PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning

Linghao Song*, Xuehai Qian†, Hai Li*, and Yiran Chen*

*University of Pittsburgh, †University of Southern California

*{linghao.song, hal66, yiran.chen}@pitt.edu, †xuehai.qian@usc.edu

ABSTRACT

Convolutional neural networks (CNNs) are the heart of deep learning applications. Recent works PRIME [1] and ISAAC [2] demonstrated the promise of using resistive random access memory (ReRAM) to perform neural computations in memory. We found that training cannot be efficiently supported with the current schemes. First, they do not consider weight update and complex data dependency in training procedure. Second, ISAAC attempts to increase system throughput with a very deep pipeline. It is only beneficial when a large number of consecutive images can be fed into the architecture. In training, the notion of batch (e.g. 64) limits the number of images can be processed consecutively, because the images in the next batch need to be processed based on the updated weights. Third, the deep pipeline in ISAAC is vulnerable to pipeline bubbles and execution stall.

In this paper, we present *PipeLayer*, a ReRAM-based PIM accelerator for CNNs that support both training and testing. We analyze data dependency and weight update in training algorithms and propose efficient pipeline to exploit inter-layer parallelism. To exploit intra-layer parallelism, we propose highly parallel design based on the notion of parallelism granularity and weight replication. With these design choices, PipeLayer enables the highly pipelined execution of both training and testing, without introducing the potential stalls in previous work. The experiment results show that, PipeLayer achieves the speedup of 42.45x compared with GPU platform on average. The average energy saving of PipeLayer compared with GPU implementation is 7.17x.

1. INTRODUCTION

It has been widely accepted that Moore's Law is ending soon [3], which means that processor performance will no longer increase at the same pace as in recent decades. On the other side, the volume of data that computer systems process has skyrocketed over the last decade. In particular, Artificial Neural Networks techniques as exemplified by deep learning [4, 5, 6] have become the state-of-the-art across a broad range of applications, including object detection [7], scene parsing [8] and visual question answering [9]). Some have even achieved human-level performance on specific tasks such as ImageNet recognition [10], Atari 2600 video games [11] and AlphaGo[12].

Convolutional neural networks (CNNs) are the heart of deep learning applications and are both compute and memory intensive. For example, AlexNet [13] performs 10^9 operations in processing just one image data. In conventional Von Neumann architecture where computation and data storage are separated, a large amount of data movements are also incurred due to the large number of layers and millions of weights. Such data movements quickly become a performance bottleneck due to limited memory bandwidth and more importantly, an energy bottleneck. A recent study [14] showed that data movements between CPUs and off-chip memory consumes two orders of magnitude more energy than a floating point operations. In an era of soon-ending Moore's law and the increasing demand of deep learning, it is urgent to reduce the data movement and computing cost.

Processing-in-memory (PIM) is an efficient technique to reduce data movements in memory hierarchy. On the other hand, the emerging non-volatile memory, metal-oxide resistive random access memory (ReRAM) [15] has been considered as a promising candidate for future memory architecture due to its high density, fast read access and low leakage power. ReRAM-based PIM is a particularly appealing option because ReRAM provides both computation and storage capability. Such design incurs no increase in cost-per-bit implied by compute logic in traditional PIM based on DRAM. Recent works [1] demonstrated that ReRAM-based PIM offer great acceleration of CNNs with low energy cost.

Despite the recent progresses [1, 2, 16], the current schemes based on ReRAM lack important features to efficiently execute complete deep learning applications. First, they only focus on testing (inference) phase of CNN but do not support the more sophisticated and intensive training (learning) phase. Second, ISAAC [2] uses a very deep pipeline to improve system throughput. However, it is only beneficial when a large number of consecutive images can be fed into the architecture. This is not true in training phase as only a limited number of consecutive image could be processed before weigh updates. Third, the deep pipeline in ISAAC also introduces pipeline bubbles. Moreover, data organization and kernel mapping are not clearly addressed in PRIME [1].

To close the gaps of ReRAM-based acceleration for complete deep learning, this paper proposes *PipeLayer*, a ReRAM-based PIM accelerator that supports complete deep learning applications. Compared to previous work, we make the following contributions.

Accelerating both training and testing. Supporting training phase is more sophisticated and challenging because it involves weight updates and complex data dependencies. The previous schemes assume that the weight values are only written to ReRAM once at the start and never change.

The simple intra- and inter-layer pipeline designed for training. To ensure high throughput for CNNs of many layers with weight updates, PipeLayer

adopts a new pipelined architecture different from [2] so that data could continuously flow into the accelerator in consecutive cycles. It is essential to support pipelined training phase. Various data input and kernel mapping schemes could be exploited using our design to balance data processing parallelism and hardware cost (i.e. the number of replicated ReRAM arrays).

Spike-based data input and output. To eliminate the overhead of DACs and ADCs, PipeLayer uses a spike-based scheme, instead of voltage-level based scheme for data input. Such design requires more cycles to inject data, however, the drawback is offset by the pipelined architecture for multiple layers. The data input scheme used in [2] is similar as our spike-based input, both eliminating DACs, but our Integration and Fire component eliminates ADCs while [2] keeps.

In the evaluation, we use ten networks. Six are popular large scale networks, AlexNet [13], and VGG-A, VGG-B, VGG-C, VGG-D, VGG-E [10], which are based on ImageNet [17]. Four networks based on MNIST [18] are built by ourselves. To evaluate PipeLayer, we build a simulator based on NVSim [19]. And the baseline is a platform with the newest released GPU, GTX 1080. The experiment results show that, PipeLayer achieves the speedup of 42.45x compared with GPU platform on average. The average energy saving of PipeLayer compared with GPU implementation is 7.17x.

The paper is organized as follows. Section 2 introduces the background of deep neural network, ReRAM and its application in neural computations. Section 3 presents the PipeLayer architecture. Section 4 presents the implementation of PipeLayer . Section 5 discusses application related issues and compares PipeLayer with ISAAC. Section 6 explains the evaluation methodology and presents experiment results. Section 7 discusses other related work. Section 8 concludes the paper.

2. BACKGROUND

2.1 Basics of Deep Neural Network

The core component of many deep learning applications (on computer vision [20, 21, 22, 23, 24], data mining [25, 26, 27, 28], language processing [29, 30, 31, 32, 33] and etc.) is convolutional neural network (CNN).

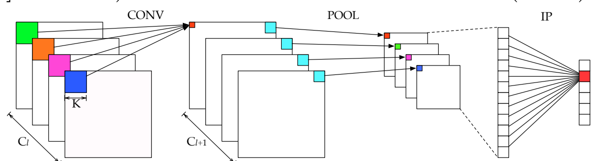


Figure 1: Convolutional Neural Network (CNN)

Figure 1 illustrates the organization of an example CNN, which has three types of layers: convolution layer, pooling layer and inner product layer. In a convolution layer, a set of kernels are convoluted with data of channels from the previous layer (layer l) to generate data for channels of next layer (layer $l+1$). d_l is a cube of data in a layer. $d_l[x, y, c]$ is the value at a point in the three dimensional data cube. We also denote the size of d in layer l as $(X_l \times Y_l \times C_l)$, so $0 \le x \le X_d - 1, 0 \le y \le Y_d - 1, 0 \le c \le C_d - 1$ and C_d is the number of channels. (x_l, y_l, c_l) indicates a point in layer l's data cube. K is the kernel composed of a set of weights. K_l is the kernel used in the computation to generate data in layer l. A kernel represents four dimensional data: the size of each dimension is K_x, K_y, C_l and C_{l+1}, where K_x and K_y are determined by algorithm (e.g. in LeNet [34], K_x and K_y are both 5).

d_{l+1} is computed as:

$$d_{l+1}[x, y, c] = \sum_{c_l=0}^{C_l-1} \sum_{k_x=0}^{K_x-1} \sum_{k_y=0}^{K_y-1} K_l[k_x, k_y, c_l, c] \times d_l[x + k_x, y + k_y, c_l] \quad (1)$$

To perform Equation (1), in total $(X_{l+1} \times Y_{l+1} \times C_{l+1} \times C_l \times K_x \times K_y)$ multiplications and $(X_{l+1} \times Y_{l+1} \times C_{l+1} \times (C_l \times K_x \times K_y - 1)$ additions are performed.

A pooling layer performs the subsampling. Taking average pooling as an example, a window of data in l is averaged to get one data point in $l+1$ as follows:

$$d_{l+1}[x, y, c] = \frac{1}{K_x K_y} \sum_{k_x=0}^{K_x-1} \sum_{k_y=0}^{K_y-1} d_l[K_x x + k_x, K_y y + k_y, c] \quad (2)$$

This average pooling operation performs $(X_{l+1} \times Y_{l+1} \times C_{l+1} \times (K_x \times K_y - 1))$ additions and $(X_{l+1} \times Y_{l+1} \times C_{l+1})$ multiplications. The multiplication could be implemented as shift operation if $(K_x \times K_y)$ is the power of 2. Max pooling is another variance, where the maximum value among values in a window in l is selected for $l+1$.

In the inner product layer, the values in data tube of l and $l+1$ are considered as a vector (denoted as $\vec{d_l}$ and $\vec{d}_{l+1}$). If the previous layer is convolution or pooling, the size of $\vec{d_l}$ is $X_l \times Y_l \times C_l$. If the previous layer is also inner product, then the size of $\vec{d_l}$ is the size of the output vector from l. $\vec{d}_{l+1}$ is a $n \times 1$ vector, n is determined by the algorithm. W_{l+1-l} is a weight matrix of size $(n \times m)$, m is the size of $\vec{d_l}$. $\vec{b}$ is a vector of bias.

The vector of $l+1$ is computed as:

$$\vec{d}_{l+1} = W_{l+1-l}\vec{d_l} + \vec{b} \quad (3)$$

This inner product operation performs $(n \times (m-1))$ additions and $(n \times m)$ multiplications.

Activation function is another important component. It is an element-wise operation and usually a nonlinear function, such as sigmoid $\frac{1}{1+e^{-x}}$ or rectified linear unit (ReLU) $max(0, x)$.

2.2 Data Forward and Backward in a Neural Network

Neural network has two phases: training (learning) phase and testing (inference) phase. In testing phase, the weights of a neural network have been determined and the task is to use the network on input samples, e.g. to recognize who is the person in an image. In testing phase, input data flow through the layers consecutively in forward direction. It is shown in Figure 2, d_{l-1} is the output of layer $l-1$ and the input of layer l. The forward process can be represented as the two equations indicated above. The computation is the same as what we discussed in Section 2.1.

Before a neural network for a specific application can be deployed, it needs to be trained and generate the

weights. This is done in training phase, where data not only move forward but also backward, — to update weights based on errors. In training phase, a cost function is defined to quantitatively evaluate how well the outputs of a neural network compare to the standard labels. We use y and t to represent the output of a neural network and the standard label respectively. An L^2 norm loss function is defined as $J(W,b) = \frac{1}{2}||y-t||_2^2$ and $J(W,b) = -\sum_{i,j} 1(y^i = t^j)\log p(y^i = t^j)$ is the softmax loss function.

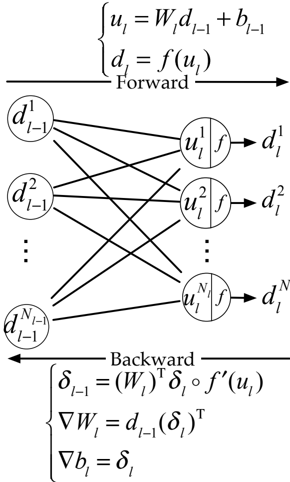


Figure 2: Two Adjacent Layers

The error δ for each layer is defined as: $\delta_l \triangleq \frac{\partial J}{\partial b_l}$. If we use an L^2 norm loss function, for the last(output) layer L, the error is $\delta_L = f'(u_L) \circ (y-t)$ where $\circ$ represents a Hadamard product, i.e. element-wise multiplications. For other layers excluding the output layer, the error is $\delta_l = (W_{l+1})^{\top}\delta_{l+1} \circ f'(u_l)$.

And with a ReLU activation function, the error can be rewritten as $\delta_l = (W_{l+1})^{\top}\delta_{l+1} \circ f'(d_l)$. So that the backward partial derivatives to W^l is $\frac{\partial J}{\partial W_l} = d_{l-1}(\delta_l)^{\top}$. And the backward partial derivatives to b_l is $\frac{\partial J}{\partial b_l} = \delta_l$. Now we can use the gradient descent method to update the weights of neural network.

We see that the training phase is more complex than testing phase due to weight updates. It also introduces more data dependencies (e.g. weight update to a layer depend on the previous layer's error and the data of the earlier forward computation). Therefore, training phase is time consuming and could take **14 to 21 days** [10].

2.3 Neural Computation using ReRAM

Recent works [1, 2, 16] demonstrated that ReRAM-based architecture could accelerate matrix-vector multiplication in neural computation. However, the current works lack important features to efficiently execute complete deep learning applications. First, they do not support training that involves weight update and complex data dependencies. Second, the deep pipeline in ISAAC [2] is not beneficial to training phase where the length of consecutive images that could enter the pipeline is limited by batch size. Also, it is vulnerable to pipeline bubble.

3. PIPELAYER ARCHIETECTURE

In this section, we first analyze the general training procedure and outline the required mechanisms to support it. Then we present the ideas of intra- and inter-layer pipeline that explore the parallelism and support training naturally.

PipeLayer architecture directly leverages ReRAM cells to perform computation without the need for extra processing units. Resistive random access memory (ReRAM or RRAM) is a type of non-volatile memory that stores information by changing the cell resistances. This paper focuses on a subset of resistive memories, — metal-oxide ReRAM, which uses metal oxide layers as switching materials. More details on the technology could be found in [1, 35, 36, 37, 38, 39, 40, 41, 42].

Our design partition the ReRAM-based main memory into two regions: morphable subarrays (Morp) and memory subarrays (Mem). The morphable subarrays can perform both computation and data (weights) storage, the two modes can be configured. In computation mode, the morphable subarrays perform matrix-vector multiplications. The memory subarrays are similar to conventional memory and have data storage capability. Due to space limit, readers could refer to [1] for details on the ReRAM basics.

3.1 Training Support

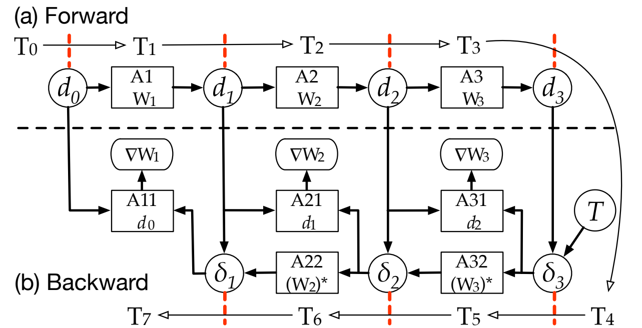


Figure 3: PipeLayer Configured for Training

For simplicity, Figure 3 shows the configuration of PipeLayer to process the training of a 3-layer CNN. The rectangles are one layer of morphable subarrays. The circles are memory subarrays to store intermediate results transferred between morphable subarrays for different layers.

Data dependencies exist in forward and backward computations. The computation timing and dependencies are shown in Figure 3. Starting from forward computation, the initial cycle is T_0, in cycle T_1, the input (d_0) enters $A1$ (morphable subarrays), which perform the matrix-vector multiplication. At the end of T_1, the results are written to a memory subarray, d_1. For clarity, we use red dashed lines to show the system state between two consecutive cycles. Note that the cycles in Figure 3 (e.g. T_0, T_1, ...) are *logical* cycles, depending on implementation, each cycle could take several physical clock cycles. We will show in Section 3.2 that the number of physical cycles is determined by parallelism and hardware resource. The term "cycle" from this point all refer to the logical cycle. The solid lines between rectangles (morphable subarrays) and circles (memory subarrays) indicate the data dependencies. The following forward layers are performed similarly.

The results of forward computation are eventually stored in d_3 before backward computation starts at T_4. The backward computations generate errors (δ_l) (l is the layer)

and partial derivatives to b and W (∇b_l and ∇W_l). As the first step, the error for the last (third) layer (δ_3) is computed in T_4. It is stored in memory subarrays served as the input for computations in the next cycle.

In T_5, two computations happen in parallel: (1) partial derivatives (∇W_3) is computed by previous results in d_2 and δ_3; (2) errors (δ_2) of the second layer is computed from δ_3. Both of the computations depend on δ_3, which is computed in T_4. ∇W_3 is stored in memory subarrays, which will be used to update weights in $A3$ and $A32$ later. Note that $(W_3)^*$ is a reordered form of W_3 and will be discussed in Section 4.3. Finally, in T_7, the partial derivatives for the first layer (∇W_1) is computed and stored in memory subarrays.

In training, batch size (denoted as B) is used to specify the number of images that can be processed before a weight update. If $B = 1$, ∇W_1, ∇W_2 and ∇W_3 are used to update the weights in $A1, A2, A3$ and $A22, A32$. If $B > 1$, ∇W_1, ∇W_2 and ∇W_3 are stored in the buffers in a special way so that the average of all partial derivatives can be computed automatically.

Training phase is more complicated than testing phase because it involves data dependencies between outputs from previous layers and the compute of partial derivatives and errors for weight update. Memory subarrays are used as buffers to store intermediate results, this greatly reduce data movements and energy consumption by avoiding data being transferred across memory hierarchy. When not necessary (e.g. with only testing), some of them can be converted to morphable subarrays.

Within a cycle, different operations could be performed depending on the phase of computation. Table 1 shows the operations in four possible cases. In an implementation, the cycle time has to allow the longest sequence of operations to fit. Note that the cycle time is not a bottleneck, because memory is much slower than processors.

3.2 Intra Layer Parallelism

3.2.1 *Naive Data Input and Kernel Mapping*

Figure 4 illustrates a naive scheme of data input and the mapping of kernels to a ReRAM array. In this example, data of layer l, kernels and data of layer $l+1$ have a size of $114 \times 114 \times 128$, $3 \times 3 \times 128 \times 256$ and $112 \times 112 \times 256$, respectively.

For each channel in layer $l+1$, the corresponding kernel has a size of $3 \times 3 \times 128$. That means the size of an input vector (yellow bar) to the ReRAM array at one cycle is 1152×1 (actually it should be 1153×1, we neglect the bias for express clarity). And in the next cycle, the kernel window shifts right or down and we have the another 1152×1 yellow bar from input. Consequently, input data enter ReRAM array in a sequential mode. It takes 12544 cycles to get all outputs of layer $l+1$.

Mapping all kernels to the same ReRAM array is not realistic. Weights of one kernel is mapped to cells of one bit line, for example, the blue cuboid is mapped to the blue bar in the array and it is the same case for the red, green and the rest cuboids, resulting in 256 bit lines and

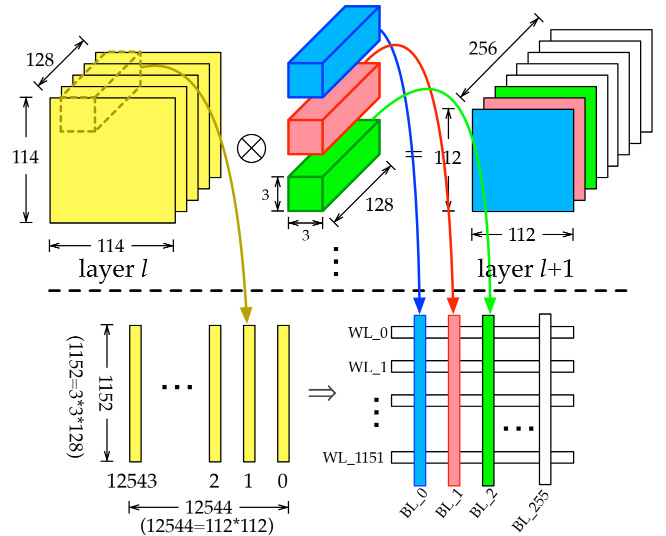


Figure 4: An Naive Scheme for Data Input and Kernel Mapping

1152 world lines for the ReRAM array. Therefore, the ReRAM array needs to have a size of 1152×256. In Section 3.2.3, we will present a more realistic design, but before that, let us explore the approach of ISAAC [2] to improve the naive design.

3.2.2 *Pipeline in ISAAC [2]*

The pipeline used in ISAAC [2] attempts to improve the throughput of the architecture by computing small tiles of a layer and then let the next layer use the partial output as input in the next cycle. It could notably improve through put by the deep pipeline. The advantage is that if a large number of input data could be fed into the pipeline continuously, after a (large) number of initial cycles to fill the pipeline, results could be generated each cycle. Unfortunately, this assumption is untrue for neural network training phase, where weights are updated at the end of a batch. The new inputs in the next batch need to be processed based on the updated weights and we cannot have a large number of consecutive input data for the pipeline. Therefore, the pipeline design in ISAAC does not fit well for training phase.

Another issue is the vulnerability to pipeline bubble, leading to execution stall. Consider a network where all kernels are of size $2 \times 2 \times 1$. One point $p50$ in layer l_5 depends on 4, 16, 64, 256 points in layer l_4, l_3, l_2 and l_1, which means the computation of $p50$ is stalled when any of the 340 ($= 4 + 16 + 64 + 256$) points is delayed, which could further stall computations depending on $p50$. Moreover, data dependencies are complex in training phase, layer l may not only depend on layer $l-1$, but also from earlier layers (e.g. δ_1 in 3 depends on δ_2 and early data d_1). It further increase the chance of stalls. [2] mentioned pipeline imbalance issue but similar issues could easily occur with more complex training phase and it is extremely hard to predict and avoid all of them.

3.2.3 *Parallelism Granularity*

The ReRAM array size could be made feasible by par-

L: Number of Layers; l: the Current Layer						
	Memory Read	Spike Driver	Morphable Subarray	Integration & Fire	Activation	Memory Subarray
Forward	✓	✓	✓	✓	✓	✓
Backward 1	d_L				$*$	∇b_L
Backward 2	∇b_l	✓	$A_{L1}(d_{l-1})$; $A_{L2}((W_{l-1})^*)$	✓	✓	∇W_l
Update	∇W_l	✓				W_l

Table 1: Break of Operations in a Cycle.

tition. We can decompose the 1152×256 matrix to a group of 18 (=9×2) matrices and map each of them to a 128×128 ReRAM array (shown in the right part of Figure 5). We can get the right results by collecting array outputs horizontally and summing them vertically.

To improve performance, we define a metric called *parallelism granularity*, denoted as G, indicating the number of duplicated copies of ReRAM arrays that store the same weights. If $G = 1$, the design is equivalent to the naive scheme. If $G = 12544$, the results of a layer could be generated in just one cycle but the hardware cost is prohibitive.

Essentially, parallelism granularity allows explore the trade-off between hardware resource of ReRAM array and performance. A good trade-off requires a carefully chosen G. Figure 5 shows an example with $G = 256$.

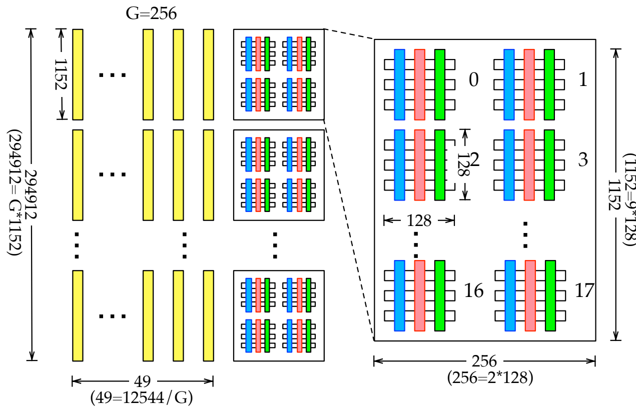


Figure 5: A Balanced Scheme for Data Input and Kernel Mapping

3.3 Inter Layer Parallelism

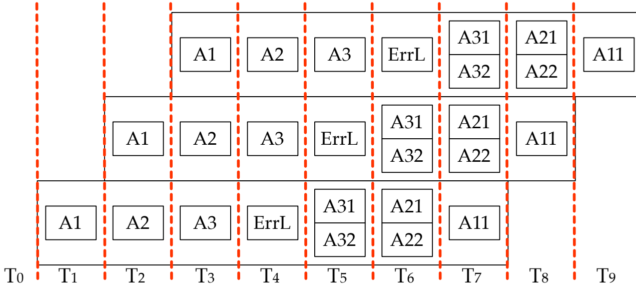
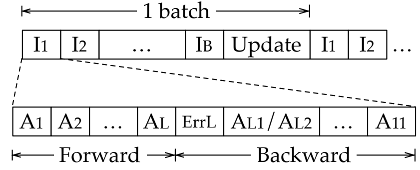


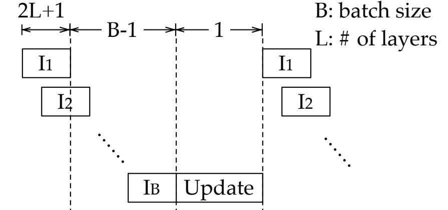
Figure 6: Training Pipeline in PipeLayer

In training, the input data are normally processed in batch. The inputs in the same batch are all processed based on the same weights at the start of the batch. The weight updates due to each input are stored and only applied to at the end of a batch. Therefore, no dependency exists among data inputs inside a batch. We propose an architecture to support pipelined training. The performance gain is due to the fact that B is normally much larger than 1 (e.g. 64), otherwise, each input needs to be processed sequentially.

Based on Figure 3, the pipelined execution is shown in Figure 6. The execution with intra- and inter-layer pipeline could achieve high throughput without the drawbacks of previous work. In the execution, a new input can enter the pipeline every cycle within a batch. At the end of batch, a new input belonging to the next batch cannot enter the pipeline until all inputs in previous batch are processed and weights are updated. In another word, a new batch has to wait the "tail" of previous batch to drain from the pipeline.



(a) Latency of PipeLayer without pipeline



(b) Latency of PipeLayer with pipeline

Figure 7: Latency of PipeLayer Architecture

Figure 7 (a) shows the latency of the baseline architecture without pipeline. Assuming the neural network has L layers, we can see that the forward computation of one input takes L cycles, the backward computation of one input takes $(L + 1)$ cycles, at the end of a batch, one cycle is dedicated to the weight update. Therefore, a batch takes $(2L + 1)B + 1$ cycles. For a total number of N inputs, the total number of training cycles is $(2L + 1)N + N/B$.

Now let us consider the pipelined execution, the latency is illustrated in Figure 7 (b). Within a batch, a new input could enter every cycle. Similar to the baseline case, the first weight update is generated after $(2L + 1)$ cycles. Then there will be $(B - 1)$ cycles until the end of batch. Finally, there is one cycle to apply all weight updates within the batch. Since the forward and backward computation are intertwined with each other, we do not distinguish the forward and backward cycles. From the above analysis, the total number of cycles to process N inputs with L layers is $(N/B)(2L + B + 1)$.

The major implication of pipelining is the increased requirement of memory subarrays. The problem is illustrated in Figure 8. Consider the same example in Figure 3, since now $A1$ will produce output every cycle, we need to have more buffers so that the data which will be used later are not over written. Specifically, referring

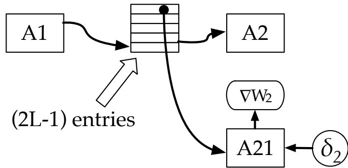


Figure 8: Memory Subarrays in PipeLayer

to Figure 3, d_1 computed in T_1 will be used after 5 cycles, in T_6. Therefore, between $A1$ and $A2$, 5 buffers are needed. $A1$ should logically keep a pointer pointing to the current buffer that the output should be written to. After an output, the pointer is increased by one. Logically we implement circular buffers. When it reaches 5, it will wrap around and return to the first entry. When the output is written again to the first entry, the useful data (d_1) used to compute ∇W_2 can be safely overwritten because ∇W_2 has been computed. According to this, in Figure 3, $A1$ will write 1st-, 2nd-, 3rd-, 4th- and 5th-entry in buffer between $A1$ and $A2$ in T_1, T_2, T_3, T_4, T_5, consecutively. Since ∇W_2 is computed from the data in the 1st-entry in the buffer in T_5, in the next cycle, $A1$ could overwrite the 1st-entry of the buffer. In general, the buffer requirement at l-th layer (assuming the total number of layers is L) is $2(L-l)+1$.

Another implication of pipelining is that at the same cycle, there will be a read and a write on the same buffer. To support this, we need to duplicate the single buffer. This happens for the buffer at d_3, δ_1, δ_2, δ_3. Table 2 compares the number of cycles and the cost of (groups of) arrays of nonpipelined and pipelined architectures.

G: Parallelism Granularity; L: Number of Layers; B: Batch Size; N: Total Number of Input Images.		
	Non-pipelined	Pipelined
Forward Cycles	LN	(N/B)(2L+B+1)
Backward Cycles	$(L+1)N+N/B$	
Morp Subarrays	$GL+G(2L-1)$	$GL+G(L-1)+BL$
Mem Subarrays	$2L$	

Table 2: Cycle and Cost of PipeLayer Architecture.

4. IMPLEMENTAION OF PIPELAYER

4.1 Overall Architecture

Figure 9 shows the architecture of PipeLayer. Our design leverages ReRAM cells to perform computation without the need for extra processing units. The design partitions the ReRAM-based main memory into two regions: morphable subarrays and memory subarrays. The morphable subarrays can perform both computation and data (weights) storage, the two modes can be configured. In computation mode, the morphable subarrays could perform matrix-vector multiplications. The memory subarrays are the same as conventional memory and have data storage capability. They are used to store intermediate results between layers and keep data that will be used in Figure 9 only shows the morphable subarrays and memory subarrays.

a. **Spike Driver**. To reduce the area and energy overhead of voltage-level based input scheme in [1], we use a weighted spike coding scheme similar to ISAAC [2]. The spike driver converts the input to a sequence of weighted spikes. In weight update, spike driver serves as write driver to tune weights stored in the ReRAM array.

b. **Integration and Fire**. It is a required component of a spike-based scheme. It integrates input current and generates output spikes. The output spikes are connected to a counter, so essentially the analog currents are converted into digital values. The data input scheme used in [2] is similar as our spike input, both eliminating DACs, but our integration and fire components eliminates ADCs while [2] keeps.

c. **Activation**. It implements the activation function defined in a CNN algorithm. When morphable subarrays are configured as memory, it is bypassed.

d. **Connection**. This component connects morphable and memory subarrays. The outputs from morphable subarrays (when they are in computation mode) need to be written to memory subarrays so that they are used as input for the morphable subarrays for the next layer in next cycle.

e. **Control**. It offloads the computation from the host CPU and orchestrates the data transfers between memory subarrays and morphable subarrays in training and testing with based on the algorithm configurations (e.g. batch size).

4.2 Component Designs

4.2.1 Spike Driver

We use a weighted spike coding scheme. For a N-bit input value, we need N time slots for potential spikes. The spike driver design is shown in Figure 9 (a). Inside the driver, N levels of reference voltage, ranging from $V_0/2^{N-1}$ to $V_0/2$ are generated. A timing control component shifts key K1 to connect one of the reference voltages at one time slot. Note that the shifting is non-decreasing: the voltage of output spike increases as time slot progresses, thus the data input is of a Least Significant Bit First (LSBF) mode. K_2 is controlled by the input data to decide whether the spike output is generated or not.

Another function of spike driver is to write or program data to a subarray, where K_1 is connected to a programing voltage. K_2 tunes the resistance of a ReRAM cell controlled by input data which serve as programing control sequences. Spike divers are reused between adjacent subarrays.

4.2.2 Integration and Fire

Figure 9 (b) shows the design of Integration and Fire circuit. A controlled current source serves as a current follower to inject an equivalent strength current as that from a bit line to a capacitor. As a result, the voltage on this capacitor is accumulated.

When the voltage on the capacitor increases to the threshold (V_{th}) of the comparator, an output spike is generated and counted by a counter. Note that the current on the bit line is the accumulation of products of spikes (input data) and ReRAM cell conductances (corresponding weights). For a fixed threshold V_{th}, a K times stronger current will makes the comparator to generate

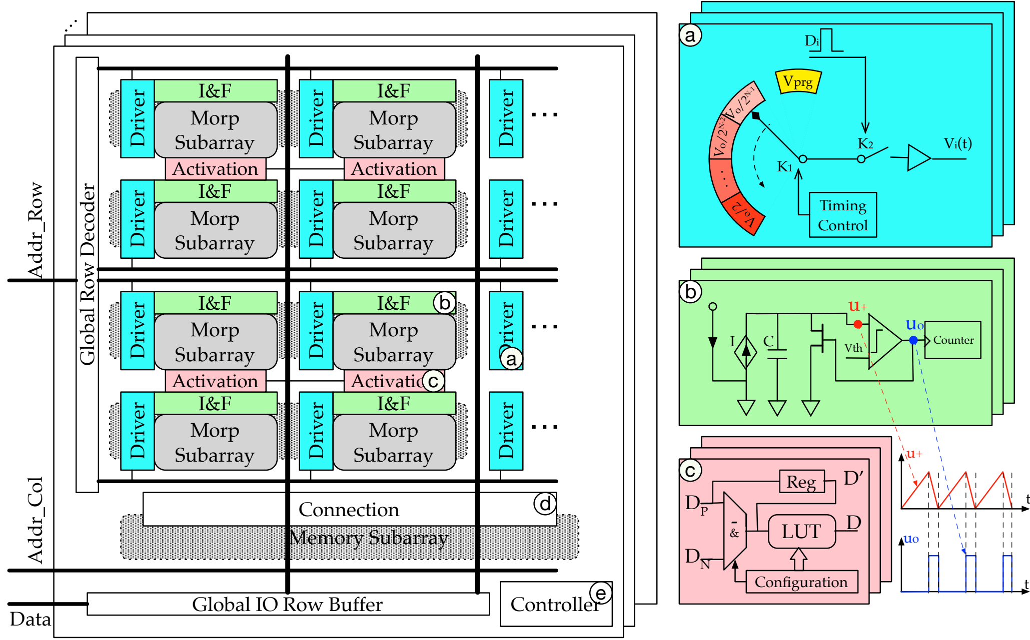


Figure 9: Architecture of PipeLayer

K times of output spikes. As a result, the number of spikes generated by the comparator is actually the accumulation of products of input data and corresponding weights. A group of integration and fire components are connected to bit lines of a subarray and they can generate an vector in parallel.

4.2.3 *Activation Function*

Figure 9 (c) shows the design of activation component. The activation component consists of a subtractor and a look up table (LUT). Since positive weights and negative weights are mapped to two subarrays, we need to collect the results D_P from a positive subarray and D_N from a negative subarray. Our activation component is configurable by different LUTs to realize the activation function defined in specific algorithm. In this work, we mainly focus on rectified linear unit (ReLU), which is widely used in deep neural networks. A register is used to keep the max value of a sequence, which realizes max pooling.

4.3 Error Backward

Before generating weight updates by partial derivatives, the error for each layer should be computed. In forward progress, *data* propagate from layer l to layer $l+1$, while in backward progress, *errors* propagate from layer l *inversely* to layer $l-1$. Figure 10 shows three kinds of error backward in convolutional neural networks.

Error backward for activation function is computed as $\delta_{l-1} = f'(u_l) \circ \delta_l$, which is an element-wise operation. The yellow squares in Figure 10 (a) represent an element of error δ_l in layer l and error δ_{l-1} in layer $l-1$. Note the activation function used is ReLU, which means $f'(\cdot)$ is either 0 or 1. The backward is to AND (&) 0 or 1 with δ_{l-1}, which is done by activation components (component c in Figure 9). Thanks to ReLU, we have $f'(u_l) = f'(d_l)$, there is no need to store intermediate data u_l in forward progress.

Figure 10 (b) shows the error backward for a pooling layer. Error elements in layer $l-1$ are copied to the positions which is the max values in windows of data in layer d_l. For example, the pink square, 3, in layer $l-1$, is copied to the lower right of the pink window in layer l, while the rest three squares in the window get 0. The four squares in layer l are propagated to the upper left, lower left, lower right and upper right of four windows in layer $l-1$. Mathematically, max pooling can be viewed as one kind of activation function. The error backward for a pooling layer is also performed in the activation component (component c in Figure 9). With d_l already stored in memory subarrays, the index for the max element in a window can be found.

For error backward, we can reuse the logic in a convolution layer. In Figure 10 (c), a 3×3 kernel is used in forward progress. In the forward progress, to generate the nine elements in the red window in layer l, all of the (light, medium and dark) blue squares in layer $l-1$ are convoluted with the 3×3 kernel and the dark blue square (with red edges) are exactly multiplied with each elements in the kernel. In the backward, to get the error for the dark blue square, we can just convolute the nine elements in the red window in layer l with the kernel.

Mathematically, error backward for convolution layer is computed as $\delta_{l-1} = \text{conv2}(\delta_l, \text{rot180}(K), \text{'full'})$. For elements at edges, zero paddings are added. Figure 11 shows the error backward for the convolution layer in Figure 4, where error in layer $l-1$ are zero-padded by 2 at each edge, so the length and width increase to 116. Also, the kernels are reordered. In this way, we can perform convolutions by using the data input and kernel mapping scheme discussed in Section 3.2.3.

4.4 Weight Update

4.4.1 *Computing Partial Derivatives*

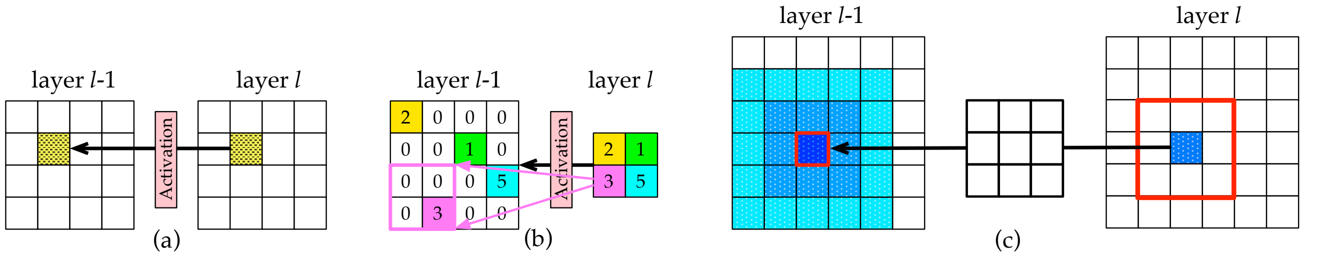


Figure 10: Error Backward for (a) Activation Function, (b) Max Pooling Layer and (c) Convolution Layer

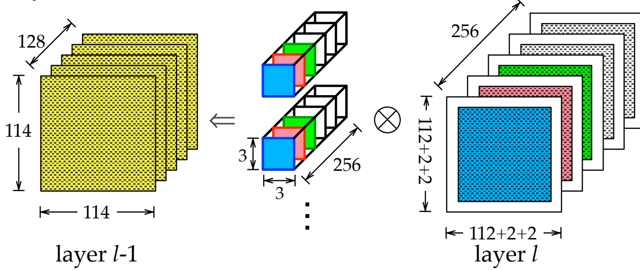


Figure 11: Error Backward for Convolution Layer by Convolution of Error of Layer l and Reordered Kernels

The partial derivatives to bias is the sum of all related error: $\nabla b = \sum_{u,v}(\delta_l)_{u,v}$. It is done by reading bitline with the input spikes representing 1. The partial derivative to a weight is the sum of element-wise products of error and patches/windows of data: $\nabla W = \sum_{u,v}(\delta_l)_{u,v} \cdot (d_{l-1})_{u,v}$. It can be converted to convolution.

In Figure 12, the convolution of one blue channel of error in layer $l-1$ and the total 128 channels of data in layer l is the partial derivatives to weights of the blue kernel (size of $3\times3\times128$). All the kernels (red, green and others) are similar. In the convolution, data in layer l (yellow slices) can be viewed as convolution kernels while error in layer $l-1$ (doted slices) can be viewed as convolution data. Therefore, the convolution is done by mapping the yellow slices to ReRAM arrays and sending the backward error to the arrays.

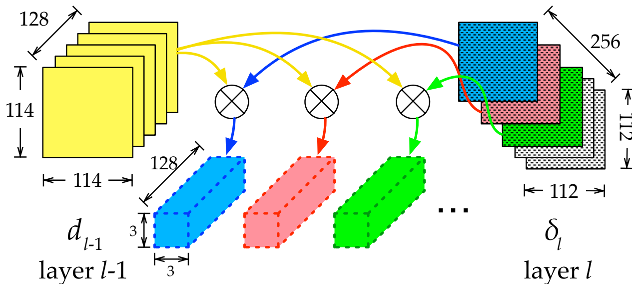


Figure 12: Computing Partial Derivatives for kernels

4.4.2 *Writing New Weights to Morphable Subarrays*

With the partial derivatives for weights ∇W and bias ∇b, old weights and bias that read from morphable subarrays. The new (updated) weights and bias can be computed and written to morphable subarrays.

In weight updating, we need to compute the average of the B (B is the batch size) partial derivatives, and this can be done by the input spikes representing $1/B$. With current accumulation property of bitline, the read out data is actually the averaged partial derivatives. At the same time, old weights are read out and new weights are computed. The LUT in activation components are bypassed when reading. Different from default reading, D_P of the subtractors are connected to old weights and D_N connected to the averaged partial derivatives. As a result, the output data are the new weights/bias, which is (old weights/bias minus averaged partial derivatives). And finally, the new weights/bias to the corresponding morphable subarrays.

5. DISCUSSION

5.1 Resolution and Accuracy

In practice, the ReRAM cell can only support limited precisions. For example, [1] used 6-bit ReRAM cells and [43] used 5 bits. Such limitation does not affect some neural networks due to their inherent error tolerance.

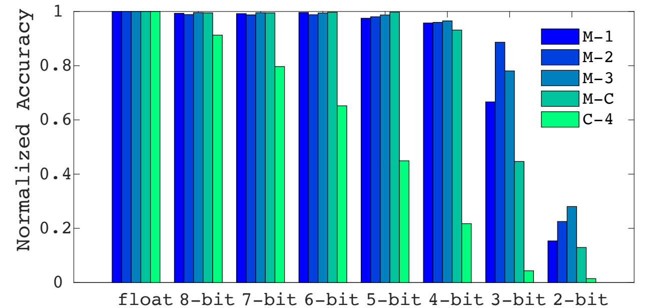


Figure 13: Tradeoff Between Resolution and Accuracy

We conducted a set of experiments to understand the trade-off between resolution (of ReRAM cells) and accuracy (of applications). We use five applications: M-1, M-2 and M-3 are multilayer perceptron neural networks while M-C and C-4 are convolutional neural networks. Figure 13 shows the normalized accuracy (to float resolution in original implementation) of using different number of bits. We see that for M-1, M-2, M-3, using 4-bit just slightly decreases the accuracy. However, the accuracy is more sensitive to resolution for convolutional neural networks, as accuracies of M-C and C-4 drop sharply from 3-bit. Especially for C-4, even at 4-bit resolution, the normalized accuracy is only around 0.2.

When higher resolution is needed, we can use multiple arrays with lower resolution, similar to [2]. The default resolution of PipeLayer is 16-bit, the same as [2, 44], and the resolution of ReRAM cells used in PipeLayer is 4-bit.

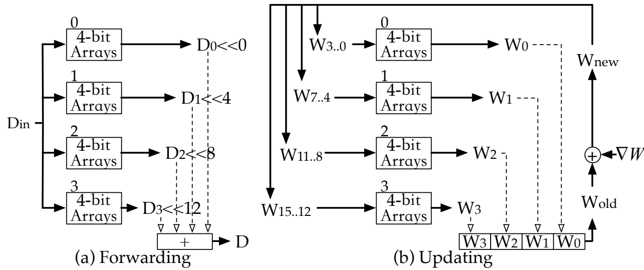


Figure 14: Resolution Compensation

In testing phase, we can easily get 16-bit results by adding four shifted 4-bit results, as shown in Figure 14 (a). Here, the same data input four groups of 4-bit arrays. The four groups store 4-bit weights for the the 15..12, 11..8, 7..4 and 3..0 segment respectively. We can shift and add the results from the four groups and get the results for 16-bit resolution. In training phase, we need to first read the old four segments of 4-bit weights then shift them to get the old weights. With the partial derivatives, we get the new weights and write back to the four groups to finish the update. It is shown in figure 14 (b).

5.2 Application Programming Interface

Thanks to the layer wise inter layer pipeline of PipeLayer , the system can be configured by layer interactively. Before discussing layer wise function invocation, we provide two functions, `Copy_to_PL` and `Copy_to_CPU` to transfer data from CPU main memory to PipeLayer or inversely. For one layer configuration, a topology set function `Topology_set` is invoked. This function configures the connections and datapath of G groups of arrays as shown in Figure 5, while G can be set by programmer or automatically optimized by compiler. Next, the weight load function `Weight_load` is invoked, which load pretrianed weights to the arrays in testing phase, or initial weights in training phase. After all layers are configured, running with pipeline can be started by function `Pipeline_Set`. Finally, one of the running mode functions `Train` or `Test` starts the execution.

6. EVALUATION

6.1 Benchmarks

The benchmark networks used are based on databases MNIST [18] and ImageNet [17]. MNIST is a classical database of handwritten digits for pattern recognition. It consists of a training set of $60K$ images and a testing set of $10K$ images. And the labels for the images range from 0 to 9. Images of MNIST are 28-by-28 gray images. ImageNet provides $14.2M$ images and $21.8K$ indexed synsets. Most of the largest and state-of-the-art performance neural networks are based on ImageNet.

For ImageNet, we select six popular large scale networks, i.e. AlexNet [13], and VGG-A, VGG-B, VGG-C, VGG-D, VGG-E [10]. The topologies and hyperparameters of these networks can be found from the references. For MNIST, we build four neural networks as our benchmarks, the hyper parameters are shown in Table 3.

In Table 3, N_1-N_2 represents an inner product layer where there are N_1 perceptrons in layer l and N_2 neurons in layer $l+1$ (e.g. $784-100$ represents an inner product layer where there are 784 neurons in the first layer and 100 neurons in the second layer). Conv$K$xC represents a convolution layer where the kernel size is K by K and the number of channels in the next layer is C.

Network	Hyper Parameters
Mnist-A	784-100-10
Mnist-B	784-500-250-10
Mnist-C	784-1500-1000-500-10
Mnist-0	conv5x10-360-10

Table 3: Hyper Parameters of Networks on MNIST

6.2 Experiment Setup

In our experiments, we compare PipeLayer with a GPU-based platform. Benchmarks running on the platform are based on the widely used framework Caffe[45].

To run the GPU platform, we set the running mode in the Caffe prototxt to GPU. The GPU used is the newest released GTX 1080. Table 4 shows the parameters of our GPU platform. The run times for GPU platform are measured by caffe and the energy costs are measured by the tool `nvidia-smi` provided by NVIDIA CUDA.

CPU: Intel Xeon E5-2630 V3, 8 cores, 2.40 GHz, $8 \times (32+32)$KB L1 Cache, 8×256KB L2 Cache, 20 MB L3 Cache.	
Memory	128 GB
Storage	1 TB
Graphic Card	NVIDIA Geforce GTX 1080
Architecture	Pascal
CUDA Cores	2560
Base Clock	1607 MHz
Compute Capability	6.1
Graphic Memory	8 GB GDDR5X
Memory Bandwidth	320 GB/s
CUDA Version	8

Table 4: Configurations of the GPU Platform

To evaluate PipeLayer, we build a simulator based on NVSim [19]. The read/write latency, read/write energy cost used in the simulator are 29.31 ns/50.88 ns per spike, 1.08 pJ/3.91 nJ per spike, which are reported in [46]. And the area model is based on data reported in [47].

6.3 Performance Results

Figure 15 shows the performance comparisons of training and testing phase. For each application, we report the execution time comparison of GPU, non-pipelined and pipelined PipeLayer (*PipeLayer*). The GPU implementation is used as the baseline and running times of benchmarks on different settings are normalized to it.

Compared to GPU platform, the geometric mean of overall (training + testing), training and testing speedup achieved by pipelined PipeLayer architecture are 33.85 x, 53.22x and 42.45x, respectively. Among all applications in both training and testing, the highest speedup achieved by non-pipelined PipeLayer is 20.81x, while for pipelined PipeLayer , the highest speedup is 146.58x.

PipeLayer architecture archives lower speedups in training phase than in testing phase. It is because training

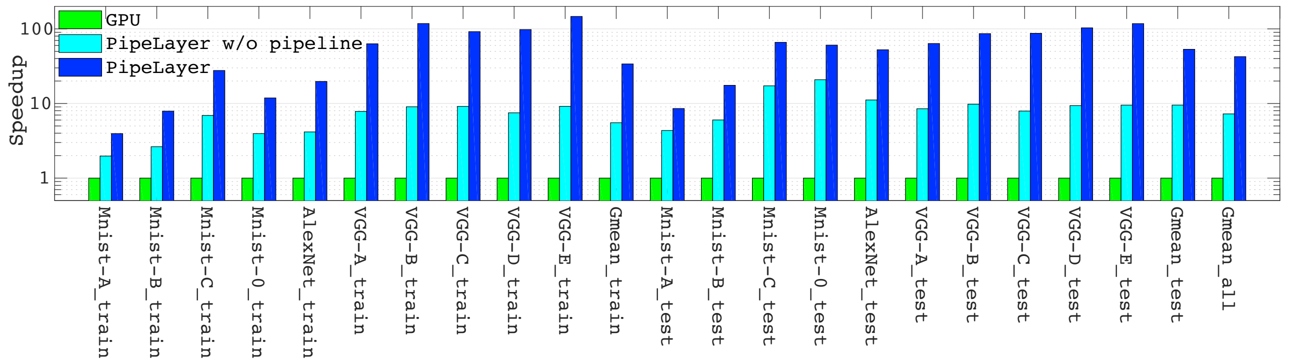


Figure 15: Speedups of Networks in Both Training and Testing

requires more intermediate data processing and weight updates.

The pipelined PipeLayer achieves a geometric mean speedup of 42.45x, significantly better than non-pipelined variants because the computations and data movements are performed in a highly parallel manner.

Intuitively, as the depth and layers of the neural networks increase, PipeLayer architecture will achieve higher speedups compared to conventional architecture. However, we found that it is not always the case. For example, the speedup of Mnist-C is larger than AlexNet in training. That is because Mnist-C is a mutilayer perceptron network, whose weights are all matrixs and can be directly mapped to ReRAM arrays.

6.4 Energy Results

Figure 16 shows *energy savings* in GPU platform and PipeLayer (pipelined). The higher the bars are, the more energy efficient the corresponding architectures are. We show results for both training and testing.

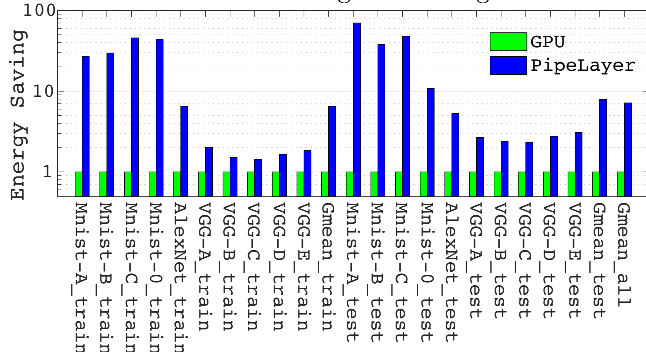


Figure 16: Energy Savings for PipeLayer

The geometric mean of energy saving compared to GPU in training and testing are 6.52x and 7.88x, respectively and the overall geometric mean of energy saving is 7.17x. The highest energy saving for training and testing are 27.03x (Mnist-C) and 70.03x (Mnist-A), respectively. We see that energy saving of PipeLayer in training is slightly less than that in testing. This is due to the extra morphable and memory subarrays needed.

In summary, we see from the results that PipeLayer architecture provides significant performance improvement with drastic energy reduction. It comes from two sources. First, the computation cost is reduced, the matrix-vector multiplications commonly used in CNNs are performed by analog circuits, which is much more energy efficient than conventional GPU implementation. Second, the data movements in memory hierarchy are greatly reduced. In PipeLayer, the data movements between different levels of memory hierarchies in conventional architectures are replaced by data movements in memory itself through layers.

6.5 Sensitivity of Parallelism Granularity

This section analyzes the effect of parallelism granularity on PipeLayer. Table 5 shows the default parallelism granularity for each convolution layer of 5 VGG networks. We use a scalar λ to enlarge or reduce the parallelism granularity of a network for λ times.

Layer	VGG-A	VGG-B	VGG-C	VGG-D	VGG-E
conv11	1024	1024	1024	1024	1024
conv12	-	1024	1024	1024	1024
conv21	256	256	256	256	256
conv22	-	256	256	256	256
conv31	64	64	64	64	64
conv32	64	64	64	64	64
conv33	-	-	64	64	64
conv34	-	-	-	-	64
conv41	16	16	16	16	16
conv42	16	16	16	16	16
conv43	-	-	16	16	16
conv44	-	-	-	-	16
conv51	4	4	4	4	4
conv52	4	4	4	4	4
conv53	-	-	4	4	4
conv54	-	-	-	-	4

Table 5: Default Parallelism Granularity G Configurations of Each Convolution Layer in VGGs

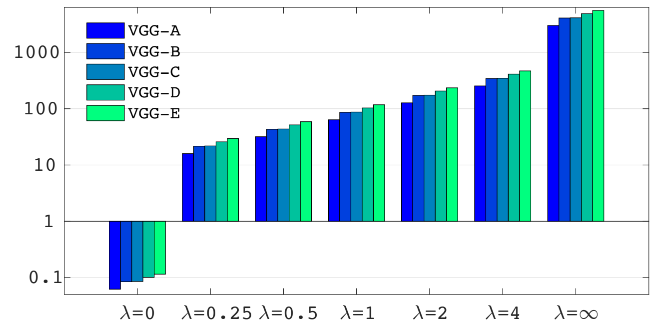


Figure 17: Speedups vs. Parallelism Granularity

In Figure 17, $\lambda = 0$ means the parallelism granularity of all layers are $G = 1$ and $\lambda = \infty$ means G is set to the maximum value for each layer. Figure 17 shows that the speedup (compared with GPU) increase monotonically

with λ. However, the increase of parallelism granularity will also cause the increase in area, as shown in Figure 18. Therefore, choosing the suitable parallelism granularity to explore the balance between speedup and area is critical. We choose the balanced parallelism granularity for the networks as the default values shown in Table 5.

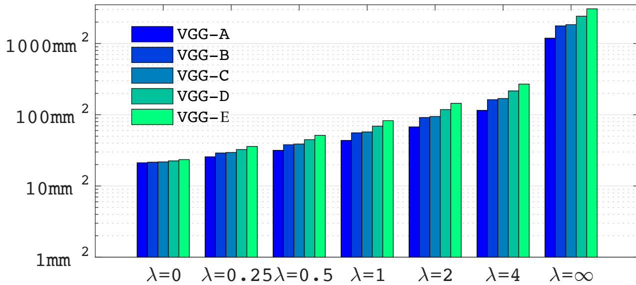


Figure 18: Area vs. Parallelism Granularity

6.6 Computation Efficiency Results

The area of PipeLayer is $82.63mm^2$. The computational efficiency of PipeLayer is $1485GOPS/s/mm^2$, higher than both DaDianNao [44] ($63.46GOPS/s/mm^2$) and ISAAC [2] ($479.0GOPS/s/mm^2$). On the other side, the power efficiency of PipeLayer is $142.9GPOS/s/W$, lower than DaDianNao ($286.4GPOS/s/W$) and ISAAC ($380.7GPOS/s/W$). In training phase, intermediate data d are used as the kernels to compute partial derivatives. As discussed in section 4.4.1, d is written to morphable subarrays for the convenience of partial derivatives computing, which means the groups of "storing" arrays are converted to "computing" arrays. For this reason, PipeLayer gains a higher computational efficiency. However, the lower power efficiency is because we write all of data to ReRAM arrays, while DaDianNao and ISAAC write to eDRAMs.

7. OTHER RELATED WORKS

PIM was a hot research topic since 1990s (e.g., IRAM [48] and DIVA [49]). Early efforts explored the integration of simple ALU [50], vectorization, SIMD [51], general processor [52], and FPGA [53] with DRAM. Unfortunately, integrating memory and computing logic into the same chip was difficult at that time [54]. Driven by the data intensive applications and the 3D-stacking technology, PIM or near data computing is resurgent, with lots of industry effort [55, 56]. Recent efforts [57, 58] decouple logic and memory designs in different dies, adopting 3D stacked memories with a logic layer that encapsulates processing units to perform computation.

Recent work also uses nonvolatile memory technologies to build ternary content addressable memories (TCAMs) [59, 60], which exploits memory cells to perform associative search operations. However, to support such search operations, these studies require redesign of nonvolatile memory cell structures to enable much larger cell sizes, which inevitably increases the cost of memory.

Many previous studies based on ReRAM for neural network acceleration, such as stand-alone accelerator [43, 61], co-processor [62] and many-core or NoC [16] architecture. Neural network acceleration are studied on other platforms, GPU [63], FPGA [64]. Significant ASIC works are [44, 65, 66, 67, 68]. The efforts of prior work focus on the co-processor architecture that accesses data from main memory in a conventional way. However, some neural network applications require a high memory bandwidth to fetch large-size input data or synaptic weights, and the data movement from memory to processors is energy-consuming (i.e. DRAM accesses consume 95% of the total energy in DianNao design [65]).

8. CONCLUSION

Convolutional neural networks (CNNs) are the key to deep learning applications. Recent works PRIME [1] and ISAAC [2] demonstrated the promise of using resistive random access memory (ReRAM) to perform neural computations in memory. However, they cannot support training efficiently due to the pipeline organizations. This paper proposes *PipeLayer*, a ReRAM-based PIM accelerator for CNNs that for the first time support both training and testing. We propose efficient pipeline to exploit intra- and inter-layer parallelism. PipeLayer enables the highly pipelined execution of both training and testing, without introducing the potential stalls in previous work. The experiment results show that, PipeLayer achieves the speedups of 42.45x compared with GPU platform on average. The average energy saving compared with GPU implementation is 7.17x.

9. ACKNOWLEDGEMENT

This work is supported in part by NSF CNS-1253424 and XPS-1337198. This work is also supported by Natural Science Foundation of China (61133004, 61502019) and Spanish Gov. European ERDF under TIN2010-21291-C02-01 and Consolider CSD2007-00050.

10. REFERENCES

[1] P. Chi *et al.*, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proc. ISCA*, 2016.

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

[3] M. M. Waldrop, "The chips are down for moore's law," *Nature News*, vol. 530, no. 7589, p. 144, 2016.

[4] K. Jarrett *et al.*, "What is the best multi-stage architecture for object recognition?," in *Proc. ICCV*, 2009.

[5] H. Larochelle *et al.*, "An empirical evaluation of deep architectures on problems with many factors of variation," in *Proc. ICML*, 2007.

[6] Q. V. Le, "Building high-level features using large scale unsupervised learning," in *Proc. ICASSP*, 2013.

[7] P. Sermanet *et al.*, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *ArXiv Preprint arXiv:1312.6229*, 2013.

[8] C. Farabet *et al.*, "Learning hierarchical features for scene labeling," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 35, no. 8, pp. 1915–1929, 2013.

[9] S. Antol *et al.*, "Vqa: Visual question answering," in *Proc. ICCV*, 2015.

[10] K. Simonyan *et al.*, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[11] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[12] D. Silver *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[13] A. Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," in *Advances in NIPS*, 2012.

[14] A. Farmahini-Farahani *et al.*, "Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules," in *Proc. HPCA*, 2015.

[15] H.-S. P. Wong *et al.*, "Metal–oxide rram," *Proceedings of the IEEE*, vol. 100, no. 6, pp. 1951–1970, 2012.

[16] X. Liu *et al.*, "Reno: a high-efficient reconfigurable neuromorphic computing accelerator design," in *Proc. DAC*, 2015.

[17] O. Russakovsky *et al.*, "ImageNet Large Scale Visual Recognition Challenge," *IJCV*, vol. 115, no. 3, pp. 211–252, 2015.

[18] Y. LeCun *et al.*, "The mnist database of handwritten digits," 1998.

[19] X. Dong *et al.*, "Nvsim: A circuit-level performance, energy, and area model for emerging non-volatile memory," in *Emerging Memory Technologies*, pp. 15–50, Springer, 2014.

[20] H. Lee *et al.*, "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations," in *Proc. ICML*, 2009.

[21] D. Ciresan *et al.*, "Multi-column deep neural networks for image classification," in *Proc. CVPR*, 2012.

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

[23] P. Sermanet *et al.*, "Convolutional neural networks applied to house numbers digit classification," in *Proc. ICPR*, 2012.

[24] M. Oquab *et al.*, "Learning and transferring mid-level image representations using convolutional neural networks," in *Proc. CVPR*, 2014.

[25] Y. LeCun *et al.*, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[26] Y. Kim, "Convolutional neural networks for sentence classification," *arXiv preprint arXiv:1408.5882*, 2014.

[27] A. G. Howard, "Some improvements on deep convolutional neural network based image classification," *arXiv preprint arXiv:1312.5402*, 2013.

[28] Y. Gong *et al.*, "Deep convolutional ranking for multilabel image annotation," *arXiv preprint arXiv:1312.4894*, 2013.

[29] R. Collobert *et al.*, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proc. ICML*, 2008.

[30] O. Abdel-Hamid *et al.*, "Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition," in *Proc. ICASSP*, 2012.

[31] N. Kalchbrenner *et al.*, "A convolutional neural network for modelling sentences," *arXiv preprint arXiv:1404.2188*, 2014.

[32] L. Deng *et al.*, "New types of deep neural network learning for speech recognition and related applications: An overview," in *Proc. ICASSP*, 2013.

[33] A. Graves *et al.*, "Speech recognition with deep recurrent neural networks," in *Proc. ICASSP*, 2013.

[34] Y. LeCun *et al.*, "Gradient-based learning applied to document recognition," *Proc. IEEE*, 1998.

[35] 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.

[36] C. Hsu *et al.*, "Self-rectifying bipolar taox/tio2 rram with superior endurance over 1012 cycles for 3d high-density storage-class memory vlsi tech," in *Proc. VLSIT*, 2013.

[37] M. K. Qureshi *et al.*, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *Proc. MICRO*, 2009.

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

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

[40] S. Yu *et al.*, "Investigating the switching dynamics and multilevel capability of bipolar metal oxide resistive switching memory," *Applied Physics Letters*, vol. 98, no. 10, p. 103514, 2011.

[41] 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, no. 6, p. 065010, 2012.

[42] 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.

[43] M. Hu *et al.*, "Dot-product engine for neuromorphic computing: programming 1t1m crossbar to accelerate matrix-vector multiplication," in *Proc. DAC*, 2016.

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

[45] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. MM*, 2014.

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

[47] R. Fackenthal *et al.*, "19.7 a 16gb reram with 200mb/s write and 1gb/s read in 27nm technology," in *Proc. ISSCC*, 2014.

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

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

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

[51] D. Elliott *et al.*, "Computational ram: The case for simd computing in memory," in *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember at ISCA*, 1997.

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

[53] M. Oskin *et al.*, *Active pages: A computation model for intelligent memory*, vol. 26. IEEE Computer Society, 1998.

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

[55] D. Zhang *et al.*, "Top-pim: throughput-oriented programmable processing in memory," in *Proc. HPDC*, 2014.

[56] 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, 2015.

[57] B. Akin *et al.*, "Data reorganization in memory using 3d-stacked dram," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 131–143, ACM, 2015.

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

[59] Q. Guo *et al.*, "A resistive tcam accelerator for data-intensive computing," in *Proc. ISCA*, 2011.

[60] Q. Guo *et al.*, "Ac-dimm: associative computing with stt-mram," in *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 189–200, ACM, 2013.

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

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

[63] S. Chetlur *et al.*, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[64] D. Mahajan *et al.*, "Tabla: A unified template-based framework for accelerating statistical machine learning," in *Proc. HPCA*, 2016.

[65] T. Chen *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, vol. 49, pp. 269–284, ACM, 2014.

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

[67] Z. Du *et al.*, "Shidiannao: shifting vision processing closer to the sensor," in *Proc. ISCA*, 2015.

[68] D. Kim *et al.*, "Neurocube: A programmable digital

neuromorphic architecture with high-density 3d memory," in *Proc. ISCA*, 2016.