Energy Efficient And Intelligent Processing-In-Memory

-From Device to Algorithm

Deliang Fan
Assistant Professor, Ph.D.
School of Electrical, Computer and Energy Engineering
Arizona State University, Tempe, AZ, 85287, USA
Email: dfan@asu.edu
https://dfan.engineering.asu.edu/

Contributing Ph.D. Students
Zhezhi He, Shaahin Angizi, Adnan Rakin, Li Yang
Outline

➢ Motivation:
  ➢ Power Wall in CMOS technology;
  ➢ Memory Wall in Von-Neumann Architecture

➢ Research Objectives and Methodologies:
  ➢ Bottom-Up: *Device & Circuits* co-design for *parallel* and *reconfigurable* in-memory logic based on Non-Volatile Memory, like STT-MRAM, SOT-MRAM, ReRAM
  ➢ Top-Down: *Architecture & Algorithm* co-optimization for data intensive *processing-in-memory* acceleration: Deep Neural Network, Data Encryption, Graph Processing, DNA Alignment, etc.

➢ Summary
Motivation: Power Wall in CMOS Device

- Low power design is a grand challenge!
- Mobile devices with extremely low power
- End of Moore’s law and Dennard Scaling
- Possible solutions?

Performance Improvement

5W power constraint

1W SOC power constraint

Mobile SoC power road map

Low power design is a grand challenge!

End of Moore’s law and Dennard Scaling

Possible solutions?

More Moore

More energy efficient, higher density

Technology Trend

Carbon nanotube Graphene TFETs III-V devices Spintronics

Post-CMOS?
Motivation: Energy Efficient In-Memory Computing

- Energy hungry data transfer
- Long memory access latency
- Limited memory bandwidth

Moving a floating point number from main memory to CPU takes two orders more energy than processing in CPU.

CPU:
- Sequential Computation
- 32-bit ALU
- Main Memory
- Instruction fetch
- Data transfer
- Multiple instruction fetch
- Multiple data transfer

GPU:
- Parallel Computation
- 32-bit GPU
- Memory
- Instruction fetch
- Data transfer
- Single instruction fetch
- Multiple data transfer

On-chip cache:
- Energy: ~5pJ
- Latency: ~10ns

Off-chip memory:
- Energy: ~640pJ
- Latency: ~100ns

Applications Demanding PIMs

- General-purpose PIM or accelerator-PIM?
- What applications are suitable for PIM (most digital PIM could do bulk bit-wise logic)?
- Application-Hardware Co-Design needed?
In-memory computing involves different memory technologies, logic-in-memory circuit designs, and PIM architecture. Software-Hardware Co-Design is needed for different applications!
**Main Research Objective and Methodology**

### Architecture: Parallel Processing-in-Memory Accelerator

**Algorithm:** Data intensive and intelligent application algorithm development for developed PIM platform

**Developed and developing PIM applications:** deep neural network, data encryption, image processing, graph processing

---

### Algorithm and Architecture Co-Design

**Architecture:** Parallel Processing-in-Memory Accelerator

**Algorithm:** Data intensive and intelligent application algorithm development for developed PIM platform

**Partial related works in 2018-2020**

**Algorithm:** D. Fan, et. al., CVPR’20/19, AAAI’20, ICCV’19, USENIX Security’20, DAC’20/19/18, ICCAD’19/18, DATE’20/19, TCAD’19/18, TMAG’20/18, etc.


---

### Top-Down Device and Circuit Co-Design

**Device & Circuit:** Parallel and Reconfigurable in-Memory Logic based on STT-MRAM, SOT-MRAM, DWM, ReRAM, DRAM, with extreme low overhead.

**Objective:** dual mode computational memory simultaneously working as memory and in-memory logic

---

### Bottom-Up Device and Circuit Co-Design

**Device & Circuit:** Parallel and Reconfigurable in-Memory Logic based on STT-MRAM, SOT-MRAM, DWM, ReRAM, DRAM, with extreme low overhead.

**Objective:** dual mode computational memory simultaneously working as memory and in-memory logic

---

**Developed and developing PIM applications:** deep neural network, data encryption, image processing, graph processing
Outline

- **Bottom Up**: Device & Circuits co-design for *parallel* and *reconfigurable* in-memory logic based on NVM
  - Memory and In-Memory Complete Boolean Logic
  - One/Two-Cycle In-Memory Full Adder leading to Fast and Parallel In-Memory Adder
  - Overcome Operand Locality Issue in Existing In-Memory Logic Designs
Spintronic Devices and MRAM

**STT-MRAM**
- **Limitations**
  - Write asymmetry
  - Reliability-limited write speed
  - Read write optimization conflicts

**SOT-MRAM**
- **Key Advantages**
  - Energy-efficient write
  - Decoupled R/W current paths
  - Separate optimization for Read and for Write

- **Limitations**
  - Requires two access transistors
  - Switching PMA MTJ requires FL engineering that involves fabrication challenge

**Features**
- ultra-low switching energy
- non-volatility
- excellent retention time
- high integration density
- CMOS compatibility

---

Dual-Mode Memory: Memory and Logic

Basic In-Memory logic – AND/NAND, OR/NOR
Dual-Mode IN-MEMORY Logic Design

- Dual mode architecture that perform both memory read-write and AND/OR logic operations.
- **Memory Mode**: charge current (~120 µA), 1ns switching speed
- **Computing Mode**: Every two bits stored in the identical column can be selected and sensed simultaneously. Through selecting different reference resistances \((EN_M, EN_{AND}, EN_{OR})\), the SA can perform basic in-memory Boolean functions (i.e. AND and OR).
Dual-Mode IN-MEMORY Logic Design

- For AND operation, $R_{\text{ref}}$ is set at the midpoint of $R_{AP} \parallel R_P = (1,0)$ and $R_{AP} \parallel R_{AP} = (1,1)$
- For OR operation, $R_{\text{ref}}$ is set at the midpoint of $R_P \parallel R_P$ and $R_P \parallel R_{AP}$
- We have performed Monte-Carlo simulation with 100000 trials. A $\sigma = 5\%$ variation is added on the Resistance-Area product ($RA_P$), and a $\sigma = 10\%$ process variation is added on the TMR.
- Sense Margin will be reduced by increasing the logic fan-in (i.e. number of parallel memory cells).
- To avoid read failure, only two fan-in in-memory logic is used in this work.

- No XOR/XNOR Logic now, intermediate data write-back needed if implemented using AND/OR
- More logic functions needed!
More Logic Functions Supported

Reconfigurable AND/NAND, OR/NOR, XOR/XNOR, Majority In-Memory Logic in one design
Reconfigurable Complete Boolean Logic

- Dual mode architecture that perform both memory and in-memory logic operations.
- Only two fan-in in-memory logic to avoid logic failure!

- Modified row/column decoder can enable either single line (memory read) or double line (logic operation).
- SA can provide bitwise AND/NAND and OR/NOR, XOR/XNOR can be realized through combinational logic gates (AND,NOR).

- **Complete Parallel Boolean Logic in one SA and one sensing cycle:** AND/NAND, OR/NOR, XOR/XNOR

---


Recent Processing-in-Memory Platforms

**Ambit: DRAM-based [4]**
- Operand locality issue
- Original data overwritten
- Multi-cycle operations
- Simple logic
- Low area overhead
- Hardware-friendly
- Exploit the full bandwidth

**Pinatubo: NVM-based [5]**
- Operand locality issue
- Modified SA
- Simple logic
- Medium area overhead
- Support one-step multi-row operations
- General platform

**RIMPA: DWM-based [6]**
- Operand locality issue
- Large area overhead
- Simple logic
- Fast MG computation
- Ultra-low power

**Issues:**
1. Only simple logic (7+ cycles for FA, intermediate data write-back)
2. Operand locality

**our solutions**
Two-Cycle In-Memory Full Adder
Reconfigurable Logic-SA

- Dual mode architecture that performs both memory and in-memory logic operations.
- Up to three fan-in in-memory logic to avoid logic failure!

Monte Carlo simulation result

- 2-input Boolean Logic in one SA and one sensing cycle: AND/NAND, OR/NOR, XOR/XNOR.
- 3-input Boolean Logic in one SA and one sensing cycle: MAJ/MIN.

D. Fan. et. al. ASPDAC 2019
In-memory Addition

- **Carry** is directly produced by ParaPIM’s MAJ function (3-row activation).

- A **Carry Latch** to store intermediate Carry outputs to be used in summation of next bits.

- **Sum** output is achieved by 2-row activated XOR followed by a 2-input XOR gate connected to it and Carry Latch.

- Enable parallel **One Computation per two Memory Cycles**.

- Assume A, B and C operands, the 2- and 3-input in-memory logic schemes generates **Sum**/Difference and **Carry**/Borrow bits very efficiently.
Parallel In-Memory Multi-bit Adder

- Parallel Matrix Addition enabled
- 2N cycles are needed for N-bit adder
One Cycle In-Memory Full Adder
Reconfigurable Logic-SA

2-input Boolean Logic (IML2x) in one SA and one sensing cycle: AND2/NAND2, OR2/NOR2.

In-memory AND2 (IML21)

- Operation: \( B \leftarrow A \)
- Function: Copy row A to row B

In-memory XOR3 (IML35)

- Operation: \( B \leftarrow A \)
- Function: Copy row A to row B

### XOR2/XNOR2

- Based on XOR3/XNOR3

### Table

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operation</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>IML2x</td>
<td>A.B</td>
<td>AND2/NAND2</td>
</tr>
<tr>
<td>IML3x</td>
<td>A.B.C</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3y</td>
<td>A.B.C</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3z</td>
<td>A.B.C</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3a</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3b</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3c</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3d</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3e</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3f</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3g</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3h</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3i</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3j</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3k</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3l</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3m</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3n</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3o</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3p</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3q</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3r</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3s</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3t</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3u</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3v</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3w</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3x</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3y</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML3z</td>
<td>A.B.C.</td>
<td>AND3/NAND3</td>
</tr>
</tbody>
</table>

### XOR3

- Operation: \( A \oplus B \oplus C \)
- Truth Table:

<table>
<thead>
<tr>
<th>( A )</th>
<th>( B )</th>
<th>( C )</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
In-memory Adder (IML36)

- **Carry** is directly produced by MAJ function (IML33).
- **Sum** output is achieved by inverted Carry signal (MIN function) for 6 out of 8 possible input combinations.
- In two extreme cases (000 and 111), the MIN signal is disconnected and Sum is achieved by NOR3 (T1:ON, T2:OFF $\rightarrow$ Sum=0) and NAND3 (T1:OFF, T2:ON $\rightarrow$ Sum=1).

- Enable parallel One Computation per One Memory Cycle.
- Assume $M1$, $M2$ and $M3$ operands, 3-input in-memory logic schemes generates **Sum**(/Difference) and **Carry**(/Borrow) bits very efficiently.
Up-to-Now: Supported **Parallel** and **Reconfigurable** In-Memory Logic in ONE-cycle

<table>
<thead>
<tr>
<th>opcode FRC</th>
<th>operation</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>IML2x</td>
<td>B ← A</td>
<td>Copy row A to Row B</td>
</tr>
<tr>
<td>IML21</td>
<td>A.B</td>
<td>AND2/NAND2</td>
</tr>
<tr>
<td>IML22</td>
<td>A+B</td>
<td>OR2/NOR2</td>
</tr>
<tr>
<td>IML3x</td>
<td>A.B.C</td>
<td>AND3/NAND3</td>
</tr>
<tr>
<td>IML31</td>
<td>A+B+C</td>
<td>OR3/NOR3</td>
</tr>
<tr>
<td>IML32</td>
<td>AB+AC+BC</td>
<td>MAJ/MIN</td>
</tr>
<tr>
<td>IML33</td>
<td>A ⊕ B</td>
<td>XOR2/XNOR2</td>
</tr>
<tr>
<td>IML34</td>
<td>A ⊕ B ⊕ C</td>
<td>XOR3/XNOR3</td>
</tr>
<tr>
<td>IML35</td>
<td>Sum/Carry</td>
<td>add/sub</td>
</tr>
</tbody>
</table>

Area Overhead

**Configuration Table for a sample 512Mb memory**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Size</th>
<th>Activation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute. Sub-array</td>
<td>512×256</td>
<td>depending on in-memory OP.</td>
</tr>
<tr>
<td>Sub-array per Mat</td>
<td>8×8</td>
<td>64</td>
</tr>
<tr>
<td>Mat per Bank</td>
<td>2×2</td>
<td>2/2 and 2/2 as row and column activations</td>
</tr>
<tr>
<td>Bank per Group</td>
<td>4×4</td>
<td>1/4 and 4/4 as row and column activations</td>
</tr>
</tbody>
</table>

2-cycle in-memory FA and its application in DNN acceleration  
D. Fan. et. al. DAC 2019, ASPDAC 2019

1-cycle in-memory FA and its application in Graph processing and DNA sequence analysis  
D. Fan. et. al. DATE 2019, DAC 2019

Add-on area breakdown

- Ctrl: 23%
- Decoders & add-on mux: 31%
- WBL & RBL Drivers: 15%
- Output Drivers: 13%
- DPU: 10%
- SA: 7%
- <1%

~5.8%

Add-on area breakdown

- Ctrl: 21%
- Output Drivers: 14%
- Drivers: 6%
- SA: 9%
- DPU: 17%
- <1%

~7.9%
Recent Processing-in-Memory Platforms

- **Ambit: DRAM-based** [4]
  - Operand locality issue
  - Original data overwritten
  - Multi-cycle operations
  - Simple logic
  - Low area overhead
  - Hardware-friendly
  - Exploit the full bandwidth

- **Pinatubo: NVM-based** [5]
  - Operand locality issue
  - Modified SA
  - Simple logic
  - Medium area overhead
  - Support one-step multi-row operations
  - General platform

- **RIMPA: DWM-based** [6]
  - Operand locality issue
  - Large area overhead
  - Simple logic
  - Fast MG computation
  - Ultra-low power

Issues:
1. Only simple logic (7+ cycles for FA, intermediate data write-back)
2. Operand locality

Reconfigurable Logic-SA, one cycle logic
2. Operand locality?

---

**Approach-1: PIMA-Logic**

D. Fan, et. al, DAC 2018

In-memory logic for data either in the same word-line or bit-line

**Approach-2: Polymorphic Logic**

D. Fan, et. al, ISVLSI 2017 (best paper award), ASPDAC2018

Observation Now: 
**Operand locality vs. Parallel computing**

New non-volatile polymorphic logic as add-on to existing SA
SOT-MRAM 2T1R Device modeling and Parameters

- Area of the SOT-MRAM accelerators (in ASP-DAC 2018 [2] and DAC 2018 [3]) consists of two main components:
  1. MRAM die area, and 2. Add-on digital processing unit area.

- MRAM die area:
  ✓ **Device level:** 1- SOT-MTJ device modeling (w.r.t. Table 1’s parameters) and 2- calculating the amount of write and sense currents.
  ✓ **Circuit level:** 1- Calculating Access Transistor size for both read (~90nm) and write (~240nm) to provide such currents. 2- Developing the layout as Figure 1; Area of each two cells was determined to be (10λ × 32λ) + (10λ × 24λ) in 45 nm process node. 3- Designing peripheral circuitry enabling PIM (Modified SA, Decoder, etc.) and calculating overhead area.
  ✓ **Architectural level:** Applying the circuit-level configurations in memory scale.

- Digital processing unit area:
  ✓ **Digital processing unit consists of different sub-component such as:**
  1. **Activation functions**, developed using lookup-table-based transformations.
  2. **Batch normalization** (BN) unit generally performs an affine function \(y = kx + h\) [1], where \(y\) and \(x\) denote the corresponding output and input feature map pixels, respectively. Therefore, we employed an internal, multiplexed CMOS adder and multiplier to perform this computation efficiently.

---

To have a fair comparison, and to explore the area, energy, latency in different PIM platforms, we first developed an iso-Capacity 32Mb-single Bank memory unit using SOT-MRAM, STT-MRAM, RRAM, SRAM, and DRAM, as shown in next page.

Notes on the designs:
- SOT-MRAM design is developed based on our design in [1].
- STT-MRAM design is developed based on our design in [1] with standard and experimentally-measured configuration available in NVSIM [2].
- RRAM design is developed based on [3] with standard default configuration available in NVSIM [2].
- SRAM design is designed based on Compute Cache [4] method with following assumptions.
- DRAM design is designed based on Ambit [5].

### Other memory technology device parameters used in NVSIM and CACTI

- DAC, IM [2], 787.

---

**Table:**

<table>
<thead>
<tr>
<th>Memory Technology</th>
<th>CellArea (F^2)</th>
<th>CellAspectRatio</th>
<th>ResistanceOn (ohm)</th>
<th>ResistanceOff (ohm)</th>
<th>ReadMode: 1</th>
<th>ResetEnergy (pJ)</th>
<th>ResetCurrent (uA)</th>
<th>ResetPulse (ns)</th>
<th>AccessType: CMOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOT-MRAM</td>
<td>70</td>
<td></td>
<td>4612</td>
<td>15221</td>
<td>current</td>
<td>0.0298</td>
<td>130</td>
<td>1</td>
<td>CMOS</td>
</tr>
<tr>
<td>STT-MRAM</td>
<td>54</td>
<td></td>
<td>3000</td>
<td>6000</td>
<td>current</td>
<td>1</td>
<td>130</td>
<td>1</td>
<td>CMOS</td>
</tr>
<tr>
<td>RRAM</td>
<td>8</td>
<td>1</td>
<td>100000000</td>
<td>100000000</td>
<td>current</td>
<td>1</td>
<td>1</td>
<td>0.16</td>
<td>None</td>
</tr>
<tr>
<td>SRAM</td>
<td>146</td>
<td>1.46</td>
<td>1.31</td>
<td>1.31</td>
<td>voltage</td>
<td>1.46</td>
<td>0.0298</td>
<td>130</td>
<td>CMOS</td>
</tr>
<tr>
<td>DRAM</td>
<td>8</td>
<td>2.0</td>
<td>0.16</td>
<td>0.16</td>
<td>voltage</td>
<td>1.46</td>
<td>0.0298</td>
<td>130</td>
<td>CMOS</td>
</tr>
</tbody>
</table>

---


Simulation results for five different **Processing-in-Memory accelerators**

**iso-capacity:** 32Mb-single Bank, Data Width: 512-bit

Table developed with TSMC (published in ISVLSI 2019, “Accelerating Deep Neural Networks in Processing-in-Memory Platforms: Analog or Digital Approach?”)

<table>
<thead>
<tr>
<th>Metrics</th>
<th>SOT-MRAM</th>
<th>STT-MRAM</th>
<th>RRAM</th>
<th>SRAM</th>
<th>DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-volatility</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Area ($mm^2$)</td>
<td>Memory: 7.06 Logic: 0.3</td>
<td>Memory: 6.22 Logic: 0.3</td>
<td>Memory: 3.34 Logic: 2.5</td>
<td>Memory: 10.38 Logic: 0.5</td>
<td>Memory: 4.53 Logic: 0.04</td>
</tr>
<tr>
<td>Read Latency (ns)</td>
<td>2.85</td>
<td>2.89</td>
<td>1.48</td>
<td>2.9</td>
<td>3.4 per access</td>
</tr>
<tr>
<td>Write Latency (ns)</td>
<td>2.59</td>
<td>11.55</td>
<td>20.9</td>
<td>2.7</td>
<td>3.4 per access</td>
</tr>
<tr>
<td>Read Dynamic Energy (nJ)</td>
<td>0.57</td>
<td>0.65</td>
<td>0.38</td>
<td>0.34</td>
<td>0.66 per access</td>
</tr>
<tr>
<td>Write Dynamic Energy (nJ)</td>
<td>0.66</td>
<td>1.2</td>
<td>2.7</td>
<td>0.38</td>
<td>0.66 per access</td>
</tr>
<tr>
<td>In-Memory Logic Energy (nJ)</td>
<td>~0.64</td>
<td>~0.79</td>
<td>~1.96</td>
<td>~0.59</td>
<td>~0.75</td>
</tr>
<tr>
<td>Leakage Power (mW)</td>
<td>550</td>
<td>722.4</td>
<td>587.6</td>
<td>5243</td>
<td>335.5</td>
</tr>
<tr>
<td>Endurance</td>
<td>[1,2] $10^{14}$ - $10^{15}$</td>
<td>[1,2] $10^{14}$ - $10^{15}$</td>
<td>up to $10^{12}$ [3,4]</td>
<td>Unlimited</td>
<td>$10^{15}$</td>
</tr>
<tr>
<td>Data over-written issue</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

PIM logic area overhead including the modified decoder and SA (8-bit ADC for RRAM)

***Data is extracted using device-to-architecture simulations. The architectural level tools [5] and [6] are extensively modified based on circuit level results. Obviously by enlarging memory size, the reported numbers change correspondingly. Read latency parameter can be used as an estimation for computation latency.***

---


Observations

**SOT-MRAM**
- The smallest write latency while having an excellent endurance.
- The smallest write dynamic energy between other Processing in-NVM platforms.
- Medium area overhead as compared to Processing-in-RRAM platform considering the iso-capacity constraint.
- Small ON/OFF ratio

**STT-MRAM**
- Long write latency compared to SOT-MRAM that leads to much larger execution time specifically in write-intensive applications such as CNNs. Small ON/OFF ratio
- Large write dynamic energy compared to SOT-MRAM.

**ReRAM**
- Not only suffers from the low endurance but also imposes large write latency, dynamic energy and computation energy.
- The low endurance issue has been addressed through “Matrix Splitting” solution [1,2] by allocating excessive memory sub-arrays, sacrificing area and consuming extra write energy and latency to do the same task.

**SRAM**
- The Largest area overhead and leakage power consumption. Volatility
- Fast write/read

**DRAM**
- Data over-written issue. This problem has been alleviated through the back-up process [3], sacrificing area and consuming extra write energy and latency to perform the same task (e.g. it takes 6 cycles to preform AND operation).
- Volatility

---
Summary

• Non-Volatile Memory, like STT-MRAM, SOT-MRAM, ReRAM, could be designed to work as dual-mode memory with both functionalities of memory and logic using innovations in device and circuit.

• With limited area overhead, we could design in-memory logic, including AND/NAND, OR/NOR, XOR/XNOR, FA in only one-cycle. It provides powerful logic functions for any further development of architectural level computational ISAs for big-data processing-in-memory accelerator designs.

• The operand locality issue in one sub-array could be solved by sacrificing the parallel computing ability of individual sub-array. It is a trade-off between specific design to choose either data rearrange to get maximal papalism or no operand locality issue
Processing-In-Memory Unit

Processing-In-Memory unit to accelerate memory/data – intensive applications.

1. Intrinsic efficient built-in in-memory logic
2. Parallel computing at each sub-array
3. Greatly reduce data communication

Challenges:
- How to design most efficient architecture to fully utilize the supported in-memory logic ISA?
- How to modify or design new computation algorithm to make it intrinsically match with the developed PIM hardware platform

D. Fan, et. al., DAC’18/19/20, ICCAD’18/19, DATE’19/20, ICCD’17/18, ASPDAC’18/19/20, TCAD’18/19, TMAG’18, TNANO’18
Main Research Objective and Methodology

**Architecture:** Parallel Processing-in-Memory Accelerator

**Algorithm:** Data intensive and intelligent application algorithm development for developed PIM platform

**Developed and developing PIM applications:** deep neural network, data encryption, image processing, graph processing

Device & Circuits: NVM + CMOS

**Objective:** Energy Efficient and Intelligent Processing-in-Memory

**Bottom-Up**

**Device & Circuit Co-Design**

**Device & Circuit:** Parallel and Reconfigurable in-Memory Logic based on STT-MRAM, SOT-MRAM, DWM, ReRAM, DRAM, with extreme low overhead.

**Objective:** dual mode computational memory simultaneously working as memory and in-memory logic

---

D. Fan, et. al., CVPR’20/19, AAAI’20, ICCV’19, USENIX Security’20, DAC’20/19/18, ICCAD’19/18, DATE’20/19, TCAD’19/18, TMAG’20/18, etc.
Application: DNN-in-Memory

- Deep Convolutional neural networks (CNNs) are reaching record-breaking accuracy in image recognition on large data-sets like ImageNet, ResNet shows a prominent recognition accuracy (96.43%) even higher than humans! (94.9%).

- Following the trend, when going deeper and denser in CNNs (e.g. ResNet employs 18-1001 layers), memory/computational resources and their communication have faced inevitable limitations called “CNN power and memory wall”) [1,2].

- Several methods have been proposed to break the wall:
  A. Compressing pre-trained networks,
  B. Quantizing parameters
  C. Pruning
  D. Convolution decomposition

- Objective: Can we build a PIM hardware friendly DNN model:
  - Remove multiplication, ideally with bit-wise logic or addition-only
  - Hardware friendly model compression
  - Without losing inference accuracy?

---

Weight Ternarization

Ternarize all model weights from floating point number to \{-1, 0, +1\} states

Benefits and Challenges:

- Model size reduced by $16X$ from 32-bit floating point number
- Convolution computation only involves addition, and thus computing complexity for hardware greatly reduced
- Challenge is how to minimize the accuracy degradation as small as possible. no degradation ideally!

D. Fan, et. al., CVPR 2019, WACV 2019
Code to download in https://github.com/elliothe/Ternarized_Neural_Network
Proposed Ternarization Method with Iterative Statistical Scaling

Network training step:

① Initialize weight with pretrained model: 1) higher accuracy; 2) converges faster than training from scratch

② Iterative weight ternarization training

③ Back propagate to update full precision weight. Note that, straight through estimator of ternarization function in the back-propagation is used to approximate gradient.

D. Fan, et. al., CVPR 2019, WACV 2019
Proposed Ternarization Method with Iterative Statistical Scaling

(1) Weight Initialization

| +0.37 | +0.53 | -0.07 |
| +0.13 | -0.82 | -0.42 |
| +0.33 | +1.21 | -0.98 |

One-time Weight initialization from pretrained model

Forward: \( w'_i = \begin{cases} \alpha \times \text{Sign}(w_{l,i}) & |w_{l,i}| \geq \Delta_{th} \\ 0 & |w_{l,i}| < \Delta_{th} \end{cases} \)  

(1)

\[ \alpha = E(|w_{l,i}|), \quad \forall \{i \mid |w_{l,i}| \geq \Delta_{th}\} \]  

(2)

Scaling factors calculated by the mean of absolute values of designated layer’s full precision weights that are greater than the thresholds

Convolution computation converts to ternary convolution without multiplication and reduced model size

\[ x_i^T \cdot w'_i = x_i^T \cdot (\alpha \cdot \text{Tern}(w_l)) = \alpha \cdot (x_i^T \cdot \text{Tern}(w_l)) \]  

(3)

D. Fan, et. al., CVPR 2019, WACV 2019
Residual Expansion to Improve Accuracy

We ternarize the whole network including the first and last layer weights

\[ x^T \cdot w' + x^T \cdot w_r' = x^T \cdot (\alpha \cdot \text{Tern}(w) + \alpha_r \cdot \text{Tern}(w_r)) \]

- Residual Expanded Layers (REL) are added to reduce accuracy loss while maintaining no-multiplication operations in DNN.

- Original layer and residual layer are ternarized from the same full precision weights with different thresholds \( \beta = (a, b) \)
### Experiments - ImageNet

<table>
<thead>
<tr>
<th>Quan. scheme</th>
<th>First layer</th>
<th>Last layer</th>
<th>Accuracy (top1/top5%)</th>
<th>Comp. rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>BWN [13]</td>
<td>Bin.</td>
<td>FP</td>
<td>FP</td>
<td>60.8/83.0</td>
</tr>
<tr>
<td>ABC-Net [12]</td>
<td>Bin.</td>
<td>FP*</td>
<td>FP*</td>
<td>68.3/87.9</td>
</tr>
<tr>
<td>ADMM [10]</td>
<td>Bin.</td>
<td>FP*</td>
<td>FP*</td>
<td>64.8/86.2</td>
</tr>
<tr>
<td>TWN [10, 11]</td>
<td>Tern.</td>
<td>FP</td>
<td>FP</td>
<td>61.8/84.2</td>
</tr>
<tr>
<td>TTN [18]</td>
<td>Tern.</td>
<td>FP</td>
<td>FP</td>
<td>66.6/87.2</td>
</tr>
<tr>
<td>ADMM [10]</td>
<td>Tern.</td>
<td>FP*</td>
<td>FP*</td>
<td>67.0/87.5</td>
</tr>
<tr>
<td>Full precision</td>
<td>-</td>
<td>FP</td>
<td>FP</td>
<td>69.75/89.07</td>
</tr>
</tbody>
</table>

- **Best accuracy achieved with the same compression rate, even with ternarized first and last layers**

FP: Full precision weights
Bin: Binary weights
Tern: Ternary weights
FP*: not reported if first and last layers are full precision

Resent structure and Imagenet datasets are used here.
14 million images with 1000 output labels

---

Experiments- ImageNet, with Residual layers

• Only 0.42% accuracy degradation in imagenet if with one residual layer for **top1 accuracy**
• The top5 accuracy even **outperforms** full precision weight
• Top1 accuracy degradation reduces with more residual layers

Resent structure and Imagenet datasets are used here
14million images with 1000 output labels

ResNet structure and Imagenet datasets are used here
14million images with 1000 output labels

<table>
<thead>
<tr>
<th></th>
<th>First</th>
<th>Last</th>
<th>Accuracy (top1/top5)</th>
<th>Accuracy gap</th>
<th>Comp. rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full precision</td>
<td>FP</td>
<td>FP</td>
<td>69.75/89.07</td>
<td>-/-</td>
<td>1×</td>
</tr>
<tr>
<td>$T_{ex}=1$</td>
<td>FP</td>
<td>FP</td>
<td>67.95/88.0</td>
<td>-1.8/-1.0</td>
<td>~ 16×</td>
</tr>
<tr>
<td>$T_{ex}=1$</td>
<td>Tern</td>
<td>Tern</td>
<td>66.01/86.78</td>
<td>-3.74/-2.29</td>
<td>~ 16×</td>
</tr>
<tr>
<td>$T_{ex}=2$</td>
<td>FP</td>
<td>FP</td>
<td>69.33/89.68</td>
<td>-0.42/+0.61</td>
<td>~ 8×</td>
</tr>
<tr>
<td>$T_{ex}=2$</td>
<td>Tern</td>
<td>Tern</td>
<td>68.05/88.04</td>
<td>-1.70/-1.03</td>
<td>~ 8×</td>
</tr>
<tr>
<td>$T_{ex}=4$</td>
<td>Tern</td>
<td>Tern</td>
<td>69.44/88.91</td>
<td>-0.31/-0.16</td>
<td>~ 4×</td>
</tr>
</tbody>
</table>

FP: Full precision weights
Bin: Binary weights
Tern: Ternary weights
FP*: not reported if first and last layers are full precision

D. Fan, et. al., CVPR 2019, WACV 2019
BD-Net: A Multiplication-less DNN with Binarized Depthwise Separable Convolution

Binarize all model weights from floating point number to \{-1, +1\} states

**Benefits and Challenges:**
- Model size reduced by at least 32X (our best results: reduced by over 64X with only 6.59% accuracy degradation in ImageNet dataset)
- Convolution computation converts to XNOR, shift and bit-counter bit-wise operations, which greatly matches with our PIM hardware platform
- Real challenge is how to minimize the accuracy degradation as small as possible. no degradation ideally!

D. Fan, et. al., ISVLSI 2018 *(best paper award)*, ICCAD 2018
Depthwise Separable Convolution

Kernel size:
\[ p \cdot q \cdot kh \cdot kw \]

Standard convolution performs feature extraction and generates a new presentation within one layer:

\[ Y = \sum_{i=1}^{p} W_i \cdot X_i; \]

where:
\[ Y \in R^{h \times w \times q}, W = R^{kh \times kw}, X = R^{h \times w} \]

** Bias is not included in Conv. layers

Computational complexity:
\[ h \cdot w \cdot kh \cdot kw \cdot p \cdot q \]

Y = \( W_j^{dw} \cdot X_i \); \[ G = \sum_{j=1}^{m \cdot p} W_j^{pw} F_j; \] i ∈ [1, p], j ∈ [1, m · p], Y ∈ R\(^{h \times w \times pm}\),

F = R\(^{h \times w}\), \( W^{dw} = R^{kh \times kw}\), \( W^{pw} = R^{1 \times 1}\), X = R\(^{h \times w}\), G ∈ R\(^{h \times w \times q}\)

Variant Depthwise separable convolution:
- Depthwise conv: Extract features w.r.t. the depthwise conv kernel.
- Pointwise conv: Linearly combine the extracted feature maps to generate new representations.

Depthwise-Separable Convolution

- **Input channel**: $p$, **output channel**: $q$, kernel size: $kh \times kw$, input tensor dimension: $h \times w \times p$

- Functionality: perform the feature extraction and combination separately.

- Hardware resource: reduce the module size of convolution layer.

- Drop-in replacement of Normal spatial Convolution layer.

- 9X smaller computational cost when $m=1$, $kh=kw=3$ (*mobilenet [1]*)

\[
\frac{h \cdot w \cdot kh \cdot kw \cdot p \cdot m + h \cdot w \cdot p \cdot m \cdot q}{h \cdot w \cdot kh \cdot kw \cdot p \cdot q} = \frac{m(kh \cdot kw + q)}{kh \cdot kw \cdot q}
\]

- ~\(1/9\) @ $m=1$ & $kh=kw=3$

  e.g. MobileNet [1]

---

**References**

BD-Net: structure

- Depth-wise separable convolution is efficient, can we push even more with 1) no multiplication, 2) more compact model size, 3) no accuracy lose?
- Using the bypass structure of Residual Network as back-bone.
- Replace the normal spatial convolution layer with depthwise separable convolution.
- Introduce binary weight to depthwise convolution part.
- Introduce binary intermediate tensor to pointwise convolution part.

• Remove Multiplication from Convolution Operation
• model size is further reduced by weight binarization
BD-Net: training

- Using straight through estimator (STE) to approximate the gradient for making binarization function differentiable [1]

\[
\text{Forward: } q = \text{Sign}(r) = \begin{cases} 
+1 & \text{if } r \geq 0 \\
-1 & \text{otherwise}
\end{cases}
\]

\[
\text{Backward: } \frac{\partial g}{\partial r} = \begin{cases} 
\frac{\partial g}{\partial q} & \text{if } |r| \leq 1 \\
0 & \text{otherwise}
\end{cases}
\]

- We keep the gradient clipping for better performance (i.e., inference accuracy) [2]

**Network training step:**

- Initialize the weight (may achieve better performance when initialize from pretrained model)

- Iterative binarize the weights of depthwise kernel

- Update the full precision weight during back-propagation

---


BD-Net: hardware cost analysis

<table>
<thead>
<tr>
<th></th>
<th>Computation Cost</th>
<th>Memory Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mul–$O(N^2)$</td>
<td>Add/Sub–$O(N)$</td>
</tr>
<tr>
<td>CNN</td>
<td>$h \cdot w \cdot kh \cdot kw \cdot p \cdot q$</td>
<td>$kh \cdot kw \cdot p \cdot q \cdot N_{\text{bit}}^{32}$</td>
</tr>
<tr>
<td>This work</td>
<td>$h \cdot w \cdot kh \cdot kw \cdot p \cdot q$</td>
<td>$kh \cdot kw \cdot p \cdot m \cdot N_{\text{bit}}^{1}$</td>
</tr>
<tr>
<td></td>
<td>$h \cdot w \cdot kh \cdot kw \cdot p \cdot m$</td>
<td>$kh \cdot kw \cdot p \cdot m \cdot N_{\text{bit}}^{m}$</td>
</tr>
<tr>
<td></td>
<td>$+h \cdot w \cdot p \cdot m \cdot q$</td>
<td>$+p \cdot m \cdot q \cdot N_{\text{bit}}^{m}$</td>
</tr>
<tr>
<td>This work/CNN</td>
<td>0</td>
<td>$\frac{1}{q \cdot N_{\text{bit}}^{32}} + \frac{m \cdot N_{\text{bit}}^{m}}{kh \cdot kw \cdot N_{\text{bit}}^{32}}$</td>
</tr>
</tbody>
</table>

- $\cdot$ Input channel: $p$, output channel: $q$, kernel size: $kh \cdot kw$, input tensor dimension: $h \cdot w \cdot p$, channel multiplier: $m$
- $N_{\text{bit}}$ is the number of bits
- We use 32bit ($N_{\text{bit}}^{n=32}$) for pointwise layer in this work.
- Channel multiplier $m$ is the hyperparameters to optimize in this work.

~1/9 when $kh=kw=3$, $m=1$

>1/9 depending on bit-width of pointwise kernel
Experiments: Cifar and ImageNet

Framework: Pytorch (Good support for depthwise convolution)

Application: Object classification

Network configuration:
MNIST: 16 input channels, 5 basic blocks, 128 hidden neuron, 64 batch size, 3×3 kernel size, 4 channel expansion.
SVHN: 128 input channels, 5 basic blocks, 512 hidden neuron, 64 batch size, 3×3 kernel size, 4 channel expansion.
CIFAR-10: 128 input channels, 5 basic blocks, 512 hidden neuron, 64 batch size, 3×3 kernel size, 4 channel expansion.

ImageNet: ResNet-18 structure. 14million images with 1000 output labels

D. Fan, et. al., ISVLSI2018 (best paper award), ICCAD2018
Our model is only 143Kb with 8 conv layers and 1 FC layer.

DNN model completely stored in on-chip cache, no need to fetch model from main memory.

PYNQ-Z1 only has 4.9Mb on chip RAM and our model only consumes 2.61 W.
hardware-aware optimization by combining ternarization and structured pruning

published in **AAAI-2020** as **spotlight** paper

• Aim to effectively integrate structured weight pruning and ternarization to boost the performance of DNN inference on hardware platform, with ultra-small accuracy degradation

![Ternarization only](image1)

{white, grey, black} denotes {-1,0,+1}

![Ours](image2)

• **Problem-1**: Disharmony between structured pruning and ternarization weights of DNN, where the naive combination shows severe accuracy degradation.

• **Solution**: Self-Adjustable Weight Penalty Clipping (SA-WPC)

• **Problem2**: The structured pruning is decoupled from the hardware architecture of DNN accelerator.

• **Solution**: Processing element-wise (PE-wise) structured pruning (i.e., length of a group of weights to be pruned is identical to the capacity of PE), which is the computation core in hardware platform (i.e. FPGA, ASIC).

Li Yang, Zhezhi He and Deliang Fan, “Harmonious Coexistence of Structured Weight Pruning and Ternarization for Deep Neural Networks,” *Thirty-Fourth AAAI Conference on Artificial Intelligence* (AAAI), Feb. 7-12 2020, New York, USA (spotlight)
Naïve combination will not work

\[ \hat{\mathcal{L}} = \mathcal{L}(f(x; \{W_l\}_{l=1}^L), t) + \lambda \sum_{l=1}^L \sum_{i=1}^{G_l} \mathcal{P}(W_{l,i}) \]

**Group-Lasso pruning**
- \( \mu = -0.0 \)
- \( \sigma = 0.07 \)

**Weight ternarization**
- \( \mu = -0.0 \)
- \( \sigma = 0.02 \)

**Naïve combine**
- \( \mu = -0.0 \)
- \( \sigma = 0.07 \)
- \( \mu = -0.0 \)
- \( \sigma = 0.04 \)

**This work**
- \( \mu = -0.0 \)
- \( \sigma = 0.08 \)

\[ \hat{w}_{l,i} = \text{Tern}(w_{l,i}, \Delta_l) = \alpha_l \cdot \begin{cases} 
+1 & w_{l,i} > \Delta_l \\
0 & -\Delta_l \leq w_{l,i} \leq \Delta_l \\
-1 & w_{l,i} < -\Delta_l 
\end{cases} \]

\[ \alpha_l = \mathbb{E}(|\{w_{l,i}\}|), \ \forall i \ |W_{l,i}| > \Delta_l \]

- Is that applying the group Lasso (i.e., weight penalty) upon entire weights contradictory to the weight ternarization method?
- Whether maintaining the weight distribution close to the original ternary counterpart helpful to mitigate the accuracy degradation?
Proposed Methodology - Self adjustable weight penalty clipping

\[ \hat{L} = L(f(x; \text{Tern}\{W_l\}_{l=1}^L), t) \]

\[ + \lambda \sum_{l=1}^L \sum_{i=1}^{G_l} \min(\|W_{l,i}\|_2; \delta_l) \]

Weight Penalty Clipping

\[ \delta_l = a \cdot \frac{1}{G_l} \sum_{i=1}^{G_l} \|W_{l,i}\|_2 \]

Two cases:

When \( \|W_{l,i}\|_2 \geq \delta_l \), the weights are not pruned

When \( \|W_{l,i}\|_2 < \delta_l \), the weights are pruned in group-wise fashion
PE-wise pruning

- Weight size is $N \times C \times K_h \times K_w$, denotes the output channel (filter), input channel, kernel height and width in current layer, respectively.
- Processing element (PE) is the basic computation core in hardware platform (i.e. FPGA, ASIC)
- The group size is defined as the capacity of one PE

Pruning group pattern:

- Filter pruning: $C \times K_h \times K_w$
- Channel pruning: $N \times K_h \times K_w$
- PE-wise pruning: $C_g \times K_h \times K_w$, $C_g \in (0, C)$
Experiments results in large scale Imagnet dataset

Table: Inference accuracy (%) of ResNet18 on ImageNet

<table>
<thead>
<tr>
<th>Quan scheme</th>
<th>First layer</th>
<th>Last layer</th>
<th>Accuracy (top1/top5)</th>
<th>Comp. rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>-</td>
<td>FP</td>
<td>69.75/89.07</td>
<td>1×</td>
</tr>
<tr>
<td>BWN</td>
<td>Bin.</td>
<td>FP</td>
<td>60.8/83.0</td>
<td>~32×</td>
</tr>
<tr>
<td>ABC-Net</td>
<td>Bin.</td>
<td>FP</td>
<td>68.3/87.9</td>
<td>~6.4×</td>
</tr>
<tr>
<td>ADMM</td>
<td>Bin.</td>
<td>FP</td>
<td>64.8/86.2</td>
<td>~32×</td>
</tr>
<tr>
<td>TWN</td>
<td>Tern.</td>
<td>FP</td>
<td>61.8/84.2</td>
<td>~16×</td>
</tr>
<tr>
<td>TTN</td>
<td>Tern.</td>
<td>FP</td>
<td>66.6/87.2</td>
<td>~16×</td>
</tr>
<tr>
<td>ADMM (He 2019)</td>
<td>Tern.</td>
<td>FP</td>
<td>67.0/87.5</td>
<td>~16×</td>
</tr>
<tr>
<td>Ours</td>
<td>Tern.</td>
<td>FP</td>
<td>68.01/88.13</td>
<td>~21.3×</td>
</tr>
</tbody>
</table>

- Comparing with weight Bin. and Tern. without pruning
- Achieve both high accuracy and compression rate
- Even better accuracy and higher compression rate than ternary-only


Li Yang, Zhezhi He and Deliang Fan, “Harmonious Coexistence of Structured Weight Pruning and Ternarization for Deep Neural Networks,” Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI), Feb. 7-12 2020, New York, USA (spotlight)
Further optimization: Make DNN flexible and dynamic

Dynamic Processing-in-Memory Accelerator for Dynamic Channel-Adaptive Deep Neural Networks

• Published in DAC 2020 and ASPDAC 2020
Motivation of dynamic neural network

**Fixed** model compression techniques  VS  **Adaptive** real-world requirement

- Quantization
- Pruning
- Low rank approximation
- Knowledge distillation

- Dynamic hardware resource allocation
- Power budget
- Throughput requirement
- Dynamic workload...

Objective: How to generate an adaptive model that could adjust its model size to meet dynamic application requirement?

Dynamic Neural Network
Dynamic Deep Neural Network

**Step 1 Sub-nets generator**
- Generate non-uniform sub-nets by pruning method
- A single DNN model consists of multiple sub-nets with different model size
- Sub-nets partial share weights

**Step 2 Fused sub-nets training**
- A single ensemble loss function for multiple objective optimization.
- Sub-nets perform inference independently at different power, speed, accuracy
Step 1: Sub-nets generator

- **optimized group Lasso based pruning --- Clipped-LASSO**

\[
\tilde{\mathcal{L}} = \mathcal{L}(f(x; \{W_l\}_{l=1}^L), t) + \lambda \sum_{l=1}^L \sum_{i=1}^{C_l} \min(\|W_{l,i}\|_2; \delta_l) \\
\text{s.t. } \delta_l = a \cdot \frac{1}{C_l} \sum_{i=1}^{C_l} \|W_{l,i}\|_2
\]

- **Problem addressed by Clipped-LASSO**
  - Avoid **group-Lasso penalty** on important weights
  - Importance is determined by **comparing L2-norm of group weights to a clipping threshold.**
  - The **threshold dynamically changes w.r.t. the layer-wise sensitivity.**
Step2: Fused subnets training

- A single ensemble loss function for multiple objective optimization

\[ L_{ens} = Loss_1 + \cdots + Loss_n \]
Experiments results

Cifar-10 – ResNet20

<table>
<thead>
<tr>
<th>Network</th>
<th>Width</th>
<th>S-NN FP</th>
<th>S-NN FP</th>
<th>Ours 16-bit</th>
<th>Ours 8-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet20</td>
<td>1.0x</td>
<td>89.8</td>
<td>91.1</td>
<td>91.0</td>
<td>91.1</td>
</tr>
<tr>
<td></td>
<td>0.75x</td>
<td>88.4</td>
<td>90.2</td>
<td>89.8</td>
<td>89.8</td>
</tr>
<tr>
<td></td>
<td>0.5x</td>
<td>85.6</td>
<td>87.5</td>
<td>87.0</td>
<td>86.9</td>
</tr>
<tr>
<td></td>
<td>0.25x</td>
<td>79.5</td>
<td>81.0</td>
<td>80.7</td>
<td>80.6</td>
</tr>
</tbody>
</table>

ImageNet = ResNet50

<table>
<thead>
<tr>
<th>Network</th>
<th>Width</th>
<th>I-NN</th>
<th>S-NN</th>
<th>Ours</th>
<th>Params(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet50</td>
<td>1.0x</td>
<td>76.1</td>
<td>76.0</td>
<td>76.6</td>
<td>25.5</td>
</tr>
<tr>
<td></td>
<td>0.75x</td>
<td>74.7</td>
<td>74.9</td>
<td>75.4</td>
<td>14.7</td>
</tr>
<tr>
<td></td>
<td>0.5x</td>
<td>72.0</td>
<td>72.1</td>
<td>72.4</td>
<td>6.9</td>
</tr>
<tr>
<td></td>
<td>0.25x</td>
<td>63.8</td>
<td>65.0</td>
<td>65.2</td>
<td>2.0</td>
</tr>
</tbody>
</table>

- Comparing with S-NN[1], choose 4 different size of sub-nets, we have better results
- To efficiently implement on Processing-In-MRAM, we further quantize both activation and weights of convolutional layers to 16-bits and 8bits with ignorable accuracy loss.
- Teacher model is also ResNet20

FP: full precision

[2] Li Yang, et. al. ASPDAC 2020

Teacher model is ResNet101, student model is ResNet50
- Comparing with I-NN and S-NN, we have better results on all different sub-nets

I-NN: the sub-nets are trained individually
Experimental results - implementation in CPU/GPU

- ResNet20 on Cifar-10
- ResNet18 on ImageNet

- Trade-off between accuracy and latency on CPU (Intel Xeon) and GPU (NVIDIA Titan-Xp)
- 4 sub-nets with different model sizes that can be switched on-the-fly
- Comparing with uniform subnets, better accuracy can be achieved under same latency

Li Yang, Zhezhi He, Yu Cao and Deliang Fan. "Non-uniform DNN Structured Subnets Sampling for Dynamic Inference". In: 57th Design Automation Conference (DAC), San Francisco, CA, July 19-23, 2020
In-Memory Convolution Engine

- A potential solution to better address storage, computation, and data transfer bottlenecks of CNNs.
- This architecture mainly consists of Image Bank, Kernel Bank, bit-wise In-Memory Convolution Engine (IMCE), and Digital Processing Unit (DPU).
- Preprocessing:
  ✓ Assume Input fmaps (I) and Kernels (W) are stored in Image Banks and Kernel Banks of memory.
  ✓ Inputs need to be constantly quantized before mapping into computational sub-arrays. This step is performed using DPU’s Quantizer and then the results are mapped to IMCE’s sub-arrays.
  ✓ IMCE is realized through the proposed SOT-MRAM based computational sub-array.

D. Fan, et. al., ICCD 2018
Cross-Layer Simulation Framework Development

Device Level:
Verilog-A model of spintronic device developed based on micromagnetic OOMMF framework, modular spintronic library.

Circuit Level:
SPICE simulation to verify logic design and extract power-delay-reliability analysis

System Level:
modified self-consistent NVSim along with an in-house developed C++ code to verify the performance, Gem5 will be used to build cycle-accurate in-memory processing unit architecture.

Application Level:
quantized deep convolution neural network and Advanced Encryption Standard (AES) algorithm are used as case study applications, to show benefits of PIM in practical applications

[12] www.eda.ncsu.edu/wiki/FreePDK
Comparison (**iso-computation (CNN as example)**, Area-Energy-Latency requirement)

- Taking LeNET to run MNIST data-set with different PIM accelerators considering the area/energy due to the computation by calculating the number of undertaken crossbars or sub-arrays.

- STT-MRAM/ReRAM/DRAM design imposes a larger latency compared with SOT-MRAM mainly due to its long intermediate data write-back.

- ReRAM design data is taken from [1]. While the required memory area of ReRAM is less than magnetic counterparts, overall it imposes larger area due to matrix splitting and extra large add-on logic area overhead (up to ~80%) [1,2].

- While DRAM accelerator imposes the least possible area compared to other PIMs with iso-capacity constraint (~1%), it needs to access multiple sub-arrays to avoid data-written problem as well as fitting the network at the same time that resulted in a larger latency and area compared to non-volatile designs.

Estimated row Performance of different PIMs without Parallelism techniques

<table>
<thead>
<tr>
<th>Parameters</th>
<th>SOT-MRAM</th>
<th>STT-MRAM</th>
<th>ReRAM</th>
<th>SRAM</th>
<th>DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area ($mm^2$)</td>
<td>0.018</td>
<td>0.015</td>
<td>0.060</td>
<td>0.64</td>
<td>0.16</td>
</tr>
<tr>
<td>(memory + logic)</td>
<td>(0.0172±0.0008)</td>
<td>(0.0143±0.0007)</td>
<td>(0.011±0.049)</td>
<td>(0.608±0.032)</td>
<td>(0.158±0.002)</td>
</tr>
<tr>
<td>Energy ($\mu J$)</td>
<td>0.74</td>
<td>1.3</td>
<td>13.5</td>
<td>1.6</td>
<td>2.1</td>
</tr>
<tr>
<td>(write-back+read-based Ops)</td>
<td>~(0.2±0.54)</td>
<td>~(0.55±0.75)</td>
<td>~(5.1±8.4)</td>
<td>~(0.42±1.18)</td>
<td>~(0.8±1.3)</td>
</tr>
<tr>
<td>Latency (ms)</td>
<td>0.4</td>
<td>2.6</td>
<td>5.8</td>
<td>0.7</td>
<td>13.5</td>
</tr>
</tbody>
</table>


For in-memory-logic, all operands are stored in memory. Unlike traditional computation, an extra data write-back is needed, which has a large effect on determining the overall energy and latency.
Performance Evaluation

ICCD 2018 [9]
256x512 columns per mat
512Mb total capacity

Vs.
ReRAM
A Prime-like [10] accelerator

GPU
NVIDIA GTX 1080Ti Pascal

ASIC

Energy Efficiency
- 18x ASIC64
- 9.2x ReRAM
- 25.6x GPU

Performance
- 18x ASIC64
- 5.4x ReRAM
- 22x GPU

- PIM-TGAN and ReRAM solutions spend less than ~20% time for memory access and data transfer.
- PIM-TGAN can efficiently utilize up to 55% of its resources.

Software: Adversarial Input Attack

adversarial example—a type of malicious inputs crafted by adding small and often imperceptible perturbations to legal inputs [1].

a major concern in many DNN-powered applications.

Our method to defend: parametric noise injection (PNI) includes trainable Gaussian noise injection at each layer of DNN’s activation or weight through solving a min-max optimization problem (published in CVPR 2019 [2])


Hardware: Adversarial Weight Attack?

Our first work discovering this is published in ICCV’19 [3] and USENIX Security’20[4]

Method: We propose a Progressive Bit Search (PBS) method which combines gradient ranking and progressive search to identify the most vulnerable bit to be flipped in DNN.

Result: 13 bit-flips out of 93 million bits to completely make ReseNet 18 malfunction in ImageNET (accuracy degrades from 69.8% to 0.1%)

A working prototype based on DRAM row-hammer attack has been developed and published in USENIX Security’2020[4]

Data-Intensive Processing-in-Memory: Data Encryption

**Objective:** Energy Efficient and Intelligent Processing-in-Memory

**Algorithm and Architecture Co-Design**

**Architecture:** Parallel Processing-in-Memory Accelerator

**Algorithm:** Data intensive and intelligent application algorithm development for developed PIM platform

**Developed and developing PIM applications:** deep neural network, data encryption, image processing, graph processing

---

**Device & Circuits Co-Design**

**Device & Circuit:** Parallel and Reconfigurable in-Memory Logic based on STT-MRAM, SOT-MRAM, DWM, ReRAM, DRAM, with extreme low overhead.

**Objective:** dual mode computational memory simultaneously working as memory and in-memory logic

---


Application: Why Energy Efficient Data Encryption?

- Big data: 2.5 quintillion ($10^{18}$) bytes of data everyday.
- IOT: 17.68 billion IOT device in 2017
- Cost of a breach has risen to $4 million per incident.
- From Data center to personal electronics, data are stored everywhere. The demand of energy efficient and high performance cryptographic components is becoming much stronger nowadays and will keep growing rapidly in the future.

This chart shows the average cost of a stolen record—for example, personally identifiable, payment, or health information on an individual—as broken out by industry.

IN-MEMORY DATA ENCRYPTION ENGINE

- Parallel, local data processing
- Short memory access latency
- Ultra-low energy
- Secure data where they stored
- Reduce data communication risk

- AES is an iterative symmetric-key cipher where both sender and receiver units use a single key for encryption and decryption.

Advanced Encryption Standard

• AES basically works on the standard input length of 16 bytes (128 bits) data organized in a 4 x 4 matrix (called the state matrix) while using three different key lengths (128, 192, and 256 bits).

• For 128-bit key length, AES encrypts the input data after 10 rounds of consecutive transformations.

• Four transformations:
  • SubBytes: LUT
  • ShiftRows: shift
  • MixColumns: XOR, shift
  • AddRoundKey: XOR
**Evaluation**

<table>
<thead>
<tr>
<th>Platforms</th>
<th>Energy (nJ)</th>
<th>Cycles</th>
<th>Area (µm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPP [5]</td>
<td>460</td>
<td>2309</td>
<td>2.5e+6</td>
</tr>
<tr>
<td>ASIC [6]</td>
<td>6.6</td>
<td>336</td>
<td>4400</td>
</tr>
<tr>
<td>CMOL [7]</td>
<td>10.3</td>
<td>470</td>
<td>320</td>
</tr>
<tr>
<td>Baseline DW [4]</td>
<td>2.4</td>
<td>1022</td>
<td>78</td>
</tr>
<tr>
<td>Pipelined DW [4]</td>
<td>2.3</td>
<td>2652</td>
<td>83</td>
</tr>
<tr>
<td>Multi-issue DW [4]</td>
<td>2.7</td>
<td>1320</td>
<td>155</td>
</tr>
<tr>
<td><strong>Ours: ASP-DAC 2018 [1]</strong></td>
<td>3.2</td>
<td>1620</td>
<td>21.8</td>
</tr>
<tr>
<td><strong>Ours: IEEE TCAD 2018 [2]</strong></td>
<td>1.74</td>
<td>2168</td>
<td>127</td>
</tr>
<tr>
<td><strong>Ours: DAC 2018 [3]</strong></td>
<td>1.5</td>
<td>872</td>
<td>27</td>
</tr>
</tbody>
</table>

- In-Memory Data Encryption based on SOT-MRAM significantly improves the data encryption performance by having the least energy consumption and latency in comparison.
- This significant improvement mainly comes from our proposed massive in-memory parallelism computing and intrinsic in-memory logic operations.

---

Data-Intensive Processing-in-Memory: Graph Processing

**Objective:** Energy Efficient and Intelligent Processing-in-Memory

**Algorithm and Architecture Co-Design**

**Device & Circuits:** NVM + CMOS

**Bottom-Up**

**Device & Circuit Co-Design**

**Objective:** dual mode computational memory simultaneously working as memory and in-memory logic

**Device & Circuit:** Parallel and Reconfigurable in-Memory Logic based on STT-MRAM, SOT-MRAM, DWM, ReRAM, DRAM, with extreme low overhead.

D. Fan, et. al., “GraphS: A Graph Processing Accelerator Leveraging SOT-MRAM” published in *Design, Automation and Test in Europe (DATE), 2019*

D. Fan. Et. al., “GraphDe...”, GLSVLSI 2019 (best paper award)
**Application: Graph Processing**

- **Huge Graphs** are Everywhere!
- Graph Topology Analysis helps us better understand the intricate connectivity of networks in practical problems.

- **Unique Challenges:**
  - Difficult to parallelize by data partitioning with current Von-Neumann architecture
  - Varying degree of parallelism over the course of execution – Unbalanced workloads
  - Locality of memory access by graph algorithms, etc.
  - I/O intensive – waits for memory fetches
  - Random access
  - Poor locality
  - Heavy conflict
Graph Processing: Partitioning & Mapping Strategy

- **Graph Partitioning is important!!** e.g. Interval-block partitioning method [3,5] to balance workloads of each PIM’s chip and maximize parallelism.

---

**Data Partitioning and Allocation**

1. **GraphQ [1]**
   - HMC

2. **GraphS [2]**
   - MRAM

3. **GraphH [3]**
   - HMC

4. **GraphR [4]**
   - ReRAM

---


Graph Processing: Operation Perspective

- **Graph Partitioning**: Interval-block partitioning method [2,3] to balance workloads of each PIM’s chip and maximize parallelism.
- **Degree centrality** needs massive number of add operation.
- It basically counts the number of valid links connected to a vertex.

**Steps:**
- Convert the graph to adjacency matrix,
- Partitioning and Mapping to Compute. Sub-arrays,
- Parallel in-memory **Addition**:

---

Graph Processing: Operation Perspective

- **Matching index** quantifies the “similarity” between two vertices based on the number of common neighbors shared by vertices.

\[
\frac{\sum \text{common neighbors}}{\sum \text{total number of neighbors}}
\]

- Steps:
  - Convert the graph to adjacency matrix,
  - Partitioning and Mapping to Compute. Sub-arrays,
  - Parallel in-memory **AND2** and **OR2**:

  1. To find the **common neighbors** of two particular vertices (e.g. V1, V2), **PIM** performs parallel **AND2** on the rows and SA’s outputs determine the matches (here, V4).

  2. Total number of neighbors is found by performing **OR2** operation on the same rows.

  3. \(\sum\) is easily implemented by add operation.

Graph Processing: Evaluation

**GraphS [1]**
512×256 sub-array-4×4 mats per bank in H-tree routing manner, 16×16 banks and 512Mb total capacity

**DRAM [2]**
An Ambit-like accelerator based TRA mechanism.

**STT-MRAM [3]**
An Pinatubo-like accelerator by modifying memory sense amplifiers.

**HMC [5]**
A conventional architecture presented in using HMC as main memory without utilizing instruction offloading functionality.

**GPU**
NVIDIA GTX 1080Ti Pascal GPU. It has 3584 CUDA cores running at 1.5GHz (11TFLOPs peak performance).

**ReRAM [4]**
An MPIM-like accelerator with 256 sub-arrays and one buffer sub-array per Bank. Each mat has 256x256 ReRAM cells.

### 3 SOCIAL NETWORK DATA-SETS

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nodes</th>
<th>Edges</th>
<th>Graph Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>ego-Facebook</td>
<td>4,039</td>
<td>88,234</td>
<td>profiles &amp; friends lists from Facebook [17]</td>
</tr>
<tr>
<td>dblp-2010</td>
<td>326,186</td>
<td>1,615,400</td>
<td>scientific collaboration network</td>
</tr>
<tr>
<td>amazon-2008</td>
<td>735,323</td>
<td>5,158,388</td>
<td>similarity among books reported by Amazon store</td>
</tr>
</tbody>
</table>

### 3 GRAPH PROCESSING TASKs:
- Degree centrality
- Matching index
- Breadth First Search (BFS)
Energy Efficiency

- **1.7x** STT-MRAM
- **3.6x** ReRAM
- **4x** DRAM

Energy Consumption and Execution time for performing 3 similar tasks:

**Energy Consumption of different accelerators**

<table>
<thead>
<tr>
<th>Accelerator</th>
<th>Energy (J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphS</td>
<td>1.2x</td>
</tr>
<tr>
<td>Piantubo</td>
<td>2.0x</td>
</tr>
<tr>
<td>MPIM</td>
<td>3.0x</td>
</tr>
<tr>
<td>Ambit</td>
<td>5.0x</td>
</tr>
</tbody>
</table>

**Execution time of different accelerators**

<table>
<thead>
<tr>
<th>Accelerator</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphS</td>
<td>1.0x</td>
</tr>
<tr>
<td>Piantubo</td>
<td>1.2x</td>
</tr>
<tr>
<td>MPIM</td>
<td>1.4x</td>
</tr>
<tr>
<td>Ambit</td>
<td>1.6x</td>
</tr>
</tbody>
</table>

Energy saving and speed-up of matching index task on GraphS and baseline HMC normalized to GPU on amazon-2008.

- **18.4x** Energy saving
- **12.6x** Speed-up

- **2.5x** STT-MRAM
- **5.3x** ReRAM
- **5.1x** DRAM

Graph Processing: Evaluation


Data-Intensive Processing-in-Memory: DNA Alignment

**Architecture**: Parallel Processing-in-Memory Accelerator

**Algorithm**: Data intensive and intelligent application algorithm development for developed PIM platform

**Developed and developing PIM applications**: deep neural network, data encryption, image processing, graph processing

---

**Device and Circuit**:
- Parallel and Reconfigurable in-Memory Logic based on STT-MRAM, SOT-MRAM, DWM, ReRAM, DRAM, with extreme low overhead.
- **Objective**: dual mode computational memory simultaneously working as memory and in-memory logic

---

**Algorithm** and **Architecture Co-Design**

**Device and Circuit Co-Design**

---

**DNA Alignment-in-memory**

---

Algorithm 1 DNA Exact Alignment-in-Memory.

---

D. Fan, et. al., “AlignS: ....” DAC, 2019
Application: DNA Alignment

- Goal: Find the complete sequence of A, C, G, T’s in DNA.
- Main Challenge: Heavily bottlenecked by Data Movement
- Solution: Processing-in-memory can alleviate the bottleneck
- Hardware-Software co-optimization needed!
- Developing mapping & filtering algorithms to fit PIM

1. Sequencing
2. Read Alignment
3. Variant Calling
4. Discovery

Bottlenecked in Alignment!!

~3.2B bps

Illumina HiSeq2000

PIM-Friendly Sequencing Algorithm

- Alignment algorithms find all occurrences of the $m$-bp short-read $R$ in the $n$-bp reference genome-$S$.

- **Burrows-Wheeler transform**!

- **BW matrix** is constructed by circulating $S$ and lexicographically sorting them. BWT of $S$ is given by the last column in the BW matrix.

- **Suffix Array (SA)** of a $S$ is a lexicographically-sorted array of the suffixes of $S$.

- **FM-Index** is built on top of BWT providing the occurrence information of each symbol in BWT.

- **Count** represents # of nt in the first column of BW

- Backward search and updating high and low pointers! to get SA interval. If low<high, $R$ has found a match in $S$.

---

Algorithm 1 DNA Exact Alignment-in-Memory.

Require : Pre-Compute and Data Mapping to DRIM: Partition pre-computed BWT, Marker Table (MT) and Suffix Array (SA) into DRIM chip.

input: DNA Short Read-R
output: positions of short read-R in reference genome-S

Step-1. Initialization:
1: \( \text{low} \leftarrow 0, \text{high} \leftarrow |S| - 1 \)

Step-2. Backward Search:
2: \( \text{for } i := |R| - 1 \text{ to } 0 \text{ do} \)
3: \( \text{low} \leftarrow \text{Bound}(M_T[low/d], R[i], \text{low}) \)
4: \( \text{high} \leftarrow \text{Bound}(M_T[high/d], R[i], \text{high}) \)
5: \( \text{if low} \geq \text{high then} \)
6: \( \text{break} & \text{return 0} \quad \triangleright \text{there is no exact alignment} \)
7: \( \text{end if} \)
8: \( \text{end for} \)

Step-3. Get matched positions from stored suffix array based on search result:
9: \( \text{for } j := \text{low} \text{ to high} - 1 \text{ do} \)
10: \( \text{positions} \leftarrow \text{MEM}(S_A[j]) \quad \triangleright \text{Read positions from Suffix Array memory} \)
11: \( \text{end for} \)

Define procedure \text{Bound}:
12: \( \text{Procedure: Bound}(M_T, nt, id) \quad \triangleright \text{compute matched interval} \)
13: \( \text{count_match} \leftarrow 0 \)
14: \( \text{for } j := 0 \text{ to } (id \mod d) \text{ do} \quad \triangleright \text{count number of nt within the BWT region} \)
15: \( \text{if XNOR\_Match(nt,BWT[id-(id mod d)+j]) == 1 then} \)
16: \( \text{count_match} = \text{count_match} + 1 \)
17: \( \text{end if} \)
18: \( \text{end for} \)
19: \( \text{marker} \leftarrow \text{MEM}(M_T[id/d], nt) \quad \triangleright \text{Read Marker Table value} \)
20: \( \text{return IM\_ADD(marker, count_match)} \)
21: \( \text{end Procedure} \)

Pre-computation needed in alignment algorithm

Read searching mainly implemented through the iteratively-used \text{Bound}(MT, nt, id) procedure.

Optimized using AlignS’s [1] functions

\text{i. MEM, ii. XNOR-match, iii. IM\_ADD}

---

Correlated Data Partitioning to fully leverage AlignS’s parallelism, and to maximize alignment computation throughput.

Localizing the computation by storing the accessed memory region of MT and BWT the same memory sub-array.

Bound function in-memory implementation:

i. Parallel Search (Comparison with Bulk XNOR2 Op)

ii. Rank Calculation (Addition)
DNA Alignment: Evaluation

**Methodology**

We generate 10 million 100-bp short read queries via **ART simulator** and align them to the human genome **Hg19**.

**Dynamic Programming** (Smith-Waterman)

- SW-based Darwin [1]
- RaceLogic [2]
- ReCAM-ReRAM [3]

**FM-Index**

- Soap2 on CPU [4]
- Soap3-dp on GPU [4]
- AligneR-ReRAM [5]
- FPGA [6]
- ASIC [7]

---

DNA Alignment: Evaluation

- AlignS stands as **third-best power-efficient** design after ASIC and AligneR

- AlignS stands as **third-fastest** design after Race and Darwin

- Throughput/Watt results:
  - ✓ 4.8x the best SW accelerator **Race Logic**
  - ✓ 1.6x **AligneR**
  - ✓ 3.4x **ASIC**
  - ✓ 67.5x **FPGA**

- Throughput/Watt/mm² results:
  - ✓ 140x **Race Logic**
  - ✓ 10% **AligneR**
  - ✓ 12x **ASIC**
  - ✓ 1570x **FPGA**

---

Applications Demanding PIMs

- General-purpose PIM or accelerator-PIM?
- What applications are suitable for PIM (most digital PIM could do bulk bit-wise logic)?
- Application-Hardware Co-Design needed?
Summary on PIM Applications

• Can every data-intensive application be mapped & accelerated by PIMs? accelerator or general-purpose? Ideally yes since many digital PIMs provide complete Boolean logic, but there are many challenges (in-memory logic circuit design, architecture, instructions, compiler) need to be addressed for a general-purpose PIM. That is why many state-of-the-art PIM is focusing on accelerator designs.

• What applications are suitable for PIM (most digital PIM could do bulk bit-wise logic)? Based on the discussed four different applications, if using PIM as an accelerator, it is better to have PIM-friendly operations that are heavily and repeatedly used. Meanwhile, data mapping/re-organizing is essential.

• Application-Hardware co-design needed? It depends. In some cases, application-hardware co-design is greatly helpful to make the application algorithm efficient for the target PIM hardware. PIM-aware optimization is essential
Summary

• Non-Volatile Memory, like STT-MRAM, SOT-MRAM, ReRAM, could be designed to work as dual-mode memory with both functionalities of memory and logic using innovations in device, circuit and architecture.

• In Device & Circuit layer, we have designed different types of in-memory logic circuit designs that could implement complete Boolean Logic, majority gate, full adder in only one cycle. These logic designs either target for highly parallel computing or to overcome the well known operand locality issue.

• Co-optimization of architecture & algorithm: The dual-mode computational memory could be utilized to accelerate data/compute-intensive applications, such as deep neural network, data encryption, image processing, graph processing, etc.

• The significant improvement mainly comes from our proposed optimized algorithm, massive in-memory parallel computing, data communication reduction and efficient in-memory logic circuits.
Thank You & Questions?

Deliang Fan
Assistant Professor, Ph.D.
School of Electrical, Computer and Energy Engineering
Arizona State University, Tempe, AZ, 85287, USA
Email: dfan@asu.edu
https://dfan.engineering.asu.edu/

Thanks to my students:
Zhezhi He, Shaahin Angizi, Adnan Rakin, Li Yang
Related publications

1. [SOCC'20] Li Yang, Zhezhi He, Shaahin Angizi and Deliang Fan, “Processing-In-Memory Accelerator for Dynamic Neural Network with Run-Time Tuning of Accuracy, Power and Latency,” 33rd IEEE International System-on-Chip Conference (SOCC), September 8-11, 2020 (invited)


13. [AAA'I20] Li Yang, Zhezhi He and Deliang Fan, “Harmonious Coexistence of Structured Weight Pruning and Ternarization for Deep Neural Networks,” Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI), Feb. 7-12 2020, New York, USA (spotlight) [pdf]


Related publications

- [CVPR’19] Zhezhi He*, Adnan Siraj Rakin* and Deliang Fan, “Parametric Noise Injection: Trainable Randomness to Improve Deep Neural Network Robustness against Adversarial Attack,” Conference on Computer Vision and Pattern Recognition (CVPR), June 16-20, 2019, Long Beach, CA, USA (* The first two authors contributed equally)
- [CVPR’19] Zhezhi He and Deliang Fan, “Simultaneously Optimizing Weight and Quantizer of Ternary Neural Network using Truncated Gaussian Approximation,” Conference on Computer Vision and Pattern Recognition (CVPR), June 16-20, 2019, Long Beach, CA, USA (accepted)
- [DAC’19] Shaahin Angizi, Jiao Sun, Wei Zhang and Deliang Fan, “AlignS: A Processing-In-Memory Accelerator for DNA Short Read Alignment Leveraging SOT-MRAM,” Design Automation Conference (DAC), June 2-6, 2019, Las Vegas, NV, USA
- [DAC’19] Zhezhi He, Jie Lin, Rickard Ewetz, Jian-Shun Yuan and Deliang Fan, “Noise Injection Adaptation: End-to-End ReRAM Crossbar Non-ideal Effect Adaption for Neural Network Mapping,” Design Automation Conference (DAC), June 2-6, 2019, Las Vegas, NV, USA
- [ICCAD’18] Shaahin Angizi, Zhezhi He and Deliang Fan, “DIMA: A Depthwise CNN In-Memory Accelerator,” IEEE/ACM International Conference on Computer Aided Design (ICCAD), Nov. 5-8, 2018, San Diego, CA, USA
- [ISVLSI’18] Zhezhi He, Shaahin Angizi, Adnan Siraj Rakin and Deliang Fan, “BD-NET: A Multiplication-less DNN with Binarized Depthwise Separable Convolution,” IEEE Computer Society Annual Symposium on VLSI, July 9-11, 2018, Hong Kong, China (Best Paper Award)
- [ISVLSI’17] F. Parveen, Z. He, S. Angizi, and D. Fan, “Hybrid Polymorphic Logic Gate with 5-Terminal Magnetic Domain Wall Motion Device,” IEEE Computer Society Annual Symposium on VLSI, July 3-5, 2017, Bochum, Germany (Best Paper Award)
- [ASPDAC’19] Shaahin Angizi, Zhezhi He and Deliang Fan, “ParaNN: A Parallel In-Situ Accelerator for Binary-Weight Deep Neural Networks,” Asia and South Pacific Design Automation Conference (ASP-DAC), Jan. 21-23, 2019, Tokyo, Japan
Related publications

- [ISVLSI’18] Zhehi He, Shaahin Angizi and Deliang Fan, “Accelerating Low Bit-Width Deep Convolution Neural Network in MRAM,” IEEE Computer Society Annual Symposium on VLSI, July 9-11, 2018, Hong Kong, CHINA (invited)
- [GLSVLSI’18] Shaahin Angizi, Zhehi He, Yu Bai, Jie Han, Mingjie Lin and Deliang Fan, “Leveraging Spintronic Devices for Efficient Approximate Logic and Stochastic Neural Network,” ACM Great Lakes Symposium on VLSI, Chicago, IL, USA, May 23-25, 2018 (invited)