PPAC: A Versatile In-Memory Accelerator for Matrix-Vector-Product-Product-Like Operations

Oscar Castañeda

Joint work with Maria Bobbett, Alexandra Gallyas-Sanhueza, and Christoph Studer
PIM breaks through the memory wall

Processing in memory (PIM)

Traditional (non-PIM) architectures

Flexible (PIM) architectures

Von Neumann-based architectures

Efficiency vs. Flexibility

ASIC → GPU → FPGA → CPU

PIM increases efficiency!

↑ Time
↑ Energy
↑ Bandwidth

Efficiency

Memory

Logic

Control

ALU
Exploring PIM architectures

- **Application-specific PIM [3]**
  - High throughput and efficient architecture tailored for a given application
  - Often only support a single task: Flexibility ↓

- **This work: PPAC**
  - An efficient yet flexible PIM accelerator

- **General-purpose PIM [1,2]**
  - Hardware support for atomistic operations (e.g., AND/NOR)
  - More complex tasks repeatedly use atomistic operations: Throughput ↓, Efficiency ↓

---

PPAC builds upon CAMs

- **Parallel Processor in Associative Content-addressable memory**
- Each CAM row compares its stored word $a_m$ with the input $x$
- **Hamming similarity**: Number of matching bits between $a_m$ and $x$
  - $h(a_m, x) = N - h(a_m, x)$, where $h$ is the Hamming distance and $N$ the number of bits in $a_m$ and $x$
- A CAM declares a match if $h(a_m, x) = N$
PPAC computes Hamming similarities

• Every row computes and processes \( \tilde{h}(a_m, x) \)
• Programmable threshold \( \delta \):
  Match if \( \tilde{h}(a_m, x) \geq \delta_m \)
  • Complete match: \( \delta_m = N \) (standard CAM)
  • Similarity match: \( \delta_m < N \)

Hamming similarity

• Single-cycle operation
✓ CAM Applications [1]
  • Network switches and routers
  • Computer caches
✓ Content Addressable Parallel Processor (CAPP) [2]
  • Based on similarity matches
✓ Particle track reconstruction [3]
✓ Approx. nearest neighbor search
  • Locality sensitive hashing
  • Indoor localization via fingerprinting

Exploiting Hamming similarities

<table>
<thead>
<tr>
<th>a</th>
<th>x</th>
<th>a XNOR x</th>
</tr>
</thead>
<tbody>
<tr>
<td>HI</td>
<td>HI</td>
<td>HI</td>
</tr>
<tr>
<td>HI</td>
<td>LO</td>
<td>LO</td>
</tr>
<tr>
<td>LO</td>
<td>HI</td>
<td>LO</td>
</tr>
<tr>
<td>LO</td>
<td>LO</td>
<td>HI</td>
</tr>
</tbody>
</table>

\[ \langle a_m, x \rangle = \sum_{n=1}^{N} a_{m,n} x_n \]

\[ \langle a_m, x \rangle = -N + 2\bar{h}(a_m, x) \]

\[
\begin{bmatrix}
\langle a_1, x \rangle \\
\vdots \\
\langle a_M, x \rangle
\end{bmatrix} = A \cdot x
\]

1-bit matrix-vector product (MVP)

- Single-cycle operation
- Binarized neural networks [1]
- mmWave/THz equalization

The PPAC architecture

- 2D array of latch-based bit-cells
- Each of the $M$ rows stores an $N$-bit word and has a row ALU
- Multiple rows are grouped in a bank

- Bit-cell with two operators
  - XNOR/AND
- All bit-cells on the same column share input and control signals
- A row is divided into subrows with local pop counts for scalability

A population count (pop count for short) corresponds to counting how many ones are there in a set of 1-bit numbers.
• Adds local pop counts to compute the row population count $r_m$

• Two accumulators:
  • First one for multi-bit vectors; includes offsets to correctly interpret $r_m$
  • Second for multi-bit matrixes

• A programmable threshold $\delta_m$ is subtracted from the second’s accumulator output to obtain $y_m$
  • Can be used as a comparator for declaring exact/similarity matches

• Pipeline stage after row population count to increase throughput
Operating MVPs in different number formats

• Key for building standard (multi-bit) unsigned and 2’s complement signed arithmetic, and other operations!
Executing an MVP with a multi-bit vector

- Assume \( \mathbf{A} \) has 1-bit entries, while \( \mathbf{x} \) has \( L \)-bit entries
- Bit-serial execution in \( L \) clock cycles
  - Operate MSBs of \( \mathbf{x} \) first (\( \mathbf{x}_L \)), LSBs last (\( \mathbf{x}_1 \))
  - Result of \( \mathbf{A}\mathbf{x}_L \) can be negated for 2’s complement arithmetic

\[
\mathbf{x} = \sum_{\ell=1}^{L} 2^{\ell-1} \mathbf{x}_\ell
\]

\[
\mathbf{A}\mathbf{x} = \sum_{\ell=1}^{L} 2^{\ell-1} \mathbf{A}\mathbf{x}_\ell
\]
Executing an MVP with a multi-bit vector

- Assume $A$ has 1-bit entries, while $x$ has $L$-bit entries
- Bit-serial execution in $L$ clock cycles
  - Operate MSBs of $x$ first ($x_L$), LSBs last ($x_1$)
  - Result of $Ax_L$ can be negated for 2’s complement arithmetic

$$x = \sum_{\ell=1}^{L} 2^{\ell-1} x_\ell$$

$$Ax = \sum_{\ell=1}^{L} 2^{\ell-1} Ax_\ell$$
**Executing an MVP with a multi-bit matrix**

- Assume $\mathbf{A}$ has $K$-bit entries and $\mathbf{x}$ has $L$-bit entries
- Bit-serial execution in $KL$ clock cycles
  - Same idea as multi-bit vectors, but memory cannot be changed fast enough [1]
  - Operate MSBs of $\mathbf{A}$ first ($\mathbf{A}_K \mathbf{x}$), LSBs last ($\mathbf{A}_1 \mathbf{x}$)
  - Result of $\mathbf{A}_k \mathbf{x}$ can be negated for 2’s complement arithmetic

Executing an MVP with a multi-bit matrix

- Assume $A$ has $K$-bit entries and $x$ has $L$-bit entries
- Bit-serial execution in $KL$ clock cycles
  - Same idea as multi-bit vectors, but memory cannot be changed fast enough [1]
  - Operate MSBs of $A$ first ($A_K x$), LSBs last ($A_1 x$)
  - Result of $A_k x$ can be negated for 2’s complement arithmetic

---

PPAC does MVPs in different number formats

$L$-bit number formats supported by PPAC

<table>
<thead>
<tr>
<th>Name</th>
<th>uint</th>
<th>int</th>
<th>oddint</th>
</tr>
</thead>
<tbody>
<tr>
<td>LO level</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>HI level</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Signed?</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- Min. value: 0, $-2^{L-1}$, $-2^L + 1$
- Max. value: $2^L - 1$, $2^{L-1} - 1$, $2^L - 1$

E.g., $L = 2$: {0,1,2,3} {-2,-1,0,1} {-3,-1,1,3}

\[ x = \sum_{\ell=1}^{L} 2^{\ell-1}x_\ell \]

- Neural network inference using low-precision int/uint numbers
- Hadamard transform using a 1-bit oddint matrix and a multi-bit int vector
- Signal processing, imaging and communications applications

For a matrix with $K$-bit entries and a vector with $L$-bit entries, an MVP is computed in $KL$ cycles.
MVPs in GF(2)

- Galois Field of Two Elements, GF(2):
  - $\times$: AND
  - $+_{\text{mod-2}}$:
- Mod-2 $+$ is the LSB of $y_m$

Cannot implement reliably in analog

GF(2) arithmetic
- Single-cycle operation
- Forward-error correction
- Decoding
  - LDPC codes
  - Polar codes
- Cryptography
  - Secure hashing
  - AES S-box computation
Computing Boolean functions with PPAC

- Sum of min-terms
  - Column → Boolean variable (complement in different column)
  - Row → Min-term
  - Bank → Logic function
- $A_{m,n} = 1$ if $m$th minterm contains $n$th variable
- PPAC can compute Boolean functions with two AND/OR/MAJ levels

![Logic function computation]
- Single-cycle operation
- Look-up table
- Programmable Logic Array (PLA)

$$Y = A \bullet B + C$$
Implementing PPAC

<table>
<thead>
<tr>
<th></th>
<th>$M$</th>
<th>$N$</th>
<th>$L$</th>
<th>$K$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Words $M$</td>
<td>16</td>
<td>16</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Word-length $N$</td>
<td>16</td>
<td>256</td>
<td>16</td>
<td>256</td>
</tr>
<tr>
<td>Area [$\mu m^2$]</td>
<td>14 161</td>
<td>72 590</td>
<td>185 283</td>
<td>783 240</td>
</tr>
<tr>
<td>Density [%]</td>
<td>75.77</td>
<td>70.45</td>
<td>75.52</td>
<td>72.13</td>
</tr>
<tr>
<td>Cell area [kGE]</td>
<td>17</td>
<td>81</td>
<td>213</td>
<td>897</td>
</tr>
<tr>
<td>Max. clock freq. [GHz]</td>
<td>1.116</td>
<td>0.979</td>
<td>0.824</td>
<td>0.703</td>
</tr>
<tr>
<td>Power (@0.9V,25°C) [mW]</td>
<td>6.64</td>
<td>45.60</td>
<td>78.65</td>
<td>381.43</td>
</tr>
<tr>
<td>Peak throughput [TOP/s]</td>
<td>0.55</td>
<td>8.01</td>
<td>6.54</td>
<td>91.99</td>
</tr>
<tr>
<td>Energy-eff. [fJ/OP]</td>
<td>12.00</td>
<td>5.69</td>
<td>12.03</td>
<td>4.15</td>
</tr>
</tbody>
</table>

- All-digital CMOS design
  - RTL design: Parameterizable and portable
  - Automated CAD tool implementation
  - Robust to process variations and easy to test

One OP is either a 1-bit multiplication or a 1-bit addition
Comparison with existing accelerators

<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>
</tr>
</thead>
<tbody>
<tr>
<td>PPAC</td>
<td>yes</td>
<td>no</td>
<td>layout</td>
<td>28</td>
<td>0.9</td>
<td>0.78</td>
<td>91 994</td>
<td>184</td>
<td>91 994</td>
<td>184</td>
</tr>
<tr>
<td>CIMA [1]</td>
<td>yes</td>
<td>yes</td>
<td>silicon</td>
<td>65</td>
<td>1.2</td>
<td>8.56</td>
<td>4 720</td>
<td>152</td>
<td>10 957</td>
<td>1 456</td>
</tr>
<tr>
<td>Bankman et al. [2]</td>
<td>no</td>
<td>yes</td>
<td>silicon</td>
<td>28</td>
<td>0.8</td>
<td>5.95</td>
<td>-</td>
<td>532</td>
<td>-</td>
<td>420</td>
</tr>
<tr>
<td>Brein [3]</td>
<td>yes</td>
<td>no</td>
<td>silicon</td>
<td>65</td>
<td>1.0</td>
<td>3.9</td>
<td>1.38</td>
<td>2.3</td>
<td>3.2</td>
<td>15</td>
</tr>
<tr>
<td>UNPU [4]</td>
<td>no</td>
<td>no</td>
<td>silicon</td>
<td>65</td>
<td>1.1</td>
<td>16</td>
<td>7 372</td>
<td>46.7</td>
<td>17 114</td>
<td>376</td>
</tr>
<tr>
<td>XNE [5]</td>
<td>no</td>
<td>no</td>
<td>layout</td>
<td>22</td>
<td>0.8</td>
<td>0.016</td>
<td>108</td>
<td>112</td>
<td>84.7</td>
<td>54.6</td>
</tr>
</tbody>
</table>

- Energy-efficiency 7.9x and 2.3x lower than mixed-signal designs in [1] and [2]
- PPAC can compute a 4-bit 256-dim. inner product in **16 clock cycles**; Neural Cache [6] requires at least **98 clock cycles**

PPAC: A versatile in-memory accelerator

- Massively-parallel engine for matrix-vector-product-like operations
  - Hamming similarity
  - Matrix-vector product
    - 1-bit or multi-bit
    - $\{\text{LO,HI}\} \rightarrow \{-1,+1\}$ or $\{\text{LO,HI}\} \rightarrow \{0,+1\}$
    - GF(2)
  - Boolean function computation
- Memory and data-path seamlessly integrated
  - High throughput, and high energy-efficiency:
    A 28nm CMOS 256x256 PPAC achieves 92 TOP/s @ 184 TOP/s/W
- Competitive performance with high flexibility

For more information, please visit vip.ece.cornell.edu
Exploiting Hamming similarities

<table>
<thead>
<tr>
<th>a</th>
<th>x</th>
<th>a XNOR x</th>
</tr>
</thead>
<tbody>
<tr>
<td>HI</td>
<td>HI</td>
<td>HI</td>
</tr>
<tr>
<td>HI</td>
<td>LO</td>
<td>LO</td>
</tr>
<tr>
<td>LO</td>
<td>HI</td>
<td>LO</td>
</tr>
<tr>
<td>LO</td>
<td>LO</td>
<td>HI</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>x</th>
<th>a • x</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>+1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>+1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
<td>+1</td>
</tr>
</tbody>
</table>

\[
\langle a_m, x \rangle = \sum_{n=1}^{N} a_{m,n} x_n \\
\langle a_m, x \rangle = -N + 2\overline{h}(a_m, x)
\]

Stores binary-valued matrix entries \( a_{m,n} \)

Stores binary-valued vector element \( x_n \)

\[ a \cdot x = \begin{bmatrix} \langle a_1, x \rangle \\ \vdots \\ \langle a_M, x \rangle \end{bmatrix} = A \cdot x \]

1-bit matrix-vector product (MVP)

- Single-cycle operation
- Binarized neural networks [1]
- mmWave/THz equalization

Operating MVPs in different number formats

- Key for building standard (multi-bit) unsigned and 2’s complement signed arithmetic, and other operations!

<table>
<thead>
<tr>
<th>a</th>
<th>x</th>
<th>a AND x</th>
</tr>
</thead>
<tbody>
<tr>
<td>HI</td>
<td>HI</td>
<td>HI</td>
</tr>
<tr>
<td>HI</td>
<td>LO</td>
<td>LO</td>
</tr>
<tr>
<td>LO</td>
<td>HI</td>
<td>LO</td>
</tr>
<tr>
<td>LO</td>
<td>LO</td>
<td>LO</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>x</th>
<th>a (\cdot) x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\(\text{addr}\), \(\text{wrEn}\), \(\text{clk}\), \(x_1\), \(\ldots\), \(x_N\), \(y_m\), \(\delta_m\), \(\text{row ALU}\)
Operating across different number formats

- So far, we have operated matrix-vector-products where both matrix and vector have \( \{ \pm 1 \} \) or \( \{0,1\} \) entries
- Operations between different number representations are possible too
  - For example, if the matrix has \( \{ \pm 1 \} \) entries, but the vector has \( \{0,1\} \) entries, then:
    \[
    \langle \mathbf{a}_m, \mathbf{x} \rangle = \bar{h}(\mathbf{a}_m, \mathbf{x}) + \bar{h}(\mathbf{a}_m, 1) - N
    \]
  - \( \bar{h}(\mathbf{a}_m, 1) \) is stored in the row ALU and only calculated if the matrix \( \mathbf{A} \) changes
Operating across different number formats

- So far, we have operated matrix-vector-products where both matrix and vector have \{\pm 1\} or \{0,1\} entries
- Operations between different number representations are possible too
  - For example, if the matrix has \{\pm 1\} entries, but the vector has \{0,1\} entries, then:
    \[
    \langle a_m, x \rangle = h(a_m, x) + h(a_m, 1) - N
    \]
  - \(h(a_m, 1)\) is stored in the row ALU and only calculated if the matrix \(A\) changes...
Computing Boolean functions with PPAC

- Sum of min-terms
  - Column → Boolean variable (complement in different column)
  - Row → Min-term
  - Bank → Logic function
- \( A_{m,n} = 1 \) if \( m \)th minterm contains \( n \)th variable
- PPAC can compute Boolean functions with two AND/OR/MAJ levels

\[
A_{m,n} = 1 \text{ if } m \text{th minterm contains } n \text{th variable}
\]

- Logic function computation
  - Single-cycle operation
  - Look-up table
  - Programmable Logic Array (PLA)

- PPAC can compute Boolean functions with two AND/OR/MAJ levels