Understanding a Modern Processing-in-Memory Architecture:

Benchmarking and Experimental Characterization

Juan Gómez Luna, Izzat El Hajj,
Ivan Fernandez, Christina Giannoula,
Geraldo F. Oliveira, Onur Mutlu

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

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

ETH Zürich

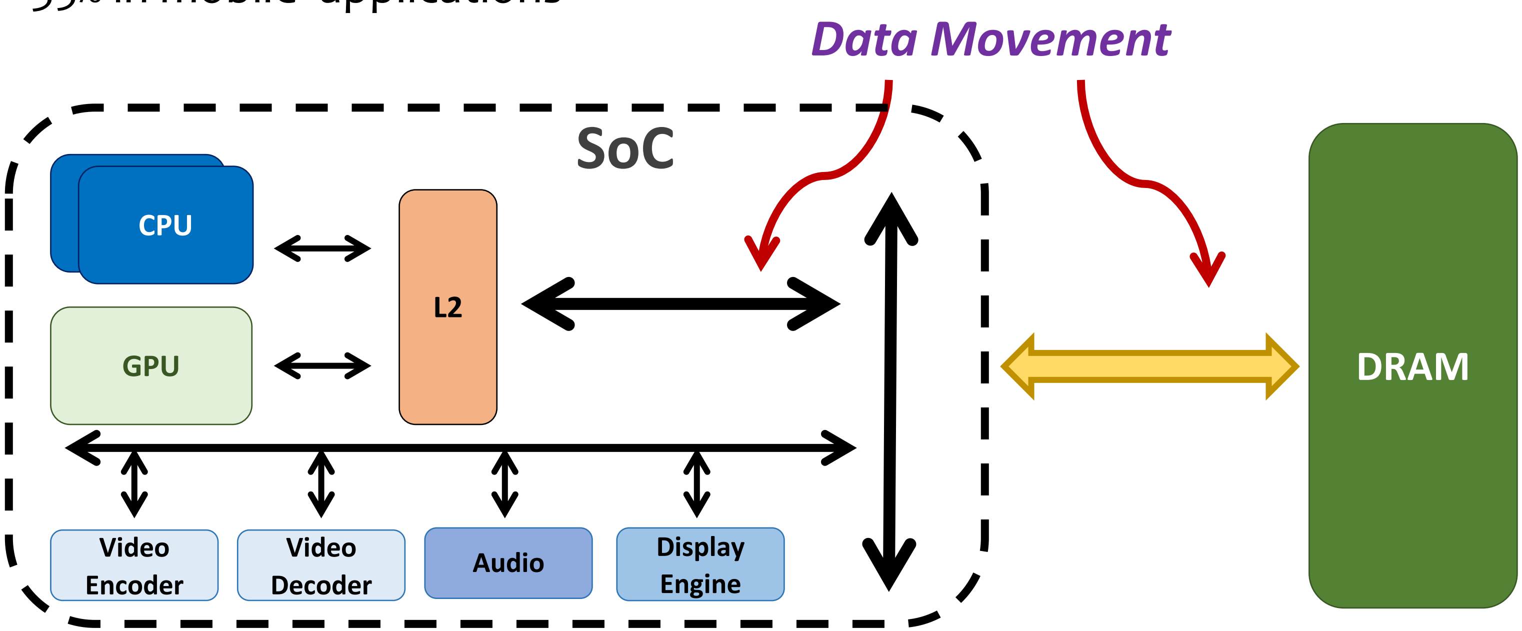
SAFARI

Executive Summary

- Data movement between memory/storage units and compute units is a major contributor to execution time and energy consumption
- Processing-in-Memory (PIM) is a paradigm that can tackle the *data movement bottleneck*
  - Though explored for +50 years, technology challenges prevented the successful materialization
- UPMEM has designed and fabricated the first publicly-available real-world PIM architecture
  - DDR4 chips embedding in-order multithreaded DRAM Processing Units (DPUs)
- Our work:
  - Introduction to UPMEM programming model and PIM architecture
  - Microbenchmark-based characterization of the DPU
  - Benchmarking and workload suitability study
- Main contributions:
  - Comprehensive characterization and analysis of the first commercially-available PIM architecture
  - **PrIM** (Processing-In-Memory) benchmarks:
    - 16 workloads that are memory-bound in conventional processor-centric systems
    - Strong and weak scaling characteristics
  - Comparison to state-of-the-art CPU and GPU
- Takeaways:
  - Workload characteristics for PIM suitability
  - Programming recommendations
  - Suggestions and hints for hardware and architecture designers of future PIM systems
  - PrIM: (a) programming samples, (b) evaluation and comparison of current and future PIM systems

Data Movement in Computing Systems

- Data movement dominates performance and is a major system energy bottleneck
- Total system energy: data movement accounts for
  - 62% in consumer applications*,
  - 40% in scientific applications★,
  - 35% in mobile applications☆

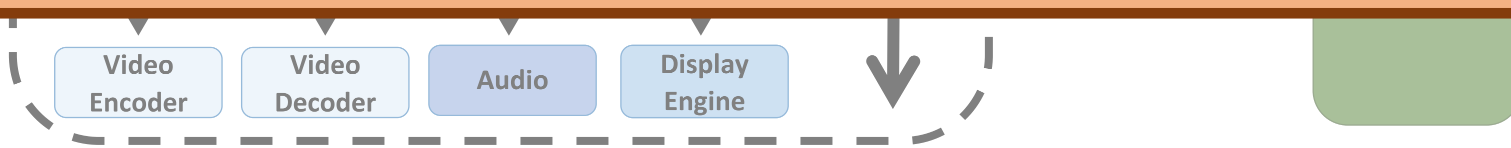


* Boroumand et al., "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," ASPLOS 2018
★ Kestor et al., "Quantifying the Energy Cost of Data Movement in Scientific Applications," IISWC 2013
☆ Pandiyan and Wu, "Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms," IISWC 2014

Data Movement in Computing Systems

- Data movement dominates performance and is a major system energy bottleneck
- Total system energy: data movement accounts for
  - 62% in consumer applications*,

Compute systems should be more data-centric



Processing-In-Memory proposes computing where it makes sense (where data resides)

* Boroumand et al., "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," ASPLOS 2018
★ Kestor et al., "Quantifying the Energy Cost of Data Movement in Scientific Applications," IISWC 2013
☆ Pandiyan and Wu, "Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms," IISWC 2014

A +50-Year-Old Paradigm

- Kautz, "Cellular Logic-in-Memory Arrays", IEEE TC 1969

IEEE TRANSACTIONS ON COMPUTERS, VOL. C-18, NO. 8, AUGUST 1969

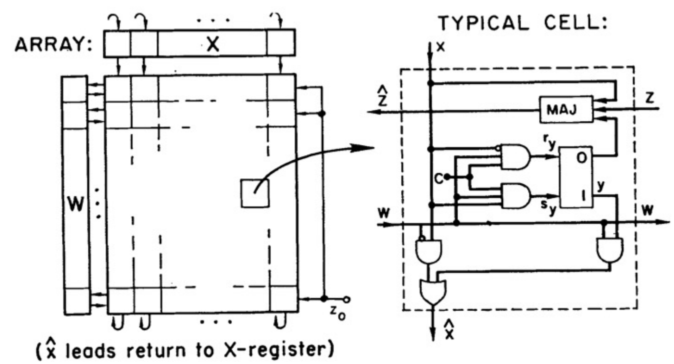
Cellular Logic-in-Memory Arrays

WILLIAM H. KAUTZ, MEMBER, IEEE

Abstract—As a direct consequence of large-scale integration, many advantages in the design, fabrication, testing, and use of digital circuitry can be achieved if the circuits can be arranged in a two-dimensional iterative, or cellular, array of identical elementary networks, or cells. When a small amount of storage is included in each cell, the same array may be regarded either as a logically enhanced memory array, or as a logic array whose elementary gates and connections can be "programmed" to realize a desired logical behavior.

In this paper the specific engineering features of such cellular logic-in-memory (CLIM) arrays are discussed, and one such special-purpose array, a cellular sorting array, is described in detail to illustrate how these features may be achieved in a particular design. It is shown how the cellular sorting array can be employed as a single-address, multiword memory that keeps in order all words stored within it. It can also be used as a content-addressed memory, a pushdown memory, a buffer memory, and (with a lower logical efficiency) a programmable array for the realization of arbitrary switching functions. A second version of a sorting array, operating on a different sorting principle, is also described.

Index Terms—Cellular logic, large-scale integration, logic arrays logic in memory, push-down memory, sorting, switching functions.



CELL EQUATIONS: $\hat{x} = \overline{w}x + wy$
$s_y = wcx, \; r_y = wc\overline{x}$
$\hat{z} = M(x, \overline{y}, z) = x\overline{y} + z(x+\overline{y})$

Fig. 1. Cellular sorting array I.

https://doi.org/10.1109/T-C.1969.222754

Processing in/near Memory: An Old Idea

- Stone, "A Logic-in-Memory Computer," IEEE TC 1970.

A Logic-in-Memory Computer

HAROLD S. STONE

Abstract—If, as presently projected, the cost of microelectronic arrays in the future will tend to reflect the number of pins on the array rather than the number of gates, the logic-in-memory array is an extremely attractive computer component. Such an array is essentially a microelectronic memory with some combinational logic associated with each storage element.

PIM Review and Open Problems

A Modern Primer on Processing in Memory

Onur Mutlu[a,b], Saugata Ghose[b,c], Juan Gómez-Luna[a], Rachata Ausavarungnirun[d]

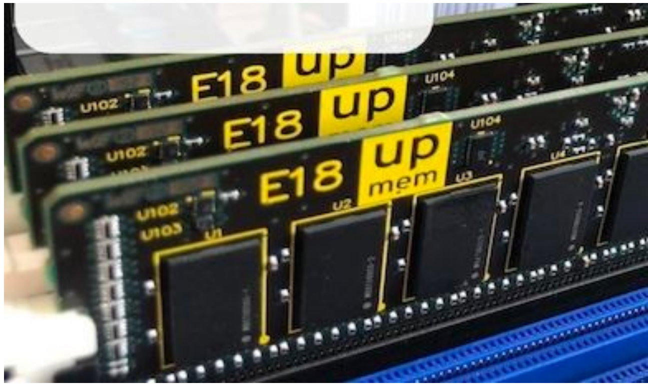
SAFARI Research Group

[a] *ETH Zürich*
[b] *Carnegie Mellon University*
[c] *University of Illinois at Urbana-Champaign*
[d] *King Mongkut's University of Technology North Bangkok*

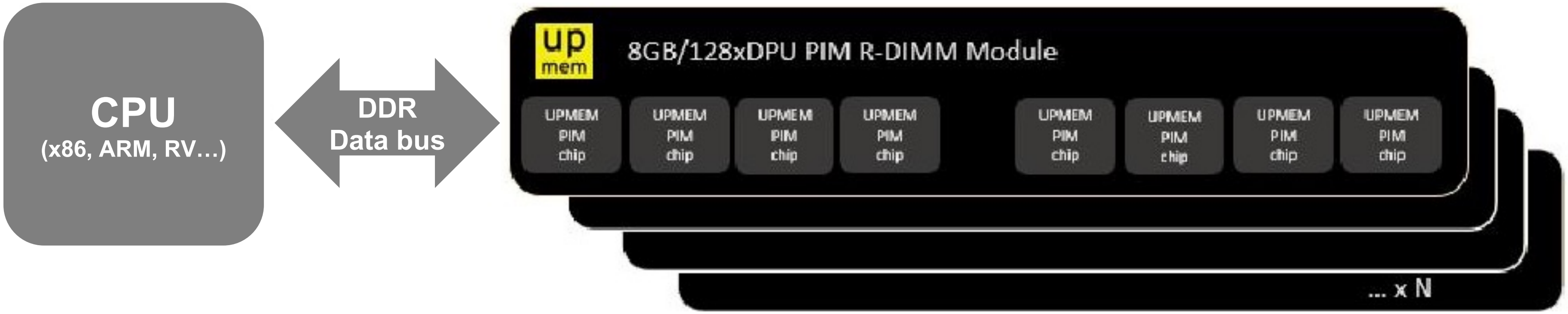
Onur Mutlu, Saugata Ghose, Juan Gomez-Luna, and Rachata Ausavarungnirun,
"A Modern Primer on Processing in Memory"
Invited Book Chapter in ***Emerging Computing: From Devices to Systems - Looking Beyond Moore and Von Neumann***, Springer, to be published in 2021.

UPMEM Processing-in-DRAM Engine (2019)

- Processing in DRAM Engine
- Includes **standard DIMM modules**, with a **large number of DPU processors** combined with DRAM chips.



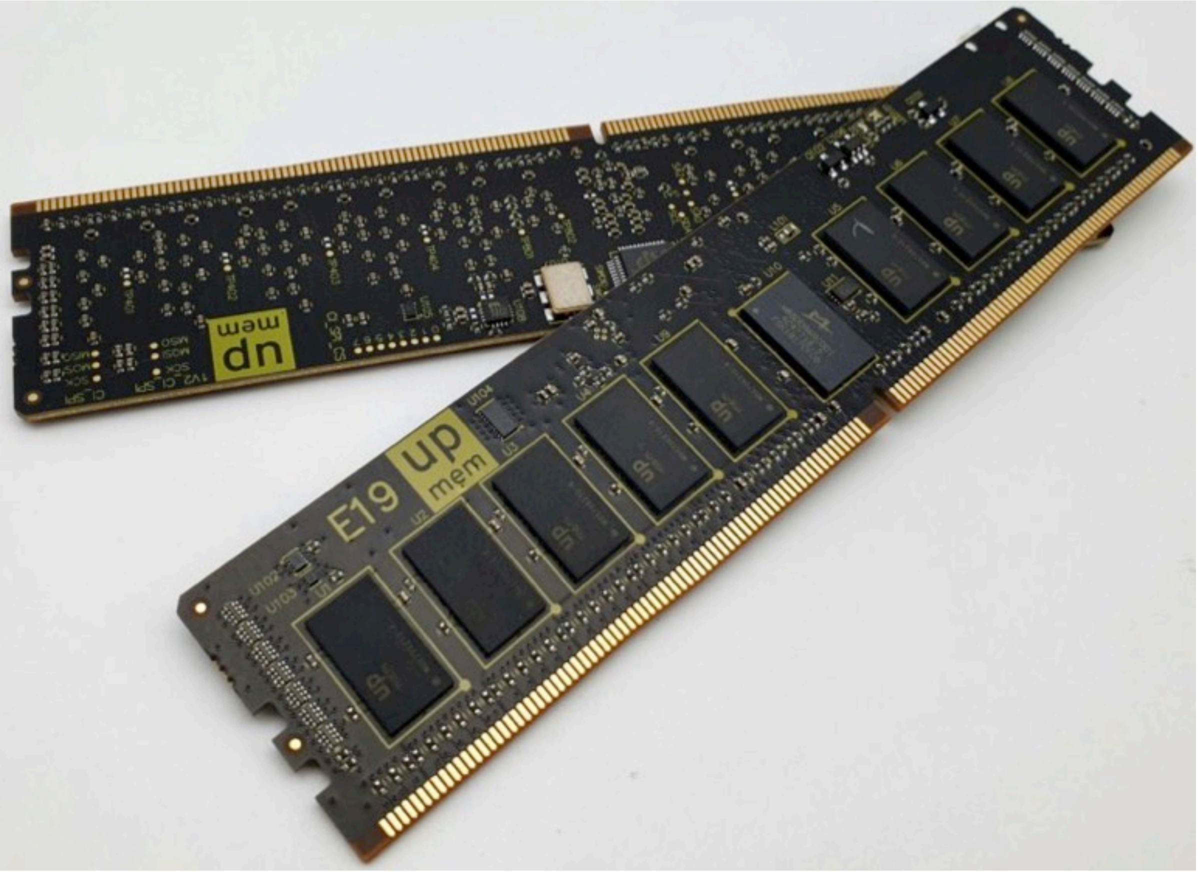
- Replaces **standard** DIMMs
  - DDR4 R-DIMM modules
    - 8GB+128 DPUs (16 PIM chips)
    - Standard 2x-nm DRAM process
  - **Large amounts of** compute & memory bandwidth



https://www.anandtech.com/show/14750/hot-chips-31-analysis-inmemory-processing-by-upmem
https://www.upmem.com/video-upmem-presenting-its-true-processing-in-memory-solution-hot-chips-2019/

UPMEM DIMMs

- E19: 8 chips/DIMM (1 rank). DPUs @ 267 MHz
- P21: 16 chips/DIMM (2 ranks). DPUs @ 350 MHz



PIM's Promises

UPMEM PIM massive benefits

- Massive speed-up
  - Massive additional compute & bandwidth
- Massive energy gains
  - Most data movement on chip
- Low cost
  - ~300$ of additional DRAM silicon
  - Affordable programming
- Massive ROI / TCO gains

Energy efficiency when computing on or off memory chip		**Server + PIM DRAM**	Server + normal DRAM
DRAM to processor 64-bit operand	pJ	~150	~3000*
Operation	pJ	~20	~10*
Server consumption	W	**~700W**	~300W
speed-up		**~ x20**	x1
energy gain		**~ x10**	x1
TCO gain		**~ x10**	x1

**Exascale Computing Trends: Adjusting to the "New Normal" for Computer Architecture; John Shalf, Computing in Science & engineering, 2013*

HOT CHIPS 31

F. Devaux, "The true Processing In Memory accelerator," HotChips 2019. doi: 10.1109/HOTCHIPS.2019.8875680

Technology Challenges

The Hurdles on the road to the Graal

- DRAM process highly constrained
  - 3x slower transistors than same node digital process
  - Logic 10 times less dense vs. ASIC process
  - Routing density dramatically lower
    - 3 metals only for routing (vs. 10+), pitch x4 larger
- Strong design choices mandatory

Take away

DRAM vs. ASIC
- Far less performing
- Wafers 2x cheaper vs. ASIC

Leapfrogging Moore's law
- **Total** Energy efficiency x10
- Massive, scalable parallelism
- Very low cost

But the PIM Graal is worth it !

HOT CHIPS 31

up mem

F. Devaux, "The true Processing In Memory accelerator," HotChips 2019. doi: 10.1109/HOTCHIPS.2019.8875680

UPMEM Patent

(12) **United States Patent**
Devaux et al.

(10) **Patent No.: US 10,324,870 B2**
(45) **Date of Patent: Jun. 18, 2019**

(54) **MEMORY CIRCUIT WITH INTEGRATED PROCESSOR**

(71) Applicant: **UPMEM**, Grenoble (FR)

(72) Inventors: **Fabrice Devaux**, La Conversion (CH); **Jean-François Roy**, Grenoble (FR)

(73) Assignee: **UPMEM**, Grenoble (FR)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **15/551,418**

(22) PCT Filed: **Feb. 12, 2016**

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,666,485 A *	9/1997	Suresh	G06F 13/1605	710/113
6,463,001 B1	10/2002	Williams		
7,349,277 B2 *	3/2008	Kinsley	G11C 11/406	365/193
8,438,358 B1 *	5/2013	Kraipak	G11C 7/04	711/167

(Continued)

FOREIGN PATENT DOCUMENTS

EP	0780768 A1	6/1997
JP	H03109661 A	5/1991
WO	2010/141221 A1	12/2010

(57) **ABSTRACT**

A memory circuit having: a memory array including one or more memory banks; a first processor; and a processor control interface for receiving data processing commands directed to the first processor from a central processor, the processor control interface being adapted to indicate to the central processor when the first processor has finished accessing one or more of the memory banks of the memory array, these memory banks becoming accessible to the central processor.

Understanding a Modern PIM Architecture

Understanding a Modern Processing-in-Memory Architecture: Benchmarking and Experimental Characterization

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

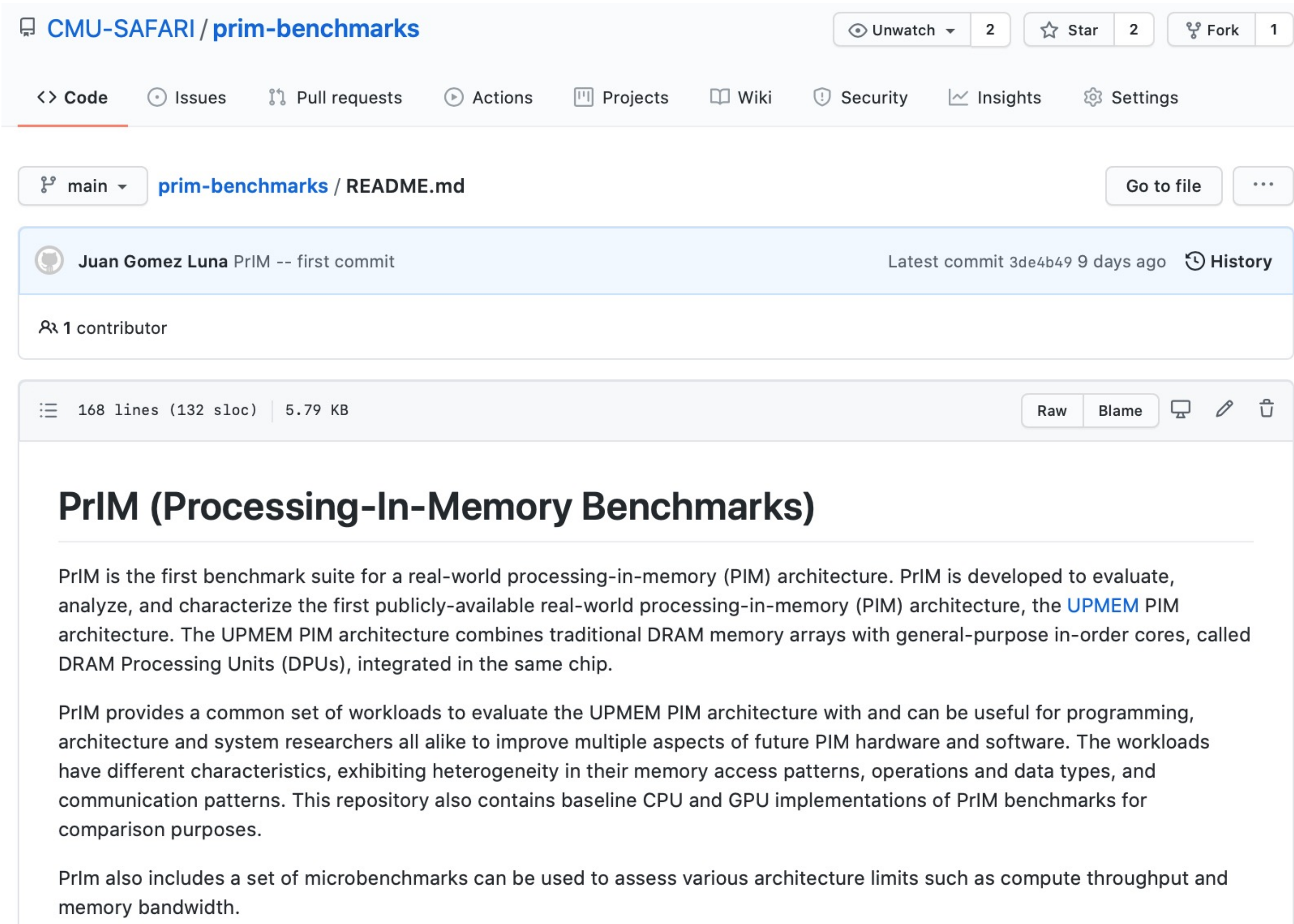
[1]ETH Zürich [2]American University of Beirut [3]University of Malaga [4]National Technical University of Athens

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



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

Observations, Recommendations, Takeaways

GENERAL PROGRAMMING RECOMMENDATIONS

1. Execute on the *DRAM Processing Units* (*DPUs*) **portions of parallel code** that are as long as possible.
2. Split the workload into **independent data blocks**, which the DPUs operate on independently.
3. Use **as many working DPUs** in the system as possible.
4. Launch at least **11 *tasklets* (i.e., software threads)** per DPU.

PROGRAMMING RECOMMENDATION 1

For data movement between the DPU's MRAM bank and the WRAM, **use large DMA transfer sizes when all the accessed data is going to be used**.

KEY OBSERVATION 7

Larger CPU-DPU and DPU-CPU transfers between the host main memory and the DRAM Processing Unit's Main memory (MRAM) banks **result in higher sustained bandwidth**.

KEY TAKEAWAY 1

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

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

Accelerator Model (I)

- UPMEM DIMMs coexist with conventional DIMMs

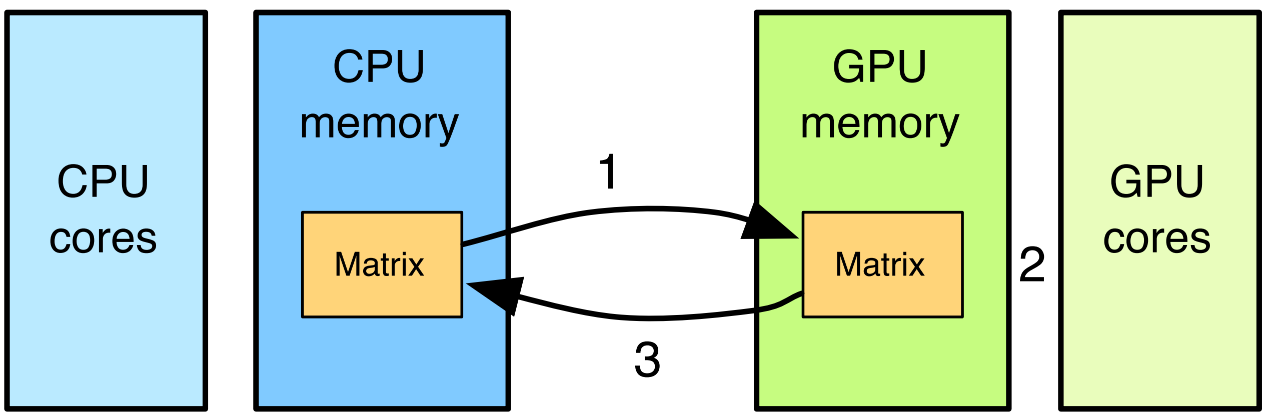
- Integration of UPMEM DIMMs in a system follows an accelerator model

- UPMEM DIMMs can be seen as a loosely coupled accelerator
  - Explicit data movement between the main processor (host CPU) and the accelerator (UPMEM)
  - Explicit kernel launch onto the UPMEM processors

- This resembles GPU computing

GPU Computing

- Computation is offloaded to the GPU
- Three steps
  - CPU-GPU data transfer (1)
  - GPU kernel execution (2)
  - GPU-CPU data transfer (3)



https://www.youtube.com/watch?v=y40-tY5WJ8A
https://safari.ethz.ch/digitaltechnik/spring2018/lib/exe/fetch.php?media=digitaldesign-2018-lecture22-gpuprogramming-afterlecture.pdf

Accelerator Model (II)

- *FIG. 6 is a flow diagram representing operations in a method of delegating a processing task to a DRAM processor according to an example embodiment*

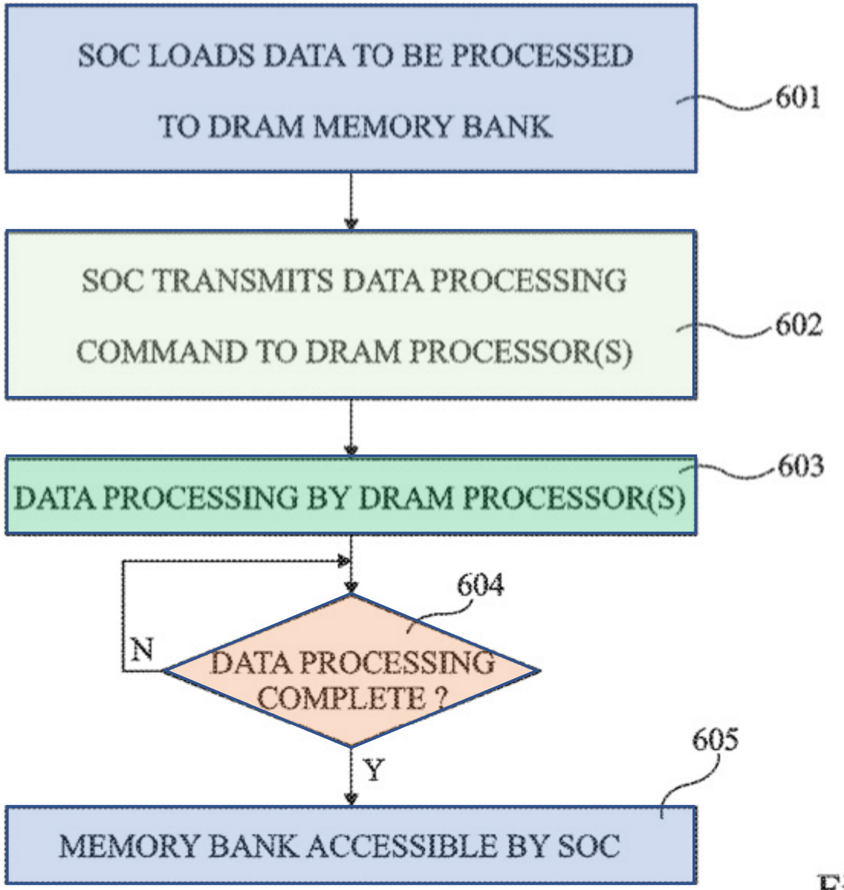


Fig 6

Fabrice Devaux, Jean-François Roy. "Memory circuit with integrated processor." US 10,324,870 B2.

System Organization (I)

- *FIG. 1 schematically illustrates a computing system comprising DRAM circuits having integrated processors according to an example embodiment*

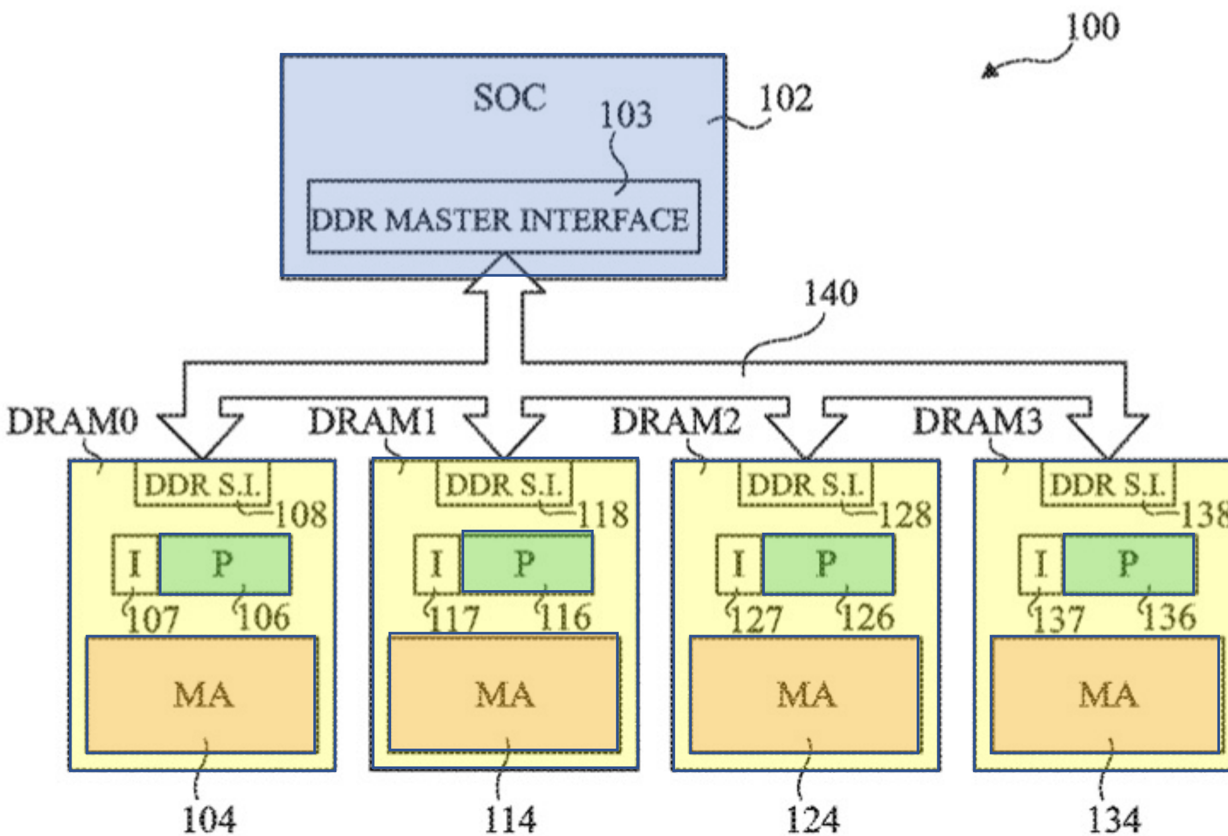
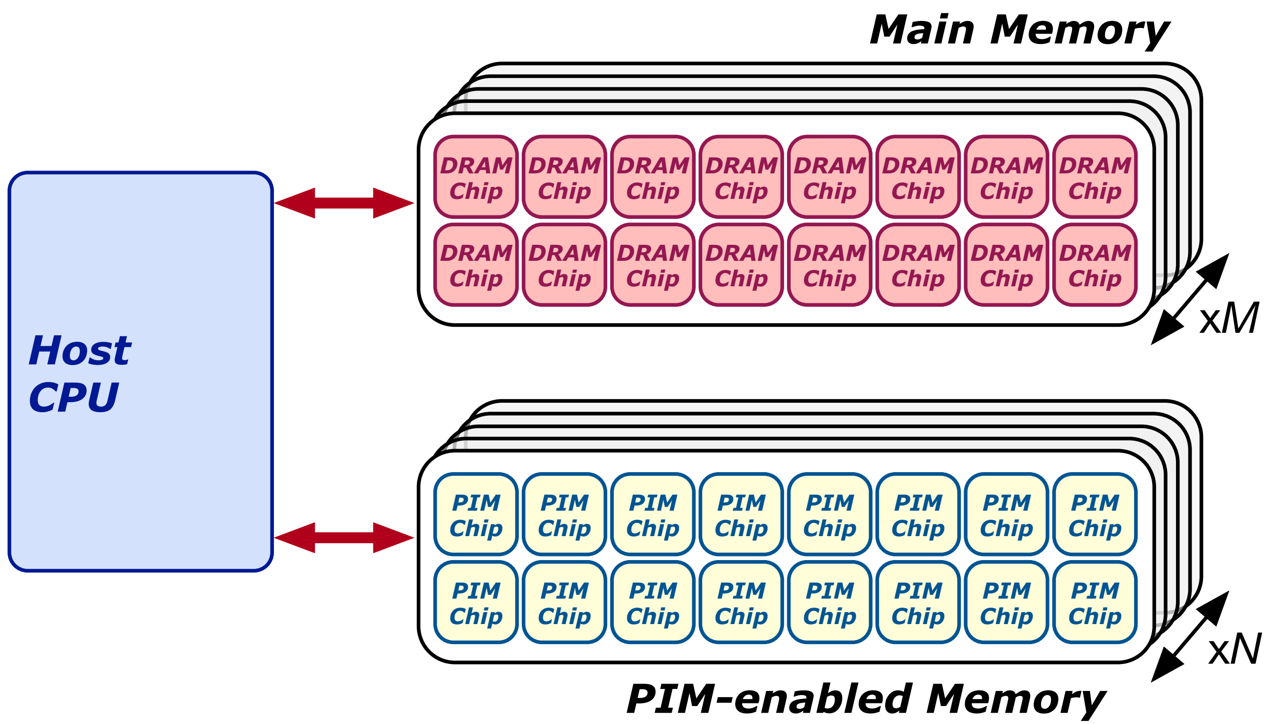


Fig 1

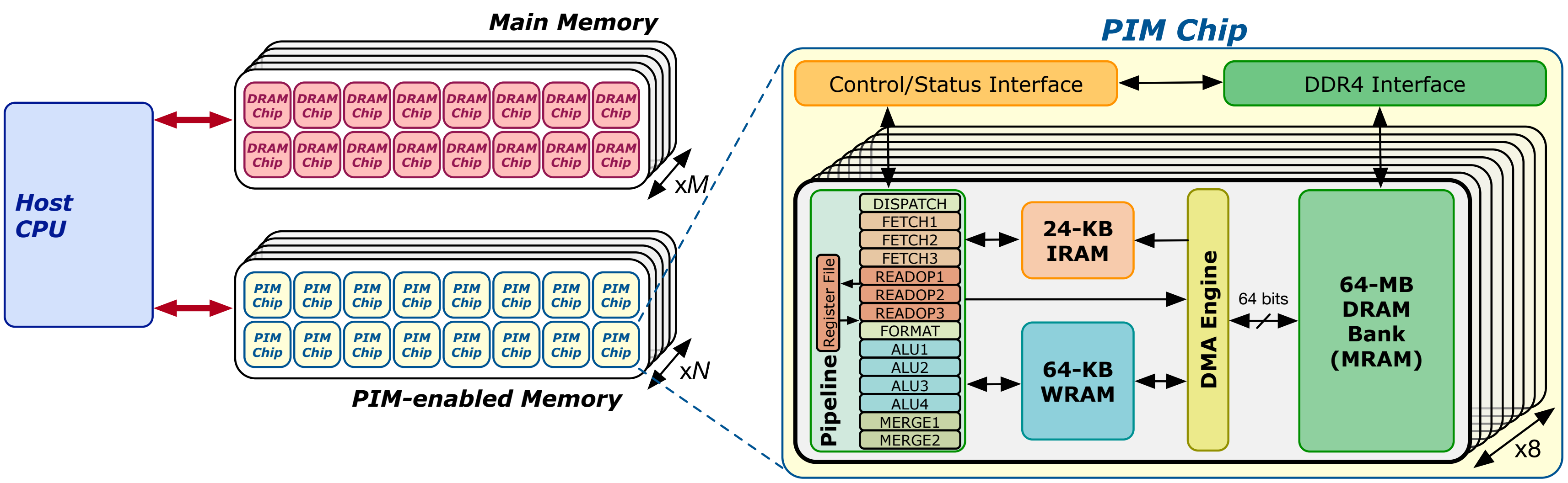
System Organization (II)

- In a UPMEM-based PIM system UPMEM DIMMs coexist with regular DDR4 DIMMs



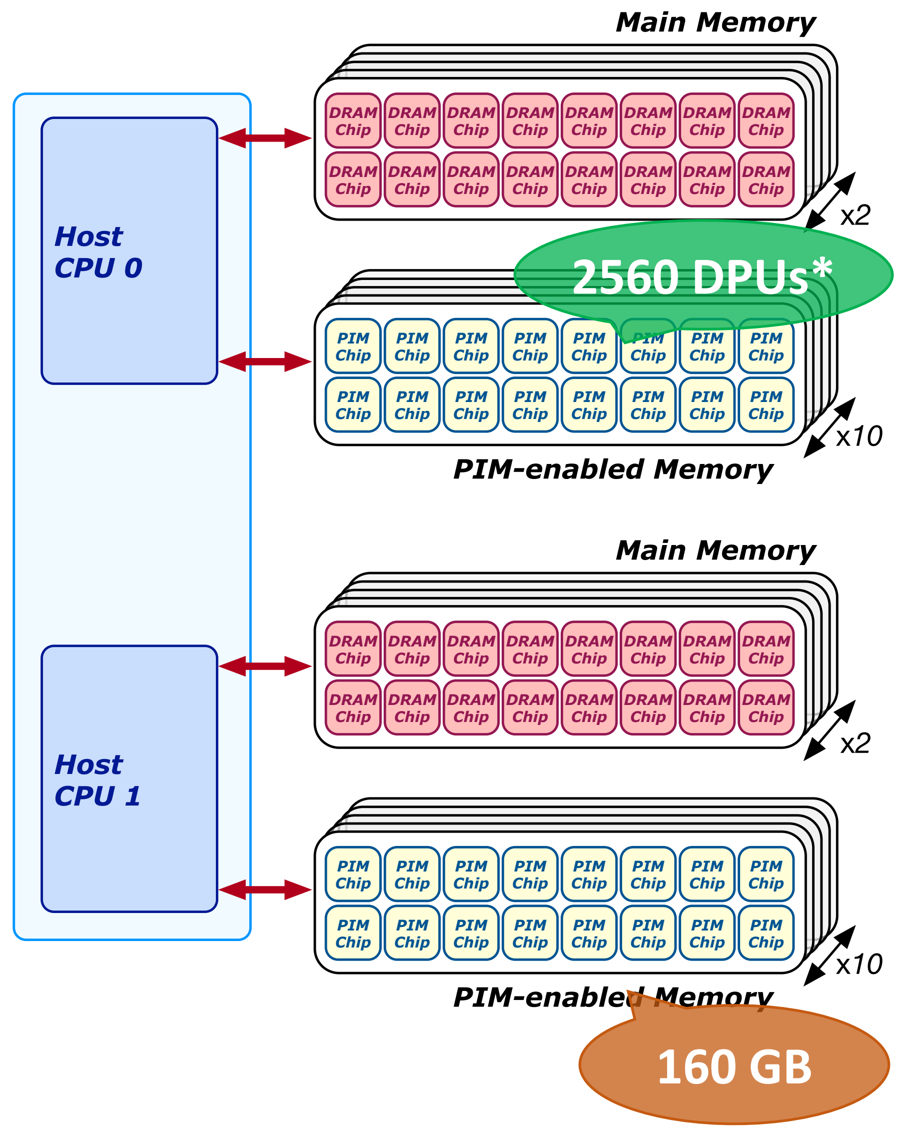
System Organization (III)

- A UPMEM DIMM contains 8 or 16 chips
  - Thus, 1 or 2 ranks of 8 chips each
- Inside each PIM chip there are:
  - 8 64MB banks per chip: Main RAM (MRAM) banks
  - 8 DRAM Processing Units (DPUs) in each chip, 64 DPUs per rank



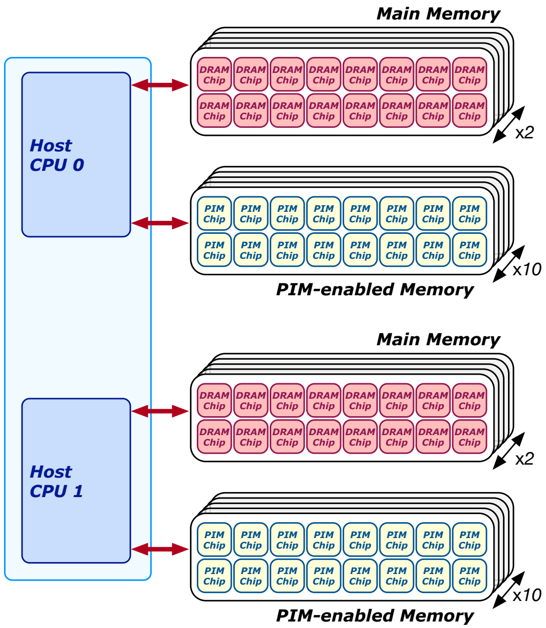
2,560-DPU System (I)

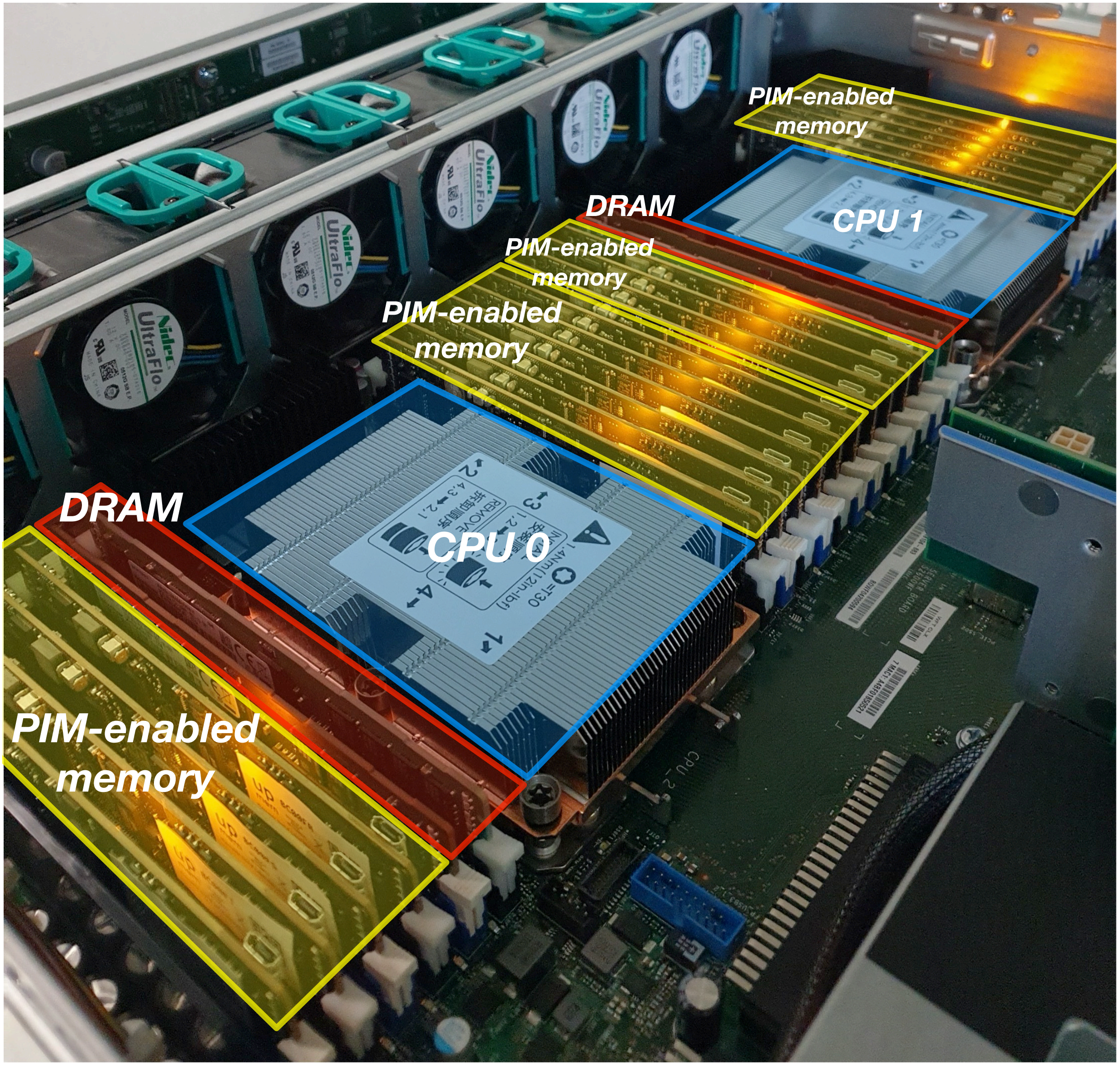
- UPMEM-based PIM system with 20 UPMEM DIMMs of 16 chips each (40 ranks)
  - P21 DIMMs
  - Dual x86 socket
    - UPMEM DIMMs coexist with regular DDR4 DIMMs
    - 2 memory controllers/socket (3 channels each)
    - 2 conventional DDR4 DIMMs on one channel of one controller



* 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.

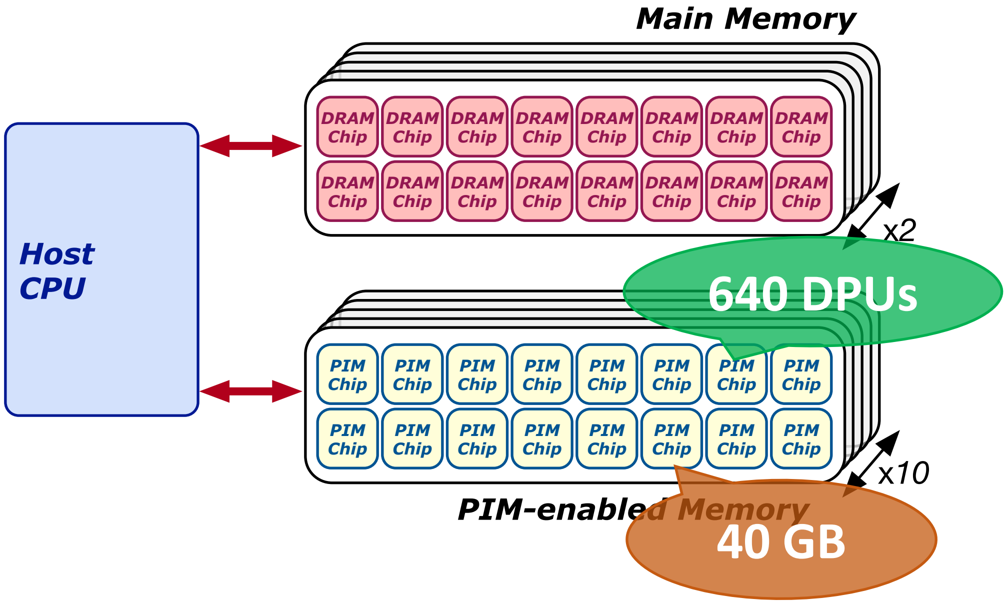
2,560-DPU System (II)





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



DPU Sharing? Security Implications?

- DPUs cannot be shared across multiple CPU processes
  - There are so many DPUs in the system that there is no need for sharing

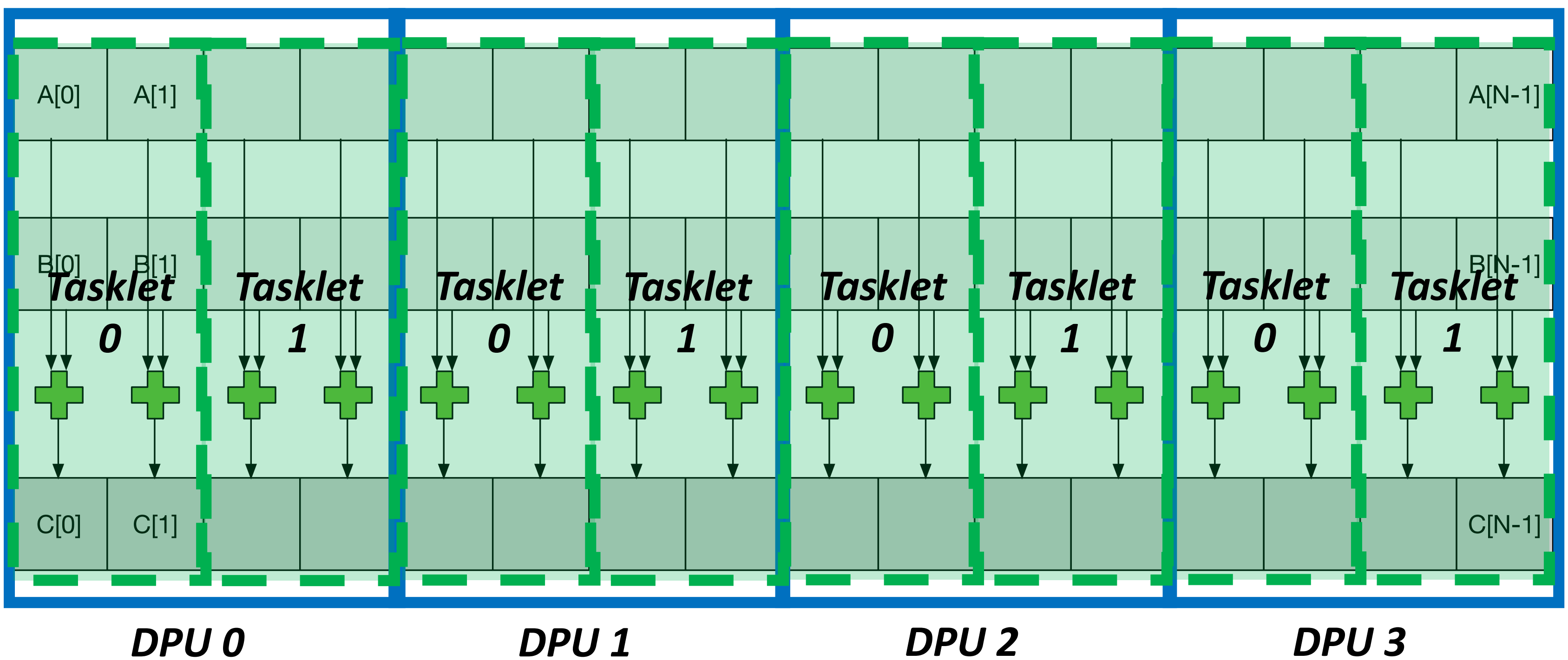
- According to UPMEM, this assumption makes things simpler
  - No need for OS
  - Simplified security implications: No side channels

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

Vector Addition (VA)

- Our first programming example
- We partition the input arrays across:
  - DPUs
  - Tasklets, i.e., software threads running on a DPU



General Programming Recommendations

- From UPMEM programming guide*, presentations★, and white papers☆

GENERAL PROGRAMMING RECOMMENDATIONS

1. Execute on the *DRAM Processing Units* (*DPUs*) **portions of parallel code** that are as long as possible.
2. Split the workload into **independent data blocks**, which the DPUs operate on independently.
3. Use **as many working DPUs** in the system as possible.
4. Launch at least **11 *tasklets* (i.e., software threads)** per DPU.

* https://sdk.upmem.com/2021.1.1/index.html
★ F. Devaux, "The true Processing In Memory accelerator," HotChips 2019. doi: 10.1109/HOTCHIPS.2019.8875680
☆ UPMEM, "Introduction to UPMEM PIM. Processing-in-memory (PIM) on DRAM Accelerator," White paper

DPU Allocation

- `dpu_alloc()` allocates a number of DPUs
  - Creates a `dpu_set`

```
struct dpu_set_t dpu_set, dpu;
uint32_t nr_of_dpus;

// Allocate DPUs
DPU_ASSERT(dpu_alloc(NR_DPUS, NULL, &dpu_set));

DPU_ASSERT(dpu_get_nr_dpus(dpu_set, &nr_of_dpus));
printf("Allocated %d DPU(s)\n", nr_of_dpus);
```

Can we allocate different DPU sets
over the course of a program?

Yes, we can. We show an example next

We deallocate a DPU set with `dpu_free()`

DPU Allocation: Needleman-Wunsch (NW)

- In NW we change the number of DPUs in the DPU set as computation progresses

```
// Top-left computation on DPUs
for (unsigned int blk = 1; blk <= (max_cols-1)/BL; blk++) {

    // If nr_of_blocks are lower than max_dpus,
    // set nr_of_dpus to be equal with nr_of_blocks
    unsigned nr_of_blocks = blk;
    if (nr_of_blocks < max_dpus) {
        DPU_ASSERT(dpu_free(dpu_set));
        DPU_ASSERT(dpu_alloc(nr_of_blocks, NULL, &dpu_set));
        DPU_ASSERT(dpu_load(dpu_set, DPU_BINARY, NULL));
        DPU_ASSERT(dpu_get_nr_dpus(dpu_set, &nr_of_dpus));
    } else if (nr_of_dpus == max_dpus) {
        ;
    } else {
        DPU_ASSERT(dpu_free(dpu_set));
        DPU_ASSERT(dpu_alloc(max_dpus, NULL, &dpu_set));
        DPU_ASSERT(dpu_load(dpu_set, DPU_BINARY, NULL));
        DPU_ASSERT(dpu_get_nr_dpus(dpu_set, &nr_of_dpus));
    }

    ...
}
```

Load DPU Binary

- `dpu_load()` loads a program in all DPUs of a `dpu_set`

```c
// Define the DPU Binary path as DPU_BINARY here
#ifndef DPU_BINARY
#define DPU_BINARY "./bin/dpu_code"
#endif

    ...

    // Load binary
    DPU_ASSERT(dpu_load(dpu_set, DPU_BINARY, NULL));

```

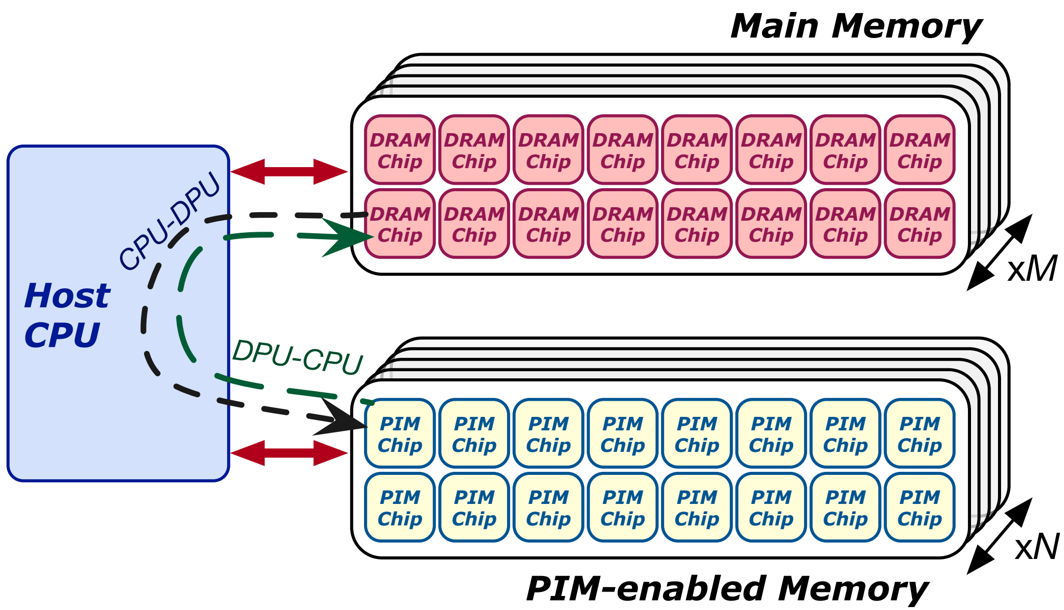
Is it possible to launch different kernels onto different DPUs?

Yes, it is possible. This enables:

- Workloads with task-level parallelism
- Different programs using different DPU sets

CPU-DPU/DPU-CPU Data Transfers

- CPU-DPU and DPU-CPU transfers
  - Between host CPU's main memory and DPUs' MRAM banks



- Serial CPU-DPU/DPU-CPU transfers:
  - A single DPU (i.e., 1 MRAM bank)
- Parallel CPU-DPU/DPU-CPU transfers:
  - Multiple DPUs (i.e., many MRAM banks)
- Broadcast CPU-DPU transfers:
  - Multiple DPUs with a single buffer

Serial Transfers

- `dpu_copy_to();`
- `dpu_copy_from();`
- We transfer (part of) a buffer to/from each DPU in the `dpu_set`
- `DPU_MRAM_HEAP_POINTER_NAME`: Start of the MRAM range that can be freely accessed by applications
  - We do not allocate MRAM explicitly

```
DPU_FOREACH (dpu_set, dpu) {
    DPU_ASSERT(dpu_copy_to(dpu, DPU_MRAM_HEAP_POINTER_NAME, 0,                                  bufferA + input_size_dpu_8bytes * i, input_size_dpu_8bytes * sizeof(T)));
    DPU_ASSERT(dpu_copy_to(dpu, DPU_MRAM_HEAP_POINTER_NAME, input_size_dpu_8bytes * sizeof(T), bufferB + input_size_dpu_8bytes * i, input_size_dpu_8bytes * sizeof(T)));
    i++;
}
```

Offset within MRAM   Pointer to main memory   Transfer size

Parallel Transfers

- We push different buffers to/from a DPU set in one transfer
  - All buffers need to be of the same size
- First, prepare (`dpu_prepare_xfer`);
  then, push (`dpu_push_xfer`)
- Direction:
  - `DPU_XFER_TO_DPU`
  - `DPU_XFER_FROM_DPU`

```
DPU_FOREACH(dpu_set, dpu, i) {
    DPU_ASSERT(dpu_prepare_xfer(dpu, bufferA + input_size_dpu_8bytes * i))
}
DPU_ASSERT(dpu_push_xfer(dpu_set, DPU_XFER_TO_DPU DPU_MRAM_HEAP_POINTER_NAME, 0, input_size_dpu_8bytes * sizeof(T) DPU_XFER_DEFAULT));

DPU_FOREACH(dpu_set, dpu, i) {
    DPU_ASSERT(dpu_prepare_xfer(dpu, bufferB + input_size_dpu_8bytes * i))
}
DPU_ASSERT(dpu_push_xfer(dpu_set, DPU_XFER_TO_DPU DPU_MRAM_HEAP_POINTER_NAME, input_size_dpu_8bytes * sizeof(T) input_size_dpu_8bytes * sizeof(T) DPU_XFER_DEFAULT));
```

Pointer to main memory

Offset within MRAM

Transfer size

Direction

Broadcast Transfers

- `dpu_broadcast_to();`
    - Only CPU to DPU
- We transfer the same buffer to all DPU in the `dpu_set`

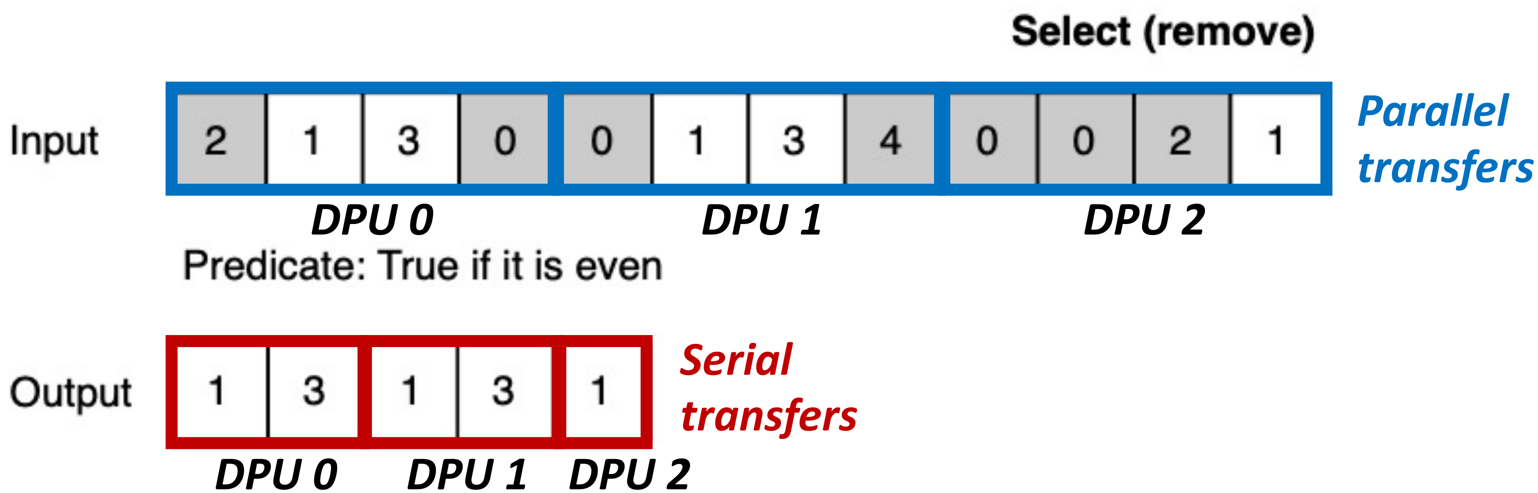
```
DPU_ASSERT(dpu_broadcast_to(dpu_set, DPU_MRAM_HEAP_POINTER_NAME, 0, bufferA, input_size_dpu * sizeof(T), DPU_XFER_DEFAULT));
```

Pointer to main memory

Transfer size

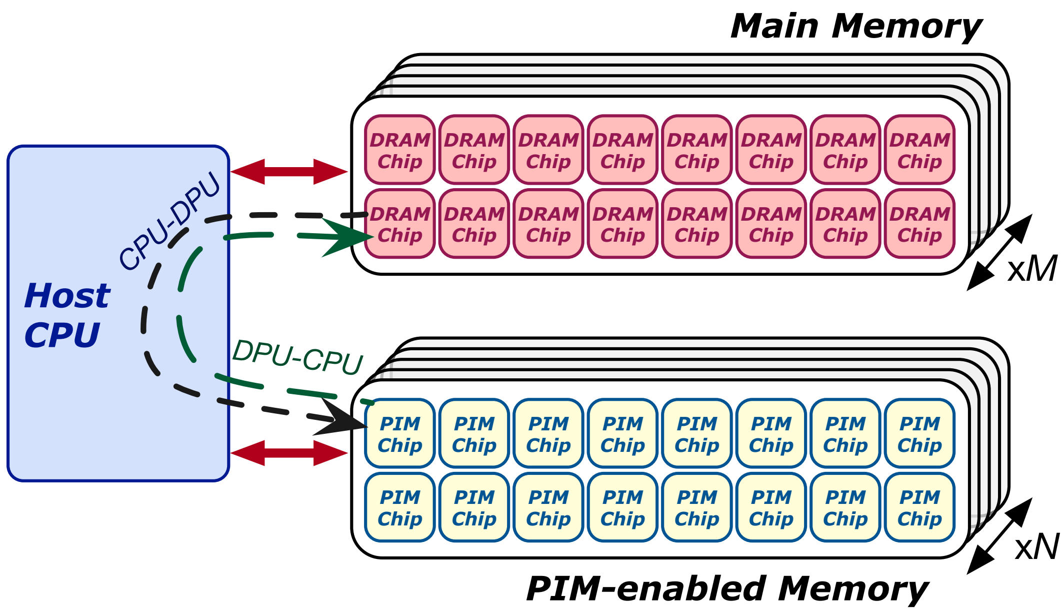
Different Types of Transfers in a Program

- An example benchmark that uses both parallel and serial transfers
- Select (SEL)
  - Remove even values



Inter-DPU Communication

- There is no direct communication channel between DPUs



- Inter-DPU communication takes places via the host CPU using CPU-DPU and DPU-CPU transfers
- Example communication patterns:
  - Merging of partial results to obtain the final result
    - Only DPU-CPU transfers
  - Redistribution of intermediate results for further computation
    - DPU-CPU transfers and CPU-DPU transfers

How Fast are these Data Transfers?

- With a microbenchmark, we obtain the sustained bandwidth of all types of CPU-DPU and DPU-CPU transfers
- Two experiments:
  - 1 DPU: variable CPU-DPU and DPU-CPU transfer size (8 bytes to 32 MB)
  - 1 rank: 32 MB CPU-DPU and DPU-CPU transfers to/from a set of 1 to 64 MRAM banks within the same rank
- We do not experiment with more than one rank
  - Preliminary experiments show that the UPMEM SDK* only parallelizes transfers within the same rank

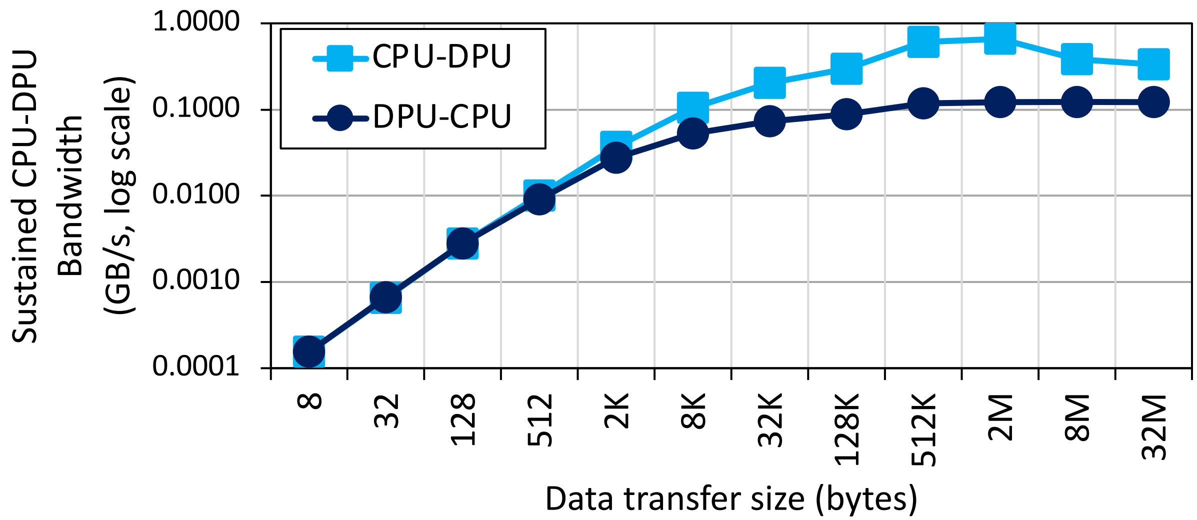
DDR4 bandwidth bounds the maximum transfer bandwidth

The cost of the transfers can be amortized, if enough computation is run on the DPUs

* UPMEM SDK 2021.1.1

CPU-DPU/DPU-CPU Transfers: 1 DPU

- Data transfer size varies between 8 bytes and 32 MB

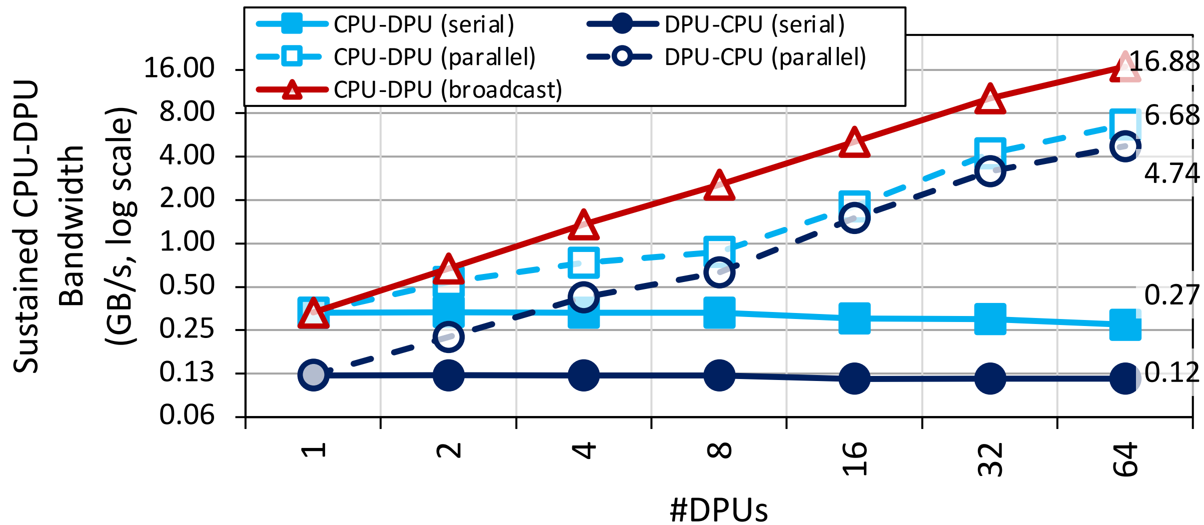


KEY OBSERVATION 7

Larger CPU-DPU and DPU-CPU transfers between the host main memory and the DRAM Processing Unit's Main memory (MRAM) banks **result in higher sustained bandwidth**.

CPU-DPU/DPU-CPU Transfers: 1 Rank (I)

- CPU-DPU (serial/parallel/broadcast) and DPU-CPU (serial/parallel)
- The number of DPUs varies between 1 and 64

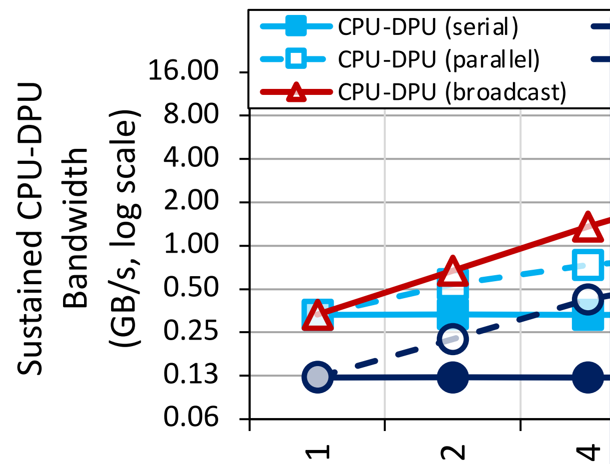


KEY OBSERVATION 8

The **sustained bandwidth of parallel CPU-DPU and DPU-CPU transfers** between the host main memory and the DRAM Processing Unit's Main memory (MRAM) banks **increases with the number of DRAM Processing Units inside a rank.**

CPU-DPU/DPU-CPU Transfers: 1 Rank (II)

- CPU-DPU (serial/parallel/broadcast) and DPU-CPU (serial/parallel)
- The number of DPUs varies between 1 and 64



KEY OBSERVATION 9

The sustained bandwidth of parallel CPU-DPU transfers is higher than the sustained bandwidth of parallel DPU-CPU transfers **due to different implementations** of CPU-DPU and DPU-CPU transfers **in the UPMEM runtime library**.

The sustained bandwidth of broadcast CPU-DPU transfers (i.e., the same buffer is copied to multiple MRAM banks) **is higher than that of parallel CPU-DPU transfers** (i.e., different buffers are copied to different MRAM banks) **due to higher temporal locality** in the CPU cache hierarchy.

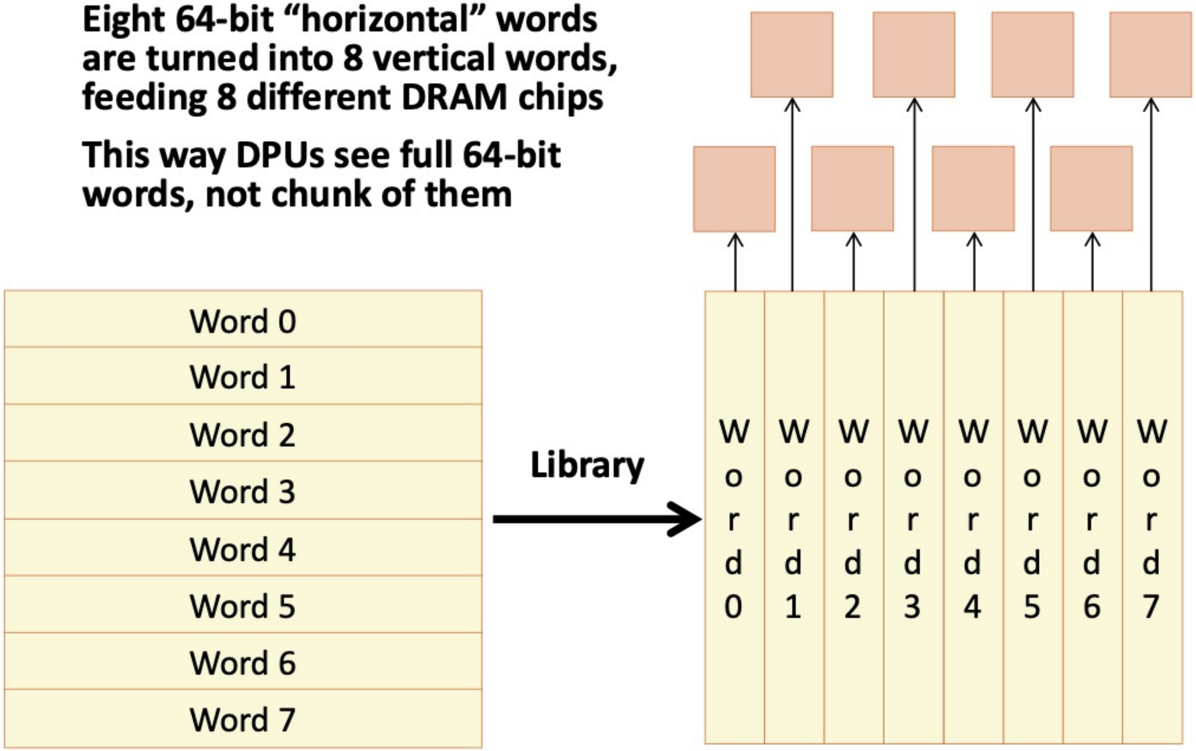
“Transposing” Library

The library feeds DPUs with correct data

Eight 64-bit “horizontal” words are turned into 8 vertical words, feeding 8 different DRAM chips

This way DPUs see full 64-bit words, not chunk of them

DRAM chip have 8-bit data bus



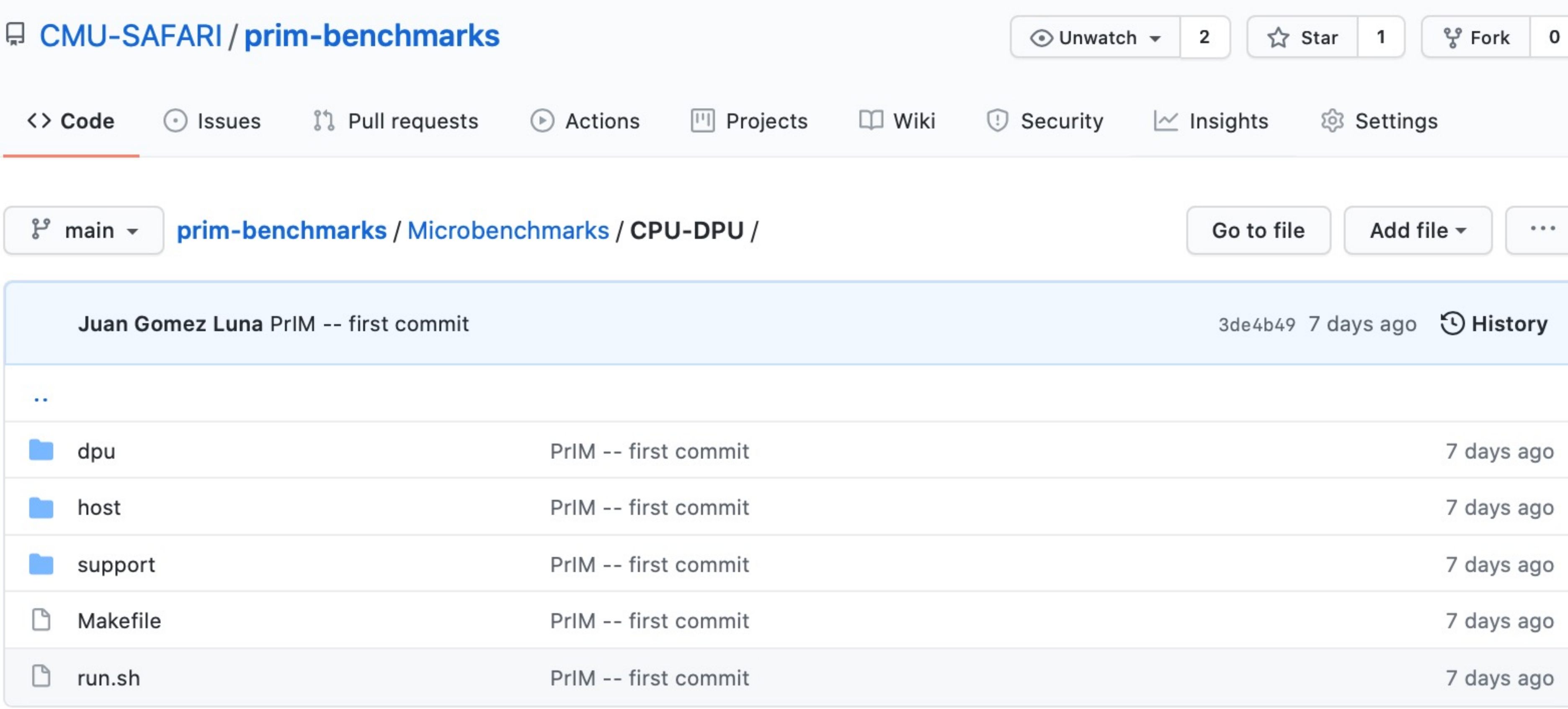
The transformation, a 8x8 matrix transposition, is done by the library inside a 64-byte cache line, thus very efficiently.

HOT CHIPS 31

F. Devaux, "The true Processing In Memory accelerator," HotChips 2019. doi: 10.1109/HOTCHIPS.2019.8875680

Microbenchmark: CPU-DPU

- CPU-DPU (serial/parallel/broadcast) and DPU-CPU (serial/parallel)



DPU Kernel Launch

- `dpu_launch()` launches a kernel on a `dpu_set`
  - `DPU_SYNCHRONOUS` suspends the application until the kernel finishes
  - `DPU_ASYNCHRONOUS` returns the control to the application
    - `dpu_sync` or `dpu_status` to check kernel completion

```
printf("Run program on DPU(s) \n");
// Run DPU kernel
DPU_ASSERT(dpu_launch(dpu_set, DPU_SYNCHRONOUS));
```

What does the asynchronous execution enable?

Some ideas:
- Task-level parallelism: concurrent execution of different kernels on different DPU sets
- Concurrent heterogeneous computation on CPU and DPUs

How to Pass Parameters to the Kernel?

- We can use serial and parallel transfers
- We pass them directly to the scratchpad memory of the DPU
  - Working RAM (WRAM): We introduce it in the next slides
- This is useful for input parameters and some results

```
// In DPU WRAM (dpu/task.c)
__host dpu_arguments_t DPU_INPUT_ARGUMENTS;
__host dpu_results_t DPU_RESULTS[NR_TASKLETS];

```

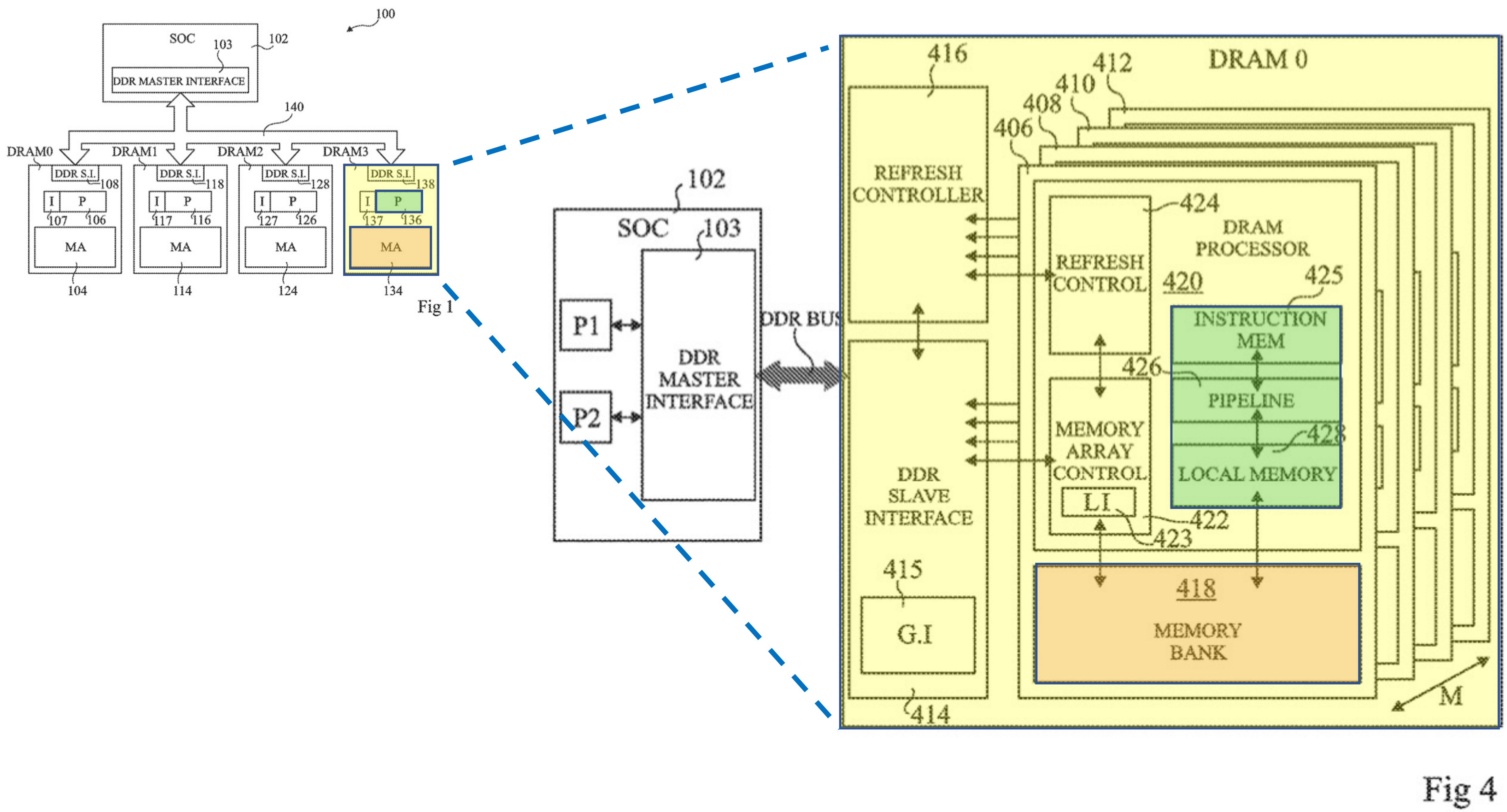
```
// Host code (host/app.c)
#ifdef SERIAL
        DPU_FOREACH (dpu_set, dpu) {
            DPU_ASSERT(dpu_copy_to(dpu, "DPU_INPUT_ARGUMENTS", 0, (const void *)&input_arguments[i], sizeof(input_arguments[0])));
            i++;
        }
#else
        DPU_FOREACH(dpu_set, dpu, i) {
            DPU_ASSERT(dpu_prepare_xfer(dpu, &input_arguments[i]));
        }
        DPU_ASSERT(dpu_push_xfer(dpu_set, DPU_XFER_TO_DPU, "DPU_INPUT_ARGUMENTS", 0, sizeof(input_arguments[0]), DPU_XFER_DEFAULT));
#endif

```

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

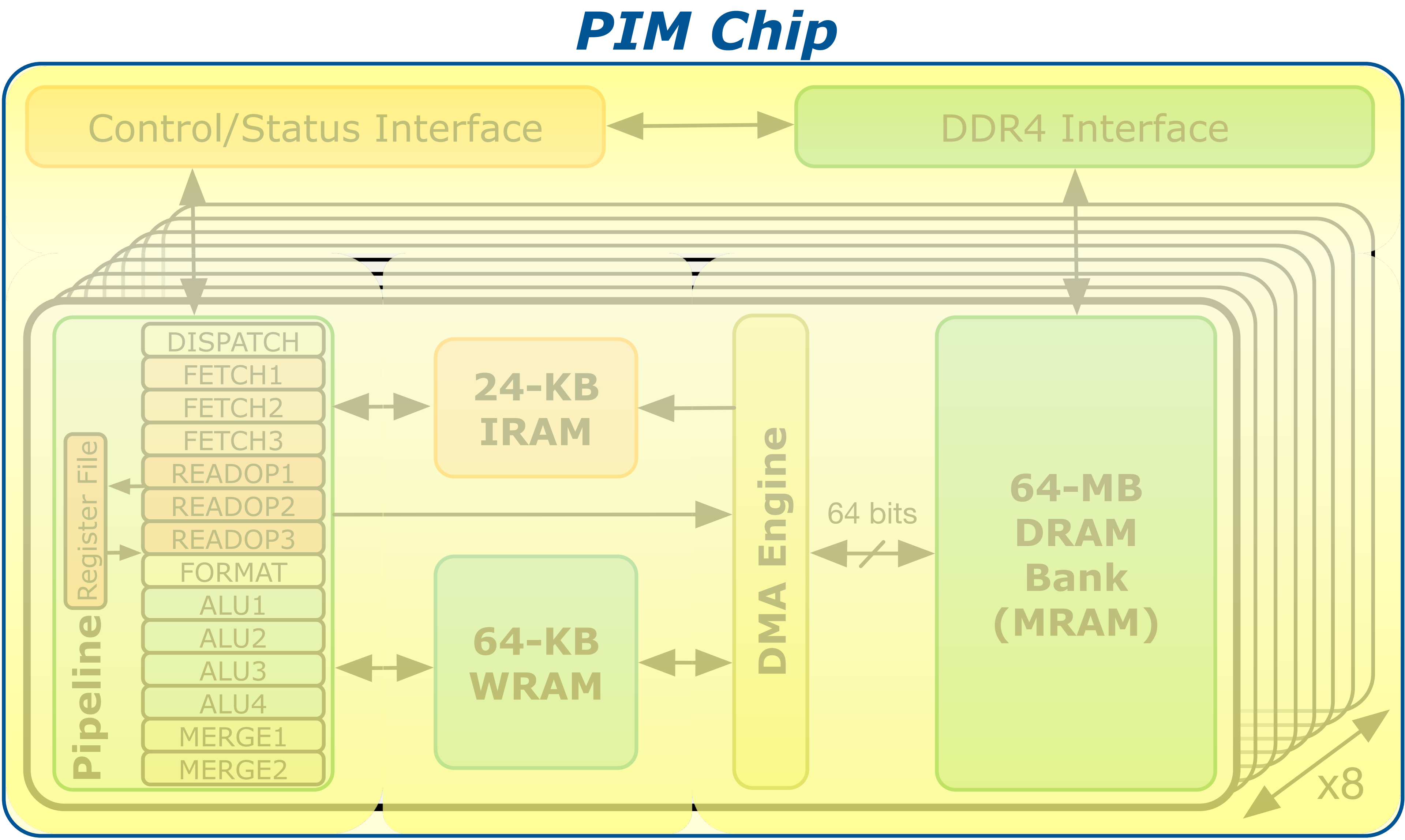
DRAM Processing Unit (I)

- *FIG. 4 schematically illustrates part of the computing system of FIG. 1 in more detail according to an example embodiment*



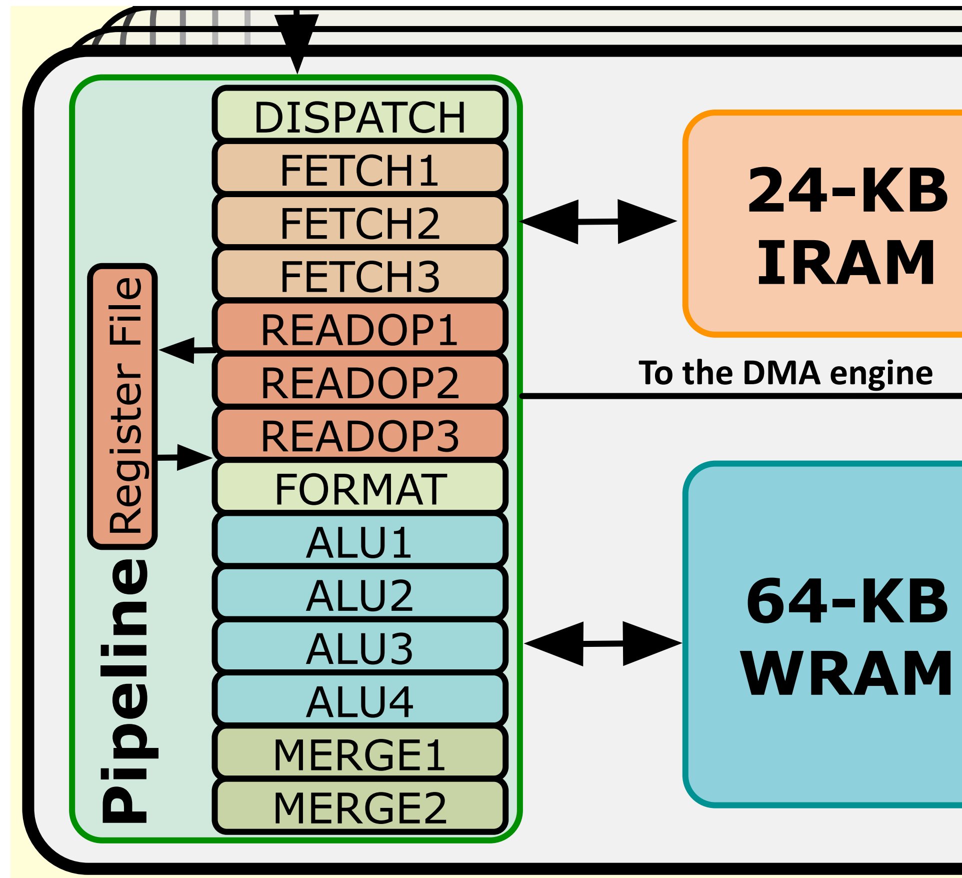
Fabrice Devaux, Jean-François Roy. "Memory circuit with integrated processor." US 10,324,870 B2.

DRAM Processing Unit (II)



DPU Pipeline

- In-order pipeline
  - Up to 350 MHz
- Fine-grain multithreaded
  - 24 hardware threads
- 14 pipeline stages
  - DISPATCH: Thread selection
  - FETCH: Instruction fetch
  - READOP: Register file
  - FORMAT: Operand formatting
  - ALU: Operation and WRAM
  - MERGE: Result formatting



Arithmetic Throughput: Microbenchmark

- Goal
  - Measure the maximum arithmetic throughput for different datatypes and operations

- Microbenchmark
  - We stream over an array in WRAM and perform read-modify-write operations
  - Experiments on one DPU
  - We vary the number of tasklets from 1 to 24
  - Arithmetic operations: add, subtract, multiply, divide
  - Datatypes: int32, int64, float, double

- We measure cycles with an accurate cycle counter that the SDK provides
  - We include WRAM accesses (including address calculation) and arithmetic operation

Microbenchmark for INT32 ADD Throughput

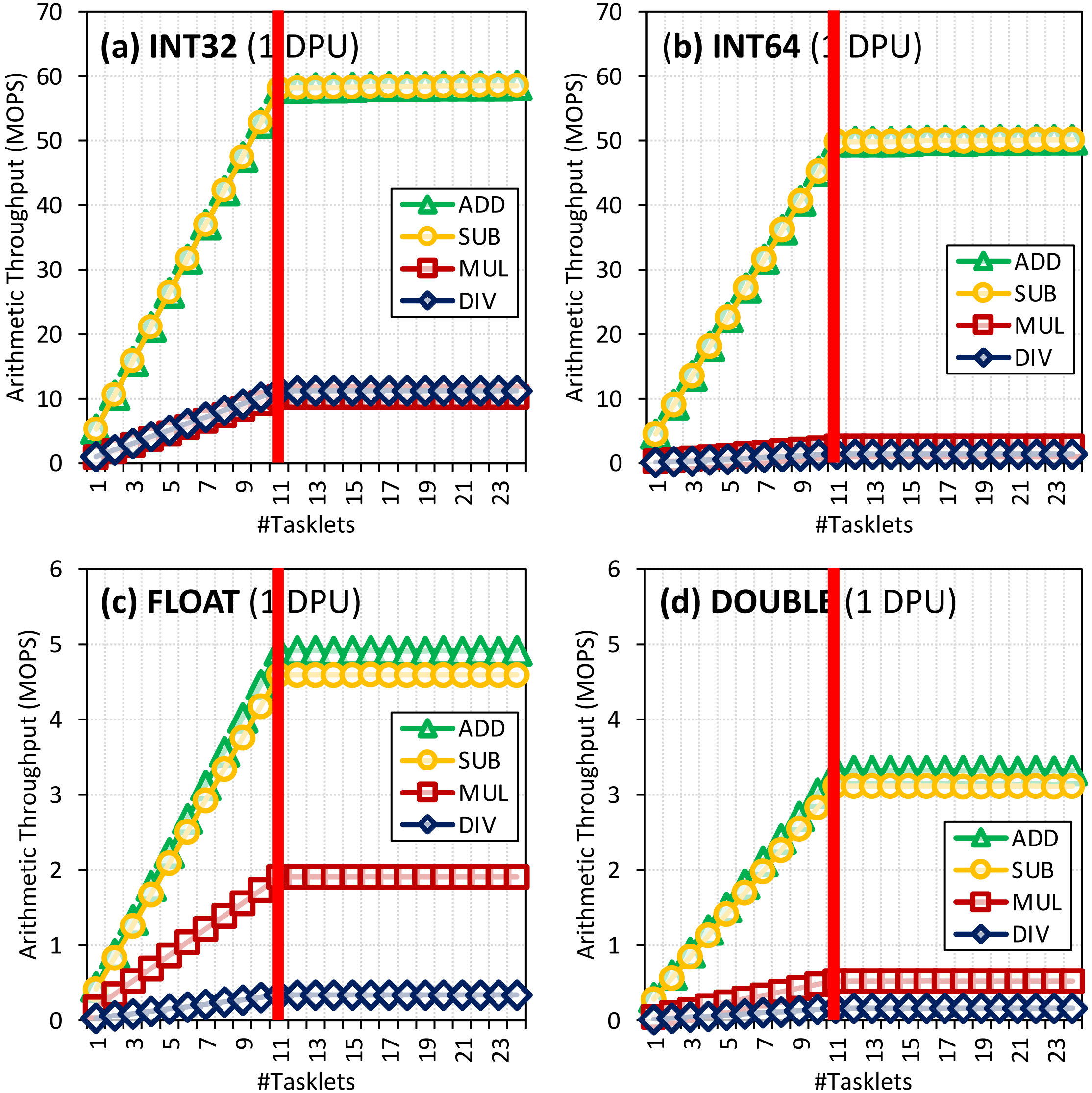
C-based code

```c
#define SIZE 256
int* bufferA = mem_alloc(SIZE * sizeof(int));
for(int i = 0; i < SIZE; i++){
    int temp = bufferA[i];
    temp += scalar;
    bufferA[i] = temp;
}
```

Compiled code (UPMEM DPU ISA)

```
  move r2, 0
.LBB0_1:                       // Loop header
  lsl_add r3, r0, r2, 2        // Address calculation
  lw r4, r3, 0                 // Load from WRAM
  add r4, r4, r1               // Add
  sw r3, 0, r4                 // Store to WRAM
  add r2, r2, 1                // Index update
  jneq r2, 256, .LBB0_1        // Conditional jump
```

Arithmetic Throughput: 11 Tasklets

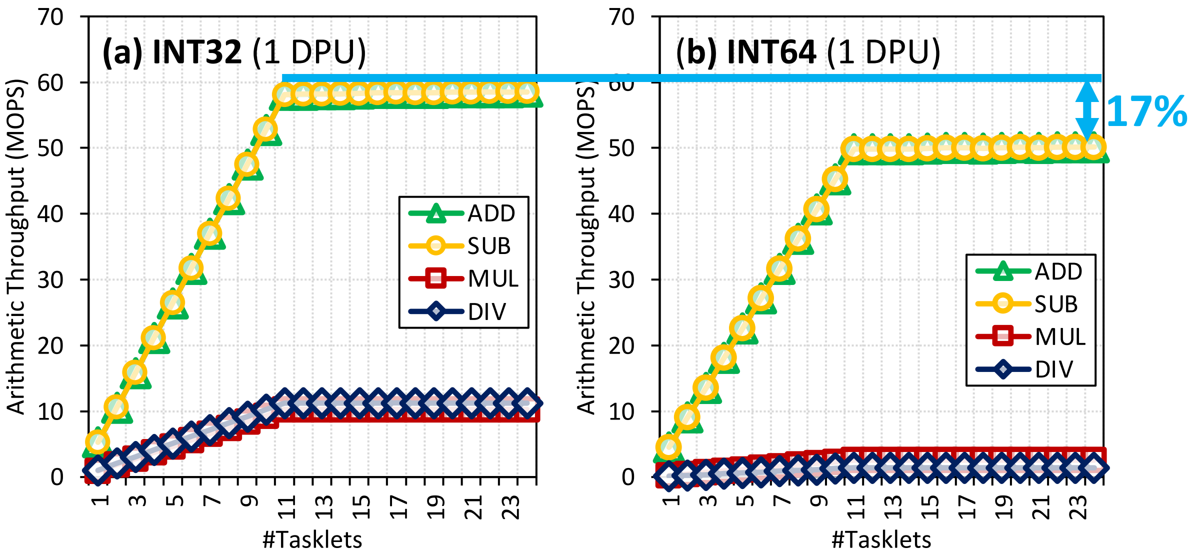


KEY OBSERVATION 1

The arithmetic throughput of a DRAM Processing Unit saturates at 11 or more tasklets.

This observation is consistent for different datatypes (INT32, INT64, UINT32, UINT64, FLOAT, DOUBLE) and operations (ADD, SUB, MUL, DIV).

Arithmetic Throughput: ADD/SUB



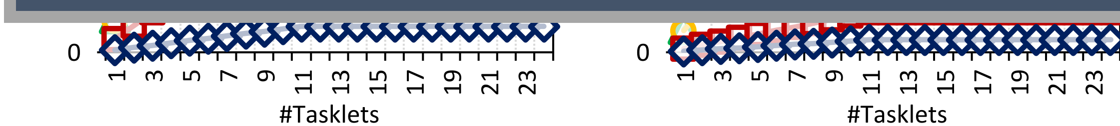
INT32 ADD/SUB are 17% faster than INT64 ADD/SUB

Can we explain the peak throughput?

Peak throughput at 11 tasklets.

One instruction retires every cycle when the pipeline is full

$$Arithmetic\ Throughput\ (in\ OPS) = \frac{frequency_{DPU}}{\#instructions}$$



Arithmetic Throughput: #Instructions

- Compiler explorer: https://dpu.dev

```c
#define BLOCK_SIZE 1024

typedef int T;
void Benchmark__32bits(T *cache_A, T scalar) {
    for (int i = 0; i < BLOCK_SIZE / sizeof(T); i++){
        ////// WRAM READ //////
        T temp = cache_A[i];

        temp += scalar; // ADD

        ////// WRAM WRITE //////
        cache_A[i] = temp;
    }
}

typedef long T_long;
void Benchmark__64bits(T_long *cache_A, T_long scalar) {
    for (int i = 0; i < BLOCK_SIZE / sizeof(T_long); i++){
        ////// WRAM READ //////
        T_long temp = cache_A[i];

        temp += scalar; // ADD
```

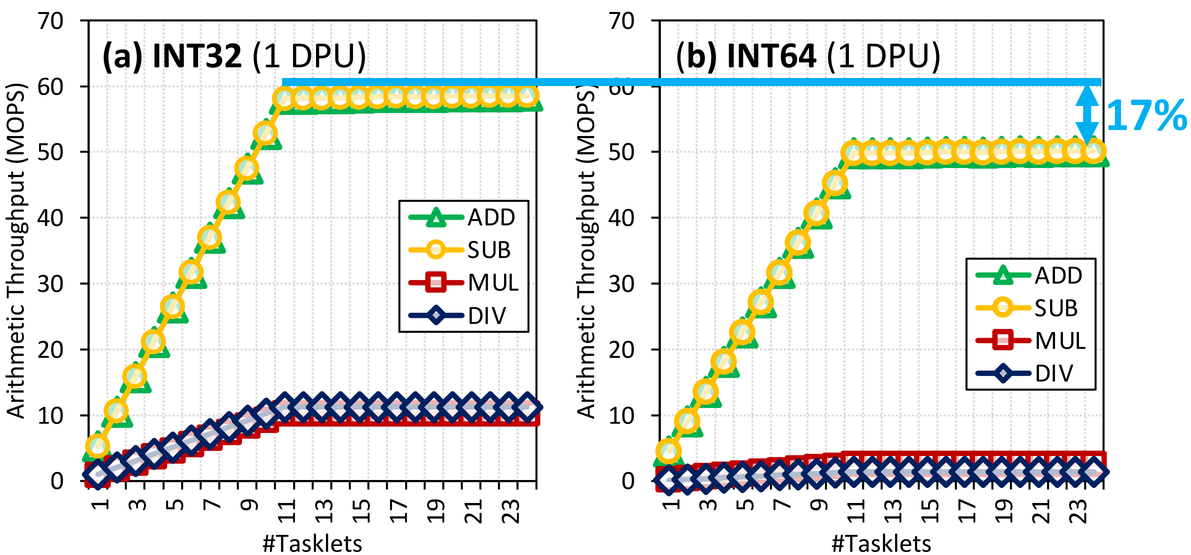
A ▾ ☐ 11010 ☐ ./a.out ☑ .LX0: ☑ .text ☑ //

```
Benchmark__32bits:
        move r2, 0
.LBB0_1:
        lsl_add r3, r0, r2, 2
        lw r4, r3, 0
        add r4, r4, r1
        sw r3, 0, r4
        add r2, r2, 1
        jneq r2, 256, .LBB0_1
        jump r23
Benchmark__64bits:
        move r1, 0
.LBB1_1:
        lsl_add r4, r0, r1, 3
        ld d6, r4, 0
        add r7, r7, r3
        addc r6, r6, r2
        sd r4, 0, d6
        add r1, r1, 1
        jneq r1, 128, .LBB1_1
        jump r23
```

6 instructions in the 32-bit ADD/SUB microbenchmark

7 instructions in the 64-bit ADD/SUB microbenchmark

Arithmetic Throughput: ADD/SUB



INT32 ADD/SUB are 17% faster than INT64 ADD/SUB

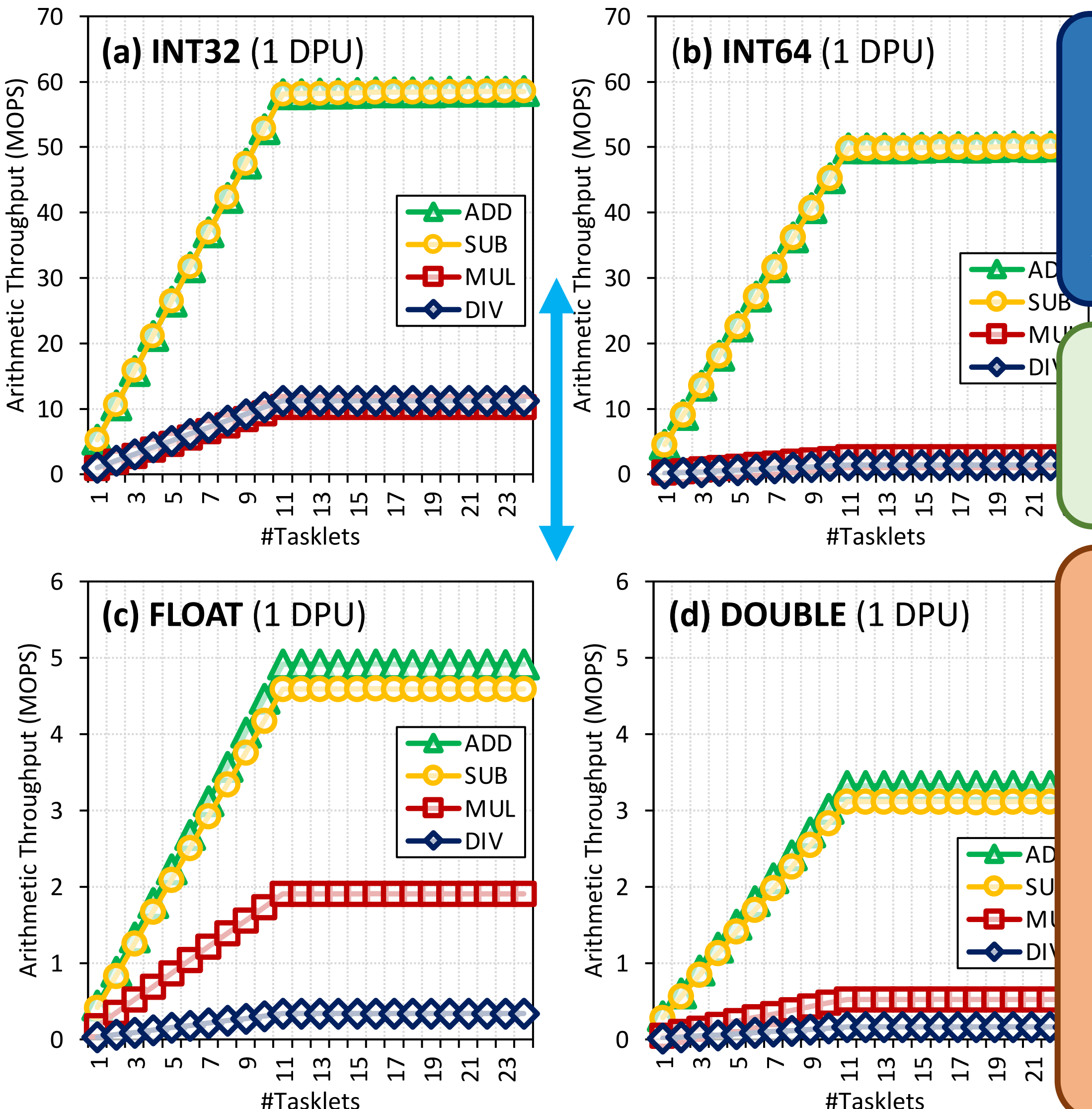
Can we explain the peak throughput?

Peak throughput at 11 tasklets.
One instruction retires every cycle when the pipeline is full

$$Arithmetic\ Throughput\ (in\ OPS) = \frac{frequency_{DPU}}{\#instructions}$$

64-bit ADD/SUB: 7 instructions → 50.00 MOPS
at $frequency_{DPU}$ = 350 MHz

Arithmetic Throughput: MUL/DIV

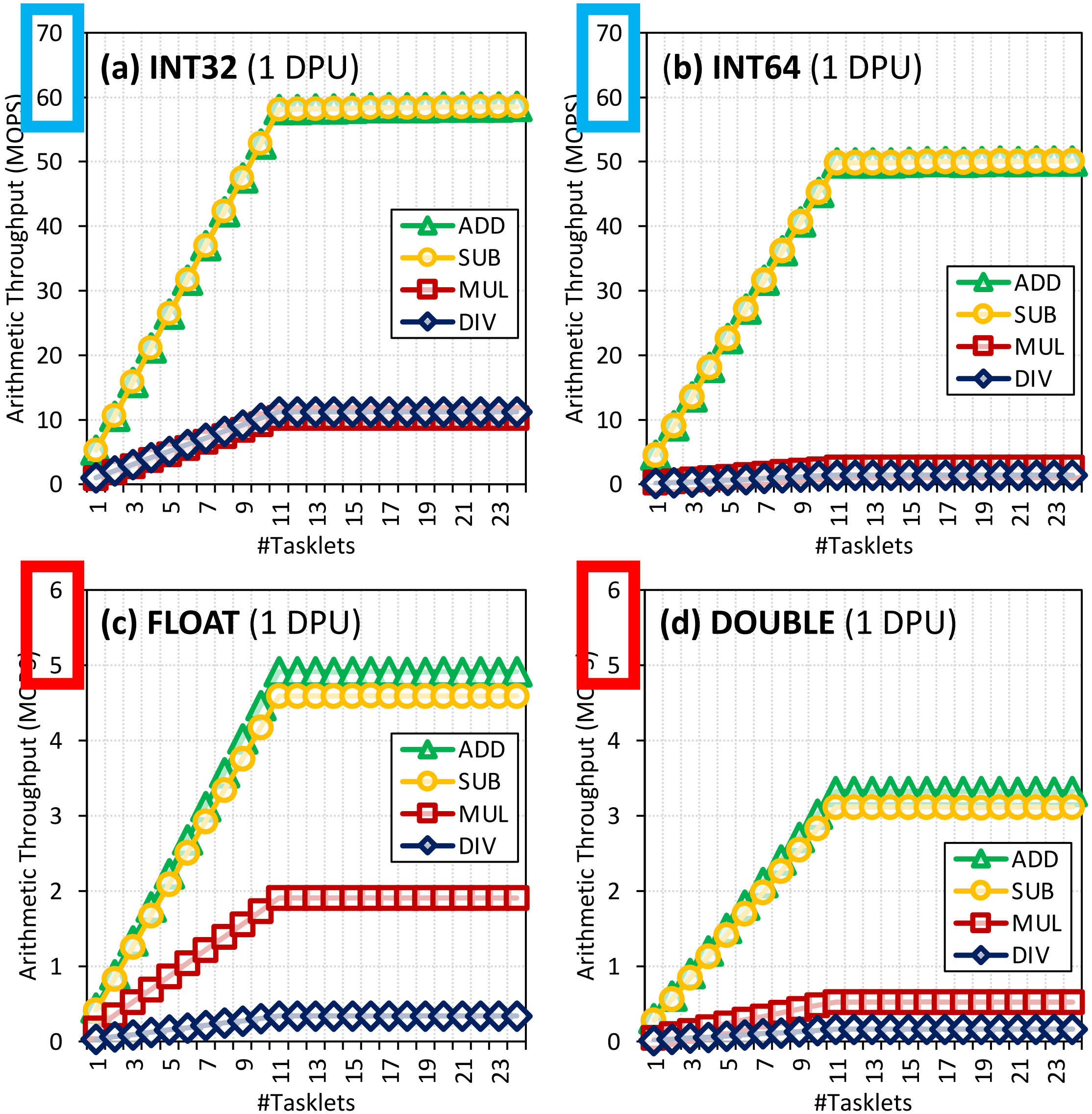


Huge throughput difference between ADD/SUB and MUL/DIV

DPUs do *not* have a 32-bit multiplier

MUL/DIV implementation is based on an instruction that performs bit shifting and addition in 1 cycle (MUL/DIV take a maximum of 32 instructions)

Arithmetic Throughput: Native Support



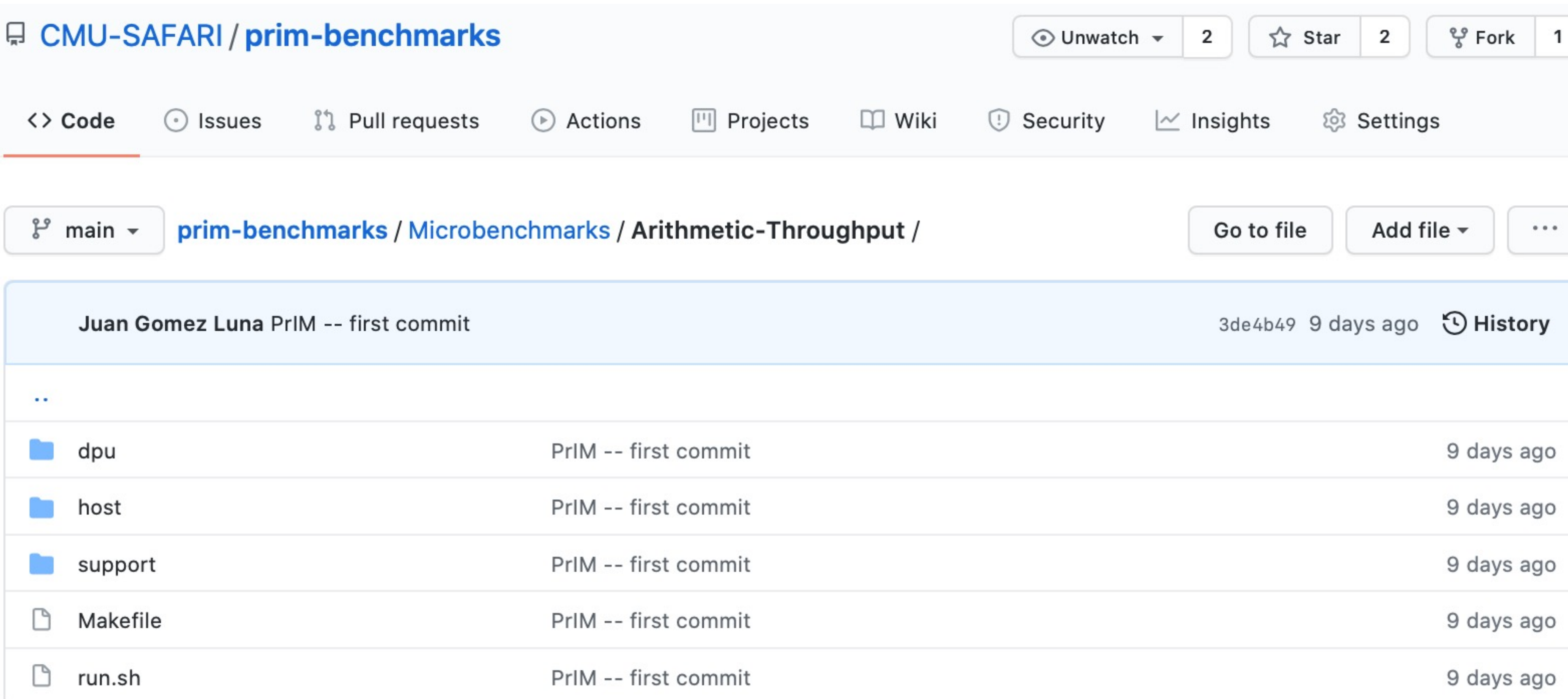
KEY OBSERVATION 2

- DPUs provide **native hardware support for 32- and 64-bit integer addition and subtraction**, leading to high throughput for these operations.

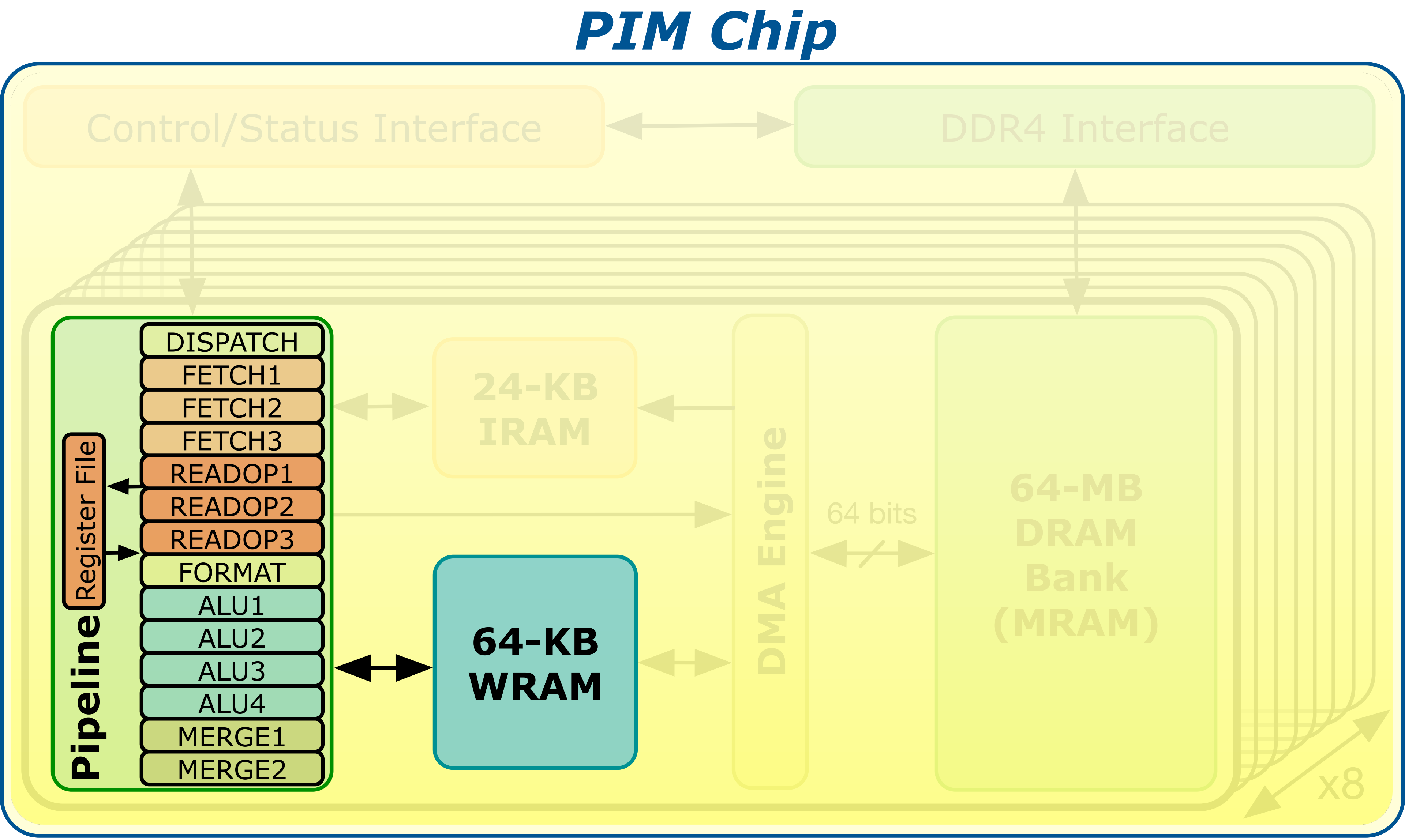
- DPUs do ***not*** **natively support 32- and 64-bit multiplication and division, and floating point operations**. These operations are **emulated by the UPMEM runtime library**, leading to much lower throughput.

Microbenchmark: Arithmetic Throughput

- Arithmetic throughput for different operations and datatypes



DPU: WRAM Bandwidth



WRAM Bandwidth: Microbenchmark

- Goal
  - Measure the WRAM bandwidth for the STREAM benchmark

- Microbenchmark
  - We implement the four versions of STREAM: COPY, ADD, SCALE, and TRIAD
  - The operations performed in ADD, SCALE, and TRIAD are addition, multiplication, and addition+multiplication, respectively
  - We vary the number of tasklets from 1 to 16
  - We show results for 1 DPU

- We do *not* include accesses to MRAM

STREAM Benchmark in WRAM

```
// COPY
for(int i = 0; i < SIZE; i++){
    bufferB[i] = bufferA[i];
}
```

8 bytes read, 8 bytes written, no arithmetic operations

```
// ADD
for(int i = 0; i < SIZE; i++){
    bufferC[i] = bufferA[i] + bufferB[i];
}
```

16 bytes read, 8 bytes written, ADD

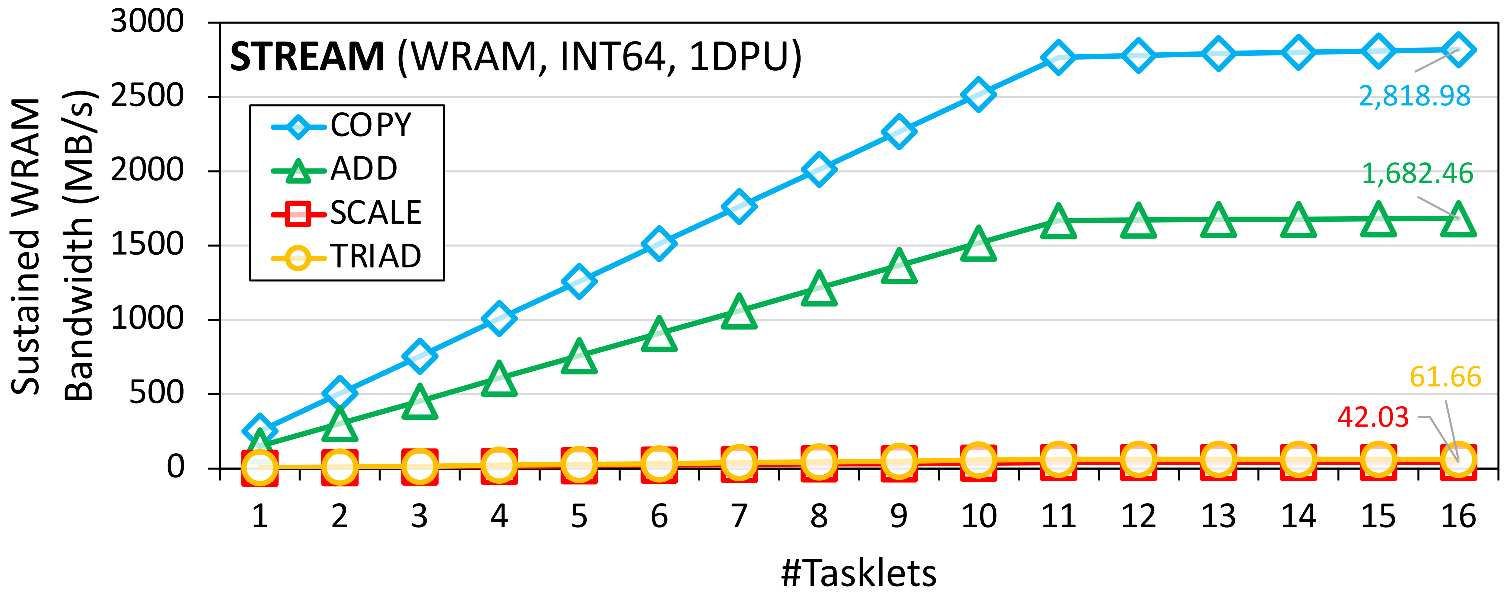
```
// SCALE
for(int i = 0; i < SIZE; i++){
    bufferB[i] = scalar * bufferA[i];
}
```

8 bytes read, 8 bytes written, MUL

```
// TRIAD
for(int i = 0; i < SIZE; i++){
    bufferC[i] = bufferA[i] + scalar * bufferB[i];
}
```

16 bytes read, 8 bytes written, MUL, ADD

WRAM Bandwidth: STREAM

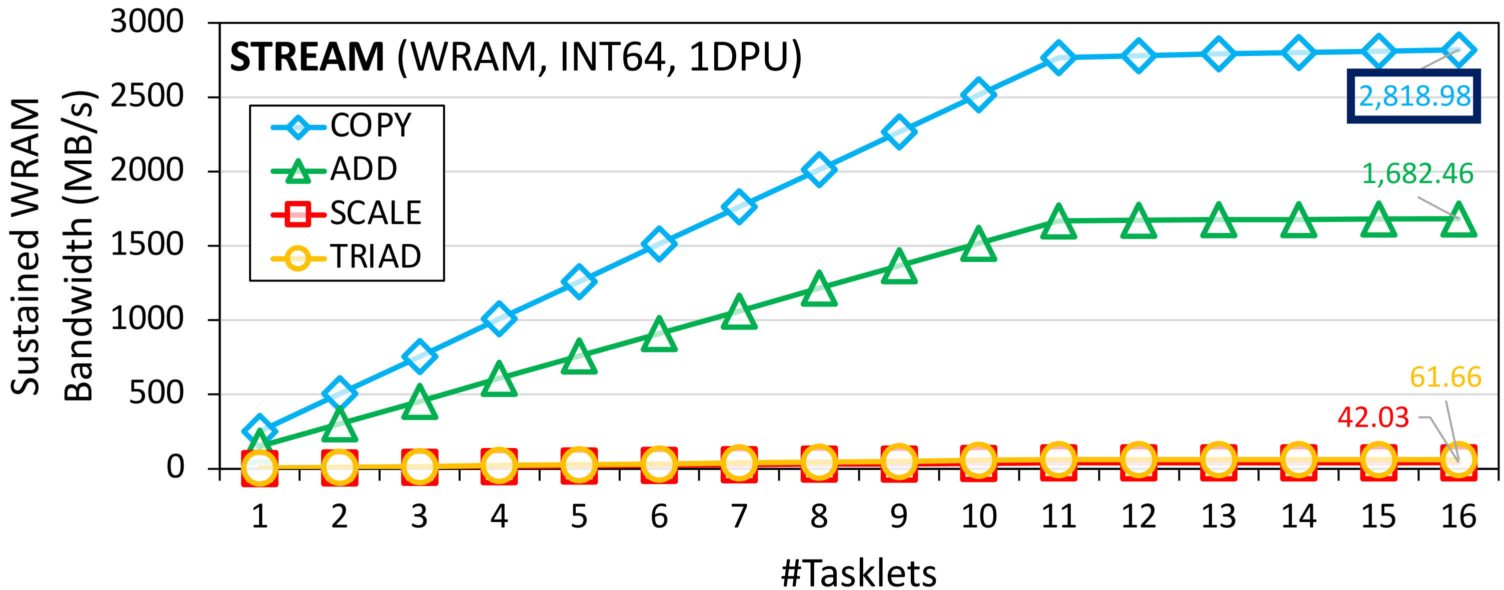


How can we estimate the bandwidth?

Assuming that the pipeline is full, and *Bytes* is the number of bytes read and written:

$$WRAM\ Bandwidth\ \left(in\ \frac{B}{S}\right) = \frac{Bytes \times frequency_{DPU}}{\#instructions}$$

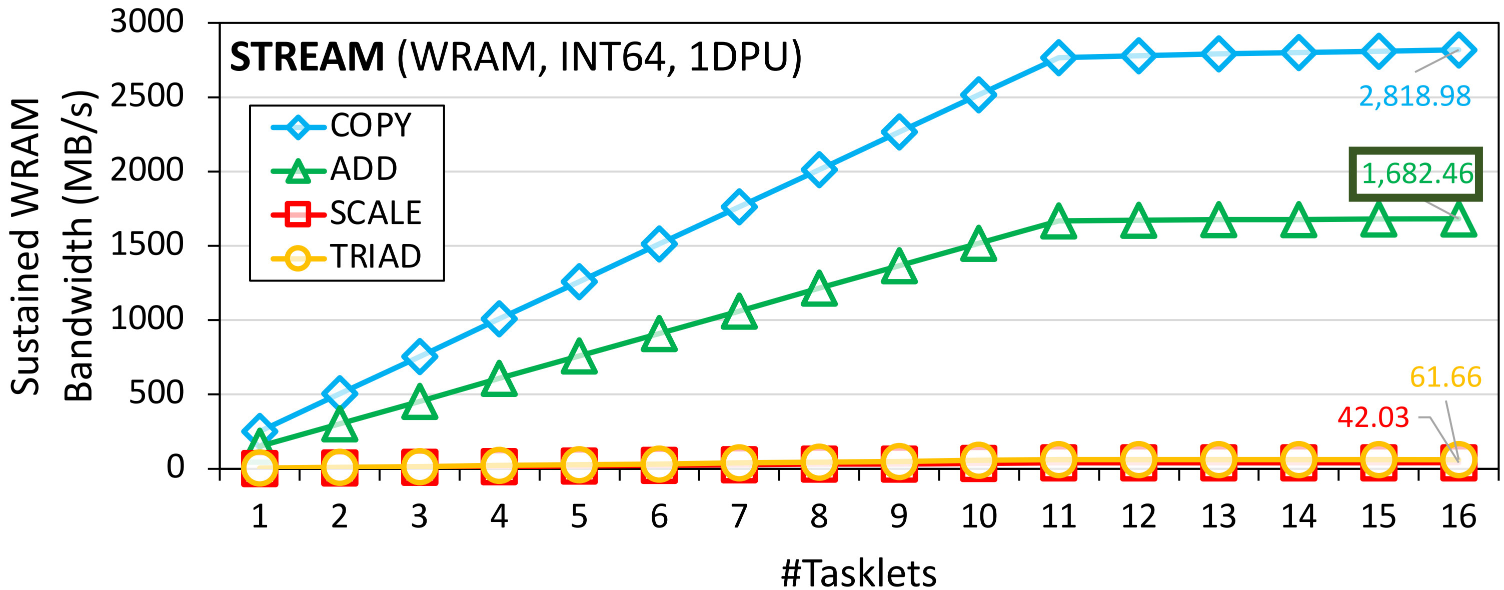
WRAM Bandwidth: COPY



COPY executes 2 instructions (WRAM load and store).
With 11 tasklets, 11 × 16 bytes in 22 cycles:

$$WRAM\ Bandwidth\ \left(in\,\frac{B}{S}\right) = 2{,}800\,\frac{MB}{s}\,at\ 350\ MHz$$

WRAM Bandwidth: ADD



$$WRAM\ Bandwidth\ \left(in\ \frac{B}{S}\right) = \frac{Bytes \times frequency_{DPU}}{\#instructions}$$

ADD executes 5 instructions (2 `ld`, `add`, `addc`, `sd`).
With 11 tasklets, 11 × 24 bytes in 55 cycles:

$$WRAM\ Bandwidth\ \left(in\ \frac{B}{S}\right) = 1{,}680\ \frac{MB}{s}\ at\ 350\ MHz$$

WRAM Bandwidth: Access Patterns

- All 8-byte WRAM loads and stores take one cycle when the DPU pipeline is full

KEY OBSERVATION 3

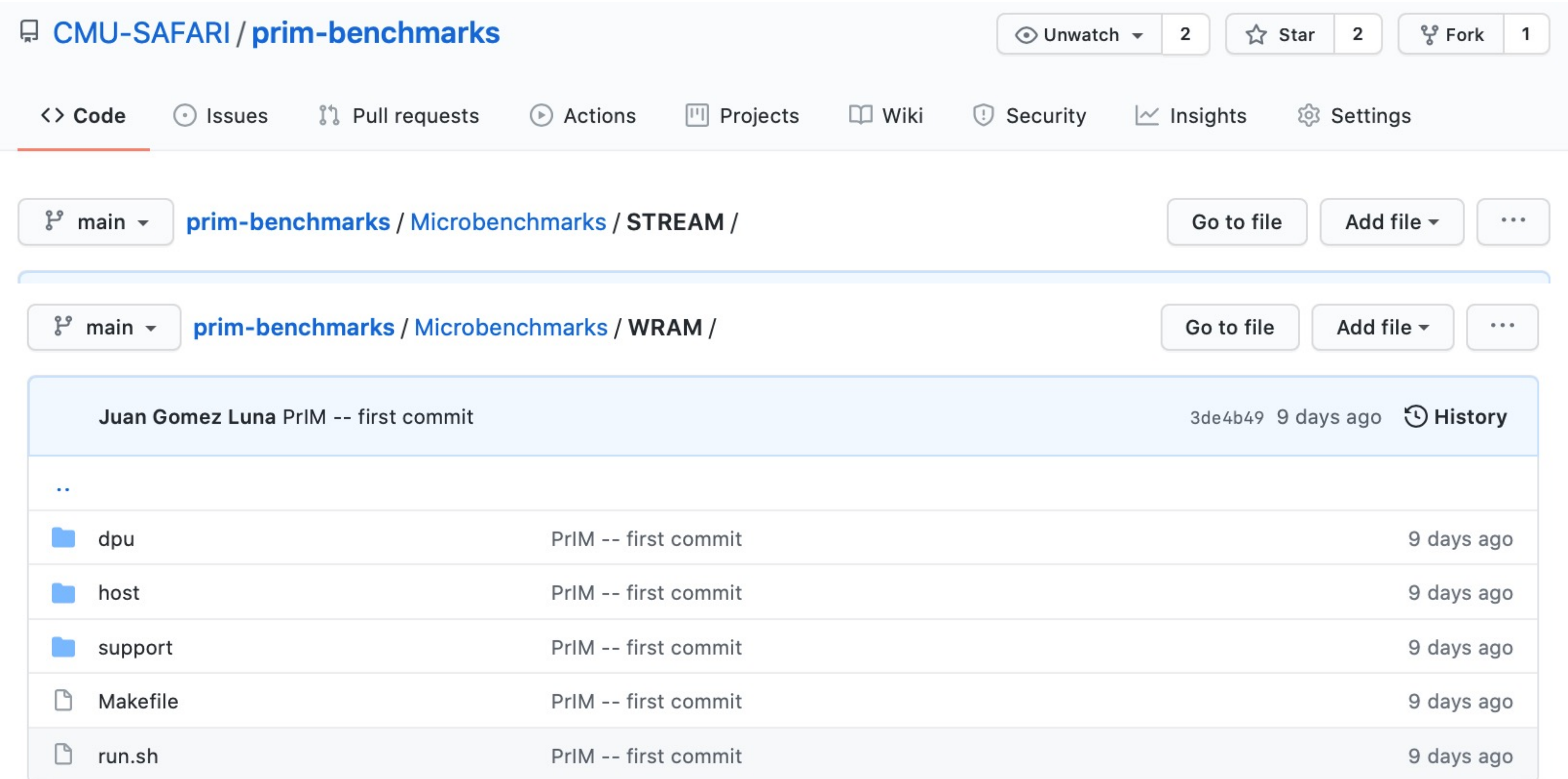
The sustained bandwidth provided by the DPU's internal Working memory (WRAM) is **independent of the memory access pattern** (either streaming, strided, or random access pattern).

All 8-byte WRAM loads and stores take one cycle, when the DPU's pipeline is full (i.e., with 11 or more tasklets).

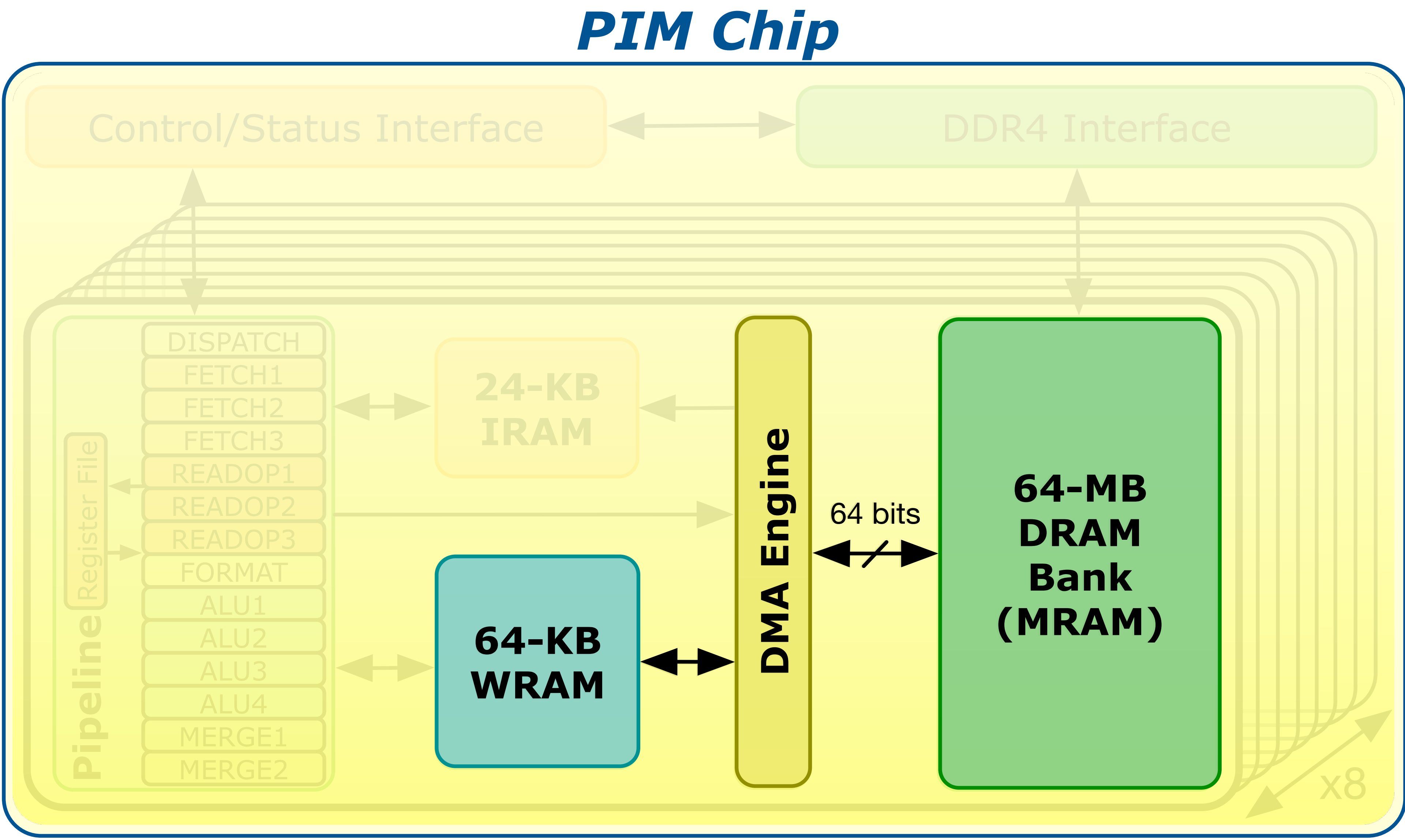
- Microbenchmark: `c[a[i]]=b[a[i]];`
  - Unit-stride: `a[i]=a[i-1]+1;`
  - Strided: `a[i]=a[i-1]+stride;`
  - Random: `a[i]=rand();`

Microbenchmark: STREAM and WRAM

- STREAM benchmark and WRAM access patterns



DPU: MRAM Latency and Bandwidth



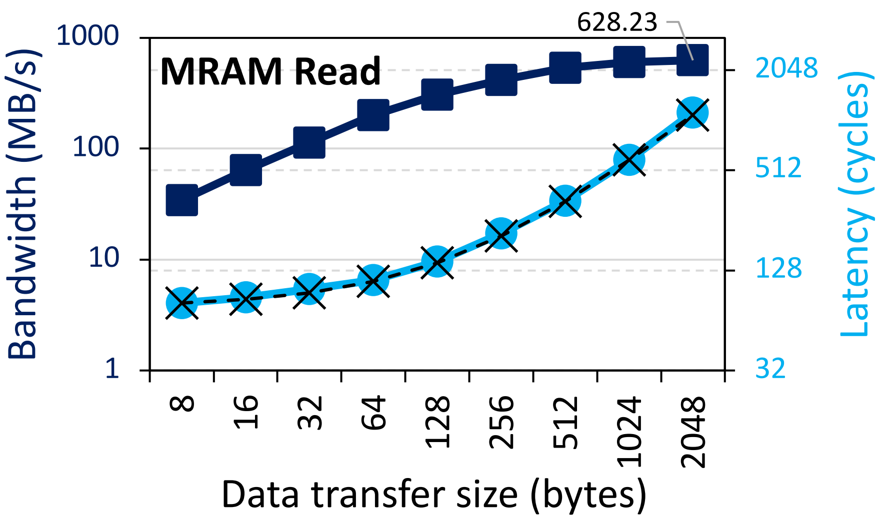
MRAM Bandwidth

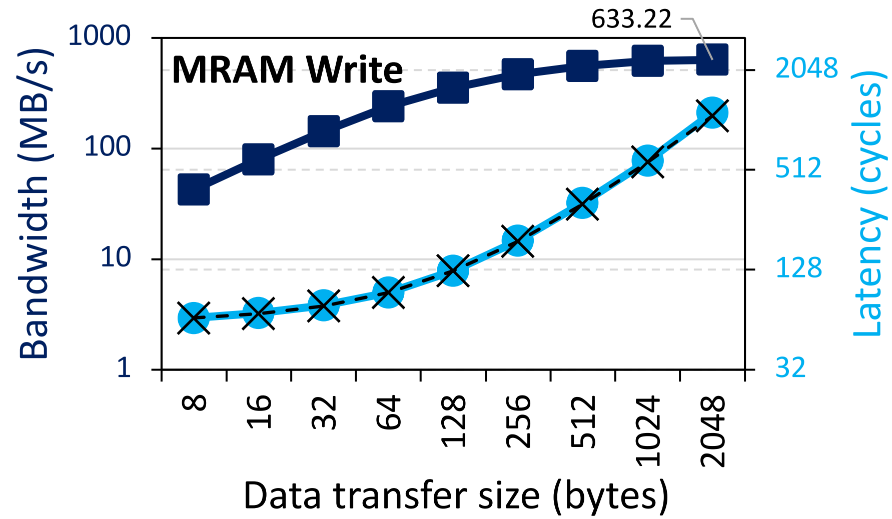
- Goal
  - Measure MRAM bandwidth for different access patterns

- Microbenchmarks
  - Latency of a single DMA transfer for different transfer sizes
    - `mram_read(); // MRAM-WRAM DMA transfer`
    - `mram_write(); // WRAM-MRAM DMA transfer`
  - STREAM benchmark
    - COPY, COPY-DMA
    - ADD, SCALE, TRIAD
  - Strided access pattern
    - Coarse-grain strided access
    - Fine-grain strided access
  - Random access pattern (GUPS)

- We do include accesses to MRAM

MRAM Read and Write Latency (I)





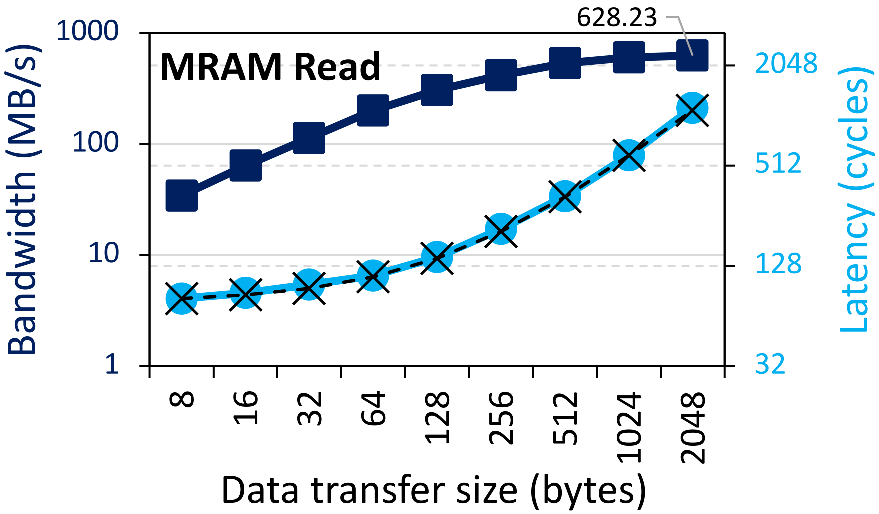
$$MRAM\ Bandwidth\ \left(in\ \frac{B}{S}\right) = \frac{size \times frequency_{DPU}}{MRAM\ Latency}$$

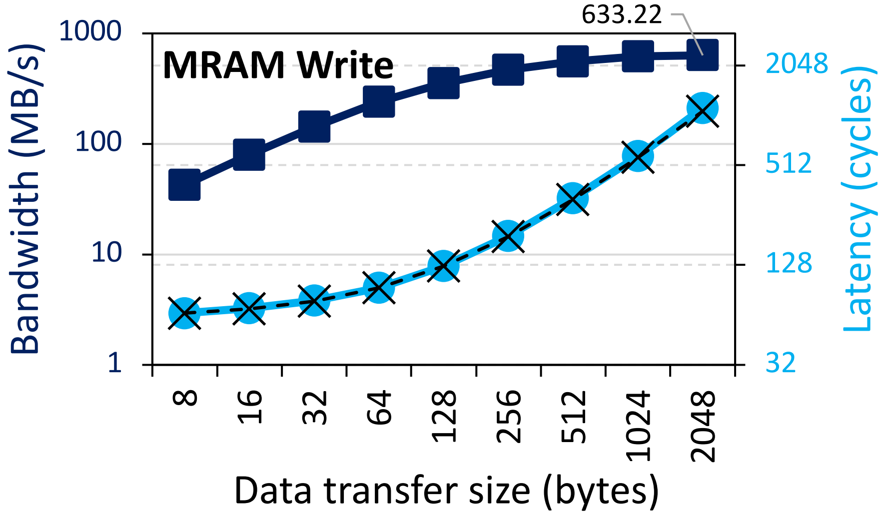
We can model the MRAM latency with a linear expression

$$MRAM\ Latency\ (in\ cycles) = \alpha + \beta \times size$$

In our measurements, β equals 0.5 cycles/byte.
Theoretical maximum MRAM bandwidth = 700 MB/s at 350 MHz

MRAM Read and Write Latency (II)

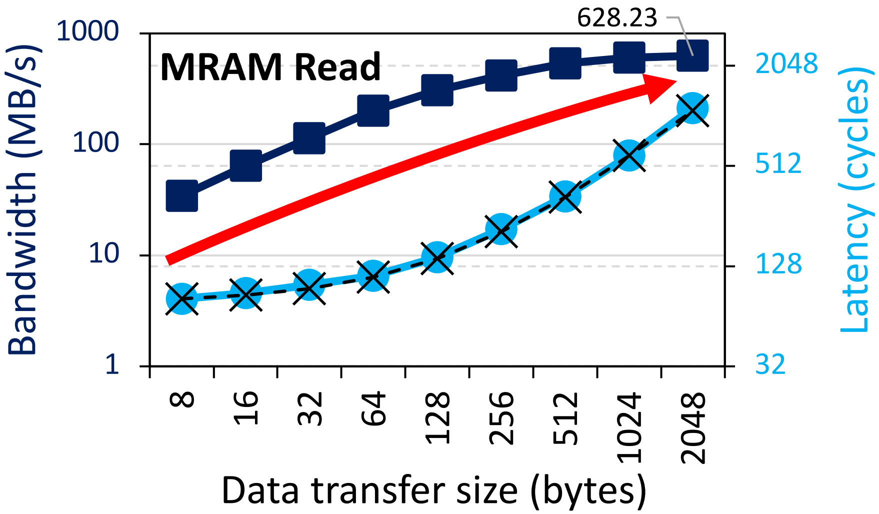


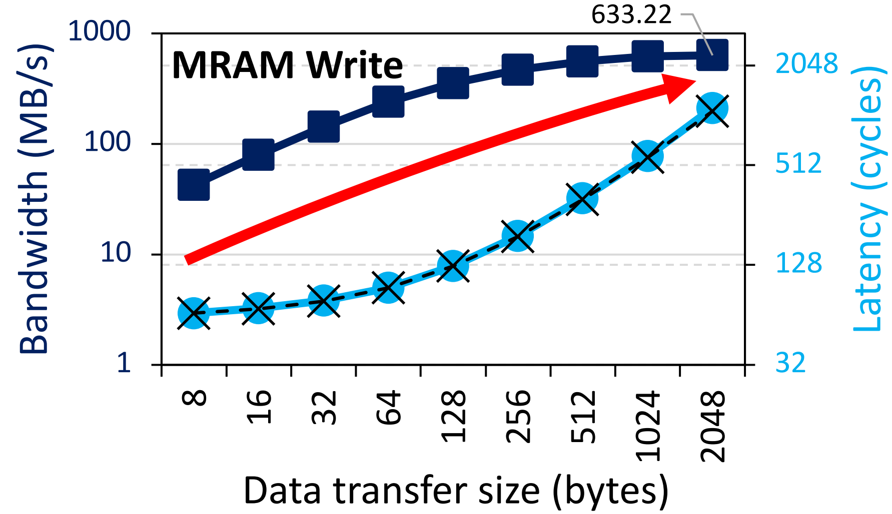


KEY OBSERVATION 4

- The DPU's **Main memory (MRAM) bank access latency increases linearly** with the transfer size.
- The maximum theoretical MRAM **bandwidth is 2 bytes per cycle**.

MRAM Read and Write Latency (III)





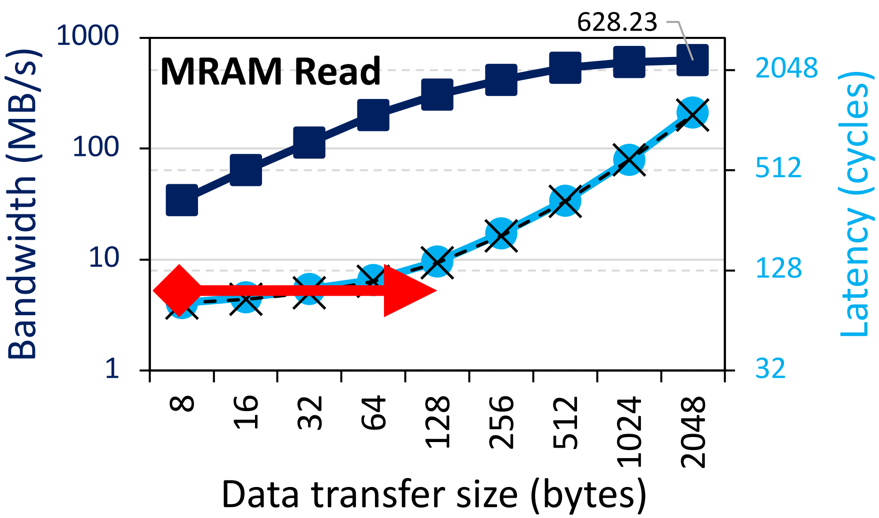
Read and write accesses to MRAM are symmetric

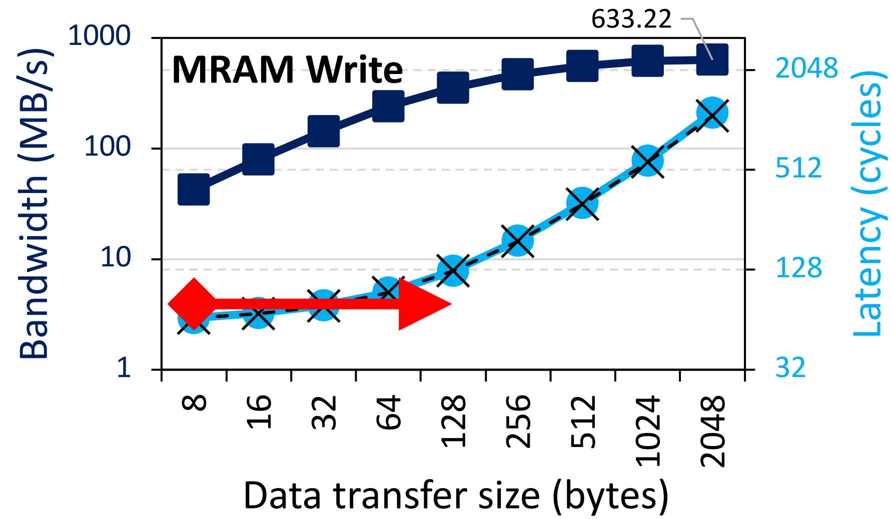
The sustained MRAM bandwidth increases with data transfer size

PROGRAMMING RECOMMENDATION 1

For data movement between the DPU's MRAM bank and the WRAM, **use large DMA transfer sizes when all the accessed data is going to be used**.

MRAM Read and Write Latency (IV)





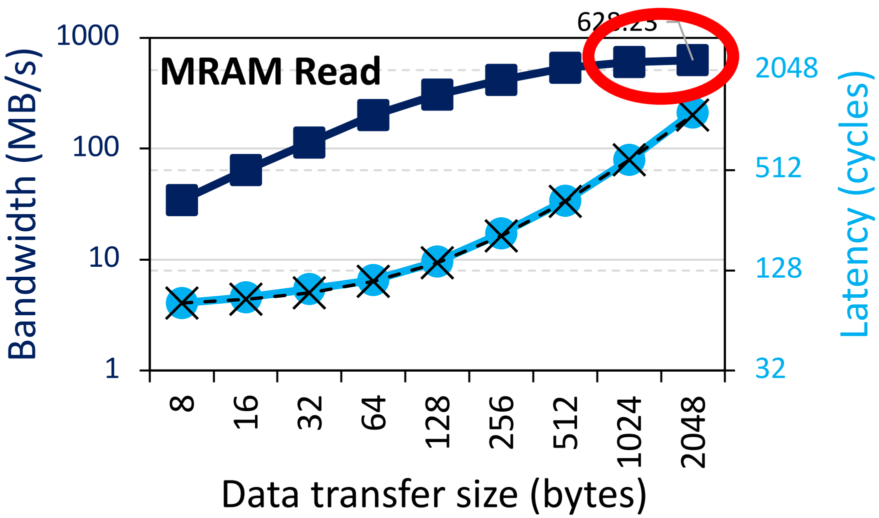
MRAM latency changes slowly between 8 and 128 bytes

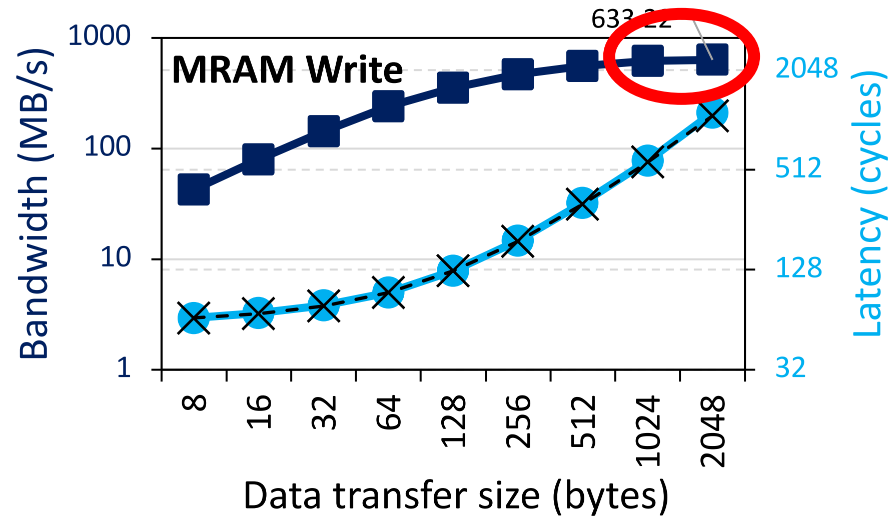
For small transfers, the fixed cost (α) dominates the variable cost ($\beta \times size$)

PROGRAMMING RECOMMENDATION 2

For small transfers between the MRAM bank and the WRAM, **fetch more bytes than necessary within a 128-byte limit**. Doing so increases the likelihood of finding data in WRAM for later accesses (i.e., the program can check whether the desired data is in WRAM before issuing a new MRAM access).

MRAM Read and Write Latency (V)





2,048-byte transfers are only 4% faster than 1,024-byte transfers

Larger transfers require more WRAM, which may limit the number of tasklets

PROGRAMMING RECOMMENDATION 3

Choose the data transfer size between the MRAM bank and the WRAM based on the program's WRAM usage, as it imposes a tradeoff between the sustained MRAM bandwidth and the number of tasklets that can run in the DPU (which is dictated by the limited WRAM capacity).

MRAM Bandwidth

- Goal
  - Measure MRAM bandwidth for different access patterns

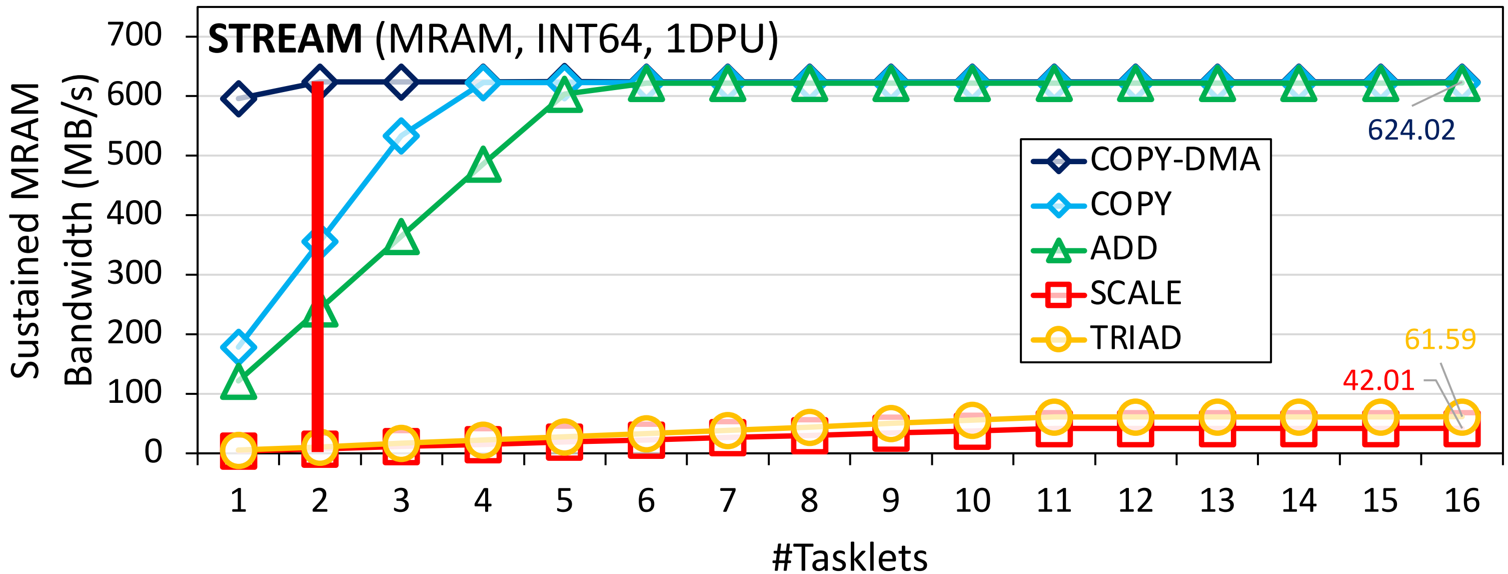
- Microbenchmarks
  - Latency of a single DMA transfer for different transfer sizes
    - `mram_read();   // MRAM-WRAM DMA transfer`
    - `mram_write(); // WRAM-MRAM DMA transfer`
  - STREAM benchmark
    - COPY, COPY-DMA
    - ADD, SCALE, TRIAD
  - Strided access pattern
    - Coarse-grain strided access
    - Fine-grain strided access
  - Random access pattern (GUPS)

- We do include accesses to MRAM

STREAM Benchmark in MRAM

```
// COPY
// Load current MRAM block to WRAM
mram_read((__mram_ptr void const*)mram_address_A, bufferA,
               SIZE * sizeof(uint64_t));

for(int i = 0; i < SIZE; i++){
     bufferB[i] = bufferA[i];
}

// Write WRAM block to MRAM
mram_write(bufferB, (__mram_ptr void*)mram_address_B,
               SIZE * sizeof(uint64_t));

// COPY-DMA
// Load current MRAM block to WRAM
mram_read((__mram_ptr void const*)mram_address_A, bufferA,
               SIZE * sizeof(uint64_t));

// Write WRAM block to MRAM
mram_write(bufferB, (__mram_ptr void*)mram_address_B,
               SIZE * sizeof(uint64_t));
```

STREAM Benchmark: COPY-DMA

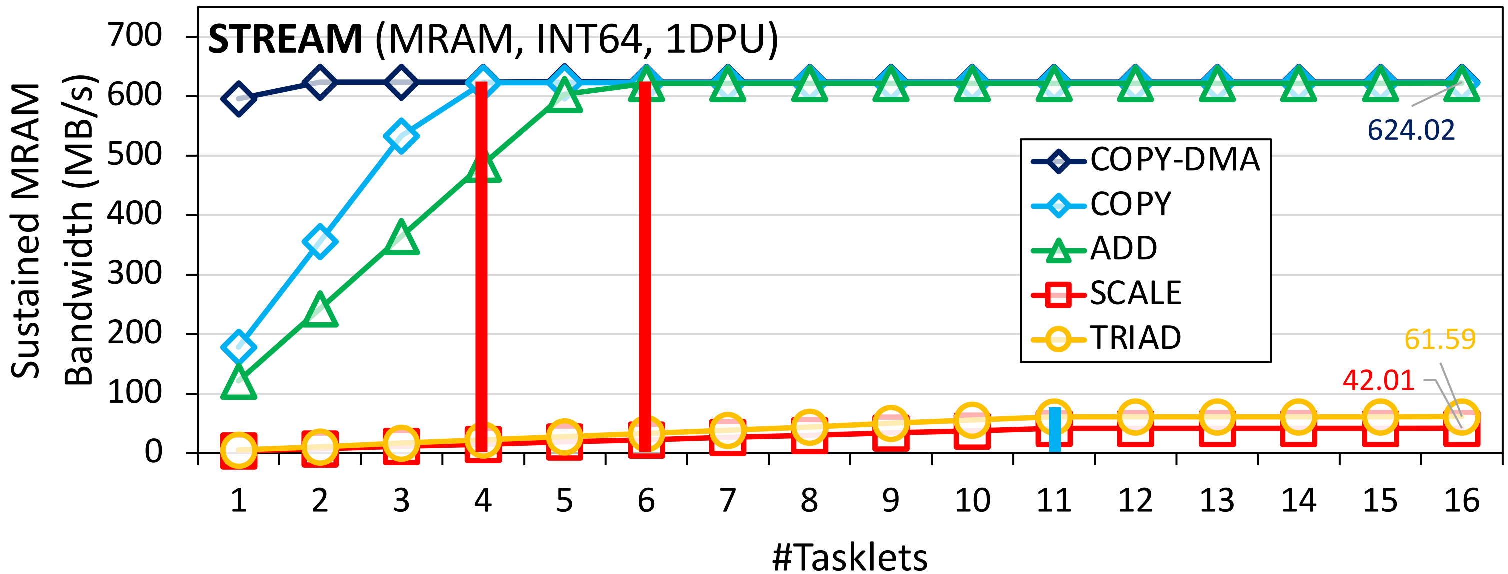


The sustained bandwidth of **COPY-DMA** is close to the theoretical maximum (700 MB/s): ~1.6 TB/s for 2,556 DPUs

COPY-DMA saturates with two tasklets, even though the DMA engine can perform only one transfer at a time

Using two or more tasklets guarantees that there is always a DMA request enqueued to keep the DMA engine busy

STREAM Benchmark: Bandwidth Saturation (I)



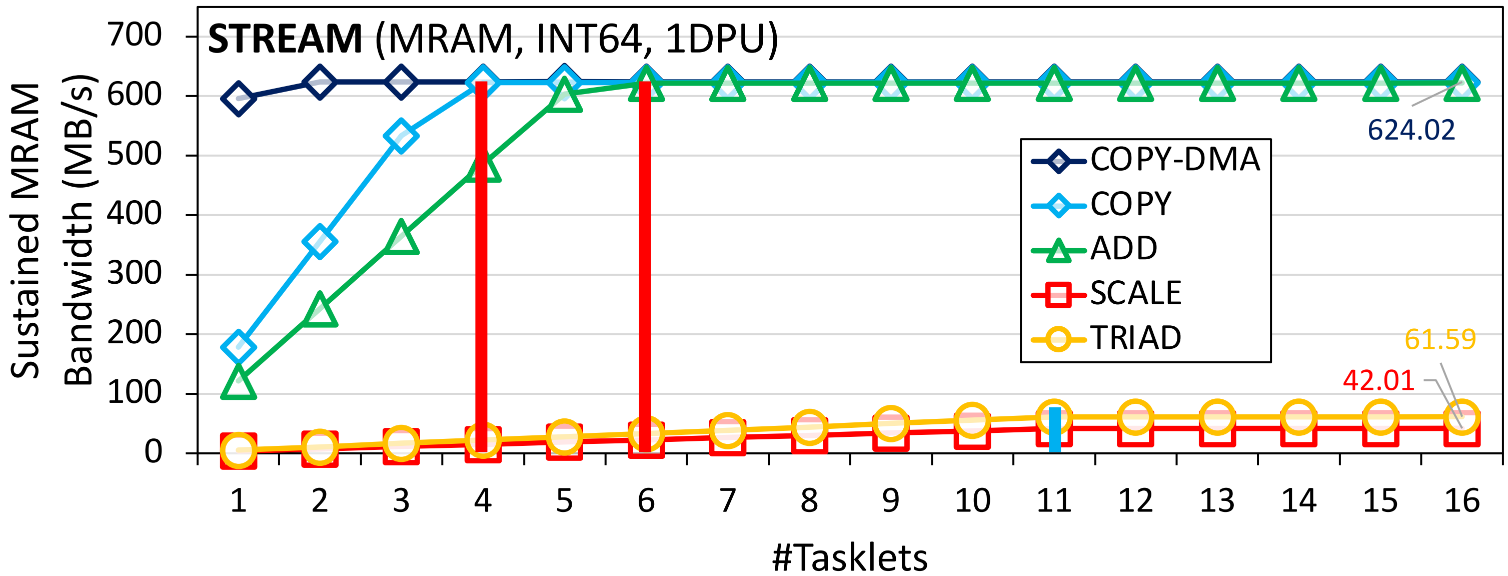
COPY and **ADD** saturate at 4 and 6 tasklets, respectively

SCALE and **TRIAD** saturate at 11 tasklets

The latency of MRAM accesses becomes longer than the pipeline latency after 4 and 6 tasklets for **COPY** and **ADD**, respectively

The pipeline latency of **SCALE** and **TRIAD** is longer than the MRAM latency for any number of tasklets (both use costly MUL)

STREAM Benchmark: Bandwidth Saturation (II)



KEY OBSERVATION 5

- **When the access latency to an MRAM bank** for a streaming benchmark (COPY-DMA, COPY, ADD) **is larger than the pipeline latency** (i.e., execution latency of arithmetic operations and WRAM accesses), the performance of the DPU saturates at a number of tasklets smaller than 11. **This is a memory-bound workload.**
- **When the pipeline latency** for a streaming benchmark (SCALE, TRIAD) **is larger than the MRAM access latency**, the performance of a DPU saturates at 11 tasklets. **This is a compute-bound workload.**

MRAM Bandwidth

- Goal
  - Measure MRAM bandwidth for different access patterns

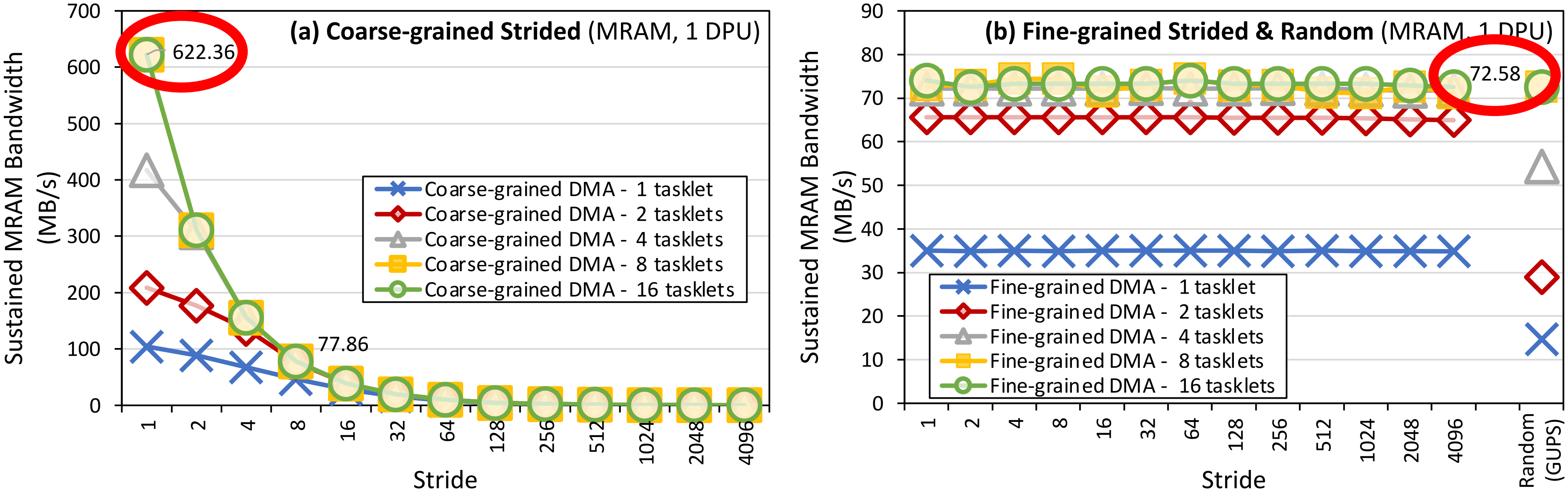
- Microbenchmarks
  - Latency of a single DMA transfer for different transfer sizes
    - `mram_read();   // MRAM-WRAM DMA transfer`
    - `mram_write(); // WRAM-MRAM DMA transfer`
  - STREAM benchmark
    - COPY, COPY-DMA
    - ADD, SCALE, TRIAD
  - Strided access pattern
    - Coarse-grain strided access
    - Fine-grain strided access
  - Random access pattern (GUPS)

- We do include accesses to MRAM

Strided and Random Access to MRAM

```
// COARSE-GRAINED STRIDED ACCESS
// Load current MRAM block to WRAM
mram_read((__mram_ptr void const*)mram_address_A, bufferA,
          SIZE * sizeof(uint64_t));
mram_read((__mram_ptr void const*)mram_address_B, bufferB,
          SIZE * sizeof(uint64_t));

for(int i = 0; i < SIZE; i += stride){
    bufferB[i] = bufferA[i];
}
// Write WRAM block to MRAM
mram_write(bufferB, (__mram_ptr void*)mram_address_B,
          SIZE * sizeof(uint64_t));


// FINE-GRAINED STRIDED & RANDOM ACCESS
for(int i = 0; i < SIZE; i += stride){
    int index = i * sizeof(uint64_t);
    // Load current MRAM element to WRAM
    mram_read((__mram_ptr void const*)(mram_address_A + index), bufferA,
              sizeof(uint64_t));

    // Write WRAM element to MRAM
    mram_write(bufferA, (__mram_ptr void*)(mram_address_B + index),
              sizeof(uint64_t));
}
```

Strided and Random Accesses (I)

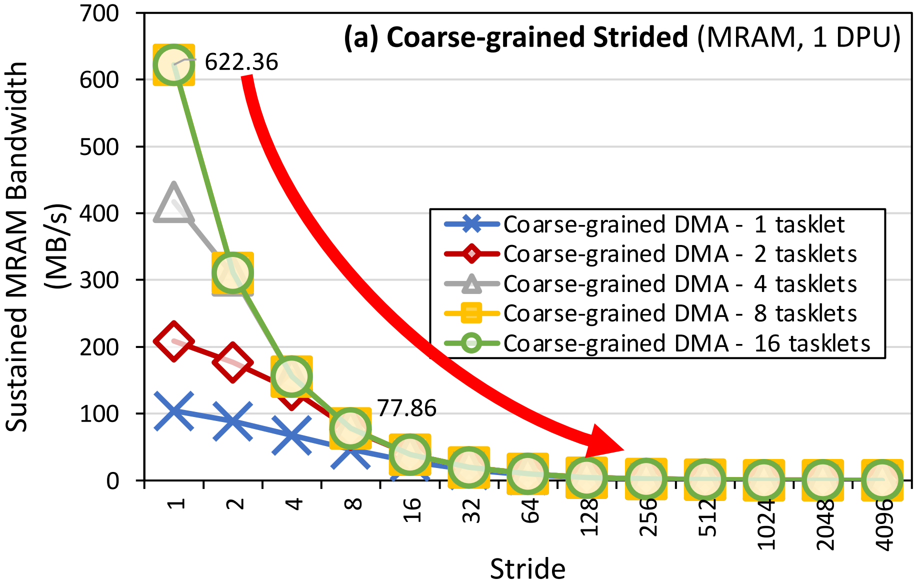


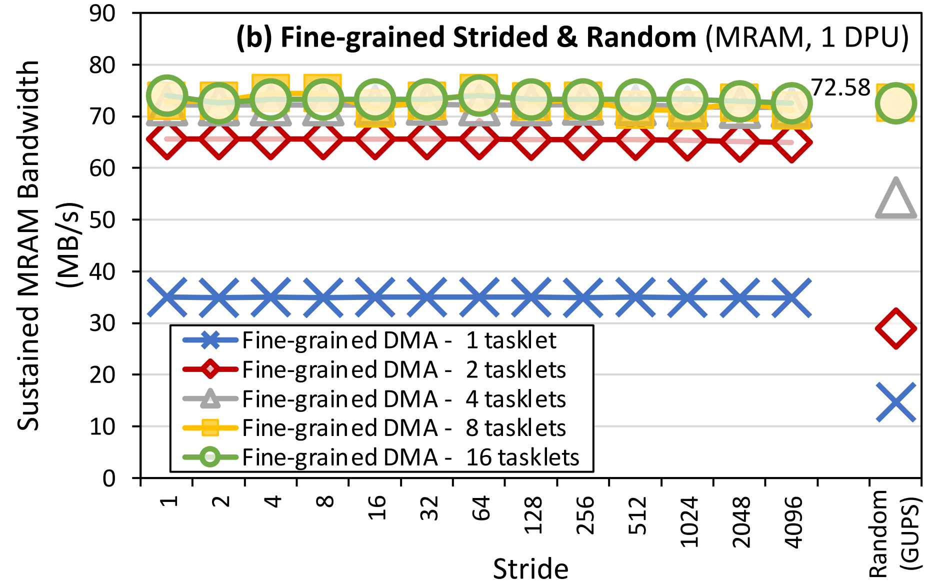
Large difference in maximum sustained bandwidth between coarse-grained and fine-grained DMA

Coarse-grained DMA uses 1,024-byte transfers, while fine-grained DMA uses 8-byte transfers

Random access achieves very similar maximum sustained bandwidth to fine-grained strided approach

Strided and Random Accesses (II)

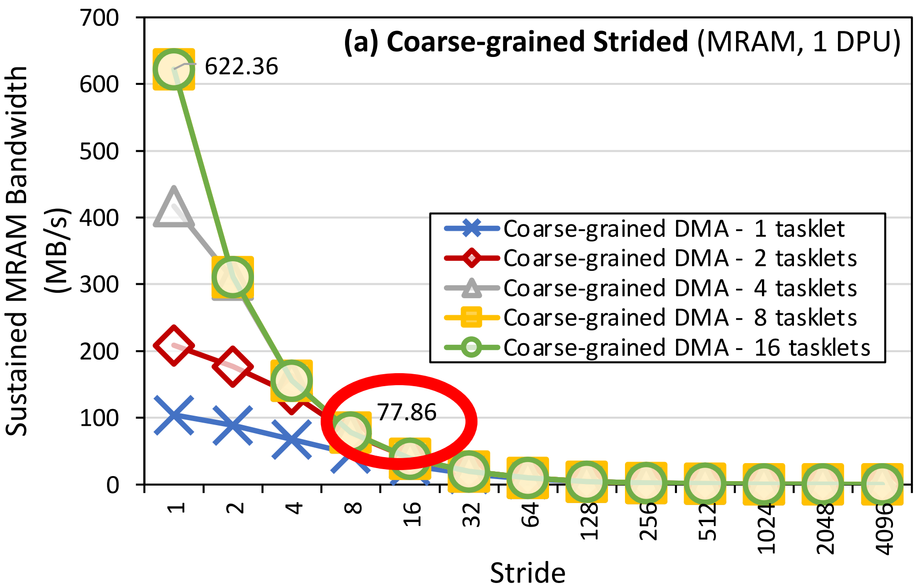


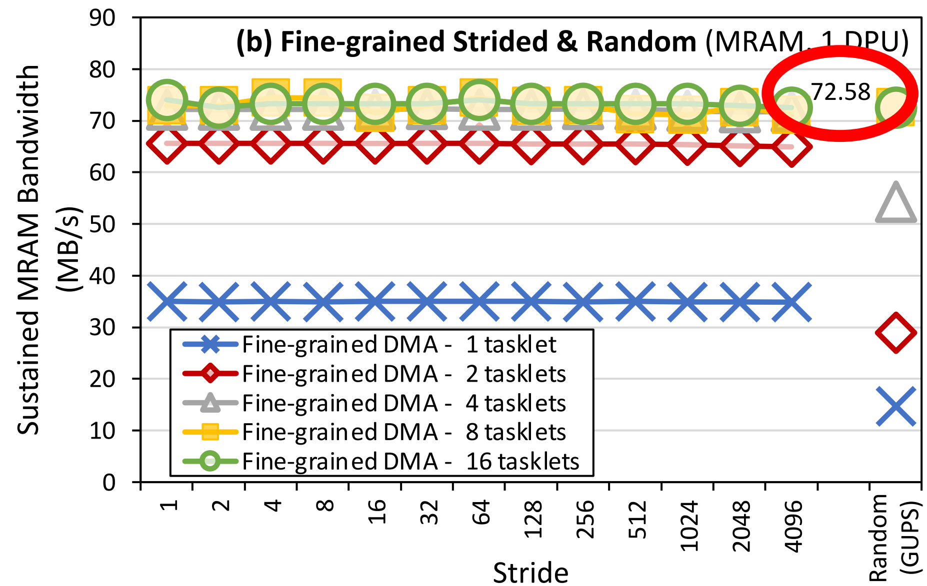


The sustained MRAM bandwidth of coarse-grained DMA decreases as the stride increases

The effective utilization of the transferred data decreases as the stride becomes larger (e.g., a stride 4 means that only one fourth of the transferred data is used)

Strided and Random Accesses (III)

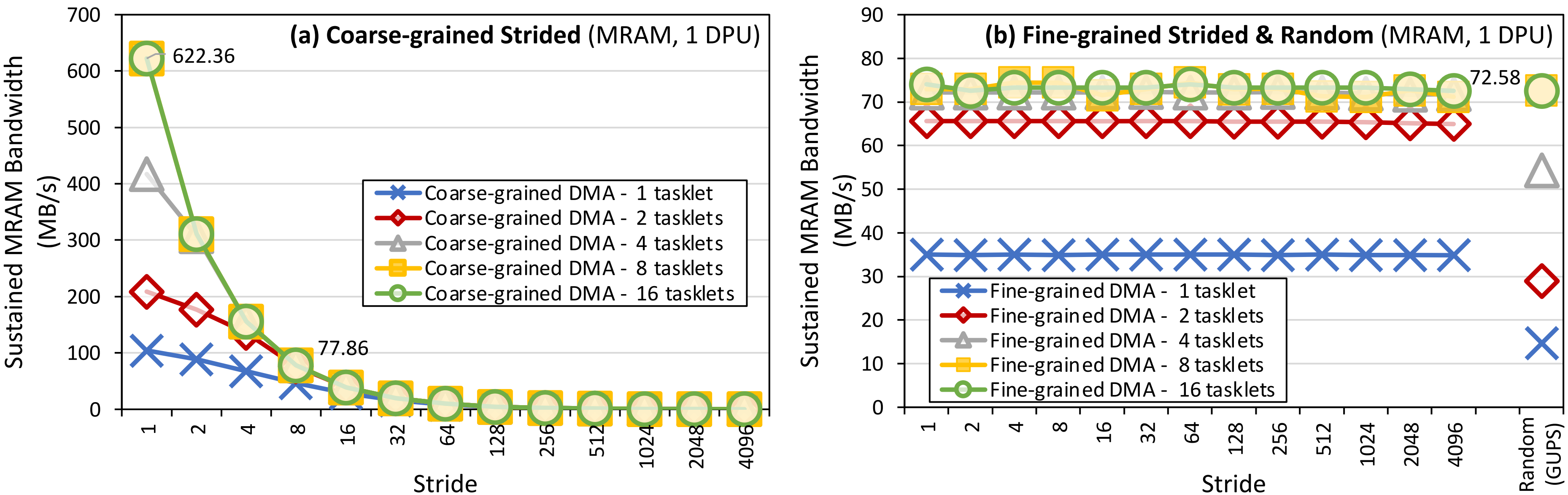




For a stride of 16 or larger, the fine-grained DMA approach achieves higher bandwidth

With stride 16, only one sixteenth of the maximum sustained bandwidth (622.36 MB/s) of coarse-grained DMA is effectively used, which is lower than the bandwidth of fine-grained DMA (72.58 MB/s)

Strided and Random Accesses (IV)

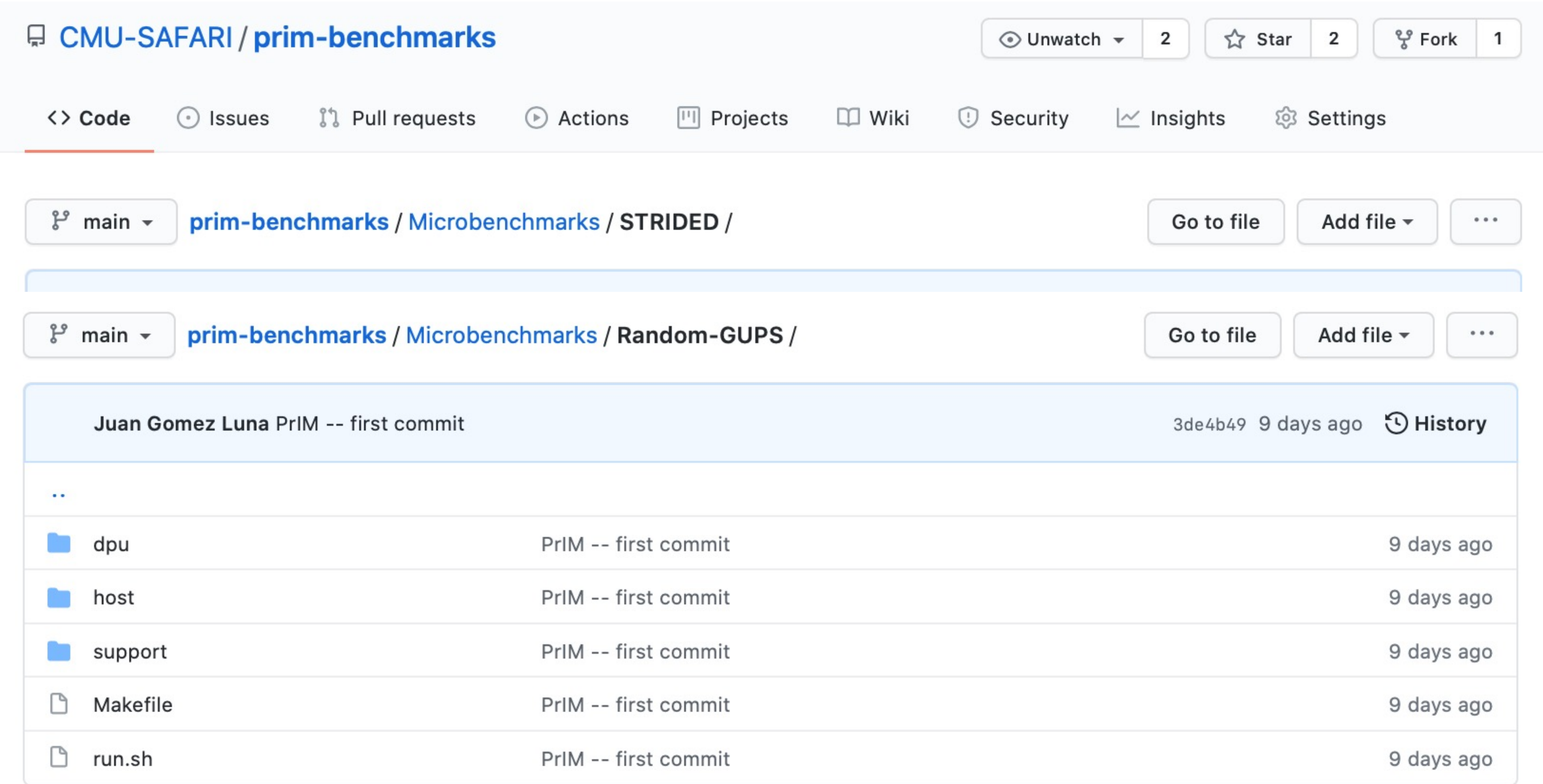


PROGRAMMING RECOMMENDATION 4

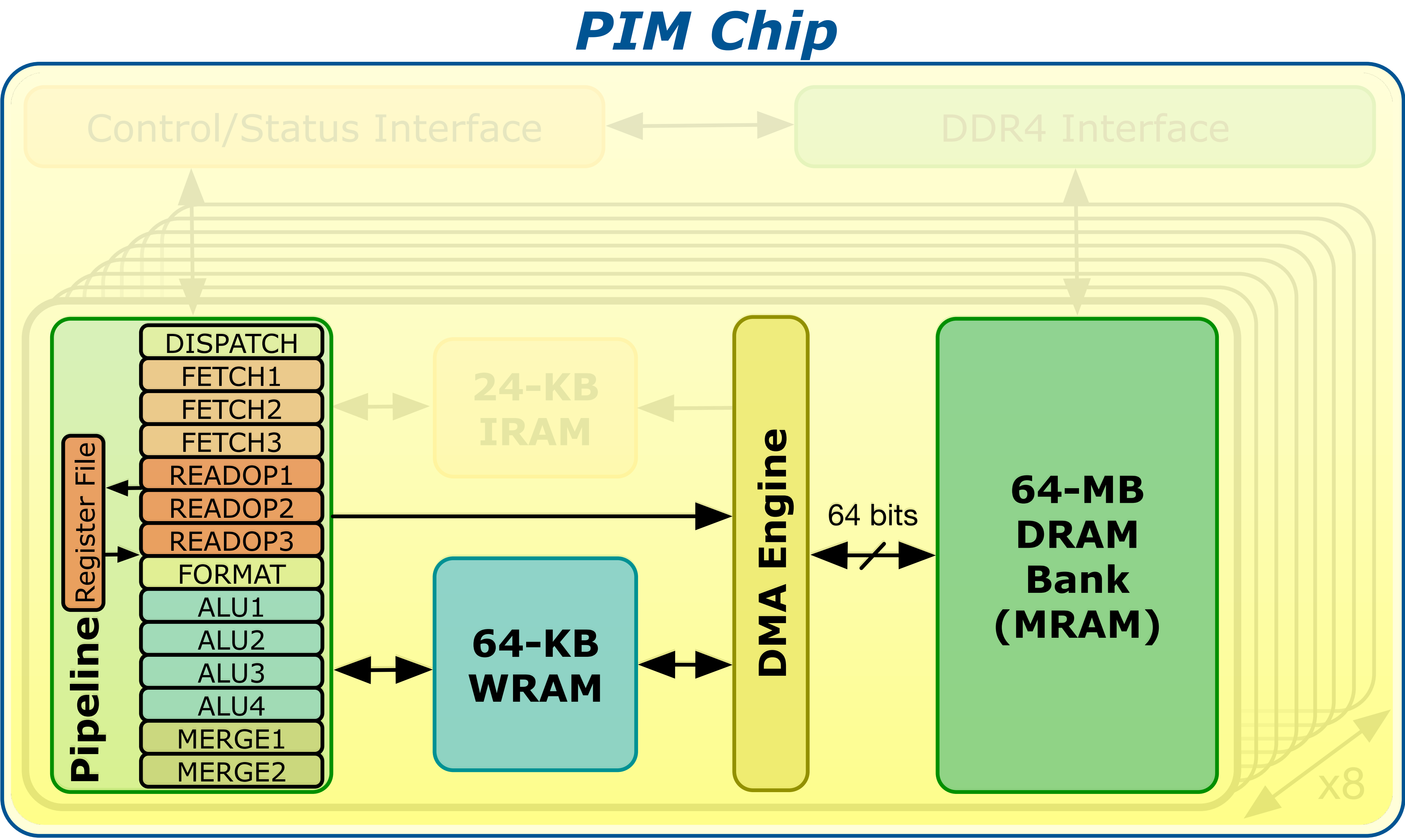
- For strided access patterns with a **stride smaller than 16 8-byte elements, fetch a large contiguous chunk** (e.g., 1,024 bytes) from a DPU's MRAM bank.
- For strided access patterns with **larger strides and random access patterns**, fetch **only the data elements that are needed** from an MRAM bank.

Microbenchmark: Strided and Random

- Strided and random accesses to MRAM



DPU: Arithmetic Throughput vs. Operational Intensity



Arithmetic Throughput vs. Operational Intensity (I)

- Goal
  - Characterize memory-bound regions and compute-bound regions for different datatypes and operations

- Microbenchmark
  - We load one chunk of an MRAM array into WRAM
  - Perform a variable number of operations on the data
  - Write back to MRAM

- The experiment is inspired by the Roofline model*

- We define operational intensity (OI) as the number of arithmetic operations performed per byte accessed from MRAM (OP/B)

- The pipeline latency changes with the operational intensity, but the MRAM access latency is fixed

*S. Williams et al., "Roofline: An Insightful Visual Performance Model for Multi-core Architectures," CACM, 2009

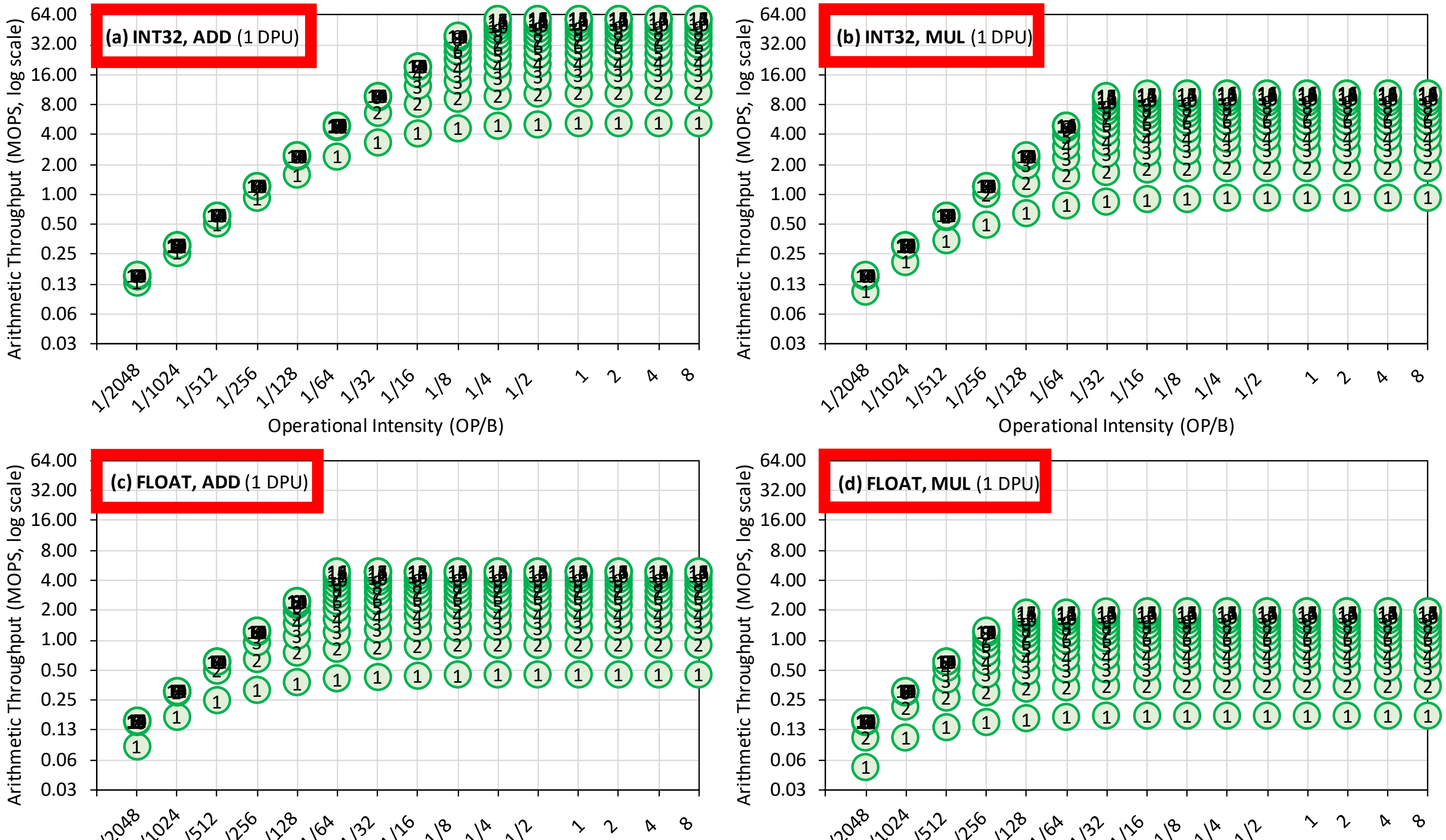
Arithmetic Throughput vs. Operational Intensity (II)

```
int repetitions = input_repeat >= 1.0 ? (int)input_repeat : 1;
int stride      = input_repeat >= 1.0 ? 1 : (int)(1 / input_repeat);

// Load current MRAM block to WRAM
mram_read((__mram_ptr void const*)mram_address_A, bufferA, SIZE * sizeof(T));

// Update
for(int r = 0; r < repetitions; r++){
    for(int i = 0; i < SIZE; i+=stride){
#ifdef ADD
        bufferA[i] += scalar; // ADD
#elif SUB
        bufferA[i] -= scalar; // SUB
#elif MUL
        bufferA[i] *= scalar; // MUL
#elif DIV
        bufferA[i] /= scalar; // DIV
#endif
    }
}

// Write WRAM block to MRAM
mram_write(bufferA, (__mram_ptr void*)mram_address_B, SIZE * sizeof(T));
```

`input_repeat` greater or equal to 1 indicates the (integer) number of repetitions per input element

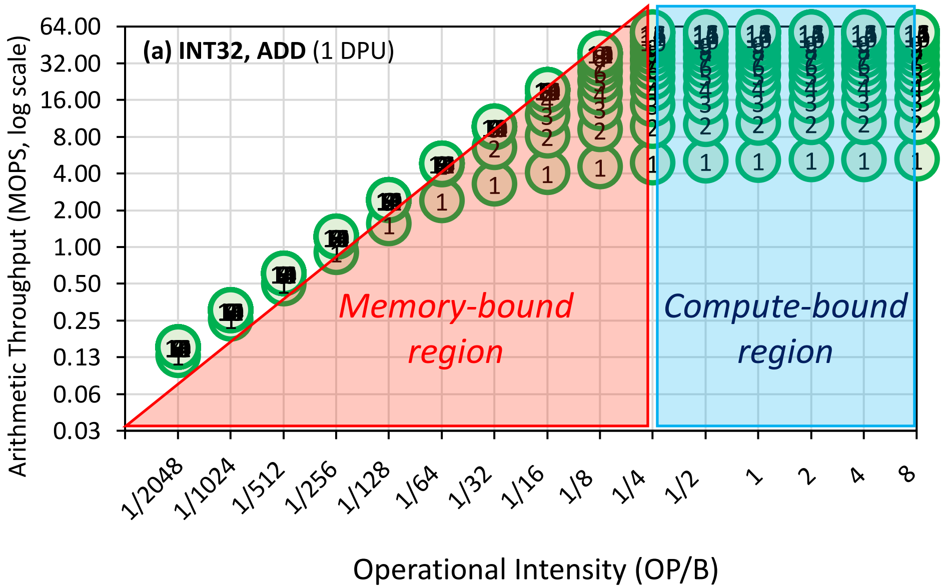
`input_repeat` smaller than 1 indicates the fraction of elements that are updated

Arithmetic Throughput vs. Operational Intensity (III)



We show results of arithmetic throughput vs. operational intensity for (a) 32-bit integer ADD, (b) 32-bit integer MUL, (c) 32-bit floating-point ADD, and (d) 32-bit floating-point MUL (results for other datatypes and operations show similar trends)

Arithmetic Throughput vs. Operational Intensity (IV)



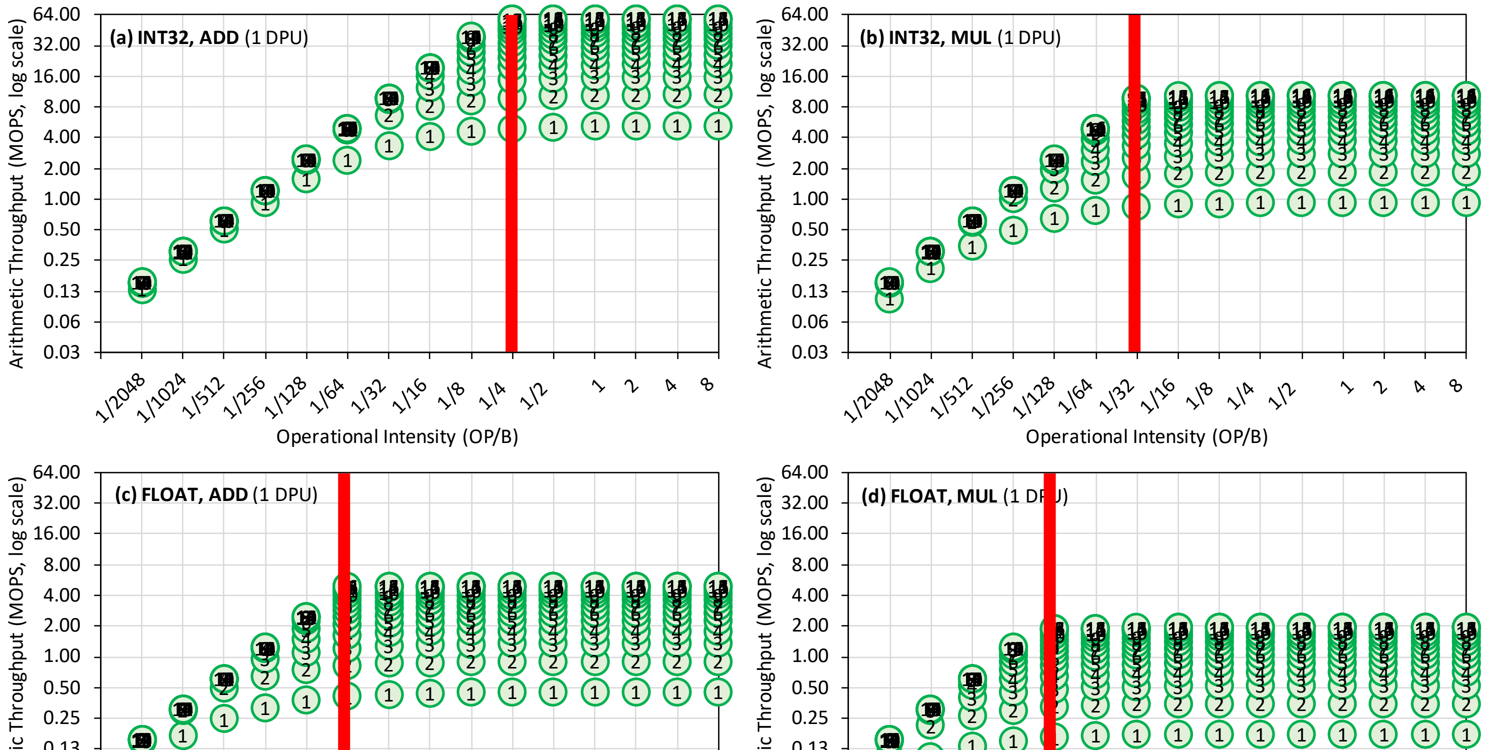
In the memory-bound region, the arithmetic throughput increases with the operational intensity

In the compute-bound region, the arithmetic throughput is flat at its maximum

The *throughput saturation point* is the operational intensity where the transition between the memory-bound region and the compute-bound region happens

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

Arithmetic Throughput vs. Operational Intensity (V)

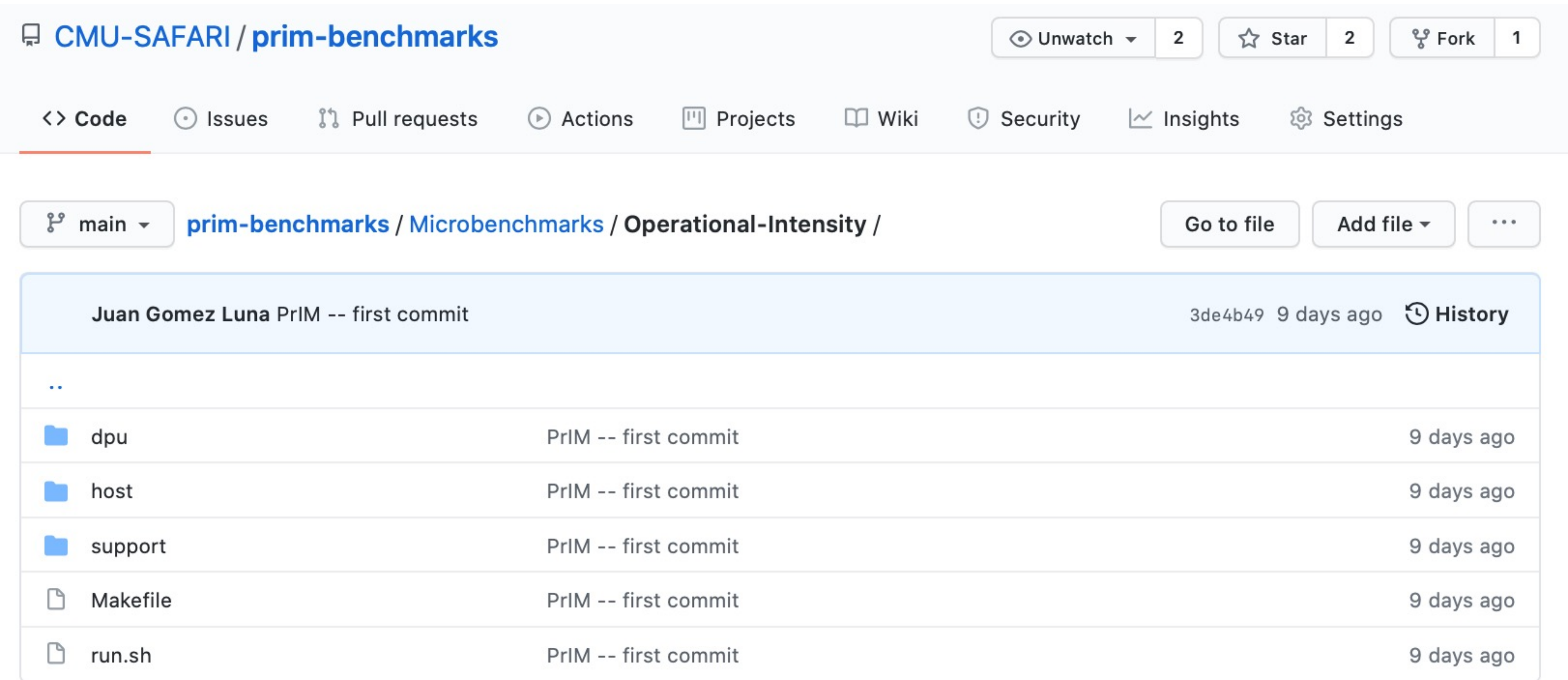


KEY OBSERVATION 6

The arithmetic throughput of a DRAM Processing Unit (DPU) saturates at low or very low operational intensity (e.g., 1 integer addition per 32-bit element). Thus, **the DPU is fundamentally a compute-bound processor.**
We expect **most real-world workloads be compute-bound in the UPMEM PIM architecture.**

Microbenchmark: Arithmetic Throughput vs. Operational Intensity

- Arithmetic Throughput versus Operational Intensity



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

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*

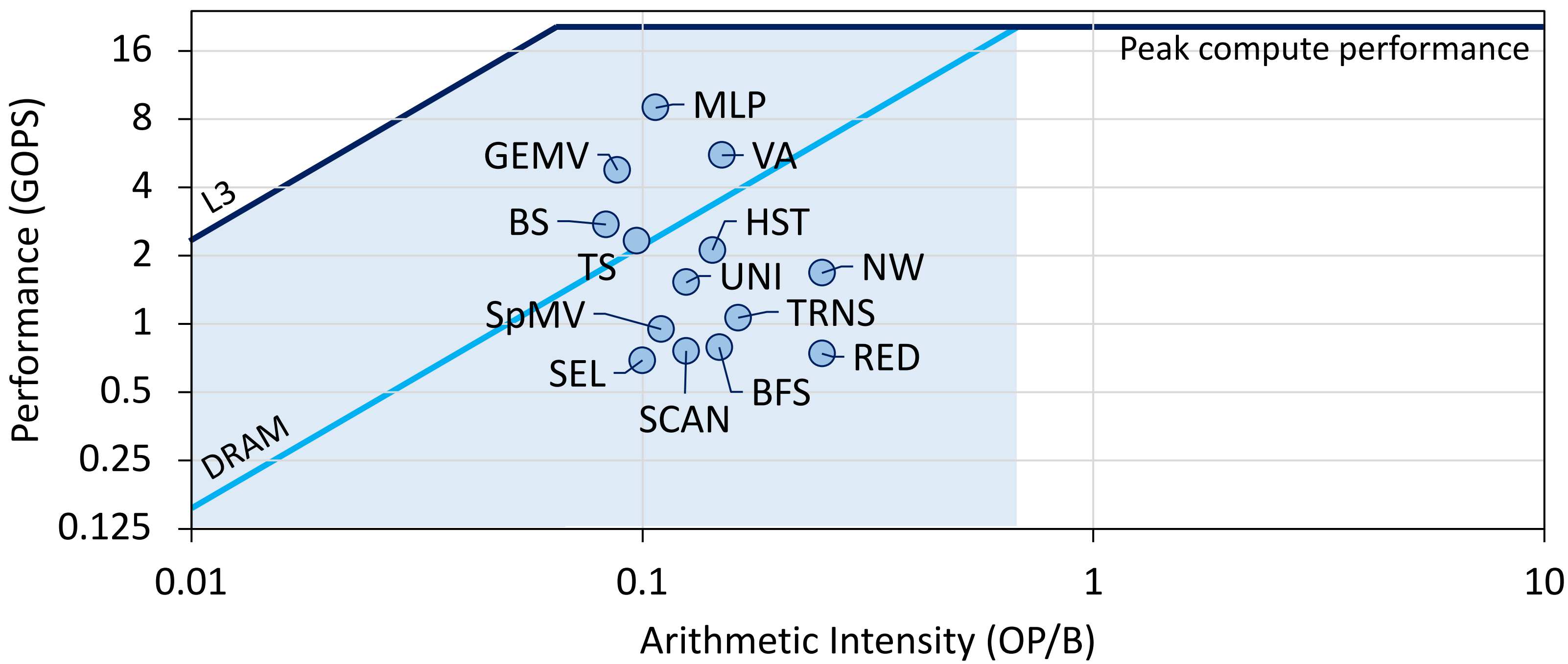
*There are two versions for two of the workloads (HST, SCAN).

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	

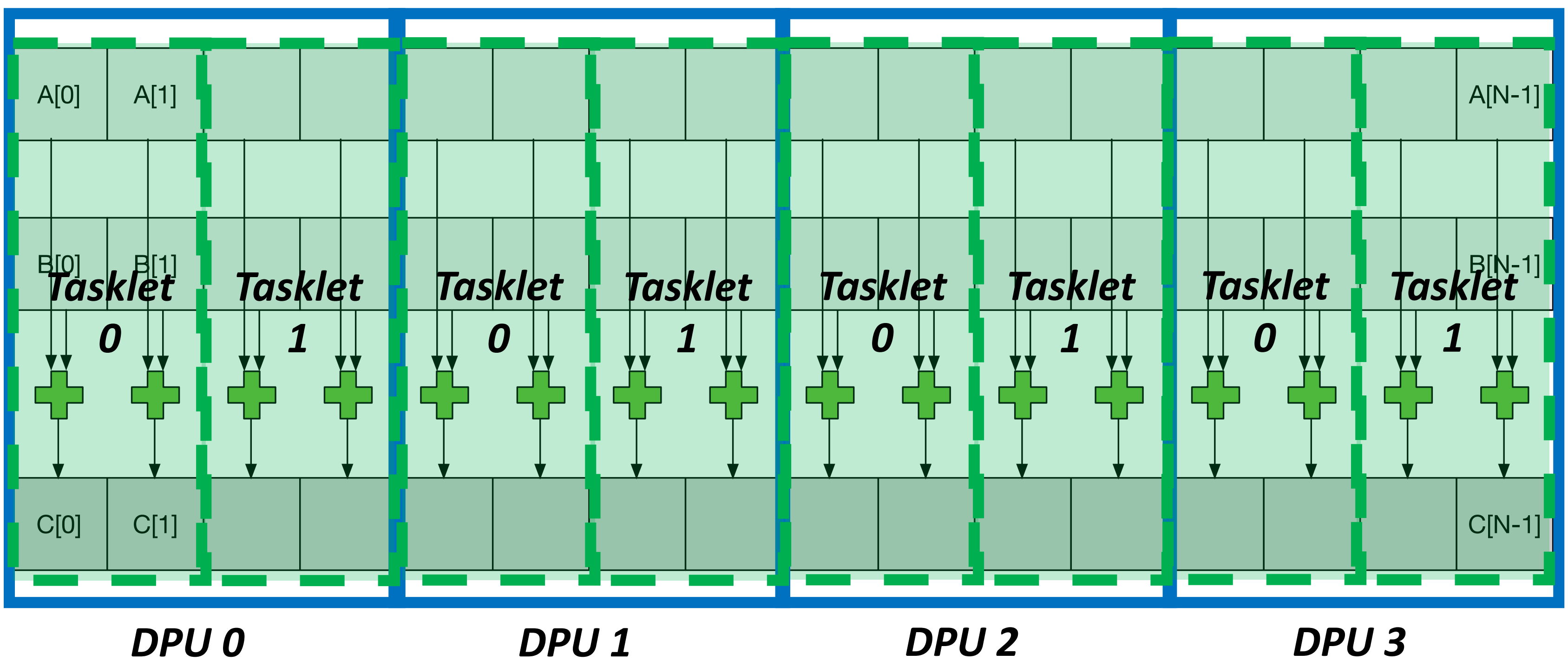
PrIM Benchmarks: Inter-DPU Communication

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

Recall: Vector Addition (VA)

- Our first programming example
- We partition the input arrays across:
  - DPUs
  - Tasklets, i.e., software threads running on a DPU



Programming a DPU Kernel (I)

- Vector addition

```c
// Vector addition kernel
int main_kernel1() {
    unsigned int tasklet_id = me()
    uint32_t input_size_dpu_bytes = DPU_INPUT_ARGUMENTS.size; // Input size per DPU in bytes
    uint32_t input_size_dpu_bytes_transfer = DPU_INPUT_ARGUMENTS.transfer_size; // Transfer input size per DPU in bytes

    // Address of the current processing block in MRAM
    uint32_t base_tasklet = tasklet_id << BLOCK_SIZE_LOG2;
    uint32_t mram_base_addr_A = (uint32_t)DPU_MRAM_HEAP_POINTER;
    uint32_t mram_base_addr_B = (uint32_t)(DPU_MRAM_HEAP_POINTER + input_size_dpu_bytes_transfer);

    // Initialize a local cache to store the MRAM block
    T *cache_A = (T *) mem_alloc(BLOCK_SIZE);
    T *cache_B = (T *) mem_alloc(BLOCK_SIZE);

    for(unsigned int byte_index = base_tasklet; byte_index < input_size_dpu_bytes; byte_index += BLOCK_SIZE * NR_TASKLETS){
        // Bound checking
        uint32_t l_size_bytes = (byte_index + BLOCK_SIZE >= input_size_dpu_bytes) ? (input_size_dpu_bytes - byte_index) : BLOCK_SIZE;

        // Load cache with current MRAM block
        mram_read((__mram_ptr void const*)(mram_base_addr_A + byte_index), cache_A, l_size_bytes);
        mram_read((__mram_ptr void const*)(mram_base_addr_B + byte_index), cache_B, l_size_bytes);

        // Computer vector addition
        vector_addition(cache_B, cache_A, l_size_bytes >> DIV);

        // Write cache to current MRAM block
        mram_write(cache_B, (__mram_ptr void*)(mram_base_addr_B + byte_index), l_size_bytes);

    }
    return 0;
}
```

Tasklet ID

Size of vector tile processed by a DPU

MRAM addresses of arrays A and B

WRAM allocation

MRAM-WRAM DMA transfers

Vector addition (see next slide)

WRAM-MRAM DMA transfer

Programming a DPU Kernel (II)

- Vector addition

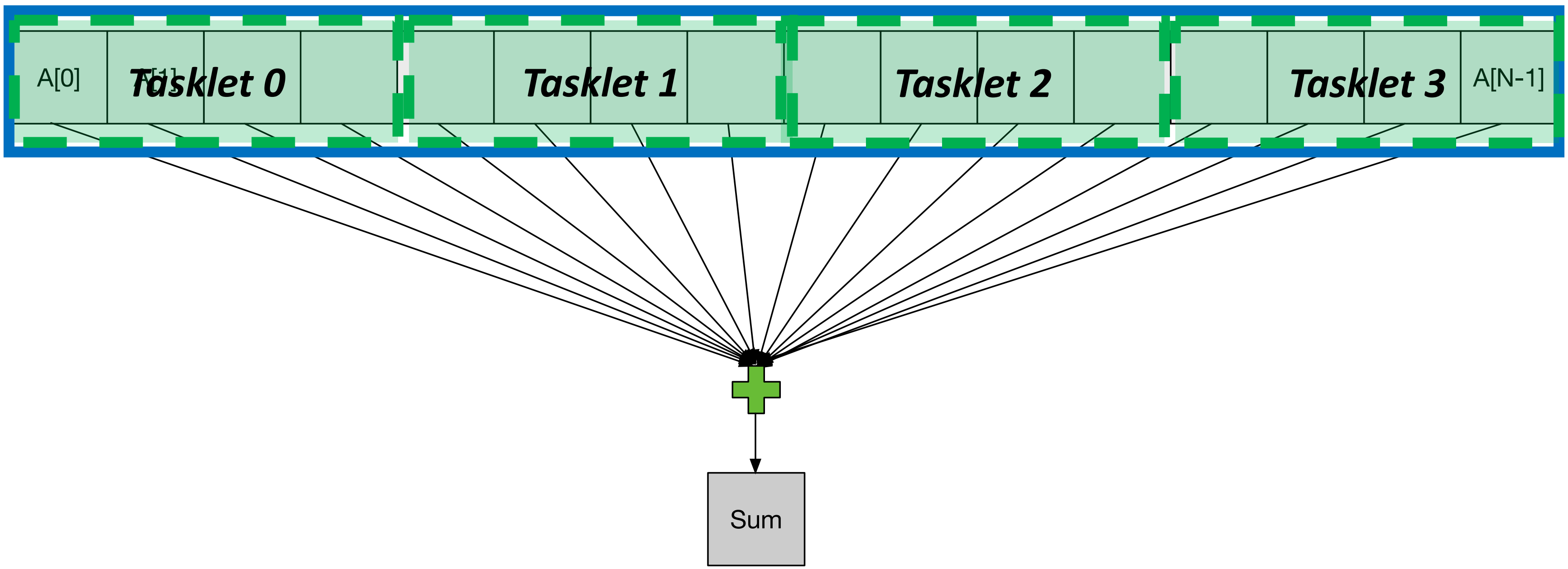
```
// vector_addition: Computes the vector addition of a cached block
static void vector_addition(T *bufferB, T *bufferA, unsigned int l_size) {

    for (unsigned int i = 0; i < l_size; i++){
        bufferB[i] += bufferA[i];
    }

}
```

Programming a DPU Kernel (III)

- A tasklet is the software abstraction of a hardware thread
- Each tasklet can have its own memory space in WRAM
  - Tasklets can also share data in WRAM by sharing pointers
- Tasklets within the same DPU can synchronize
  - Mutual exclusion
    - `mutex_lock(); mutex_unlock();`
  - Handshakes
    - `handshake_wait_for(); handshake_notify();`
  - Barriers
    - `barrier_wait();`
  - Semaphores
    - `sem_give(); sem_take();`

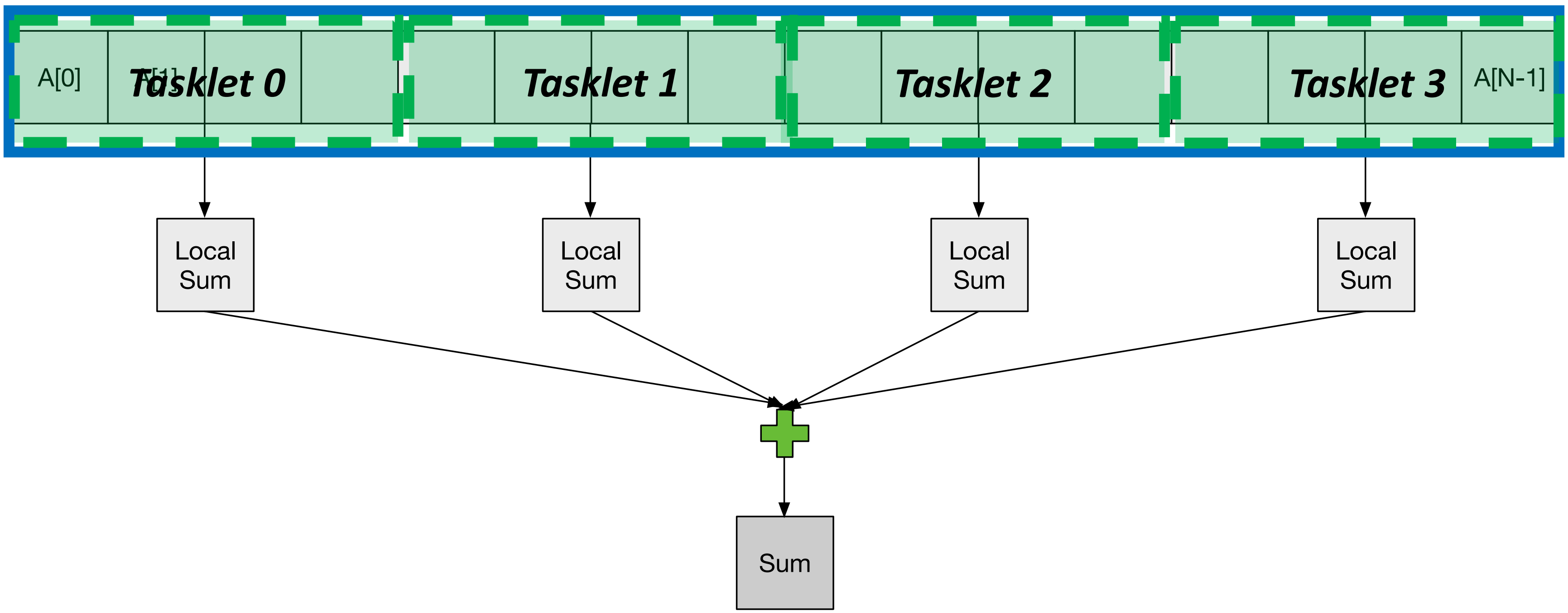
Parallel Reduction (I)

- Tasklets in a DPU can work together on a parallel reduction



Parallel Reduction (II)

- Each tasklet computes a local sum



Parallel Reduction (III)

- Each tasklet computes a local sum

```
for(unsigned int byte_index = base_tasklet; byte_index < input_size_dpu_bytes; byte_index += BLOCK_SIZE * NR_TASKLETS){

    // Bound checking
    uint32_t l_size_bytes = (byte_index + BLOCK_SIZE >= input_size_dpu_bytes) ? (input_size_dpu_bytes - byte_index) : BLOCK_SIZE;

    // Load cache with current MRAM block
    mram_read((__mram_ptr void const*)(mram_base_addr_A + byte_index), cache_A, l_size_bytes);

    // Reduction in each tasklet
    l_count += reduction(cache_A, l_size_bytes >> DIV);

}
// Copy local count to shared array in WRAM
message[tasklet_id] = l_count;
```

Accumulate in a local sum

Copy local sum into WRAM

Final Reduction

- A single tasklet can perform the final reduction

```c
for(unsigned int byte_index = base_tasklet; byte_index < input_size_dpu_bytes; byte_index += BLOCK_SIZE * NR_TASKLETS){

    // Bound checking
    uint32_t l_size_bytes = (byte_index + BLOCK_SIZE >= input_size_dpu_bytes) ? (input_size_dpu_bytes - byte_index) : BLOCK_SIZE;

    // Load cache with current MRAM block
    mram_read((__mram_ptr void const*)(mram_base_addr_A + byte_index), cache_A, l_size_bytes);

    // Reduction in each tasklet
    l_count += reduction(cache_A, l_size_bytes >> DIV);

}
// Copy local count to shared array in WRAM
message[tasklet_id] = l_count;
```

Accumulate in a local sum

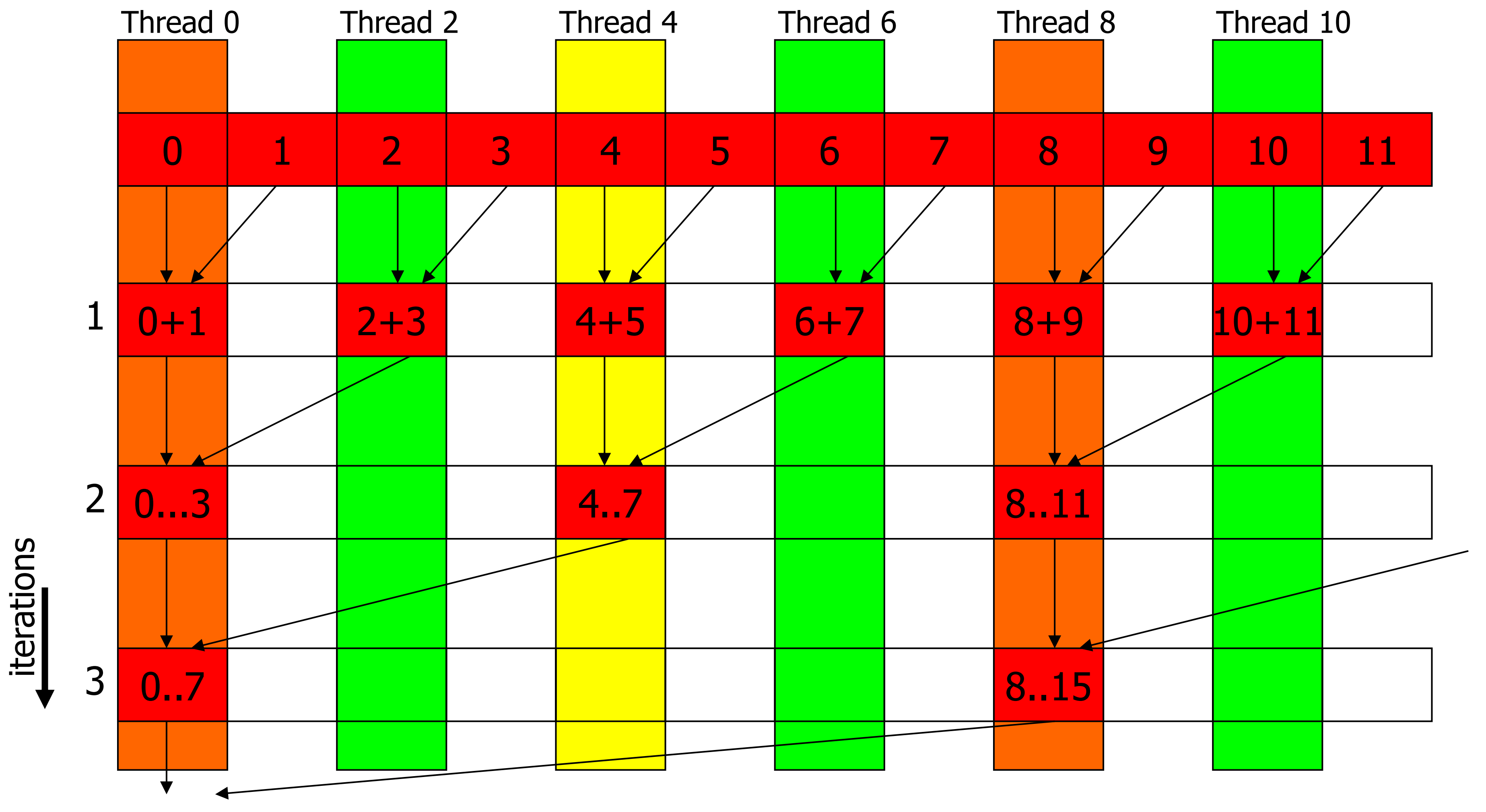
Copy local sum into WRAM

```c
// Single-thread reduction
// Barrier
barrier_wait(&my_barrier);

if(tasklet_id == 0){
    #pragma unroll
    for (unsigned int each_tasklet = 1; each_tasklet < NR_TASKLETS; each_tasklet++){
        message[0] += message[each_tasklet];
    }

    // Total count in this DPU
    result->t_count = message[0];
}
```

Barrier synchronization

Sequential accumulation

Vector Reduction: Naïve Mapping



Slide credit: Hwu & Kirk

https://www.youtube.com/watch?v=y40-tY5WJ8A
https://safari.ethz.ch/digitaltechnik/spring2018/lib/exe/fetch.php?media=digitaldesign-2018-lecture22-gpuprogramming-afterlecture.pdf

Using Barriers: Tree-Based Reduction

- Multiple tasklets can perform a tree-based reduction
  - After every iteration tasklets synchronize with a barrier
  - Half of the tasklets retire at the end of an iteration

```
// Barrier
barrier_wait(&my_barrier);

#pragma unroll
for (unsigned int offset = 1; offset < NR_TASKLETS; offset <<= 1){

    if((tasklet_id & (2*offset - 1)) == 0){
        message[tasklet_id] += message[tasklet_id + offset];
    }

    // Barrier
    barrier_wait(&my_barrier);
}
```

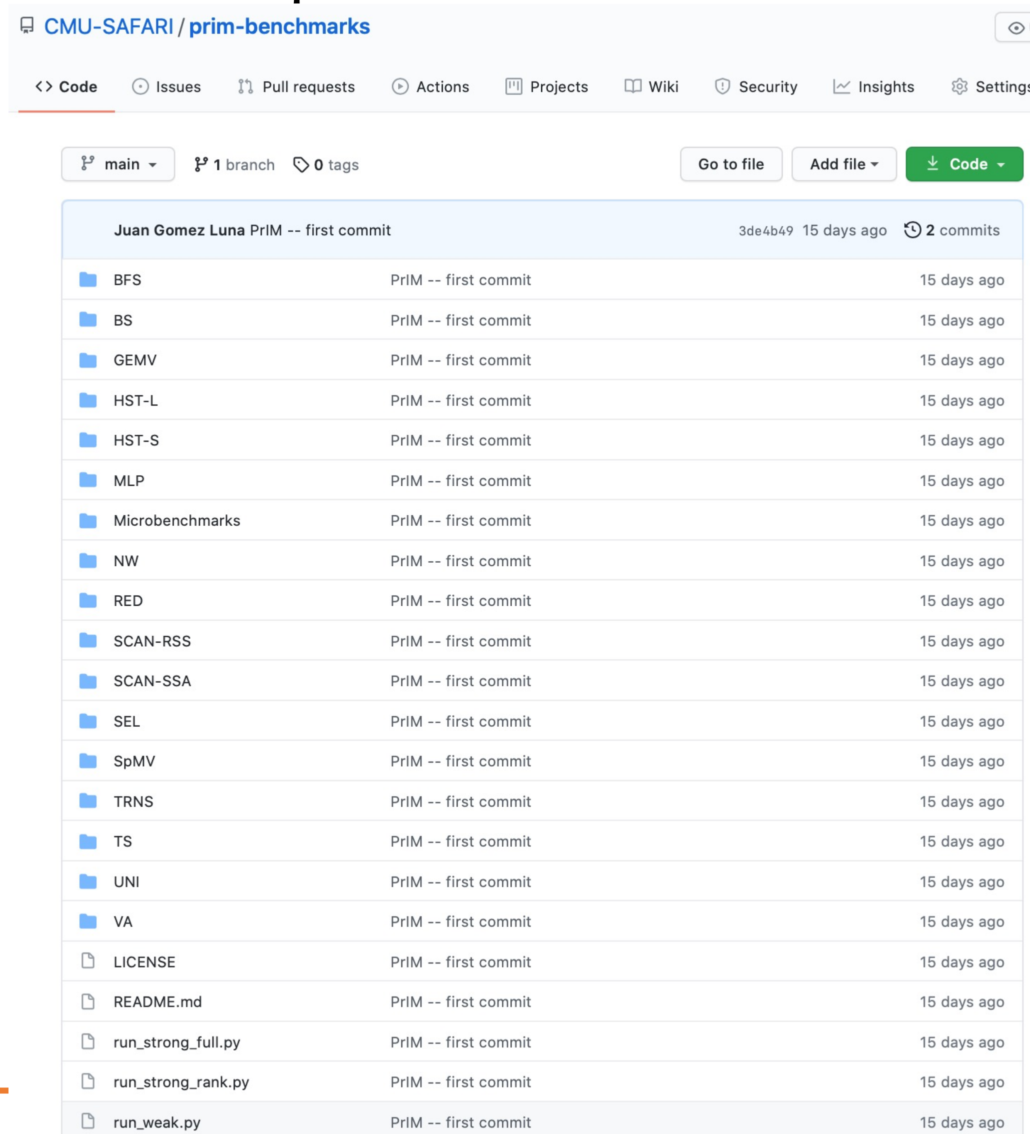
"offset" tasklets working

Barrier synchronization

PrIM also includes a handshake-based tree-based reduction.
We compare single-tasklet, barrier-based, and handshake-based versions in the Appendix of the paper

PrIM Benchmarks

- 16 benchmarks and scripts for evaluation



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

Datasets

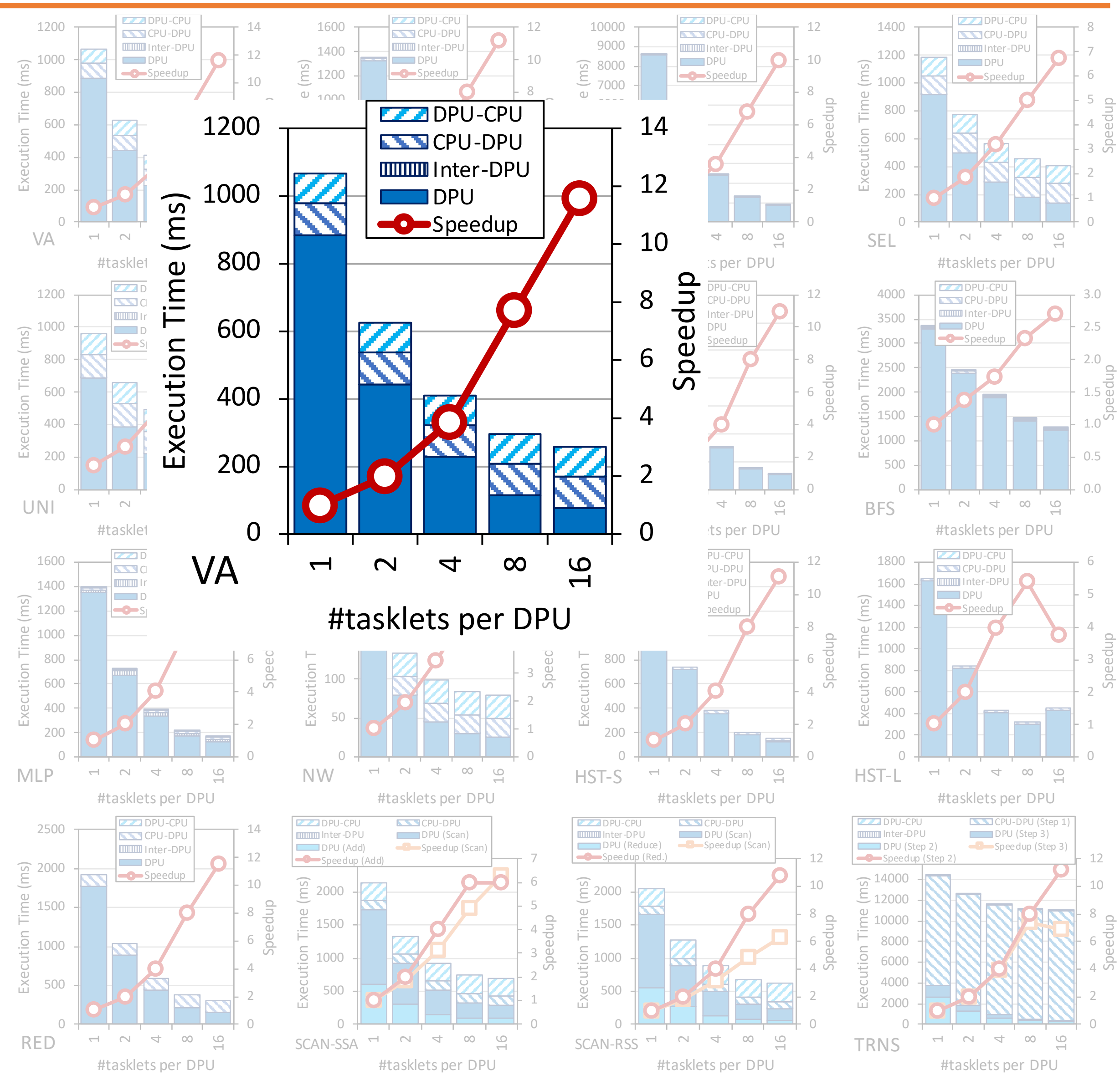
- Strong and weak scaling experiments

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

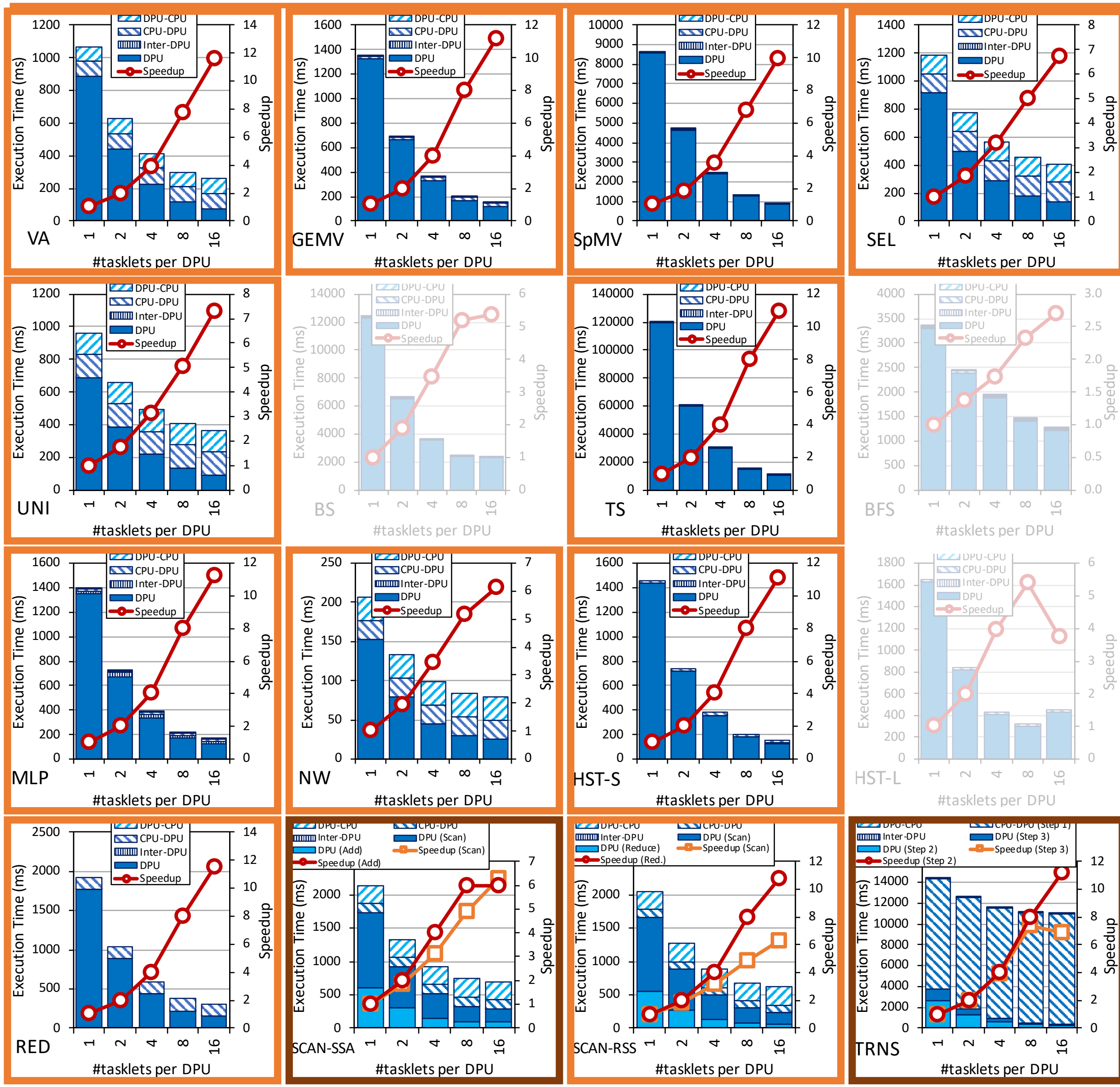
The PrIM benchmarks repository includes
all datasets and scripts used in our evaluation
https://github.com/CMU-SAFARI/prim-benchmarks

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)



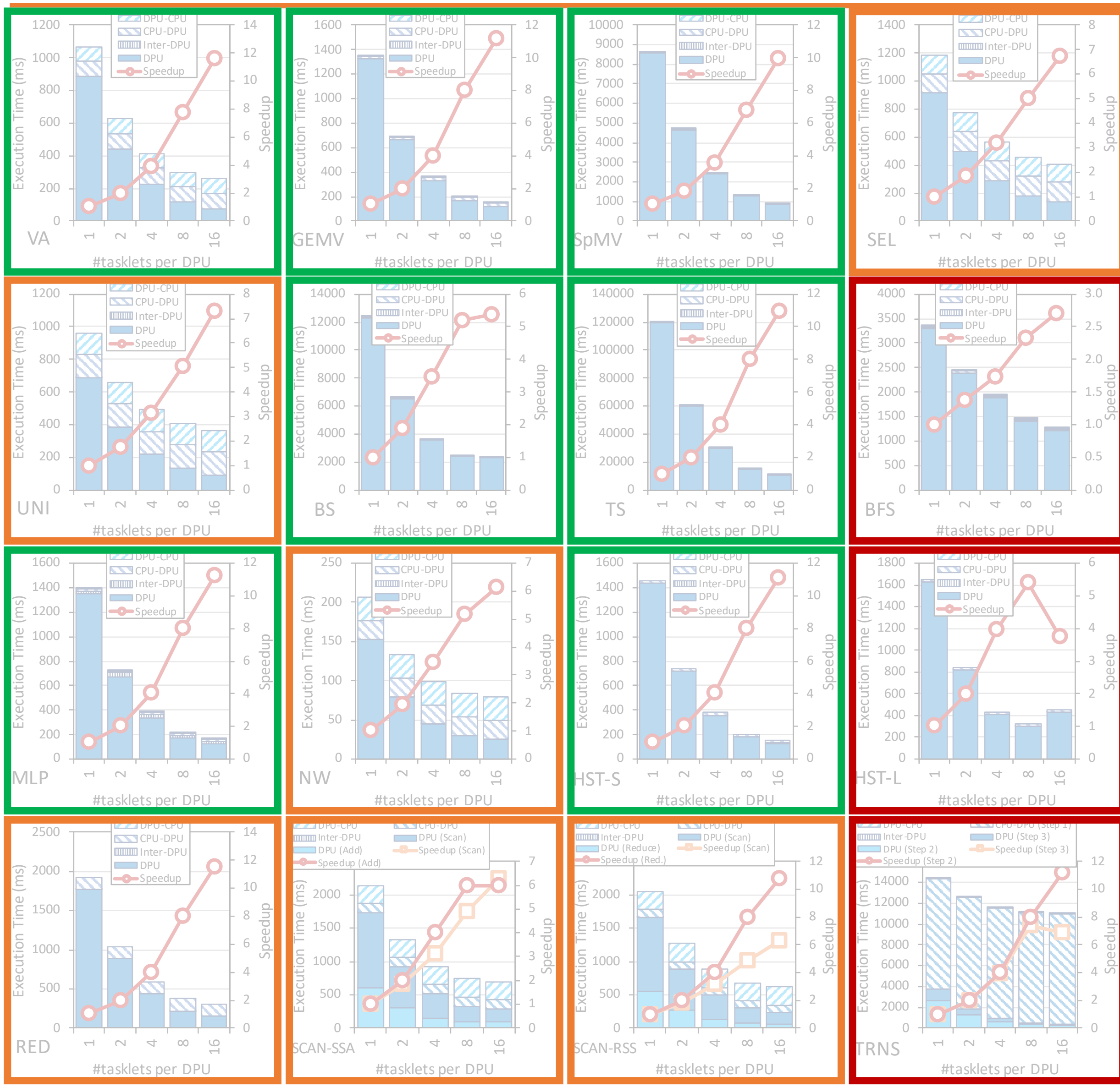
VA, GEMV, SpMV, SEL, UNI, TS, MLP, NW, HST-S, RED, SCAN-SSA (Scan kernel), SCAN-RSS (both kernels), and TRNS (Step 2 kernel), the best performing number of tasklets is 16

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)

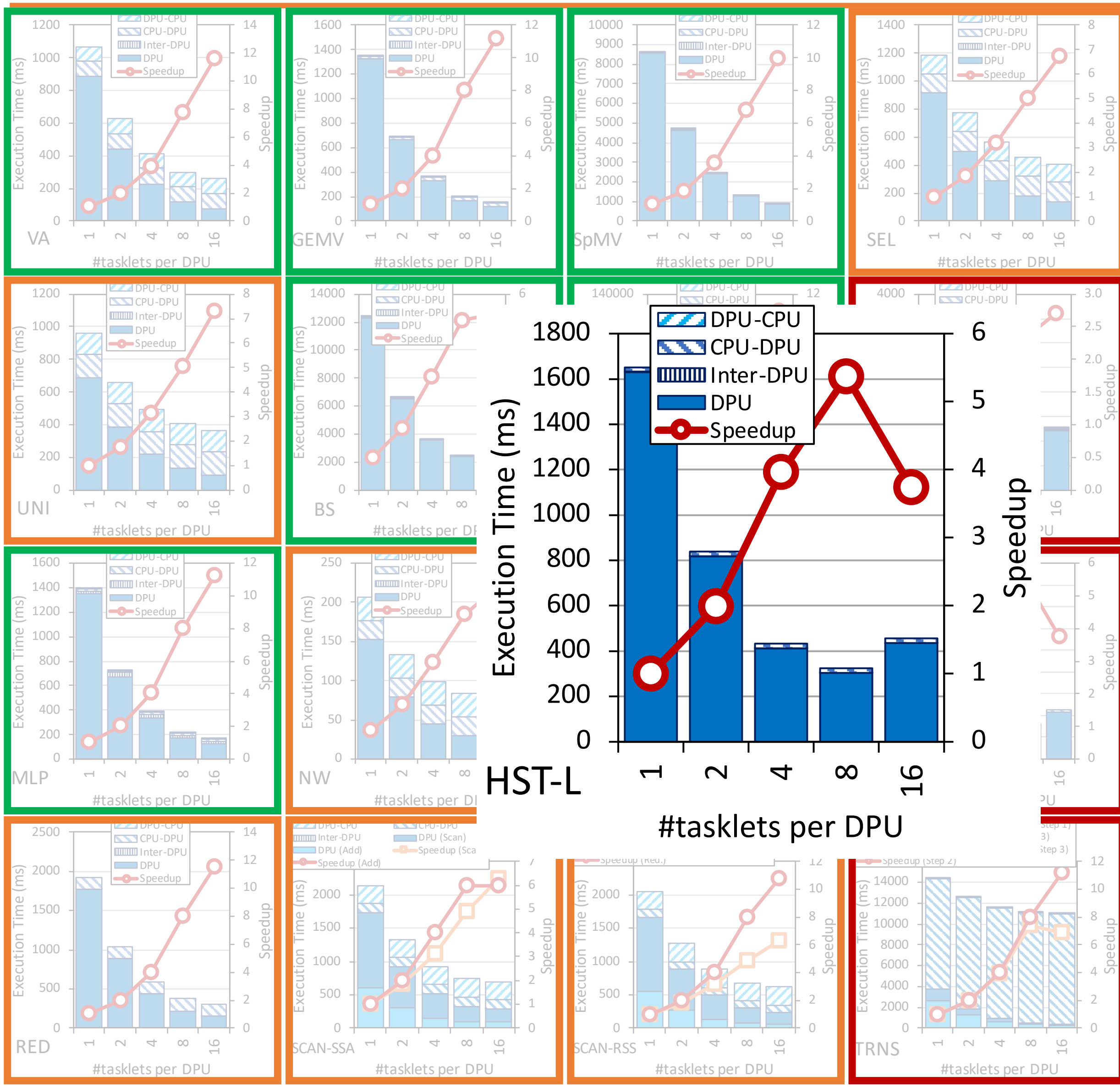


VA, GEMV, SpMV, BS, TS, MLP, HST-S do not use intra-DPU synchronization primitives

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 synchronization primitives

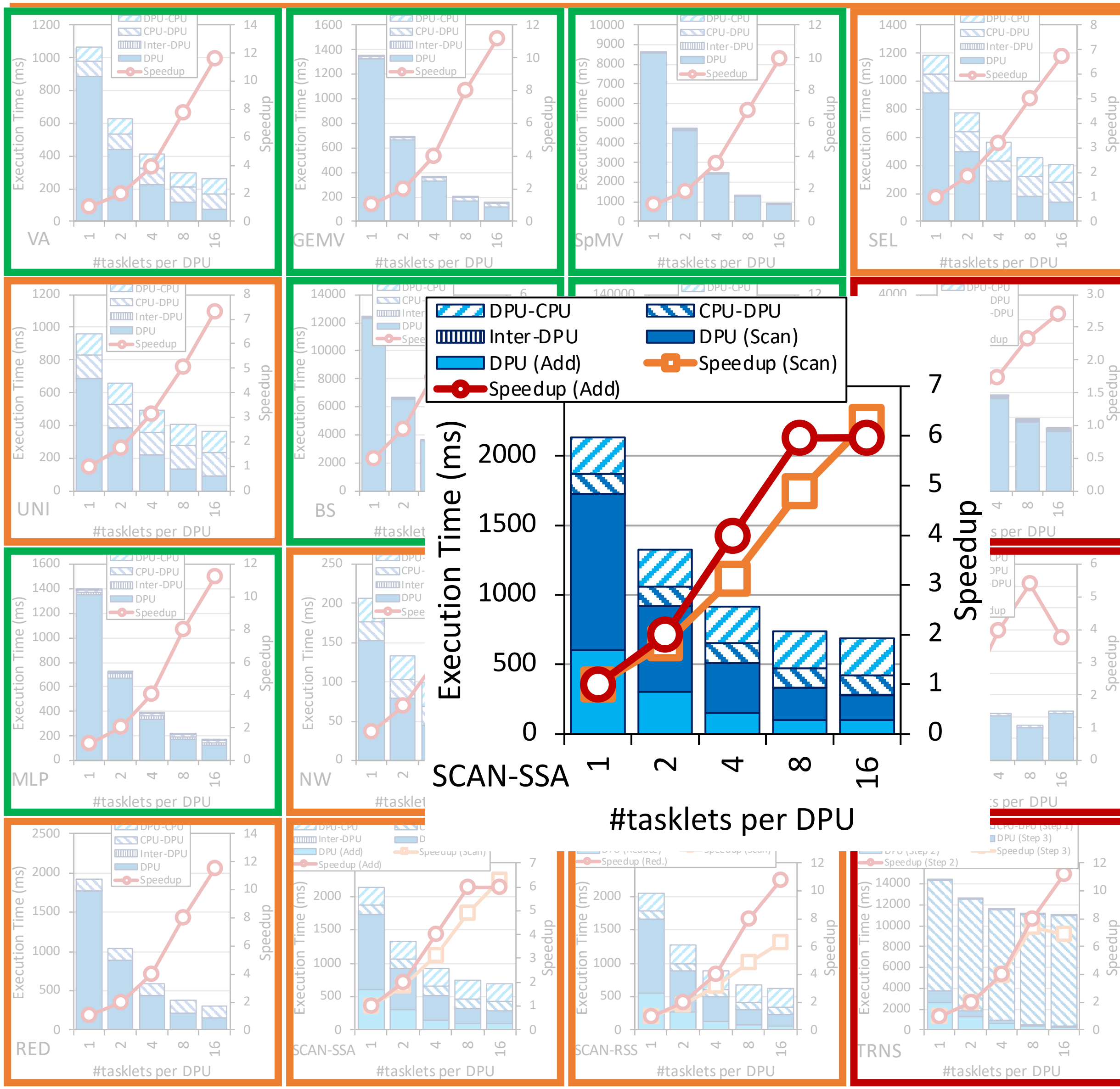
In SEL, UNI, NW, RED, SCAN-SSA (Scan kernel), SCAN-RSS (both kernels), synchronization is lightweight

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)

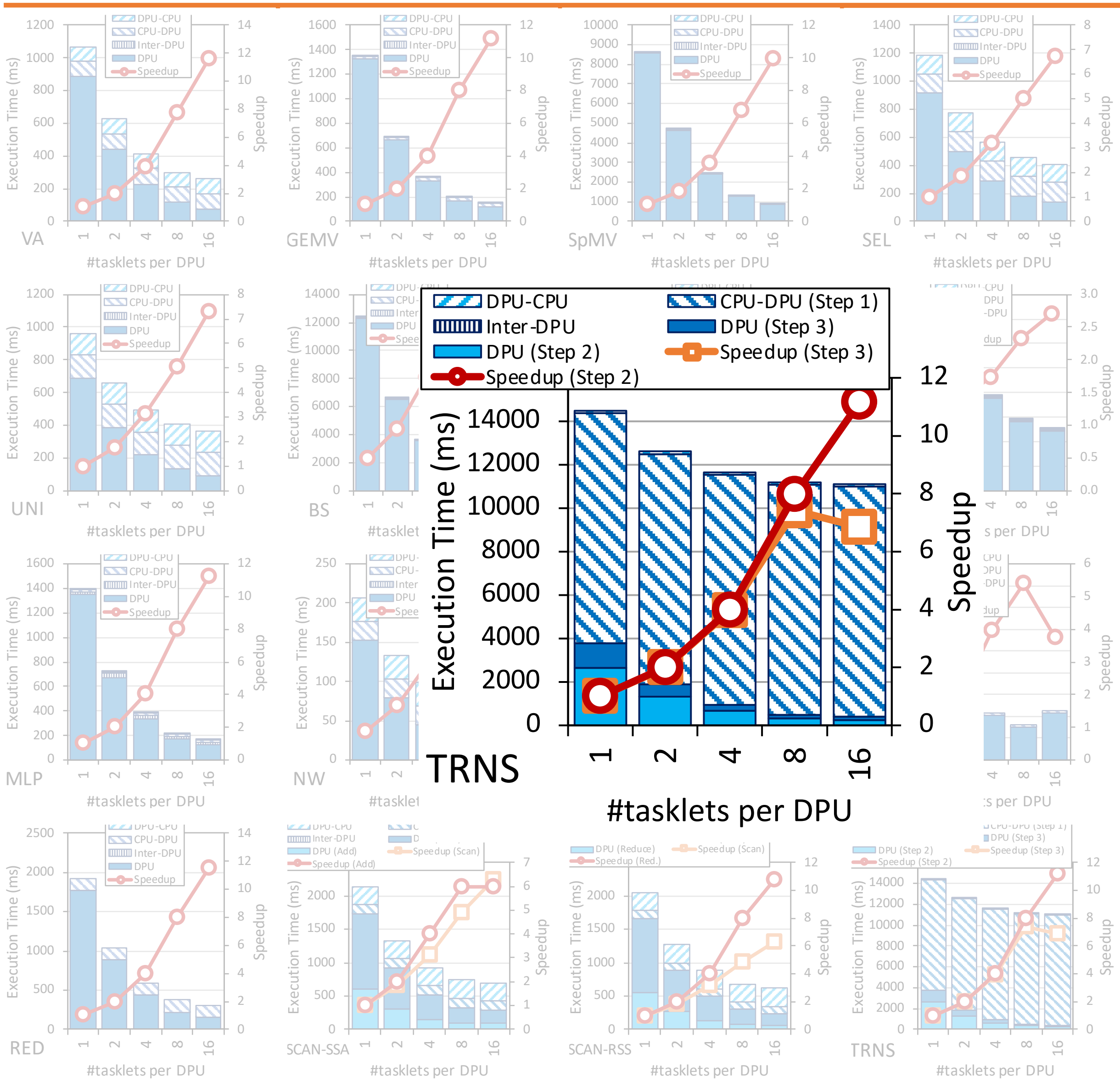


SCAN-SSA (Add kernel) is not compute-intensive. Thus, performance saturates with less that 11 tasklets (recall STREAM ADD).
BS shows similar behavior

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

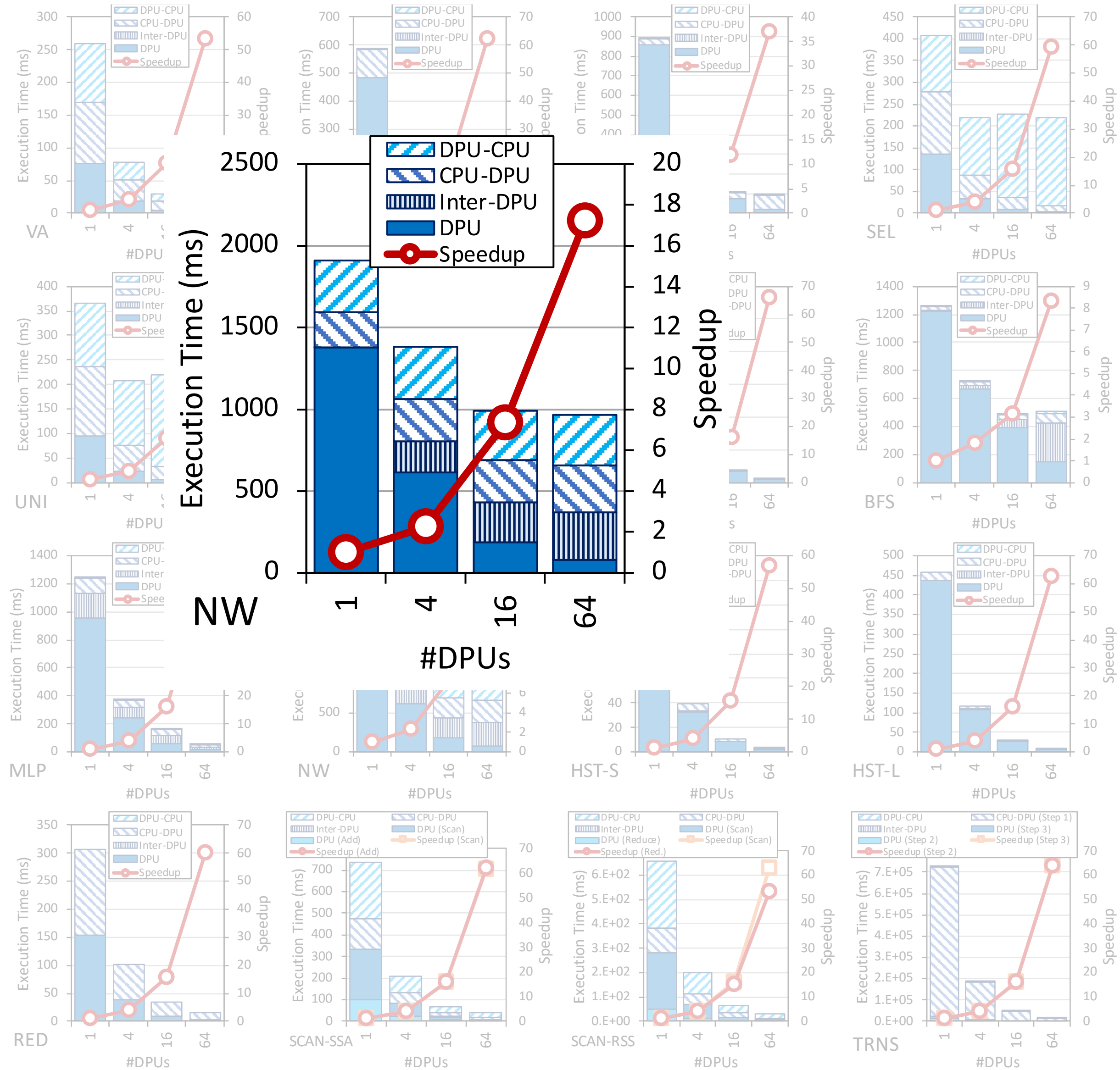
TRNS performs step 1 of the matrix transposition via the CPU-DPU transfer.
Using small transfers (8 elements) does not exploit full CPU-DPU bandwidth

KEY OBSERVATION 13

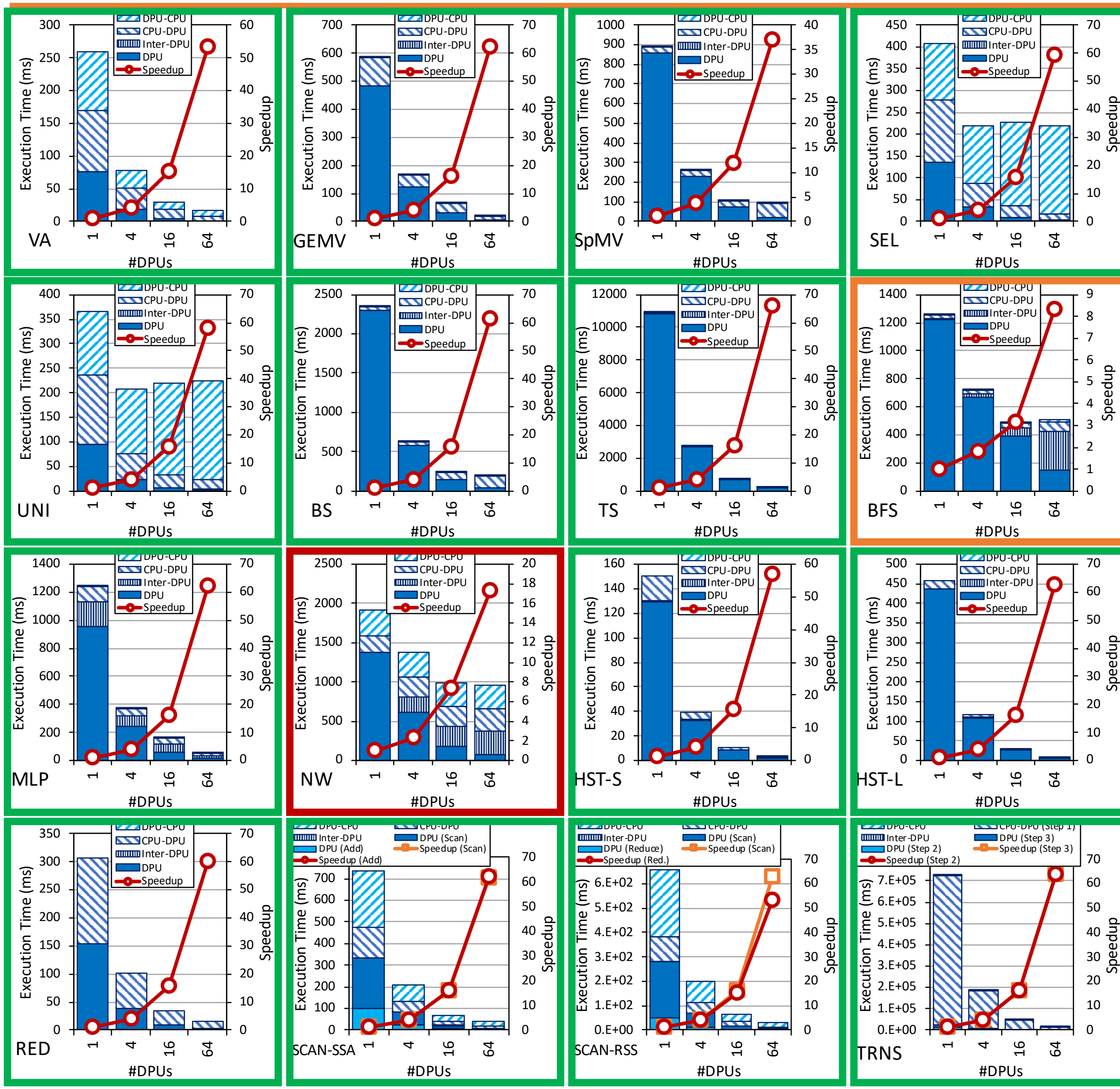
Transferring large data chunks from/to the host CPU is preferred for input data and output results due to higher sustained CPU-DPU/DPU-CPU bandwidths.

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)



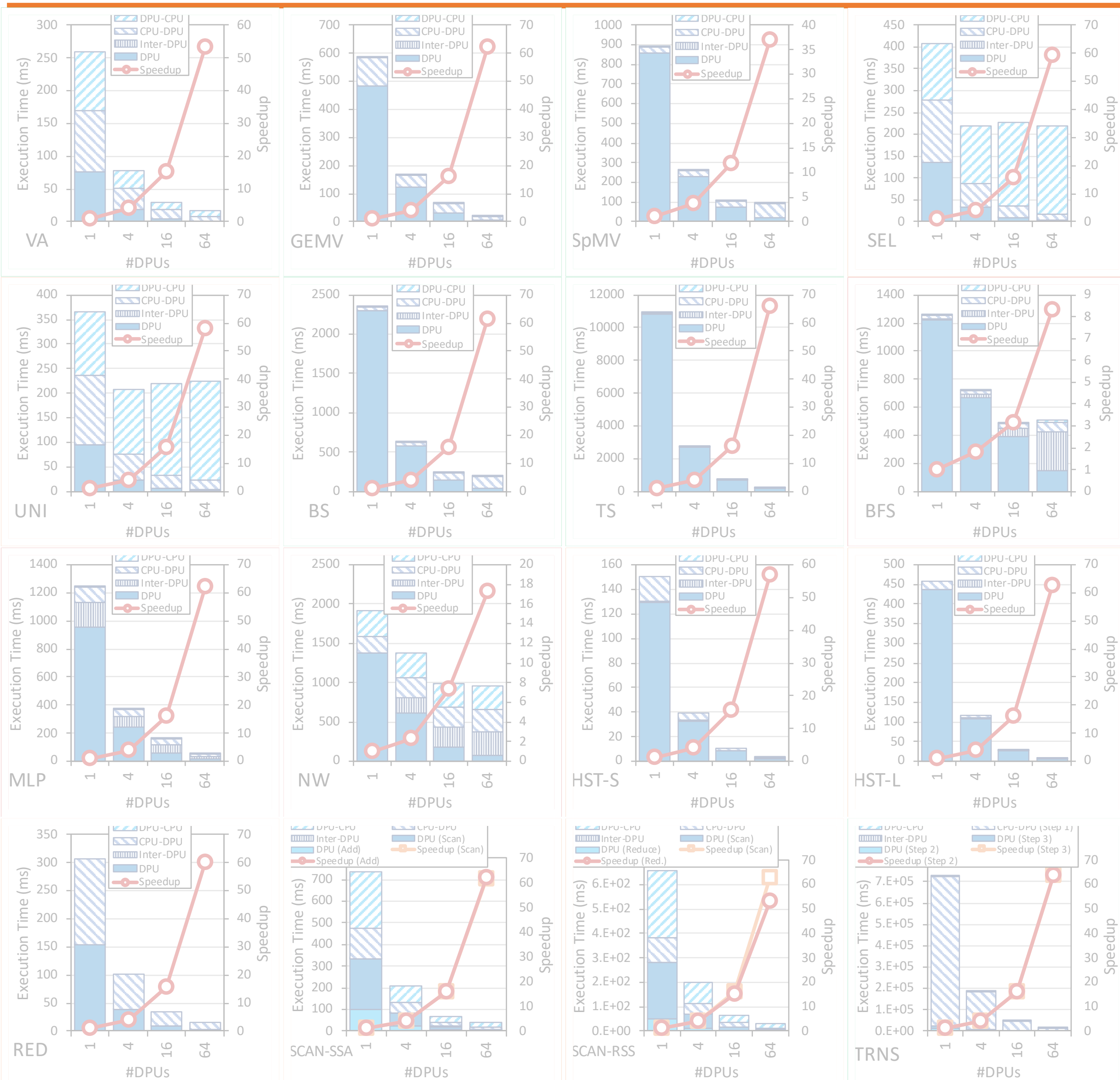
VA, GEMV, SpMV, SEL, UNI, BS, TS, MLP, HST-S, HSTS-L, RED, SCAN-SSA (both kernel), SCAN-RSS (both kernels), and TRNS (both kernels) scale linearly with the number of DPUs

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)

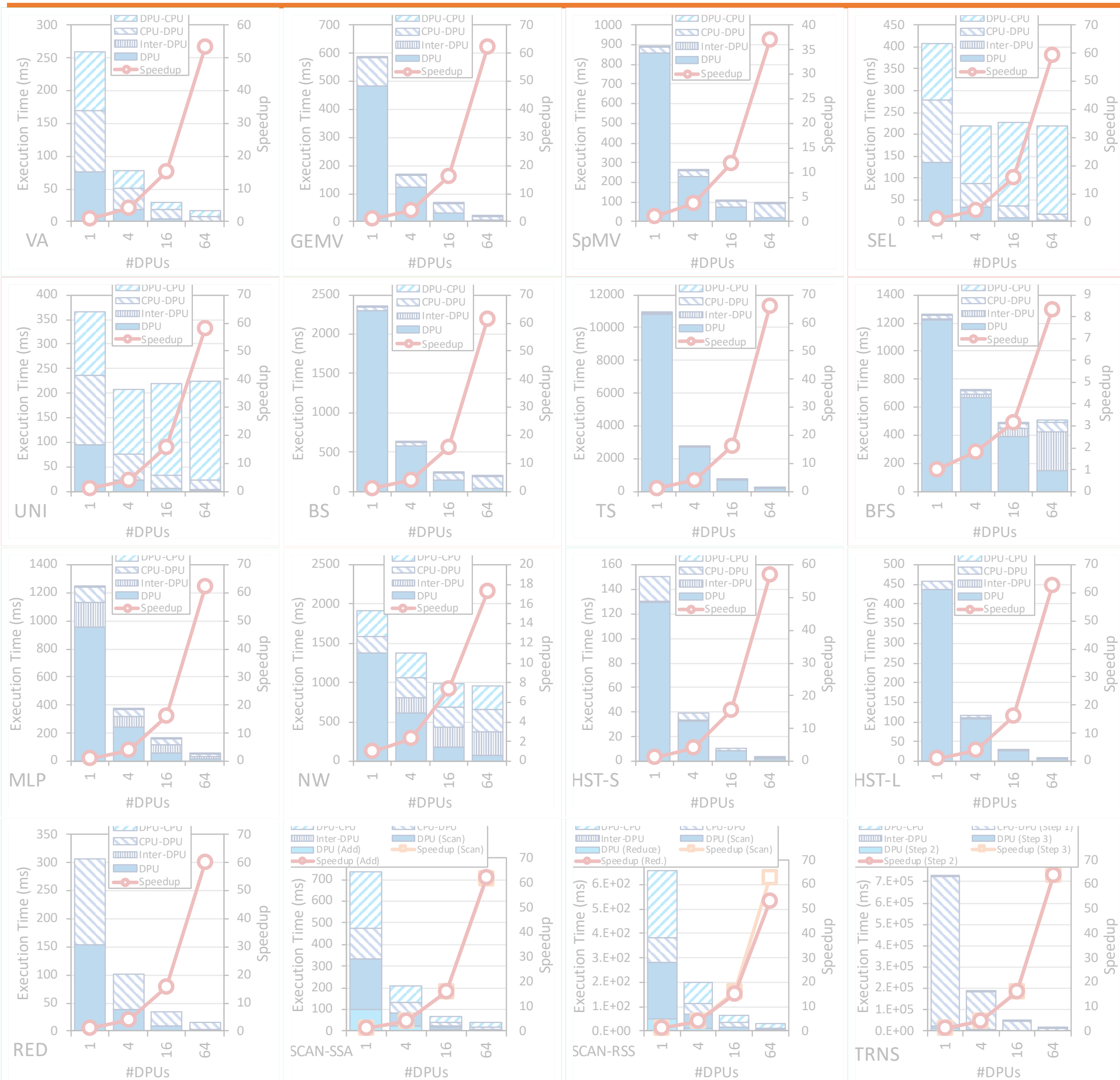


VA, GEMV, SpMV, BS, TS, TRNS do not need inter-DPU synchronization

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

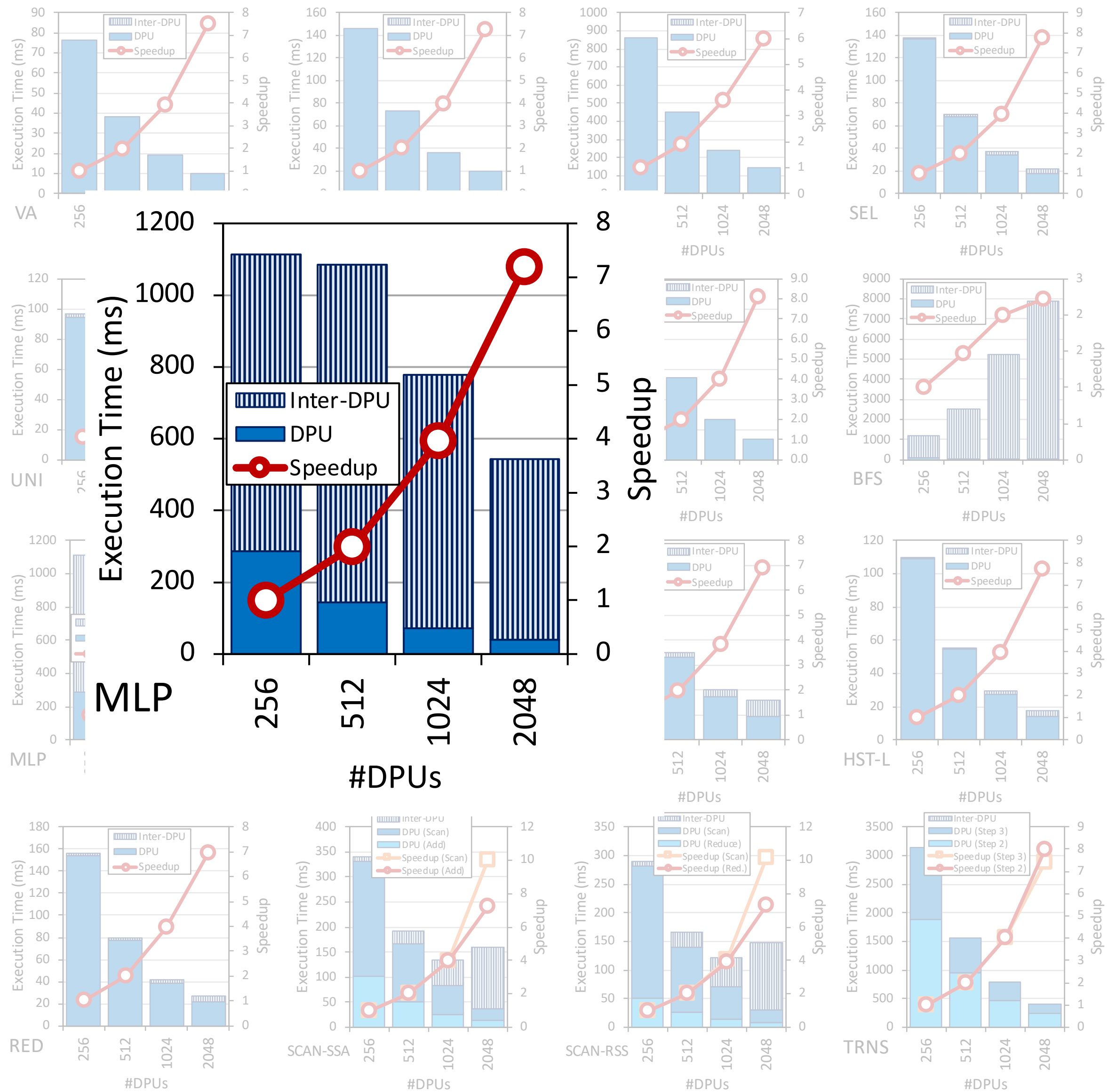
SpMV, SEL, UNI, BFS cannot use parallel transfers, as the transfer size per DPU is not fixed

PROGRAMMING RECOMMENDATION 5

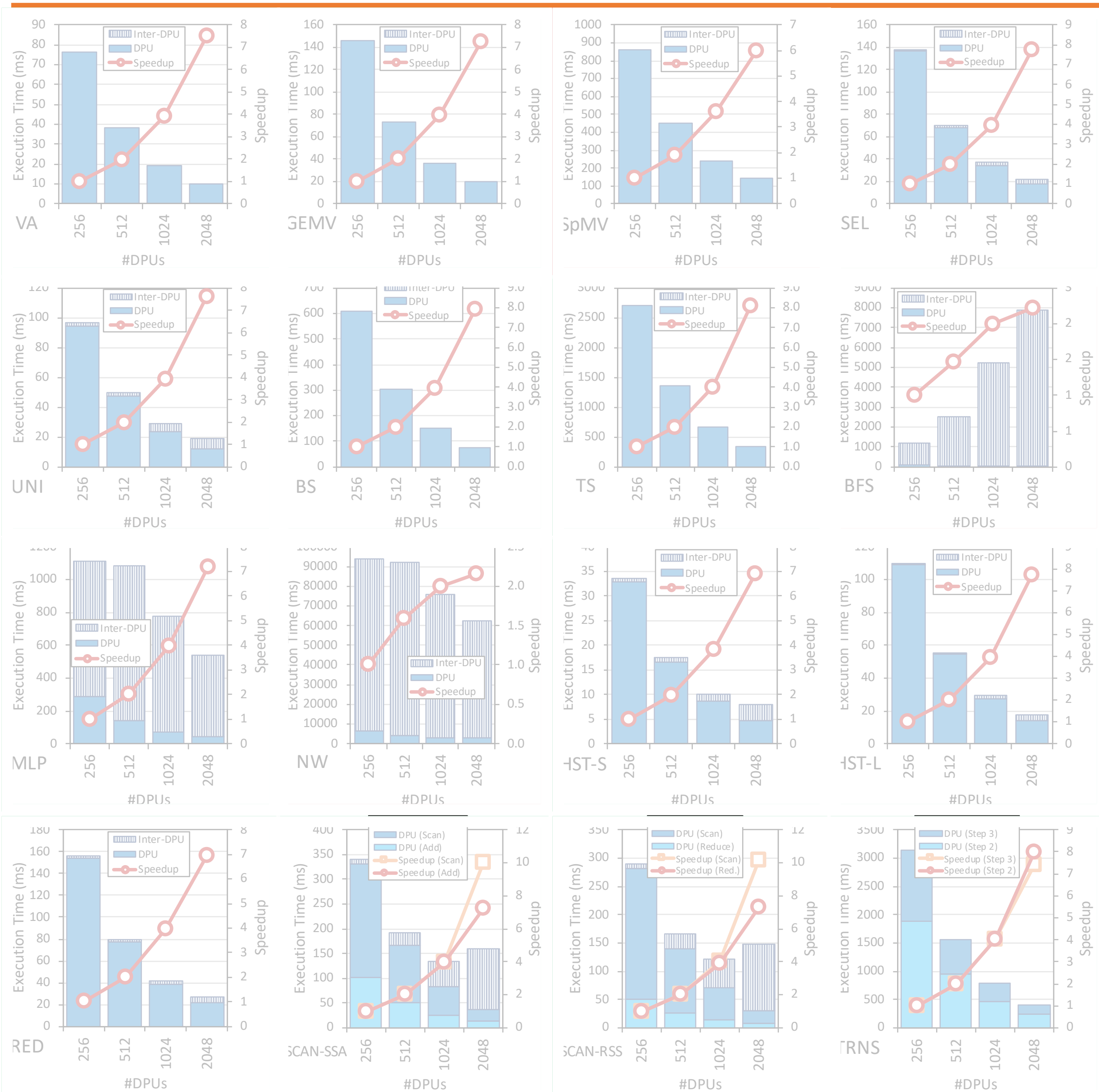
Parallel CPU-DPU/DPU-CPU transfers inside a rank of DPUs are recommended for real-world workloads when all transferred buffers are of the same size.

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)



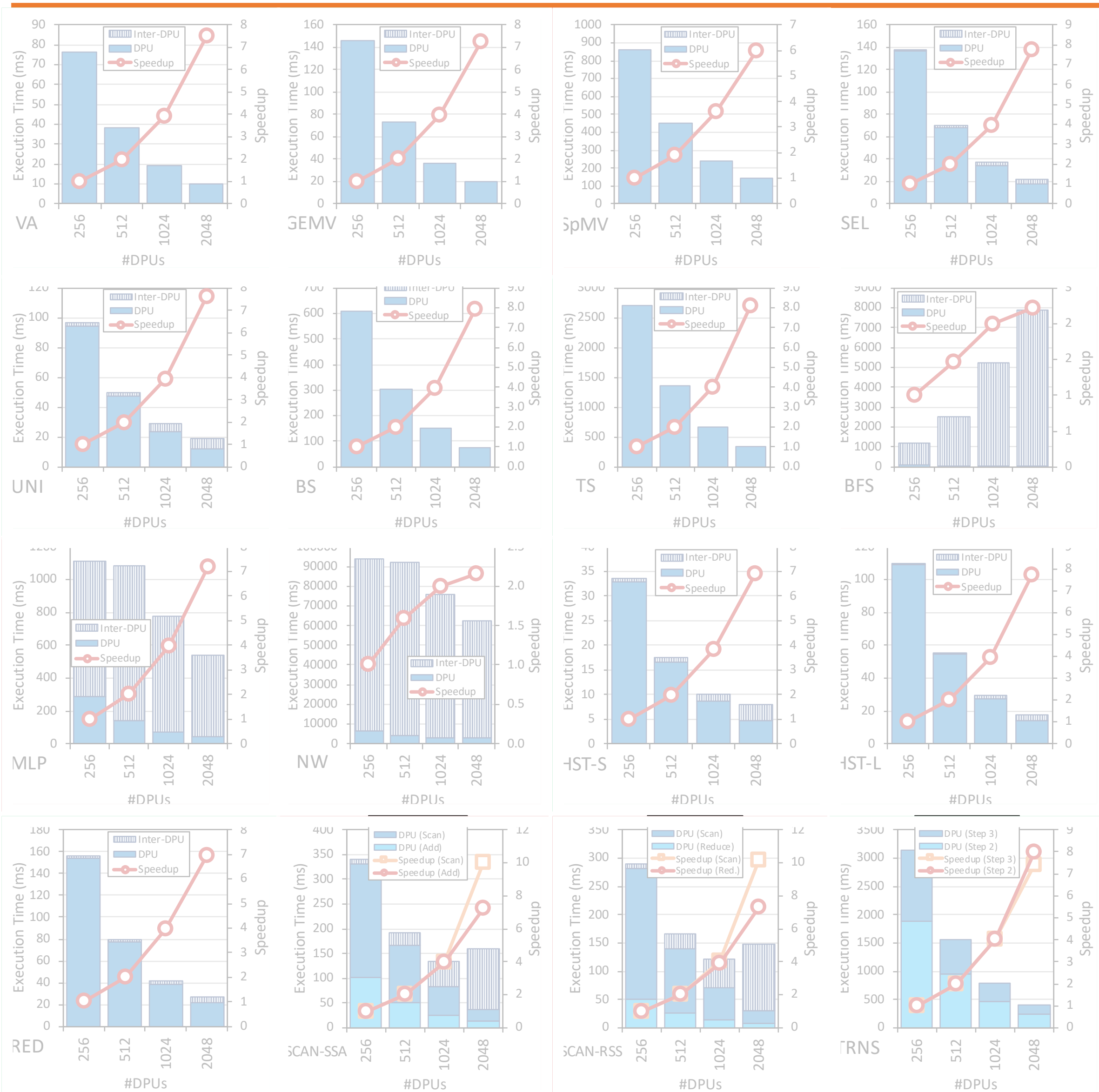
VA, GEMV, SEL, UNI, BS, TS, MLP, HST-S, HSTS-L, RED, SCAN-SSA (both kernel), SCAN-RSS (both kernels), and TRNS (both kernels) scale linearly with the number of DPUs

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

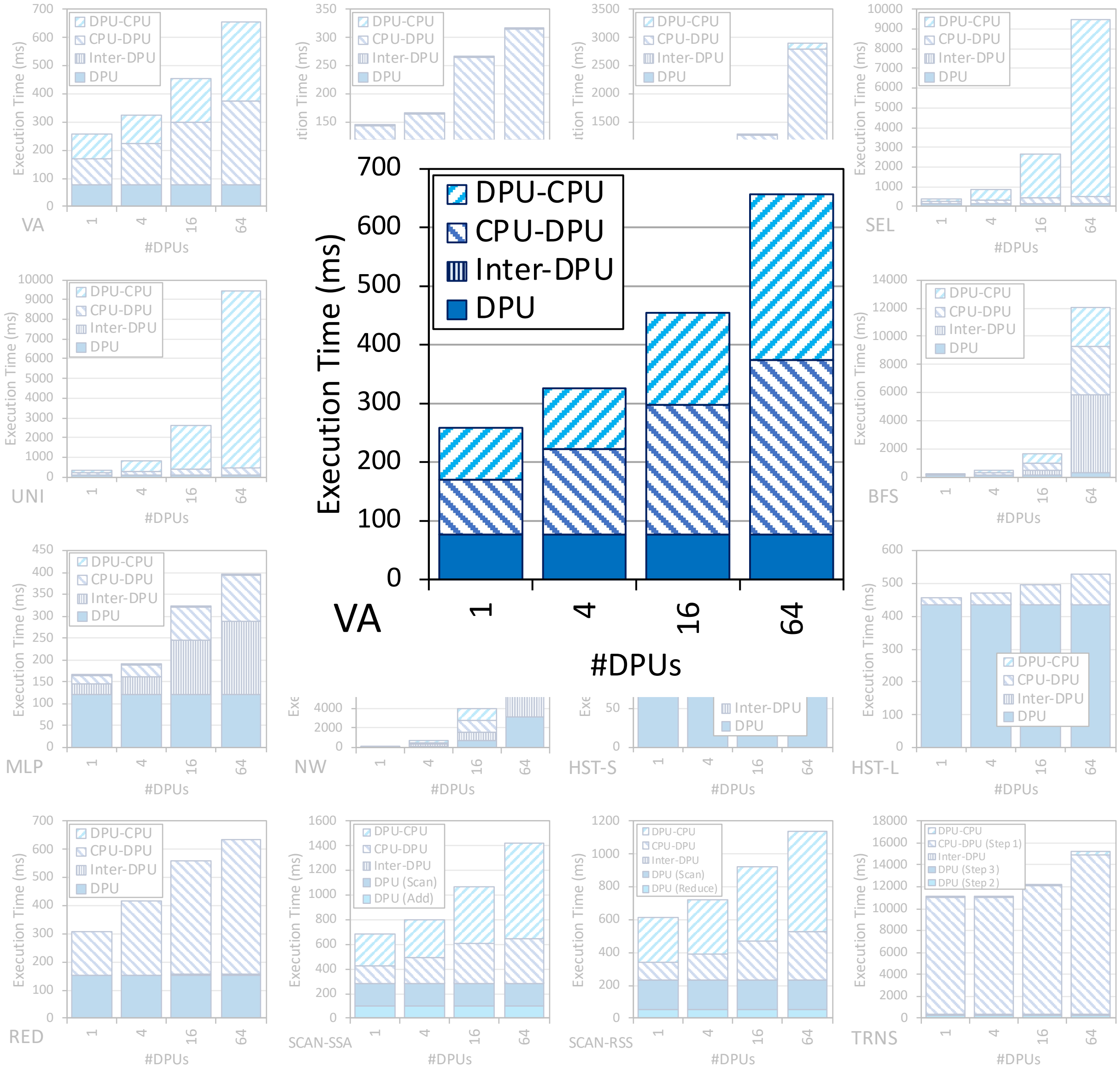
The overhead of merging partial results from DPUs in the host CPU is tolerable across all PrIM benchmarks that need it.

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).

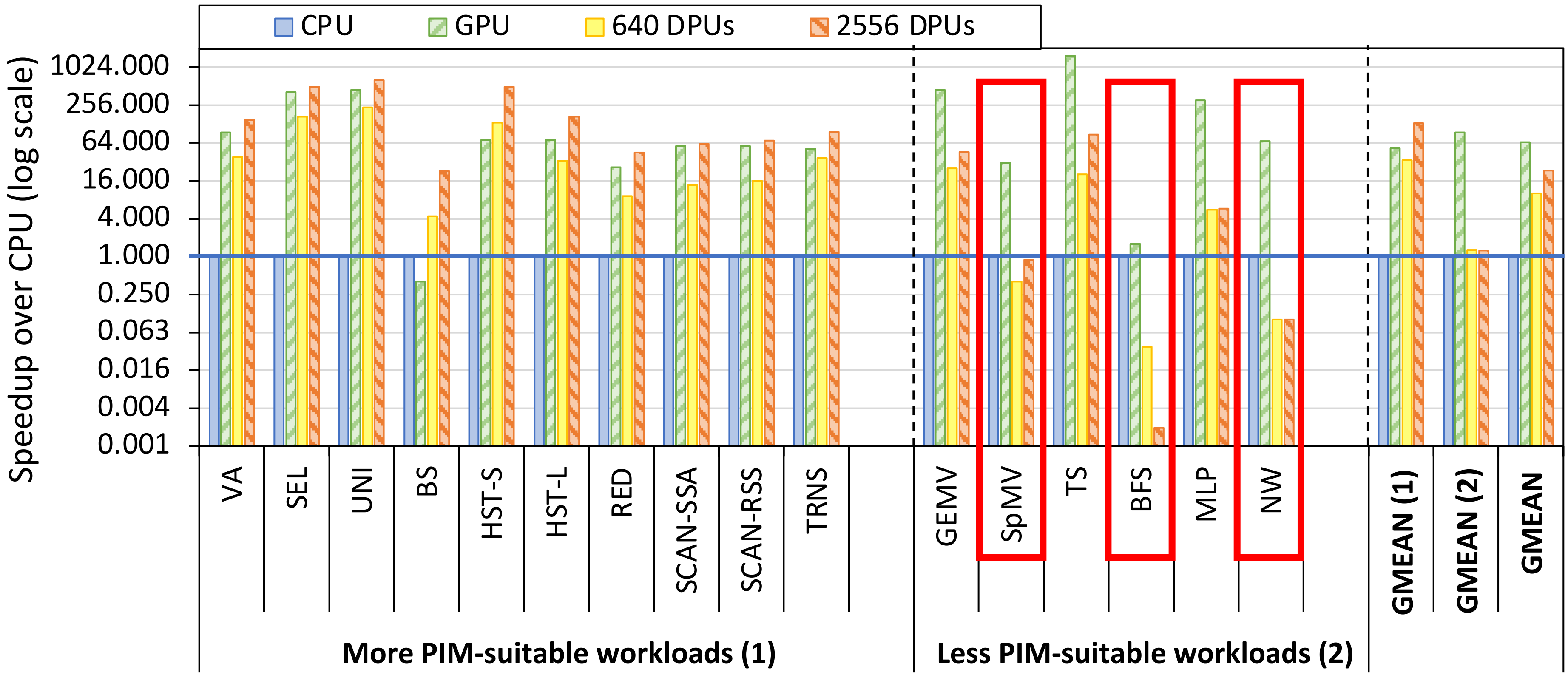
KEY OBSERVATION 18

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

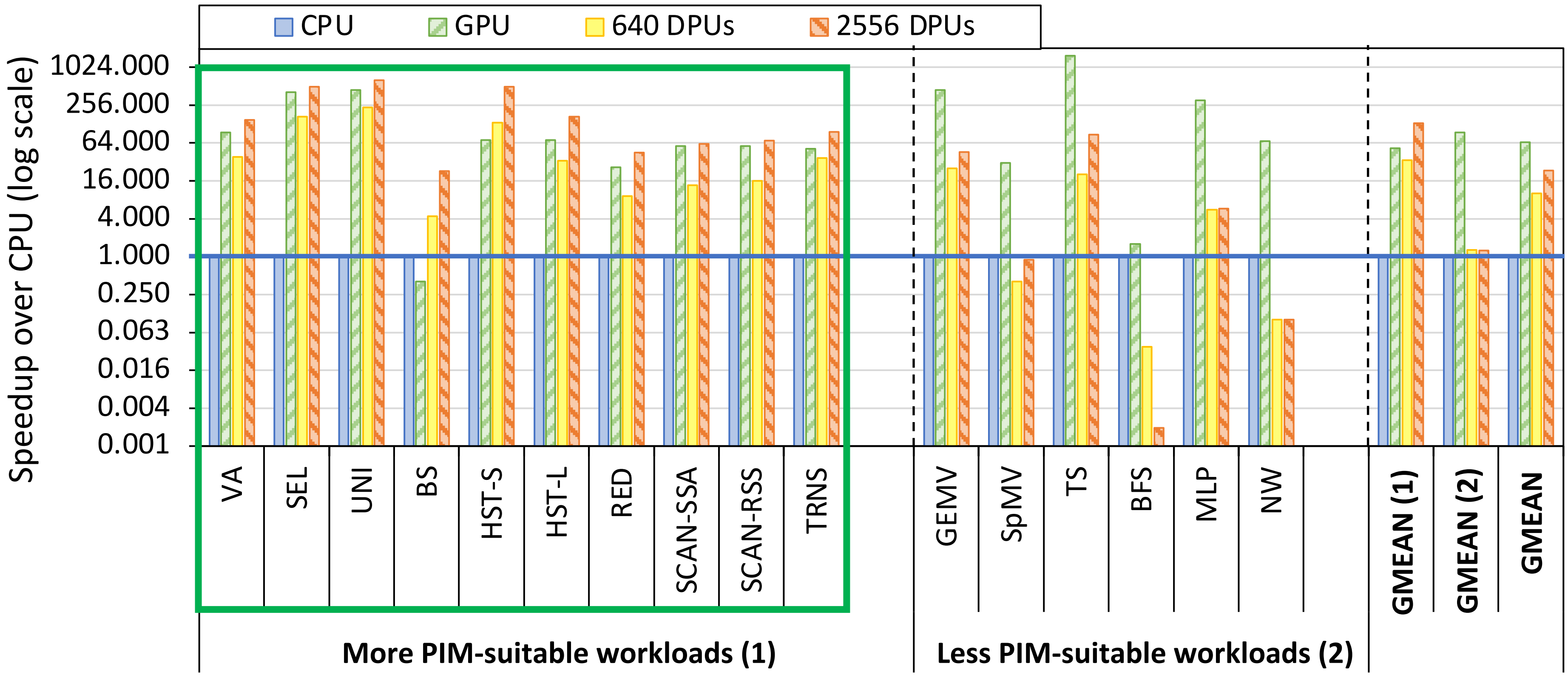
CPU/GPU: Performance Comparison (I)



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

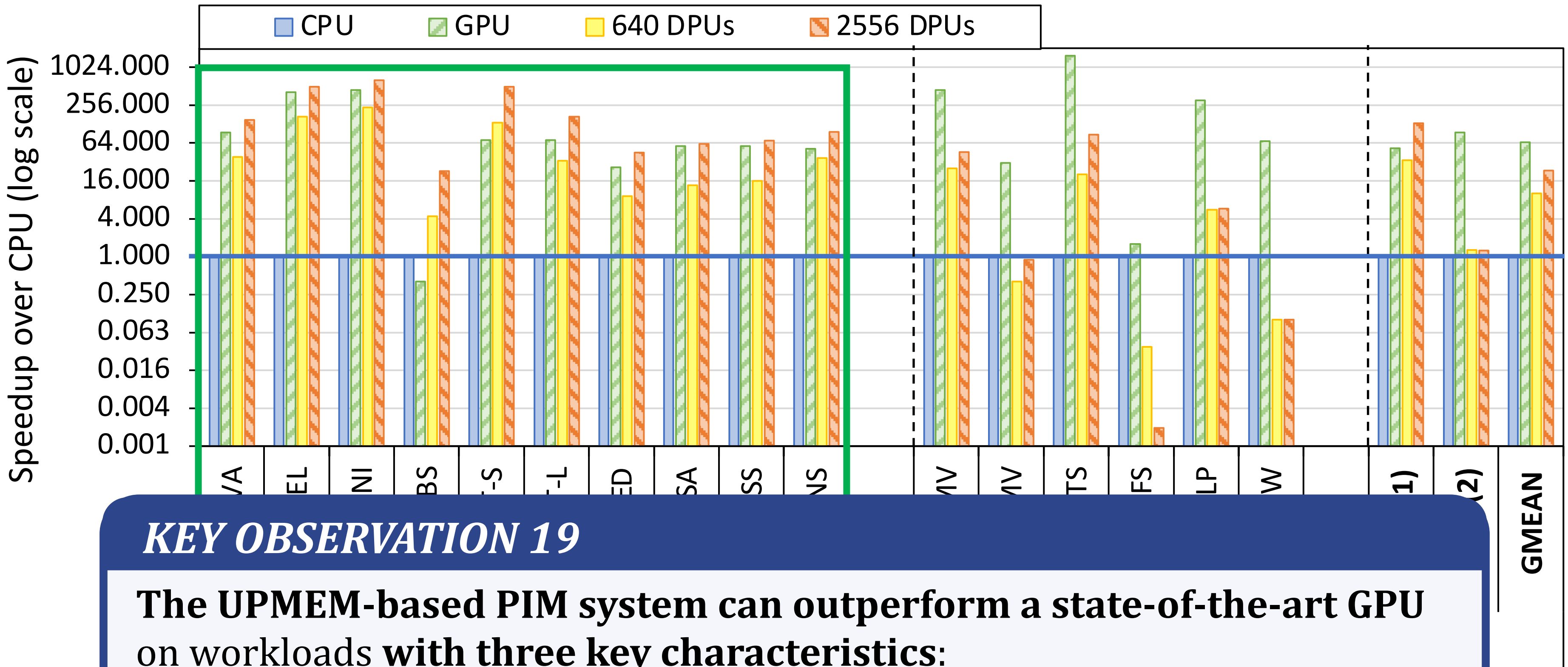
CPU/GPU: Performance Comparison (II)



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)



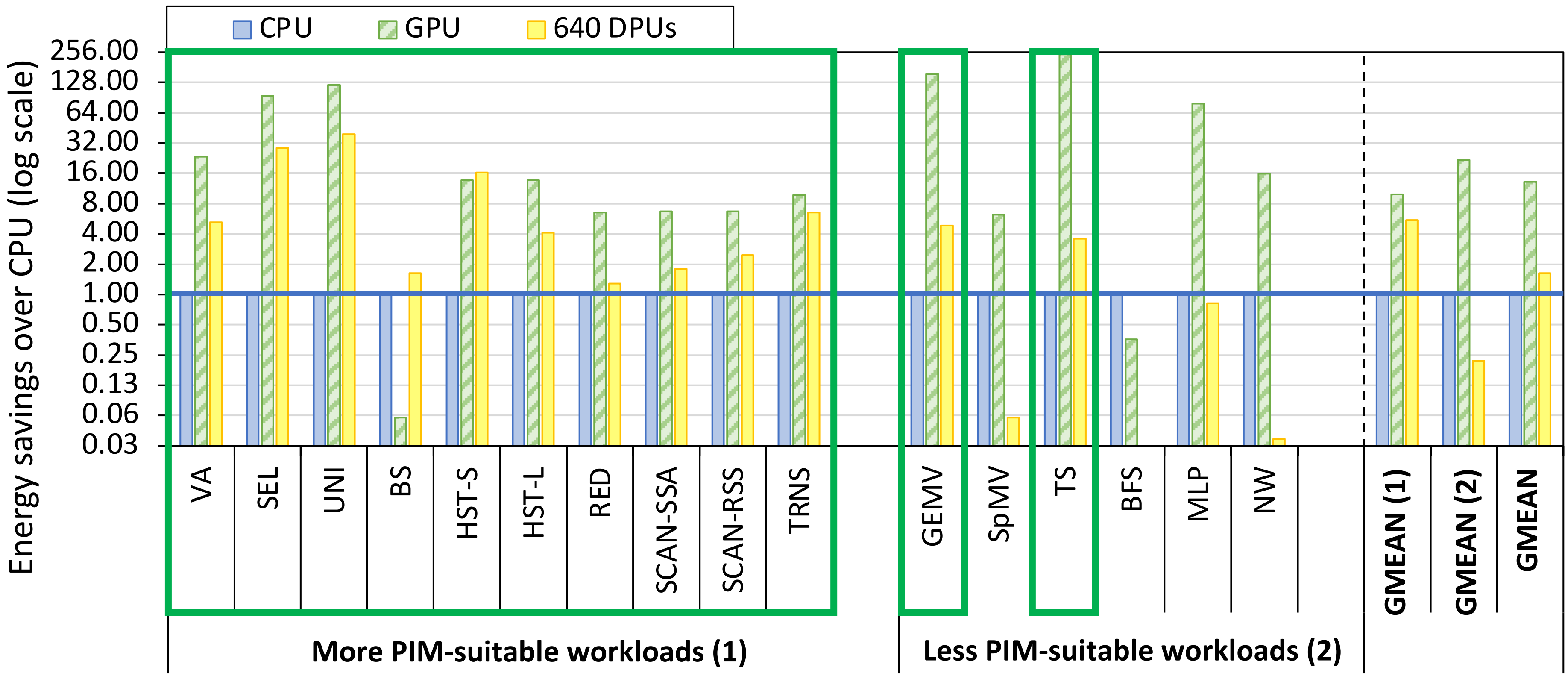
KEY OBSERVATION 19

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**.

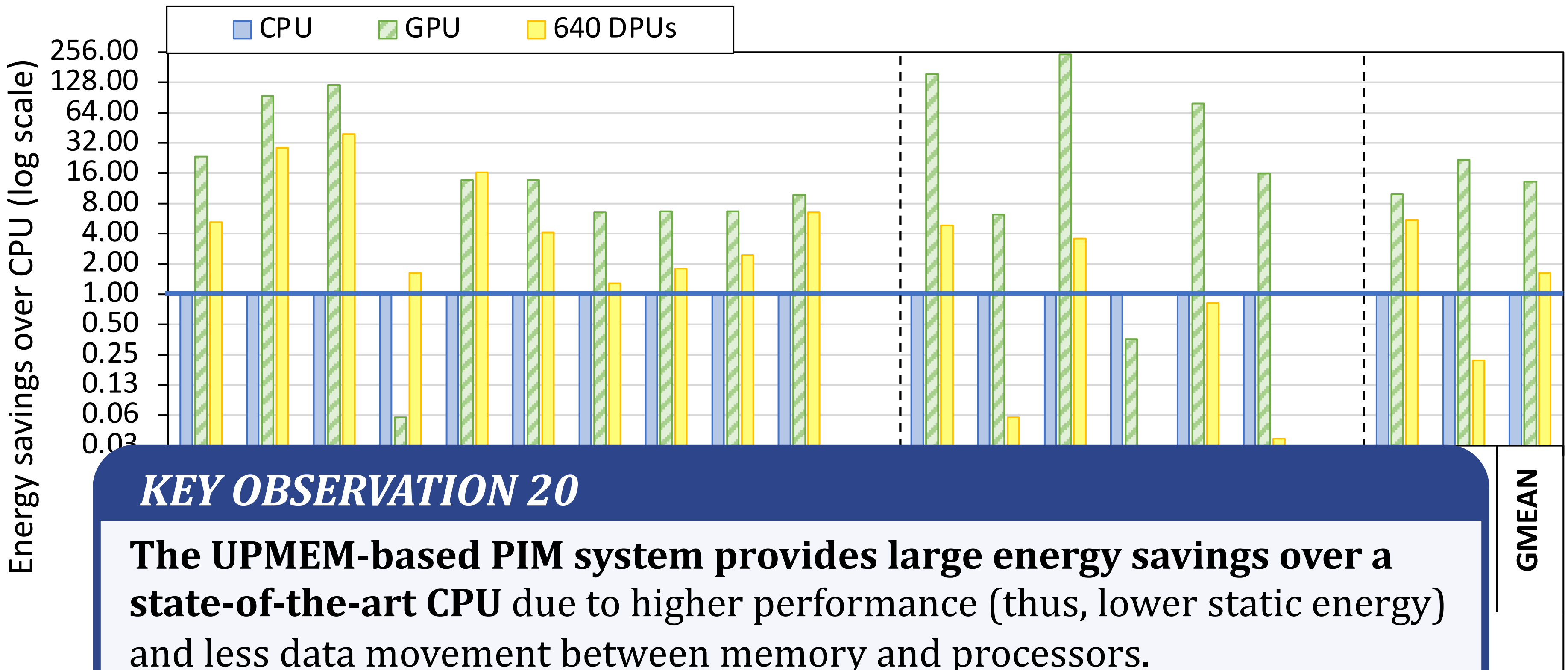
CPU/GPU: Energy Comparison (I)



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

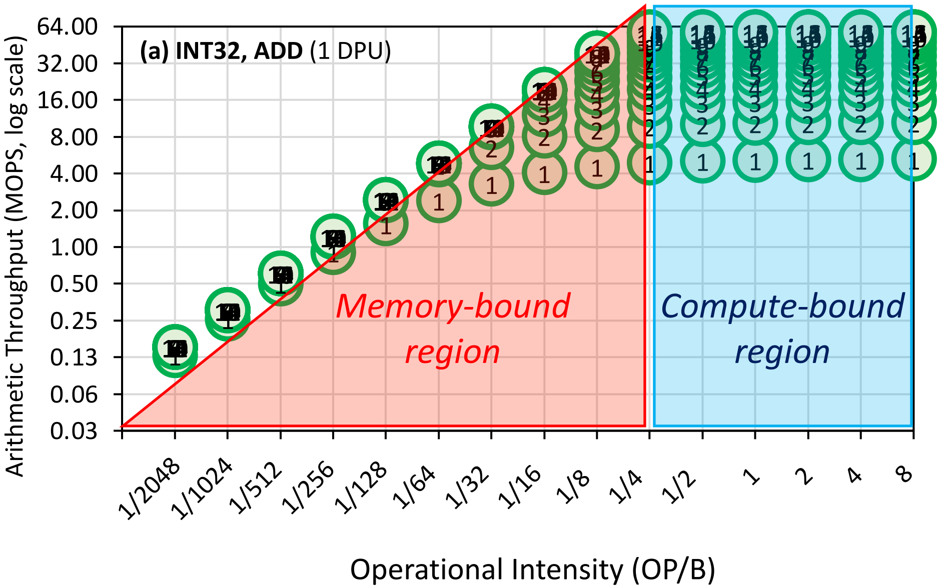
The UPMEM-based PIM system provides large energy savings over a state-of-the-art CPU due to higher performance (thus, lower static energy) and less data movement between memory and processors.

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

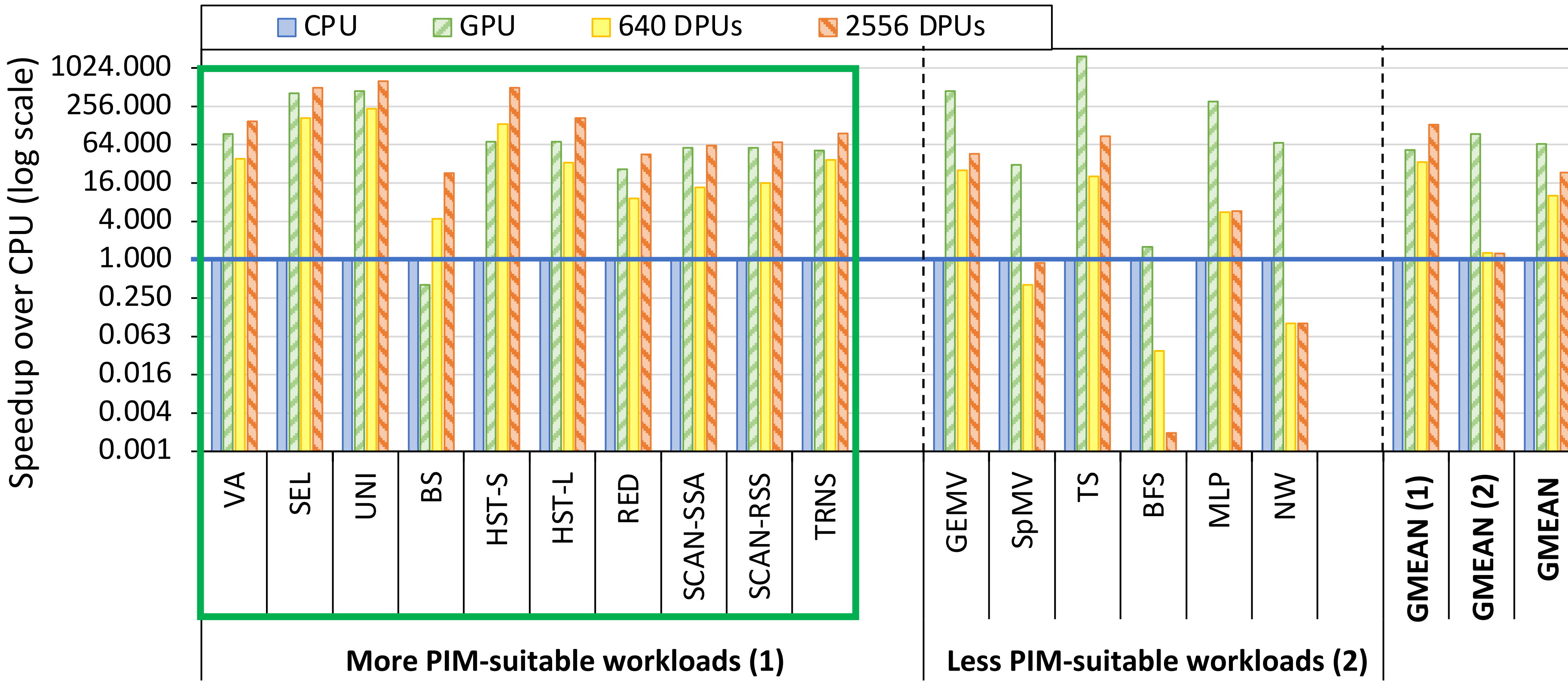


The throughput saturation point is as low as ¼ OP/B, i.e., 1 integer addition per every 32-bit element fetched

KEY TAKEAWAY 1

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

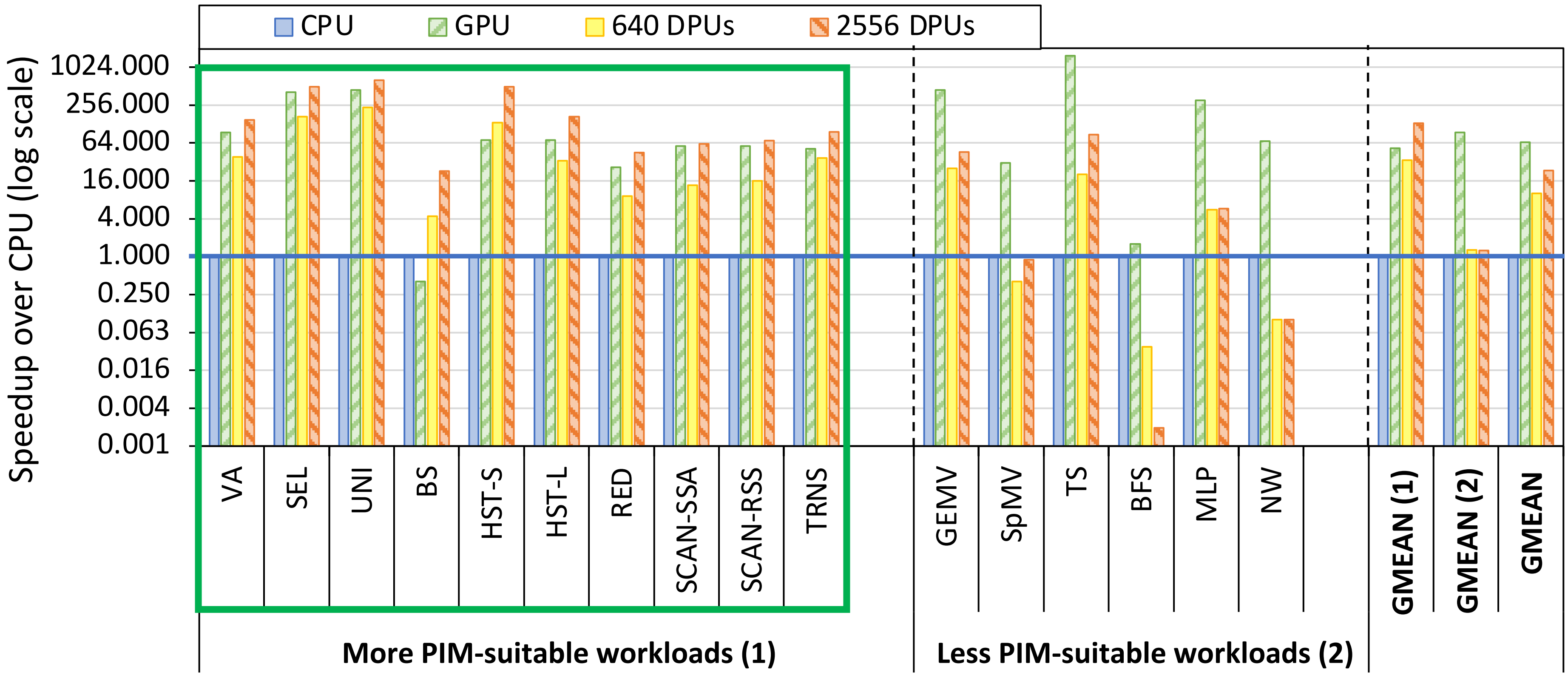
Key Takeaway 2



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 and energy efficiency on most of PrIM benchmarks.**

• UPMEM-based PIM systems **outperform state-of-the-art GPUs on a majority of 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.

Executive Summary

- Data movement between memory/storage units and compute units is a major contributor to execution time and energy consumption
- Processing-in-Memory (PIM) is a paradigm that can tackle the *data movement bottleneck*
  - Though explored for +50 years, technology challenges prevented the successful materialization
- UPMEM has designed and fabricated the first publicly-available real-world PIM architecture
  - DDR4 chips embedding in-order multithreaded DRAM Processing Units (DPUs)
- Our work:
  - Introduction to UPMEM programming model and PIM architecture
  - Microbenchmark-based characterization of the DPU
  - Benchmarking and workload suitability study
- Main contributions:
  - Comprehensive characterization and analysis of the first commercially-available PIM architecture
  - **PrIM** (Processing-In-Memory) benchmarks:
    - 16 workloads that are memory-bound in conventional processor-centric systems
    - Strong and weak scaling characteristics
  - Comparison to state-of-the-art CPU and GPU
- Takeaways:
  - Workload characteristics for PIM suitability
  - Programming recommendations
  - Suggestions and hints for hardware and architecture designers of future PIM systems
  - PrIM: (a) programming samples, (b) evaluation and comparison of current and future PIM systems

Understanding a Modern PIM Architecture

Understanding a Modern Processing-in-Memory Architecture: Benchmarking and Experimental Characterization

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

CMU-SAFARI / **prim-benchmarks**

Unwatch 2 | Star 2 | Fork 1

Code | Issues | Pull requests | Actions | Projects | Wiki | Security | Insights | Settings

main | prim-benchmarks / **README.md** | Go to file

Juan Gomez Luna PrIM -- first commit | Latest commit 3de4b49 9 days ago | **History**

1 contributor

168 lines (132 sloc) | 5.79 KB | Raw | Blame

PrIM (Processing-In-Memory Benchmarks)

PrIM is the first benchmark suite for a real-world processing-in-memory (PIM) architecture. PrIM is developed to evaluate, analyze, and characterize the first publicly-available real-world processing-in-memory (PIM) architecture, the UPMEM PIM architecture. The UPMEM PIM architecture combines traditional DRAM memory arrays with general-purpose in-order cores, called DRAM Processing Units (DPUs), integrated in the same chip.

PrIM provides a common set of workloads to evaluate the UPMEM PIM architecture with and can be useful for programming, architecture and system researchers all alike to improve multiple aspects of future PIM hardware and software. The workloads have different characteristics, exhibiting heterogeneity in their memory access patterns, operations and data types, and communication patterns. This repository also contains baseline CPU and GPU implementations of PrIM benchmarks for comparison purposes.

PrIm also includes a set of microbenchmarks can be used to assess various architecture limits such as compute throughput and memory bandwidth.

Understanding a Modern Processing-in-Memory Architecture:

Benchmarking and Experimental Characterization

Juan Gómez Luna, Izzat El Hajj,
Ivan Fernandez, Christina Giannoula,
Geraldo F. Oliveira, Onur Mutlu

el1goluj@gmail.com

https://arxiv.org/pdf/2105.03814.pdf
https://github.com/CMU-SAFARI/prim-benchmarks

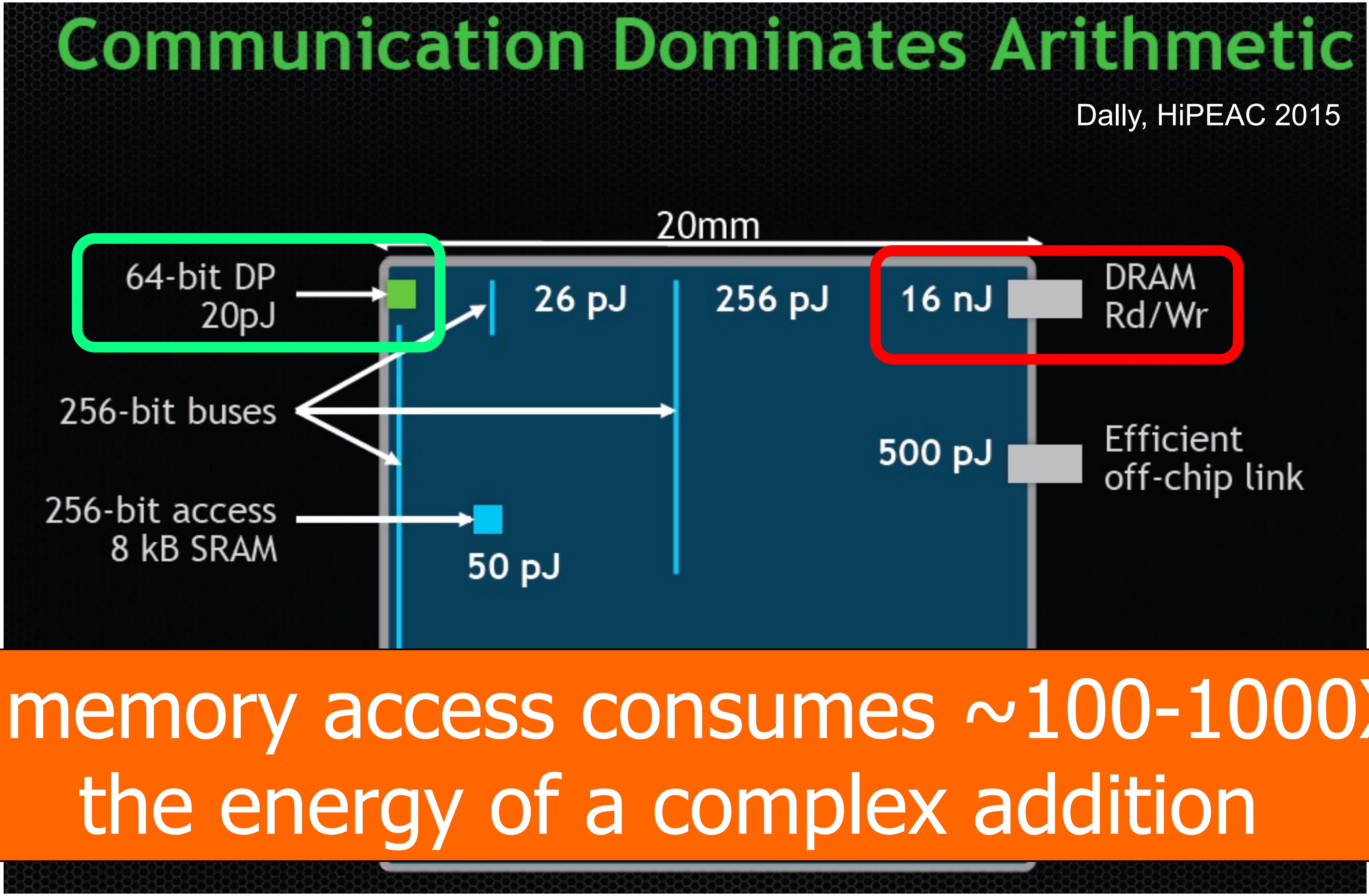
ETH Zürich

SAFARI

Resources

- UPMEM SDK documentation
  - https://sdk.upmem.com/master/00_ToolchainAtAGlance.html
- Fabrice Devaux's presentation at HotChips 2019
  - https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8875680
- Onur's lectures and talks

Data Movement vs. Computation Energy



A memory access consumes ~100-1000X the energy of a complex addition

O. Mutlu, "Processing Data Where It Makes Sense: Enabling In-Memory Computation," keynote talk at MST 2017
https://people.inf.ethz.ch/omutlu/pub/onur-MST-Keynote- EnablingInMemoryComputation-October-27-2017-unrolled-FINAL.pptx

Characterization of UPMEM PIM

- Microbenchmarks
  - Pipeline throughput
  - STREAM benchmark: WRAM, MRAM
  - Strided accesses and GUPS
  - Throughput vs. Operational intensity
  - CPU-DPU data transfers
- Real-world benchmarks
  - Dense linear algebra
  - Sparse linear algebra
  - Databases
  - Graph processing
  - Bioinformatics
  - Etc.

Banner Colors

This is a question or an observation

This is an answer from, e.g., UPMEM documentation or our own research

This is an idea or a discussion starter, an opportunity for brainstorming

DPU Sharing? Security Implications?

- DPUs cannot be shared across multiple CPU processes
  - There are so many DPUs in the system that there is no need for sharing

- According to UPMEM, this assumption makes things simpler

Is it possible to perform RowHammer bit flips?
Can we attack the previous or the next application that runs on a DPU?

RowHammer patents and Giray's paper?

More Questions and Ideas?

How do we handle memory coherence,
memory oversubscription, etc.?

They are programmer's responsibility

A software library to handle
memory management transparently to programmers

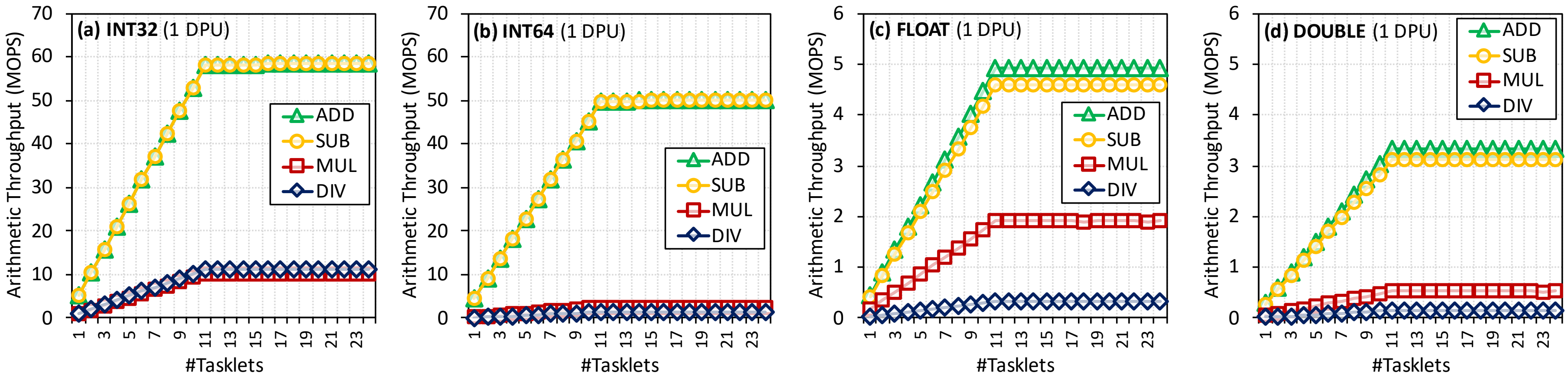
ASPLOS 2010

An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems

Isaac Gelado Javier Cabezas
Nacho Navarro
Universitat Politecnica de Catalunya
{igelado, jcabezas, nacho}@ac.upc.edu

John E. Stone Sanjay Patel
Wen-mei W. Hwu
University of Illinois
{jestone, sjp, hwu}@illinois.edu

Arithmetic Throughput (II)

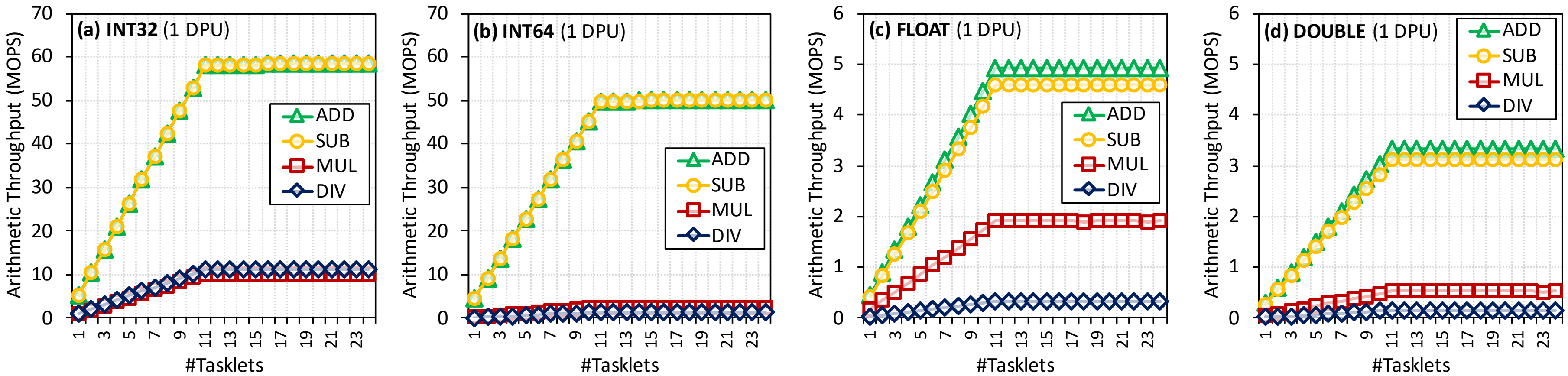


Huge throughput difference between add/sub and mul/div

DPUs do not have a 32-bit multiplier.
mul/div implementation is based on bit shifting and addition: maximum of 32 cycles (instructions) to complete

There is an 8-bit multiplier in the pipeline.
Would it be possible to use it for more efficient implementation?

Arithmetic Throughput (III)

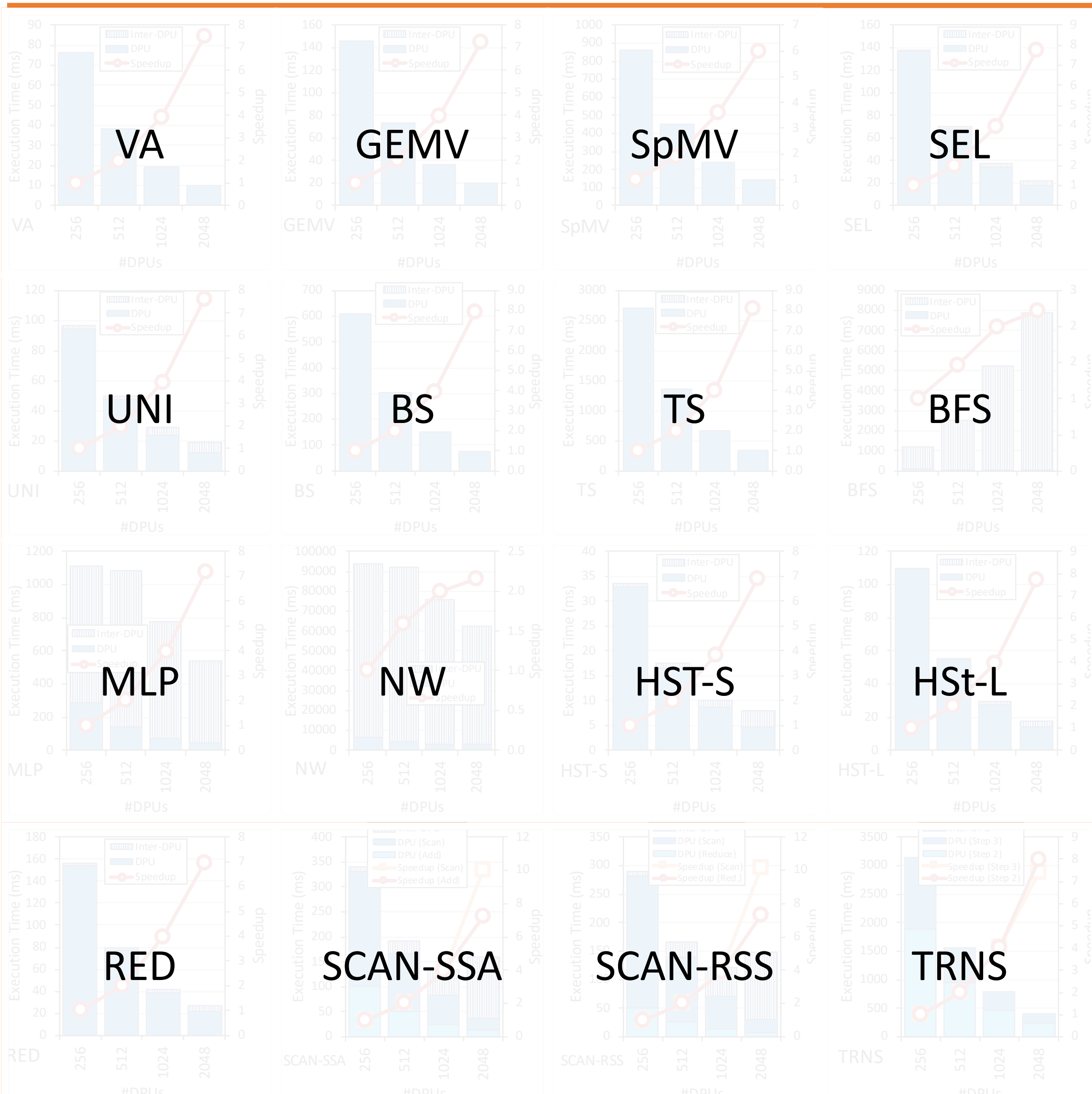


Huge throughput difference between int32/int64 and float/double

DPUs do not have floating point units.
Software emulation for floating point computations

More efficient algorithms based on other formats?
E.g., posit, TF32?

Strong Scaling: 32 Ranks

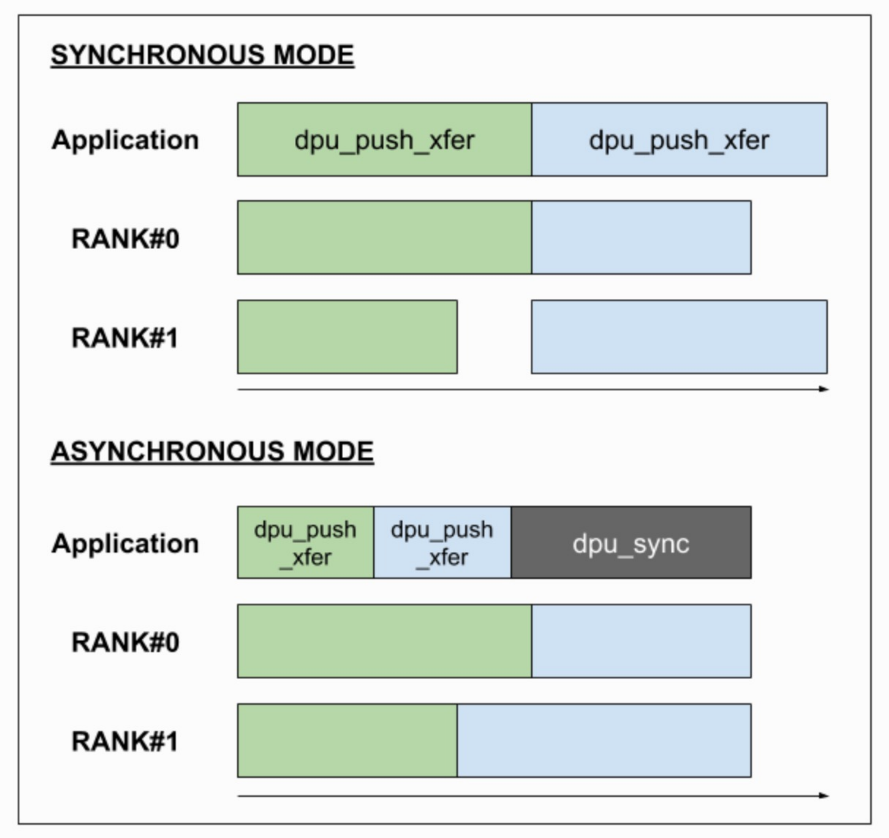


DSLs, High-level Programming

- Tangram

Backup: CPU-DPU Data Transfers

- Parallel asynchronous mode
  - Two transfers to a set of two ranks



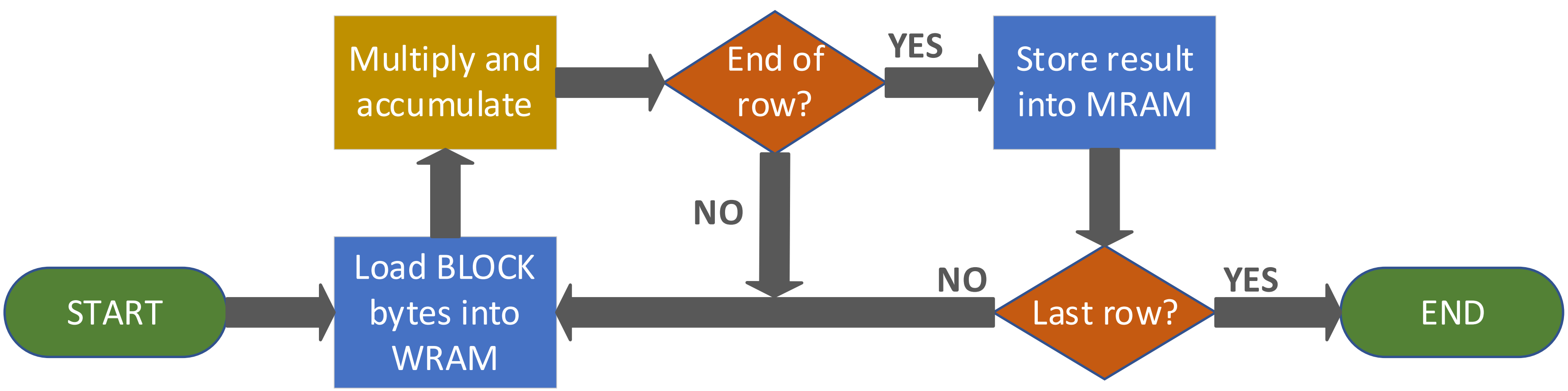
https://sdk.upmem.com/master/032_DPURuntimeService_HostCommunication.html#dpu-rank-transfer-interface-label

GEMV: Parallelization Approach

- GEMV (general matrix-vector multiplication)

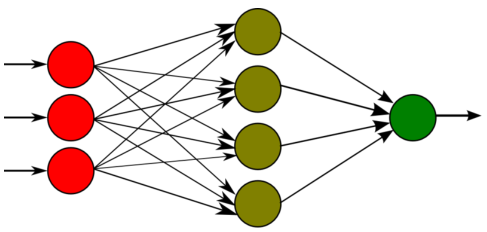
$$\begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 0 \\ 6 & 0 & 0 & 7 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 5 \\ 1 \\ 8 \end{bmatrix} = \begin{bmatrix} 4 \\ 47 \\ 5 \\ 68 \end{bmatrix}$$

- Workload distribution
  - chunk_size = (num_rows / (nr_ranks * nr_dpus)), to each DPU
  - chunk_size / NR_TASKLETS, to each tasklet

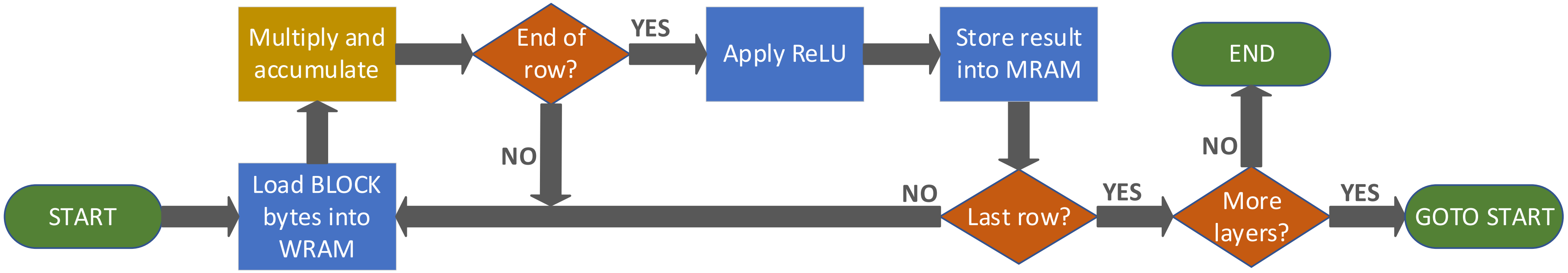


MLP: Parallelization Approach

- MLP (multi-layer perceptron), based on GEMV



- Workload distribution
  - chunk_size = (num_rows / (nr_ranks * nr_dpus)), to each DPU
  - chunk_size / NR_TASKLETS, to each tasklet



PIM Review and Open Problems

Processing Data Where It Makes Sense: Enabling In-Memory Computation

Onur Mutlu[a,b], Saugata Ghose[b], Juan Gómez-Luna[a], Rachata Ausavarungnirun[b,c]

[a] *ETH Zürich*
[b] *Carnegie Mellon University*
[c] *King Mongkut's University of Technology North Bangkok*

Onur Mutlu, Saugata Ghose, Juan Gomez-Luna, and Rachata Ausavarungnirun, **"Processing Data Where It Makes Sense: Enabling In-Memory Computation"**
Invited paper in Microprocessors and Microsystems ***(MICPRO)***, June 2019.
[arXiv version]

https://arxiv.org/pdf/1903.03988.pdf

PIM Review and Open Problems (II)

A Workload and Programming Ease Driven Perspective of Processing-in-Memory

Saugata Ghose[†] Amirali Boroumand[†] Jeremie S. Kim[†§] Juan Gómez-Luna[§] Onur Mutlu[§†]

[†]*Carnegie Mellon University* [§]*ETH Zürich*

Saugata Ghose, Amirali Boroumand, Jeremie S. Kim, Juan Gomez-Luna, and Onur Mutlu,
"Processing-in-Memory: A Workload-Driven Perspective"
Invited Article in IBM Journal of Research & Development, Special Issue on Hardware for Artificial Intelligence, to appear in November 2019.
[Preliminary arXiv version]