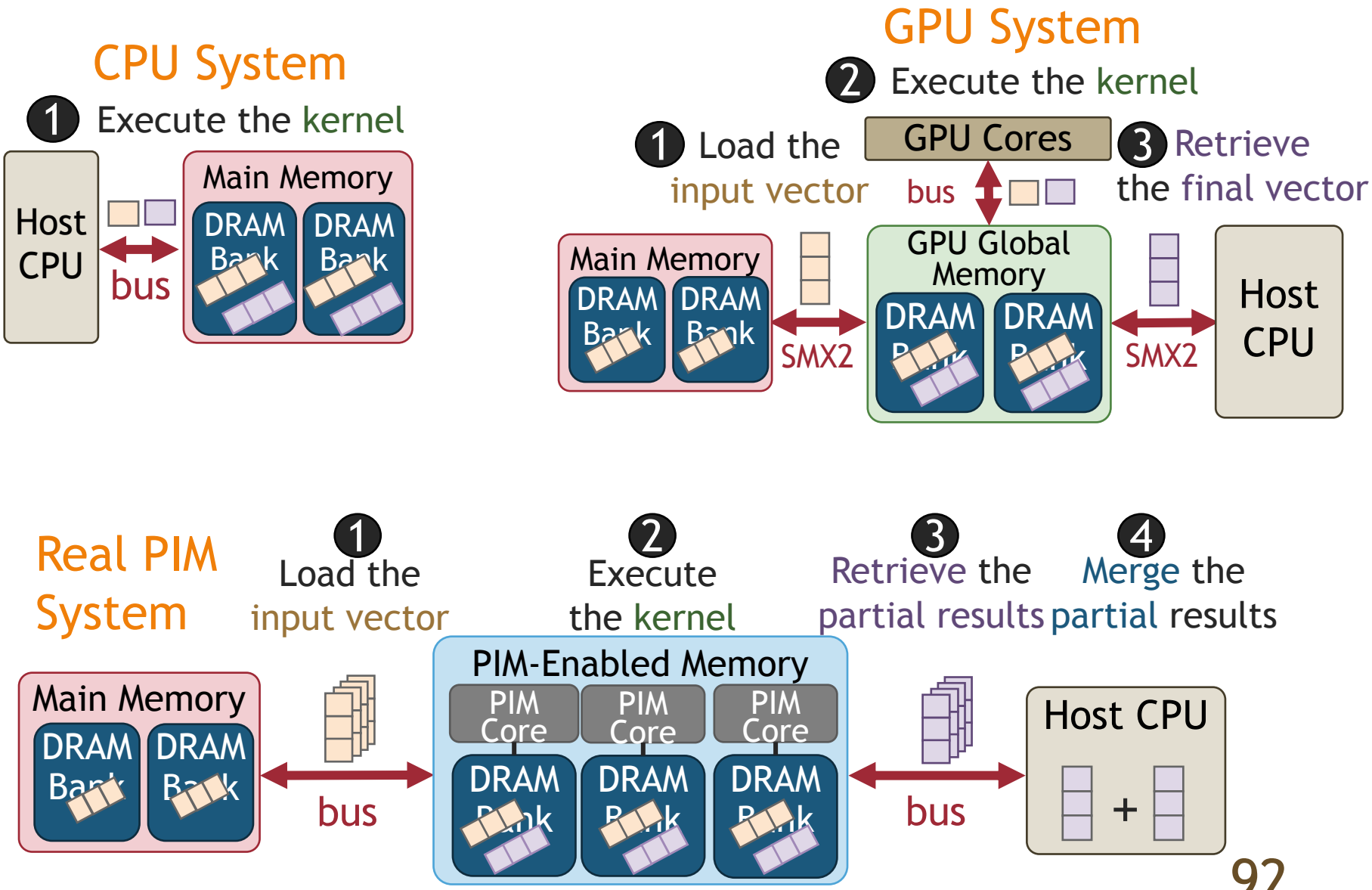
Expensive data transfers to/from PIM-enabled memory performed via the narrow memory bus impose significant performance overhead to end-to-end SpMV execution. Thus, it is hard to fully exploit all available PIM cores of the system.



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
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

Processor-Centric

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
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

Processor-Centric

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
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

Processor-Centric

Memory-Centric

CPU/GPU Comparisons

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

Processor-Centric

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.			

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

Centric

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 Paper and Repo

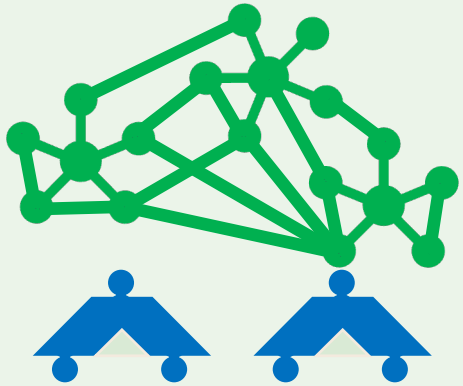
- Appears at SIGMETRICS 2022

SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Systems

CHRISTINA GIANNOULA, ETH Zürich, Switzerland and National Technical University of Athens, Greece
IVAN FERNANDEZ, ETH Zürich, Switzerland and University of Malaga, Spain
JUAN GÓMEZ-LUNA, ETH Zürich, Switzerland
NECTARIOS KOZIRIS, National Technical University of Athens, Greece
GEORGIOS GOUMAS, National Technical University of Athens, Greece
ONUR MUTLU, ETH Zürich, Switzerland

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

https://www.youtube.com/watch?v=5kaOsJKlGrE



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

Machine Learning Training

An Experimental Evaluation of Machine Learning Training on a Real Processing-in-Memory System

Juan Gómez Luna, Yuxin Guo, Sylvan Brocard,
Julien Legriel, Remy Cimadomo, Geraldo F. Oliveira,
Gagandeep Singh, Onur Mutlu

https://arxiv.org/pdf/2207.07886.pdf
https://github.com/CMU-SAFARI/pim-ml
juang@ethz.ch

ETH Zürich SAFARI upmem

Thursday, May 25, 2023

ISPASS 2023 Version

- Presented at ISPASS 2023

Evaluating Machine Learning Workloads on Memory-Centric Computing Systems

Juan Gómez-Luna[1] Yuxin Guo[1] Sylvan Brocard[2] Julien Legriel[2]
Remy Cimadomo[2] Geraldo F. Oliveira[1] Gagandeep Singh[1] Onur Mutlu[1]
[1]ETH Zürich [2]UPMEM

https://people.inf.ethz.ch/omutlu/pub/MLonUPMEM-PIM_ispass23.pdf

Source code: https://github.com/CMU-SAFARI/pim-ml

https://youtu.be/60pkaI5AeM4

Executive Summary

- Training machine learning (ML) algorithms is a computationally expensive process, frequently memory-bound due to repeatedly accessing large training datasets
- Memory-centric computing systems, i.e., with Processing-in-Memory (PIM) capabilities, can alleviate this *data movement bottleneck*
- Real-world PIM systems have only recently been manufactured and commercialized
  - UPMEM has designed and fabricated the first publicly-available real-world PIM architecture
- Our goal is to understand the potential of modern general-purpose PIM architectures to accelerate machine learning training
- Our main contributions:
  - PIM implementation of several classic machine learning algorithms: linear regression, logistic regression, decision tree, K-means clustering
  - Workload characterization in terms of quality, performance, and scaling
  - Comparison to their counterpart implementations on processor-centric systems (CPU and GPU)
    - PIM version of `DTR` is **27x** / **1.34x** faster than the CPU / GPU version, respectively
    - PIM version of `KME` is **2.8x** / **3.2x** faster than the CPU / GPU version, respectively
  - Source code: https://github.com/CMU-SAFARI/pim-ml
- Experimental evaluation on a real-world PIM system with 2,524 PIM cores @ 425 MHz and 158 GB of DRAM memory
- Key observations, takeaways, and recommendations for ML workloads on general-purpose PIM systems

Outline

Machine learning workloads

Processing-in-memory

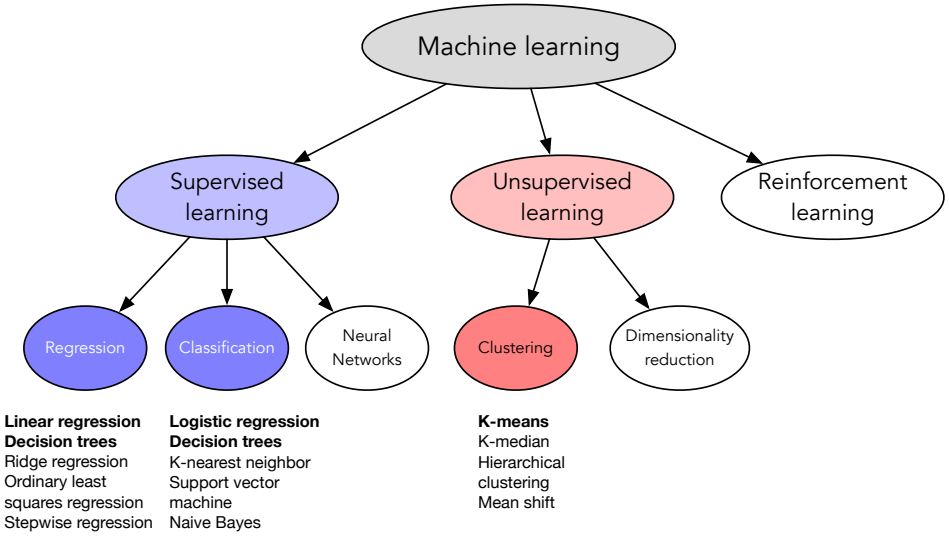
PIM implementation of ML workloads

Evaluation

- Quality Metrics
- Analysis of PIM Kernels
- Performance Scaling
- Comparison to CPU and GPU

Machine Learning Workloads

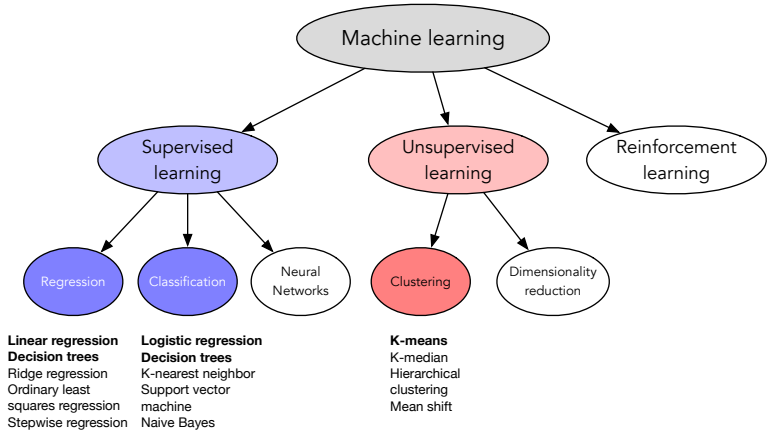
- Machine learning training with large amounts of data is a computationally expensive process, which requires many iterations to update an ML model's parameters

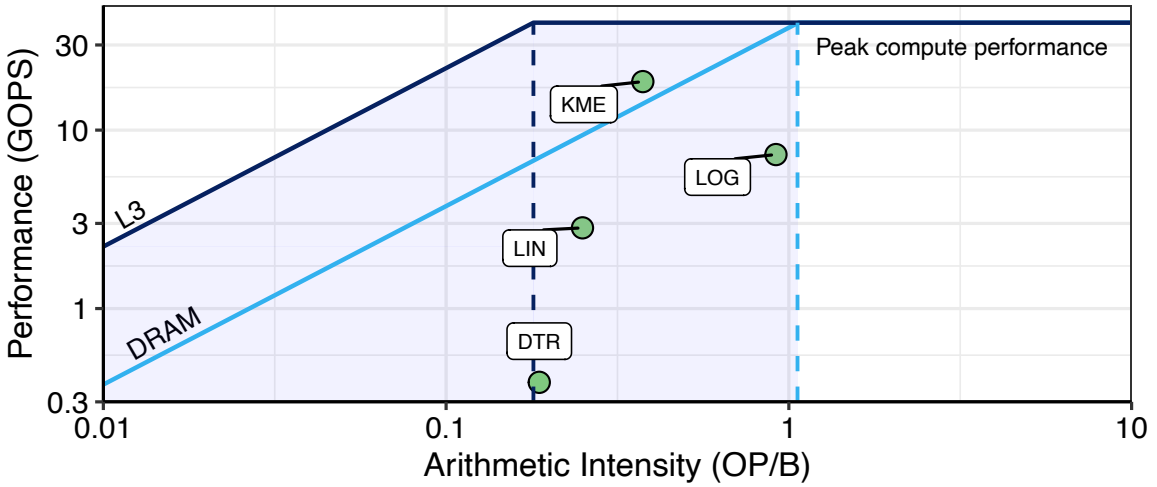


- Frequent data movement between memory and processing elements to access training data
- The amount of computation is not enough to amortize the cost of moving training data to the processing elements
  - Low arithmetic intensity
  - Low temporal locality
  - Irregular memory accesses

Machine Learning Workloads: Our Goal

- Our goal is to study and analyze how real-world general-purpose PIM can accelerate ML training
- Four representative ML algorithms: linear regression, logistic regression, decision tree, K-means
- Roofline model to quantify the memory boundedness of CPU versions of the four workloads





All workloads fall in the memory-bound area of the Roofline

Outline

Machine learning workloads

Processing-in-memory

PIM implementation of ML workloads

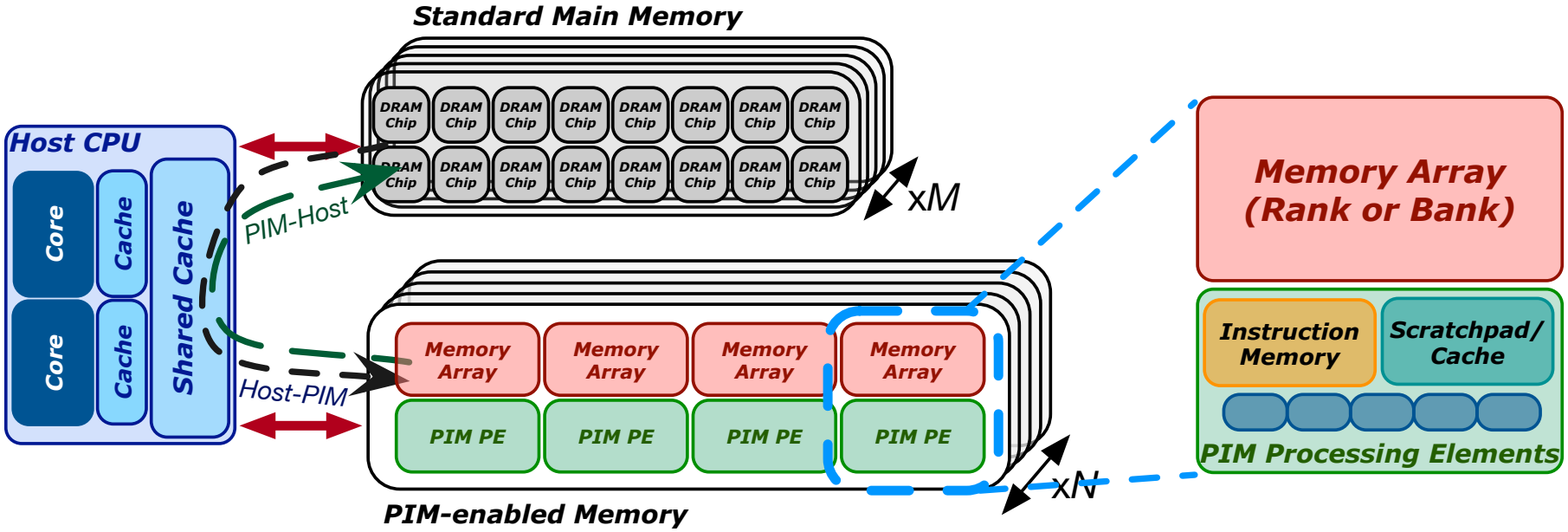
Evaluation

- Quality Metrics
- Analysis of PIM Kernels
- Performance Scaling
- Comparison to CPU and GPU

Processing-in-Memory (PIM)

- PIM is a computing paradigm that advocates for memory-centric computing systems, where processing elements are placed near or inside the memory arrays
- Real-world PIM architectures are becoming a reality
  - UPMEM PIM, Samsung HBM-PIM, Samsung AxDIMM, SK Hynix AiM, Alibaba HB-PNM
- These PIM systems have some common characteristics:
  1. There is a host processor (CPU or GPU) with access to (1) standard main memory, and (2) PIM-enabled memory
  2. PIM-enabled memory contains multiple PIM processing elements (PEs) with high bandwidth and low latency memory access
  3. PIM PEs run only at a few hundred MHz and have a small number of registers and small (or no) cache/scratchpad
  4. PIM PEs may need to communicate via the host processor

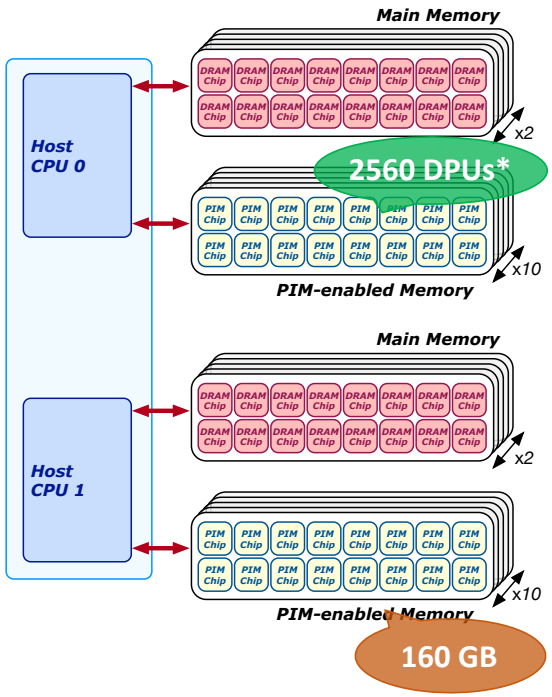
A State-of-the-Art PIM System



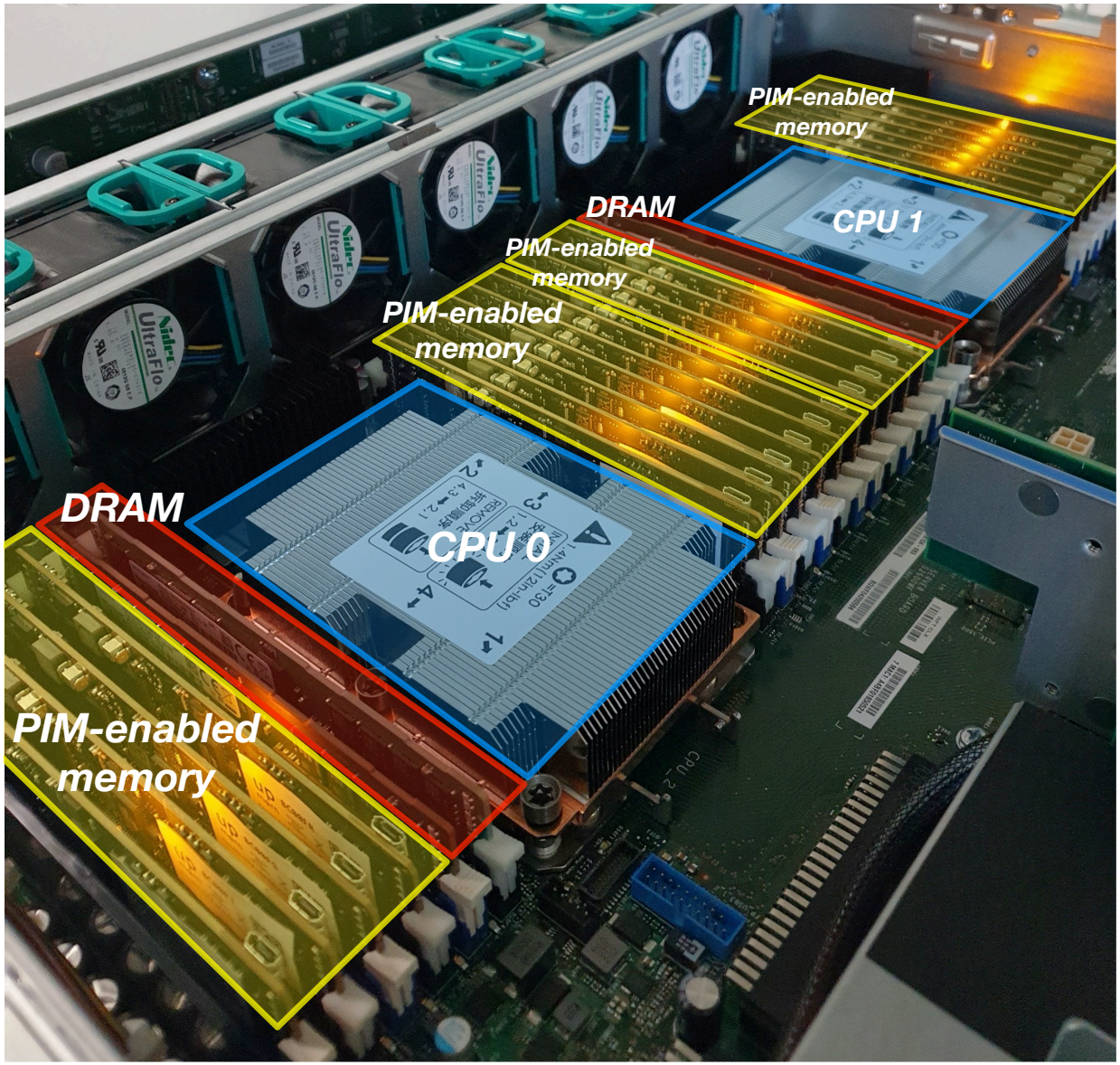
- In our work, we use the UPMEM PIM architecture
  - General-purpose processing cores called *DRAM Processing Units* (*DPUs*)
    - Up to 24 PIM threads, called *tasklets*
    - 32-bit integer arithmetic, but multiplication/division are emulated*, as well as floating-point operations
  - 64-MB DRAM bank (*MRAM*), 64-KB scratchpad (*WRAM*)

* 8-bit integer multiplication is natively supported

2,560-DPU UPMEM PIM System



- 20 UPMEM DIMMs of 16 chips each (40 ranks)
- Dual x86 socket
- UPMEM DIMMs coexist with regular DDR4 DIMMs
  - 2 memory controllers/socket (3 channels each)
  - 2 conventional DDR4 DIMMs on one channel of one controller



* There are some faulty DPUs in the system that we use in our experiments. Thus, the maximum number of DPUs we can use is 2,524

Outline

Machine learning workloads

Processing-in-memory

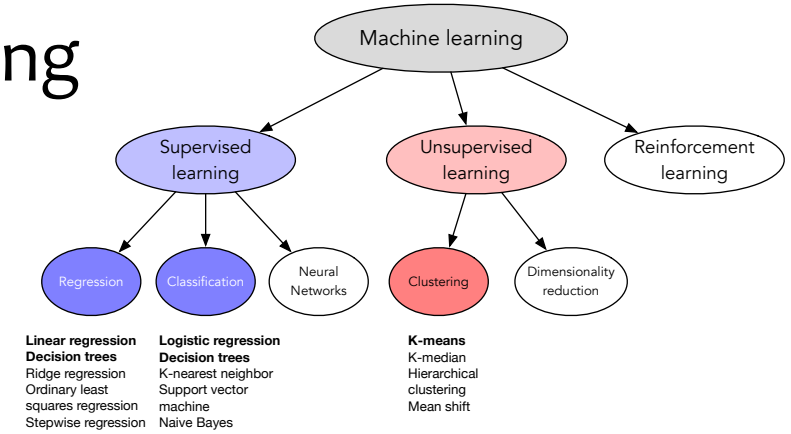
PIM implementation of ML workloads

Evaluation

- Quality Metrics
- Analysis of PIM Kernels
- Performance Scaling
- Comparison to CPU and GPU

ML Training Workloads

- Four widely-used machine learning workloads:
  - Linear regression (LIN)
  - Logistic regression (LOG)
  - Decision tree (DTR)
  - K-means clustering (KME)



- Diversity of our ML training workloads:
  - Memory access patterns
  - Operations and datatypes
  - Communication/synchronization

Learning approach	Application	Algorithm	Short name	Memory access pattern			Computation pattern		Communication/synchronization	
				Sequential	Strided	Random	Operations	Datatype	Intra PIM Core	Inter PIM Core
Supervised	Regression	**Linear Regression**	LIN	Yes	No	No	mul, add	float, int32_t	barrier	Yes
	Classification	**Logistic Regression**	LOG	Yes	No	No	mul, add, exp, div	float, int32_t	barrier	Yes
		Decision Tree	DTR	Yes	No	No	compare, add	float	barrier, mutex	Yes
Unsupervised	Clustering	**K-Means**	KME	Yes	No	No	mul, compare, add	int16_t, int64_t	barrier, mutex	Yes

Linear Regression

- Linear regression (`LIN`) is a supervised learning algorithm where the predicted output variable has a linear relation with the input variable
  - We use *gradient descent* as the optimization algorithm to find the minimum of the loss function
- Our PIM implementation divides the training dataset (*X*) equally among PIM cores
  - PIM threads compute dot products of row vectors and weights
  - Each dot product is compared to the observed value *y* to compute a partial gradient value
  - Partial gradient values are reduced and sent to the host
- Four versions of `LIN`:
  - `LIN-FP32`: training datasets of 32-bit real values
  - `LIN-INT32`: 32-bit fixed-point representation
  - `LIN-HYB`: hybrid precision (8-bit, 16-bit, 32-bit)
  - `LIN-BUI`: custom multiplication based on 8-bit built-in multiplication

Custom Integer Multiplication

Default integer multiplication

C code

```
1  result = X[i] * W[i];          // X and W are in WRAM (scratchpad)
```

UPMEM ISA

```
1  lbs r3, r2, 0                  // Load 1 byte from X[i]
2  lsl_add r2, r20, r1, 1         // Address of W[i]: r2=r20+(r1<<1)
3  lhs r4, r2, 0                  // Load 2 bytes from W[i]
4  mul_ul_ul r2, r4, r3, small, 0x80000378 // r2=r4(l)*r3(l)
5  mul_sh_ul r5, r4, r3           // r5=r4(h)*r3(l)
6  lsl_add r2, r2, r5, 8          // r2=r2+(r5<<8)
7  mul_sh_ul r5, r3, r4           // r5=r3(h)*r4(l)
8  lsl_add r2, r2, r5, 8          // r2=r2+(r5<<8)
9  mul_sh_sh r3, r4, r3           // r3=r4(h)*r3(h)
10 lsl_add r2, r2, r3, 16, true, 0x80000378 //r2=r2+(r3<<16)
```

Custom integer multiplication

C code

```
1  __builtin_mul_sl_ul_rrr(templ, X[i], W[i]);
2  __builtin_mul_sl_sh_rrr(temph, X[i], W[i]);
3  result = (temph << 8) + templ;
```

UPMEM ISA

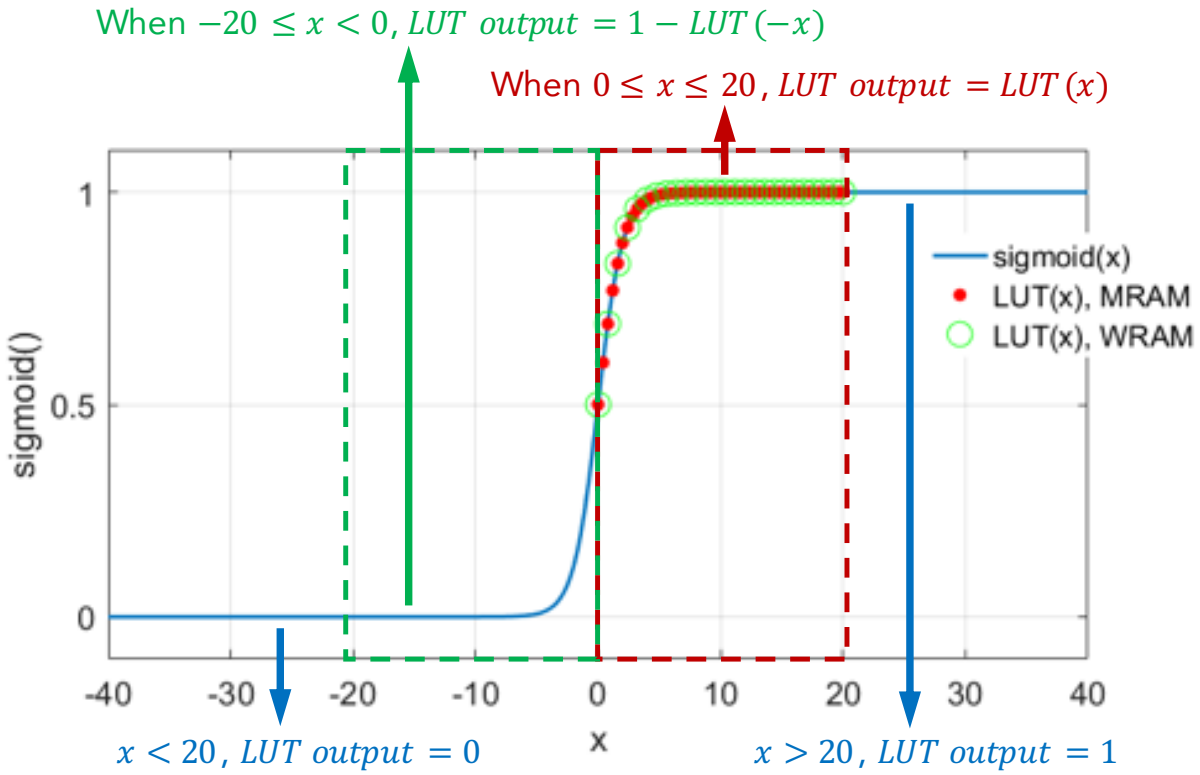
```
1  lbs r4, r4, 0                  // Load 1 byte from X[i]
2  lsl_add r5, r20, r3, 1         // Address of W[i]: r5=r20+(r1<<1)
3  lhs r5, r5, 0                  // Load 2 bytes from W[i]
4  mul_sl_ul r6, r4, r5           // r6=r4(l)*r5(l)
5  mul_sl_sh r4, r4, r5           // r4=r4(l)*r5(h)
6  add r2, r6, r2                 // r2=r2+r6
7  lsl_add r2, r2, r4, 8          // r2=r2+(r4 << 8)
```

Logistic Regression

- Logistic regression (LOG) is a supervised learning algorithm used for classification, which outputs probability values for each input observation variable or vector
  - *Sigmoid* function to map predicted values to probabilities
- Our PIM implementation follows the same workload distribution pattern as our linear regression implementation
- Six versions of LOG:
  - LOG-FP32: training datasets of 32-bit real values, Sigmoid approximated with Taylor series
  - LOG-INT32: 32-bit fixed-point representation, Taylor series
  - LOG-INT32-LUT: Sigmoid calculation with a lookup table (LUT)
    - LOG-INT32-LUT(MRAM): LUT in MRAM
    - LOG-INT32-LUT(WRAM): LUT in WRAM
  - LOG-HYB-LUT: hybrid precision (8-bit, 16-bit, 32-bit), LUT in WRAM
  - LOG-BUI-LUT: custom multiplication based on 8-bit built-in multiplication, LUT in WRAM

LUT-based Sigmoid Calculation

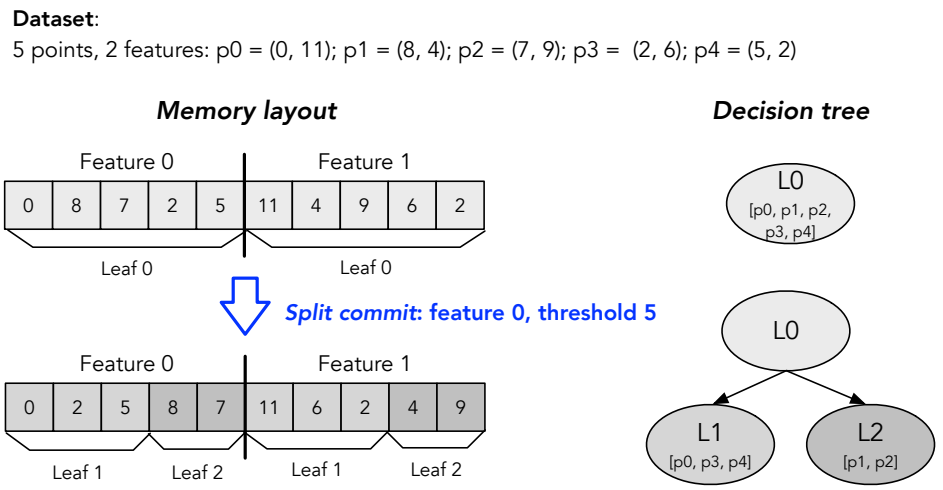
- We take advantage of the fact that Sigmoid is symmetric
- The LUT size depends on the boundary (e.g., 20) and the number of bits for the decimal part of the fixed-point representation (e.g., 10)
  - 20 x 1024 entries (with 16-bit entries) = 40 KB



Decision Tree

- Decision trees (DTR) are tree-based methods used for classification and regression, which partition the feature space into *leaves*, with a simple prediction model in each leaf
- Our PIM implementation partitions the training set among PIM cores, which compute partial *Gini* scores to evaluate the host's *split* decisions
- The host sends commands to the PIM cores:
  - *Split commit* to split a tree leaf
  - *Split evaluate* to evaluate a split
  - *Min-max* to query minimum/maximum values of a feature in a tree leaf

- Data layout in split commit to maximize memory bandwidth with streaming accesses
- This data layout also ensures memory accesses in streaming in split evaluate



K-Means Clustering

- K-means (`KME`) is an iterative clustering method used to find groups in a dataset which have not been explicitly labeled
- Our PIM implementation distributes the dataset evenly over the PIM cores
- PIM threads evaluate which centroid is the closest one to each point of the training set
  - Counter and accumulator per coordinate (per centroid)
- Then, the host recalculates the centroids
- Convergence to a local optimum when the updated centroid's coordinates are within a threshold (*Frobenius norm*)

Outline

Machine learning workloads

Processing-in-memory

PIM implementation of ML workloads

Evaluation

- Quality Metrics
- Analysis of PIM Kernels
- Performance Scaling
- Comparison to CPU and GPU

Evaluation Methodology

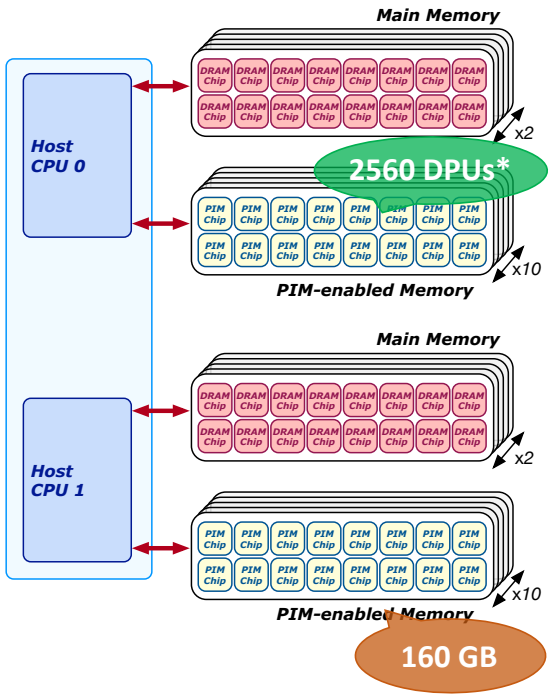
- Synthetic and real datasets

ML Workload	Synthetic Datasets†		Real Datasets
	Strong Scaling (1 PIM core | 256-2048 PIM cores)	Weak Scaling (per PIM core)	
Linear regression	2,048 samples, 16 attr. (0.125 MB) | 6,291,456 samples, 16 attr. (384 MB)	2,048 samples, 16 attr. (0.125 MB)	SUSY [232, 233]
Logistic regression	2,048 samples, 16 attr. (0.125 MB) | 6,291,456 samples, 16 attr. (384 MB)	2,048 samples, 16 attr. (0.125 MB)	Skin segmentation [234]
Decision tree	60,000 samples, 16 attr. (3.84 MB) | 153,600,000 samples, 16 attr. (9830 MB)	600,000 samples, 16 attr. (38.4 MB)	Higgs boson [232, 235] | Criteo [236]
K-Means	10,000 samples, 16 attr. (0.64 MB) | 25,600,000 samples, 16 attr. (1640 MB)	100,000 samples, 16 attr. (6.4 MB)	Higgs boson [232, 235] | Criteo [236]

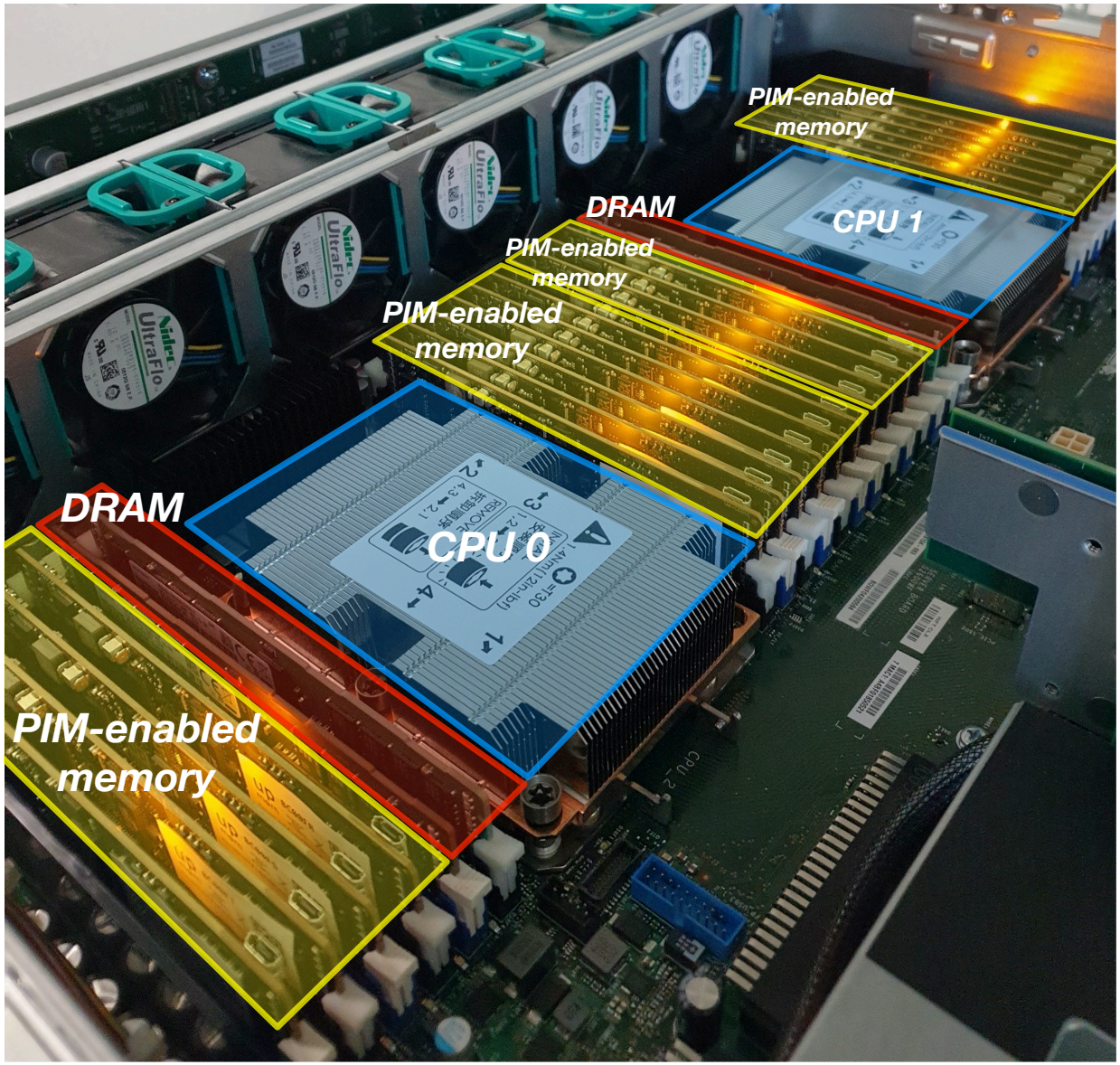
† Format = Samples (dataset elements), Attributes (Size in MB).

- Evaluated systems
  - UPMEM PIM system with 2,524 PIM cores @ 425 MHz and 158 GB of DRAM
  - Intel Xeon Silver 4215 CPU
  - NVIDIA A100 GPU
- We evaluate:
  - Quality metrics
  - Performance of PIM kernels
  - Performance scaling
  - Comparison to CPU and GPU

2,560-DPU UPMEM PIM System



- 20 UPMEM DIMMs of 16 chips each (40 ranks)
- Dual x86 socket
- UPMEM DIMMs coexist with regular DDR4 DIMMs
  - 2 memory controllers/socket (3 channels each)
  - 2 conventional DDR4 DIMMs on one channel of one controller



* There are some faulty DPUs in the system that we use in our experiments. Thus, the maximum number of DPUs we can use is 2,524

Outline

Machine learning workloads

Processing-in-memory

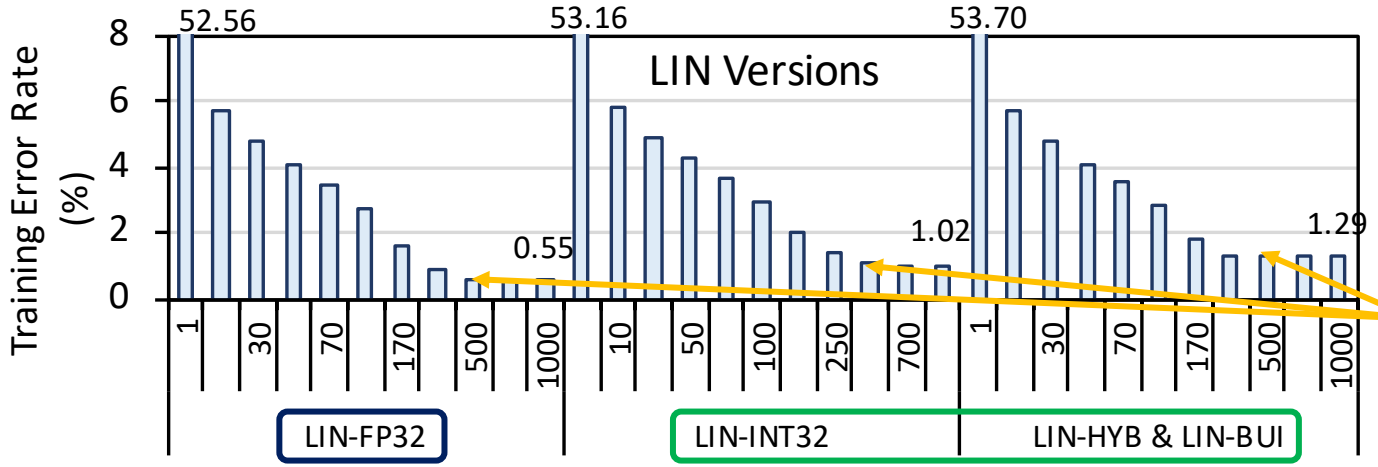
PIM implementation of ML workloads

Evaluation

- Quality Metrics
- Analysis of PIM Kernels
- Performance Scaling
- Comparison to CPU and GPU

Evaluation: Quality Metrics: LIN

- Linear regression



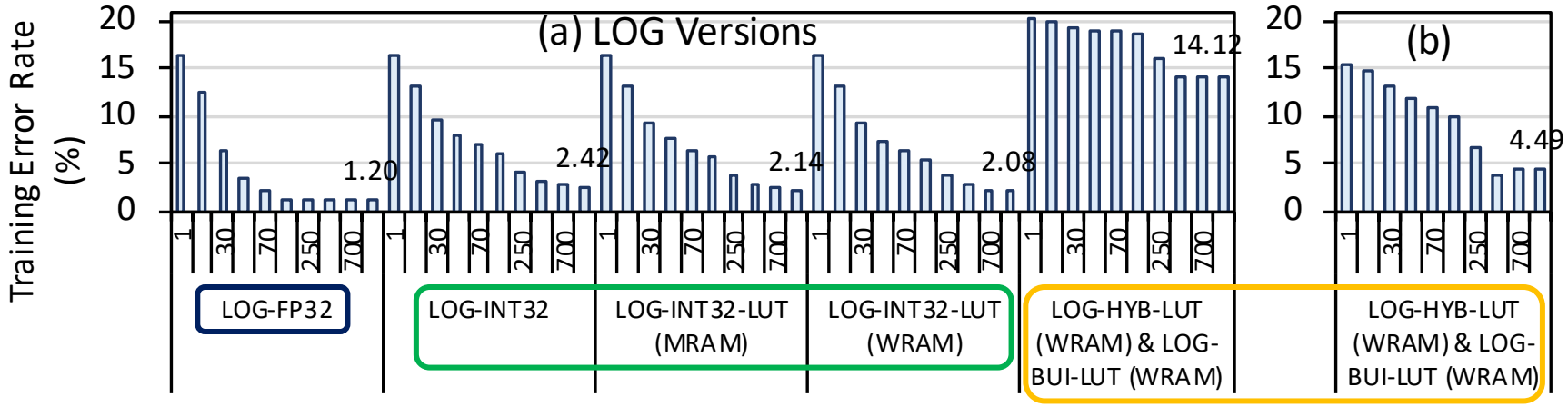
The training error rate flattens after 500 iterations

Training error rate of LIN-FP32 is the same as the CPU version

For the integer versions, the training error rate remains low and close to that of LIN-FP32

Evaluation: Quality Metrics: LOG

- Logistic regression



Training error rate of `LOG-FP32` is the same as the CPU version

LUT-based versions obtain lower training error rates than `LOG-INT32`, since they use exact values, not approximations

Reduced-precision datatypes increase the training error rate, which heavily depends on the number of decimal numbers of the samples (e.g., 4 in (a), 2 in (b))

Evaluation: Quality Metrics

- Linear regression
  - Training error rate of `LIN-FP32` is the same as the CPU version
  - For integer versions, it remains low and close to that of `LIN-FP32`
- Logistic regression
  - LUT-based versions obtain lower training error rates that `LOG-INT32`, since they use exact values, not approximations
- Decision tree
  - Training accuracy only slightly lower than that of the CPU version
- K-means clustering
  - Same *Calinski-Harabasz score* and *adjusted Rand index* of PIM and CPU versions

We maintain the accuracy of all workloads
(or keep it close to the CPU baseline)

Outline

Machine learning workloads

Processing-in-memory

PIM implementation of ML workloads

Evaluation

- Quality Metrics
- Analysis of PIM Kernels
- Performance Scaling
- Comparison to CPU and GPU

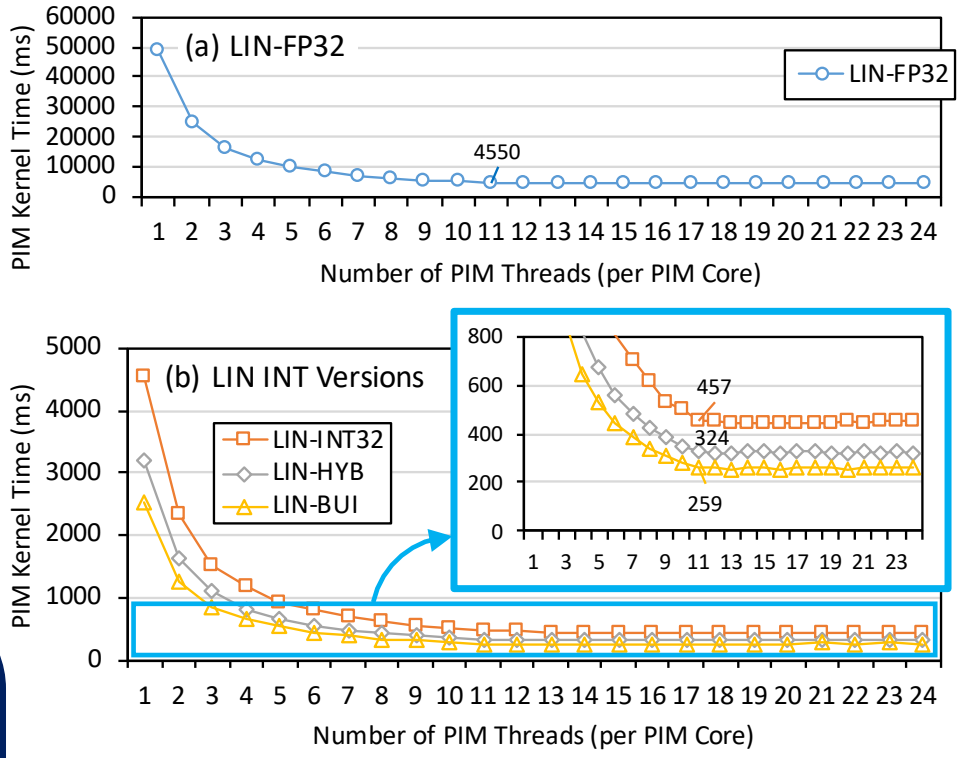
Evaluation: Analysis of PIM Kernels (I)

- Linear regression

All versions saturate at 11 or more PIM threads

Fixed-point representation accelerates the kernel by an order of magnitude over FP32

Key Takeaway 1. Workloads with **arithmetic operations or datatypes not natively supported** by PIM cores run at low performance due to instruction emulation (e.g., FP in UPMEM PIM).



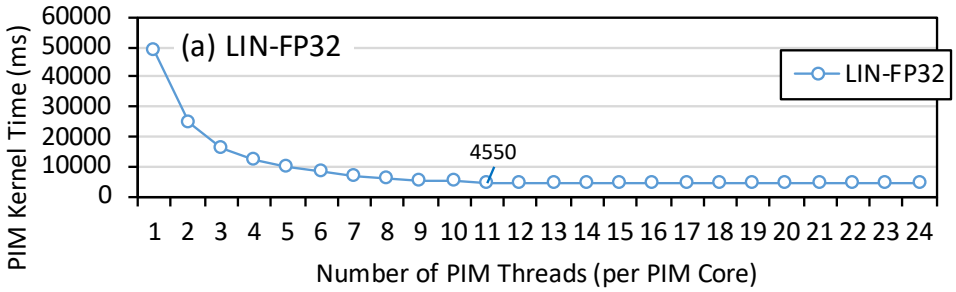
Recommendation 1. Use **fixed-point representation,** without much accuracy loss, if PIM cores do not support FP.

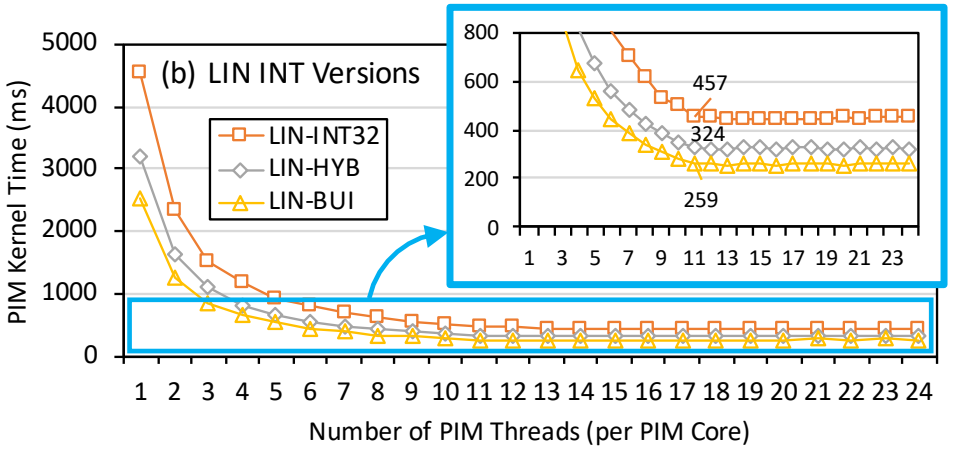
Evaluation: Analysis of PIM Kernels (II)

- Linear regression

`LIN-HYB` is 41% faster than `LIN-INT32`

`LIN-BUI` provides an additional 25% speedup





Recommendation 2. **Quantization** can take advantage of native hardware support. **Hybrid precision** can significantly improve performance.

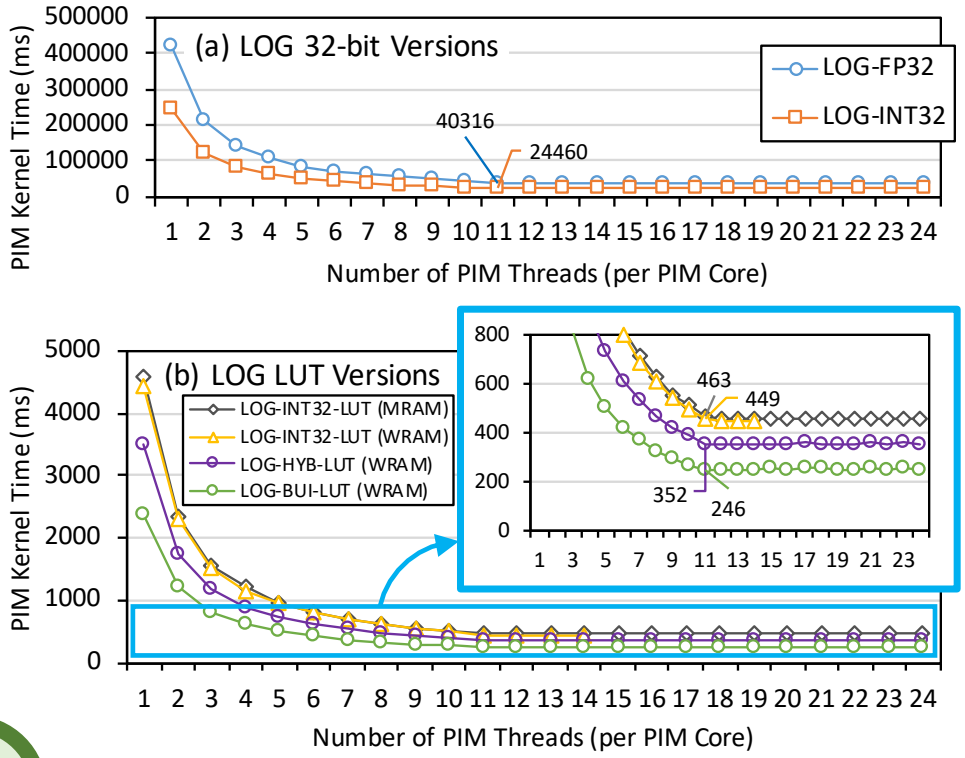
Recommendation 3. Programmers/better compilers can **optimize code by leveraging native instructions** (e.g., 8-bit integer multiplication in UPMEM) .

Evaluation: Analysis of PIM Kernels (III)

- Logistic regression

Very high kernel time of LOG-FP32 and LOG-INT32 due to Sigmoid approximation

LOG-INT32-LUT(MRAM) is 53x faster than LOG-INT32



Recommendation 4. Convert computation to memory accesses by **keeping pre-calculated operation results** (e.g., LUTs, memoization) **in memory**.

LOG-HYB-LUT is 28% faster than LOG-INT32-LUT

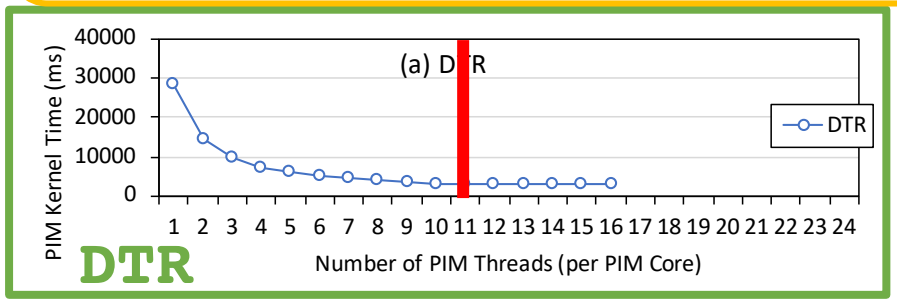
LOG-BUI-LUT provides an additional 43% speedup

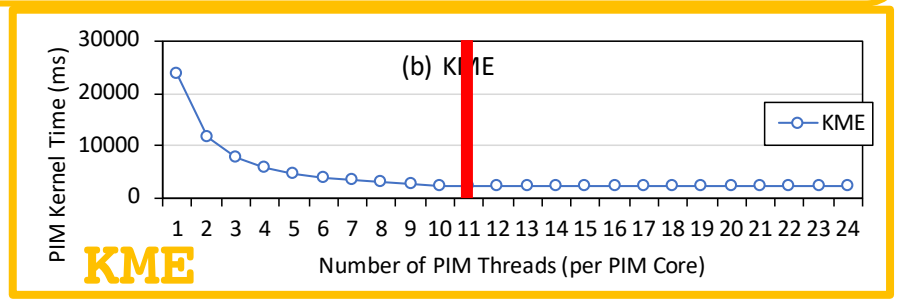
Evaluation: Analysis of PIM Kernels (IV)

- Linear regression, logistic regression, decision tree, K-means clustering

The performance of all kernels saturates at 11 or more PIM threads. In the UPMEM PIM architecture, this means that the pipeline latency hides the memory latency

As a result, these kernels are compute-bound on the UPMEM PIM architecture



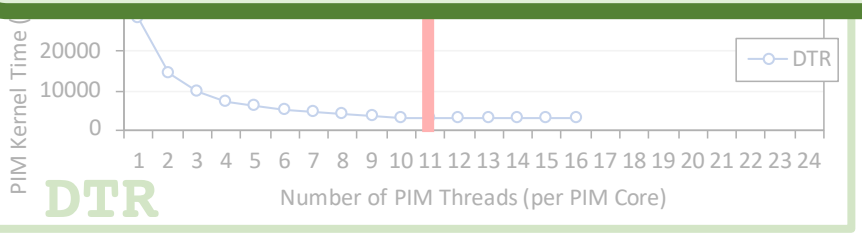


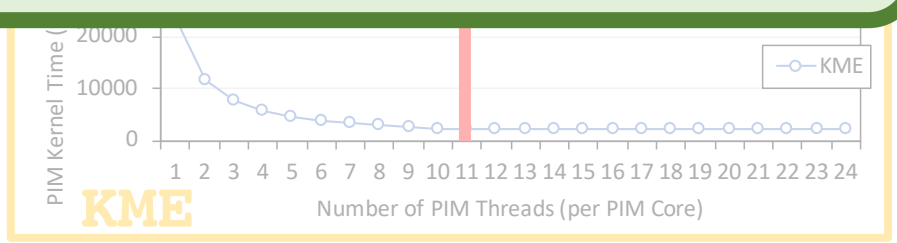
Evaluation: Analysis of PIM Kernels (V)

- Linear regression, logistic regression, decision tree, K-means clustering

Key Takeaway 2. ML workloads that are memory-bound due to low arithmetic intensity in CPU/GPU **become compute-bound when running on PIM.**

Recommendation 6. Maximize the utilization of PIM cores by **keeping their pipeline fully busy.**





Outline

Machine learning workloads

Processing-in-memory

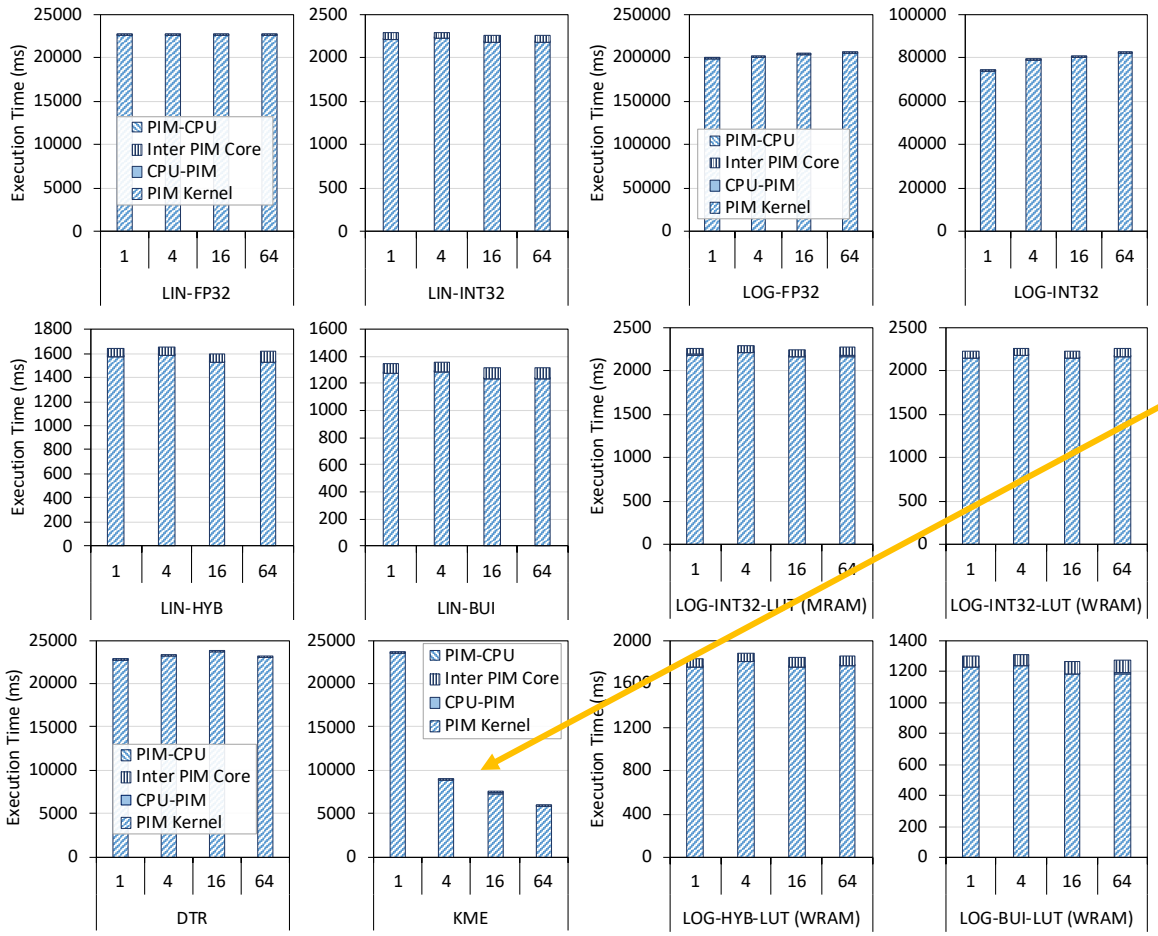
PIM implementation of ML workloads

Evaluation

- Quality Metrics
- Analysis of PIM Kernels
- Performance Scaling
- Comparison to CPU and GPU

Evaluation: Performance Scaling (I)

- Weak scaling: 1 to 64 PIM cores



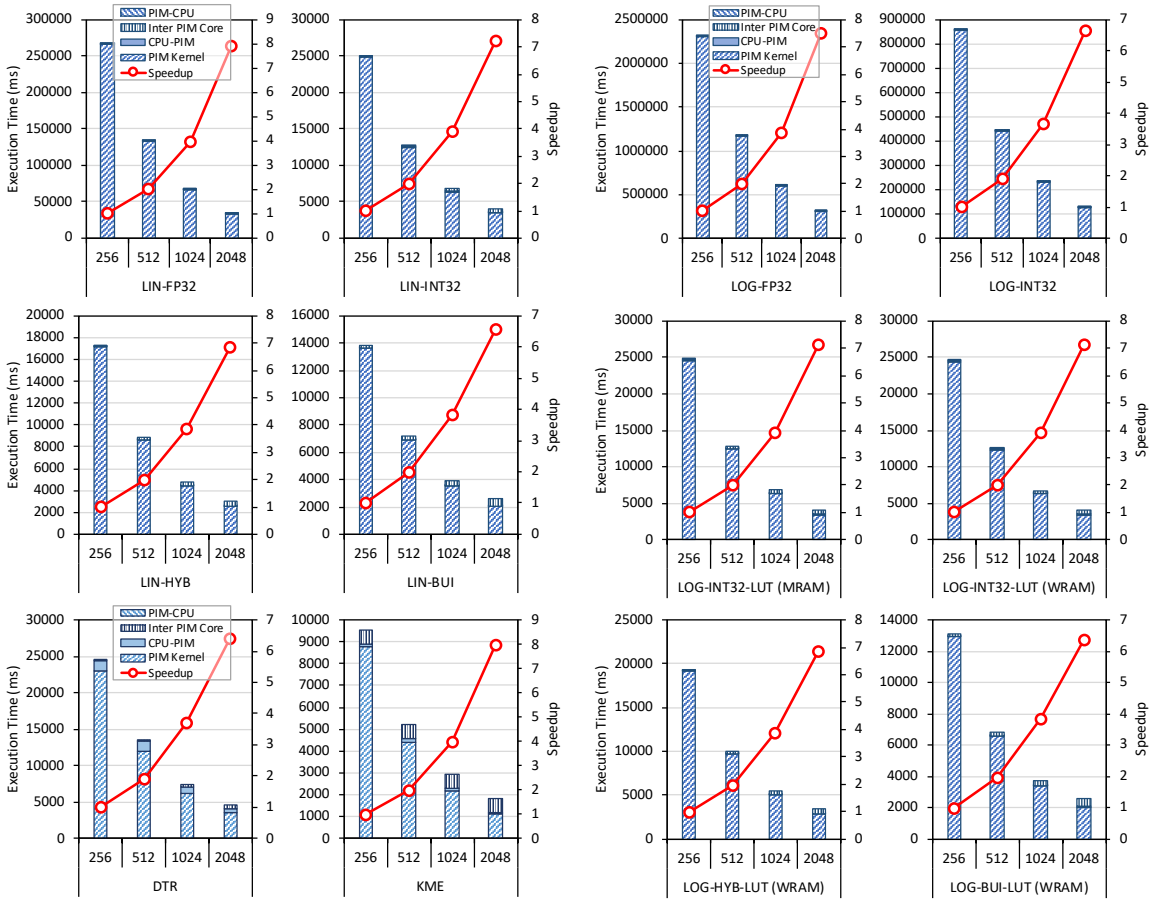
PIM kernel time of `LIN`, `LOG`, and `DTR` scales linearly with the number of PIM cores

`KME` converges with fewer iterations on a larger dataset

The sum of CPU-PIM, Inter PIM core, and PIM-CPU takes less than 7% of the total execution time in all cases

Evaluation: Performance Scaling (II)

- Strong scaling: 256 to 2,048 PIM cores

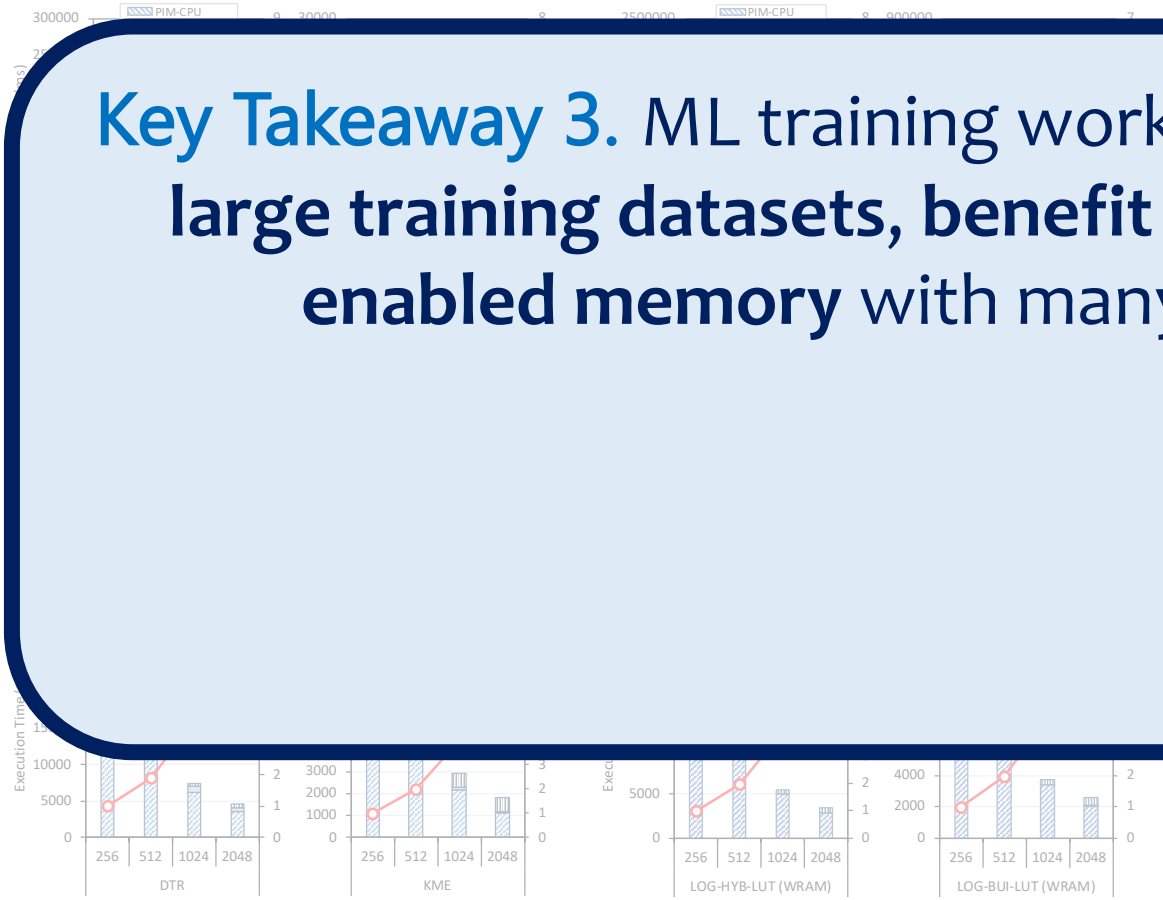


PIM kernel time scales linearly with the number of PIM cores

Little overhead from inter PIM core communication and communication between host and PIM cores

Evaluation: Performance Scaling (II)

- Strong scaling: 256 to 2,048 PIM cores



Key Takeaway 3. ML training workloads, which need **large training datasets, benefit from large PIM-enabled memory** with many PIM cores.

between host and PIM cores

Outline

Machine learning workloads

Processing-in-memory

PIM implementation of ML workloads

Evaluation

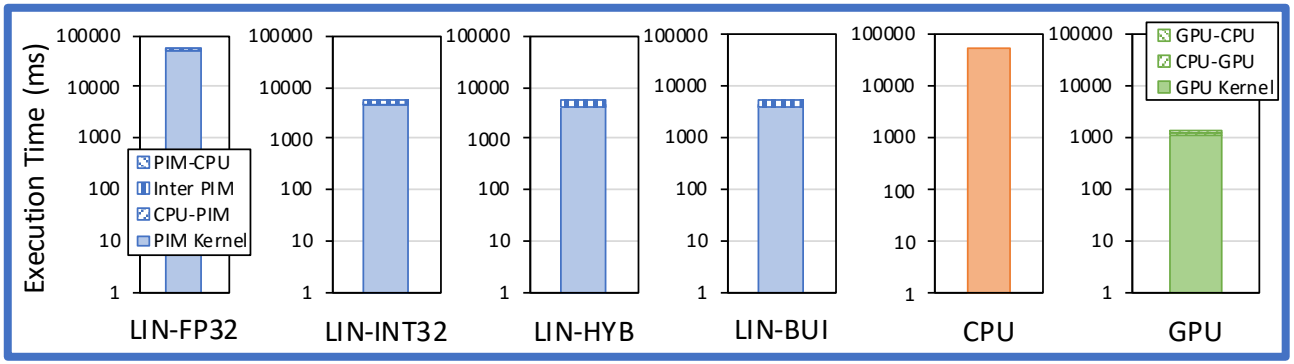
- Quality Metrics
- Analysis of PIM Kernels
- Performance Scaling
- Comparison to CPU and GPU

Comparison to CPU and GPU (I)

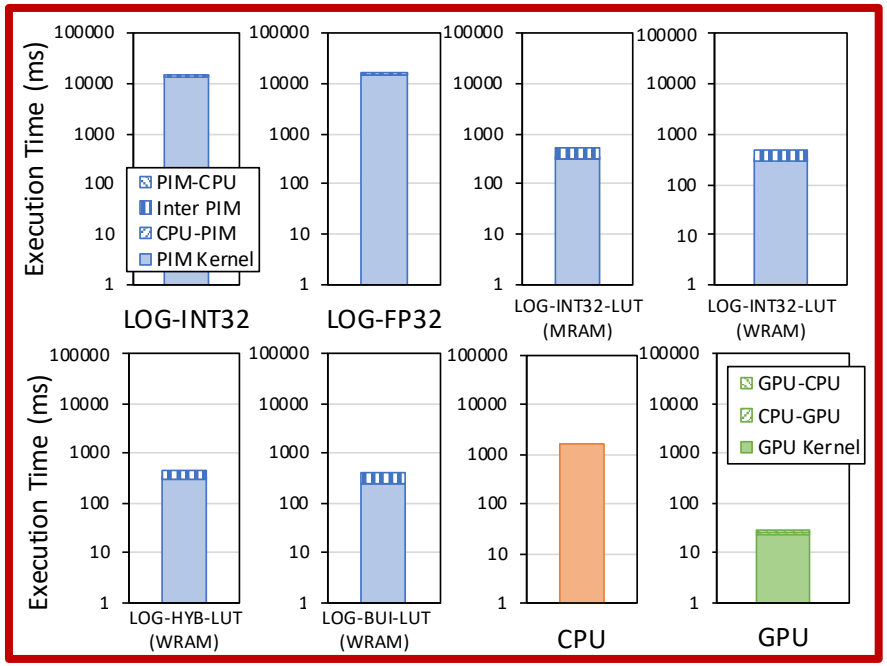
- Linear regression and logistic regression

PIM versions are heavily burdened when they use operations that are not natively supported by the hardware

LIN



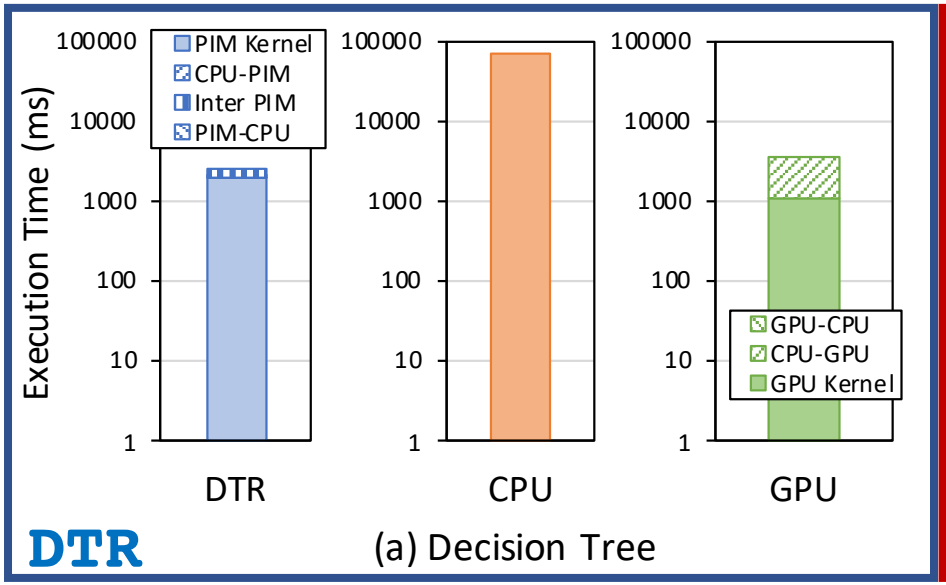
LOG

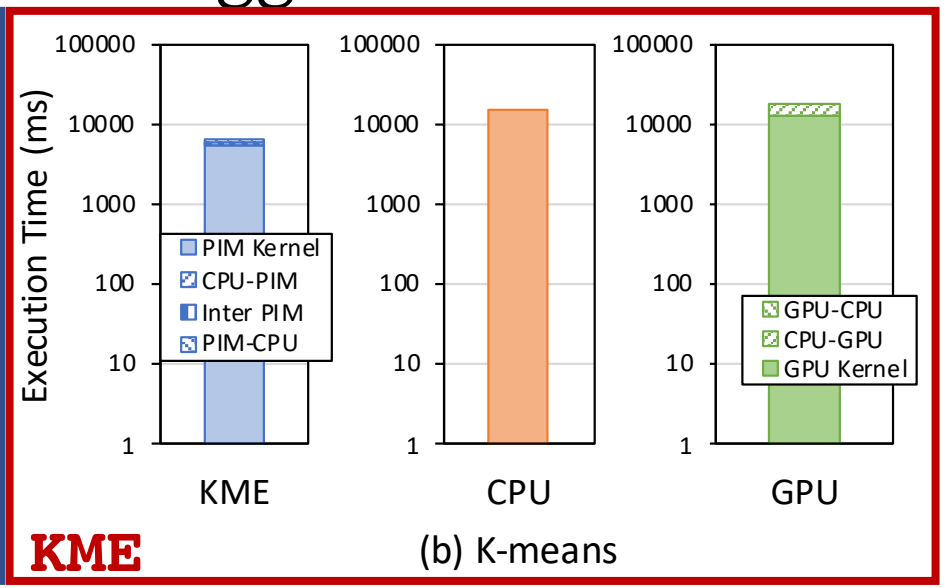


Several optimizations reduce the execution time considerably (LIN/LOG up to 10x/3.9x faster than CPU) and close the gap with GPU performance (LIN/LOG still 4x/16x slower than GPU)

Comparison to CPU and GPU (II)

- Decision tree and K-means with Higgs boson dataset





PIM version of DTR is **27x** faster than the CPU version and **1.34x** faster than the GPU version

PIM version of KME is **2.8x** faster than the CPU version and **3.2x** faster than the GPU version

Long arXiv Version

- Additional implementation details
- More evaluation results
- Extended observations, takeaways, and recommendations

An Experimental Evaluation of Machine Learning Training on a Real Processing-in-Memory System

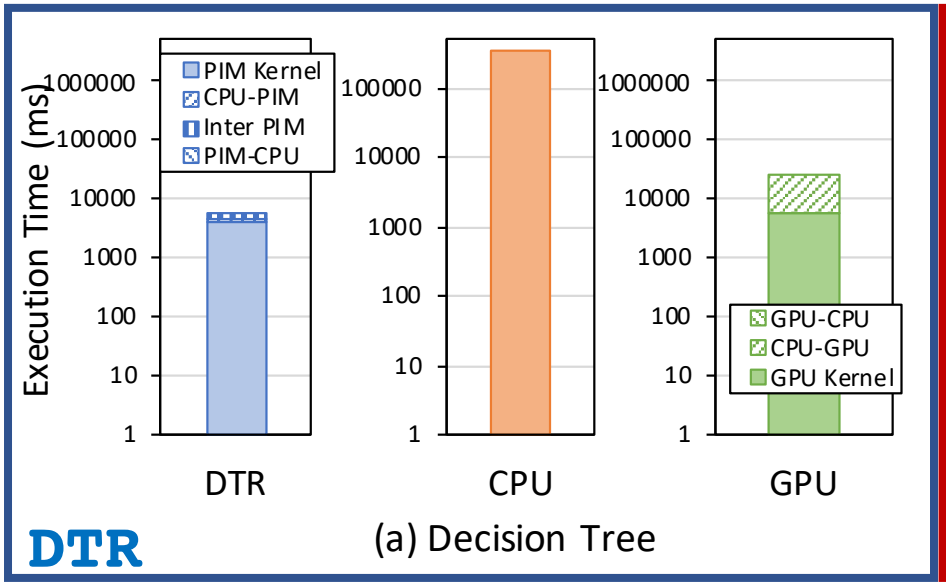
Juan Gómez-Luna[1] Yuxin Guo[1] Sylvan Brocard[2] Julien Legriel[2]
Remy Cimadomo[2] Geraldo F. Oliveira[1] Gagandeep Singh[1] Onur Mutlu[1]
[1]ETH Zürich [2]UPMEM

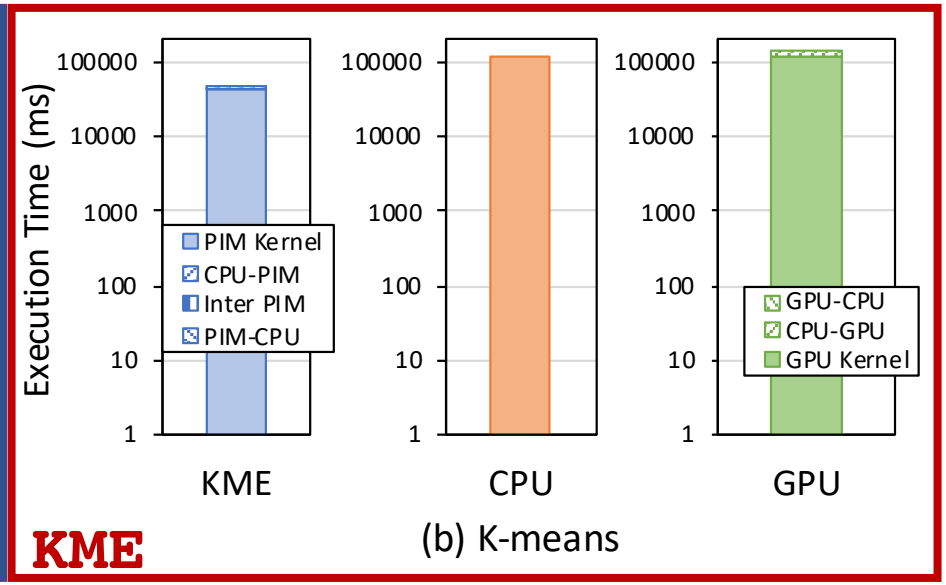
https://arxiv.org/pdf/2207.07886.pdf

Source code: https://github.com/CMU-SAFARI/pim-ml

Comparison to CPU and GPU (III)

- Decision tree and K-means with Criteo 1TB dataset



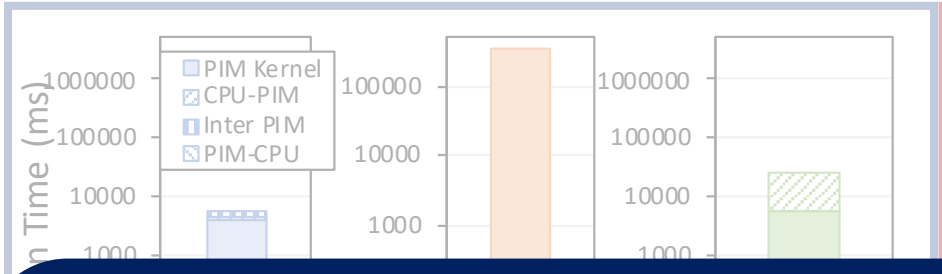


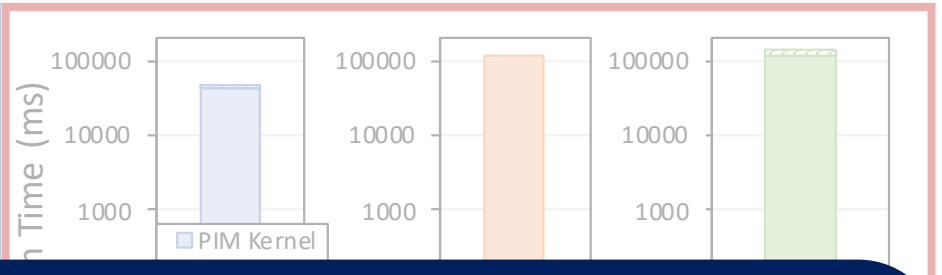
PIM version of DTR is **62x** faster than the CPU version and **4.5x** faster than the GPU version

PIM version of KME is **2.7x** faster than the CPU version and **3.2x** faster than the GPU version

Comparison to CPU and GPU (IV)

- Decision tree and K-means with Criteo 1TB dataset





Key Takeaway 4. **ML workloads that require mainly operations natively supported** by the PIM architecture, such as decision tree and K-means clustering, **outperform their CPU and GPU counterparts.**

faster than the CPU version and **4.5x** faster than the GPU version

faster than the CPU version and **3.2x** faster than the GPU version

Long arXiv Version

- Additional implementation details
- More evaluation results
- Extended observations, takeaways, and recommendations

An Experimental Evaluation of Machine Learning Training on a Real Processing-in-Memory System

Juan Gómez-Luna[1] Yuxin Guo[1] Sylvan Brocard[2] Julien Legriel[2]
Remy Cimadomo[2] Geraldo F. Oliveira[1] Gagandeep Singh[1] Onur Mutlu[1]
[1]ETH Zürich [2]UPMEM

https://arxiv.org/pdf/2207.07886.pdf

Source code: https://github.com/CMU-SAFARI/pim-ml

Short arXiv Version

- Presented at ISVLSI 2022

Machine Learning Training on a Real Processing-in-Memory System

Juan Gómez-Luna[1] Yuxin Guo[1] Sylvan Brocard[2] Julien Legriel[2]
Remy Cimadomo[2] Geraldo F. Oliveira[1] Gagandeep Singh[1] Onur Mutlu[1]

[1]ETH Zürich [2]UPMEM

https://arxiv.org/pdf/2206.06022.pdf

Source code: https://github.com/CMU-SAFARI/pim-ml

https://youtu.be/CVX8n-X-5wI

ISPASS 2023 Version

- Presented at ISPASS 2023

Evaluating Machine Learning Workloads on Memory-Centric Computing Systems

Juan Gómez-Luna[1] Yuxin Guo[1] Sylvan Brocard[2] Julien Legriel[2]
Remy Cimadomo[2] Geraldo F. Oliveira[1] Gagandeep Singh[1] Onur Mutlu[1]
[1]ETH Zürich [2]UPMEM

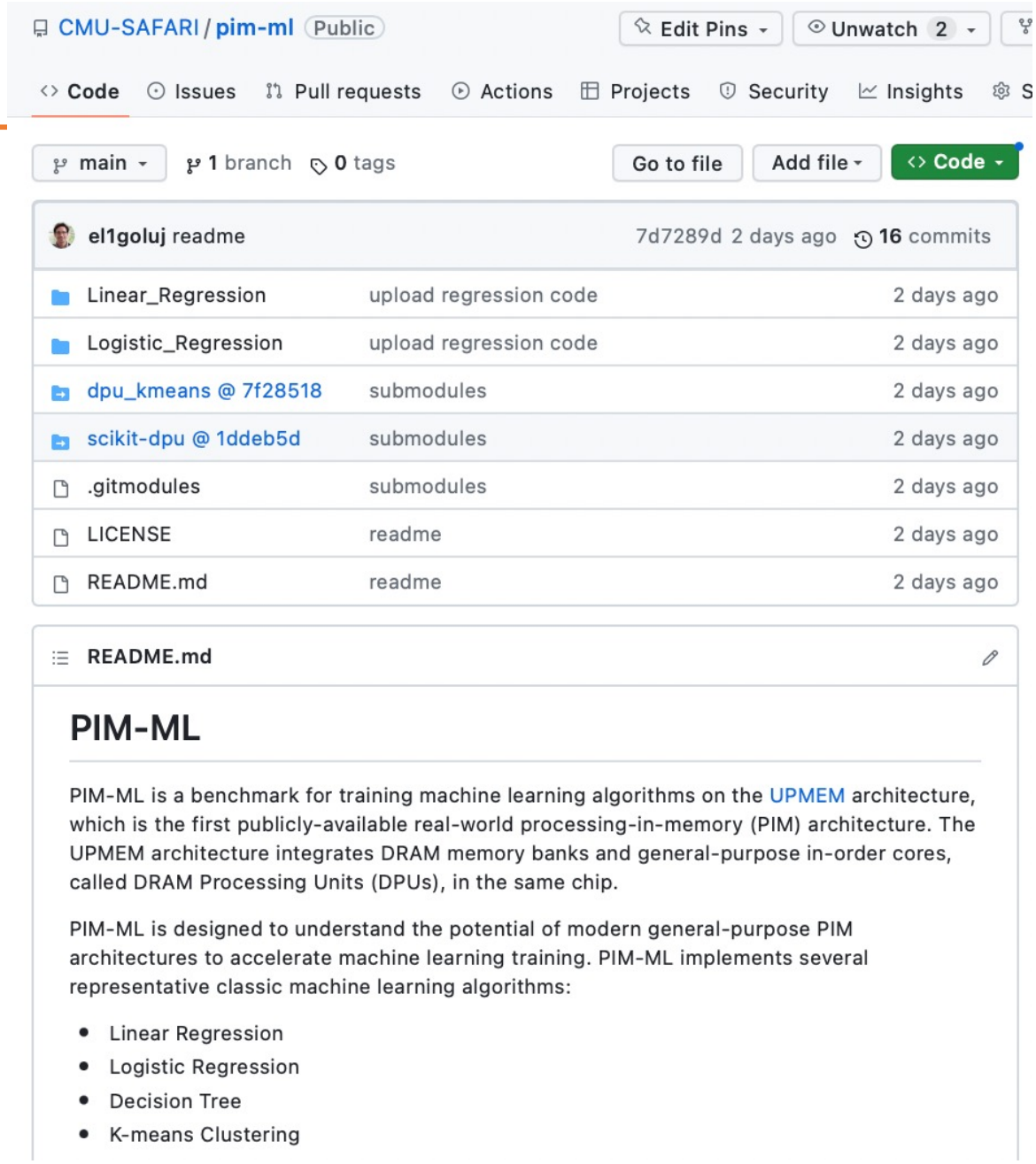
https://people.inf.ethz.ch/omutlu/pub/MLonUPMEM-PIM_ispass23.pdf

Source code: https://github.com/CMU-SAFARI/pim-ml

https://youtu.be/60pkaI5AeM4

Source Code

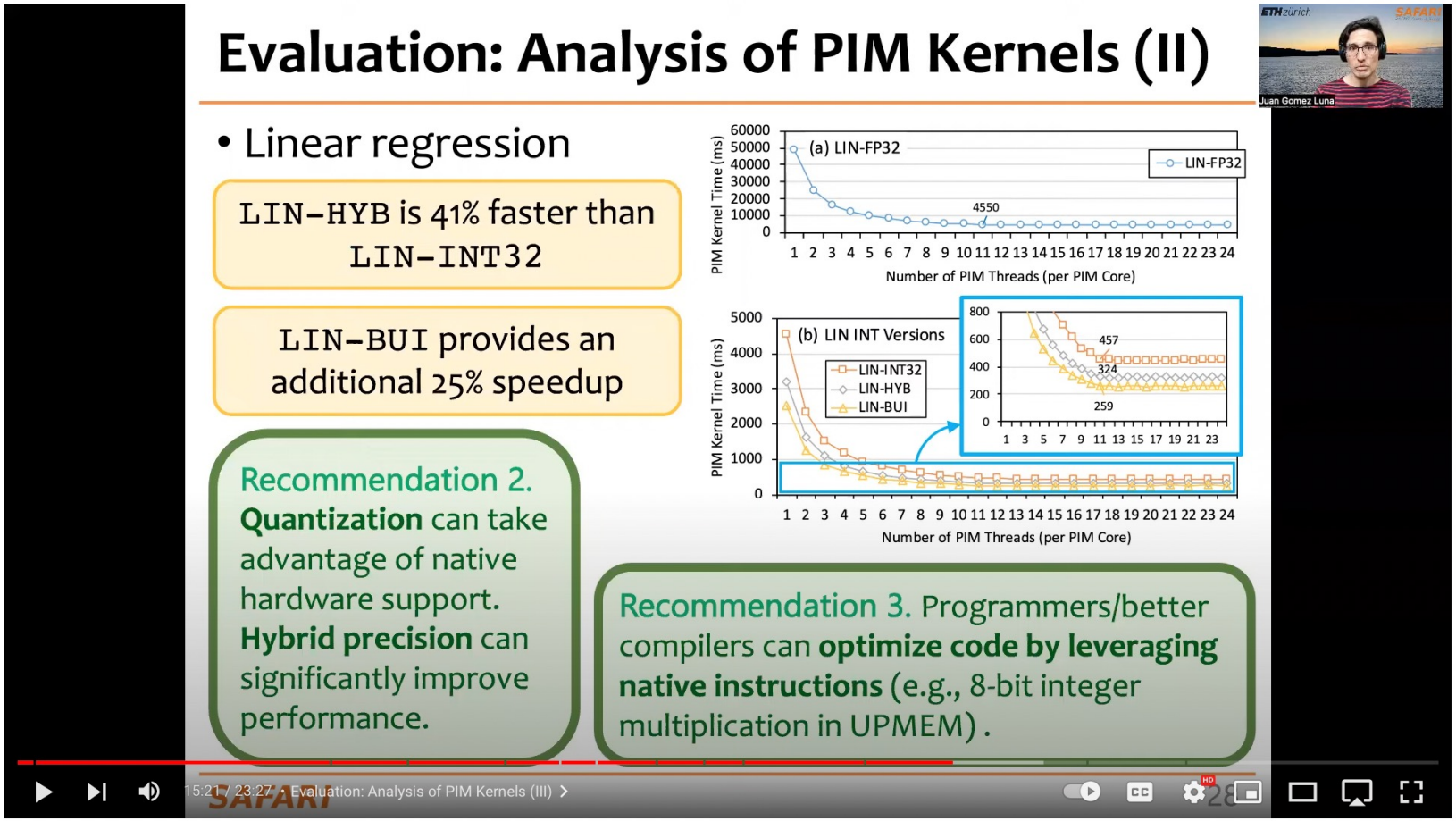
- https://github.com/CMU-SAFARI/pim-ml



Executive Summary

- Training machine learning (ML) algorithms is a computationally expensive process, frequently memory-bound due to repeatedly accessing large training datasets
- Memory-centric computing systems, i.e., with Processing-in-Memory (PIM) capabilities, can alleviate this *data movement bottleneck*
- Real-world PIM systems have only recently been manufactured and commercialized
  - UPMEM has designed and fabricated the first publicly-available real-world PIM architecture
- Our goal is to understand the potential of modern general-purpose PIM architectures to accelerate machine learning training
- Our main contributions:
  - PIM implementation of several classic machine learning algorithms: linear regression, logistic regression, decision tree, K-means clustering
  - Workload characterization in terms of quality, performance, and scaling
  - Comparison to their counterpart implementations on processor-centric systems (CPU and GPU)
    - PIM version of `DTR` is **27x** / **1.34x** faster than the CPU / GPU version, respectively
    - PIM version of `KME` is **2.8x** / **3.2x** faster than the CPU / GPU version, respectively
  - Source code: https://github.com/CMU-SAFARI/pim-ml
- Experimental evaluation on a real-world PIM system with 2,524 PIM cores @ 425 MHz and 158 GB of DRAM memory
- Key observations, takeaways, and recommendations for ML workloads on general-purpose PIM systems

Lecture on PIM-ML



PIM Course: Lecture 12: ML Training on a Real PIM Architecture (Spring 2023)

Onur Mutlu Lectures
33.4K subscribers
Subscribed
9
Share
Clip
Save

234 views 2 weeks ago Livestream - Data-Centric Architectures: Fundamentally Improving Performance and Energy (Spring 2023)
Projects & Seminars, ETH Zürich, Spring 2023
Data-Centric Architectures: Fundamentally Improving Performance and Energy

An Experimental Evaluation of Machine Learning Training on a Real Processing-in-Memory System

Juan Gómez Luna, Yuxin Guo, Sylvan Brocard,
Julien Legriel, Remy Cimadomo, Geraldo F. Oliveira,
Gagandeep Singh, Onur Mutlu

https://arxiv.org/pdf/2207.07886.pdf
https://github.com/CMU-SAFARI/pim-ml
juang@ethz.ch

ETH Zürich SAFARI upmem

Thursday, May 25, 2023

Genomics Sequence Alignment

High Throughput Sequence Alignment Using Real Processing-in-Memory Systems

Safaa Diab, Amir Nassereldine, Mohammed Alser,
Juan Gómez Luna, Onur Mutlu, Izzat El Hajj

https://arxiv.org/pdf/2208.01243.pdf
https://github.com/CMU-SAFARI/alignment-in-memory
juang@ethz.ch

ETH Zürich

SAFARI

Thursday, June 8, 2023

Sequence Alignment on Processing-in-Memory

- Published in Bioinformatics (2023)

A Framework for High-throughput Sequence Alignment using Real Processing-in-Memory Systems

Safaa Diab [1] Amir Nassereldine [1] Mohammed Alser [2] Juan Gómez Luna [2]
Onur Mutlu [2] Izzat El Hajj [1]

[1]American University of Beirut [2]ETH Zürich

https://arxiv.org/pdf/2204.02085.pdf
https://arxiv.org/pdf/2208.01243.pdf
Source code: https://github.com/safaad/aim

Executive Summary

- Genome sequence alignment is typically memory-bound on conventional processor-centric systems (CPU, GPU)
- Processing-in-memory (PIM) has the potential to overcome this data movement bottleneck by placing compute capabilities near/inside memory arrays
- We present Alignment-in-Memory (AiM), a PIM framework for genome sequence alignment on general-purpose PIM systems
  - AiM supports multiple alignment algorithms: NW, SWG, GenASM, WFA
  - Implemented for the UPMEM PIM architecture
- Our evaluation shows that sequence alignment on PIM can be significantly faster than on CPUs and GPUs

Outline

Background & Motivation:
Sequence Alignment on CPU

Processing-in-memory to Alleviate the Data Movement Bottleneck

Alignment-in-memory: A Framework for Sequence Alignment on PIM Systems

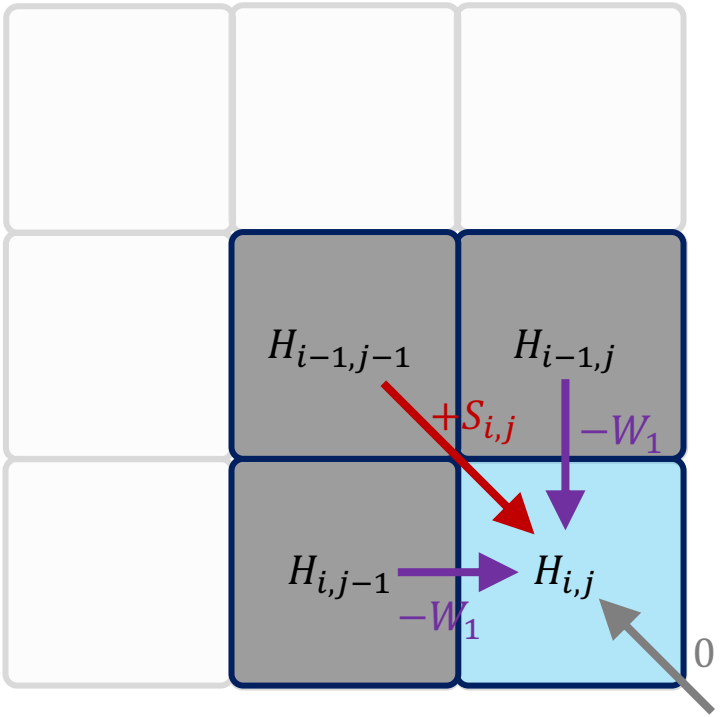
Evaluation

Genome Analysis

- A genome is the complete set of an organism's genetic instructions
- DNA is a string of base pairs (characters): A, C, G, T
  - The human genome contains 3.2 billion base pairs
  - Genome analysis helps to understand genetic variations, predict the presence and abundance of microbes, monitor disease outbreaks, develop personalized medicine, etc.
- Sequencing machines provide only randomized fragments (reads), which need to be mapped to the reference genome
  - Depending on the sequencing technology, there are short (50-300 bp) and long reads (10K-100K bp)
- A key step in the read mapping process is sequence alignment
  - Quadratic-time dynamic programming algorithms, e.g., Smith-Waterman, Needleman-Wunsch
  - State-of-the-art algorithms: GenASM, WFA, WFA-adaptive

Sequence Alignment: Smith-Waterman (I)

- Smith-Waterman performs local sequence alignment
  - We compare two sequences A and B
  - For any two elements of A and B, we can obtain a similarity score
  - $S_{i,j} = \begin{cases} +3, A_i = B_j \\ -3, A_i \neq B_j \end{cases}$
- Gap penalty (linear)
  - $W_1 = 2$
  - $W_k = kW_1$
- Scoring matrix
  - $H_{ij} = max \begin{cases} H_{i-1,j-1} + S_{i,j} \\ H_{i-1,j} - W_1 \\ H_{i\ j-1} - W_1 \\ 0 \end{cases}$



Sequence Alignment: Smith-Waterman (II)

- Let's align two sequences: AACC and AATC
- First, we compute the scoring matrix

		A	A	T	C
	0	0	0	0	0
A	0	3	3	1	0
A	0	3	6	4	2
C	0	1	4	3	7
C	0	0	2	1	6

$$H_{1,1} = \max(H_{0,0} + S_{1,1}, H_{0,1} - W_1, H_{1,0} - W_1, 0)$$
$$= \max(0 + 3, 0 - 2, 0 - 2, 0) = 3$$

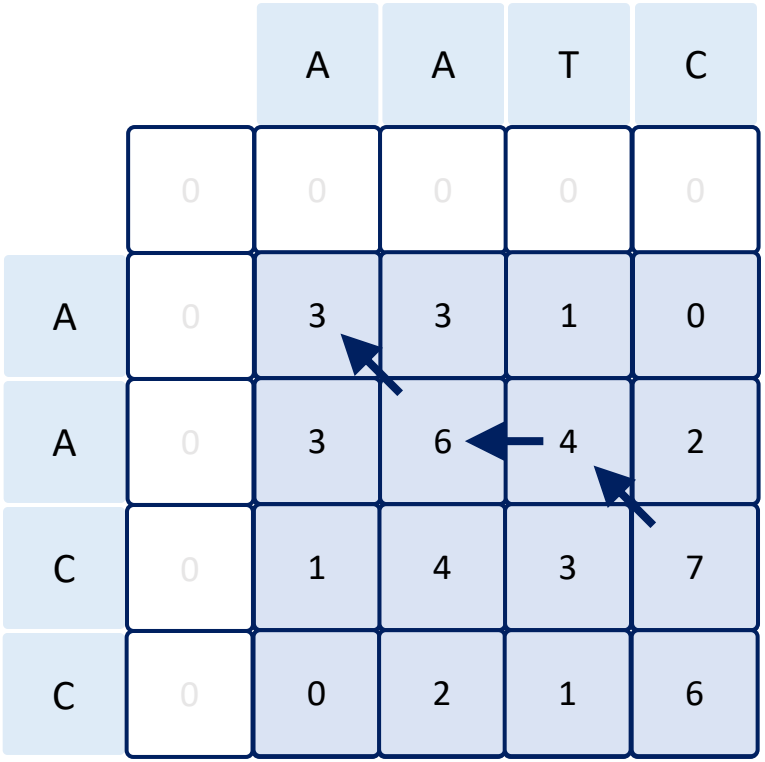
$$H_{2,1} = \max(H_{1,0} + S_{2,1}, H_{1,1} - W_1, H_{2,0} - W_1, 0)$$
$$= \max(0 + 3, 3 - 2, 0 - 2, 0) = 3$$

$$H_{1,2} = \max(H_{0,1} + S_{1,2}, H_{0,2} - W_1, H_{1,1} - W_1, 0)$$
$$= \max(0 + 3, 0 - 2, 3 - 2, 0) = 3$$

To calculate one antidiagonal, we need the two previous antidiagonals

Sequence Alignment: Smith-Waterman (III)

- Let's align two sequences: AACC and AATC
- Second, traceback and alignment



A A T C
| | | |
A A - C

For the traceback process, we need the direction of the score calculation

Sequence Alignment Algorithms

Needleman-Wunsch (NW)

D		A	T	A
	0	4	8	12
A	4	0	4	8
T	8	4	0	4
C	12	8	4	2
A	16	12	8	4

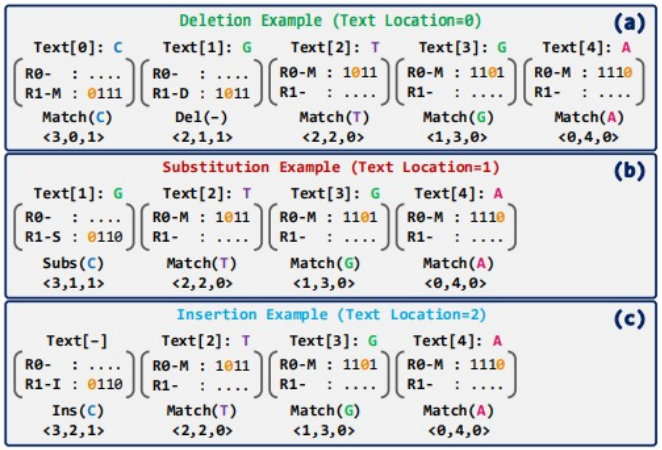
Smith-Waterman-Gotoh (SWG)

M		A	T	A
	0	5	6	7
A	5	0	5	6
T	6	5	0	5
C	7	6	5	2
A	8	7	6	5

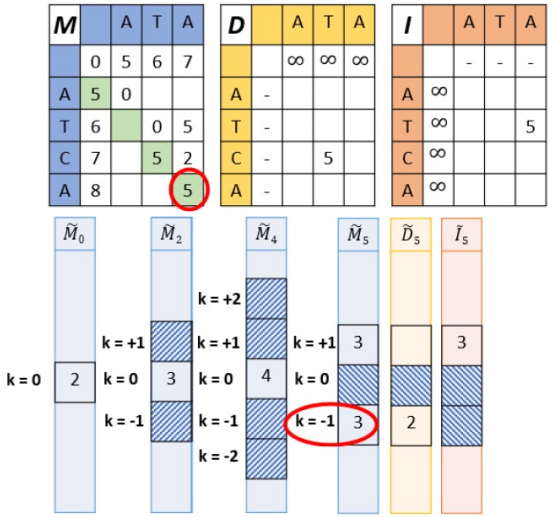
D		A	T	A
		∞	∞	∞
A	-	10	11	12
T	-	5	10	11
C	-	6	5	10
A	-	7	6	7

I		A	T	A
		-	-	-
A	∞	10	5	6
T	∞	11	10	5
C	∞	12	11	10
A	∞	13	12	11

GenASM



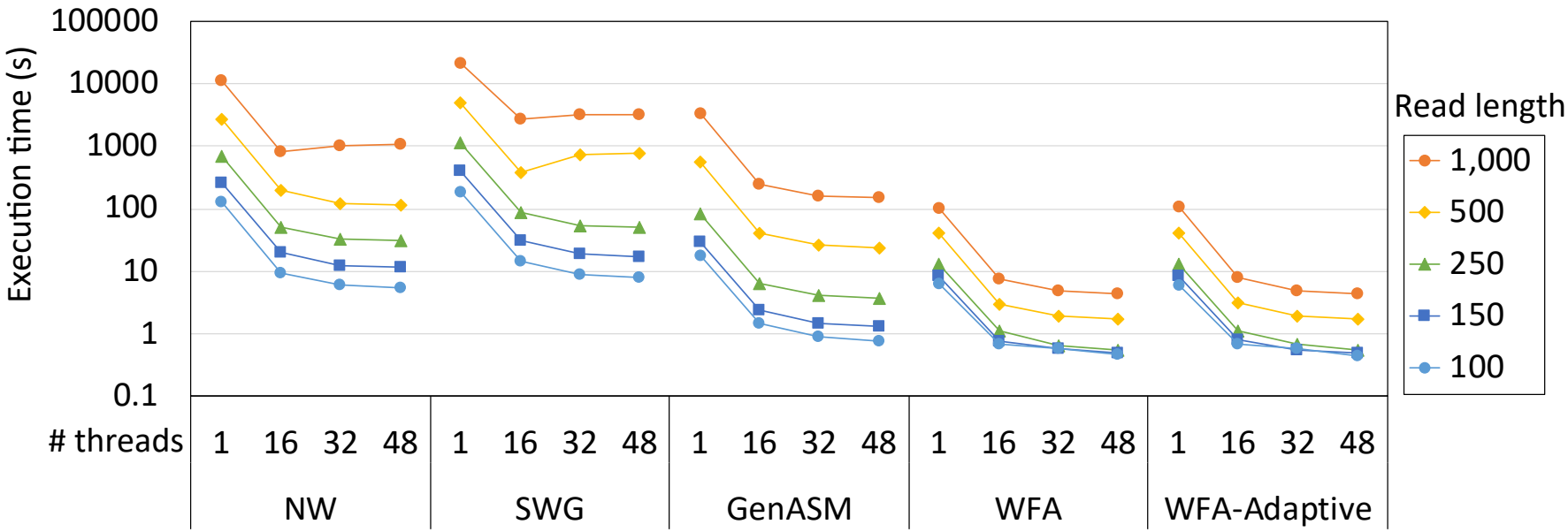
Wavefront Algorithm (WFA)



Senol Cali et al. "GenASM: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis." MICRO 2020
Marco-Sola et al. "Fast gap-affine pairwise alignment using the wavefront algorithm." Bioinformatics (2021)

Motivation: Alignment on CPU (I)

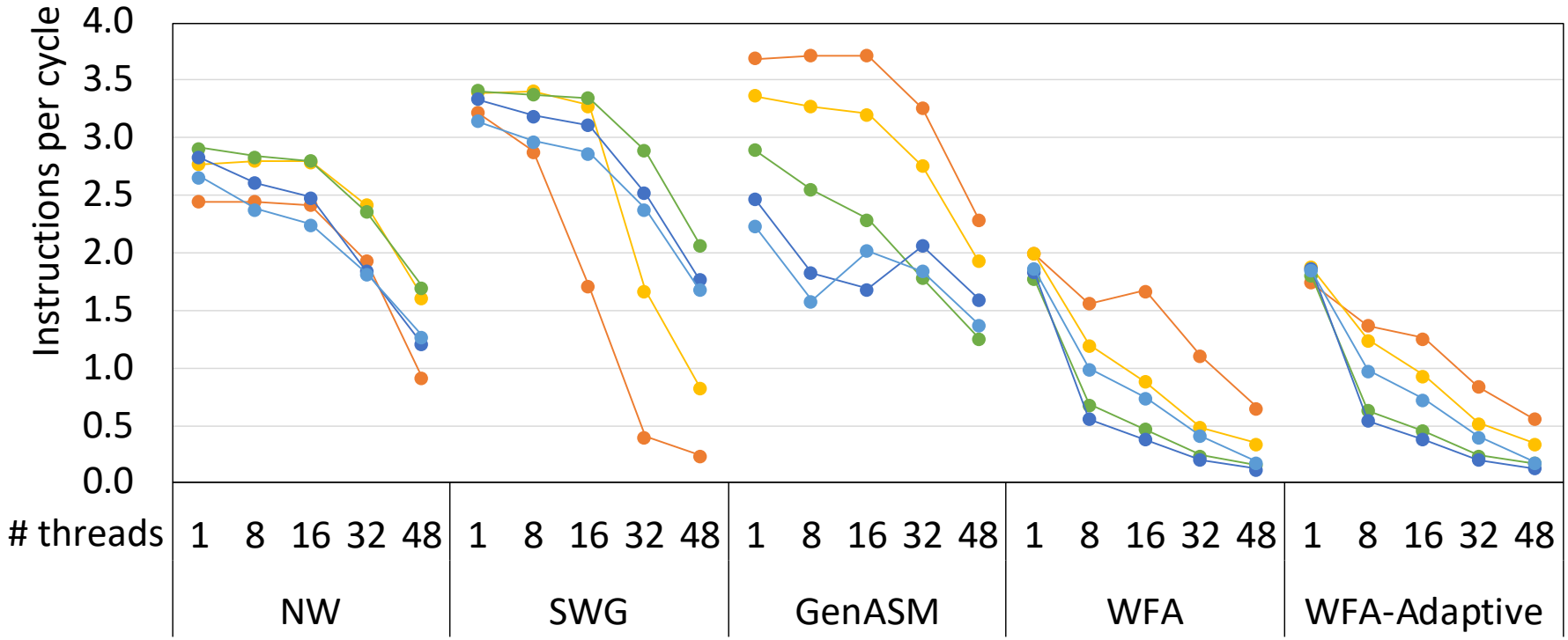
- Scaling of sequence alignment on CPU
  - Execution time



Performance tends to saturate
as the number of CPU threads grows

Motivation: Alignment on CPU (II)

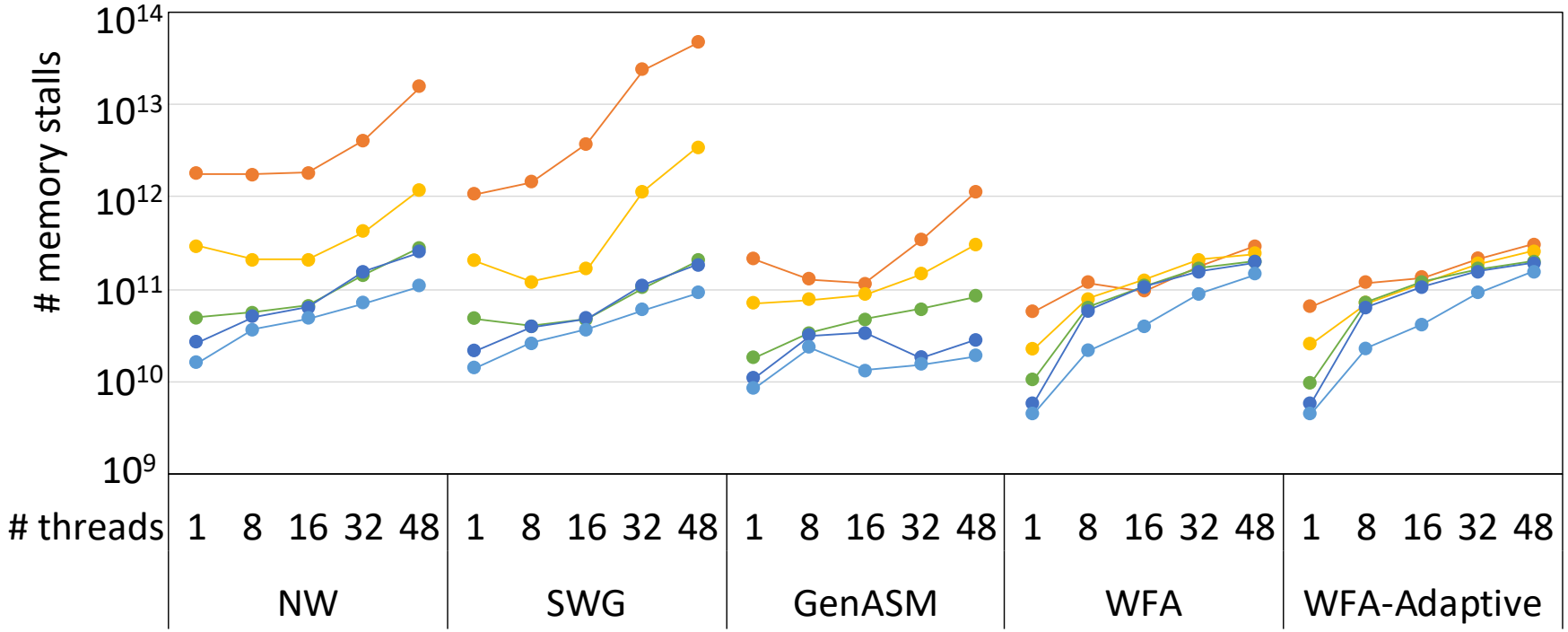
- Scaling of sequence alignment on CPU
  - Instructions per cycle



As the number of CPU threads grows, IPC decreases.
This means that threads spend more time idle

Motivation: Alignment on CPU (III)

- Scaling of sequence alignment on CPU
  - Memory stalls



As the number of CPU threads grows,
threads spend more time waiting for memory

Outline

Background & Motivation:
Sequence Alignment on CPU

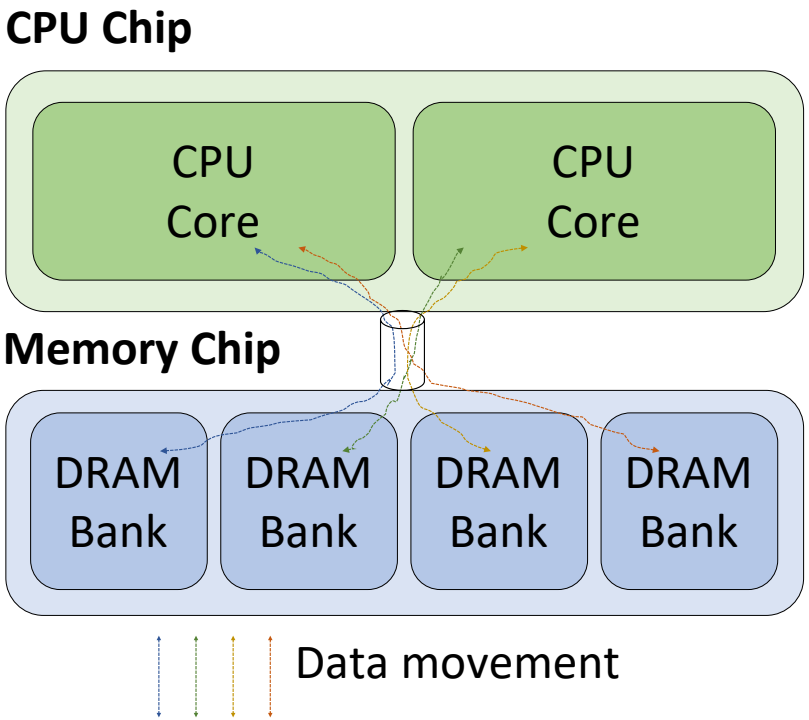
Processing-in-memory to Alleviate the Data Movement Bottleneck

Alignment-in-memory: A Framework for Sequence Alignment on PIM Systems

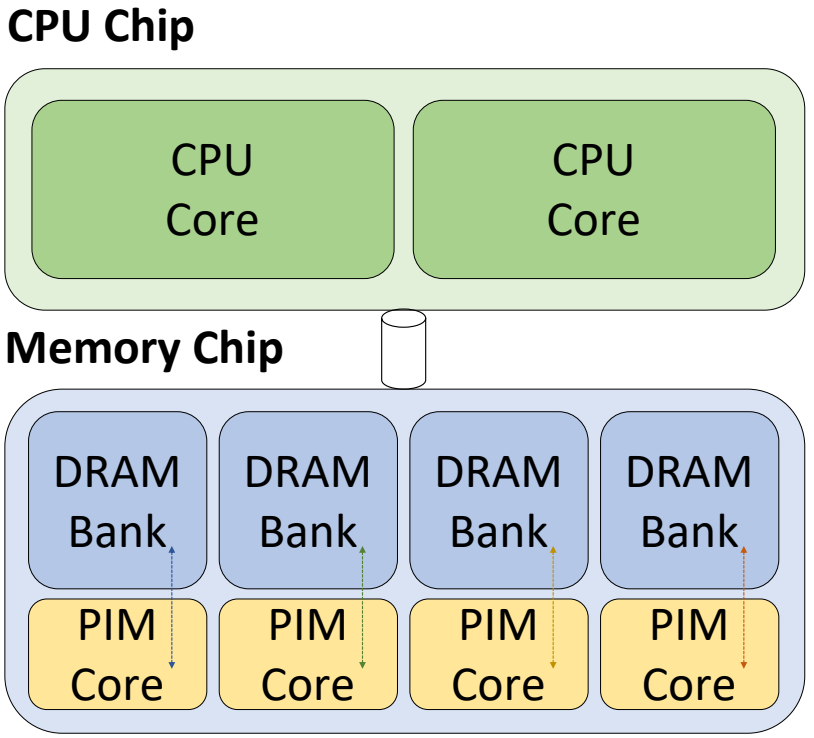
Evaluation

PIM to Minimize Data Movement

- Processing-in-memory can alleviate the data movement bottleneck



Conventional CPU processing



Processing-in-memory (PIM)

Outline

Background & Motivation:
Sequence Alignment on CPU

Processing-in-memory to Alleviate the Data Movement Bottleneck

Alignment-in-memory: A Framework for Sequence Alignment on PIM Systems

Evaluation

Alignment-in-Memory (AiM)

- Supported algorithms

Needleman-Wunsch (NW)

D		A	T	A
	0	4	8	12
A	4	0	4	8
T	8	4	0	4
C	12	8	4	2
A	16	12	8	4

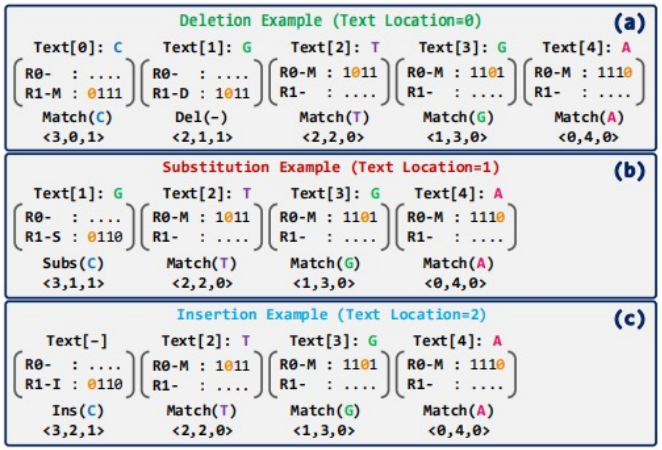
Smith-Waterman-Gotoh (SWG)

M		A	T	A
	0	5	6	7
A	5	0	5	6
T	6	5	0	5
C	7	6	5	2
A	8	7	6	5

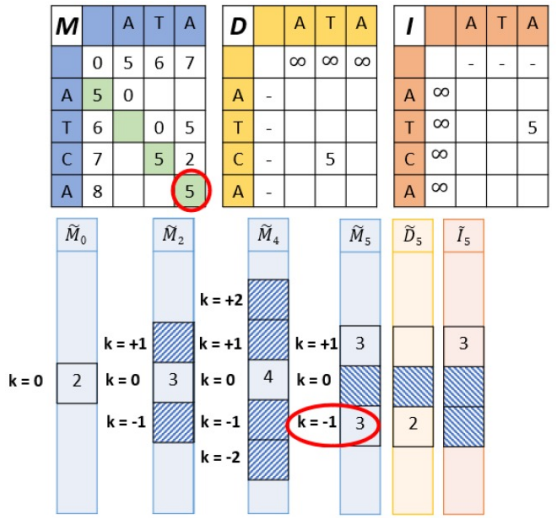
D		A	T	A
		∞	∞	∞
A	-	10	11	12
T	-	5	10	11
C	-	6	5	10
A	-	7	6	7

I		A	T	A
		-	-	-
A	∞	10	5	6
T	∞	11	10	5
C	∞	12	11	10
A	∞	13	12	11

GenASM



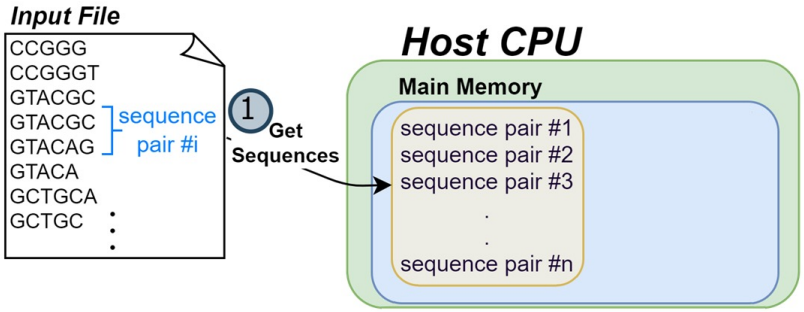
Wavefront Algorithm (WFA)



Senol Cali et al. "GenASM: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis." MICRO 2020
Marco-Sola et al. "Fast gap-affine pairwise alignment using the wavefront algorithm." Bioinformatics (2021)

Alignment-in-Memory (AiM)

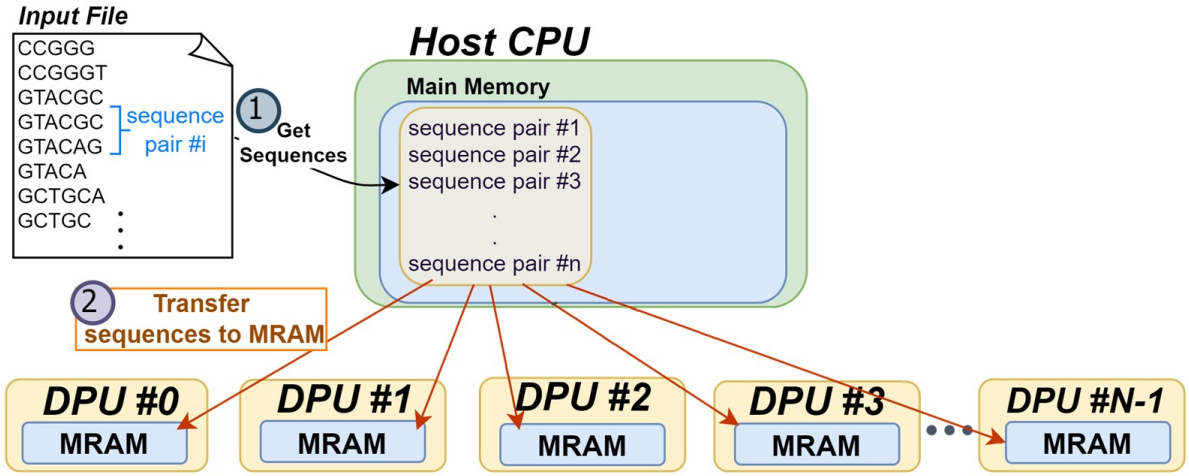
- A framework for sequence alignment on PIM systems



(1) The host CPU reads sequence pairs from an input file and stores them in main memory

Alignment-in-Memory (AiM)

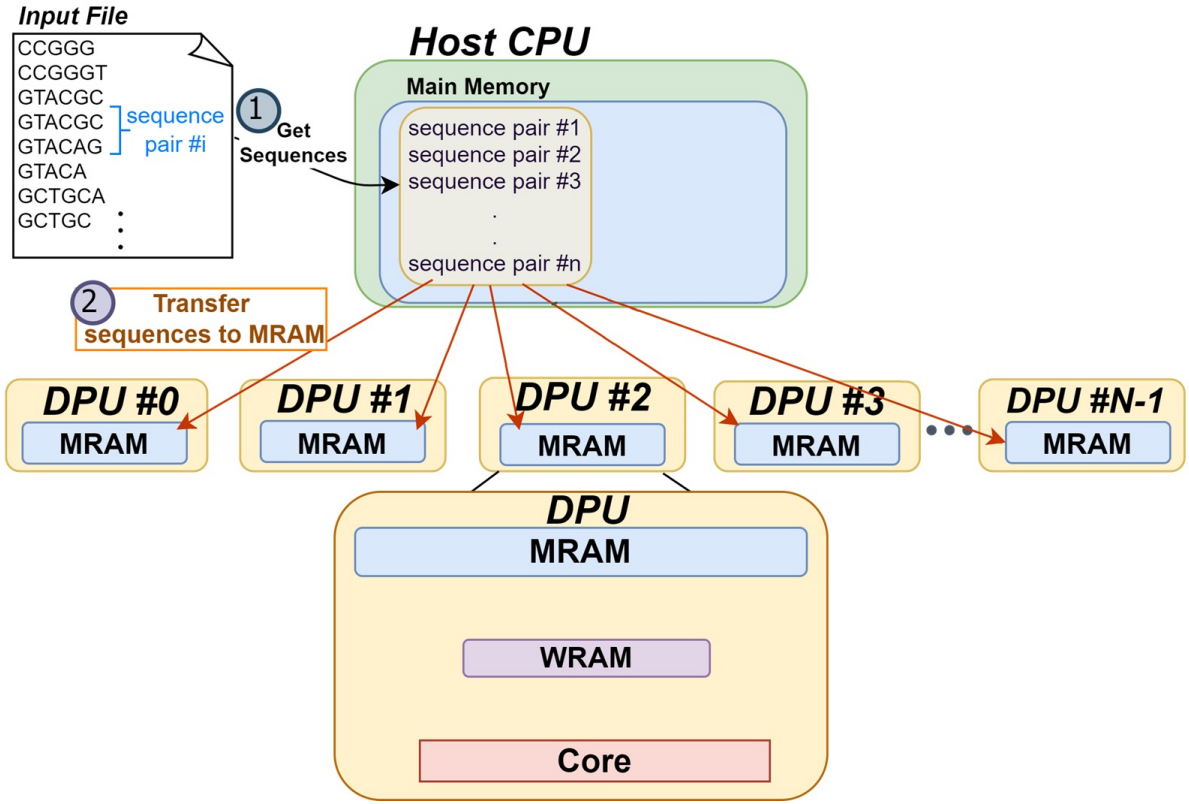
- A framework for sequence alignment on PIM systems



(2) The sequence pairs are evenly partitioned across PIM cores (DPUs) and transferred to the DRAM banks (MRAM) in the PIM DIMMs

Alignment-in-Memory (AiM)

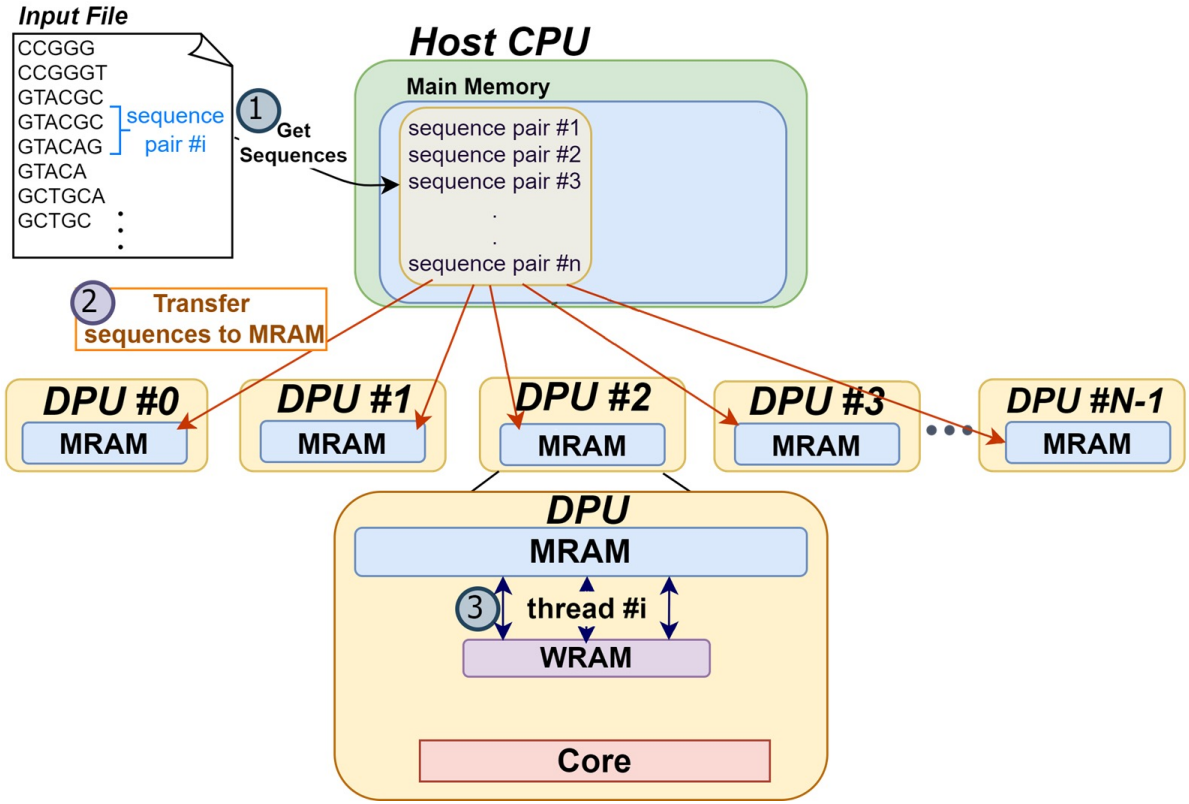
- A framework for sequence alignment on PIM systems



In each PIM core (DPU) different threads will align different sequence pairs. This approach avoids any inter-thread or inter PIM core synchronization

Alignment-in-Memory (AiM)

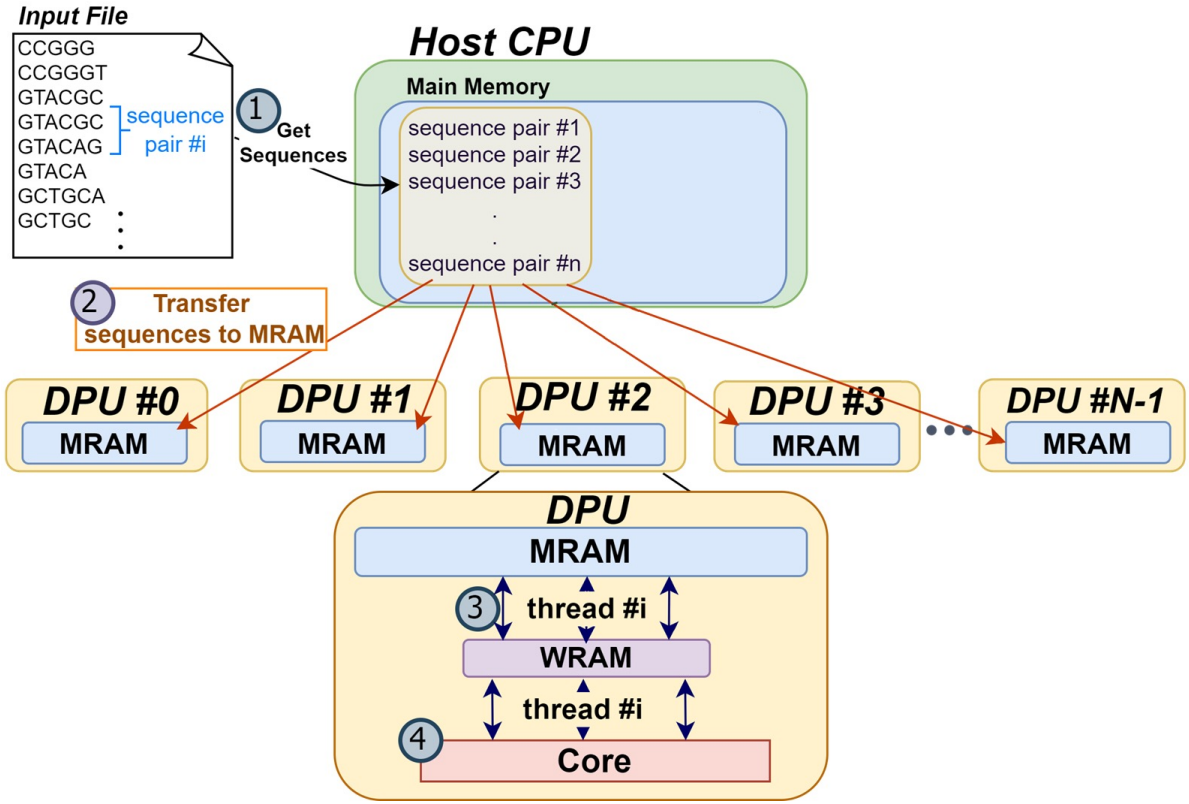
- A framework for sequence alignment on PIM systems



(3) Each thread copies a sequence pair from the DRAM bank (MRAM) to the scratchpad (WRAM) using a DMA transfer

Alignment-in-Memory (AiM)

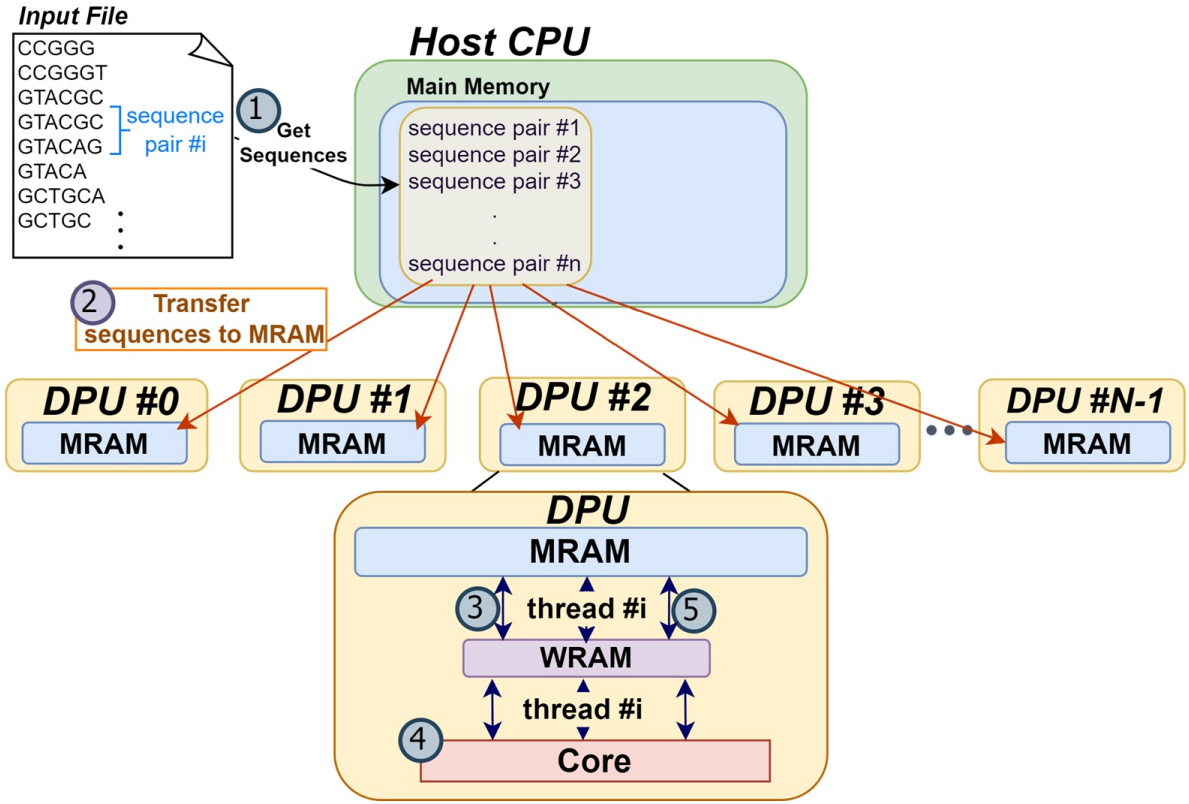
- A framework for sequence alignment on PIM systems



(4) Threads align the sequence pairs and extract the alignment operations using traceback

Alignment-in-Memory (AiM)

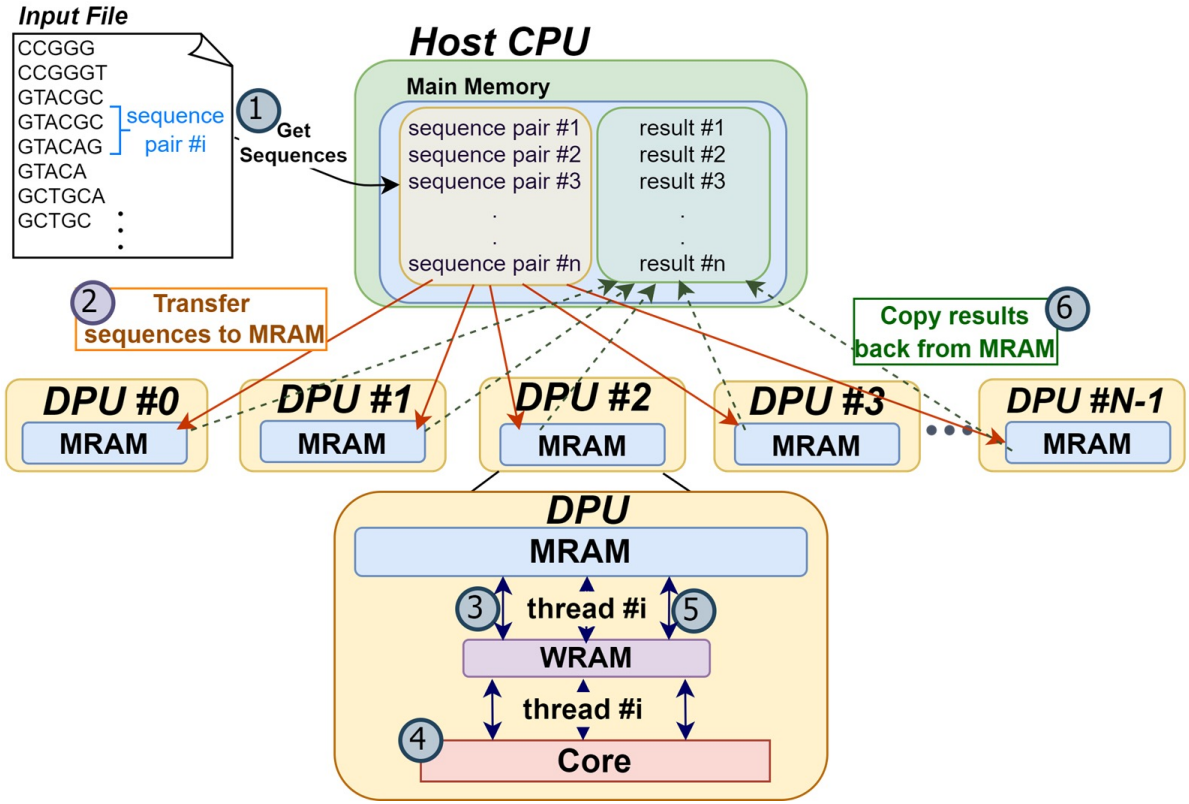
- A framework for sequence alignment on PIM systems



(5) Threads write the alignment score and operations to the DRAM bank via a DMA write transfer

Alignment-in-Memory (AiM)

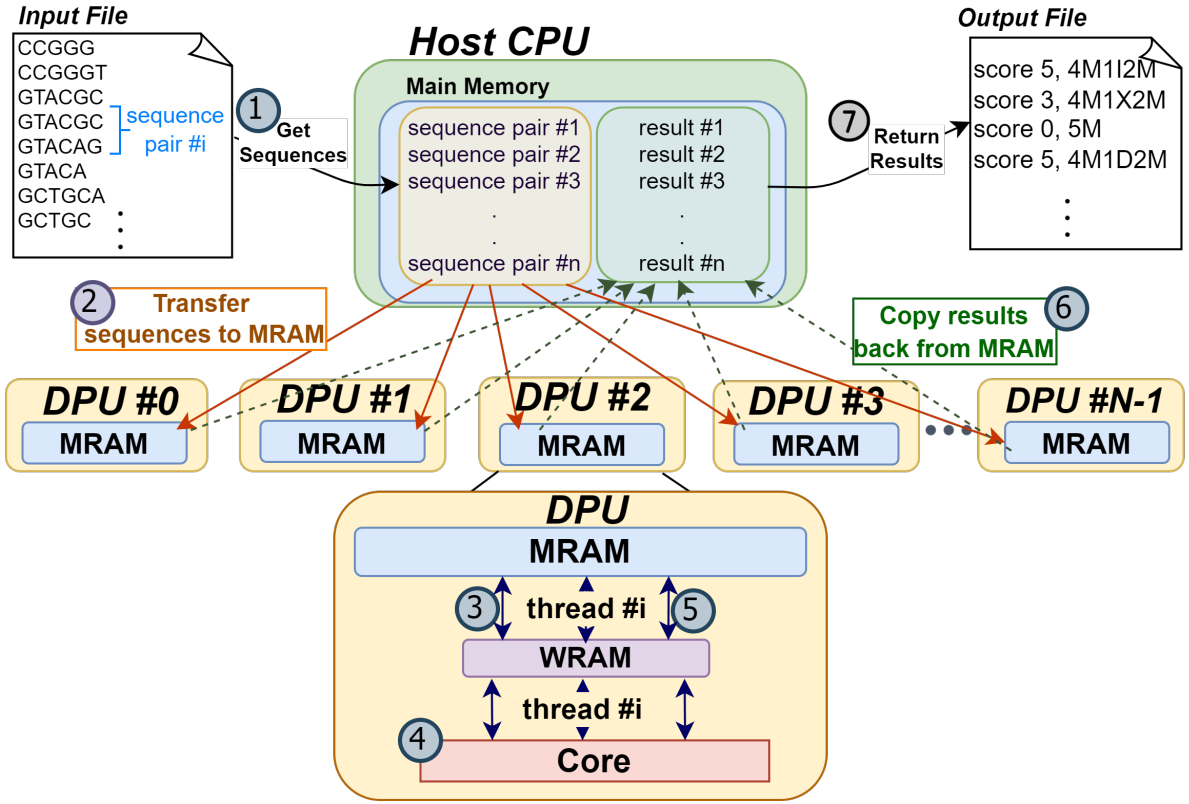
- A framework for sequence alignment on PIM systems



(6) Once the execution on the PIM core has finished, the host CPU reads the alignment score and operations of each sequence pair

Alignment-in-Memory (AiM)

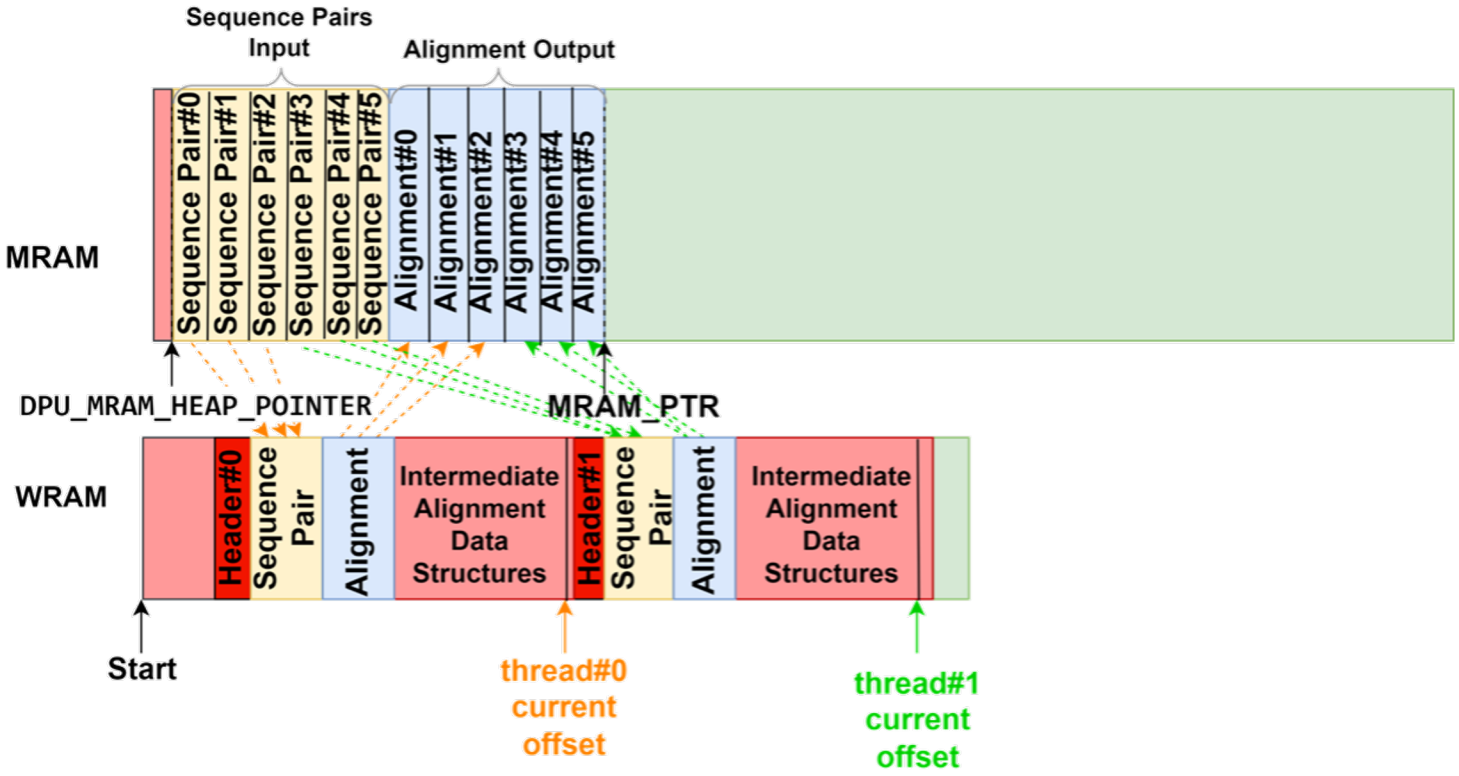
- A framework for sequence alignment on PIM systems



(7) The host CPU writes the alignment results to an output file

Data Management (I)

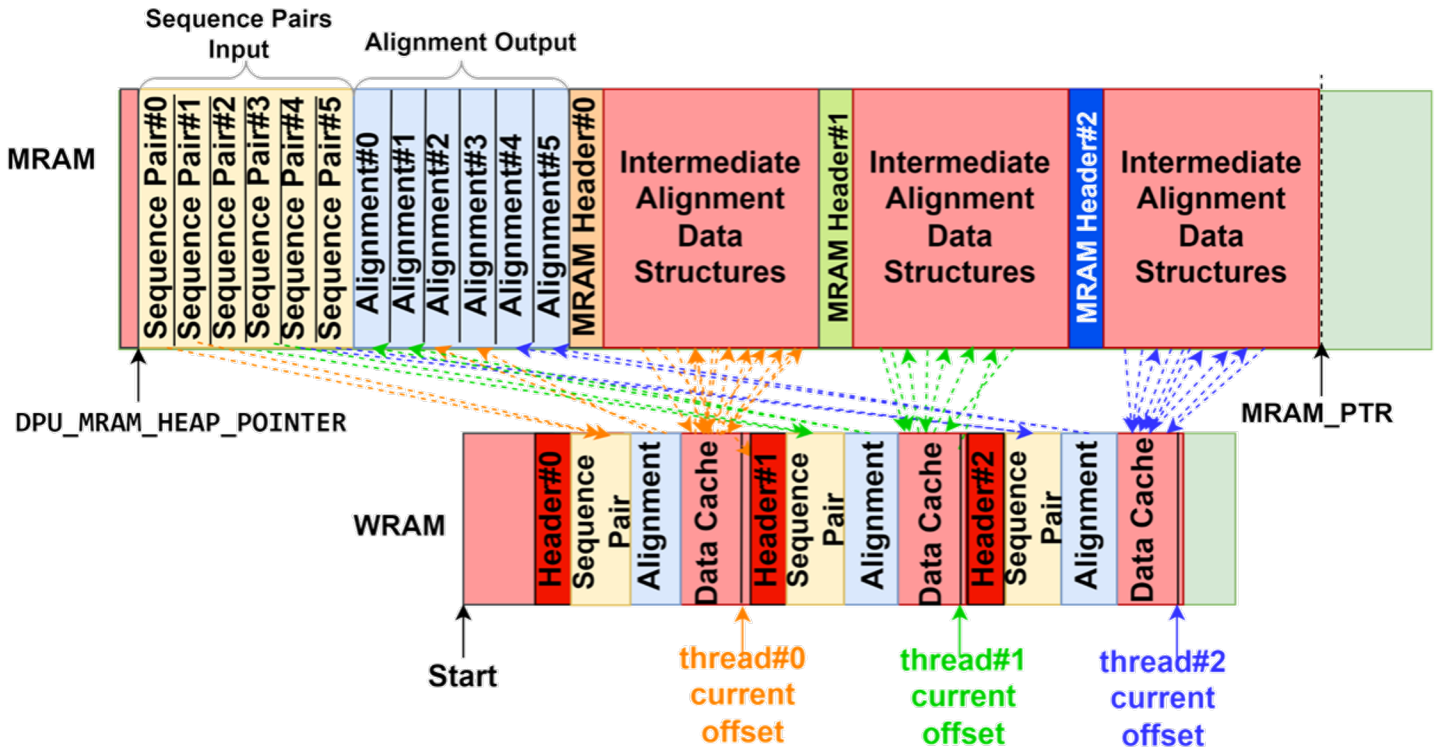
- Using WRAM only for intermediate data structures



The scratchpad (WRAM) is fast but has limited capacity.
As a result, larger sequences need larger intermediate data structures, which may limit the number of concurrent PIM threads

Data Management (II)

- Using WRAM+MRAM for intermediate data structures



Data cache in the scratchpad (WRAM) is loaded on demand.
On average, longer access latency than WRAM only, but it is possible to run more concurrent PIM threads

Data Management (III)

- WFA and WFA-adaptive use intermediate data structures of varying size
  - Wavefront components
- Thus, they require dynamic memory allocation
- UPMEM SDK provides an incremental dynamic memory allocator
  - However, this memory allocator is shared by all PIM threads, which requires synchronization
- We design a custom per-thread memory allocator for low-overhead dynamic memory allocation

Outline

Background & Motivation:
Sequence Alignment on CPU

Processing-in-memory to Alleviate the Data Movement Bottleneck

Alignment-in-memory: A Framework for Sequence Alignment on PIM Systems

Evaluation

Evaluation Methodology

- Real and synthetic datasets

Read Lengths	Edit Distances	Description
100	0-5%	Real, Accession# ERR240727 *
150	0-5%	Real, Accession# SRR826460 *
250	0-5%	Real, Accession# SRR826471 *
500, 1000, 5000, 10000	0-5%	Synthetic ^

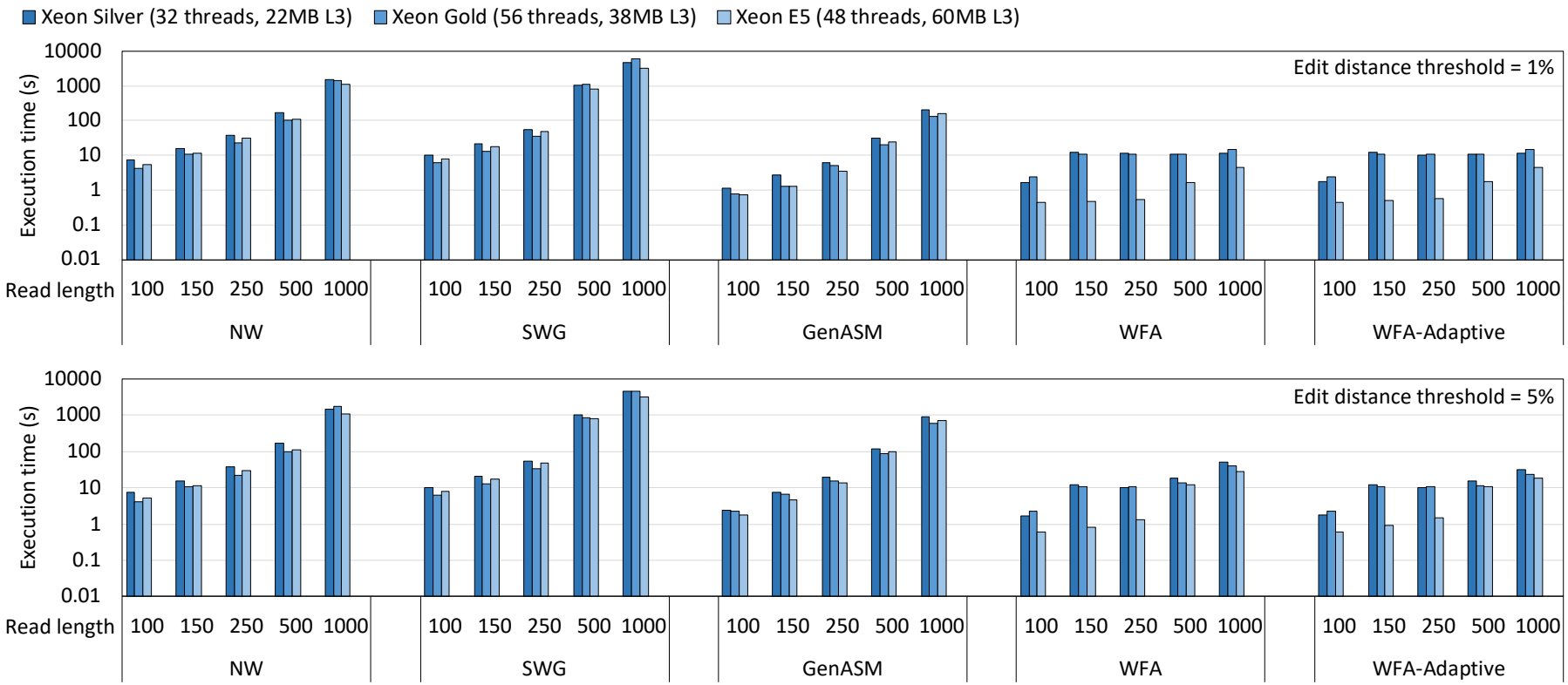
*https://www.ebi.ac.uk/ena/browser/view
^https://github.com/smarco/WFA2-lib

- Evaluated systems
  - UPMEM PIM system with 2,560 PIM cores @ 425 MHz and 160 GB of DRAM
  - Three CPU systems

System	1	2	3
CPU	Intel Xeon Silver 4215	Intel Xeon Gold 5120	Intel Xeon E5-2697 v2
Process node	14 nm	14 nm	22 nm
Sockets	2	2	2
Cores	16	28	24
Threads	32	56	48
Frequency	2.50 GHz	2.20 GHz	2.70 GHz
L3 cache	22 MB	38 MB	60 MB
Memory	256 GB	64 GB	32 GB
CPU TDP	170 W	210 W	260 W

Evaluation: PIM vs. CPU (I)

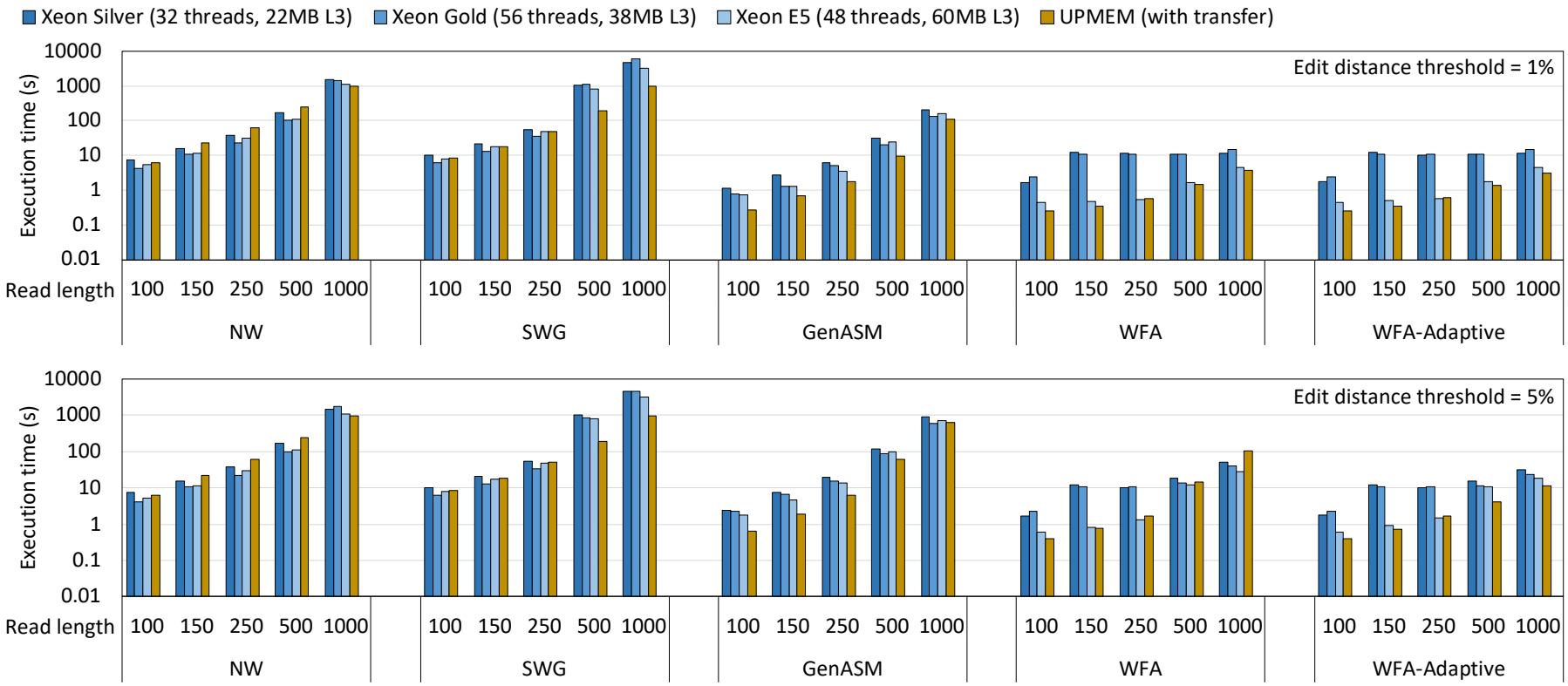
- Evaluation on three CPU systems



The CPU system with the largest L3 cache is the best performing one, which aligns with the memory boundedness of sequence alignment

Evaluation: PIM vs. CPU (II)

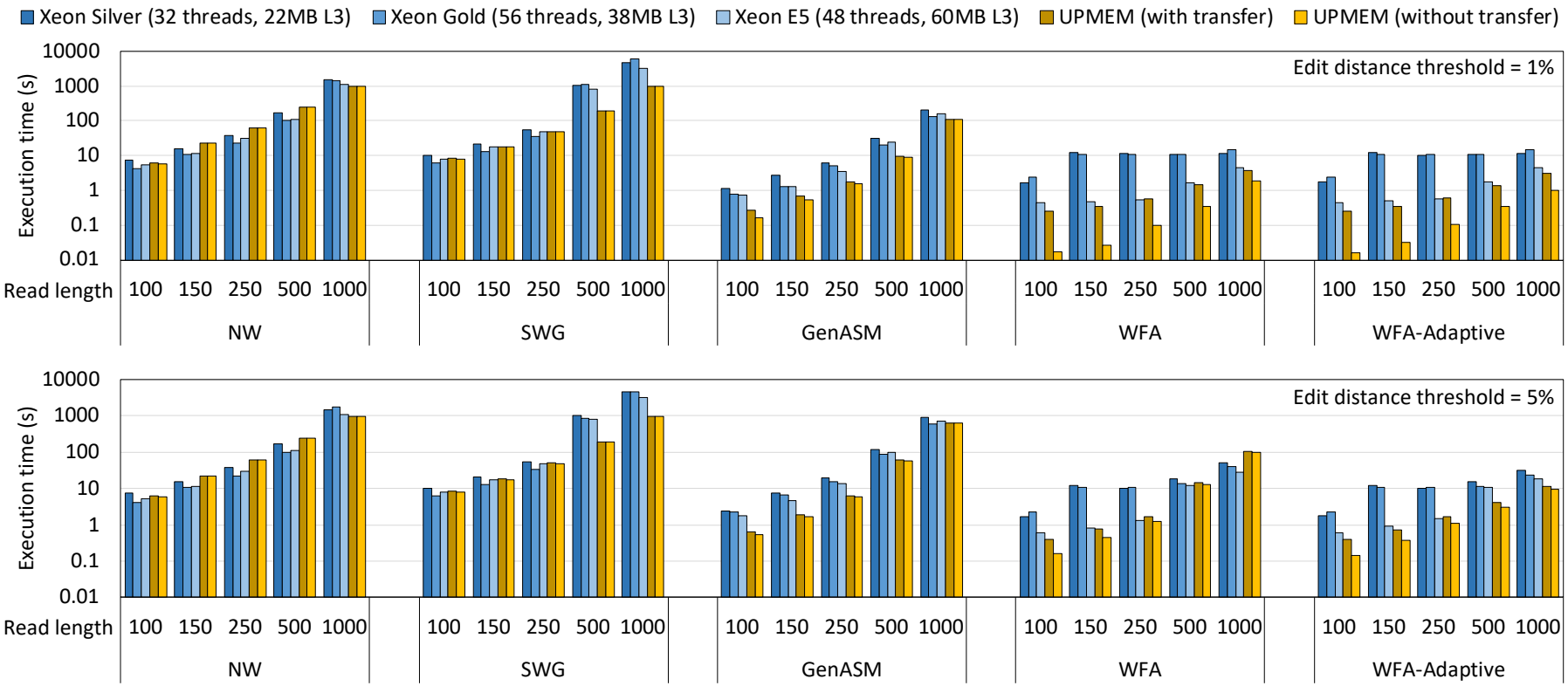
- PIM execution time including CPU-PIM/PIM-CPU transfers



PIM outperforms CPU systems in a majority of experiments
(up to 4.06× for SWG, up to 1.83× for WFA, up to 2.56× for WFA-adaptive)

Evaluation: PIM vs. CPU (III)

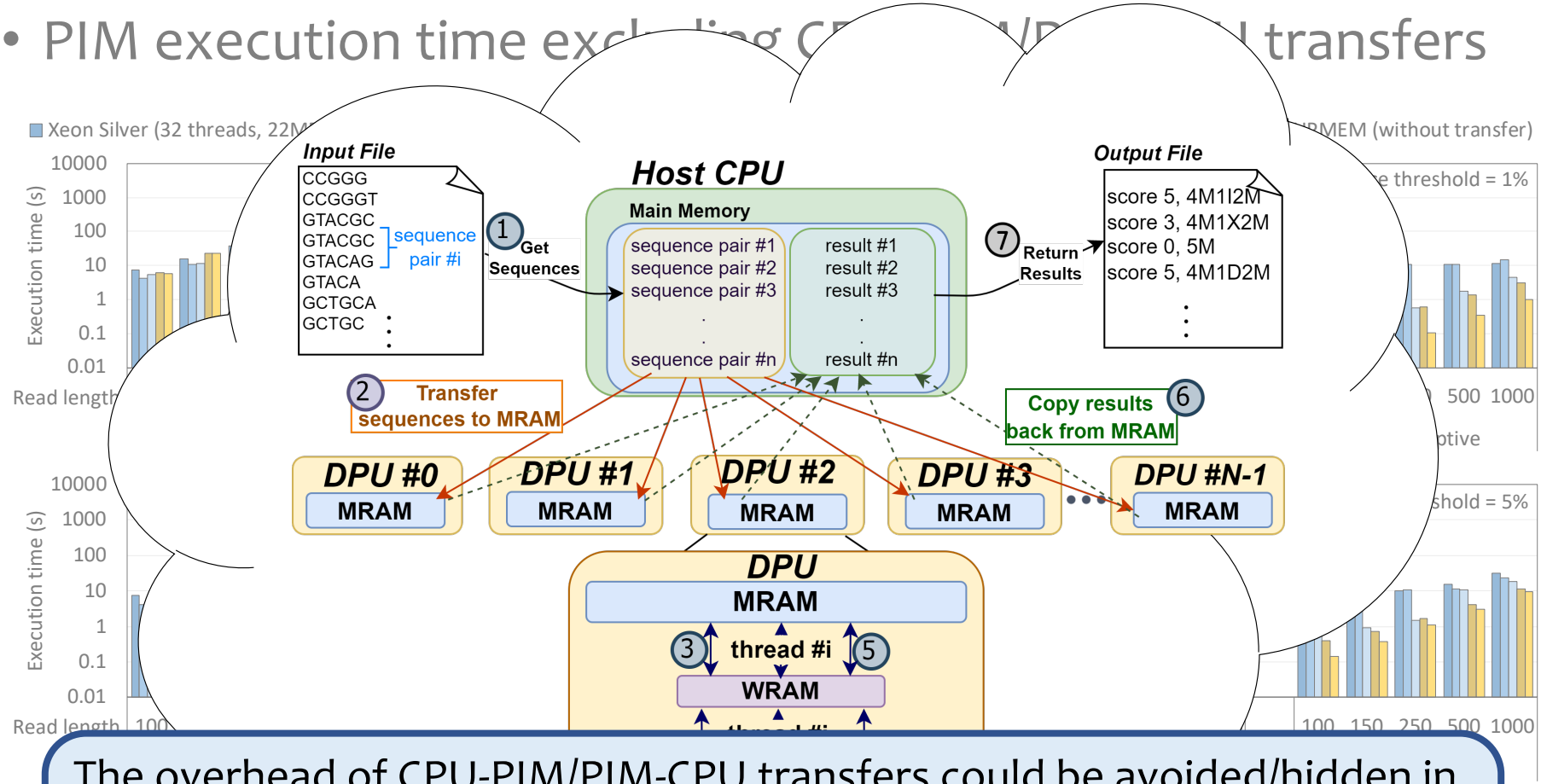
- PIM execution time excluding CPU-PIM/PIM-CPU transfers



PIM speedups over CPU increase significantly when data transfer times are not included (up to 25.93× for WFA, up to 28.14× for WFA-adaptive)

Evaluation: PIM vs. CPU (III)

- PIM execution time excluding CPU-PIM/PIM-CPU transfers

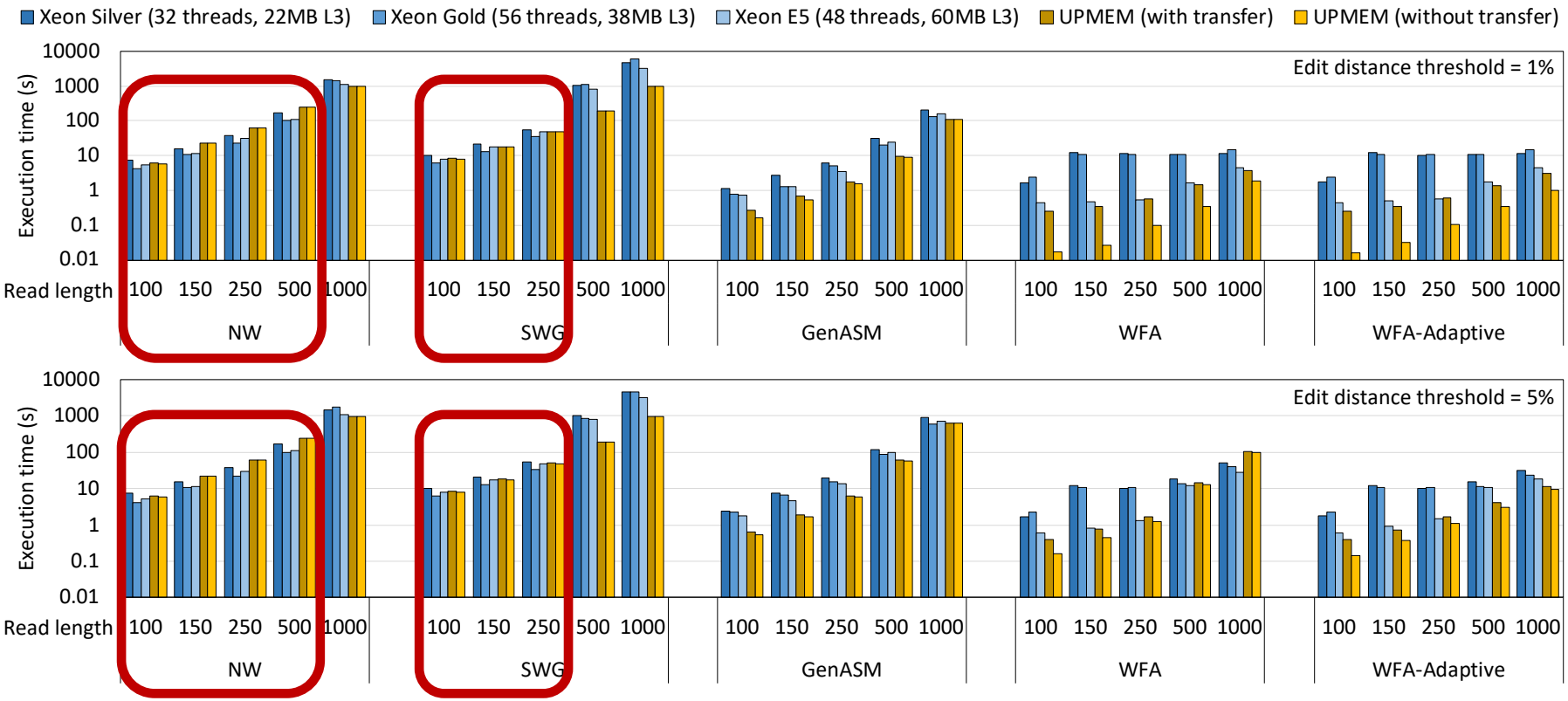


The overhead of CPU-PIM/PIM-CPU transfers could be avoided/hidden in future PIM systems that:

- Use PIM DIMMs as main memory, or
- Overlap data transfers and computation

Evaluation: PIM vs. CPU (IV)

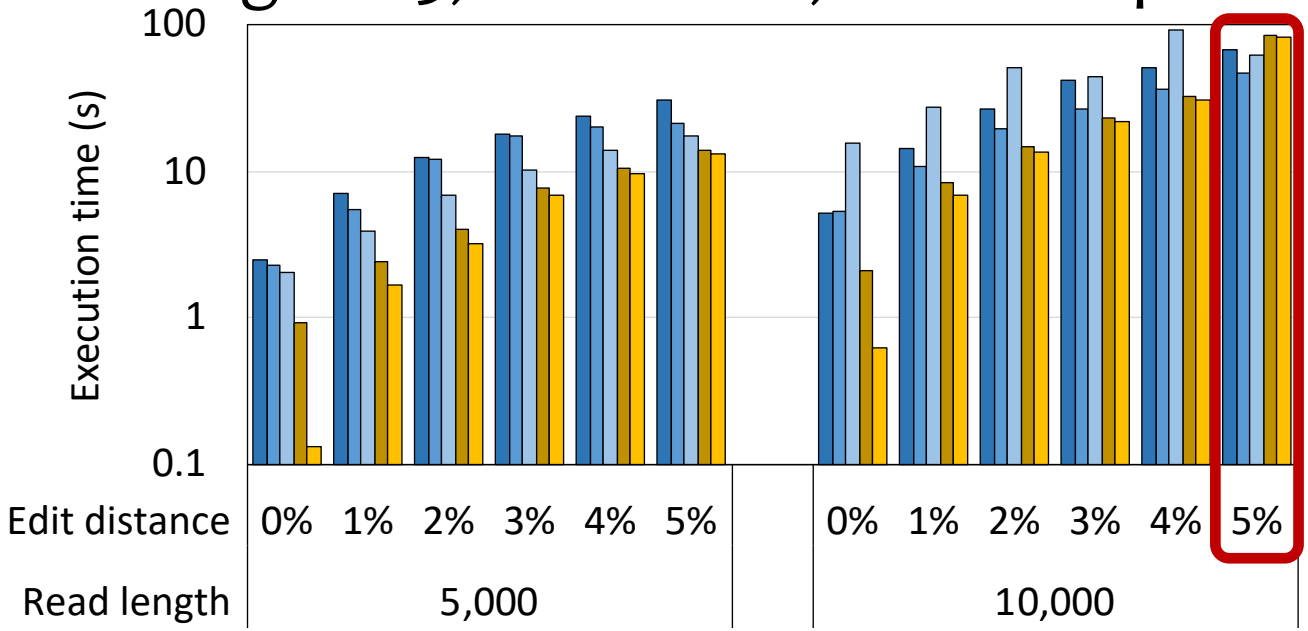
- Algorithms with regular access patterns



PIM does not outperform CPU for algorithms with regular access patterns at small read lengths

WFA-Adaptive with Large Sequences

- WFA and WFA-adaptive have linear space complexity
  - Unlikely to exceed MRAM capacity
- Large read lengths: 5,000 and 10,000 base pairs

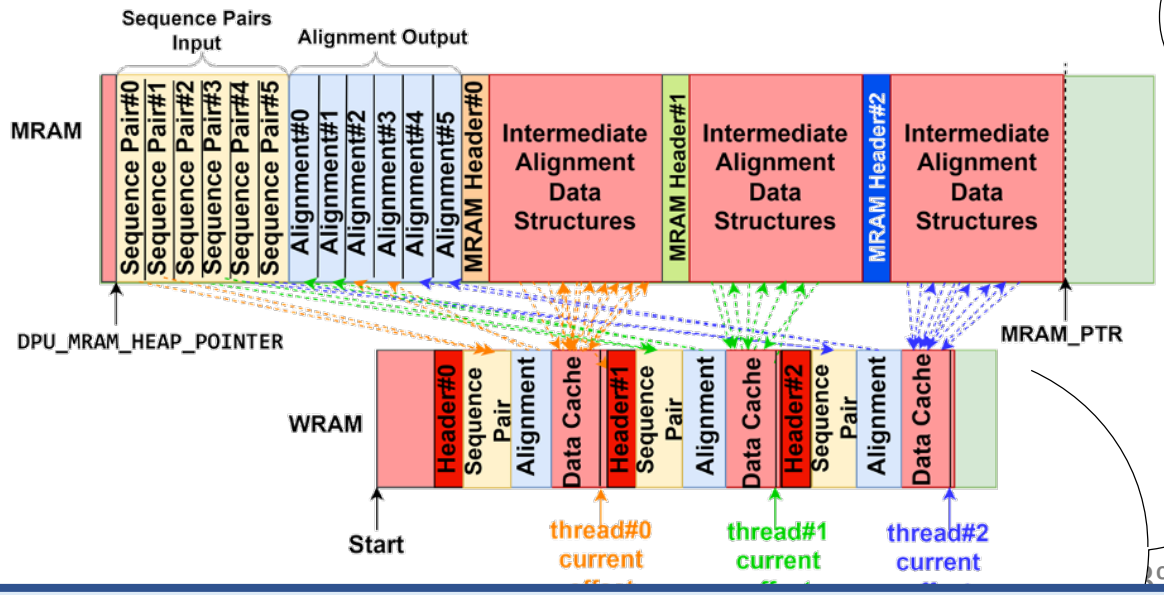


PIM continues to outperform the CPU systems for large read lengths

Scalability is currently limited by WRAM capacity

WFA-Adaptive with Large Sequences

- WFA and WFA-adaptive h[illegible] complexity
  - Unli[illegible]
- Lar[illegible]



Possible software solutions for (even) larger sequences:

- Reduce WRAM footprint by streaming partial wavefront components from MRAM, or
- Multiple PIM threads per alignment

Future UPMEM PIM cores may have larger WRAM capacity

Evaluation: PIM vs. GPU

- WFA on PIM vs. WFA on GPU

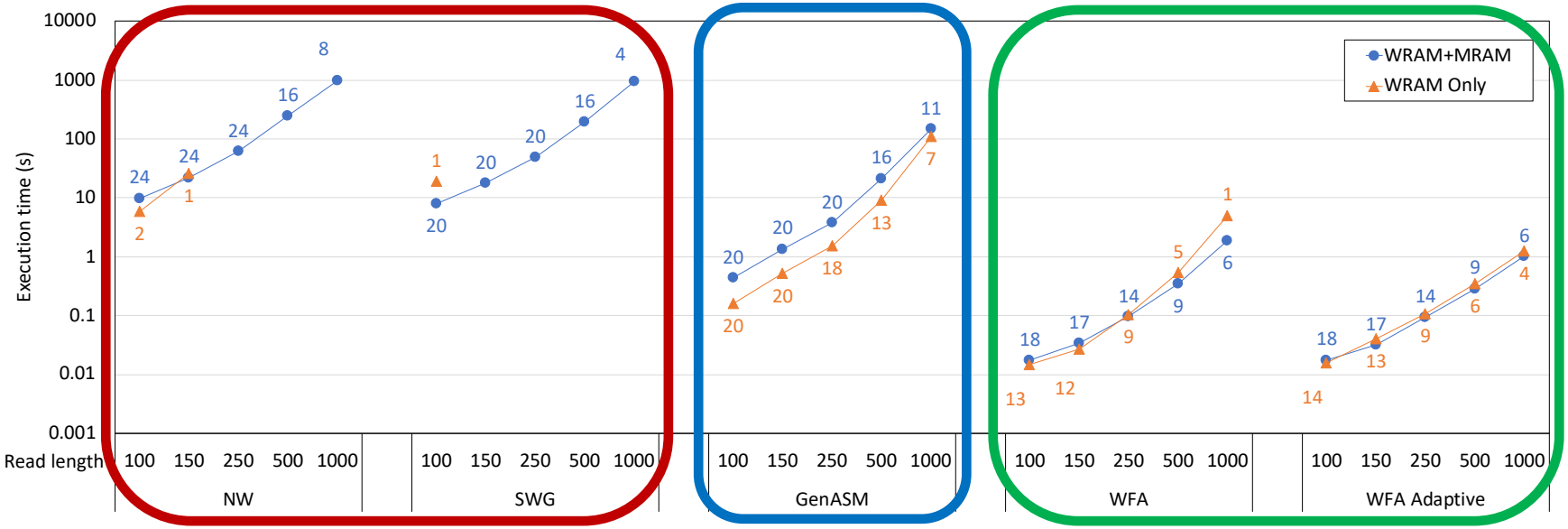
Sequence length	Edit distance	Throughput (alignments per second)		Throughput improvement
		WFA-GPU	UPMEM (with transfer)	
150	2%	9.09M	12.97M	1.42×
	5%	5.56M	7.03M	1.27×
1,000	2%	1.43M	1.10M	0.77×
	5%	370K	434K	1.17×
10,000	2%	25.0K	66.9K	2.68×
	5%	5.56K	11.81K	2.12×

PIM outperforms GPU in a majority of experiments

Aguado-Puig et al. "WFA-GPU: Gap-affine pairwise alignment using GPUs." bioRxiv (2022)

Intermediate Data Structures

- WRAM only vs. WRAM+MRAM for intermediate data structures



For large data structures (NW, SWG), WRAM only does not scale

For algorithms that use small data structures (GenASM), WRAM only is better

For algorithms that use medium-sized data structures (WFA, WFA-adaptive), WRAM only is better for short reads while WRAM+MRAM is better for long reads

Sequence Alignment on Processing-in-Memory

- Published in Bioinformatics (2023)

A Framework for High-throughput Sequence Alignment using Real Processing-in-Memory Systems

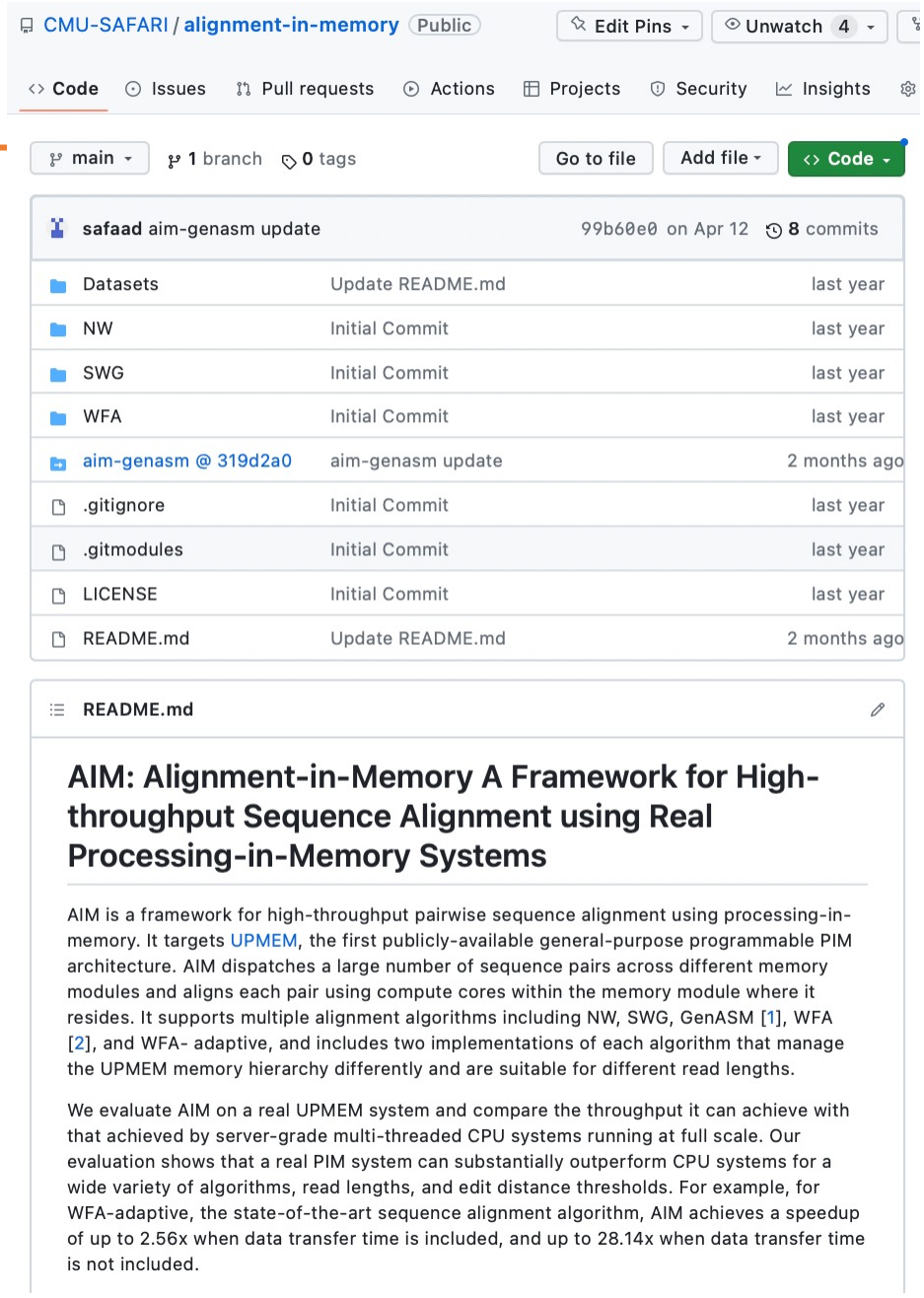
Safaa Diab [1] Amir Nassereldine [1] Mohammed Alser [2] Juan Gómez Luna [2]
Onur Mutlu [2] Izzat El Hajj [1]

[1]American University of Beirut [2]ETH Zürich

https://arxiv.org/pdf/2204.02085.pdf
https://arxiv.org/pdf/2208.01243.pdf
Source code: https://github.com/safaad/aim

Source Code

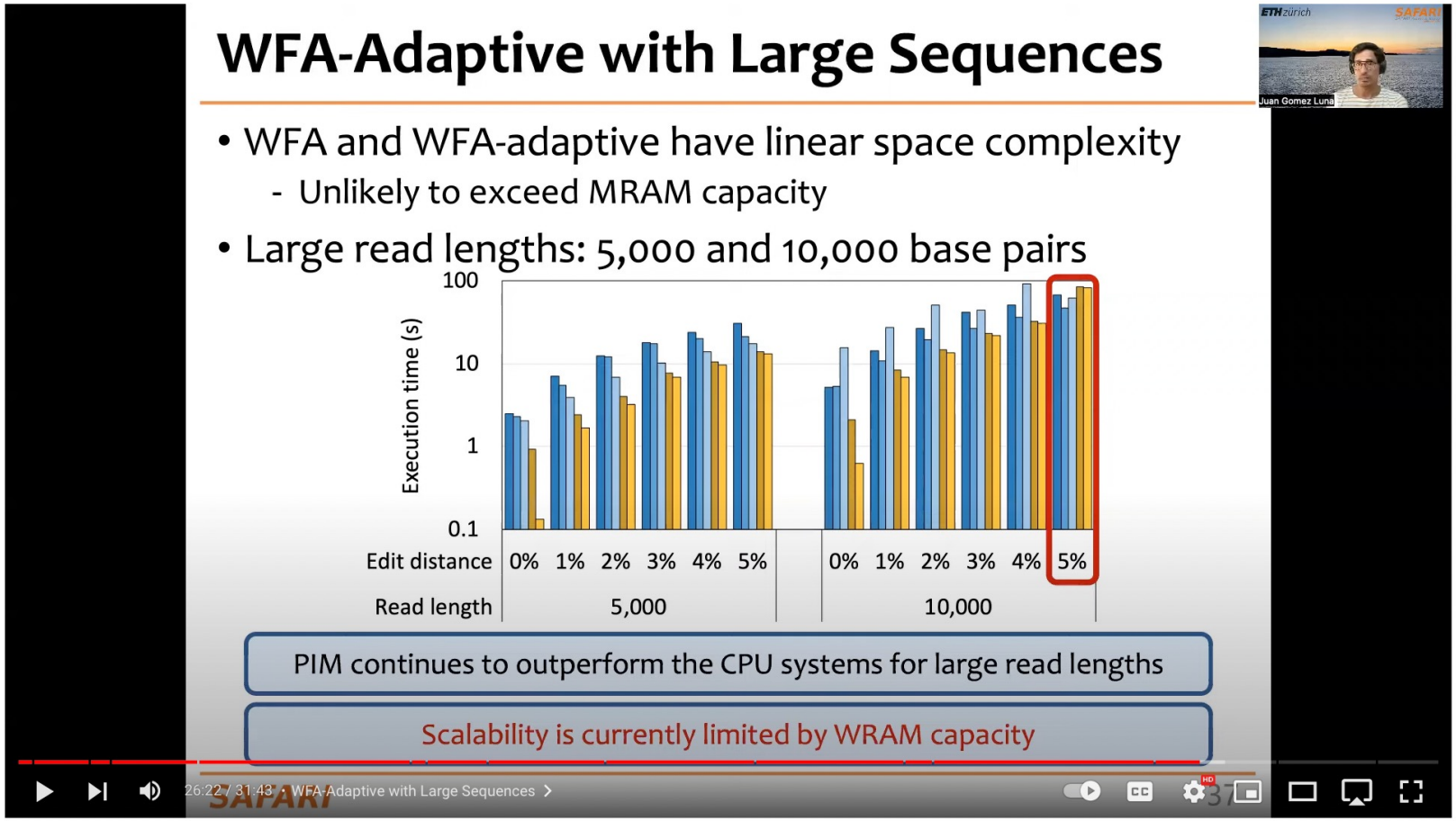
- https://github.com/CMU-SAFARI/alignment-in-memory
- https://github.com/safaad/aim



Executive Summary

- Genome sequence alignment is typically memory-bound on conventional processor-centric systems (CPU, GPU)
- Processing-in-memory (PIM) has the potential to overcome this data movement bottleneck by placing compute capabilities near/inside memory arrays
- We present Alignment-in-Memory (AiM), a PIM framework for genome sequence alignment on general-purpose PIM systems
  - AiM supports multiple alignment algorithms: NW, SWG, GenASM, WFA
  - Implemented for the UPMEM PIM architecture
- Our evaluation shows that sequence alignment on PIM can be significantly faster than on CPUs and GPUs

Lecture on Alignment-in-Memory



PIM Course: Lecture 14: Genome Sequence Alignment on PIM (Spring 2023)

Onur Mutlu Lectures
33.4K subscribers
Subscribed
5 Share Clip Save

161 views 5 days ago Livestream - Data-Centric Architectures: Fundamentally Improving Performance and Energy (Spring 2023)
Projects & Seminars, ETH Zürich, Spring 2023
Data-Centric Architectures: Fundamentally Improving Performance and Energy

High Throughput Sequence Alignment Using Real Processing-in-Memory Systems

Safaa Diab, Amir Nassereldine, Mohammed Alser,
Juan Gómez Luna, Onur Mutlu, Izzat El Hajj

https://arxiv.org/pdf/2208.01243.pdf
https://github.com/CMU-SAFARI/alignment-in-memory
juang@ethz.ch

ETH Zürich

SAFARI

Thursday, June 8, 2023

Transcendental Functions

TransPimLib:

Efficient Transcendental Functions for Processing-in-Memory Systems

Maurus Item, Juan Gómez Luna, Yuxin Guo,
Geraldo F. Oliveira, Mohammad Sadrosadati, Onur Mutlu

https://arxiv.org/pdf/2304.01951.pdf
https://github.com/CMU-SAFARI/transpimlib
juang@ethz.ch

ETH Zürich

SAFARI

Thursday, June 1, 2023

ISPASS 2023 Version

- Presented at ISPASS 2023

TransPimLib: Efficient Transcendental Functions for Processing-in-Memory Systems

Maurus Item Juan Gómez-Luna Yuxin Guo
Geraldo F. Oliveira Mohammad Sadrosadati Onur Mutlu
ETH Zürich

https://people.inf.ethz.ch/omutlu/pub/TransPIMLib_ispass23.pdf

Source code: https://github.com/CMU-SAFARI/transpimlib

https://youtu.be/lqqf4eaaEE4

Executive Summary

- Processing-in-Memory (PIM) promises to alleviate the *data movement bottleneck*
- However, current real-world PIM systems have very constrained hardware, which results in limited instruction sets
  - Difficulty/impossibility of computing complex operations, such as transcendental functions (e.g., trigonometric, exp, log) and other hard-to-calculate functions (e.g., square root)
  - These functions are important for modern workloads, e.g., activation functions in machine learning applications
- TransPimLib is the first library for transcendental and other hard-to-calculate functions on general-purpose PIM systems
  - CORDIC-based and LUT-based methods for trigonometric functions, hyperbolic functions, exponentiation, logarithm, square root, etc.
  - Source code: https://github.com/CMU-SAFARI/transpimlib
- We implement TransPimLib for the UPMEM PIM architecture and evaluate its methods in terms of performance, accuracy, memory requirements, and setup time
  - Three real workloads (Blackscholes, Sigmoid, Softmax)

Outline

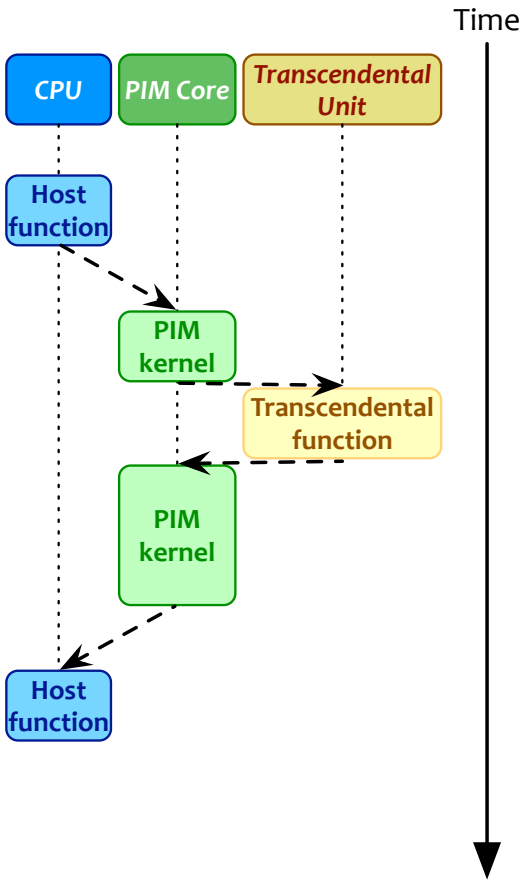
Processing-in-memory
and transcendental functions

TransPimLib:
A library for transcendental
and other hard-to-calculate functions

Evaluation

How to Calculate Transcendental Functions in a PIM System?

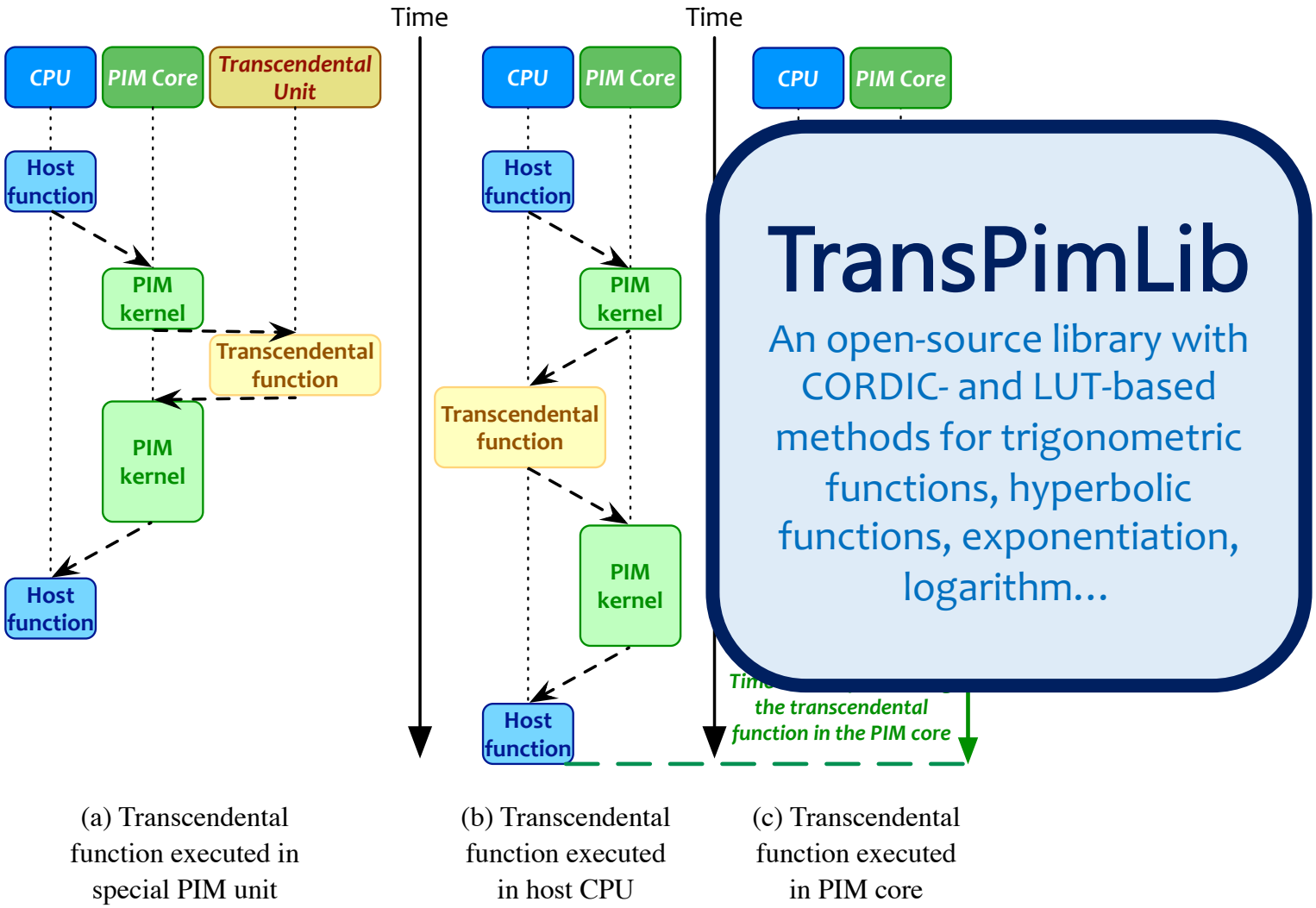
- Three possible alternatives



(a) Transcendental function executed in special PIM unit

How to Calculate Transcendental Functions in a PIM System?

- Three possible alternatives



(a) Transcendental function executed in special PIM unit

(b) Transcendental function executed in host CPU

(c) Transcendental function executed in PIM core

Outline

Processing-in-memory
and transcendental functions

TransPimLib:
A library for transcendental
and other hard-to-calculate functions

Evaluation

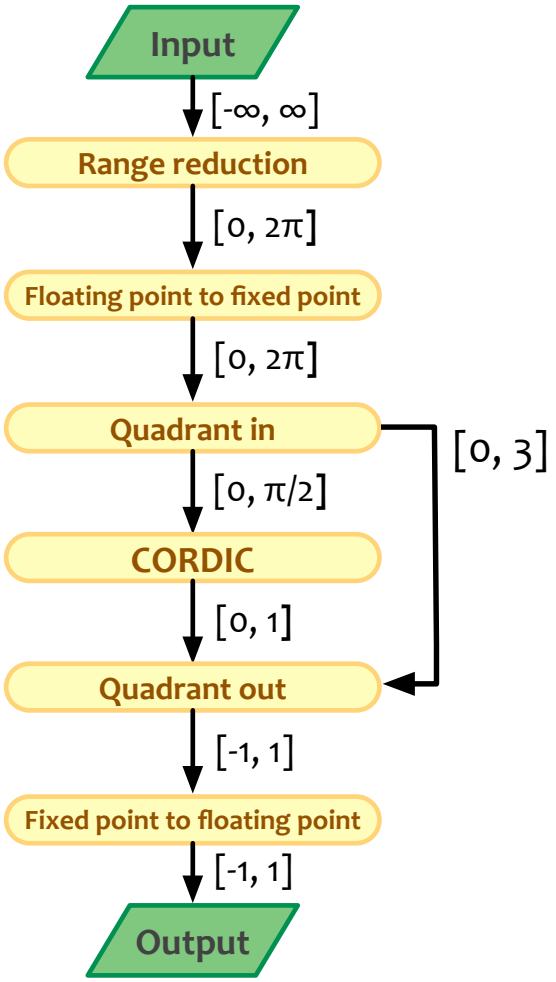
TransPimLib: Implementation

- Various methods to calculate transcendental functions:
  - Taylor approximation, minimax polynomials, CORDIC, LUTs
- CORDIC is an iterative method that uses bit-shifts, additions, and table lookups
  - In rotation mode, CORDIC computes the function value for an input θ by rotating a vector [1, 0] iteratively
  - The rotation is done by multiplying the vector and a matrix
  - The matrix represents the rotation angle, which decreases in each iteration
- Fuzzy Lookup Tables (LUTs) return an (approximate) output $f(x)$ for each input x
  - A function $a(x)$ returns an address to access the LUT
  - The table returns $LUT(a(x)) \simeq f(x)$
  - To generate the LUT, we need a helper function $a^{-1}()$, such that $x = a(a^{-1}(x))$
  - LUTs' accuracy improves with interpolation:

    $f(x) \simeq LUT(a(x)) + LUT(a(x)+1) - LUT(a(x)) \cdot \Delta$

TransPimLib: CORDIC-based Methods

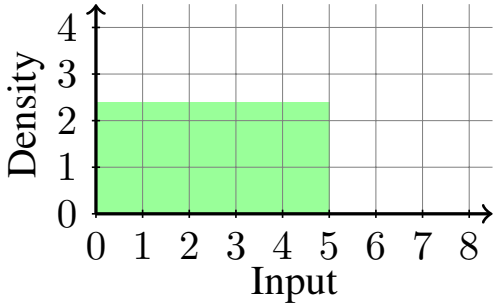
- TransPimLib contains CORDIC implementations of trigonometric (sin, cos, tan) and hyperbolic (sinh, cosh, tanh) functions, exponentiation, logarithm, and square root
- Example: Sine function

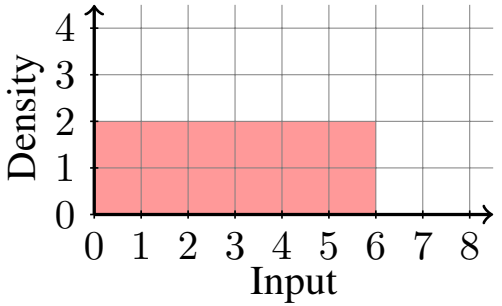


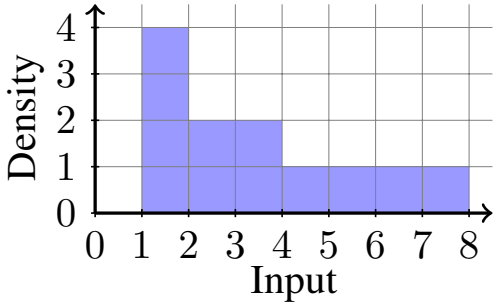
TransPimLib: LUT-based Methods

- Multiplication-based LUT (M-LUT)
  - Regular spacing between table entries
  - $a(x) = round((x - p) \cdot k)$, where k represents the LUT density
- LDEXP-based LUT (L-LUT)
  - Multiplication is cheaper if we multiply by 2^n
  - *ldexp(arg, exp)* to perform arg $\cdot\ 2^{exp}$
  - $a(x) = round((x - p) \cdot 2^n)$
    - k is a power-of-two, which results in less precision but avoids multiplication
- Direct Float Conversion-based LUT (D-LUT)
  - $a(x)$ uses the last n bits of the exponent and p bits of the mantissa
  - Piece-wise linear density: 2^n steps of 2^p addresses

Map interval [0, 5] to a 12-entry LUT

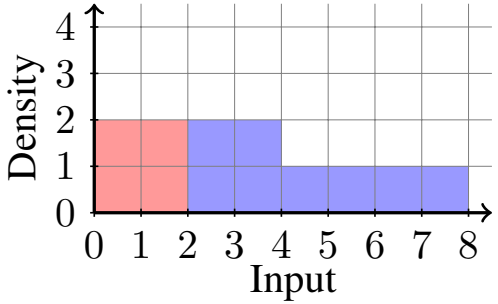


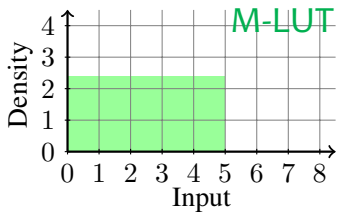


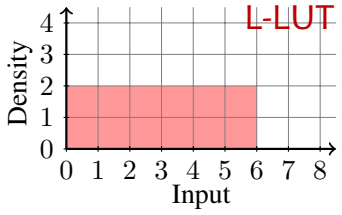


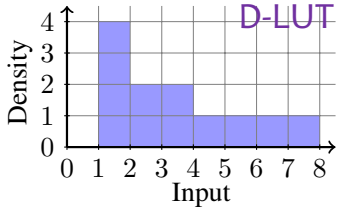
TransPimLib: Combined Methods

- Direct Float Conversion + LDEXP-based LUT (DL-LUT)
  - Uses an L-LUT between 0 and the smallest exponent and a D-LUT for larger inputs









- CORDIC+L-LUT (CORDIC+LUT)
  - Replaces the first few iterations of CORDIC with a LUT
  - Flexible tradeoff between computing cost, table size, and precision

TransPimLib: Supported Functions

	Supported Functions									
Implementation Method	**sin**	**cos**	**tan**	**sinh**	**cosh**	**tanh**	**exp**	**log**	**sqrt**	**GELU**
CORDIC	✓	✓	✓	✓	✓	✓	✓	✓	✓	
M-LUT	✓	✓	✓				✓	✓	✓	
M-LUT+Interp.	✓	✓	✓				✓	✓	✓	
L-LUT	✓	✓	✓				✓	✓	✓	
L-LUT+Interp.	✓	✓	✓				✓	✓	✓	
D-LUT+Interp.	✓					✓				✓
DL-LUT+Interp.	✓					✓				✓
CORDIC+LUT	✓	✓	✓	✓	✓	✓	✓			

Based on our preliminary analysis, we provide the most suitable methods for each of the supported functions (other than sine).

Outline

Processing-in-memory
and transcendental functions

TransPimLib:
A library for transcendental
and other hard-to-calculate functions

Evaluation

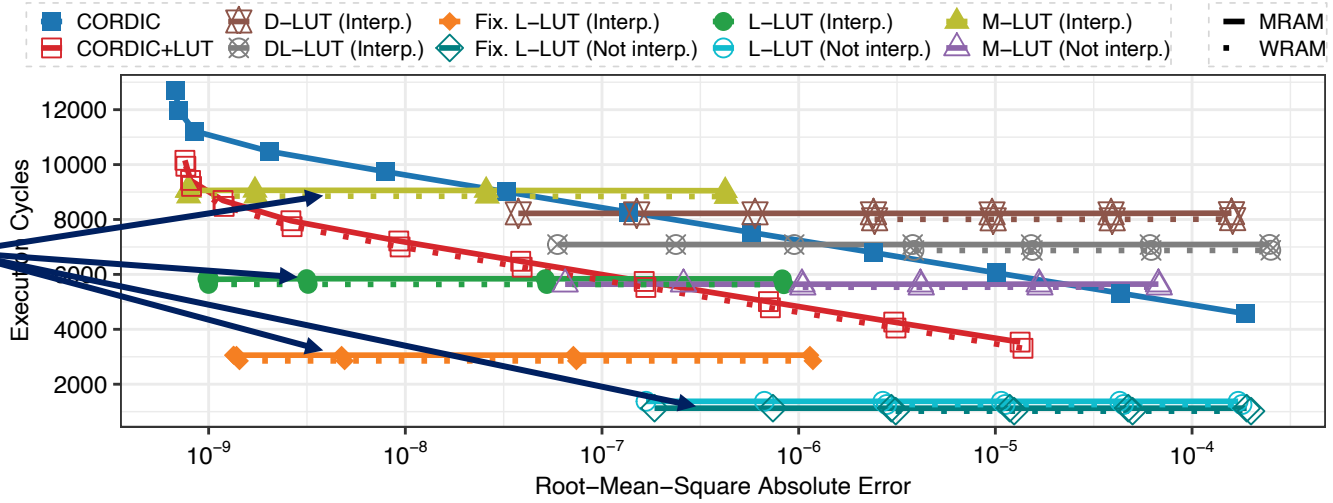
Evaluation Methodology

- Evaluated systems
  - UPMEM PIM system with 2,545 PIM cores @ 350 MHz and 159 GB of DRAM
  - 2-socket Intel Xeon CPU (32 cores)
- Microbenchmarks
  - Performance evaluation
    - We measure execution cycles
  - Accuracy evaluation
    - Root-mean-square absolute error (RMSE) with respect to the CPU with the standard math library
  - Setup time
    - Generation on the host CPU and transfers to the PIM side
  - Memory consumption
    - All tables and variables allocated in the DRAM bank of a PIM core
  - We use sine, as a representative function
- Real-world Benchmarks
  - Blackscholes: exp, log, sqrt, cumulative normal distribution (CNDF)
  - Sigmoid
  - Softmax

Microbenchmark Results: Performance (I)

- We measure the execution cycles for an accuracy range between 10^{-4} and 10^{-9}
- LUT-based versions place the LUT in either the PIM core's DRAM bank (MRAM) or the scratchpad (WRAM)

Performance of LUT-based methods is independent of the accuracy



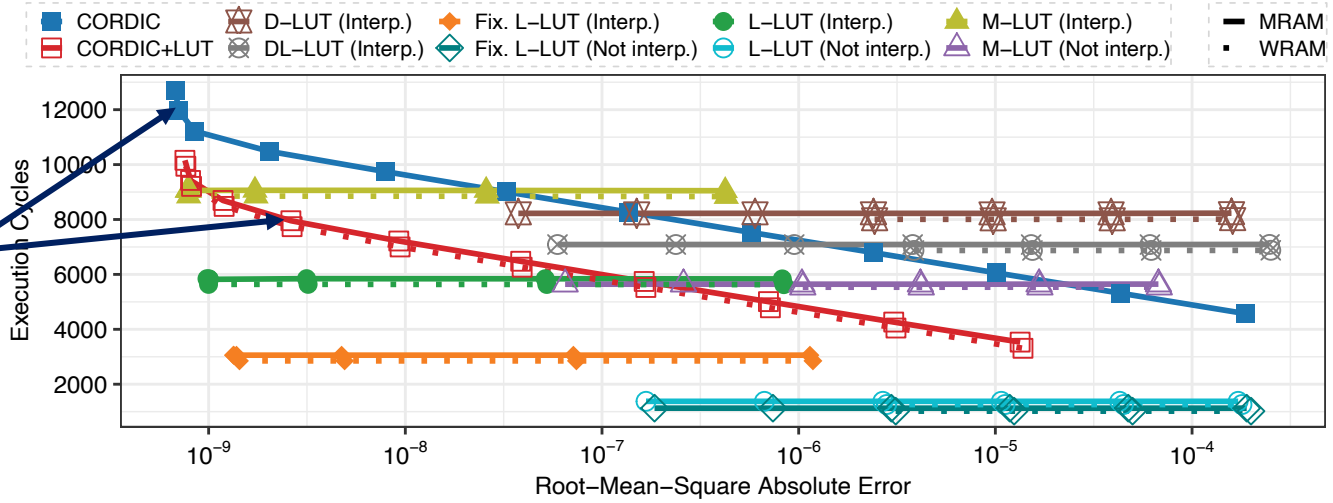
Execution cycles depend on the number of multiplications:
- Interp. M-LUT: 2 FP multiplications
- Non-interp. M-LUT and inter. L-LUT: 1 FP multiplication
- Non-interp. L-LUT: No FP multiplication

Fixed-point version of the L-LUT
- Interp. Fix. L-LUT doubles the performance of inter. L-LUT due to faster fixed-point multiplication

Microbenchmark Results: Performance (II)

- We measure the execution cycles for an accuracy range between 10^{-4} and 10^{-9}
- CORDIC-based methods take more execution cycles to provide higher accuracy

CORDIC accuracy increases with each iteration of the CORDIC algorithm



CORDIC+LUT runs faster than **CORDIC**, as it replaces the initial iterations with an L-LUT query

At some point (~10^{-9}), further increasing the LUT size or CORDIC iterations does not improve accuracy

Little benefit from placing LUTs in the scratchpad (WRAM) instead of the DRAM bank (MRAM)

Microbenchmark Results: Performance (III)

- We measure the execution cycles for an accuracy range between 10^{-4} and 10^{-9}
- CORDIC-based methods take more execution cycles to provide

Key Takeaway 1

Interpolated L-LUT methods (lookup table with LDEXP operation) offer the **best tradeoff** in terms of performance and accuracy

CORDIC+LUT runs faster than CORDIC, as it replaces the initial iterations with an L-LUT query

At some point (~10^{-9}), further increasing the LUT size or CORDIC iterations does not improve accuracy

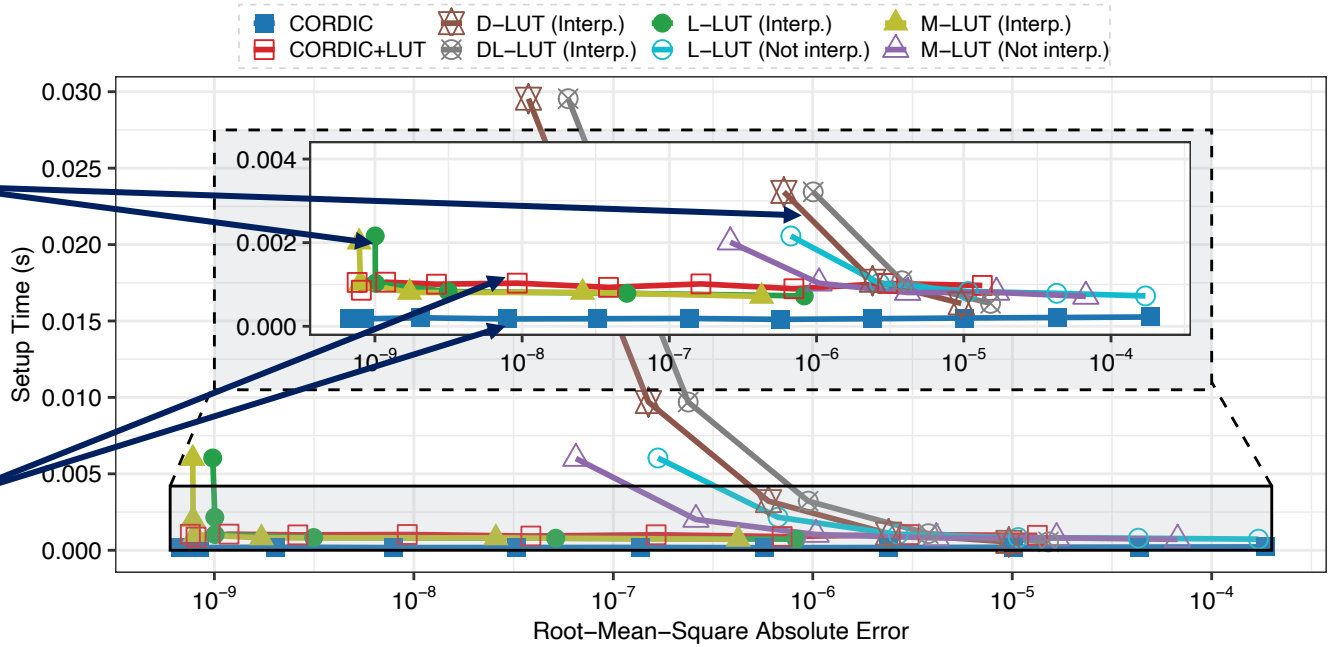
Little benefit from placing LUTs in the scratchpad (WRAM) instead of the DRAM bank (MRAM)

Microbenchmark Results: Setup Time (I)

- The setup time can also impact the decision of what method to use

For LUT-based methods, setup times increase with LUT size

CORDIC methods have flat setup times



CORDIC methods can provide higher overall performance
(i.e., setup time + PIM kernel time) than LUT-based methods
when the total number of transcendental functions in a workload is low.
For example, we estimate ~40 sine operations (see paper)

Microbenchmark Results: Setup Time (II)

- The setup time can also impact the decision of what method to use

Key Takeaway 2

CORDIC-based methods are preferable when a PIM kernel needs to execute **just a few transcendental functions** due to their low setup time in the host CPU

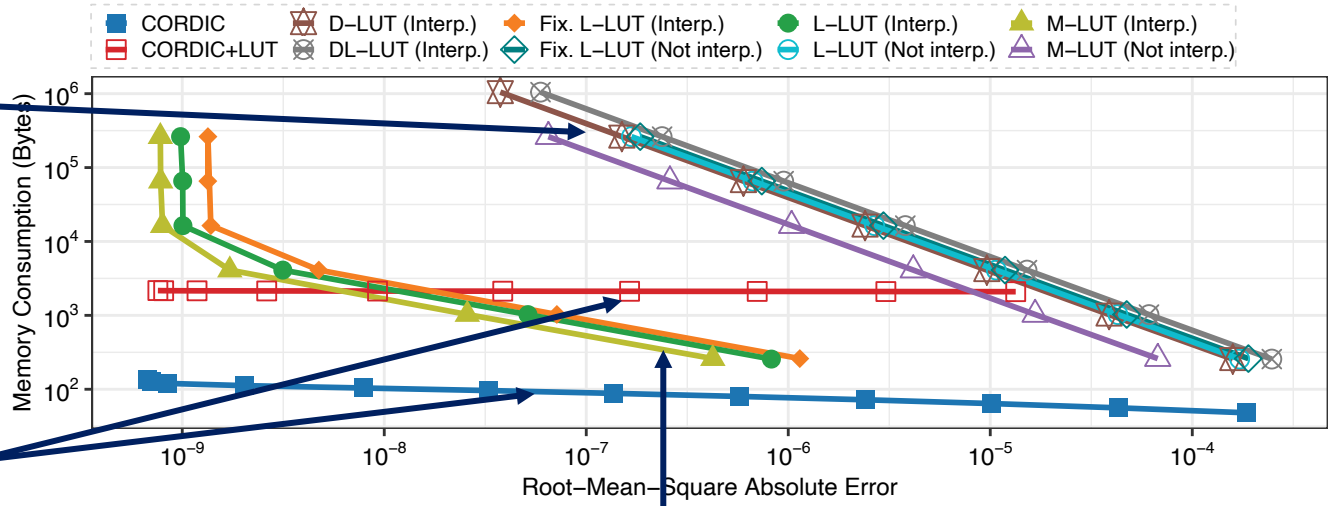
CORDIC methods can provide higher overall performance
(i.e., setup time + PIM kernel time) than LUT-based methods
when the total number of transcendental functions in a workload is low.
For example, we estimate ~40 sine operations (see paper)

Microbenchmark Results: Memory (I)

- We also obtain the memory consumption (in bytes) in the DRAM bank of a PIM core

Accuracy of non-interp. LUT methods is limited by the available memory

Memory consumption of CORDIC methods does *not* increase exponentially



Interpolation is an effective way of increasing accuracy without increasing LUT size

Microbenchmark Results: Memory (II)

- We also obtain the memory consumption (in bytes) in

Key Takeaway 3

Interpolated L-LUT methods offer a good tradeoff in terms of accuracy, execution cycles, and memory consumption.

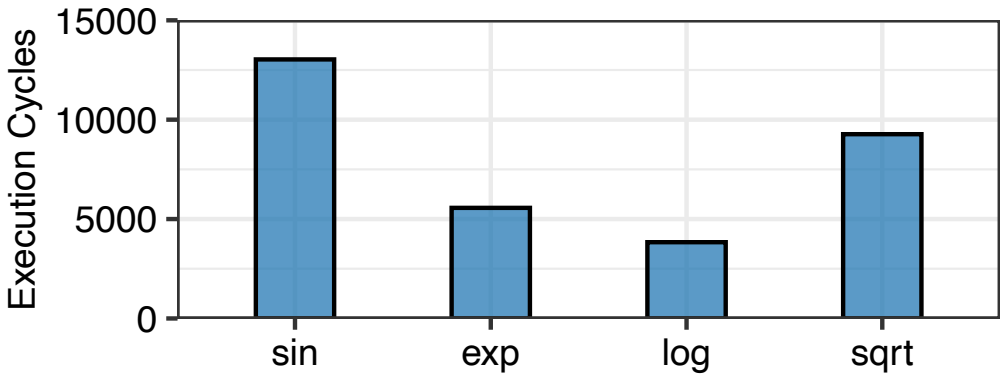
However, **CORDIC and CORDIC+LUT methods** are recommended for applications that require high accuracy, where the available memory is limited (e.g., needed for large datasets)

Other Supported Functions (I)

- The general trends for other functions supported by TransPimLib are similar to those of the sine function
- Some major differences:
  1. Tangent calculation takes around 2-3 times more cycles than sine calculation, as it requires
     a) Calculation of sine and cosine
     b) A floating-point division
  2. Some supported functions require range reduction and/or range extension

a) The cost differs between functions, as it depends on specific mathematical identity needed for the conversion

b) But range reduction/extension is only necessary depending on the actual range of input values



Other Supported Functions (II)

- Some major differences:
  3. Activation functions *tanh* and GELU do not require range reduction/extension and are approximately linear in most parts

Key Takeaway 4

D-LUT and DL-LUT methods are well-suited for activation functions, such as *tanh* and *GELU*, which (1) do not require range extension, and (2) are approximately linear in most parts.

D-LUT and DL-LUT are faster than interpolated L-LUT, while providing similar accuracy

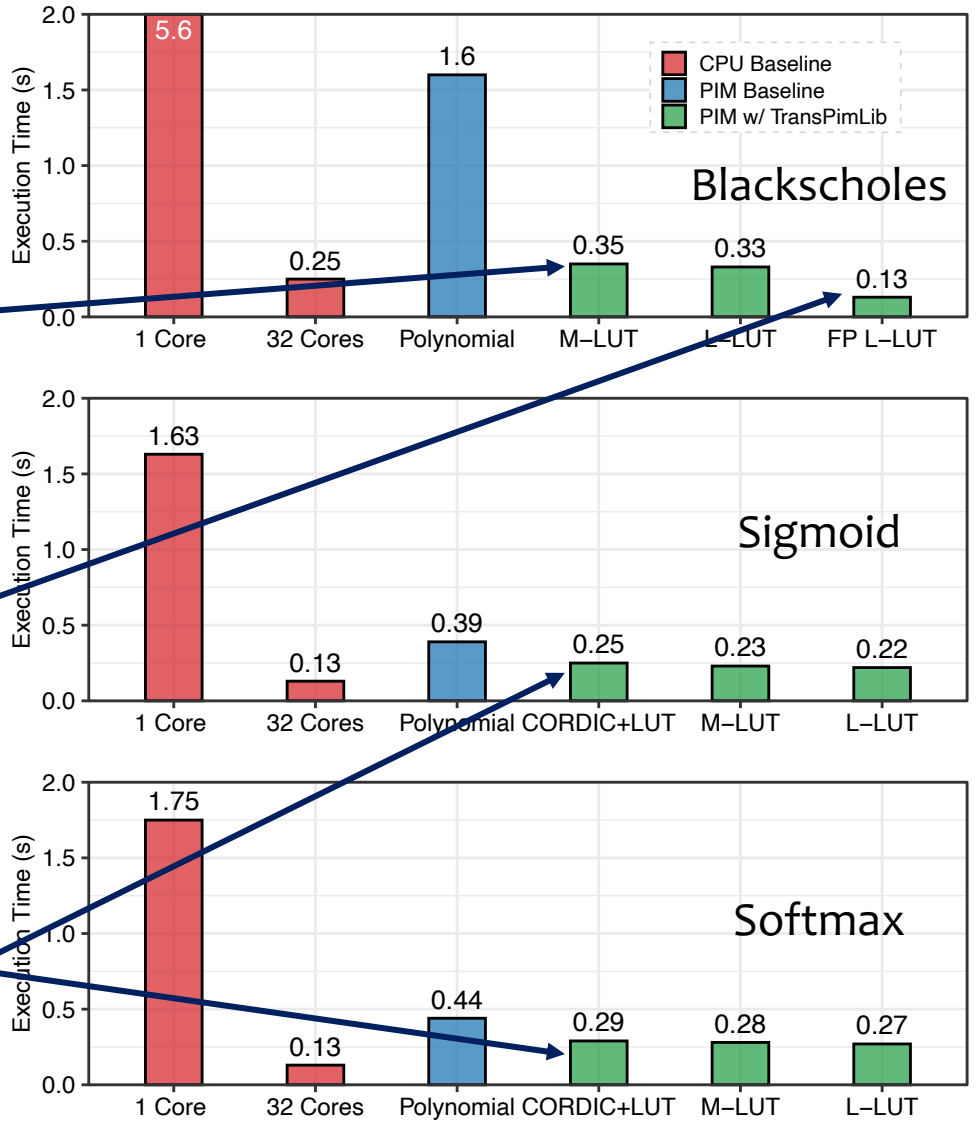
Real-world Benchmark Results (I)

- 1 & 32 CPU cores
- PIM baseline: Polynomial

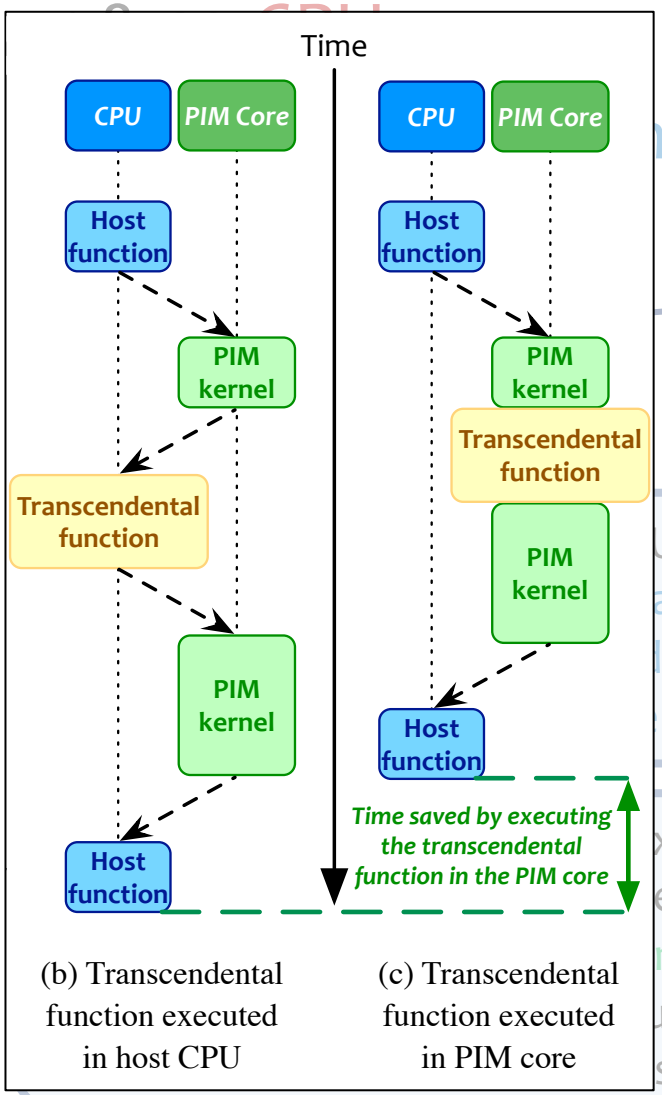
For Blackscholes, TransPimLib is 5-12x faster than the PIM baseline

Fixed-point L-LUT is 92% faster than the 32-thread CPU baseline

For Sigmoid and Softmax, TransPimLib outperforms the PIM baseline and shows that it can save data movement from executing activation functions in the host CPU



Real-world Benchmark Results (II)



Key Takeaway 5

TransPimLib can **reduce data movement from PIM cores to the CPU** (Fig. (b)) for applications running on the PIM cores.

As a result, the execution of transcendental functions in the PIM cores (Fig. (c)) could be 6–8× faster than the execution in the host CPU

More in the Paper

- Background on CORDIC and Fuzzy Lookup Tables
- How to use TransPimLib (APIs)
- Additional observations and takeaways

TransPimLib: Efficient Transcendental Functions for Processing-in-Memory Systems

Maurus Item Juan Gómez-Luna Yuxin Guo
Geraldo F. Oliveira Mohammad Sadrosadati Onur Mutlu
ETH Zürich

https://people.inf.ethz.ch/omutlu/pub/TransPIMLib_ispass23.pdf
https://arxiv.org/pdf/2304.01951.pdf

TransPimLib: arXiv Version

TransPimLib: A Library for Efficient Transcendental Functions on Processing-in-Memory Systems

Maurus Item Juan Gómez-Luna Yuxin Guo
Geraldo F. Oliveira Mohammad Sadrosadati Onur Mutlu
ETH Zürich

https://arxiv.org/pdf/2304.01951.pdf

Source code: https://github.com/CMU-SAFARI/transpimlib

https://youtu.be/lqqf4eaaEE4

ISPASS 2023 Version

- Presented at ISPASS 2023

TransPimLib: Efficient Transcendental Functions for Processing-in-Memory Systems

Maurus Item Juan Gómez-Luna Yuxin Guo
Geraldo F. Oliveira Mohammad Sadrosadati Onur Mutlu
ETH Zürich

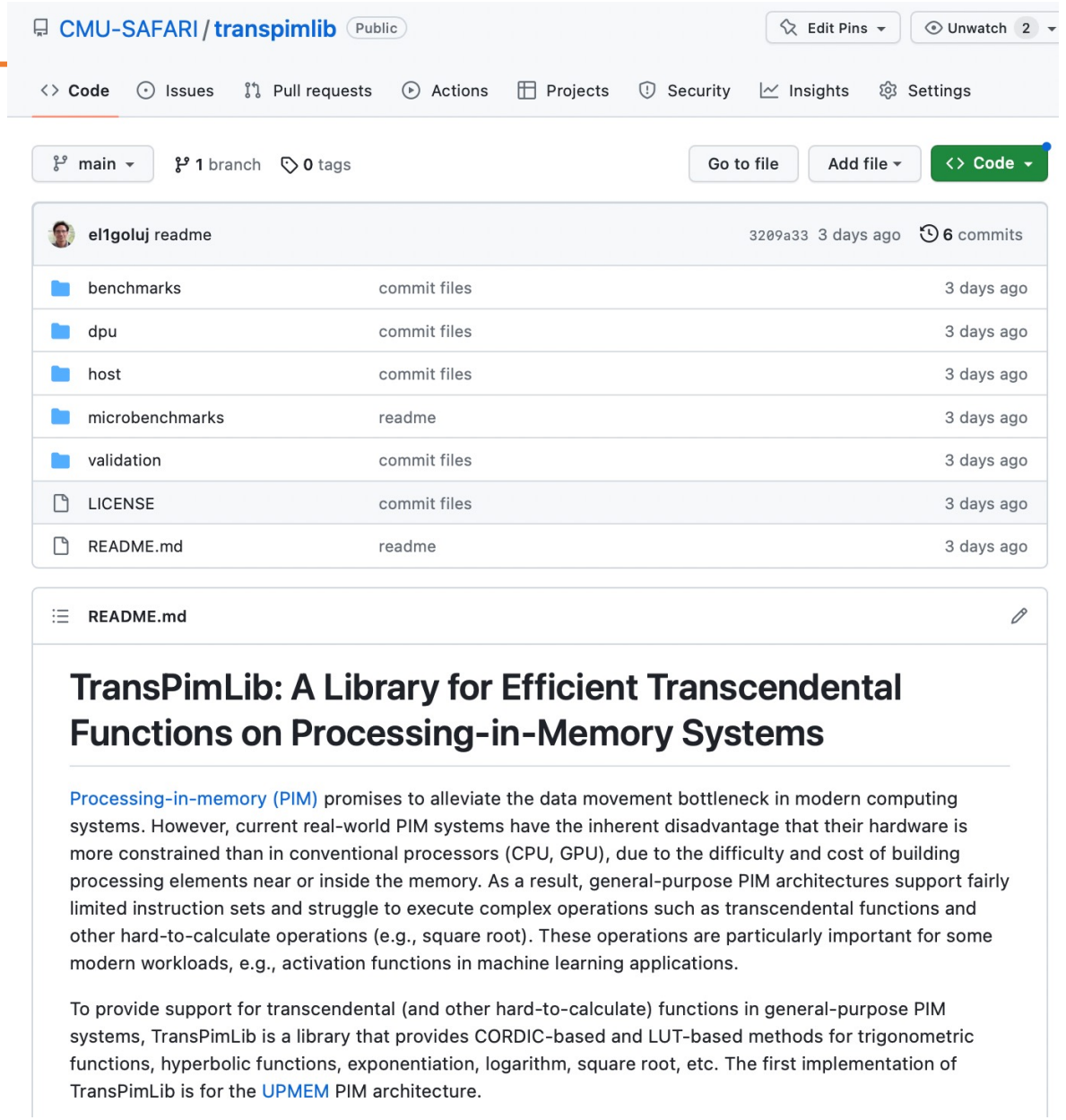
https://people.inf.ethz.ch/omutlu/pub/TransPIMLib_ispass23.pdf

Source code: https://github.com/CMU-SAFARI/transpimlib

https://youtu.be/lqqf4eaaEE4

Source Code

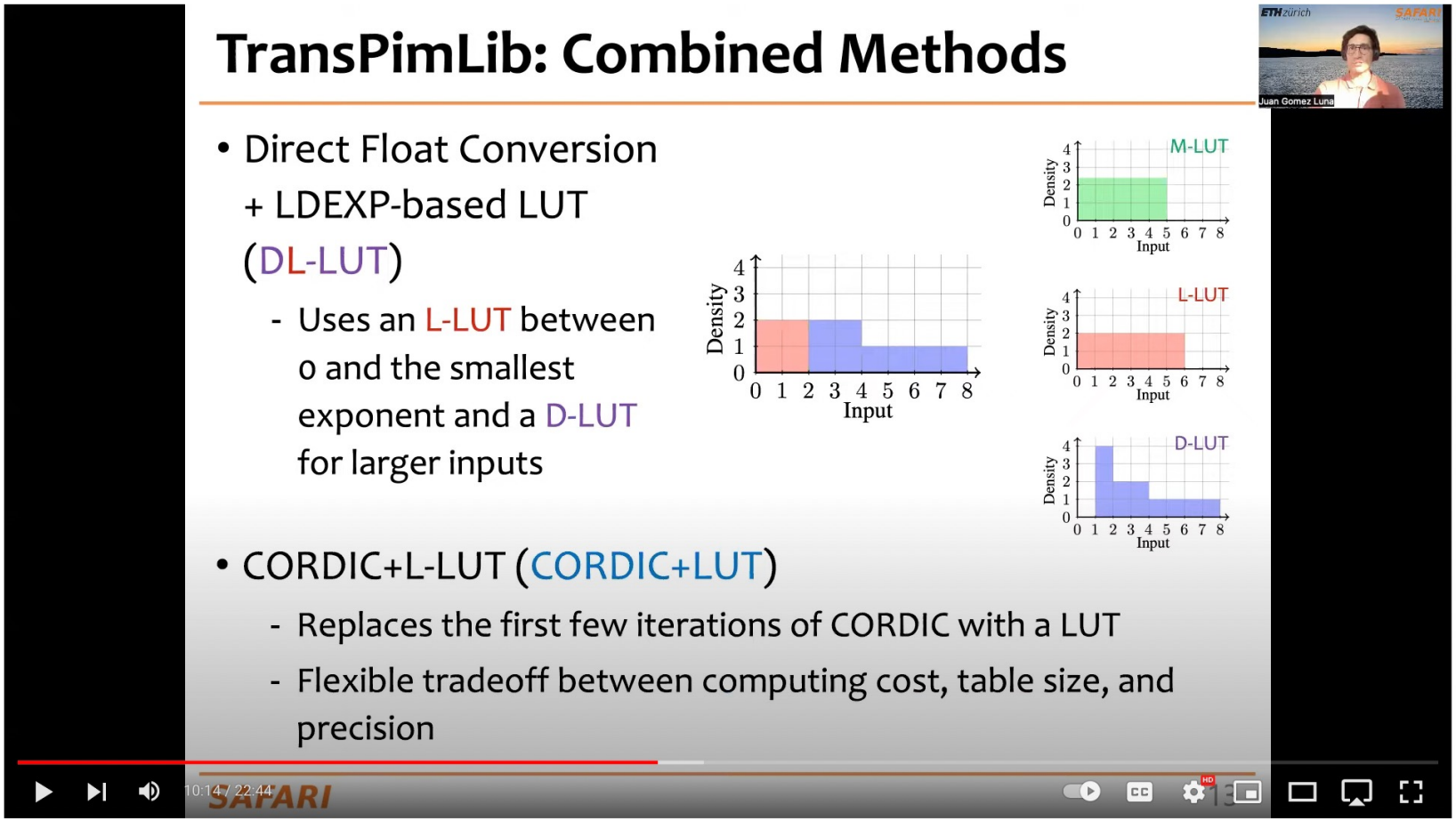
- https://github.com/CMU-SAFARI/transpimlib



Executive Summary

- Processing-in-Memory (PIM) promises to alleviate the *data movement bottleneck*
- However, current real-world PIM systems have very constrained hardware, which results in limited instruction sets
  - Difficulty/impossibility of computing complex operations, such as transcendental functions (e.g., trigonometric, exp, log) and other hard-to-calculate functions (e.g., square root)
  - These functions are important for modern workloads, e.g., activation functions in machine learning applications
- TransPimLib is the first library for transcendental and other hard-to-calculate functions on general-purpose PIM systems
  - CORDIC-based and LUT-based methods for trigonometric functions, hyperbolic functions, exponentiation, logarithm, square root, etc.
  - Source code: https://github.com/CMU-SAFARI/transpimlib
- We implement TransPimLib for the UPMEM PIM architecture and evaluate its methods in terms of performance, accuracy, memory requirements, and setup time
  - Three real workloads (Blackscholes, Sigmoid, Softmax)

Lecture on TransPimLib



PIM Course: Lecture 13: Efficient Transcendental Functions on PIM (Spring 2023)

Onur Mutlu Lectures
33.4K subscribers
Subscribed
7
Share
Clip
Save

148 views 12 days ago Livestream - Data-Centric Architectures: Fundamentally Improving Performance and Energy (Spring 2023)
Projects & Seminars, ETH Zürich, Spring 2023
Data-Centric Architectures: Fundamentally Improving Performance and Energy

TransPimLib:

Efficient Transcendental Functions for Processing-in-Memory Systems

Maurus Item, Juan Gómez Luna, Yuxin Guo,
Geraldo F. Oliveira, Mohammad Sadrosadati, Onur Mutlu

https://arxiv.org/pdf/2304.01951.pdf
https://github.com/CMU-SAFARI/transpimlib
juang@ethz.ch

ETH Zürich

SAFARI

Thursday, June 1, 2023

Homomorphic Operations

Evaluating Homomorphic Operations on a Real-World Processing-In-Memory System

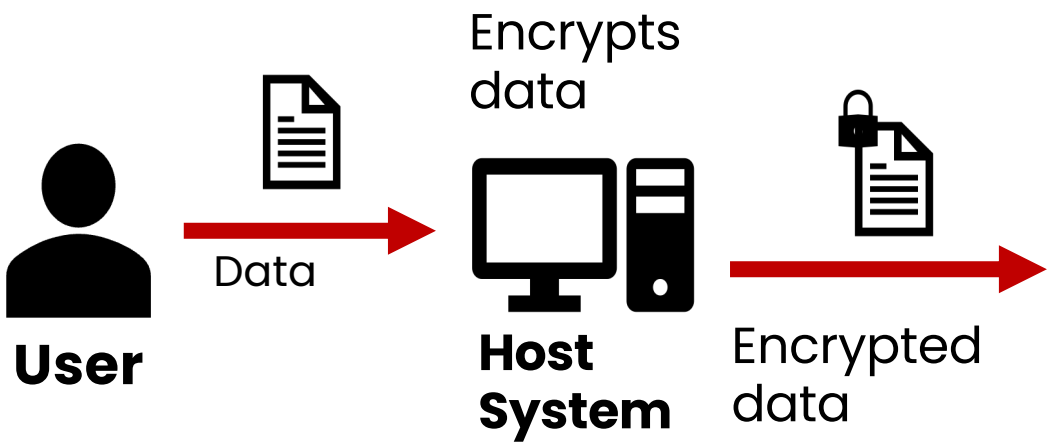
Harshita Gupta* Mayank Kabra*
Juan Gómez-Luna Konstantinos Kanellopoulos
Onur Mutlu

SAFARI

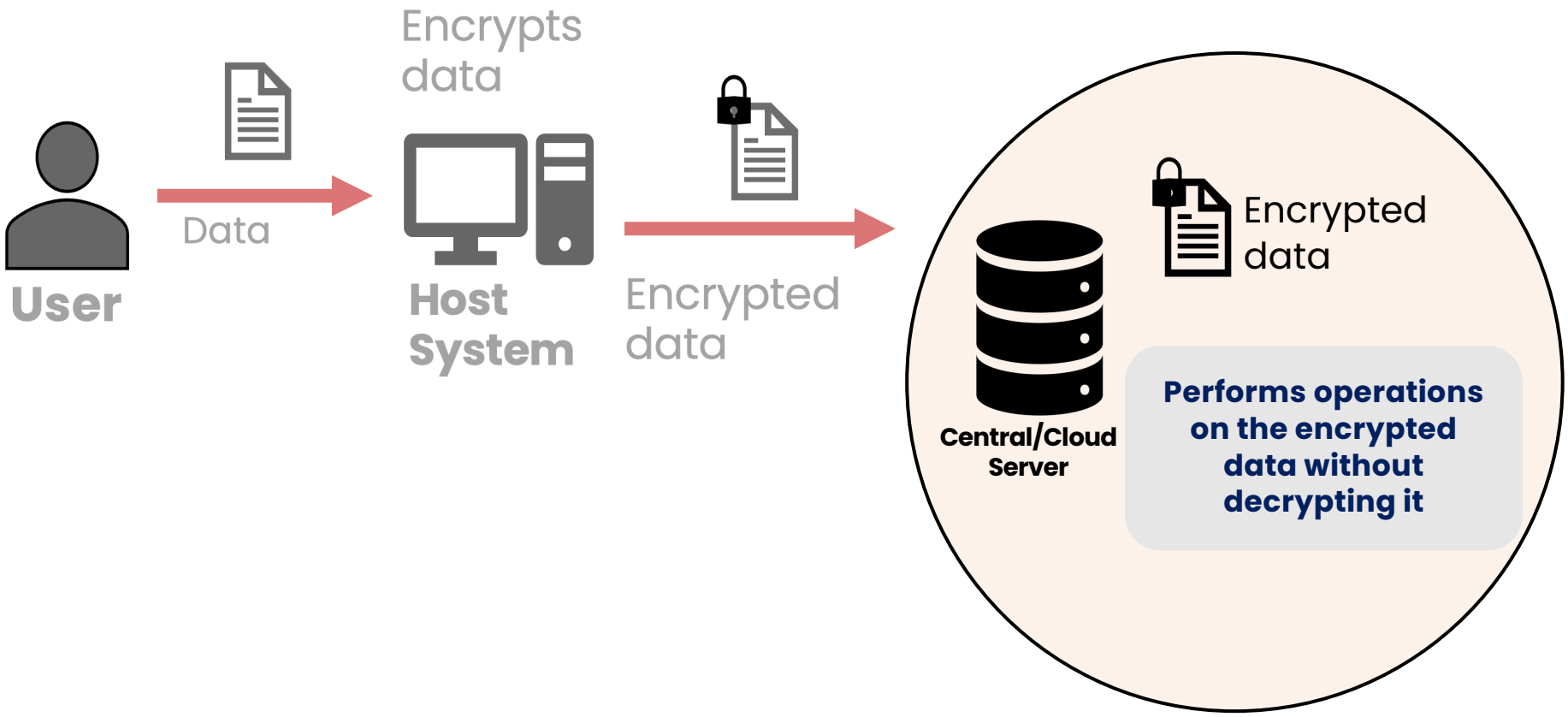
ETH zürich

Homomorphic Encryption

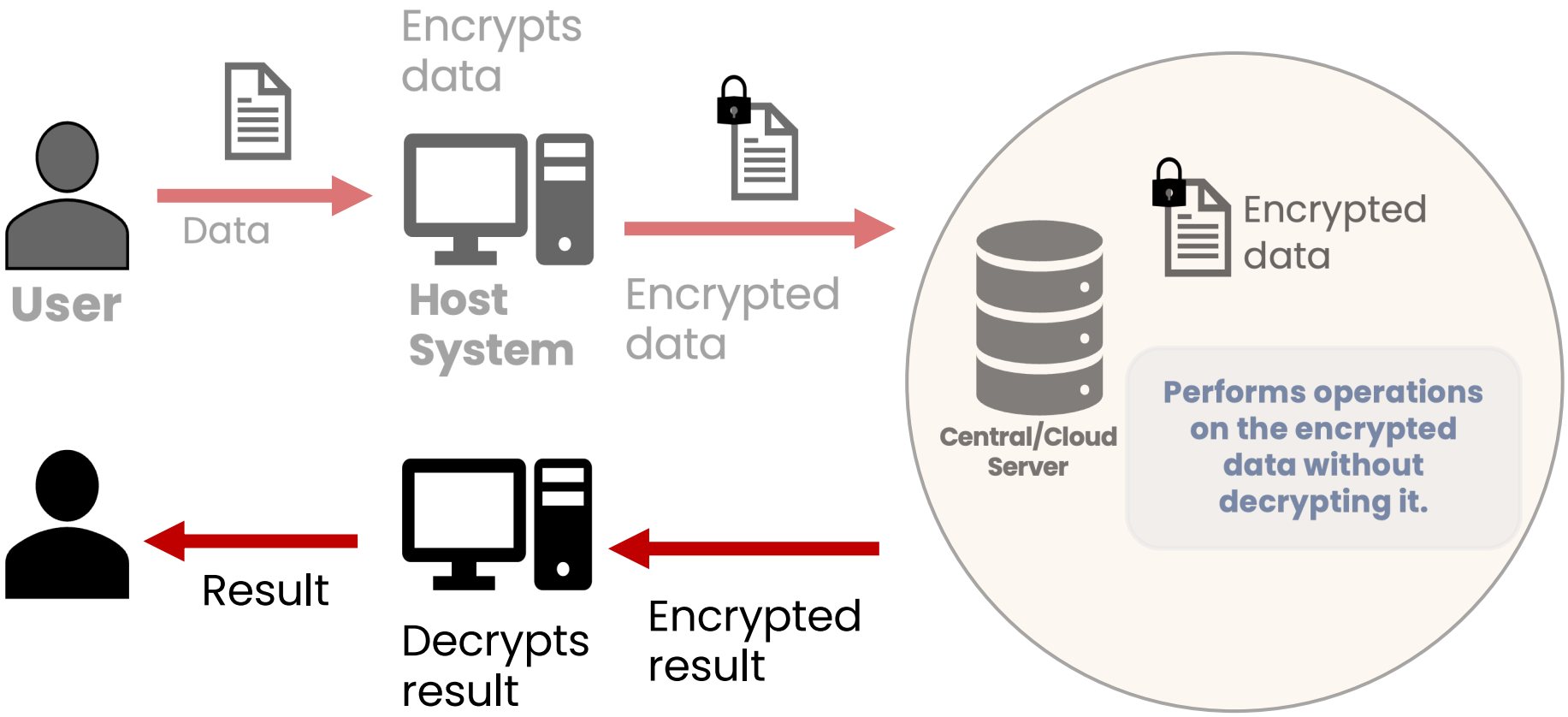
Homomorphic Encryption



Homomorphic Encryption



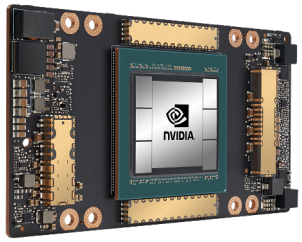
Homomorphic Encryption

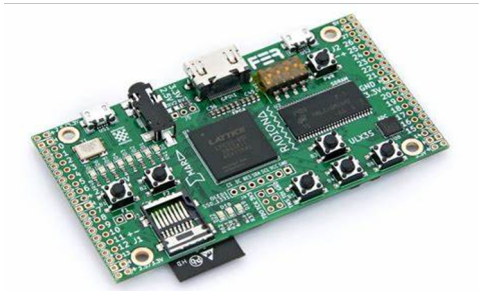


Motivation

Homomorphic operations suffer from
large memory capacity and data movement bottlenecks

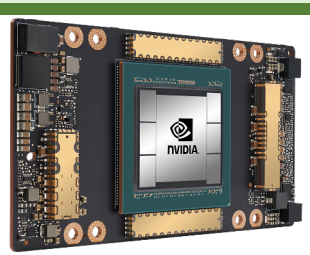
Acceleration Techniques





Motivation

These approaches face challenges in **resource limitations, data movement, and practical implementation**

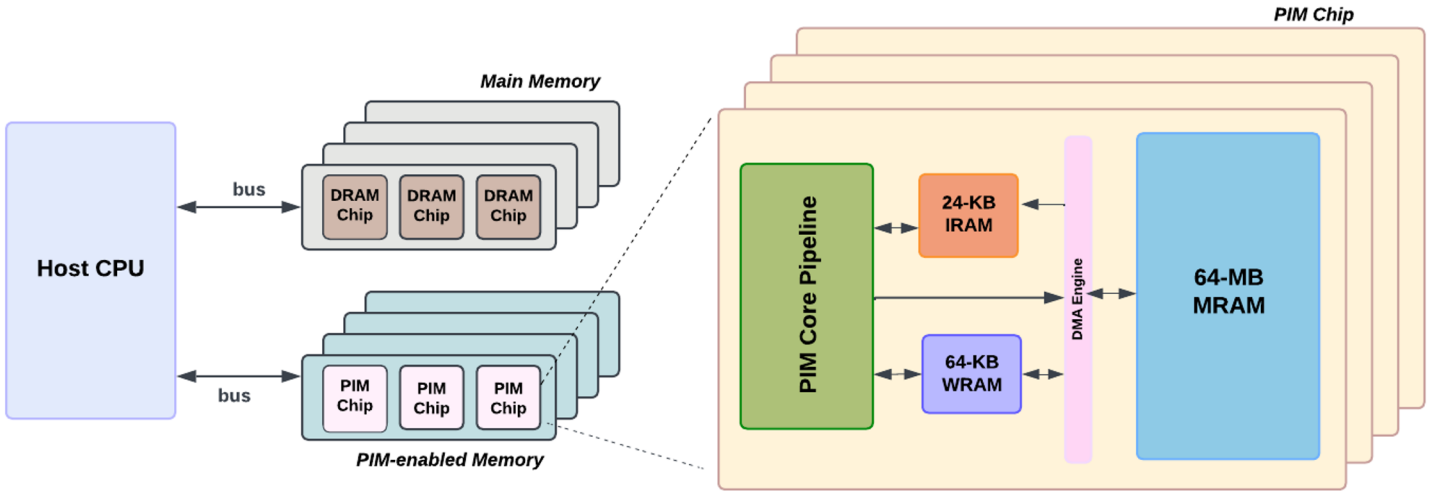


Our Goal

Evaluate the suitability of real-world general-purpose processing-in-memory (PIM) architectures to perform homomorphic operations.

Our Goal

Evaluate the suitability of real-world general-purpose processing-in-memory (PIM) architectures to perform homomorphic operations.

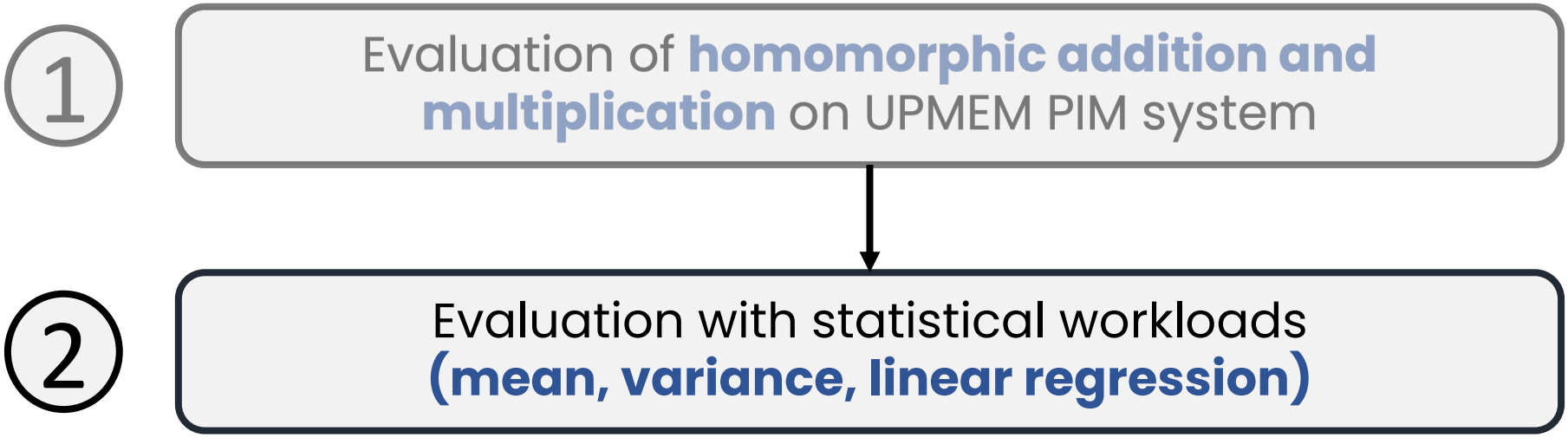


UPMEM: First Real World PIM Architecture

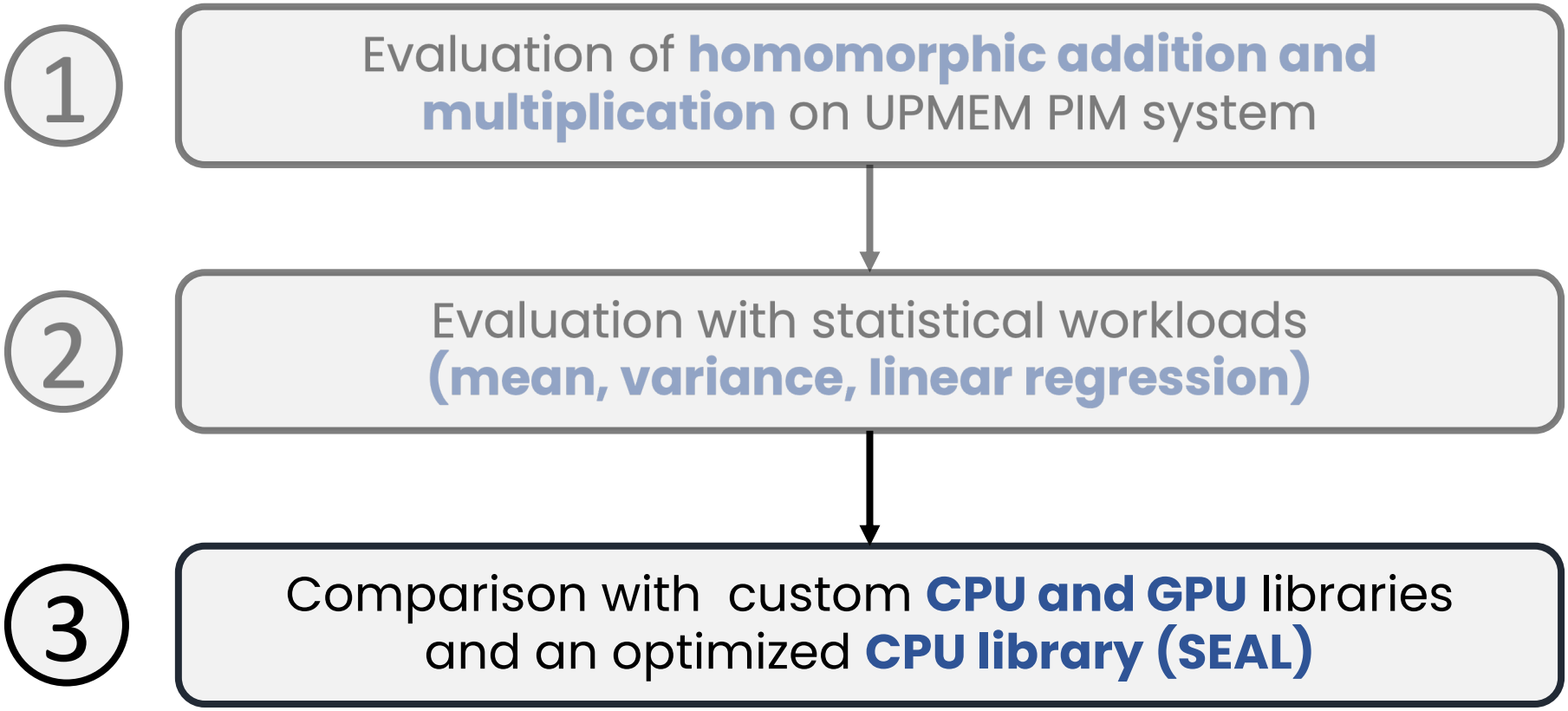
Evaluation Methodology

① Evaluation of **homomorphic addition and multiplication** on UPMEM PIM system

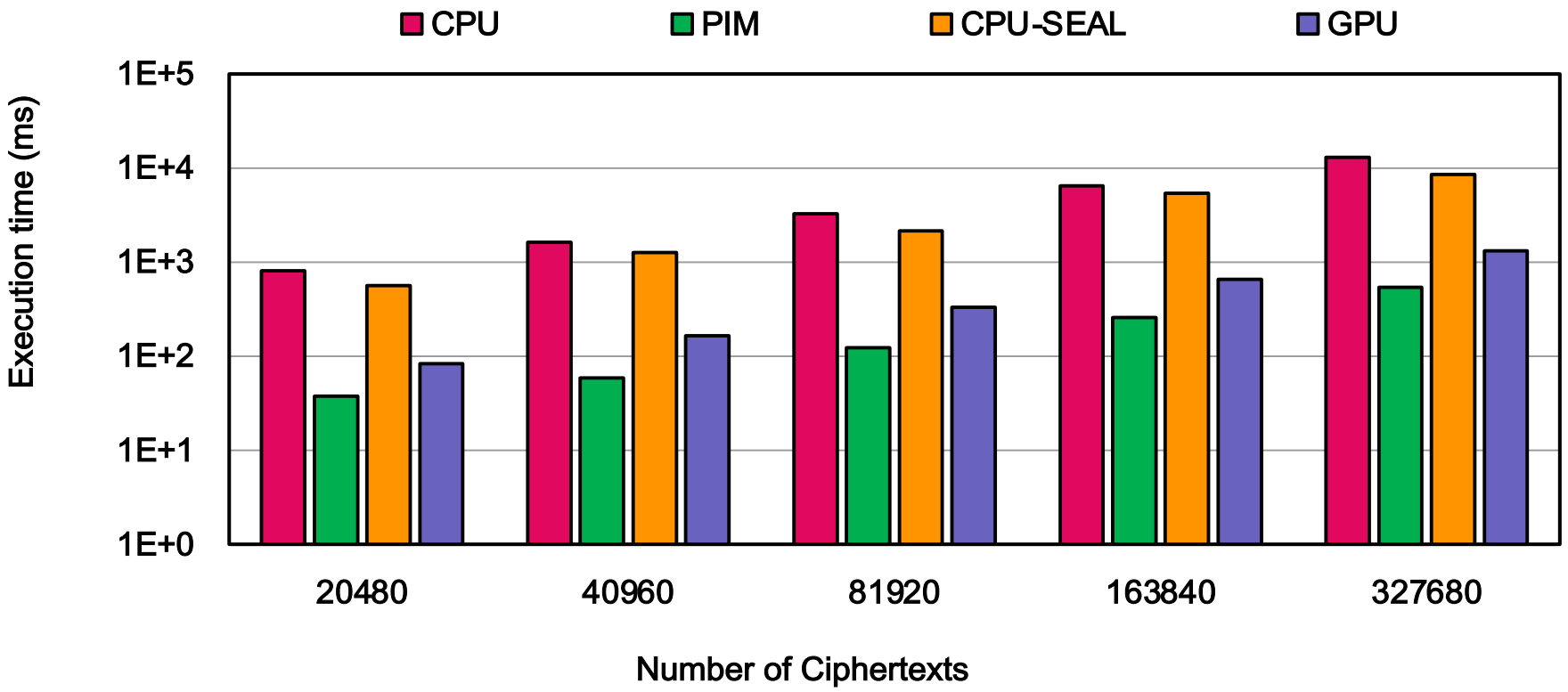
Evaluation Methodology



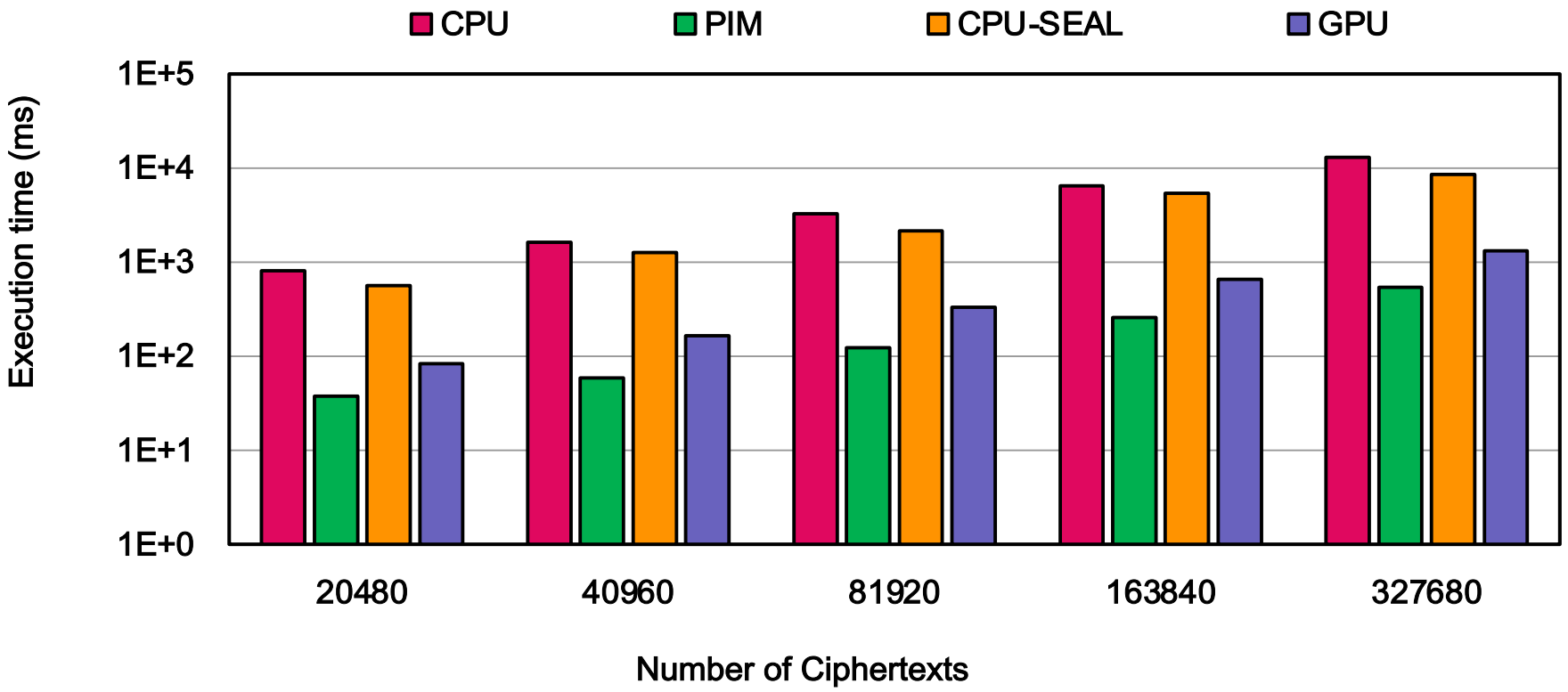
Evaluation Methodology



Evaluation: Homomorphic Addition



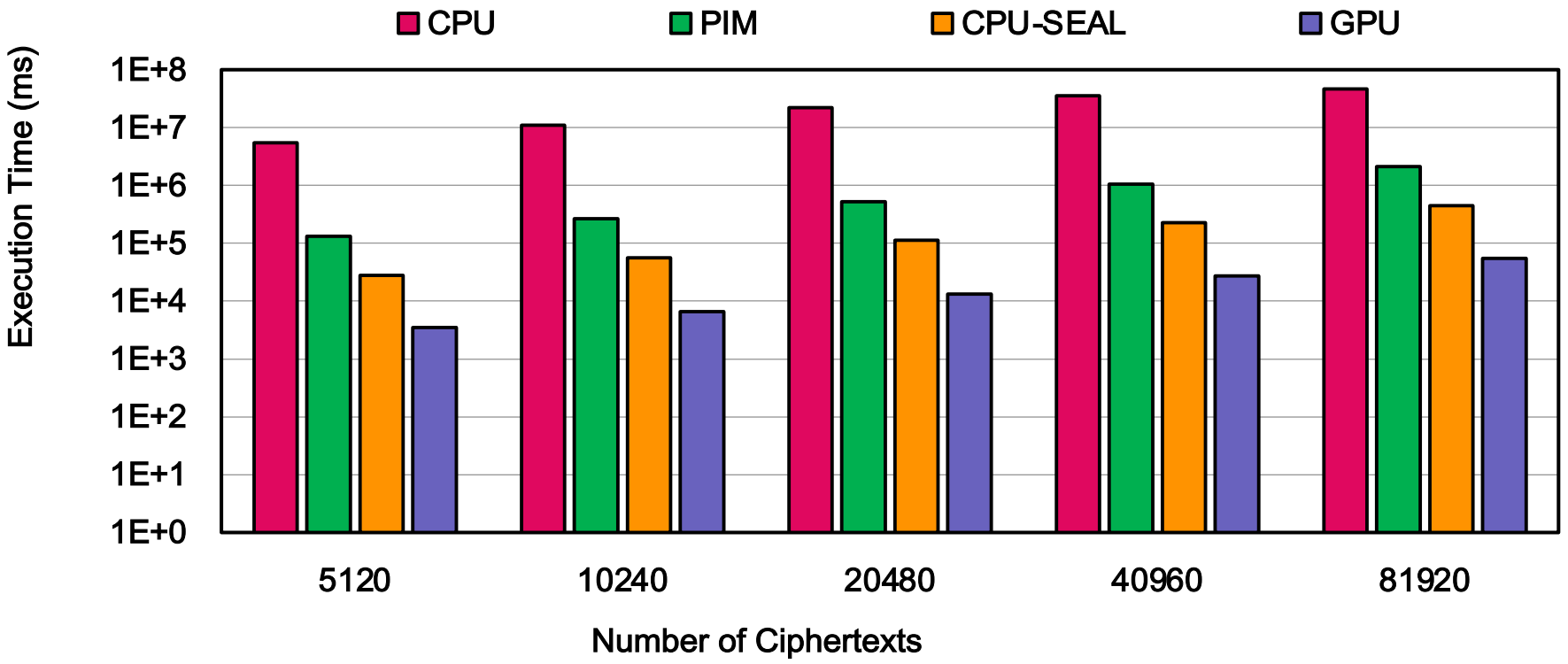
Evaluation: Homomorphic Addition



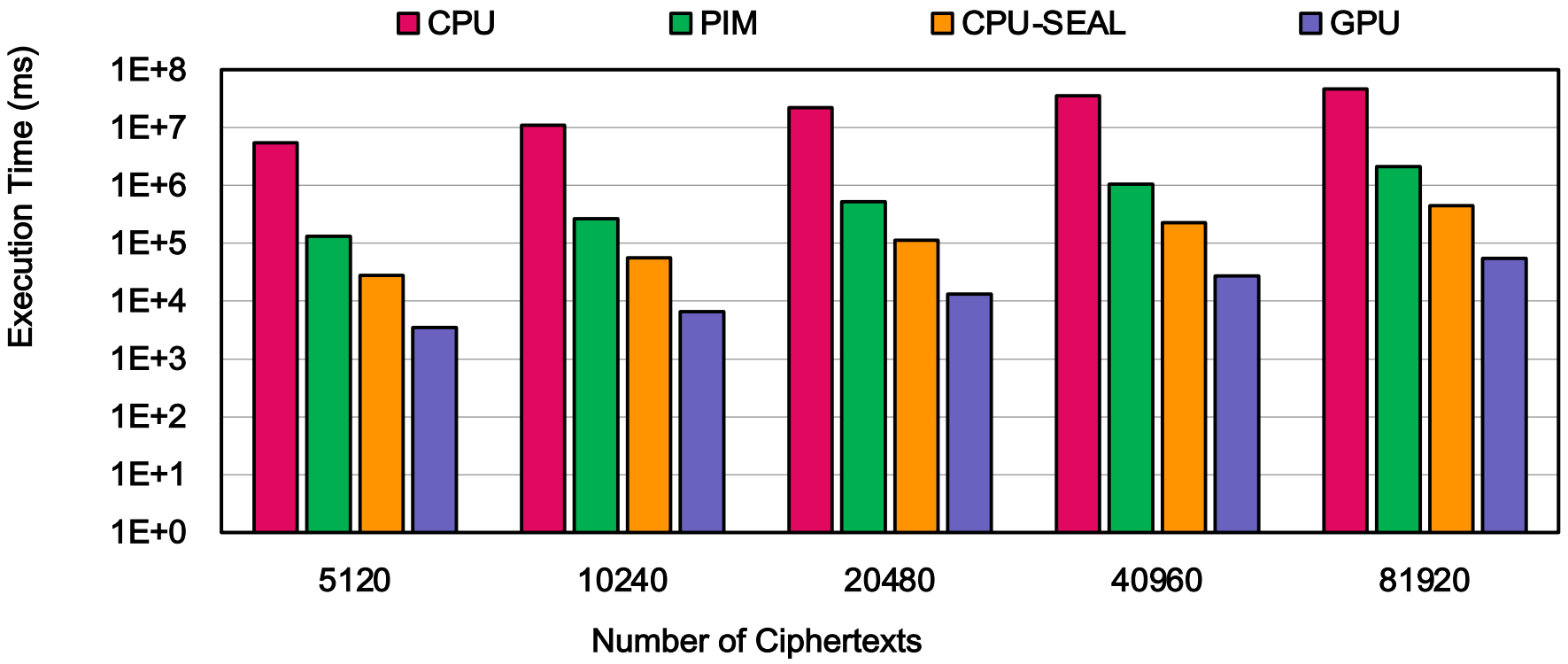
50 – 100× speedup provided by PIM over CPU

2 – 15× speedup over GPU

Evaluation: Homomorphic Multiplication

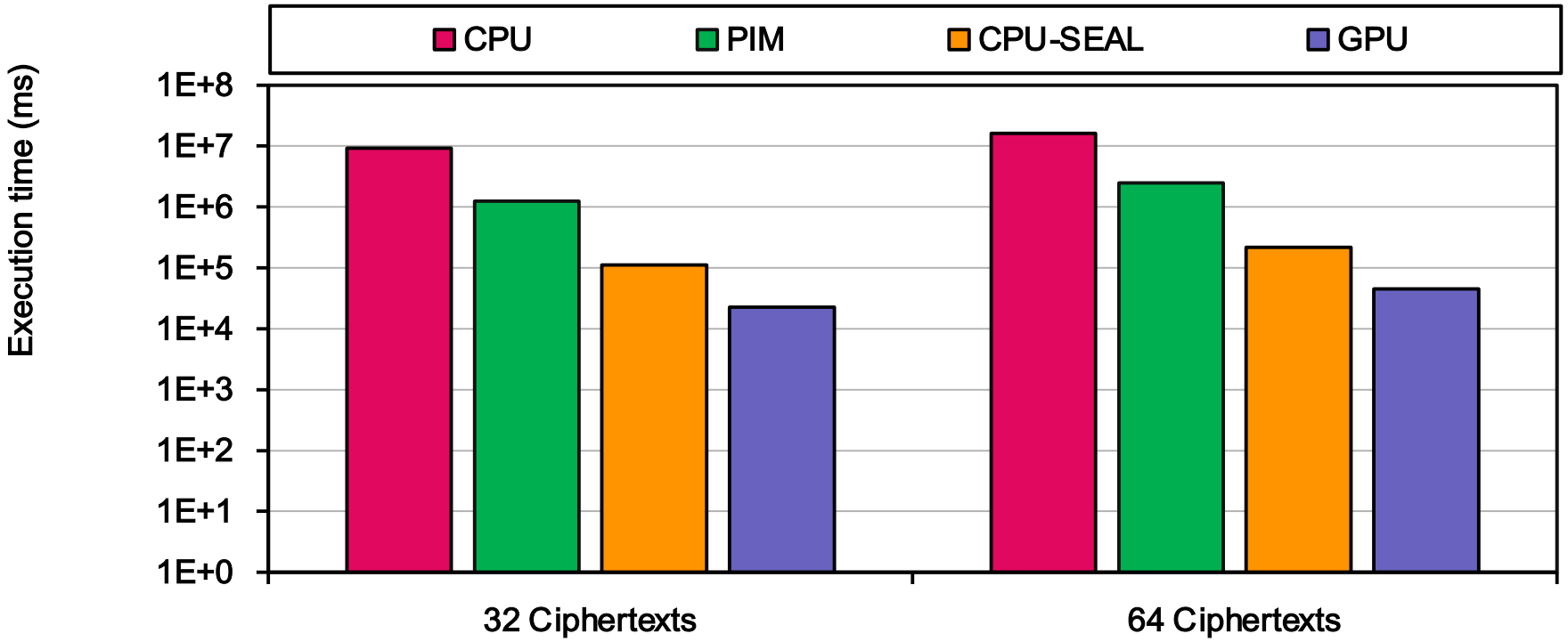


Evaluation: Homomorphic Multiplication

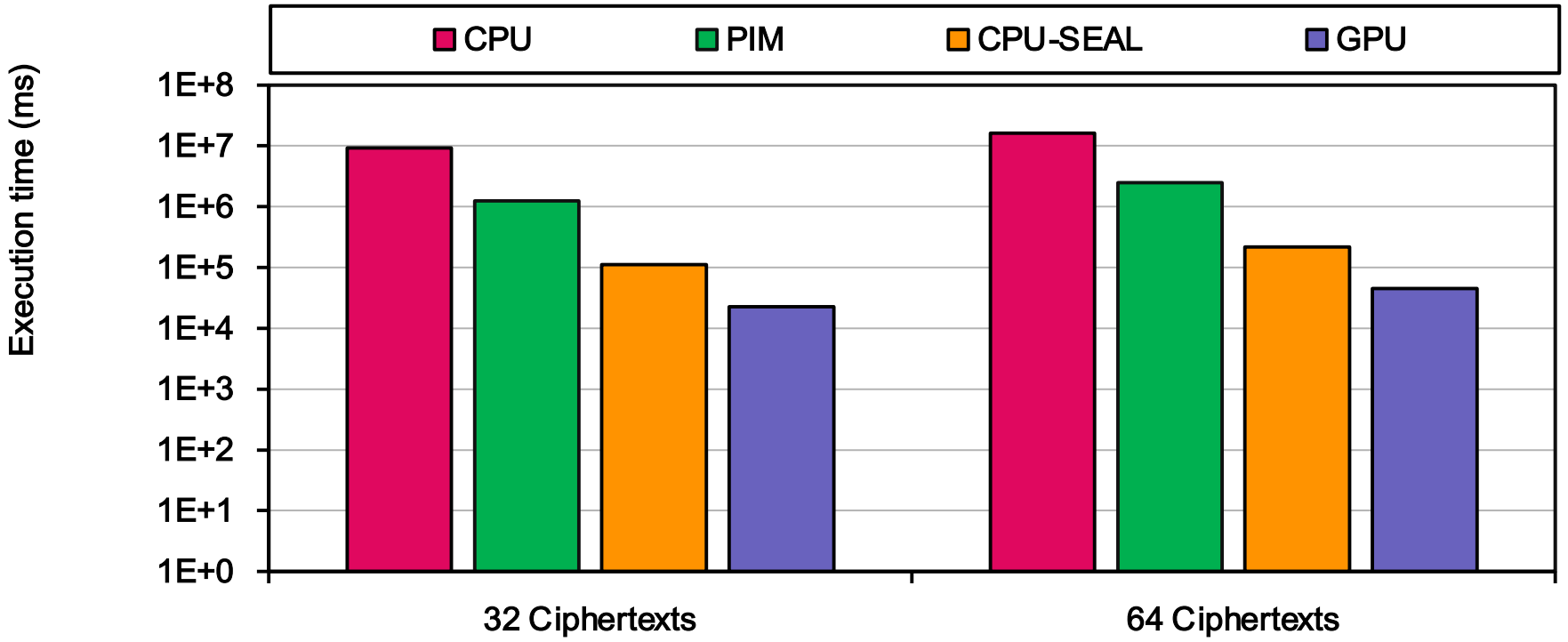


PIM lags 10 – 15× behind the GPU due to the lack of native multiplication support

Evaluation: Linear Regression



Evaluation: Linear Regression



PIM is 6.4–7.5x faster than the custom CPU implementation

CPU-SEAL and GPU are faster than PIM

Key Takeaways

① With **native hardware support for 32-bit integer addition** and large number of PIM cores, the **UPMEM PIM system outperforms** CPU and GPU for homomorphic addition.

Key Takeaways

① With **native hardware support for 32-bit integer addition** and large number of PIM cores, the **UPMEM PIM system outperforms** CPU and GPU for homomorphic addition.

② The **lack of native support for 32-bit integer multiplication hampers the performance** of PIM for homomorphic multiplication.

Key Takeaways

1. With **native hardware support for 32-bit integer addition** and large number of PIM cores, the **UPMEM PIM system outperforms** CPU and GPU for homomorphic addition.

2. The **lack of native support for 32-bit integer multiplication hampers the performance** of PIM for homomorphic multiplication.

3. The **computational power of PIM scales with memory capacity** via the addition of more memory banks and PIM cores.

Executive Summary

PROBLEM

Homomorphic encryption (HE) enables computation on encrypted data, however it suffers from **memory usage and data movement bottlenecks**.

Executive Summary

PROBLEM

Homomorphic encryption (HE) enables computation on encrypted data, however it suffers from **memory usage and data movement bottlenecks**.

GOAL

Our **goal** is to extensively evaluate **homomorphic operations** on real-world general-purpose processing-in-memory architectures.

Executive Summary

PROBLEM

Homomorphic encryption (HE) enables computation on encrypted data, however it suffers from **memory usage and data movement bottlenecks**.

GOAL

Our **goal** is to extensively evaluate **homomorphic operations** on real-world general-purpose processing-in-memory architectures.

CONTRIBUTIONS

1. **Novel Implementation:** First to implement homomorphic addition and multiplication on a real PIM system.
2. **Performance Evaluation:** Evaluated the performance of PIM across different HE security levels with real-world workloads: (i) mean, (ii) variance, (iii) linear regression.
3. **Key Insights:** Revealed PIM system capabilities and trade-offs for cryptography.

Executive Summary

PROBLEM

Homomorphic encryption (HE) enables computation on encrypted data, however it suffers from **memory usage and data movement bottlenecks**.

GOAL

Our **goal** is to extensively evaluate **homomorphic operations** on real-world general-purpose processing-in-memory architectures.

CONTRIBUTIONS

1. **Novel Implementation:** First to implement homomorphic addition and multiplication on a real PIM system.
2. **Performance Evaluation:** Evaluated the performance of PIM across different HE security levels with real-world workloads: (i) mean, (ii) variance, (iii) linear regression.
3. **Key Insights:** Revealed PIM system capabilities and trade-offs for cryptography.

EVALUATIONS

1. PIM is **50–100×** faster than CPU and **2–15×** faster than GPU in vector addition.
2. PIM outperforms CPU by **40–50×** in vector multiplication but lags **10–15×** behind GPU.
3. For statistical operations, PIM achieves **30× to 300×** improvement over CPU and **10× to 30×** over GPU.

Evaluating Homomorphic Operations on a Real-World Processing-In-Memory System

Harshita Gupta* Mayank Kabra*
Juan Gómez-Luna Konstantinos Kanellopoulos
Onur Mutlu

SAFARI

ETH zürich

MICRO 2023 Tutorial

Real-world Processing-in-Memory Systems for Modern Workloads

Accelerating Modern Workloads

on a General-purpose PIM System

Dr. Juan Gómez Luna

Professor Onur Mutlu

ETH zürich

SAFARI

Sunday, October 29, 2023