PRIME: A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory

Ping Chi*, Shuangchen Li*, Cong Xu†, Tao Zhang‡, Jishen Zhao§, Yongpan Liu¶, Yu Wang¶ and Yuan Xie*
*Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106, USA
†HP Labs, Palo Alto, CA 94304, USA; ‡NVIDIA Corporation, Santa Clara, CA 95950, USA
§Department of Computer Engineering, University of California, Santa Cruz, CA 95064, USA
¶Department of Electronic Engineering, Tsinghua University, Beijing 100084, China
*Email: {pingchi, shuangchenli, yuanxie}@ece.ucsb.edu

***Abstract*—Processing-in-memory (PIM) is a promising solution to address the "memory wall" challenges for future computer systems. Prior proposed PIM architectures put additional computation logic in or near memory. The emerging metal-oxide resistive random access memory (ReRAM) has showed its potential to be used for main memory. Moreover, with its crossbar array structure, ReRAM can perform matrix-vector multiplication efficiently, and has been widely studied to accelerate neural network (NN) applications. In this work, we propose a novel PIM architecture, called PRIME, to accelerate NN applications in ReRAM based main memory. In PRIME, a portion of ReRAM crossbar arrays can be configured as accelerators for NN applications or as normal memory for a larger memory space. We provide microarchitecture and circuit designs to enable the morphable functions with an insignificant area overhead. We also design a software/hardware interface for software developers to implement various NNs on PRIME. Benefiting from both the PIM architecture and the efficiency of using ReRAM for NN computation, PRIME distinguishes itself from prior work on NN acceleration, with significant performance improvement and energy saving. Our experimental results show that, compared with a state-of-the-art neural processing unit design, PRIME improves the performance by ∼2360× and the energy consumption by ∼895×, across the evaluated machine learning benchmarks.**

***Keywords*-processing in memory; neural network; resistive random access memory**

I. Introduction

Conventional computer systems adopt separate processing (CPUs and GPUs) and data storage components (memory, flash, and disks). As the volume of data to process has skyrocketed over the last decade, data movement between the processing units (PUs) and the memory is becoming one of the most critical performance and energy bottlenecks in various computer systems, ranging from cloud servers to end-user devices. For example, the data transfer between CPUs and off-chip memory consumes two orders of magnitude more energy than a floating point operation [1]. Recent progress in processing-in-memory (PIM) techniques introduce promising solutions to the challenges [2]–[5], by leveraging 3D memory technologies [6] to integrate computation logic with the memory.

Recent work demonstrated that some emerging non-volatile memories, such as metal-oxide resistive random access memory (ReRAM) [7], spin-transfer torque magnetic RAM (STT-RAM) [8], and phase change memory (PCM) [9], have the capability of performing logic and arithmetic operations beyond data storage. This allows the memory to serve both computation and memory functions, promising a radical renovation of the relationship between computation and memory. Among them, ReRAM can perform matrix-vector multiplication efficiently in a crossbar structure, and has been widely studied to represent synapses in neural computation [10]–[15].

Neural network (NN) and deep learning (DL) have the potential to provide optimal solutions in various applications including image/speech recognition and natural language processing, and are gaining a lot of attention recently. The state-of-the-art NN and DL algorithms, such as multi-layer perceptron (MLP) and convolutional neural network (CNN), require a large memory capacity as the size of NN increases dramatically (e.g., 1.32GB synaptic weights for Youtube video object recognition [16]). High-performance acceleration of NN requires high memory bandwidth since the PUs are hungry for fetching the synaptic weights [17]. To address this challenge, recent special-purpose chip designs have adopted large on-chip memory to store the synaptic weights. For example, DaDianNao [18] employed a large on-chip eDRAM for both high bandwidth and data locality; TrueNorth utilized an SRAM crossbar memory for synapses in each core [19]. Although those solutions effectively reduce the transfer of synaptic weights between the PUs and the off-chip memory, the data movement including input and output data besides synaptic weights is still a hinderance to performance improvement and energy saving. Instead of integrating more on-chip memory, PIM is a promising solution to tackle this issue by putting the computation logic into the memory chip, so that NN computation can enjoy the large memory capacity and sustain high memory bandwidth via in-memory data communication at the same time.

*Shuangchen Li and Ping Chi have equal contribution. This work is supported in part by NSF 1461698, 1500848, and 1533933, and DOE grant DE-SC0013553, and a grant from Qualcomm.

In this work, we propose a novel PIM architecture for efficient NN computation built upon ReRAM crossbar arrays, called PRIME, processing in ReRAM-based main memory. ReRAM has been proposed as an alternative to build the next-generation main memory [20], and is also a good candidate for PIM thanks to its large capacity, fast read speed, and computation capability. In our design, a portion of memory arrays are enabled to serve as NN accelerators besides normal memory. Our circuit, architecture, and software interface designs allow these ReRAM arrays to dynamically reconfigure between memory and accelerators, and also to represent various NNs. The current PRIME design supports large-scale MLPs and CNNs, which can produce the state-of-the-art performance on varieties of NN applications, e.g. top classification accuracy for image recognition tasks. Distinguished from all prior work on NN acceleration, PRIME can benefit from both the efficiency of using ReRAM for NN computation and the efficiency of the PIM architecture to reduce the data movement overhead, and therefore can achieve significant performance gain and energy saving. As no dedicated processor is required, PRIME incurs very small area overhead. It is also manufacture friendly with low cost, since it remains as the memory design without requirement for complex logic integration or 3D stacking.

The contribution of this paper is summarized as follows:

- We propose a ReRAM main memory architecture, which contains a portion of memory arrays (full function subarrays) that can be configured as NN accelerators or as normal memory on demand. It is a novel PIM solution to accelerate NN applications, which enjoys the advantage of in-memory data movement, and also the efficiency of ReRAM based computation.
- We design a set of circuits and microarchitecture to enable the NN computation in memory, and achieve the goal of low area overhead by careful design, e.g. reusing the peripheral circuits for both memory and computation functions.
- With practical assumptions of the technologies of using ReRAM crossbar arrays for NN computation, we propose an input and synapse composing scheme to overcome the precision challenge.
- We develop a software/hardware interface that allows software developers to configure the full function subarrays to implement various NNs. We optimize NN mapping during compile time, and exploit the bank-level parallelism of ReRAM memory for further acceleration.

II. Background and Related Work

This session presents the background and related work on ReRAM basics, NN computation using ReRAM, and PIM.

A. ReRAM Basics

Resistive random access memory, known as ReRAM, is a type of non-volatile memory that stores information

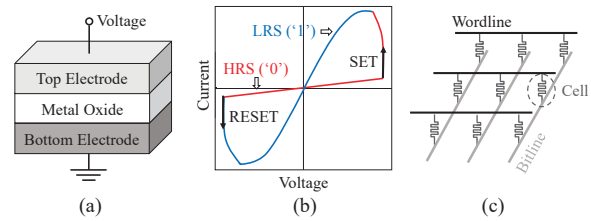

Figure 1. (a) Conceptual view of a ReRAM cell; (b) I-V curve of bipolar switching; (c) schematic view of a crossbar architecture.

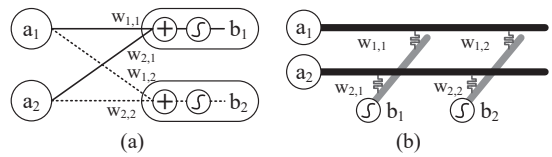

Figure 2. (a) An ANN with one input/output layer; (b) using a ReRAM crossbar array for neural computation.

by changing cell resistances. The general definition does not specify the resistive switching material. This work focuses on a subset of resistive memories, called metal-oxide ReRAM, which uses metal oxide layers as switching materials.

Figure 1(a) demonstrates the metal-insulator-metal (MIM) structure of a ReRAM cell: a top electrode, a bottom electrode, and a metal-oxide layer sandwiched between them [7]. By applying an external voltage across it, a ReRAM cell can be switched between a high resistance state (HRS) and a low resistance state (LRS), which are used to represent the logic "0" and "1", respectively.

Figure 1(b) shows the I-V characteristics of a typical bipolar ReRAM cell. Switching a cell from HRS (logic "0") to LRS (logic "1") is a SET operation, and the reverse process is a RESET operation. To SET the cell, a positive voltage that can generate sufficient write current is required. To RESET the cell, a negative voltage with a proper magnitude is necessary. The reported endurance of ReRAM is up to 10^{12} [21], [22], making the lifetime issue of ReRAM-based memory less concerned than PCM based main memory whose endurance has been assumed between 10^6-10^8 [23].

An area-efficient array organization for ReRAM is crossbar structure as shown in Figure 1(c) [24]. There are two common approaches to improve the density and reduce the cost of ReRAM: multi-layer crossbar architecture [25]–[27] and multi-level cell (MLC) [28]–[30]. In MLC structure, ReRAM cells can store more than one bit of information in a single cell with various levels of resistance. This MLC characteristic can be realized by changing the resistance of ReRAM cell gradually with finer write control. Recent work has demonstrated 7-bit MLC ReRAM [31].

Due to crossbar architecture's high density, ReRAM has been considered as a cost-efficient replacement of DRAM to build next-generation main memory [20]. The read latency of ReRAM can be comparable to that of DRAM while its write latency is significantly longer than that of DRAM (*e.g.* 5×). Several architectural techniques were proposed [20] to

improve the write performance, bridging the performance gap between the optimized ReRAM and DRAM within 10%. In this work, we adopt a similar performance optimized design of the ReRAM based main memory [20].

B. Accelerating NNs in Hardware

Artificial neural networks (ANNs) are a family of machine learning algorithms inspired by the human brain structure. Generally, they are presented as network of interconnected neurons, containing an input layer, an output layer, and sometimes one or more hidden layers. Figure 2(a) shows a simple neural network with an input layer of two neurons, an output layer of two neurons, and no hidden layers. The output b_j is calculated as,

$$b_j = \sigma(\sum_{\forall i} a_i \cdot w_{i,j}), \tag{1}$$

where a_i are input data, $w_{i,j}$ is synaptic weights, and σ is a non-linear function, for $i = 1, 2$, and $j = 1, 2$.

In the era of big data, machine learning is widely used to learn from and make predictions on a large amount of data. With the advent of deep learning, some neural network algorithms such as convolutional neural networks (CNNs) and deep neural networks (DNNs) start to show their power and effectiveness across a wide range of applications [17], [18]. Researchers have also utilized NNs to accelerate approximate computing [32], [33].

Prior studies [19], [34], [35] strive to build neuromorphic systems with CMOS-based neurons and synapses. However, doing so introduces substantial design challenges due to the huge area occupied by thousands of transistors used to implement numerous neurons and synapses. Alternatively, ReRAM is becoming a promising candidate to build area-efficient synaptic arrays for NN computation [10]–[13], as it emerges with crossbar architecture. Recently, Presioso *et al.* fabricated a 12×12 ReRAM crossbar prototype with a fully operational neural network, successfully classifying 3×3-pixel black/white images into 3 categories [12]. Figure 2(b) shows an example of using a 2×2 ReRAM crossbar array to execute the neural networks in Figure 2(a). The input data a_i is represented by analog input voltages on the wordlines. The synaptic weights $w_{i,j}$ are programmed into the cell conductances in the crossbar array. Then the current flowing to the end of each bitline is viewed as the result of the matrix-vector multiplication, $\sum_i a_i \cdot w_{i,j}$. After sensing the current on each bitline, the neural networks adopt a non-linear function unit to complete the execution.

Implementing NNs with ReRAM crossbar arrays requires specialized peripheral circuit design. For example, digital-to-analog converters (DACs) and analog-to-digital converters (ADCs) are needed for analog computing. Also, a sigmoid unit as well as a substraction unit is required, since matrices with positive and negative weights are implemented as two separated crossbar arrays.

There are a lot of studies on using ReRAM for NN computation, from stand-alone accelerator [10], [14], [15], co-processor [11], to many-core or NoC [36] architecture. Recently, a full-fledged NN accelerator design based on ReRAM crossbars have been proposed, named ISAAC [37]. Most prior work exploits ReRAM either as DRAM/flash replacement [20], [27], [38] or as synapses for NN computation [10]–[13], [36]. In this work, PRIME is a morphable ReRAM based main memory architecture, where a portion of ReRAM crossbar arrays are enabled with the NN computation function, referred as full function subarrays. When NN applications are running, PRIME can execute them with the full function subarrays to improve performance or energy efficiency; while no NN applications are executed, the full function subarrays can be freed to provide extra memory capacity.

There are also many other studies on accelerating NNs on the platforms of GPU [16], [39], [40], FPGA [41]–[43] and ASIC [17]–[19], [34], [35], [44]. The DianNao series [17], [18], [44] are good examples of ASIC-based NN accelerators; furthermore, the first instruction set architecture for NN accelerators has been proposed, called Cambricon [45]. Distinguished from the existing work on accelerating NNs in hardware, PRIME proposes a PIM solution for the first time to our best knowledge. Most prior work focused on the co-processor architecture, in which data are accessed from main memory in a conventional way, as shown in Figure 3(a). Since many NN applications require high memory bandwidth to fetch large-size input data and synaptic weights, the data movement between memory and processor is both time-consuming and energy-consuming. As reported, DRAM accesses consume 95% of the total energy in DianNao design [17]. To address this challenge, some recent work on ASIC put more memory on chip for synaptic weight storage [18], [19], [34], [35]. However, the issue still exists due to the transfer of input and output data. In this work, we propose to accelerate NNs in a PIM architecture, moving the computing resources to the memory side by adapting a portion of ReRAM crossbar arrays in the main memory as NN accelerator. It takes advantage of the large internal bandwidth of the main memory, and makes the data movement minimal. Recently, we see a lot of work that focused on spiking neural networks (SNNs), e.g. TrueNorth [19], [34], [35]. ReRAM can also implement SNN [13]. Making PRIME to support SNN is our future work.

C. Processing-in-memory (PIM)

PIM is not a new concept, and there has been a lot of work on it since 1990s, e.g., IRAM [46], [47] and DIVA [48]. Early efforts explored integrating simple ALU [49], vectorization [46], SIMD [50], general-purpose processors [51], and FPGA [52] with DRAM. Unfortunately, the idea of integrating performance-optimized logic with density-optimized

memory aroused a lot of criticism from the cost-sensitive memory industry [53]. Recently, driven by the data intensive applications and the 3D-stacking technology, PIM or near data computing (NDC) is resurgent, with lots of industry effort (e.g., IBM [54], AMD [4], and Samsung [55]). Recent efforts [2]–[5], [56] decouple logic and memory designs in different dies, adopting 3D stacked memories with a logic layer that encapsulates processing units to perform computation, as shown in Figure 3(b). This architecture design is compatible with the hybrid memory cube (HMC) [57] and high bandwidth memory (HBM) [58].

PRIME is a distinct solution from either early or recent PIM work. Instead of adding logic to memory, PRIME utilizes the memory arrays themselves for computing, hence area overhead is very small. The add-on hardware in PRIME to enable the computation function consist of simple modifications of the existing memory peripheral circuits, which are more manufacture friendly than integrating complex logic into the memory die. Moreover, PRIME does not rely on 3D-stacking technology, exempt from its high cost and thermal problems. Also, while previous work focused on database and graph processing applications [3], [5], [59], PRIME aims at accelerating NN applications.

Recent work also employs nonvolatile memory technologies (ReRAM, PCM, and STT-RAM) to build ternary content addressable memories (TCAMs), which exploits memory cells to perform associative search operations [60]–[62]. However, to support such search operations, it requires a redesign of their memory cell structures which makes the cell sizes larger and inevitably increases the memory cost. Compared to these TCAM designs, PRIME obviates memory cell redesign, and can support more sophisticated computation than TCAMs.

III. PRIME Architecture

We propose processing in ReRAM-based main memory, PRIME, which efficiently accelerates NN computation by leveraging ReRAM's computation capability and the PIM architecture. Figure 3(c) depicts an overview of our design. While most previous NN acceleration approaches require additional processing units (PU) (Figure 3(a) and (b)), PRIME directly leverages ReRAM cells to perform computation without the need for extra PUs. To achieve this, as shown in Figure 3(c), PRIME partitions a ReRAM bank into three regions: memory (Mem) subarrays, full function (FF) subarrays, and Buffer subarrays.

The Mem subarrays only have data storage capability (the same as conventional memory subarrays). Their microarchitecture and circuit designs are similar to a recent design of performance-optimized ReRAM main memory [20]. The FF subarrays have both computation and data storage capabilities, and they can operate in two modes. In memory mode, the FF subarrays serve as conventional memory; in computation mode, they can execute NN computation. There

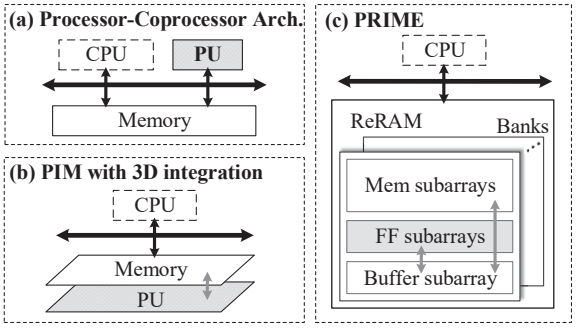

Figure 3. (a) Traditional shared memory based processor-coprocessor architecture, (b) PIM approach using 3D integration technologies, (c) PRIME design.

is a PRIME controller to control the operation and the reconfiguration of the FF subarrays. The Buffer subarrays serve as data buffers for the FF subarrays, and we use the memory subarrays that are closest to the FF subarrays as Buffer subarrays. They are connected to the FF subarrays through private data ports, so that buffer accesses do not consume the bandwidth of the Mem subarrays. While not being used as data buffers, the Buffer subarrays can also be used as normal memory. From Figure 3(c), we can find that for NN computation the FF subarrays enjoy the high bandwidth of in-memory data movement, and can work in parallel with CPU, with the help of the Buffer subarrays.

This section describes the details of our microarchitecture and circuit designs of the FF subarrays, the Buffer subarrays, and the PRIME controller. These designs are independent of the technology assumptions for ReRAM based computation. For generality, we assume that the input data have P_{in} bits, the synaptic weights have P_w bits, and the output data have P_o bits. With practical assumptions, the precision of ReRAM based NN computation is a critical challenge. We discuss the precision issue and propose a scheme to overcome it in Section III-D. Finally, more details are given about implementing NN algorithms with our hardware design.

A. FF Subarray Design

The design goal for FF subarray is to support both storage and computation with a minimum area overhead. To achieve this goal, we maximize the reuse of peripheral circuits for both storage and computation.

1) Microarchitecture and Circuit Design: To enable the NN computation function in FF subarrays, we modify decoders and drivers, column multiplexers (MUX), and sense amplifiers (SA) as shown in Figure 4.

Decoder and Driver. We add several components in decoders and drivers marked as light blue in Figure 4 Ⓐ. First, we attach multi-level voltage sources to the wordlines to provide accurate input voltages. NN computation requires that all input data are simultaneously fed into the corresponding wordline. Therefore, we add a latch to control the input voltage. The control signals determine the combination of voltage sources that provide the demanding input voltage.

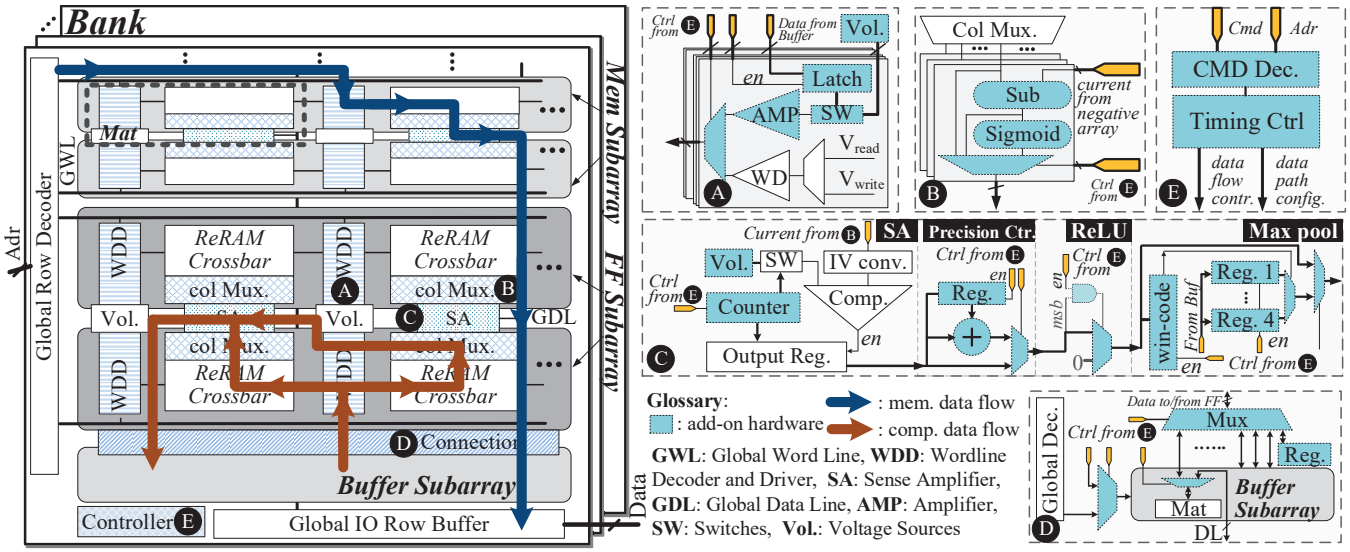

Figure 4. The PRIME architecture. Left: bank structure. The blue and red bold lines represent the directions of the data flow for normal memory and for computation, respectively. Right: functional blocks modified/added in PRIME. (A) Wordline driver with multi-level voltage sources; (B) column multiplexer with analog subtraction and sigmoid circuitry; (C) reconfigurable SA with counters for multi-level outputs, and added ReLU and 4-1 max pooling function units; (D) connection between the FF and Buffer subarrays; (E) PRIME controller.

Second, to drive the analog signals transferring on the wordlines, we employ a separate current amplifier on each wordline. Third, rather than two voltage levels used in the memory mode (for read and write, respectively), NN computation requires $2^{P_{in}}$ levels of input voltages. We employ a multiplexer to switch the voltage driver between memory and computation modes. Finally, we employ two crossbar arrays store positive and negative weights, respectively, and allow them to share the same input port.

Column Multiplexer. In order to support NN computation, we modify the column multiplexers in ReRAM by adding the components marked in light blue in Figure 4 B. The modified column multiplexer incorporates two analog processing units: an analog subtraction unit and a non-linear threshold (sigmoid) unit [63]. The sigmoid unit can be bypassed in certain scenarios, e.g. when a large NN is mapped to multiple crossbar arrays. In addition, in order to allow FF subarrays to switch bitlines between memory and computation modes, we attach a multiplexer to each bitline to control the switch. Since a pair of crossbar arrays with positive and negative weights require one set of such peripheral circuits, we only need to modify half of the column multiplexers. After analog processing, the output current is sensed by local SAs.

Sense Amplifier. Figure 4 C shows the SA design with the following modifications as marked in light blue in the figure. First, NN computation requires SAs to offer much higher precision than memory does. We adopt a P_o-bit ($P_o \leq 8$) precision reconfigurable SA design that has been tested through fabrication [64]. Second, we allow SA's precision to be configured as any value between 1-bit and P_o-bit, controlled by the counter as shown in Figure 4 C. The result is stored in the output registers. Third, we allow low-precision ReRAM cells to perform NN computation with a high-precision weight, by developing a precision control circuit that consists of a register and an adder. Fourth, we add a hardware unit to support ReLU function, a function in the convolution layer of CNN. The circuit checks the sign bit of the result. It outputs zero when the sign bit is negative and the result itself otherwise. Finally, a circuit to support 4-1 max pooling is included. More details are discussed in Section III-E.

Buffer Connection. Figure 4 D shows the communication between the FF subarrays and the Buffer subarrays. We enable an FF subarray to access any physical location in a Buffer subarray to accommodate the random memory access pattern in NN computation (e.g., in the connection of two convolutional layers). To this end, extra decoders and multiplexers are employed in the buffer connection unit. Additionally, we allow the data transfer to bypass the Buffer subarray in certain scenarios, e.g. when the output of one mat is exactly the input of another. After bypassing the Buffer subarrays, we employ a register as an intermediate data storage.

Benefits of Our Design are two-fold. First, our design efficiently utilizes the peripheral circuits by sharing them between memory and computation functions, which significantly reduces the area overhead. For example, in a typical ReRAM-based neuromorphic computing system [10], DACs and ADCs are used for input and output signal conversions; in a ReRAM-based memory system, SAs and write drivers are required for read and write operations. Yet, SAs and ADCs serve similar functions, while write drivers and DACs do similar functions. In PRIME, instead of using both,

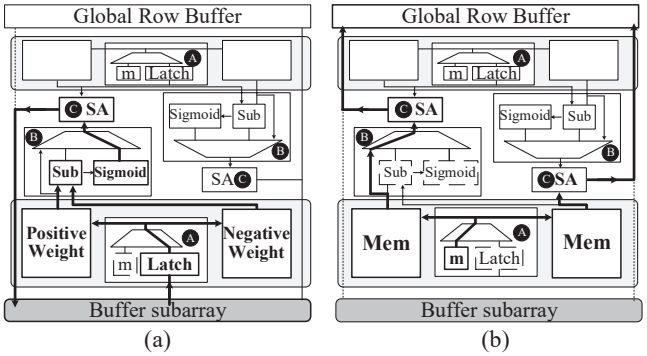

Figure 5. An example of the configurations of FF subarrays. (a) Computation mode; (b) memory mode.

we reuse SAs and write drivers to serve ADC and DAC functions by slightly modifying the circuit design. Second, we enable the FF subarrays to flexibly and efficiently morph between memory and computation modes.

2) Morphing Between Two Modes: Figure 5 shows two FF subarrays that are configured into computation and memory modes, respectively. The black bold lines in the figure demonstrate the data flow in each configuration. As shown in Figure 5(a), in computation mode, the FF subarray fetches the input data of the NN from the Buffer subarray into the latch of the wordline decoder and driver. After the computation in the crossbar arrays that store positive and negative weights, their output signals are fed into the subtraction unit, and then the difference signal goes into the sigmoid unit. The analog output is converted to digital signal by the SA is written back to the Buffer subarray. As shown in Figure 5(b), in memory mode, the input comes from the read/write voltage selection (denoted by an **m** box), and the output bypasses the subtraction and sigmoid units.

The morphing between memory and computation modes involves several steps. Before the FF subarrays switch from memory mode to computation mode, PRIME migrates the data stored in the FF subarrays to certain allocated space in Mem subarrays, and then writes the synaptic weights to be used by computation into the FF subarrays. When data preparations are ready, the peripheral circuits are reconfigured by the PRIME controller, and the FF subarrays are switched to computation mode and can start to execute the mapped NNs. After completing the computation tasks, the FF subarrays are switched back to memory mode through a wrap-up step that reconfigures the peripheral circuits.

B. Buffer Subarrays

The goal of the Buffer subarrays is two-fold. First, they are used to cache the input and output data for the FF subarrays. Benefiting from the massive parallelism of matrix-vector multiplication provided by ReRAM crossbar structures, the computation itself takes a very short time. Moreover, the data input and output may be serial, and their latencies become potential bottlenecks. Therefore, it is necessary to cache the input and output data. Second, the FF subarrays can communicate with the Buffer subarrays directly without the involvement of the CPU, so that the CPU and the FF subarrays can work in parallel.

We choose to configure the adjacent memory subarray to the FF subarrays as the Buffer subarray, which is close to both the FF subarrays and the global row buffer so as to minimize the delay. We do not utilize the local row buffer because it is not large enough to serve typical NNs. We do not implement the buffer with low-latency SRAM due to its large area and cost overhead.

As described in Section III-A1, the Buffer subarray and the FF subarrays are connected by the connection unit which enables the FF subarrays to access any data in the buffer. To fetch data for the FF subarrays, the data are first loaded from a Mem subarray to the global row buffer, and then they are written from the row buffer to the Buffer subarray. These two steps have to be done in serial due to the resource conflict, i.e. the global data lines (GDL). The communication between the Buffer subarray and the FF subarrays is independent with the communication between the Mem subarray and the globe row buffer. Therefore, when PRIME is accelerating NN computation, CPU can still access the memory and work in parallel. To write the data from the Buffer subarray to memory, the data go through the global row buffer to the corresponding Mem subarray.

C. PRIME Controller

Figure 4 E illustrates the PRIME controller that decodes instructions and provides control signals to all the peripheral circuits in the FF subarrays. A key role of the controller is to configure the FF subarrays in memory and computation modes. Table I lists the basic commands used by the controller. The left four commands generate control signals for the multiplexers in Figure 4, including the function selection of each mat among programming synaptic weights, computation, and memory, and also the input source selection for computation, either from the Buffer subarray or from the output of the previous layer. These commands are performed once during each configuration of the FF subarrays. The right four commands in Table I control the data movement. They are applied during the whole computation phase.

Table I
PRIME CONTROLLER COMMANDS

Datapath Configure	**Data Flow Control**
prog/comp/mem [mat adr][0/1/2]	fetch [mem adr] to [buf adr]
bypass sigmoid [mat adr] [0/1]	commit [buf adr] to [mem adr]
bypass SA [mat adr][0/1]	load [buf adr] to [FF adr]
input source [mat adr][0/1]	store [FF adr] to [buf adr]

D. Overcoming the Precision Challenge

The precision issue is one of the most critical challenges for ReRAM based NN computation. It contains several aspects: input precision, synaptic weight (or cell resistance) precision, output (or analog computation) precision, and

their impacts on the results of NN applications (e.g. the classification accuracy of image recognition tasks).

Previous work has employed 1-bit to 12-bit synaptic weights for ReRAM based NN computation [11]–[13]. There have been active research going on with improving the resistance precision of MLC ReRAM cells. With a simple feedback algorithm, the resistance of a ReRAM device can be tuned with 1% precision (equivalent to 7-bit precision) for a single cell and about 3% for the cells in crossbar arrays [31], [65].

The latest results of the Dot-Product Engine project from HP Labs reported that, for a 256×256 crossbar array, given full-precision inputs (e.g. usually 8-bit for image data), 4-bit synaptic weights can achieve 6-bit output precision, and 6-bit synaptic weights can achieve 7-bit output precision, when the impacts of noise on the computation precision of ReRAM crossbar arrays are considered [66].

We evaluated the impacts of input and synaptic weight precisions on a handwritten digit recognition task using LeNet-5, a well-known CNN, over the MNIST database [67]. We adopt the dynamic fixed point data format [68], and apply it to represent the input data and synaptic weights of every layer. From the results as shown in Figure 6, for this NN application, 3-bit dynamic fixed point input precision and 3-bit dynamic fixed point synaptic weight precision are adequate to achieve 99% classification accuracy, causing negligible accuracy loss compared with the result of floating point data format. The results indicate that NN algorithms are very robust to the precisions of input data and synaptic weights.

Our PRIME design can be adapted to different assumptions of input precision, synaptic weight precision, and output precision. According to the state-of-the-art technologies used in ReRAM based NN computation, one practical assumption is that: the input voltage have only 3-bit precision (i.e. 8 voltage levels), and the ReRAM cells can only represent 4-bit synaptic weights (i.e. 16 resistance levels), and the target output precision is 6-bit. The data format we use is dynamic fixed point [68]. To achieve high computation accuracy with conservative assumptions, we propose an input and synapse composing scheme, which can use two 3-bit input signals to compose one 6-bit input signal and two 4-bit cells to represent one 8-bit synaptic weight.

1) Input and Synapse Composing Scheme: We present the input and synapse composing algorithm first, and then present the hardware implementation. Table II lists the notations.

If the computation in a ReRAM crossbar array has full accuracy, the result should be

$$R_{\text{full}} = \sum_{i=1}^{2^{P_N}} \left(\sum_{k=1}^{P_{\text{in}}} I_k^i 2^{k-1} \cdot \sum_{k=1}^{P_w} W_k^i 2^{k-1}\right), \tag{2}$$

which has $(P_{\text{in}} + P_w + P_N)$-bit full precision. Since the target output is P_o-bit, we will take the highest P_o-bit of

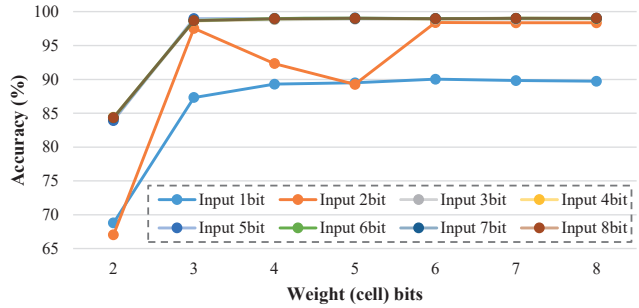

Figure 6. The precision result.

Table II
NOTATION DESCRIPTION.

P_{in}, P_o, P_w	the number of bits for input/output/synaptic weights
P_N	the number of inputs to a crossbar array is 2^{P_N}
I_k^i, W_k^i	the kth bit of the ith input signal/synaptic weight
Ih_k^i, Il_k^i	the kth bit of HIGH/LOW-bit part of the ith input
Wh_k^i, Wl_k^i	the kth bit of HIGH/LOW-bit part of the ith weight

R_{full}. Then, the target result is denoted as shifting R_{full} to the right by $(P_{\text{in}} + P_w + P_N - P_o)$ bits:

$$R_{\text{target}} = R_{\text{full}} \gg (P_{\text{in}} + P_w + P_N - P_o). \tag{3}$$

Now each input signal and synaptic weight are composed of two parts: high-bit part and low-bit part. We have,

$$\text{input: } \sum_{k=1}^{P_{\text{in}}} I_k^i 2^{k-1} = \sum_{k=1}^{P_{\text{in}}/2} (Ih_k^i 2^{k-1} \cdot 2^{P_{\text{in}}/2} + Il_k^i 2^{k-1}) \tag{4}$$

$$\text{weight: } \sum_{k=1}^{P_w} W_k^i 2^{k-1} = \sum_{k=1}^{P_w/2} (Wh_k^i 2^{k-1} \cdot 2^{P_w/2} + Wl_k^i 2^{k-1}). \tag{5}$$

Then, R_{full} will contain four parts (i.e., HH-part, HL-part, LH-part, and LL-part),

$$R_{\text{full}} = \sum_{i=1}^{2^{P_N}} \{ 2^{\frac{P_w+P_{\text{in}}}{2}} \cdot \underbrace{\sum_{k=1}^{P_{\text{in}}/2} Ih_k^i 2^{k-1} \sum_{k=1}^{P_w/2} Wh_k^i 2^{k-1}}_{HH-part} \tag{6}$$

$$+ 2^{\frac{P_w}{2}} \cdot \underbrace{\sum_{k=1}^{P_{\text{in}}/2} Il_k^i 2^{k-1} \sum_{k=1}^{P_w/2} Wh_k^i 2^{k-1}}_{HL-part}$$

$$+ 2^{\frac{P_{\text{in}}}{2}} \cdot \underbrace{\sum_{k=1}^{P_{\text{in}}/2} Ih_k^i 2^{k-1} \sum_{k=1}^{P_w/2} Wl_k^i 2^{k-1}}_{LH-part} + \underbrace{\sum_{k=1}^{P_{\text{in}}/2} Il_k^i 2^{k-1} \sum_{k=1}^{P_w/2} Wl_k^i 2^{k-1}}_{LL-part} \}. \tag{7}$$

Here, we rewrite R_{full} as

$$R_{\text{full}} = 2^{\frac{P_w+P_{\text{in}}}{2}} \cdot R_{\text{full}}^{\text{HH}} + 2^{\frac{P_w}{2}} \cdot R_{\text{full}}^{\text{HL}} + 2^{\frac{P_{\text{in}}}{2}} \cdot R_{\text{full}}^{\text{LH}} + R_{\text{full}}^{\text{LL}}. \tag{8}$$

We can also denote R_{target} with four parts:

$$R_{\text{target}} = R_{\text{tar}}^{\text{HH}} + R_{\text{tar}}^{\text{HL}} + R_{\text{tar}}^{\text{LH}} + R_{\text{tar}}^{\text{LL}}. \tag{9}$$

In equation (8), if the output of each R_{full} part is only P_o-bit, then,

- $R_{\text{tar}}^{\text{HH}}$: take all the P_o bits of $R_{\text{full}}^{\text{HH}}$ result
- $R_{\text{tar}}^{\text{HL}}$: take the highest $P_o - \frac{P_{\text{in}}}{2}$ bits of $R_{\text{full}}^{\text{HL}}$ result

- $R_{\text{tar}}^{\text{LH}}$: take the highest $P_o - \frac{P_w}{2}$ bits of $R_{\text{full}}^{\text{LH}}$ result
- $R_{\text{tar}}^{\text{LL}}$: take the highest $P_o - \frac{P_{\text{in}}+P_w}{2}$ bits of $R_{\text{full}}^{\text{LL}}$.

According to our assumptions, we have $P_{\text{in}} = 6$ (composed of two 3-bit signals), $P_w = 8$ (composed of two 4-bit cells), and $P_{\text{out}} = 6$ (enabled by 6-bit precision reconfigurable sense amplifiers). The target result should be the summation of three components: all the 6 bits of $R_{\text{full}}^{\text{HH}}$ output, the highest 3 bits of $R_{\text{full}}^{\text{HL}}$ output, and the highest 2 bits of $R_{\text{tar}}^{\text{LH}}$ output.

To implement synapse weight composing, P_{in} is loaded to the latch in the WL driver as shown in Figure 4 A. According to the control signal, the high-bit and low-bit parts of the input are fed to the corresponding crossbar array sequentially. To implement synapse composing, the high-bit and low-bit parts of the synaptic weights are stored in adjacent bitlines of the corresponding crossbar array. As shown in Equation (9), R_{target} consists of four components. They are calculated one by one, and their results are accumulated with the adder in Figure 4 C. The right shift operation, i.e. taking the highest several bits of a result, can be implemented by the reconfigurable SA. To take the highest n-bit of a result, we simply configure the SA as an n-bit SA.

E. Implementing NN Algorithms

MLP/Fully-connected Layer: Matrix-vector Multiplication. Matrix-vector multiplication is one of the most important primitives in NN algorithms, as shown in Figure 2 and Equation (1). The ReRAM crossbar arrays are used to implement it: the weight matrix is pre-programmed in ReRAM cells; the input vector is the voltages on the wordlines driven by the drivers (as shown in Figure 4 A); the output currents are accumulated at the bitlines. The synaptic weight matrix is separated into two matrices: one storing the positive weights and the other storing the negative weights. They are programmed into two crossbar arrays. A subtraction unit (as shown in Figure 4 B) is used to subtract the result of the negative part from that of the positive part.

MLP/Fully-connected Layer: Activation Function. Our circuit design supports two activation functions: sigmoid and ReLU. Sigmoid is implemented by the sigmoid unit in Figure 4 B, and ReLU is implemented by the ReLU unit in Figure 4 C. These two units can be configured to bypass in some scenarios.

Convolution Layer. The computation of the convolution layer is described as follows,

$$f_i^{\text{out}} = max(\sum_{j=1}^{n_{\text{in}}} f_j^{\text{in}} \otimes g_{i,j} + b_i, 0),\ 1 \le i \le n_{\text{out}}, \quad (10)$$

where f_j^{in} is the j-th input feature map, and f_i^{out} is the i-th output feature map, $g_{i,j}$ is the convolution kernel for f_j^{in} and f_i^{out}, b_i is the bias term, and n_{in} and n_{out} are the numbers of the input and output feature maps, respectively.

To implement the summation of n_{in} convolution operations ($f_j^{\text{in}} \otimes g_{i,j}$) plus b_i, all the elements of j convolution kernels $g_{i,j}$ are pre-programmed in the ReRAM cells of one BL or more BLs if they cannot fit in one, and the elements of f_j^{in} are performed as input voltages. We also write b_i in ReRAM cells, and regard the corresponding input as "1". Each BL will output the whole or part of the convolution result. If more BLs are used, it takes one more step to achieve the final result. Next, the $\max(x, 0)$ function is executed by the ReLU logic in Figure 4 C.

Pooling Layer. To implement max pooling function, we adopt 4:1 max pooling hardware in Figure 4 C, which is able to support n:1 max pooling with multiple steps for $n > 4$. For 4:1 max pooling, first, four inputs $\{a_i\}$ are stored in the registers, $i = 1, 2, 3, 4$; second, we execute the dot products of $\{a_i\}$ and six sets of weights [1,-1,0,0], [1,0,-1,0], [1,0,0,-1], [0,1,-1,0], [0,1,0,-1], [0,0,1,-1] by using ReRAM to obtain the results of $(a_i - a_j)$, $i \neq j$; next, the signs of their results are stored in the Winner Code register; finally, according to the code, the hardware determines the maximum and outputs it. Mean pooling is easier to implement than max pooling, because it can be done with ReRAM and does not require extra hardware. To perform n:1 mean pooling, we simply pre-program the weights $[1/n, \cdots, 1/n]$ in ReRAM cells, and execute the dot product of the inputs and the weights to obtain the mean value of n inputs.

Local Response Normalization (LRN) Layer. Currently, PRIME does not support LRN acceleration. We did not add the hardware for LRN, because state-of-the-art CNNs do not contain LRN layers [69]. When LRN layers are applied, PRIME requires the help of CPU for LRN computation.

IV. System-Level Design

In this section, we present the system-level design of PRIME. The software-hardware interface framework is described. Then, we focus on the optimization of NN mapping and data allocation during compile time. Next, we introduce the operating system (OS) support for switching FF subarrays between memory and computation modes at run time.

A. Software-Hardware Interface

Figure 7 shows the stack of PRIME to support NN programming, which allows developers to easily configure the FF subarrays for NN applications[1]. From software programming to hardware execution, there are three stages: programming (coding), compiling (code optimization), and code execution. In the programming stage, PRIME provides application programming interfaces (APIs) so that they allow developers to: 1) map the topology of the NN to the FF subarrays, *Map_Topology*, 2) program the synaptic weights

[1]Due to the space limit, we only depict the key steps at high level while the design details of the OS kernel, compiler, and tool chains are left as engineering work.

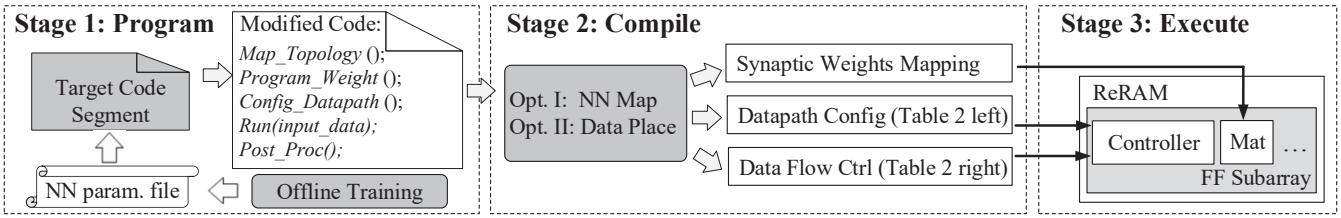

Figure 7. The software perspective of PRIME: from source code to execution.

into mats, *Program_Weight*, 3) configure the data paths of the FF subarrays, *Config_Datapath*, 4) run computation, *Run*, and 5) post-process the result, *Post_Proc*. In our work, the training of NN is done off-line so that the inputs of each API are already known (*NN param.file*). Prior work explored to implement training with ReRAM crossbar arrays [12], [70]–[74], and we plan to further enhance PRIME with the training capability in future work.

In the compiling stage, the NN mapping to the FF subarrays and the input data allocation are optimized (as described in Section IV-B). The output of compiling is the metadata for synaptic weights mapping, data path configuration, and execution commands with data dependency and flow control. The metadata is also the input for the execution stage. In the execution stage, PRIME controller writes the synaptic weights to the mapped addresses in the FF subarrays; then it (re-)configures the peripheral circuits according to the *Datapath Configure* commands (Table I left) to set up the data paths for computation; and finally, it executes *Data Flow Control* commands (Table I right) to manage data movement into or out of the FF subarrays at runtime.

B. Compile Time Optimization

1) NN Mapping Optimization: The mapping of the NN topology to the physical ReRAM cells is optimized during compile time. For different scales of NNs, we have different optimizations.[2]

Small-Scale NN: Replication. When an NN can be mapped to a single FF mat, it is small-scale. Although we can simply map a small-scale NN to some cells in one mat, the other cells in this mat may be wasted. Moreover, the speedup for very small NNs is not obvious, because the latency of the peripheral circuits may overwhelm the latency of matrix-vector multiplication on ReRAM cells. Our optimization is to replicate the small NN to different independent portions of the mat. For example, to implement a $128-1$ NN, we duplicate it and map a $256-2$ NN to the target mat. This optimization can also be applied to convolution layers. Furthermore, if there is another FF mat available, we can also duplicate the mapping to the second mat, and then the two mats can work simultaneously, as long as the Buffer subarray has enough bandwidth.

[2]In our technical report [75], we provide a detailed NN mapping example: how to map CNN-1 in Table III to PRIME.

Medium-Scale NN: Split-Merge. When an NN cannot be mapped to a single FF mat, but can fit to the FF subarrays of one bank, it is medium-scale. During the mapping at compile time, a medium-scale NN has to be split into small-scale NNs, and then their results are merged. For example, to implement a $512-512$ NN on PRIME with $256-256$ mats, it is split into four $256-256$ parts ($[M_{1,1}, M_{1,2}; M_{2,1}, M_{2,2}]$) and mapped to four different mats. After they finish computation, the results of $M_{1,1}$ and $M_{2,1}$ are added to get the first 256 elements of the final result, and the sum of the results of $M_{1,2}$ and $M_{2,2}$ forms the second 256 elements of the final result.

Large-Scale NN: Inter-Bank Communication. A large-scale NN is one NN that cannot be mapped to the FF subarrays in a single bank. Intuitively, we can divide it into several medium-scale trunks and map each trunk to the same bank serially in several stages. This naive solution requires reprogramming the FF subarrays at every stage, and the latency overhead of reprogramming may offset the speedup. Alternatively, PRIME allows to use multiple banks to implement a large-scale NN. These banks can transfer data to each other and run in a pipelined fashion to improve the throughput. Like prior work [76], the inter-bank data movement is implemented by exploiting the internal data bus shared by all the banks in a chip. PRIME controller manages the inter-bank communication, and can handle arbitrary network connections. If all the banks are used to implement a single NN, PRIME can handle a maximal NN with $\sim 2.7\times10^8$ synapses, which is larger than the largest NN that have been mapped to the existing NPUs (TrueNorth [34], 1.4×10^7 synapses). In Section V, we implement an extremely large CNN on PRIME, *VGG-D* [69] which has 1.4×10^8 synapses.

2) Bank-level Parallelism and Data Placement: Since FF subarrays reside in every bank, PRIME intrinsically inherits bank-level parallelism to speed up computation. For example, for a small-scale or medium-scale NN, since it can be fitted into one bank, the FF subarrays in all the banks can be configured the same and run in parallel. Considering FF subarrays in each bank as an NPU, PRIME contains 64 NPUs in total (8 banks$\times$8 chips) so that 64 images can be processed in parallel. To take advantage of the bank-level parallelism, the OS is required to place one image in each bank and to evenly distribute images to all the banks. As current page placement strategies expose memory latency or

bandwidth information to the OS [77], [78], PRIME exposes the bank ID information to the OS, so that each image can be mapped to a single bank. For large-scale NNs, they can still benefit from bank-level parallelism as long as we can map one replica or more to the spare banks.

C. Run Time Optimization

When FF subarrays are configured for NN applications, the memory space is reserved and supervised by the OS so that it is invisible to other user applications. However, during the runtime, if none or few of their crossbar arrays are used for computation, and the page miss rate is higher than the predefined threshold (which indicates the memory capacity is insufficient), the OS is able to release the reserved memory addresses as normal memory. It was observed that the memory requirement varies among workloads, and prior work has proposed to dynamically adjust the memory capacity by switching between SLC and MLC modes in PCM-based memory [79]. The page miss rate curve can be tracked dynamically by using either hardware or software approaches [80]. In our design, the granularity to flexibly configure a range of memory addresses for either computation or memory is crossbar array (mat): when an array is configured for computation, it stores multi-bit synaptic weights; when an array is used as normal memory, it stores data as single-bit cells. The OS works with the memory management unit (MMU) to keep all the mapping information of the FF subarrays, and decides when and how much reserved memory space should be released, based on the combination of the page miss rate and the utilization of the FF subarrays for computation.

V. Evaluation

In this section, we evaluate our PRIME design. We first describe the experiment setup, and then present the performance and energy results and estimate the area overhead.

A. Experiment Setup

Benchmark. The benchmarks we use (*MlBench*) comprise six NN designs for machine learning applications, as listed in Table III. *CNN-1* and *CNN-2* are two CNNs, and *MLP-S/M/L* are three multilayer perceptrons (MLPs) with different network scales: small, medium, and large. Those five NNs are evaluated on the widely used *MNIST* database of handwritten digits [67]. The sixth NN, *VGG-D*, is well known for ImageNet ILSVRC [69]. It is an extremely large CNN, containing 16 weight layers and 1.4×10^8 synapses, and requiring $\sim 1.6 \times 10^{10}$ operations.

PRIME Configurations. There are 2 FF subarrays and 1 Buffer subarray per bank (totally 64 subarrays). In FF subarrays, for each mat, there are 256×256 ReRAM cells and eight 6-bit reconfigurable SAs; for each ReRAM cell, we assume 4-bit MLC for computation while SLC for memory; the input voltage has 8 levels (3-bit) for computation while 2 levels (1-bit) for memory. With our input and synapse composing scheme, for computation, the input and output are 6-bit dynamic fixed point, and the weights are 8-bit.

Table III
THE BENCHMARKS AND TOPOLOGIES.

MlBench		*MLP-S*	784-500-250-10
CNN-1	conv5x5-pool-720-70-10	*MLP-M*	784-1000-500-250-10
CNN-2	conv7x10-pool-1210-120-10	*MLP-L*	784-1500-1000-500-10
VGG-D	conv3x64-conv3x64-pool-conv3x128-conv3x128-pool conv3x256-conv3x256-conv3x256-pool-conv3x512 conv3x512-conv3x512-pool-conv3x512-conv3x512 conv3x512-pool-25088-4096-4096-1000		

Table IV
CONFIGURATIONS OF CPU AND MEMORY.

Processor	4 cores; 3GHz; Out-of-order
L1 I&D cache	Private; 32KB; 4-way; 2 cycles access;
L2 cache	Private; 2MB; 8-way; 10 cycles access;
ReRAM-based Main Memory	16GB ReRAM; 533MHz IO bus; 8 chips/rank; 8 banks/chip; tRCD-tCL-tRP-tWR 22.5-9.8-0.5-41.4 (ns)

Table V
THE CONFIGURATIONS OF COMPARATIVES.

Description		Data path	Buffer
pNPU-co	Parallel NPU [17] as co-processor	16×16 multiplier 256-1 adder tree	2KB in/out 32KB weight
pNPU-pim	PIM version of parallel NPU, 3D stacked to each bank		

Methodology. We compare PRIME with several counterparts. The baseline is a CPU-only solution. The configurations of CPU and ReRAM main memory are shown in Table IV, including key memory timing parameters for simulation. We also evaluate two different NPU solutions: using a complex parallel NPU [17] as a co-processor (pNPU-co), and using the NPU as a PIM-processor through 3D stacking (pNPU-pim). The configurations of these comparatives are described in Table V.

We model the above NPU designs using Synopsys Design Compiler and PrimeTime with $65nm$ TSMC CMOS library. We also model ReRAM main memory and our PRIME system with modified NVSim [81], CACTI-3DD [82] and CACTI-IO [83]. We adopt Pt/TiO2-x/Pt devices [65] with $R_{\text{on}}/R_{\text{off}} = 1k\Omega/20k\Omega$ and 2V SET/RESET voltage. The FF subarray is modeled by heavily modified NVSim, according to the peripheral circuit modifications, i.e., write driver [84], sigmoid [63], and sense amplifier [64] circuits. We built a trace-based in-house simulator to evaluate different systems, including CPU-only, PRIME, NPU co-processor, and NPU PIM-processor.

B. Performance Results

The performance results for *MlBench* are presented in Figure 8. *MlBench* benchmarks use large NNs and require high memory bandwidth, and therefore they can benefit from PIM. To demonstrate the PIM advantages, we evaluate two pNPU-pim solutions: pNPU-pim-x1 is a PIM-processor

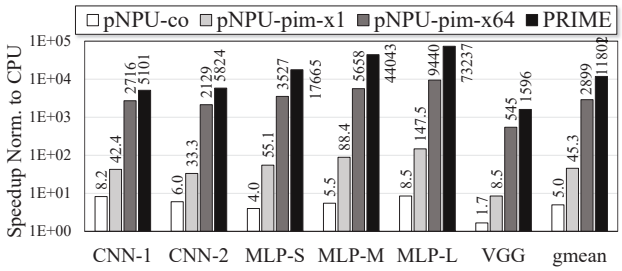

Figure 8. The performance speedups (*vs.* CPU).

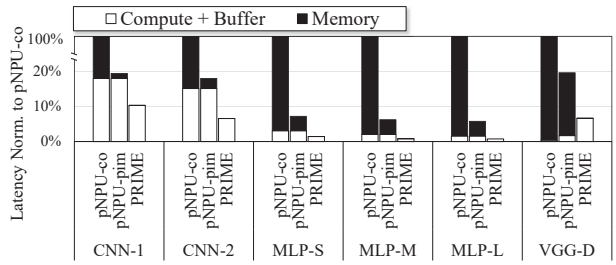

Figure 9. The execution time breakdown (*vs.* pNPU-co).

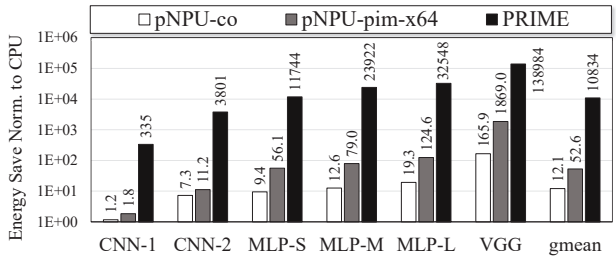

Figure 10. The energy saving results (*vs.* CPU).

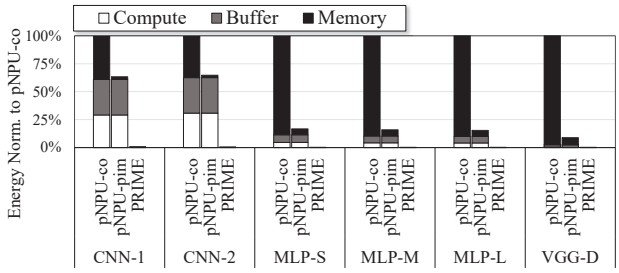

Figure 11. The energy breakdown (*vs.* pNPU-co).

with a single parallel NPU stacked on top of memory; and pNPU-pim-x64 with 64 NPUs, for comparison with PRIME which takes advantages of bank-level parallelism (64 banks). By comparing the speedups of pNPU-co and pNPU-pim-x1, we find that the PIM solution has a $9.1\times$ speedup on average over a co-processor solution. Among all the solutions, PRIME achieves the highest speedup over the CPU-only solution, about $4.1\times$ of pNPU-pim-x64's. PRIME achieves a smaller speedup in *VGG-D* than other benchmarks, because it has to map the extremely large *VGG-D* across 8 chips where the data communication between banks/chips is costly. The performance advantage of PRIME over the 3D-stacking PIM solution (pNPU-pim-x64) for NN applications comes from the efficiency of using ReRAM for NN computation, because the synaptic weights have already been pre-programmed in ReRAM cells and do not require data fetches from the main memory during computation. In our performance and energy evaluations of PRIME, we do not include the latency and energy consumption of configuring ReRAM for computation, because we assume that once the configuration is done, the NNs will be executed for tens of thousands times to process different input data.

Figure 9 presents the breakdown of the execution time normalized to pNPU-co. To clearly show the breakdown, we evaluate the results of pNPU-pim with one NPU, and PRIME without leveraging bank parallelism for computation. The execution time is divided into two parts, computation and memory access. The computation part also includes the time spent on the buffers of NPUs or the Buffer subarrays of PRIME in managing data movement. We find that pNPU-pim reduces the memory access time a lot, and PRIME further reduces it to zero. Zero memory access time does not imply that there is no memory access, but it means that the memory access time can be hidden by the Buffer subarrays.

C. Energy Results

The energy saving results for *MlBench* are presented in Figure 10. Figure 10 does not show the results of pNPU-pim-x1, because they are the same with those of pNPU-pim-x64. From Figure 10, PRIME shows its superior energy-efficiency to other solutions. pNPU-pim-x64 is several times more energy efficient than pNPU-co, because the PIM architecture reduces memory accesses and saves energy. The energy advantage of PRIME over the 3D-stacking PIM solution (pNPU-pim-x64) for NN applications comes from the energy efficiency of using ReRAM for NN computation.

Figure 11 provides the breakdown of the energy consumption normalized to pNPU-co. The total energy consumptions are divided into three parts, computation energy, buffer energy, and memory energy. From Figure 11, pNPU-pim-x64 consumes almost the same energy in computation and buffer with pNUP-co, but saves the memory energy by 93.9% on average by decreasing the memory accesses and reducing memory bus and I/O energy. PRIME reduces all the three parts of energy consumption significantly. For computation, ReRAM based analog computing is very energy-efficient. Moreover, since each ReRAM mat can store 256×256 synaptic weights, the cache and memory accesses to fetch the synaptic weights are eliminated. Furthermore, since each ReRAM mat can execute as large as a $256-256$ NN at one time, PRIME also saves a lot of buffer and memory accesses to the temporary data. From Figure 11, CNN benchmarks consume more energy in buffer and less energy in memory than MLP benchmarks. The reason is that the convolution layers and pooling layers of CNN usually have a small number of input data, synaptic weights, and output data, and buffers are effective to reduce memory accesses.

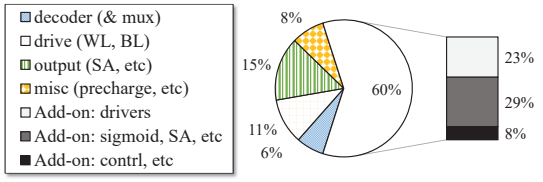

Figure 12. Area Overhead of PRIME.

D. Area Overhead

Given two FF subarrays and one Buffer subarray per bank (64 subarrays in total), PRIME only incurs 5.76% area overhead. The choice of the number of FF subarrays is a tradeoff between peak GOPS and area overhead. Our experimental results on *Mlbench* (except VGG-D) show that the utilities of FF subarrays are 39.8% and 75.9% on average before and after replication, respectively. For *VGG-D*, the utilities of FF subarrays are 53.9% and 73.6% before and after replication, respectively. Figure 12 shows the breakdown of the area overhead in a mat of an FF subarray. There is 60% area increase to support computation: the added driver takes 23%, the subtraction and sigmoid circuits take 29%, and the control, the multiplexer, and etc. cost 8%.

VI. Conclusion

This paper proposed a novel processing in ReRAM-based main memory design, PRIME, which substantially improves the performance and energy efficiency for neural network (NN) applications, benefiting from both the PIM architecture and the efficiency of ReRAM-based NN computation. In PRIME, part of the ReRAM memory arrays are enabled with NN computation capability. They can either perform computation to accelerate NN applications or serve as memory to provide a larger working memory space. We present our designs from circuit-level to system-level. With circuit reuse, PRIME incurs an insignificant area overhead to the original ReRAM chips. The experimental results show that PRIME can achieves a high speedup and significant energy saving for various NN applications using MLP and CNN.

References

[1] S. W. Keckler *et al.*, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, 2011.

[2] B. Akin *et al.*, "Data reorganization in memory using 3D-stacked DRAM," in *Proc. ISCA*, 2015.

[3] J. Ahn *et al.*, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proc. ISCA*, 2015.

[4] D. Zhang *et al.*, "TOP-PIM: Throughput-oriented programmable processing in memory," in *Proc. HPDC*, 2014.

[5] S. H. Pugsley *et al.*, "NDC: Analyzing the impact of 3D-stacked memory+ logic devices on mapreduce workloads," in *Proc. ISPASS*, 2014.

[6] J. T. Pawlowski, "Hybrid memory cube: breakthrough DRAM performance with a fundamentally re-architected DRAM subsystem,," in *Proc. of Hot Chips Symposium*, 2011.

[7] H.-S. Wong *et al.*, "Metal-oxide RRAM," *Proc. of the IEEE*, vol. 100, no. 6, pp. 1951–1970, 2012.

[8] A. Vincent *et al.*, "Spin-transfer torque magnetic memory as a stochastic memristive synapse," in *Proc. ISCAS*, 2014.

[9] G. Burr *et al.*, "Experimental demonstration and tolerancing of a large-scale neural network (165,000 synapses), using phase-change memory as the synaptic weight element," in *Proc. IEDM*, 2014.

[10] M. Hu *et al.*, "Hardware realization of BSB recall function using memristor crossbar arrays," in *Proc. DAC*, 2012.

[11] B. Li *et al.*, "Memristor-based approximated computation," in *Proc. ISLPED*, 2013.

[12] M. Prezioso *et al.*, "Training and operation of an integrated neuromorphic network based on metal-oxide memristors," *Nature*, vol. 521, no. 7550, pp. 61–64, 2015.

[13] Y. Kim *et al.*, "A reconfigurable digital neuromorphic processor with memristive synaptic crossbar for cognitive computing," *J. Emerg. Technol. Comput. Syst.*, vol. 11, no. 4, pp. 38:1–38:25, 2015.

[14] Z. Chen *et al.*, "Optimized learning scheme for grayscale image recognition in a RRAM based analog neuromorphic system," in *Proc. IEDM*, 2015.

[15] G. W. Burr *et al.*, "Large-scale neural networks implemented with non-volatile memory as the synaptic weight element: Comparative performance analysis (accuracy, speed, and power)," in *Proc. IEDM*, 2015.

[16] A. Coates *et al.*, "Deep learning with COTS HPC systems," in *Proc. ICML*, 2013.

[17] T. Chen *et al.*, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. ASPLOS*, 2014.

[18] Y. Chen *et al.*, "DaDianNao: A machine-learning supercomputer," in *Proc. MICRO*, 2014.

[19] P. Merolla *et al.*, "A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm," in *Proc. CICC*, 2011.

[20] C. Xu *et al.*, "Overcoming the challenges of crossbar resistive memory architectures," in *Proc. HPCA*, 2015.

[21] M.-J. Lee *et al.*, "A fast, high-endurance and scalable non-volatile memory device made from asymmetric Ta2O5-x/TaO2-x bilayer structures," *Nature Materials*, vol. 10, no. 8, pp. 625–630, 2011.

[22] C.-W. Hsu *et al.*, "Self-rectifying bipolar TaOx/TiO2 RRAM with superior endurance over 10^{12} cycles for 3D high-density storage-class memory," in *Proc. VLSIT*, 2013.

[23] M. K. Qureshi *et al.*, "Enhancing lifetime and security of PCM-based main memory with Start-Gap wear leveling," in *Proc. MICRO*, 2009.

[24] D. Niu *et al.*, "Design trade-offs for high density cross-point resistive memory," in *Proc. ISLPED*, 2012.

[25] A. Kawahara *et al.*, "An 8Mb multi-layered cross-point ReRAM macro with 443MB/s write throughput," in *Proc. ISSCC*, 2012.

[26] S. Yu *et al.*, "3D vertical RRAM - scaling limit analysis and demonstration of 3D array operation," in *Proc. VLSIT*, 2013.

[27] C. Xu *et al.*, "Architecting 3D vertical resistive memory for next-generation storage systems," in *Proc. ICCAD*, 2014.

[28] S. Yu *et al.*, "Investigating the switching dynamics and multilevel capability of bipolar metal oxide resistive switching memory," *Applied Physics Letters*, vol. 98, p. 103514, 2011.

[29] M.-C. Wu *et al.*, "A study on low-power, nanosecond operation and multilevel bipolar resistance switching in ti/zro2/pt nonvolatile memory with 1t1r architecture," *Semiconductor Science and Technology*, vol. 27, p. 065010, 2012.

[30] L. Zhang *et al.*, "SpongeDirectory: Flexible sparse directories utilizing multi-level memristors," in *Proc. PACT*, 2014.

[31] F. Alibart *et al.*, "High precision tuning of state for memristive devices by adaptable variation-tolerant algorithm," *Nanotechnology*, vol. 23, no. 7, p. 075201, 2012.

[32] H. Esmaeilzadeh *et al.*, "Neural acceleration for general-purpose approximate programs," in *Proc. MICRO*, 2012.

[33] R. St. Amant *et al.*, "General-purpose code acceleration with limited-precision analog computation," in *Proc. ISCA*, 2014.

[34] S. K. Esser *et al.*, "Cognitive computing systems: Algorithms and applications for networks of neurosynaptic cores," in *Proc. IJCNN*, 2013.

[35] J. Seo *et al.*, "A 45nm CMOS neuromorphic chip with a scalable architecture for learning in networks of spiking neurons," in *Proc. CICC*, 2011.

[36] T. M. Taha *et al.*, "Exploring the design space of specialized multicore neural processors," in *Proc. IJCNN*, 2013.

[37] A. Shafiee *et al.*, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proc. ISCA*, 2016.

[38] M. Jung *et al.*, "Design of a large-scale storage-class RRAM system," in *Proc. ICS*, 2013.

[39] D. C. Cireşan *et al.*, "Flexible, high performance convolutional neural networks for image classification," in *Proc. IJCAI*, 2011.

[40] J. Schmidhuber, "Multi-column deep neural networks for image classification," in *Proc. CVPR*, 2012.

[41] S. Sahin *et al.*, "Neural network implementation in hardware using FPGAs," in *Neural Information Processing*, vol. 4234, pp. 1105–1112, 2006.

[42] C. Farabet *et al.*, "Cnp: An FPGA-based processor for convolutional networks," in *Proc. FPL*, 2009.

[43] J.-Y. Kim *et al.*, "A 201.4 GOPS 496 mW real-time multi-object recognition processor with bio-inspired neural perception engine," *JSSC*, vol. 45, no. 1, pp. 32–45, 2010.

[44] D. Liu *et al.*, "Pudiannao: A polyvalent machine learning accelerator," in *Proc. ASPLOS*, 2015.

[45] S. Liu *et al.*, "Cambricon: An instruction set architecture for neural networks," in *Proc. ISCA*, 2016.

[46] C. Kozyrakis *et al.*, "Scalable processors in the billion-transistor era: IRAM," *Computer*, vol. 30, no. 9, pp. 75–78, 1997.

[47] D. Patterson *et al.*, "Intelligent ram (iram): The industrial setting, applications, and architectures," in *Proc. ICCD*, 1997.

[48] J. Draper *et al.*, "The architecture of the DIVA processing-in-memory chip," in *Proc. ICS*, 2002.

[49] M. Gokhale *et al.*, "Processing in memory: The terasys massively parallel PIM array," *Computer*, vol. 28, no. 4, pp. 23–31, 1995.

[50] D. Elliott *et al.*, "Computational RAM: The case for SIMD computing in memory," in *Workshop on Mixing Logic and DRAM at ISCA*, 1997.

[51] T. Yamauchi *et al.*, "A single chip multiprocessor integrated with DRAM," in *Workshop on Mixing Logic and DRAM at ISCA*, 1997.

[52] M. Oskin *et al.*, "Active pages: a computation model for intelligent memory," in *Proc. ISCA*, 1998.

[53] R. Balasubramonian *et al.*, "Near-data processing: Insights from a micro-46 workshop," *Micro, IEEE*, vol. 34, no. 4, pp. 36–42, 2014.

[54] R. Nair *et al.*, "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, pp. 17:1–17:14, 2015.

[55] Z. Guz *et al.*, "Real-time analytics as the killer application for processing-in-memory," in *Proc. WoNDP*, 2014.

[56] N. S. Mirzadeh *et al.*, "Sort vs. hash join revisited for near-memory execution," in *Proc. ASBD*, 2015.

[57] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *Proc. VLSIT*, 2012.

[58] D. U. Lee *et al.*, "A 1.2V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV," in *Proc. ISSCC*, 2014.

[59] S. Li *et al.*, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proc. DAC*, 2016.

[60] F. Alibart *et al.*, "Hybrid CMOS/nanodevice circuits for high throughput pattern matching applications," in *Proc. AHS*, 2011.

[61] Q. Guo *et al.*, "A resistive TCAM accelerator for data-intensive computing," in *Proc. MICRO*, 2011.

[62] Q. Guo *et al.*, "AC-DIMM: Associative computing with STT-MRAM," in *Proc. ISCA*, 2013.

[63] B. Li *et al.*, "RRAM-based analog approximate computing," *TCAD*, vol. 34, no. 12, pp. 1905–1917, 2015.

[64] J. Li *et al.*, "A novel reconfigurable sensing scheme for variable level storage in phase change memory," in *Proc. IMW*, 2011.

[65] L. Gao *et al.*, "A high resolution nonvolatile analog memory ionic devices," in *Proc. NVMW*, 2013.

[66] M. Hu *et al.*, "Dot-product engine: Programming memristor crossbar arrays for efficient vector-matrix multiplication," in *ICCAD'15 Workshop on "Towards Efficient Computing in the Dark Silicon Era"*, 2015.

[67] Y. Lecun *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[68] M. Courbariaux *et al.*, "Low precision storage for deep learning," *CoRR*, vol. abs/1412.7024, 2014.

[69] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. ICLR*, 2015.

[70] F. Alibart *et al.*, "Pattern classification by memristive crossbar circuits using ex situ and in situ training," *Nature communications*, vol. 4, 2013.

[71] M. Hu *et al.*, "BSB training scheme implementation on memristor-based circuit," in *Proc. CISDA*, 2013.

[72] B. Li *et al.*, "Training itself: Mixed-signal training acceleration for memristor-based neural network," in *Proc. ASP-DAC*, 2014.

[73] B. Liu *et al.*, "Digital-assisted noise-eliminating training for memristor crossbar-based analog neuromorphic computing engine," in *Proc. DAC*, 2013.

[74] B. Liu *et al.*, "Reduction and IR-drop compensations techniques for reliable neuromorphic computing systems," in *Proc. ICCAD*, 2014.

[75] P. Chi *et al.*, "Processing-in-memory in ReRAM-based main memory," *SEAL-lab Technical Report*, no. 2015-001, 2015.

[76] V. Seshadri *et al.*, "RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization," in *Proc. MICRO*, 2013.

[77] B. Verghese *et al.*, "Operating system support for improving data locality on CC-NUMA compute servers," in *Proc. ASPLOS*, 1996.

[78] N. Agarwal *et al.*, "Page placement strategies for GPUs within heterogeneous memory systems," in *Proc. ASPLOS*, 2015.

[79] M. K. Qureshi *et al.*, "Morphable memory system: A robust architecture for exploiting multi-level phase change memories," in *Proc. ISCA*, 2010.

[80] P. Zhou *et al.*, "Dynamic tracking of page miss ratio curve for memory management," in *Proc. ASPLOS*, 2004.

[81] X. Dong *et al.*, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *TCAD*, vol. 31, no. 7, pp. 994–1007, 2012.

[82] K. Chen *et al.*, "CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory," in *Proc. DATE*, 2012.

[83] N. P. Jouppi *et al.*, "CACTI-IO: CACTI with off-chip power-area-timing models," in *Proc. ICCAD*, 2012.

[84] C. Xu *et al.*, "Understanding the trade-offs in multi-level cell ReRAM memory design," in *Proc. DAC*, 2013.