

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

Geraldo F. Oliveira

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

Onur Mutlu

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

1

Introduction

2

The Programmability Barrier

3

SimplePIM Overview

Management, Communication & Processing Interfaces

Evaluation Results

4

DaPPA Overview

DaPPA Main Components

Evaluation Results

5

Conclusion

Outline

1

Introduction

2

The Programmability Barrier

3

SimplePIM Overview

Management, Communication & Processing Interfaces

Evaluation Results

4

DaPPA Overview

DaPPA Main Components

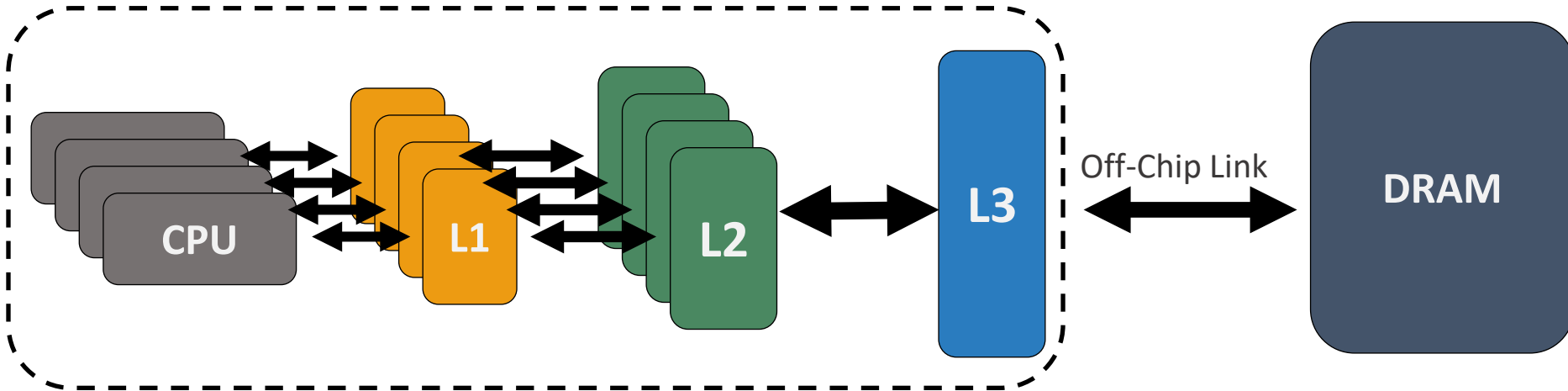
Evaluation Results

5

Conclusion

Data Movement Bottleneck: The Problem

Data movement is a major bottleneck
in modern computer architectures

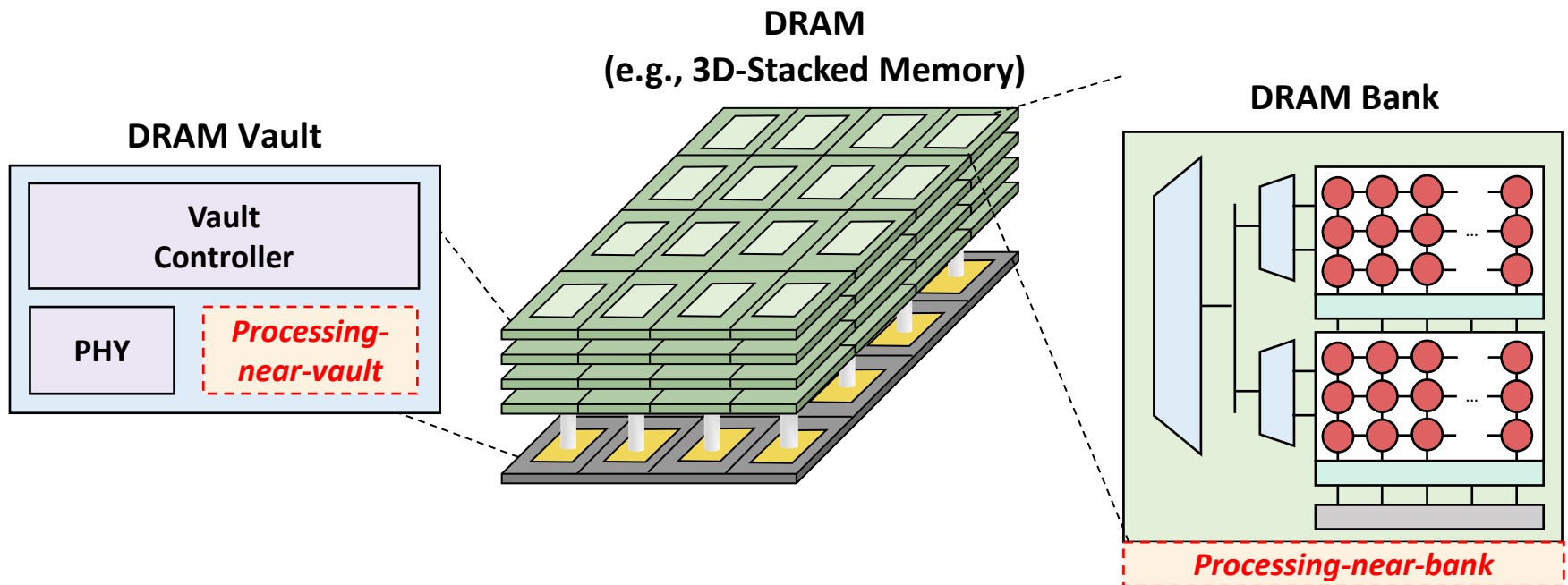


Over 60% of the total system energy is spent on
data movement¹

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)

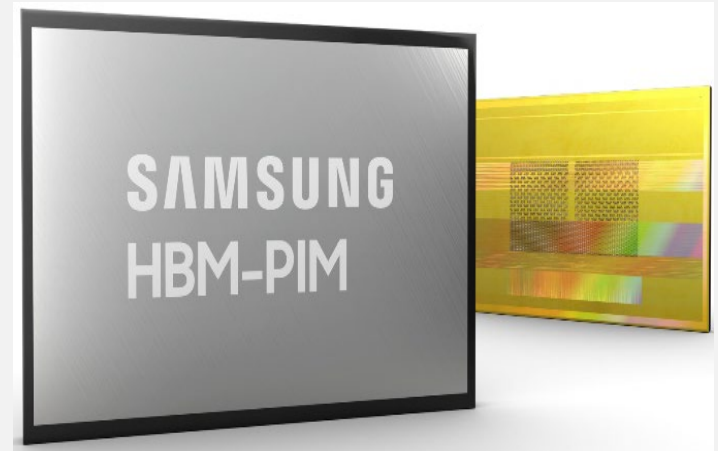
Processing-in-Memory architectures are
now commercially available

SK hynix AiM



Near-DRAM-bank processing
for neural networks

Alibaba HB-PNM

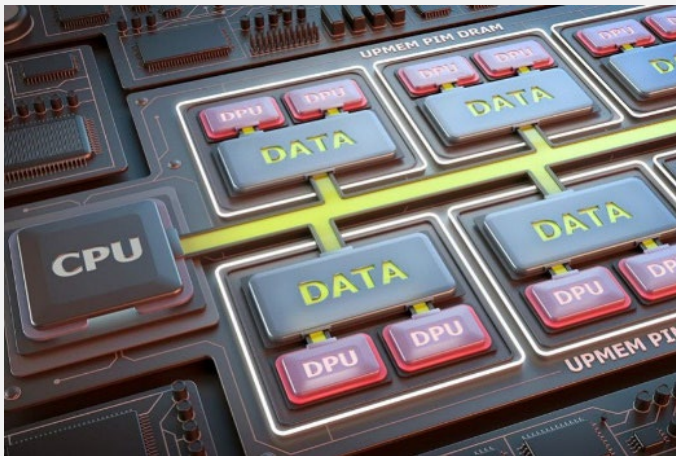


Hybrid bonding with logic
for recommendation systems

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

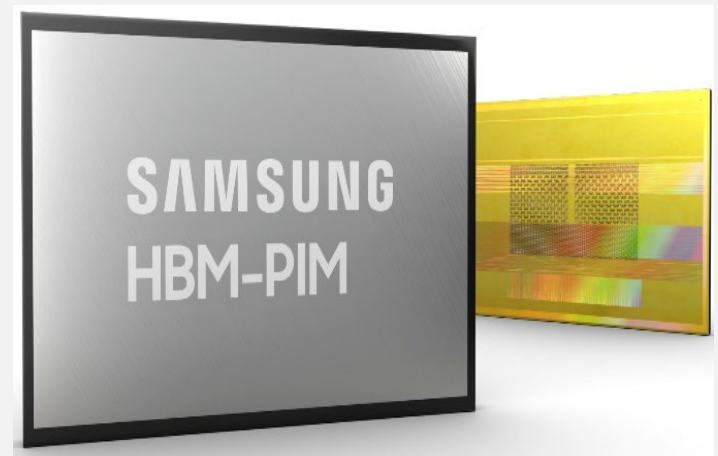
Processing-in-Memory architectures are
now commercially available

UPMEM



Near-DRAM-bank processing
for general-purpose computing

Samsung HBM-PIM

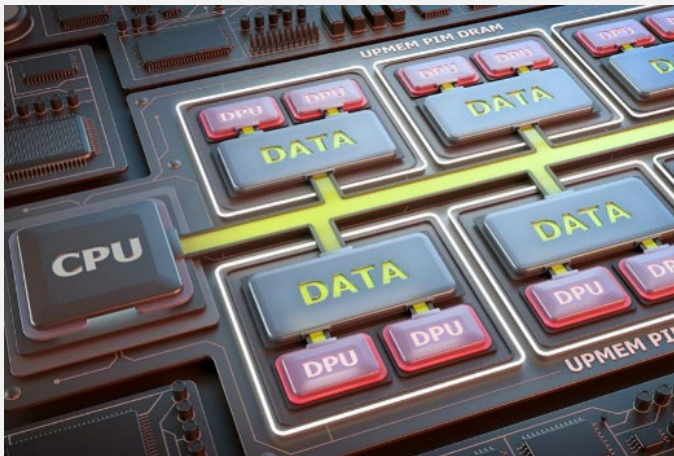


Near-DRAM-bank processing
for neural networks

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

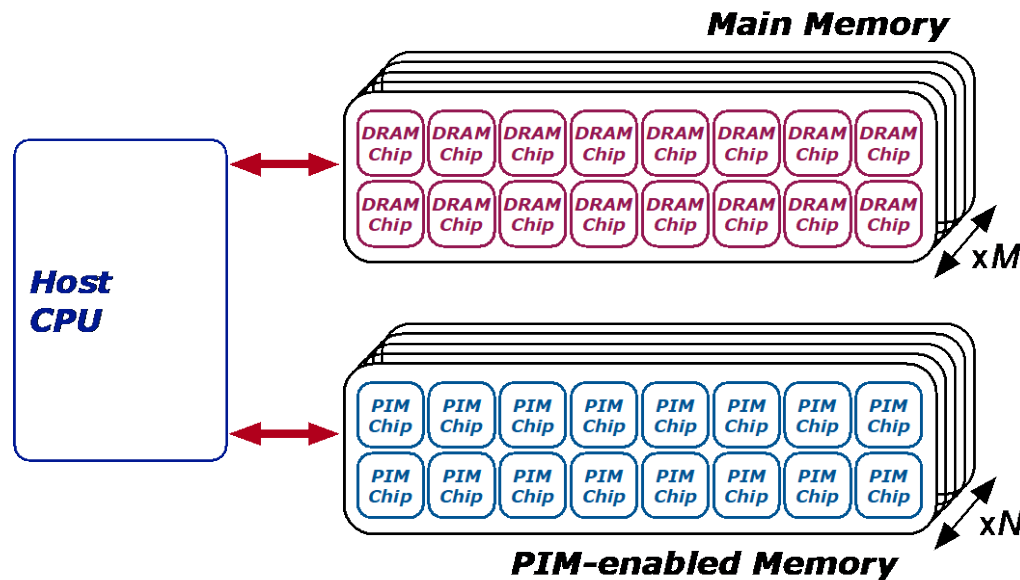
Samsung HBM-PIM



Near-DRAM-bank processing
for neural networks

Processing-in-Memory: The UPMEM Architecture (I)

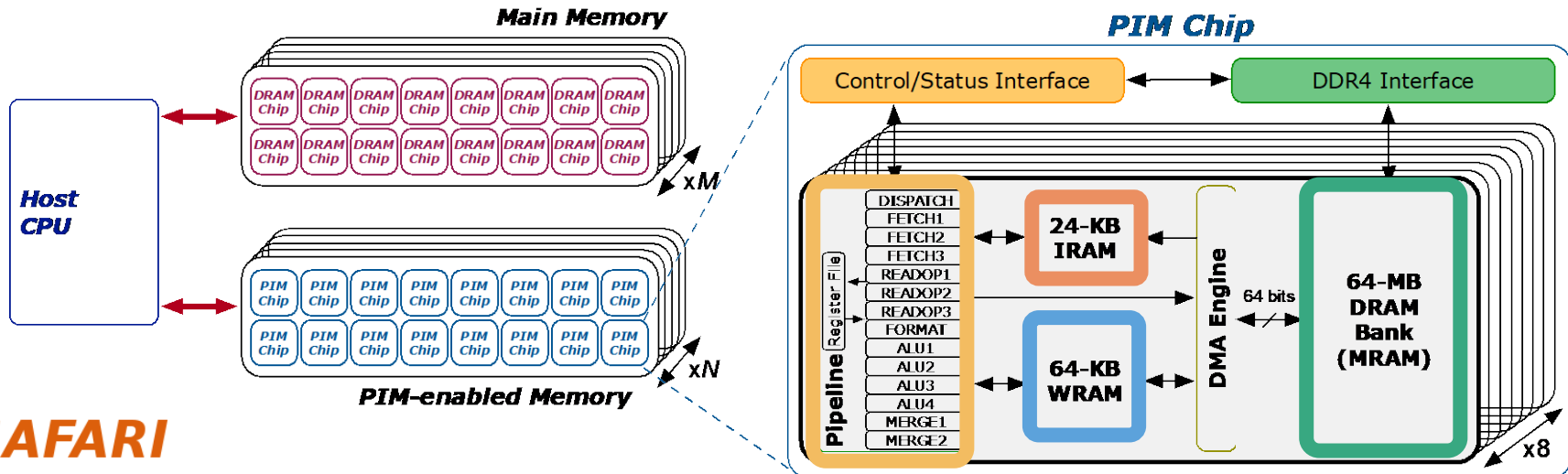
- In a UPMEM-based PIM system UPMEM DIMMs coexist with regular DDR4 DIMMs



Integration of UPMEM PIM in a system follows
an accelerator model

Processing-in-Memory: The UPMEM Architecture (II)

- 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: Main RAM (MRAM)** banks
 - 8 **24-KB instruction cache: Instruction RAM (IRAM)**
 - 8 **64KB scratchpad memory: Working RAM (WRAM)**
 - 8 **GP in-order cores: DRAM Processing Units (DPUs)**



Outline

1

Introduction

2

The Programmability Barrier

3

SimplePIM Overview

Management, Communication & Processing Interfaces

Evaluation Results

4

DaPPA Overview

DaPPA Main Components

Evaluation Results

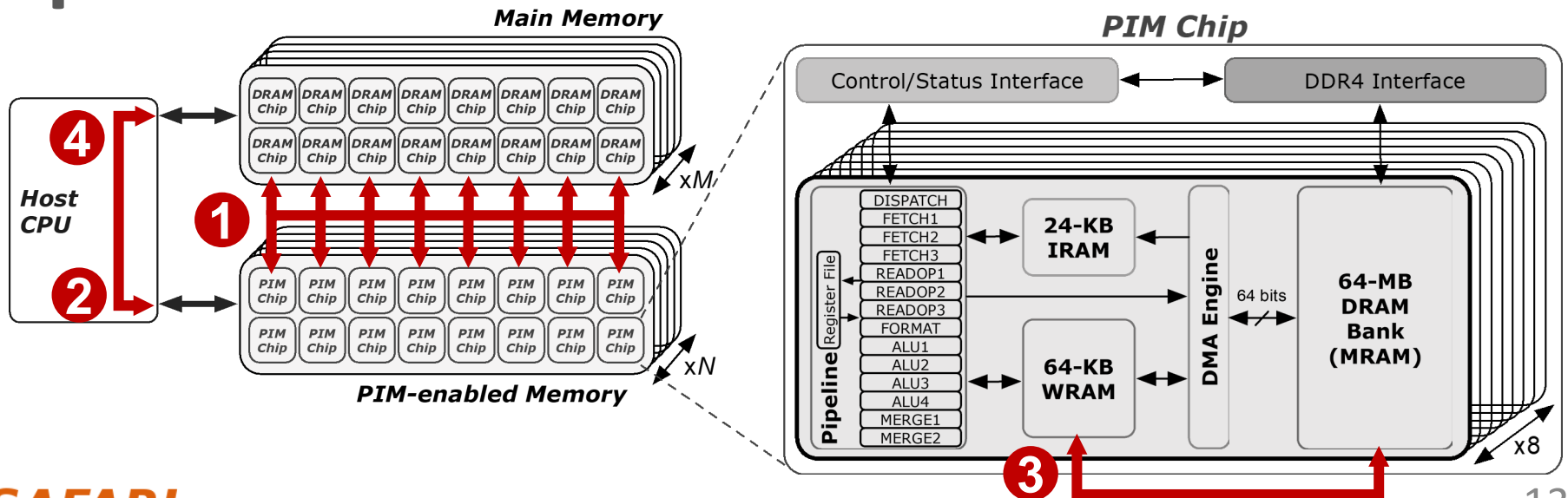
5

Conclusion

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;
mmio_read((__mmio_ptr void const*)(mmio_base_addr_A + byte_index),
          cache_A, l_size_bytes);
mmio_read((__mmio_ptr void const*)(mmio_base_addr_B + byte_index),
          cache_B, l_size_bytes);
vector_addition(cache_B, cache_A, l_size_bytes >> DIV);
mmio_write(cache_B, (__mmio_ptr void*)(mmio_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:

Align
data

Collect
parameters

Distribute
parameters

Launch
computation

Collect
results

Manage
scratchpad

Orchestrate
computation

Goal:

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

1

Introduction

2

The Programmability Barrier

3

SimplePIM Overview

Management, Communication & Processing Interfaces

Evaluation Results

4

DaPPA Overview

DaPPA Main Components

Evaluation Results

5

Conclusion

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 Chen¹ Juan Gómez-Luna¹ Izzat El Hajj² Yuxin Guo¹ Onur Mutlu¹
¹ETH Zürich ²American 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**

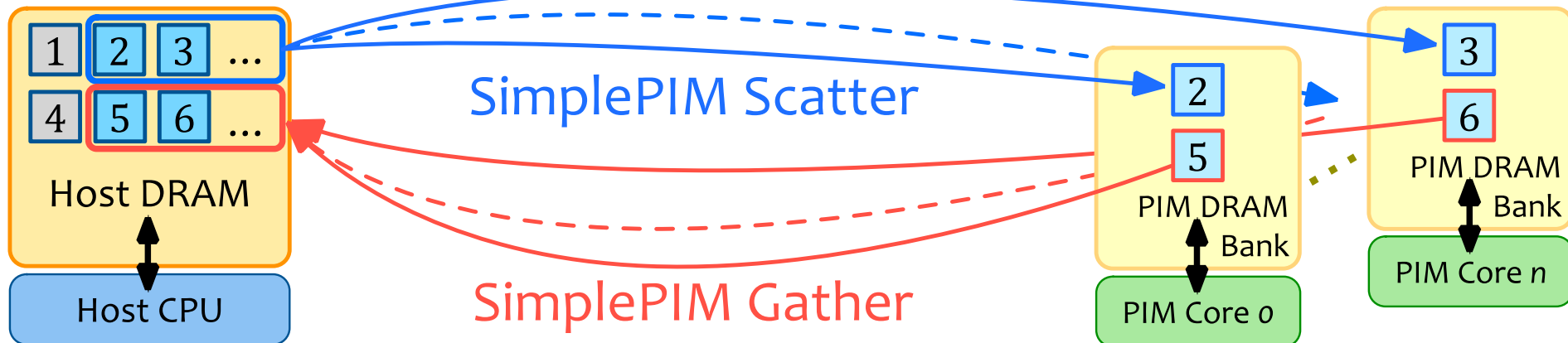
- Distributes an array to PIM DRAM banks

```
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);
```

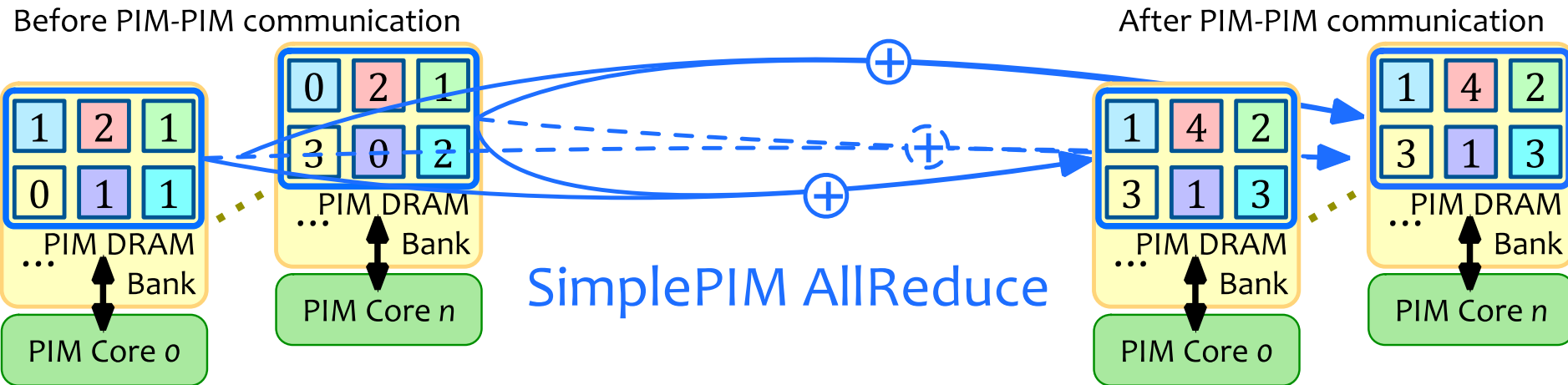


SimplePIM Programming Framework: Communication Interface (III)

• 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);
```

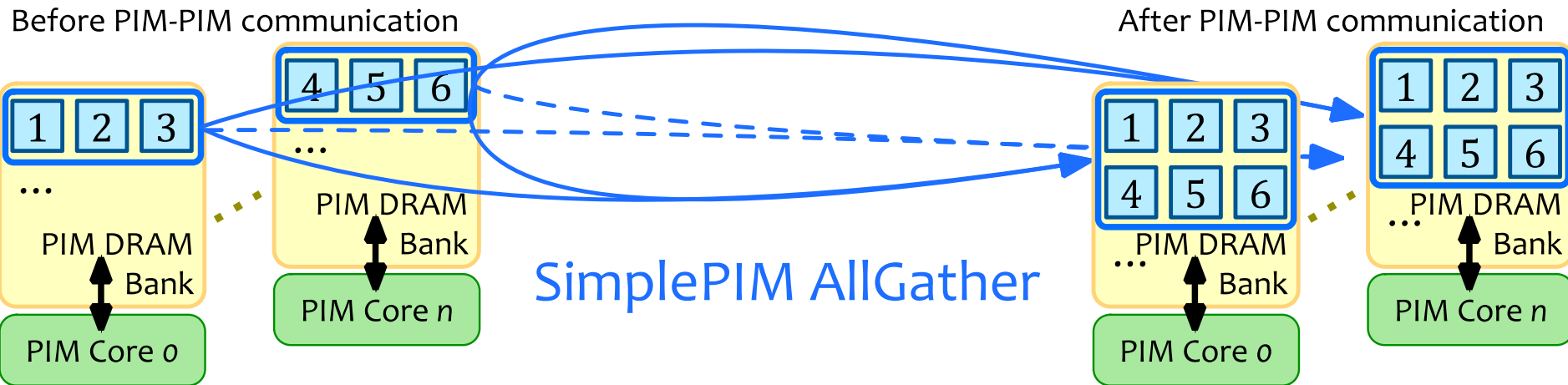


SimplePIM Programming Framework: Communication Interface (IV)

• 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);
```

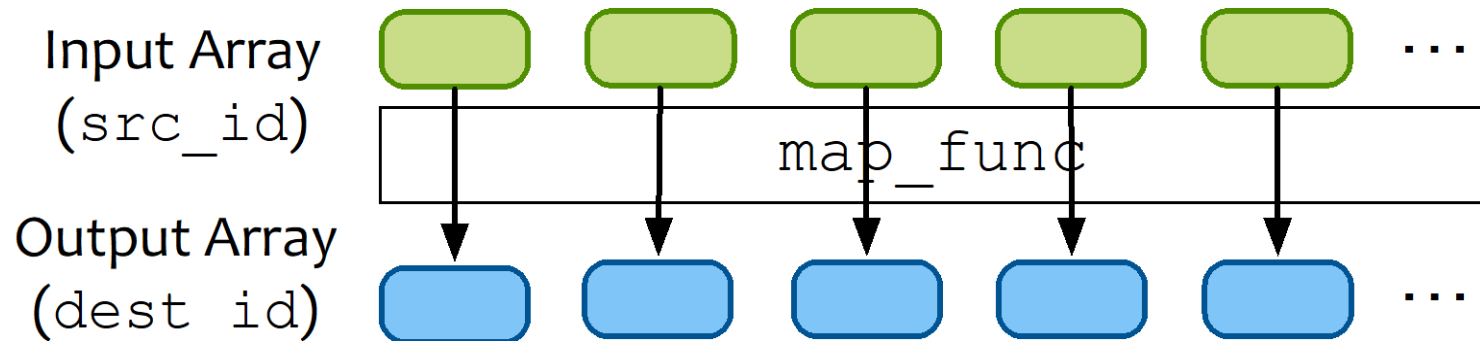


SimplePIM Programming Framework: Processing Interface (I)

- **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);
```

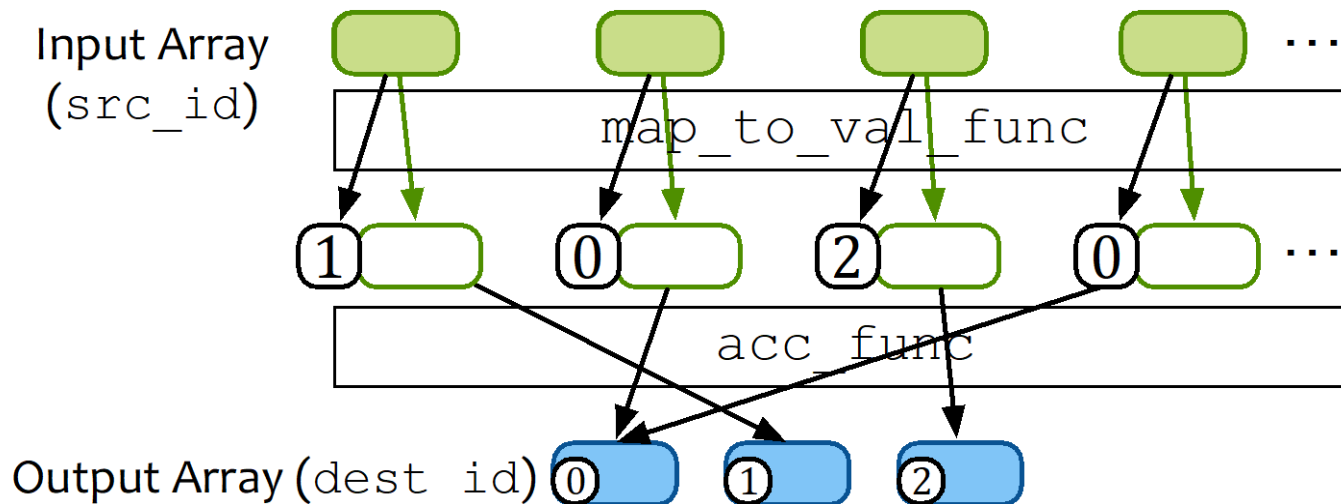


SimplePIM Programming Framework: Processing Interface (II)

• 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);
```

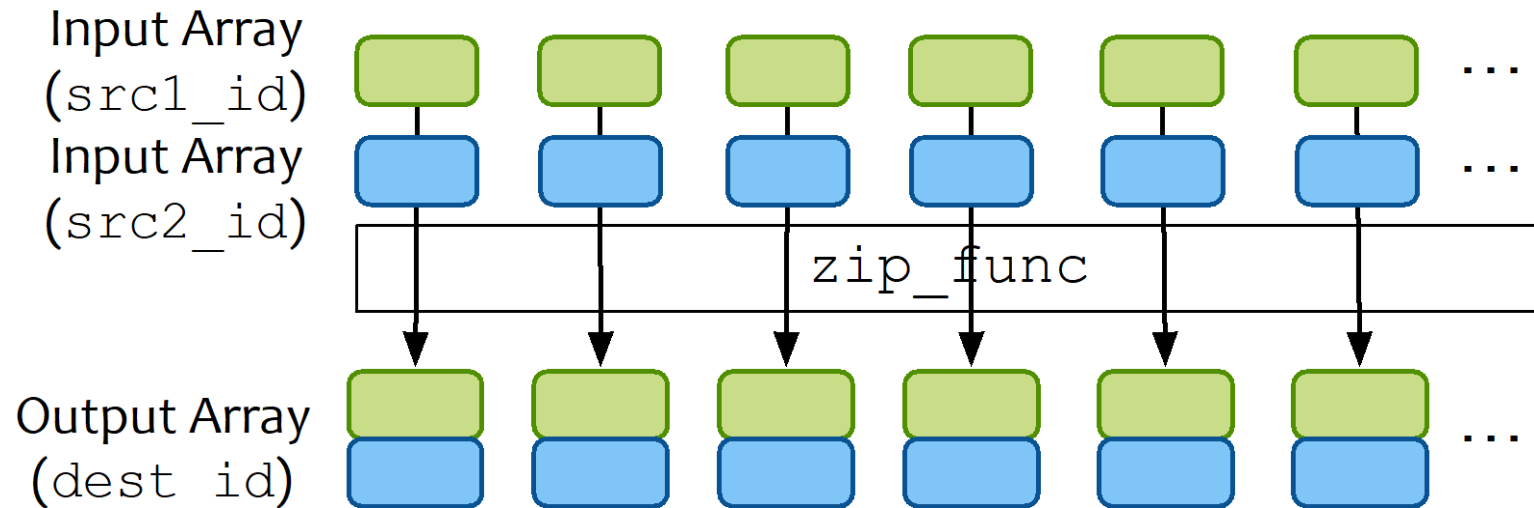


SimplePIM Programming Framework: Processing Interface (III)

- **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

1

Introduction

2

The Programmability Barrier

3

SimplePIM Overview

Management, Communication & Processing Interfaces

Evaluation Results

4

DaPPA Overview

DaPPA Main Components

Evaluation Results

5

Conclusion

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);
```

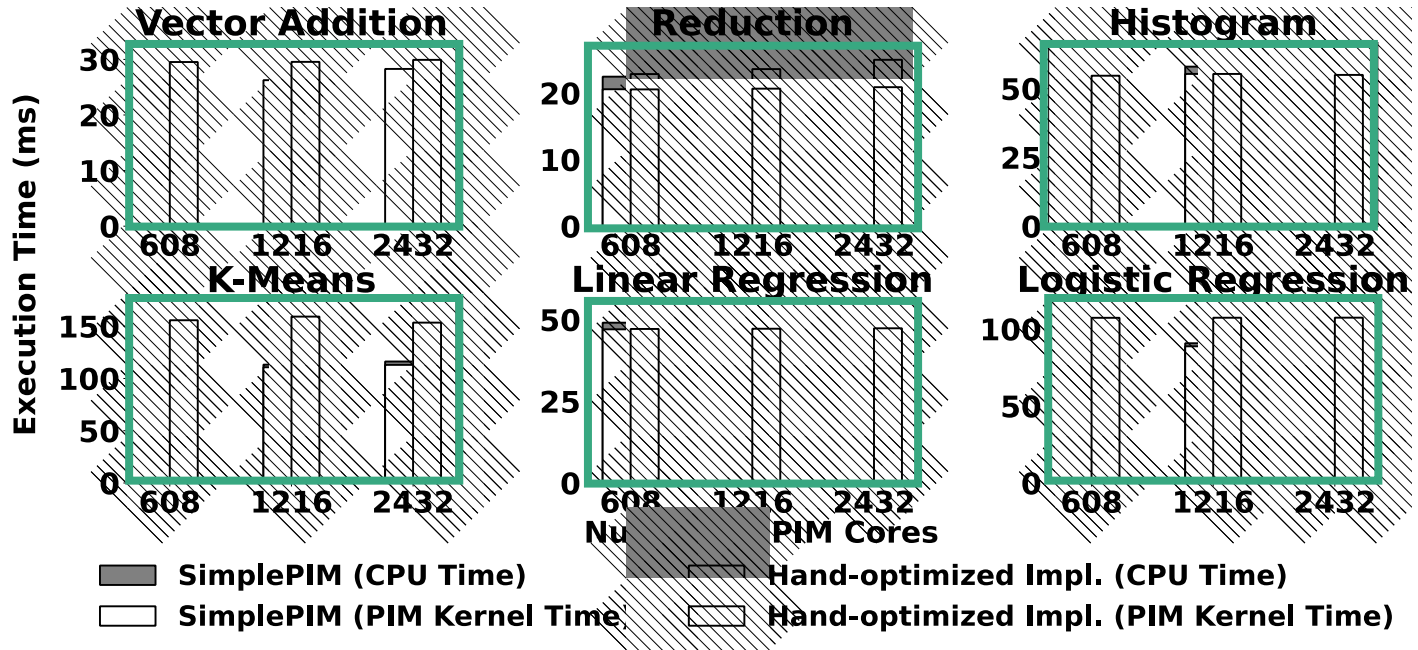

Evaluation Results: Productive Improvement (III)

- 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×**

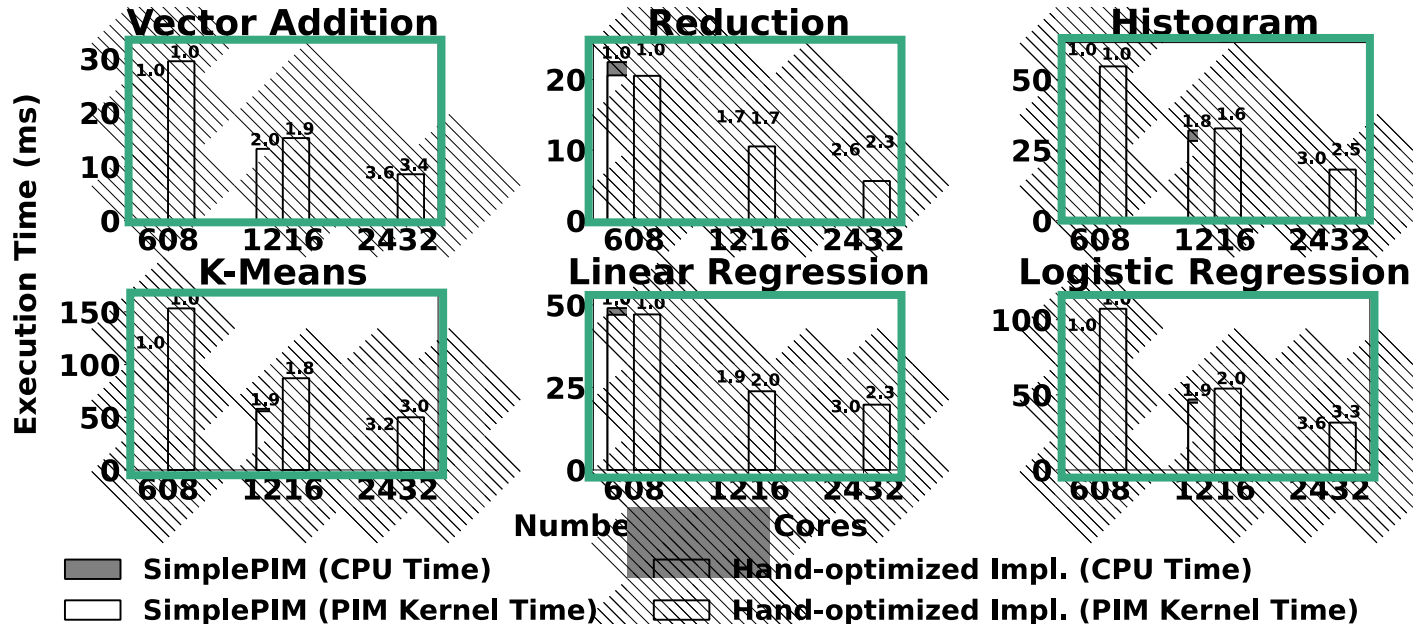
Evaluation Results: Weak Scaling Analysis



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>

CMU-SAFARI / SimplePIM Public

Notifications F

<> Code Issues Pull requests Actions Projects Security Insights

main Branches Tags Code

13 Commits

- benchmarks
- lib
- .gitignore
- LICENSE
- README.md

README MIT license

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

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:

About

SimplePIM is the first high-level programming framework for real-world processing-in-memory (PIM) architectures. Described in the PACT 2023 paper by Chen et al. (<https://arxiv.org/pdf/2310.01893.pdf>).

- Readme
- MIT license
- Activity
- Custom properties
- 18 stars
- 6 watching
- 4 forks

Report repository

Releases

No releases published

Packages

No packages published

Contributors 3

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 Chen¹ Juan Gómez-Luna¹ Izzat El Hajj² Yuxin Guo¹ Onur Mutlu¹
¹ETH Zürich ²American University of Beirut

Outline

1

Introduction

2

The Programmability Barrier

3

SimplePIM Overview

Management, Communication & Processing Interfaces

Evaluation Results

4

DaPPA Overview

DaPPA Main Components

Evaluation Results

5

Conclusion

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

1

Introduction

2

The Programmability Barrier

3

SimplePIM Overview

Management, Communication & Processing Interfaces

Evaluation Results

4

DaPPA Overview

DaPPA Main Components

Evaluation Results

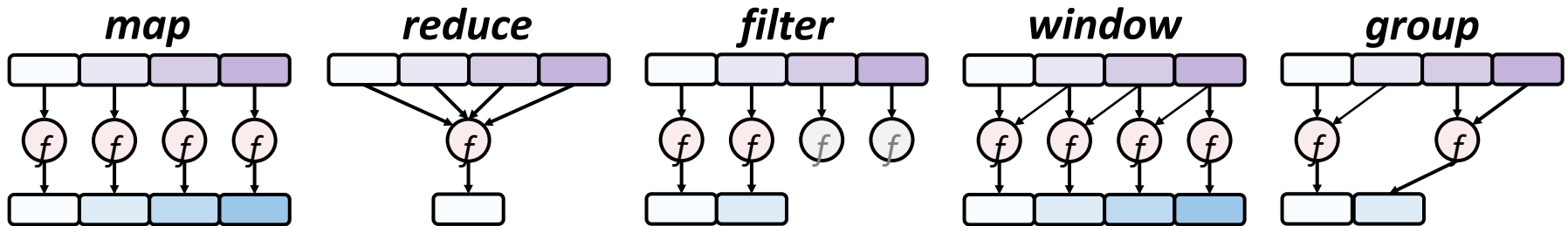
5

Conclusion

DaPPA: Data-Parallel Pattern APIs

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

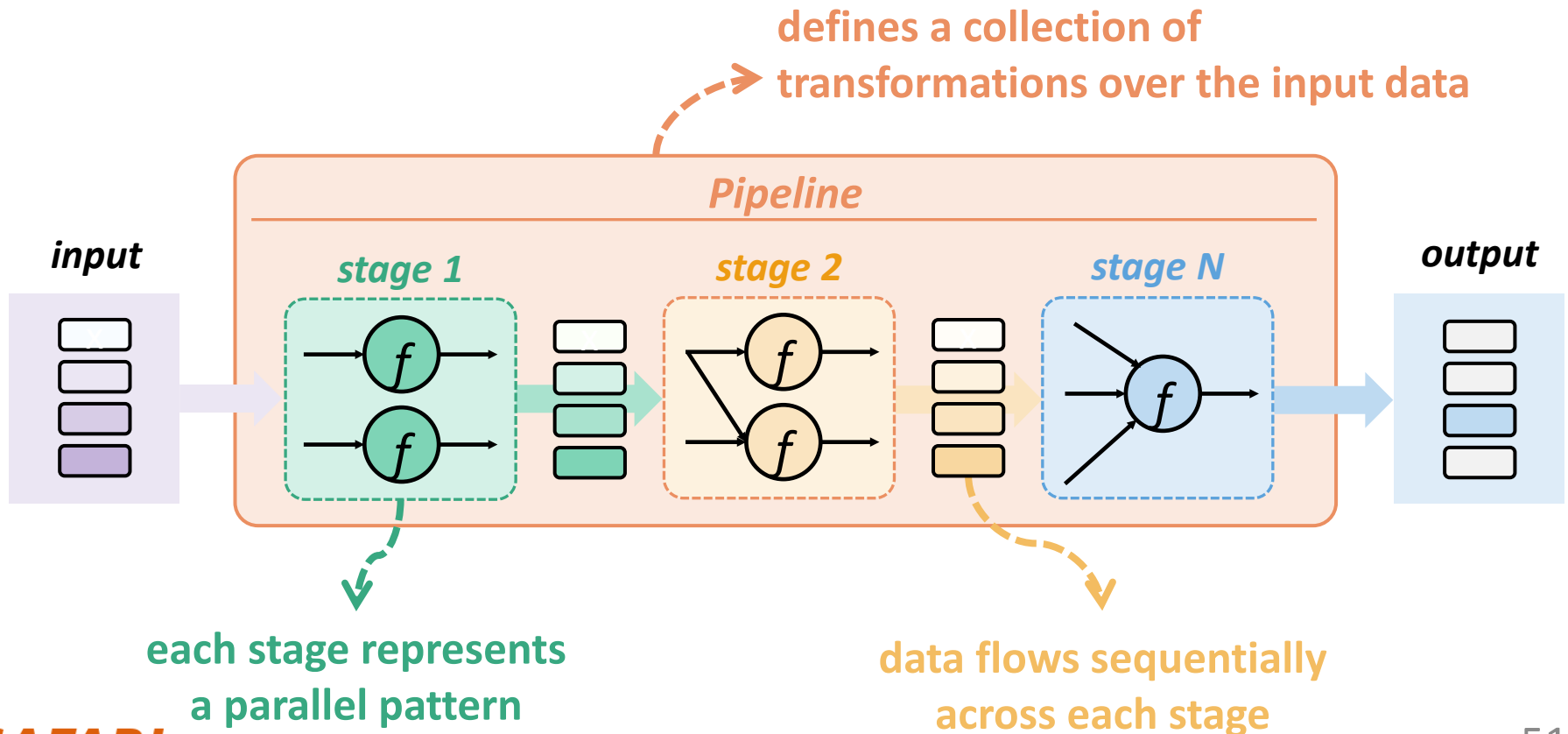
- Skeleton and pattern-based parallel programming are a **common abstraction for parallel architectures**
 - M. Cole, “*Bringing Skeletons Out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming*,” *Parallel Computing*, 2004
- DaPPA supports **five primary** data-parallel patterns



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

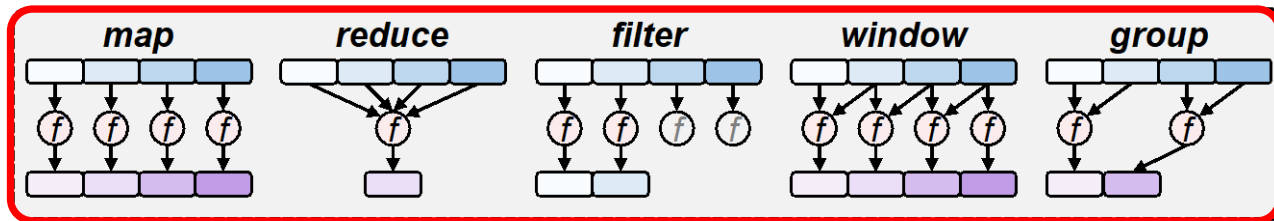
Example of DaPPA's implementation of a
vector dot product operation

reduce
 $C = A_0B_0 +$
map $A_1B_1 +$
 $A_2B_2 +$
 A_3B_3
target
computation

DaPPA: Putting All Together

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

❶ *data-parallel pattern APIs*



reduce

$$C = A_0B_0 +$$

map

$$A_1B_1 +$$
$$A_2B_2 +$$
$$A_3B_3$$

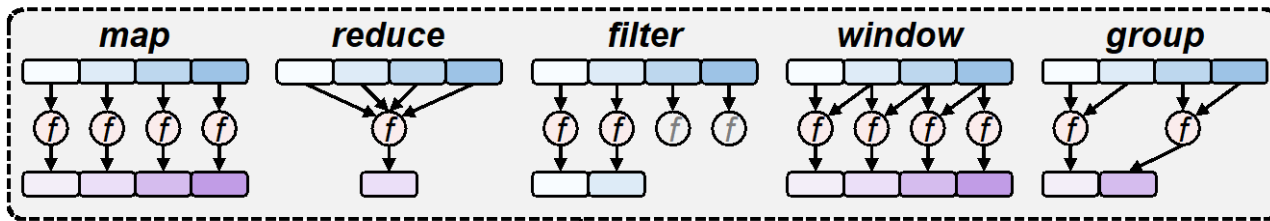
target

computation

DaPPA: Putting All Together

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

❶ data-parallel pattern APIs



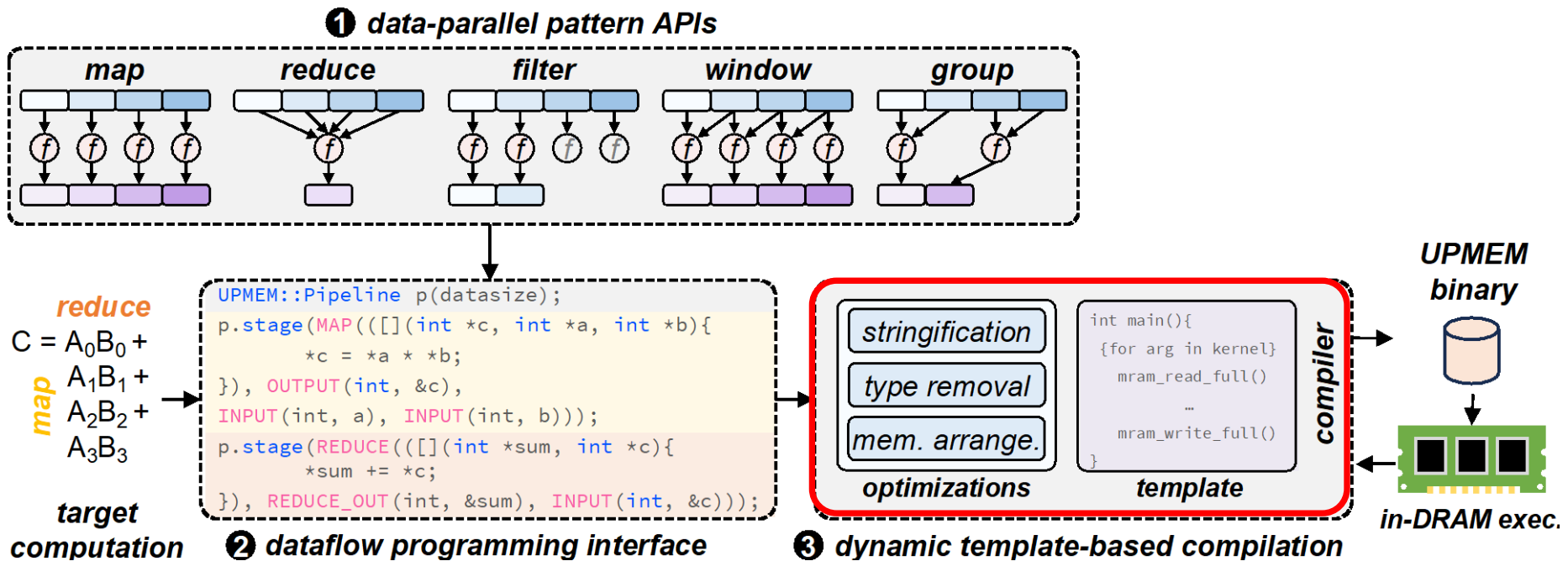
reduce
 $C = A_0B_0 +$
map $A_1B_1 +$
 $A_2B_2 +$
 A_3B_3
target
computation

```
UPMEM::Pipeline p(datasize);  
p.stage(MAP([[int *c, int *a, int *b]{  
    *c = *a * *b;  
}], OUTPUT(int, &c),  
INPUT(int, a), INPUT(int, b)));  
p.stage(REDUCE([[int *sum, int *c]{  
    *sum += *c;  
}], REDUCE_OUT(int, &sum), INPUT(int, &c)));
```

❷ dataflow programming interface

DaPPA: Putting All Together

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



Outline

1

Introduction

2

The Programmability Barrier

3

SimplePIM Overview

Management, Communication & Processing Interfaces

Evaluation Results

4

DaPPA Overview

DaPPA Main Components

Evaluation Results

5

Conclusion

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

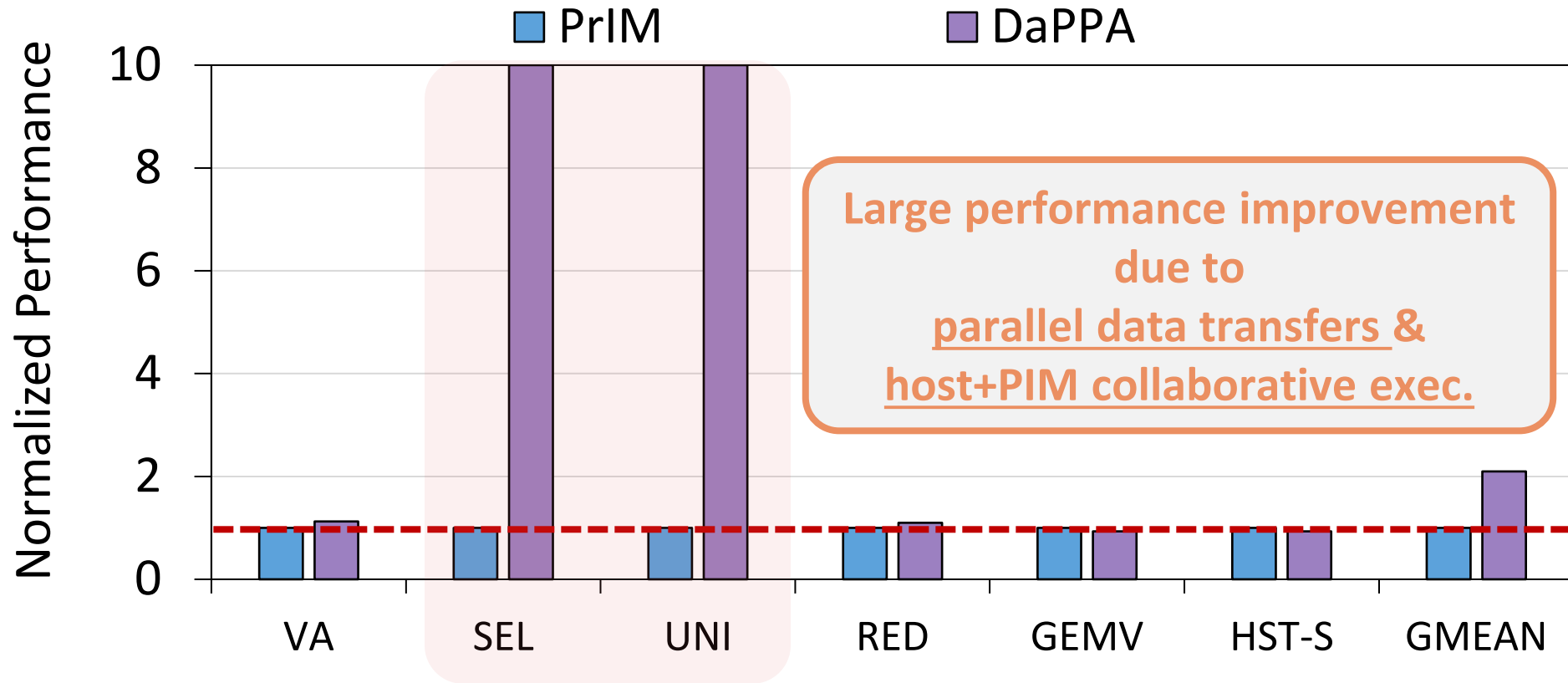
- **Workloads:** 6 workloads from the PRIM benchmark suite

- Vector addition (**VA**); Select (**SEL**); Unique (**UNI**); Reduce (**RED**); General matrix-vector multiply (**GEMV**); Histogram small (**HST-S**)

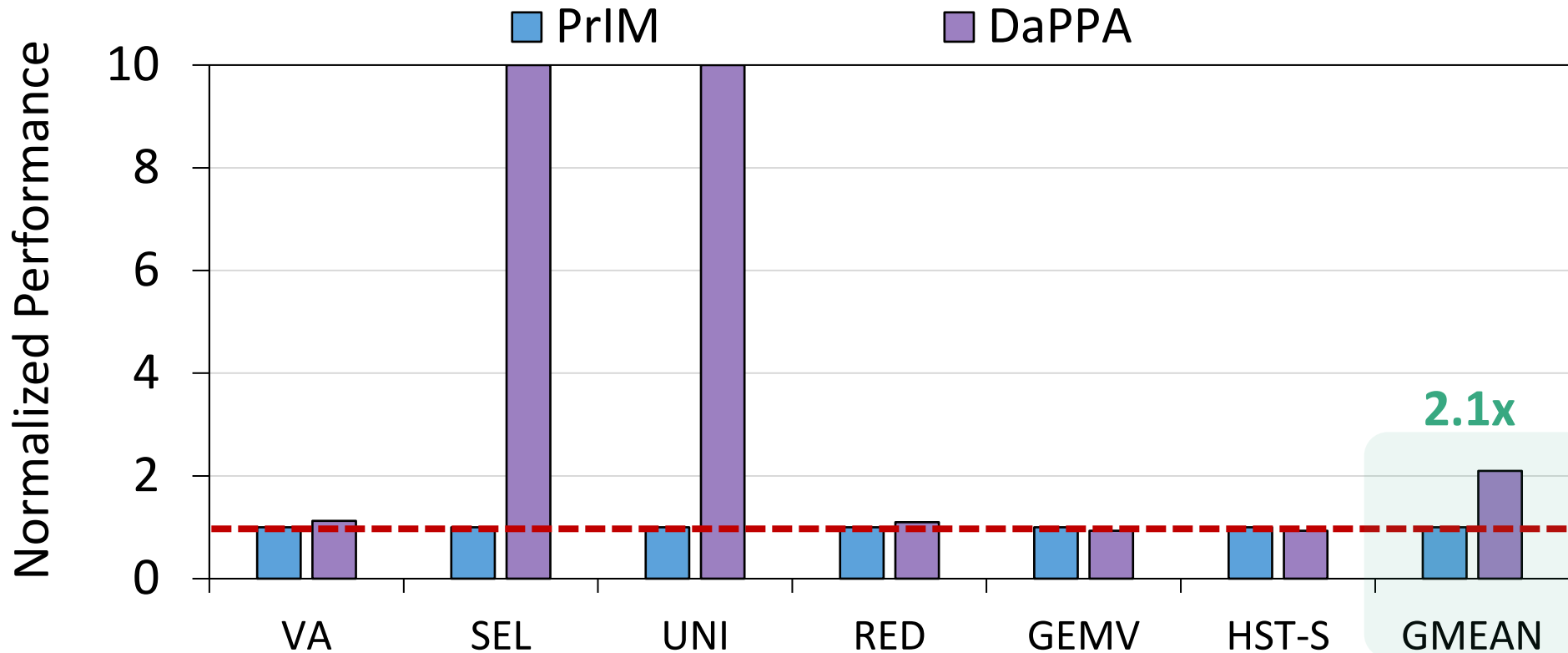
- **Metrics**

- **End-to-end execution time** (average of 10 runs)
- **Programming complexity** (in lines of code)

Evaluation: Performance Analysis

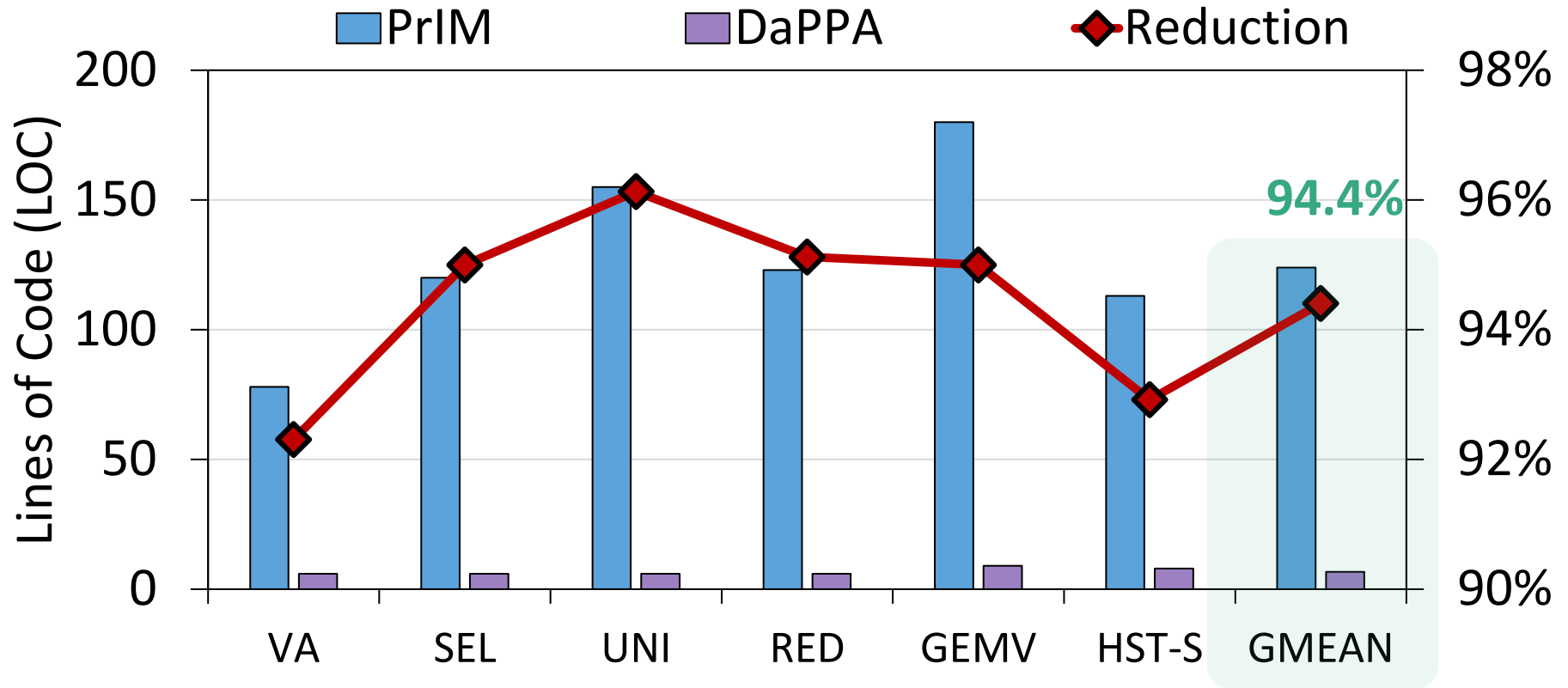


Evaluation: Performance Analysis



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

1

Introduction

2

The Programmability Barrier

3

SimplePIM Overview

Management, Communication & Processing Interfaces

Evaluation Results

4

DaPPA Overview

DaPPA Main Components

Evaluation Results

5

Conclusion

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)

FMS Posters on PIM

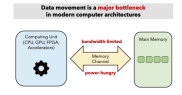
MIMDRAM: An End-to-End Processing-Using-DRAM System for High-Throughput, Energy-Efficient and Programmer-Transparent Multiple-Instruction Multiple-Data Processing

Geraldo F. Oliveira, Ataberk Olgun, Abdullah Giray Yaglikçi, F. Nisa Bostanci, Juan Gómez-Luna, Saugata Ghose, Onur Mutlu



Data Movement Bottleneck

Data movement is a major bottleneck in modern computer architectures

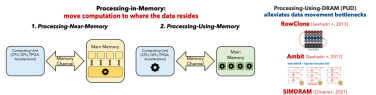


Over 60% of the total system energy is spent on data movement

Processing-in-Memory: Overview & Landscape

Processing-in-Memory: **more computation to where the data resides**

1. Processing-Near-Memory
2. Processing-Using-Memory



Processing-Near-Memory leverages additional logic for computation

Processing-Using-Memory uses memory's operational principles for computation

Limitations of PUD Systems

PUD systems suffer from three sources of inefficiency due to the large and rigid DRAM access granularity


1. SIMD Underutilization
2. Limited Computation Support
3. Challenging Programming Model

Our Goal: Design a flexible PUD system that overcomes the three limitations caused by DRAM's large and rigid access granularity

MIMDRAM: A Hardware/Software Co-Designed PUD System

MIMDRAM is a hardware/software co-designed PUD system that enables fine-grained PUD computation at low cost and programming effort

Key Mechanism: MIMDRAM is a hardware/software co-designed PUD system that enables fine-grained PUD computation at low cost and programming effort



Five-Driven DRAM for PUD Execution

1. Memory access is driven by the user program
2. The user program is driven by the user program
3. The user program is driven by the user program
4. The user program is driven by the user program
5. The user program is driven by the user program

Key Idea


Leverage fine-grained DRAM for processing-using-DRAM

1. Improves SIMD Utilization
2. Enables Low-Cost Interconnects for Reduction of DRAM and CPU Energy
3. Eases Programmability

Software Support

Three new LLVM-based passes targeting PUD execution

1. source code
2. loop re-arrangement
3. cost
4. scheduling
5. code generation



Evaluation

Experimental Setup

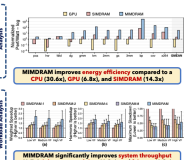
- CPU: Intel Core i7-8650U
- GPU: NVIDIA A100-GPU
- PUD: Processing-Using-Memory (PUD) on DRAM (3.0, 10.0, 20.0)
- Hardware/Software Co-Designed PUD System: MIMDRAM

Workloads

- 12 workloads from Polybench, Rodinia, Phoronix, and SPECint2000
- 48M pre-programmed application mixes

Evaluation

MIMDRAM improves energy efficiency compared to a CPU (30.6x), GPU (4.3x), and SIMDRAM (4.3x)



Conclusion

Problem: Processing-Using-DRAM (PUD) suffers from three limitations caused by DRAM's large and rigid access granularity

Goal: Design a flexible PUD system that overcomes the three limitations caused by DRAM's large and rigid access granularity

Key Mechanism: MIMDRAM, a hardware/software co-designed PUD system

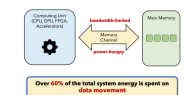
Key Results: MIMDRAM achieves 30.6x, 4.3x, and 4.3x energy efficiency over CPU, GPU, and SIMDRAM, respectively

PUMA: Efficient and Low-Cost Memory Allocation and Alignment Support for Processing-Using-Memory Architectures

Geraldo F. Oliveira, Emanuele G. Esposito, Juan Gómez-Luna, Onur Mutlu

Data Movement Bottleneck

Data movement is a major bottleneck in modern computer architectures

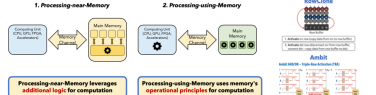


Over 60% of the total system energy is spent on data movement

Processing-in-Memory: Overview & Landscape

Processing-in-Memory: **more computation to where the data resides**

1. Processing-Near-Memory
2. Processing-Using-Memory



Processing-Near-Memory leverages additional logic for computation

Processing-Using-Memory uses memory's operational principles for computation

The Memory Allocation Issue

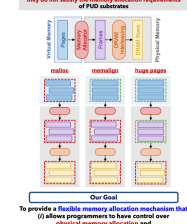
Traditional memory allocators cannot take full advantage of PUD architectures since they do not take the memory allocation requirements of PUD subtrees

Key Idea: Use the internal DRAM mapping information, together with heuristics that are aligned to page address and size and (2) physically contiguous

PUMA memory allocator

```

void* code;
int main() {
    int arr[1024];
    for (int i = 0; i < 1024; i++) {
        arr[i] = i;
    }
    return 0;
}
    
```



Our Goal: To provide a flexible memory allocation mechanism that (1) enables programmer to have control over physical memory allocation and (2) enables PUD execution from the operating system viewpoint.

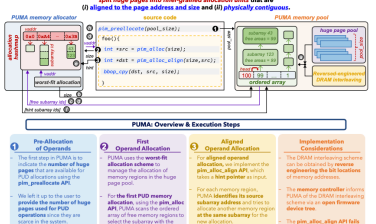
PUMA: Efficient Memory Allocation & Alignment

use the internal DRAM mapping information, together with heuristics that are aligned to page address and size and (2) physically contiguous

Key Idea: Use the internal DRAM mapping information, together with heuristics that are aligned to page address and size and (2) physically contiguous

PUMA Overview & Execution Steps

1. Pre-allocation of Operations
2. First Operational Allocation
3. Aligned Operational Allocation



Our Goal: To provide a flexible memory allocation mechanism that (1) enables programmer to have control over physical memory allocation and (2) enables PUD execution from the operating system viewpoint.

Evaluation

Experimental Setup

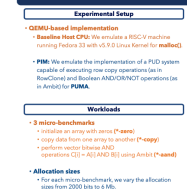
- Baseline Host CPU: We emulate a RISC-V machine running Firefox 33 with v5.0 Linux kernel for macOS
- PUM: We emulate the implementation of a PUD system capable of executing tree-type operations such as RedAnd and BitAnd AND/OR/NOT operations (as in Analytic PUMA)

Workloads

- 3 below benchmarks
- Includes an array with zeros (Zeros)
- Copy data from one array to another (Copy)
- Performs dot product AND
- Operations (0-1) AND (0-1) using And1 (xrand)

Allocation sizes

- For each micro-benchmark, we vary the allocation size from 2000 MiB to 4 MiB.



PUMA can lead to performance gains for large allocation sizes

Conclusion

Background: Processing-in-Memory (PIM) alleviates the performance and energy bottlenecks caused by data

Problem: PUD architectures require a restrictive data layout and alignment for computation

Goal: Design a flexible memory allocation mechanism that allows programs to have control over physical memory allocation, enabling PUD execution

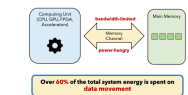
Key Results: Our evaluation shows that PUMA requires no extra hardware and enables workloads to run faster and with less energy

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

Geraldo F. Oliveira, Alain Kohli, Juan Gómez-Luna, Onur Mutlu

Data Movement Bottleneck

Data movement is a major bottleneck in modern computer architectures

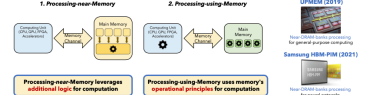


Over 60% of the total system energy is spent on data movement

Processing-in-Memory: Overview & Landscape

Processing-in-Memory: **more computation to where the data resides**

1. Processing-Near-Memory
2. Processing-Using-Memory



Processing-Near-Memory leverages additional logic for computation

Processing-Using-Memory uses memory's operational principles for computation

The Programmability Issue

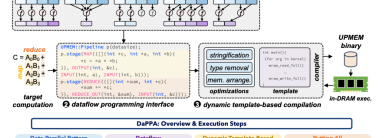
General purpose PIM architectures (e.g., UPMEM) often employ a custom programming interface

Our Goal: To ease programmability for the UPMEM architectures allowing a programmer to write efficient PIM-friendly code without the need to explicitly manage hardware resources.

DaPPA: A Data-Parallel Processing-in-Memory Architecture

DaPPA Overview & Execution Steps

1. Data-Parallel Patterns APIs
2. Dataflow Programming Interface
3. Dynamic Template-Based Compilation
4. Putting All Together



Our Goal: To ease programmability for the UPMEM architectures allowing a programmer to write efficient PIM-friendly code without the need to explicitly manage hardware resources.

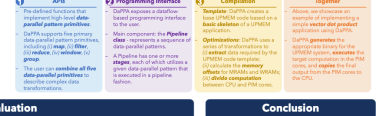
Key Idea

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

DaPPA Overview & Execution Steps

DaPPA Overview & Execution Steps

1. Data-Parallel Patterns APIs
2. Dataflow Programming Interface
3. Dynamic Template-Based Compilation
4. Putting All Together



Evaluation

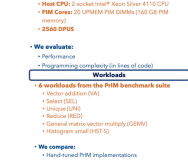
Experimental Setup

- UPMEM system
- Host CPU: Intel Core i7-8650U
- GPU: NVIDIA A100-GPU
- PUD: Processing-Using-Memory (PUD) on DRAM (3.0, 10.0, 20.0)
- Hardware/Software Co-Designed PUD System: MIMDRAM

Workloads

- 6 workloads from the PIM benchmark suite
- Vector addition (ADD)
- Matrix multiplication (MM)
- Matrix-vector multiplication (MV)
- Histogram small (HIST-S)

Our Goal: To ease programmability for the UPMEM architectures allowing a programmer to write efficient PIM-friendly code without the need to explicitly manage hardware resources.



DaPPA significantly improves speed and performance compared to hand-coded implementations

Conclusion

Background: Processing-in-Memory (PIM) alleviates the performance and energy bottlenecks caused by data

Problem: Programming the UPMEM system requires non-trivial effort from the programmer

Goal: Design a programming framework for the UPMEM architectures allowing one to write PIM code without prior knowledge of the hardware

Key Results: Our evaluation shows that DaPPA requires no extra hardware and enables workloads to run faster and with less energy

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

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

Geraldo F. Oliveira

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

Onur Mutlu