Memristive Memory Processing Unit (mMPU) Real Processing in Memory using Memristors

Nishil Talati, Rotem Ben Hur, Nimrod Wald, Ameer Haj Ali, Ben Perach, Natan Peled, Ronny Ronen and Shahar Kvatinsky

Technion – Israel Institute of Technology

Yale80 in 2019, July 2, 2019
The ASIC$^2$ Group

- Emerging technologies: Design, Simulation, Modeling, Applications
- Explore computation beyond von Neumann architectures

Memory design

Simulation tools

Efficient processors

Hardware security

Mixed signal & RF circuits

Cytomorphic electronics

Neuromorphic computing

Memristive Memory Processing Unit (mMPU)
Talk Summary in a Single Slide

A memristor memory cell

NOR logic Gate (MAGIC)

Crossbar Compatible

SIMD computing in memory

True Processing in Memory
The von Neumann Bottleneck

Latency

Energy

<table>
<thead>
<tr>
<th>Operation</th>
<th>Energy/Op (45nm)</th>
<th>Cost (vs. Add)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add operation</td>
<td>0.18 pJ</td>
<td>1X</td>
</tr>
<tr>
<td>Load from on-chip SRAM</td>
<td>11 pJ</td>
<td>~60X</td>
</tr>
<tr>
<td>Send to off-chip DRAM</td>
<td>640 pJ</td>
<td>~3600X</td>
</tr>
</tbody>
</table>

Pedram et al., IDT, 2017

The von Neumann Machine

Processing In Memory (PIM) – Higher Performance, Lower Energy!
The Problem

Processing of data is performed far away from the data

62.7% of the total system energy is spent on **data movement**

We Need A Paradigm Shift To …

- Enable computation with minimal data movement
- Compute where it makes sense (**where data resides**)  
- Make computing architectures more **data-centric**
In-/Near-/Out- Memory Computing

• **OUT**: All computations are done out of the memory array → data movement, dedicated processing units
• **NEAR**: Some computations done out of the memory array → data movement
• **IN**: All computations are performed in the memory array

---

**IN**: Real Processing in Memory

Commands
Data movement

On-Chip Controller

Peripheral circuit

Memristive Memory Array

---

On-Chip Controller

Peripheral circuit

Memristive Memory Array

---

Processing Element

Read Inputs
Write Results

Controller

Memristive Memory Array

---

mMPU: Potential Solution to the von Neumann Bottleneck

Moving from conventional DRAM to memristive memory

mMPU: performing computation *USING* the memristive memory cells
Outline

• Background

• **Memristor basics**
  • Logic using memory cells
  • Processing in Memory - the mMPU
  • System design using the mMPU
• Conclusions
Memristor: The Fourth Basic Element

Memristor – Memory Resistor
Resistor with Varying Resistance

Low resistive state ($R_{ON}$, LRS)
High resistive state ($R_{OFF}$, HRS)

Decrease in resistance
Increase in resistance
Applications for Memristors

Memory

Logic circuits

Analog circuits

Neuromorphic computing

Security

Basics
Important Memristors Attributes for Logic

• Hysteresis – state is preserved
• Distinct high/low resistance states (HRS/LRS) – binary applications
• Threshold current/voltage for switching
Logic Families Using Memristors

Outline

• Background
• Memristor basics

• **Logic using memory cells**
  • Processing in Memory - the mMPU
  • System design using the mMPU
• Conclusions
MAGIC – Memristor Aided LoGIC

Initialize OUT to $R_{ON}$

$R_{ON} = \text{Logic '1'}$

$R_{OFF} = \text{Logic '0'}$

$R_{OFF} \gg R_{ON}$

<table>
<thead>
<tr>
<th>$IN_1$</th>
<th>$IN_2$</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

MAGIC NOR in Memristive Crossbar

Crossbar Compatible

Functionally complete
MAGIC NOR in a Memristive Memory
Parallel Execution of MAGIC Gates

Efficient SIMD Realization

N. Talati et al., “Logic Design within Memristive Memories using Memristor-Aided IoGIC (MAGIC),” TNANO, 2016
Outline

• Background
• Memristor basics
• Logic using memory cells
• **Processing in Memory - the mMPU**
• System design using the mMPU
• Conclusions
The Idea: Throughput Improvement

• Locate element computation to a single row
  => Execution of a single instance in each row in parallel
• Implement desired function using NOR/NOT sequence

\[
\text{Throughput} = \frac{\#\text{instances}}{\text{Latency}}
\]

\[
\begin{align*}
\mathbf{a}_0 & \quad \mathbf{b}_0 \\
\mathbf{a}_1 & \quad \mathbf{b}_1 \\
\mathbf{a}_2 & \quad \mathbf{b}_2 \\
\vdots & \quad \vdots \\
\mathbf{a}_n & \quad \mathbf{b}_n
\end{align*}
\]

\[
f^n: \quad \begin{align*}
\mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n \quad \mathbf{b}_0, \mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n
\end{align*}
\]

= \quad \begin{align*}
\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_n
\end{align*}
Example: In-Memory Parallel Execution

\[ \text{OR}(A_i, B_i) \quad \forall i = 1, \ldots, N \]

\( A \text{ or } B = \neg(\text{A nor } B) \)
Example: In-Memory Parallel Execution

\[ \text{NOT}(A + B) \]

\[ OR(A_i, B_i) \quad \forall i = 1, \ldots, N \]

\[ A \text{ or } B = \text{not}(A \text{ nor } B) \]

**Total**: Latency: 2 steps  
Area: (2+2) cells  
Energy: 2 operations

* Per element, ignoring initialization
Hierarchy of Logical Functions

Matrix multiplication
POW
SQRT
ADD
AND
MUL
XOR
OR
DIV
SUB
COPY
NAND

Convolution
LOG

MAGIC NOR/NOT

Complete logic family
Example: Full Adder (1)

\[ S = A \oplus B \oplus C_{in} \]
\[ C_{out} = A \cdot B + B \cdot C_{in} + A \cdot C_{in} \]

- Generate NOR/NOT sequence
- Existing CAD tools do it very well!
- Can reuse interim results to save space!

**Total**:
- Latency: 12 steps\(^2\)
- Area: \(\leq (3+12)\) cells
- Energy: 12 operations

---
\(^1\) Per element, ignoring initialization
\(^2\) Can be done w/ 9 NORs

Example: Full Adder (2)

full_adder_12gates.v.in/out+gates = 3/2+10:
1: T1 = NOT I1 % alloc: R3 = NOT I1
2: T2 = NOT I2 % alloc: R4 = NOT I2
3: T5 = NOR2 T1,T2 % alloc: R7 = NOR2 R3,R4
4: T4 = NOR2 I1,I2 % alloc: R6 = NOR2 I1,I2
5: T6 = NOR2 T5,T4 % alloc: R8 = NOR2 R7,R6
6: T8 = NOR2 T6,I3 % alloc: R10 = NOR2 R8,I3
7: T7 = NOT T6 % alloc: R9 = NOT R8
8: T3 = NOT I3 % alloc: R5 = NOT I3
9: T9 = NOR2 T7,T3 % alloc: R11 = NOR2 R9,R5
10: O11 = NOR2 T8,T9 % alloc: R1 = NOR2 R10,R11
11: T10 = NOR2 T5,T9 % alloc: R12 = NOR2 R7,R11
12: O12 = NOT T10 % alloc: R2 = NOT R12

full_adder_12gates.v.in/out+gates = 3/2+10:
1: T1 = NOT I1 % alloc: R3 = NOT I1
2: T2 = NOT I2 % alloc: R4 = NOT I2
3: T5 = NOR2 T1,T2 % alloc: R2 = NOR2 R3,R1
4: T4 = NOR2 I1,I2 % alloc: R3 = NOR2 I1,I2
5: T6 = NOR2 T5,T4 % alloc: R1 = NOR2 R2,R3
6: T8 = NOR2 T6,I3 % alloc: R3 = NOR2 R1,I3
7: T7 = NOT T6 % alloc: R4 = NOT R1
8: T3 = NOT I3 % alloc: R1 = NOT I3
9: T9 = NOR2 T7,T3 % alloc: R5 = NOR2 R4,R1
10: O11 = NOR2 T8,T9 % alloc: R1 = NOR2 R3,R5
11: T10 = NOR2 T5,T9 % alloc: R3 = NOR2 R2,R5
12: O12 = NOT T10 % alloc: R2 = NOT R3

Naïve column allocation

Smart column allocation

No Cell Reuse

12 cells

↓

5 cells

Best Cell Reuse

Total*:
Latency: 12 steps
Area (max): (3+12) cells
Area (min): (3+5) cells
Energy: 12 operations

* Per element, ignoring initialization

Smart Reuse ➔ Column usage reduction!
Example: N-bit Multiplication

- One partial product at a time
- One adder at a time
- Requires \( \sim 300 < 512 \) cells for \( N=16 \)

- Sequence of adders
- Partial products

- Input 1
- Input 2

N bits  N bits  3N bits  12N bits  2N bits

- Showcase PIM implementation of complex operation
- \( N=16 \) \( \rightarrow \) \( \sim 3700 \) NOR operations \( (O(N^2)) \)
- Naïve column allocation: \( \sim 3700 \) Cells \( (O(N^2)) \)
- Optimized manual/automictic allocation: \( \sim 300 \) Cells! \( (O(N)) \)

- Area
  - 3700
  - 300

- Total*: 
  - Latency: \( \sim 3700 \) steps
  - Area (max): \( \sim 3700 \) cells
  - Area (min): \( \sim 300 \) cells
  - Energy: \( \sim 3700 \) operations

* Per element, ignoring initialization

Outline

• Background
• Memristor basics
• Logic using memory cells
• Processing in Memory - the mMPU

• **System design using the mMPU**
• Conclusions
System Design using the mMPU

Memory Design

Periphery Design

mMPU Controller Design and Optimization

Applications

Programming Model

CPU

mMPU Controller

mMPU

28
Challenge 1
mMPU Controller μ-architecture
Challenge 1: Arithmetic/Logical Operations in the mMPU

- **Arithmetic/Logical Operations in the mMPU**
Challenge 2: Code Generation and Storage Allocation

- (adapted from) R. Ben Hur, et al., “Synthesis and Mapping of Boolean Functions for Memristor Aided Logic (MAGIC)”, ICCAD 2017
Challenge 2a: Mapping into a Limited-size Array

- Netlist → DAG*
- Traverse (DFS) – similar to register allocation
- The execution order influences:
  - How many cells are enough for the execution
  - How many initialization cycles are necessary

*DAG: Direct Acyclic Graph
### Challenge 2a: Mapping into a Limited-size Array

**Minimum:** 8 columns (cells) → 16 cycles for execution of $N$ instances

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Netlist</th>
<th>Physical Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$w_1 = \text{NOT } A$</td>
<td>$R_2 = \text{NOT } I_1$</td>
</tr>
<tr>
<td>2</td>
<td>$w_2 = \text{NOT } B$</td>
<td>$R_1 = \text{NOT } I_2$</td>
</tr>
<tr>
<td>3</td>
<td>$w_5 = \text{NOR2 } w_1, w_2$</td>
<td>$R_4 = \text{NOR2 } R_2, R_1$</td>
</tr>
<tr>
<td>4</td>
<td>$w_4 = \text{NOR2 } A, B$</td>
<td>$R_3 = \text{NOR2 } I_1, I_2$</td>
</tr>
<tr>
<td>5</td>
<td>$w_6 = \text{NOR2 } w_5, w_4$</td>
<td>$R_5 = \text{NOR2 } R_4, R_3$</td>
</tr>
<tr>
<td>6</td>
<td><strong>ReUsing 3 registers = 2 1 3</strong></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>$w_8 = \text{NOR2 } w_6, \text{Cin}$</td>
<td>$R_2 = \text{NOR2 } R_5, I_3$</td>
</tr>
<tr>
<td>8</td>
<td>$w_7 = \text{NOT } w_6$</td>
<td>$R_1 = \text{NOT } R_5$</td>
</tr>
<tr>
<td>9</td>
<td>$w_3 = \text{NOT } \text{Cin}$</td>
<td>$R_3 = \text{NOT } I_3$</td>
</tr>
<tr>
<td>10</td>
<td><strong>ReUsing 1 registers = 5</strong></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>$w_9 = \text{NOR2 } w_7, w_3$</td>
<td>$R_5 = \text{NOR2 } R_1, R_3$</td>
</tr>
<tr>
<td>12</td>
<td><strong>ReUsing 2 registers = 1 3</strong></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>$S = \text{NOR2 } w_8, w_9$</td>
<td>$R_1 = \text{NOR2 } R_2, R_5$</td>
</tr>
<tr>
<td>14</td>
<td>$w_{10} = \text{NOR2 } w_5, w_9$</td>
<td>$R_3 = \text{NOR2 } R_4, R_5$</td>
</tr>
<tr>
<td>15</td>
<td><strong>ReUsing 3 registers = 2 4 5</strong></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>$C_{out} = \text{NOT } w_{10}$</td>
<td>$R_2 = \text{NOT } R_3$</td>
</tr>
</tbody>
</table>
Challenge 2a: Latency/Area Relative to Unrestricted Area (No Reuse)

At the cost of a 6.2% latency degradation, latency overhead reduces to 2.3% when area decreases by 5.4x. Area decreases by 5.8x.

EPFL Benchmarks:
- Area - Minimum area
- Area - Minimum area + max{5%,10}
- #Cycles - Minimum area
- #Cycles - Minimum area + max{5%,10}

Relative Latency, Relative Area
Challenge 3: Data Movement

Processing-in-memory Instruction

PIM_ADD
#Op1, #Op2, #Out

Relative locations of op1 and op2:
- Same MAT, aligned
- Same MAT, not aligned
- Different MATs
- Different banks

DIMM

Bank

Internal Bus

Bank I/O

Row Dec

Subarray

MLT

Memristive cell

BL

WL

System

35
Challenge 3: Modes of Data Transfer

Intra-MAT data transfer

Intra-bank data transfer

Inter-bank data transfer

N. Talati et al., "Practical Challenges in Delivering the Promises of Real Processing-in-Memory Machines," DATE 2018
More mMPU Research Challenges

• Defining the mMPU operation
  • Instructions; Micro-Operation Format
  • Control Storage
  • Optimized compilation of mMPU “program”
  • Passing info from the mMPU to the runtime systems

• Data mapping & alignment
  • "Where is address A+1?"
  • Which variables stay in the same MAT?
  • Cost of data alignment
  • Memory controller to support PIM

• Understanding the electrical limitations
• Rigorous evaluation of mMPU benefits
• Endurance aspects

....
Conclusions

• Data transfer → the von Neumann bottleneck
• MAGIC – the scalable building block for processing in memory
• mMPU – a real processing-in-memory machine
  • Requires a new computing paradigm
• Our group aims to develop a working end-to-end mMPU system
Thanks!