SIMDRAM: A Framework for Bit-Serial SIMD Processing using DRAM

P&S Processing-in-Memory
Fall 2022
10 January 2023

Nastaran Hajinazar*
Sven Gregorio
Minesh Patel
Juan Gómez–Luna

Geraldo F. Oliveira*
Joao Ferreira
Mohammed Alser
Onur Mutlu

Saugata Ghose
Nika Mansouri Ghiasi
Executive Summary

• **Motivation**: Processing-using-Memory (PuM) architectures can effectively perform bulk bitwise computation

• **Problem**: Existing PuM architectures are not widely applicable
  – Support only a limited and specific set of operations
  – Lack the flexibility to support new operations
  – Require significant changes to the DRAM subarray

• **Goals**: Design a processing-using-DRAM framework that:
  – Efficiently implements complex operations
  – Provides the flexibility to support new desired operations
  – Minimally changes the DRAM architecture

• **SIMDRAM**: An end-to-end processing-using-DRAM framework that provides the programming interface, the ISA, and the hardware support for:
  1. Efficiently computing complex operations
  2. Providing the ability to implement arbitrary operations as required
  3. Using a massively-parallel in-DRAM SIMD substrate

• **Key Results**: SIMDRAM provides:
  – 88x and 5.8x the throughput and 257x and 31x the energy efficiency of a baseline CPU and a high-end GPU, respectively, for 16 in-DRAM operations
  – 21x and 2.1x the performance of the CPU and GPU over seven real-world applications
Outline

1. Processing-using-DRAM

2. Background

3. SIMDGRAM
   Processing-using-DRAM Substrate
   Framework

4. System Integration

5. Evaluation

6. Conclusion
# Outline

1. Processing-using-DRAM

2. Background

3. SIMDREAM
   - Processing-using-DRAM Substrate
   - Framework

4. System Integration

5. Evaluation

6. Conclusion
Data Movement Bottleneck

- Data movement is a major bottleneck

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

Bandwidth-limited and power-hungry memory channel

1 A. Boroumand et al., “Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks,” ASPLOS, 2018
Processing-in-Memory (PIM)

- **Processing-in-Memory**: moves computation closer to where the data resides
  - Reduces/eliminates the need to move data between processor and DRAM
Processing-using-Memory (PuM)

- **PuM**: Exploits analog operation principles of the memory circuitry to perform computation
  - Leverages the large internal bandwidth and parallelism available inside the memory arrays

- A common approach for PuM architectures is to perform bulk bitwise operations
  - Simple logical operations (e.g., AND, OR, XOR)
  - More complex operations (e.g., addition, multiplication)
Outline

1. Processing-using-DRAM
2. Background
3. SIMDGRAM
   Processing-using-DRAM Substrate
   Framework
4. System Integration
5. Evaluation
6. Conclusion
Inside a DRAM Chip

- Subarray (2D Array of DRAM Cells)
- Sense Amplifiers
- Row Buffer
- DRAM Bank
- DRAM Chips
- DRAM Module
DRAM Cell Operation

1. ACTIVATE (ACT)

2. READ/WRITE

3. PRECHARGE (PRE)
DRAM Cell Operation (1/3)

1. **ACTIVATE (ACT)**

2. **READ/WRITE**

3. **PRECHARGE (PRE)**

1. **raise wordline**

2. **capacitor loses charge to bitline**

3. **enable sense amplifier**

4. **amplify deviation in the bitline**

- Storage capacitor
- Access transistor
- Bitline
- Sense amplifier
- Enable
- Wordline
- Half $V_{DD} + \delta$

SAFARI
DRAM Cell Operation (2/3)

1. ACTIVATE (ACT)
2. READ/WRITE
3. PRECHARGE (PRE)

- Storage capacitor
- Access transistor
- Wordline
- Bitline
- Read/write charge
- Latched in sense amplifier
- Sense amplifier
- Enable
DRAM Cell Operation (3/3)

1. **lower wordline**
   - wordline
   - storage capacitor
   - access transistor

2. **precharge bitline** for next access
   - $\frac{1}{2}V_{DD}$
   - bitline

3. **disable sense amplifier**
   - enable
   - sense amplifier

1. **ACTIVATE (ACT)**
2. **READ/WRITE**
3. **PRECHARGE (PRE)**
RowClone: In-DRAM Row Copy (1/2)

Row copy command sequence:
1. ACTIVATE (ACT)
2. ACTIVATE (ACT)
3. PRECHARGE (PRE)
RowClone: In-DRAM Row Copy (2/2)

1. ACTIVATE source row A
2. bitline will be pulled to charge level of row A
3. ACTIVATE destination row B
4. charge level of source row A will be copied to destination row B

Row copy command sequence:
1. ACTIVATE (ACT)
2. ACTIVATE (ACT)
3. PRECHARGE (PRE)

SAFARI

2 V. Seshadri et al., "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization", MICRO, 2013
Triple-Row Activation: Majority Function

1. ACTIVATE (ACT)
2. PRECHARGE (PRE)

Majority function command sequence:

V. Seshadri et al., "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology", MICRO, 2017
Triple-Row Activation: Majority Function

1. **ACTIVATE** three rows simultaneously → **triple-row activation**

2. **bitline** will be pulled to the majority of cells A, B, and C

3. **values in cells A, B, C** will be overwritten with the majority output

4. **PRECHARGE** bitline for next access

---

**SAFARI**

**MAJ(A, B, C) = MAJ(V_{dd}, V_{dd}, 0) = V_{dd}**

---

**Majority function command sequence**

1. **ACTIVATE (ACT)**
2. **PRECHARGE (PRE)**

---

Ambit: In-DRAM Bulk Bitwise AND/OR

$MAJ(A, B, 0) = \text{AND}(A, B)$

$MAJ(A, B, 1) = \text{OR}(A, B)$
Ambit: Subarray Organization

- 1006 regular data rows
- 2 pre-initialized rows
- Regular row decoder
- Bitwise decoder
- Less than 1% of overhead in existing DRAM chips

V. Seshadri et al., “Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology”, MICRO, 2017
PuM: Prior Works

- DRAM and other memory technologies that are capable of performing **computation using memory**

**Shortcomings:**

- Support **only basic** operations (e.g., Boolean operations, addition)
  - Not widely applicable

- Support a **limited** set of operations
  - Lack the flexibility to support new operations

- Require **significant changes** to the DRAM
  - Costly (e.g., area, power)
PuM: Prior Works

• DRAM and other memory technologies that are capable of performing computation using memory

Shortcomings:

• Support only basic operations (e.g., Boolean operations, addition) - Not widely applicable

• Support a limited set of operations - Lack the flexibility to support new operations

• Require significant changes to the DRAM - Costly (e.g., area, power)

Need a framework that aids general adoption of PuM, by:

- Efficiently implementing complex operations

- Providing flexibility to support new operations
Our Goal

**Goal:** Design a PuM framework that

- **Efficiently** implements **complex** operations
- Provides the **flexibility** to support new desired operations
- **Minimally** changes the DRAM architecture
Outline

1. Processing-using-DRAM

2. Background

3. SIMDRAM
   Processing-using-DRAM Substrate Framework

4. System Integration

5. Evaluation

6. Conclusion
Key Idea

**SIMDRAM**: An end-to-end processing-using-DRAM framework that provides the programming interface, the ISA, and the hardware support for:

- Efficiently computing complex operations in DRAM
- Providing the ability to implement arbitrary operations as required
- Using an in-DRAM massively-parallel SIMD substrate that requires minimal changes to DRAM architecture
Outline

1. Processing-using-DRAM
2. Background
3. SIMDGRAM
   - Processing-using-DRAM Substrate
   - Framework
4. System Integration
5. Evaluation
6. Conclusion
SIMDRAM: PuM Substrate

• SIMDGRAM framework is built around a DRAM substrate that enables two techniques:

  (1) Vertical data layout
  most significant bit (MSB)
  least significant bit (LSB)

  (2) Majority-based computation
  \[ C_{out} = AB + AC_{in} + BC_{in} \]

Pros compared to the conventional horizontal layout:
  • Implicit shift operation
  • Massive parallelism

Pros compared to AND/OR/NOT-based computation:
  • Higher performance
  • Higher throughput
  • Lower energy consumption
Outline

1. Processing-using-DRAM
2. Background
3. SIMDRAAM
   Processing-using-DRAM Substrate
   Framework
4. System Integration
5. Evaluation
6. Conclusion
SIMDRAM Framework

User Input

Desired operation

AND/OR/NOT logic

Step 1: Generate MAJ logic

MAJ

MAJ/NOT logic

Step 2: Generate sequence of DRAM commands

| ACT/PRE |
| ACT/PRE |
| ACT/PRE |
| ACT/ACT/PRE |
| done |

μProgram

SIMDRAM Output

New SIMDRAM μProgram

μProgram

Main memory

 ISA

New SIMDRAM instruction

User Input

SIMDRAM-enabled application

foo () {

bbop_new

}

Step 3: Execution according to μProgram

Control Unit

Memory Controller

μProgram

Instruction result in memory

SAFEARI
SIMDRAM Framework: Step 1

**User Input**

Desired operation

AND/OR/NOT logic

**Step 1: Generate MAJ logic**

**MAJ/NOT logic**

**Step 2: Generate sequence of DRAM commands**

- ACT/PRE
- ACT/PRE
- ACT/PRE
- ACT/ACT/PRE
- done

**SIMDRAM Output**

New SIMDRAM µProgram

µProgram

Main memory

bbop_new

New SIMDRAM instruction

**User Input**

SIMDRAM-enabled application

foo () {
    bbop_new
}

**Step 3: Execution according to µProgram**

Control Unit

µProgram

Memory Controller

**SIMDRAM Output**

Instruction result in memory

ACT/PRE

ACT/PRE

ACT/PRE

ACT/PRE/PRE

done
Step 1: Naïve MAJ/NOT Implementation

Naïvely converting AND/OR/NOT-implementation to MAJ/NOT-implementation leads to an unoptimized circuit.
Step 1: Efficient MAJ/NOT Implementation

Step 1 generates an optimized MAJ/NOT-implementation of the desired operation

Greedy optimization algorithm

Part 2

SIMDRAM Framework: Step 2

**Step 1: Generate MAJ logic**

User Input

Desired operation

AND/OR/NOT logic

Step 2: Generate sequence of DRAM commands

SIMDRAM Output

New SIMD RAM µProgram

µProgram

Main memory

SIMDRAM Output

Instruction result in memory

**Step 3: Execution according to µProgram**

User Input

SIMDRAM-enabled application

foo () {

    bbop_new

}

SIMDRAM Output

Instruction result in memory

**SIMDRAM Framework:**

- **User Input:** Desired operation AND/OR/NOT logic
- **Step 1: Generate MAJ logic:** MAJ/NOT logic
- **Step 2: Generate sequence of DRAM commands:**
  - ACT/PRE
  - ACT/PRE
  - ACT/PRE
  - ACT/ACT/PRE
  - done
- **SIMDRAM Output:** New SIMD RAM µProgram
- **Control Unit:** MAJ logic
- **Memory Controller:** µProgram
- **SIMDRAM-enabled application:**

```python
foo() {
    bbop_new
}
```
Step 2: μProgram Generation

• **μProgram**: A series of microarchitectural operations (e.g., ACT/PRE) that SIMDRAAM uses to execute SIMDRAAM operation in DRAM

• **Goal of Step 2**: To generate the μProgram that executes the desired SIMDRAAM operation in DRAM

Task 1: Allocate DRAM rows to the operands

Task 2: Generate μProgram
Step 2: μProgram Generation

• **μProgram**: A series of microarchitectural operations (e.g., ACT/PRE) that SIMDGRAM uses to execute SIMDGRAM operation in DRAM

• **Goal of Step 2**: To generate the μProgram that executes the desired SIMDGRAM operation in DRAM

---

Task 1: Allocate DRAM rows to the operands

Task 2: Generate μProgram
Task 1: Allocating DRAM Rows to Operands

- Allocation algorithm considers two constraints specific to processing-using-DRAM

![Diagram showing subarray organization with regular row decoder and bitwise decoder with binary values and highlighted compute rows]

**Constraint 1:** Limited number of rows reserved for computation

**Compute rows**
Task 1: Allocating DRAM Rows to Operands

- Allocation algorithm considers two constraints specific to processing-using-DRAM

Constraint 2: Destructive behavior of triple-row activation

Overwritten with MAJ output
Task 1: Allocating DRAM Rows to Operands

- Allocation algorithm:
  - Assigns as many inputs as the number of free compute rows
  - All three input rows contain the MAJ output and can be reused
Step 2: μProgram Generation

• **μProgram**: A series of microarchitectural operations (e.g., ACT/PRE) that SIMDRAIM uses to execute SIMDRAIM operation in DRAM

• **Goal of Step 2**: To generate the μProgram that executes the desired SIMDRAIM operation in DRAM

**Task 1: Allocate DRAM rows to the operands**

**Task 2: Generate μProgram**
Task 2: Generate an initial µProgram

1. Generate µProgram
Task 2: Optimize the µProgram

Initial µProgram

1. Copy A to reserved row (ACT/ACT/PRE)
2. Copy B to reserved row (ACT/ACT/PRE)
3. Copy $C_{in}$ to reserved row (ACT/ACT/PRE)
4. Execute MAJ (ACT/PRE)
5. Copy $C_{out}$ to destination row (ACT/PRE)

1. Generate µProgram
2. Optimize
Task 2: Optimize the µProgram

Initial µProgram

1. **Copy** A to reserved row (ACT/ACT/PRE)
2. **Copy** B to reserved row (ACT/ACT/PRE)
3. **Copy** C\textsubscript{in} to reserved row (ACT/ACT/PRE)
4. Execute MAJ (ACT/PRE)
5. Copy C\textsubscript{out} to destination row (ACT/PRE)

Coalesce row copies

1. Generate µProgram
2. Optimize
Task 2: Optimize the μProgram

Initial μProgram

1. Generate μProgram
   1. Copy A to reserved row (ACT/ACT/PRE)
   2. Copy B to reserved row (ACT/ACT/PRE)
   3. Copy $C_{in}$ to reserved row (ACT/ACT/PRE)
   4. Execute MAJ (ACT/PRE)
   5. Copy $C_{out}$ to destination row (ACT/PRE)

2. Optimize

Merge MAJ + row copy
Task 2: Optimize the µProgram

1. Generate µProgram

2. Optimize

Optimized µProgram

1. Copy A, B, C_in to reserved rows (ACT/ACT/PRE)

2. Execute MAJ and copy C_out to destination row (ACT/ACT/PRE)

Coalesce row copies

Merge MAJ + row copy
Task 2: Generate N-bit Computation

- **Final µProgram** is optimized and computes the desired operation for operands of N-bit size in a bit-serial fashion

**Optimized µProgram**

Repeat N times:

1. Copy A, B, C_{in} to reserved rows (ACT/ACT/PRE)
2. Execute MAJ and copy C_{out} to destination row (ACT/ACT/PRE)

**SAFARI**
Task 2: Generate μProgram

- **Final μProgram** is optimized and computes the desired operation for operands of N-bit size in a bit-serial fashion.

```
Repeat N times:
1. Copy A, B, C in to reserved rows (ACT/ACT/PRE)
2. Execute MAJ and copy C out to destination row (ACT/ACT/PRE)

Stored in a reserved DRAM region for future use

A new SIMDGRAM instruction (called bbop) added to CPU ISA
```

Final μProgram
SIMDRAM Framework: Step 3

**User Input**
- Desired operation
- AND/OR/NOT logic

**Step 1: Generate MAJ logic**
- MAJ/NOT logic

**Step 2: Generate sequence of DRAM commands**
- ACT/PRE
- ACT/PRE
- ACT/PRE
- ACT/ACT/PRE
- done

**SIMDRAM Output**
- New SIMDRAM µProgram
  - µProgram
  - Main memory
  - ISA

**User Input**
- SIMDRAM-enabled application
  ```
  foo () {
    bbop_new
  }
  ```

**Step 3: Execution according to µProgram**
- Control Unit
- µProgram
  - ACT/PRE
  - ACT/PRE
  - ACT/PRE
  - ACT/PRE/PRE
  - done

**SIMDRAM Output**
- Instruction result in memory
  - ACT/PRE
Step 3: µProgram Execution

- **SIMDRAM control unit**: handles the execution of the µProgram at runtime

- Upon receiving a **bbop instruction**, the control unit:
  1. Loads the µProgram corresponding to SIMDRAM operation
  2. Issues the sequence of DRAM commands (ACT/PRE) stored in the µProgram to SIMDRAM subarrays to perform the in-DRAM operation

```plaintext
SIMDRAM-enabled application

foo () {
    bbop_new
}
```

**SAFARI**
# Outline

1. Processing-using-DRAM
2. Background
3. SIMDREAM
   - Processing-using-DRAM Substrate
   - Framework
4. System Integration
5. Evaluation
6. Conclusion
System Integration

- Efficiently transposing data
- Programming interface
- Handling page faults, address translation, coherence, and interrupts
- Handling limited subarray size
- Security implications
- Limitations of our framework
# System Integration

- Efficiently transposing data
- Programming interface
- Handling page faults, address translation, coherence, and interrupts
- Handling limited subarray size
- Security implications
- Limitations of our framework
Transposing Data

• **SIMDRAM** operates on *vertically-laid-out* data

• **Other system components** expect data to be laid out *horizontally*

*Challenging* to share data between SIMDRAM and CPU
Transposition Unit

Last-Level Cache

Object Tracker (OT)

Vertical → Horizontal Transpose

Transpose Buffer

Store Unit

Vertical → Horizontal Transpose

Transpose Buffer

Fetch Unit

Memory Controller

Horizontal → Vertical Transpose

Transpose Buffer

SAFARI
Efficiently Transposing Data

Low impact on the throughput of SIMDGRAM operations

Low area cost (0.06 mm²)
System Integration

- Efficiently transposing data
- Programming interface
- Handling page faults, address translation, coherence, and interrupts
- Handling limited subarray size
- Security implications
- Limitations of our framework
## Programming Interface

- Four new SIMDRAAM ISA extensions

<table>
<thead>
<tr>
<th>Type</th>
<th>ISA Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>bbop_trsp_init address, size, n</td>
</tr>
<tr>
<td>Input Operation</td>
<td>bbop_op dst, src, size, n</td>
</tr>
<tr>
<td></td>
<td>bbop_op dst, src_1, src_2, size, n</td>
</tr>
<tr>
<td>Predication</td>
<td>bbop_if_else dst, src_1, src_2, select, size, n</td>
</tr>
</tbody>
</table>
# Programming Interface

- Four new SIMDGRAM ISA extensions

<table>
<thead>
<tr>
<th>Type</th>
<th>ISA Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td><code>bbop_trsp_init address, size, n</code></td>
</tr>
</tbody>
</table>
### Programming Interface

- Four new SIMD RAM ISA extensions

<table>
<thead>
<tr>
<th>Type</th>
<th>ISA Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>bbop_trsp_init address, size, n</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1-Input Operation</td>
<td>bbop_op dst, src, size, n</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Programming Interface

- Four new SIMD RAM ISA extensions

<table>
<thead>
<tr>
<th>Type</th>
<th>ISA Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td><code>bbop_trsp_init address, size, n</code></td>
</tr>
<tr>
<td>1-Input Operation</td>
<td><code>bbop_op dst, src, size, n</code></td>
</tr>
<tr>
<td>2-Input Operation</td>
<td><code>bbop_op dst, src_1, src_2, size, n</code></td>
</tr>
</tbody>
</table>
Programming Interface

- Four new SIMDGRAM ISA extensions

<table>
<thead>
<tr>
<th>Type</th>
<th>ISA Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td><code>bbop_trsp_init address, size, n</code></td>
</tr>
<tr>
<td>1-Input Operation</td>
<td><code>bbop_op dst, src, size, n</code></td>
</tr>
<tr>
<td>2-Input Operation</td>
<td><code>bbop_op dst, src_1, src_2, size, n</code></td>
</tr>
<tr>
<td>Predication</td>
<td><code>bbop_if_else dst, src_1, src_2, select, size, n</code></td>
</tr>
</tbody>
</table>
Code Using SIMDGRAM Instructions

1 int size = 65536;
2 int elm_size = sizeof(uint8_t);
3 uint8_t *A, *B, *C = (uint8_t *) malloc(size * elm_size);
4 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5 ... 
6 for (int i = 0; i < size; ++ i){
7     bool cond = A[i] > pred[i];
8     if (cond)
9         C[i] = A[i] + B[i];
10    else
11        C[i] = A[i] - B[i];
12 }

← C code for vector add/sub with predicated execution

Equivalent code using SIMDGRAM operations →

1 int size = 65536;
2 int elm_size = sizeof(uint8_t);
3 uint8_t *A, *B, *C = (uint8_t *) malloc(size * elm_size);
4 
5 bbop_trsp_init(A, size, elm_size);
6 bbop_trsp_init(B, size, elm_size);
7 bbop_trsp_init(C, size, elm_size);
8 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9 // D, E, F store intermediate data
10 uint8_t *D, *E = (uint8_t *) malloc(size * elm_size);
11 bool *F = (bool *) malloc(size * sizeof(bool));
12 ... 
13 bbop_add(D, A, B, size, elm_size);
14 bbop_sub(E, A, B, size, elm_size);
15 bbop_greater(F, A, pred, size, elm_size);
16 bbop_if_else(C, D, E, F, size, elm_size);
Code Using SIMDRAVM Instructions

```
int size = 65536;
int elm_size = sizeof(uint8_t);
uint8_t *A, *B, *C = (uint8_t *) malloc(size * elm_size);
uint8_t *pred = (uint8_t *) malloc(size * elm_size);
...
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];
}
```

← C code for vector add/sub with predicated execution

Equivalent code using SIMDRAVM operations →

```
int size = 65536;
int elm_size = sizeof(uint8_t);
uint8_t *A, *B, *C = (uint8_t *) malloc(size * elm_size);

bbop_trsp_init(A, size, elm_size);
bbop_trsp_init(B, size, elm_size);
bbop_trsp_init(C, size, elm_size);
uint8_t *pred = (uint8_t *) malloc(size * elm_size);
// D, E, F store intermediate data
uint8_t *D, *E = (uint8_t *) malloc(size * elm_size);
bool *F = (bool *) malloc(size * sizeof(bool));
...
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);
```
Code Using SIMDGRAM Instructions

```c
int size = 65536;
int elm_size = sizeof (uint8_t);
uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
uint8_t *pred = (uint8_t *) malloc(size * elm_size);

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];
}
```

← C code for vector add/sub with predicated execution

Equivalent code using SIMDGRAM operations →

```c
int size = 65536;
int elm_size = sizeof(uint8_t);
uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);

bbop_trsp_init(A , size , elm_size);
bbop_trsp_init(B , size , elm_size);
bbop_trsp_init(C , size , elm_size);
uint8_t *pred = (uint8_t *) malloc(size * elm_size);
// D, E, F store intermediate data
uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
bool *F = (bool *) malloc (size * sizeof(bool));

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);
```
Code Using SIMDRAPI Instructions

```
1 int size = 65536;
2 int elm_size = sizeof (uint8_t);
3 uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5 ...
6 for (int i = 0; i < size ; ++ i){
7   bool cond = A[i] > pred[i];
8      if (cond)
9         C [i] = A[i] + B[i];
10     else
11         C [i] = A[i] - B [i];
12 }
```

Equivalent code using SIMDRAPI operations

```
1 int size = 65536;
2 int elm_size = sizeof(uint8_t);
3 uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4
5 bbop_trsp_init(A , size , elm_size);
6 bbop_trsp_init(B , size , elm_size);
7 bbop_trsp_init(C , size , elm_size);
8 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9 // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12 ...
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```
Code Using SIMDRAM Instructions

< C code for vector add/sub with predicated execution

```c
1 int size = 65536;
2 int elm_size = sizeof (uint8_t);
3 uint8_t *A, *B, *C = (uint8_t *) malloc(size * elm_size);
4 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
...
6 for (int i = 0; i < size ; ++ i){
7    bool cond = A[i] > pred[i];
8    if (cond)
9        C [i] = A[i] + B[i];
10   else
11      C [i] = A[i] - B[i];
12 }
```

Equivalent code using SIMDGRAM operations →

```c
1 int size = 65536;
2 int elm_size = sizeof(uint8_t);
3 uint8_t *A, *B, *C = (uint8_t *) malloc(size * elm_size);
4
5 bbop_trsp_init(A , size , elm_size);
6 bbop_trsp_init(B , size , elm_size);
7 bbop_trsp_init(C , size , elm_size);
8 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9 // D, E, F store intermediate data
10 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
11 bool *F = (bool *) malloc (size * sizeof(bool));
12...
13 bbop_add(D , A , B , size , elm_size);
14 bbop_sub(E , A , B , size , elm_size);
15 bbop_greater(F , A , pred , size , elm_size);
16 bbop_if_else(C , D , E , F , size , elm_size);
```
Code Using SIMD RAM Instructions

```c
1 int size = 65536;
2 int elm_size = sizeof (uint8_t);
3 uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5 ...
6 for (int i = 0; i < size ; ++ i){
7   bool cond = A[i] > pred[i];
8     if (cond)
9       C [i] = A[i] + B[i];
10    else
11       C [i] = A[i] - B [i];
12 }
```

```c
1 int size = 65536;
2 int elm_size = sizeof(uint8_t);
3 uint8_t *A , *B , *C = (uint8_t *) malloc(size * elm_size);
4 bbop_trsp_init(A , size , elm_size);
5 bbop_trsp_init(B , size , elm_size);
6 bbop_trsp_init(C , size , elm_size);
7 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
8 // D, E, F store intermediate data
9 uint8_t *D , *E = (uint8_t *) malloc (size * elm_size);
10 bool *F = (bool *) malloc (size * sizeof(bool));
11 ...
12 bbop_add(D , A , B , size , elm_size);
13 bbop_sub(E , A , B , size , elm_size);
14 bbop_greater(F , A , pred , size , elm_size);
15 bbop_if_else(C , D , E , F , size , elm_size);
```

← C code for vector add/sub with predicated execution

Equivalent code using SIMD RAM operations →
Code Using SIMDGRAM Instructions

1 int size = 65536;
2 int elm_size = sizeof(uint8_t);
3 uint8_t *A, *B, *C = (uint8_t *) malloc(size * elm_size);
4 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
5 ...
6 for (int i = 0; i < size; ++i){
7    bool cond = A[i] > pred[i];
8    if (cond)
9       C[i] = A[i] + B[i];
10   else
11      C[i] = A[i] - B[i];
12 }

← C code for vector add/sub with predicated execution

Equivalent code using SIMDGRAM operations →

1 int size = 65536;
2 int elm_size = sizeof(uint8_t);
3 uint8_t *A, *B, *C = (uint8_t *) malloc(size * elm_size);
4   
5 bbop_trsp_init(A, size, elm_size);
6 bbop_trsp_init(B, size, elm_size);
7 bbop_trsp_init(C, size, elm_size);
8 uint8_t *pred = (uint8_t *) malloc(size * elm_size);
9   / D, E, F store intermediate data
10 uint8_t *D, *E = (uint8_t *) malloc(size * elm_size);
11 bool *F = (bool *) malloc(size * sizeof(bool));
12 ...
13 bbop_add(D, A, B, size, elm_size);
14 bbop_sub(E, A, B, size, elm_size);
15 bbop_greater(F, A, pred, size, elm_size);
16 bbop_if_else(C, D, E, F, size, elm_size);
More in the Paper

SIMDRAM: An End-to-End Framework for Bit-Serial SIMD Computing in DRAM

*Nastaran Hajinazar\textsuperscript{1,2} \quad *Geraldo F. Oliveira\textsuperscript{1} \quad Sven Gregorio\textsuperscript{1} \quad João Dinis Ferreira\textsuperscript{1}
Nika Mansouri Ghiasi\textsuperscript{1} \quad Minesh Patel\textsuperscript{1} \quad Mohammed Alser\textsuperscript{1} \quad Saugata Ghose\textsuperscript{3}
Juan Gómez-Luna\textsuperscript{1} \quad Onur Mutlu\textsuperscript{1} \quad
\textsuperscript{1}ETH Zürich \quad \textsuperscript{2}Simon Fraser University \quad \textsuperscript{3}University of Illinois at Urbana–Champaign

- Handling page faults, address translation, coherence, and interrupts
- Handling limited subarray size
- Security implications
- Limitations of our framework
1. Processing-using-DRAM

2. Background

3. SIMDREAM
   Processing-using-DRAM Substrate
   Framework

4. System Integration

5. Evaluation

6. Conclusion
Methodology: Experimental Setup

- **Simulator**: gem5

**Baselines:**
- A multi-core CPU (Intel Skylake)
- A high-end GPU (Nvidia Titan V)
- Ambit: a state-of-the-art in-memory computing mechanism

**Evaluated SIMDRAM configurations** (all using a DDR4 device):
- **1-bank**: SIMDRAM exploits 65’536 SIMD lanes (an 8 kB row buffer)
- **4-banks**: SIMDRAM exploits 262’144 SIMD lanes
- **16-banks**: SIMDRAM exploits 1’048’576 SIMD lanes
Methodology: Workloads

Evaluated:

• 16 complex in-DRAM operations:
  - Absolute
  - Addition/Subtraction
  - BitCount
  - Equality/ Greater/Greater Equal
  - Predication
  - ReLU
  - AND-/OR-/XOR-Reduction
  - Division/Multiplication

• 7 real-world applications
  - BitWeaving (databases)
  - LeNET (Neural Networks)
  - TPH-H (databases)
  - VGG-13/VGG-16 (Neural Networks)
  - kNN (machine learning)
  - brightness (graphics)
Throughput Analysis

Average normalized throughput across all 16 SIMDGRAM operations

SIMDRAM significantly outperforms all state-of-the-art baselines for a wide range of operations.
Energy Analysis

Average normalized energy efficiency across all 16 SIMDRAM operations

SIMDRAM is more energy-efficient than all state-of-the-art baselines for a wide range of operations
Real-World Application

Average speedup across 7 real-world applications

SIMDRAM effectively and efficiently accelerates many commonly-used real-world applications
More in the Paper

• Evaluation:

  - Reliability
  - Data movement overhead
  - Data transposition overhead
  - Area overhead
  - Comparison to in-cache computing
More in the Paper

• Evaluation:

  - Reliability
  - Data movement overhead
  - Data transposition overhead
  - Area overhead
  - Comparison to in-cache computing

SIMDRAM: An End-to-End Framework for Bit-Serial SIMD Computing in DRAM

*Nastaran Hajinazar\textsuperscript{1,2}  
Nika Mansouri Ghiasi\textsuperscript{1}  
*Geraldo F. Oliveira\textsuperscript{1}  
Minesh Patel\textsuperscript{1}  
Juan Gómez-Luna\textsuperscript{1}  
Sven Gregorio\textsuperscript{1}  
Mohammed Alser\textsuperscript{1}  
Onur Mutlu\textsuperscript{1}  
João Dinis Ferreira\textsuperscript{1}  
Saugata Ghose\textsuperscript{3}

\textsuperscript{1}ETH Zürich  \textsuperscript{2}Simon Fraser University  \textsuperscript{3}University of Illinois at Urbana–Champaign

SAFARI  
Conclusion

- **SIMDRAM**: An end-to-end processing-using-DRAM framework that provides the programming interface, the ISA, and the hardware support for:
  1. Efficiently computing complex operations
  2. Providing the ability to implement arbitrary operations as required
  3. Using a massively-parallel in-DRAM SIMD substrate

- **Key Results**: SIMDRAM provides:
  - 88x and 5.8x the throughput and 257x and 31x the energy efficiency of a baseline CPU and a high-end GPU, respectively, for 16 in-DRAM operations
  - 21x and 2.1x the performance of the CPU and GPU over seven real-world applications

- **Conclusion**: SIMDRAM is a promising PuM framework
  - Can ease the adoption of processing-using-DRAM architectures
  - Improve the performance and efficiency of processing-using-DRAM architectures
SIMDRAM: A Framework for Bit-Serial SIMD Processing using DRAM

P&S Processing-in-Memory
Fall 2022
10 January 2023

Nastaran Hajinazar*
Sven Gregorio
Minesh Patel
Juan Gómez–Luna

Geraldo F. Oliveira*
Joao Ferreira
Mohammed Alser
Onur Mutlu

Nika Mansouri Ghiasi
Saugata Ghose

SAFARI  SFU  ETH Zürich  UNIVERSITY OF ILLINOIS