Optimizing Weight Mapping and Data Flow for Convolutional Neural Networks on Processing-In-Memory Architectures

Xiaochen Peng, *Student Member, IEEE*, Rui Liu, *Student Member, IEEE*, and Shimeng Yu, *Senior Member, IEEE*

***Abstract*— Recent state-of-the-art deep convolutional neural networks (CNNs) have shown remarkable success in current intelligent systems for various tasks, such as image/speech recognition and classification. A number of recent efforts have attempted to design custom inference engines based on processing-in-memory (PIM) architecture, where the memory array is used for weighted sum computation, thereby avoiding the frequent data transfer between buffers and computation units. Prior PIM designs typically unroll each 3D kernel of the convolutional layers into a vertical column of a large weight matrix, where the input data needs to be accessed multiple times. In this paper, in order to maximize both weight and input data reuse for PIM architecture, we propose a novel weight mapping method and the corresponding data flow which divides the kernels and assign the input data into different processing-elements (PEs) according to their spatial locations. As a case study, resistive random access memory (RRAM) based 8-bit PIM design at 32 nm is benchmarked. The proposed mapping method and data flow yields ~2.03x speed up and ~1.4x improvement in throughput and energy efficiency for ResNet-34, compared with the prior design based on the conventional mapping method. To further optimize the hardware performance and throughput, we propose an optimal pipeline architecture, with ~50% area overhead, it achieves overall 913x and 1.96x improvement in throughput and energy efficiency, which are 132476 FPS and 20.1 TOPS/W, respectively.**

***Index Terms*— Convolutional neural network, processing in memory, hardware accelerator, resistive random access memory.**

I. Introduction

THE neuro-inspired computing has been utilized in a broad range of applications and cloud services. In particular, the deep convolutional neural networks (CNNs) have shown remarkable breakthroughs in various applications, including speech recognition and image classification. To achieve higher accuracy, the popular state-of-the-art CNNs tend to have a large number of high-dimensional convolutional layers, where the input data could exceed hundreds of megabytes, and need over tens of thousands operations for each input pixel. Hence, the conventional von Neumann platforms (e.g. CPUs/GPUs and/or FPGAs) are facing challenges for implementing deep CNN operations, which require high bandwidth and power consumption for intensive data processing and communication. It is thus desirable to design custom CNN accelerators, to implement the large and deep algorithms on-chip efficiently, and achieve significant speed-up and power reduction compared with the conventional platforms.

A number of recent efforts have attempted to design CMOS application-specific-integrated-circuit (ASIC) accelerators by academia, such as Eyeriss [1], Envision [2] and DNPU [3], where a group of multiply-accumulate (MAC) units are utilized to perform digital dot-product, and accumulate the partial-sums to get the output data. At each operation, both weight and input data are read out from on-chip buffers and transferred to corresponding MAC unit, after the output data are accumulated, they will be sent back to the buffers and wait to be scheduled for next operation. Among the ASIC accelerators, TPU [4] from Google which employs 8-bit digital matrix multipliers and systolic array based data flow, has achieved commercial success.

However, in most CMOS based accelerators, the data storage is based on SRAM buffers. It is well known that normally a single SRAM cell could occupy $>150F^2$ (where F is the technology feature size), while the data load-in and read-out are operated in a row-by-row fashion for such conventional SRAM arrays. Due to the limited on-chip SRAM capacity (typically a few MB), the accelerators have to load in large amount of input and weight data from off-chip memory (i.e. DRAM). This leads to high energy consumption for data movement, especially on interconnect, on-chip buffer access and off-chip DRAM access.

Therefore, an area- and energy-efficient approach called processing-in-memory (PIM) is proposed, where the computation is embedded into the memory directly, i.e., by analog computation using the column current summation. PIM could be applied to SRAM design with modified bit-cells to enable parallel access [5]. However, for inference engine where instant on and dynamic power-gating is needed, the volatility of SRAM is a major drawback. Instead, emerging non-volatile

Manuscript received May 16, 2019; revised September 27, 2019; accepted December 3, 2019. Date of publication December 25, 2019; date of current version April 1, 2020. This work was supported in part by the ASCENT, one of the SRC/DARPA JUMP Centers, under Grant NSF-CCF-1903951 and Grant NSF-CCF-1740225, in part by the SRC under Contract 2018-NC-2762, and in part by Samsung Semiconductor, Inc. This article was recommended by Associate Editor M.-F. Chang. *(Corresponding author: Xiaochen Peng.)*

X. Peng and S. Yu are with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA (e-mail: xpeng76@gatech.edu; shimeng.yu@ece.gatech.edu).

R. Liu was with the School of Electrical, Computer, and Energy Engineering, Arizona State University, Tempe, AZ 85281 USA. She is now with the Design Group of Synopsys, Mountain View, CA 94043 USA.

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSI.2019.2958568

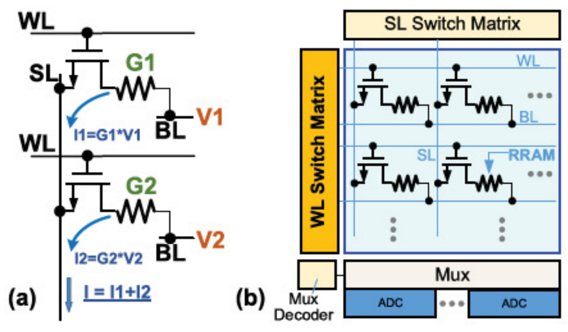

Fig. 1. (a) Using a sense-line (SL) current to perform sum of dot-products. (b) Pseudo-crossbar array with peripheral circuits Switch Matrix, MUX+MUX Decoder and ADC.

memory (eNVMs) [6] based PIM designs are of great interests. Among the emerging technologies, the resistive random access memory (RRAM) has been proved to be a promising candidate for practical large-scale integration. For example, the embedded RRAM processes have been demonstrated at 40nm by TSMC [7] and at 22nm by Intel [8]. With the mature one-transistor-one-resistor (1T1R) pseudo-crossbar structure, the analog vector-matrix multiplication can be efficiently performed by exploiting the multilevel states of RRAM devices as "analog" synapses [9].

Fig. 1 shows the pseudo-crossbar array that can perform "analog" matrix-vector multiplication, where the word lines (WLs) could be turned on simultaneously to achieve parallel dot-product computation. The input vectors are represented as voltage inputs of the bit-lines (BLs), and the dot-product value will be the current passing through each RRAM cell, such that the sum of dot-products in each column will be naturally accumulated along the SL, and finally obtained by an analog-to-digital converter (ADC) at the end of each SL. During the write operation (i.e. weight loading), WL could be individually activated to enable row-by-row programming.

To implement the convolutional layers, a "naïve" weight mapping method is to unroll each 3D kernel into a long vertical column, such that a group of 3D kernels could be unrolled and mapped to a large 2D matrix. In the deep CNNs, the depth of input and output feature maps tends to become deeper, which could be thousands by thousands. Hence, using a single large matrix to implement one convolutional layer may cause slow-access and extra energy consumption, thus, array partitioning [10] can be introduced to parallelize the computation into multiple sub-arrays. It is noted that the input data will be reused significantly for convolution as the sliding window moves over the input feature map, thus it is necessary to mitigate the unnecessary latency and energy waste in interconnect and buffers due to this input data reuse. In this work, we focus on a RRAM based pipelined PIM architecture that supports 8-bit weight and 8-bit activation for CNN inference, which exploits a novel weight mapping and data flow to maximize both input and weight data reuse. We benchmark the hardware performance of the proposed architecture, using a circuit-level macro model called NeuroSim [11] at 32 nm CMOS node (a practical node considering the recent demonstrations by TSMC [7] and Intel [8]), and compare the new design with a baseline which is based on the conventional "naïve" mapping method and data flow. This is an extension of our prior conference paper [12]. The new materials added to this journal paper include architecture design and analysis based on ResNet [13] for ImageNet [14], hybrid mapping method to maximize memory utilization and pipeline system to speed up the process and minimize leakage, which helps to improve both the throughput and energy efficiency.

II. Related Work

In this section, we provide a brief survey on several related works on PIM accelerator designs. We compare these state-of-the-art works with our approach, which mainly focus on optimizing the mapping method and data flow for PIM accelerators.

ISAAC [15] design used crossbar memory arrays as analog dot-product engines for CNN acceleration. In particular, the supporting digital components required in analog PIM accelerators were defined. The balance was explored between high throughput and area overhead of ADCs, DACs and on-chip eDRAM buffers. A balanced inter-layer pipelined system was implemented, yielding significant improvement of throughput, energy and computational density, compared with the prior DaDianNao [16] architecture. PRIME [17] proposed an architecture based on RRAM memory arrays, where part of the memory arrays was enabled for neural functional computation, and the others served as memory to provide a larger working memory space. With reuse of the peripheries, the benchmark results showed that PRIME could achieve speed-up and energy saving for various neural networks.

PipeLayer [18] revealed the limitation of ISAAC for on-chip training, where the notion of batch-based training limited the number of images that could be processed efficiently. It analyzed the data dependency and transformation in training phase, and propose a highly intra-layer parallel design, which enabled the highly pipelined execution of both on-chip training and testing phases. The benchmark results showed that the proposed design could achieve $\sim 42\times$ speed-up and $\sim 7\times$ energy saving compared with GPU.

Moreover, since the requirements of computation and storage for deep CNNs are usually high, there is great attention to develop low-precision neural networks to avoid the significant overhead of hardware resource, networks such as BNN [19] and XNOR-Net [20] have been proved to achieve relatively reasonable accuracy as well as significant reductions on computational resources. The prior work [21] proposed an optimized accelerator that was tailored for such binary neural networks, which employed bitwise convolution with massive parallelism to achieve high energy efficiency. The benchmark results show that, based on digital RRAM-crossbar, the design achieved $\sim 1.6\times$ faster and $\sim 296\times$ more energy efficiency than GPU.

However, recently deep CNNs tend to grow deeper and deeper, which requires more hardware resources on-chip, and leads to a problem of area overhead even for the compact PIM accelerators. Unlike the conventional designs based on 2D architectures, in [22], the authors proposed a multilayer CMOS-RRAM accelerator based on 3D heterogeneous integration that could further support more parallelism; and with adoption of ternary neural networks, to significantly compress the size of synaptic weights and neural activations. The proposed accelerator could achieve remarkable improvements of throughput and energy efficiency, as well as the area saving compared with 3D CMOS-ASIC implementation, or 2D CPU implementation. This work presented a promising solution to further overcome the area overhead of PIM architectures for large-scale deep CNNs.

These prior works mostly focused on architecture optimization for circuit-level or system-level, and barely discussed about the detailed reshape process of high-dimensional kernels. In this work, we mainly focus on optimizing the mapping method of weight matrix on PIM accelerators and data flow to maximize the input data reuse. To benchmark our approach, we implement a RRAM-based PIM accelerator for 8-bit CNN inference. The reason for choosing 8-bit precision for weights and activations is because recent works [23] and [24] provided fully 8-bit quantization of all the data paths including weights, activations, gradients and errors, which realized bit-wise operations for both training and testing, and achieved competitive accuracy on ResNet18/34/50 models on ImageNet dataset, compared with conventional floating point frameworks.

III. Background

A. Computation of CNN in PIM Architectures

The process of how the convolutional layer computes the outputs is shown in Fig. 2: in layer<n>, the size of input feature maps (IFMs) is W×W×D (where D is the depth of input feature channel), which are the outputs from layer<n-1>. The size of each 3D kernel is K×K×D, with kernel depth of N (i.e. there are N such 3D kernels), thus the total size of the kernels in layer<n> will be K×K×D×N. To get the outputs, a group of IFMs (with size K×K×D) will be selected at each time, and to be multiplied and accumulated with N kernels with size K×K×D, then each of them will generate a 1×1×1 output, the output from the top kernel (shown as light orange cube) goes to the front, and the output from the bottom kernel (shown as dark orange cube) goes to the back, thus, in total there will be 1×1×N outputs.

As shown in Fig. 2, it could be considered that, the kernels are "sliding over" the IFMs, and perform elementwise multiplications with a certain stride, and then the products of each elementwise multiplications in each 3D kernel will be summed up to get the final outputs, it is easy to detect that during the "kernel sliding", part of the input data will be reused for the computation of the next output. If we consider same-padding of the IFMs with a stride equals to one, it is straightforward to know that the output feature maps (OFMs) of layer<n+1> will be W×W×N, here the N (kernel depth) defines the depth of output feature channel.

B. A Conventional Weight Mapping Method of CNN

In the PIM architectures, the kernels (weights) will be mapped into crossbar arrays as conductance of each cross-point (memory device), such that, this 3D elementwise multiplication will be transformed to a dot-product multiplication. Since all the dot-products in each 3D kernel will be summed up to get the final output, it is easier to get the final output by just unrolling each 3D kernel into a long vertical column, and with the nature of crossbar array, the partial sums from each cross-point will be automatically summed through each SL. In this way, a single 3D kernel will be mapped into a long column, thus the total kernels in each convolutional layer will form a large weight matrix.

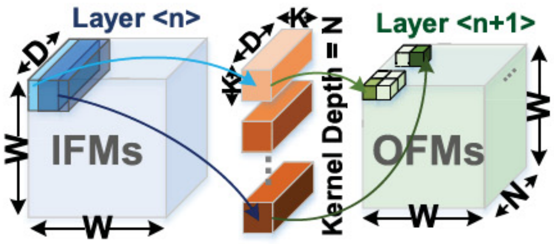

Fig. 2. Example of convolutional layer computation.

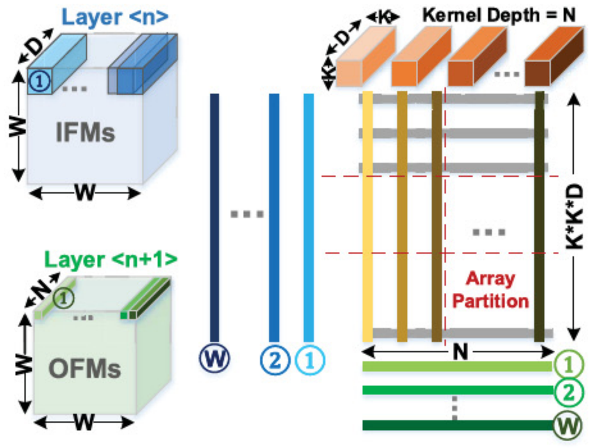

Fig. 3. Conventional "naïve" mapping method of input and weight data, with kernel moving in multiple cycles [25].

Fig. 3 shows this conventional weight mapping method [25] (as baseline in this work). With the same example in Fig. 2, one 3D kernel with size K×K×D could be unrolled to a long vertical column whose length equals to K×K×D, and with kernel depth equals to N, there are N such vertical columns in total. Thus, in layer<n>, the kernels could be mapped into a large weight matrix, whose length and width equals to K×K×D and N, respectively. In this way, to get the total OFMs, at each cycle, a part of IFMs (shown in blue cubes) will be multiplied with each 3D kernels. For example, at the beginning, the right-top part of IFMs (dark blue cube) will be unrolled into a long vertical column and applied to the rows of that large weight matrix. If we assume a single OFM has size of W×W, with channel depth of N, at each cycle, there are N such OFM in total, we call the front-channel OFM as the 1st OFM, and the back-channel one as the Nth OFM. In this way, the sum of dot-products from the first kernel (leftmost column in the weight matrix) will be the 1st element in the first OFM, the sum of dot-products from the second kernel will be the 1st element in the second OFM, and so on, thus, at the first cycle, we could get the 1st elements of every OFM channel, from front to back (as shown in light green row in

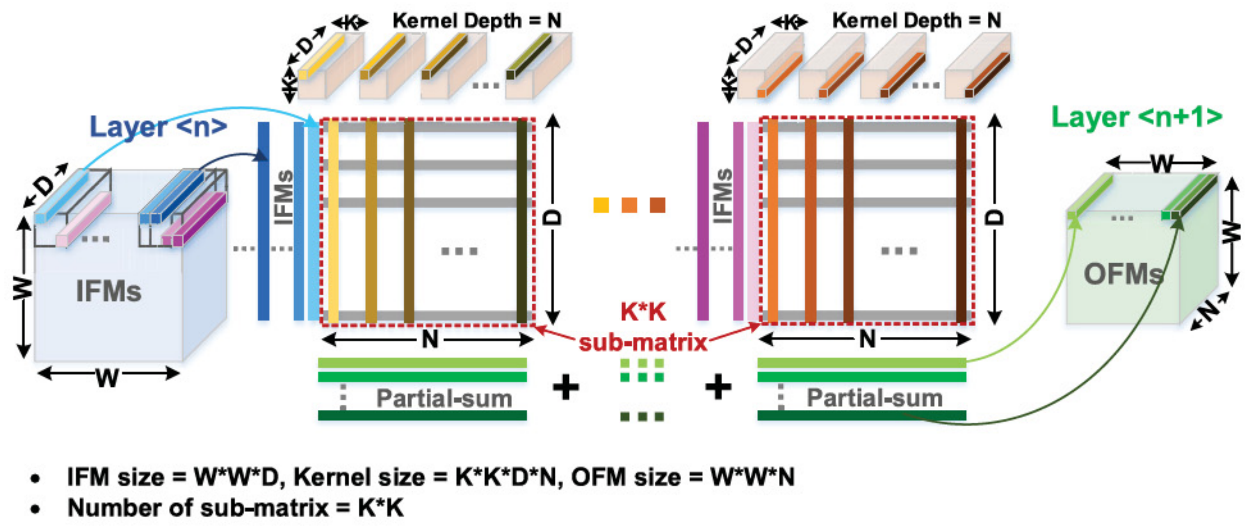

Fig. 4. Proposed novel mapping method that map the weights along the spatial location to a group of sub-matrix. K×K kernel is mapped to K×K sub-matrices (or processing elements, PEs). Partial sums are accumulated by adder tree.

size 1×1×N). In the same way, at the second cycle, the kernels will "slide over" the inputs with a stride (equals to one in this example), after the dot-product operation, we will get all the 2ed elements in each OFM channel. Thus, to generate the total OFMs in layer<n>, we need to "slide" the kernels over the IFMs by W×W times, i.e. we need W×W cycles to finish the computation. It should be noted that, during the entire operation, a part of the IFMs used in the prior cycle will always be reused at current cycle. Considering about the huge amount of dot-product operations in convolutional layers, these frequent revisiting of input data could cause a significant energy and latency waste.

To practically map and operate large convolutional layers on chip, array partitioning [10] is introduced, which could cut a single large matrix into several sub-arrays, and parallelize the computation efficiently. In popular deep CNNs, the kernel size normally varies a lot across different layers, thus the unrolled weight matrix size is quite different, which leads to various number of sub-arrays to be used to represent different layers. In this case, it is very difficult for us to reuse the unrolled input data among various number of sub-arrays, since it needs different control circuits and signals for different layer, to send a certain segment of input data to a specific group of sub-arrays. In other words, unless we customize the hardware for a specific neural network, it is impractical to design the reconfigurable interconnects among different sub-arrays (for input data reuse) for general deep CNNs. Therefore, it is crucial to design a novel mapping method and data flow that could maximize input data reuse, where the weight data and input data can be mapped according to their spatial location, and the hardware can actually "slide over" the input data with a global control unit.

IV. Novel Weight Mapping and Data Flow

A. A Novel Weight Mapping Method

To realize the input data reuse practically, in this work, we propose a novel mapping method as Fig. 4 shows. Unlike the conventional mapping method, where all the 3D kernels are unrolled into a large matrix, we map the weights at different spatial location of each kernel into different sub-matrices. In other words, with the same example as Fig. 2 and Fig. 3 show (where the kernel size is K×K×D×N), if we cut each K×K×D kernel along its first and second dimension, we will get several 1×1×D partitioned kernel data, and for each kernel, there are K×K of them. According to the spatial location of partitioned data in each kernel, we define which group of these partitioned data should belong to. For example, all the partitioned data who are locating at the left-most and top-most position at each kernel, will be considered as one group, and implemented into one sub-matrix, the height and width of each sub-matrix should equal to 1×1×D and N. Hence, K×K sub-matrixes are needed for the kernels (whose first and second dimension equal to K and K), since each sub-matrix has size D×N, the size of total weight matrix will be K×K×D×N, which equals to the size of unrolled matrix from conventional mapping method (as Fig. 3 shows). Similarly, the input data which should be assigned to various spatial location in each kernel, will be sent to the corresponding sub-matrix respectively.

In deep CNNs, the dimension of kernels can be very large, thus, even though the kernels are partitioned and mapped into different sub-matrix, the size of each sub-matrix could still be relatively large (around 512×512). In this case, each sub-matrix can be represented by a group of sub-arrays which makes the sub-matrix to be large enough to hold the kernels from various layers, as the Fig. 6 (c) shows, such group of sub-arrays with the necessary input and output buffers and accumulation modules can be defined as a processing element (PE). It should be noted that array partitioning within the PE is helpful to maximize the memory utilization. Since the kernels from some convolutional layers (normally the first couple of layers) could be shallow and small, which will not fill the entire PE and cause memory waste. With a group of partitioned sub-arrays, those shallow kernels could be duplicated in different sub-arrays and take multiple input

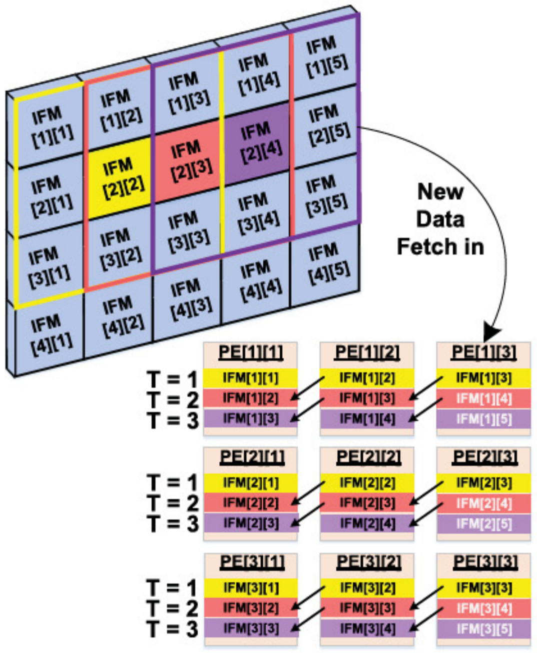

Fig. 5. An example of how the kernels "slide over" the IFMs, and how the IFMs being transferred among PEs in multiple cycles.

data to generate independent outputs simultaneously. In this case, those shallow convolutional layers which have shallow kernels but large IFMs could be speeded up significantly.

B. Data Flow to Maximize IFM Reuse

Based on the proposed novel mapping method, which cuts the kernels into several PEs according to their spatial locations, and assign the input data into corresponding ones, it is possible to efficiently reuse the IFMs among these PEs. As shown in Fig. 5, it is an example of mapping and processing a convolutional layer with the 3×3 kernel. At the very beginning, all the input data are assigned to the corresponding PEs, i.e. at T=1, a input vector with length D, called IFM[1][1] is assigned to PE[1][1], similarly IFM[1][2] is assigned to PE[1][2], IFM[1][3] is assigned to PE[1][3], and so on. From each PE, a partial sum of size 1×1×N will be generated, and these partial sums from these 9 PEs will be summed up along their third dimension (number of columns in each PE, which equals to N in this example) by adder trees, and get the final OFM[2][2] (with size 1×1×N). At the next cycle, the IFMs which will be used for the next computation are transferred to the neighboring PEs, and the useless IFMs will be released. For example, at T=2, IFM[1][1] is released (will not be used anymore), IFM[1][2] is transferred from PE[1][2] to PE[1][1], IFM[1][3] is transferred from PE[1][3] to PE[1][2], and so on. Finally, by the end of second cycle, the partial sums from the 9 PEs will be accumulated and generate the OFM[2][3]. It is clear in the example that, with the novel data flow, only 1/3 input data are newly introduced from upper-level buffers (e.g. global buffer or off-chip DRAM), and 2/3 of them are reused from the neighbor PEs.

Thus, by passing the used IFMs in the same direction as the kernel "slides over" the inputs, the IFMs can be reused among the PEs efficiently. For a more general concept, with kernel size K×K, and stride equals to S, every time only S/K of required input data are newly transferred, the rest (K-S)/K of input data are reused among neighboring PEs.

V. Architecture Design

A. Overall Architecture

In Fig. 6 (a), it shows the top-level diagram of a RRAM-based PIM architecture based on the proposed novel mapping method and data flow, which contains multiple tiles, accumulation units, pooling and activation units, L3 buffer and the global control. With the mature embedded RRAM process [8], it is possible to implement all the kernels of the entire deep CNN algorithm on chip, the accumulation units are needed to sum up the partial sums from various tiles, and the summed up results will then be sent to activation units (or/and pooling units if necessary), and finally sent back to L3 buffer, the global control will schedule the generated feature map in L3 buffer to another group of tiles to operate the following layers.

Fig. 6 (b) shows a single tile, which contains multiple PEs with routers and L2 buffer. Within a tile, the routers make it possible to communicate among PEs and transfer partial sums from PEs to accumulation units. In the novel mapping, the number of PEs equal to the kernel size, e.g. 9 PEs for 3×3 kernel. Fig. 6 (c) shows a PE which contains 16 sub-arrays and L1 buffer, where the 16 sub-arrays could support enough storage for the deepest layer in ResNet (kernel depth is 512), the intra-PE accumulation units are used to sum up partial sums from sub-arrays, the output buffer will collect the accumulated partial sums and wait to be sent out by inter-PE routers.

To implement a Y-bit analog synaptic weight, we could use N × M-bit cells (where Y=N×M) as a group, such that each column in the group could represent from LSB to MSB of the partial sums. Fig. 6 (d) shows that in this work, we use 4× 2-bit RRAM cells to implement 8-bit weights, as 2-bit RRAM has been demonstrated in prototype chip [26]. To eliminate the challenges of using digital-to-analog converter (DAC) to represent the analog input voltages in a small dynamic range, which could also leads to inaccuracy with RRAM nonlinear I-V relation [27], we represent the 8-bit fixed-point neuron activations with eight sequential input voltage pulses through eight cycles. For each row, if the input vector bit is 1, then the row will be selected for weighted-sum operation (read out), otherwise the row will be skipped. To access all the cells on the selected row, the WLs are activated through the WL switch matrix, and the weighted sum currents from the columns are digitalized by ADCs. Then we use the shift-add and register modules to shift and accumulate the partial sums of the 8-bit sequential inputs.

Since the ADCs always dominate the area, it is impractical to use high-precision ADCs at the edge of RRAM sub-arrays, we have to truncate the precision of partial sums for ADCs to minimize the area and energy overhead. According to the prior work on binary neural network where 3-bit ADC was reportedly sufficient for partial-sum-collecting

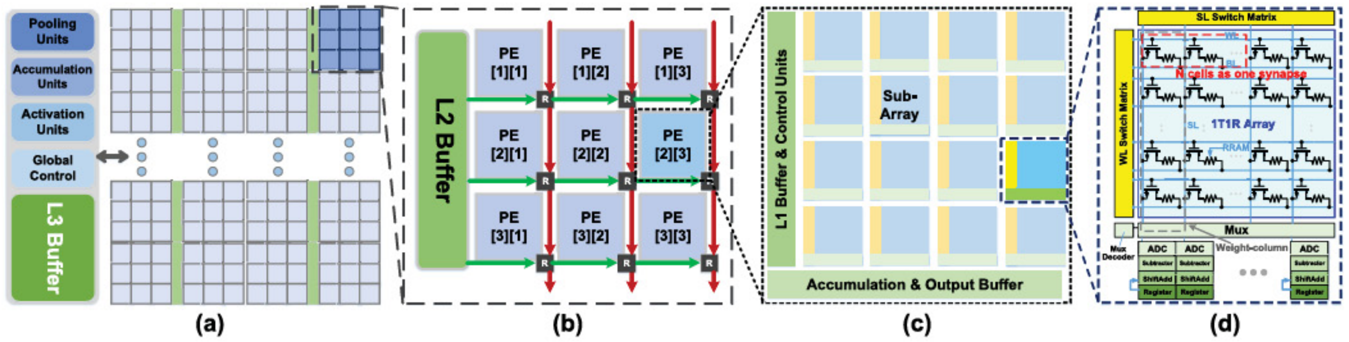

Fig. 6. The diagram of (a) RRAM based PIM architecture; (b) a tile with multiple PEs (e.g. 9 PEs for 3×3 kernel), routers and L2 buffer; (c) a PE contains 16 subarrays, L1 buffer and control units, accumulation modules and output buffer; (d) parallel 8-bit RRAM sub-array (e.g. $4 \times$ 2-bit/cell RRAM cells as one synapse).

in 128×128 sub-array [28], we analyzed the effect of ADC precision on 8-bit networks, as a result, the 5-bit ADC is sufficient to provide high accuracy (~90%) in 128×128 sub-array with 2-bit RRAM.

For this inference engine, we assume the write-verify protocol [29] is applied to accurately tune the conductance to be the target value, so the non-ideality of weight-update (such as non-linearity, asymmetry and variations) will not affect the actual conductance value. In addition, since the weight range in the network is trained to be zero-centered, for example, in range $[-1, 1]$, we use the minimum conductance to represent the -1, and use the maximum conductance to represent the $+1$, then we use several dummy columns to map the zero center of weight, and effectively cancel the off-current for limited on/off ratio for the RRAM devices.

The sub-array size is set to 128×128, to guarantee maximum memory efficiency with the following considerations. In the most shallow convolutional layers, the depth of feature channels (which defines number of rows in each sub-matrix) could be smaller than 128, thus, if we use larger sub-array size (i.e. 256×256), at least half of the memory will not be used. Also, we do not further downsize our sub-array to 64×64, it is well known that the periphery dominates the area of sub-array, and thus degrading the area efficiency (i.e. comparing four 64×64 sub-arrays with one 128×128 sub-array, the area efficiency of the latter is higher).

B. Memory Efficiency

With the novel mapping method, we could consider that, for each layer, the depth of IFM defines the number of rows of each sub-matrix, and the kernel depth defined the number of columns of each sub-matrix, while the kernel size defined the number of sub-matrixes. In this work, we assume the PE as minimum computation unit for each layer, i.e. we do not mix more than one convolutional layers into same PE. Otherwise, extra controllers are needed to be implemented inside each PE, to assign data of different layers to separate L1 and output buffer locations. This could cause "over-design" of PEs, as only a few shallow layers can be mixed in one PE. Moreover, mixing more than one layers in one PE could make it harder to pipeline, since it requires more buffers, even accumulation and activation units in side each PE, which is impractical for chip design. As only the shallow layers has

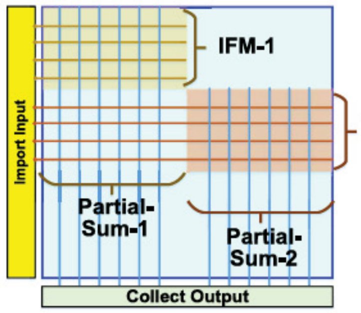

Fig. 7. Example of sub-matrix duplication inside a sub-array.

small sub-matrixes which cannot fill the entire PE, it is not necessary to implement extra control units inside each PE, to assign different sub-arrays for different layers. However, normally the shallow layers has large IFMs, which need much longer operation time compared with the deeper layers, thus, it is desired to speed up the shallow layers. While the sub-matrixes cannot fully filled inside each PE for these layers, we could duplicate the sub-matrixes as much as possible, which makes it possible for more IFMs to be operated at same time, thus improve the memory efficiency and speed up the entire process.

As Fig. 1 (b) shows, the BL switch matrix is used to import input vectors, while the ADCs (and the following peripheries) are used to collect the partial-sums. It should be noted that, the purpose of sub-matrix duplication is to operate more IFMs at the same time. With the pseudo-crossbar structure, the number of duplication inside one sub-array is limited by the hardware resource of data-import and output-collect. Fig. 7 shows an example of sub-matrix duplication inside a sub-array. When a sub-matrix is implemented from the left-top of the sub-array, its duplication cannot be placed below it (i.e. share the same SLs), since the outputs generated from different input vectors cannot be added along the SLs. Similarly, the duplication cannot be placed beside it (i.e. share the same WLs and BLs), since it is meaningless to calculate same outputs multiple times. Thus, the maximum number of duplication of sub-matrix inside the sub-array can be defined as:

$$Min\left(\frac{\#\ row\ in\ subarray}{\#\ row\ in\ submatrix}, \frac{\#\ column\ in\ subarray}{\#\ column\ in\ submatrix}\right)$$

It is noticeable that, in most deep CNNs, the channel depth of IFM from first convolutional layer is normally set to 3 (since input is RGB image), and kernel depth is normally much larger than that value (for example, in VGG [30] and ResNet [13], the kernel depth of first convolutional layer is 64). Thus, with the sub-array size to be 128×128, and number of sub-arrays inside a PE is 16, if we use novel mapping method to map the first convolutional layer in ResNet, we need 49 PEs (7×7 kernel size) where inside each PE, we could only duplicate the kernels by 32 times, the memory efficiency is only $\sim$2.34%. However, in conventional mapping method, the kernels could be unrolled into a matrix with size 147×64, which could be implemented by 2 sub-arrays, thus there is only 1 PE used and the kernels could be duplicated by 8 times, the memory efficiency is $\sim$28.7%. It is clear to find that, to maximize the memory efficiency, when the channel depth is much smaller than the number of row in sub-array, it is better to map the kernels with conventional mapping method. This drawback will not affect deeper convolutional layers (with much larger channel depth), since they do not have the duplication limitation.

The fully-connected (FC) layers are simple 2D vector-matrix multiplication, which can also be considered as special convolutional layers, but with kernel size equals to 1×1. For example, in the ResNet, the size of weight-matrix in the FC layer equals to 4096×1000, it can be considered that, the IFM size is $1\times1\times4096$, each kernel size is $1\times1\times4096$, and the kernel depth is 1000, finally the output size is $1\times1\times1000$. Thus, the mapping of FC layer stays same for both conventional and novel mapping method, i.e. directly map the weight-matrix into a group of PEs (or sub-arrays).

It should be noted that, there are residual blocks in the ResNet models, where the OFMs from previous layer will be accumulated with the OFMs from current layer, for example, the OFMs from layer<N> will be accumulated with the OFMs from layer<N+3>. In our design, we consider it as path to bypass the OFMs from previous layers and add to the current ones, which can be processed in the L3 buffer and global accumulation units directly.

In summary, the optimal design option is to use the hybrid mapping method, where for FC layer and the convolutional layers with very small IFM channel depth (normally the first layer) use conventional mapping method to maximize memory efficiency, and the other convolutional layers use novel mapping method to maximize input data reuse and minimize data transfer cost. Table I shows the implementation of ResNet-34, where the first convolutional layer and the only FC layer use conventional mapping method, the rest of them use novel mapping method. By using the weight-matrix duplication as discussed above, the number of duplication equals to the speed-up ratio, as Table I shows, the first layer could be speeded up by 8 times, layer-2 to layer-7 could be speeded up by 32 times, layer-8 to layer-15 will be speeded up by 16 times, and so on. It should be noted that, these duplication comes for free with our novel architecture design, where the PE is assumed to be the minimum computation units.

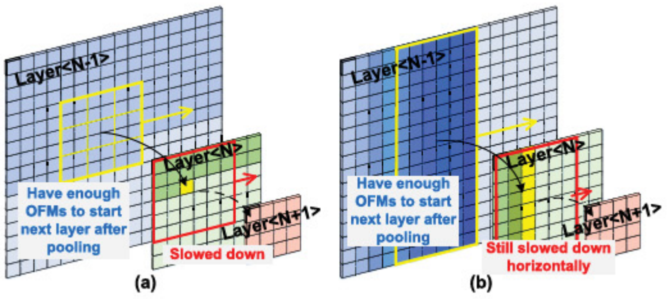
Layer<N-1>
Layer<N>
Layer<N+1>
Have enough OFMs to start next layer after pooling
Slowed down
(a)
Layer<N-1>
Layer<N>
Layer<N+1>
Have enough OFMs to start next layer after pooling
Still slowed down horizontally
(b)

Fig. 8. Example of processing convolutional layers, (a) no speed-up; (b) import a block of IFMs to speed up the convolutional layers vertically.

C. Pipeline System

In deep CNNs, the OFMs of current layer will be the IFMs of next layer. Since the OFMs tend to become deeper but smaller (i.e. for size W×W×N, the W becomes smaller, and N becomes larger) from shallow layers to deeper layers, according to the operation in section III (i.e. it needs W×W cycles to get total OFMs in a convolutional layer), the OFMs from shallow layers need much longer time to be generated than the deeper layers.

A conventional scheduling is to process the algorithm layer-by-layer (L-by-L), i.e. we do not start next layer until we get the whole OFMs of current layer. The advantage of this L-by-L operation is: the shallow layers (which need much longer time to generate OFMs for next layers) will not affect the speed of deeper layers. However, the drawback is: there will be a long period of time that most of the hardware resources are staying in idle (when some specific layer is processing), which leads to standby leakage problem.

The natural solution is pipeline system, which makes every single PE works all the time. It should be noted that, since the speed of the first couples of convolutional layers are quite slow, the pipeline could slow down the deeper layers since they need to wait for prior layers' OFMs. For example, in VGG [30], the OFMs size is 224×224 after first two convolutional layers, then passing a pooling layer, the OFMs become to 112×112, which will be used as IFMs for the third convolutional layer and generate its OFMs with the same size; it is clear to find that the time to get the IFMs of the third layer is 224×224, and the time for the third layer to generate its OFMs is 112×112. However, if the convolutional layers are pipelined, there is obviously a speed mismatch between the pooling layer and third layer, and it leads to a weak pipeline system which could cause data mismatch. Thus, it is necessary to speed up the shallow layers in the pipeline system, and build up a balanced pipeline system.

There are various methods to speed up the convolutional layers, a basic concept is to further duplicate the weight matrix (i.e. duplicate PEs in this work, since PE is the minimum computation unit for each layer), and thus process multiple OFMs at the same time. However, various method of importing IFMs to the duplicated weight matrix could cause different effects. For example, Fig. 8 (a) shows the example of how the convolutional layers operate without any weight duplication

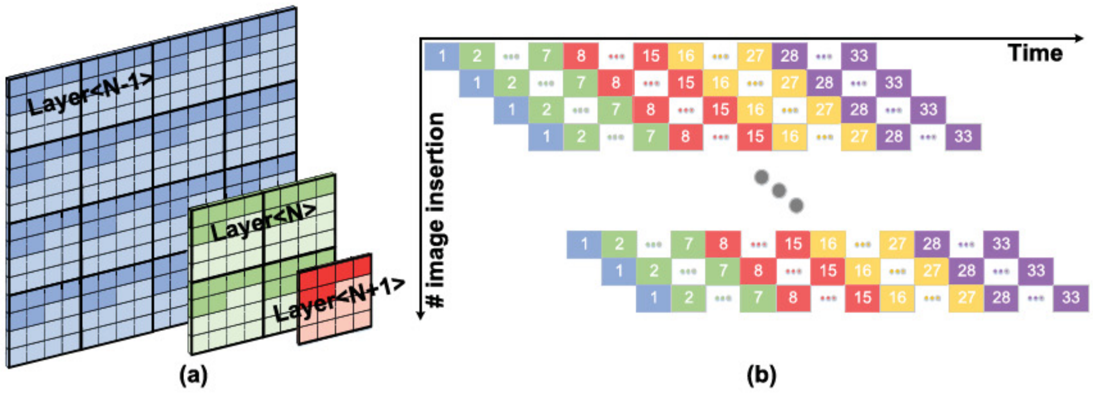

Fig. 9. Example of (a) speed up each layer by "tiling down the OFMs"; (b) pipeline system to process ResNet-34, where each layer is a stage that can process one image, thus there could be 33 images in total through the entire process.

(speed-up). Fig. 8 (b) shows the case when we import a vertical block of IFMs to speed up, it is true that the processing of OFMs has been speeded up vertically, however, the speed of OFMs' processing in horizontal will still slow down the next layer's operation. Thus, the processing of each convolutional layer should be speeded up both vertically and horizontally.

In deep CNNs, there are mainly two methods to downsize their OFMs from layer to layer: 1) use 2×2 pooling layer, which will downsize both the height and width of OFMs by 2 [30]; 2) use special convolutional layers with stride equals to 2, and automatically generate the OFMs whose height and width equal to the half of its IFMs [13]. Of course, the downsize ratio can be other value rather than 2, but we can considered that, in deep CNNs, the size of OFMs from each convolutional layer is always proportional to others' by an integer factor. Thus, it is possible to speed up the slow layers by these integer factors, and balance the pipeline system.

As mentioned in section V-B, the shallow layers have large OFMs but small weight matrix, which make it possible for weight duplication and naturally speed up the process. As a case study, in ResNet-34, the OFMs' size of layer-1 is 112×112, the OFMs' size of layer-2 to layer-7 is 56×56, the OFMs' size of layer-8 to layer-15 is 28×28, the OFMs' size of layer-16 to layer-27 is 14×14, and the OFMs' size of layer-28 to layer-33 is 7×7. An extreme design is to speed up each layer based on the fasted layers (layer-28 to layer-33), i.e. 7×7 cycles. According to the integral multiples of each layer to the fastest ones, the layer-1 needs to be speeded up by 256 times, layer-2 to layer-7 need to be speeded up by 64 times, layer-8 to layer-15 need 16 times speed-up and layer-16 to layer-27 need 4 times speed-up. According to Table I, we only need to further duplicate the weights (tiles) for layer-1 to layer-7, since most of them can naturally support weight duplication, and already satisfy the speed-up requirement.

Fig. 9 (a) shows the example of how to speed up each layer by "tiling down the OFMs" of each layer, thus to make every layer has the same processing speed. It should be noted that, this pipeline system is an extremely optimal design, where all the layers are tightly pipelined, i.e. each layer is considered as a pipeline stage, and during a certain period of time, each layer is processing different image. Thus, we need to implement extra buffers on-chip to store independent OFMs from various layers, considering the memory tiles dominate the total area, the buffer overload is negligible, while the throughput improvement is significant.

TABLE I
RESNET-34 CONFIGURATION AND HARDWARE DESIGN

Layer #	Type	Kernel Size	# PE	# Duplication
1	Conv.	(64,3,7,7)	1	8
2	Conv.	(64,64,3,3)	9	32
3	Conv.	(64,64,3,3)	9	32
4	Conv.	(64,64,3,3)	9	32
5	Conv.	(64,64,3,3)	9	32
6	Conv.	(64,64,3,3)	9	32
7	Conv.	(64,64,3,3)	9	32
8	Conv.	(128,64,3,3)	9	16
9	Conv.	(128,128,3,3)	9	16
10	Conv.	(128,128,3,3)	9	16
11	Conv.	(128,128,3,3)	9	16
12	Conv.	(128,128,3,3)	9	16
13	Conv.	(128,128,3,3)	9	16
14	Conv.	(128,128,3,3)	9	16
15	Conv.	(128,128,3,3)	9	16
16	Conv.	(256,128,3,3)	9	8
17	Conv.	(256,256,3,3)	9	4
18	Conv.	(256,256,3,3)	9	4
19	Conv.	(256,256,3,3)	9	4
20	Conv.	(256,256,3,3)	9	4
21	Conv.	(256,256,3,3)	9	4
22	Conv.	(256,256,3,3)	9	4
23	Conv.	(256,256,3,3)	9	4
24	Conv.	(256,256,3,3)	9	4
25	Conv.	(256,256,3,3)	9	4
26	Conv.	(256,256,3,3)	9	4
27	Conv.	(256,256,3,3)	9	4
28	Conv.	(512,256,3,3)	9	2
29	Conv.	(512,512,3,3)	9	1
30	Conv.	(512,512,3,3)	9	1
31	Conv.	(512,512,3,3)	9	1
32	Conv.	(512,512,3,3)	9	1
33	Conv.	(512,512,3,3)	9	1
34	FC.	(1000,4096)	16	1

VI. Benchmark and Discussion

A. Simulation Setup

To evaluate the hardware performance of the RRAM based PIM architecture, we utilized a circuit-level macro model NeuroSim [11] to estimate the area, latency and energy based on at 32nm CMOS node where RRAM integration is feasible from the industry's perspective. We use an 8-bit inference architecture, and implement the ResNet-34 on-chip, where the hardware configurations are designed for the optimal pipeline as mentioned in section V-C. We set the "naïve" weight mapping method with conventional H-tree routing as the baseline, to evaluate the latency and energy saving in interconnect and buffer in this work.

Table II shows the component parameters of the proposed architecture. The energy is given in energy per unit (or module and PE) per operation (or bit). For example, the energy of ADC is 22.45 pJ/op, it is the energy for all the ADCs in one sub-array to process one operation, here the operation means once the sub-array is activated and given a vector of input pulse, the ADC will get one group of outputs from the sub-array. It should be noted that, considering about the area overhead, we have grouped the columns and shared every 8 columns with one ADC. In order to get the outputs of all the columns in one sub-array, the ADCs need to work 8 operations. Moreover, since the input precision is set to be 8-bit, the entire sub-array need to operate 8 times, and meanwhile these outputs will be shifted and accumulated. In addition, in the PE level, the sub-array energy is 25.04 nJ/op, which tells the total energy for one single sub-array to do one 8-bit vector-matrix multiplication (8-bit inputs multiply 8-bit weights and accumulate the dot-products). This includes all the operations in sub-array level as discussed, such as sharing and switching the ADCs among columns and shift-adds for high-precision inputs.

TABLE II
Architecture Parameters

Component	Spec.		Energy	Area (mm^2)
Architecture Level				
PE	Number	1560	0.202 uJ/PE/op	198.12
L3 Buffer	Device	SRAM	0.254 pJ/bit	7.39
	Size	128KB		
	Bus Width	512-bit		
	Number	24		
Pooling unit	Precision	8-bit	0.044 pJ/unit/op	1.037
	Number	8192		
Accumulation Units	Max Bit	26-bit	9.795 pJ/unit/op	38.5
	Number	6144		
Activation Units	Precision	8-bit	0.014 pJ/unit/op	0.08
	Number	6144		
L2 Buffer	Device	SRAM	0.132 pJ/bit	3.9
	Size	16KB		
	Bus Width	256-bit		
	Number	48		
Architecture Total (ResNet-34)				249.03
PE Level (1560 PEs on chip)				
Sub-array	Number	16	25.04 nJ/op	0.101
	Device	Register		
L1 Buffer	Size	4Kb	0.064 pJ/bit	0.0073
	Bus Width	128-bit		
Adder Tree	Max Bit	16-bit	15.90 pJ/op	0.01
	Number	128		
Output Buffer	Device	Register	0.064 pJ/bit	0.0073
	Size	4Kb		
	Bus Width	128-bit		
PE Total				0.127
Sub-array Level (only shows key modules)				
RRAM array	Size	128*128	0.37 pJ/op	0.0003
	Cell Precision	2-bit		
WL Switch Matrix	Used to select WLs during read & write		0.27 pJ/op	0.0004
SL Switch Matrix	Used to select SLs during write, not used in read		/	0.0002
Mux & Mux Decoder	To select columns in read		0.24 pJ/op	0.0005
	3-bit Mux Decoder for "8 column share 1 ADC"			
ADC	Precision	5-bit	22.45 pJ/op	0.0036
	Number	16		
Shift-Add & DFF	Precision	14-bit	1.6 pJ/op	0.0014
	Number	16		
Sub-Array Total				0.0063

B. Results and Discussion

According to the novel weight mapping and data flow in this work, instead of passing the entire K×K IFMs, every time the PE groups only takes the new S×K IFMs from L2 buffer, and the used (K−S)×K IFMs will be moved forward to neighboring PEs and be reused for the computation of next OFMs (where K is the kernel size, and S is stride size). Hence, compared to the baseline (K×K×D, where D is the channel depth of IFMs), only a small amount of bits (1×K×D) will be visited in buffers, and interconnect will transfer much less data every time in this work. Here we estimate the total latency and energy for one ImageNet image inference on quantized 8-bit ResNet-34. It should be noted that, we do not consider any speed-up and pipeline in the analysis of interconnect and buffer, since the speed-up and pipeline will not change the data flow in novel architecture, as Fig. 9 (a) shows.

In baseline (without any speed-up and pipeline), due to the limitation of bus width and the number of bits that can be read out at each access for different buffer size, around ~22% of the total latency and ~18% of dynamic energy consumption are caused by the interconnect and buffers according to the simulation result, while the leakage energy due to idle PEs could occupy ~20% of total energy.

Fig. 10 shows the latency and energy consumption of interconnect (normalized to the first convolutional layer) and buffers (normalized to the last convolutional layer) along the convolutional layers in ResNet-34. For interconnect, the size of weight matrix used to map the synaptic weights (the first layer has the minimum weight matrix) determines the latency

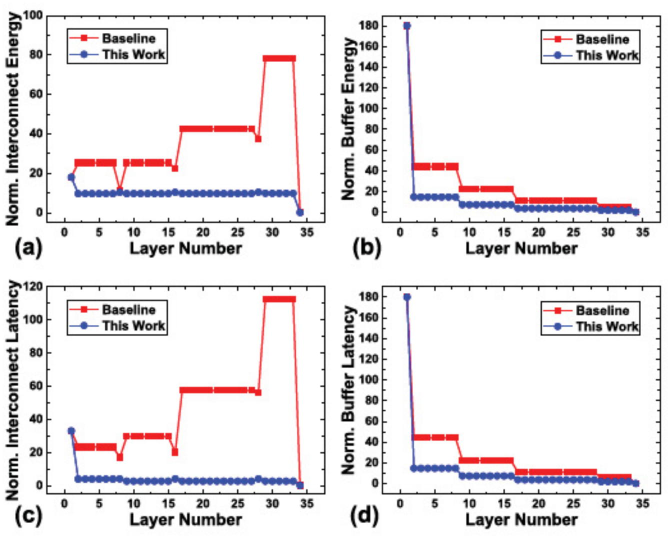

Fig. 10. (a) Normalized energy of interconnect and (b) buffer, (c) Normalized latency of interconnect and (d) buffer for both baseline and this work.

and energy consumption, while for the buffers, the input feature sizes (the last layer has smallest input size) determine those. For the baseline design, in principle, the latency and energy consumption should be higher in deeper layer, but there are some drops in layer-8, −16 and −28. The reason why latency and energy consumption drop dramatically, is because there are pooling layers before those layers, which cause 4× decrease of transferring input data, but the results increase dramatically after that, because the weight matrix are doubled and leading to longer transferring distance (e.g. weight matrix size in layer-27 is 3×3×256×256, while in layer-28 is 3×3×512×512). For the proposed design, since the input reuse is maximized, each time only 1/3 of the input as for baseline (kernel size is 3×3) will be transferred from buffer to the nearest PEs, and the input being reused will be transferred among the neighboring PEs through a very short distance simultaneously, thus the bus width will no longer limit the latency that much (as it does in baseline) and the results of interconnect do not vary too much along layers. While the results of buffer vary along layers, because of the limitation of bits that can read out from buffers (interface of buffer) at each access time. Overall, the novel mapping and data flow can help to save ∼90% of latency and ∼68% of energy consumption of interconnect and buffers.

Table III summarizes the estimation results of the 8-bit RRAM architecture for all the convolutional layers in ResNet-34 for three cases: the baseline, the novel mapping without any speed-up and weight-duplication, and the novel mapping with the optimal pipeline design as shown in Fig. 9 (b).

It should be noted that, from baseline to novel architecture, the only difference is the mapping method and dataflow of the large convolutional layers (layer-2 to layer-33, while layer-1 with small weight matrix keeps conventional mapping method to guarantee memory efficiency). Compared with baseline, in novel architecture, the latency and energy improvement of interconnect and buffer leads to ∼2.03× increase of throughput, and ∼1.4× improvement of energy efficiency,

TABLE III
RESNET-34 BENCHMARK RESULTS

Architecture		8-bit ResNet-34	
Chip Area (mm^2)	L-by-L	165	
	Pipelined	249	
Hardware Performance		**FPS**	**TOPS/W**
Baseline		145	10.22
Novel Archi.		294	14.27
Improvement I (Baseline to Novel)		**× 2.03**	**× 1.4**
Pipelined Novel Archi.		132,476	20.1
Improvement II (Baseline to Pipelined Novel)		**× 913**	**× 1.96**

which are 294 frame per second (FPS) and 14.27 TOPS/W respectively.

With the optimal pipelining as Fig. 9 (b) shows, by adding extra tiles and corresponding peripheries, each single layer could process an image, and thus there are 33 images could be processed on-chip at the same time. Since the shallow layers are speeded up, the latency for each pipeline stage is further optimized, i.e. within such small period of time (7×7 cycles as discussed in section V-C), the system will send out final outputs for one image. Finally, the throughput is increased by 913× by the speed-up and extreme pipeline system, and achieves ∼1.96× improvement of energy efficiency by minimizing the leakage energy (avoid idle tiles). This optimal pipeline design achieves 132,476 FPS and 20.1 TOPS/W in hardware performance, with ∼50% overload of chip area caused by weight duplication (compared with L-by-L case). We show two extreme cases (L-by-L and optimal pipeline), in practice, between the L-by-L and optimal pipelined designs, there are some design options by trading-off the throughput and chip area.

VII. CONCLUSION

In this paper, we propose an 8-bit RRAM based PIM architecture, which is implemented based on a novel mapping method and data flow, to maximize weight and input data reuse. To analyze the latency and energy saving of interconnect and buffers, we set a baseline which uses conventional mapping method and H-tree routing. We used NeuroSim [11] to estimate the area, latency and energy of an inference engine for ResNet-34 benchmark at 32nm, which shows at least ∼90% save of latency and 68% save of energy in interconnect and buffers. This novel mapping and data flow achieves overall ∼2.03× speed up and ∼1.4× improvement in energy efficiency. With an optimal speed-up and pipeline design based on this novel architecture, the throughput could be improved by 913×, and the energy efficiency could be increased by 1.96×, which are 132,476 FPS and 20.1 TOPS/W, respectively. The area overhead of this optimal design is ∼50%, while this overhead could be minimized by trading-off with the throughput. The proposed novel mapping method and data flow as well as optimization schemes such as weight duplication and pipeline could be applied to other PIM architectures with SRAM and other eNVM technologies.

References

[1] Y. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Jan./Feb. 2016, pp. 262–263.

[2] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "Envision: A 0.26-to-10 TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28 nm FDSOI," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2017, pp. 246–247.

[3] D. Shin, J. Lee, J. Lee, and H.-J. Yoo, "DNPU: An 8.1 TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2017, pp. 240–241.

[4] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Architecture (ISCA)*, Jun. 2017, pp. 1–12.

[5] X. Si *et al.*, "A twin-8T SRAM computation-in-memory macro for multiple-bit CNN-based machine learning," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2019, pp. 396–398.

[6] S. Yu and P. Chen, "Emerging memory technologies: Recent trends and prospects," *IEEE Solid-State Circuits Mag.*, vol. 8, no. 2, pp. 43–56, Jun. 2016.

[7] C.-C. Chou *et al.*, "An N40 256K$\times$44 embedded RRAM macro with SL-precharge SA and low-voltage current limiter to improve read and write performance," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2018, pp. 478–480.

[8] P. Jain *et al.*, "A 3.6 Mb 10.1 Mb/mm^2 embedded non-volatile ReRAM macro in 22 nm FinFET technology with adaptive forming/set/reset schemes yielding down to 0.5 V with sensing time of 5 ns at 0.7 V," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2019, pp. 212–214.

[9] S. Yu, "Neuro-inspired computing with emerging nonvolatile memorys," *Proc. IEEE*, vol. 106, no. 2, pp. 260–285, Feb. 2018.

[10] P.-Y. Chen and S. Yu, "Partition SRAM and RRAM based synaptic arrays for neuro-inspired computing," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2016, pp. 2310–2313.

[11] P.-Y. Chen, X. Peng, and S. Yu, "NeuroSim: A circuit-level macro model for benchmarking neuro-inspired architectures in online learning," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 37, no. 12, pp. 3067–3080, Dec. 2018.

[12] X. Peng, R. Liu, and S. Yu, "Optimizing weight mapping and data flow for convolutional neural networks on RRAM based processing-in-memory architecture," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2019, pp. 1–5.

[13] S. Wu *et al.*, "Training and inference with integers in deep neural networks," in *Proc. Int. Conf. Learn. Representations (ICLR)*, 2018, pp. 1–14.

[14] O. Russakovsky *et al.*, "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, Dec. 2015.

[15] A. Shafiee *et al.*, "ISAAC: A convolutional neural network accelerator with *in-situ* analog arithmetic in crossbars," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Architecture (ISCA)*, Jun. 2016, pp. 14–26.

[16] Y. Chen *et al.*, "DaDianNao: A machine-learning supercomputer," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Cambridge, U.K, Dec. 2014, pp. 609–622.

[17] P. Chi *et al.*, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *Proc. ACM/IEEE Int. Symp. Comput. Architecture (ISCA)*, Jun. 2016, pp. 27–39.

[18] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A pipelined ReRAM-based accelerator for deep learning," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 541–552.

[19] I. Hubara *et al.*, "Binarized neural networks," in *Proc. 30th Conf. Neural Inf. Process. Syst. (NIPS)*, Mar. 2016, pp. 1–13.

[20] M. Rastegari *et al.*, "XNOR-Net: ImageNet classification using binary convolutional neural networks," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, Oct. 2016, pp. 525–532.

[21] L. Ni, Z. Liu, H. Yu, and R. V. Joshi, "An energy-efficient digital ReRAM-crossbar-based CNN with bitwise parallelism," *IEEE J. Explor. Solid-State Computat. Devices Circuits*, vol. 3, no. 2, pp. 37–46, Dec. 2017.

[22] H. Huang, L. Ni, K. Wang, Y. Wang, and H. Yu, "A highly parallel and energy efficient three-dimensional multilayer CMOS-RRAM accelerator for tensorized neural network," *IEEE Trans. Nanotechnol.*, vol. 17, no. 4, pp. 645–656, Jul. 2018.

[23] Y. Yang *et al.*, "Training high-performance and large-scale deep neural networks with full 8-bit integers," 2019, *arXiv:1909.02384*. [Online]. Available: https://arxiv.org/abs/1909.02384

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[25] T. Gokmen, M. Onen, and W. Haensch, "Training deep convolutional neural networks with resistive cross-point devices," *Frontiers Neurosci.*, vol. 11, no. 10, pp. 1–13, Oct. 2017.

[26] S.-S. Sheu *et al.*, "A 4Mb embedded SLC resistive-RAM macro with 7.2 ns read-write random-access time and 160 ns MLC-access capability," in *Proc. IEEE Int. Solid-State Circuits Conf.*, Feb. 2011, pp. 200–202.

[27] P.-Y. Chen *et al.*, "Technology-design co-optimization of resistive cross-point array for accelerating learning algorithms on chip," in *Proc. Des. Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2015, pp. 854–859.

[28] X. Sun, S. Yin, X. Peng, R. Liu, J.-S. Seo, and S. Yu, "XNOR-RRAM: A scalable and parallel resistive synaptic architecture for binary neural networks," in *Proc. Des. Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 1423–1428.

[29] L. Gao, P.-Y. Chen, and S. Yu, "Programming protocol optimization for analog weight tuning in resistive memories," *IEEE Electron Device Lett.*, vol. 36, no. 11, pp. 1157–1159, 2015.

[30] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Int. Conf. Learn. Representations (ICLR)*, 2015, pp. 1–14.

Xiaochen Peng (S'17) received the B.S. degree in automatic system from the Hefei University of Technology in 2014 and the M.S. degree in electrical engineering from Arizona State University in 2016. She is currently pursuing the Ph.D. degree in electrical and computer engineering with the Georgia Institute of Technology, Atlanta, GA, USA. Her research interests include development of device-to-system benchmarking framework for machine learning accelerators and design of emerging-device-based hardware implementation for neural networks.

Rui Liu (S'16) received the B.S. degree from Xidian University, Xi'an, China, in 2011, the M.S. degree from Peking University, Beijing, China, in 2014, and the Ph.D. degree in electrical engineering from Arizona State University in 2018. Her research interests include emerging non-volatile memory device/architecture design, radiation effects in RRAM devices and array architectures, hardware design for security systems, and new computing paradigm exploration.

Shimeng Yu (M'14–SM'19) received the B.S. degree in microelectronics from Peking University, Beijing, China, in 2009, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, USA, in 2011 and 2013, respectively. He is currently an Associate Professor of electrical and computer engineering with the Georgia Institute of Technology, Atlanta, GA, USA. He has published more than 90 journal articles and more than 140 conference articles with citations more than 9000 and H-index 47. His research interests are emerging nano-devices and circuits with a focus on the resistive memories for different applications including machine/deep learning, neuromorphic computing, and hardware security. He was a recipient of the NSF Faculty Early CAREER Award in 2016, the IEEE Electron Devices Society (EDS) Early Career Award in 2017, the ACM Special Interests Group on Design Automation (SIGDA) Outstanding New Faculty Award in 2018, and the Semiconductor Research Corporation (SRC) Young Faculty Award in 2019.