

# SimplePIM:

## A Software Framework for Productive and Efficient Processing-in-Memory

Jinfan Chen, Juan Gómez Luna, Izzat El Hajj, Yuxin Guo, Onur Mutlu

<https://arxiv.org/pdf/2310.01893.pdf>

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

[juang@ethz.ch](mailto:juang@ethz.ch)

# Executive Summary

---

- **Processing-in-Memory** (PIM) promises to alleviate the *data movement bottleneck*
- Real PIM hardware is now available, e.g., UPMEM PIM
- However, **programming real PIM hardware is challenging**, e.g.:
  - Distribute data across PIM memory banks,
  - Manage data transfers between host cores and PIM cores, and between PIM cores,
  - Launch PIM kernels on the PIM cores, etc.
- **SimplePIM** is a high-level programming framework for real PIM hardware
  - Iterators such as map, reduce, and zip
  - Collective communication with broadcast, scatter, and gather
- Implementation on UPMEM and evaluation with six different workloads
  - Reduction, vector add, histogram, linear/logistic regression, K-means
  - **4.4x fewer lines of code** compared to hand-optimized code
  - Between 15% and 43% **faster than hand-optimized code** for three workloads
- Source code: <https://github.com/CMU-SAFARI/SimplePIM>

# Outline

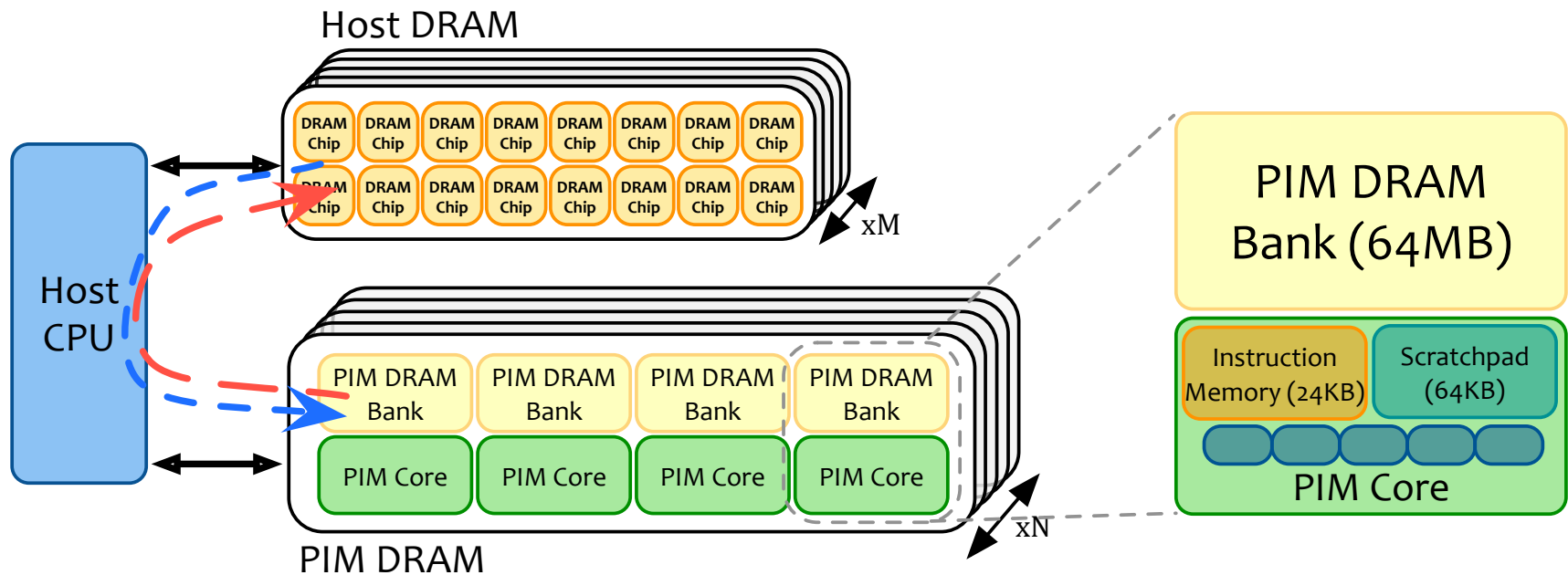
---

Processing-in-memory  
and PIM programming

SimplePIM:  
A high-level programming framework for  
processing-in-memory

Evaluation

# A State-of-the-Art PIM System



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

# Programming a PIM System (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;
}
```

# Programming a PIM System (II)

---

- PIM programming is challenging
  - Manage data movement between host DRAM and PIM DRAM
    - Parallel, serial, broadcast, and gather/scatter transfers
  - Manage data movement between PIM DRAM bank and scratchpad
    - 8-byte aligned and maximum of 2,048 bytes
  - Multithreaded programming model
  - Inter-thread synchronization
    - Barriers, handshakes, mutexes, and semaphores

## Our Goal

Design a **high-level programming framework** that abstracts these hardware-specific complexities and provides a **clean yet powerful interface** for ease of use and **high program performance**

# Outline

---

Processing-in-memory  
and PIM programming

SimplePIM:  
A high-level programming framework for  
processing-in-memory

Evaluation

# The SimplePIM Programming Framework

---

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

- **Management interface**

- Metadata for PIM-resident arrays

- **Communication interface**

- Abstractions for host-PIM and PIM-PIM communication

- **Processing interface**

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



# 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);
```

# The SimplePIM Programming Framework

---

- SimplePIM provides standard abstractions to build and deploy applications on PIM systems
  - Management interface
    - Metadata for PIM-resident arrays
  - Communication interface
    - Abstractions for host-PIM and PIM-PIM communication
  - Processing interface
    - Iterators (map, reduce, zip) to implement workloads

# Host-to-PIM Communication: Broadcast

- SimplePIM 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);
```



# Host-to-PIM Communication: Scatter/Gather

- SimplePIM Scatter

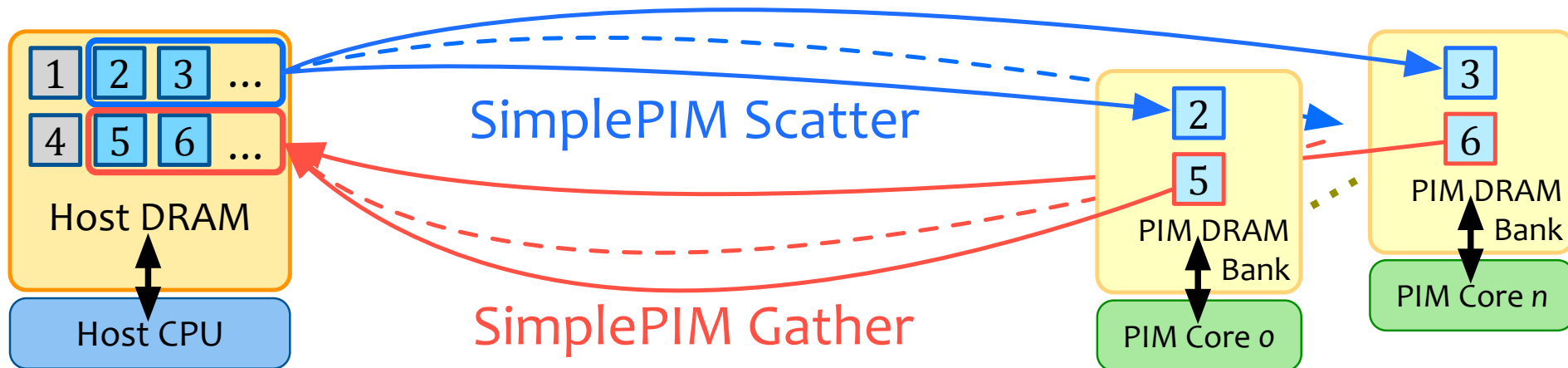
- Distributes an array to PIM DRAM banks

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

- SimplePIM Gather

- Collects portions of an array from PIM DRAM banks

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

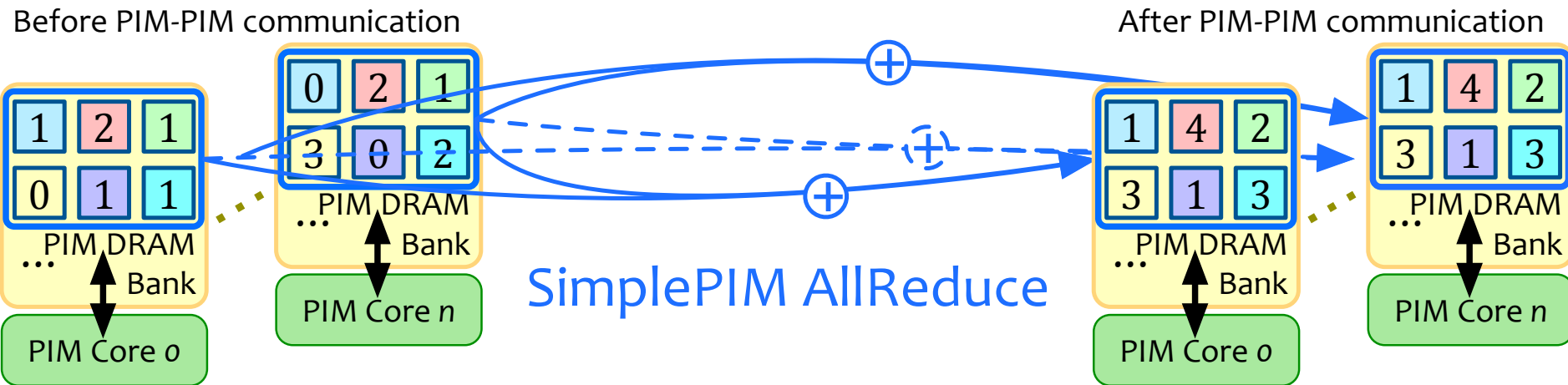


# PIM-PIM Communication: AllReduce

- SimplePIM 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);
```

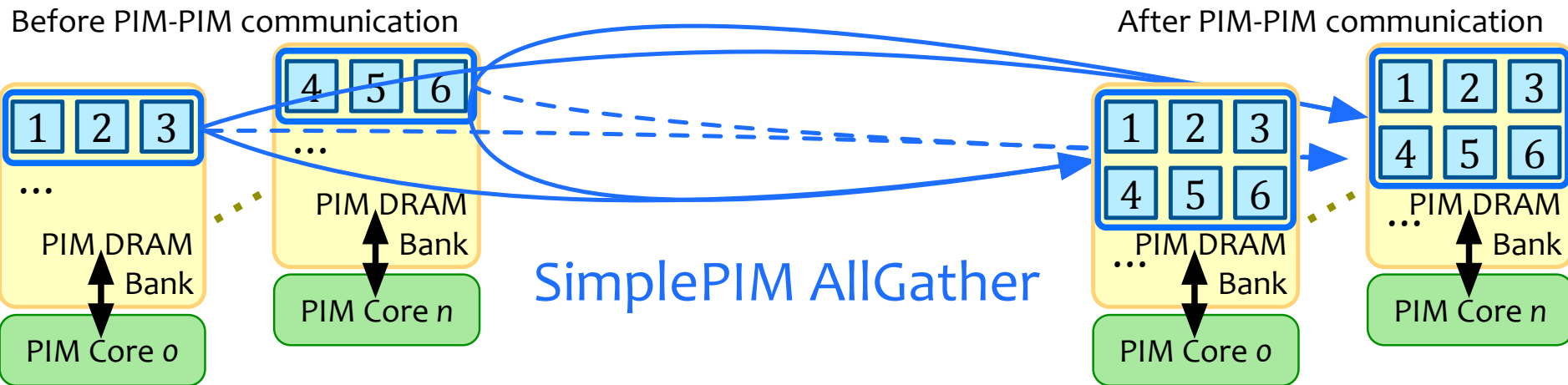


# PIM-PIM Communication: AllGather

- SimplePIM 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);
```



# The SimplePIM Programming Framework

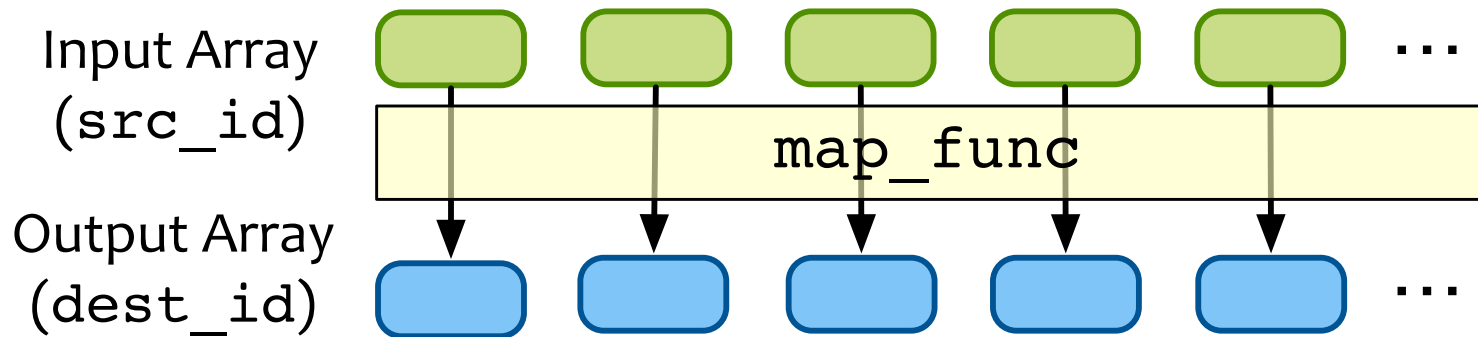
---

- SimplePIM provides standard abstractions to build and deploy applications on PIM systems
  - Management interface
    - Metadata for PIM-resident arrays
  - Communication interface
    - Abstractions for host-PIM and PIM-PIM communication
  - Processing interface
    - Iterators (map, reduce, zip) to implement workloads

# Processing Interface: Map

- 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);
```



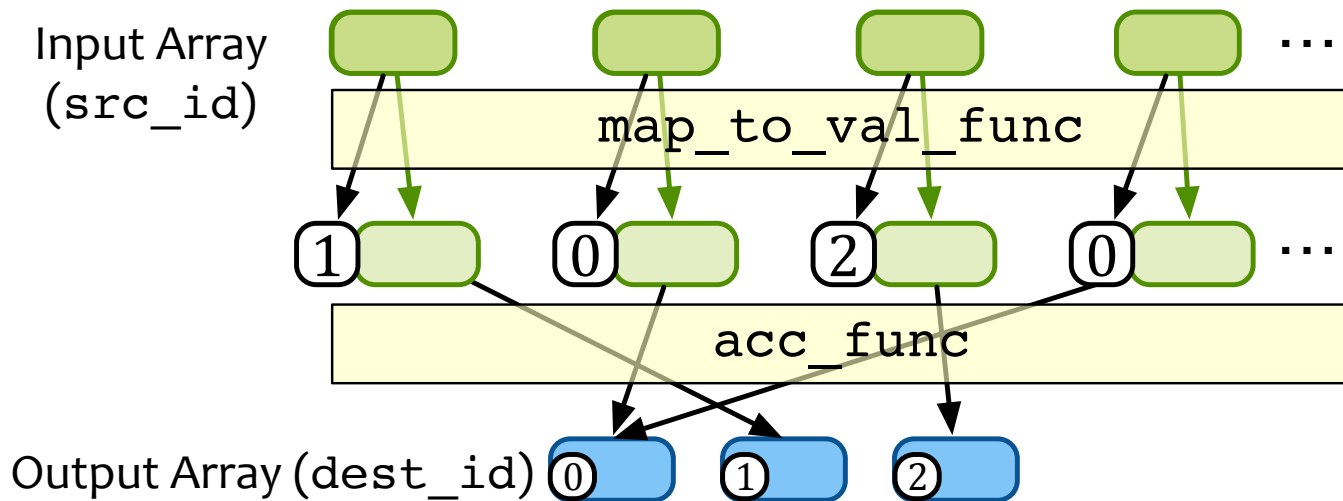


# Processing Interface: Reduction

- 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);
```

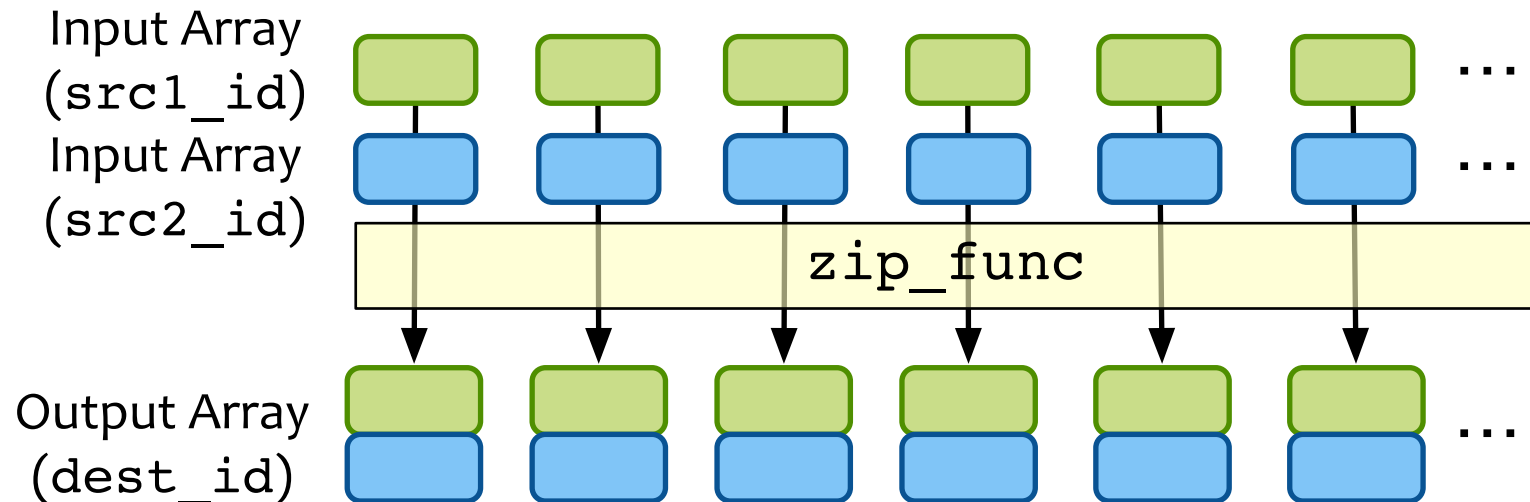


# Processing Interface: Zip

- 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's UPMEM Implementation

---

- Communication interface
  - SimplePIM automatically handles alignment requirements and inserts padding as needed
- Processing interface
  - Array map
    - Invokes PIM cores and PIM threads, and handles PIM DRAM-scratchpad transfers
  - Array reduction
    - Shared accumulator reduction
    - Thread-private accumulator reduction
  - Array zip
    - Lazy approach to minimize data copying

# General Code Optimizations

---

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

# More in the Paper

---

- Strength reduction
- Loop unrolling

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

Jinfan Chen<sup>1</sup>   Juan Gómez-Luna<sup>1</sup>   Izzat El Hajj<sup>2</sup>   Yuxin Guo<sup>1</sup>   Onur Mutlu<sup>1</sup>  
<sup>1</sup>ETH Zürich   <sup>2</sup>American University of Beirut

• FUNCTIONALITY

- Adju   <https://arxiv.org/pdf/2310.01893.pdf>

# Outline

---

Processing-in-memory  
and PIM programming

SimplePIM:  
A high-level programming framework for  
processing-in-memory

Evaluation

# 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

# Productivity 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;
}
```



# Productivity 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);
```

# Productivity Improvement (III)

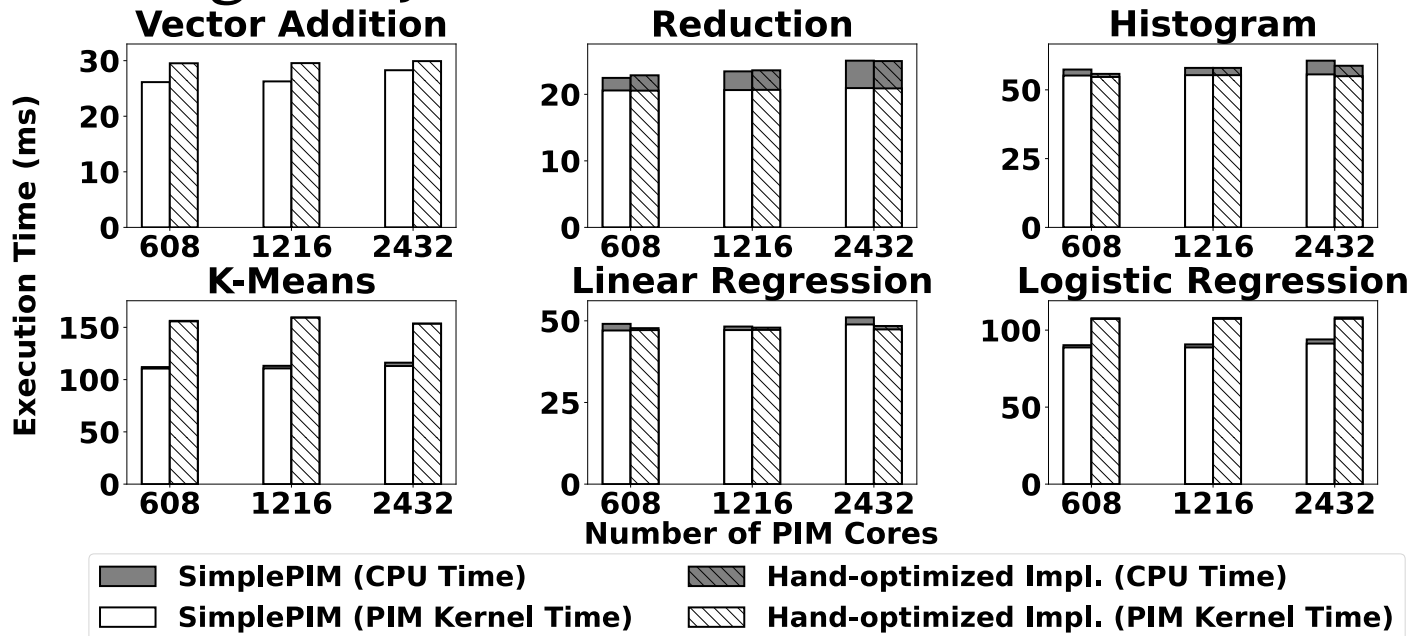
- Lines of code (LoC) reduction

	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×

# Performance Evaluation (I)

- Weak scaling analysis

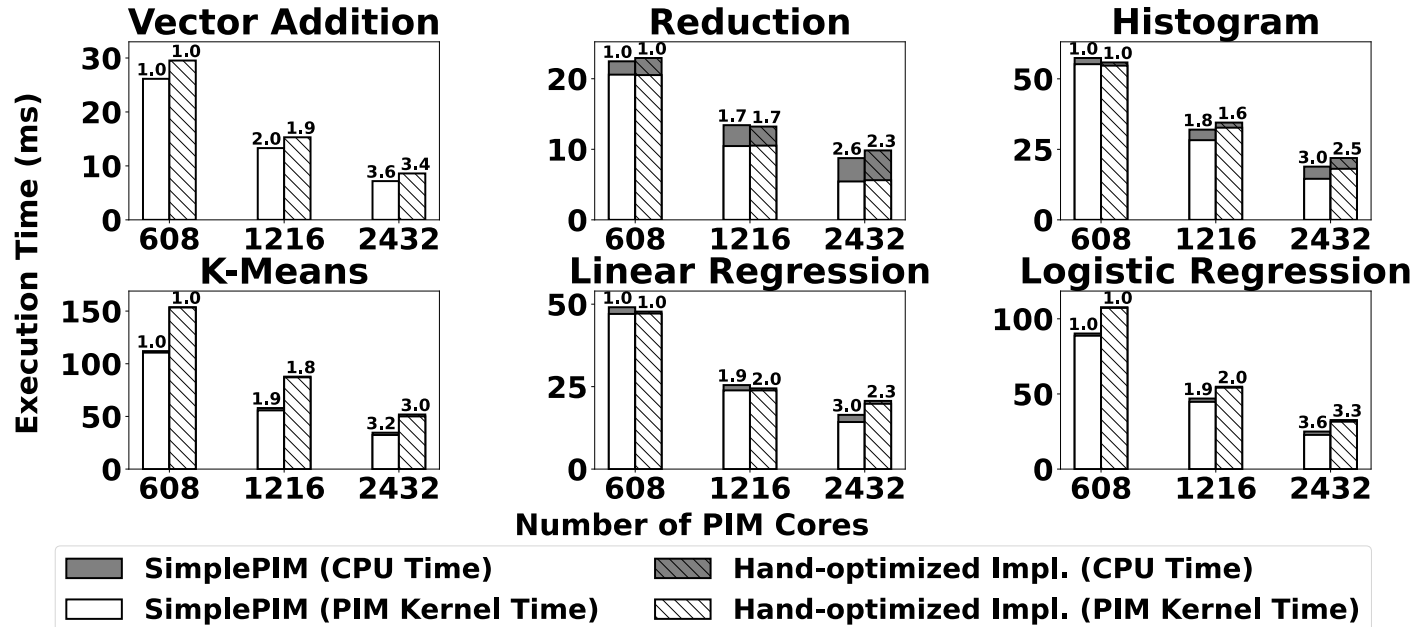


SimplePIM achieves **comparable performance** for reduction, histogram, and linear regression

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

# Performance Evaluation (II)

- Strong scaling analysis

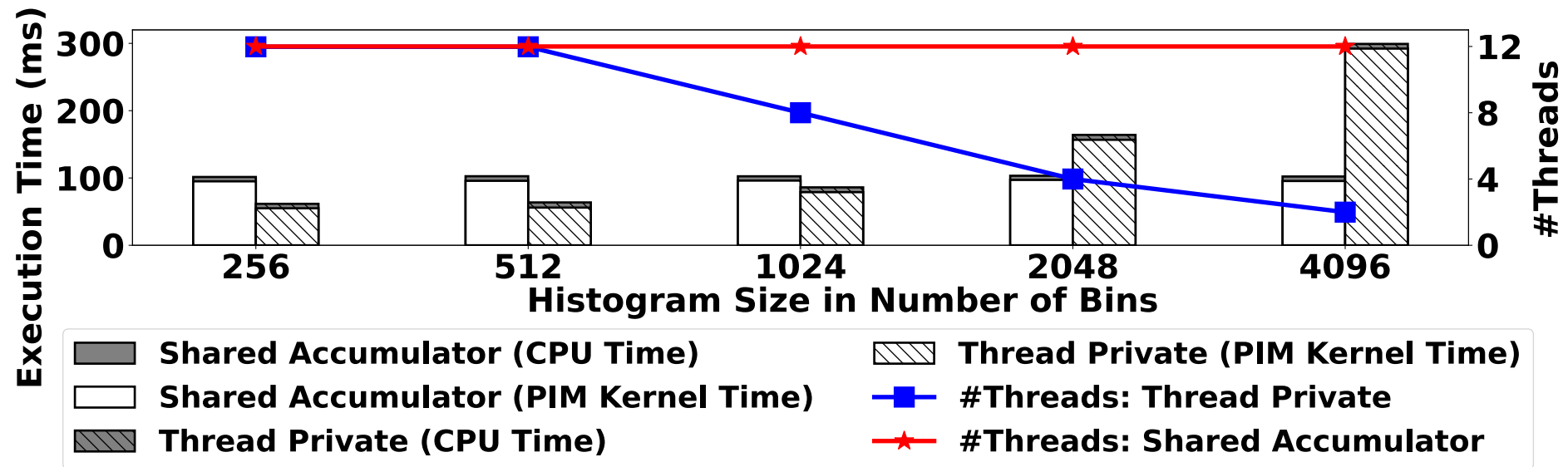


SimplePIM scales better than hand-optimized implementations for reduction, histogram, and linear regression

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

# Variants of Array Reduction

- Shared accumulator version versus thread-private version for histogram



The thread-private version is **up to 70% faster** than the shared accumulator version for histograms of 256-1024 bins

The number of active PIM threads of the thread-private version reduces after 1024 bins due to limited scratchpad size

# Discussion

---

- SimplePIM is devised for PIM architectures with
  - A host processor with access to standard main memory and PIM-enabled memory
  - PIM processing elements (PEs) that communicate via the host processor
  - The number of PIM PEs scales with memory capacity
- SimplePIM emulates the communication between PIM cores via the host processor
- Other parallel patterns can be incorporated in future work
  - Prefix sum and filter can be easily added
  - Stencil and convolution would require fine-grained scatter-gather for halo cells
  - Random access patterns would be hard to support

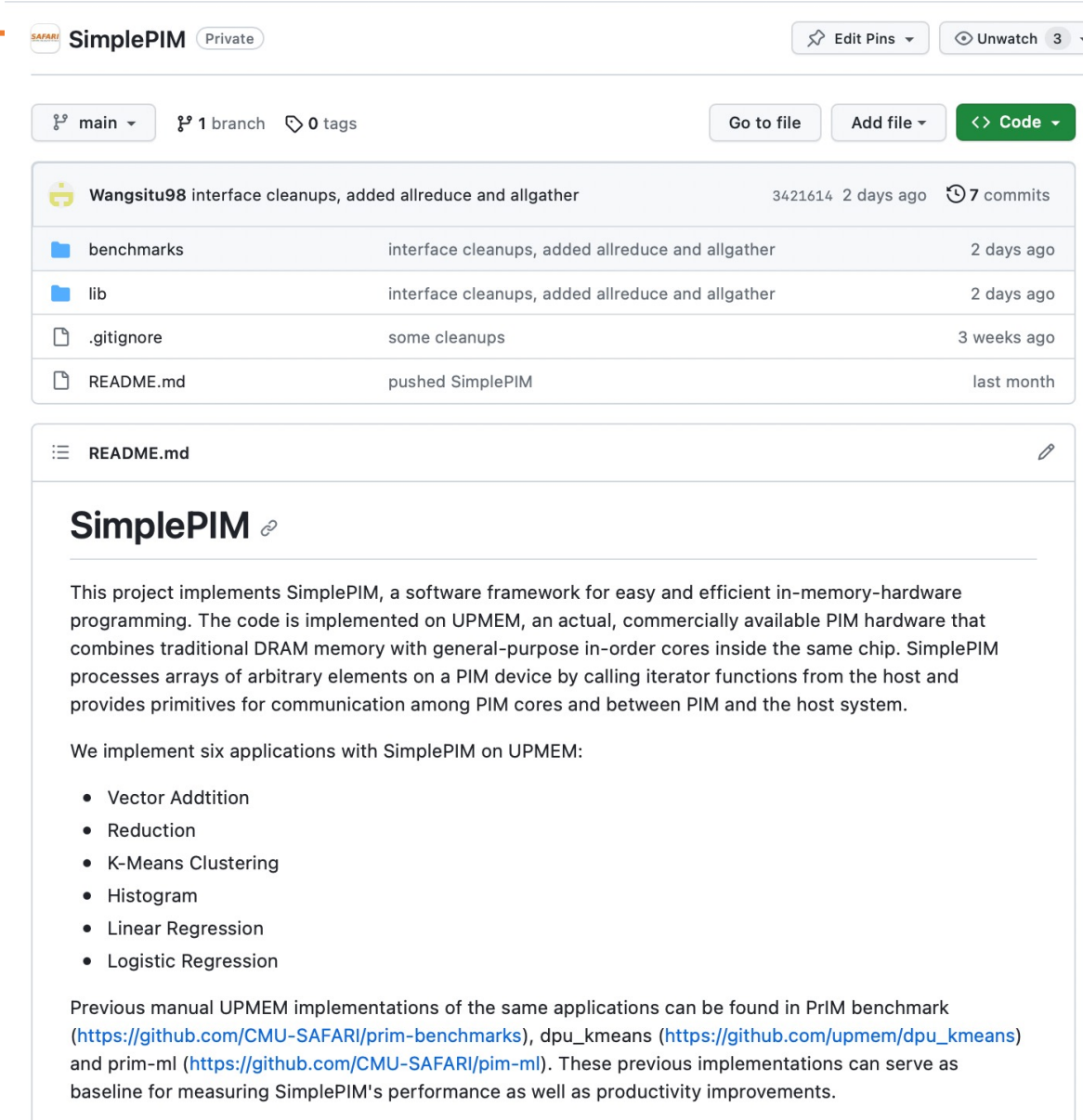
## SimplePIM: A Software Framework for Productive and Efficient Processing-in-Memory

Jinfan Chen<sup>1</sup>   Juan Gómez-Luna<sup>1</sup>   Izzat El Hajj<sup>2</sup>   Yuxin Guo<sup>1</sup>   Onur Mutlu<sup>1</sup>  
<sup>1</sup>ETH Zürich   <sup>2</sup>American University of Beirut

<https://arxiv.org/pdf/2310.01893.pdf>

# Source Code

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



The screenshot shows the GitHub repository for SimplePIM. At the top, it says 'SimplePIM Private' with 'Edit Pins' and 'Unwatch 3' options. Below that, it shows 'main' branch, '1 branch', and '0 tags'. There are buttons for 'Go to file', 'Add file', and 'Code'. The commit history shows a commit by Wangsitu98 with the message 'interface cleanups, added allreduce and allgather' from 2 days ago. Below the commit history is a table of files:

File	Commit Message	Time
benchmarks	interface cleanups, added allreduce and allgather	2 days ago
lib	interface cleanups, added allreduce and allgather	2 days ago
.gitignore	some cleanups	3 weeks ago
README.md	pushed SimplePIM	last month

Below the table is the README.md file content:

## SimplePIM

This project implements SimplePIM, a software framework for easy and efficient in-memory-hardware programming. The code is implemented on UPMEM, an actual, commercially available PIM hardware that combines traditional DRAM memory with general-purpose in-order cores inside the same chip. SimplePIM processes arrays of arbitrary elements on a PIM device by calling iterator functions from the host and provides primitives for communication among PIM cores and between PIM and the host system.

We implement six applications with SimplePIM on UPMEM:

- Vector Addition
- Reduction
- K-Means Clustering
- Histogram
- Linear Regression
- Logistic Regression

Previous manual UPMEM implementations of the same applications can be found in PRIM benchmark (<https://github.com/CMU-SAFARI/prim-benchmarks>), dpu\_kmeans ([https://github.com/upmem/dpu\\_kmeans](https://github.com/upmem/dpu_kmeans)) and prim-ml (<https://github.com/CMU-SAFARI/pim-ml>). These previous implementations can serve as baseline for measuring SimplePIM's performance as well as productivity improvements.



# Executive Summary

---

- **Processing-in-Memory** (PIM) promises to alleviate the *data movement bottleneck*
- Real PIM hardware is now available, e.g., UPMEM PIM
- However, **programming real PIM hardware is challenging**, e.g.:
  - Distribute data across PIM memory banks,
  - Manage data transfers between host cores and PIM cores, and between PIM cores,
  - Launch PIM kernels on the PIM cores, etc.
- **SimplePIM** is a high-level programming framework for real PIM hardware
  - Iterators such as map, reduce, and zip
  - Collective communication with broadcast, scatter, and gather
- Implementation on UPMEM and evaluation with six different workloads
  - Reduction, vector add, histogram, linear/logistic regression, K-means
  - **4.4x fewer lines of code** compared to hand-optimized code
  - Between 15% and 43% **faster than hand-optimized code** for three workloads
- 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, Onur Mutlu

<https://arxiv.org/pdf/2310.01893.pdf>

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

[juang@ethz.ch](mailto:juang@ethz.ch)