NNPIM: A Processing In-Memory Architecture for Neural Network Acceleration

Saransh Gupta, *Student Member, IEEE,* Mohsen Imani, *Student Member, IEEE,*
Harveen Kaur, and Tajana Rosing, *Fellow, IEEE*

***Abstract*—Neural networks (NNs) have shown great ability to process emerging applications such as speech recognition, language recognition, image classification, video segmentation, and gaming. It is therefore important to make NNs efficient. Although attempts have been made to improve NNs' computation cost, the data movement between memory and processing cores is the main bottleneck for NNs' energy consumption and execution time. This makes the implementation of NNs significantly slower on traditional CPU/GPU cores. In this paper, we propose a novel processing in-memory architecture, called NNPIM, that significantly accelerates neural network's inference phase inside the memory. First, we design a crossbar memory architecture that supports fast addition, multiplication, and search operations inside the memory. Second, we introduce simple optimization techniques which significantly improves NNs' performance and reduces the overall energy consumption. We also map all NN functionalities using parallel in-memory components. To further improve the efficiency, our design supports weight sharing to reduce the number of computations in memory and consecutively speedup NNPIM computation. We compare the efficiency of our proposed NNPIM with GPU and the state-of-the-art PIM architectures. Our evaluation shows that our design can achieve 131.5× higher energy efficiency and is 48.2× faster as compared to NVIDIA GTX 1080 GPU architecture. Compared to state-of-the-art neural network accelerators, NNPIM can achieve on an average 3.6× higher energy efficiency and is 4.6× faster, while providing the same classification accuracy.**

***Index Terms*—Non-volatile memory, Processing in-Memory, Neural Networks**

I. Introduction

The emergence of *Internet of Things* (IoT) has significantly increased the size of application data sets required to be processed [1]. These large data sets encourage the use of algorithms which automatically extract useful information from them and artificial neural networks for this purpose are being investigated widely. In particular, deep neural networks (NNs) demonstrate superior effectiveness for diverse classification problems, image processing, video segmentation, speech recognition, computer vision and gaming [2]–[5]. Although many NN models are implemented on high-performance computing architectures, such as parallelizable GPGPUs, running neural networks on the general purpose processors is still slow, energy hungry, and prohibitively expensive.

Earlier work proposed several FPGA-based and ASIC designs [6]–[9] to accelerate neural networks. However, these techniques pose a critical technical challenge due to the cost of data movement, since they require dedicated memory blocks, e.g., SRAM, to store the large amount of network weights and input signals. Prior work exploits several techniques to optimize the enormous cost, yet the memory still takes up to 90% of the total energy consumption to perform NN inference tasks even in the ASIC design [7].

Processing in-memory (PIM) is a promising solution to address the data movement issue by implementing logic within memory [10]–[16]. Instead of sending a large amount of data to the processing cores for computation, PIM performs a part of computation tasks, e.g., bit-wise computations, inside the memory, thus avoiding the memory access bottleneck and accelerating the application performance significantly. Some research work proposes PIM-based neural network accelerators which keep the input data and trained weights inside memory [10], [17]–[19]. For example, work in [19] shows that memristor devices can model the computations in each neuron. They store trained weights of each neuron as device resistance values and pass current representing the input values in a way similar to spiking-based neuromorphic computing. They only support two functionalities in memory, addition and multiplication, while other important operations such as activation functions are implemented using CMOS-based logic, which would make the fabrication expensive. In addition, Analog to Digital Converters (ADCs) and Digital to Analog Converters (DACs) used by their design do not scale along with memory device technology and take the majority of power (61%). In this context, the ADC/DAC-based computation would not be an appropriate solution to design PIM-based NN accelerators.

In this paper, we propose a novel NN accelerator, called neural network processing in-memory (NNPIM), which significantly reduces the overhead of data movements while supporting all the NN functionalities completely in memory. To realize such computation, our design first analyzes computation flows of a NN model and encodes key NN operations for a specialized PIM-enabled accelerator. The proposed NNPIM supports three layers popularly used for designing a NN model: fully-connected, convolution, and pooling layer. We divide the computation tasks of the networks into four operations, multiplication, addition, activation function, and pooling. Our accelerator supports all of these operations inside a crossbar memory. Our evaluation shows that our design can achieve 131.5× higher energy efficiency and is 48.2× faster as compared to NVIDIA GTX 1080 GPU architecture. Compared to state-of-the-art neural network accelerator, NNPIM can achieve on average 3.6× higher energy efficiency and is 4.6×

All authors are with the Department of Computer Science and Engineering, University of California San Diego, La Jolla, CA, 92093.
E-mail: {sgupta, moimani, hak133, tajana}@ucsd.edu

faster, while providing the same classification accuracy.

II. Background and Related Work

A NN model consists of multiple layers which have multiple neurons. These layers are stacked on top of each other in a hierarchical formation, so each layer takes the output of previous layer as input and forwards its output to the next layer. In this paper, we focus on three types of layers that are most commonly utilized in practical neural network designs: (i) convolution layers, (ii) fully connected layers, and (iii) pooling layers. In neural network, each neuron takes a vector of inputs from neurons of the preceding layer $X = \langle X_0, \cdots, X_n \rangle$, then computes its output as follows:

$$\varphi(\sum_{i=1}^{n} W_i X_i + b)$$

where W_i and X_i correspond to a weight and an input respectively, b is a bias parameter, and $\varphi(.)$ is a nonlinear activation function. Prior to the execution of NNs, parameters W_i and b are learned in a training process. For inference, the pre-trained parameters are used to compute the outputs of each neuron, called *activation units*. A neuron produces one activation unit based on two main operations, the weighted accumulation, i.e. $\sum W_i X_i$, and the activation function, i.e. $\varphi(.)$. By processing all the computation through the layers, also known as *feed-forward* procedure, it produces multiple outputs which are used for the final prediction. Two basic operations are associated to the weighted accumulation: multiplication and addition. Thus, the key technical challenge is how to reduce the size of two input sets.

Modern neural network algorithms are executed on different types of platforms such as GPU, FPGAs and ASIC chips [6], [20]–[23]. Prior works attempt to fully utilize existing cores to accelerate neural networks. For example, for a neural network-based image classification, GPU showed high performance improvement (up to two orders of magnitudes) over CPU-based implementation [21]. Several research works show hardware-based accelerators can further improve the efficiency of neural networks. DaDianNao proposed a series of ASIC designs which accelerate neural networks [24]. To fully utilize data locality, they employed high-bandwidth on-chip eDRAM blocks instead of using SRAM-based synapses [25]. Work in [26] proposed parallel CNN accelerators which use GPGPUs, FPGAs or ASICs and work based on stochastic computing. In their design, the main computation still relies on CMOS-based cores, thus suffering from the data movement issue. In contrast, the proposed NNPIM accelerator does not rely on any additional processing cores.

The capability of non-volatile memories (NVMs) to act as both storage and a processing unit has encouraged research in processing in-memory. Resistive RAM (RRAM) is one such memory which stores data in the form of its resistance [12]. Many logic families have been proposed which implement basic logic operations in-memory. Memory-Aided Logic (MAGIC) [27] is one of the many proposed logic families for RRAM. It uses the resistive nature of ReRAM to implement logic purely in memory without the need for any special sense amplifiers or requirement of a unique memory architecture. MAGIC implements the logic NOR operation in crossbar memory and uses it as the basis for other operations. Prior work also tried to use RRAMs to design PIM-based neural network accelerators [18], [19]. These designs use multi-level memristor devices which perform the multiplication and addition operations by converting digital values to analog signals. However, these approaches have potential design issues. Their designs require analog and digital-mixed circuits, e.g., ADC and DAC, which do not scale with CMOS technology. The neural network operations other than multiplication and addition still rely on the CMOS-based logic, increasing fabrication costs. In contrast, we design the NNPIM accelerator which supports all neural network computations inside the memory without using costly ADC/DAC blocks.

III. PIM-based Neural Network Acceleration

A. PIM for Neural Network

Processing in-memory supports essential functionalities among different memory rows. These operations should be general enough to benefit many applications. Neural network computation is based on a few basic operations, so executing them in-memory can allow us to run whole application inside a memory. This would reduce data movement issue and accelerate any network locally in memory.

In inference, neural networks use a combination of convolution, pooling, and fully connected layers to process or classify the data. There are two types of data in neural networks: **(i)** a large number of trained weights, which we call them network model and **(ii)** the input data which is processed by the network. The main computation in neural network involves processing the input data over network using the trained weights. It leads to several computations between weights and inputs. The goal of PIM is to locally perform operations between these inputs and weights inside a memory block, such that there is no need to send data up to processor. To support all the required operations in memory, we design a PIM architecture which can perform addition, multiplication, activation function, and pooling locally in memory. These operations are managed inside a memory using simple controllers.

The memory architecture used in this work supports the following functions on the same hardware:

Addition/Multiplication: Our design can execute the addition of three data values, in memory, by activating their corresponding rows. If more values have to be added at the same time, our design implements addition in a tree structure. The multiplication inside memory is performed in a similar way, by generating all possible partial products and adding them in parallel in memory. We talk about details of hardware implementation in Section III-B.

Activation Function: Traditionally, *Sigmoid* function has been used as an activation function [28]. This function is defined as: $S(x) = 1/1 + e^{-x}$. Implementing this functionality in memory requires modeling exponential operations. Our design can handle this operation by using the Taylor expansion of the *Sigmoid* function and considering the first few terms to approximate the *Sigmoid* function. The Taylor

expansion only consists of addition and multiplication. We can easily implement any function in memory as long as it is representable by Taylor expansion and the more terms we consider in Taylor expansion, the better the model is for activation functions. Prior work showed that it is not necessary to use *Sigmoid* as an activation function. Instead, using simple "Rectified Linear Unit" clamped at a certain point (e.g. X=a) could provide similar or better accuracy than *Sigmoid*. In that case, the activation function can be implemented using a single comparator which checks if input X surpasses a value a. Note that in case of rectified linear unit, activation function can be processed simply inside a controller.

Pooling: Our hardware implements in-memory pooling using nearest search operation. PIM stores the output of convolution layer inside a memory block with nearest search capability. The NNPIM pooling unit can be logically viewed as a lookup table, where each data is present in a separate row of the crossbar memory. Further, the data within a pooling window may or may not be present in consecutive rows. For each pooling window, our search-based pooling activates the corresponding rows, i.e. charges the wordlines, and then applies voltages at the bitlines corresponding to MIN/MAX operation [29]–[31]. The wordline which discharges first is the output of the pooling window. To find the maximum value, our design searches for a row with the closest distance (maximum similarity) to inf value. This inf value in hardware is the maximum value which can be represented by hardware. Using this block, we can search for the MAX value among the selected rows inside the memory. Similarly, the MIN pooling can be implemented by searching for the smallest value in lookup table ($-\inf$).

The implementation of average pooling is equivalent to addition operation followed by division. However, division is a difficult operation in memory. To avoid the use of a special division accelerator/circuit, the weights in the layer preceding the pooling layer are normalized before being encoded in NNPIM. Hence, the average operation simply becomes an addition operation. The data corresponding to a pooling window is added using data intensive addition operation discussed in Section III-B.

B. In-Memory Addition/Multiplication

In-memory operations are in general slower than the corresponding CMOS based implementations. This is worsened by the serial nature of previously proposed PIM techniques. In this section, we propose a fast adder for memristive memories, which introduces parallelism in addition and optimizes its latency. Our design is based on the idea of carry save addition (CSA) and adapts it for in-memory computation. We further use a Wallace-tree inspired structure to leverage the fast *3:2* reduction of our new in-memory adder design. The implementation of this new adder is made feasible by the configurable interconnects which we previously proposed in [12].

We use MAGIC `NOR` [27] to execute logic functions in memory due to its simplicity and independence of execution from data in memory. An execution voltage, V_0, is applied to the bitlines of the inputs (in case of `NOR` in a row) or wordlines of the outputs (in case of `NOR` in a column) in order to evaluate `NOR`, while the bitlines of the outputs (`NOR` in a row) or wordlines of the inputs (`NOR` in a column) are grounded. The work in [32] extends this idea to implement adder in a crossbar. It executes a pattern of voltages in order to evaluate sum (S) and carry (C_{out}) bits of 1-bit full addition (inputs being A,B,C) given by

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

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

Here, C_{out} is realized as a series of 4 `NOR` operations while S is obtained by 3 `NOT` operations (evaluation of A', B', and C') followed by 5 `NOR` operations. A `NOT` operation is implemented as a `NOR` operation with 1 input. Extending this 1-bit addition to N-bit addition requires propagating carry between different bits, consuming N times the latency of 1-bit addition. We define a cycle time ($= 1.1ns$) as the time taken to implement one `MAGIC NOR` operation. This design takes $12N+1$ cycles to add two N-bit numbers.

The design in [32] is good for small numbers but as the length of numbers increases, time taken increases linearly. A $N \times M$ multiplication requires addition of M partial products, each of size N bits, to generate a $(N+M)$-bit product. This takes $(M-1)\cdot(12(N-1)+1)$ cycles to obtain the final product.

Figure 1 describes our fast addition which we implement in memory using MAGIC NOR. Figure 1(a) shows carry save addition. Here, $S1[n]$ and $C1[n]$ are the sum and carry-out bits, respectively of 1-bit addition of $A1[n]$, $A2[n]$, and $A3[n]$. The 1-bit adders do not propagate the carry bit and generate two outputs. This makes the n additions independent of each other. The proposed adder exploits this property of CSA. Since, `MAGIC` execution scheme doesn't depend upon the operands of addition, multiple addition operations can execute in parallel if the inputs are mapped correctly. The design utilizes the memory unit proposed in [12], which supports shifting operations, to implement CSA like behaviour. The latency of this *3:2* reduction, 3 inputs to 2 outputs, is same as that of a 1-bit addition (*i.e.*, 13 cycles) irrespective of the size of operands. The two numbers can then be added serially, consuming $12N+1$ cycles. This totals to $12N+14$ cycles while the previous adder would take $24N-22$ cycles. The difference increases linearly with the size of inputs.

Figure 1(b) shows the Wallace-tree inspired structure we use to add multiple numbers (9 n-bit numbers in this case). At every stage of execution, the available addends are divided in groups of three. The addends are then added using a separate adder (as described above) for each group, generating two outputs per group. The additions in the same stage of execution are independent and can occur in parallel to each other. Our configurable interconnect, introduced in [12], arranges the outputs of this stage in groups of three for addition in the next stage. This structure takes a total of four stages for *9:2* reduction, having the same delay as that of four 1-bit additions. At the end of the tree structure we are left with two $(N+3)$-bit numbers which can then be added serially. The tree-structured addition reduces the delay substantially

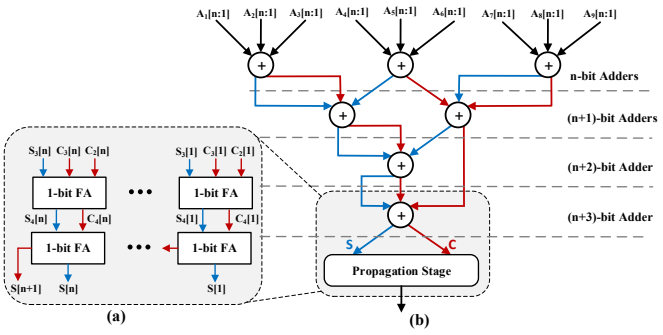


Fig. 1. (a) Carry save addition (b) Tree structured addition of 9 n-bit numbers

as carry propagation happens only in the last stage, unlike the conventional approach where carry is propagated at every step of addition. Although this speed up comes at the cost of increased energy consumption and number of writes in memory, it is acceptable because the latency is reduced by large margins as shown in Section V.

To extend the idea to multiplication, our design in [12] generates partial products by looking at one of the input operands and copy-shift the other one. We then use the fast addition explained above to add the partial products together. This provides a huge latency improvement over the previous in-memory adder designs. We further optimize the massive amount of shift operations involved in such computations. Our design divides the crossbar memory into multiple data and processing blocks [12] which we describe later in Section IV-D. A data-processing block pair contains a configurable interconnect between the two blocks which accelerates shifting and enables shifting multiple bits in parallel [12].

C. In-Memory Search Operation

As discussed in Section III-A MIN/MAX pooling involves searching the memory block for the data nearest to $-\inf/\inf$. An efficient way to perform these search operation is to implement it in-memory. We utilize the inherent characteristics of capacitors to discharge differentially to search for the nearest (least hamming distance) data. We also use a voltage application technique which enables efficient nearest data search based on binary distance. We apply this technique to search for the minimum or maximum value among the outputs of convolution layers.

For a search in conventional CAM, the match-lines (MLs) are pre-charged to V_{dd} and then bitlines are driven with V_{dd} or 0 depending upon the input query. The MLs of rows with more number of matches discharge earlier. The line to discharge first is the one with minimum mismatch with the input query. To give binary weight to the bits, the authors in [30] modify the bitline driving voltage. Suppose a stage contains m bits $(m-1:0)$. The bitlines which were earlier driven with V_{dd} are now driven with a voltage $V_i = V_{dd}/2^{(m-1-i)}$ where i denotes the index of a bit in the stage. Here, a bit with higher index are driven by a higher voltage, giving it more weight than the lower bits. Hence, a match in the most significant bit results in faster ML discharging current than lower indices. We exploit this difference to implement MIN/MAX pooling as done in [30].

IV. NNPIM Design

A. NNPIM Overview

In conventional systems, sensors are connected to the processing system. The output from the sensors is placed into local storage (often NVM). At the time of processing, the core reads the data stored in memory sequentially. Once the data is processed, the outputs are stored back in the memory. NNPIM acts as an accelerator accompanying the processor. It acts as a secondary memory such that the output from sensors, is sent to NNPIM instead of the main memory. Now, since the model is already stored in NNPIM, the sensor data can be processed in NNPIM without involving the data transfers in conventional systems. The output of the network is generated and stored in NNPIM and can be sent to the processor when requested.

As described before, an inference task in neural network involves multiplying inputs with the weights, which are calculated during the training phase. Once a network is trained, the weights remain constant and do not change over different inference tasks. The previously proposed hardware designs to accelerate neural networks do not exploit this property of neural networks. In such cases, multiplication with fixed weights is computationally as expensive as that with variable weights.

NNPIM uses this fixed nature of weights to reduce the complexity of in-memory neural network multiplications. Instead of using the weights directly, NNPIM breaks down the weights into simpler factors. These factors are chosen such that multiplying a number with them just requires a shift and add/subtract operation. Hence, instead of exhaustively generating all the partial products and adding them, we rely on the fixed nature of weights to pre-process them and calculate their "multiplication-friendly" factors. All these computations utilize the PIM operations proposed in Section III.

A neural network usually involves a large number of weights. Using this large number of weights restricts the enhancements which in-memory processing can provide. We realize that the memory requirement and energy consumption of NNPIM depend on the number of weights. Hence, we use weight sharing to reduce the number of unique weights in each neuron [17], [33]. Since all the computations in NNPIM happen in-memory, we design NNPIM such that this reduction in weights directly results in a decrease in the number of memory blocks required for computations.

B. Weight Clustering in NNPIM

The conventional NN requires a large number of multiplications. We leverage shared weights to reduce number of operations, *i.e.* multiple inputs of each neuron share the same value, however, a naive implementation of weight sharing can result in undesirable loss of accuracy. We devise a greedy algorithm to select the near optimized shared weights that reduce the loss of accuracy; instead of applying shared weights to the already trained NN, we train the NN in a way that weight sharing does not impose much loss of accuracy.

The weights of each layer are fixed in the inference phase; in order to share the weights, the clustering algorithm is applied on the fixed weights. Assuming that a fully-connected layer maps N neurons into M outputs, the corresponding matrix $W_{M\times N}$ is clustered once and a single set of weights are generated for the whole matrix. For convolution layers, the weights corresponding to different output channels are clustered separately: a convolution layer mapping N channels into M channels using a weight tensor $W_{h\times h\times N\times M}$ is divided into M different tensors and each tensor is clustered separately, resulting in M different weights.

After clustering, each weight is replaced by their closest centroids [33]. The objective of clustering is to minimize the within cluster sum of squares (WCSS):

$$\min_{c_{i1},\ldots,c_{iN_{clusters}}} (WCSS = \sum_{k=1}^{N_{clusters}} \sum_{W_{ij}^{l}\in c_{ik}} ||W_{ij}^{l} - c_{ik}||^2) \quad (2)$$

where $C = \{c_{i1}, c_{i2}, \ldots, c_{iN_{clusters}}\}$ are the cluster centroids. We use K-means algorithm for clustering.

Weight clustering essentially finds the best matches that can represent this distribution, and replaces all parameters with their closest centroids. Weight clustering is often accompanied by some degree of additive error, $\Delta e = e_{clustered} - e_{baseline}$.

To compensate for this error, our algorithm retrains the neural network based on the new weight constraints. After each retraining, our design again clusters the weights and estimates the quality of the classification using the new cluster centers. The procedure of weight clustering and retraining continues until the estimated error becomes smaller than a desired level. Otherwise the retraining procedure stops after a pre-specified number of epochs. Figure 2 shows the accuracy of neural network for MNIST dataset during different retraining iterations. The result shows that retraining improves the classification accuracy by finding a suitable clusters for each neuron weights.

One major advantage of weight sharing is that it can significantly reduce the number of required multiplications. Each neuron in neural network multiplies several input data, say n, with pre-stored weights. Therefore, each neuron requires to multiply n input-weight pairs. Using weight clustering, the number of distinct weights in each neuron can be reduced to k, where $k << n$. Instead of multiplying all input-weight pairs, we can simply add all inputs which share the same weight and finally multiply the result of addition with the weight value. This method reduces the number of multiplications in each neuron from n to k. This significantly accelerates the NNPIM computation, since in PIM the multiplication performs much slower than addition. Moreover, our hardware enables fast addition of multiple input vectors in-memory. Hence, the inputs corresponding to the same weight can first be added together using carry save addition. Then, the result can be multiplied with the weight. In other words, multiple multiplications are broken down into a large addition and a multiplication. In this way, we reduce the number of computations required as well as the complexity of operations involved.

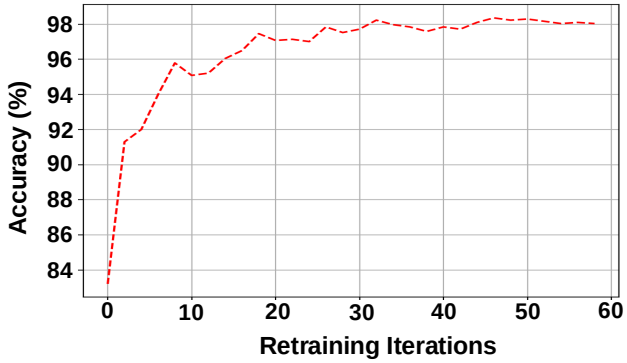


Fig. 2. An example of the MNIST classification accuracy during different retraining iterations when the NN weights are shared into eight cluster centers.

C. NNPIM Multiplication

The multiplier in Section III-B performs exhaustive binary multiplication. It generates a partial product for each '1' present in the multiplier and performs addition. Although this approach is general and works for all applications but it can lead to unnecessary latency overheads in certain cases. For example, multiplication by 255 (b11111111) would require generation of 8 partial products, corresponding to each '1', and their subsequent addition. The same operation can executed by multiplying by 256, i.e. shifting by 8 bits, and then subtracting the multiplicand from the obtained result.

Bernstein algorithm [34] factorizes the constant multiplier into factors which are a power of 2 or a power of 2 ± 1. It uses branch and bound based search pruning and finds the factors based on a formulation for their costs. Figure 3 gives an example of how the algorithm can reduce the number of operations. In this case, binary multiplication takes 6 instructions whereas the factor-based multiplication takes only 4 instructions. The binary method is the worst case factorization which can be obtained using the algorithm.

Using this algorithm involves finding suitable factors. It can be time consuming and may add unwanted latency if the operands change frequently. However, such an algorithm can be useful if one of the operands is constant. In that case, the constant operand can be factorized once and these factors can be referenced every time the constant is involved in multiplication. This makes such factorization suitable for neural networks, where the weights are always constant and only the inputs are variable. NNPIM exploits this property by storing the factors of the weights and using these factors for computations. We now discuss two ways in which we use Bernstein algorithm to improve computations in neural networks. One approach aims to minimize the energy consumption of the design while the other approach presents a latency-optimized technique.

Energy-Optimized NNPIM: The hardware in Section III-B utilizes carry save addition to reduce the latency of multiplication. However, in order to minimize the propagation of carry and reduce the latency, it implements a large number of partially redundant parallel operations. This consumes significant amount of energy. A naive energy-efficient design would

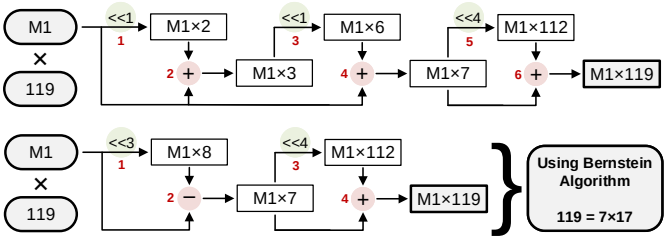


Fig. 3. Example of Bernstein's Algorithm

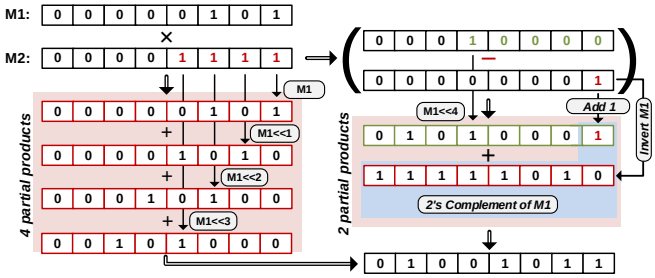


Fig. 4. Generating the partial products in latency-optimized NNPIM

process all partial products serially, adding two at a time. Such a design is intuitive but does not exploit the constant operands in neural networks. The inference phase of neural networks involves multiplication of many input vectors with weights obtained from the training phase and fixed during inference. This phase is defined by multiplication of variables, i.e. input vectors, with constants, i.e. weights, making it a suitable application for Bernstein algorithm. We can accelerate the testing phase by factorizing the weights and using these factors instead of actual weights for computation. For the example discussed before, binary implementation requires 6 serial shift or add operations, while NNPIM only requires 4 serial shift, add, or subtract operations.

Latency-Optimized NNPIM: The above approach based on Bernstein algorithm is perfect when the total energy consumption of the design is the major concern. Bernstein algorithm reduces the number of operations required but does not necessarily accelerate the overall in-memory processing. In carry save addition, carry is propagated only in the end to minimize the time taken to compute the final product. Breaking the weights into smaller factors requires the computation of multiple intermediate products to achieve the final output. Factorizing 119 into 7 and 17 leads to two carry propagation stages instead of one. Since carry propagation is the bottleneck in the multiplication process, many such operations make it impossible to gain time from the reduced number of instructions.

In order to reduce latency, NNPIM uses an adder structure similar to that in Section III-B while taking into consideration the constant operand in neural networks. It exploits the fact that in binary representation, a sequence of 1s, for example $b00011111$, can be written as a difference of two shifted 1s, i.e. $b00011111 = b00100000 - b00000001$. Instead of generating multiple shifted partial products, NNPIM generates only two partial products. It is similar to Booth's recoding but differs in the way it is implemented in memory. Instead of applying the operations serially as in the case of Booth's recoding, we modify subtraction to make it suitable for parallel execution. To maintain uniformity by executing only addition instructions, NNPIM simplifies subtraction as shown in Figure 4. In the figure, generation of 2's complement of M1 involves inversion of M1 and addition of 1. Inversion is a single MAGIC NOR step, where all the bits can be inverted in parallel. Moreover, 1 is added to the shifted version of M1. The LSB of the shifted M1 is always 0, converting the addition of 1 ($Add1$ in Figure 4) to a simple SET operation on LSB. The two partial products can then be added normally as in case of a conventional multiplication.

The above technique may not be applicable directly since it is highly unlikely for the weights to always be a sequence of 1's. Hence, we propose a modified version of Bernstein algorithm which is suitable for carry save addition. Instead of breaking down the constant operands into smaller factors, we break them down into chunks of continuous 1s as shown in Figure 5. These smaller parts of constants are then reduced using the same concept as discussed above. Since this approach generates two partial products for a series of 1s, reduction is done only when there are more than 2 consecutive 1's. In the example shown in Figure 5, the binary execution would require 11 partial products, but the optimized one generates just 6 partial products. Unlike the factors obtained by Bernstein algorithm, these partial products are added in parallel using carry save addition. This reduces the latency of NNPIM significantly.

Figure 6 compares the energy-optimized and latency-optimized approaches for 32-bit multiplication. The result shows that energy-optimized approach can provide 2.3× energy efficiency as compared to latency-optimized approach, while the latency-optimized can be 1.8× faster.

D. NNPIM Architecture

Figure 7 details the architecture of the proposed NNPIM. Figure 7a shows the overview of the architecture of NNPIM. Each neuron in NN has a corresponding computation unit. Each of this unit is made up of several computation sub-units, one for every weight corresponding to the inputs of the neuron. Every unit has an additional computation sub-unit which is responsible for accumulation of all the multiplication results for a neuron and implementing the activation function, which we call activation unit. The outputs from all the activation units are sent to the pooling unit. In case pooling is not required, the output of activation units is used directly for the next layer.

NNPIM is entirely based on crossbar memory. The crossbar structure is divided into smaller blocks, upper blocks and lower blocks as shown in Figure 7b. All these blocks are architecturally and functionally the same as described in [12]. Each computation sub-unit as well as activation unit is one such block pair (pair of one upper and one lower block). All the computations for a weight are executed in the corresponding block pair. Hence, a neuron with N weights will have N computation sub-units which implies N block pairs. The

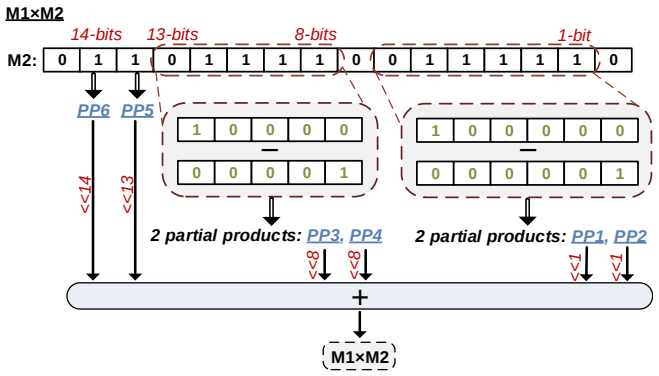


Fig. 5. Optimizing NNPIM by reducing the complexity of weights

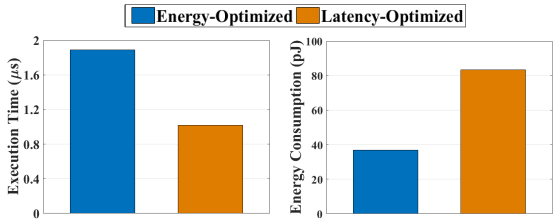


Fig. 6. Execution time and energy consumption of 32-bit NNPIM multiplication in energy and latency-optimized.

major peripheral circuitry including the bitline and wordline controllers, sense amplifiers, row/column decoders, etc. are shared by all these pairs.

Each upper block is connected to the corresponding lower block via configurable interconnects as shown in Figure 7c. These interconnects are collection of switches, similar to a barrel shifter, which connects the bitlines of the two blocks. b_n and b'_n are bitlines coming into and going out of the interconnect respectively. The select signals, s_n control the amount of shift. These interconnects can connect cells with different bitlines together. For example, they can connect $b_n, b_{n+1}, b_{n+2}, \ldots$ incoming bitlines to, say, $b'_{n+4}, b'_{n+5}, b'_{n+6}, \ldots$ outgoing bitlines, respectively, hence enabling the flow of current between the cells on different bitlines of blocks. This kind of a structure makes the otherwise slow shifting operations energy efficient and fast, having the latency same as that of a normal copy operation. It is important because neural networks involve large number of shift operations (mainly due to multiplication), which could be a bottleneck if not dealt at the hardware level.

All the outputs of multiplication for a neuron are accumulated and Taylor expanded activation function is implemented in the activation unit, which is made up of the same sub-unit as described above. The outputs of all these units are sent to the pooling unit. This pooling unit is a usual crossbar memory which doesn't require splitting the memory into multiple blocks. The pooling unit works on the in-memory search operations described in Section III-C. The outputs from all the activation units are written and the outputs closest to $+inf/-inf$ are selected for MAX/MIN pooling.

In a general purpose implementation, the weights would be stored in memory and the inputs would get multiplied with the stored weights in parallel in different blocks. However, such an architecture will not be able to take advantage of the optimizations proposed in the previous sections. NNPIM uses a control-store architecture, where a control word for a block is defined by a shared operand and the corresponding local control vector (CV). Instead of storing the actual fixed weights in the memory, we pre-program the control words in the memory. These control words are optimized based on the techniques proposed before. The memory unit loads a control word and implements the operation without worrying about the actual weights.

The shared controller for the bitline and wordline, takes in 2-bit operands as shown in Figure 7b. Each operand, detailed in Figure 7d, corresponds to a specific function required by NNPIM for computations. Each pair of upper and lower blocks in our architecture has an independent shift controller which governs the bit shifts between the two blocks. The shift controller is a simple circuit which activates a particular select line depending upon the control vector sent to it. The control vector has two fields: (i) active flag which indicates whether the shift controller is active in that cycle and (ii) a 5-bit field indicating the amount of shift. A computational unit has a common shared operand list, while each sub-unit (i.e. each block pair) has its own CV list. A memory with N block pairs has N configurable interconnects and hence, N shift controllers. Each operand sent to the shared controller has a corresponding control vector for each shift controller. Our architecture enables independent shifts among different pairs of blocks while introducing very little overhead as shown in Section V.

Example: Figure 8 shows sample execution of two NNPIM multiplications in parallel, $In1 \times W1$ and $In2 \times W2$. After applying the optimization described in Section IV-C, the first multiplication results in 5 partial products while the second multiplication results in 6 partial products. The partial products generation by a shift and subtraction (i and ii in Figure 8) take three operations each. Here, in order to reduce the number of operations, the shifts before and after the subtraction are combined together. Also, the last operation in the example is not required by $W1$. So, the enable bit in the control vector for $W1$ is set to zero.

E. In-Memory Parallelism

NNPIM uses a blocked memory structure as shown in Figure 7b. Here, each block processes computation corresponding to one weight. Since each block pair in NNPIM has a shift controller, all these blocks can independently implement multiplication in parallel and computation for multiple weights can happen simultaneously. The number of computations possible in parallel directly effect the number of neurons that can be processed in parallel. This is limited by the size of the memory available. Assume that our memory allows for 2k block pairs. In a network where each neuron has 512 weights corresponding to 512 inputs, our memory can implement just 4 (=2k/512) neurons in parallel. This can be a bottleneck in large networks.

Weight sharing turns out to be useful in such cases as it restricts the number of unique weights for each neurons,

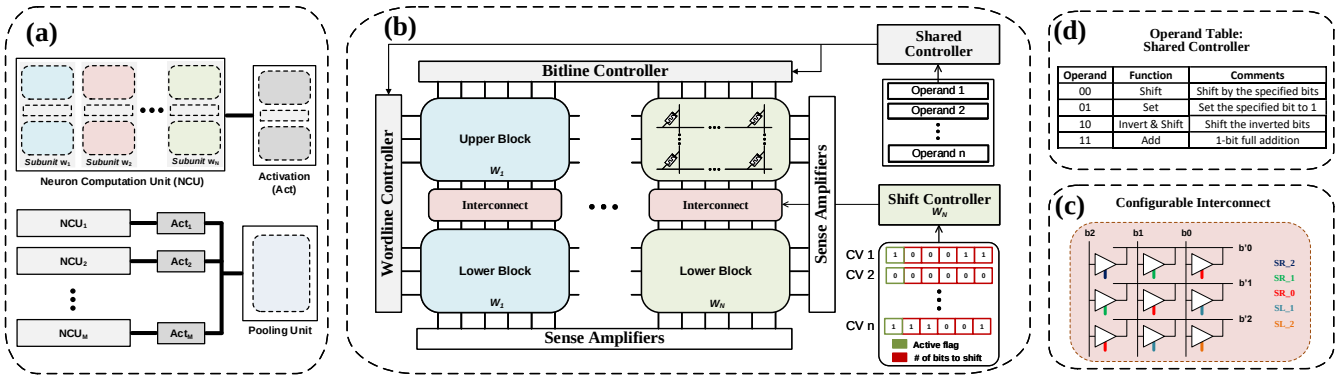


Fig. 7. Architecture overview of the proposed NNPIM. (a) Overall view of neural network implementation in-memory; (b) in-memory implementation of neuron; (c) circuit for configurable interconnect; (d) functions used in NNPIM.

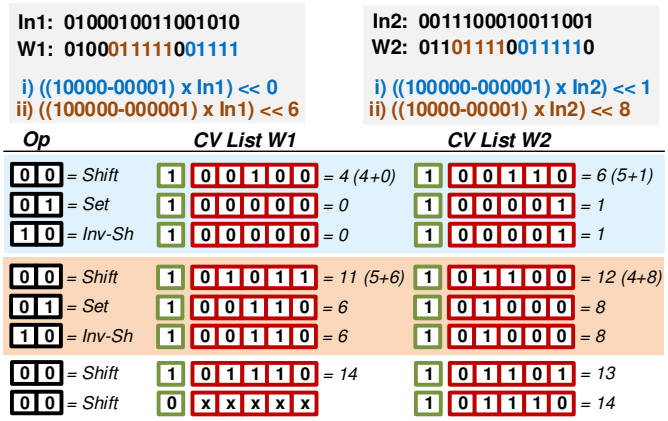


Fig. 8. Operands and control vectors for two parallel NNPIM multiplications.

thereby enabling the execution of more neurons in parallel. For the case discussed above, the number of neurons possible to be executed in parallel increases from 4 to 32 when the number of weights are restricted to 64. This further increases to 64, 128, and 256 when the number of weights are restricted to 32, 16, and 8 respectively. More the number of neurons executed in parallel, lesser is the overall latency of the network. Hence, weight sharing not only reduces the number of computations but also increases the overall performance of the network as further verified in Section V.

V. Experimental Results

A. Experimental Setup

We designed the NNPIM framework support, which retrains NN models for the accelerator configuration, in C++ while exploiting two back-ends, Scikit-learn library [35] for clustering and Tensorflow [36] for the model training and verification. For the accelerator design, we use Cadence Virtuoso tool for circuit-level simulations and calculate energy consumption and performance of all the NNPIM memory blocks. The NNPIM controller has been designed using System Verilog and synthesized using Synopsys Design Compiler in 45nm TSMC technology. The sense amplifier is similar to that used in [12]. Each sense amplifier reads one bit with latency and energy consumption of 150ps and 9.1fJ respectively. We use VTEAM memristor model [37] for our memory design simulations with R_{ON} and R_{OFF} of $10k\Omega$ and $10M\Omega$ respectively [12], [38]. The reduction in the ratio of Ron and Roff affects NNPIM performance. A smaller ratio would increase the delay of MAGIC NOR. However, for the device model used in the paper, the increase in delay is negligible when the ratio is 1:100 but increases more than 3x for the ratio 1:10 [32]. We compare the proposed NNPIM accelerator with GPU-based DNN implementations, running on NVIDIA GPU GTX 1080. The performance and energy of GPU are measured by the `nvidia-smi` tool. We used a batch size of 16 on GPU and for all tested applications, and the GPU utilization was higher than 85% and was on an average 89%.

B. Workloads

We compare the efficiency of the proposed PIM and GPU by running four general OpenCL applications including: *Sobel, Robert, Fast Fourier transform (FFT)* and *DwHaar1D*. For image processing we use random images from *Caltech 101* [39] library, while for non-image processing applications inputs are generated randomly. Majority of these applications consists of additions and multiplications. The other common operations such as square root has been approximated by these two functions in the source code.

We also evaluate the efficiency of the proposed NNPIM over six popular neural network applications, similar to work in [17]:

Handwriting classification (MNIST) [40]: MNIST includes images of handwritten digits. The objective is to classify an input image to one of ten digits, 0 . . . 9.

Voice Recognition (ISOLET) [41]: ISOLET consists of speech signals collected from 150 speakers. The goal is to classify the vocal signal into one of 26 English letters.

Indoor Localization (INDOOR) [42]: We designed a NN model for the indoor localization dataset. This NN localizes into one of 13 places where there is high loss in GPS signals.

Activity Recognition (HAR) [43]: The dataset includes signals collected from motion sensors for 8 subjects performing 19 different activities. The objective is to recognize the class of human activities.

Object Recognition (CIFAR) [44]: CIFAR-10 and CIFAR-100 are two datasets which include 50000 training and 10000

TABLE I
NN MODELS AND BASELINE ERROR RATES FOR 6 APPLICATIONS (INPUT LAYER - *IN*, FULLY CONNECTED LAYER - *FC*, CONVOLUTION LAYER - *C*, AND POOLING LAYER - *PL*.)

Dataset	Network Topology	Error
MNIST	$IN: 784,\ FC: 512,\ FC: 512,\ FC: 10$	1.5%
ISOLET	$IN: 617,\ FC: 512,\ FC: 512,\ FC: 26$	3.6%
INDOOR	$IN: 520,\ FC: 512,\ FC: 512,\ FC: 13$	4.2%
HAR	$IN: 561,\ FC: 512,\ FC: 512,\ FC: 19$	1.7%
CIFAR-10	$IN: 32\times32\times3, CV: 32\times3\times3, PL: 2\times2,$	14.4%
CIFAR-100	$CV: 64\times3\times3, CV: 64\times3\times3, FC: 512,\ FC: 10\ (100)$	42.3%
ImageNet	VGG-16 [**?**]	28.5%
ImageNet	GoogleNet [48]	15.6%

TABLE II
QUALITY LOSS OF DIFFERENT NN APPLICATIONS DUE TO WEIGHT SHARING.

Dataset	8 weights	16 weights	32 weights	64 weights
MNIST	1.1%	0.26%	0%	0%
ISOLET	0.33%	0.12%	0%	0%
INDOOR	0.38%	0.24%	0.13%	0%
HAR	2.1%	0.32%	0.14%	0%
CIFAR-10	1.2%	0.29%	0.09%	0%
CIFAR-100	2.4%	1.2%	0.8%	0%
ImageNet (VGG)	4.6%	2.5%	1.0%	0%
ImageNet (GoogleNet)	6.3%	3.1%	0.9%	0%

testing images belonging to 10 and 100 classes, respectively. The goal is to classify an input image to the correct category, e.g., animals, airplane, automobile, ship, truck, etc.

ILSVRC2012 Image Classification (ImageNet) [45]: This dataset contains about 1200000 training samples and 50000 validation samples. The objective is to classify each image to one of 1000 categories.

Table I presents the NN topologies and baseline error rates for the original models before weight sharing. The error rate is defined by the ratio of the number of misclassified data to the total number of a testing dataset. Each NN model is trained using stochastic gradient descent with momentum [46]. In order to avoid overfitting, Dropout [47] is applied to fully-connected layers with a drop rate of 0.5. In all the NN topologies, the activation functions are set to "Rectified Linear Unit" (ReLU), and a "Softmax" function is applied to the output layer.

C. NNPIM & Dataset Size

Figure 9 shows the energy savings and performance improvements of running applications on PIM, normalized to GPU energy and performance. For each application, the size of input dataset increases from 1Kb to 1GB. In traditional cores, the energy and performance of computation consists of two terms: computation and data movement. In small dataset (˜KB), the computation cost is dominant, while running applications with large datasets (˜GB), the energy and performance of consumption are bound by the data movement rather than computation cost. This data movement is due to small cache size of transitional core which increases the number of cache miss. Consecutively, this degrades the energy consumption and performance of data movement between the memory and caches. In addition, large number of cache misses, significantly slows down the computation in traditional cores. In contrast, in proposed PIM architecture the dataset is already stored in the memory and computation is major cost. Therefore, regardless of dataset size (the dataset can fit on PIM), the PIM energy and performance of increases linearly by the dataset size. Although the memory-based computation in the PIM is slower than transitional CMOS-based computation (*i.e.* floating point units in GPU), in processing the large dataset, the proposed PIM works significantly faster than GPU. In terms of energy, the memory-based operations in PIM is more energy efficient than GPU. Our evaluation shows that for most applications using datasets larger than 200MB (which is true for many IoT applications), proposed PIM is much faster and more energy efficient than GPU. With 1GB dataset, the PIM design can achieve 28× energy savings, 4.8× performance improvement as compared to GPU architecture.

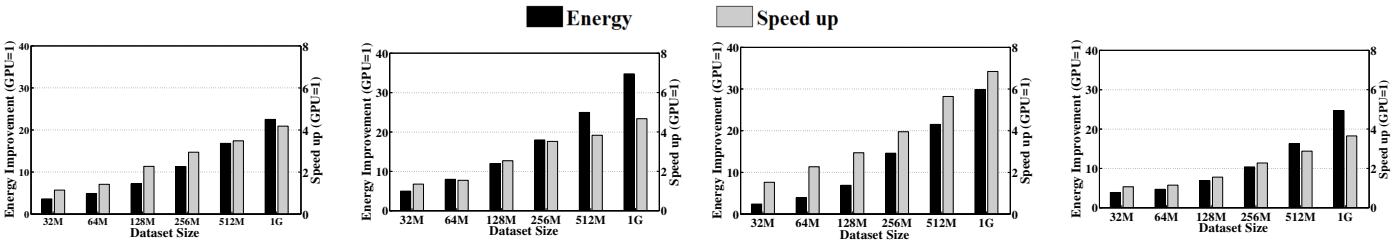
D. Comparison of NNPIM with Previous PIM Implementations

Figure 10 compares the performance efficiency of the proposed design with the state-of-the-art prior work [32], [49]. The work in [32] computes addition in-memory using MAGIC logic family, while the work in [49] uses the complementary resistive switching to perform addition inside the crossbar memory. Our evaluation comparing the energy and performance of addition of N operands of length N bits each shows that the proposed PIM can achieve at least 2× speed up compared to previous designs in exact mode. Proposed PIM can be at least 6× faster with 99.9% accuracy using the approximation techniques proposed in [12]. The proposed design is even better since the calculations for previous work do not include the latency involved in shift operations. This improvement comes at the expense of the overhead of interconnect circuitry and its control logic. However, the next best adder, *i.e.,* the PC-Adder [49] uses multiple arrays each having different wordline and bitline controllers, introducing a lot of area overhead. This overhead is not present in our design since all the blocks share the same controllers.

E. NNPIM & Weight Sharing

We compare the efficiency and accuracy of the NNPIM over different application with and without weight sharing. Table II shows the impact of weight sharing on the classification accuracy of NNPIM. Table II shows the NNPIM quality loss (QL) for different applications when the number of shared weights in each neuron changes from 8 to 64. The QL is defined as the difference between NNPIM accuracy with and without weight sharing. Our evaluation shows that a network with 64 shared weights can provide the same accuracy as a design without weight sharing. Further reducing the number of weight to 8 reduces the classification accuracy of applications. For instance, CIFAR-10 and CIFAR-100 lose 1.2% and 2.8% quality respectively when the number of shared weights decreases to 8.

NNPIM exploits this weight sharing in order to accelerate neural network computation by reducing the multiplication cost. Figure 11 shows the energy consumption and memory



(a) Sobel (b) Robert (c) FFT (d) DwtHaar1D

Fig. 9. Energy consumption and speedup of the proposed design in exact mode normalized to GPU vs different dataset sizes.

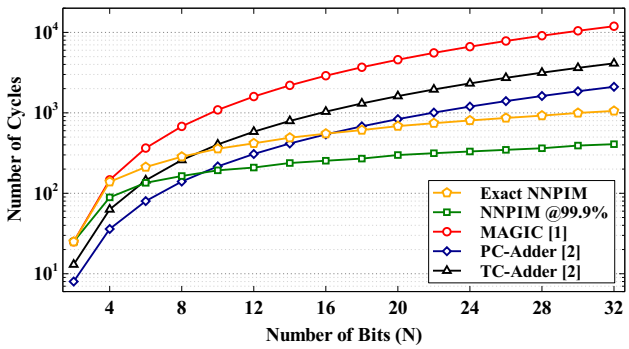


Fig. 10. Performance comparison of the proposed design with previous work for addition of N operands, each sized N bits

requirement of NNPIM running different applications with different weight sharing. The reported improvements are compared to energy consumption of the same applications running on NVIDIA GTX 1080 GPU. The energy efficiency of NNPIM significantly improves as the number of shared weights reduce. Our evaluation shows that NNPIM without weight sharing provides 14.6× energy efficiency improvement as compared to GPU architecture. We observe that NNPIM gets energy efficiency improvements from removing the data movement cost and efficient in-memory computation. However, in terms of performance the NNPIM advantage comes mostly from addressing the data movement issue.

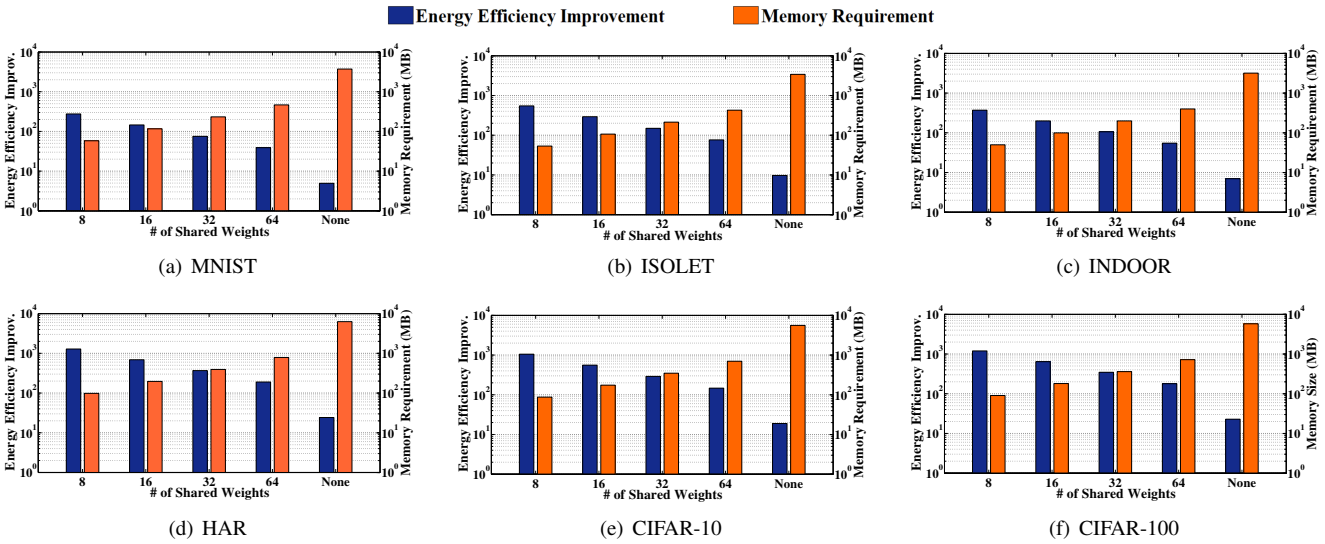
The NNPIM advantages are more obvious on large networks such as CIFAR-10 and CIFAR-100, since these networks have more data movement. Weight sharing can significantly improve the NNPIM efficiency by reducing the computation cost. The results show that NNPIM using 32-bit fixed point operations and 64 shared weights provides 131.5× energy efficiency improvement and 48.2× speedup as compared to GPU architecture at 0% quality loss. With 1% and 2% quality loss, the average energy efficiency improvements of NNPIM increase to 235.6× and 384.0× respectively. Weight sharing does not impact the performance of NNPIM since all neurons in a layer are implemented in parallel and consecutive layers are processed serially.

Figure 11 also shows the required NNPIM memory size for different amounts of weight sharing. NNPIM requires significantly lower memory size for PIM operation as compared to NNPIM without weight sharing. As our results show, decreasing the number of weights by half, reduces the number of required multiplications by half. Our evaluation over all applications indicates that by reducing the number of weights to 64, NNPIM will provide maximum quality while using 7.8× less memory as compared to NNPIM without weight sharing. Similarly, ensuring less than 1% and 2% quality loss, NNPIM uses 12.4× and 15.6× lower memory size as compared to NNPIM without weight sharing.

F. Energy Consumption and Performance

In this section we compare the energy consumption and execution time of NNPIM with DaDianNao [24] and ISAAC [19], the state-of-the-art NN accelerators. All designs have been tested over six different applications. For NN accelerators, we select the best configuration reported in the papers [19], [24]. For instance, ISAAC design works at 1.2GHz and uses 8-bits ADC, 1-bit DAC, 128×128 array size where each memristor cell stores 2 bits. DaDianNao works at 600MHz, with 36MB eDRAM size (4 per tile), 16 neural functional units, and 128-bit global bus. We see that of the previously proposed designs, ISAAC performs better over all datasets. Figure 12 shows the energy efficiency improvement and speedup of NNPIM (32-bit fixed point operations and 64 shared weights), DaDianNao [24] and ISAAC [19] as compared to NVIDIA GTX 1080 GPU architecture. For MNIST, ISOLET, and INDOOR the GPU can process each input at 4.6ms, 4.3ms, and 4.1ms respectively. For larger inputs and networks such as VGG, GPU execution is 7.2ms (138 image/s). Our evaluation shows that NNPIM outperforms both DaDianNao and ISAAC for all applications. For example, benchmarking with MNIST, proposed NNPIM can provide 2.8× energy efficiency improvement and 2.9× speedup as compared to DaDianNao. These improvements are higher for ImageNet and CIFAR-100, as NNPIM has higher computational efficiency on large networks. Our design can achieve 5.8× (1.5×) energy efficiency improvement and 6.6× (2.7×) speedup as compared to DaDianNao (ISAAC) while providing the same classification accuracy on all applications. At 1% quality loss, the NNPIM energy efficiency grows 11.3× and 3.1× as compared to DaDianNao and ISAAC respectively.

Since NNPIM removes the costly weight-input multiplications, it can achieve higher accuracy for deep networks with large number of weights. We compare NNPIM efficiency on four different networks with fully connected layers, designed for MNIST dataset. Note that the goal of this experiment is to show the impact of network size on NNPIM efficiency, not on



(a) MNIST (b) ISOLET (c) INDOOR

(d) HAR (e) CIFAR-10 (f) CIFAR-100

Fig. 11. Energy consumption and memory size requirement of NNPIM with and without weight sharing.

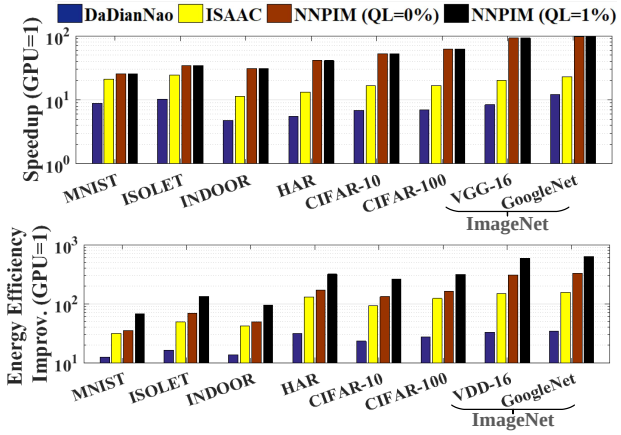


Fig. 12. Comparing the energy consumption and execution time of proposed NNPIM with state-of-the-art NN accelerators.

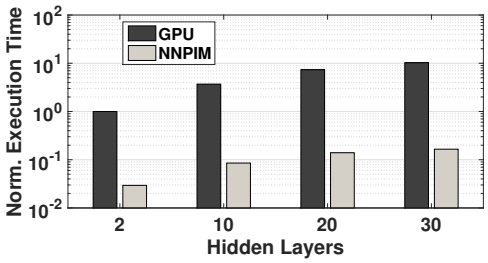


Fig. 13. Comparison of GPU and NNPIM efficiency running FC networks with different number of hidden layers.

the classification accuracy. We choose four configurations with 2, 10, 20, and 30 hidden layers, each with 4096 neurons. For example, the first network with two hidden layer has the following topology: $\{IN : 784,\ FC : 4096,\ FC : 4096,\ FC : 10\}$. Figure 13 compares the execution time of GPU and NNPIM running the same networks. All results are reported for case of using 64 shared weights. Our evaluation shows that as the size of these networks increases, GPU execution time increases significantly while the NNPIM execution time changes with much lower rate. For example, NNPIM is 34.6x faster than Nvidia GTX 1080 for a network with two hidden FC layers. This further improves to 61.7x for a network with 100 hidden FC layers. The higher efficiency of NNPIM in large network comes from (i) NNPIM's ability to parallelize each layer operations, (ii) addressing the costly data movement between memory and computing unit.

G. Computation Efficiency

An NNPIM sub-unit consists of a pair of two 64×256 memristive crossbar arrays, connected by the interconnect. The area of this sub-unit is 301.74 um^2 at 45nm process node, which includes the area of the required peripherals. To calculate area, we estimate the occupied area by each component of NNPIM. For example, for crossbar memory, we estimated the area by using NVSim tool. Similarly, the area of the interconnect and the peripheral circuits, such as the controller, column/row decoders, are calculated using Synopsys Design Compiler. The total number of blocks combined together dictates the amount of parallelism achieved. We made an accelerator with the total area similar to previous designs to have a fair comparison. NNPIM consists of 256K of sub-units with the total area of 79.10 mm^2 at 45nm process node. Table III compares the computation efficiency of NNPIM with other state-of-the-art accelerators. NNPIM has an areal computation efficiency of 1853.9 $GOP/s/mm^2$, which is higher than all other accelerators. This is primarily due to the fact that unlike other accelerators, NNPIM doesn't use big ADCs/DACs. These circuit components occupy the major part of area in previous designs. Moreover, the computation efficiency of NNPIM with respect to power is 432.9 $GOP/s/W$. Although NNPIM doesn't involve power-hungry circuit components, it uses larger crossbars which results in similar power efficiency

TABLE III
COMPUTATION EFFICIENCY OF NNPIM VS OTHER ACCELERATORS.

Metric	NNPIM	DaDianNao [24]	ISAAC [19]	AtomLayer [50]	PipeLayer [51]
$GOP/s/mm^2$	1853.9	63.4	479.0	475.6	1485.0
$GOP/s/W$	432.9	286.4	380.7	682.5	142.9

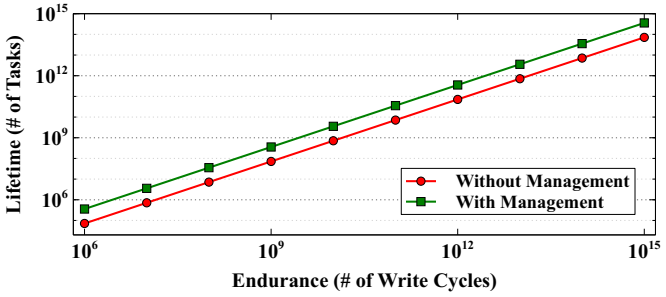


Fig. 14. Lifetime analysis of NNPIM with change in the endurance of device.

than most accelerators. Only AtomLayer [50] is 1.6× more power efficient than NNPIM since it reduces frequent ReRAM writes.

H. Lifetime Analysis

The MAGIC NOR logic results in many write operations. This may decrease reliability given the low endurance of the commercially available ReRAM devices. However, NNPIM reduces the number of switches required for neural network inference by reducing the required computation for each multiplication (Section IV-C) and decreasing the number of shifts (Section IV-D). Moreover, we implement simple endurance management technique to increase the lifetime of our design. As shown in Section III-B, implementing logic operations using MAGIC NOR generates some intermediate states. To store these intermediate states, some processing rows are reserved in a memory block which are used by all logic operations in the block. Hence, processing rows are the most active and experience the worst endurance degradation. In order to increase the lifetime of the memory, we change the rows allocated for processing overtime. This distributes the degradation across the block instead of being concentrated to a few rows, effectively reducing the worst case degradation per cell. It results in increase in the lifetime of the device. For example, for memory blocks with 64 (1024) rows, and with 12 of them reserved for processing, this management increases the lifetime of device by ˜5× (˜85×).

We perform a sensitivity study of the lifetime of NNPIM in terms of the number of classification tasks that can be performed. In our study, we vary the endurance of a cell from 10^6 to 10^{15} writes (W). To calculate the lifetime, we first calculate the worst case device state changes per memory cell (S_m) required for one inference task with different networks. Then, W/S_m gives the total number of classification tasks possible. Figure 14 shows the way the number of classification tasks change with change in the endurance. We observe that for the memory with endurance of 10^{12} writes, NNPIM can perform 3.5×10^{11} classification tasks.

I. Area Overhead

Comparing the area overhead of NNPIM to conventional crossbar memory shows that the NNPIM adds 37.2% to the area of the chip, of which 25% is for the shifter used for multiplication, 9.3% for the modified sense amplifiers, and 2.9% for the registers storing the network weights. Weight sharing significantly reduces the NN model size and the required hardware to process the weights. NNPIM area overhead is significantly lower compared to prior PIM-based DNN accelerators (87.7% in [19]) which uses eDRAM buffers and large ADCs and DACs to covert the data from digital to analog and analog to digital.

VI. CONCLUSION

In this paper, we propose NNPIM which aims at accelerating the inference phase of neural networks. We introduce a new processing in-memory based architecture to efficiently implement the huge amount of computations involved in the inference phase of neural networks. We use weight sharing to reduce the computational requirement of NNs. We exploit the consistency of weights at the hardware level by making multiplication operations more efficient using the proposed techniques. NNPIM is highly parallelizable and can process neurons belonging to a layer in parallel. Our evaluation shows that our design can achieve 131.5× higher energy efficiency and 48.2× speedup as compared to GPU architecture.

VII. ACKNOWLEDGMENT

This work was supported in part by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA and NSF grants #1730158 and #1527034.

REFERENCES

[1] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[2] L. Cavigelli, D. Bernath, M. Magno, and L. Benini, "Computationally efficient target classification in multispectral image data with deep neural networks," in *Target and Background Signatures II*, vol. 9997, p. 99970L, International Society for Optics and Photonics, 2016.

[3] C. Clark and A. Storkey, "Training deep convolutional neural networks to play go," in *International Conference on Machine Learning*, pp. 1766–1774, 2015.

[4] K. Srinivas, B. K. Rani, and A. Govrdhan, "Applications of data mining techniques in healthcare and prediction of heart attacks," *International Journal on Computer Science and Engineering (IJCSE)*, vol. 2, no. 02, pp. 250–255, 2010.

[5] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[6] 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 *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[7] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 267–278, IEEE Press, 2016.

[8] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170, ACM, 2015.

[9] 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.

[10] C. Liu, M. Hu, J. P. Strachan, and H. H. Li, "Rescuing memristor-based neuromorphic design with high defects," in *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 87, ACM, 2017.

[11] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2016.

[12] 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.

[13] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, pp. 105–117, 2016.

[14] 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.

[15] 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.

[16] S. Gupta, "Processing in memory using emerging memory technologies," Master's thesis, University of California, San Diego, 2018.

[17] 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.

[18] 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 *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 27–39, IEEE Press, 2016.

[19] 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," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[20] 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," *ACM Sigplan Notices*, vol. 49, no. 4, pp. 269–284, 2014.

[21] D. C. Ciresan, U. Meier, J. Masci, L. Maria Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," in *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, vol. 22, p. 1237, Barcelona, Spain, 2011.

[22] M. Samragh, M. Ghasemzadeh, and F. Koushanfar, "Customizing neural networks for efficient fpga implementation," in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, pp. 85–92, IEEE, 2017.

[23] 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.

[24] 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.

[25] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. S. Modha, "A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm," in *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, pp. 1–4, IEEE, 2011.

[26] A. Ren, Z. Li, C. Ding, Q. Qiu, Y. Wang, J. Li, X. Qian, and B. Yuan, "Sc-dcnn: highly-scalable deep convolutional neural network using stochastic computing," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 405–418, ACM, 2017.

[27] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "MAGIC-Memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[28] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[29] M. Imani, Y. Kim, and T. Rosing, "Mpim: Multi-purpose in-memory processing using configurable resistive memory," in *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, pp. 757–763, IEEE, 2017.

[30] M. Imani, S. Gupta, A. Arredondo, and T. Rosing, "Efficient query processing in crossbar memory," in *Low Power Electronics and Design (ISLPED, 2017 IEEE/ACM International Symposium on*, pp. 1–6, IEEE, 2017.

[31] 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.

[32] 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.

[33] 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.

[34] R. Bernstein, "Multiplication by integer constants," *Software: practice and experience*, vol. 16, no. 7, pp. 641–652, 1986.

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[36] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, (Savannah, GA), pp. 265–283, USENIX Association, 2016.

[37] 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.

[38] P. Knag, W. Lu, and Z. Zhang, "A native stochastic computing architecture enabled by memristors," *IEEE Transactions on Nanotechnology*, vol. 13, no. 2, pp. 283–293, 2014.

[39] "Caltech Library." http://www.vision.caltech.edu/Image_Datasets/Caltech101/.

[40] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits, 1998," *URL http://yann. lecun. com/exdb/mnist*, vol. 10, p. 34, 1998.

[41] "Uci machine learning repository." http://archive.ics.uci.edu/ml/datasets/ISOLET.

[42] "Uci machine learning repository." https://archive.ics.uci.edu/ml/datasets/UJIIndoorLoc.

[43] "Uci machine learning repository." https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Activities.

[44] "The cifar dataset." https://www.cs.toronto.edu/~kriz/cifar.html.

[45] 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.

[46] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, pp. 1139–1147, 2013.

[47] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[48] 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.

[49] 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.

[50] X. Qiao, X. Cao, H. Yang, L. Song, and H. Li, "Atomlayer: a universal reram-based cnn accelerator with atomic layer computation," in *Proceedings of the 55th Annual Design Automation Conference*, p. 103, ACM, 2018.

[51] 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*, pp. 541–552, IEEE, 2017.

Saransh Gupta is a Ph.D. student in the Department of Computer Science and Engineering at the University of California San Diego. He is a member of System Energy Efficiency Laboratory (SEELab). He received his B.E. (Hons) in Electrical and Electronics Engineering from Birla Institute of Technology & Science, Pilani - K.K. Birla Goa Campus in 2016 and M.S. in Electrical and Computer Engineering from University of California San Diego in 2018. His research interests include circuit, architecture, and system level aspects of emerging computing paradigms.

Mohsen Imani received his M.S. and BCs degrees from the School of Electrical and Computer Engineering at the University of Tehran in March 2014 and September 2011 respectively. From September 2014, he is a Ph.D. student in the Department of Computer Science and Engineering at the University of California San Diego, CA, USA. He is a project leader at System Energy Efficient Laboratory (SeeLab) where he is mentoring several graduate and undergraduate students on different computer engineering projects from circuit to system level. Mr. Imani's research focuses on computer architecture, machine learning and brain-inspired computing.

Harveen Kaur received her MS in Computer Science from University of California at San Diego in 2018, and B.Tech in Electronics and Communication Engineering from Indian Institute of Technology, Delhi in 2014. Her research interests include computer architecture, operating, and embedded systems. She was a member of the System Energy Efficient Laboratory (SEELAB), University of California at San Diego.

Tajana Simunic Rosing is a Professor, a holder of the Fratamico Endowed Chair, and a director of System Energy Efficiency Lab at UCSD. She is currently heading the effort in SmartCities as a part of DARPA and industry funded TerraSwarm center. During 2009-2012 she led the energy efficient datacenters theme as a part of the MuSyC center. Her research interests are energy efficient computing, embedded and distributed systems. Prior to this she was a full time researcher at HP Labs while being leading research part-time at Stanford University. She finished her PhD in 2001 at Stanford University, concurrently with finishing her Masters in En-gineering Management. Her PhD topic was Dynamic Management of Power Consumption. Prior to pursuing the PhD, she worked as a Senior Design Engineer at Altera Corporation.