DSPIMM: A Fully Digital SParse In-Memory Matrix Vector Multiplier for Communication Applications

Amitesh Sridharan*, Fan Zhang*, Yang Sui†, Bo Yuan†, Deliang Fan*
*School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ, USA
†Department of Electrical and Computer Engineering, Rutgers University
Email: asridh25@asu.edu, fzhang95@asu.edu, yang.sui@rutgers.edu, bo.yuan@soe.rutgers.edu, dfan@asu.edu

***Abstract*—Channel decoders are key computing modules in wired/wireless communication systems. Recently neural network (NN)-based decoders have shown their promising error-correcting performance because of their end-to-end learning capability. However, compared with the traditional approaches, the emerging neural belief propagation (NBP) solution suffers higher storage and computational complexity, limiting its hardware performance. To address this challenge and develop a channel decoder that can achieve high decoding performance and hardware performance simultaneously, in this paper we take a first step towards exploring SRAM-based in-memory computing for efficient NBP channel decoding. We first analyze the unique sparsity pattern in the NBP processing, and then propose an efficient and fully Digital Sparse In-Memory Matrix vector Multiplier (DSPIMM) computing platform. Extensive experiments demonstrate that our proposed DSPIMM achieves significantly higher energy efficiency and throughput than the state-of-the-art counterparts.**

***Index Terms*—Sparsity, In-Memory-Computing, SRAM, MAC, Neural Decoder.**

I. Introduction

Thanks to their powerful error-correcting capabilities, modern channel codes, such as low-density parity check (LDPC) [1], polar [2], and Turbo [3] codes, have been widely used in numerous real-world wired and wireless communication systems, including but not limited to 5G, Wi-Fi, StarLink, Ethernet, etc. In general, given a fixed channel code, its error-correcting performance is mainly determined by the *decoder*. Recently, neural belief propagation (NBP), as a neural network (NN)-based approach, has shown very promising decoding performance across different types of channel codes [4]–[7]. By unfolding the original iterative belief propagation procedure to form a sparse feedforward neural network, NBP makes the key scaling parameters, which were previously set in a heuristic way, can now be directly learned from the data, significantly improving the error-correcting capability of channel codes.

Hardware Challenge of NBP Decoder. Despite its attractive algorithmic advantage, NBP decoder is facing a severe challenge in hardware performance. The integration of NN into decoding process, though improving error-correcting performance, brings much higher storage and computation overhead. Because channel decoders are typically deployed in the real-time and/or low-power communication systems, the significantly increasing complexity, if not properly addressed, may hinder the widespread adoption of this promising technique.

IMC-NBP: A Double-Win Solution. Fortunately, we discover that the emerging hardware challenge for NBP decoder can be effectively addressed via in-memory computing (IMC), a technique that has been well-studied to develop low-power general NN hardware [8]–[11]. Considering NBP is essentially a type of specialized sparse feedforward neural network model, applying IMC to its hardware design, is naturally a very promising strategy towards achieving high hardware performance while preserving high decoding performance.

Which Type of IMC for NBP? Motivated by such promising benefits, in this paper, we propose to develop energy-efficient high-performance in-memory computing-based neural BP decoder. Since there exist numerous types of IMC techniques in the market, e.g., SRAM, RRAM, MRAM, etc., the very first design knob we need to consider is the most suitable IMC approach for NBP decoder. Our in-depth analysis concludes that digital in-SRAM computing is the best candidate for building the desired NBP decoder. This is because compared to AI applications, wired/wireless communication have very stringent requirement on error rate (at least 10^{-4} and above) and data rate, calling for a noiseless compute environment and high throughput. To that end, digital in-SRAM computing is an ideal candidate for NBP hardware because of its accurate computation, low read/write latency, and high flexibility.

Questions to be Answered. Considering NBP decoder is a sparse neural network with a unique sparsity pattern and activation function, a customized solution, instead of the existing general in-SRAM hardware, is desired to fully deliver its algorithmic promise. More specifically, several technical questions need to be answered. For instance, how should we properly leverage the unique structured and unstructured sparse patterns, which currently cannot be supported by the existing digital SRAM-based IMC implementations? What is the efficient way to map new computing flow and operation on the IMC circuits?

Technical Preview and Contributions. In this paper, we perform systematic investigations to answer these questions, and then develop the corresponding hardware solutions. The main contributions are summarized as follows:

1) We, for the first time, design and develop an end-to-end, energy-efficient high-speed SRAM-based in-memory computing system for neural BP channel decoding, namely *DSPIMM*.
2) We propose an efficient and digital bit-serial in-memory matrix-vector multiplication (MVM) module using a novel 8T compute SRAM bit-cell circuit design, fully supporting the unique sparsity pattern in NBP decoding.
3) We propose a greedy weight compression and localization

This work is supported in part by the National Science Foundation under Grant No.2003749 and No. 2144751

979-8-3503-2348-1/23/$31.00 ©2023 IEEE

(GWCL) algorithm, which properly leverages the structured and unstructured sparsity pattern, to realize efficient data mapping and sparse computing.

4) We conduct extensive experiments showing the great energy efficiency and power improvement of our DSPIMM platform. We also systematically benchmark with other state-of-the-art counterparts.

II. Neural BP Algorithm

According to coding theory, a (N, K) channel code is uniquely defined by a $(N-K)$-by-N binary parity check matrix ($\boldsymbol{H}$), which can also be interpreted as a bipartite graph consisting of N variable nodes and $(N-K)$ check nodes. Suppose we use v to denote the v-th variable node in the node set V and c to denote the c-th check node in the node set C, respectively. Also, we use $E = \{e_{(c,v)} = (c, v) : H(c, v) = 1, v \in V, c \in C\}$ to denote the set of edges connecting the two types of nodes. Here the $e_{(c,v)}$ connecting the c-th check node and the v-th variable node corresponds to one 1-valued entry ("$H(c, v) = 1$") of $\boldsymbol{H}$.

The key idea of NBP decoding [12] is to perform message update in the unfolded bipartite graph. As illustrated in Fig. 1, the neurons denoted as orange and green circles represent $u_{c\to v}$ and $u_{v\to c}$, which are the messages (i.e., "belief") transmitted from the v-th variable node to the c-th check node and from the v-th variable node to the c-th check node at the t-th iteration through edges E, respectively. Different from traditional belief propagation, NBP treats the connections between $u_{c\to v}$ and $u_{v\to c}$ as trainable weights instead of the pre-set heuristics. Next, we summarize the overall dataflow and compute steps of Neural BP decoding. Initially, an NBP decoder receives the log-likelihood ratios (LLRs) $\boldsymbol{l} \in \mathbb{R}^n$ of the received codeword $\boldsymbol{r}$ as:

$$l_v = \log \frac{Pr(x_v = 1|r_v)}{Pr(x_v = 0|r_v)}. \tag{1}$$

Then the variable nodes and check nodes iteratively update the LLR messages during the entire Neural BP decoding process. The specific update principle of the LLR message in each iteration go through the following five steps:

Step 1: Structured Sparse Matrix-Vector Multiplication (SSP-MVM). At the t-th iteration, $u^t_{v\to c}$ can be calculated as:

$$\boldsymbol{u^t_{v\to c}} = \boldsymbol{W_1 l_v} + \boldsymbol{W_2 u^{t-1}_{c\to v}}, \tag{2}$$

where we define the first term as $\boldsymbol{k^t_{v\to c}} = \boldsymbol{W_1 l_v}$ and second term $\boldsymbol{q^t_{v\to c}} = \boldsymbol{W_2 u^{t-1}_{c\to v}}$. For the first term $\boldsymbol{k^t_{v\to c}} = \boldsymbol{W_1 l_v}$, the matrix format can be formulated as:

$$\begin{bmatrix} (k^t_{v\to c})_1 \\ \vdots \\ (k^t_{v\to c})_D \end{bmatrix} = \begin{bmatrix} 0,0,0,w,\dots \\ 0,0,w,0,\dots \\ \vdots \end{bmatrix} \begin{bmatrix} (l_v)_1 \\ \vdots \\ (l_v)_N \end{bmatrix} \tag{3}$$

with input vector $\boldsymbol{l_v} \in \mathbb{R}^N$ and weight matrix $\boldsymbol{W}_1 \in \mathbb{R}^{D\times N}$ that has one non-zero entry in each row (corresponding to v), denoted by golden connections between l_v (pink circles) and $u_{v\to c}$ (green circles).

Step 2: Unstructured Sparse Matrix Vector Multiplication (USP-MVM) and Accumulation. It requires another matrix multiplication followed by an addition with the results from the previous step. For the second term $\boldsymbol{q^t_{v\to c}} = \boldsymbol{W_2 u^{t-1}_{c\to v}}$, the matrix format can be formulated as:

$$\begin{bmatrix} (q^t_{v\to c})_1 \\ \vdots \\ (q^t_{v\to c})_D \end{bmatrix} = \begin{bmatrix} 0,0,0,0,\dots \\ 0,0,0,w,\dots \\ \vdots \end{bmatrix} \begin{bmatrix} (u^{t-1}_{c\to v})_1 \\ \vdots \\ (u^{t-1}_{c\to v})_D \end{bmatrix} \tag{4}$$

with input vector $\boldsymbol{u^{t-1}_{c\to v}} \in \mathbb{R}^D$ and weight matrix $\boldsymbol{W}_2 \in \mathbb{R}^{D\times D}$ that has non-zero entries at the positions corresponding to $N(v)\backslash c$, where $N(v) = \{c \in C : e_{(c,v)} \in E\}$ and $M(c) = \{v \in V : e_{(c,v)} \in E\}$ are the neighbors of variable node v and check node c, respectively. $\boldsymbol{W}_2$ is denoted by the red connections between $u_{c\to v}$ (orange circles) and $u_{v\to c}$ (green circles). The $\boldsymbol{u^t_{v\to c}}$ is calculated as the summation of $\boldsymbol{k}$ and $\boldsymbol{q}$ from Eq. 3 and 4 as:

$$\begin{bmatrix} (u^t_{v\to c})_1 \\ \vdots \\ (u^t_{v\to c})_D \end{bmatrix} = \begin{bmatrix} (k^t_{v\to c})_1 \\ \vdots \\ (k^t_{v\to c})_D \end{bmatrix} + \begin{bmatrix} (q^t_{v\to c})_1 \\ \vdots \\ (q^t_{v\to c})_D \end{bmatrix}. \tag{5}$$

Step 3&4: Min-Sum and Dot-Product Computation. The $u^t_{c\to v}$ is calculated by min-sum operation [7] as follows:

$$u^t_{c\to v} = w_{3_{c\to v}} \min_{v'\in M(c)\backslash v} |u^t_{v'\to c}| \prod_{v'\in M(c)\backslash v} \text{sign}(u^t_{v'\to c}), \tag{6}$$

where $\boldsymbol{w_3} \in \mathbb{R}^D$ is learnable. This step is denoted by connections between $u_{v\to c}$ (green circles) and $u_{c\to v}$ (orange circles) including blue connections (min-sum operations) and black connections (weight dot products).

Step 5: S_V Calculation. Then, the final soft output after the t-th iteration can be calculated as:

$$\boldsymbol{s^t_v} = \boldsymbol{l_v} + \boldsymbol{W}_4 \times \boldsymbol{u^t_{c'\to v}} \tag{7}$$

$$\begin{bmatrix} (s^t_v)_1 \\ \vdots \\ (s^t_v)_N \end{bmatrix} = \begin{bmatrix} (l_v)_1 \\ \vdots \\ (l_v)_N \end{bmatrix} + \begin{bmatrix} 0,0,0,0,\dots \\ 0,w,0,0,\dots \\ \vdots \end{bmatrix} \begin{bmatrix} (u^t_{v'\to c})_1 \\ \vdots \\ (u^t_{v'\to c})_D \end{bmatrix}, \tag{8}$$

where $\boldsymbol{W}_4 \in \mathbb{R}^{N\times D}$ with non-zero elements correspond to $N(v)$, denoted by golden connections between $u_{c\to v}$ (orange circles) and s_v (yellow circles).

Remark. As shown in the neural BP computation flow (Fig. 1), the majority of learn-able weights $\boldsymbol{W_1, W_2, W_4}$ hold **high sparsity** with a special pattern under the matrix format. For instance, because $\boldsymbol{W}_1 \in \mathbb{R}^{D\times N}$ has only one non-zero weight in each row, the sparsity of $\boldsymbol{W_1}$ is $\frac{N-1}{N}\%$, which can easily achieve 90% when $N \geq 10$ and 99% when $N \geq 100$.

III. Proposed *DSPIMM* Platform

A. Architecture and Data Flow

Fig. 2(A) demonstrates the overall architecture of our DSPIMM for NBP. It supports all the required five algorithm steps as shown in the corresponding circuit model in Fig. 2(A1), (A2), (A3), and (A4). Note that, steps 1 and 2 are mainly MVMs and are implemented using our IMC modules, where the corresponding circuits are shown in Fig. 2(B) and (C).

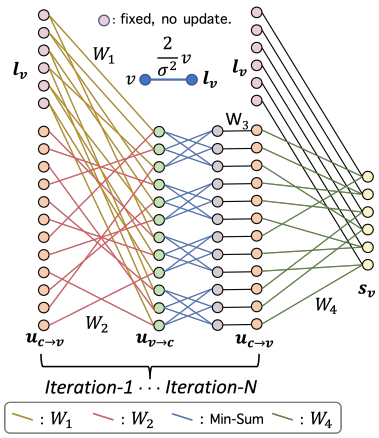


Fig. 1. Example of neural BP decoding procedure.

1) W_1, Structured Sparse Matrix Memory: denoted by Fig. 2(A1). This corresponds to the NBP step 1. The IMC array with golden halos denotes the memory of W_1 weight matrices (i.e., golden connections in Fig. 1). Each IMC array size is 256x256 which translates to 8KB per block. The Control, I/O logic, and Input Buffers are shared among 4 IMC sub-arrays. The control and I/O logic help in scattering the stored weights in the Input Buffer (compiler-generated SRAMs) to the IMC sub-Arrays. They also store the Inputs, i.e., l_v vectors, and scatter them during compute mode. Then, the compute/partial product outputs are collected in the output/partial product buffers (Using SRAM compiler generated Register Files).

2) W_2, Unstructured Sparse (USP) Matrix Memory: Fig. 2(A2) shows the USP-Matrix Memory. The memory organization is similar to SSP Matrix memory, with 128x256 - 4KB IMC arrays. They carry W_2 (red halo) as well as W_4 (green halo) since both follow a similar computing pattern. This USP Matrix Memory is responsible for the sparse MVMs in Step 2 and Step 5 of the NBP algorithm.

3) Global Addition: After SSP and USP-MVM, the stored partial products in the output buffers are streamed into parallel global adders to perform the addition operations on the two MVM outputs. This completes the compute of the green dots in Fig. 1 denoting the end of Step 2 of NBP algorithm.

4) MinSum Compute and dot products (Fig. 2 A3&A4): These modules perform steps 3 and 4. The $Minsum$ and dot-product instructions do not have common operands, hence are not suitable for IMC. Thus, we leverage the digital comparators in parallel to compute the $MinSum$. The $Minsum$ outs along with the sign bits from the output buffers of USP-MM and W_3 weights are sent to the dot-product engine (DPE) to compute Step 4. The W_3 weight matrix is usually large and uncompressed. So a Register file is used to buffer a portion of W_3 weights to be streamed onto the DPE. when finished, the remaining data will be fetched from off-chip.

5) S_v Calculation: Step 5 needs to be performed only once after several iterations of Steps 1 through 4. It is performed by loading l_v onto the output buffers of the SSP-MVM memory and using the computational sub-array of the USM-MVM (green halos) to perform MVM between W_4 and the u_vc of the previous iteration. Now, the global adders are used to sum the output buffers of USP Matrix Memory and SSP Matrix Memory containing the MVM outs and l_v respectively.

B. 6+2T (8T) SRAM bitcell design for in-memory computing

To implement in-memory computing (IMC), specifically for matrix multiplication in this work, we propose a 6+2T SRAM compute bit-cell (CBC) as shown in Fig. 3(a) to implement 1 bit partial product and then the peripheral shift & accumulator circuits implement the rest for multi-bit matrix multiplication. For memory function, a traditional 6T SRAM bitcell is used. For compute, the bit-cell is augmented with two additional transistors - T1 and T2. Together, they perform the 'AND' function or a 1'b dot-product within the memory cell, between operand-1 (weight bit - w/wb) and operand-2 (external input bit - IWL). The weight bit (w and its complementary - wb) is stored in the cross-coupled inverters of bitcell which are connected with the gate terminals of T2 and T1, respectively. The other operand-2 is from the input world line (i.e., IWL), which goes to the source terminal of T2. Note that, as the name suggests, IWL is broadcasted to the entire worldline, providing inputs to all 8T CBCs in that row that store multi-bits of the weight parameters. The last signal is a VSS/GND connecting to the source terminal of T1. Finally, the AND or 1'b Dot product out (DPO) is obtained from the common drain terminals of T1 and T2, where the truth table is given in Fig. 3(b).

C. Bit-Serial Matrix Vector Multiplication (MVM) in-memory

For multi-bit MVM, the multi-bit weight operand is stored in the memory and the other input operand is streamed through the IWLs. The matrix operand stored in the memory is transposed before storage, this will put a single row of the matrix elements into a single column of the memory array. It is done to (1) perform dot products between all elements in one column of the second operand (through IWL) and all elements in a single row of the first operand. (2) Since IWLs are shared amongst rows, the same second operand column can be used to multiply with all the first operand rows, performing a parallel NXM (Op.1) * $MX1$ (Op.2) *vector dot-product*. Then, the accumulation of dot-products of every column in memory (corresponding to the row of the first operand) is implemented using adder trees to complete MVM. Due to bit-serial design, the IWLs can stream only 1 bit at a time, a shift accumulation is designed to respect the bit-position of the multi-bit operands streamed through the IWLs, where data flow is shown in Fig. 2(D).

D. Structured Sparse Matrix Vector Multiplication (SSP-MVM)

MVM-in-memory is dense and intensive, meaning all bit-cells are active and used for compute. But for NBP, the W_1 is extremely sparse with the special pattern as described in section II. So to leverage such property, we develop a hardware-friendly compression/encoding algorithm that localizes the weights and eliminates all zeros from being stored, thereby ignored for compute.

1) Greedy weight compression and localization (GWCL) algorithm for SSP-MVM: Fig.4 shows an example. It parses through the weight matrix and only stores the non-zero weights in memory. But, such an operation scatters the weights across

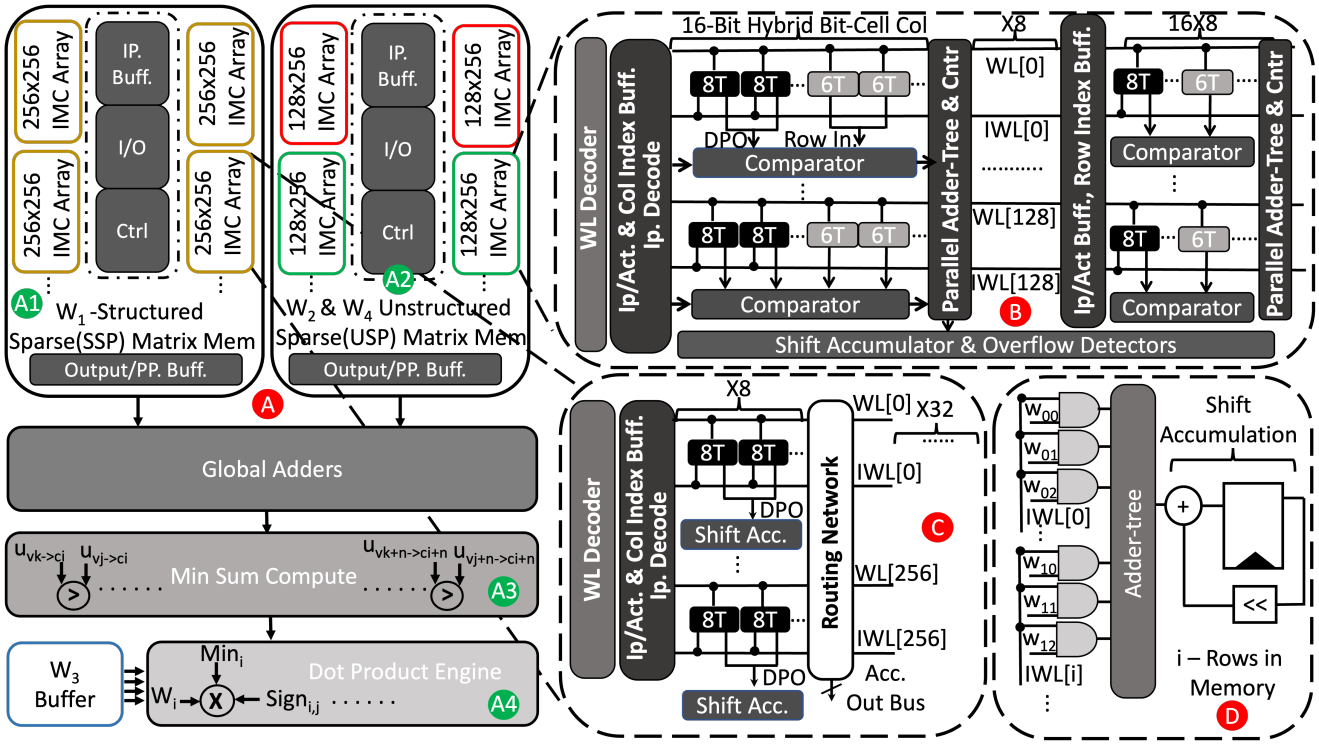
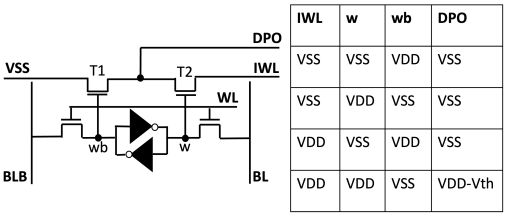


Fig. 2. (A) DSPIMM architecture (B) USP Weight In-Memory Compute (C) Structured Sparse Weight In-Memory Compute (D) Data flow of Bit-Serial MVM.



IWL	w	wb	DPO
VSS	VSS	VDD	VSS
VSS	VDD	VSS	VSS
VDD	VSS	VDD	VSS
VDD	VDD	VSS	VDD-Vth

Fig. 3. (a) 6+2T(8T) Compute Bit-Cell (b) Truth table of 2T AND/Dot-Product

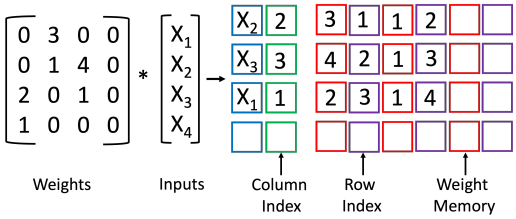


Fig. 4. EGWCL Algorithm example

memory, breaking the structure of matrix multiplication. To solve this, during the skip of zero weights, our circuit needs to be aware of (1) which input needs to be multiplied with which weight, and (2) which DPOs (post multiplication) need to be added together.

For (1), whenever the algorithm faces a non-zero weight, it stores not only the weight value but also the column index of this weight next to the input buffers. Through this, when a new input is streamed in, the column indices can dictate which row of input buffer should be streamed onto the IWLs for compute. Since the memory leverages the sharing of IWLs over several bit-cells to achieve high parallelism, during compression, all weights belonging to the same column are stored in a single row. For (2), the sparsity of W_1 matrix is structured and only one element per row is a non-zero weight. Since all elements in a row are added up for MVM(after dot product), the resulting accumulation in this case will be the dot-product of the input and weight alone. Therefore, due to the nature of this sparsity pattern, no accumulation is necessary. Hence, no circuitry is required for accumulation or decoding the accumulation of the scattered weights. In summary, the GWCL algorithm works in two stages:

Stage 1: Ignores zero-weights and greedily stores non-zero weights, it also stores the corresponding column index alongside the input buffer.

Stage 2: If encountering a weight belonging to a column previously stored, it stores the weight in the same row as that of the previously encountered column index to enable the parallel multiplication for a shared IWL.

2) SSP-MVM In-Memory Compute Circuit and Architecture (Fig. 2(C)): The one-time sparse weight compression discussed above is done off-line and mapped to our IMC arrays. During inference, a new set of inputs is fetched every iteration, so a decoding circuitry is designed to map the newly fetched inputs using column indices. It consists of a set of comparators that compare the indices of the new inputs against the stored column indices (next to the input buffers) and map the inputs to the input buffers of the respective rows. Then, these inputs will be streamed onto the IWLs in a bit-serial fashion for performing partial product. Since no accumulation is required, the DPOs are directly sent to the shift accumulators which completes the 8b8b dot-product. In summary, the implemented SSP-MVM IMC architecture has 32 8-bit columns, each column has 256 rows and each row of the 8-bit column consists of a shift accumulator and each 8-bit column have a routing network to route all the accumulated outputs.

E. Unstructured Sparse Matrix Vector Mult. (USP-MVM)

The above SSP-MVM has a fixed sparse pattern with one-hot element in a row, enabling us to skip accumulation. But, the unstructured sparse W_2 and W_4 matrices do not follow this pattern, with multiple non-zero elements in a row.

1) Enhanced Greedy Algorithm for USP-MVM: To adapt our hardware for USP-MVM, we enhance our GWCL algorithm to also support USP weights. The main difference here is that weights need to be accumulated and are scattered all over the memory array. To complete MVM, it needs a way to identify which DPOs (post input stream-in) require accumulation, hence an additional operation is performed alongside Stage 1, which is, the row indices of the weights are also stored alongside the weight memory. The reason is that, in an MVM between matrix A and B, the column of operand B is multiplied by the row of operand A, after which the dot-products accumulate together. Mapping such a process to our IMC memory array means the accumulation only happens to the dot-products of weights in the same row. So, by storing the weight indices during accumulation, we only need to accumulate the dot-products resulting from weights having the same row-indices. In summary, the Enhanced GWCL (EGWCL) is:

Stage 1: Store column indices of all non-zero weights next to the input buffers; store the row-indices next to the non-zero weights, ignoring the zero weights.

Stage 2: If the newly encountered weight has a column index that is previously stored next to the input buffers, it stores the weight in the subsequent column of the same memory row corresponding to the column index.

2) USP-MVM IMC Circuit and Architecture (Fig. 2(b)): As per our EGWCL algorithm, both the weights and row indices are stored in the weight memory. Since only the weights are used for compute and the row-indices are used for decoding the compressed weights for accumulation, a traditional 6T SRAM bitcell is used for storing the row indices and 8T CBC is used for storing weights. For a given memory size $m \times n$, $log(m)$ bits are required to represent the row indices. We use 128-bit rows memory with 256-bit columns. The 8-bit CBC and the 8-bit 6T SRAM together form a 16-Bit Hybrid Bit-Cell Column. So, we have 16 columns of the 16-Bit Hybrid Bit-Cells in total. As for the inputs, the column indices are stored alongside the input buffers in flops. When a new input is fetched, the comparator-based decoder is designed to parse through the column indices of the input matrix and store the corresponding inputs onto the input buffers. Then, they are streamed onto IWLs for dot product computing. The CBCs in every 16-bit column will hold 8-bit DPOs that will be accumulated next. The EGWCL algorithm scatters the weights across the memory, so multiple rows of the weight matrix can be present in a single column of the memory array. Circuitry is required to (1) identify which rows are present and (2) parse through and accumulate all the weights in the column. For (1), we attach comparators to every word, which enables reading of the weight indices directly from 6T bitcells. These comparators take in the row index as well as the 8'DPO and compare the row-index against a generated index. If it compares it outputs the 8'DPO, else it outputs a 0. For (2), we need to identify all the row-indices present in a single column. So a Mod-Counter is placed in every column. Every counter is given the first and last index present in the corresponding column to parse through all the row indices of that column. The output of these counters is sent to the comparators, providing indices to compare against and identify the weights needs to be accumulated.

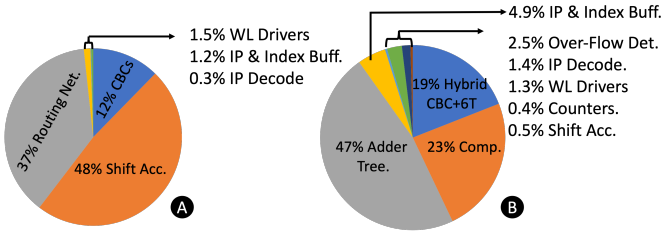


Fig. 5. Area Breakdown of (a) SSP Matrix Mem (b) USP Matrix Mem

3) Overflows: The row indices of the accumulated DPOs are also stored alongside the flops in the shift accumulators (Propagated from counters in the adder tree). This is to tackle an inherent drawback called overflow that arises due to the nature of the EGWCL algorithm. When the weights are compressed, there is a chance that weights from a single row (W_2 matrix) can span multiple columns (memory). In this case, these values need to be accumulated. So, to keep track of which row the partial product is being computed, the row index is also stored in flops for every column of the memory array. After the counters parse through all the row-indices, the overflow detectors present alongside the shift accumulators accumulate all the weights that belong to the same row. This completes the USP-MVM.

IV. Evaluation and Results

A. Experiment Setup

Cadence Spectre is used for all custom circuitry, designed using TSMC 28nm to verify functionality and to check for latency and power consumption. The area evaluation of custom circuitry is done by making layouts in Cadence Virtuoso. For an SSP-MVM Memory, we simulate a 128x256 memory array. For USP-MVM, we simulate a 256x256 array. For all digital components, we use Synopsis Design Compiler to synthesize the gate netlist. For all reported code lengths, VCD files are generated using SDF annotated post-synthesized RTL simulations. These VCD files are used in Synopsys PrimePower for reporting the power numbers. The Post-synthesized netlist is used in Synopsys PrimeTime to obtain latency numbers.

B. Experiment Results

Since this is the first work to demonstrate NBP in an SRAM-based IMC, we compare our work with other popular LDPC channel decoding hardware implementations in Table III, even using different algorithms. The reason is that LDPC is the most commonly used channel code in real-world applications, and its hardware decoder design receives the most attention as compared to other channel codes. Ours achieves the best energy efficiency and lowest power. We also evaluate the efficacy of our compression algorithm in Table I. It clearly achieves memory savings that match the sparsity ratio.

TABLE I
GWCL ALGORITHM MEMORY BENEFITS(EXCLUDES INDEX MEMORY)

Code Length/ Weight Memory	121		672		1056	
	Uncompressed	GWCL algorithm	Uncompressed	GWCL algorithm	Uncompressed	GWCL algorithm
W_1	73.2KB	0.6KB	1.5MB	2.2KB	3.7MB	3.52KB
W_2	366KB	2.4KB	5MB	5.5KB	12.3MB	8.7KB
W_3	366KB	N/A	4.9MB	N/A	12.3MB	N/A
W_4	73.2KB	0.6KB	1.5MB	2.2KB	3.7MB	3.52KB

TABLE II
COMPARISON WITH STATE-OF-THE-ART SRAM BASED IMC ACCELERATORS.

Reference	**USP-MVM(This Work)**	**SSP-MVM(This Work)**	**ISSCC'21 [13]**	**ESSCIRC'19 [14]**	**ISSCC'22 [10]**
Technology	28nm	28nm	22nm	65nm	28nm
Array Size	2KB(Weights)+2KB(Indices)	8KB	8KB	0.8KB	2KB
Bit-Cell overhead	1T per bit-cell(8'b w 2T, 8'b w/o 2T)	2T Per Bit-Cell	4T per bitcell	XOR+MUX+FA/ Bitcell	2T Per Bit Cell
Sparsity Level	50%	>99%(Fixed)	50%	50%	50%
Macro Size	0.187mm^2	0.7673mm^2	0.202mm^2	0.242mm^2	0.049mm^2
Performance(GOPs)(8b8b)	**786.18**	**1927.3**	917	N/A	2035(4b1b)
Efficiency(TOPs/W)(8b8b)	12.92	**29.56**	24.7	2.06(16b)	154(4b1b)
Latency(8b8b)	**10.42ns(9 Cycles)**	**8.501ns(8 Cycles)**	18ns	NA	∼20ns
Implementation	Synthesis	Synthesis	Post-Silicon	Post-Silicon	Post-Silicon

*USP-MVM throughput - 128(No of weights in a column)*2(Multiply+Accumulate)*2(50% Sparsity)*16(No of cols)/10.42ns=786GOP/s.
**SSP-MVM throughput - 256(No of weights in a column)*2(Multiply+Accumulate)*32(No of columns)/8.5ns=1927.3GOP/s

TABLE III
COMPARISON WITH PRIOR LDPC IMPLEMENTATIONS

	This Work	**TCAS'21 [15]**	**VLSI'18 [16]**
Code Length	1056	1027	2048
Core Area	1.32mm^2	2.24mm^2	16.2mm^2
Frequency	783Mhz	1000Mhz	862Mhz
Throughput	224Gb/s @4it	833Gb/s@4it	588Gb/s@5it
Area Efficiency	169.7Gb/s/mm^2	371.9Gb/s/mm^2	36.3Gb/s/mm^2
Energy Efficiency	**1374.2Gb/s/W**	109.605Gb/s/W	44.21Gb/s/W
Latency	57.465ns@4it	38ns@4it	69.6@5it
Power	0.163W	7.6W	13.3W
Node	28nm	16nm	28nm
Algorithm	neural-BP	Layered	Finite Alphabet

TABLE IV
POWER BREAKDOWN

SSP-Matrix Mem(256x256)		USP-Matrix Mem(128x256)	
Hardware	Power (mW)	Hardware	Power(mW)
Bit-Cell array(8T)	11.6mW	Bit-cell array(6T+8T)	4.2mW
Shift Accumulator	46.73mW	Comparator	21.3mW
Routing Network	R+C Parasitics	Adder Tree	18.7mW
IP Index+IP Buff.	6.3mW	Ip Index + Ip Buff.	4.22mW
Decoder	0.88mW	Shift Accumulator	6.74mW
Ip Decode	1.3mW	Overflow + Counters	4.63mW
Total	66.81mW	Total	59.79mW

For IMC performance, we draw comparisons with state-of-art IMC designs that have MAC operation as their core in Table II. Compared to existing SRAM-based IMC platforms, the USP-MVM module and the SSP-MVM module achieve the best TOP/s metric. The USP-MVM achieves a throughput almost equal to [13] even though it is only 1/4th in size and our SSP-MVM IMC module can complete an 8b8b MAC one cycle faster from skipping accumulations. The complete area and power breakdown for all sub-modules in USP-MVM and SSP-MVM are shown in Fig. 5 and Table IV. All above detailed hardware evaluation and bench-marking show great performance improvement and hopefully, our design could serve as a benchmark for future neural decoder implementations.

V. CONCLUSION

In this work, we propose a novel SRAM-based IMC circuit and architecture to implement the Neural BP channel decoding algorithm. We utilize the sparse nature of the algorithm by proposing IMC algorithm-hardware co-design to perform sparse MVMs whose operands have fixed (algorithm specific) or generic unstructured sparse patterns. Our proposed IMCs achieve the best throughput out of state-of-the-art IMC MAC implementations and significantly higher energy efficiency than state-of-the-art LDPC decoder hardware.

REFERENCES

[1] R. Gallager. Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28, 1962.

[2] E. Arikan. Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels. *IEEE Transactions on information Theory*, 55(7):3051–3073, 2009.

[3] C. Berrou et al. Near shannon limit error-correcting coding and decoding: Turbo-codes. 1. In *Proceedings of ICC '93 - IEEE International Conference on Communications*, volume 2, pp. 1064–1070 vol.2, 1993.

[4] E. Nachmani et al. Learning to decode linear codes using deep learning. In *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 341–346, 2016.

[5] S. Cammerer et al. Scaling deep learning-based decoding of polar codes via partitioning. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pp. 1–6, 2017.

[6] T. Gruber et al. On deep learning-based channel decoding. In *2017 51st Annual Conference on Information Sciences and Systems (CISS)*, pp. 1–6.

[7] L. Lugosch et al. Neural offset min-sum decoding. In *2017 IEEE International Symposium on Information Theory (ISIT)*, pp. 1361–1365.

[8] A. Biswas et al. Conv-sram: An energy-efficient sram with in-memory dot-product computation for low-power convolutional neural networks. *IEEE JSSC*, 2018.

[9] J. Yue et al. 14.3 a 65nm computing-in-memory-based cnn processor with 2.9-to-35.8 tops/w system energy efficiency using dynamic-sparsity performance-scaling architecture and energy-efficient inter/intra-macro data reuse. In *IEEE ISSCC*, 2020.

[10] D. Wang et al. Dimc: 2219tops/w 2569f2/b digital in-memory computing macro in 28nm based on approximate arithmetic hardware. In *2022 ISSCC*, volume 65, pp. 266–268, 2022.

[11] A. Sridharan et al. A 1.23-ghz 16-kb programmable and generic processing-in-sram accelerator in 65nm. In *ESSCIRC 2022- IEEE 48th European Solid State Circuits Conference (ESSCIRC)*, pp. 153–156, 2022.

[12] E. Nachmani et al. Deep learning methods for improved decoding of linear codes. *IEEE Journal of Selected Topics in Signal Processing*, 12(1):119–131, 2018.

[13] Y.-D. Chih et al. 16.4 an 89tops/w and 16.3tops/mm2 all-digital sram-based full-precision compute-in memory macro in 22nm for machine-learning edge applications. In *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, volume 64, pp. 252–254, 2021.

[14] H. Kim et al. A 1-16b precision reconfigurable digital in-memory computing macro featuring column-mac architecture and bit-serial computation. In *ESSCIRC 2019 - IEEE 45th European Solid State Circuits Conference*.

[15] M. Li et al. High-speed ldpc decoders towards 1 tb/s. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(5):2224–2233, 2021.

[16] R. Ghanaatian et al. A 588-gb/s ldpc decoder based on finite-alphabet message passing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(2):329–340, 2018.