FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision

Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing
Department of Computer Science and Engineering, UC San Diego, La Jolla, CA 92093, USA
{moimani, sgupta, yek048, tajana}@ucsd.edu

ABSTRACT

Processing In-Memory (PIM) has shown a great potential to accelerate inference tasks of Convolutional Neural Network (CNN). However, existing PIM architectures do not support high precision computation, e.g., in floating point precision, which is essential for training accurate CNN models. In addition, most of the existing PIM approaches require analog/mixed-signal circuits, which do not scale, exploiting insufficiently reliable multi-bit Non-Volatile Memory (NVM). In this paper, we propose FloatPIM, a fully-digital scalable PIM architecture that accelerates CNN in both training and testing phases. FloatPIM natively supports floating-point representation, thus enabling accurate CNN training. FloatPIM also enables fast communication between neighboring memory blocks to reduce internal data movement of the PIM architecture. We evaluate the efficiency of FloatPIM on ImageNet dataset using popular large-scale neural networks. Our evaluation shows that FloatPIM supporting floating point precision can achieve up to 5.1% higher classification accuracy as compared to existing PIM architectures with limited fixed-point precision. FloatPIM training is on average 303.2× and 48.6× (4.3× and 15.8×) faster and more energy efficient as compared to GTX 1080 GPU (PipeLayer [1] PIM accelerator). For testing, FloatPIM also provides 324.8× and 297.9× (6.3× and 21.6×) speedup and energy efficiency as compared to GPU (ISAAC [2] PIM accelerator) respectively.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Hardware** → **Emerging technologies**; • **Computing methodologies** → *Machine learning*.

KEYWORDS

Processing in-memory, Non-volatile memory, Deep Neural Network, Machine learning acceleration

ACM Reference Format:
Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision . In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3307650.3322237

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISCA '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322237

1 INTRODUCTION

Artificial neural networks, in particular deep learning [3, 4], have wide range of applications in diverse areas including: object detection [5], self driving car, and translation [6]. Recently, in some specific tasks such as AlphaGo [7] and ImageNet Recognition [8], deep learning algorithms presented human-level performance. Convolutional neural networks (CNN) are the most commonly used deep learning models [5, 9]. Processing CNNs in conventional von Neumann architectures is inefficient as these architectures have separate memory and computing units. The on-chip caches do not have enough capacity to store all data for large size CNNs with hundreds of layers and millions of weights. This consequently creates a large amount of data movement between the processing cores and memory units which significantly slows down the computation.

Processing in-memory (PIM) is a promising solution to address the data movement issue [10]. ISAAC [2] and PRIME [11] exploit analog characteristics of non-volatile memory to support matrix multiplication in memory. These architectures transfer the digital input data into an analog domain and pass the analog signal through a crossbar ReRAM to compute matrix multiplication. The matrix values are stored as multi-bit memristors in a crossbar memory. Although these PIM-based designs presented superior efficiency, there are several limitations when using PIM for CNN training. First, the precision of the design is bounded to fixed-point precision as determined by the number of multi-bit memristors used to represent a value. However, CNN models often need to be trained with floating point precision to achieve high classification accuracy [12, 13]. For example, GoogleNet, trained with 32-bit fixed point values, achieves 3% lower classification accuracy than the one trained with 32-bit floating points. In addition, earlier work showed that, without enough precision, the model training is likely to diverge or provide low accuracy [12–14]. Most commercial CNN accelerators train their models using floating point precision, e.g., bfloat16 [15]. The bfloat16 is a half precision floating point format utilized in Intel AI processors, such as Nervana NNP-L1000, Xeon processors, and Intel FPGAs [16–18], Google Cloud TPUs [19–21], and TensorFlow [21, 22].

Another limitation is that the state-of-the-art PIM-based designs utilize costly digital-to-analog (DAC) and analog-to-digital converter (ADC) blocks. For example, recent work in [23] designed an analog-based memristive accelerator to support floating point operations. However, the mixed-signal ADC/DAC blocks take the majority of the chip area and power, e.g., 98% of the total area and 89% of the total power, and do not scale as fast as the CMOS technology does [2]. In addition, prior PIM designs use multi-bit memristor devices that are not sufficiently reliable for commercialization unlike commonly-used single-level NVMs, e.g., Intel 3D Xpoint [24]. Their very expensive write operations frequently

occur during the training. For example, work in [25–27] extend the application of analog crossbar memory to accelerate training, but they still have expensive converter units and multi-bit devices. PipeLayer [1] modified the ISAAC [2] pipeline architecture and use spike-based input to eliminate ADC and DAC blocks. However, the computation of PipeLayer still happens on the converted data and its precision limits to fixed-point operations.

In this paper, we propose FloatPIM, a novel high precision PIM architecture, which significantly accelerates CNNs in both training and testing with the floating-point representation. This paper presents the following main contributions:

- **FloatPIM directly supports floating-point representations, thus enabling high precision CNN training and testing.** To the best of our knowledge, FloatPIM is the first PIM-based CNN training architecture that exploits analog properties of the memory without explicitly converting data into the analog domain. FloatPIM is flexible in that it works with floating-point as well as fixed-point precision.
- **FloatPIM implements the PIM operations directly on a digital data stored in memory using a scalable architecture.** All computations in FloatPIM are done with bitwise NOR operation on a single bit bipolar resistive devices. This eliminates the overhead of ADC and DAC blocks to transfer data between the analog and digital domain. It also completely eliminates the necessity of the multi-bit memristors, thus simplifying manufacturing.
- **We introduce several key design features that optimize the CNN computations in PIM designs.** FloatPIM breaks the computation into computing and data transfer phases. In the computing mode, all blocks are working in parallel to compute the matrix multiplication and convolution tasks. During the data transfer mode, FloatPIM enables a pipelined, row-parallel data transfer between neighboring memory blocks. This significantly reduces the cost of internal data movement.
- **We evaluate the efficiency of FloatPIM on popular large-scale networks with comparisons to the state-of-the-art solutions.** In this paper, we show how FloatPIM accelerates computations of AlexNet, VGGNet, GoogleNet, and SqueezeNet for ImageNet dataset [28]. In terms of accuracy, FloatPIM supporting floating point precision can achieve up to 5.1% higher classification accuracy than the one using fixed point representation. In terms of efficiency, our evaluation shows that FloatPIM in training can achieve 303.2× and 48.6× (4.3× and 15.8×) speedup and energy efficiency as compared to the state-of-the-art GPU (PipeLayer PIM accelerator [1]). In training, FloatPIM provides 324.8× and 297.9× (6.3× and 21.6×) speedup and energy efficiency as compared to GPU (ISAAC PIM accelerator [2]) respectively.

2 BACKGROUND

2.1 DNN Training

Figure 1a show an example of neural networks in a fully-connected layer, where each neuron is connected to all neurons in the previous layer using weights. Figure 1a shows the computation of a single neuron in the feed-forward pass. The outputs of the neurons in the previous layer are multiplied with the weight matrix, and the

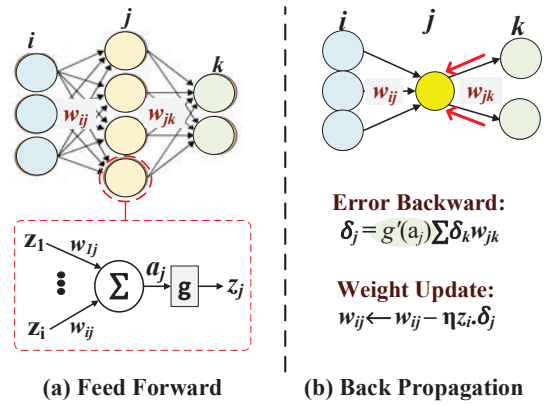


Figure 1: DNN computation during (a) feed-forward and (b) back-propagation.

results are accumulated in each neuron (a_j). The result of accumulation passes through an activation function (g). This function is traditionally a *Sigmoid* [29], but recently Rectangular Linear Unit (ReLU) is the most commonly used [3]. The activation results are used as the input for the neurons in the next layer.

The goal of the training is to find the network weights using the gradient descent method. It runs in two main steps: feed-forward and back-propagation. In the feed-forward step, it examines the quality of the current neural network model for classifying a predefined number of training data points, also known as batch size. It then stores all intermediate neurons values (Z_i) and the derivatives of the activation function $g'(a_j)$ for all data point in a batch. The next step is to update the neural network weights, often referred to the back-propagation step. Figure 1b illustrates the back-propagation that performs two major tasks: error backward and weight update.

Error backward: Back-propagation, first measures the loss function in the CNN output layer using:

$$J = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{k}(y_j^{(i)})$$

Based on a chain rule, it identifies the gradient of the loss function to each weight in the previous layer using:

$$\frac{dJ}{da_j} = \sum_k \frac{dJ}{da_k}\frac{da_k}{da_j}$$

The error vector in a layer j (δ_j) is computed backward depending on the error vector in the layer k (δ_k) and derivatives of the activation in a layer j ($g'(a_j)$). Assuming $\delta_j = -dJ/da_j$, the following equation defines the gradient of the entropy loss for each neuron in the layer j:

$$\delta_j \; where \begin{cases} (t_j - y_j), & \text{if } j \textit{ is an output unit} \\ -g'(a_j)\sum_k \delta_k W_{jk}, & \text{if } j \textit{ is a hidden unit} \end{cases}$$

In CNN, the convolution layer trains in a similar way to the fully-connected layer, but with higher computation complexity. This is because each output element in the convolution layers depends on the movement of the convolution kernel through a range of the

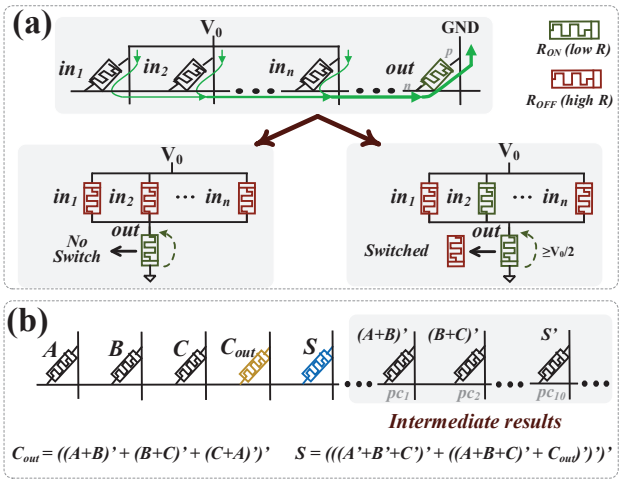


Figure 2: Digital PIM operations. (a) NOR operation. (b) 1-bit addition.

input matrix. The following equation shows how convolution gets the gradient of a loss function to each input:

$$\frac{dJ}{da^j_{r,s}} = \sum_{r,s\in Q} \frac{dJ}{da^k_Q} \cdot \frac{da^k_Q}{da^j_{r,s}} = \sum_{r,s\in Q} \delta^k_Q \frac{da^k_Q}{da^j_{r,s}}$$

Similar to the expansion of the equation in the fully-connected layers, we have:

$$\frac{dJ}{da^j_{r,s}} = \sum_a \sum_b \underbrace{[g'_{a_{r,s}} \cdot \sum_{m=0}^{k_1-1} \sum_{n=0}^{K_2-1} \delta^k_{a-m,b-n} W^k_{m,n}]}_{\delta^k_{r,s}} * Z^i_{i+m',j+n'}$$

where $*$ denotes the convolution.

Weight update: Finally, the weights are updated by subtracting the current weights from the $\eta.z_i.\delta_j$ matrix:

$$W_{ij} \longleftarrow W_{ij} - \eta \frac{dJ}{dW_{ij}} = W_{ij} - \eta \delta_j Z_i$$

where η is a learning rate, and Z_i is the output of the neurons after the activation function in the layer i. Note that both $g'(a_j)$ and Z_i are calculated and stored during the feed-forward step.

2.2 Digital Processing In-Memory

Processing in-memory digitally involves input-based switching of memristor, unlike the conventional memristor processing which uses ADC/DAC blocks to convert data between analog and digital domains. Digital PIM performs the computation directly on the stored values in the memory without reading them out or using any sense amplifier. Digital PIM has been designed in literature [30–34] and fabricated in [35], to implement logic using memristor switching. The output device switches between two resistive states, R_{ON} (low resistive state, '1') and R_{OFF} (high resistive state, '0'), whenever the voltage across the device, i.e., p and n terminals shown in Figure 2a, exceeds a threshold [36]. This property can be exploited to implement NOR gate in the digital memory by applying a fixed voltage, V_0 across the memristor devices [30]. The output memristor

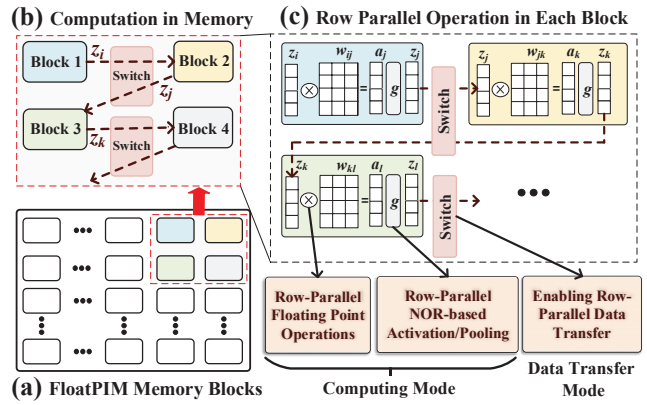


Figure 3: Overview of FloatPIM.

is initialized to R_{ON} in the beginning. To execute NOR in a row, an execution voltage, V_0, is applied at the p terminals of the inputs while the p terminal of the output memristor is grounded, as shown in Figure 2. The aim is to switch the output memristor from R_{ON} to R_{OFF} when one or more inputs stored '1.' value (low resistance). Since NOR is a universal logic gate, it can be used to implement other logic operations like addition [37, 38] and multiplication [39]. For example, 1-bit addition (inputs being A, B, C) can be represented in the form of NOR as,

$$C_{out} = ((A+B)' + (B+C)' + (C+A)')'. \quad (1a)$$

$$S = (((A' + B' + C')' + ((A+B+C)' + C_{out})')')'. \quad (1b)$$

Here, C_{out} and S are the generated carry and sum bits of addition. Also, $(A+B+C)'$, $(A+B)'$, and A' represent $NOR(A,B,C)$, $NOR(A,B)$, and $NOR(A,A)$ respectively. Figure 2b visualizes the implementation of 1-bit addition in a memristor-based crossbar memory. The processing cells, pc, store the intermediate results and are not used to store data. Digital processing in-memory achieves maximum performance when the operands are present in the same row because, in this configuration, all the bits of an operand are accessible by all the bits of the other operand. This increases the flexibility in implementing operations in memory.

In-memory operations are in general slower than the corresponding CMOS-based implementations. This is because memristor devices are slow in switching. However, this PIM architecture can provide significant speedup with large parallelism. PIM can support addition and multiplications in parallel, irrespective of the number of rows. For example, to add values stored in different columns of memory, it takes the same amount of time for PIM to process the addition in a single row or all memory rows. However, the processing time in conventional cores highly depends on the data size.

3 FLOATPIM OVERVIEW

In this paper, we propose a digital and scalable processing in-memory architecture (FloatPIM), which accelerates CNNs in both training and testing phases with precise floating-point computations. Figure 3a shows the overview of the FloatPIM architecture consisting of multiple crossbar memory blocks. As an example, Figure 3b shows how three adjacent layers (recall the structure of layers and notations shown in Figure 1a) are mapped to the

FloatPIM memory blocks to perform the feed-forward computation. Each memory block represents a layer, and stores the data used in either testing (i.e., weights) or training (i.e., weights, the output of each neuron before activation, and the derivative of the activation function (g')), as shown in Figure 3c. With the stored data, the FloatPIM performs with two phases: (i) computing phase and (ii) data transfer phase. During the computing phase, all memory blocks work in parallel, where each block processes an individual layer using PIM operations. Then, in the data transfer phase, the memory blocks transfer their outputs to the blocks corresponding to the next layers, i.e., to proceed either the feed-forward or back-propagation. The switches are shown in Figure 3b control the data transfer flows.

In Section 4, we present how each FloatPIM memory block performs CNN computations for a layer. The block supports in-memory operations for key CNN computations, including vector-matrix multiplication, convolution, and pooling (Section 4.1.) We also support the activation functions like ReLU and Sigmoid in memory. MIN/MAX pooling operations are implemented using in-memory search operations. Our proposed design optimizes each of the basic operations to provide high performance. For example, for the convolution which requires shifting convolution kernels across different parts of an input matrix, we design shifter circuits that allow accessing weight vectors across different rows of the input matrix. The feed-forward step is performed entirely inside memory by executing the basic PIM operations (Section 4.2.) FloatPIM also performs all the computations of the back-propagation with the same key operations and hardware to the one used in the feed-forward (Section 4.3.)

In Section 5, we describe how the memory blocks compose the entire FloatPIM architecture. FloatPIM further accelerates the feed-forward and back-propagation by fully utilizing the parallelism provided in the PIM architecture, e.g., row/block-parallel PIM operations. We show how these tasks can be parallelized for both feed-forward and back-propagation across a batch, i.e., multiple inputs at a time. It uses multiple data copies pre-stored in different blocks in memory. Section 6 presents in-depth circuit-level details of the PIM-based floating point addition and multiplication.

4 CNN COMPUTATION IN A FLOATPIM BLOCK

In this section, we show how a FloatPIM memory block performs the training/testing task[1] of a single CNN layer. Figure 4 shows a high-level illustration of the training procedure of a fully-connected layer in FloatPIM. As discussed in Section 2, CNN training has two steps: feed-forward and back-propagation. During the feed-forward step, FloatPIM processes the input data in a pipeline stage. For each data point, FloatPIM stores two intermediate neuron values: (i) the output of each neuron after the activation function (Z_i) and (ii) the gradient of activation function for the accumulated results ($g'(a_j)$). In the back-propagation step, FloatPIM first measures the loss function in the last output layer and accordingly updates the weights of each layer using the intermediate values stored during the feed-forward step. As Figure 4b and c show, the error sequentially propagates and updates the weights in the previous layer.

[1]Please note that only with the feed-forward step, FloatPIM supports the testing task, i.e., inference, where an input data processes through different CNN layers.

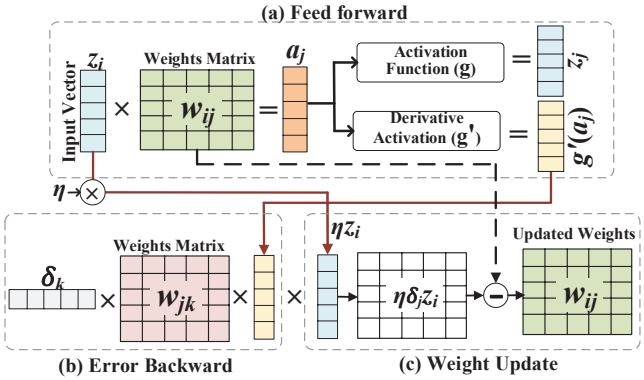


Figure 4: Overview of CNN Training.

CNNs use similar operations for both the fully-connected and convolution layers. In the feed-forward step, the CNN computations are vector-matrix multiplication for the fully-connected layers and convolution operation for the convolution layers. In the back-propagation, FloatPIM uses the same vector-matrix multiplication to update the weights for fully-connected layers, while the weights of the convolution layers are updated using the in-memory vector-matrix multiplication and convolution. In the next subsection, we first describe how FloatPIM support basic testing/training operations of a single CNN layer in digital PIM.

4.1 Building Blocks of CNN Training/Inference

Vector-Matrix Multiplication: One of the key operations of CNN computation is vector-matrix multiplication. The vector-matrix multiplication is accomplished by multiplications of the stored inputs and weights, and addition to accumulating the results of the multiplications. Figure 5a shows an example of the vector-matrix multiplication. As discussed in Section 2.2, the in-memory operations on digital data can perform in a row-parallel way, by performing the NOR-based operations on the data located in different columns. Thus, the input-weight multiplication can be processed by the row-parallel PIM operation. In contrast, the subsequent addition cannot be done in the row-parallel way as its operands are located in different rows. This hinders achieving maximum parallelism that the digital PIM operations offer.

Figure 5b shows how our design implements row-parallel operations by locating the data in a PIM-compatible manner. FloatPIM stores multiple copies of the input vector horizontally and the transposed weight matrix in memory (W_{ij}^T). FloatPIM first performs the multiplication of the input columns with each corresponding column of the weight matrix. The multiplication result is written in another column of the same memory block. Finally, FloatPIM accumulates the stored multiplication results column-wise with multiple PIM addition operations to the other column.

FloatPIM enables the multiplication and accumulation to perform independent of the number of rows. Let us assume that each multiplication and addition take T_{Mul} and T_{Add} latencies respectively. Thus, we require $M \times T_{Mul}$ and $N \times T_{Add}$ latencies to perform the multiplication and accumulation respectively, where the size of the weight matrix is M by N.

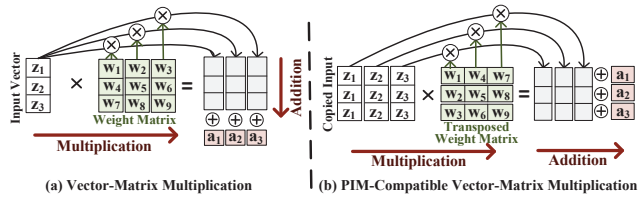
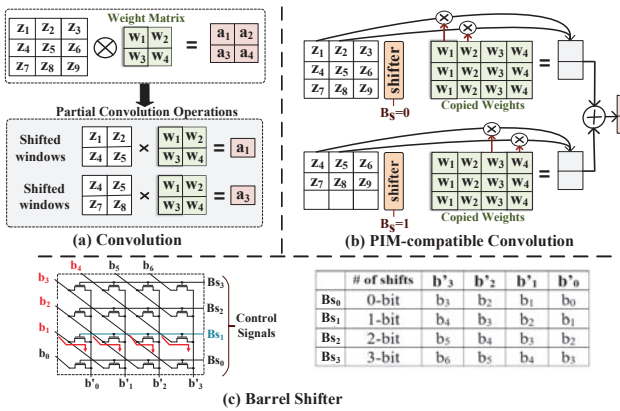


Figure 5: Vector-matrix multiplication.



	# of shifts	b'_3	b'_2	b'_1	b'_0
Bs_0	0-bit	b_3	b_2	b_1	b_0
Bs_1	1-bit	b_4	b_3	b_2	b_1
Bs_2	2-bit	b_5	b_4	b_3	b_2
Bs_3	3-bit	b_6	b_5	b_4	b_3

Figure 6: Convolution operation.

Convolution: As shown in Figure 6a, the convolution layer consists of many multiplications, where a shared weight kernel shifts and multiplies with an input matrix. A naive way to implement the convolution is to write all the partial convolutions for each window movement by reading and writing the convolution weights repeatedly in memory. However, this method has high-performance overhead in PIM, since non-volatile memories (NVMs) have slow write operation.

FloatPIM addresses this issue by replacing the convolution with light-weight interconnect logic for the multiplication operation. Figure 6b illustrates the proposed method which consists of two parts: (i) It writes all convolution weights in a single row and then copies them in other rows using the row-parallel write operation that happens just in two cycles. This method enables the input values to be multiplied with any convolution weights stored in another column. (ii) It exploits a configurable interconnect to virtually model the shift procedure of the convolution kernel. This interconnect is a barrel shifter which connects two parts of the same memory.

Figure 6c shows the structure of a barrel shifter that provides a 3-bits shift operation as an example. Depending on the B_S control signals, a barrel shifter connects different $\{b_1, \ldots, b_6\}$ bits to $\{b'_1, \ldots, b'_4\}$. The number of required shift operations depends on the size of the convolution windows. For the example shown in Figure 6b, for a 2×2 convolution window, the barrel shifter supports a single shift operation using $B_S = 0$ or $B_S = 1$ control signal. Similarly, for a $n \times n$ convolution kernel, the number of shift operation is $n-1$. Our FloatPIM supports up to a 7×7 convolution kernel, and it covers all the tested popular CNN structures. Note that FloatPIM can also support n larger than 7 by rewriting shifted input matrices into other columns.

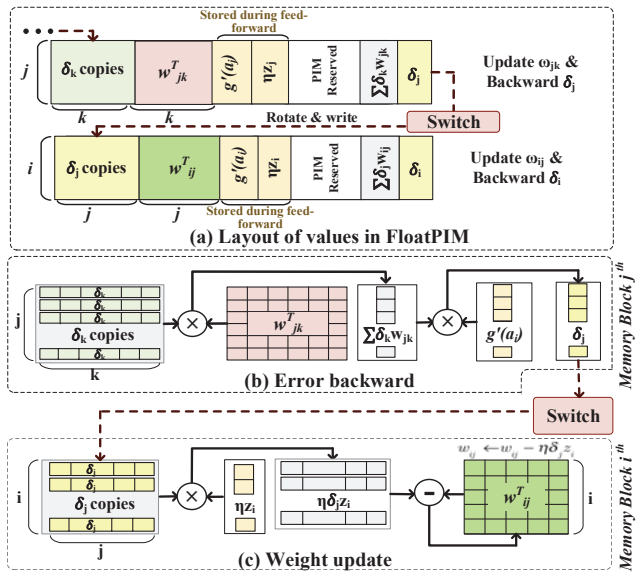


Figure 7: Back-propagation of FloatPIM.

Row-Parallel Write Both vector-matrix multiplication and convolution require to copy the input or weight vectors in multiple rows. Since writing multiple rows sequentially would degrade performance, FloatPIM supports a row-parallel write operation that writes the same value to all rows only in two cycles. In the first cycle, the block activates all columns containing "1" by connecting the corresponding bitlines to V_{SET} voltage, while the row driver sets the wordlines for the destination rows to zero. It writes 1s on all the selected memory cells at the same time. In the second cycle, the column driver connects only the bitlines which carry "0" bit to the zero voltage, while the row driver sets the wordlines to V_{RESET}. This writes the input to all memory rows.

MAX/MIN Pooling: The goal of MAX (MIN) pooling layer is to find a maximum (minimum) values among the neuron's output in the previous layer. To implement pooling in memory, we use a crossbar memory with the capability of searching for the nearest value. Work in [40] exploited different supply voltages to give weight to different bitlines and enable the nearest search capability. Using this hardware, we implement MAX pooling by searching for a value which has the nearest similarity to the largest possible value. Similarity the MIN pooling can be implemented by searching for a row of a memory which has the closest distance to the minimum possible value. Since the values are floating point, the search happens in two phases. First, we find value with the highest exponent; then for values with the same maximum exponent, we search to find a value with the which has the largest mantissa.

4.2 Feed-Forward Acceleration

There are three major types of CNN layers: fully-connected, convolution, and pooling layers. For each type of the three layers, we exploit different data allocation mechanisms to enable high parallelism and perform the computation tasks with minimal internal data movement. For the fully connected layer, the main computation is vector-matrix multiplication. CNN weights (W_{ij}) are stored as a matrix in memory and multiplied with the input vector stored

in a different column. This multiplication and addition can happen between the memory columns using the same approach we introduce for PIM-compatible vector-matrix multiplication. The convolution is another commonly used operation in the deep neural network, which is implemented using the PIM-compatible convolution hardware introduced in Section 4.1.

After the fully connected and convolution layers, there is an activation function. We perform activation functions with a sequence of in-memory NOR operations. For example, we perform the ReLU function by subtracting all neuron's output from the ReLU threshold value (THR). This subtraction can happen in a row parallel way, where the same THR value is written in another column of the memory block (all rows). Finally, we write the threshold value in a row-parallel way in all memory rows that the subtracted results have positive sign bits. For the neuron's output with negative sign bits, we can avoid subtraction and instead write 0 value on all such rows. We also support non-linear activation functions, e.g., Sigmoid, using the PIM-based multiplication and addition based on Taylor expansion. For example, for Sigmoid, we consider the first three terms of the Taylor expansion ($1/2 + 1/4a_i - 1/48a_i^3$). The Taylor expansion is implemented in memory as a series of the control signals on the pre-activation vector stored in a column of a crossbar memory. First, we exploit in-memory multiplications to calculate different powers of the pre-activation values, e.g, a_3, in a row parallel way. Then, we multiply the values with a pre-stored Taylor expansion coefficient, e.g., $1/4, 1/48$, stored in reserved columns of the same memory. Finally, the result of activation can be calculated using addition and subtraction. Note that our approach parallelizes the activation function for all neuron's output of a DNN layer which is stored in a single column but different rows of a memory block. Moreover, since FloatPIM does not use separate hardware modules for any layers but implements them using basic memory operations. Hence, with no changes to memory and minimal modifications to the architecture, FloatPIM can support the fusion of multiple layers.

4.3 Back-Propagation Acceleration

Figure 7 shows the CNN training phases in fully-connected layer: (i) Error backward, where the error propagates through different CNN layers. (ii) Weight update, which calculates the new CNN weights depending on the propagated error.

Fully-Connected Layer: Figure 7 shows the overview of the DNN operations to update the error vector (δ). Figure 7a shows the layout of the pre-stored values in each memory block in order to perform the back-propagation. Each memory block stores the weights, the output of neurons (Z) and derivatives of the activation ($g'(a)$) in a block for each layer.

During the back-propagation, δ vector is the only input to each memory block. The error vector propagates backward in the networks. The error backward starts with multiplying the weights of the jth layer (W_{jk}) with the δ_k error vector. To enhance the performance of this multiplication, we copy the same δ_k vector on the j rows of the memory (as shown in Figure 7b). The multiplication of the transposed weights and copied δ_k matrix is performed in a row parallel way.

Finally, FloatPIM accumulates all stored multiplication results ($\sum \delta_j W_{jk}$). One way is to use $k * bw$-bits columns that stores all the results of the multiplications where bw is the values bit-width. Instead, we design an in-memory multiply-accumulation (MAC) operation which reuses the memory columns for the accumulation. FloatPIM consecutively performs multiplication and addition operations. This reduces the number of required columns to bw-bits, and results in significant improvements in the area efficiency per computation. Assuming that T_{Mul} and T_{Add} take for multiplication and addition respectively, the multiplication of the weight and δ_k matrix is computed in $(T_{Mul} + T_{Add}) \times k$. Since FloatPIM performs the computation in a row-parallel way, the performance of computation is independent on j. The result of $\sum \delta_j W_{jk}$ is a vector with j elements (Figure 7b). This vector multiplies element-wise by $g'(a_j)$ vector in S cycles and row-parallel way. Note that during feed-forward the $g'(a_j)$ is written in a suitable memory location which enables column-wise multiplication with no internal data movement.

The result of the multiplication is a δ_j error vector, and it is sent to the next memory block to update the weights (Figure 7c). The error vector is used for both updating the weights (W_{ij}) and computing the backward error vector (δ_i) in a layer i. Next, FloatPIM transfers the δ_j vector to the next memory block which is responsible to update the W_{ij} weights. The δ_j vector is copied in i memory rows next to the W_{ij}^T matrix using the copy operation.

For the weight update, the δ_j matrix is multiplied with ηZ_i vector, where ηZ_i is calculated and stored during the feed-forward step. This takes $j \times T_{Mul}$. As Figure 7b shows, the result of the multiplication is a matrix with $j \times i$ elements. Finally, FloatPIM updates the weights by subtracting W_{ij}^T from the $\eta \delta_j Z_i$ matrix. This subtraction happens column by column and the result will be rewritten in the same column as the new weight matrix. This reduces the number of required memory columns from $k \times bw$ to bw columns.

Convolution Layer: There are a few differences between the feed-forward and convolution layers in the back-propagation step. Unlike the feed-forward layer, the error term is defined as a matrix, i.e., the error backward computes the error matrix in a layer j (δ^j) depending on the error matrix in a layer k (δ^k). The update on the error matrix happens by computing the convolution of the δ^j and weight matrix, where the size of weights is usually much smaller than δ $(m, n << r, s)$. This operation can be implemented in-memory using the same hardware we used to accelerate the convolution in the feed-forward layer. Next, the generated matrix from the convolution is multiplied with the derivatives of the activation function (g'), which is already stored in memory during the feed-forward step. It is computed the same PIM functionalities used for the fully-connected layers. Finally, the generated matrix is convolved with Z^i which is the matrix corresponding to the output of the previous CNN layer. When a pooling layer is used, the Z^i is the output of that layer.

5 FLOATPIM ARCHITECTURE

Figure 8 shows the overview of the proposed FloatPIM architecture processing multiple CNN layers. FloatPIM consists of 32 tiles, where each tile has 256 crossbar memory blocks which have row and column drivers (A). To support the convolution kernel, we exploit the barrel shifter in each memory block. In both feed-forward and back-propagation, FloatPIM needs to send the data to the next memory block in order to continue the computation. FloatPIM

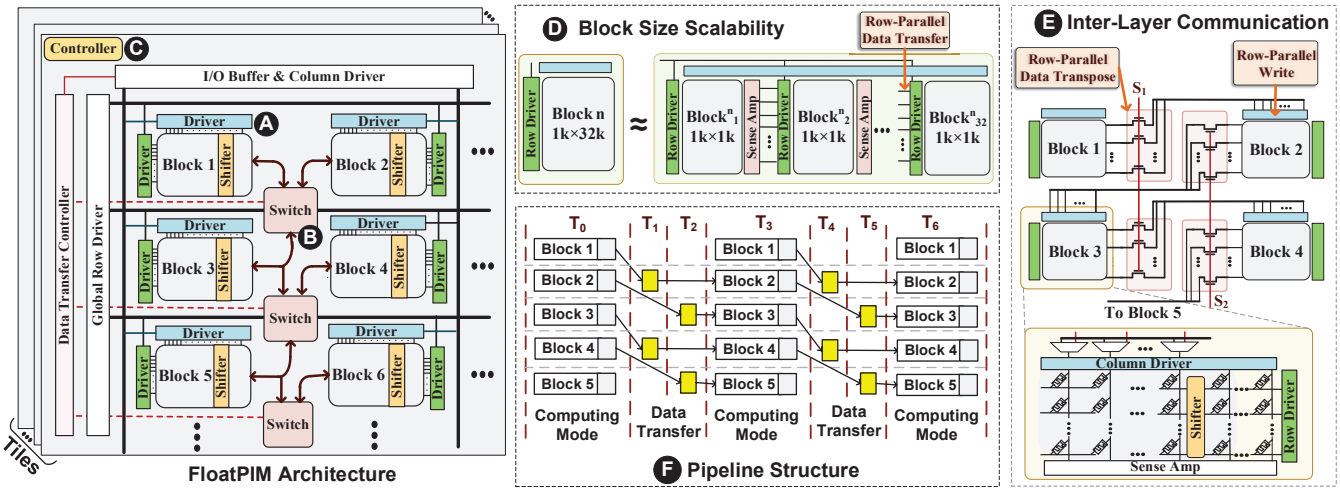


Figure 8: FloatPIM memory architecture.

exploits switches which enable parallelized data transfer between the neighboring blocks (Ⓑ). The controller block calculates the loss function and controls the row-driver, column driver and the switches used for fast data transfer (Ⓒ).

5.1 Block Size Scalability

Due to the existing challenges in the crossbar memory [41, 42], each memory block is assumed to have a size of $1K \times 1K$. However, to enable each block to process a single DNN layer, each block needs much larger bitline, e.g., $16 - 32K$ to store input/output and weight matrix. In addition, since our design extends NOR operation to addition/ multiplication, it requires reserved bitlines to keep the intermediate result of the computation. For N-bit addition and multiplication, each memory row requires to put 12 and $16N - 19$ bits respectively for storing intermediate operations. However, since only one of the additions/ multiplications happens at a time in a memory block, the intermediate cells for addition and multiplication are shared. In bfloat16, this results in only 93 additional memory cells to support 7-bit mantissa multiplication. Thanks to the scalability of the FloatPIM, i.e., working on digital data, FloatPIM performs the computation on a few cascaded memory sub-blocks (Ⓓ). All blocks are controlled by the same column driver, but different row drivers that enable fine-tune block activation. FloatPIM transfers data between two neighbor blocks in a row-parallel way by reading the output of a block and writing it into the next memory. Regardless of the number of values/rows, the execution time of this parallel data transfer only depends on the bit-width of the values ($N + 1$ cycles for N-bit data transfer). Cascading the blocks comes at the expense of increasing the cost in internal data movement. Our evaluation shows that cascading a block into 32 sub-blocks increases FloatPIM execution time only by 3.8% (less than 3.4% energy overhead) as compared to assuming ideal $1k \times 32K$ block size.

5.2 Inter-layer Communication

During the data transfer phase, the results computed in one memory block are written to another memory block as an input for the next computation phase. Let us assume two fully-connected CNN layers are mapped to two neighborhood blocks. The results of the PIM operation of the first memory block need to be written as an input in the second block. Assuming a CNN layer with 1K neurons, we need to write 1K values to process the next computation.

To speedup the write, we design a switch which enables fast data transfer between the neighboring memory blocks. The data transfer between the blocks happen with rotation and write operations. For example, in the feed-forward step, the generated vertical output vector needs to be rotated and copied into several rows of the next memory block (explained in Section 4.1). Similarly, in back-propagation the generated δ vector of backward error needs to be rotated and written in the next memory block to process the weight update (explained in Section 4.3). The circuit in Figure 8(Ⓑ) shows how FloatPIM supports the rotation and write operations between the blocks. FloatPIM locates the memory blocks such that the tail of the neighbor blocks face together. Then, it exploits switches to connect the adjacent memory blocks. Each block is connected to its two adjacent neighbors. During the computation phase, these switches are in off mode. So, each memory block can individually perform its computation. Then, during the data transfer phase, FloatPIM connects the blocks together in order to move data in a row parallel way. For example, connecting S_1 switches writes each column of the Block 1 into a row of Block 2. This data transfer can happen in a bit-serial and row-parallel way. Similarly, activation S_2 control signal connects the Block 2 to Block 3.

Figure 8Ⓕ shows the functionality of FloatPIM memory blocks working in a pipeline structure. Each memory block models the computation of either a fully-connected or a convolution layer. At the first cycle (T_0), the switches are disconnected and all the memory blocks are in the computing mode and work in parallel. Then, FloatPIM works in the data transfer mode for two cycles. In the first transferring cycles (T_1), all odd blocks send their output values to their neighboring blocks with even indices ($S_1 = 1, S_2 = 0$). In the second transferring cycle (T_2), the even blocks are sending their generated output values to their neighbor odd blocks ($S_1 = 0$, $S_2 = 1$). This enables to complete all the required data transfers only within the two consecutive steps. For example, switches can

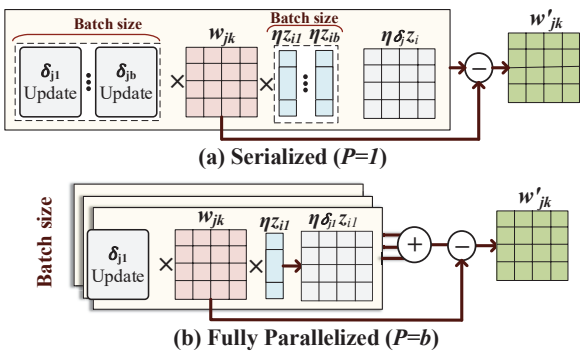


Figure 9: FloatPIM training parallelism in a batch.

transfer a vector of values with bw-bits in $2 * bw$ cycles, regardless of the number of rows. Each FloatPIM tile can process data points in the pipeline structure with $d = T_0 + T_1 + T_2$ cycle width, where T_0 and $T_1 + T_2$ are the computing and data transfer cycles, respectively.

It should be noted that FloatPIM takes care of non-consecutive, but the periodic connection in neural network layers, such as ResNet [43]. For example, in ResNet, the output of each layer can be used as an input in the next two consecutive layers. For these cases, FloatPIM requires a different pipeline stage, wherein the data transfer mode the output of a particular layer is sequentially sent to the second and then the third block. For the cases with the non-periodic connection, the controller reads the value of the block in a row-parallel way and writes them into another memory block. However, since these cases are not common, the proposed inter-block communication can still significantly improve efficiency.

5.3 FloatPIM Parallelism

The proposed design parallelizes computations across memory blocks and data located in different memory rows. In this section, we describe other parallelization strategies exploited in our implementation.

Parallelization of Feed-Forward: The CNN training happens in batch size windows (b). The batch size indicates the number of training data points which process in feed-forward before the back-propagation happen. In the feed-forward step, there is no dependency between the computation of different inputs in a batch; thus the feed-forward computation can be parallelized for all data points in a batch as well. In order to enable feed forward parallelism, FloatPIM replicates the CNN weights in different memory blocks, where each memory can process the information of a single data point in a batch. The number of tiles determines the feed-forward parallelism. FloatPIM can work in the highest performance if FloatPIM can parallelize the computation of all data point in a batch; otherwise, it reuses the memory blocks to perform the computation of multiple data points. In that case, each memory block needs to store the weights corresponding to multiple layers in order to avoid the costly write operation during feed-forward. In Section 7.6, we explore the impact of the number of FloatPIM tiles.

Parallelization of Back-Propagation: FloatPIM keeps all intermediate neuron values (Z and g') in memory and updates the weights accordingly. Figure 9 shows the functionality of FloatPIM memory blocks updating the weights of a CNN layer when there are b data points in a batch. In the back-propagation, FloatPIM cannot parallelize the computation in different layers, but the computation of different data points in a batch can be parallelized in each layer.

FloatPIM may store the intermediate values of all data points in a batch in a single memory block and processes them sequentially (Figure 9a). It results in a lower power and memory requirement. The efficiency depends on how many data points in a batch is processed by a block. When P is less than b, we call this low power configuration as FloatPIM-LP. To further improve the performance, FloatPIM can parallelize the computation of different data points in a batch by processing them in separated memory blocks (Figure 9b). Each memory block stores the information from the feed-forward in a specific batch, while all memory blocks need to store the same weight matrix. The error backward for all blocks performs in parallel. To update the weights, FloatPIM collects the $\eta.\delta.Z$ vectors from all memory blocks that process a data point in a batch. The combined vectors are subtracted from the stored weight matrix, and the updated weight matrix is written back into all memory blocks in parallel. We call this fully-parallelized strategy as FloatPIM-HP.

6 IN-MEMORY FLOATING POINT COMPUTATION

This work represents the very first implementation of floating point addition and multiplication in the crossbar memory. A floating point number consists of a binary number string with three different parts: a sign bit, an exponent part, and a fractional value. For example, the IEEE 754 32-bit floating point notation consists of a sign bit, eight exponent bits, and 23 fractional bits. The first bit in the floating point notation (A_{32}) represents the sign bit, where '0' represents a positive number. The next eight bits represent the exponent of the binary numbers ($A_{31}, \ldots, A_{24}$), ranging from -126 to 127. The following 23 bits ($A_{23}, \ldots, A_1$) represent the fractional part, also known as mantissa, which has a value between 1 and 2.

6.1 FloatPIM Multiplication

Floating point multiplication involves: (i) XORing the sign bits, (ii) addition of exponent bits, and (iii) fixed-point multiplication of mantissa bits. In-memory floating point multiplication requires storing the two operands and writing the result in the same row of another column. In FloatPIM, we XOR the sign bits and add the exponent bits using multiple NOR operations [37, 38]. While XOR takes 6 cycles, addition takes $13N_e$ cycles, where N_e is the number of exponent bits. The mantissa bits are multiplied in the way presented in [39]. While these operations are sequential, they can be parallelized over all rows in the memory. The latency and energy of FloatPIM multiplication can be formulated as:

$$T_{Mul} = (12N_e + 6.5N_m^2 - 7.5N_m - 2)T_{NOR}$$

$$E_{Mul} = (12N_e + 6.5N_m^2 - 7.5N_m - 2)E_{NOR}$$

6.2 FloatPIM Addition

Floating point addition involves: (i) left-shifting the decimal point (right-shifting mantissa) to make the exponents same, (ii) addition of shifted mantissa, (iii) normalizing the result. Assume the two floating point numbers to be added are A and B, where A_s (B_s), A_e

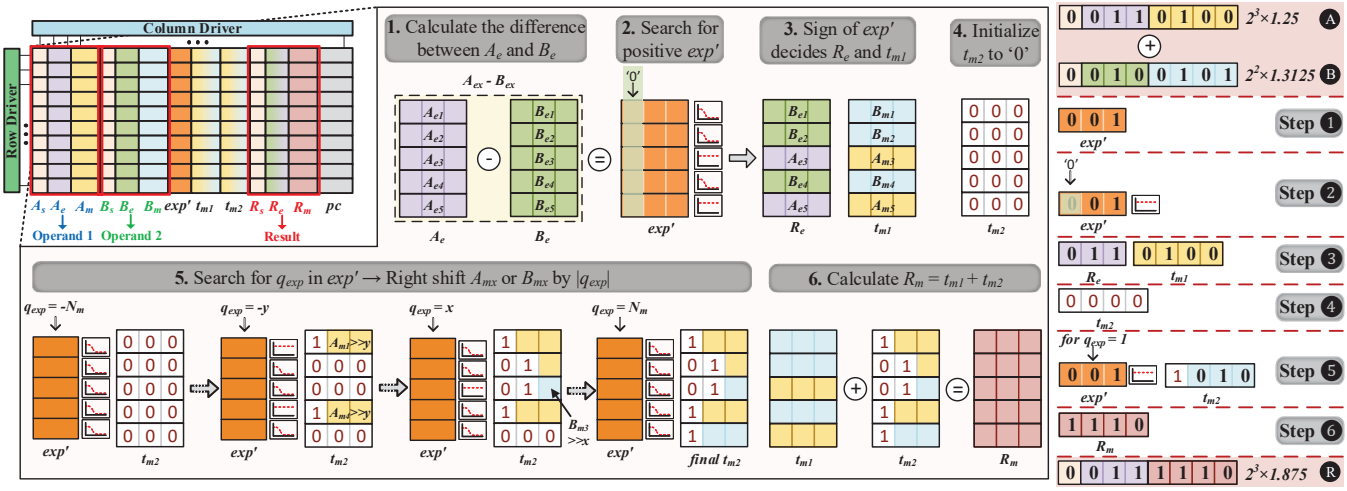


Figure 10: In-memory implementation of floating point addition.

(B_e), and A_m (B_m) represent the sign, exponent, and mantissa bits of A (B). We calculate the difference $A_e - B_e$ using an in-memory fixed point subtraction and store the result as exp'. This subtraction is implemented using multiple NOR operations as discussed earlier Based on exp', either A_m is shifted by $-exp'$ (if $exp' < 0$) or B_m is shifted by exp' (if $exp' > 0$). We may accomplish it by reading out exp', and then shifting the mantissa bits accordingly. However, it does not parallelize the operations over multiple rows, resulting in high latencies.

In this paper, we propose a novel alternative approach that handles exp' based shift by using exact search operation shown in [44]. Figure 10 shows an illustration of the proposed procedure. We create a new exponent, t_e, and two mantissas, t_{m1} and t_{m2}, to be added together. Here, t_e is the greater of A_e and B_e. t_{m1} is equal to the mantissa of the number with greater exponent, while t_{m2} is equal to the shifted mantissa. To identify the greater exponent, we search for '0' in the memory column containing the sign bit of exp'. For all the matched (unmatched) rows, t_e and t_{m1} are equal to A_e (B_e) and A_m (B_m) respectively. The copy operations for old exponents and mantissas, i.e., t_e and t_{m1}, are performed by column-wise NOT operations, eliminating any read/write operation. Next, t_{m2} in all rows is first initialized to '0's. We then search for each number in the range $\pm N_m$ in the columns containing exp'. Each search query q_{exp} where $-N_m \leq q_{exp} < 0$, t_{m2} is equal to A_m right-shifted by $|q_{exp}|$. On the other hand, when $0 \leq q_{exp} \leq N_m$, t_{m2} is equal to B_m right-shifted by $|q_{exp}|$. $(N_m - |q_{exp}|)$th bit of t_{m2} is set to '1' to incorporate the hidden digit in floating point representation. Shifting operation can be in turn carried out by simply copying the data with a NOT operation at the target location. Finally, t_{m1} and t_{m2} are added using fixed-point in-memory addition and stored as t'_m. To normalize t'_m used as final mantissa, for all $exp' \neq 0$, if the addition of t_{m1} and t_{m2} results in a carry, t'_m is right-shifted by one bit, while t_e is incremented by 1. If $exp' = 0$, t'_m is right-shifted by one bit and t_e is incremented by 1. Additionally, if the carry is generated in this case, the MSB of the shifted t'_m is set to '1'. The new t'_m (denoted as t_m) and t_e represent the output mantissa and exponent bits, respectively. The latency and energy of FloatPIM addition can be formulated as:

$$T_{Add} = (3 + 16N_e + 19N_m + N_m^2)T_{NOR} + (2N_m + 1)T_{search}$$

$$E_{Add} = 2(N_m + 1)E_{search} + 12(N_e + N_m)E_{NOR} + N_m E_{reset} + [2(N_e + N_m) + N_m^2/2 + N_m/2 + 1](E_{set} + E_{reset})$$

where the values of energy and execution time of the basic operations can be found here:

E_{set}	E_{reset}	E_{NOR}	E_{search}	T_{NOR}	T_{search}
$23.8fJ$	$0.32fJ$	$0.29fJ$	$5.34pJ$	$1.1ns$	$1.5ns$

7 EVALUATION

7.1 Experimental Setup

We have designed and used a cycle-accurate simulator based on Tensorflow [45, 46] which emulates the memory functionality during the DNN training and testing phases. For the accelerator design, we use HSPICE for circuit-level simulations to measure the energy consumption and performance of all the FloatPIM floating-point/fixed-point operations in 28nm technology. The energy consumption and performance are also cross-validated using NVSim [47]. We used System Verilog and Synopsys *Design Compiler* [48] to implement and synthesize the FloatPIM controller. For parasitics, we used the same simulation setup considered by work in [37]. The robustness of all proposed circuits, i.e., interconnect, has been verified by considering 10% process variations on the size and threshold voltage of transistors using 5000 Monte Carlo simulations. FloatPIM works with any bipolar resistive technology which is the most commonly used in existing NVMs. Here, we adopt memristor device with a VTEAM model [36]. The model parameters of the memristor, as listed in Table 1, are chosen to produce switching delay of 1ns, a voltage pulse of 1V and 2V for RESET and SET operations in order to fit practical devices [30]. Table 2 summarizes the device characteristics for each FloatPIM component. FloatPIM consists of 32 tiles, where each has 256 memory blocks to cover all the tested CNN structures. Each tile takes $0.96mm^2$ area and consumes 7.64mW power. In total,

Table 1: VTEAM Model Parameters for Memristor

k_{on}	$-216.2m/sec$	$V_{T,ON}$	$-1.5V$	x_{off}	$3nm$
k_{off}	$0.091m/sec$	$V_{T,OFF}$	$0.3V$	R_{ON}	$10k\Omega$
$\alpha_{on}, \alpha_{off}$	4	x_{on}	0	R_{OFF}	$10M\Omega$

Table 2: FloatPIM Parameters

Component	Params	Spec	Area	Power
Crossbar Array	size	1Mb	$3449.6\mu m^2$	6.14mW
Shifter	shift	6 levels	$19.26\mu m^2$	0.69mW
Switches	number	1K-bits	$32.69\mu m^2$	0.42mW
Max Pool	number	1	$80\mu m^2$	0.38mW
Controller	number	1	$401.4\mu m^2$	0.65mW
Memory Block	size	1Mb	$3{,}468.8\mu m^2$	6.83mW
Tile	number size	256 Blocks 256Mb	$0.96mm^2$	7.64mW
Total	number size	32 Tiles 8Gb	**30.64**mm^2	**62.60W**

Table 3: Workloads

	Model Size	Number of Layers Conv	FC	Classification Error
AlexNet [28]	224MB	5	3	27.4%
GoogleNet [49]	54MB	57	1	15.6%
VGGNet [8]	554MB	13	3	17.5%
SqueezeNet [50]	6MB	26	1	25.9%

FloatPIM takes $30.64mm^2$ area and consumes 62.60W power on average.

7.2 Workload

We perform our experiment on ImageNet [28] which is a large dataset with about 1.2M training samples and 50K validation samples. The objective is to classify each image to one of 1000 categories. We tested with four popular large-scale networks, i.e., AlexNet [28], VGGNet [8], GoogleNet [49], and SqueezeNet [50] to classify ImageNet dataset, summarized in Table 3. We compare the proposed FloatPIM with GPU-based DNN implementations (Conv:convolution, FC:fully-connected). The experiments are performed using Tensorflow [46] running on NVIDIA GPU GTX 1080. The performance and energy of GPU are measured by the `nvidia-smi` tool.

7.3 FloatPIM & Data Representation

Table 4 reports the classification error rate of different networks when they train with floating point and fixed point representation. For float precision, we used 32-bit floating point (Float-32) and bfloat16 (bFloat) [43], a commonly used representation in many CNN accelerators. For fixed-point precision, we used a 32-bit fixed point (Fixed-32) and 16-bit fixed point (Fixed-16) representations for FloatPIM training. For all networks, we perform the testing using Fixed-32 precision. To achieve maximum classification accuracy, it is essential to train CNN models using floating point representation. For example, using Fixed-16 and Fixed-32 for training, VGGNet provides 5.2% and 2.6% lower classification accuracy as compared to the same network trained based on bFloat. In addition, we observe that for all applications, bFloat can provide the same accuracy as Float-32, while computationally processes in a much faster way.

Table 4: Error rate comparison and PIM supports.

	Float-32	bFloat-16	Fixed-32	Fixed-16
AlexNet	27.4%	27.4%	29.6%	31.3%
GoogleNet	15.6%	15.6%	18.5%	21.4%
VGGNet	17.5%	17.7%	21.4%	23.1%
SqueezeNet	25.9%	26.1%	29.6%	32.1%
PIM Designs Support				
	Float-32	**bFloat-16**	**Fixed-32**	**Fixed-16**
ISAAC [2]	✗	✗	✓	✓
PipeLayer [1]	✗	✗	✓	✓
FloatPIM	✓	✓	✓	✓

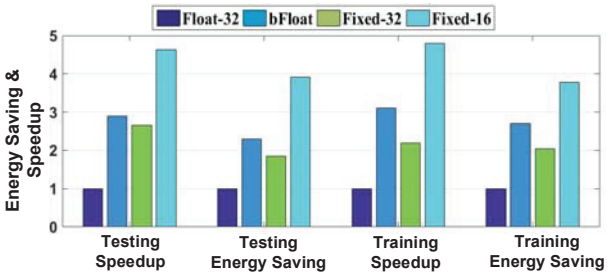


Figure 11: FloatPIM energy saving and speedup using floating point and fixed point representations.

This is because FloatPIM works based on the bitwise `NOR` operation, thus it can simply ignore processing the least significant bits of mantissas in floating point representation in order to accelerate the computation. Table 4 lists the supported computation precision by two recent PIM-based CNN accelerators [1, 2]. All existing PIM architectures can support CNN acceleration just using fixed-point values, which results in up to 5.1% lower classification accuracy than floating point precision supported by FloatPIM.

Figure 11 shows the speedup and energy saving of FloatPIM, on average for the four CNN models, using the fixed point and floating point representation for the CNN training and testing. All results are normalized to Float-32. Our evaluation shows that FloatPIM using bFloat can achieve 2.9× speedup and 2.5× energy savings as compared to FloatPIM using Float-32, while providing similar classification accuracy. In addition, FloatPIM using bFloat model can provide higher efficiency than Fixed-32. For example, FloatPIM using bFloat can achieve 1.5× speedup, 1.42× energy efficiency as compared to Fixed-32.

7.4 FloatPIM Training

Figure 12 compares the performance and energy efficiency of FloatPIM with the GPU-based implementation and *PipeLayer* [1] which is a state-of-the-art hardware accelerating CNN training using ISAAC [2] hardware. For PipeLayer, we used read/write latency of 29.31ns/50.88ns and energy of 1.08pJ/ 3.91nJ per spike as reported in the reference paper [1]. In addition, we used $\lambda = 4$ which provides reasonable efficiency.

During training, CNN requires a significantly large memory size to store the feed-forward information of different data points in a batch. For large networks, this information cannot fit on the GPU memory, thus it results in slow training. Our evaluation shows that

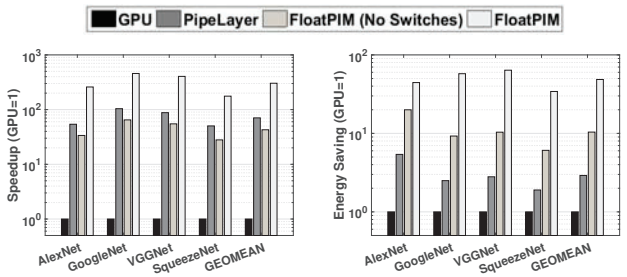


Figure 12: FloatPIM efficiency during training.

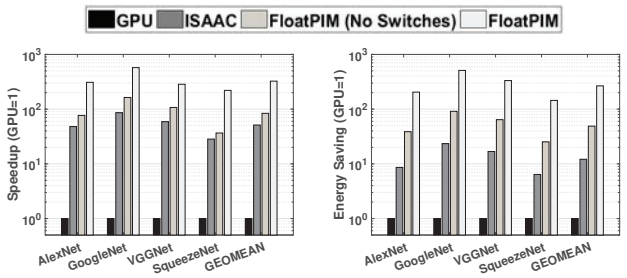


Figure 13: FloatPIM efficiency during the testing.

FloatPIM can achieve on average 303.2× speedup and 48.6× energy efficiency in training as compared to GPU-based approach. The higher efficiency of the FloatPIM is more obvious on the CNNs with more number of convolution layers. Figure 12 also compares FloatPIM efficiency over PipeLayer when it enables and disables the in-parallel data transfer between the memory blocks. Our evaluation shows that FloatPIM without parallelized data transfer provides 1.6× lower speedup, but 3.5× higher energy efficiency as compared to the PipeLayer. However, exploiting switches significantly accelerates the FloatPIM computation by removing the internal data movement between the neighboring blocks. Our evaluation shows that FloatPIM enabling in-parallel data transfer can achieve on average 4.3× speedup and 15.8× energy efficiency as compared to PipeLayer. The higher energy efficiency of FloatPIM comes from (i) its digital-based operation which avoids paying the extra cost of transferring data between the digital and analog/spike domain; (ii) the higher density of the FloatPIM which enables significantly better parallelism. The PipeLayer computing precision is bounded to fixed point operations, while FloatPIM provides the floating point precision which is essential for the highly accurate CNN training.

7.5 FloatPIM Testing

Figure 13 compares the performance and energy consumption of FloatPIM with NVIDIA GPU and ISAAC [2] which is the state-of-the-art PIM-based DNN accelerator. ISAAC works at 1.2GHz and uses 8-bits ADC, 1-bit DAC, 128×128 array size where each memristor cell stores 2 bits. We used the same parameters reported on the paper for the implementation [2]. We used FloatPIM (32-Tiles configuration) with and without in-parallel data transfer between the memory blocks. All execution time and energy results are normalized to GPU results. Our evaluation shows that both PIM-based architectures, i.e., ISAAC and FloatPIM, have significantly higher efficiency than GPU, since they address the data movement issue which is the main computation bottleneck of the conventional cores.

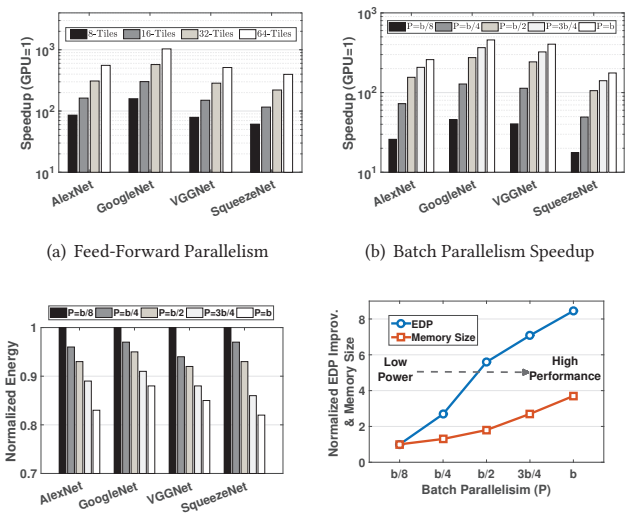
The results show that FloatPIM using bFloat implementation can achieve on average 6.3× and 21.6× (324.8× and 297.9×) speedup and energy efficiency improvement as compared to ISAAC (GPU-based approach). Our evaluation shows that FloatPIM with no in-parallel data transfer (no switches) can still provide 1.7× speedup and 3.9× energy efficiency as compared to ISAAC. FloatPIM provides the following advantages to ISAAC. (i) It eliminates the cost of internal data movement between memory blocks, which is a major bottleneck of most PIM architectures. (ii) It removes the necessity of using costly ADC and DAC blocks which takes the major portion of the ISAAC area and power. In addition, these mixed-signal blocks do not scale as fast as the CMOS technology does. (iii) FloatPIM is fully digital and scalable architecture which can work as accurate as the original floating point representation, while the precision of analog-based design limits to the fixed point representation.

7.6 Impacts of Parallelism

Feed-Forward: In the feed-forward step, we define the parallelism as the number of data points that can be processed in parallel. As discussed in Section 5.3, to improve the feed-forward performance, we can exploit different FloatPIM tiles to process different training data points in parallel. Figure 14a shows the impact of the number of tiles on FloatPIM performance speedup. We observed that increasing the number of tiles improves the performance of feed-forward. For example, FloatPIM using 32-tiles can achieve 1.83× higher performance as compared to FloatPIM with 16-tiles.

Back-Propagation: Unlike the feed-forward, the back propagation has dependencies between the CNN layers. This eliminates parallelizing the computation of different layers. However, in each layer, FloatPIM can parallelize the computation of different data points in a batch. In the low power design (FloatPIM-LP), a single block of memory processes a small set of data points in a batch ($P = b/8$). In contrast, the high-performance mode (FloatPIM-HP) can process all data points in a batch process in a single memory block ($p = b$). For the evaluation of this section, we consider $b = 128$ batch size for all the networks. Figure 14b,c show the speedup and normalized energy consumption of FloatPIM for a different level of parallelism. Our evaluation shows that increasing the parallelism from $p = b/8$ to $p = b$ improves the FloatPIM performance on average by 78.3×.

This parallelism comes at the cost of lowering the energy efficiency and increasing the effective memory size. The lower energy efficiency is due to the cost of error vector aggregation from different memory blocks in order to update the weights. FloatPIM-HP provides on average 15.4% lower energy efficiency than FloatPIM-LP. In addition, FloatPIM-HP requires to replicate the weight of CNN layer in all blocks corresponding to different data points in a batch. Figure 14d shows the normalized energy-delay product (EDP) and memory size of FloatPIM using different back-propagation parallelism. The results are normalized to FloatPIM-LP with the serialized process. Our evaluation shows that FloatPIM-HP can provide 8.2× higher EDP improvement while requiring 3.9× larger memory as compared to FloatPIM-LP.



(a) Feed-Forward Parallelism (b) Batch Parallelism Speedup

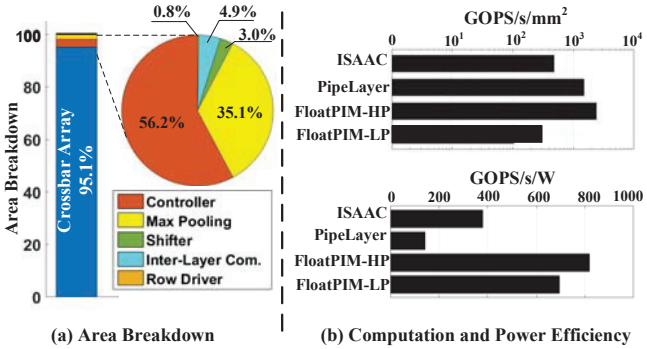
(c) Batch Parallelism Energy (d) Batch Parallelism Trade-off

Figure 14: The impact of parallelism on efficiency.

7.7 Computation/Power Efficiency

Unlike other PIM-based accelerators, FloatPIM makes very small changes to the existing crossbar memory. FloatPIM in 32-Tiles configuration takes 30.64mm^2 area. Our evaluation shows that in FloatPIM 95.1% of the area has been occupied by crossbar memory. The extra interconnects and switches added to enable fast convolution and inter-block connection only take 0.15% and 0.24% of the total FloatPIM area (Figure 15a). In addition, FloatPIM does not require to have fine-tuned control on the row/column drivers. To perform column-wise NOR operation, we require to only select 3 bitlines at a time. Similarly, the row driver can be activated on the entire memory rows (for computation) or a single row (for read/write operations). Our evaluation shows that multi-row activation results in less than 0.01% area overhead. Similarly, the controller takes about 3.0% of total chip area.

Figure 15b compares the computation (the number of 16-bit operations performed per second per mm^2) and power efficiency (the number of 16-bit operations performed per watt) of the FloatPIM with ISAAC [2] and PipeLayer [1]. Since FloatPIM supports floating point operations, we report the results as the number of floating point operations (*FLOPS*), while for other PIM designs we report it as the number of operations (*OPS*). Our result shows that FloatPIM can achieve 2,392.4 $GFLOPS/s/mm^2$ and 302.3 $GFLOPS/s/mm^2$ computation efficiency in high performance and low power modes respectively. The higher efficiency of FloatPIM-HP as compared to ISAAC (479.0 $GOPS/s/mm^2$) and PipeLayer (1,485 $GOPS/s/mm^2$) comes from its higher density which enables more computation to happen in the same memory area. For example, ISAAC uses ADC and DAC blocks which take a large portion of the area. In addition, PipeLayer still requires to generate spike which results in lower efficiency. In the low power mode, FloatPIM utilizes memory blocks with a large bitline size (in order to process all data points in a batch). This increases the area while the amount of computations stays the same, regardless of the bitline size.



(a) Area Breakdown (b) Computation and Power Efficiency

Figure 15: (a) FloatPIM area breakdown, (b) efficiency comparisons.

In terms of power, FloatPIM can provide much higher efficiency than both ISAAC and PipeLayer. FloatPIM removes the necessity of the costly internal data movement between the FloatPIM blocks by using the same memory block for both storage and computing. Our evaluation shows that FloatPIM in high performance and low power modes can achieve 818.4 $GFLOPS/s/W$ and 695.1 $GFLOPS/s/W$ power efficiency which are higher than both ISAAC (380.7 $GOPS/s/W$) and PipeLayer (142.9 $GOPS/s/W$) design.

7.8 Endurance Management

FloatPIM operations involve switching of memristor devices. This may affect the memory lifetime, given the endurance limits of the commercially available ReRAM devices. We implement an endurance management technique to increase the lifetime of our design. As discussed before, FloatPIM reserves some memory columns to store the intermediate states while processing. These columns are the most active and experience the worst endurance degradation. To increase the lifetime of the memory, we change the columns allocated for processing over time. This distributes the degradation across the block instead of being concentrated to a few columns, effectively reducing the worst case degradation per cell. It results in an increase in the lifetime of the device. For example, for memory blocks in FloatPIM with 1024 columns, and with 93 of them reserved for processing (in case of bfloat16), this management increases the lifetime of the device by ~11×. We also perform a sensitivity study of the lifetime of FloatPIM in terms of the number of classification tasks that can be performed. We observe that for the memory with endurance of 10^9 (10^{15}) writes, FloatPIM can perform 3.1×10^8 (3.1×10^{14}) classification tasks.

8 RELATED WORK

There are several recent studies adopting alternative low-precision arithmetics for DNN training [51]. work in [52, 53] proposed DNN training on hardware with hybrid dynamic fixed-point and floating point precision. However, in terms of convolutions neural network, the work in [14, 54] showed that fixed-point is not the most suitable representation for CNN training. Instead, the training can perform with lower bits of floating point values.

Modern neural network algorithms are executed on different types of platforms such as GPU, FPGAs, and ASIC chips [55–63].

Prior work attempted to fully utilize existing cores to accelerate neural networks. However, in their design the main computation still relies on CMOS-based cores, thus has limited parallelism. To address data movement issue, work in [64] proposed a neural cache architecture which re-purposes caches for parallel in-memory computing. Work in [65] modified DRAM architecture to accelerate DNN inference by supporting matrix multiplication in memory. In contrast, FloatPIM performs a row-parallel and non-destructive bitwise operation inside non-volatile memory block without using any sense amplifier. FloatPIM also accelerates DNN in both training and testing modes.

The capability of non-volatile memories (NVMs) to act as both storage and a processing unit has encouraged research in processing in-memory (PIM). Work in[10, 66] designed NVM-based Boltzmann machine capable of solving a broad class deep learning and optimization problems. Work in [2, 11] used ReRAM-based crossbar memory to perform matrix multiplication in memory and accordingly designed architecture to design PIM-based accelerator for CNN inference. Work in [25–27] used the same crossbar memory to accelerate CNN training. Work in [23] exploited the conventional analog-based memristive accelerator to support floating point operations. They exploit the exponent locality of data and the limited precision of floating point operations to enable floating point operations on fixed-point hardware. In contrast, we enable floating point operations inherently in memory and do not rely on data pre-processing and scheduling, making FloatPIM a general floating point accelerator which is independent of data. In addition, the analog approaches require mixed-signal circuits, e.g., ADC and DAC, which do not scale as fast as the CMOS technology scales. Work in [1] proposed *PipeLayer*, a PIM-based architecture based on [2] to accelerate CNN training by exploiting inter-layer and intra-layer parallelism. PipeLayer eliminates using ADC and DAC blocks by using the spike-based approach. However, similar to other PIM architectures, PipeLayer precision limits to fixed-point operations. In addition, it uses not sufficiently reliable multi-bit memristors, which are hard to program especially during training with a large number of writes.

Prior works exploited digital PIM operations to accelerate different applications such as DNNs [67–70], brain-inspired computing [31, 71], object recognition [72], graph processing [73, 74], and database applications [40, 75]. However, those designs do not support high precision computation and incur significant internal data movement.

9 CONCLUSION

In this paper, we proposed FloatPIM, the first PIM-based DNN training architecture that exploits analog properties of the memory without explicitly converting data into the analog domain. FloatPIM is a flexible PIM-based accelerator that works with floating-point as well as fixed-point precision. FloatPIM addresses the internal data movement issue of the PIM architecture by enabling in-parallel data transfer between the neighboring blocks. We have evaluated the efficiency of FloatPIM on a wide range of practical networks. Our evaluation shows that FloatPIM can achieve on average 4.3× and 15.8× (6.3× and 21.6×) higher speedup and energy efficiency as compared to PipeLayer (ISAAC), the state-of-the-art PIM accelerator, during training (testing).

ACKNOWLEDGEMENTS

This work was partially supported by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, and also NSF grants #1730158 and #1527034. Authors would like to thank Yunhui Guo for the initial discussion.

REFERENCES

[1] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, IEEE, 2017.
[2] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 14–26, IEEE Press, 2016.
[3] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.
[4] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.
[5] C. Dong, C. C. Loy, K. He, and X. Tang, "Learning a deep convolutional network for image super-resolution," in *European conference on computer vision*, pp. 184–199, Springer, 2014.
[6] L. Deng, D. Yu, *et al.*, "Deep learning: methods and applications," *Foundations and Trends® in Signal Processing*, vol. 7, no. 3–4, pp. 197–387, 2014.
[7] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.
[8] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
[9] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*, pp. 525–542, Springer, 2016.
[10] M. N. Bojnordi and E. Ipek, "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pp. 1–13, IEEE, 2016.
[11] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 27–39, IEEE Press, 2016.
[12] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.
[13] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," *arXiv preprint arXiv:1412.7024*, 2014.
[14] C. Louizos, M. Reisser, T. Blankevoort, E. Gavves, and M. Welling, "Relaxed quantization for discretized neural networks," *arXiv preprint arXiv:1810.01875*, 2018.
[15] "Bfloat16 floating point format.." https://en.wikipedia.org/wiki/Bfloat16_floating-point_format.
[16] "Intel xeon processors and intel fpgas.." https://venturebeat.com/2018/05/23/intel-unveils-nervana-neural-net-l-1000-for-accelerated-ai-training/.
[17] "Intel xeon and fpga lines." https://www.top500.org/news/intel-lays-out-new-roadmap-for-ai-portfolio/.
[18] "Nnp-l1000." https://www.tomshardware.com/news/intel-neural-network-processor-lake-crest,37105.html.
[19] "Google cloud.." https://cloud.google.com/tpu/docs/tensorflow-ops.
[20] "Tpu repository with tensorflow 1.7.0.." https://blog.riseml.com/comparing-google-tpuv2-against-nvidia-v100-on-resnet-50-c2bbb6a51e5e?gi=51a90720b9dd.
[21] J. V. Dillon, I. Langmore, D. Tran, E. Brevdo, S. Vasudevan, D. Moore, B. Patton, A. Alemi, M. Hoffman, and R. A. Saurous, "Tensorflow distributions," *arXiv preprint arXiv:1711.10604*, 2017.
[22] "Google. 2018-05-08. retrieved 2018-05-23. in many models this is a drop-in replacement for float-32.." https://www.youtube.com/watch?v=vm67WcLzfvc&t=2555.
[23] B. Feinberg, U. K. R. Vengalam, N. Whitehair, S. Wang, and E. Ipek, "Enabling scientific computing on memristive accelerators," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 367–382, IEEE, 2018.
[24] "Intel and micron produce breakthrough memory technology.." http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology.
[25] M. Cheng, L. Xia, Z. Zhu, Y. Cai, Y. Xie, Y. Wang, and H. Yang, "Time: A training-in-memory architecture for memristor-based deep neural networks," in *Proceedings*

of the 54th Annual Design Automation Conference 2017, p. 26, ACM, 2017.
[26] Y. Cai, T. Tang, L. Xia, M. Cheng, Z. Zhu, Y. Wang, and H. Yang, "Training low bitwidth convolutional neural network on rram," in *Proceedings of the 23rd Asia and South Pacific Design Automation Conference*, pp. 117–122, IEEE Press, 2018.
[27] Y. Cai, Y. Lin, L. Xia, X. Chen, S. Han, Y. Wang, and H. Yang, "Long live time: improving lifetime for training-in-memory engines by structured gradient sparsification," in *Proceedings of the 55th Annual Design Automation Conference*, p. 107, ACM, 2018.
[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.
[29] L. K. Hansen and P. Salamon, "Neural network ensembles," *IEEE transactions on pattern analysis and machine intelligence*, vol. 12, no. 10, pp. 993–1001, 1990.
[30] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "MagicâĂŤmemristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.
[31] S. Gupta, M. Imani, and T. Rosing, "Felix: Fast and energy-efficient logic in memory," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–7, IEEE, 2018.
[32] A. Siemon, S. Menzel, R. Waser, and E. Linn, "A complementary resistive switch-based crossbar array adder," *IEEE journal on emerging and selected topics in circuits and systems*, vol. 5, no. 1, pp. 64–74, 2015.
[33] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-based material implication (IMPLY) logic: design principles and methodologies," *TVLSI*, vol. 22, no. 10, pp. 2054–2066, 2014.
[34] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "Memristive switches enable stateful logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.
[35] B. C. Jang, Y. Nam, B. J. Koo, J. Choi, S. G. Im, S.-H. K. Park, and S.-Y. Choi, "Memristive logic-in-memory integrated circuits for energy-efficient flexible electronics," *Advanced Functional Materials*, vol. 28, no. 2, 2018.
[36] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny, "Vteam: A general model for voltage-controlled memristors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 8, pp. 786–790, 2015.
[37] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic design within memristive memories using memristor-aided logic (magic)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 635–650, 2016.
[38] M. Imani, S. Gupta, and T. Rosing, "Ultra-efficient processing in-memory for data intensive applications," in *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 6, ACM, 2017.
[39] A. Haj-Ali *et al.*, "Efficient algorithms for in-memory fixed point multiplication using magic," in *IEEE ISCAS*, IEEE, 2018.
[40] M. Imani, D. Peroni, Y. Kim, A. Rahimi, and T. Rosing, "Efficient neural network acceleration on gpgpu using content addressable memory," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1026–1031, IEEE, 2017.
[41] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 476–488, IEEE, 2015.
[42] A. Nag, R. Balasubramonian, V. Srikumar, R. Walker, A. Shafiee, J. P. Strachan, and N. Muralimanohar, "Newton: Gravitating towards the physical limits of crossbar acceleration," *IEEE Micro*, vol. 38, no. 5, pp. 41–49, 2018.
[43] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.
[44] A. Ghofrani, A. Rahimi, M. A. Lastras-Montaño, L. Benini, R. K. Gupta, and K.-T. Cheng, "Associative memristive memory for approximate computing in gpus," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 6, no. 2, pp. 222–234, 2016.
[45] F. Chollet, "keras." https://github.com/fchollet/keras, 2015.
[46] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.
[47] X. Dong, C. Xu, N. Jouppi, and Y. Xie, "Nvsim: A circuit-level performance, energy, and area model for emerging non-volatile memory," in *Emerging Memory Technologies*, pp. 15–50, Springer, 2014.
[48] D. Compiler, R. User, and M. Guide, "Synopsys," *Inc., see http://www. synopsys. com*, 2000.
[49] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.
[50] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.
[51] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaev, G. Venkatesh, *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.
[52] M. Drumond, T. Lin, M. Jaggi, and B. Falsafi, "End-to-end dnn training with block floating point arithmetic," *arXiv preprint arXiv:1804.01526*, 2018.
[53] D. Das, N. Mellempudi, D. Mudigere, D. Kalamkar, S. Avancha, K. Banerjee, S. Sridharan, K. Vaidyanathan, B. Kaul, E. Georganas, *et al.*, "Mixed precision training of convolutional neural networks using integer operations," *arXiv preprint arXiv:1802.00930*, 2018.
[54] C. De Sa, M. Leszczynski, J. Zhang, A. Marzoev, C. R. Aberger, K. Olukotun, and C. Ré, "High-accuracy low-precision training," *arXiv preprint arXiv:1803.03383*, 2018.
[55] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 17, IEEE Press, 2016.
[56] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 382–394, ACM, 2017.
[57] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, vol. 49, pp. 269–284, ACM, 2014.
[58] V. Aklaghi, A. Yazdanbakhsh, K. Samadi, H. Esmaeilzadeh, and R. Gupta, "Snapea: Predictive early activation for reducing computation in deep convolutional neural networks," ISCA, 2018.
[59] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. W. Fletcher, "Ucnn: Exploiting computational reuse in deep neural networks via weight repetition," *arXiv preprint arXiv:1804.06508*, 2018.
[60] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, *et al.*, "C ir cnn: accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 395–408, ACM, 2017.
[61] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, *et al.*, "Can fpgas beat gpus in accelerating next-generation deep neural networks?," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 5–14, ACM, 2017.
[62] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 243–254, IEEE, 2016.
[63] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 609–622, IEEE Computer Society, 2014.
[64] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," *arXiv preprint arXiv:1805.03718*, 2018.
[65] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 288–301, ACM, 2017.
[66] M. N. Bojnordi and E. Ipek, "The memristive boltzmann machines," *IEEE Micro*, vol. 37, no. 3, pp. 22–29, 2017.
[67] M. Imani, M. Samragh, Y. Kim, S. Gupta, F. Koushanfar, and T. Rosing, "Rapidnn: In-memory deep neural network acceleration framework," *arXiv preprint arXiv:1806.05794*, 2018.
[68] S. Gupta, M. Imani, H. Kaur, and T. S. Rosing, "Nnpim: A processing in-memory architecture for neural network acceleration," *IEEE Transactions on Computers*, 2019.
[69] M. Imani, S. Gupta, and T. Rosing, "Genpim: Generalized processing in-memory to accelerate data intensive applications," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1155–1158, IEEE, 2018.
[70] S. Salamat, M. Imani, S. Gupta, and T. Rosing, "Rnsnet: In-memory neural network acceleration using residue number system," in *2018 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–12, IEEE, 2018.
[71] M. Imani, A. Rahimi, D. Kong, T. Rosing, and J. M. Rabaey, "Exploring hyperdimensional associative memory," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 445–456, IEEE, 2017.
[72] Y. Kim, M. Imani, and T. Rosing, "Orchard: Visual object recognition accelerator based on approximate in-memory processing," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 25–32, IEEE, 2017.
[73] M. Zhou, M. Imani, S. Gupta, and T. Rosing, "Gas: A heterogeneous memory architecture for graph processing," in *Proceedings of the International Symposium on Low Power Electronics and Design*, p. 27, ACM, 2018.
[74] M. Zhou, M. Imani, S. Gupta, Y. Kim, and T. Rosing, "Gram: graph processing in a reram-based computational memory," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pp. 591–596, ACM, 2019.
[75] M. Imani, S. Gupta, S. Sharma, and T. Rosing, "Nvquery: Efficient query processing in non-volatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.