SIMDRAM: A Framework for Bit-Serial SIMD Processing using DRAM
[ASPLOS, 2021]

Presenter: Geraldo F. Oliveira  Seminar in Computer Architecture  25.04.2024

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

SAFARI
Outline

1. Processing-using-DRAM
2. Background
3. SIMDRAAM
   Processing-using-DRAM Substrate Framework
4. System Integration
5. Evaluation
6. Conclusion
<table>
<thead>
<tr>
<th>Section</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Processing using-DRAM</td>
<td></td>
</tr>
<tr>
<td>2. Background</td>
<td></td>
</tr>
<tr>
<td>3. SIMD RAM</td>
<td>Processing using-DRAM Substrate</td>
</tr>
<tr>
<td></td>
<td>Framework</td>
</tr>
<tr>
<td>4. System</td>
<td>Integration</td>
</tr>
<tr>
<td>5. Evaluation</td>
<td></td>
</tr>
<tr>
<td>6. Conclusion</td>
<td></td>
</tr>
</tbody>
</table>
Data Movement Bottleneck

- Data movement is a major bottleneck

More than 60% of the total system energy is spent on data movement\(^1\)

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. SIMDREAM
   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

SAFARI
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
5. capacitor charge is restored
DRAM Cell Operation (2/3)

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

- Storage capacitor
- Access transistor
- Wordline
- Bitline
- Enable
- Sense amplifier
- $V_{DD}$
- 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

5. PRECHARGE bitline for next access

½ V_DD

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

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
1. ACTIVATE three rows simultaneously → triple-row activation

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

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

4. PRECHARGE bitline for next access

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

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

MAJ \( (A, B, 1) = \text{OR} (A, B) \)

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

1006 regular data rows

2 pre-initialized rows

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. SIMDRA M
   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. SIMDRAM
   Processing-using-DRAM Substrate Framework

4. System Integration

5. Evaluation

6. Conclusion
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)

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

  (2) Majority-based computation

  \[ C_{out} = AB + AC_{in} + BC_{in} \]

  Pros compared to AND/OR/NOT-based computation:

  • Higher performance
  • Higher throughput
  • Lower energy consumption
Outline

1. Processing-using-DRAM
2. Background
3. SIMDRAM
   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/NOT logic

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

μProgram

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

**Step 3: Execution according to μProgram**

Control Unit

**SIMDRAM Output**

New SIMDRAM μProgram

μProgram

Main memory

bbop_new

New SIMDRAM instruction

**SIMDRAM Output**

Instruction result in memory

SAFARI
SIMDRAM Framework: Step 1

**Step 1: Generate MAJ logic**

User Input

- Desired operation
  - AND/OR/NOT logic

**MAJ/NOT logic**

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

<table>
<thead>
<tr>
<th>µProgram</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACT/PRE</td>
</tr>
<tr>
<td>ACT/PRE</td>
</tr>
<tr>
<td>ACT/PRE</td>
</tr>
<tr>
<td>ACT/PRE/PRE</td>
</tr>
<tr>
<td>done</td>
</tr>
</tbody>
</table>

**Step 3: Execution according to µProgram**

- foo () {
  - bbop_new
}

**SIMDRAM Output**

- New SIMDRAM µProgram
- µProgram
- Main memory

- bbop_new
- ISA

**SIMDRAM Output**

- Instruction result in memory

- ACT/PRE
Step 1: Naïve MAJ/NOT Implementation

Naïvely converting AND/OR/NOT-implementation to MAJ/NOT-implementation leads to an unoptimized circuit.
Step 1 generates an **optimized** MAJ/NOT-implementation of the desired operation

\[ \text{A} \quad \text{B} \quad \text{C} \quad \text{out} \]

SIMDRAM Framework: Step 2

**User Input**

Desired operation

AND/OR/NOT logic

**Step 1: Generate MAJ logic**

MAJ

**MAJ/NOT logic**

**μProgram**

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

<table>
<thead>
<tr>
<th>ACT/PRE</th>
<th>ACT/PRE</th>
<th>ACT/PRE</th>
<th>ACT/ACT/PRE</th>
<th>done</th>
</tr>
</thead>
</table>

**SIMDRAM Output**

New SIMDRAM μProgram

μProgram

Main memory

bbop_new

New SIMDRAM instruction

**Step 3: Execution according to μProgram**

Control Unit

μProgram

**Memory Controller**

Instruction result in memory

SAFARI
Step 2: µProgram Generation

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

• **Goal of Step 2**: To generate the µProgram that executes the desired SIMDRAM 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 SIMDRAM uses to execute SIMDRAM operation in DRAM

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

```
<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

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

```
subarray organization
```
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 SIMDRAM uses to execute SIMDRAM operation in DRAM

• **Goal of Step 2**: To generate the μProgram that executes the desired SIMDRAM operation in DRAM
Task 2: Generate an initial µProgram

1. Generate µProgram

Initial µProgram
Task 2: Optimize the µProgram

1. Generate µProgram

2. Optimize

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)
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)

Coalesce row copies

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

1. Generate µProgram
2. Optimize

Merge $\text{MAJ} + \text{row copy}$
Task 2: Optimize the μProgram

1. Generate 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)

2. Optimize

SAFARI
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.

![Diagram of MAJ circuit]

1. **Generate µProgram**
   - Copy A, B, C\_in to reserved rows (ACT/ACT/PRE)
   - Execute MAJ and copy C\_out to destination row (ACT/ACT/PRE)

2. **Optimize**
   - Final µProgram

3. **Generate N-bit computation**

---

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

- **Final \(\mu\text{Program}\)** is optimized and computes the desired operation for operands of \(N\)-bit size in a bit-serial fashion

- Final \(\mu\text{Program}\) is stored in a reserved DRAM region for future use

- A new SIMDRAM instruction (called \textit{bbop}) added to CPU ISA
SIMDRAM Framework: Step 3

**User Input**

Desired operation

AND/OR/NOT logic

**Step 1: Generate MAJ logic**

MAJ

MAJ/NOT logic

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

<table>
<thead>
<tr>
<th>ACT/PRE</th>
<th>ACT/PRE</th>
<th>ACT/PRE</th>
<th>ACT/ACT/PRE</th>
<th>done</th>
</tr>
</thead>
</table>

**SIMDRAM Output**

New SIMDRAM \( \mu \)Program

\( \mu \)Program

Main memory

bbop_new

New SIMDRAM instruction

**User Input**

SIMDRAM-enabled application

foo () {

bbop_new

}

**Step 3: Execution according to \( \mu \)Program**

Control Unit

\( \mu \)Program

Memory Controller

**SIMDRAM Output**

Instruction result in memory

ACT/PRE

ACT/PRE

ACT/PRE

ACT/PRE/PRE

done
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

```c
SIMDRAM-enabled application
foo () {
    bbop_new
}
```
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
<table>
<thead>
<tr>
<th>1. Processing-using-DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Background</td>
</tr>
<tr>
<td>3. SIMDRAM</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>
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 SIMDRAM 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 SIMDREAM operations

- SIMDREAM - 1 Bank
- SIMDREAM - 4 Banks
- SIMDREAM - 16 Banks

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
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

[ASPLOS, 2021]

Presenter: Geraldo F. Oliveira

Seminar in Computer Architecture 25.04.2024

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

SAFARI  SFU SIMON FRASER UNIVERSITY ETH Zürich UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Strengths (Sorted)

• First work to propose a concrete methodology to generate arbitrary PuM operations
  - Extends the applicability of PuM

• Explores a new execution model (i.e., SIMD) for PuM computing
  - Turns a DRAM array into a very wide SIMD engine
  - Great for throughput-oriented workloads

• Extensive evaluation, including alternative PuM architectures
  - Comparison to processing-using-caches highlights important trade-offs

• Well-written and detailed paper
  - The paper presents a holistic system integration design

SAFARI
5.6 SIMDram Limitations

We note three key limitations of the current version of the SIMDram framework.

- **Floating-Point Operation**: SIMDram supports only integer and fixed-point operations, enabling floating-point operations in DRAM while maintaining low area overheads is a challenge. For example, for floating-point addition, the IEEE 754 FP32 format [58] requires shifting the mantissa by the difference of the exponents of elements. Since each bitline stores a data element in SIMDram, shifting the value stored in one bitline without compromising the values stored in other bitlines at low cost is only feasible.

- **Operations That Require Shuffling Data Across Bitlines**: Different from prior work (e.g., Dordor [45]), SIMDram does not add any extra circuitry to perform bit-shift operations. Instead, SIMDram stores data in a vertical layout and can perform explicit bit-shift operations (if needed) by orchestrating row copies. Even though this approach enables SIMDram to implement a large range of operations, it is not possible to perform shuffling and reduction operations *across bitlines* without the inclusion of dedicated bit-shifting circuitry. This is due to the lack of physical connections across bitlines, which can be solved by building a bit-shift engine across different subarrays.

- **Synchronization Between Concurrent In-DRAM Operations**: SIMDram can be easily modified to enable concurrent execution of distinct operations across different subarrays in DRAM. However, this would require the implementation of software or hardware synchronization primitives to orchestrate the computation of a single task across different subarrays. Ideas that are similar to SynCron [45] can be beneficial.
Weaknesses (Sorted) - II

• The bit-serial execution model incurs non-trivial costs
  - Data transposition
  - High-latency PuM operations for high-precision computation

• Very-wide SIMD engines have failed in the past
  - Applications have varying degrees of SIMD parallelism
  - Extracting data parallelism from an application is hard

• Programmability is done manually
  - An assembly-like programming model hurts programmability and efficiency

• Evaluation is done using simulation
  - It is unclear if SIMDRAM ideas can be ported to off-the-shelf DRAM chips
Takeaways

• Processing-using-DRAM systems can execute **key arithmetic operations** at high throughput and high energy efficiency

• One can implement arbitrary PuM operations by orchestrating **PuM primitives** (row copies and majority)
Discussion Points

• **Tip 1:** build upon the identified weaknesses

• **Tip 2:** be concrete!

• **The bit-serial execution model incurs non-trivial costs**
  - Data transposition
  → Which hardware changes are required to operate on horizontally laid-out data?
  - We need a way to allow carry to propagate across DRAM columns.
  - Option 1: Directly interconnect DRAM columns: what is the cost?
  - Option 2: Use alternative arithmetic algorithms: what are the challenges?

• **Very-wide SIMD engines have failed in the past**
  - Applications have varying degrees of SIMD parallelism
  - Extracting data parallelism from an application is hard
  → How can SIMD underutilization and energy waste be avoided?
  - From the hardware point-of-view: operate at a smaller number of columns at work
  - From a software point-of-view: exploit other types of parallelism (e.g., application parallelism)
Tips: Preparing Your Presentation

• **Main components of your presentation**
  - Title Slide
  - Executive Summary
  - Outline
  - Presentation Flow

• **Show, don't tell!**

• **Draw takeaways and conclusions for the audience**

• **Be prepared: practice, practice, practice!**

• **Style matters!**
Title Slide

SIMDRAM: A Framework for Bit-Serial SIMD Processing using DRAM
[ASPLOS, 2021]

Presenter: Geraldo F. Oliveira  Seminar in Computer Architecture  25.04.2024

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

Paper information: full title, venue, authors list
Executive Summary

- Inform the audience what the presentation is about from the start
- **Motivation** (optional), **Problem**, **Goal**, **Key Idea/Key Mechanism**, **Key Results**

**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 & Presentation Flow

• An **outline** gives **structure to your presentation** and informs/reminds the reader of the current topic of discussion

**Outline**

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

• **Presentation flow** should go from a **high** (introduction, background, key ideas) to a **low level of abstraction** (implementation details, evaluation)
During an ACTIVATE, the wordline of the target row is asserted, which connects all cells along the row to their respective bitlines. Each bitline shares charge with its corresponding cell capacitor, and the resulting bitline voltage shift is sensed and amplified by the bitline’s sense amplifier. Once the sense amplifiers finish amplification, the row buffer contains the values originally stored within the cells along the asserted wordline.
Draw Takeaways & Conclusions

• By **explicitly drawing takeaways and conclusions**, you ensure that the audience gets the correct message

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)

Need a framework that aids **general adoption of PuM**, by:
- Efficiently implementing **complex operations**
- Providing flexibility to support **new operations**
Be Prepared: Practice, Practice, Practice!

• Being **comfortable with your slides** and knowing what is coming next is crucial!

• **Script your talk** for consistent practice rounds

---

jMDRAM is an end-to-end framework that provides the user with the ability to implement an arbitrary operation in DRAM.

The framework is composed of three main steps which are illustrated in this picture.

**CLICK** It takes as user input the AND/OR/NOT representation of a desired operation.

**CLICK** Step 1 builds an efficient MAJ/NOT representation of a given desired operation from its AND/OR/NOT-based implementation. Specifically, this step takes as input a desired operation and uses logic optimization to minimize the number of logic primitives (and, therefore, the computation latency) required to perform the operation.

**CLICK** The second step allocates DRAM rows to the operation’s inputs and outputs and generates the required sequence of DRAM commands to execute the desired operation.

**CLICK** This step’s output is a µProgram, i.e., the optimized sequence of DRAM commands that is stored in main memory and will be used to execute the operation at runtime. The µProgram is stored in main memory and will be used to execute the operation at runtime.

**CLICK** A new SIMDRAM instruction called bmem is created as an interface to the µProgram, and then added to the CPU ISA.

**CLICK** The third step executes the µProgram to perform the operation. SIMDRAM uses a control unit in the memory controller that transparently issues the sequence of commands to DRAM, as dictated by the µProgram. Once the µProgram is complete, the result of the operation is held in DRAM.
Style Matters!

• Your keen eye is instrumental in finding patterns!

Do not distract the audience with the following:

- Tipos (typos)
- Inconsistencies
  • e.g., different fonts being used together
- Misalignments
- Poorly set vertical and horizontal alignments

• Beauty is subjective, but there are good practices

- Start from a good template
- Composition rules are good starting points:
  rule of thirds, use of empty spaces, repetition, font hierarchy ...
- When using colors, think about the basics of color theory

• Avoid the Calibri font at all costs!

  “It’s the end of an era, but Calibri’s designer, Lucas de Groot, has no qualms about letting his typeface rest for a bit. “It’s a relief,” he says.” [WIRED, 2021]
SIMDRAM: A Framework for Bit-Serial SIMD Processing using DRAM

[ASPLOS, 2021]

Presenter: Geraldo F. Oliveira  Seminar in Computer Architecture  25.04.2024

Nastaran Hajinazar*  Sven Gregorio  Joao Ferreira  Mohammed Alser
Minesh Patel  Juan Gómez-Luna  Onur Mutlu  Saugata Ghose