

# Understanding a Modern Processing-in-Memory Architecture

Juan Gómez Luna, Izzat El Hajj,  
Iván Fernández, Christina Giannoula,  
Geraldo F. de Oliveira, Onur Mutlu

# Executive Summary

---

- **Processing-in-Memory** is a paradigm that can tackle the **data movement bottleneck**
- Though promising, **there were not real-world devices** that represent a baseline for our research
  - Simulation models for our PIM architecture proposals, where the baseline is typically the host CPU (or GPU)
- UPMEM has designed and fabricated **the first publicly-available real-world PIM architecture**
  - DDR4 chips embedding in-order multithreaded DPUs
- Goals
  - **Introduction** to UPMEM programming model and PIM architecture
  - **Understanding** the UPMEM PIM architecture

# Outline

---

- Introduction
  - Accelerator Model
  - System Integration
- UPMEM PIM Programming
  - Vector Addition
  - DPU Allocation
  - CPU-DPU Data Transfers
  - DPU Kernel Launch and Execution
- DRAM Processing Unit
  - Arithmetic throughput
- DPU Kernel Execution
  - Vector Addition
  - Tasklet Synchronization
    - Parallel Reduction
- Characterization of the UPMEM PIM Architecture

# Outline

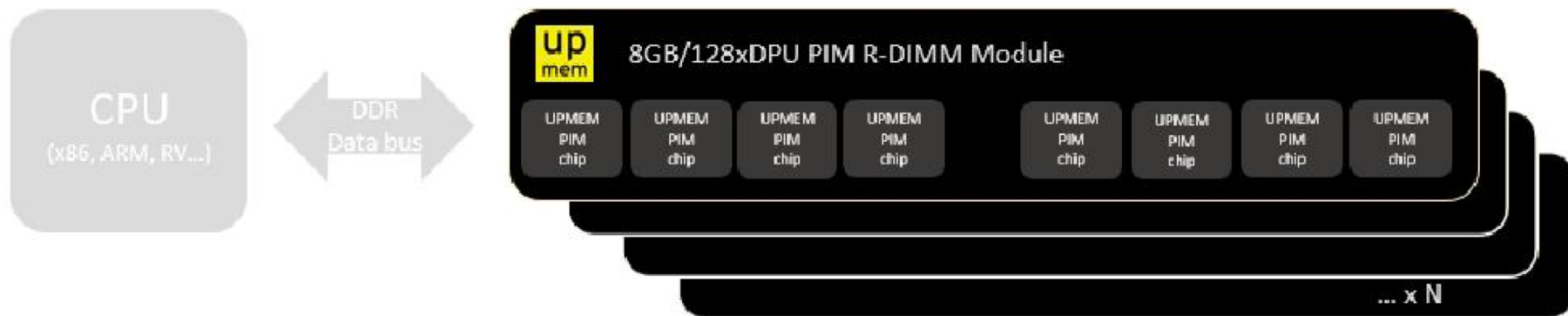
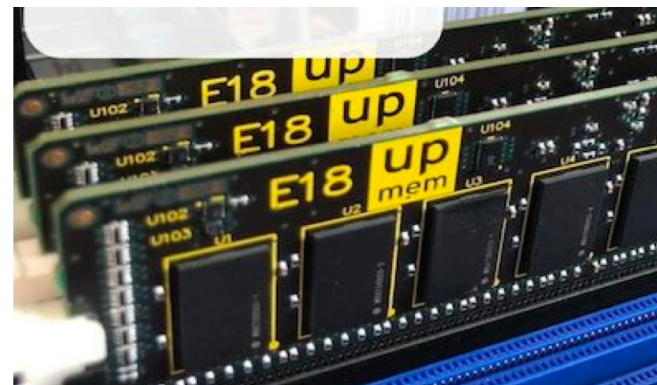
---

- Introduction
  - Accelerator Model
  - System Integration
- UPMEM PIM Programming
  - Vector Addition
  - DPU Allocation
  - CPU-DPU Data Transfers
  - DPU Kernel Launch and Execution
- DRAM Processing Unit
  - Arithmetic throughput
- DPU Kernel Execution
  - Vector Addition
  - Tasklet Synchronization
    - Parallel Reduction
- Characterization of the UPMEM PIM Architecture



# 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



# 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*

Copyright UPMEM® 2019

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 04,2020 at 13:55:41 UTC from IEEE Xplore. Restrictions apply.

HOT CHIPS 31

up  
mem

# 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

---

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

Onur Mutlu<sup>a,b</sup>, Saugata Ghose<sup>b</sup>, Juan Gómez-Luna<sup>a</sup>, Rachata Ausavarungnirun<sup>b,c</sup>

<sup>a</sup>*ETH Zürich*

<sup>b</sup>*Carnegie Mellon University*

<sup>c</sup>*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]*

# PIM Review and Open Problems (II)

---

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

Saugata Ghose<sup>†</sup>   Amirali Boroumand<sup>†</sup>   Jeremie S. Kim<sup>†§</sup>   Juan Gómez-Luna<sup>§</sup>   Onur Mutlu<sup>§†</sup>

<sup>†</sup>*Carnegie Mellon University*

<sup>§</sup>*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]

# 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

But the PIM Graal is worth it !

## 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

Copyright UPMEM® 2019

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 04, 2020 at 13:55:41 UTC from IEEE Xplore. Restrictions apply.

HOT CHIPS 31



# UPMEM Patent

(12) <b>United States Patent</b>		(10) <b>Patent No.:</b>	<b>US 10,324,870 B2</b>		
<b>Devaux et al.</b>		(45) <b>Date of Patent:</b>	<b>Jun. 18, 2019</b>		
(54) <b>MEMORY CIRCUIT WITH INTEGRATED PROCESSOR</b>		(56)	<b>References Cited</b>		
		U.S. PATENT DOCUMENTS			
(71) Applicant:	UPMEM, Grenoble (FR)	5,666,485 A *	9/1997	Suresh ..... G06F 13/1605 710/113	
(72) Inventors:	Fabrice Devaux, La Conversion (CH); Jean-François Roy, Grenoble (FR)	6,463,001 B1	10/2002	Williams	
		7,349,277 B2 *	3/2008	Kinsley ..... G11C 11/406 365/193	
(73) Assignee:	UPMEM, Grenoble (FR)	8,438,358 B1 *	5/2013	Kraipak ..... G11C 7/04 711/167	
(*) Notice:	Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.		(Continued)		
		FOREIGN PATENT DOCUMENTS			
(21) Appl. No.:	15/551,418	EP	0780768 A1	6/1997	
		JP	H03109661 A	5/1991	
(22) PCT Filed:	Feb. 12, 2016	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.

# Accelerator Model (I)

---

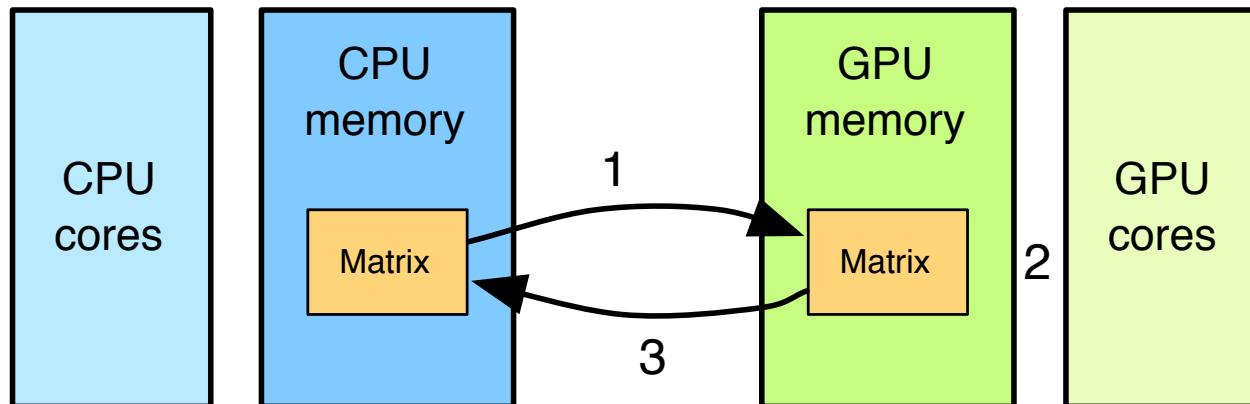
- Integration of UPMEM DIMMs in a system follows an **accelerator model**
  - UPMEM DIMMs coexist with conventional DIMMs
- UPMEM DIMMs can be seen as a **loosely coupled accelerator**
  - Explicit data movement between the main processor (CPU) and the accelerator (UPMEM)
  - Explicit kernel launch onto 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)



# 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

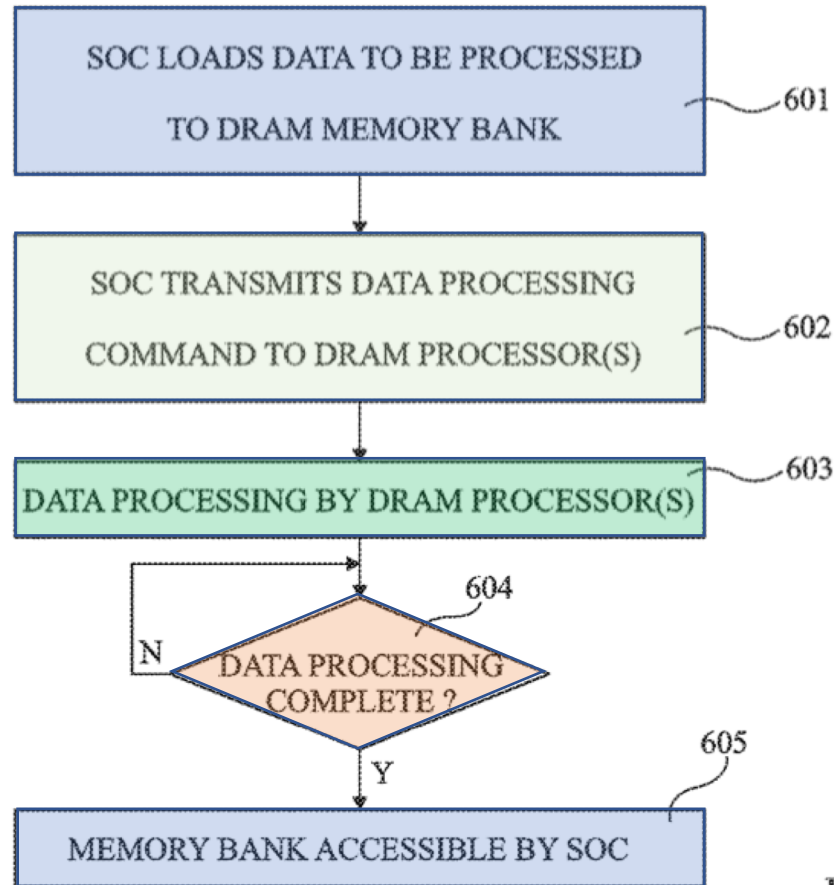


Fig 6

# System Organization (I)

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

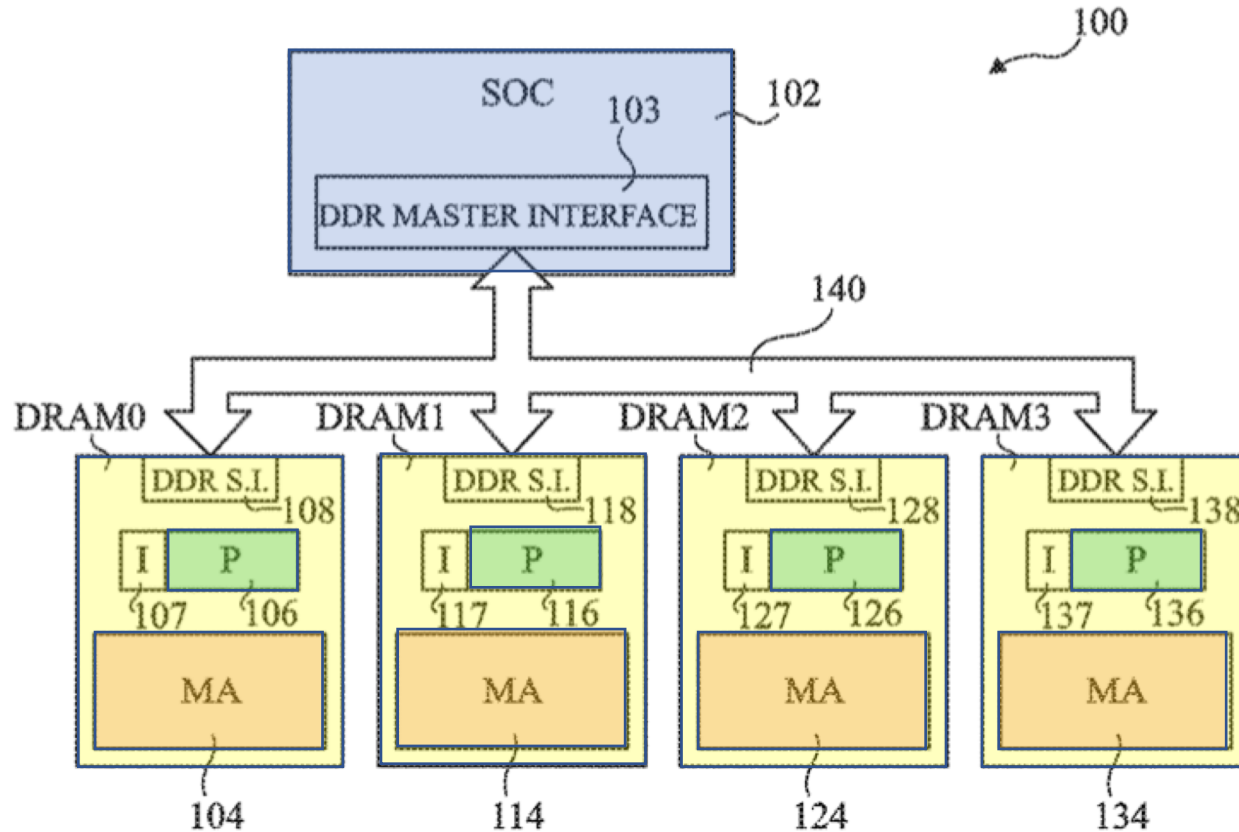
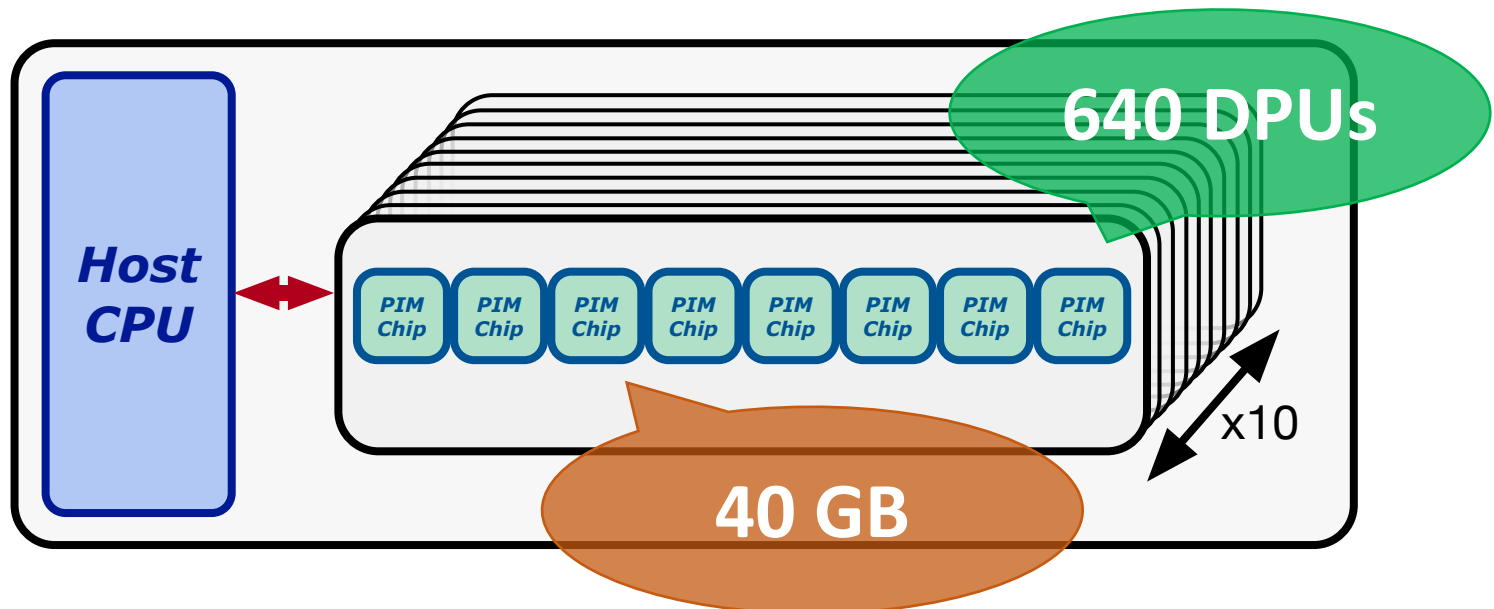


Fig 1

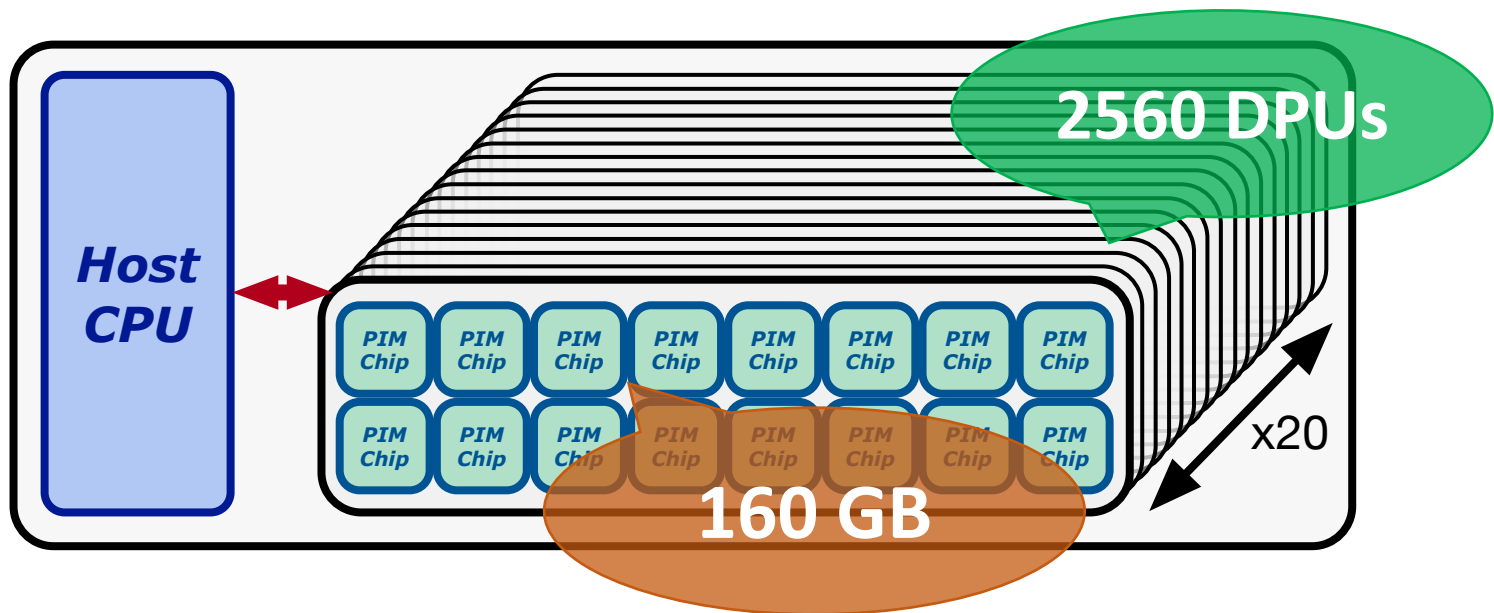
# System Organization (II)

- UPMEM DIMMs coexist with regular DDR4 DIMMs
- Current setup with 10 UPMEM DIMMs (10 ranks) of 8 chips each
  - 8 DRAM Processing Units (DPUs) in each chip, 64 DPUs per rank
  - 8 64MB banks per chip: MRAM (Main RAM) banks
  - x86 socket with 2 memory controllers (3 channels each)
    - 2 conventional DIMMs on the same channel of one controller



# System Organization (III)

- Full-blown configuration
  - 20 DIMMs of 16 chips each
    - With 8 DPUs per chip, 2560 DPUs
    - 160 GB of MRAM



# 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

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

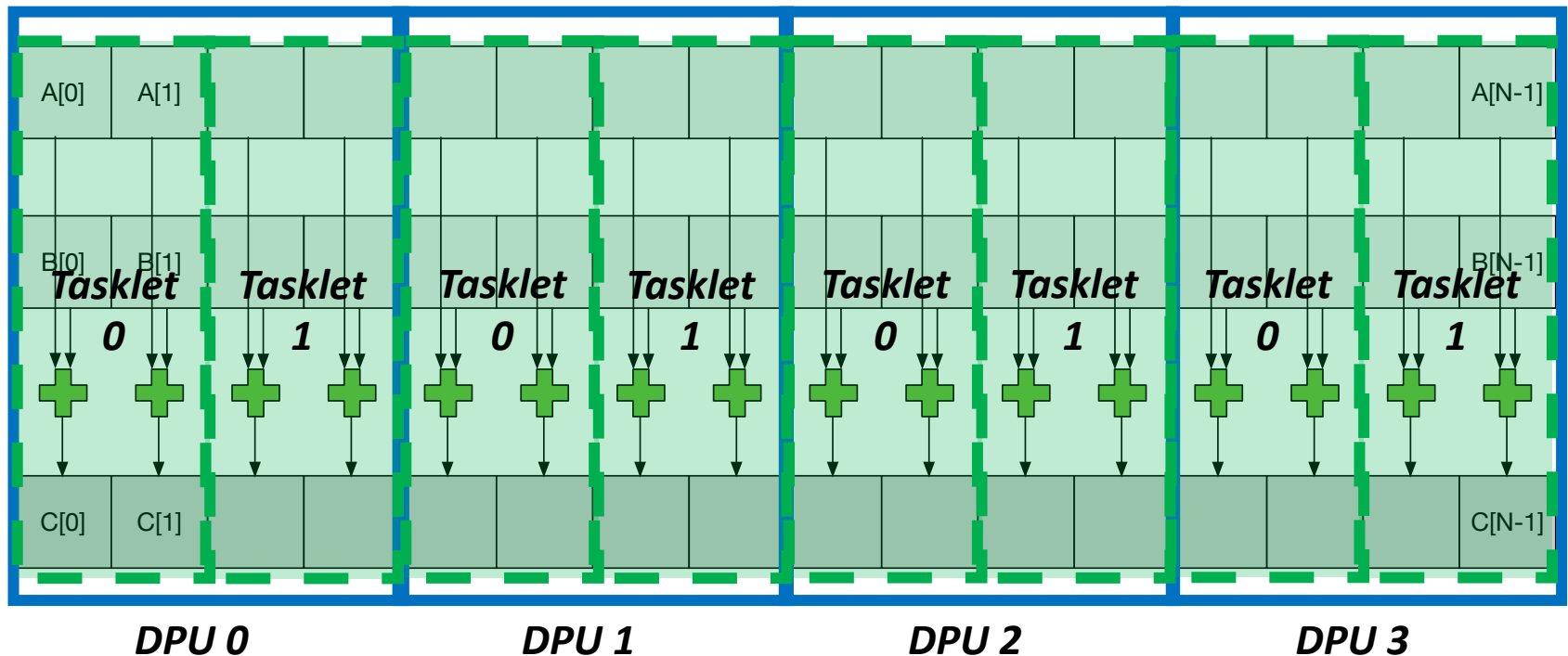
# Outline

---

- Introduction
  - Accelerator Model
  - System Integration
- UPMEM PIM Programming
  - Vector Addition
  - DPU Allocation
  - CPU-DPU Data Transfers
  - DPU Kernel Launch and Execution
- DRAM Processing Unit
  - Arithmetic throughput
- DPU Kernel Execution
  - Vector Addition
  - Tasklet Synchronization
    - Parallel Reduction
- Characterization of the UPMEM PIM Architecture

# Vector Addition

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






# 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. Btw, we deallocate a DPU set with `dpu_free ( )`

# DPU Allocation: Needleman-Wunsch

- 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`

```
// 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?

It is possible:  
More complex workloads with **task-level parallelism**

# CPU-DPU Data Transfers (I)

- 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 * i, input_size_dpu * sizeof(T)))  
    DPU_ASSERT(dpu_copy_from(dpu, DPU_MRAM_HEAP_POINTER_NAME, input_size_dpu * sizeof(T), bufferB + input_size_dpu * i, input_size_dpu * sizeof(T)))  
    i++;  
}
```

	Offset within MRAM	Pointer to main memory	Transfer size
--	--------------------	------------------------	---------------

# CPU-DPU Data Transfers (II)

- Parallel transfers

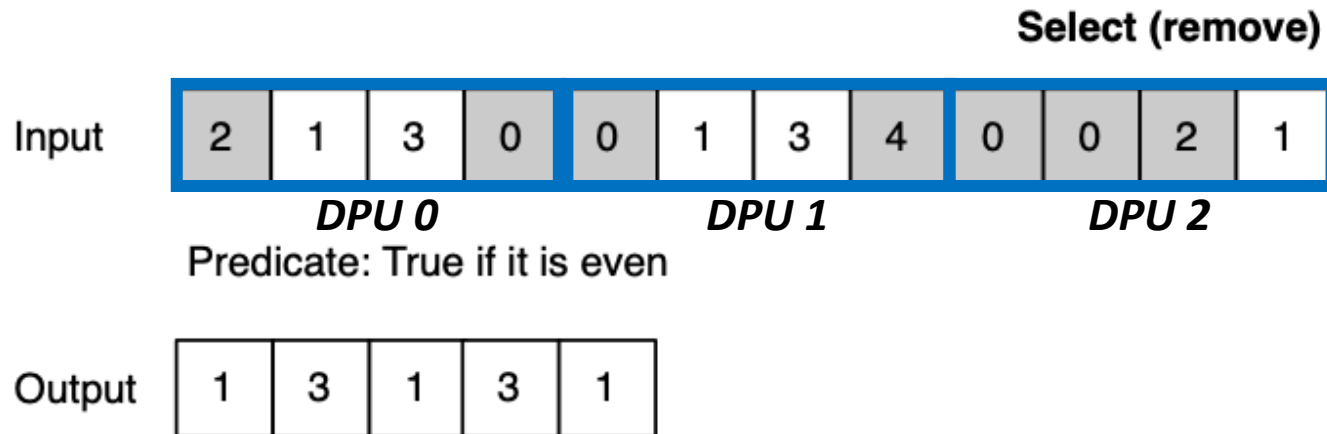
- We push different buffers to/from a DPU set in one transfer
- First, prepare (`dpu_prepare_xfer`);  
then, push (`dpu_push_xfer`)
- Direction:
  - `DPU_XFER_TO_DPU`
  - `DPU_XFER_FROM_DPU`

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

Direction

# CPU-DPU Data Transfers (III)

- An example benchmark we use both parallel and serial transfers
- SELECT



# How Fast are these Data Transfers?

---

- Serial and parallel transfers
  - **Load** (CPU-to-DPU) and **Retrieve** (DPU-to-CPU)

DDR4 bandwidth bounds the maximum transfer bandwidth

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

# “Transposing” Library

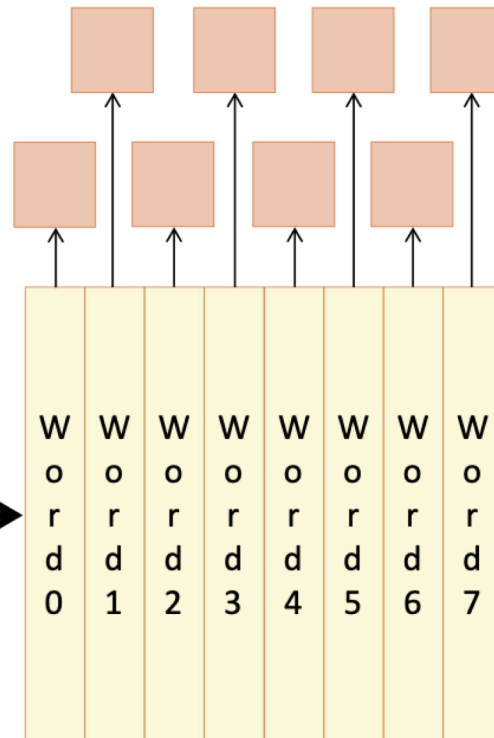
## The library feeds DPUs with correct data

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

Word 0
Word 1
Word 2
Word 3
Word 4
Word 5
Word 6
Word 7

Library



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.

Copyright UPMEM® 2019

Authorized licensed use limited to: ETH BIBLIOTHEK ZURICH. Downloaded on September 04, 2020 at 13:55:41 UTC from IEEE Xplore. Restrictions apply.

HOT CHIPS 31






# 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");  
DPU_ASSERT(dpu_launch(dpu_set, DPU_SYNCHRONOUS));
```



What does the asynchronous execution enable?

Some ideas:

- Overlapping **data transfers and DPU** computation
- Concurrent **heterogeneous computation** on CPU and DPUs
- **Task-level parallelism**: concurrent execution of different kernels on different DPU sets

# How do Pass Parameters to the Kernel?

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

```
// In DPU WRAM
__host dpu_arguments_t DPU_INPUT_ARGUMENTS;
__host dpu_results_t DPU_RESULTS[NR_TASKLETS];

...

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

# 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

# Outline

---

- Introduction
  - Accelerator Model
  - System Integration
- UPMEM PIM Programming
  - Vector Addition
  - DPU Allocation
  - CPU-DPU Data Transfers
  - DPU Kernel Launch and Execution
- **DRAM Processing Unit**
  - Arithmetic throughput
- DPU Kernel Execution
  - Vector Addition
  - Tasklet Synchronization
    - Parallel Reduction
- Characterization of the UPMEM PIM Architecture

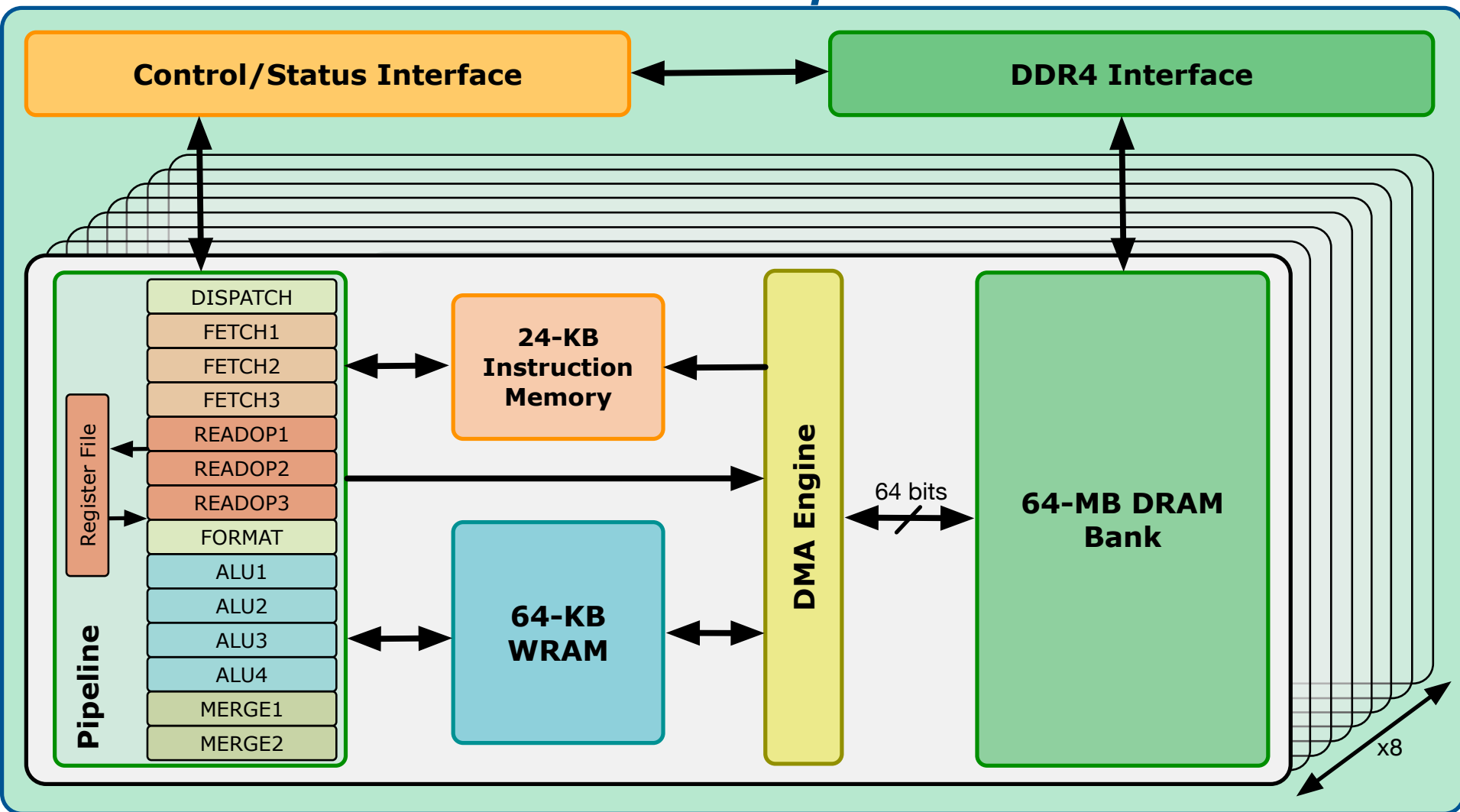
- FIG. 4 schematically illustrates part of the computing system of FIG. 1 in more detail according to an example embodiment

Fig 1

Fig 4

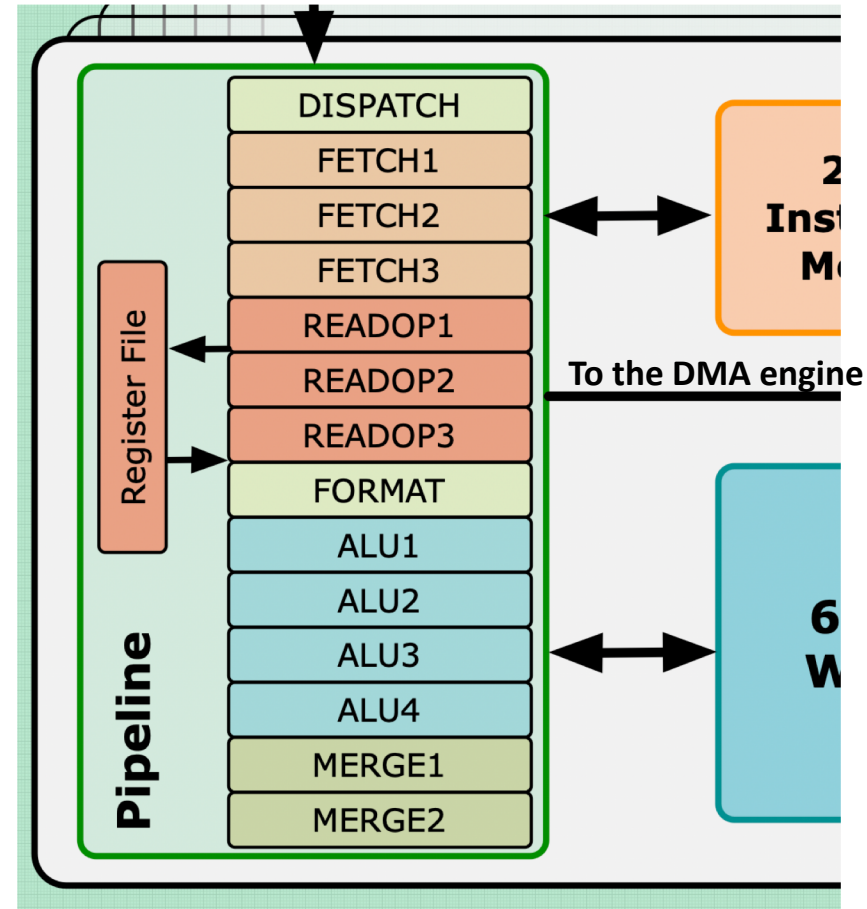
# DRAM Processing Unit (II)

## *PIM Chip*

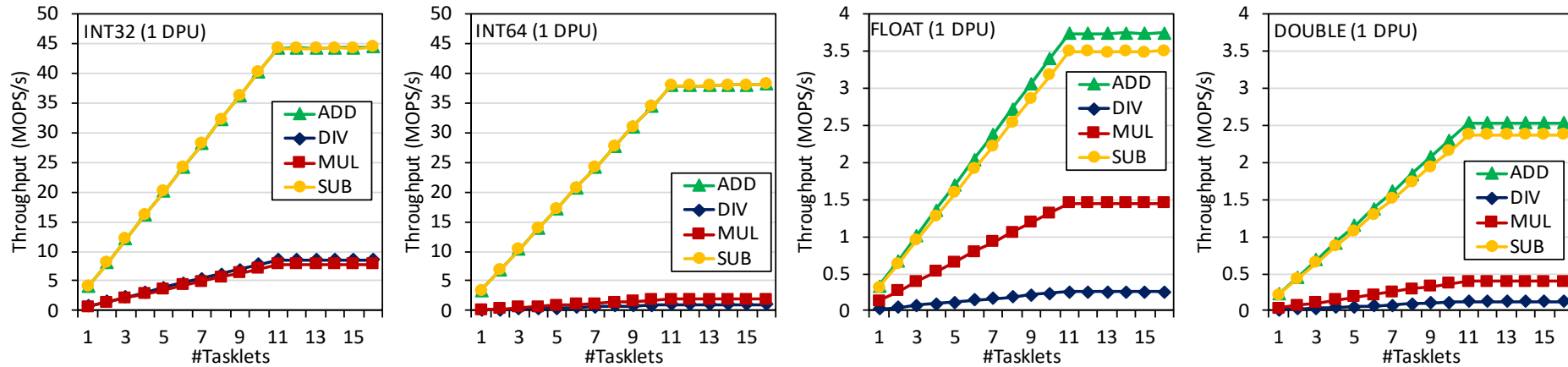


# DPU Pipeline

- In-order pipeline
  - 400 MHz in next generation
  - 267 MHz in current setup
- Multithreading
  - 24 hardware threads
- 14 pipeline stages
  - DISPATCH: Thread selection
  - FETCH
  - READOP
  - FORMAT: Operand formatting
  - ALU: Operation and WRAM
  - MERGE: Result formatting
- 11 tasklets for peak throughput



# Arithmetic Throughput (I)



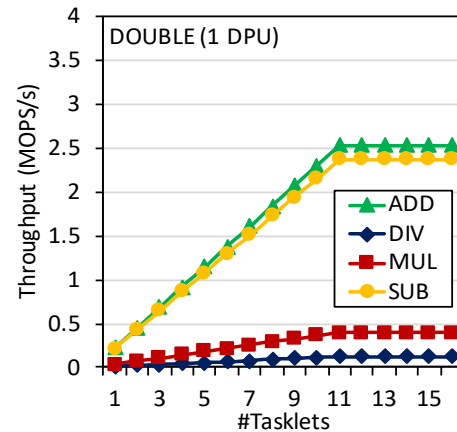
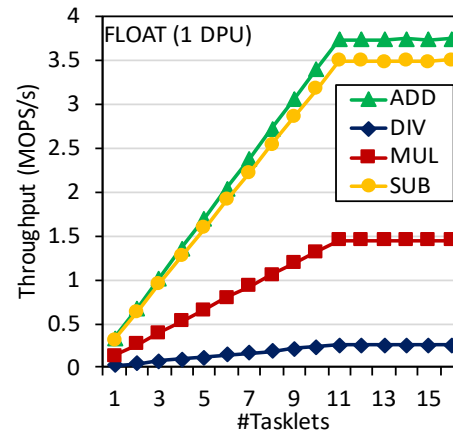
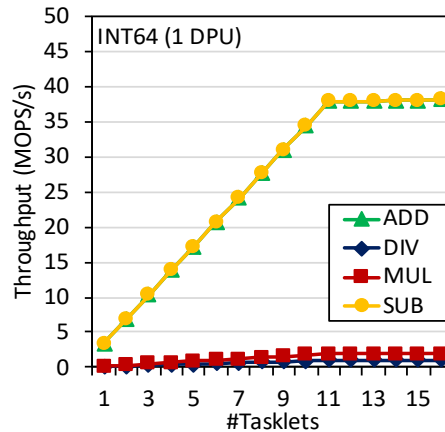
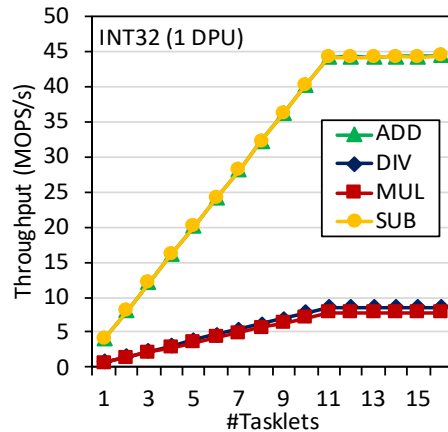
Huge throughput difference between add/sub and mul/div

DPU's 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



# Arithmetic Throughput (II)



Huge throughput difference between  
int32/int64 and float/double

DPU's do not have floating point units.  
Software emulation for floating point computations

More efficient algorithms based on **other formats**?

# Outline

---

- Introduction
  - Accelerator Model
  - System Integration
- UPMEM PIM Programming
  - Vector Addition
  - DPU Allocation
  - CPU-DPU Data Transfers
  - DPU Kernel Launch and Execution
- DRAM Processing Unit
  - Arithmetic throughput
- DPU Kernel Execution
  - Vector Addition
  - Tasklet Synchronization
    - Parallel Reduction
- Characterization of the UPMEM PIM Architecture

# Programming a DPU Kernel (I)

- Vector addition

```
/**
 * @fn task_main
 * @brief main function executed by each tasklet
 * @output vector addition
 */
int main kernel1() {
    unsigned int tasklet_id = me(); Tasklet ID

    uint32_t input_size_dpu = DPU_INPUT_ARGUMENTS.size / sizeof(T); Size of vector tile processed by a DPU

    // Address of the current processing block in MRAM MRAM addresses of arrays A and B
    uint32_t mram_base_addr_A = (uint32_t)(DPU_MRAM_HEAP_POINTER + (tasklet_id << BLOCK_SIZE_LOG2));
    uint32_t mram_base_addr_B = (uint32_t)(DPU_MRAM_HEAP_POINTER + (tasklet_id << BLOCK_SIZE_LOG2) + input_size_dpu * sizeof(T));

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

    for(unsigned int byte_index = 0; byte_index < input_size_dpu * sizeof(T); byte_index += BLOCK_SIZE * NR_TASKLETS){

        // Load cache with current MRAM block
        mram_read((__mram_ptr void const*)(mram_base_addr_A + byte_index), cache_A, BLOCK_SIZE);
        mram_read((__mram_ptr void const*)(mram_base_addr_B + byte_index), cache_B, BLOCK_SIZE); MRAM-WRAM DMA transfers

        // Computer vector addition
        vector_addition(cache_B, cache_A); Vector addition (see next slide)

        // Write cache to current MRAM block
        mram_write(cache_B, (__mram_ptr void*)(mram_base_addr_B + byte_index), BLOCK_SIZE); WRAM-MRAM DMA transfer
    }
}
```

# Programming a DPU Kernel (II)

---

- Vector addition

```
/**
 * @fn vector_addition
 * @brief computes the vector addition of a cached block of 256 bytes
 * @param bufferA the buffer address A
 * @param bufferB the buffer address B - output
 * @return void
 */
void vector_addition(T *bufferB, T *bufferA) {
    for (unsigned int i = 0; i < BLOCK_SIZE / sizeof(T); i++){
        bufferB[i] += bufferA[i];
    }
}
```

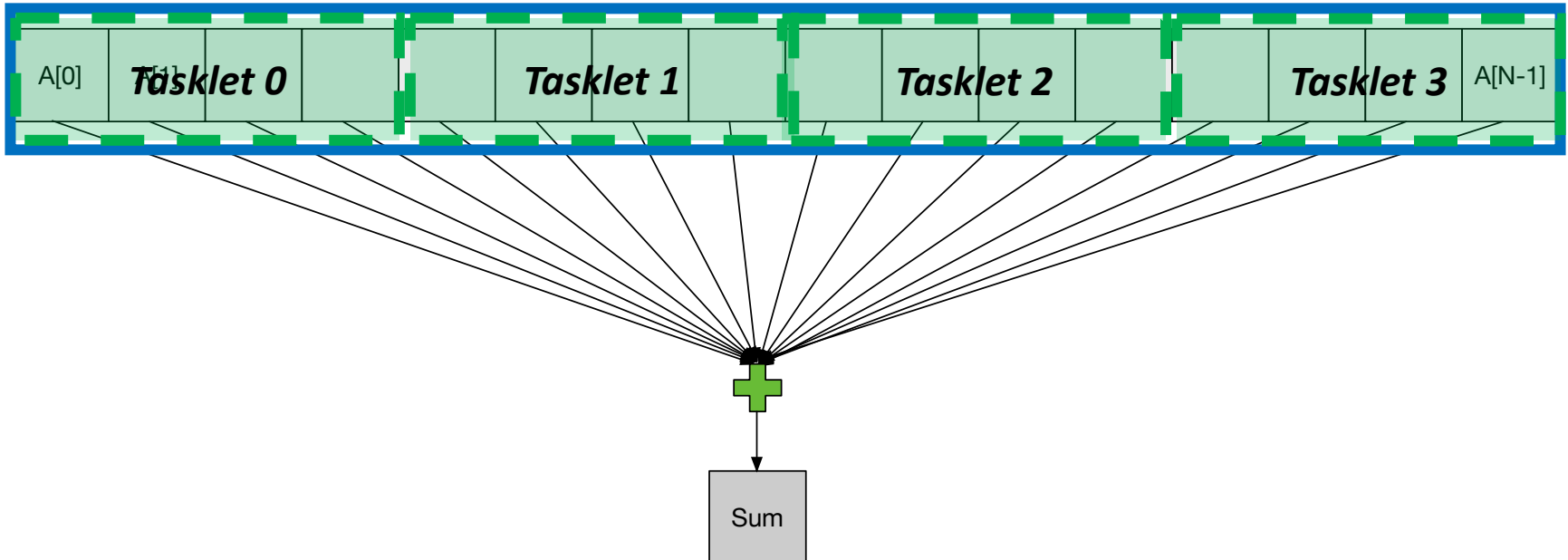
# Programming a DPU Kernel (III)

---

- 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();`

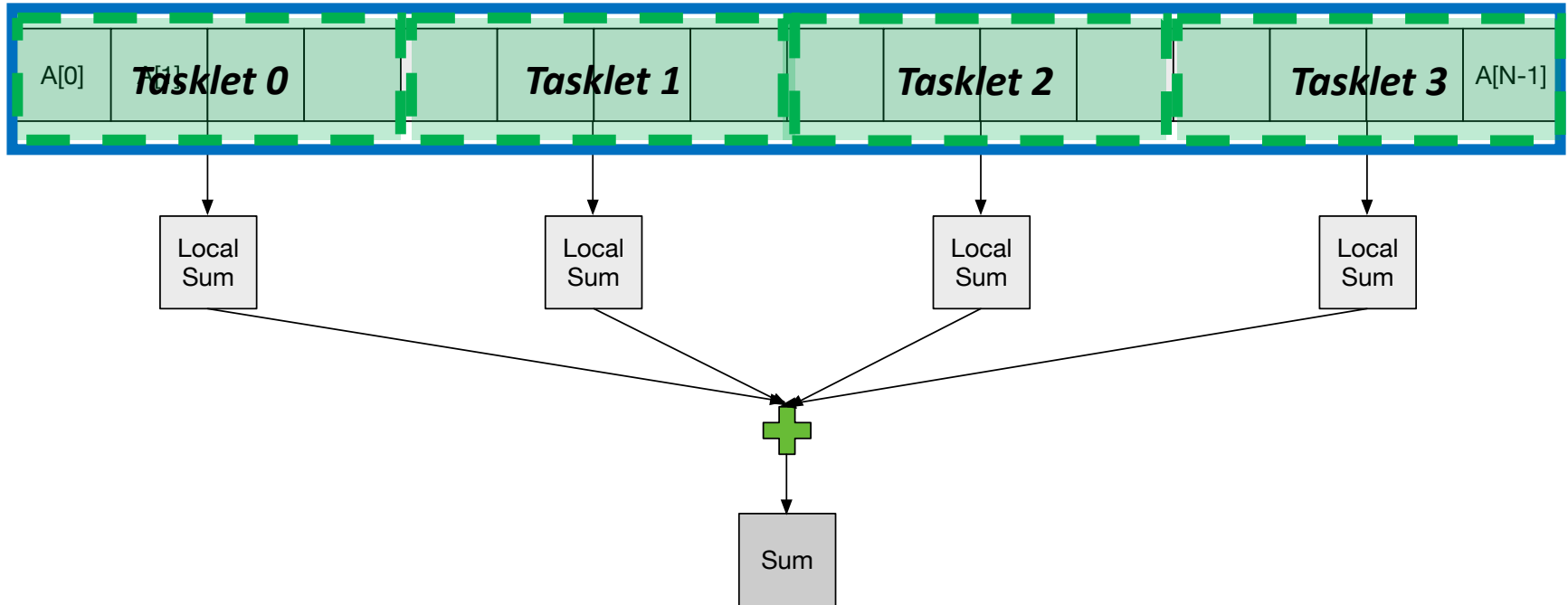
# Parallel Reduction (I)

- 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

```
// Local per-tasklet reduction
for(unsigned int byte_index = 0; byte_index < input_size_dpu * sizeof(T); byte_index += BLOCK_SIZE * NR_TASKLETS){

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

    // Reduction in each tasklet
    l_count += reduction(cache_A); Accumulate in a local sum
}

// Local counts to shared array in WRAM
message[tasklet_id] = l_count; Copy local sum into WRAM
```



# Final Reduction

- A single tasklet can perform the final reduction

```
// Local per-tasklet reduction
for(unsigned int byte_index = 0; byte_index < input_size_dpu * sizeof(T); byte_index += BLOCK_SIZE * NR_TASKLETS){

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

    // Reduction in each tasklet
    l_count += reduction(cache_A); Accumulate in a local sum
}

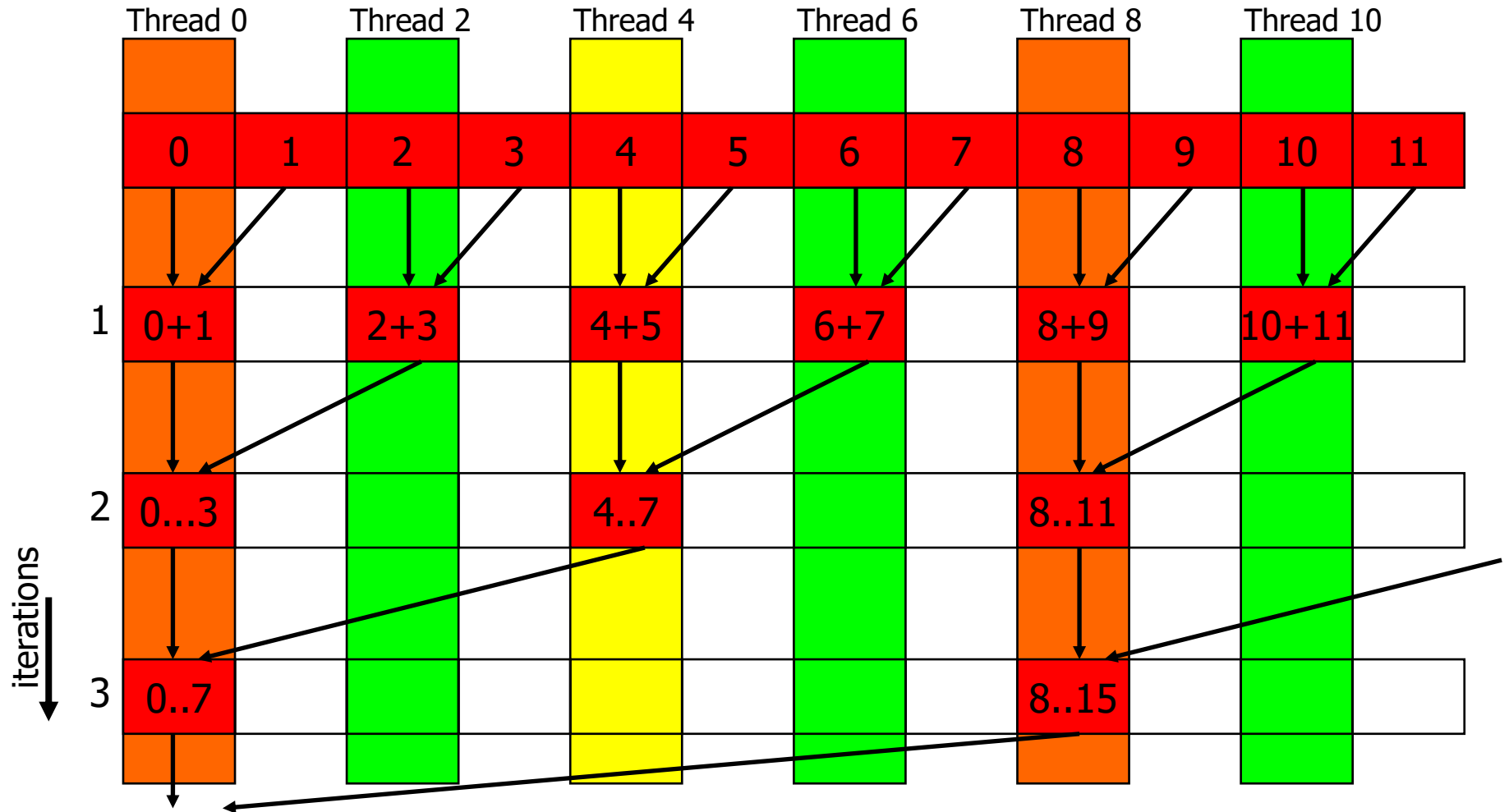
// Local counts to shared array in WRAM
message[tasklet_id] = l_count; Copy local sum into WRAM

// Barrier
barrier_wait(&my_barrier); Barrier synchronization

// Single-thread reduction
if(tasklet_id == 0)
    for (unsigned int each_tasklet = 1; each_tasklet < NR_TASKLETS; each_tasklet++){
        message[0] += message[each_tasklet]; Sequential accumulation
    }

// Total count in this DPU
if(tasklet_id == 0){
    result->t_count = message[tasklet_id];
}
```

# Vector Reduction: Naïve Mapping (I)



Slide credit: Hwu & Kirk

# 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 = NR_TASKLETS/2; offset > 0; offset >>=1){

    if(tasklet_id < offset){

        message[tasklet_id] += message[tasklet_id + offset]; "offset" tasklets working

    }

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

# Outline

---

- Introduction
  - Accelerator Model
  - System Integration
- UPMEM PIM Programming
  - Vector Addition
  - DPU Allocation
  - CPU-DPU Data Transfers
  - DPU Kernel Launch and Execution
- DRAM Processing Unit
  - Arithmetic throughput
- DPU Kernel Execution
  - Vector Addition
  - Tasklet Synchronization
    - Parallel Reduction
- Characterization of the UPMEM PIM Architecture

# 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.

# Resources

---

- UPMEM SDK documentation
  - [https://sdk.upmem.com/master/00\\_ToolchainAtAGlance.html](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>

# Executive Summary

---

- **Processing-in-Memory** is a paradigm that can tackle the **data movement bottleneck**
- Though promising, **there were not real-world devices** that represent a baseline for our research
  - Simulation models for our PIM architecture proposals, where the baseline is typically the host CPU (or GPU)
- UPMEM has designed and fabricated **the first publicly-available real-world PIM architecture**
  - DDR4 chips embedding in-order multithreaded DPUs
- Goals
  - **Introduction** to UPMEM programming model and PIM architecture
  - **Understanding** the UPMEM PIM architecture

# Understanding a Modern Processing-in-Memory Architecture

Juan Gómez Luna, Izzat El Hajj,  
Iván Fernández, Christina Giannoula,  
Geraldo F. de Oliveira, Onur Mutlu