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

P&S Ramulator
Spring 2022

Nastaran Hajinazar*  Geraldo F. Oliveira*
Sven Gregorio  Joao Ferreira  Nika Mansouri Ghiasi
Minesh Patel  Mohammed Alser  Saugata Ghose
Juan Gómez–Luna  Onur Mutlu

SAFARI  SFU  SIMON FRASER UNIVERSITY  ETH Zürich  UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
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
1. Processing-using-DRAM
2. Background
3. SIMDRAAM
   Processing-using-DRAM Substrate
   Framework
4. System Integration
5. Evaluation
6. Conclusion
# Outline

1. Processing-using-DRAM

2. Background

3. SIMDGRAM
   - 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

\[\text{Computing Unit (CPU, GPU, FPGA, Accelerators)} \quad \text{Memory channel} \quad \text{Main Memory (DRAM)}\]

\[\text{SAFARI} \quad 1\text{A. Boroumand et al., “Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks,” ASPLOS, 2018}\]
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. SIMDRAM
   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 charge shifts to bitline

3. enable sense amplifier

4. amplify deviation in the bitline

\[
\frac{1}{2} V_{DD} + \delta
\]
DRAM Cell Operation (2/3)

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

- wordline
- storage capacitor
- access transistor
- bitline
- enable
- sense amplifier

read/write charge latched in sense amplifier
DRAM Cell Operation (3/3)

1. lower wordline

2. precharge bitline for next access

3. disable 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)

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:\n
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

**Majority function command sequence**:  
1. ACTIVATE (ACT)  
2. PRECHARGE (PRE)  

\[ \text{MAJ}(A, B, C) = \text{MAJ}(V_{\text{dd}}, V_{\text{dd}}, 0) = V_{\text{dd}} \]

3 V. Seshadri et al., "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology", MICRO, 2017
Ambit: In-DRAM Bulk Bitwise AND/OR

MAJ (A, B, 0) = AND (A, B)

MAJ (A, B, 1) = OR (A, B)

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

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)

- 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. SIMDGRAM
   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
<table>
<thead>
<tr>
<th>Outline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Processing-using-DRAM</td>
</tr>
<tr>
<td>2. Background</td>
</tr>
<tr>
<td>3. SIMDRAAM</td>
</tr>
<tr>
<td>Processing-using-DRAM Substrate</td>
</tr>
<tr>
<td>Framework</td>
</tr>
<tr>
<td>4. System Integration</td>
</tr>
<tr>
<td>5. Evaluation</td>
</tr>
<tr>
<td>6. Conclusion</td>
</tr>
</tbody>
</table>
SIMDRAM: PuM Substrate

- SIMDRAM 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

\[ C_{out} = A \times B + A \times C_{in} + B \times C_{in} \]

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. SIMDGRAM
   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

Step 3: Execution according to μProgram

Control Unit

μProgram

Memory Controller

SIMDRAM Output

New SIMDRAM μProgram

μProgram

Main memory

New SIMDRAM instruction

ISA

User Input

SIMDRAM-enabled application

foo () {
    bbop_new
}
SIMDRAM Framework: Step 1

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

SIMDRAM Output

New SIMDRAM µProgram
µProgram

Main memory

μProgram

ISA

Step 3: Execution according to µProgram

SIMDRAM Output

Instruction result in memory

Control Unit

Memory Controller

Safety

foo () {
    bbop_new
}

Control Unit

μProgram

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

Greedy optimization algorithm

Part 2

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

SIMDRAM Framework: Step 2

**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

New SIMDRAM instruction

**User Input**

SIMDRAM-enabled application

foo () {
    bbop_new
}

**Step 3: Execution according to µProgram**

Control Unit

µProgram

Memory Controller

Instruction result in memory

ACT/PRE
Step 2: µProgram Generation

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

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

---

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

**Task 2**: Generate µProgram
Step 2: \( \mu \text{Program} \) Generation

- **\( \mu \text{Program} \):** A series of microarchitectural operations (e.g., ACT/PRE) that SIMDREAM uses to execute SIMDREAM operation in DRAM

- **Goal of Step 2:** To generate the \( \mu \text{Program} \) that executes the desired SIMDREAM operation in DRAM

---

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

**Task 2:** Generate \( \mu \text{Program} \)
Task 1: Allocating DRAM Rows to Operands

- Allocation algorithm considers two constraints specific to processing-using-DRAM:
  - Constraint 1: Limited number of rows reserved for computation
  - Constraint 2: Regular row decoder
  - Constraint 3: Bitwise decoder

Subarray organization

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 SIMDAM uses to execute SIMDAM operation in DRAM

• **Goal of Step 2**: To generate the µProgram that executes the desired SIMDAM 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

1. Generate 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)

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** \( C_{\text{in}} \) to reserved row (ACT/ACT/PRE)
4. Execute MAJ (ACT/PRE)
5. Copy \( C_{\text{out}} \) to destination row (ACT/PRE)

Coalesce row copies

1. Generate μProgram
2. Optimize

SAFARI
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)

Merge MAJ + row copy

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

1. Generate μProgram
   - Copy A, B, Cin to reserved rows (ACT/ACT/PRE)
   - Execute MAJ and copy Cout to destination row (ACT/ACT/PRE)

2. Optimize
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\text{in} to reserved rows (ACT/ACT/PRE)

2. Execute MAJ and copy C\text{out} to destination row (ACT/ACT/PRE)

1. Generate µProgram
2. Optimize
3. Generate N-bit computation

**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.

  - **Final µProgram**
    - Stored in a reserved DRAM region for future use
    - A new SIMDRAAM instruction *(called bbop)* added to CPU ISA
SIMDRAM Framework: Step 3

**User Input**

Desired operation
AND/OR/NOT logic

**Step 1: Generate MAJ logic**

MAJ

**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**

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
Outline

1. Processing-using-DRAM
2. Background
3. SIMDRAm
   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
Efficiently Transposing Data

Low impact on the throughput of SIMD RAM operations

Low area cost (0.06 mm$^2$)
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 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>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 SIMDRAI 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>
</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>
</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>
</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>
<tr>
<td>Predication</td>
<td><code>bbop_if_else dst, src_1, src_2, select, size, n</code></td>
</tr>
</tbody>
</table>
Code Using SIMDRAM 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)
10.  else
12. }

Equivalent code using SIMDRAM 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. 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
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];
}
```

```
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

Equivalent code using SIMDARAM operations →

```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];
}
```
Code Using SIMD RAM 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 SIMD RAM 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

```
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];
}
```

Equivalent code using SIMDGRAM 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 SIMDRA.M 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 code for vector add/sub with predicated execution
        C[i] = A[i] + B[i];
    } else {
        C[i] = A[i] - B[i];
    }
}
```

Equivalent code using SIMDRA.M 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);
bbob_sub(E, A, B, size , elm_size);
bbob_greater(F , A, pred , size , elm_size);
bbob_if_else(C, D, E, F, size , elm_size);
```
# Code Using SIMDARAM Instructions

C code for vector add/sub with predicated execution

```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];
}
```

Equivalent code using SIMDARAM 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);
```
More in the Paper

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

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

\(^1\)ETH Zürich \(^2\)Simon Fraser University \(^3\)University of Illinois at Urbana–Champaign

- coherences, and interrupts
- Handling limited subarray size
- Security implications
- Limitations of our framework
Outline

1. Processing-using-DRAM
2. Background
3. SIMDGRAM
   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 SIMDARAM configurations (all using a DDR4 device):
  - 1-bank: SIMDARAM exploits 65’536 SIMD lanes (an 8 kB row buffer)
  - 4-banks: SIMDARAM exploits 262’144 SIMD lanes
  - 16-banks: SIMDARAM 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)
  - TPH-H (databases)
  - kNN (machine learning)
  - LeNET (Neural Networks)
  - VGG-13/VGG-16 (Neural Networks)
  - brightness (graphics)
SIMDRAM significantly outperforms all state-of-the-art baselines for a wide range of operations.
Energy Analysis

Average normalized energy efficiency across all 16 SIMDGRAM 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*¹,²  
Nika Mansouri Ghiasi¹  
*Geraldo F. Oliveira*¹  
Minesh Patel¹  
Juan Gómez-Luna¹  
Sven Gregorio¹  
Mohammed Alser¹  
Onur Mutlu¹  
João Dinis Ferreira¹  
Saugata Ghose³

¹ETH Zürich ²Simon Fraser University ³University of Illinois at Urbana–Champaign

**SAFARI**  
Outline

1. Processing-using-DRAM
2. Background
3. SIMDGRAM
   Processing-using-DRAM Substrate Framework
4. System Integration
5. Evaluation
6. Conclusion
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 Ramulator
Spring 2022

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 URBANA–CHAMPAIGN
Task 1: Allocating DRAM Rows to Operands

Input: Full Adder Optimized MIG

Output: Row-to-operand allocation

<table>
<thead>
<tr>
<th>Level 0</th>
<th>Level 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>v2</td>
</tr>
<tr>
<td>A</td>
<td>C_in</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>C_in</td>
</tr>
<tr>
<td>B</td>
<td>Out1</td>
</tr>
<tr>
<td>Out2</td>
<td>S</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v1</th>
<th>v2</th>
<th>v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C_in</td>
<td>A</td>
</tr>
<tr>
<td>C_in</td>
<td>B</td>
<td>C_in</td>
</tr>
<tr>
<td>B</td>
<td>Out1</td>
<td>Out2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
<th>Row-to-operand allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>A  DCC0</td>
</tr>
<tr>
<td></td>
<td>C_in T0</td>
</tr>
<tr>
<td></td>
<td>B  T1</td>
</tr>
<tr>
<td>v2</td>
<td>A  T2</td>
</tr>
<tr>
<td></td>
<td>C_in T3</td>
</tr>
<tr>
<td></td>
<td>B  DCC1</td>
</tr>
<tr>
<td>v3</td>
<td>Out1 T0</td>
</tr>
<tr>
<td></td>
<td>A  T1</td>
</tr>
<tr>
<td></td>
<td>Out2 DCC1</td>
</tr>
</tbody>
</table>
Task 1: Allocating DRAM Rows to Operands

Input: Full Adder Optimized MIG

Output: Row-to-operand allocation
Task 1: Allocating DRAM Rows to Operands

Input: Full Adder Optimized MIG

Output: Row-to-operand allocation

<table>
<thead>
<tr>
<th>Action</th>
<th>Level  = 0</th>
<th>Level  = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent(A) = { }</td>
<td>(T0, T1, T2, T3) {DCC0, DCC1}</td>
<td>(T0, T1, T2, T3) {DCC1}</td>
</tr>
<tr>
<td>A ← freeRow()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>parent(Cin) = { }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cin ← freeRow()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Free Rows:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v1</td>
<td>A</td>
<td>DCC0</td>
</tr>
<tr>
<td></td>
<td>Cin</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>v2</td>
<td>A</td>
<td>Cin</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>Mapping:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v1</td>
<td>A</td>
<td>DCC0</td>
</tr>
<tr>
<td></td>
<td>Cin</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>v1</td>
<td>Cin</td>
<td>T0</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

Phase 0

Output:

<table>
<thead>
<tr>
<th>v1</th>
<th>v2</th>
<th>v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>DCC0</td>
<td>Out1</td>
</tr>
<tr>
<td>Cin</td>
<td>T0</td>
<td>T1</td>
</tr>
<tr>
<td>B</td>
<td>DCC1</td>
<td>T2</td>
</tr>
<tr>
<td>Cin</td>
<td>T3</td>
<td>T1</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>Out2</td>
</tr>
</tbody>
</table>

Phase 1

Level 0

Level 1

Next node

Next phase
Task 1: Allocating DRAM Rows to Operands

Input: Full Adder Optimized MIG

Output: Row-to-operand allocation

Level = 0
Task 1: Allocating DRAM Rows to Operands

Input: Full Adder Optimized MIG

Output: Row-to-operand allocation

Level 0

Level 1

parent(A) = {}
A ← freeRow()

parent(Cin) = {}
Cin ← freeRow()

parent(B) = {}
B ← freeRow()

parent(A) = {}
A ← freeRow()

Level = 0

Next phase

Next node
Task 1: Allocating DRAM Rows to Operands

Input: Full Adder Optimized MIG

Output: Row-to-operand allocation
Task 1: Allocating DRAM Rows to Operands

Input: Full Adder Optimized MIG

Output: Row-to-operand allocation

Level 0

Level 1

Phase 0

Phase 1

Level 1

Level 0

1. Level = 0

2. Next node

3. Level = 1

4. Next phase

SAFARI
Task 1: Allocating DRAM Rows to Operands

Input: Full Adder Optimized MIG

Output: Row-to-operand allocation
Step 3: Operation Execution

• The SIMD RAM control unit works as follows:
Step 3: Operation Execution

- The SIMDGRAM control unit works as follows:
Step 3: Operation Execution

- The SIMDRAM control unit works as follows:
Step 3: Operation Execution

- The SIMDGRAM control unit works as follows:
Step 3: Operation Execution

- The SIMD RAM control unit works as follows:
Step 3: Operation Execution

- The SIMDRAM control unit works as follows:
Step 3: Operation Execution

- The SIMD RAM control unit works as follows:

[Diagram description]

1. From CPU to bbop FIFO
2. µProgram Scratchpad to bbop_op
3. 1024 µProgram from µProgram Memory
4. µPC to µOp Memory
5. µOp Processing FSM
6. en_decrement
7. is_zero

[Diagram details]

- µOp 0, ... to µOp 63
- µOp Memory with µOp 0, µOp 1, ... µOp 63
- µRegister Addressing Unit
- µRegister File

[Flow details]

- From CPU to FIFO
- From µProgram Scratchpad to FIFO
- 1024 µProgram from Memory
- µPC to µOp Memory
- µOp Processing FSM
- AAP/AP to Memory Controller

[Additional notes]

- Step by step execution flow through the control unit
- µOp Memory and µRegister File interaction
- Branch target determination