ABUMPIMP 2024: 2nd Minisymposium on Applications and Benefits of a UPMEM Commercial Massively Parallel Processing-In-Memory Platform

Software Frameworks for Productive and Effective Processing-in-Memory Architectures

Geraldo F. Oliveira

(https://geraldofojunior.github.io/)

Onur Mutlu

ETH zürich

SAFARI

Brief Self Introduction

- **Geraldo F. Oliveira**
  - Researcher @ SAFARI Research Group since November 2017
  - Soon, I will defend my PhD thesis, advised by Onur Mutlu
  - https://geraldofojunior.github.io/
  - geraldofojunior@gmail.com (best way to reach me)
  - https://safari.ethz.ch

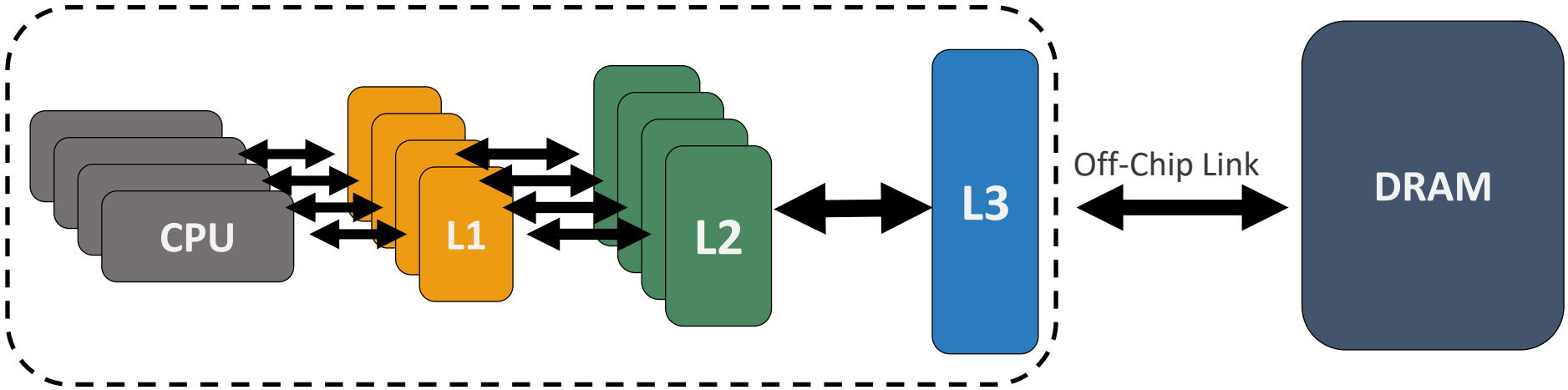
- **Research in:**
  - Computer architecture, computer systems, hardware security
  - Memory and storage systems
  - Hardware security, safety, predictability
  - Fault tolerance
  - Hardware/software cooperation
  - …

Outline

Outline

Data Movement Bottleneck:
The Problem

Data movement is a major bottleneck
in modern computer architectures

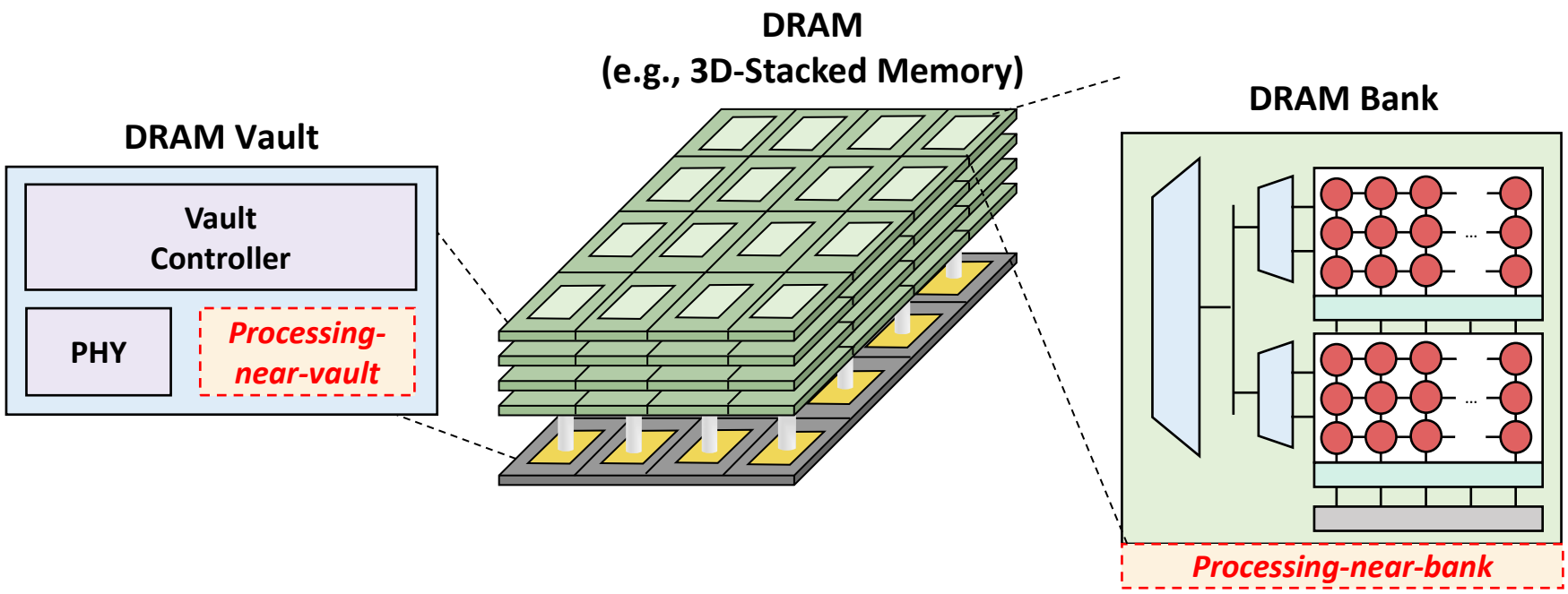


Over 60% of the total system energy is spent on data movement[1]

[1] A. Boroumand et al., "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," ASPLOS, 2018

Processing-in-Memory:
Overview

Processing-in-Memory (PIM) architectures alleviate data movement by **doing computation where the data resides**



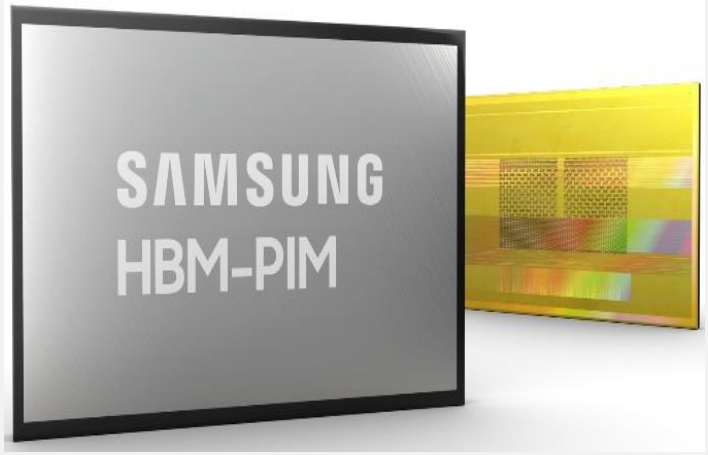
Processing-in-Memory:
Landscape of Real Systems (I)

Processing-in-Memory architectures are
now commercially available

SK hynix AiM

Near-DRAM-bank processing
for neural networks

Alibaba HB-PNM

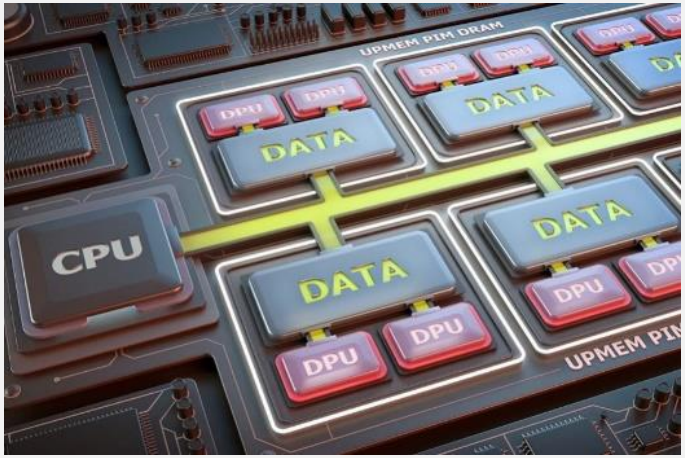


Hybrid bonding with logic
for recommendation systems

Processing-in-Memory:
Landscape of Real Systems (II)

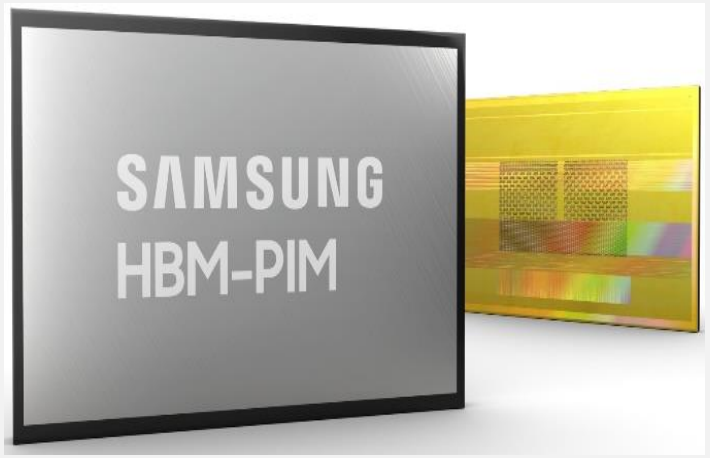
Processing-in-Memory architectures are
now commercially available

UPMEM



Near-DRAM-bank processing
for general-purpose computing

Samsung HBM-PIM

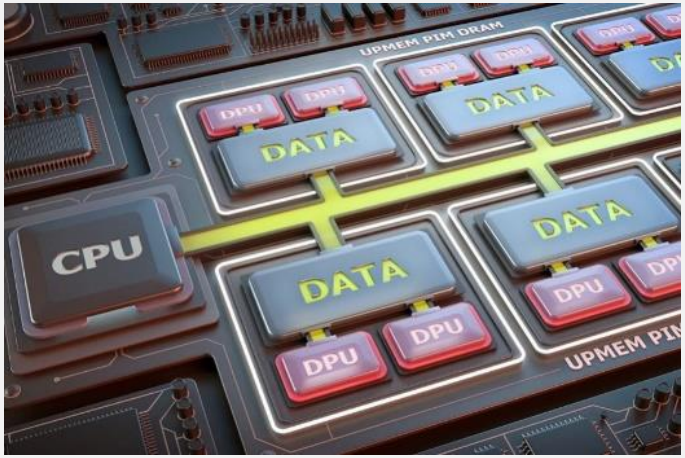


Near-DRAM-bank processing
for neural networks

Processing-in-Memory:
Landscape of Real Systems (II)

Processing-in-Memory architectures are now commercially available

UPMEM



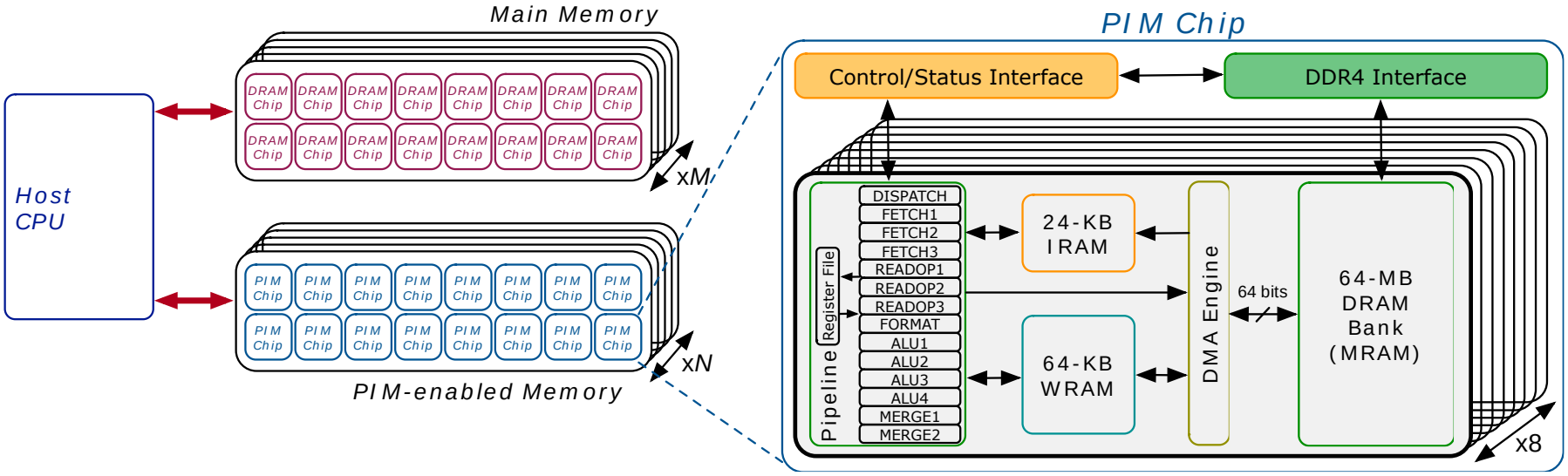
Near-DRAM-bank processing
for general-purpose computing

Samsung HBM-PIM

Near-DRAM-bank processing
for neural networks

Processing-in-Memory:
The UPMEM Architecture

UPMEM: Near-DRAM-bank processing
for general-purpose computing



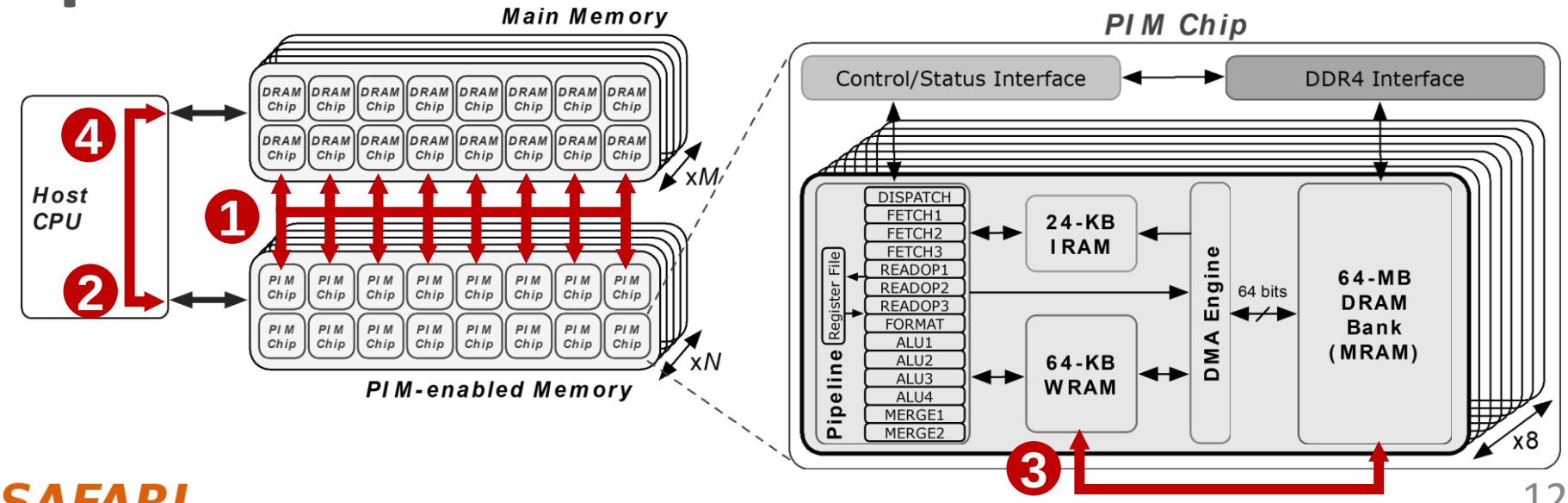
Integration of UPMEM PIM in a system follows an accelerator model

Outline

The Programmability Barrier:
Overview

Programming the UPMEM-based system requires:

1. Splitting input data and computation across PIM chips
2. Transferring input data from main memory to PIM chips
3. Manually handling caching in PIM's scratchpad memory
4. Transferring output data from PIM chips to main memory



The Programmability Barrier:
Vector Addition Example

Programmer's Tasks:

The Programmability Barrier:
Vector Addition Example

Programmer's Tasks:

Align data

```
const unsigned int input_size = 1073741824; // Example value
const unsigned int input_size_8bytes =
                        ((input_size * sizeof(T)) % 8) != 0
                        ? roundup(input_size, 8) : input_size;

const unsigned int input_size_dpu = divceil(input_size, nr_of_dpus);
const unsigned int input_size_dpu_8bytes =
                        ((input_size_dpu * sizeof(T)) % 8) != 0
                        ? roundup(input_size_dpu, 8) : input_size_dpu;
```

The Programmability Barrier:
Vector Addition Example

Programmer's Tasks:

Align data | Collect parameters

```
unsigned int kernel = 0;
dpu_arguments_t input_arguments[NR_DPUS];
for(i=0; i<nr_of_dpus-1; i++) {
  input_arguments[i].size = input_size_dpu_8bytes * sizeof(T);
  input_arguments[i].transfer_size = input_size_dpu_8bytes * sizeof(T);
  input_arguments[i].kernel  = kernel;
}
input_arguments[nr_of_dpus-1].size =
  (input_size_8bytes - input_size_dpu_8bytes * (NR_DPUS-1)) * sizeof(T);
input_arguments[nr_of_dpus-1].transfer_size =
  input_size_dpu_8bytes * sizeof(T);
input_arguments[nr_of_dpus-1].kernel = kernel;
```

The Programmability Barrier:
Vector Addition Example

Programmer's Tasks:

Align data	Collect parameters	Distribute parameters

```
DPU_ASSERT(dpu_push_xfer(dpu_set, DPU_XFER_TO_DPU, "DPU_INPUT_ARGUMENTS",
                         0, sizeof(input_arguments[0]), DPU_XFER_DEFAULT));
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));
```

The Programmability Barrier:
Vector Addition Example

Programmer's Tasks:

Align data	Collect parameters	Distribute parameters	Launch computation

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

The Programmability Barrier:
Vector Addition Example

Programmer's Tasks:

Align data	Collect parameters	Distribute parameters	Launch computation	Collect results

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

The Programmability Barrier:
Vector Addition Example

Programmer's Tasks:

Align data	Collect parameters	Distribute parameters	Launch computation	Collect results	Manage scratchpad

```
barrier_wait(&my_barrier);
uint32_t input_size_dpu_bytes = DPU_INPUT_ARGUMENTS.size;
uint32_t input_size_dpu_bytes_transfer =
                                DPU_INPUT_ARGUMENTS.transfer_size;
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);
T *cache_A = (T *) mem_alloc(BLOCK_SIZE);
T *cache_B = (T *) mem_alloc(BLOCK_SIZE);
```

The Programmability Barrier:
Vector Addition Example

Programmer's Tasks:

Align data	Collect parameters	Distribute parameters	Launch computation	Collect results	Manage scratchpad	Orchestrate computation

```
for (int byte_index = base_tasklet; byte_index < input_size_dpu_bytes;
      byte_index += BLOCK_SIZE * NR_TASKLETS){
uint32_t l_size_bytes = (byte_index + BLOCK_SIZE >=
                    input_size_dpu_bytes)
                    ? (input_size_dpu_bytes - byte_index) : BLOCK_SIZE;
   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);
   vector_addition(cache_B, cache_A, l_size_bytes >> DIV);
   mram_write(cache_B, (__mram_ptr void*)(mram_base_addr_B + byte_index)
              l_size_bytes);
}
```

The Programmability Barrier:
Vector Addition Example

Programmer's Tasks:

Align data	Collect parameters	Distribute parameters	Launch computation	Collect results	Manage scratchpad	Orchestrate computation

Goal:

Just write my kernel

```
static void vector_addition(T *bufferB, T *bufferA, int l_size){
    for (unsigned int i = 0; i < l_size; i++){
        bufferB[i] += bufferA[i];
    }
}
```

The Programmability Barrier:
Summary

Programmer's Tasks:

Goal:

Align data | Collect parameters | Distribute parameters | Launch computation | Collect results | Manage scratchpad | Orchestrate computation | Just write my kernel

Problem

Programming the UPMEM system leads to non-trivial effort → requires knowledge of the underlying hardware and manual fine-grained data movement handling

```
    }
}
```

Our Goal

Goal

To ease programmability for the UPMEM system,
allowing a programmer to write
efficient PIM-friendly code
without the need to
explicitly manage hardware resources

Outline

SimplePIM:

A Software Framework for Productive and Efficient Processing in Memory

- Jinfan Chen, Juan Gómez-Luna, Izzat El Hajj, Yuxin Guo, and Onur Mutlu,
  "SimplePIM: A Software Framework for Productive and Efficient Processing in Memory"
  Proceedings of the 32nd International Conference on Parallel Architectures and Compilation Techniques (***PACT***),
  Vienna, Austria, October 2023.
  [Slides (pptx) (pdf)]
  [SimplePIM Source Code]

SimplePIM: A Software Framework for Productive and Efficient Processing-in-Memory

Jinfan Chen[1] Juan Gómez-Luna[1] Izzat El Hajj[2] Yuxin Guo[1] Onur Mutlu[1]

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

SimplePIM Programming Framework: Overview

SimplePIM provides standard abstractions to build and deploy applications on PIM systems

1 **Management interface**

→ Metadata for PIM-resident arrays

2 **Communication interface**

→ Abstractions for host-PIM and PIM-PIM communication

3 **Processing interface**

→ Iterators (`map`, `reduce`, `zip`) to implement workloads

SimplePIM Programming Framework:
Management Interface

- **Metadata for PIM-resident arrays**
  - `array_meta_data_t` describes a PIM-resident array
  - `simple_pim_management_t` for managing PIM-resident arrays

- **`lookup`**: Retrieves all relevant information of an array

```
array_meta_data_t* simple_pim_array_lookup(const char* id,
simple_pim_management_t* management);
```

- **`register`**: Registers the metadata of an array

```
void simple_pim_array_register(array_meta_data_t* meta_data,
simple_pim_management_t* management);
```

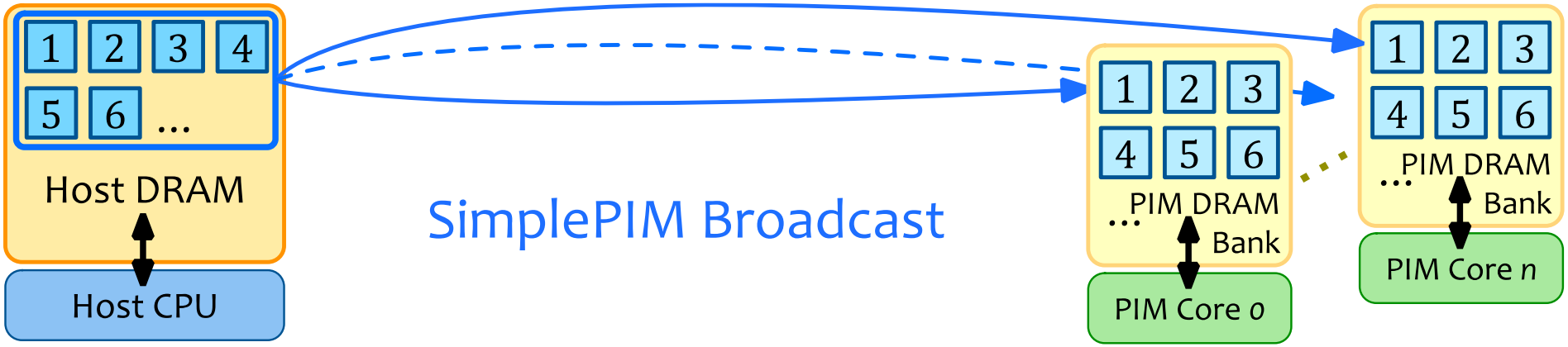
- **`free`**: Removes the metadata of an array

```
void simple_pim_array_free(const char* id, simple_pim_management_t* management);
```

SimplePIM Programming Framework:
Communication Interface (I)

- **SimplePIM Host-to-CPU Broadcast**
  - Transfers a host array to all PIM cores in the system

```
void simple_pim_array_broadcast(char* const id, void* arr, uint64_t len,
uint32_t type_size, simple_pim_management_t* management);
```



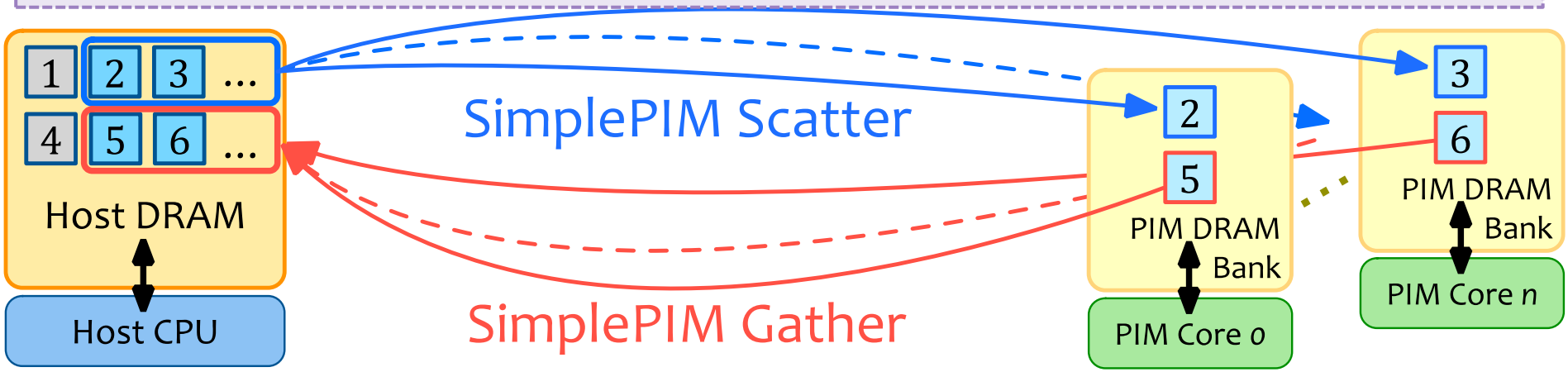
SimplePIM Programming Framework:
Communication Interface (II)

- **Host-to-PIM SimplePIM Scatter**
  - Distributes an array to PIM DRAM banks

```
void simple_pim_array_scatter(char* const id, void* arr, uint64_t len,
uint32_t type_size, simple_pim_management_t* management);
```

- **Host-to-PIM SimplePIM Gather**
  - Collects portions of an array from PIM DRAM banks

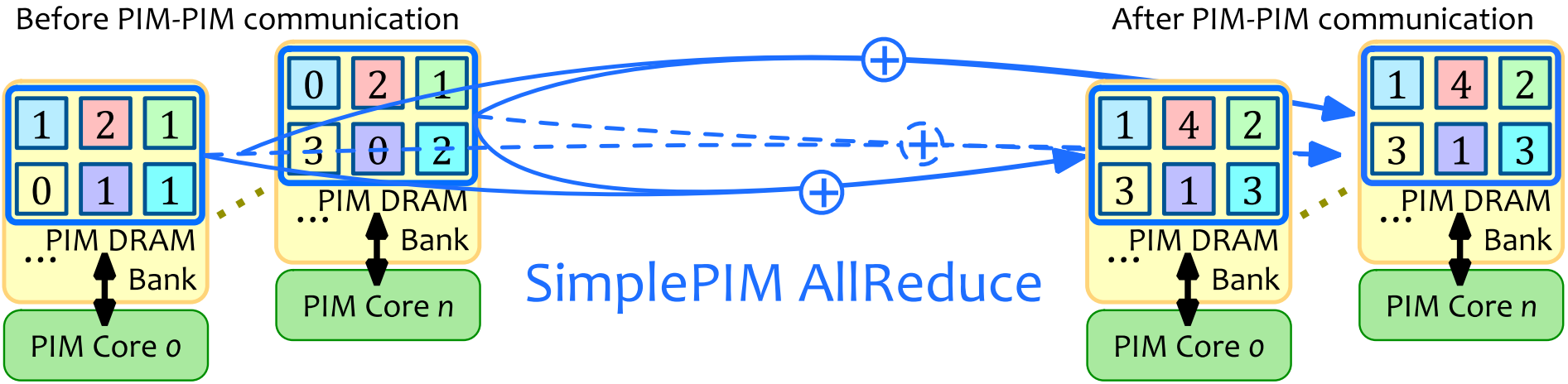
```
void* simple_pim_array_gather(char* const id, simple_pim_management_t*
management);
```



SimplePIM Programming Framework:
Communication Interface (III)

- **PIM-to-PIM Communication: AllReduce**
  - Used for algorithm synchronization
  - The programmer specifies an accumulative function

```
void simple_pim_array_allreduce(char* const id, handle_t* handle,
simple_pim_management_t* management);
```



SimplePIM Programming Framework:
Communication Interface (IV)

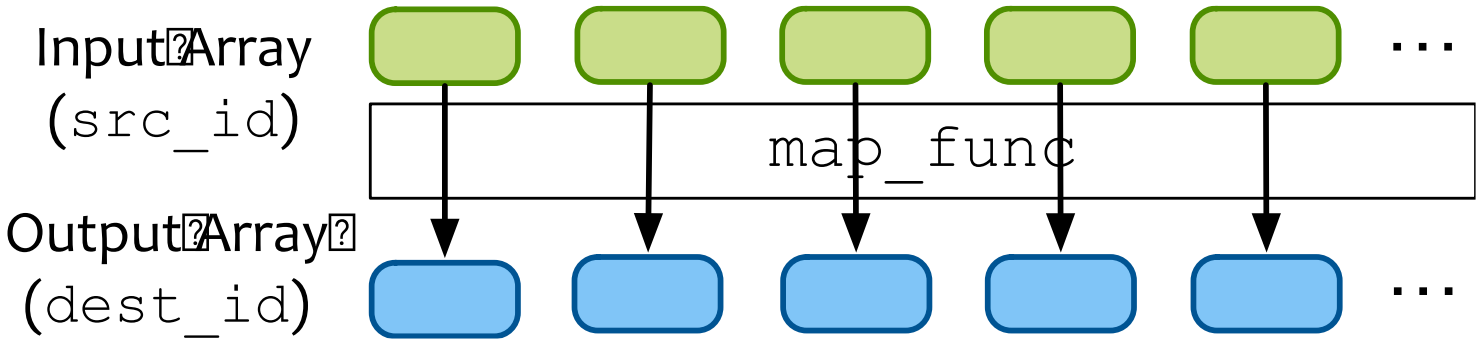
- **PIM-to-PIM Communication: AllGather**
  - Combines array pieces and distributes the complete array to all PIM cores

```
void simple_pim_array_allgather(char* const id, char* new_id,
simple_pim_management_t* management);
```

SimplePIM Programming Framework: Processing Interface (I)

- **Array Map**
  - Applies `map_func` to every element of the data array

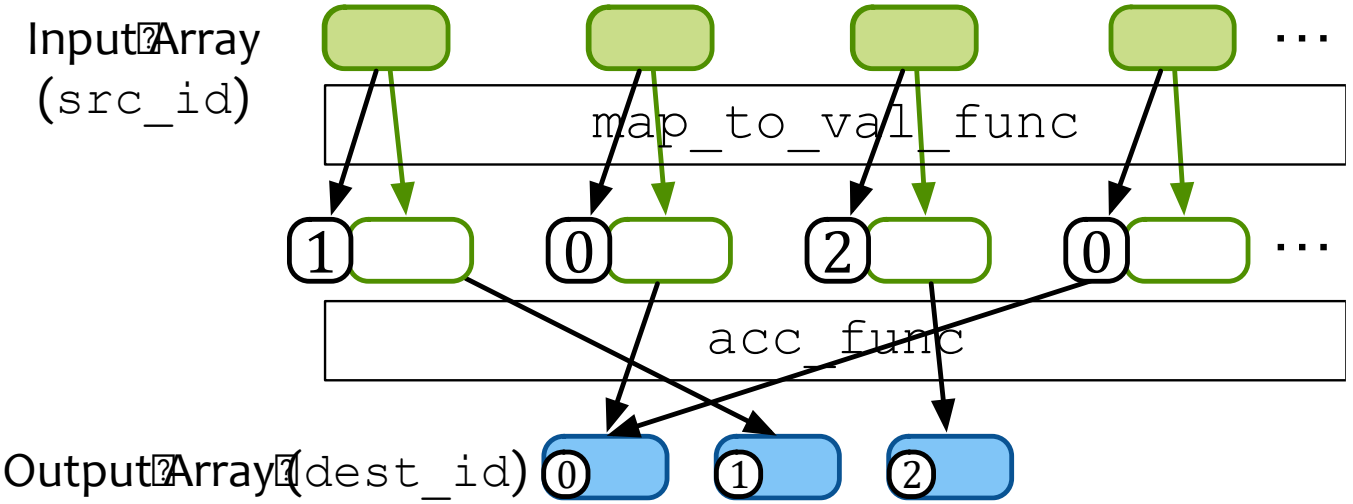
```
void simple_pim_array_map(const char* src_id, const char* dest_id,
uint32_t output_type, handle_t* handle, simple_pim_management_t* management);
```



SimplePIM Programming Framework: Processing Interface (II)

- **Array Reduction**
  - The `map_to_val_func` function transforms an input element to an output value and an output index
  - The `acc_func` function accumulates the output values onto the output array

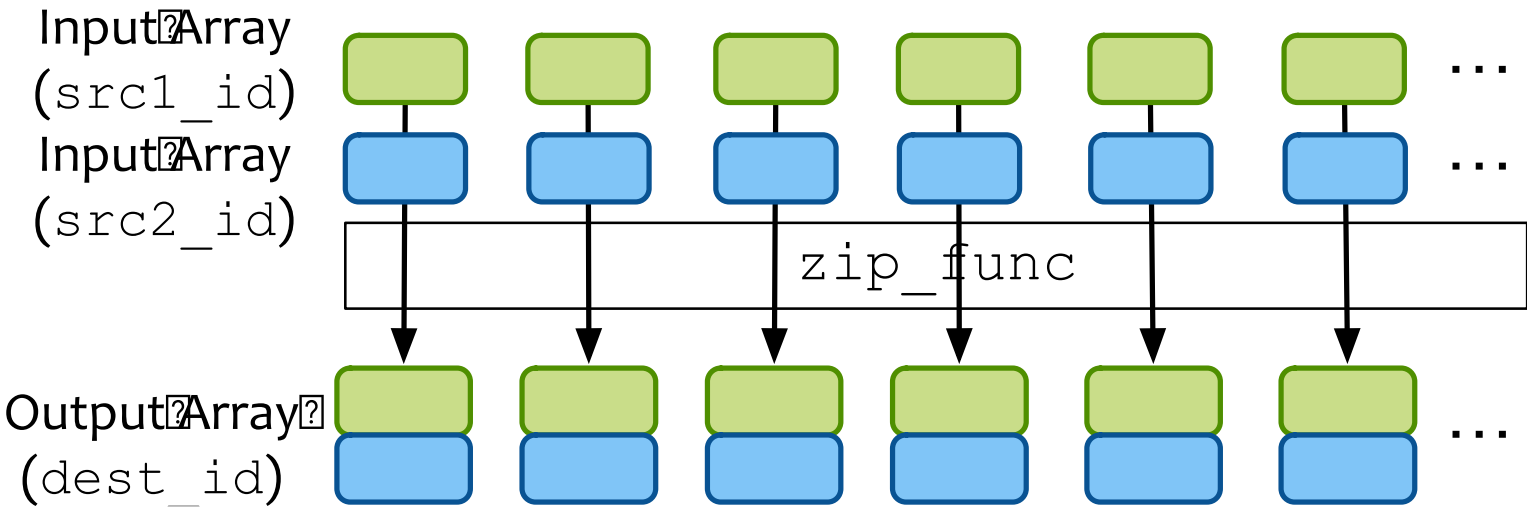
```
void simple_pim_array_red(const char* src_id, const char* dest_id,
uint32_t output_type, uint32_t output_len, handle_t* handle,
simple_pim_management_t* management);
```



SimplePIM Programming Framework: Processing Interface (III)

- **Array Zip**
  - Takes two input arrays and combines their elements into an output array

```
void simple_pim_array_zip(const char* src1_id, const char* src2_id,
const char* dest_id, simple_pim_management_t* management);
```



SimplePIM Programming Framework:
General Code Optimizations

- Strength reduction
- Loop unrolling
- Avoiding boundary checks
- Function inlining
- Adjustment of data transfer sizes

Outline

Evaluation Results:
Evaluation Methodology

- **Evaluated system**
  - UPMEM PIM system with 2,432 PIM cores with 159 GB of PIM DRAM
- **Real-world Benchmarks**
  - Vector addition
  - Reduction
  - Histogram
  - K-Means
  - Linear regression
  - Logistic regression
- Comparison to hand-optimized codes in terms of programming productivity and performance

Evaluation Results:
Productive Improvement (I)

- Example: Hand-optimized histogram with UPMEM SDK

```
... // Initialize global variables and functions for histogram
int main_kernel() {
  if (tasklet_id == 0)
    mem_reset(); // Reset the heap
  ... // Initialize variables and the histogram
  T *input_buff_A = (T*)mem_alloc(2048); // Allocate buffer in scratchpad memory

  for (unsigned int byte_index = base_tasklet; byte_index < input_size; byte_index += stride) {
    // Boundary checking
    uint32_t l_size_bytes = (byte_index + 2048 >= input_size) ? (input_size - byte_index) : 2048;
    // Load scratchpad with a DRAM block
    mram_read((const __mram_ptr void*)(mram_base_addr_A + byte_index), input_buff_A, l_size_bytes);
    // Histogram calculation
    histogram(hist, bins, input_buff_A, l_size_bytes/sizeof(uint32_t));
  }
  ...
  barrier_wait(&my_barrier); // Barrier to synchronize PIM threads
  ... // Merging histograms from different tasklets into one histo_dpu
  // Write result from scratchpad to DRAM
  if (tasklet_id == 0)
    if (bins * sizeof(uint32_t) <= 2048)
      mram_write(histo_dpu, (__mram_ptr void*)mram_base_addr_histo, bins * sizeof(uint32_t));
    else
      for (unsigned int offset = 0; offset < ((bins * sizeof(uint32_t)) >> 11); offset++) {
        mram_write(histo_dpu + (offset << 9), (__mram_ptr void*)(mram_base_addr_histo +
                  (offset << 11)), 2048);
      }
  return 0;
}
```

Evaluation Results:
Productive Improvement (II)

- Example: SimplePIM histogram

```
// Programmer-defined functions in the file "histo_filepath"
void init_func (uint32_t size, void* ptr) {
  char* casted_value_ptr = (char*) ptr;
  for (int i = 0; i < size; i++)
    casted_value_ptr[i] = 0;
}

void acc_func (void* dest, void* src) {
  *(uint32_t*)dest += *(uint32_t*)src;
}

void map_to_val_func (void* input, void* output, uint32_t* key) {
  uint32_t d = *((uint32_t*)input);
  *(uint32_t*)output = 1;
  *key = d * bins >> 12;
}

// Host side handle creation and iterator call
handle_t* handle = simple_pim_create_handle("histo_filepath", REDUCE, NULL, 0);

// Transfer (scatter) data to PIM, register as "t1"
simple_pim_array_scatter("t1", src, bins, sizeof(T), management);

// Run histogram on "t1" and produce "t2"
simple_pim_array_red("t1", "t2", sizeof(T), bins, handle, management);
```

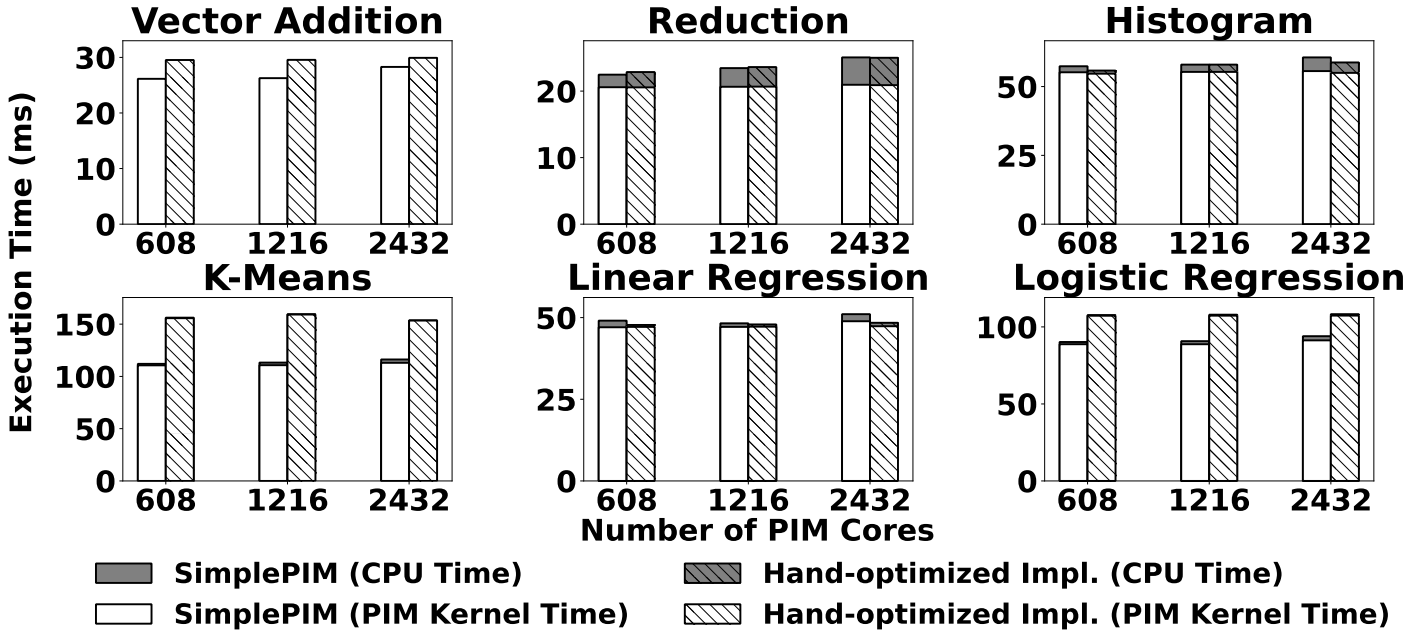
Evaluation Results:
Productive Improvement (III)

- Lines of code (LoC) reduction

	SimplePIM	Hand-optimized	LoC Reduction
Reduction	14	83	5.93×
Vector Addition	14	82	5.86×
Histogram	21	114	5.43×
Linear Regression	48	157	3.27×
Logistic Regression	59	176	2.98×
K-Means	68	206	3.03×

SimplePIM reduces the number of lines of effective code by a factor of 2.98× to 5.93×

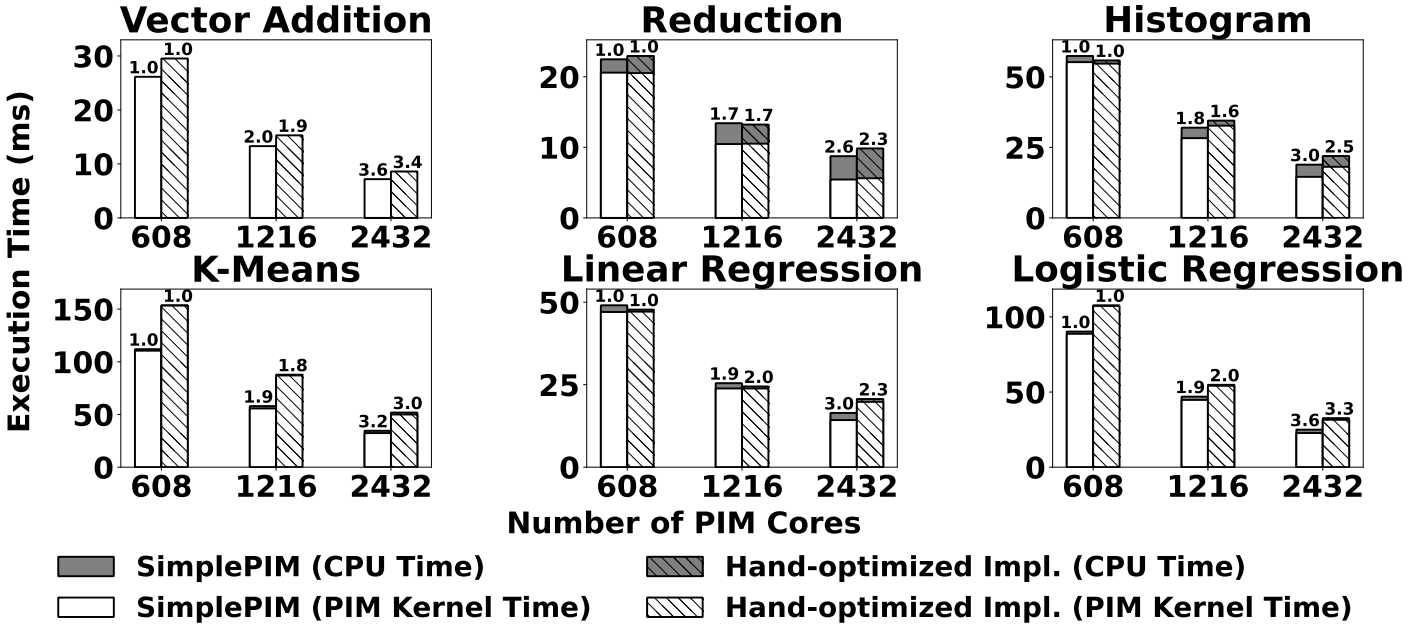
Evaluation Results:
Weak Scaling Analysis



SimplePIM achieves comparable performance for `reduction`, `histogram`, and `linear regression`

SimplePIM outperforms hand-optimized implementations for `vector addition`, `logistic regression`, and `k-means` by 10%-37%

Evaluation Results:
Strong Scaling Analysis

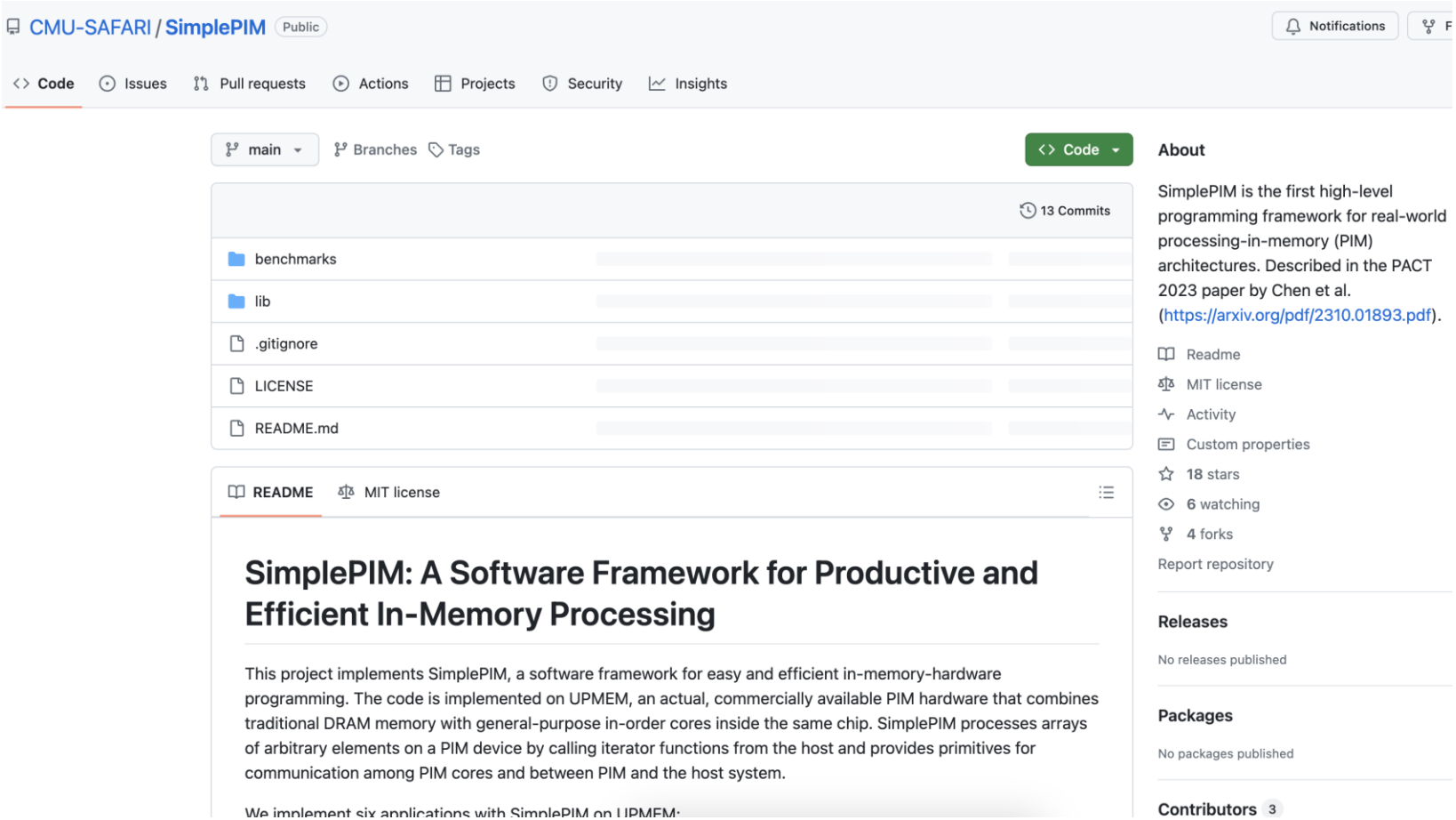


SimplePIM scales better than hand-optimized implementations for `reduction`, `histogram`, and `linear regression`

SimplePIM outperforms hand-optimized implementations for `vector addition`, `logistic regression`, and `k-means` by 15%-43%

Source Code

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



SimplePIM:

A Software Framework for Productive and Efficient Processing in Memory

- Jinfan Chen, Juan Gómez-Luna, Izzat El Hajj, Yuxin Guo, and Onur Mutlu,
  "SimplePIM: A Software Framework for Productive and Efficient Processing in Memory"
  Proceedings of the 32nd International Conference on Parallel Architectures and Compilation Techniques ***(PACT)***,
  Vienna, Austria, October 2023.
  [Slides (pptx) (pdf)]
  [SimplePIM Source Code]

Outline

DaPPA:

A Data-Parallel Framework for Processing-in-Memory Architectures

- Geraldo F. Oliveira, Alain Kohli, David Novo, Juan Gómez-Luna, Onur Mutlu
  "DaPPA: A Data-Parallel Framework for Processing-in-Memory Architectures,"
  arXiv:2310.10168 [cs.AR]
  2nd Place ACM Student Research Competition at *the 32nd International Conference on Parallel Architectures and Compilation Techniques* (***PACT***), Vienna, Austria, October 2023.

DaPPA: A Data-Parallel Framework for Processing-in-Memory Architectures

Geraldo F. Oliveira⋆ Alain Kohli⋆ David Novo‡ Juan Gómez-Luna⋆ Onur Mutlu⋆

⋆*ETH Zürich* ‡*LIRMM, Univ. Montpellier, CNRS*

1. Motivation & Problem

The increasing prevalence and growing size of data in modern applications have led to high costs for computation in traditional *processor-centric computing* systems. To mitigate these costs, the *processing-in-memory* (PIM) [1–6] paradigm moves computation closer to where the data resides, reducing the need to move data between memory and the processor. Even though the concept of PIM has been first proposed in the 1960s [7, 8], real-world PIM systems have only recently

face [15, 16] that abstracts the hardware components of the UPMEM system. Using this key idea, DaPPA transforms a data-parallel pattern-based application code into the appropriate UPMEM-target code, including the required APIs for data management and code partition, which can then be compiled into a UPMEM-based binary *transparently* from the programmer. While generating UPMEM-target code, DaPPA implements several code optimizations to improve end-to-end performance.

DaPPA:
Key Idea & Overview

Key Idea

Leverage an intuitive data-parallel pattern-based interface for PIM programming

DaPPA , a Data-Parallel PIM Architecture that automatically distributes input and gathers output data, handles memory management, and parallelizes work across PIM cores

DaPPA is composed of three main components:

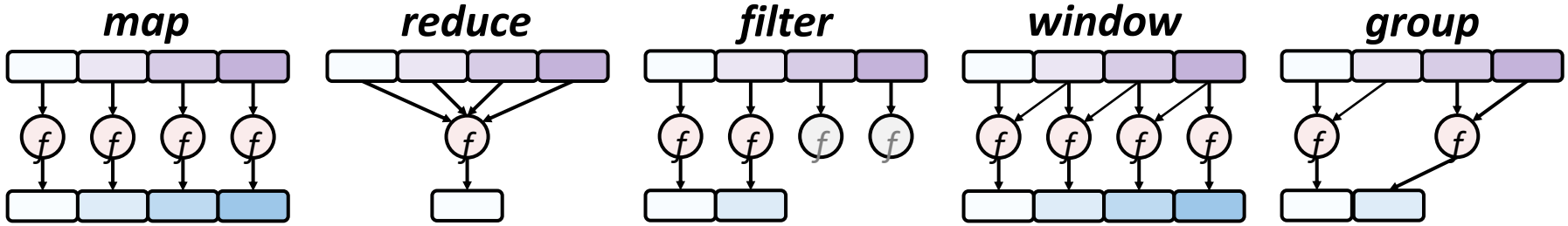
1. **Data-Parallel Pattern APIs**
2. **Dataflow Programming Interface**
3. **Dynamic Template-Based Compilation**

Outline

DaPPA:
Data-Parallel Pattern APIs

Pre-defined functions that implement high-level
data-parallel pattern primitives

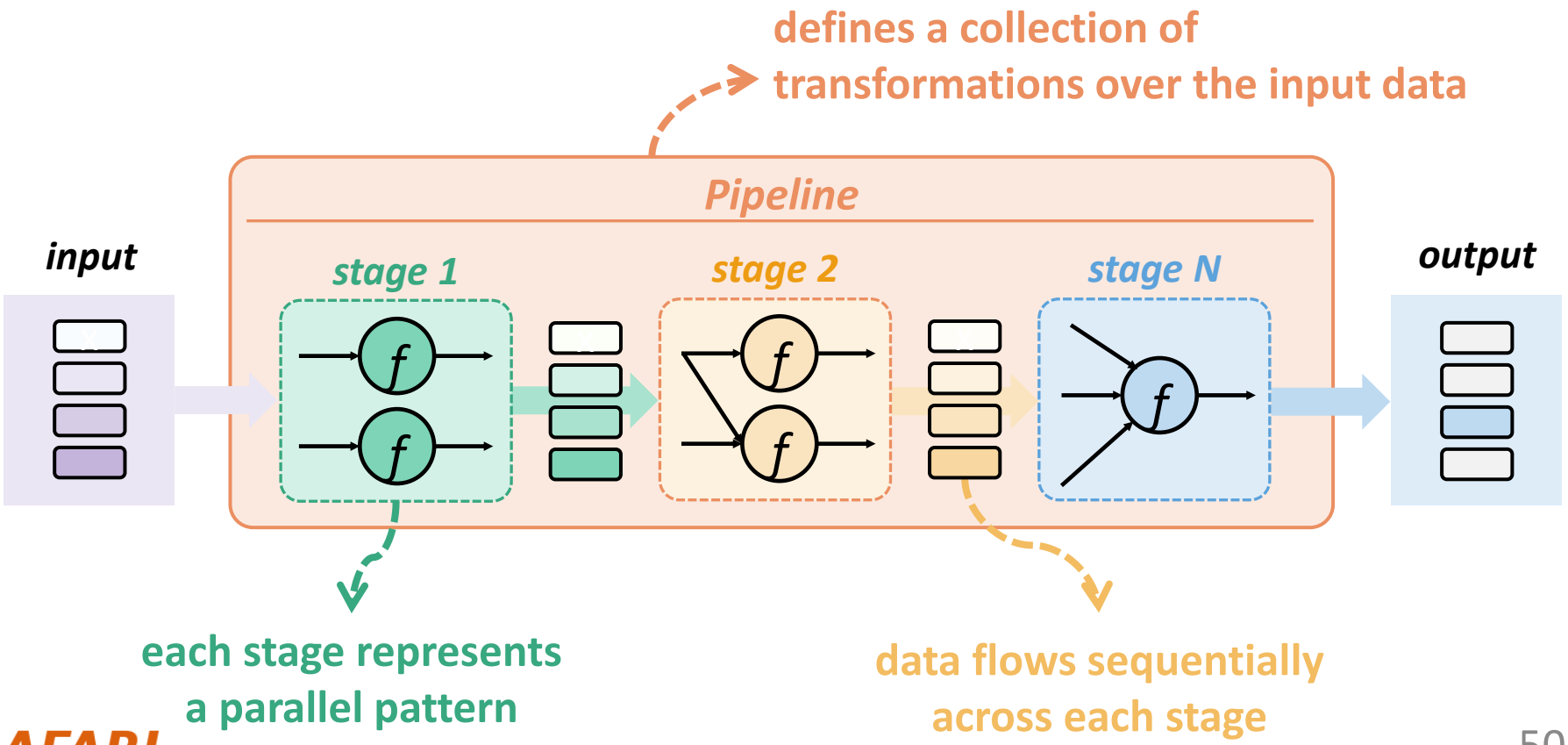
- Skeleton and pattern-based parallel programming are a **common abstraction for parallel architectures**
  - M. Cole, "*Bringing Skeletons Out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming*," Parallel Computing, 2004
- DaPPA supports **five primary** data-parallel patterns



The user can combine all five data-parallel primitives to describe complex data transformations

DaPPA:
Dataflow Programming Interface

DaPPA exposes to the user a
dataflow-based programming interface



DaPPA:
Dynamic Template-Based Compilation

DaPPA uses a dynamic template-based compilation to generate PIM code **in two main steps**

1 **Templating**: DaPPA creates a base UPMEM code based on a ***basic skeleton*** of a UPMEM application

- We use the **Inja C++ templating engine**

2 **Optimizations**: DaPPA uses a series of transformations to

- ***extract*** data required by the UPMEM code template
- calculate the ***memory offsets*** for MRAMs and WRAMs
- ***divide computation*** between CPU and PIM cores

DaPPA compiles and executes each stage in a Pipeline per time → allows for runtime optimizations

DaPPA:
Putting All Together

Example of DaPPA's implementation of a
vector dot product operation

reduce

$C = A_0B_0 +$
$A_1B_1 +$
$A_2B_2 +$
A_3B_3

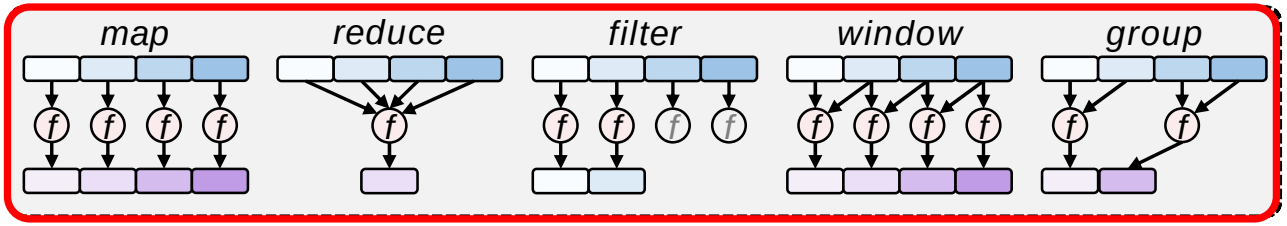
map

target computation

DaPPA:
Putting All Together

Example of DaPPA's implementation of a
vector dot product operation

❶ *data-parallel pattern APIs*



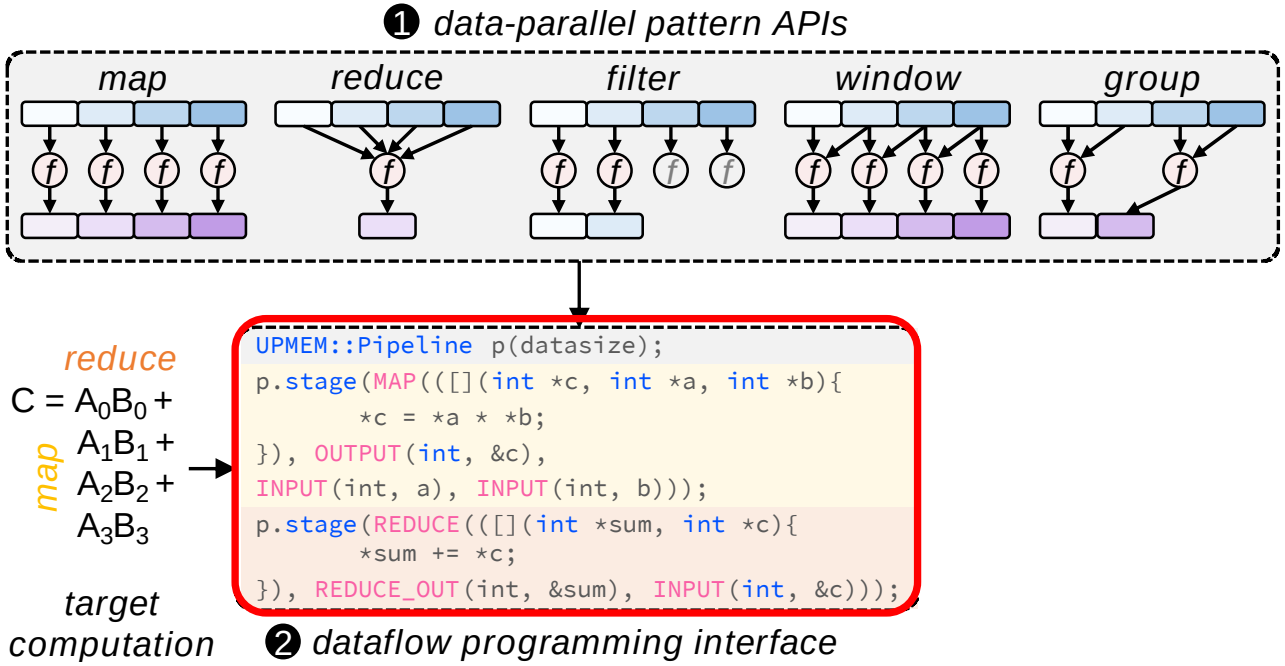
reduce
map

$C = A_0B_0 + A_1B_1 + A_2B_2 + A_3B_3$

target computation

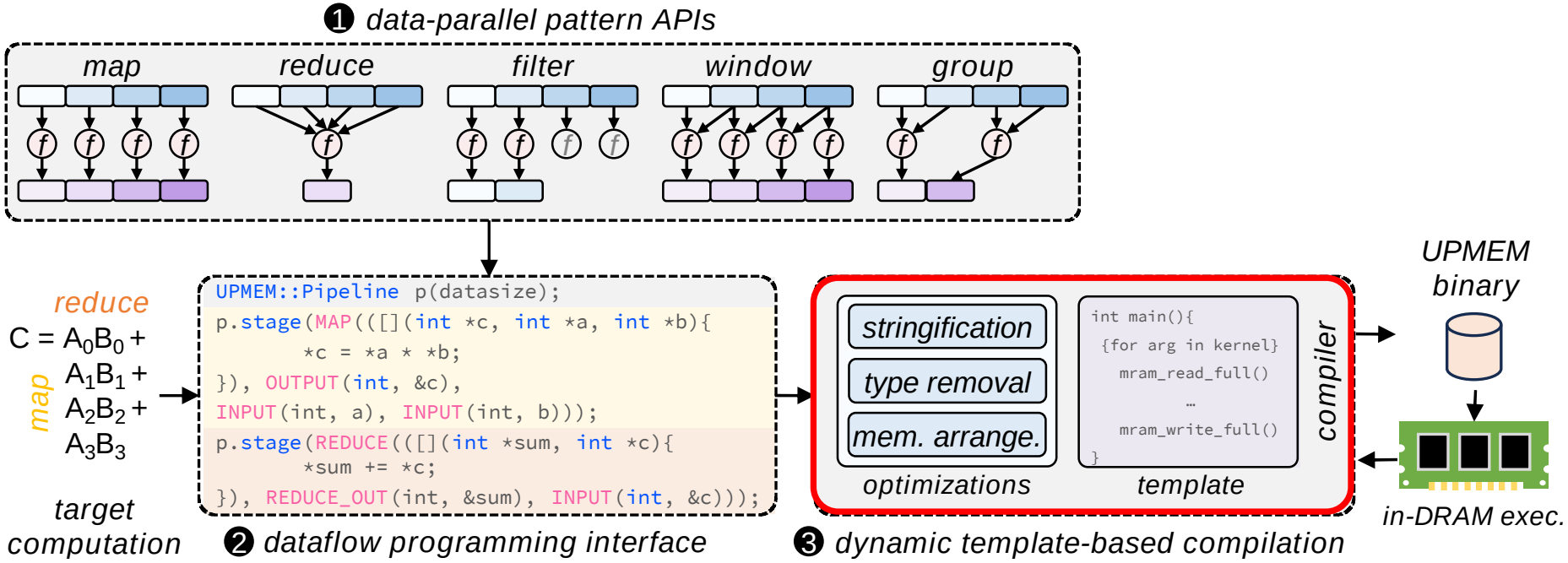
DaPPA:
Putting All Together

Example of DaPPA's implementation of a
vector dot product operation



DaPPA:
Putting All Together

Example of DaPPA's implementation of a
vector dot product operation

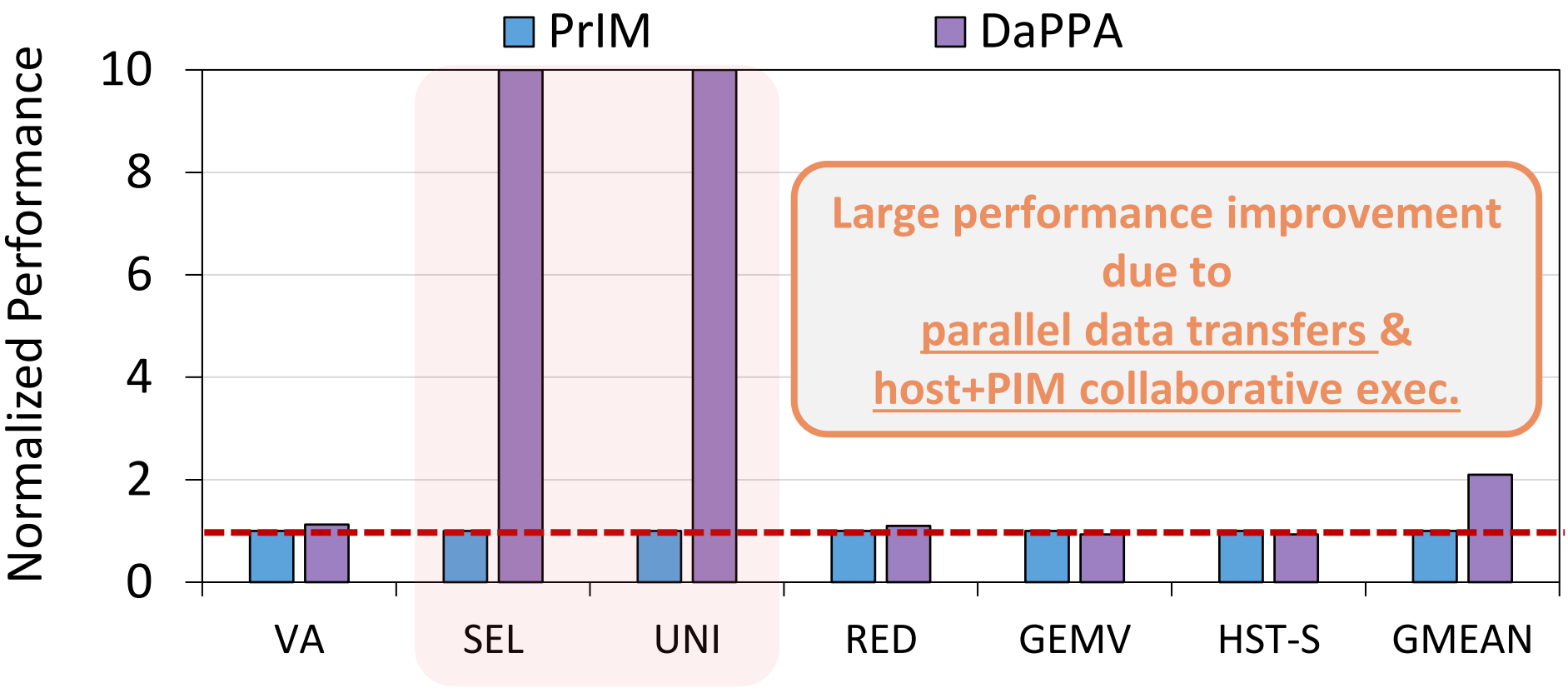


Outline

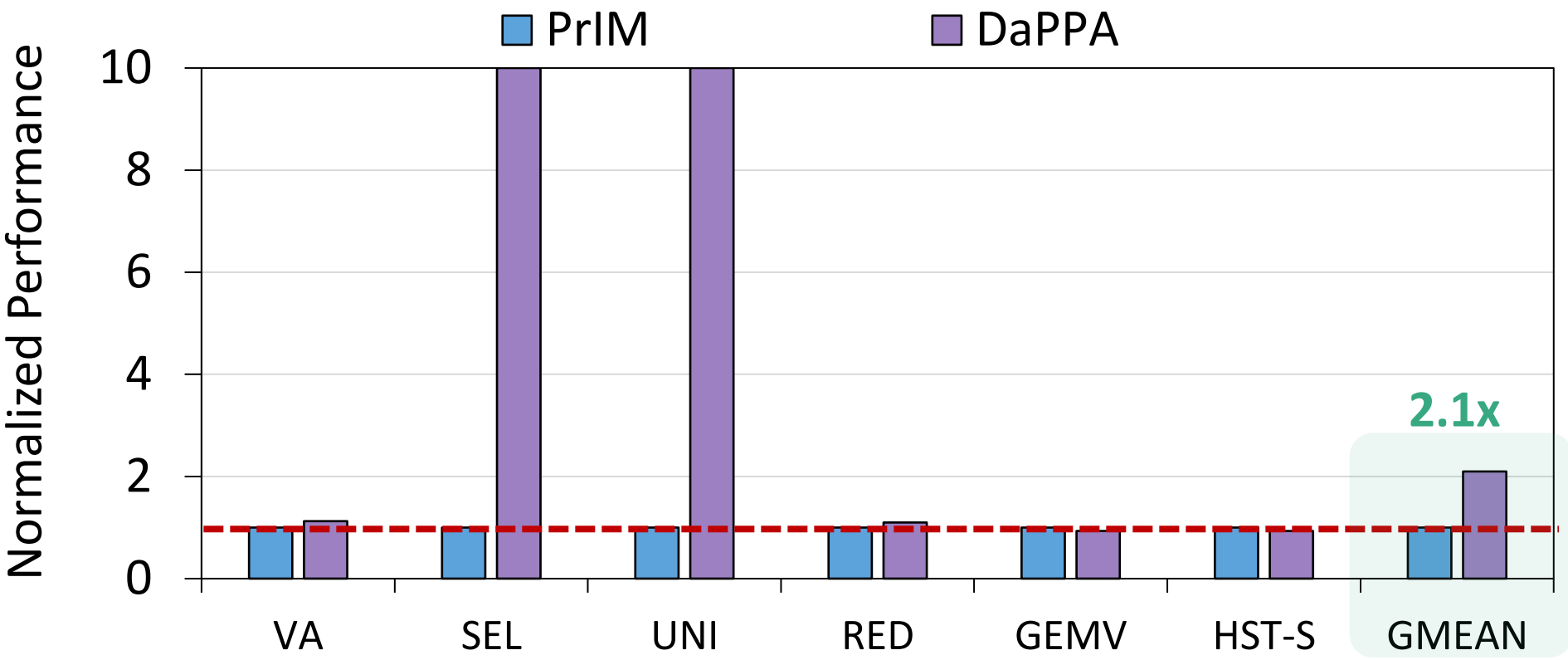
Evaluation:
Methodology Overview

- **Evaluation Setup**
  - **Host CPU:** 2-socket Intel® Xeon Silver 4110 CPU
  - **PIM Cores:** 20 UPMEM PIM DIMMs (160 GB PIM memory)
  - **2560 DPUs** in total
- **Workloads:** 6 workloads from the PrIM benchmark suite
  - Vector addition (**VA**); Select (**SEL**); Unique (**UNI**); Reduce (**RED**); General matrix-vector multiply (**GEMV**); Histogram small (**HST-S**)
- **Metrics**
  - **End-to-end execution time** (average of 10 runs)
  - **Programming complexity** (in lines of code)

Evaluation:
Performance Analysis

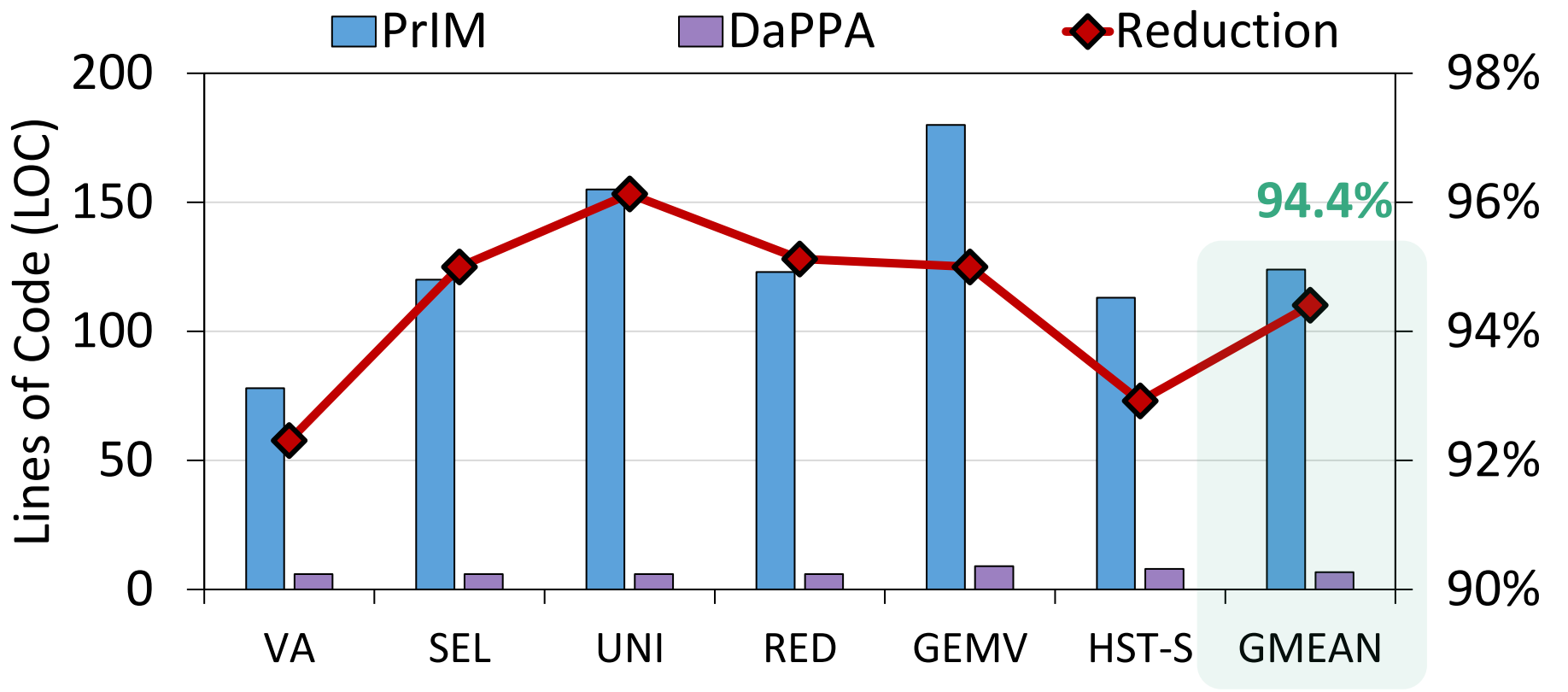


Evaluation:
Performance Analysis



DaPPA significantly improves end-to-end performance compared to hand-tuned implementations

Evaluation:
Programming Complexity Analysis



DaPPA significantly reduces programming complexity by abstracting hardware components

Evaluation:
Comparison to State-of-the-Art

SimplePIM [Chen+, PACT'23]: a framework that uses
(1) iterator functions and (2) primitives for communication
to aid PIM programmability

Compared to SimplePIM, DaPPA provides three key benefits

1. **Higher abstraction level** → The programmer does *not* need to **manually specify communication patterns** used during computation
2. **Support for more parallel patterns** → DaPPA supports **two more parallel primitives** (window and group), and allows the **mixing of parallel patterns**
3. **Further execution optimizations** → DaPPA allows using idle host resources for **collaborative execution**

DaPPA improves state-of-the-art frameworks for PIM programmability

DaPPA:

A Data-Parallel Framework for Processing-in-Memory Architectures

- Geraldo F. Oliveira, Alain Kohli, David Novo, Juan Gómez-Luna, Onur Mutlu
  "DaPPA: A Data-Parallel Framework for Processing-in-Memory Architectures,"
  arXiv:2310.10168 [cs.AR]
  2nd Place ACM Student Research Competition at *the 32nd International Conference on Parallel Architectures and Compilation Techniques* (***PACT***), Vienna, Austria, October 2023.

DaPPA: A Data-Parallel Framework for Processing-in-Memory Architectures

Geraldo F. Oliveira⋆ Alain Kohli⋆ David Novo‡ Juan Gómez-Luna⋆ Onur Mutlu⋆

⋆*ETH Zürich* ‡*LIRMM, Univ. Montpellier, CNRS*

1. Motivation & Problem

The increasing prevalence and growing size of data in modern applications have led to high costs for computation in traditional *processor-centric computing* systems. To mitigate these costs, the *processing-in-memory* (PIM) [1–6] paradigm moves computation closer to where the data resides, reducing the need to move data between memory and the processor. Even though the concept of PIM has been first proposed in the 1960s [7, 8], real-world PIM systems have only recently

face [15, 16] that abstracts the hardware components of the UPMEM system. Using this key idea, DaPPA transforms a data-parallel pattern-based application code into the appropriate UPMEM-target code, including the required APIs for data management and code partition, which can then be compiled into a UPMEM-based binary *transparently* from the programmer. While generating UPMEM-target code, DaPPA implements several code optimizations to improve end-to-end performance.

Outline

Conclusion

Problem: Programming general-purpose processing-in-memory (PIM) systems require **non-trivial effort**

- The programmer needs to (1) have **knowledge of the hardware** and (2) **manually manage** data movement

Goal: Ease **programmability** of general-purpose PIM systems

Two Approaches: SimplePIM & DaPPA

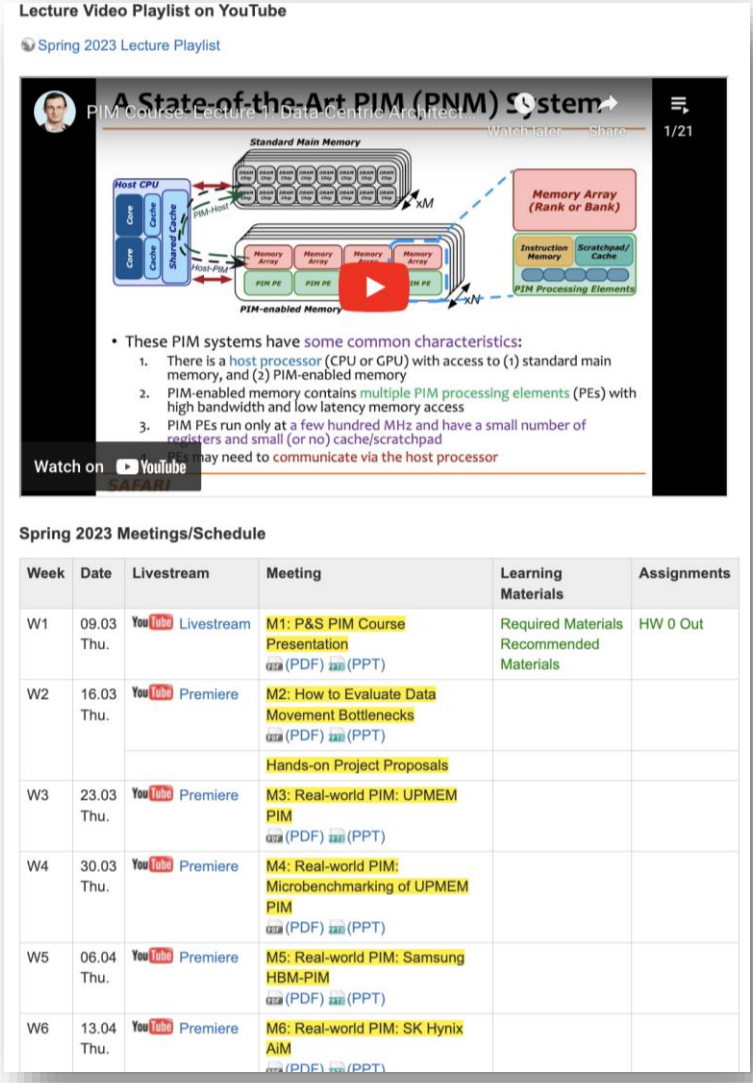
- provides a **data-parallel pattern-based programming interface** that abstracts hardware components;
- automatically **distributes** input and **gathers** output data, **handles** memory management, and **parallelizes** works across PIM cores

Key Results: Our evaluation shows that SimplePIM & DaPPA

- **outperform** hand-tuned workloads
- **reduce** programming complexity (in lines of code)

Processing-in-Memory Course:
Spring 2023

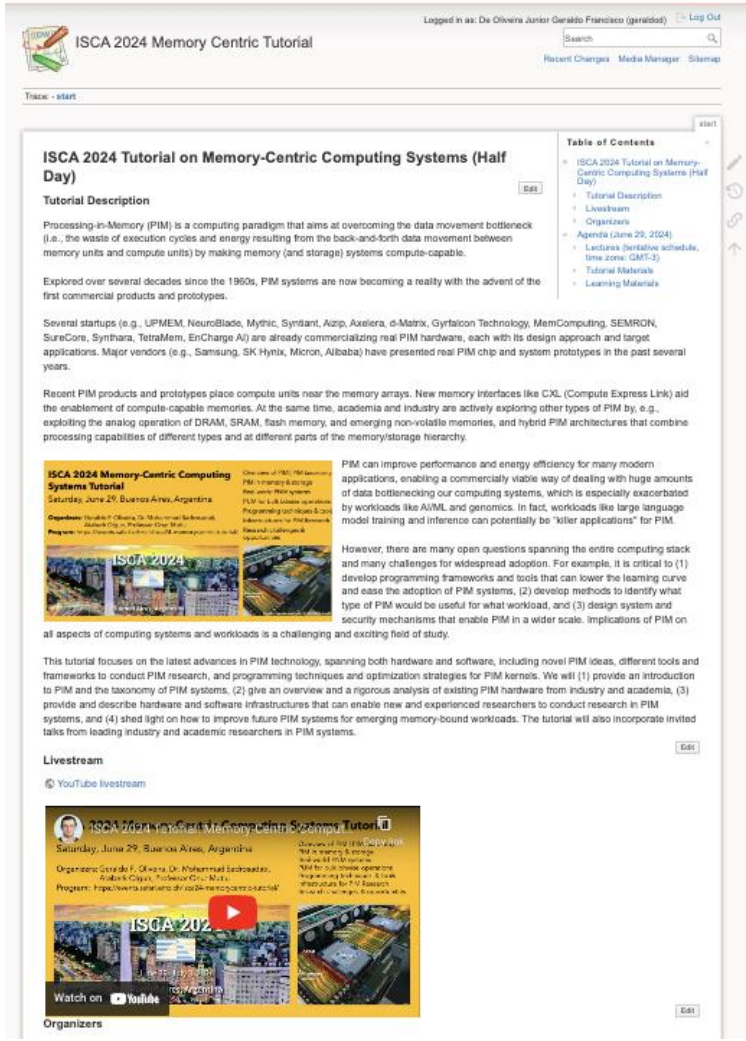
- **Spring 2023 Edition**
  - https://safari.ethz.ch/projects_and_seminars/spring2023/doku.php?id=processing_in_memory
- **Fall 2022 Edition**
  - https://safari.ethz.ch/projects_and_seminars/fall2022/doku.php?id=processing_in_memory
- **YouTube Livestream**
  - https://www.youtube.com/playlist?list=PL5Q2soXY2Zi_EObuoAZVSq_o6UySWQHvZ
- **Bachelor's Course:**
  - Elective at ETH Zurich
  - Introduction to PIM systems (from industry and academia)
  - Tutorial on using real PIM infrastructure
  - Potential research exploration



https://www.youtube.com/onurmutlulectures

PIM Tutorial:
ISCA & MICRO 2024

- ISCA 2024 Edition
  - June 29th: Lectures, hands-on labs, and invited lectures
  - https://events.safari.ethz.ch/isca24-memorycentric-tutorial/doku.php?id=start
- MICRO 2024 Edition (Upcoming)
  - November 2nd: Lectures, hands-on labs, and invited lectures
  - https://events.safari.ethz.ch/micro24-memorycentric-tutorial/doku.php?id=start
- YouTube Livestream
  - https://www.youtube.com/watch?v=KV2MXvcBgb0



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, 2023

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

ABUMPIMP 2024: 2nd Minisymposium on Applications and Benefits of a UPMEM Commercial Massively Parallel Processing-In-Memory Platform

Software Frameworks for Productive and Effective Processing-in-Memory Architectures

Geraldo F. Oliveira

(https://geraldofojunior.github.io/)

Onur Mutlu

ETH zürich

SAFARI