2nd Workshop on Memory-Centric Computing:

Processing-Using-Memory - Part II

Dr. Geraldo F. Oliveira

https://geraldofojunior.github.io

ICS 2025

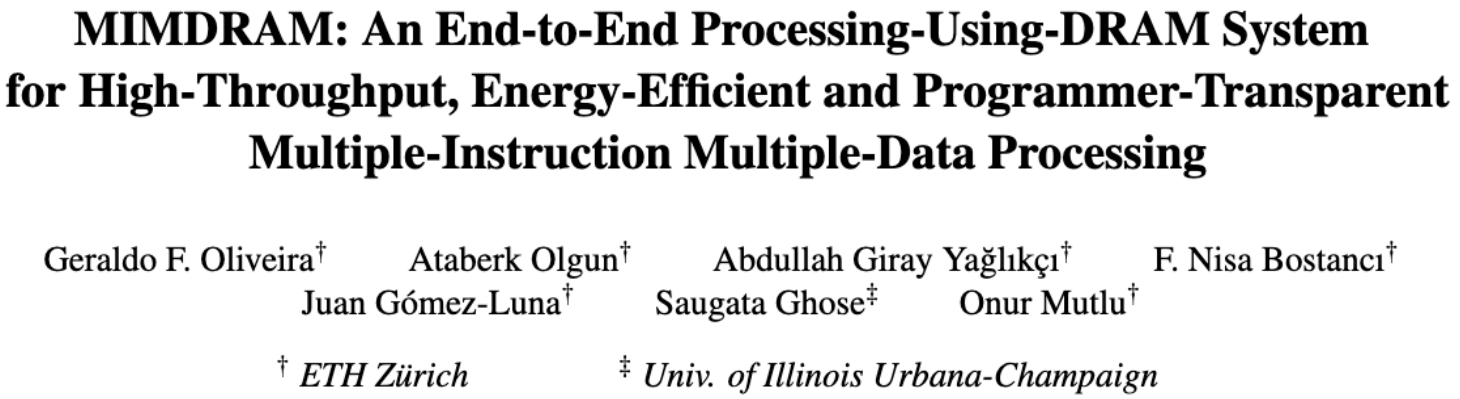
08 June 2025

SAFARI

ETH zürich

SIMDRAM Follow-Up: MIMDRAM

- Geraldo F. Oliveira, Ataberk Olgun, Abdullah Giray Yağlıkçı, F. Nisa Bostancı, Juan Gómez-Luna, Saugata Ghose, and Onur Mutlu **"MIMDRAM: An End-to-End Processing-Using-DRAM System for High-Throughput, Energy-Efficient and Programmer-Transparent Multiple-Instruction Multiple-Data Computing"**
  *Proceedings of the 30th International Symposium on High-Performance Computer Architecture (**HPCA**)*, Edinburgh, Scotland, March 2024.



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 Yağlıkçı† F. Nisa Bostancı†
Juan Gómez-Luna† Saugata Ghose‡ Onur Mutlu†

† *ETH Zürich* ‡ *Univ. of Illinois Urbana-Champaign*

https://arxiv.org/pdf/2402.19080.pdf

Limitations of PUD Systems:
Overview

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

1 SIMD Underutilization
- due to data parallelism variation within and across applications
- leads to throughput and energy waste

2 Limited Computation Support
- due to a lack of low-cost interconnects across columns
- limits PUD operations to only parallel map constructs

3 Challenging Programming Model
- due to a lack of compiler support for PUD systems
- creates a burden on programmers, limiting PUD adoption

Limitations of PUD Systems:
Overview

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

1 SIMD Underutilization
- due to data parallelism variation within and across applications
- leads to throughput and energy waste

2 Limited Computation Support
- due to a lack of low-cost interconnects across columns
- limits PUD operations to only parallel map constructs

3 Challenging Programming Model
- due to a lack of compiler support for PUD systems
- creates a burden on programmers, limiting PUD adoption

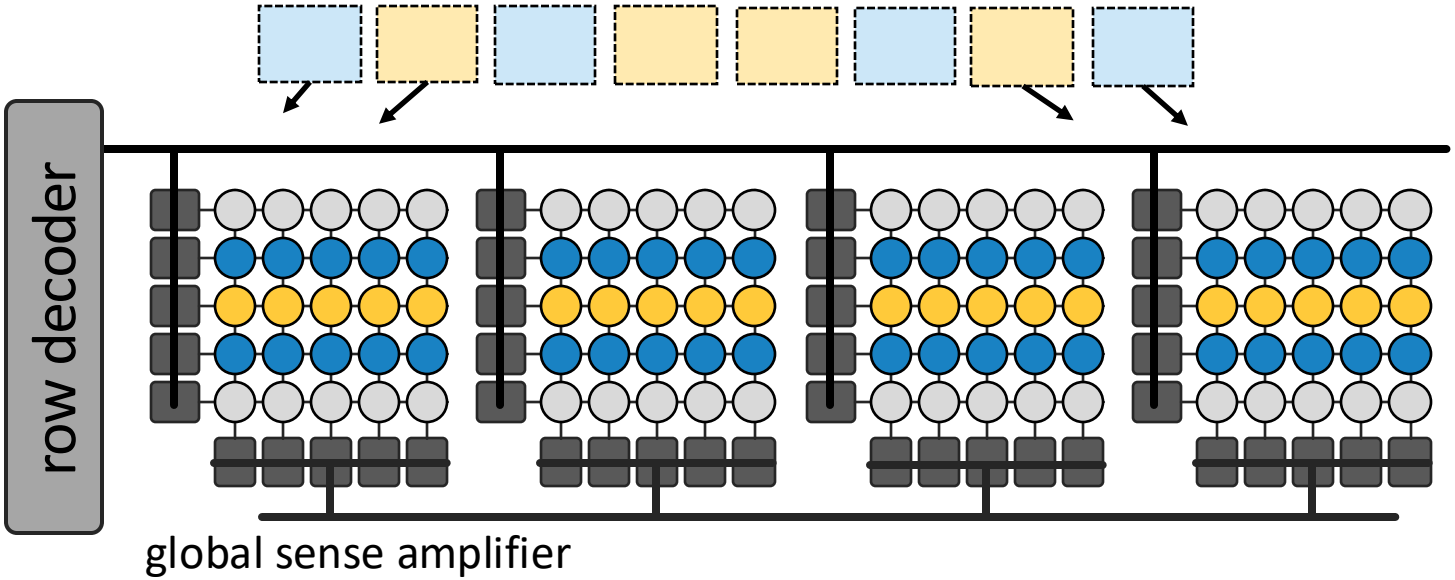
Limitations of PUD Systems:
Underutilization of SIMD Lanes (I)

Application Analysis:

quantify the fraction of SIMD parallelism in real applications

Maximum Vectorization Factor:

maximum number of scalar operands that fit into a SIMD register

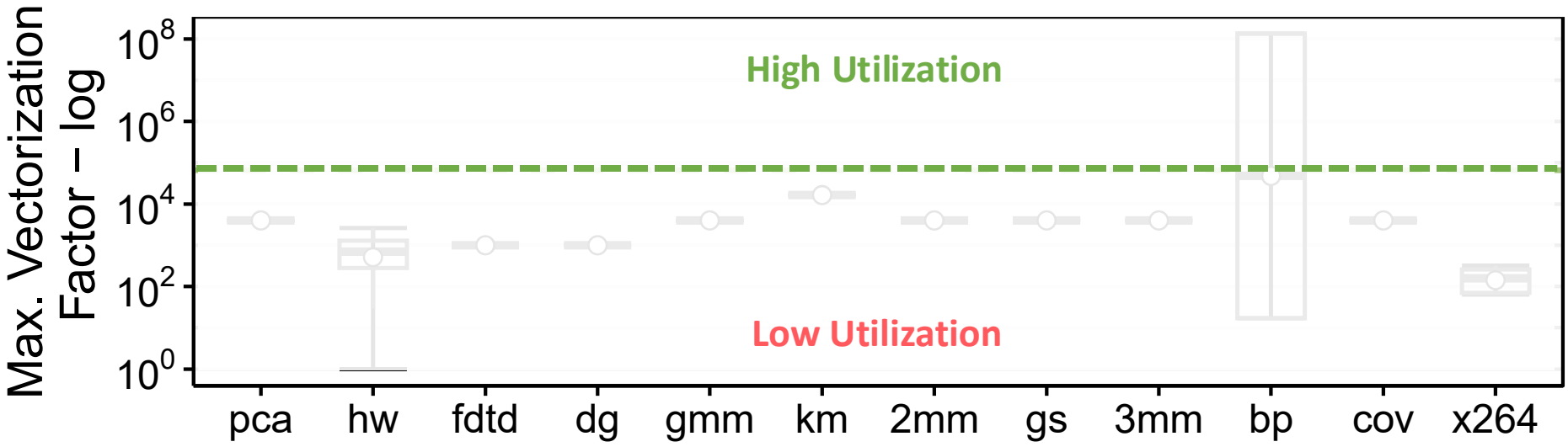


Ideal maximum vectorization factor = # DRAM columns (e.g., 65,536)

Limitations of PUD Systems:
Underutilization of SIMD Lanes (II)

Application Analysis:

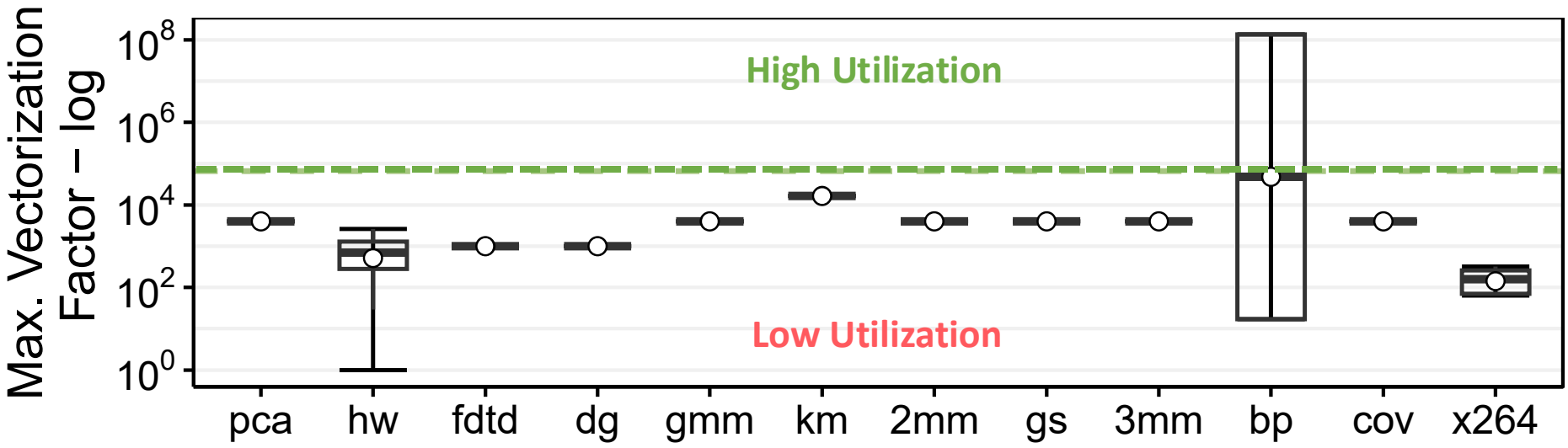
quantify the fraction of SIMD parallelism in real applications



Limitations of PUD Systems:
Underutilization of SIMD Lanes (II)

Application Analysis:

quantify the fraction of SIMD parallelism in real applications



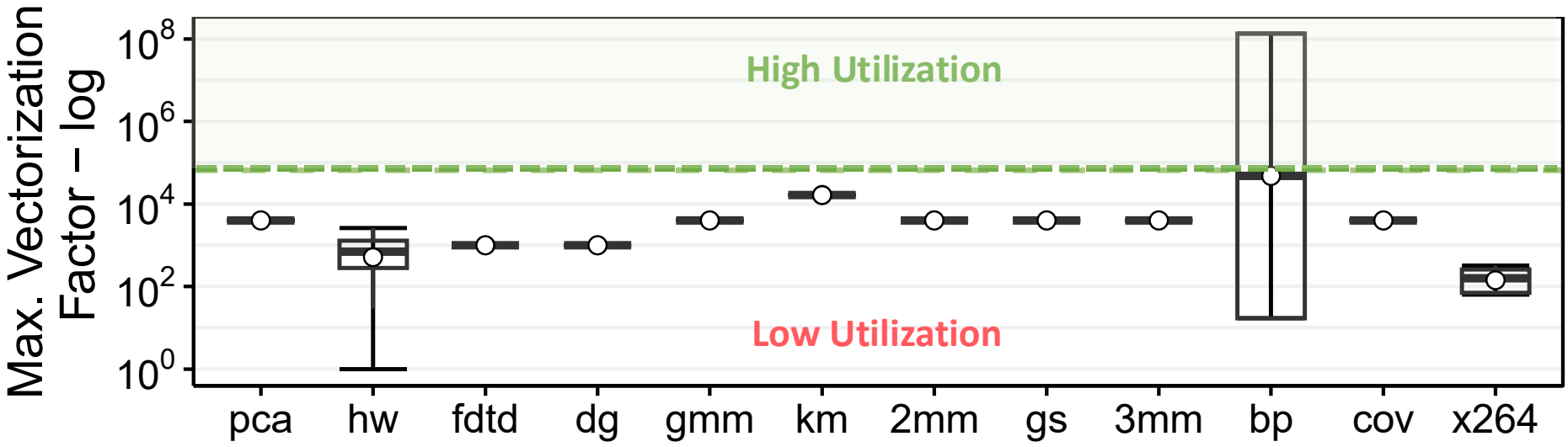
Takeaway

SIMD parallelism significantly varies within a single application and across different applications

Limitations of PUD Systems:
Underutilization of SIMD Lanes (III)

Application Analysis:

quantify the fraction of SIMD parallelism in real applications



Takeaway

A small fraction of vectorized loops have a large enough maximum vectorization factor to fully exploit the SIMD parallelism of PUD systems

Limitations of PUD Systems:
Overview

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

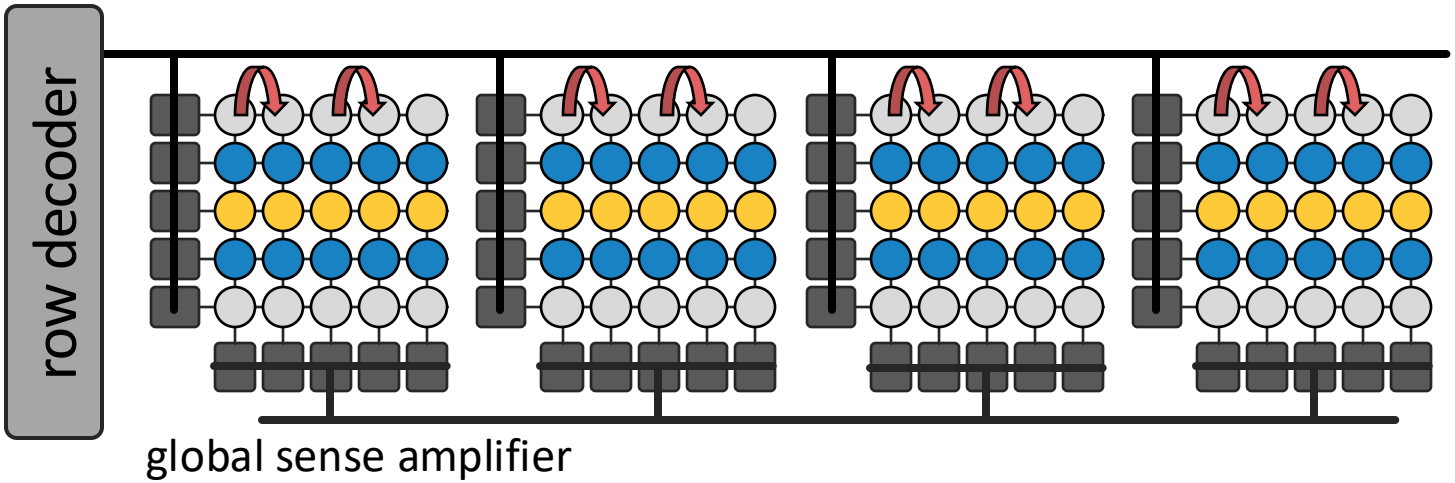
1 **SIMD Underutilization**
- due to data parallelism variation within and across applications
- leads to throughput and energy waste

2 **Limited Computation Support**
- due to a lack of low-cost interconnects across columns
- limits PUD operations to only parallel map constructs

3 **Challenging Programming Model**
- due to a lack of compiler support for PUD systems
- creates a burden on programmers, limiting PUD adoption

Limitations of PUD Systems:
Limited Computation Support

PUD systems do not support vector reduction at low area cost since data movement is bounded to within a DRAM column



no direct communication path across columns

Takeaway

Directly connecting all DRAM columns using a custom all-to-all interconnect leads to large (i.e., 21%) area cost

Limitations of PUD Systems:
Overview

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

1 SIMD Underutilization
- due to data parallelism variation within and across applications
- leads to throughput and energy waste

2 Limited Computation Support
- due to a lack of low-cost interconnects across columns
- limits PUD operations to only parallel map constructs

3 Challenging Programming Model
- due to a lack of compiler support for PUD systems
- creates a burden on programmers, limiting PUD adoption

Limitations of PUD Systems:
Challenging Programming Model

Programmer's Tasks:

Goal:

Just write my kernel

High-level code for

C[i] = (A[i] > pred[i])? A[i] + B[i] : A[i] – B[i]

```
for (int i = 0; i < size ; ++ i){
    bool cond = A[i] > pred[i];
    if (cond) C[i] = A[i] + B[i];
    else  C[i] = A[i] - B[i];
}
```

Limitations of PUD Systems:
Challenging Programming Model

Programmer's Tasks:

Map & align data structures

Goal:

Just write my kernel

High-level code for

C[i] = (A[i] > pred[i])? A[i] + B[i] : A[i] – B[i]

```
for (int i = 0; i < size ; ++ i){
   bool cond = A[i] > pred[i];
   if (cond) C[i] = A[i] + B[i];
   else  C[i] = A[i] - B[i];
}
```

Limitations of PUD Systems:
Challenging Programming Model

Programmer's Tasks:

- Map & align data structures
- Identify array boundaries

Goal:

- Just write my kernel

High-level code for

C[i] = (A[i] > pred[i])? A[i] + B[i] : A[i] – B[i]

```
for (int i = 0; i < size ; ++ i){
   bool cond = A[i] > pred[i];
   if (cond) C[i] = A[i] + B[i];
   else  C[i] = A[i] - B[i];
}
```

Limitations of PUD Systems:
Challenging Programming Model

Programmer's Tasks:

Map & align data structures	Identify array boundaries	Manually unroll loop	Map C to PUD instructions

Goal:

Just write my kernel

High-level code for

C[i] = (A[i] > pred[i])? A[i] + B[i] : A[i] – B[i]

```
for (int i = 0; i < size ; ++ i){
    bool cond = A[i] > pred[i];
    if (cond) C[i] = A[i] + B[i];
    else  C[i] = A[i] - B[i];
}
```

Limitations of PUD Systems:
Challenging Programming Model

Programmer's Tasks:

Map & align data structures	Identify array boundaries	Manually unroll loop	Map C to PUD instructions	Orchestrate data movement

Goal: Just write my kernel

High-level code for

C[i] = (A[i] > pred[i])? A[i] + B[i] : A[i] – B[i]

```
for (int i = 0; i < size ; ++ i){
   bool cond = A[i] > pred[i];
   if (cond) C[i] = A[i] + B[i];
   else  C[i] = A[i] - B[i];
}
```

Limitations of PUD Systems:
Challenging Programming Model

Programmer's Tasks:

Map & align data structures	Identify array boundaries	Manually unroll loop	Map C to PUD instructions	Orchestrate data movement	**Goal:** Just write my kernel

PUD's assembly-like code for

C[i] = (A[i] > pred[i])? A[i] + B[i] : A[i] – B[i]

```
bbop_trsp_init(A , size , elm_size);
bbop_trsp_init(B , size , elm_size);
bbop_trsp_init(C , size , elm_size);

bbop_add(D , A , B , size , elm_size);
bbop_sub(E , A , B , size , elm_size);
bbop_greater(F , A , pred , size , elm_size);
bbop_if_else(C , D , E , F , size , elm_size);
```

Problem & Goal

Problem

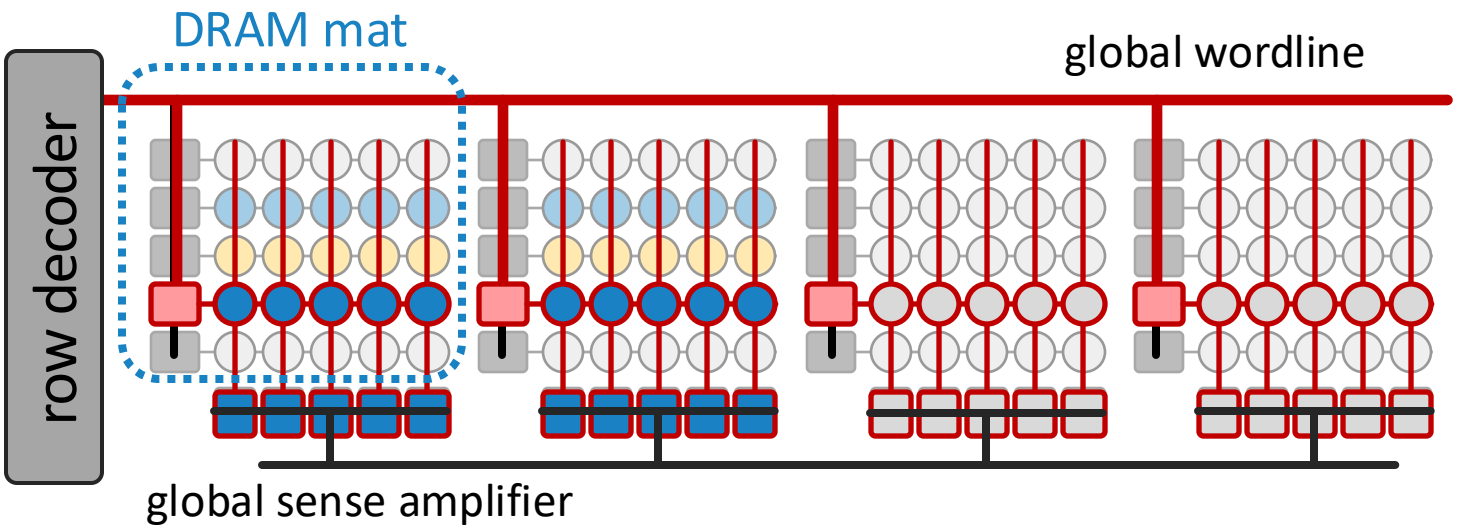
Processing-Using-DRAM's large and rigid granularity limits its applicability and efficiency for different applications

Goal

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

MIMDRAM:
Key Idea (I)

DRAM's hierarchical organization can enable <u>fine-grained access</u>



Key Issue:

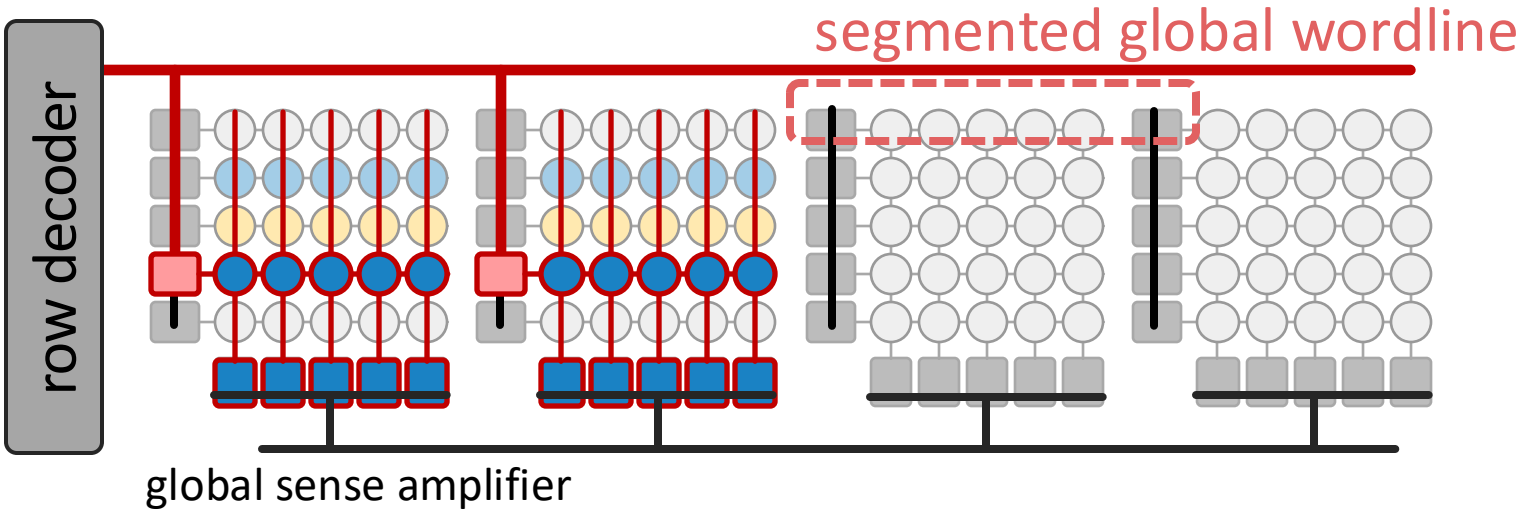
on a DRAM access, the global wordline propagates across all DRAM mats

Fine-Grained DRAM:

segments the global wordline to access individual DRAM mats

MIMDRAM:
Key Idea (II)

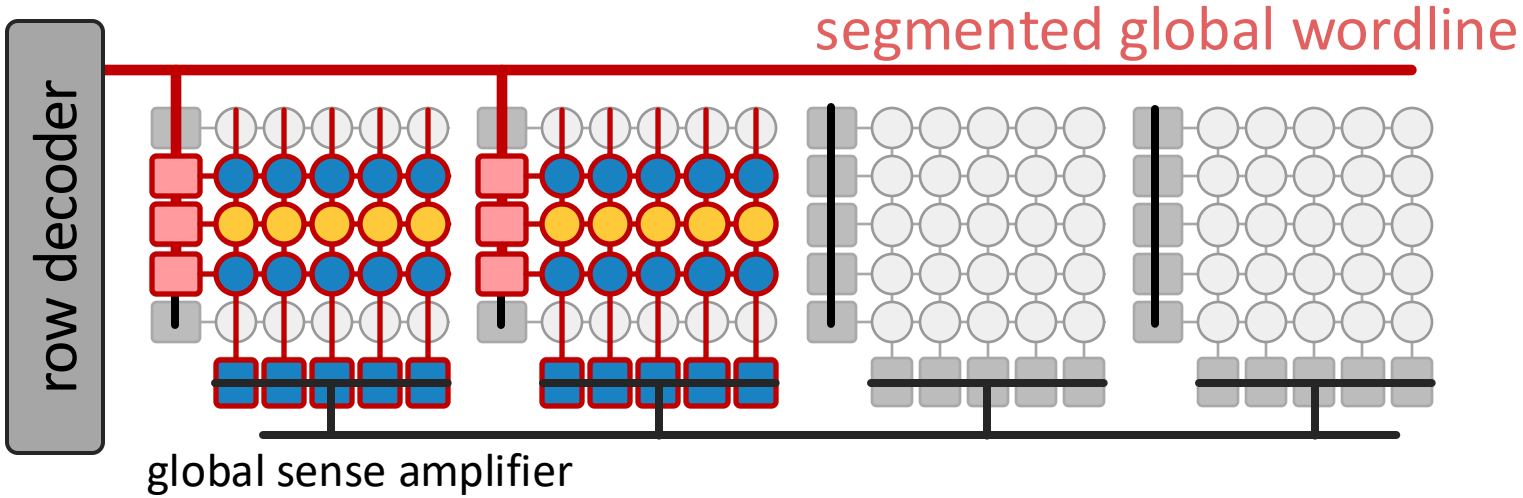
Fine-Grained DRAM:
segments the global wordline to access individual DRAM mats



Fine-grained DRAM for energy-efficient DRAM access:

[Cooper-Balis+, 2010]: Fine-Grained Activation for Power Reduction in DRAM
[Udipi+, 2010]: Rethinking DRAM Design and Organization for Energy-Constrained Multi-Cores
[Zhang+, 2014]: Half-DRAM
[Ha+, 2016]: Improving Energy Efficiency of DRAM by Exploiting Half Page Row Access
[O'Connor+, 2017]: Fine-Grained DRAM
[Olgun+, 2024]: Sectored DRAM

MIMDRAM:
Key Idea (III)

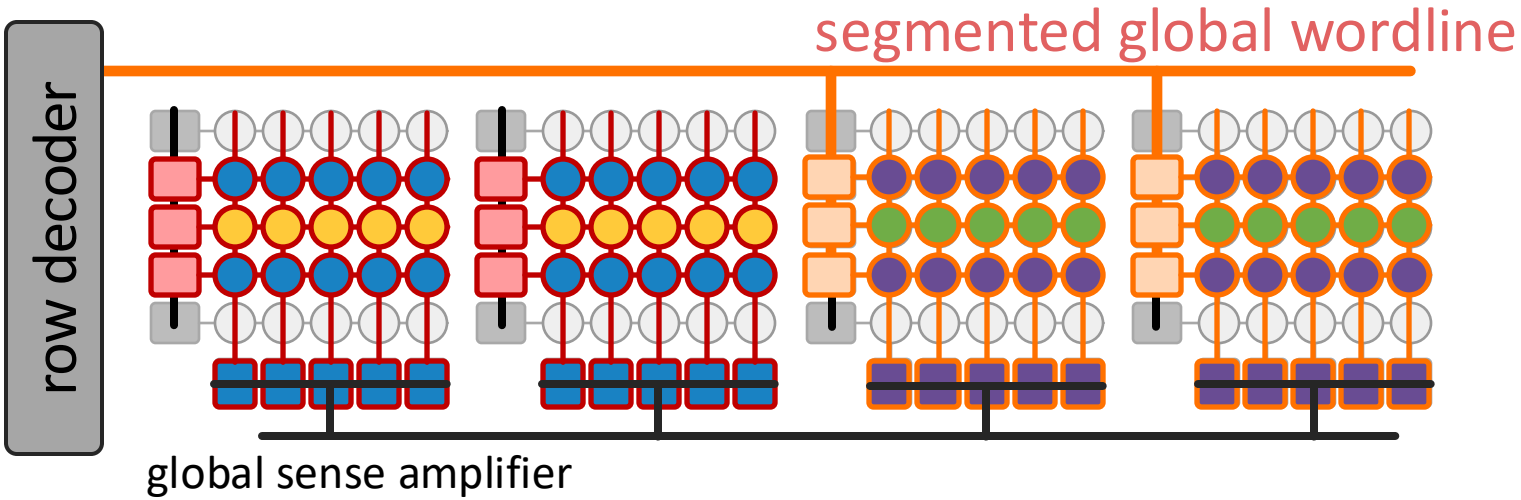


Fine-grained DRAM for processing-using-DRAM:

1 Improves SIMD utilization

- for a single PUD operation, only access the DRAM mats with target data

MIMDRAM:
Key Idea (III)

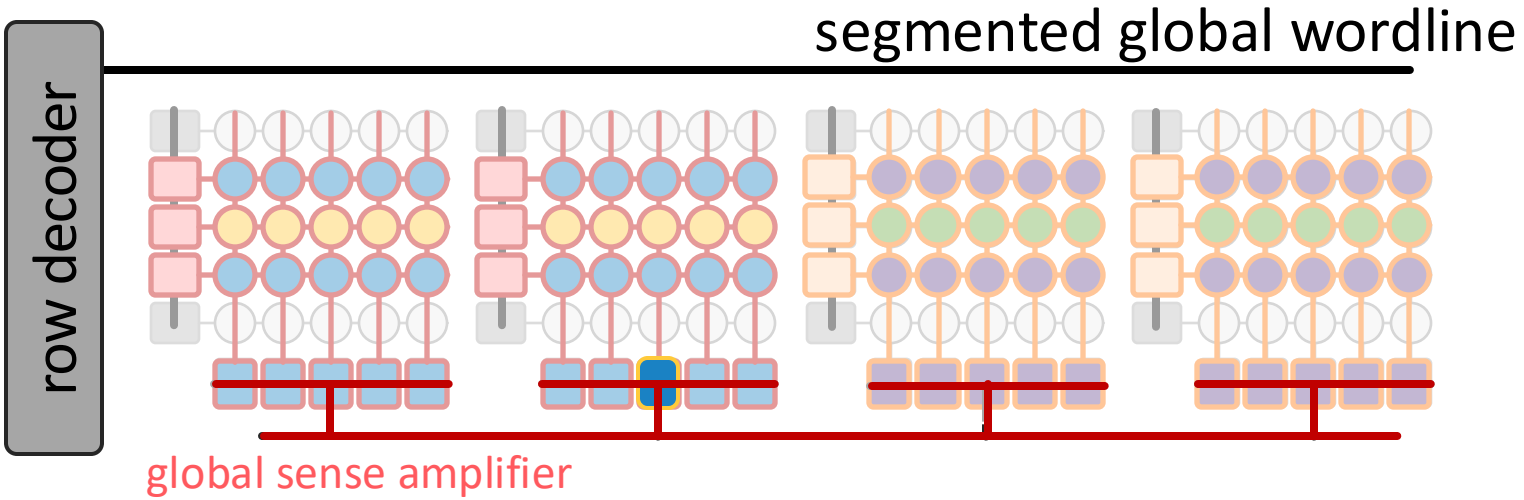


Fine-grained DRAM for processing-using-DRAM:

1 Improves SIMD utilization

- for a single PUD operation, only access the DRAM mats with target data
- for multiple PUD operations, execute independent operations concurrently
  → **multiple instruction, multiple data (MIMD) execution model**

MIMDRAM:
Key Idea (III)



Fine-grained DRAM for processing-using-DRAM:

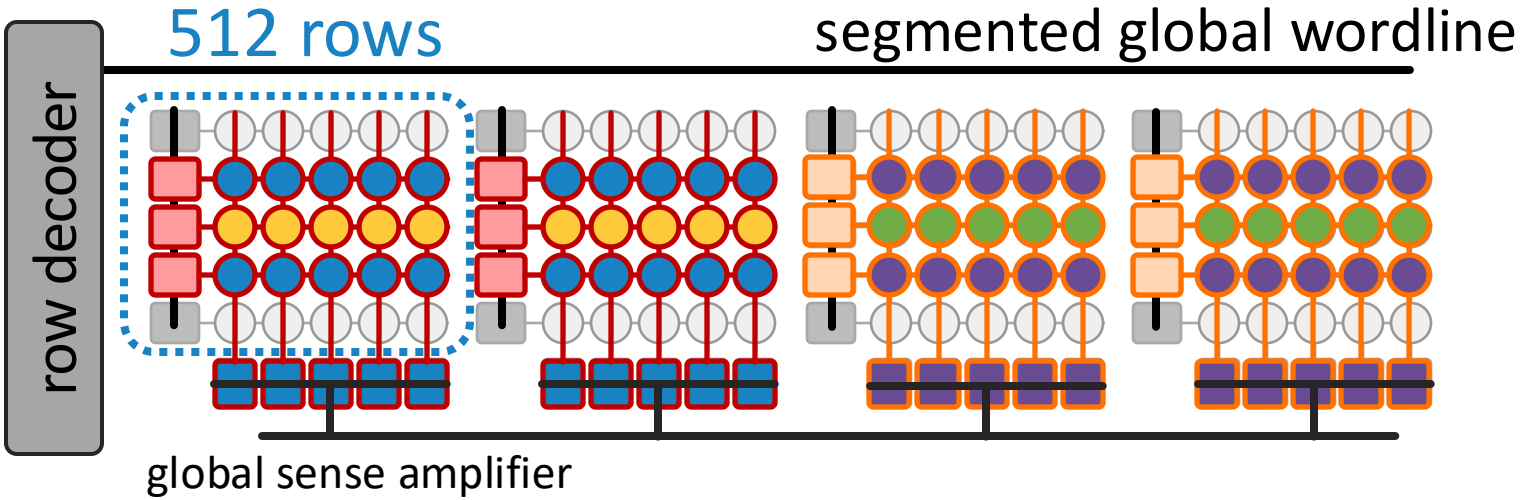
1 Improves SIMD utilization

- for a single PUD operation, only access the DRAM mats with target data
- for multiple PUD operations, execute independent operations concurrently
  → multiple instruction, multiple data (MIMD) execution model

2 Enables low-cost interconnects for vector reduction

- global and local data buses can be used for inter-/intra-mat communication

MIMDRAM:
Key Idea (III)



Fine-grained DRAM for processing-using-DRAM:

1 Improves SIMD utilization
- for a single PUD operation, only access the DRAM mats with target data
- for multiple PUD operations, execute independent operations concurrently
  → **multiple instruction, multiple data (MIMD) execution model**

2 Enables low-cost interconnects for vector reduction
- global and local data buses can be used for inter-/intra-mat communication

3 Eases programmability
- SIMD parallelism in a DRAM mat is on par with vector ISAs' SIMD width

MIMDRAM:
Overview

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

Main components of MIMDRAM:

1 Hardware

- DRAM array modification to enable fine-grained PUD computation
- inter- and intra-mat interconnects to enable PUD vector reduction
- control unit design to orchestrate PUD execution

2 Software

- compiler support to transparently generate PUD instructions
- system support to map and execute PUD instructions

MIMDRAM:
Overview

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

Main components of MIMDRAM:

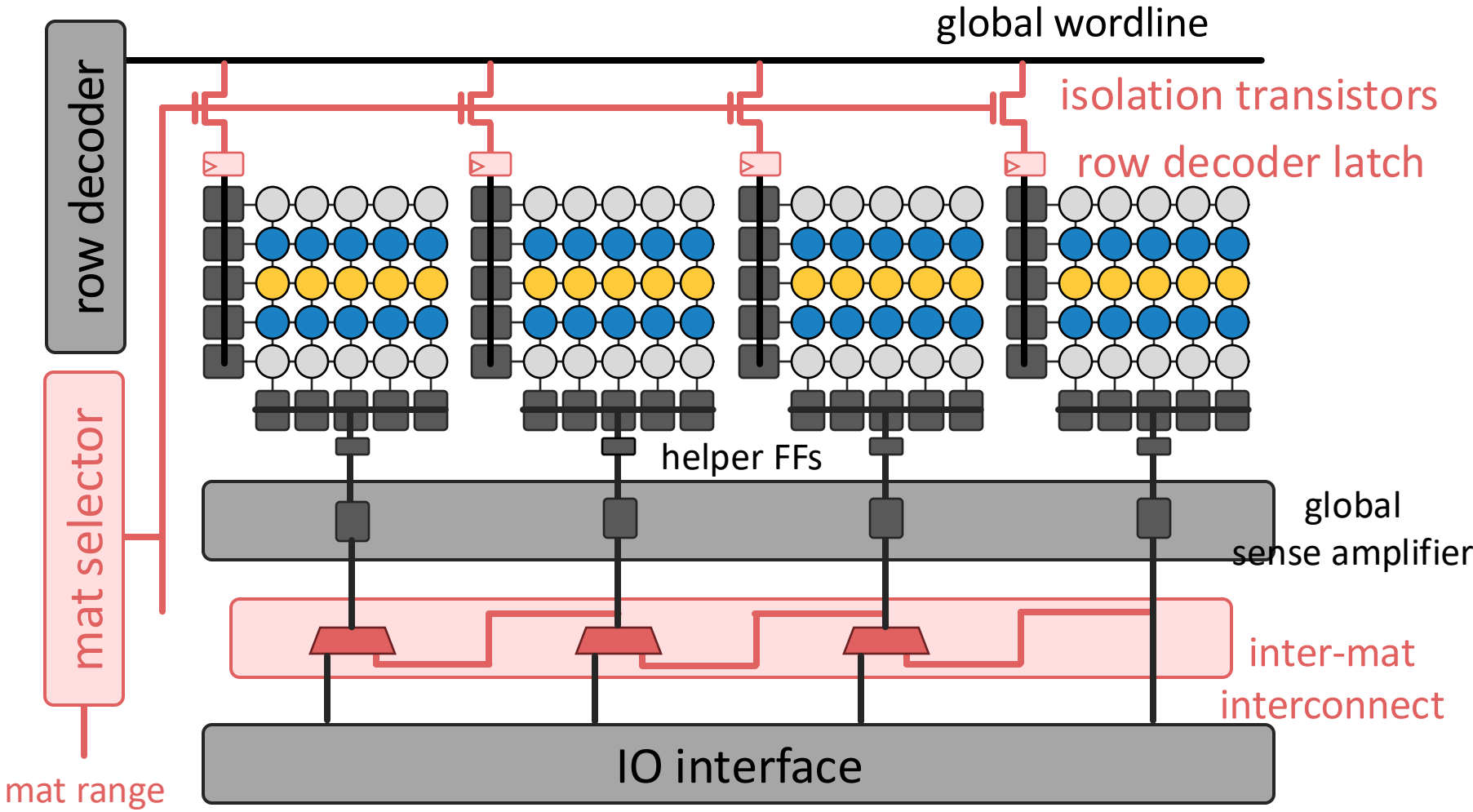
1 Hardware

- **DRAM array modification to enable fine-grained PUD computation**
- **inter- and intra-mat interconnects to enable PUD vector reduction**
- **control unit design to orchestrate PUD execution**

2 Software

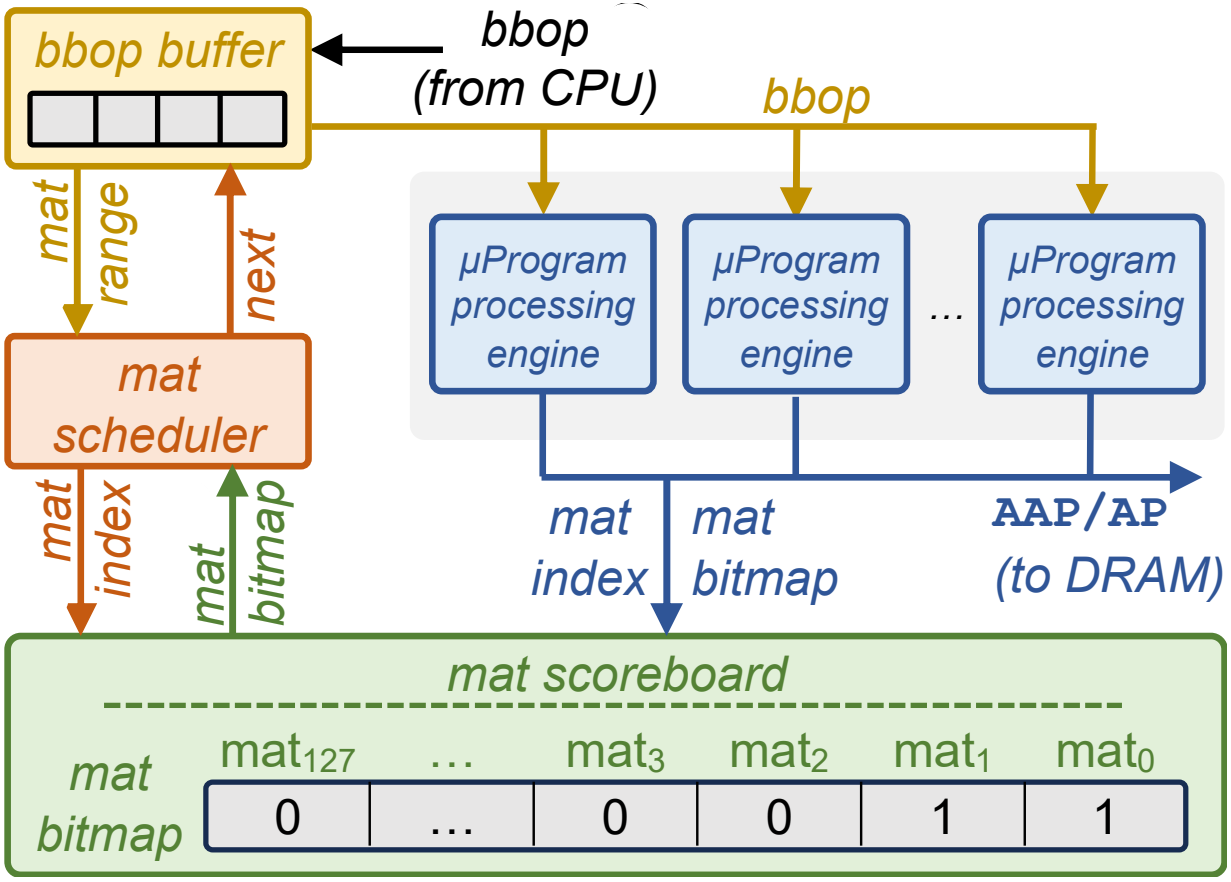
- new compiler support to transparently generate PUD instructions
- system support to map and execute PUD instructions

MIMDRAM:
Modifications to DRAM Chip



MIMDRAM:
Control Unit Design

The control unit schedules and orchestrates the execution of multiple PUD operations transparently



MIMDRAM:
Overview

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

Main components of MIMDRAM:

1 Hardware

- DRAM array modification to enable fine-grained PUD computation
- inter- and intra-mat interconnects to enable PUD vector reduction
- control unit design to orchestrate PUD execution

2 Software

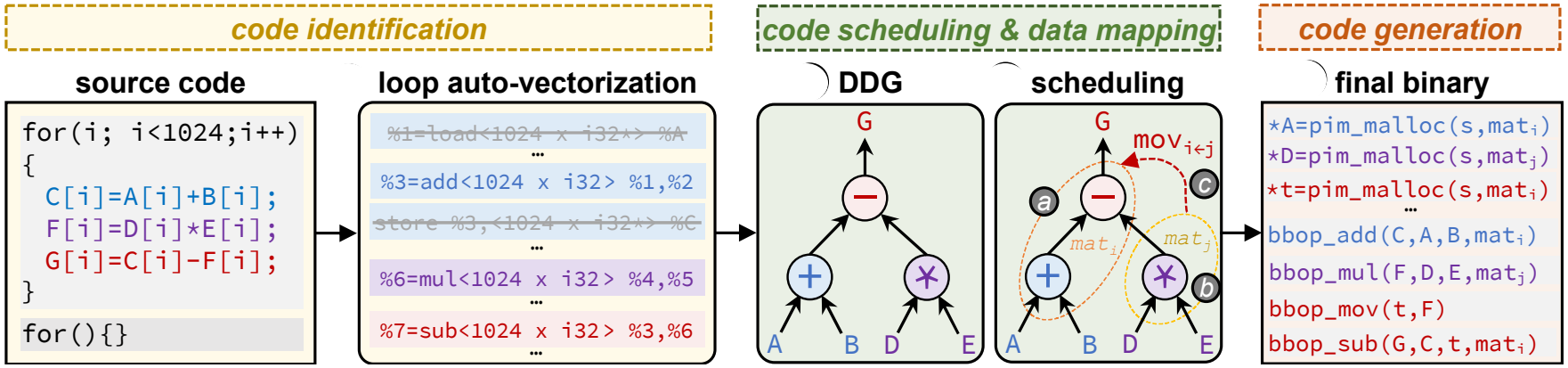
- **new compiler support to transparently generate PUD instructions**
- **system support to map and execute PUD instructions**

MIMDRAM:
Compiler Support (I)

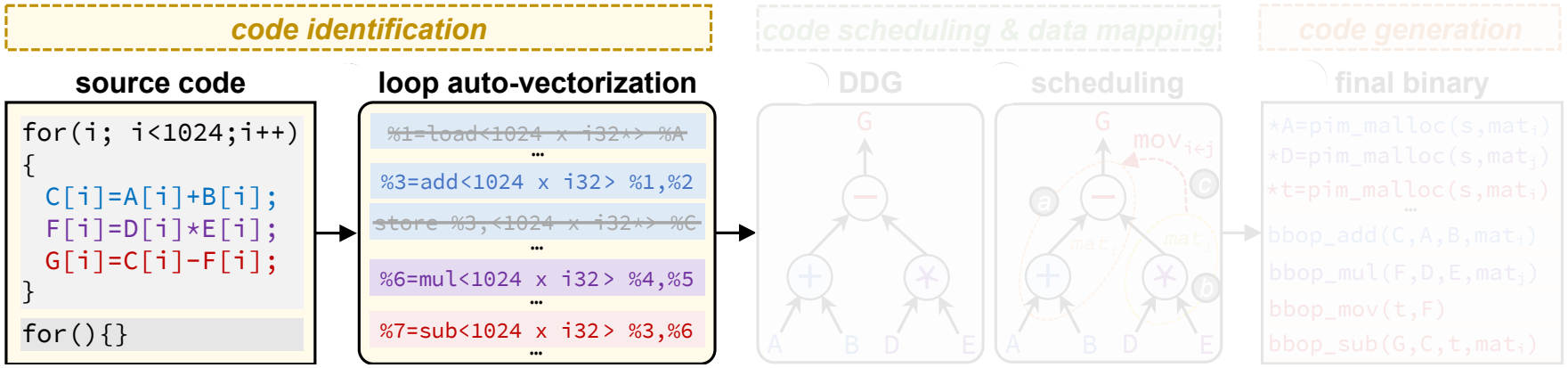
Goal

Transparently:
extract SIMD parallelism from an application, and
schedule PUD instructions while maximizing utilization

Three new LLVM-based passes targeting PUD execution



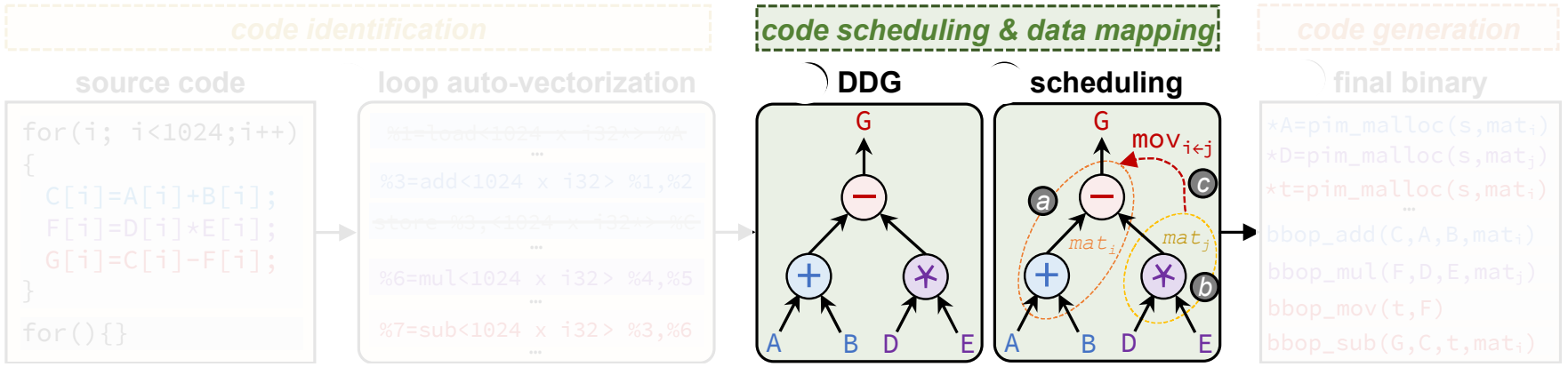
MIMDRAM:
Compiler Support (II)



Goal

Identify SIMD parallelism, generate PUD instructions, and set the appropriate vectorization factor

MIMDRAM:
Compiler Support (II)



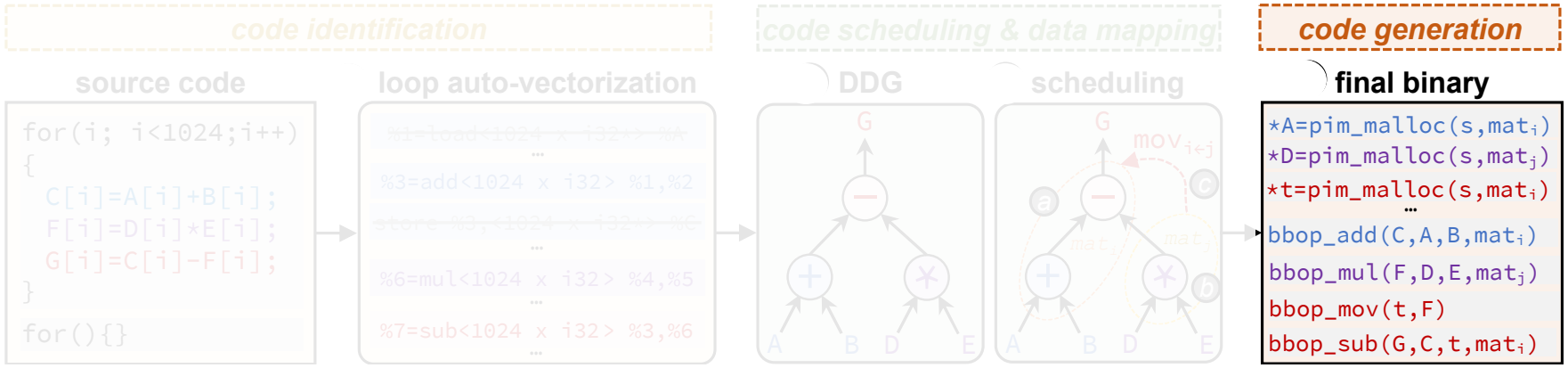
Goal

Identify SIMD parallelism, generate PUD instructions, and set the appropriate vectorization factor

Goal

Improve SIMD utilization by allowing the distribution of independent PUD instructions across DRAM mats

MIMDRAM:
Compiler Support (III)



Goal

Identify SIMD parallelism, generate PUD instructions, and set the appropriate vectorization factor

Goal

Improve SIMD utilization by allowing the distribution of independent PUD instructions across DRAM mats

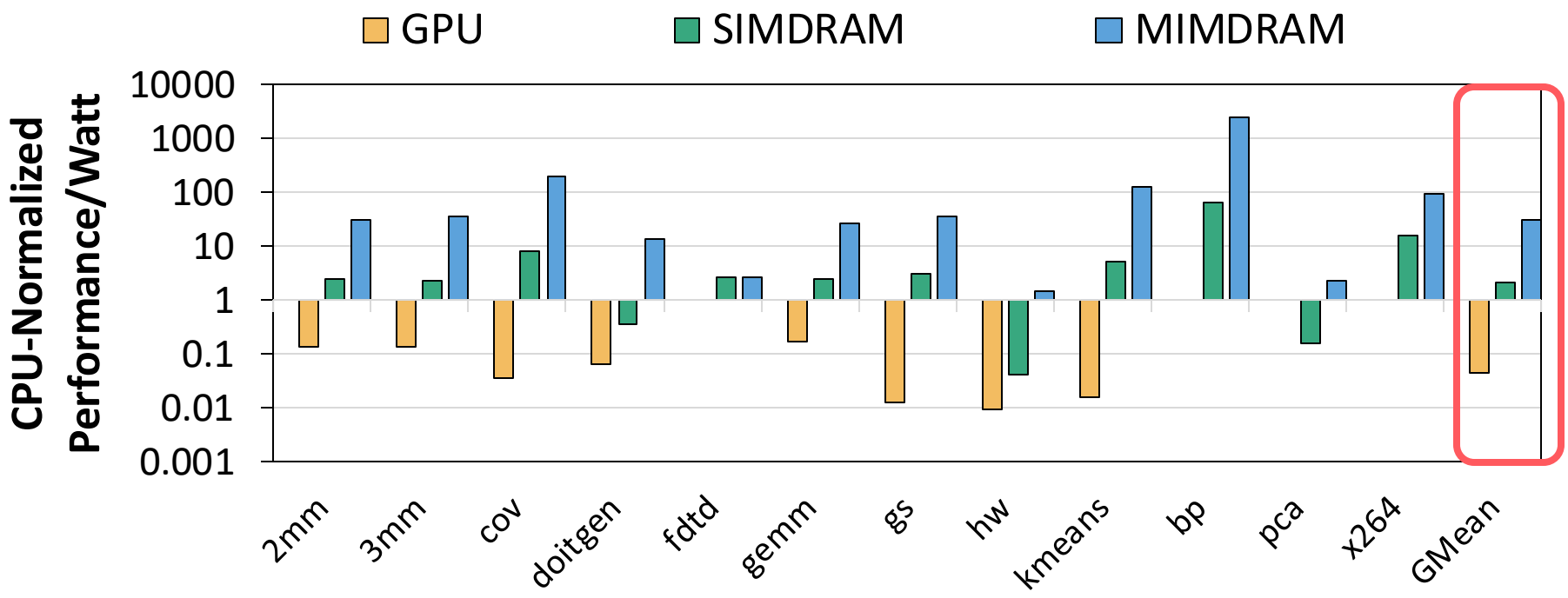
Goal

Generate the appropriate binary for data allocation and PUD instructions

MIMDRAM:
System Support

- Instruction set architecture
- Execution & data transposition
- Data coherence
- Address translation
- Data allocation & alignment
- Mat label translation

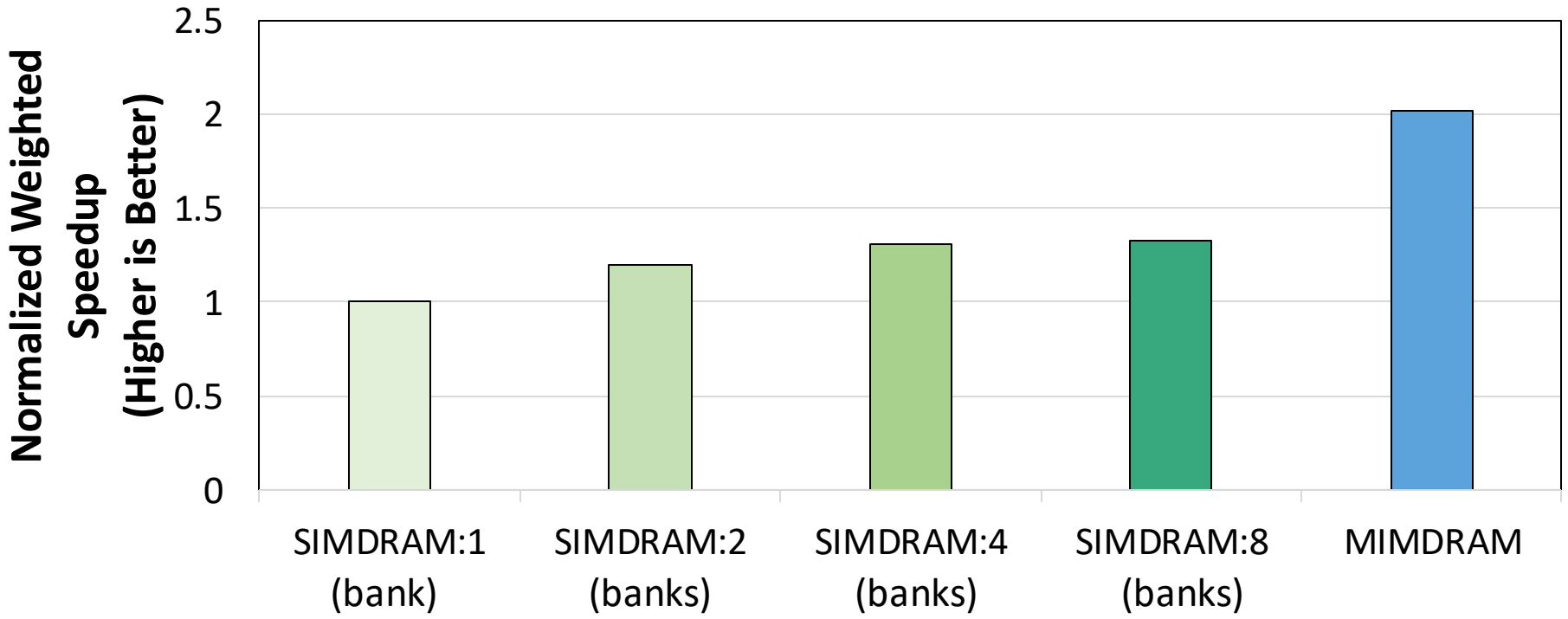
Evaluation:
Single Application Analysis – Energy Efficiency



Takeaway

MIMDRAM significantly improves energy efficiency compared to CPU (30.6x), GPU (6.8x), and SIMDRAM (14.3x)

Evaluation:
Multi-Programmed Workload Analysis



Takeaway

MIMDRAM significantly improves system throughput (1.68x) compared to SIMDRAM

Evaluation:
More in the Paper

- **MIMDRAM with subarray and bank-level parallelism**
  - MIMDRAM provides significant performance gains compared to the baseline CPU (**13.2x**) and GPU (**2x**)

- **Comparison to DRISA and Fulcrum for multi-programmed workloads**
  - MIMDRAM achieves system throughput on par with DRISA and Fulcrum

- **MIMDRAM's SIMD utilization versus SIMDRAM**
  - MIMDRAM provides **15.6x** the utilization of SIMDRAM

- **Area analysis**
  - MIMDRAM adds small area cost to a DRAM chip (**1.11%**) and CPU die (**0.6%**)

Outline

- **Processing-Using-Memory (PUM) Solutions**
  - Review of DRAM Background, RowClone, and Ambit
  - SIMDRAM & MIMDRAM: Generalizing PUM Computing
  - **pLUTo: Extending Operation Support with LUTs**
  - PUM in Off-the-Shelf DRAM Chips
  - PUM Beyond Boolean/Arithmetic and DRAM
- Processing-Near-Memory (PNM) Solutions
  - Overview
  - Accelerating Neural Networks & Databases
  - Real PNM Systems
- Barriers for Adoption (and Current Solutions)
- Conclusion

In-DRAM Lookup-Table Based Execution

João Dinis Ferreira, Gabriel Falcao, Juan Gómez-Luna, Mohammed Alser, Lois Orosa, Mohammad Sadrosadati, Jeremie S. Kim, Geraldo F. Oliveira, Taha Shahroodi, Anant Nori, and Onur Mutlu,
"pLUTo: Enabling Massively Parallel Computation in DRAM via Lookup Tables"
*Proceedings of the 55th International Symposium on Microarchitecture (**MICRO**)*, Chicago, IL, USA, October 2022.
[Slides (pptx) (pdf)]
[Longer Lecture Slides (pptx) (pdf)]
[Lecture Video (26 minutes)]
[arXiv version]
[Source Code (Officially Artifact Evaluated with All Badges)]
Officially artifact evaluated as available, reusable and reproducible.

pLUTo: Enabling Massively Parallel Computation in DRAM via Lookup Tables

João Dinis Ferreira[§] Gabriel Falcao[†] Juan Gómez-Luna[§] Mohammed Alser[§]
Lois Orosa[§▽] Mohammad Sadrosadati[§] Jeremie S. Kim[§] Geraldo F. Oliveira[§]
Taha Shahroodi[‡] Anant Nori[⋆] Onur Mutlu[§]
[§]*ETH Zürich* [†]*IT, University of Coimbra* [▽]*Galicia Supercomputing Center* [‡]*TU Delft* [⋆]*Intel*

https://arxiv.org/pdf/2104.07699.pdf

Limitations of Processing-using-DRAM

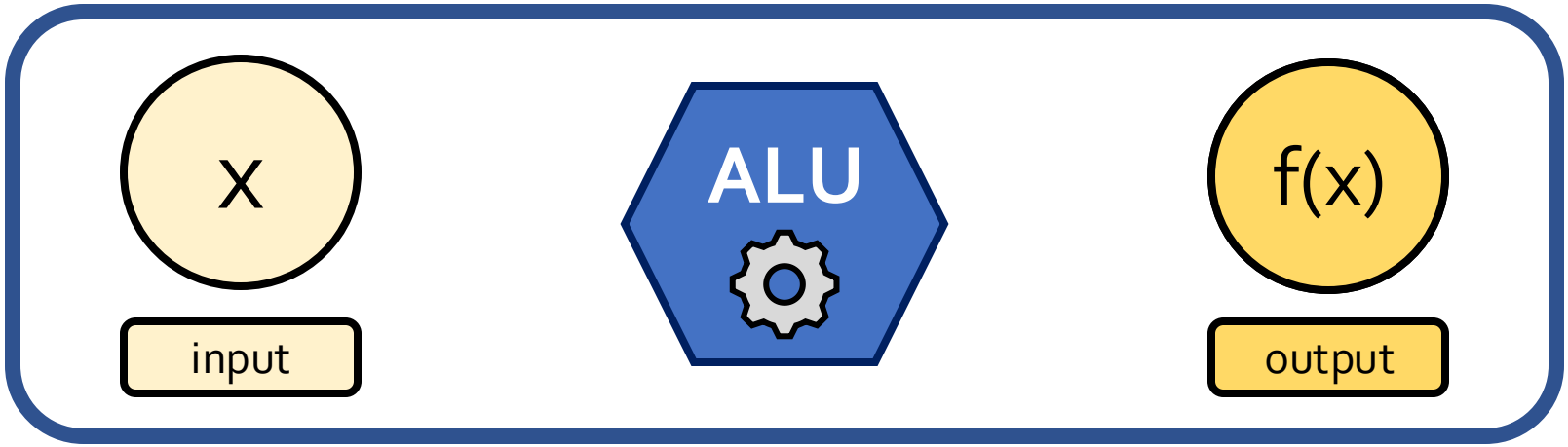
Data Movement	*RowClone, Seshadri+ 2013* *LISA, Chang+ 2013*
Bitwise Operations	*Ambit, Seshadri+ 2017*
Bit Shifting	*DRISA, Li+ 2017*
Arithmetic Operations	*SIMDRAM, Hajinazar & Oliveira+ 2021*

Existing Processing-using-DRAM architectures only support a limited range of operations

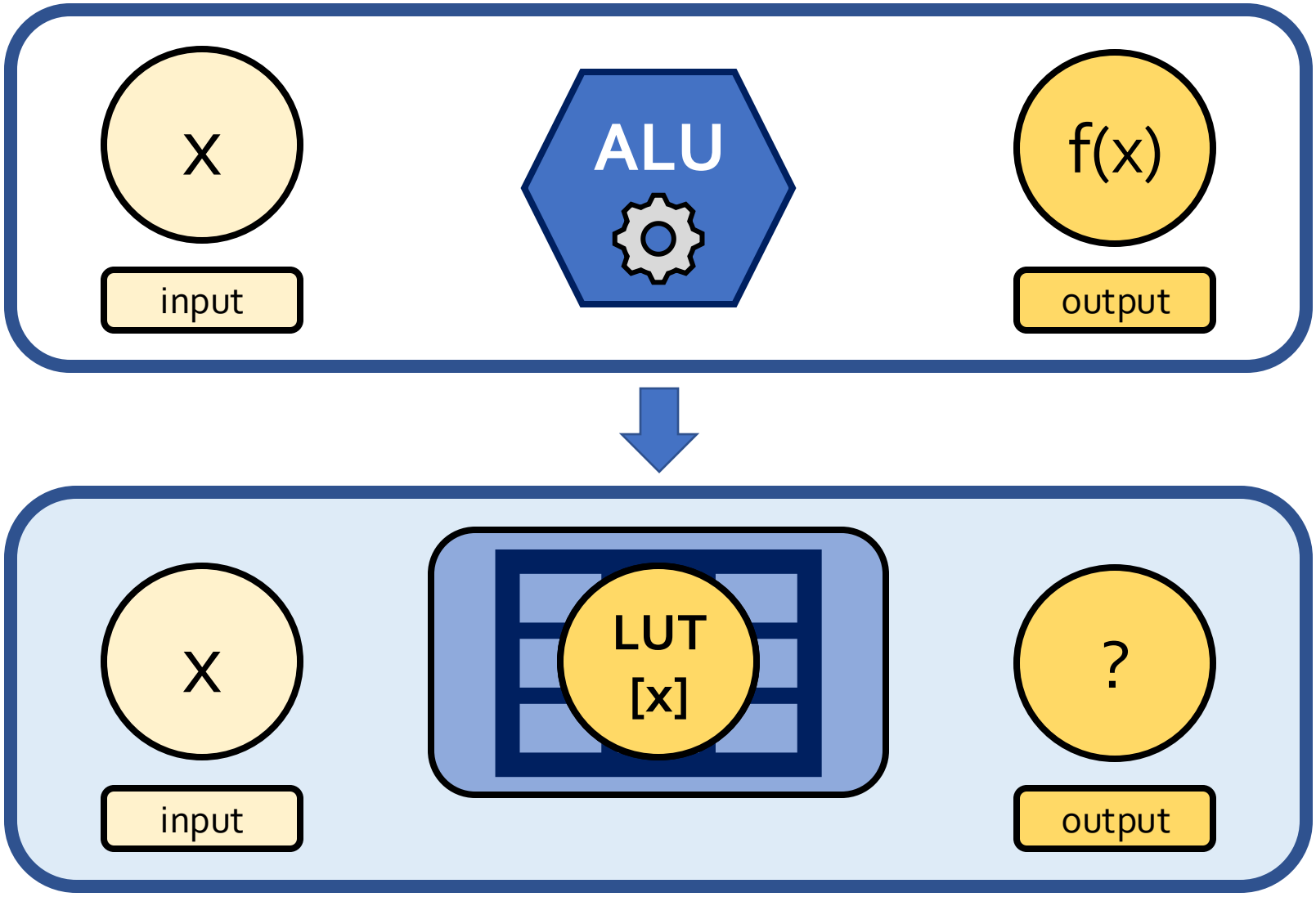
The Goal of pLUTo

Extend **Processing-using-DRAM to support the execution of** ***arbitrarily complex operations***

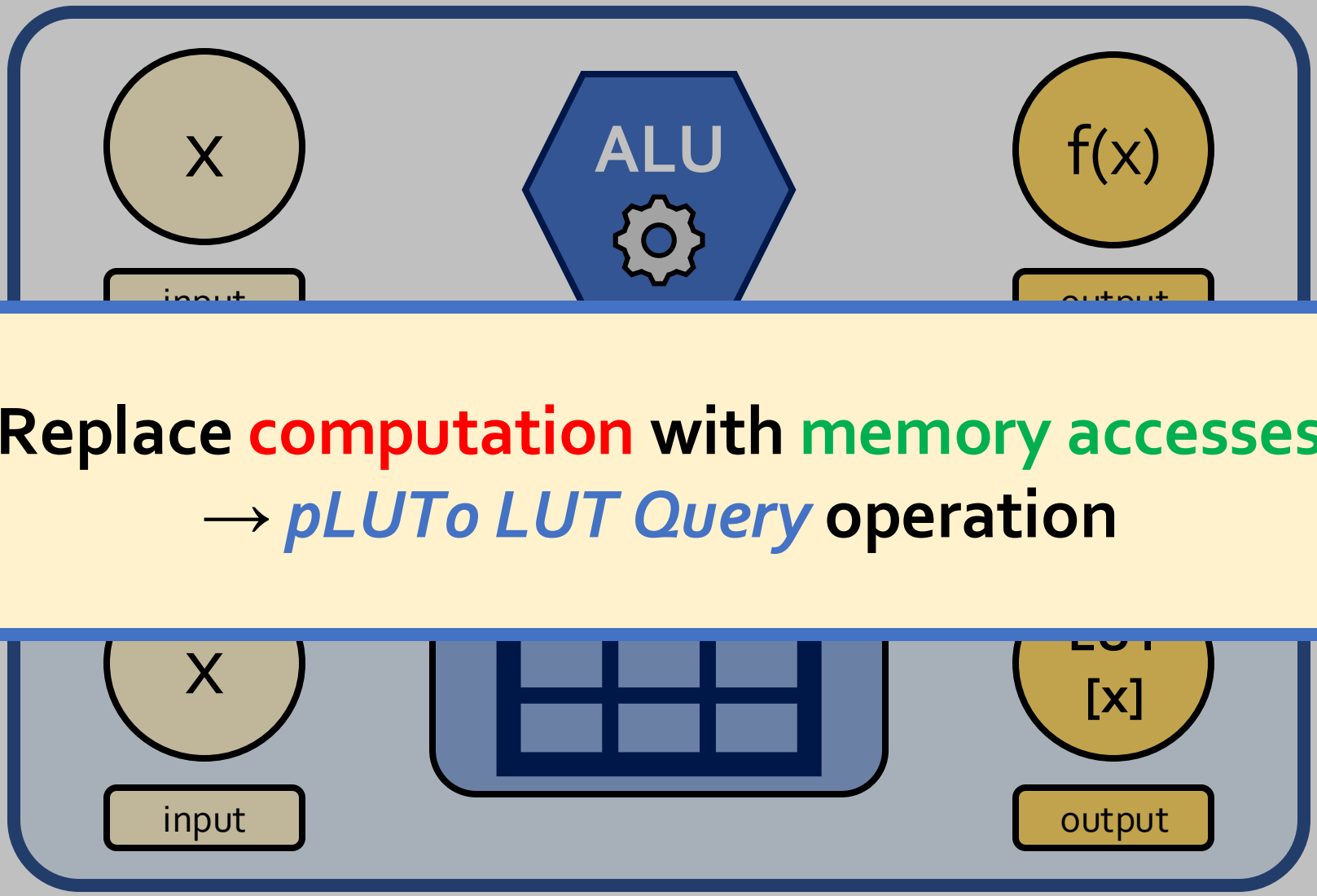
pLUTo: Key Idea



pLUTo: Key Idea



pLUTo: Key Idea



Replace computation with memory accesses
→ *pLUTo LUT Query* operation

In-DRAM pLUTo LUT Query: Setup

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

	Prime numbers	
LUT index	i	f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

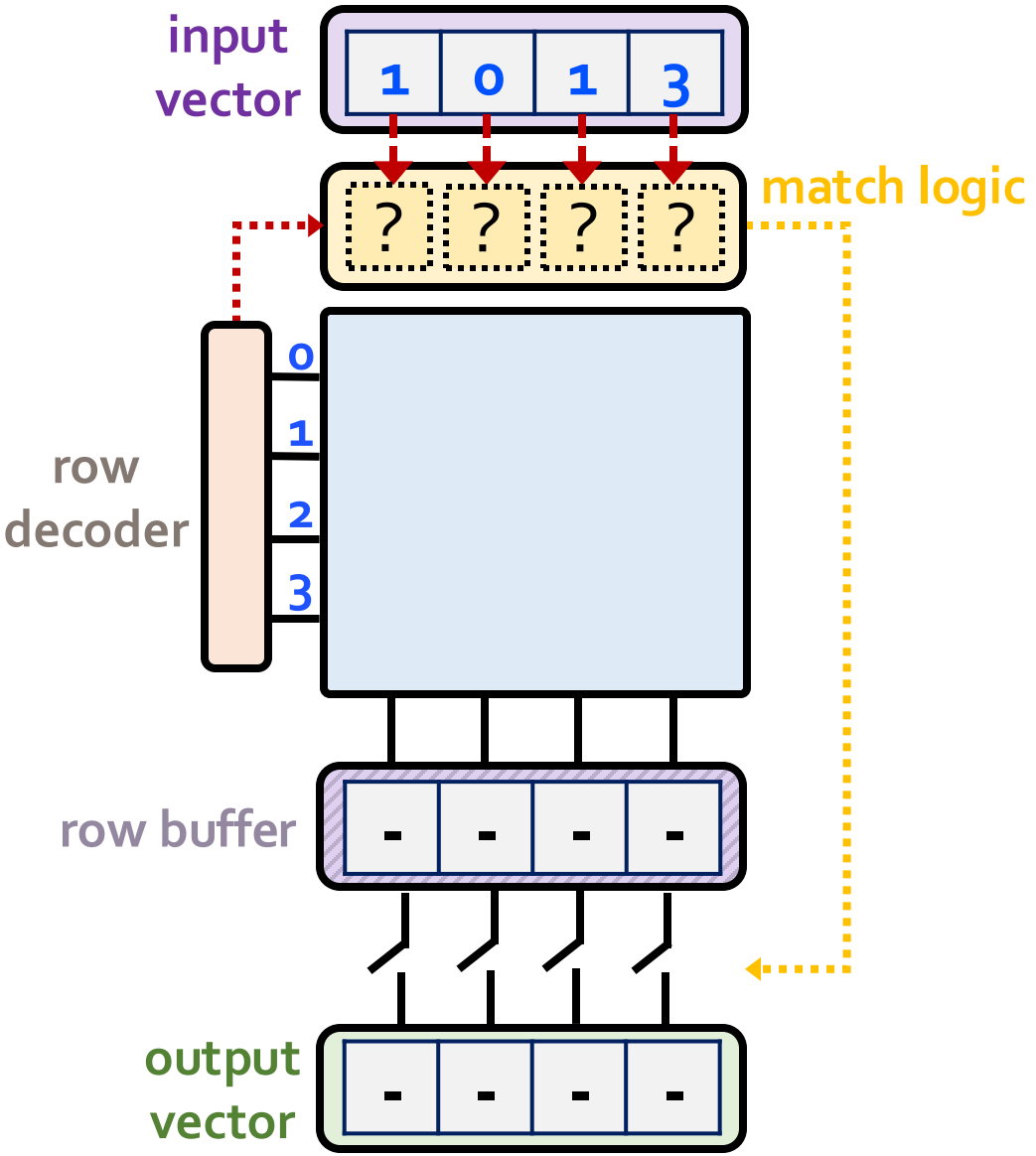
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Setup

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

LUT index	Prime numbers i	Prime numbers f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

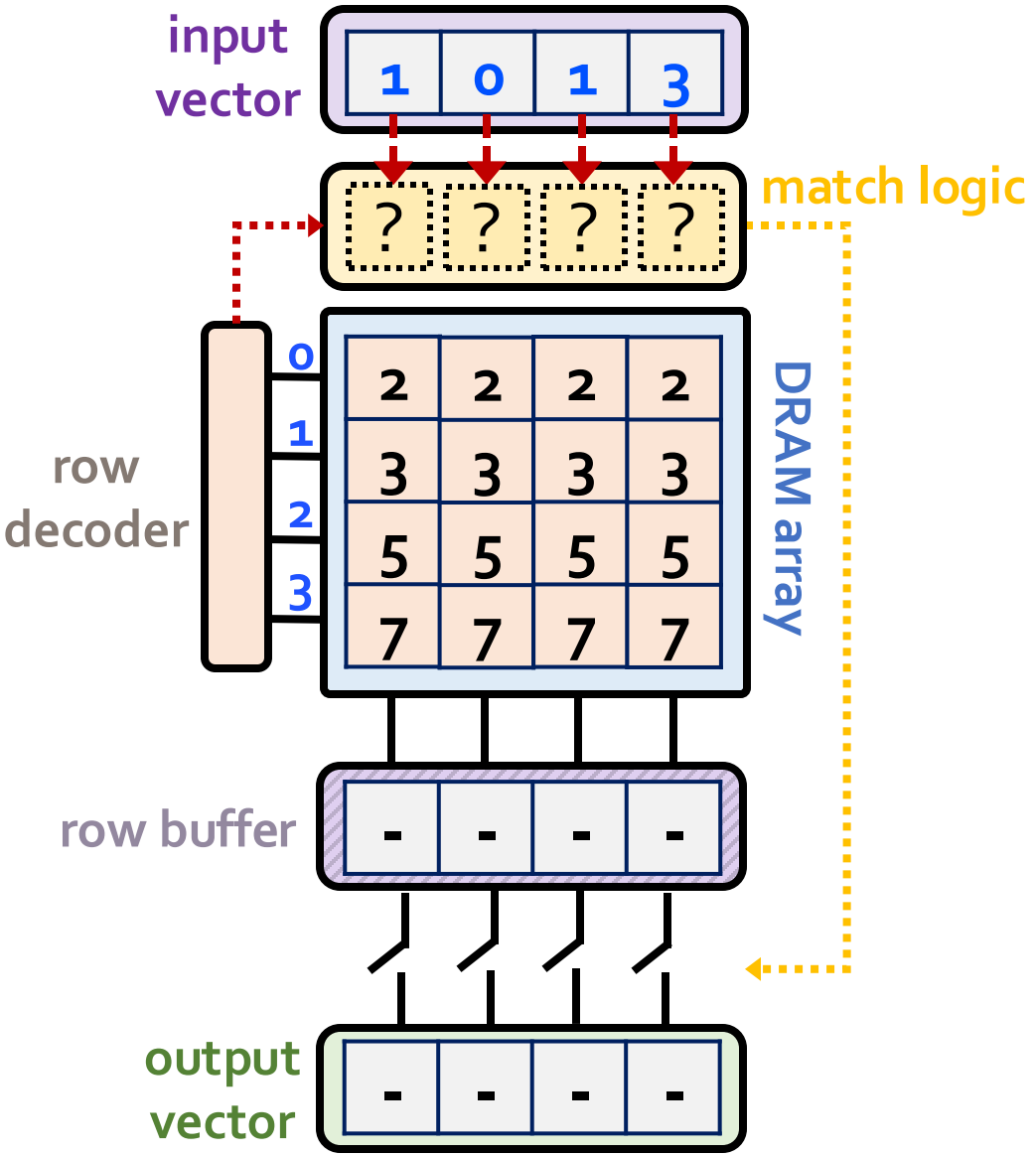
lookup table

1	0	1	3

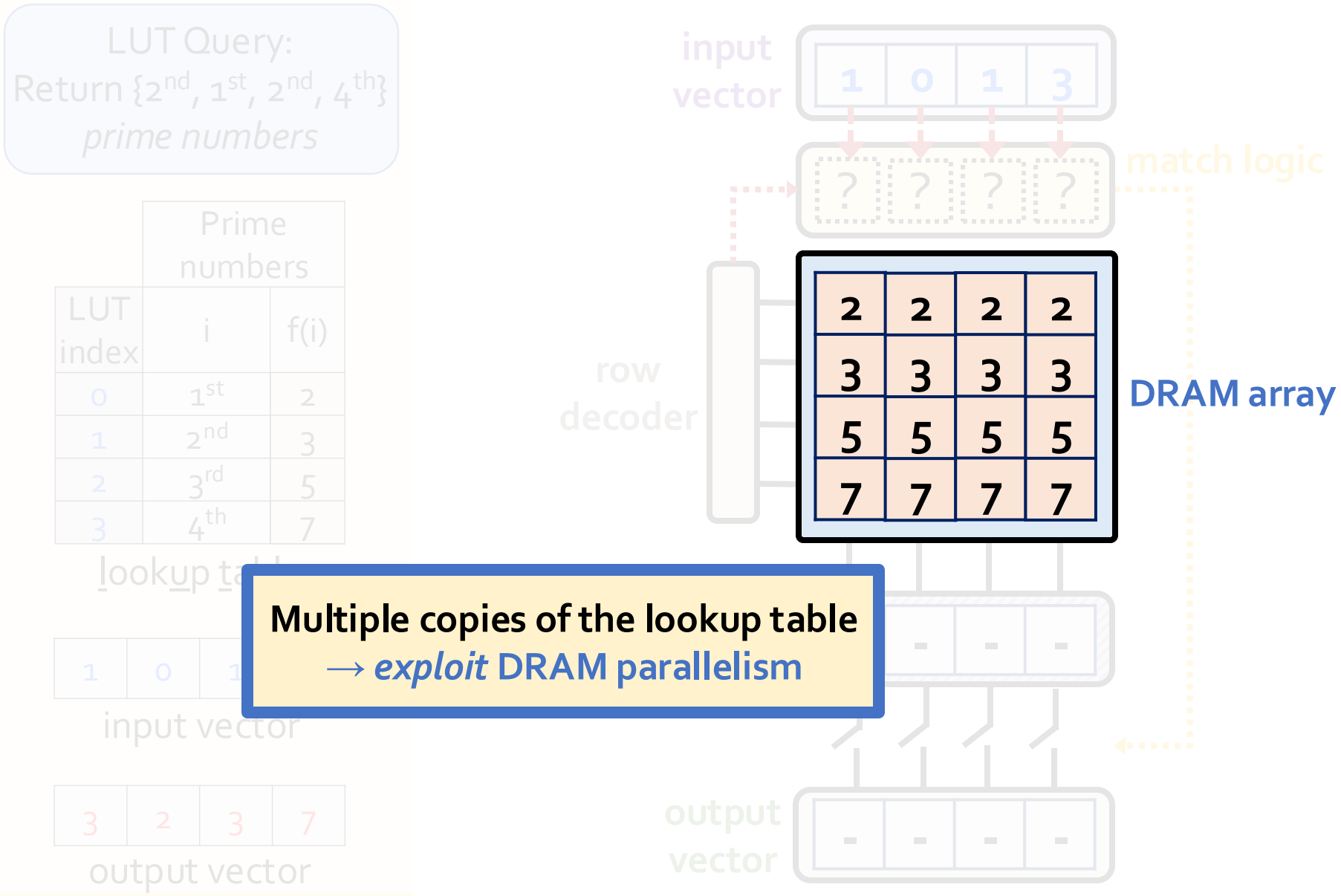
input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Setup



In-DRAM pLUTo LUT Query: Setup

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

LUT index	Prime numbers: i	Prime numbers: f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

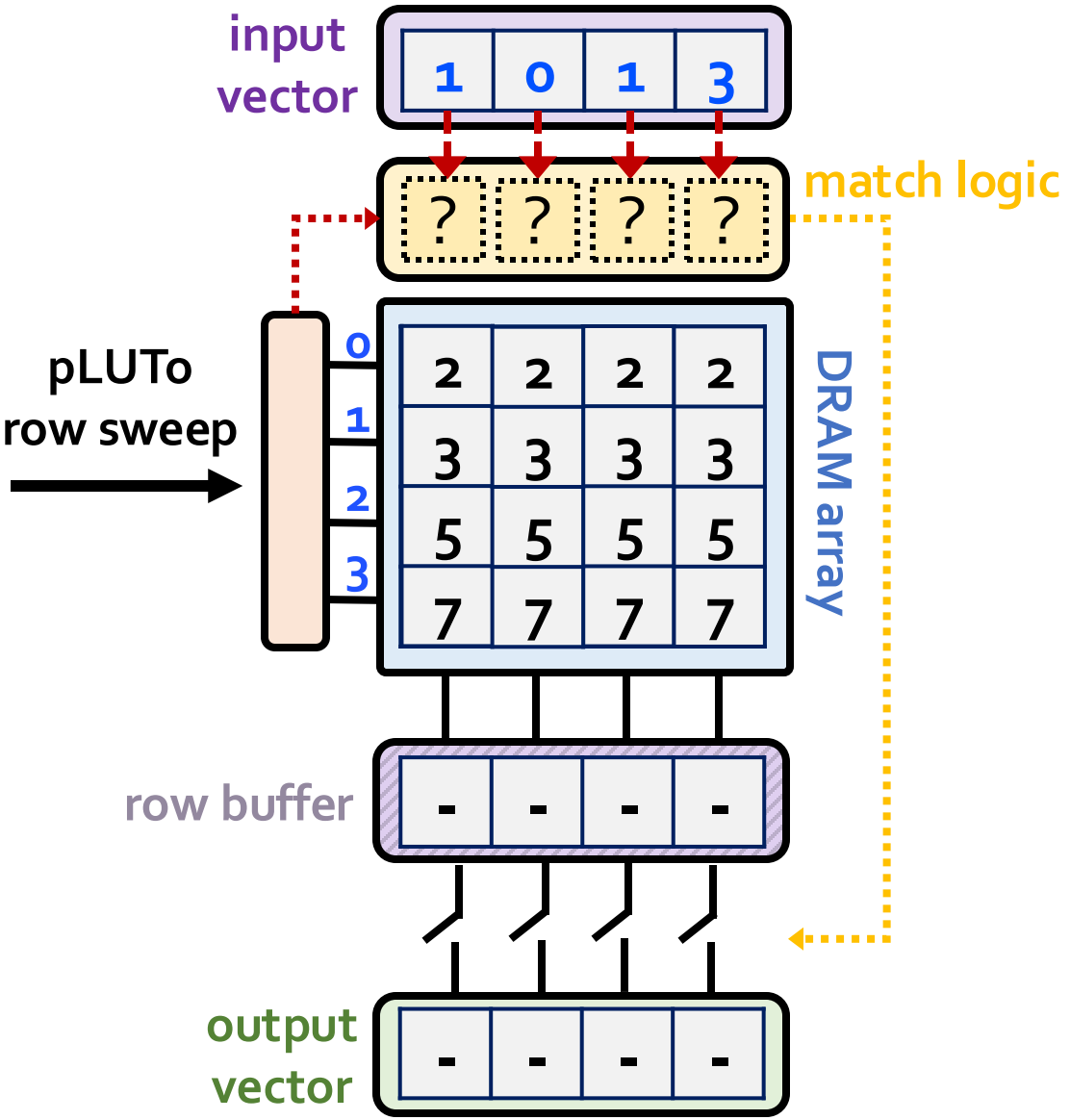
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Step 1

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

LUT index	Prime numbers i	f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

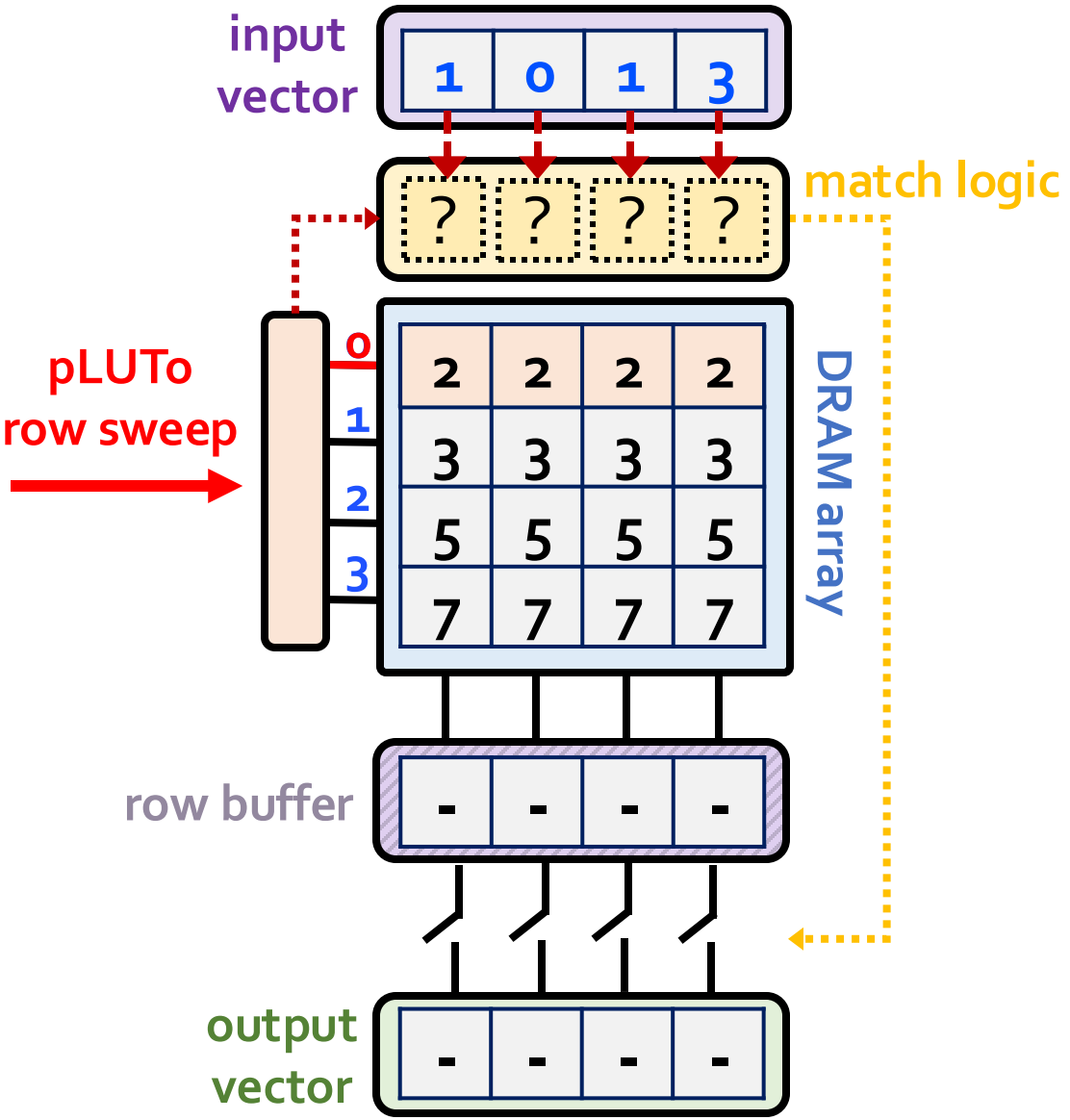
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Step 1

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

LUT index	Prime numbers: i	Prime numbers: f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

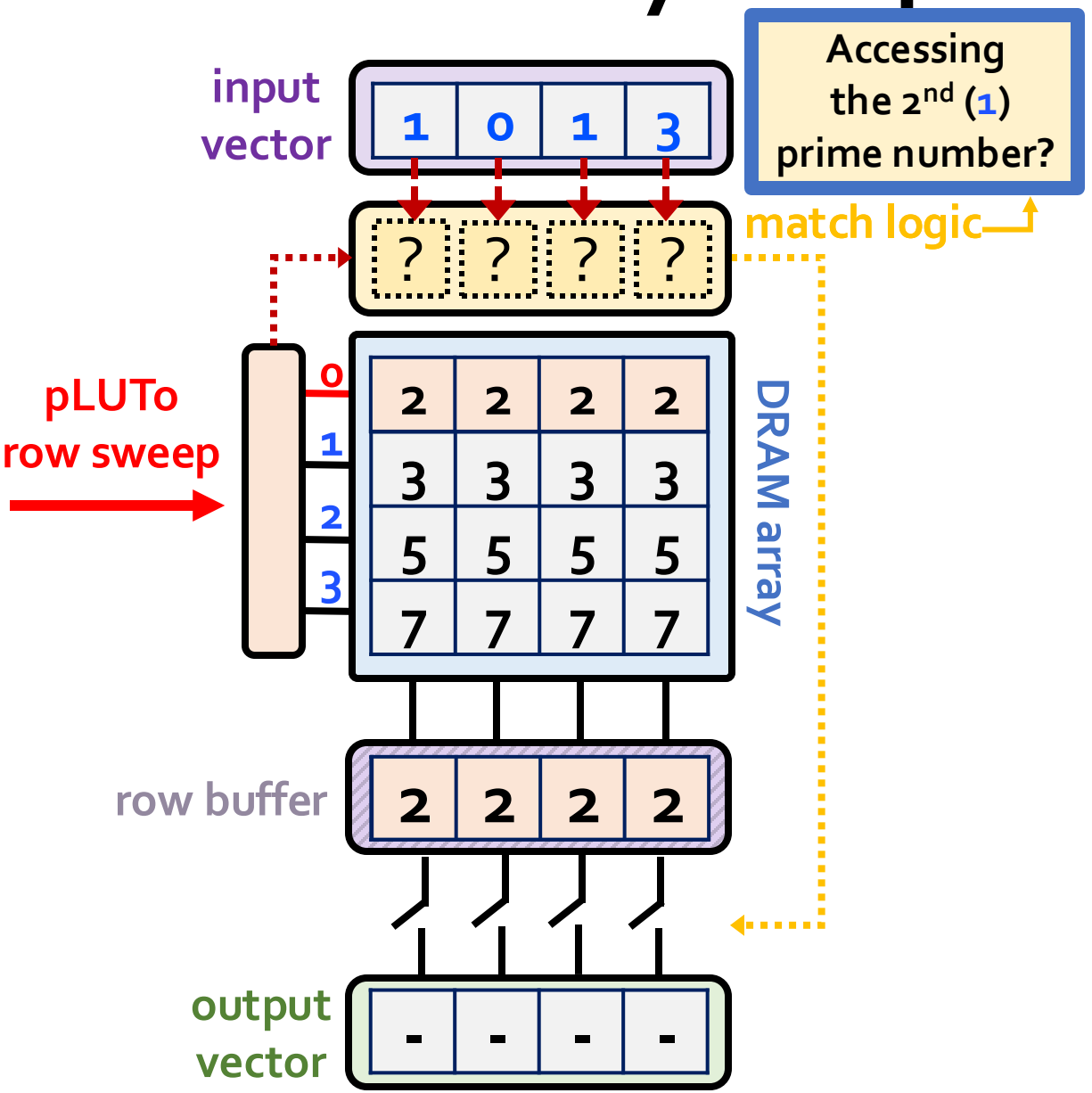
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Step 1

LUT Query:
Return {2nd, 1st, 2nd, 4th} *prime numbers*

LUT index	Prime numbers i	f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

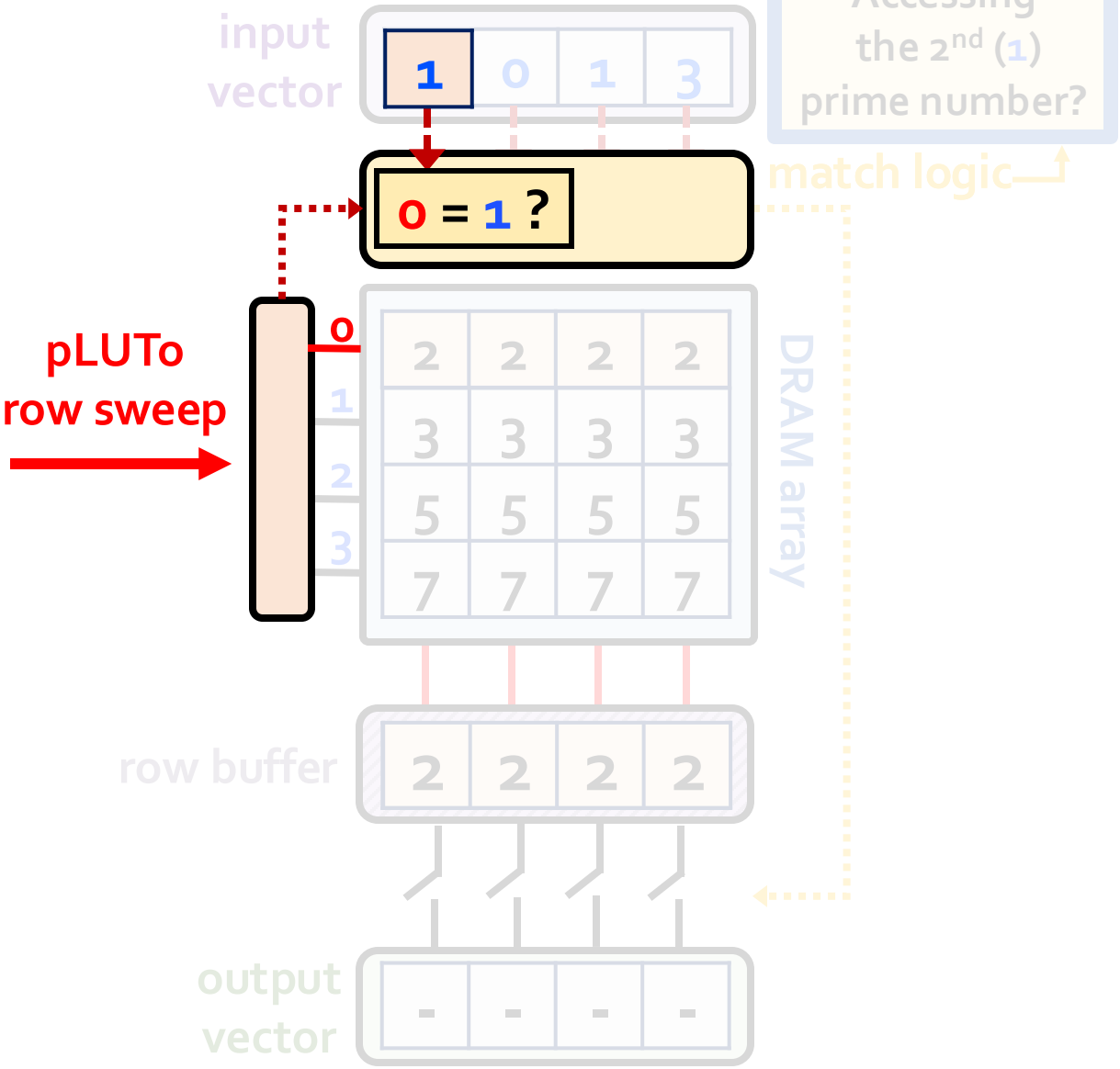
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Step 1

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

LUT index	Prime numbers: i	Prime numbers: f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

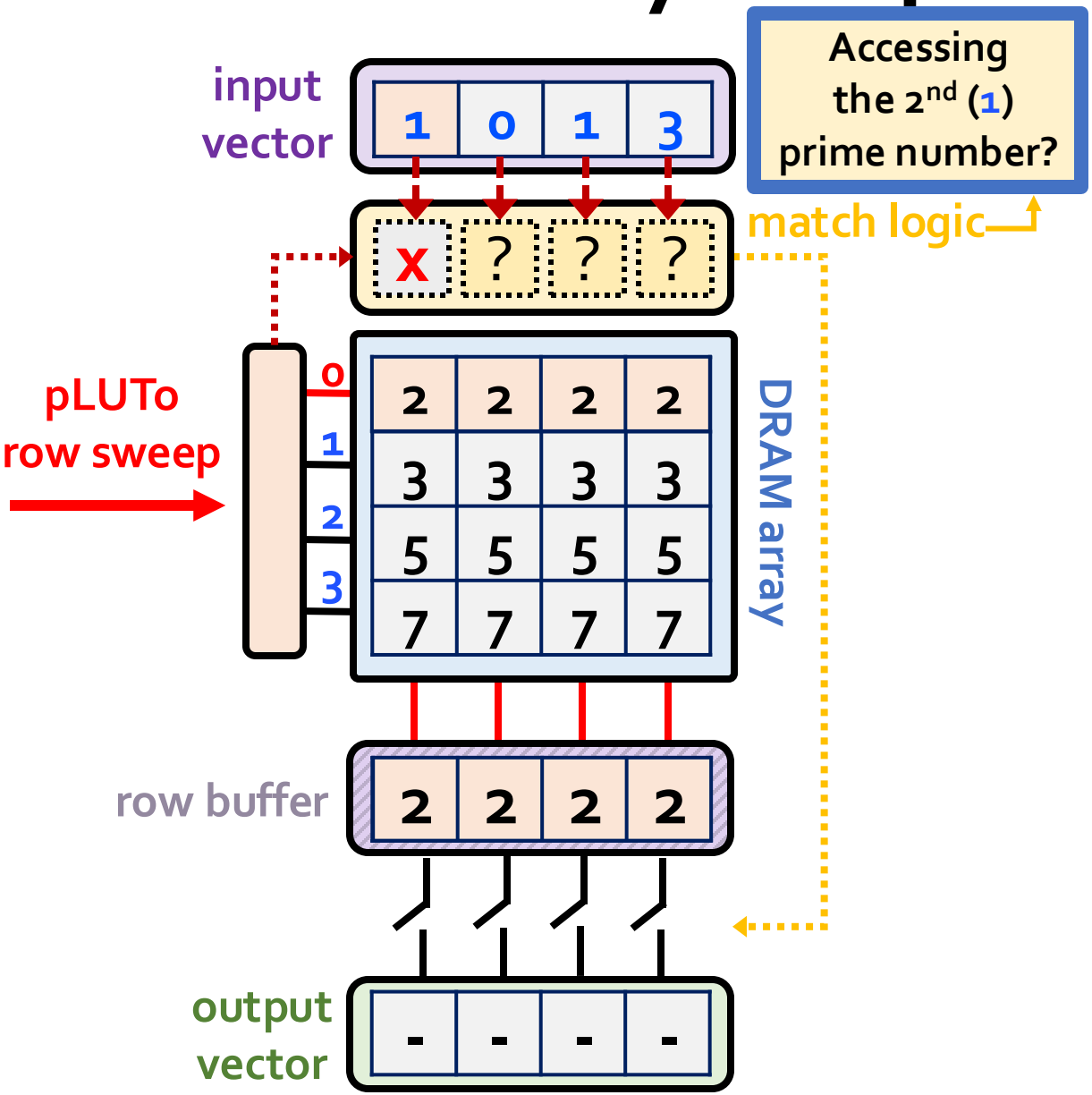
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Step 1

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

LUT index	Prime numbers: i	Prime numbers: f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

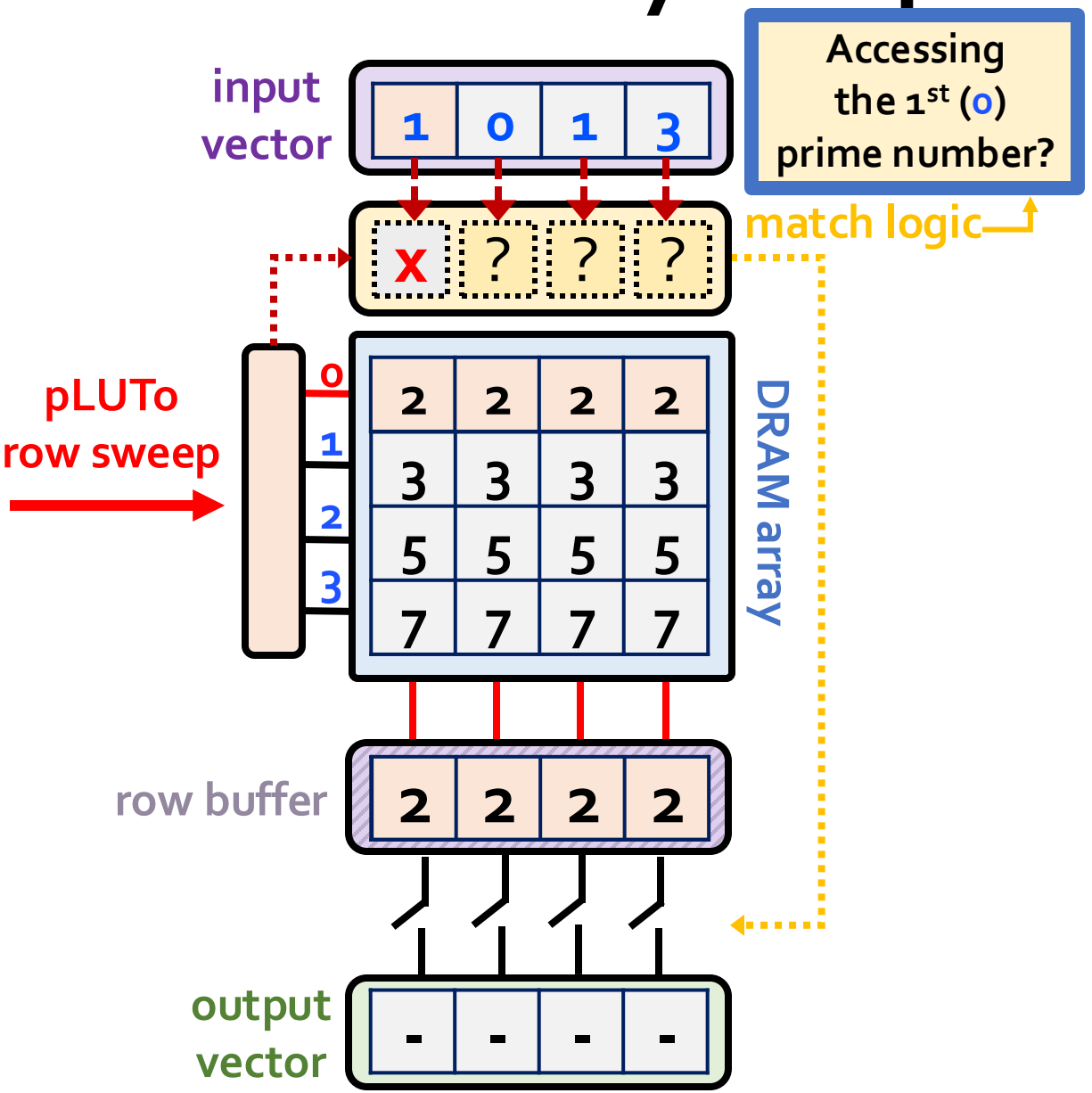
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Step 1

LUT Query:
Return {2nd, 1st, 2nd, 4th} *prime numbers*

LUT index	Prime numbers i	f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

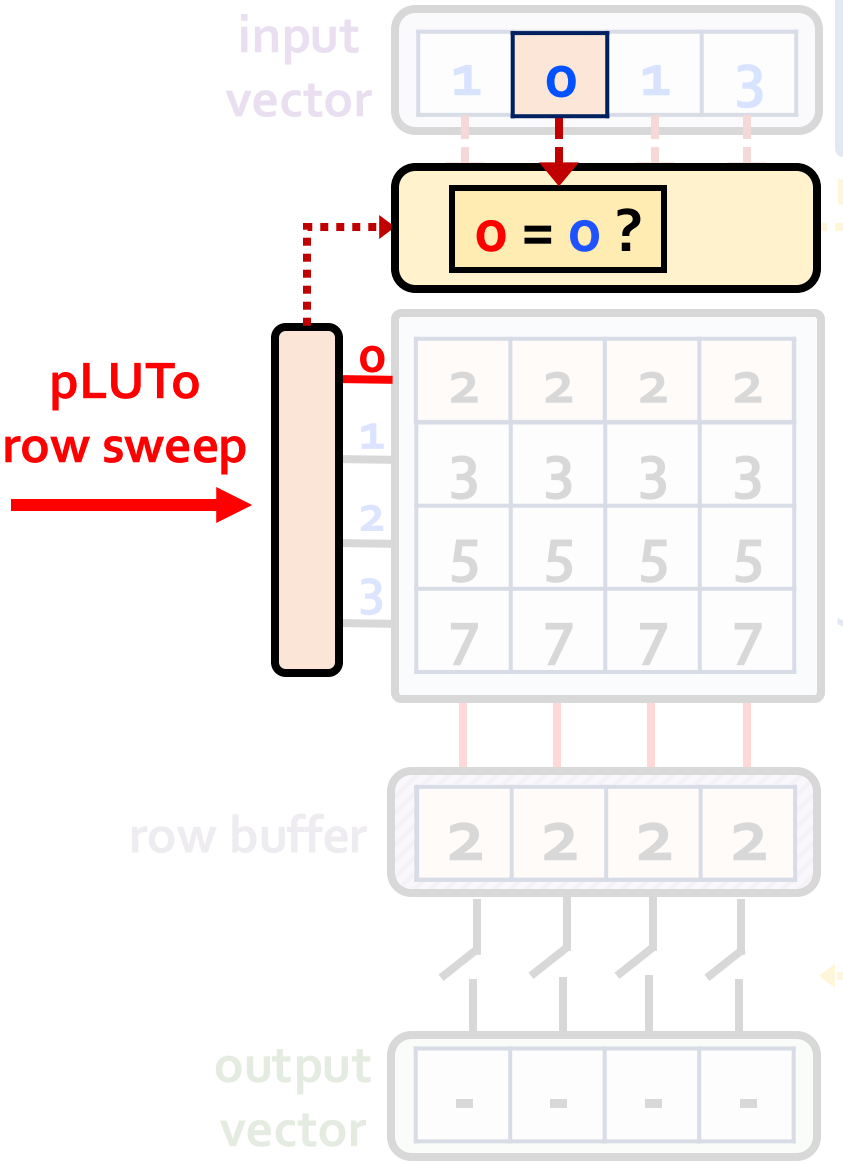
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Step 1

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

LUT index	Prime numbers i	f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

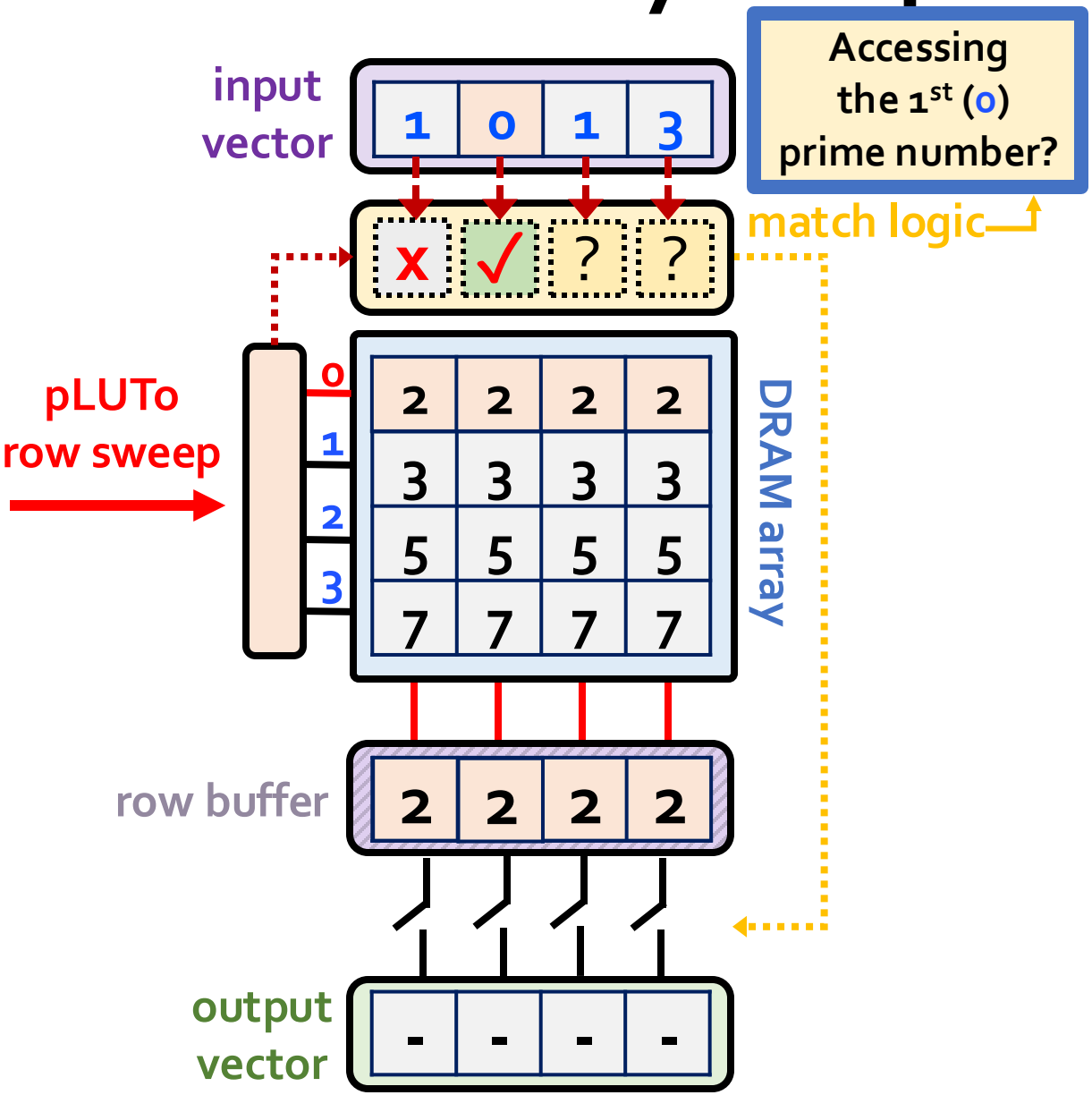
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Step 1

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

LUT index	Prime numbers: i	Prime numbers: f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

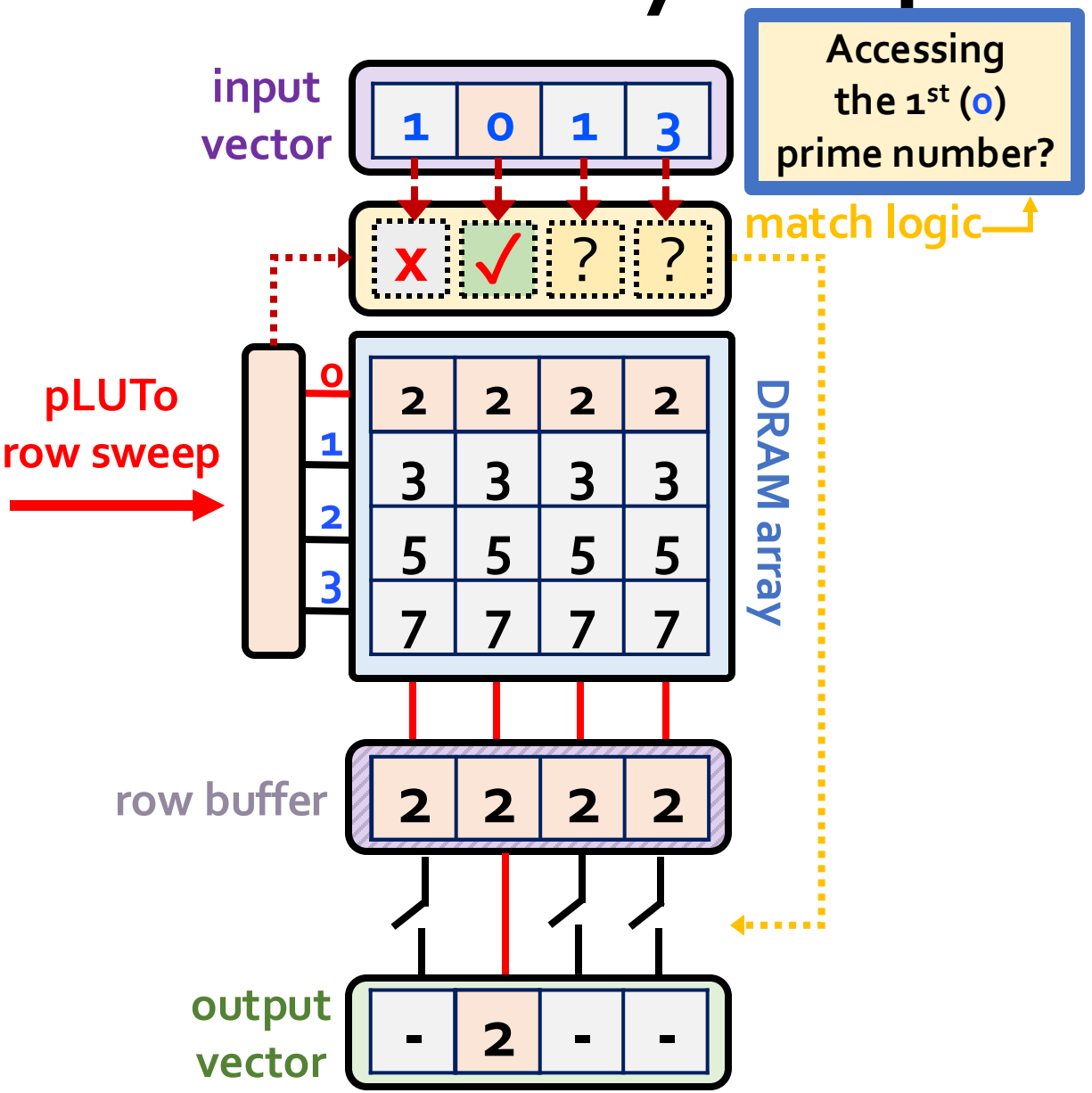
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Step 1

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

LUT index	Prime numbers: i	Prime numbers: f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

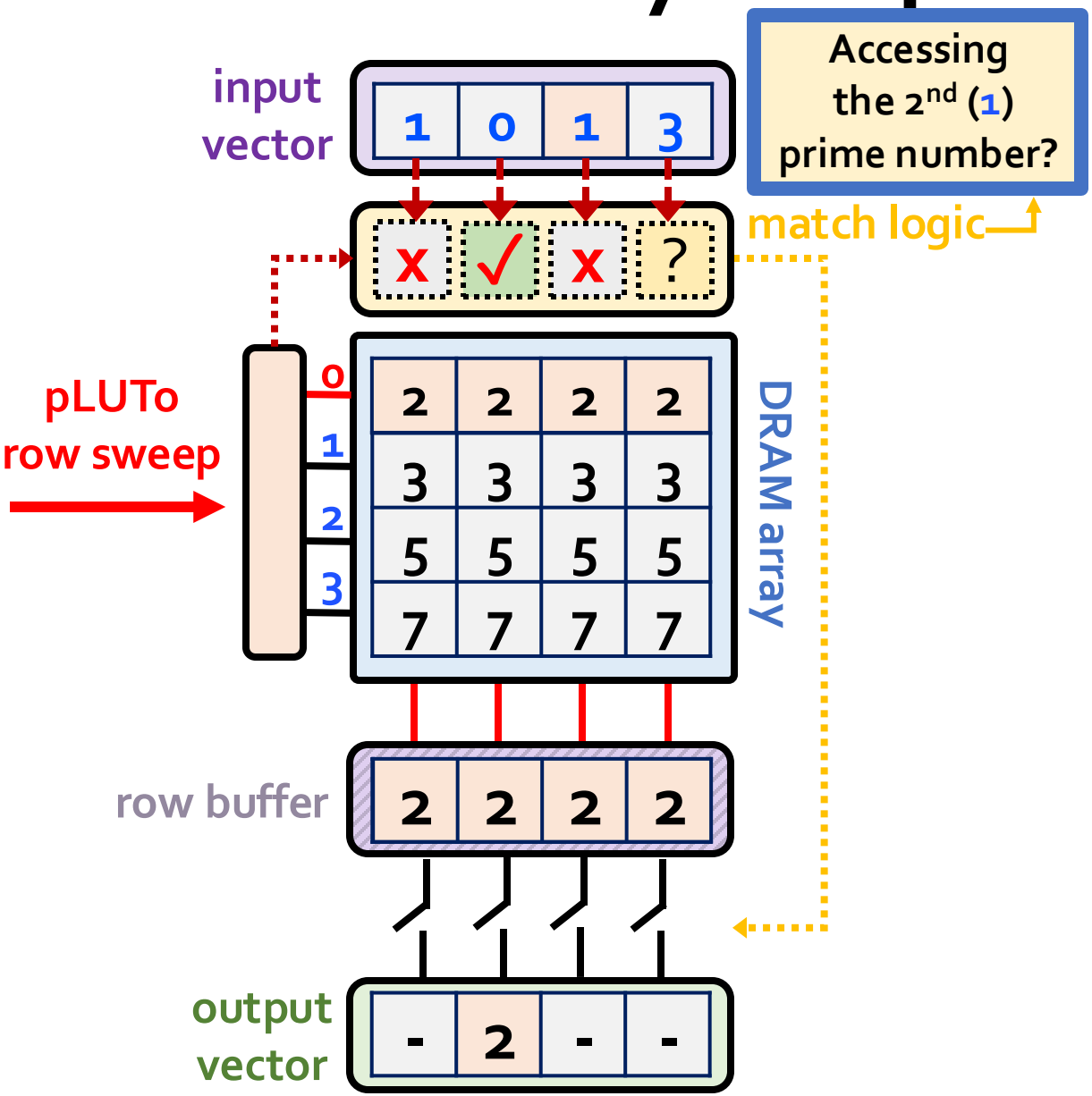
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Step 1

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

LUT index	Prime numbers: i	Prime numbers: f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

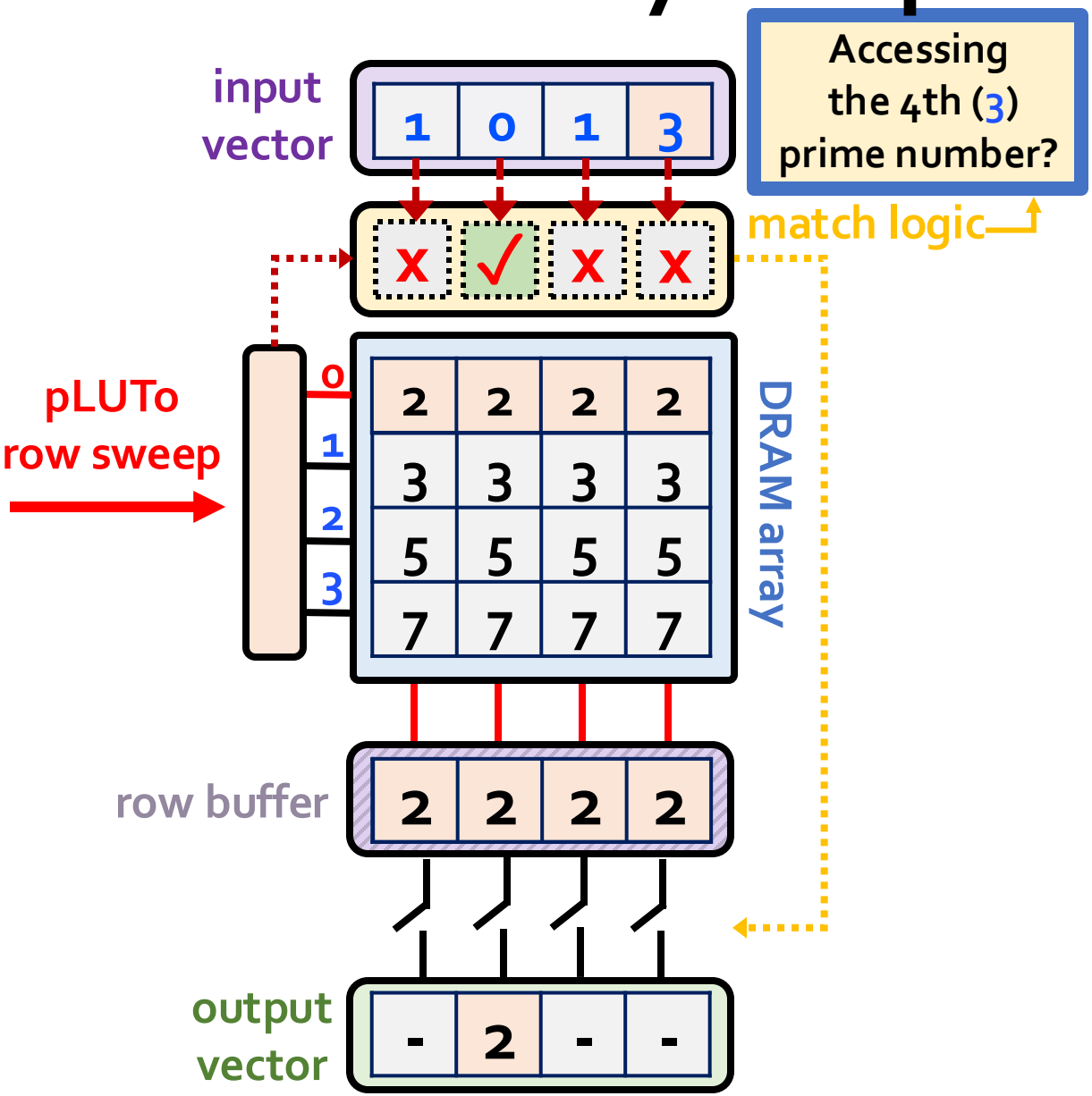
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Step 2

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

LUT index	Prime numbers i	f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

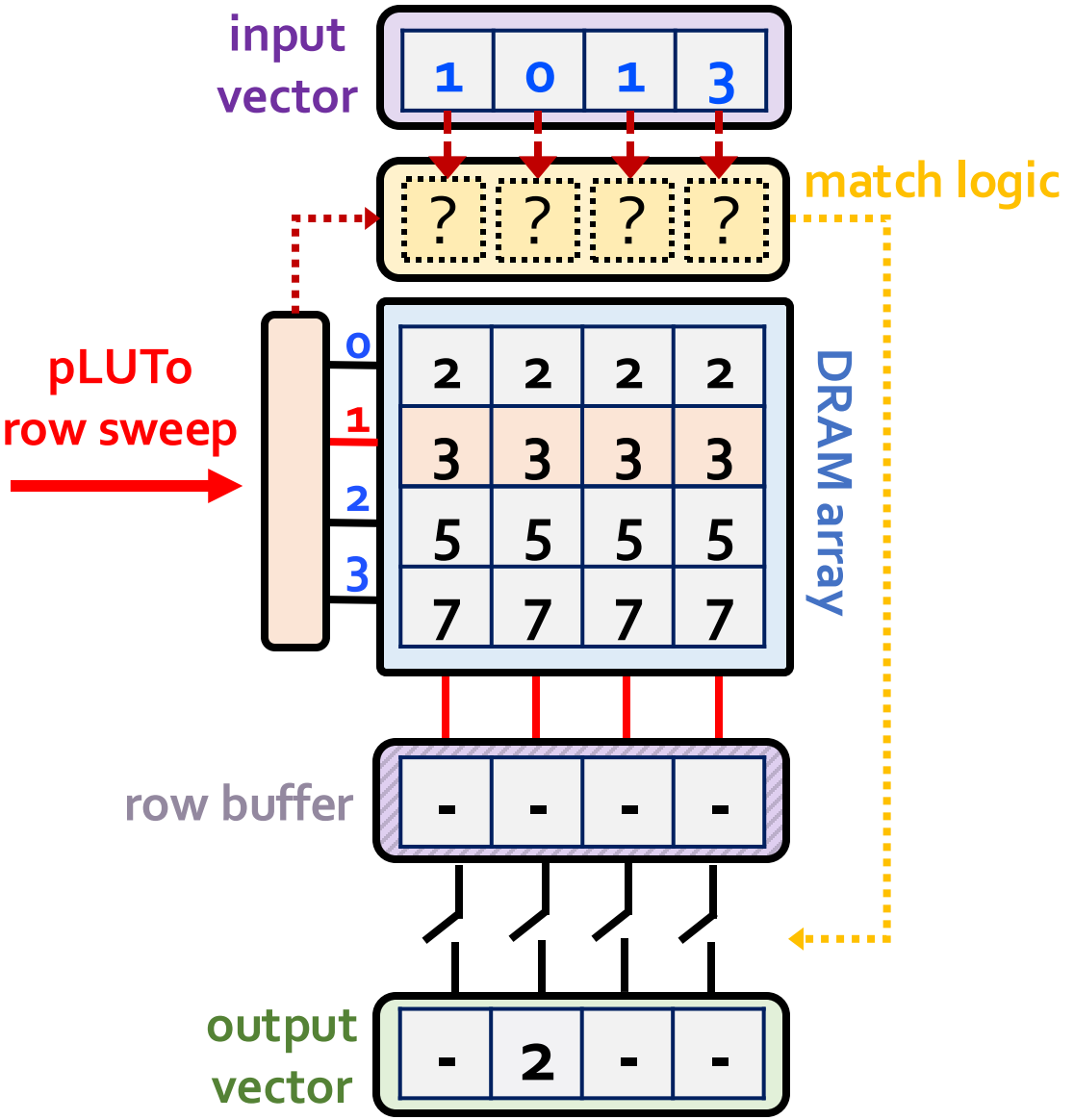
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Step 2

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

LUT index	Prime numbers i	f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

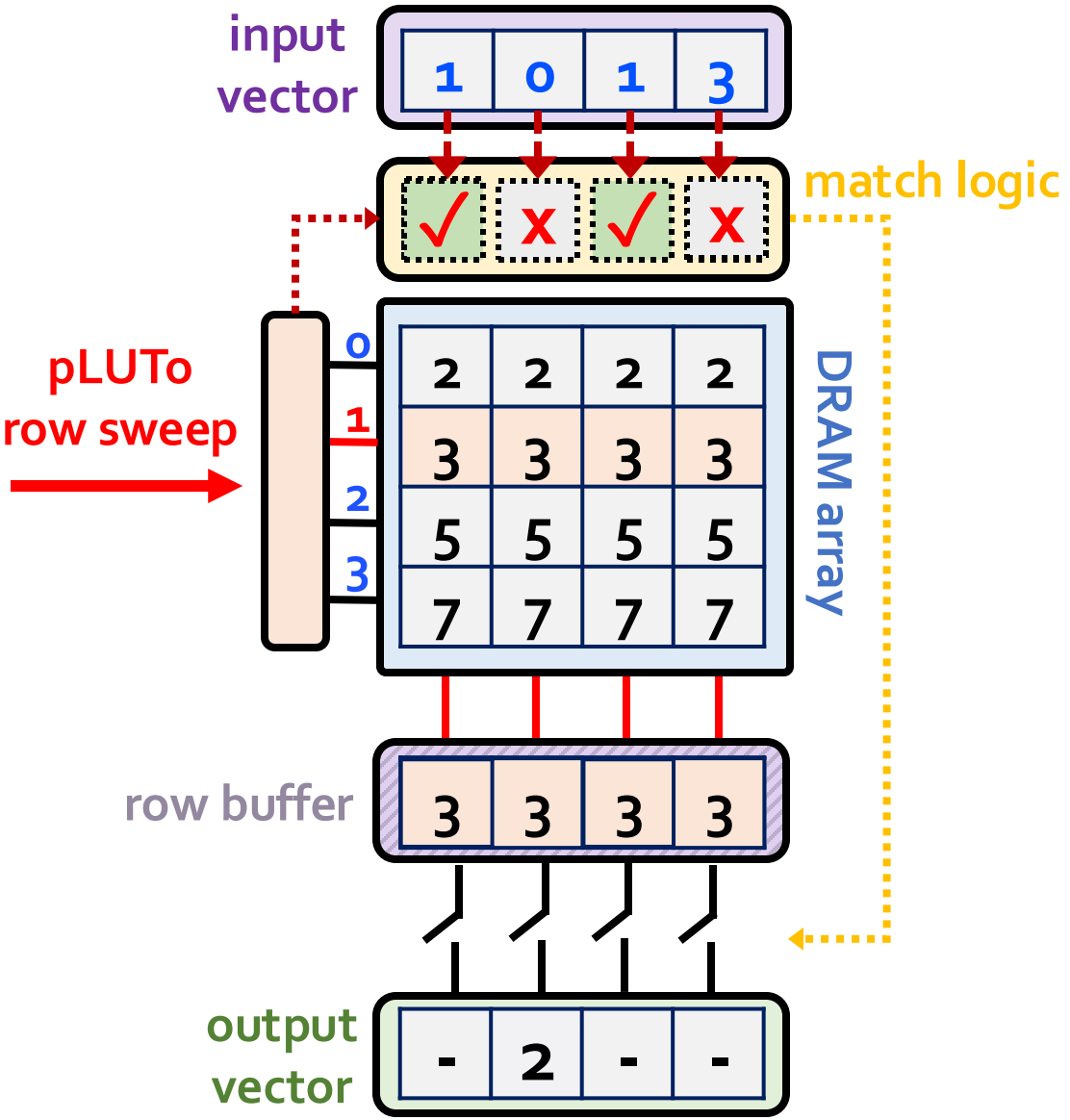
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Step 2

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

LUT index	Prime numbers i	f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

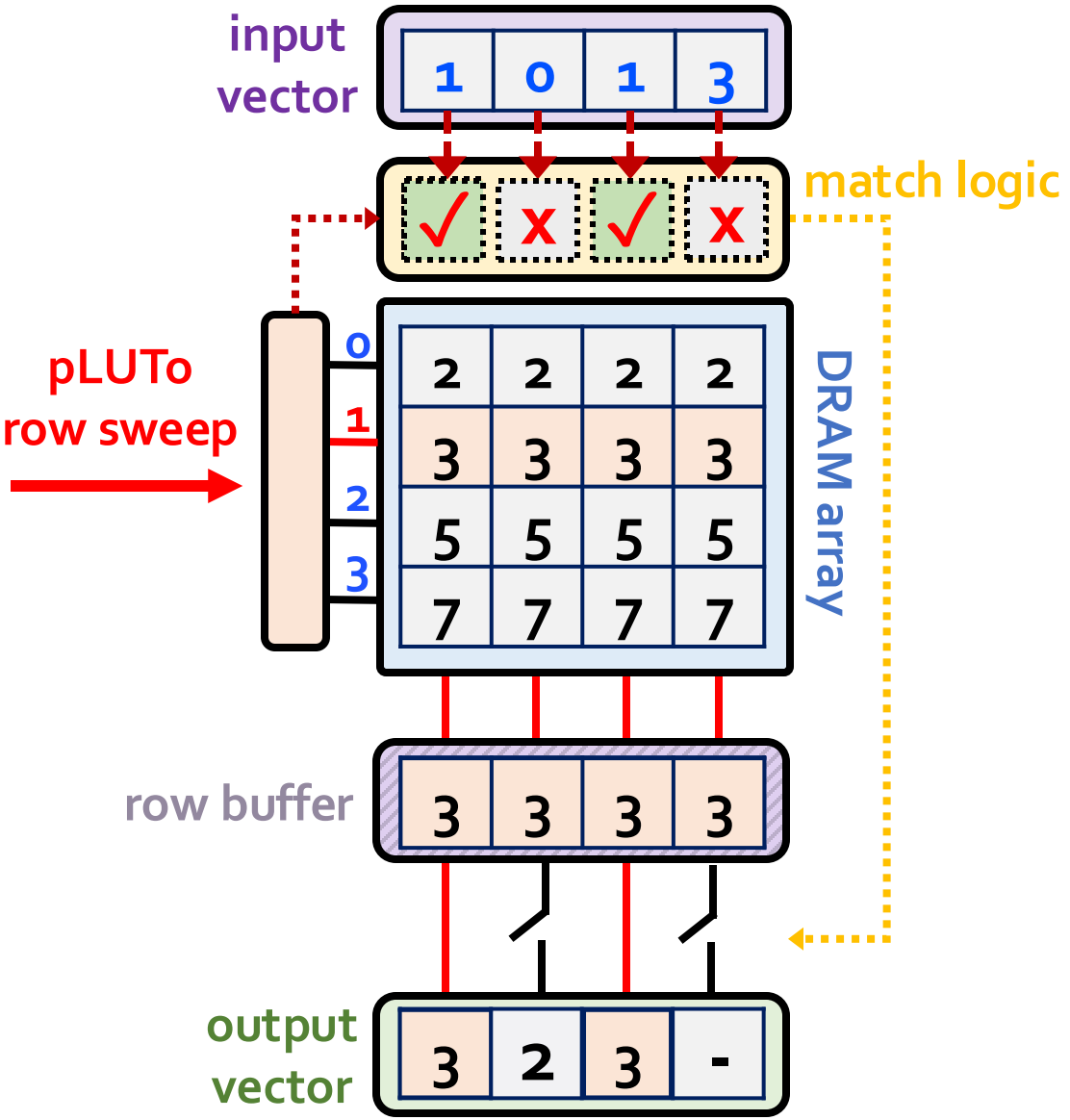
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Step 3

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

LUT index	Prime numbers i	f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

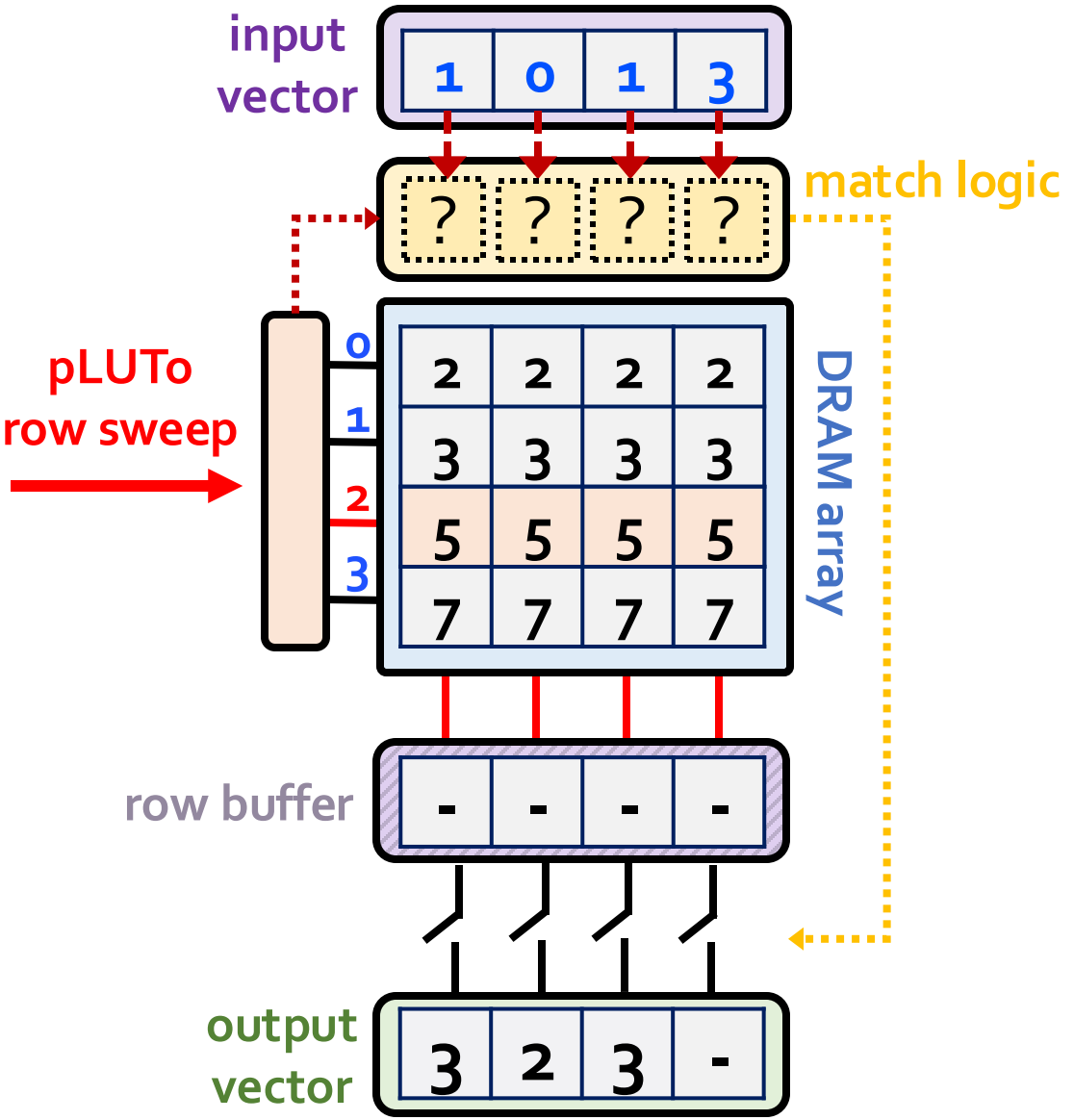
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Step 3

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

LUT index	Prime numbers i	f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

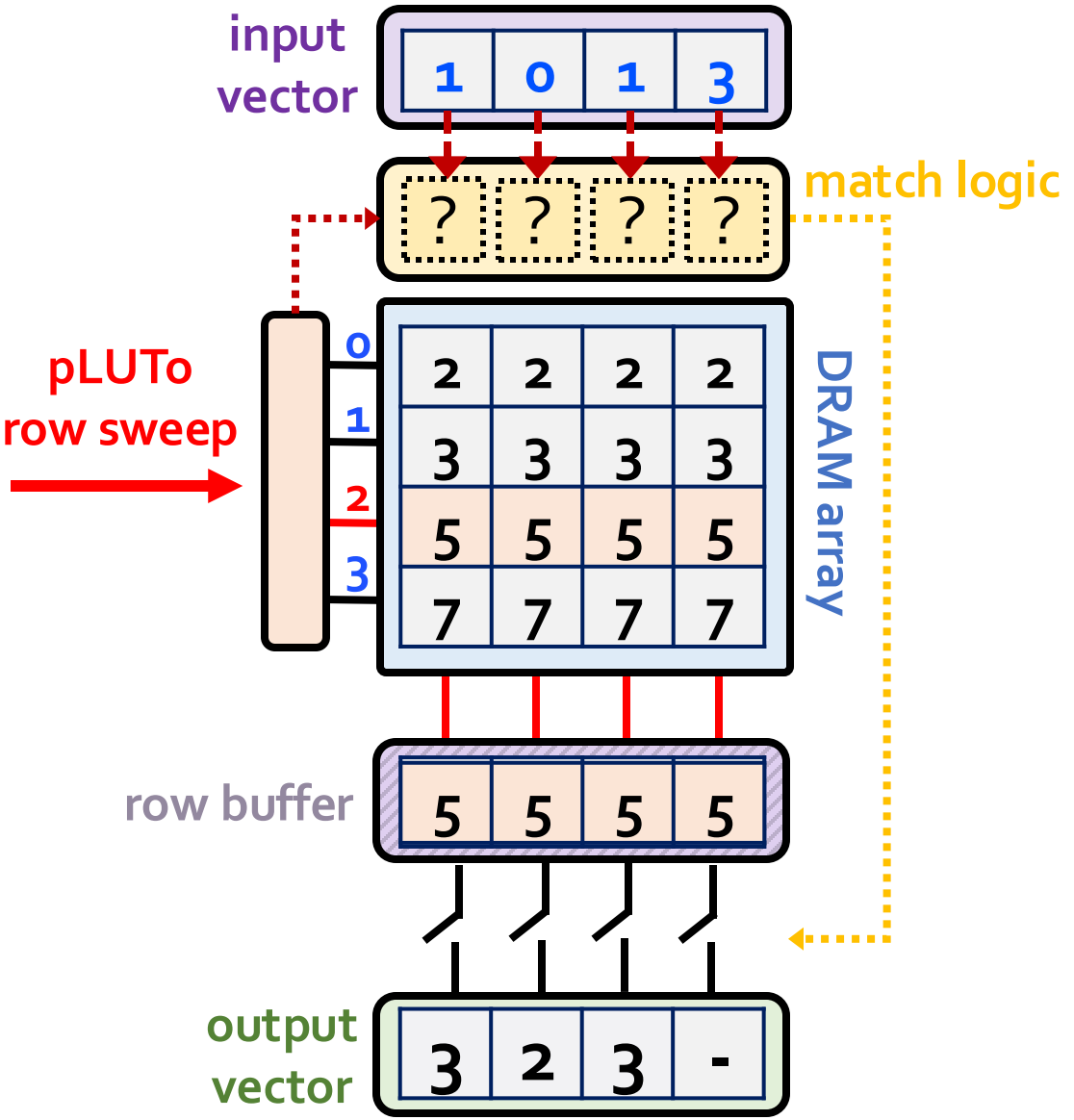
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Step 3

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

LUT index	Prime numbers: i	Prime numbers: f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

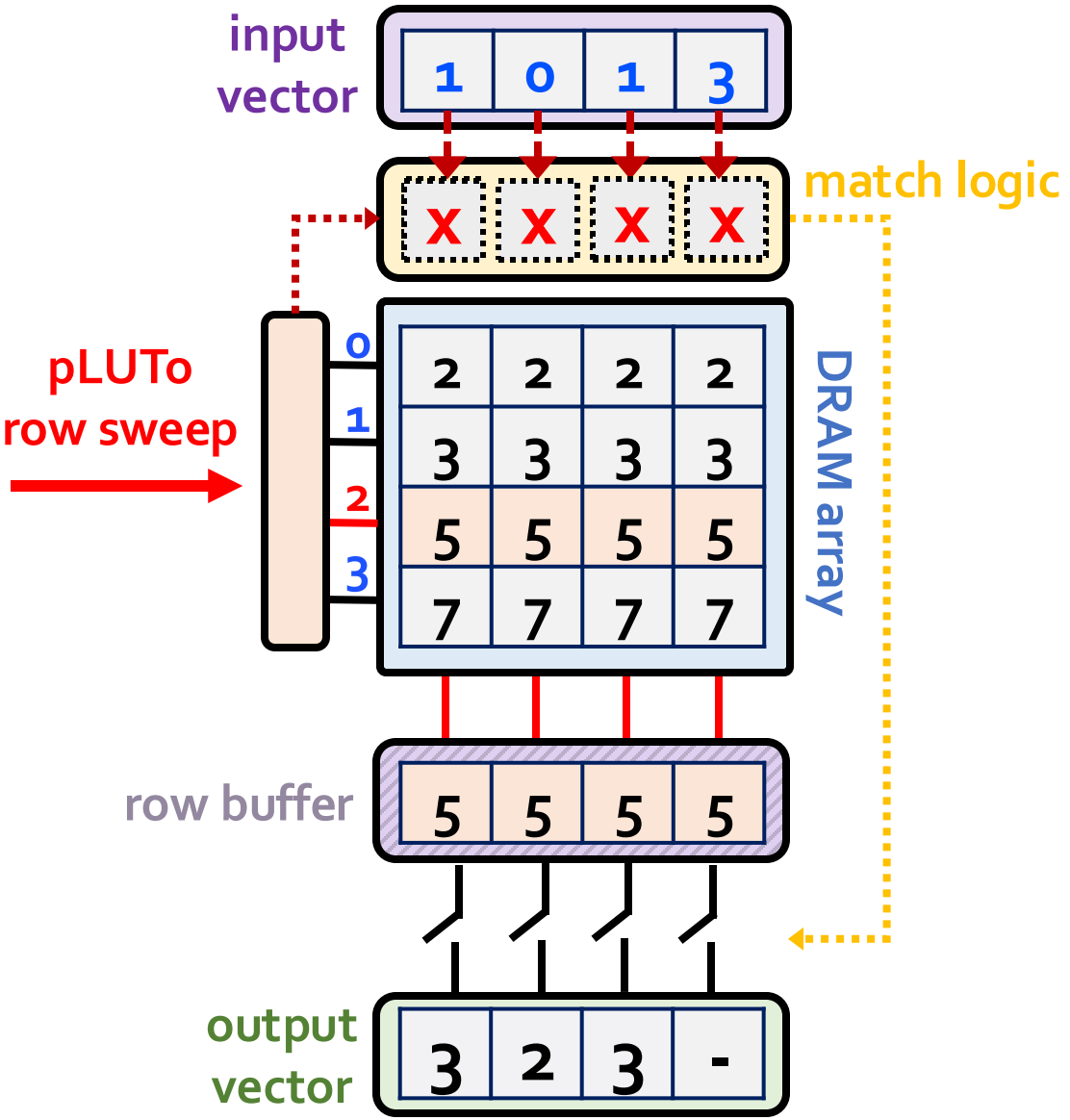
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Step 4

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

LUT index	Prime numbers i	f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

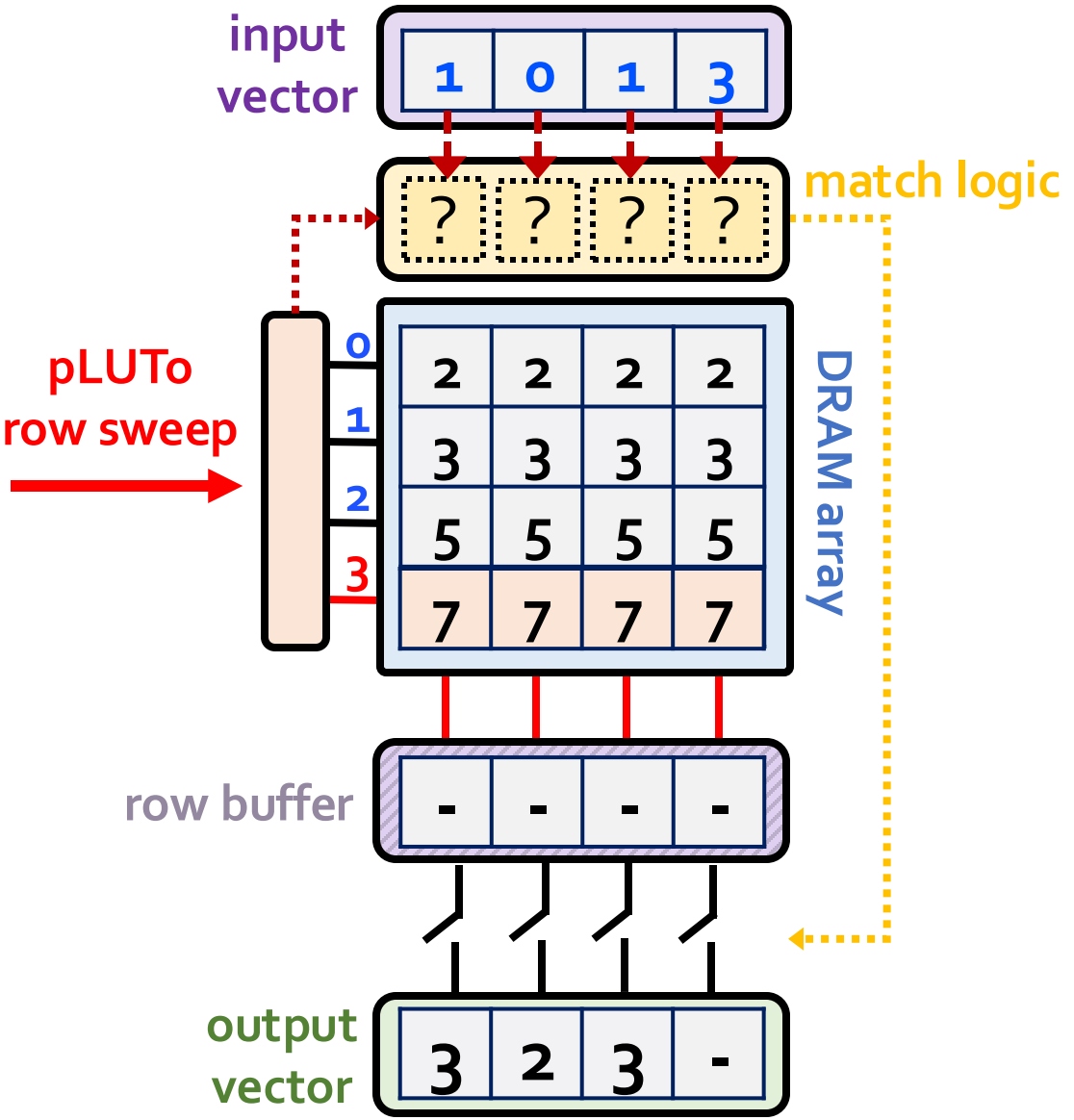
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Step 4

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

LUT index	Prime numbers i	f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

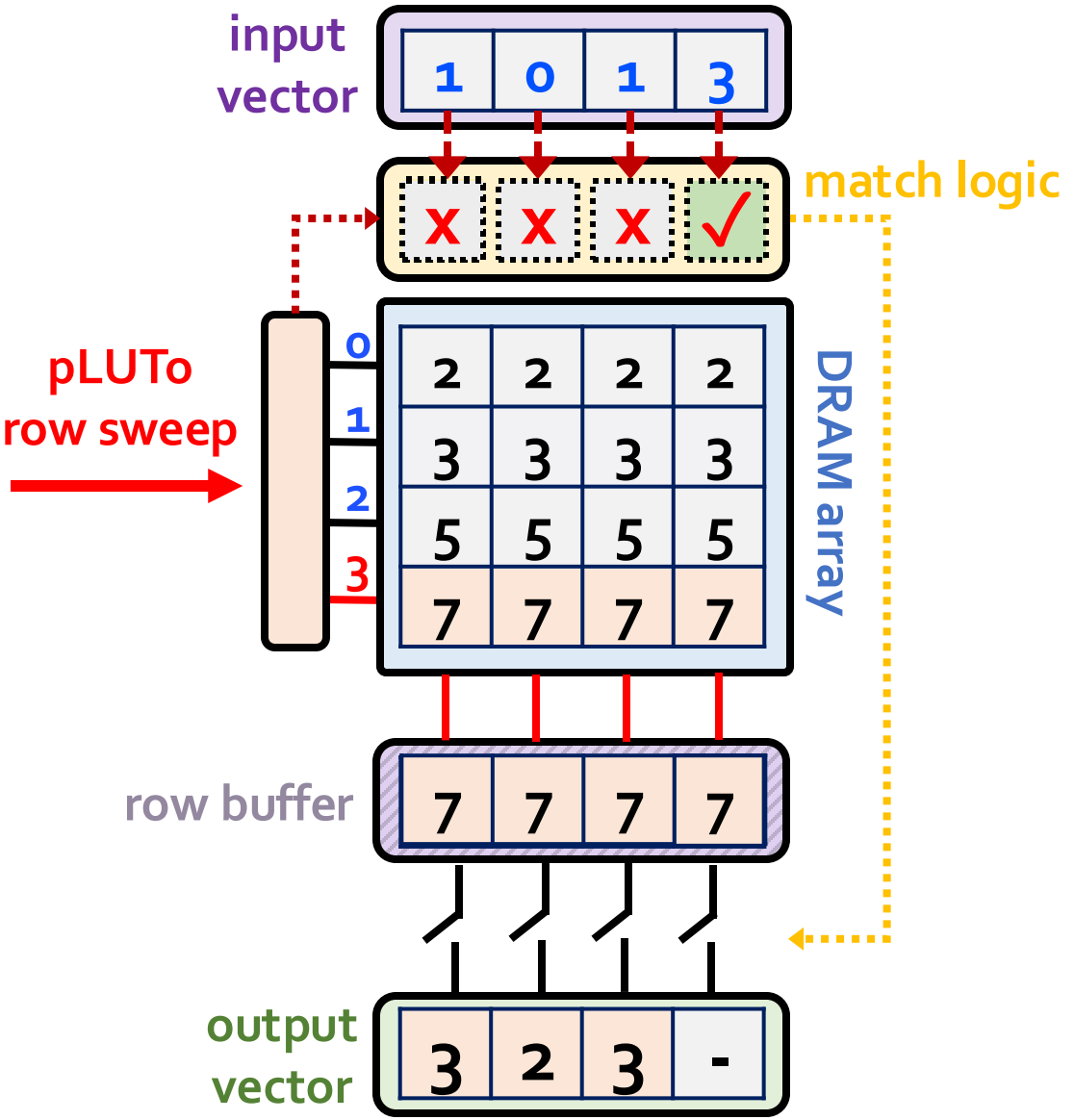
lookup table

1	0	1	3

input vector

3	2	3	7

output vector



In-DRAM pLUTo LUT Query: Step 4

LUT Query:
Return {2nd, 1st, 2nd, 4th}
prime numbers

LUT index	Prime numbers i	f(i)
0	1st	2
1	2nd	3
2	3rd	5
3	4th	7

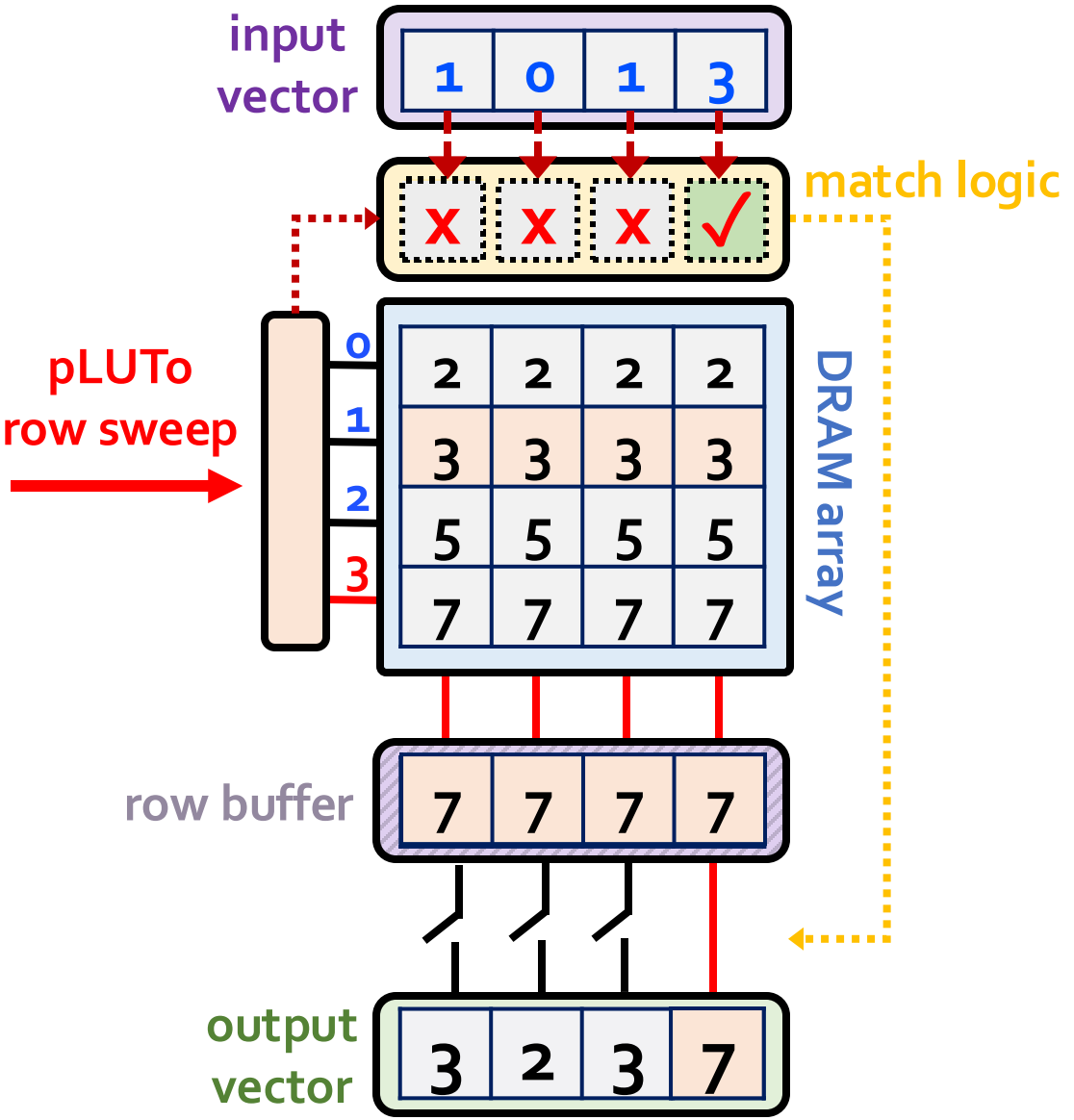
lookup table

1	0	1	3

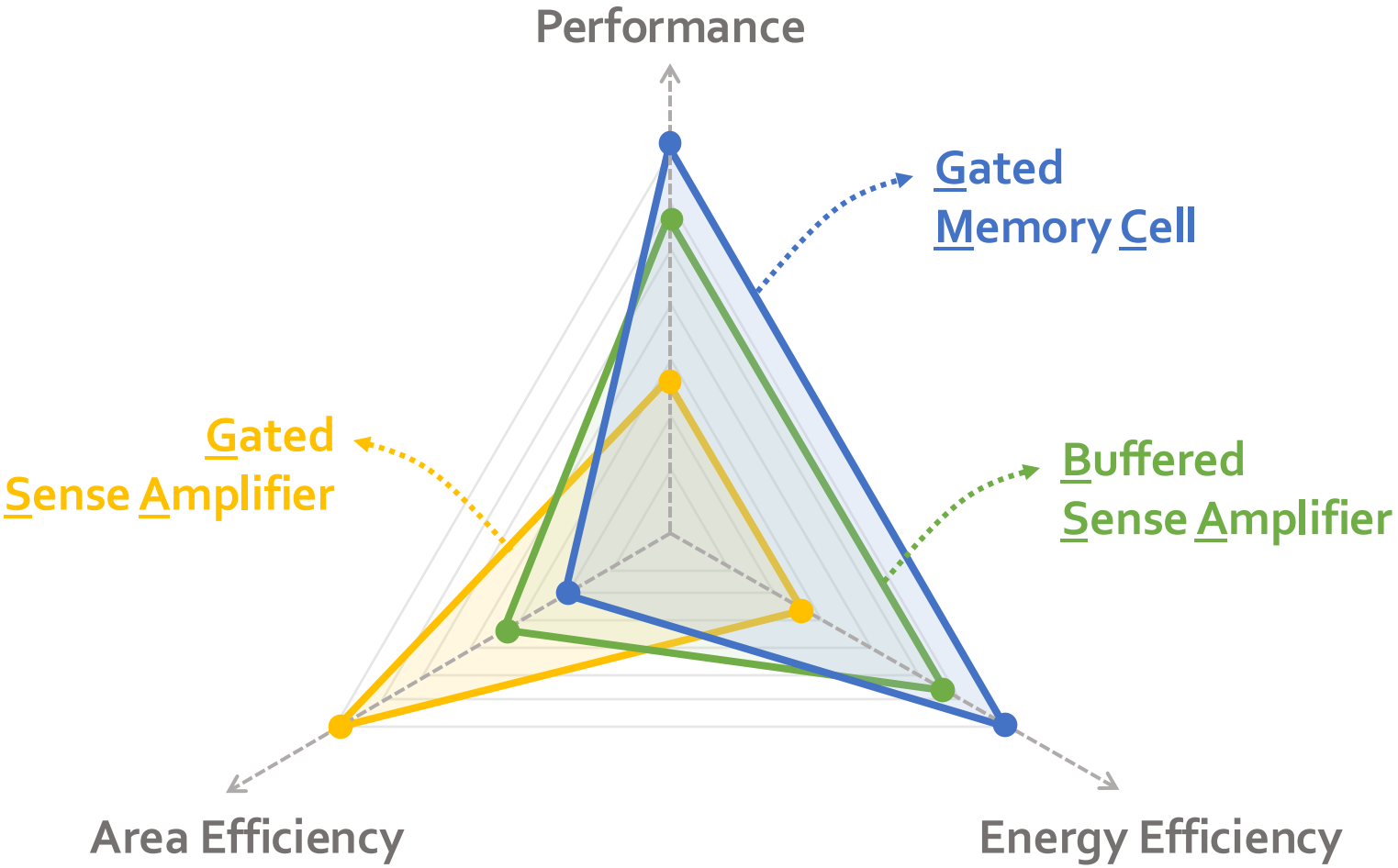
input vector

3	2	3	7

output vector

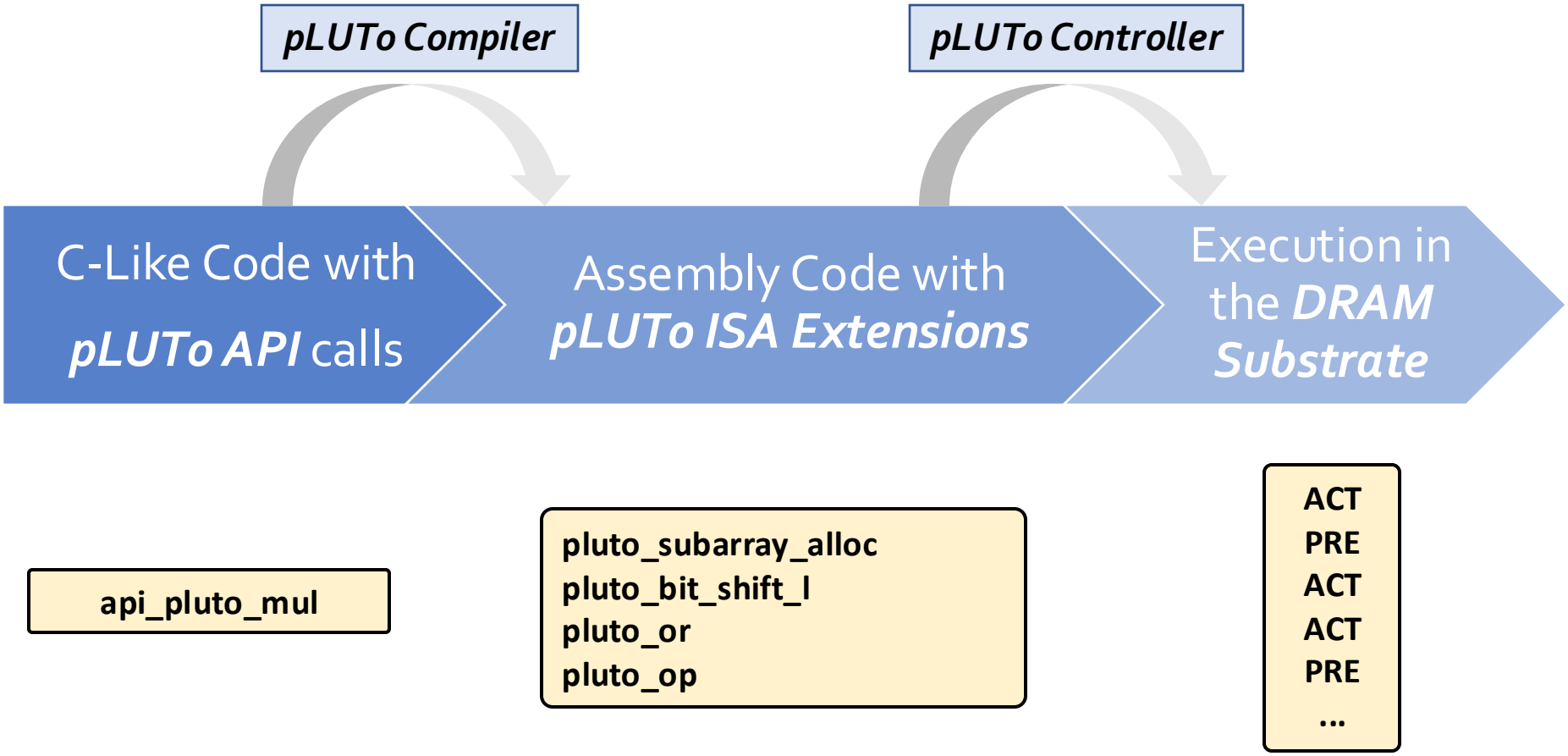


pLUTo Designs: Tradeoff Space

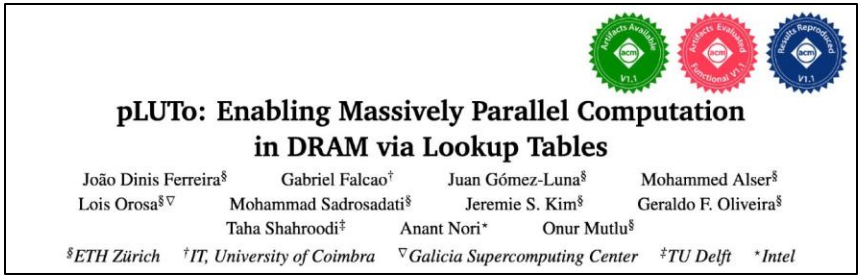


pLUTo designs cover a *broad design space* and provide *different* performance, energy, and area efficiency

System Integration



More in the Paper



pLUTo: Enabling Massively Parallel Computation in DRAM via Lookup Tables

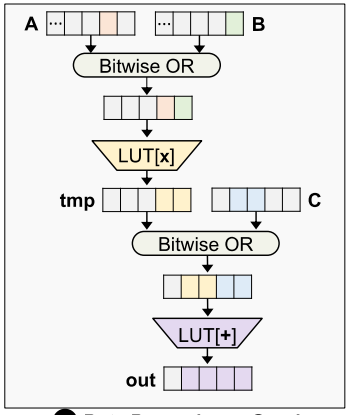
João Dinis Ferreira[§] Gabriel Falcao[†] Juan Gómez-Luna[§] Mohammed Alser[§]
Lois Orosa[§▽] Mohammad Sadrosadati[§] Jeremie S. Kim[§] Geraldo F. Oliveira[§]
Taha Shahroodi[‡] Anant Nori[⋆] Onur Mutlu[§]
[§]ETH Zürich [†]IT, University of Coimbra [▽]Galicia Supercomputing Center [‡]TU Delft [⋆]Intel

```c
uint2_t *A,*B,*C = (uint2_t *)malloc(input_size*2); // Inputs
uint4_t *out = (uint4_t *)malloc(input_size*4); // Output

// Array initialization
// ...
// Multiply-and-add loop
for (int i = 0; i < input_size; i++) {
      out[i] = A[i]*B[i] + C[i];
}
```

(a) Reference C Code

```c
// Array allocation
uint2_t *A, *B   = pluto_malloc(size=input_size, bitwidth=2);
uint2_t *C, *tmp = pluto_malloc(size=input_size, bitwidth=4);
uint4_t *out     = pluto_malloc(size=input_size, bitwidth=5);

// Multiply-and-add loop
for (int i = 0; i < input_size/row_size; i++){
   api_pluto_mul(in1 = A, in2 = B,   out = tmp, bitwidth = 2);
   api_pluto_add(in1 = C, in2 = tmp, out = out, bitwidth = 4);
}
```

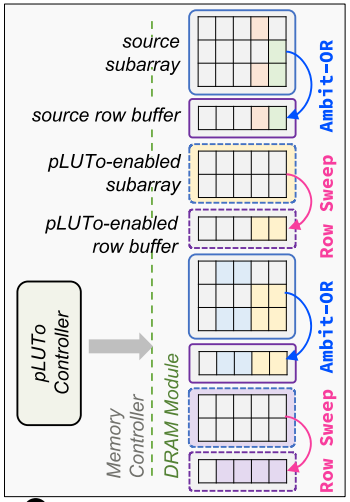
(b) pLUTo API Code



(d) Data Dependency Graph

```
# Array allocation
pluto_row_alloc $prg0, input_size, 2 # Allocate A
pluto_row_alloc $prg1, input_size, 2 # Allocate B
pluto_row_alloc $prg2, input_size, 4 # Allocate C
pluto_row_alloc $prg3, input_size, 4 # Allocate tmp
pluto_row_alloc $prg4, input_size, 5 # Allocate out

# Allocate and load LUTs
pluto_subarray_alloc $lut_rg0, "mul2_lut_file.dat"
pluto_subarray_alloc $lut_rg1, "add4_lut_file.dat"

# Allocate temporary row for OR operation
pluto_row_alloc $prg5, input_size, 8

# Multiply-and-add loop
div $r0, input_size, row_size # Initialize loop counter
LOOP:
   pluto_bit_shift_l $pgr0, 4 # Shift A 4 bits to the left
   pluto_or $prg5, $prg0, $prg1 # $prg5 <- A | B
   pluto_op $prg3, $prg5, $lut_rg0, 256, 4 # tmp <- LUT[A|B]
   pluto_bit_shift_l $pgr3, 4 # Shift tmp 4 bits to the left
   pluto_or $prg5, $prg3, $prg2 # $pgr5 <- tmp | C
   pluto_op $prg4, $prg5, $lut_rg1, 256, 8 # out <- LUT[tmp|C]
   # Update input addresses
   subi $r0, 0     # decrement loop counter
   bne $r0, LOOP   # next loop iteration
```

(c) pLUTo ISA Instructions

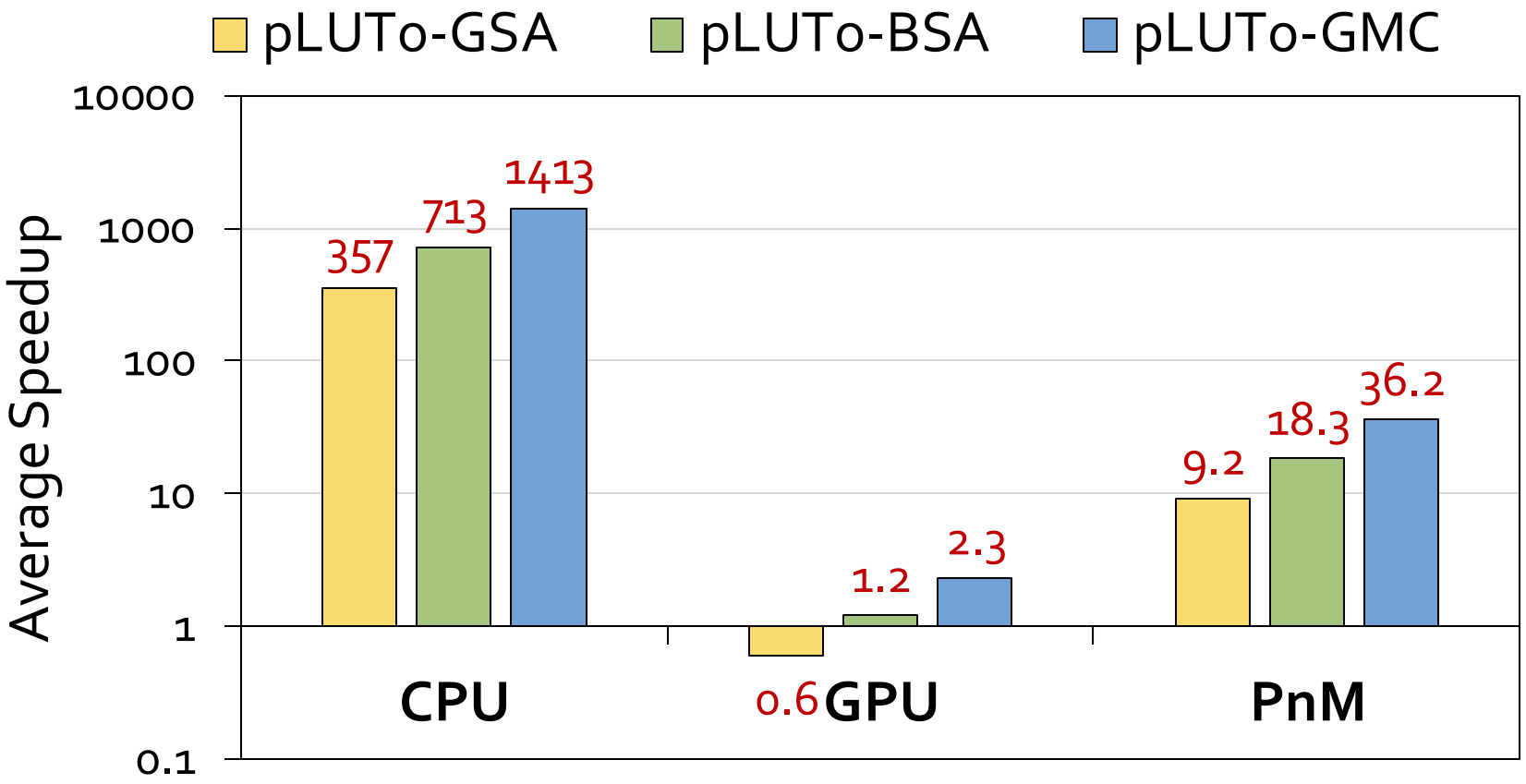


(e) pLUTo Controller & Execution

https://arxiv.org/pdf/2104.07699.pdf

Performance

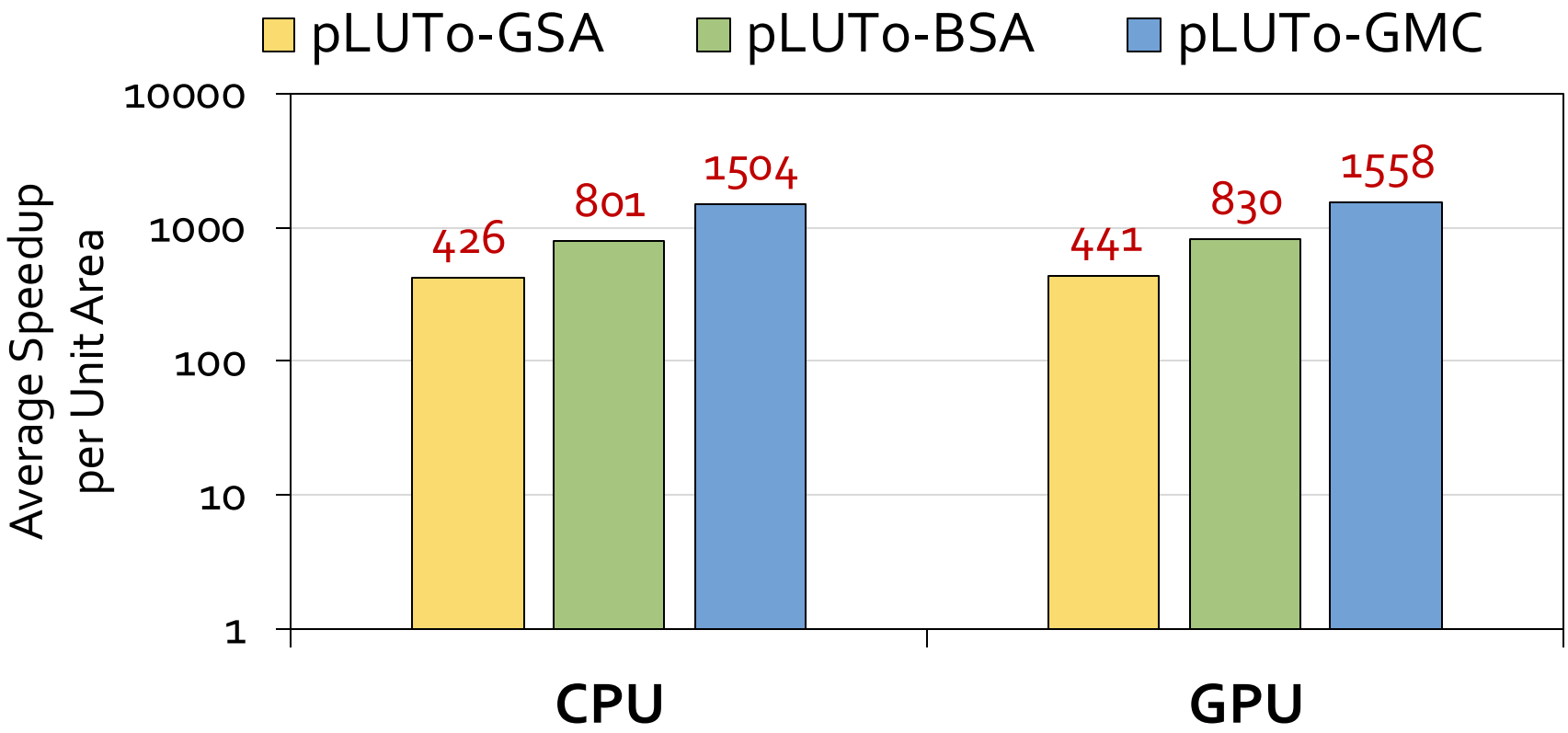
Average speedup across 7 real-world workloads



pLUTo *significantly outperforms* CPU, GPU and PnM baselines

Performance (normalized to area)

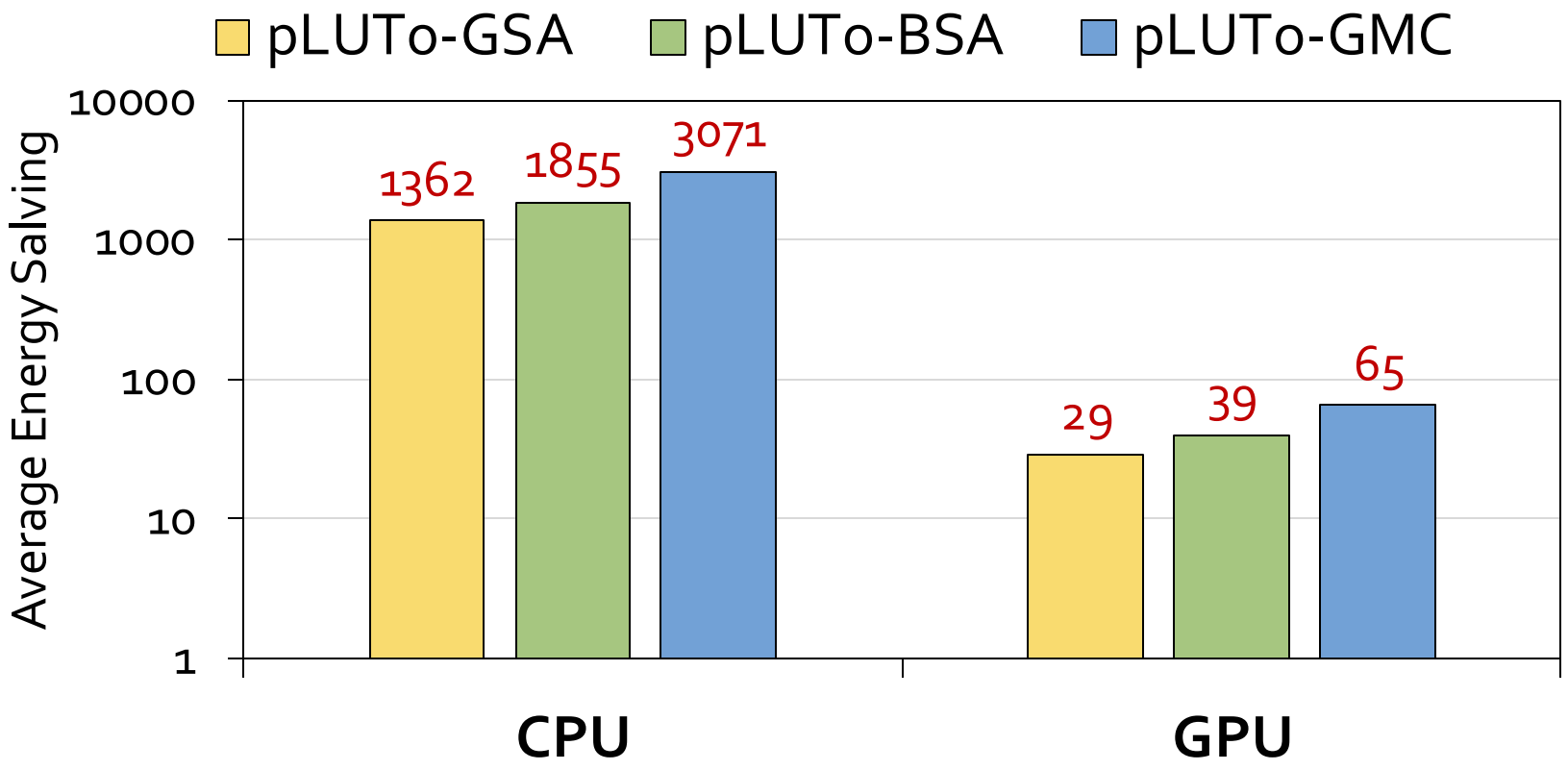
Average speedup normalized to area across 7 real-world workloads



pLUTo provides *substantially higher* performance per unit area than *both* the CPU and the GPU

Energy Consumption

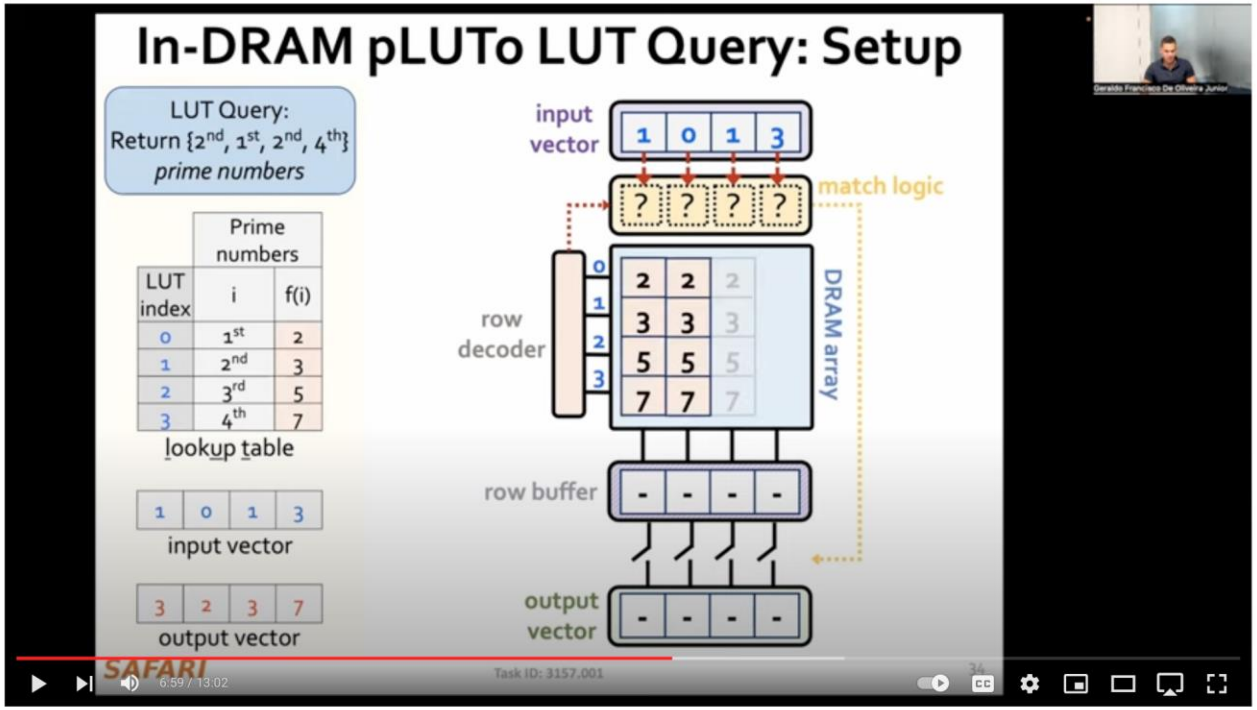
Average energy consumption across 7 real-world workloads



pLUTo *significantly reduces energy consumption* compared to processor-centric architectures for various workloads

SRC TECHCON Presentation

- Geraldo F. Oliveira
  - pLUTo: Enabling Massively Parallel Computation in DRAM via Lookup Tables
  - https://arxiv.org/pdf/2104.07699.pdf



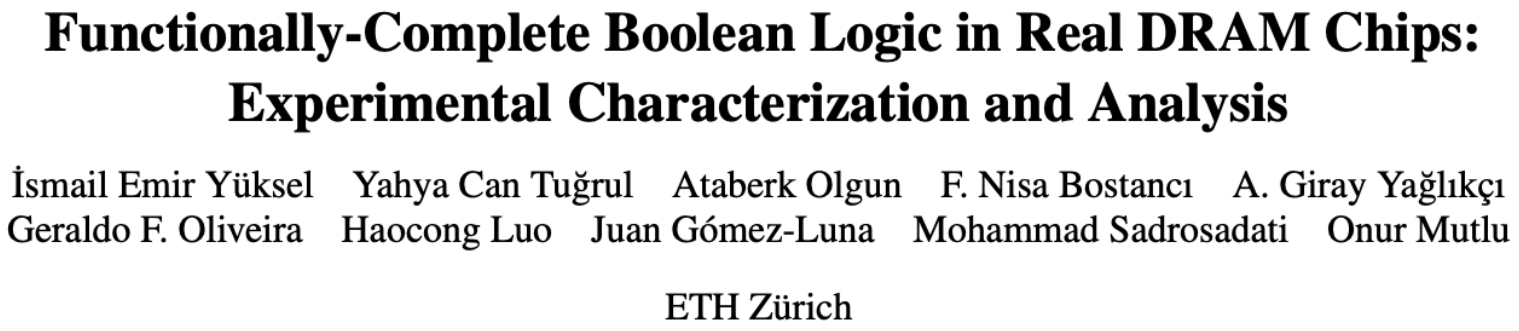
https://youtu.be/9t1FJQ6nNw4?si=bhylWCLZde2DC7os

Outline

- **Processing-Using-Memory (PUM) Solutions**
  - Review of DRAM Background, RowClone, and Ambit
  - SIMDRAM & MIMDRAM: Generalizing PUM Computing
  - pLUTo: Extending Operation Support with LUTs
  - **PUM in Off-the-Shelf DRAM Chips**
  - PUM Beyond Boolean/Arithmetic and DRAM
- Processing-Near-Memory (PNM) Solutions
  - Overview
  - Accelerating Neural Networks & Databases
  - Real PNM Systems
- Barriers for Adoption (and Current Solutions)
- Conclusion

Bulk Bitwise Operations in Real DRAM Chips

- Ismail Emir Yüksel, Yahya Can Tugrul Ataberk Olgun, F. Nisa Bostancı, A. Giray Yaglıkçı, Geraldo F. Oliveira, Haocong Luo, Juan Gómez-Luna, Mohammad Sadrosadati, Onur Mutlu, "**Functionally-Complete Boolean Logic in Real DRAM Chips: Experimental Characterization and Analysis**," *Proceedings of the 30th International Symposium on High-Performance Computer Architecture (**HPCA**)*, Edinburgh, Scotland, March 2024.



Functionally-Complete Boolean Logic in Real DRAM Chips: Experimental Characterization and Analysis

İsmail Emir Yüksel Yahya Can Tuğrul Ataberk Olgun F. Nisa Bostancı A. Giray Yağlıkçı
Geraldo F. Oliveira Haocong Luo Juan Gómez-Luna Mohammad Sadrosadati Onur Mutlu

ETH Zürich

https://arxiv.org/pdf/2402.18736

The Capability of COTS DRAM Chips

We **demonstrate** that **COTS DRAM chips:**

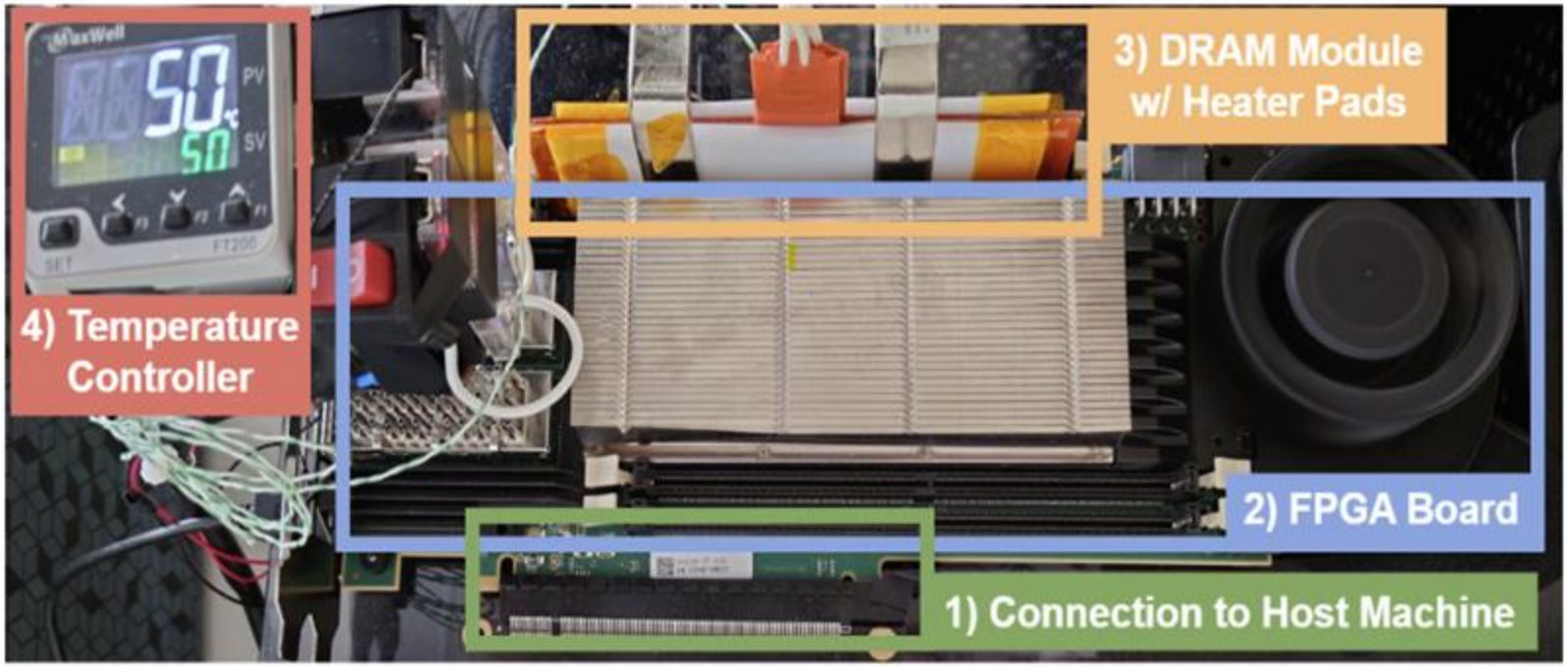
1 Can **simultaneously activate** up to **48 rows** in **two neighboring subarrays**

2 Can perform **NOT operation** with up to **32 output operands**

3 Can perform up to **16-input AND, NAND, OR, and NOR** operations

DRAM Testing Infrastructure

- Developed from DRAM Bender [Olgun+, TCAD'23]*
- Fine-grained control over DRAM commands, timings, and temperature



*Olgun et al., "DRAM Bender: An Extensible and Versatile FPGA-based Infrastructure to Easily Test State-of-the-art DRAM Chips," TCAD, 2023.

DRAM Chips Tested

- 256 DDR4 chips from two major DRAM manufacturers
- Covers different die revisions and chip densities

Chip Mfr.	#Modules (#Chips)	Die Rev.	Mfr. Date[a]	Chip Density	Chip Org.	Speed Rate
SK Hynix	9 (72)	M	N/A	4Gb	x8	2666MT/s
	5 (40)	A	N/A	4Gb	x8	2133MT/s
	1 (16)	A	N/A	8Gb	x8	2666MT/s
	1 (32)	A	18-14	4Gb	x4	2400MT/s
	1 (32)	A	16-49	8Gb	x4	2400MT/s
	1 (32)	M	16-22	8Gb	x4	2666MT/s
Samsung	1 (8)	F	21-02	4Gb	x8	2666MT/s
	2 (16)	D	21-10	8Gb	x8	2133MT/s
	1 (8)	A	22-12	8Gb	x8	3200MT/s

The Capability of COTS DRAM Chips

We **demonstrate** that **COTS DRAM chips:**

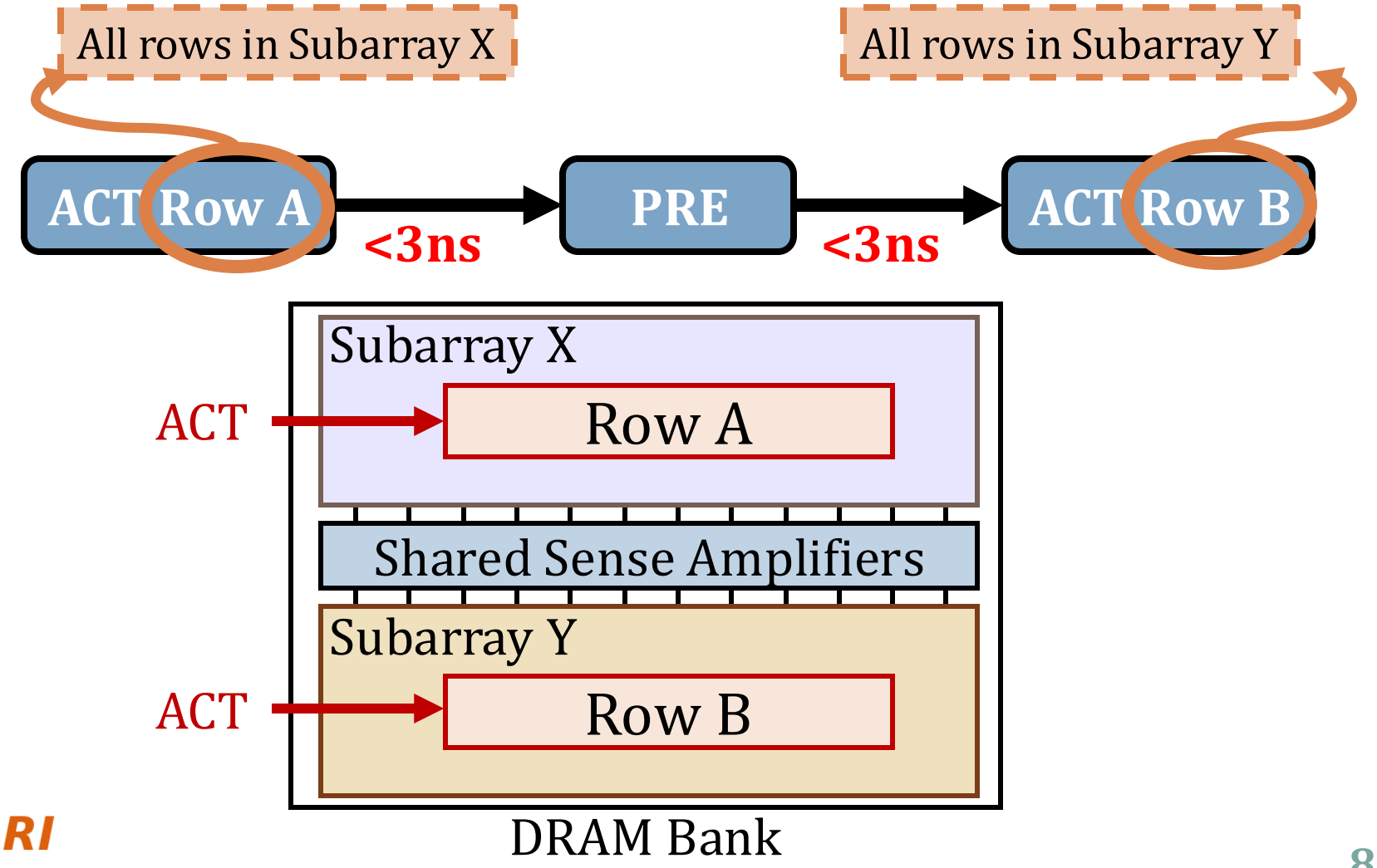
1 Can **simultaneously activate** up to **48 rows** in **two neighboring subarrays**

2 Can perform **NOT operation** with **up to 32** output operands

3 Can perform **up to 16-input AND, NAND, OR, and NOR** operations

Characterization Methodology

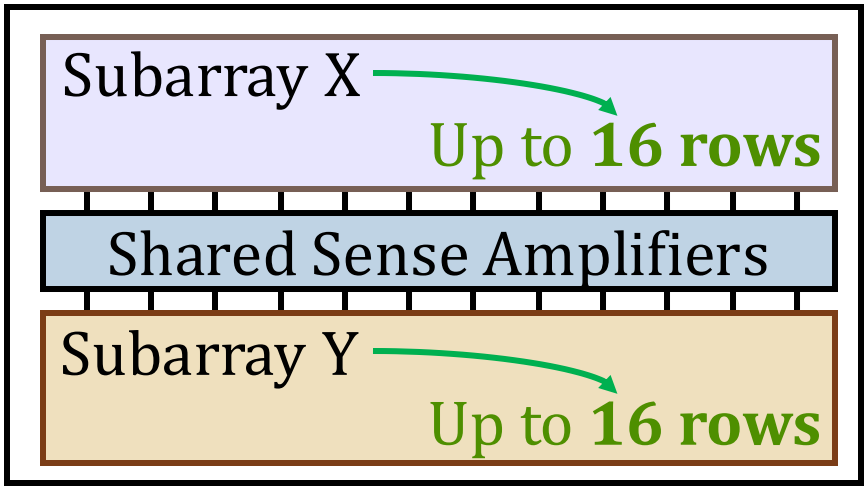
- To understand **which and how many** rows are simultaneously activated
  - **Sweep** Row A and Row B addresses



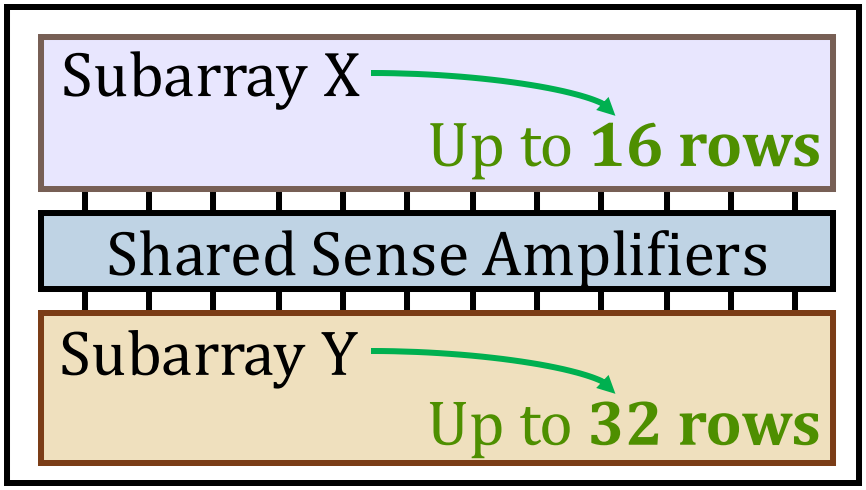
Key Results

COTS DRAM chips have **two distinct** sets of activation patterns in **neighboring subarrays** when two rows are activated with **violated timings**

Exactly the same number of rows in each subarray are activated	**Twice as many** rows in one subarray **compared to its neighbor subarray** are activated



A total of **32 rows**



A total of **48 rows**

The Capability of COTS DRAM Chips

We **demonstrate** that **COTS DRAM chips:**

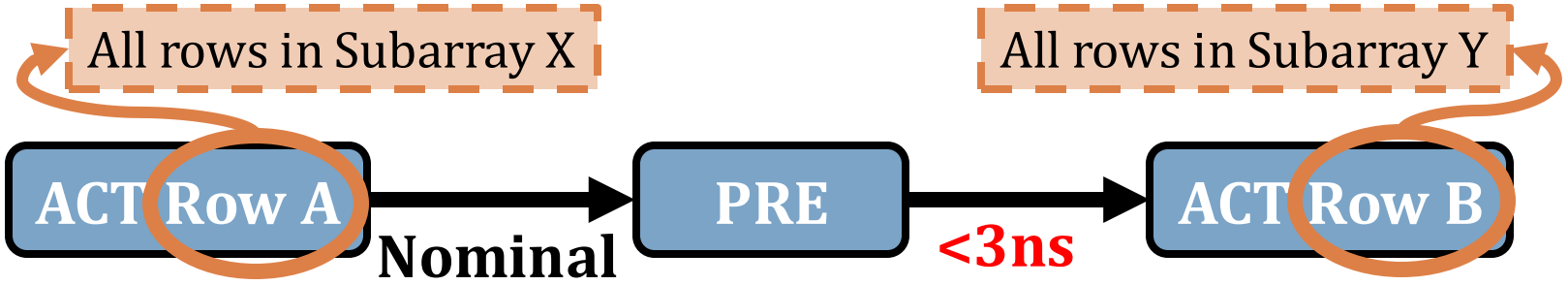
1. Can simultaneously activate up to 48 rows in two neighboring subarrays

2. Can perform **NOT operation** with **up to 32** output operands

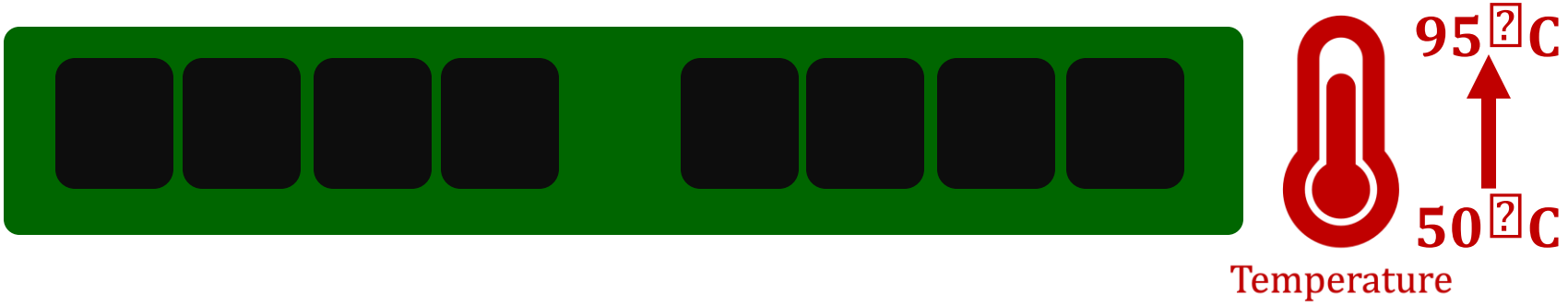
3. Can perform up to 16-input AND, NAND, OR, and NOR operations

Characterization Methodology

- Sweep **Row A and Row B addresses**



- Sweep **DRAM chip temperature**



Key Takeaways from In-DRAM NOT Operation

Key Takeaway 1

COTS DRAM chips can perform NOT operations with up to 32 destination rows

Key Takeaway 2

Temperature has a small effect on the reliability of NOT operations

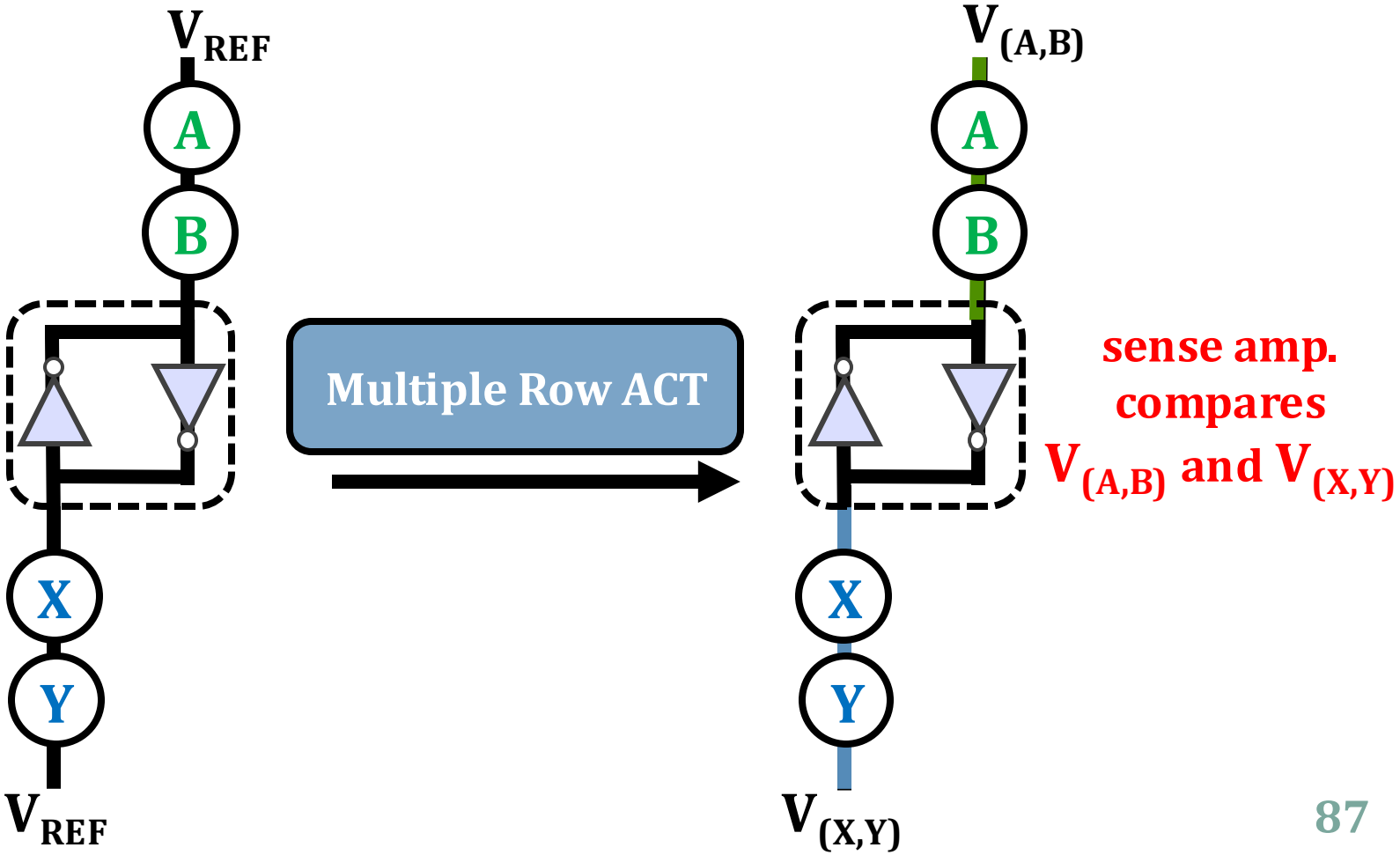
The Capability of COTS DRAM Chips

We **demonstrate** that **COTS DRAM chips:**

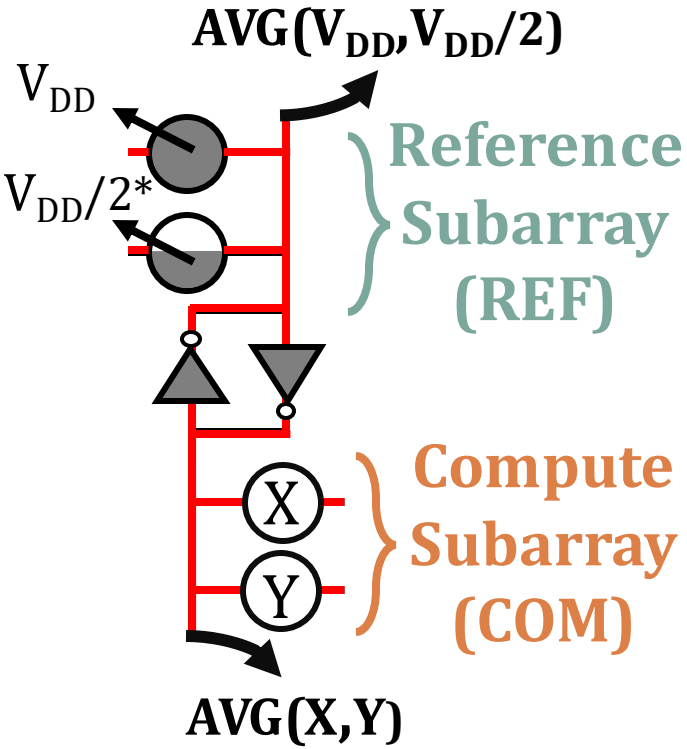
1. Can **simultaneously activate** up to **48 rows in two neighboring subarrays**
2. Can perform **NOT operation** with **up to 32** output operands
3. Can perform **up to 16-input AND, NAND, OR, and NOR** operations

Key Idea

Manipulate the bitline voltage to express
a wide variety of functions using
multiple-row activation in neighboring subarrays

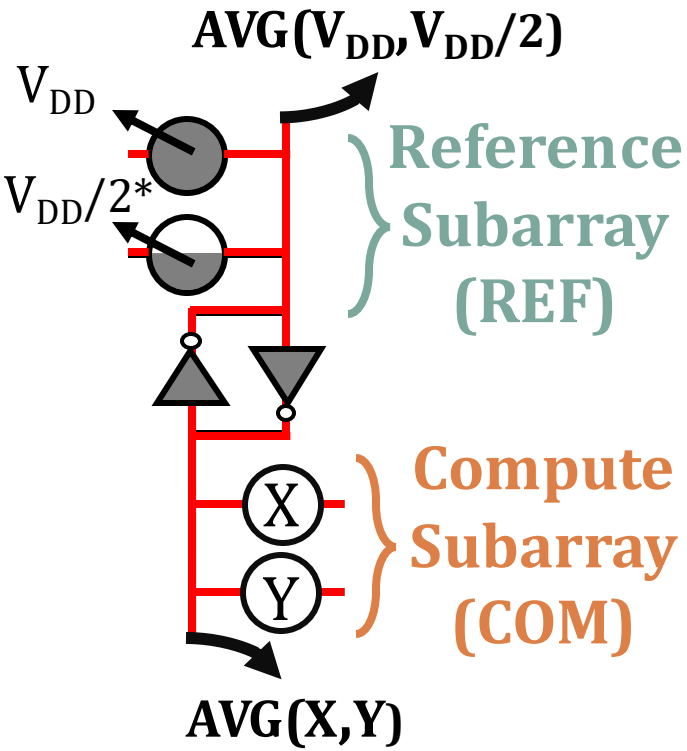


Two-Input AND and NAND Operations



*Gao et al., "FracDRAM: Fractional Values in Off-the-Shelf DRAM," in MICRO, 2022.

Two-Input AND and NAND Operations



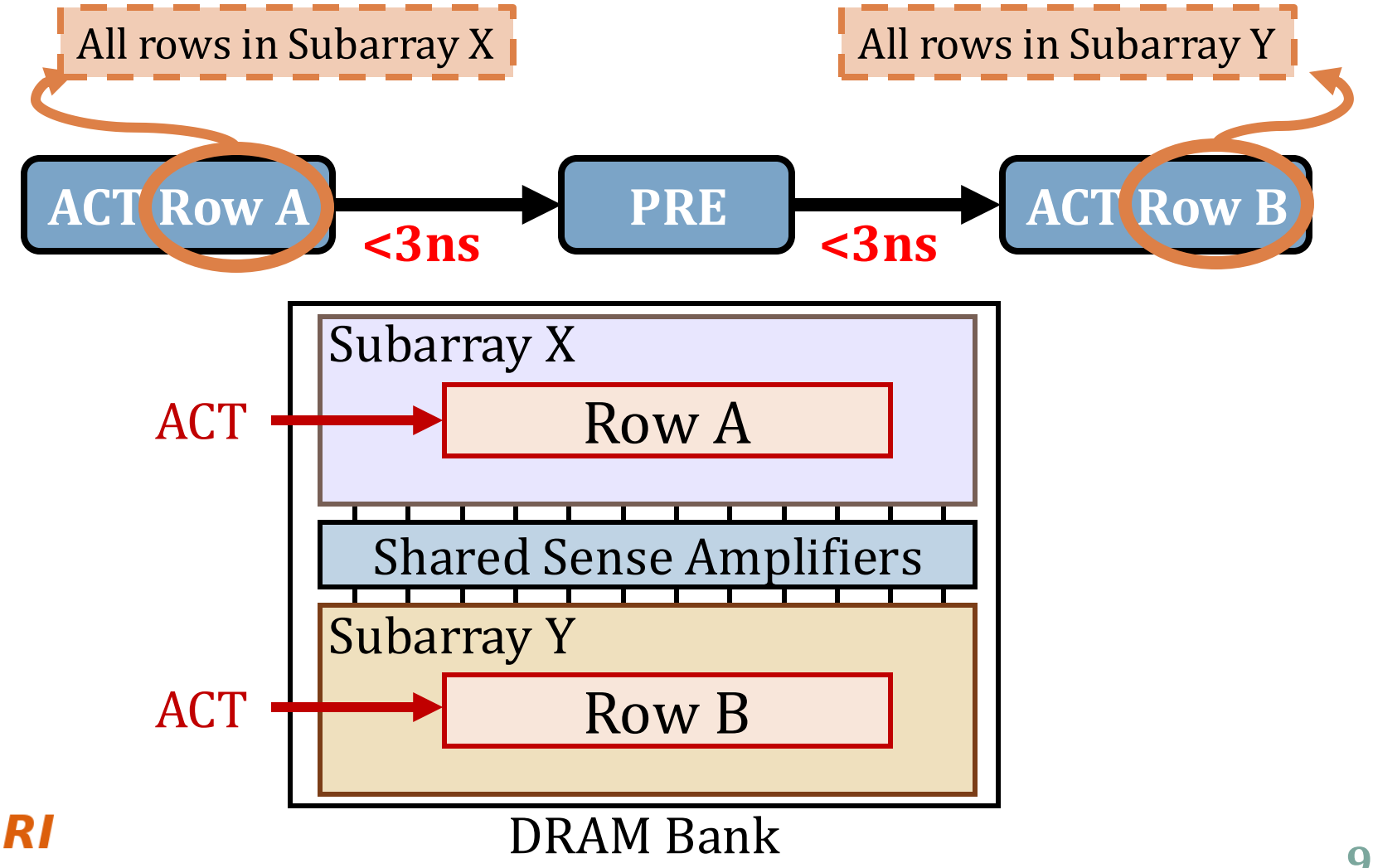
V_{DD}=1 & GND = 0

X	Y	COM	REF
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0
		AND	NAND

*Gao et al., "FracDRAM: Fractional Values in Off-the-Shelf DRAM," in MICRO, 2022.

Characterization Methodology

- **Sweep** Row A and Row B addresses



Key Takeaways from In-DRAM Operations

Key Takeaway 1

COTS DRAM chips can perform {2, 4, 8, 16}-input AND, NAND, OR, and NOR operations

Key Takeaway 2

COTS DRAM chips can perform AND, NAND, OR, and NOR operations with very high reliability

Key Takeaway 3

Data pattern slightly affects the reliability of AND, NAND, OR, and NOR operations

Real Processing Using Memory Prototype

- End-to-end RowClone & TRNG using off-the-shelf DRAM chips
- Idea: Violate DRAM timing parameters to mimic RowClone

PiDRAM: A Holistic End-to-end FPGA-based Framework for Processing-in-DRAM

Ataberk Olgun[§†] Juan Gómez Luna[§] Konstantinos Kanellopoulos[§] Behzad Salami[§*]
Hasan Hassan[§] Oğuz Ergin[†] Onur Mutlu[§]

[§]ETH Zürich [†]TOBB ETÜ [*]BSC

https://arxiv.org/pdf/2111.00082.pdf

https://github.com/cmu-safari/pidram

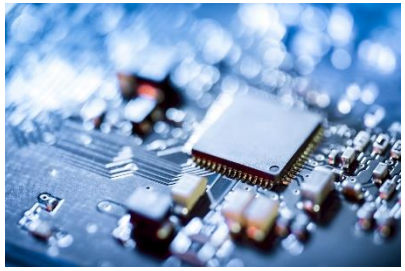
https://www.youtube.com/watch?v=qeukNs5XI3g&t=4192s

PiDRAM

Goal: Develop a **flexible** platform to explore **end-to-end** implementations of PuM techniques

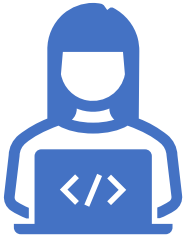
- Enable rapid integration via key components

Hardware



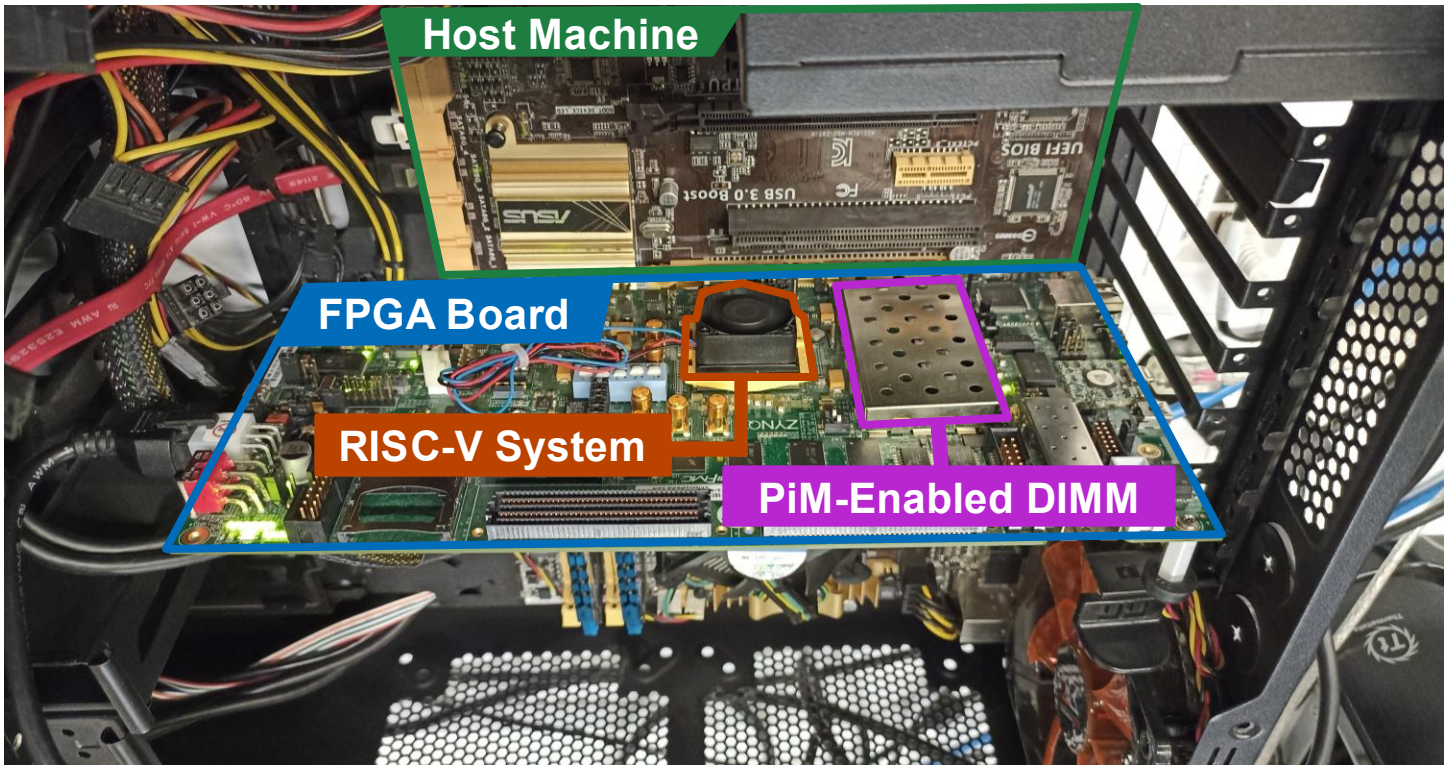
1. Easy-to-extend Memory Controller
2. ISA-transparent PuM Controller

Software



1. Extensible Software Library
2. Custom Supervisor Software

Real Processing Using Memory Prototype

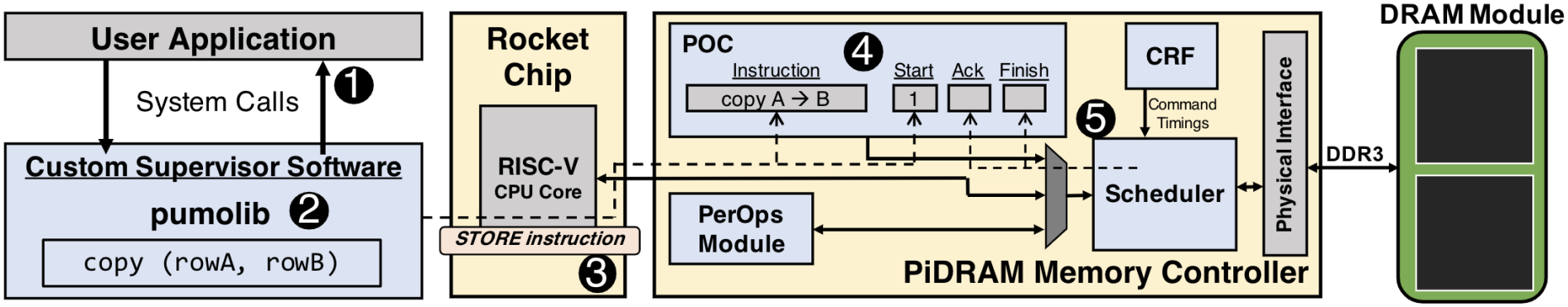


https://arxiv.org/pdf/2111.00082.pdf

https://github.com/cmu-safari/pidram

https://www.youtube.com/watch?v=qeukNs5XI3g&t=4192s

PiDRAM Workflow



1- User application interfaces with the OS via system calls

2- OS uses PuM Operations Library (pumolib) to convey operation related information to the hardware *using*

3- STORE instructions that target the memory mapped registers of the PuM Operations Controller (POC)

4- POC oversees the execution of a PuM operation (e.g., RowClone, bulk bitwise operations)

5- Scheduler arbitrates between regular (load, store) and PuM operations and issues DRAM commands with custom timings

Real Processing Using Memory Prototype

README.md

Building a PiDRAM Prototype

To build PiDRAM's prototype on Xilinx ZC706 boards, developers need to use the two sub-projects in this directory. `fpga-zynq` is a repository branched off of UCB-BAR's fpga-zynq repository. We use `fpga-zynq` to generate rocket chip designs that support end-to-end DRAM PuM execution. `controller-hardware` is where we keep the main Vivado project and Verilog sources for PiDRAM's memory controller and the top level system design.

Rebuilding Steps

1. Navigate into `fpga-zynq` and read the README file to understand the overall workflow of the repository
   - Follow the readme in `fpga-zynq/rocket-chip/riscv-tools` to install dependencies
2. Create the Verilog source of the rocket chip design using the `ZynqCopyFPGAConfig`
   - Navigate into zc706, then run `make rocket CONFIG=ZynqCopyFPGAConfig -j<number of cores>`
3. Copy the generated Verilog file (should be under zc706/src) and overwrite the same file in `controller-hardware/source/hdl/impl/rocket-chip`
4. Open the Vivado project in `controller-hardware/Vivado_Project` using Vivado 2016.2
5. Generate a bitstream
6. Copy the bitstream (system_top.bit) to `fpga-zynq/zc706`
7. Use the `./build_script.sh` to generate the new `boot.bin` under `fpga-images-zc706`, you can use this file to program the FPGA using the SD-Card
   - For details, follow the relevant instructions in `fpga-zynq/README.md`

You can run programs compiled with the RISC-V Toolchain supplied within the `fpga-zynq` repository. To install the toolchain, follow the instructions under `fpga-zynq/rocket-chip/riscv-tools`.

Generating DDR3 Controller IP sources

We cannot provide the sources for the Xilinx PHY IP we use in PiDRAM's memory controller due to licensing issues. We describe here how to regenerate them using Vivado 2016.2. First, you need to generate the IP RTL files:

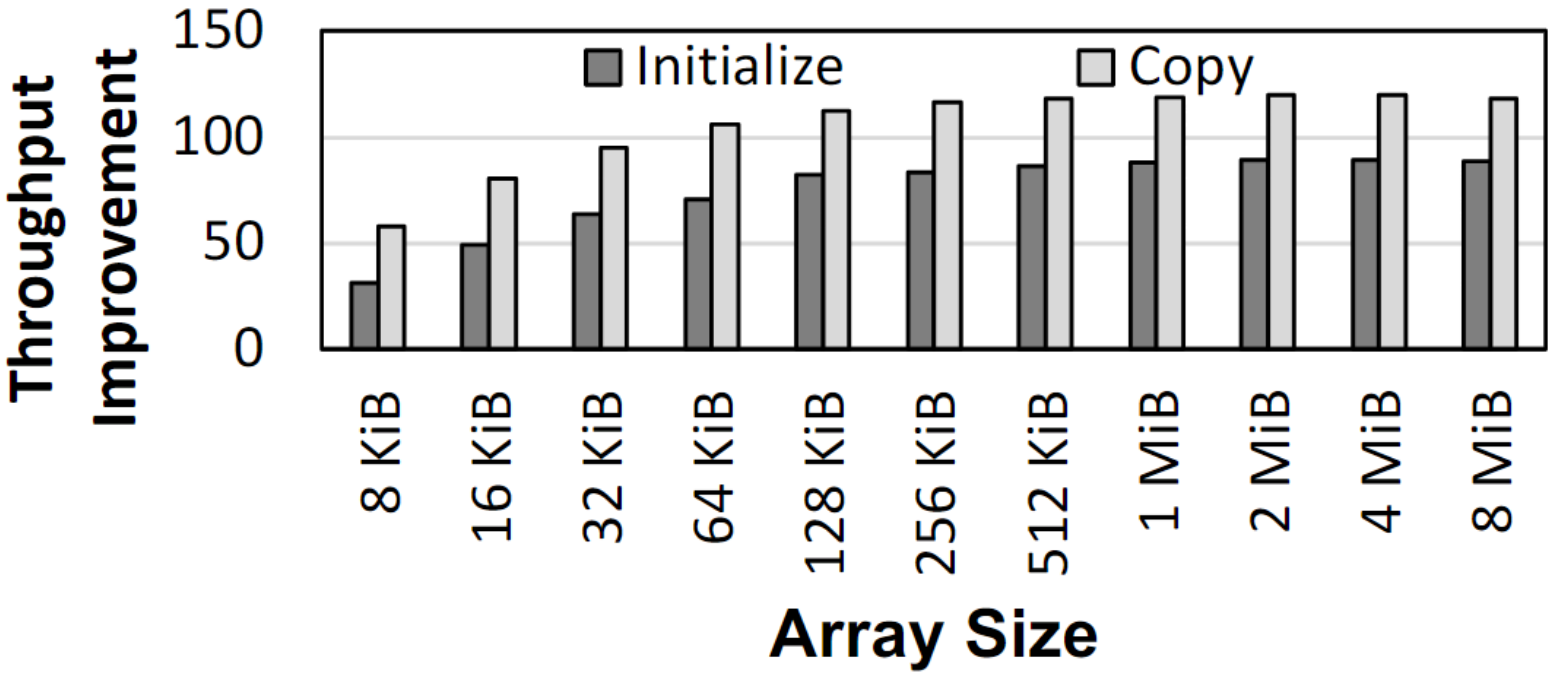
1- Open IP Catalog
2- Find "Memory Interface Generator (MIG 7 Series)" IP and double click

https://arxiv.org/pdf/2111.00082.pdf

https://github.com/cmu-safari/pidram

https://www.youtube.com/watch?v=qeukNs5XI3g&t=4192s

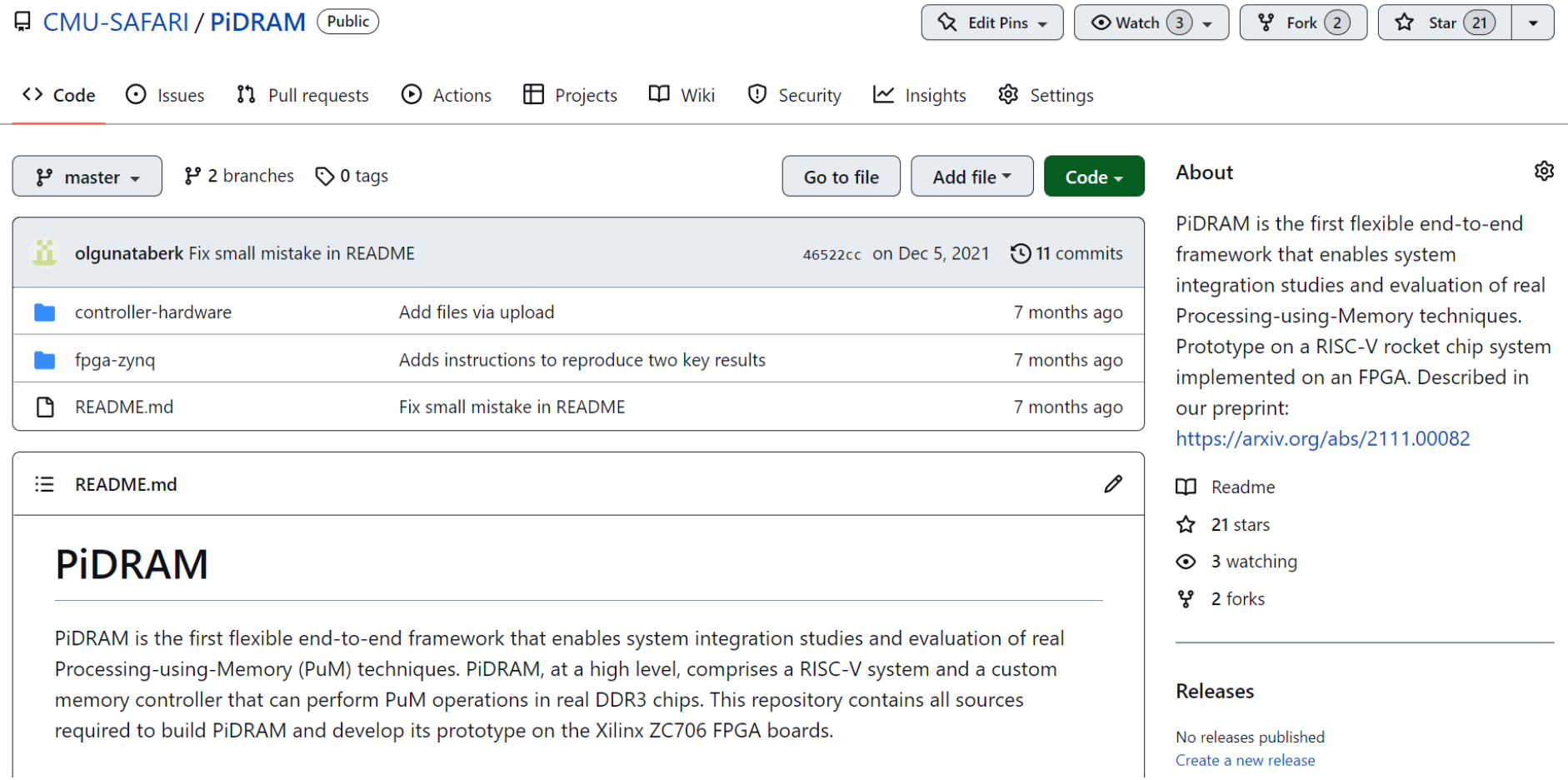
Microbenchmark Copy/Initialization Throughput



In-DRAM Copy and Initialization improve throughput by 119x and 89x

PiDRAM is Open Source

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



Extended Version on ArXiv

https://arxiv.org/abs/2111.00082

Computer Science > Hardware Architecture

[Submitted on 29 Oct 2021 (v1), last revised 19 Dec 2021 (this version, v3)]

PiDRAM: A Holistic End-to-end FPGA-based Framework for Processing-in-DRAM

Ataberk Olgun, Juan Gómez Luna, Konstantinos Kanellopoulos, Behzad Salami, Hasan Hassan, Oğuz Ergin, Onur Mutlu

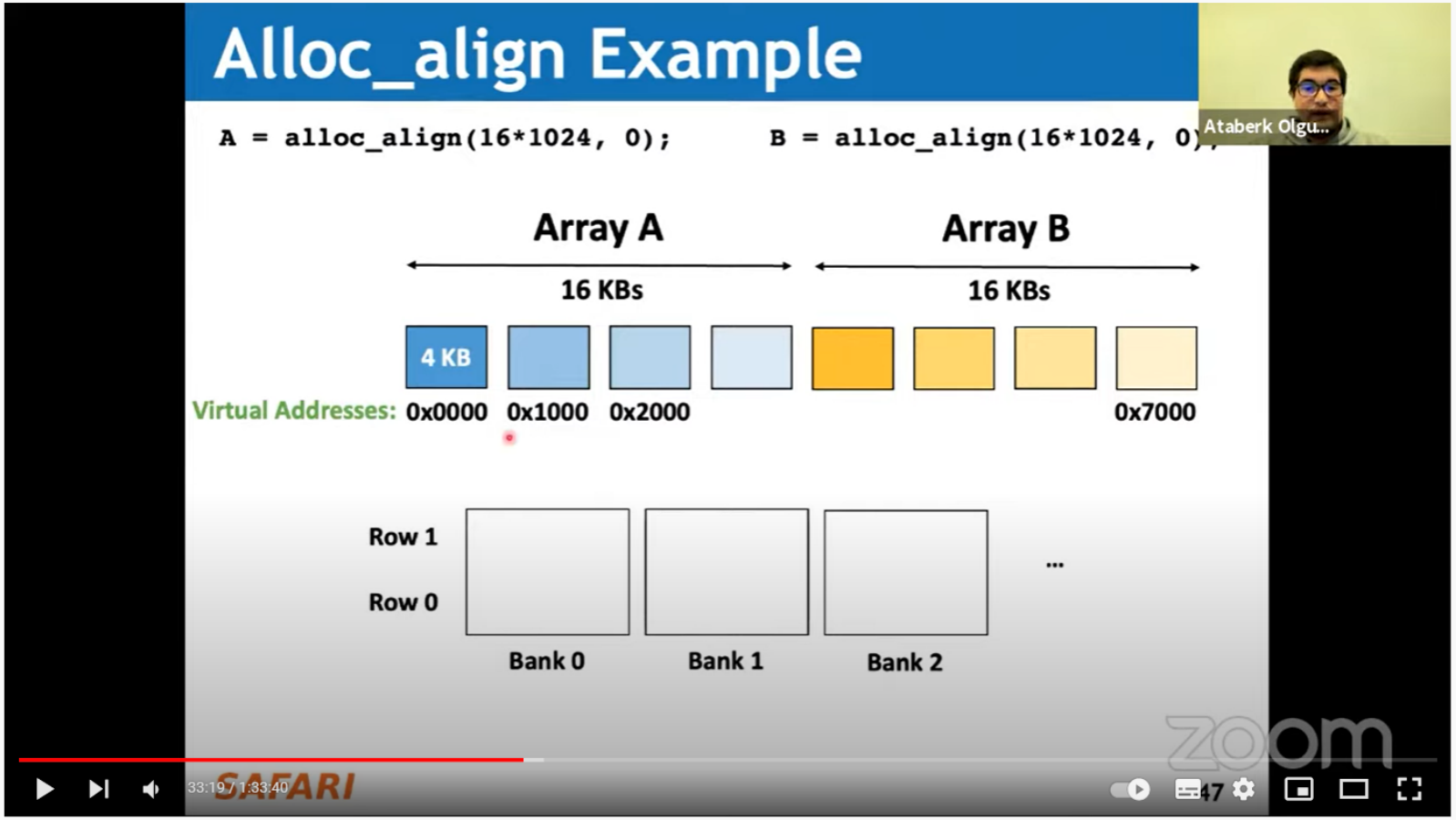
Processing-using-memory (PuM) techniques leverage the analog operation of memory cells to perform computation. Several recent works have demonstrated PuM techniques in off-the-shelf DRAM devices. Since DRAM is the dominant memory technology as main memory in current computing systems, these PuM techniques represent an opportunity for alleviating the data movement bottleneck at very low cost. However, system integration of PuM techniques imposes non-trivial challenges that are yet to be solved. Design space exploration of potential solutions to the PuM integration challenges requires appropriate tools to develop necessary hardware and software components. Unfortunately, current specialized DRAM-testing platforms, or system simulators do not provide the flexibility and/or the holistic system view that is necessary to deal with PuM integration challenges.

We design and develop PiDRAM, the first flexible end-to-end framework that enables system integration studies and evaluation of real PuM techniques. PiDRAM provides software and hardware components to rapidly integrate PuM techniques across the whole system software and hardware stack (e.g., necessary modifications in the operating system, memory controller). We implement PiDRAM on an FPGA-based platform along with an open-source RISC-V system. Using PiDRAM, we implement and evaluate two state-of-the-art PuM techniques: in-DRAM (i) copy and initialization, (ii) true random number generation. Our results show that the in-memory copy and initialization techniques can improve the performance of bulk copy operations by 12.6x and bulk initialization operations by 14.6x on a real system. Implementing the true random number generator requires only 190 lines of Verilog and 74 lines of C code using PiDRAM's software and hardware components.

Comments: 15 pages, 12 figures
Subjects: **Hardware Architecture (cs.AR)**
Cite as: arXiv:2111.00082 **[cs.AR]**
(or arXiv:2111.00082v3 **[cs.AR]** for this version)
https://doi.org/10.48550/arXiv.2111.00082

Long Talk + Tutorial on Youtube

https://youtu.be/s_z_S6FYpC8



Processing in Memory Course: Meeting 6: End-to-end Framework for Processing-using-Memory - Fall'21

615 views • Streamed live on 9 Nov 2021 • Project & Seminar, ETH Zürich, Fall 202 Show more

25 Dislike Share Download Clip Save ...

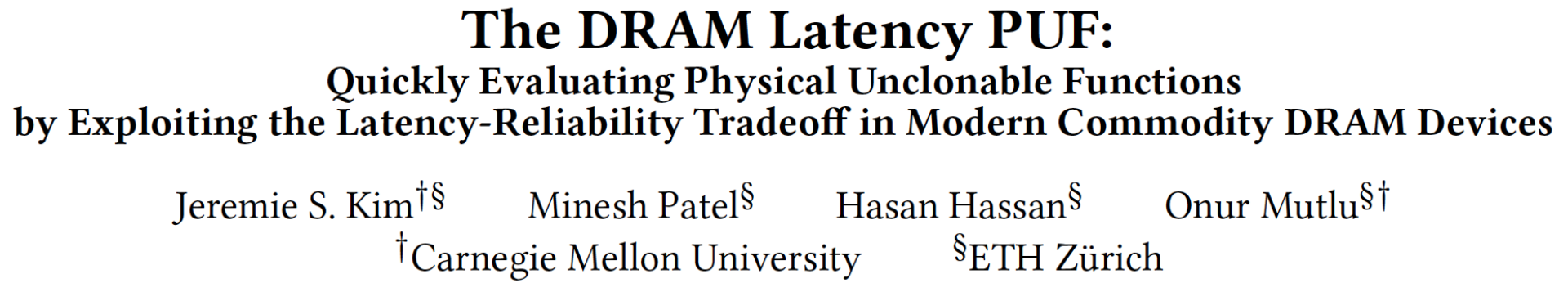
Onur Mutlu Lectures
25.7K subscribers
SUBSCRIBED

Outline

- **Processing-Using-Memory (PUM) Solutions**
  - Review of DRAM Background, RowClone, and Ambit
  - SIMDRAM & MIMDRAM: Generalizing PUM Computing
  - pLUTo: Extending Operation Support with LUTs
  - PUM in Off-the-Shelf DRAM Chips
  - **PUM Beyond Boolean/Arithmetic and DRAM**
- Processing-Near-Memory (PNM) Solutions
  - Overview
  - Accelerating Neural Networks & Databases
  - Real PNM Systems
- Barriers for Adoption (and Current Solutions)
- Conclusion

In-DRAM Physical Unclonable Functions

- Jeremie S. Kim, Minesh Patel, Hasan Hassan, and Onur Mutlu,
  "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern DRAM Devices"
  *Proceedings of the 24th International Symposium on High-Performance Computer Architecture (**HPCA**)*, Vienna, Austria, February 2018.
  [Lightning Talk Video]
  [Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)]
  [Full Talk Lecture Video (28 minutes)]

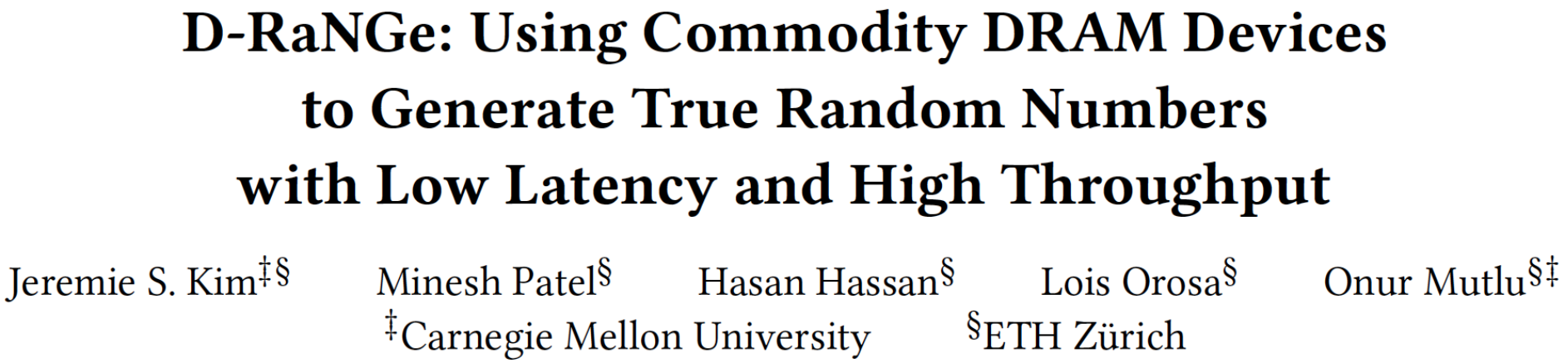


The DRAM Latency PUF:
Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern Commodity DRAM Devices

Jeremie S. Kim[†§] Minesh Patel[§] Hasan Hassan[§] Onur Mutlu[§†]
[†]Carnegie Mellon University [§]ETH Zürich

In-DRAM True Random Number Generation

- Jeremie S. Kim, Minesh Patel, Hasan Hassan, Lois Orosa, and Onur Mutlu,
  "D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput"
  *Proceedings of the 25th International Symposium on High-Performance Computer Architecture (**HPCA**)*, Washington, DC, USA, February 2019.
  [Slides (pptx) (pdf)]
  [Full Talk Video (21 minutes)]
  [Full Talk Lecture Video (27 minutes)]
  Top Picks Honorable Mention by IEEE Micro.



D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput

Jeremie S. Kim[‡§] Minesh Patel[§] Hasan Hassan[§] Lois Orosa[§] Onur Mutlu[§‡]
[‡]Carnegie Mellon University [§]ETH Zürich

In-DRAM True Random Number Generation

- Ataberk Olgun, Minesh Patel, A. Giray Yaglikci, Haocong Luo, Jeremie S. Kim, F. Nisa Bostanci, Nandita Vijaykumar, Oguz Ergin, and Onur Mutlu,
  "QUAC-TRNG: High-Throughput True Random Number Generation Using Quadruple Row Activation in Commodity DRAM Chips"
  *Proceedings of the 48th International Symposium on Computer Architecture (**ISCA**)*, Virtual, June 2021.
  [Slides (pptx) (pdf)]
  [Short Talk Slides (pptx) (pdf)]
  [Talk Video (25 minutes)]
  [SAFARI Live Seminar Video (1 hr 26 mins)]

QUAC-TRNG: High-Throughput True Random Number Generation Using Quadruple Row Activation in Commodity DRAM Chips

Ataberk Olgun[§†] Minesh Patel[§] A. Giray Yağlıkçı[§] Haocong Luo[§]
Jeremie S. Kim[§] F. Nisa Bostancı[§†] Nandita Vijaykumar[§⊙] Oğuz Ergin[†] Onur Mutlu[§]

[§]*ETH Zürich* [†]*TOBB University of Economics and Technology* [⊙]*University of Toronto*

In-DRAM True Random Number Generation

- F. Nisa Bostanci, Ataberk Olgun, Lois Orosa, A. Giray Yaglikci, Jeremie S. Kim, Hasan Hassan, Oguz Ergin, and Onur Mutlu,
  "DR-STRaNGe: End-to-End System Design for DRAM-based True Random Number Generators"
  *Proceedings of the 28th International Symposium on High-Performance Computer Architecture (**HPCA**)*, Virtual, April 2022.
  [Slides (pptx) (pdf)]
  [Short Talk Slides (pptx) (pdf)]

DR-STRaNGe: End-to-End System Design for DRAM-based True Random Number Generators

F. Nisa Bostancı[†§] Ataberk Olgun[†§] Lois Orosa[§] A. Giray Yağlıkçı[§]
Jeremie S. Kim[§] Hasan Hassan[§] Oğuz Ergin[†] Onur Mutlu[§]

[†]*TOBB University of Economics and Technology* [§]*ETH Zürich*

https://arxiv.org/pdf/2201.01385.pdf

Pinatubo: RowClone and Bitwise Ops in PCM

Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-volatile Memories

Shuangchen Li[1]*, Cong Xu[2], Qiaosha Zou[1,5], Jishen Zhao[3], Yu Lu[4], and Yuan Xie[1]

University of California, Santa Barbara[1], Hewlett Packard Labs[2]
University of California, Santa Cruz[3], Qualcomm Inc.[4], Huawei Technologies Inc.[5]
{shuangchenli, yuanxie}ece.ucsb.edu[1]

Pinatubo: RowClone and Bitwise Ops in PCM

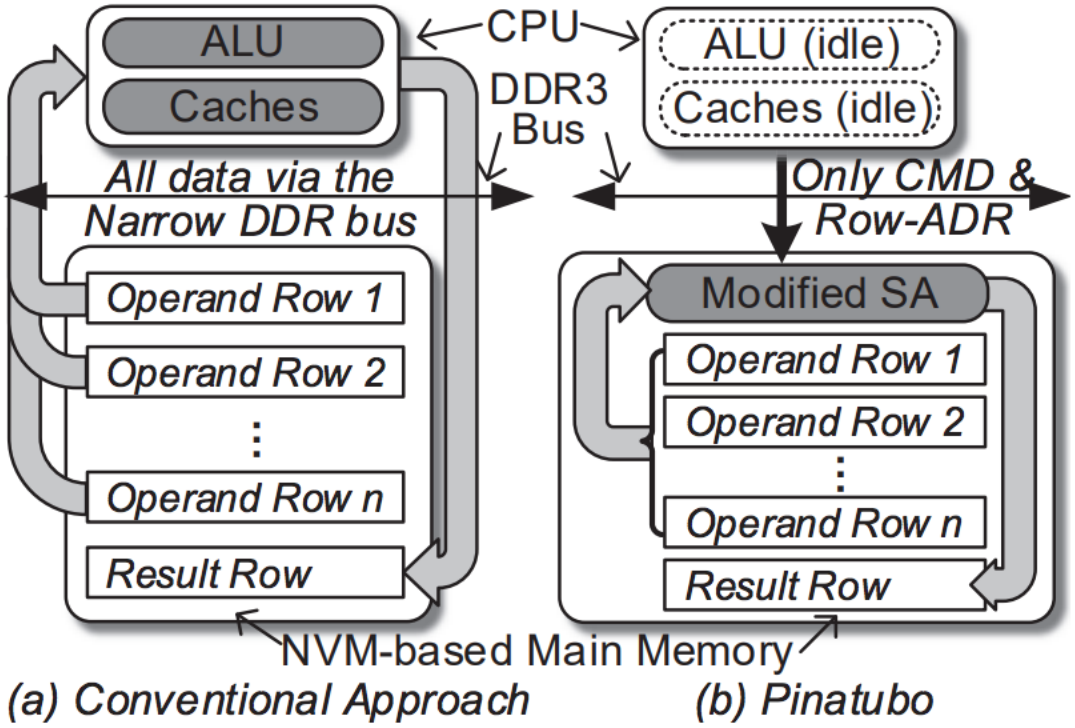


Figure 2: Overview: (a) Computing-centric approach, moving tons of data to CPU and write back. (b) The proposed Pinatubo architecture, performs n-row bitwise operations inside NVM in one step.

In-Flash Bulk Bitwise Execution

- Jisung Park, Roknoddin Azizi, Geraldo F. Oliveira, Mohammad Sadrosadati, Rakesh Nadig, David Novo, Juan Gómez-Luna, Myungsuk Kim, and Onur Mutlu,
  "Flash-Cosmos: In-Flash Bulk Bitwise Operations Using Inherent Computation Capability of NAND Flash Memory"
  Proceedings of the 55th International Symposium on Microarchitecture (***MICRO***), Chicago, IL, USA, October 2022.
  [Slides (pptx) (pdf)]
  [Longer Lecture Slides (pptx) (pdf)]
  [Lecture Video (44 minutes)]
  [arXiv version]

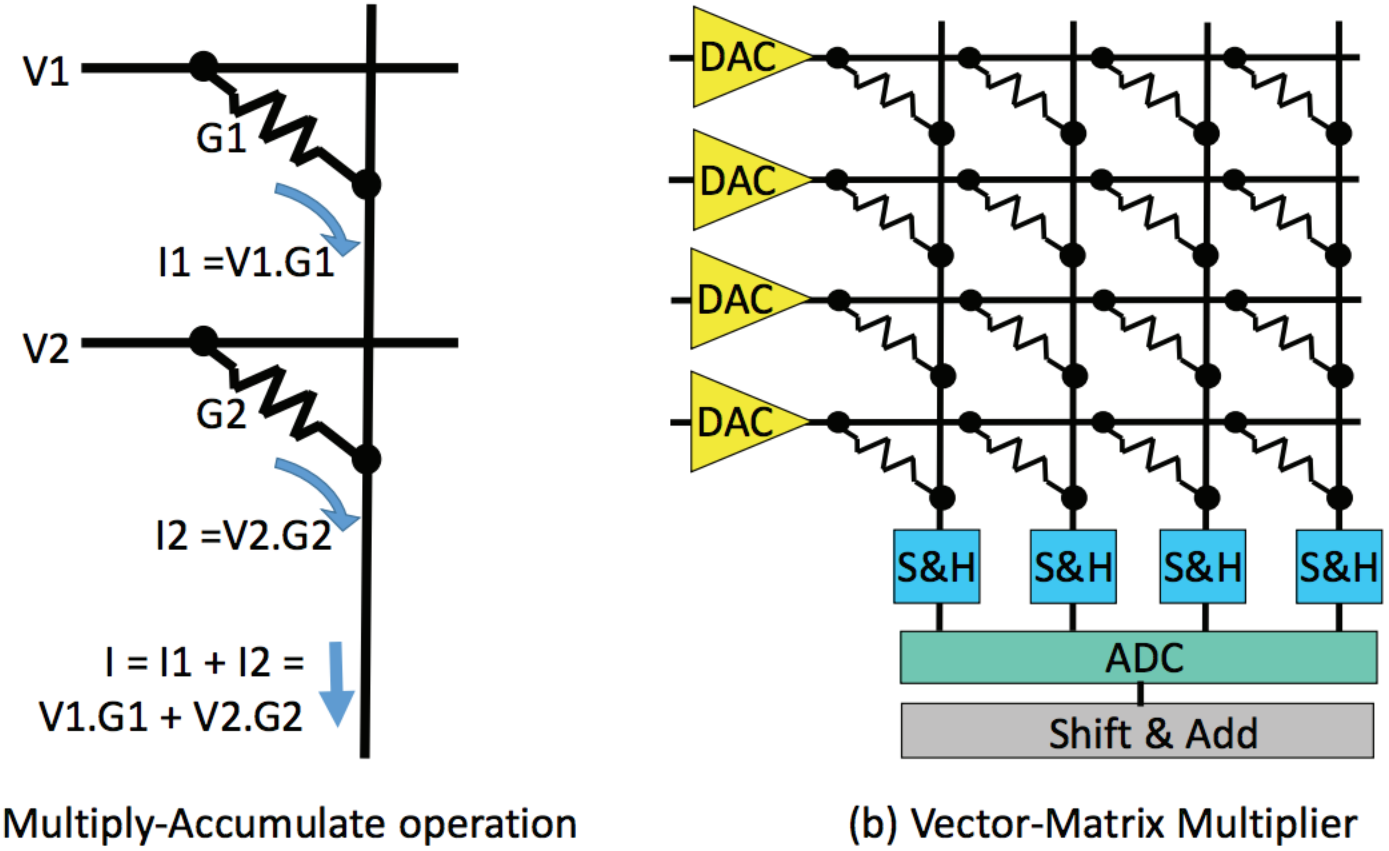
Flash-Cosmos: In-Flash Bulk Bitwise Operations Using Inherent Computation Capability of NAND Flash Memory

Jisung Park[§∇] Roknoddin Azizi[§] Geraldo F. Oliveira[§] Mohammad Sadrosadati[§]
Rakesh Nadig[§] David Novo[†] Juan Gómez-Luna[§] Myungsuk Kim[‡] Onur Mutlu[§]

[§]*ETH Zürich* [∇]*POSTECH* [†]*LIRMM, Univ. Montpellier, CNRS* [‡]*Kyungpook National University*

https://arxiv.org/pdf/2209.05566.pdf

Aside: In-Memory Crossbar Computation

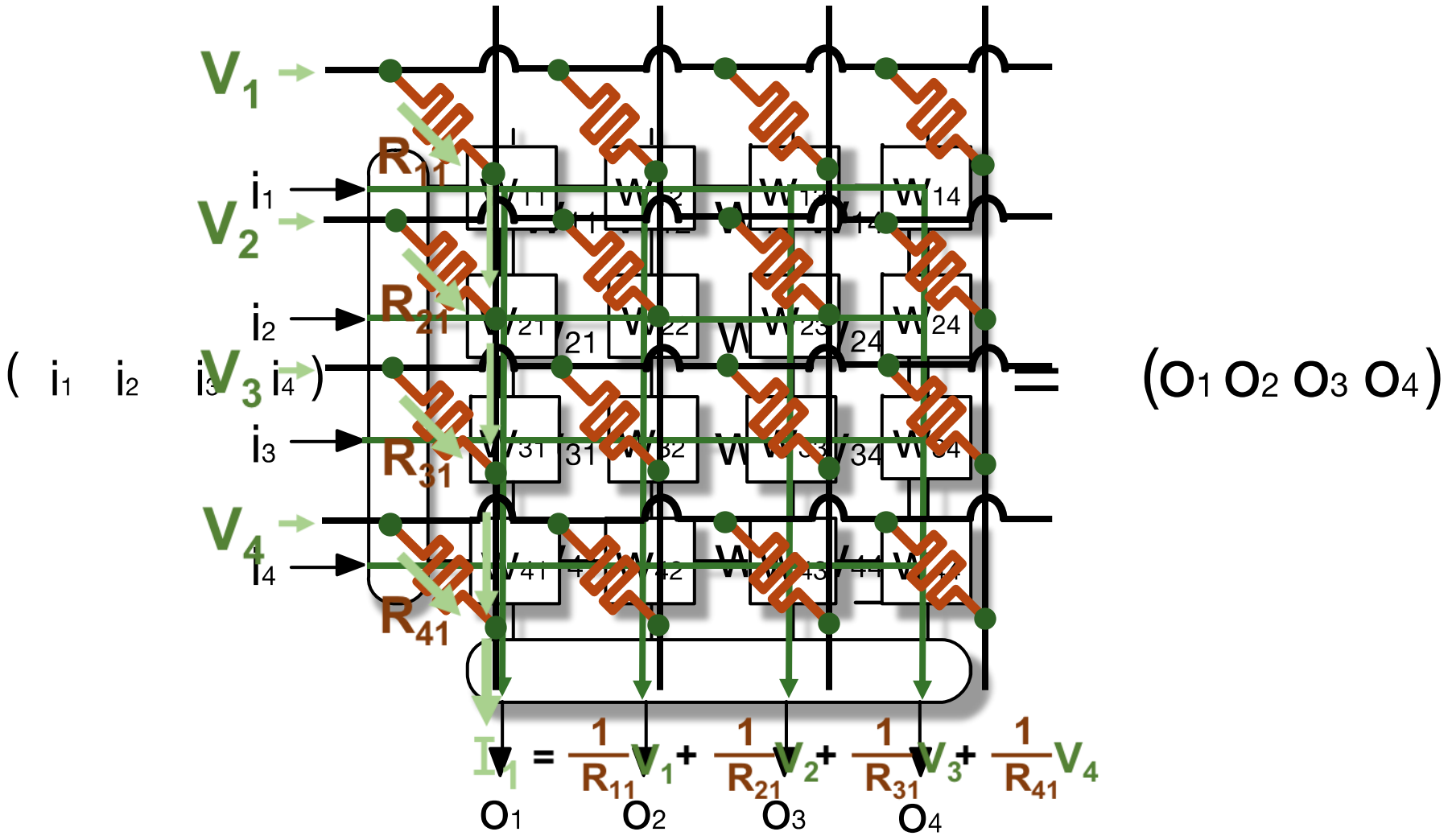


(a) Multiply-Accumulate operation

(b) Vector-Matrix Multiplier

Fig. 1. (a) Using a bitline to perform an analog sum of products operation. (b) A memristor crossbar used as a vector-matrix multiplier.

Aside: In-Memory Crossbar Computation



2nd Workshop on Memory-Centric Computing:

Processing-Using-Memory - Part II

Dr. Geraldo F. Oliveira

https://geraldofojunior.github.io

ICS 2025

08 June 2025

SAFARI

ETH zürich