2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED) | 978-1-6654-3922-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISLPED52811.2021.9502474

FPRA: A Fine-grained Parallel RRAM Architecture

Xiao Liu[1] Minxuan Zhou[1] Rachata Ausavarungnirun[2] Sean Eilert[3]
Ameen Akel[3] Tajana Rosing[1] Vijaykrishnan Narayanan[4] Jishen Zhao[1]

[1]*University of California, San Diego* [2]*King Mongkut's University of Technology North Bangkok*
[3]*Micron Technology, Inc.* [4]*Pennsylvania State University*

***Abstract*—Emerging resistive memory (RRAM) based crossbar array is a promising technology to accelerate neural network applications. RRAM-based CNN accelerators support a high-degree of intra-layer and inter-layer parallelism. The intra-layer parallelism duplicates kernels for each network layer while the inter-layer parallelism allows execution of each layer when a portion of input data is available. However, previously proposed RRAM-based accelerators do not leverage data sharing between duplicate kernels leading to significant idleness of crossbar arrays during inference. This shared data creates data dependencies that stall the processing of the next layer in the pipeline.**

To address these issues, we propose Fine-grained Parallel RRAM Architecture (FPRA), a novel architectural design, to improve parallelism for pipeline-enabled RRAM-based accelerators. FPRA addresses the data sharing issue with kernel batching and data sharing aware memory. Kernel batching rearranges the layout of the kernels and minimizes the data dependencies created by the input shared data. The data sharing aware memory uniformly buffers the input and output data for each layer, efficiently dispatching data to duplicate kernels while reducing the amount of data transferred between layers. We evaluate FPRA on eight popular image recognition CNN models with various configurations in a cycle-accurate simulator. We find that FPRA manages to achieve 2.0× average latency speedup, and 2.1× average throughput increase, as compared to the state-of-the-art RRAM-based accelerators.

I. Introduction

The emergence of non-volatile memory (NVRAM) provides an alternative for memory devices. One representative NVRAM – Resistive RAM (RRAM), or memristor, enables not only data storage ability, but also computation ability [31]. This duality blurs the boundary between memory and computation and is also referred to as processing-in-memory (PIM). PIM architecture may achieve higher performance and better power efficiency, compared to the conventional storage-only memory architecture [3].

The RRAM-based PIM architecture has been intensively explored to accelerate deep neural network (DNN) applications in recent years [5], [23], [25]. Multiple RRAM cells form an array structure called "crossbar", which can conduct a multiply-accumulate operation in a single cycle. This Single Instruction, Multiple Data (SIMD) design allows parallel matrix multiplications and convolutions, which are the fundamental operations in various DNNs. For instance, calculations in fully connected layers and convolutional layers in convolutional neural networks (CNN) are essentially matrix multiplications and convolutions. Prior works demonstrate SIMD architecture can achieve better performance compared to CMOS-based application-specific integrated circuits (ASIC) accelerators [5], [23], [25], and higher energy efficiency compared to GPU [2], [25].

However, existing RRAM-based DNN accelerators compromise performance due to the data dependency. The data dependency resulting from shared data in the existing pipeline design degrades the performance of the accelerator. Based on our analysis of the pipeline that consists of both inter-layer and intra-layer parallelisms [23], [25], we discover that the existing designs spend up to 50% of inference time on idle cycles because of the shared data induced dependency. As we demonstrate in Section II, data sharing between duplicate kernels intensifies the data dependencies between different layers, which leads to significant pipeline stalls during the execution. Our experimental results show that idle cycles caused by shared data can take up to 50% of the single inference latency for certain network models.

Some existing RRAM-based accelerators attempt to provide flexibility and reconfigurability [11], [12] while ignoring the shared data and the pipeline architecture. Other existing accelerators attempt to address the issue by abandoning the pipeline design [6], [19]. However, these solutions underutilize the on-chip area of RRAM, which leads to lower throughput. To obtain performance benefits of the pipeline and address data sharing issue, we introduce Finer-grained Parallel RRAM Architecture (FPRA), a RRAM-based accelerator design that provides high parallelism by resolving the data dependency due to shared data in duplicate kernels. FPRA is based on the insight that the proper arrangement of the duplicate kernels and unified data buffering of input and output data can resolve the shared data induced dependency. FPRA introduces *kernel-batching* to group the computation of the duplicate kernels of the same layer, and *data sharing aware memory* to uniformly manage the input and output data of duplicate kernels.

This paper makes the following contributions:

- We discover that shared data induces data dependency, and evaluate the corresponding inefficiency in the pipeline design of the RRAM-based NN accelerators.
- We propose FPRA, a highly parallel RRAM-based NN accelerator. FPRA consists of two main mechanisms: kernel batching, and data sharing aware memory.
- We evaluate FPRA on eight image recognition DNN models. Our results show 2.0× speedup[1] and 2.1× throughput increase against the state-of-art baseline.

II. Background & Motivation

A. RRAM-based NN Accelerators

Accelerator architecture. As Figure 1 (a) shows, the memristors are placed in a grid such that they are horizontally connected by the *wordlines* and vertically connected by the *bitlines*. For each memristor, the output current on the wordline is the product of memristor conductance and input voltage on the bitline (Kirchoff's Law). The currents from all the memristors connected with the same wordline are summed up. Hence, the

[1]Speedup represents the speed up of a single inference latency.

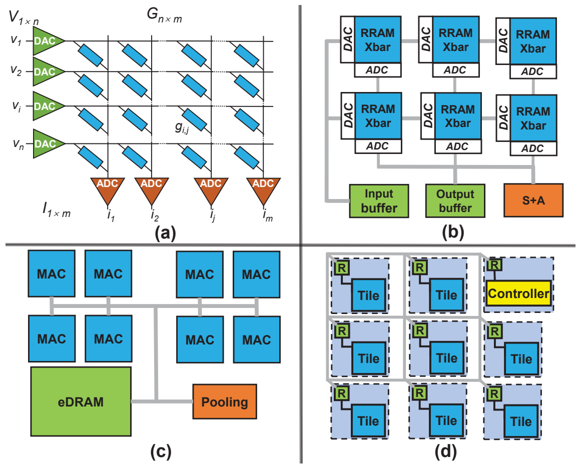

Fig. 1. Overview of RRAM-based DNN accelerator.

input voltages vector $V_{1\times n}$ is able to conduct matrix multiplication with the memristors conductance matrix $G_{n\times m}$ to get the output currents vector $I_{1\times m}$, i.e., $I_{1\times m} = V_{1\times n} \times G_{n\times m}$. Such crossbar arrays, along with analog converters (DACs), analog to digital converters (ADCs), and the shift-and-add (S+A) unit form a *multiply accumulator* (MAC).

To process a neural network, MACs have pooling unit and embedded DRAM (eDRAM) to process non-matrix-multiplication operations in the network (Figure 1 (c)). Certain number of MACs, along with shared pooling unit and eDRAM, form a *tile*. As shown in Figure 1 (d), a tile is the highest-level computational component. Tiles and chip controller are interconnected with routers to form a 2D mesh-connected network-on-chip (NoC). Tiles use routers to forward input and output data to each other. The chip controller is in charge of pre-processing and I/O operations.

Inference using RRAM-based accelerators. To process an DNN inference, the crossbar arrays process the convolutional and fully connected layers and the pooling units process the pooling layers [5], [23]. The accelerator processes the convolutional layer in the *weight stationary* form [27] by assigning weights as the conductance of the crossbar array (i.e., $G_{n\times m}$ on Figure 1 (a)). The three-dimensional kernel flattens to a single dimension which fits into a column of the crossbar array. The output on the bitline directly corresponds to output of the convolution. Weights of a kernel can be stored over the different columns of a crossbar array. Considering the limitations on the size of the crossbar array and the bitwidth of RRAM cells, all the weights of the same kernel are usually spread across multiple crossbar arrays (i.e., a column of crossbar arrays shown in Figure 1 (b)).

Existing parallelisms in RRAM-based accelerator. Similar to the CMOS-based NN accelerators [4], [13], RRAM-based accelerators possess *intra-layer* parallelism. In the intra-layer parallelism, the computations of all the parameters of a kernel are concurrent. RRAM-based accelerators obtain intra-layer parallelism through mapping the kernel parameters to the crossbar arrays. To further improve parallelism, RRAM-based accelerators can also duplicate kernel parameters to different groups of crossbar arrays, such that different parts of the input feature map are processed concurrently.

The pipeline design in the RRAM-based accelerators introduces *inter-layer parallelism*. RRAM-based accelerators can deploy more computational units within the same chip area because of the size advantage of the memristor. Existing RRAM-based accelerators [23], [25] deploy all the network layers on an RRAM-based accelerator at the same time. To do that, the accelerator allocates tiles for each layer, such that each layer becomes a pipeline stage. This pipeline design allows each layer to start computation when the required kernel input pixels are ready [23].

We estimate the theoretical maximum throughput of the RRAM based accelerator (with the configuration in Table I) to be 6.5×10^3 GOP/s (e.g., giga 16-bit operations per second) for each tile. Figure 2 compares the theoretical throughput with the throughput of four RRAM-accelerated neural networks executing on the simulator. Results show a significant performance gap between theoretical throughput and simulated throughput. Based on our detailed analysis in the following section, we find that the majority of the performance gap is the data dependency related pipeline idleness.

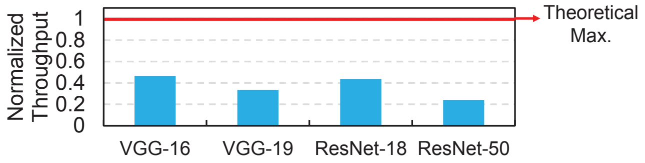

Fig. 2. The throughput of four neural networks. Numbers are normalized to the theoretical maximum throughput of the accelerator.

B. Shared Data Induced Data Dependency

Shared data between duplicate kernels emerges when pipelining and kernel duplication are applied at the same time. In kernel duplication, the feature map is evenly divided by the kernels [23]. Different kernels compute different parts of the output feature map such that none of the computation is repeated. However, when the kernel size is larger than 1×1, parts of the input feature map overlap among the duplicate kernels. Figure 3 shows the data sharing of a convolutional layer with four duplicate 2×2-sized kernels. The input feature map can be partitioned in a different way. However, shared data still exists among duplicate kernels.

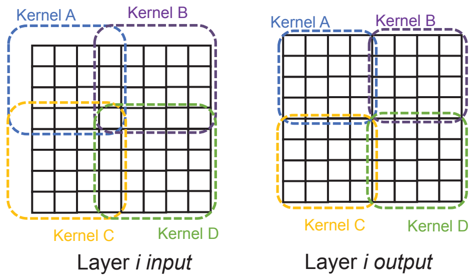

Fig. 3. Shared data in a convolutional layer with four duplicate kernels.

The shared data does not affect intra-layer parallelism by itself when all the input feature maps are prepared at the start of computation. The pipeline naturally has the data dependencies between consecutive layers. The shared data, however, further aggravates the data dependencies because the shared data blocks multiple kernels in the subsequent layers. Figure 4 illustrates an example of shared data induced data dependency in the first two convolutional layers in a DNN model. These two layers both use a 2×2-sized kernel and both kernels are duplicated twice. As the figure shows, all the output feature maps are divided into two halves. In each layer, two duplicate kernels share the middle part of the pixels of the input feature maps. During the computation, each kernel computes a single output pixel within a compute cycle.

Starting at the top left corner of its input, the kernel moves from left to right to perform computations. The first layer has all its input feature maps (a) ready at the beginning. Therefore, none of the kernels (b, c) in layer 1 are blocked. In layer 2, kernel B (f) is blocked due to the missing input data (output pixel 4 in the layer 1) (d). Kernel B needs to wait until the output pixel 4 from layer 1 is computed. We may change the order of computation in layer 1 to force kernel A to compute the shared data first. This, however, blocks the computation of kernel A (e) in layer 2. With more duplicate kernels and deeper network layers, more shared data exists in the input feature map. This aggravates the data dependency problem, inducing more idleness of the duplicate kernel.

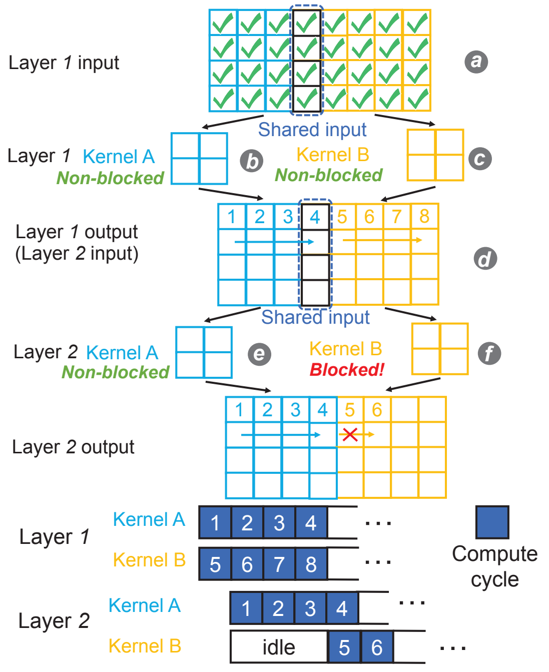

Fig. 4. An example of data sharing in the first two layers of a DNN model.

To quantify the inefficiency of the shared data induced data dependency, we simulate a pipeline-enabled RRAM-based accelerator (configuration details in Section IV) and breakdown the single inference latency, as shown in Figure 5. We breakdown the latency into five categories: computation time of crossbar arrays, idle time resulting from shared data related dependency, communication time (e.g., time for synchronization between tiles and MACs), data movement time (e.g., timing for transferring input and output data), and others.

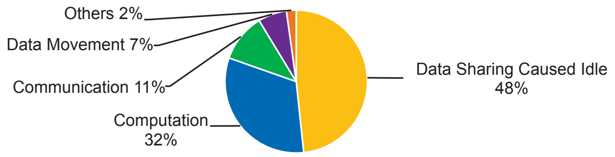

Fig. 5. Single inference latency breakdown.

Due to the pipeline, crossbar arrays belonging to different layers have significantly different cycles on each type of operation. We collect the cycles for each type of operation and calculate the breakdown for all the crossbar arrays of each layer. We calculate the overall breakdown with the geometric mean of each layer. We observe that the shared data induced idleness forms a significant portion, contributing to nearly 50% of the overall latency. This can be attributed

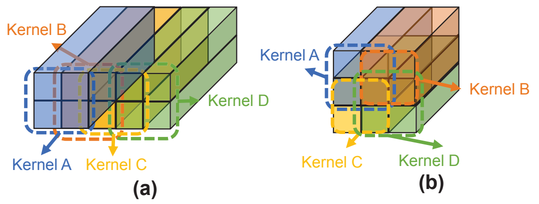

Fig. 6. Two kernel batching types: (a) linear batching, and (b) square batching.

to pending shared data blocking computations of multiple kernels. Therefore, arranging the sequence of computations on shared data becomes critical to the performance of pipeline-enabled RRAM accelerators.

III. Finer-grained Parallel RRAM Architecture

To alleviate the shared data induced data dependencies in RRAM-based accelerators, we propose Finer-grained Parallel RRAM Architecture (FPRA). FPRA aims to achieve higher parallelism with finer-grained coordination of data between different kernels and different layers. FPRA is a architecture design consisting of two mechanisms: *kernel batching* and *data sharing aware memory*. The kernel batching groups duplicate kernels and uniformly manages their data forwarding and computation. The data sharing aware memory stores shared input/output data between all batched kernels and manages data for all the kernels together to reduce unnecessary data duplication.

Kernel batching. The shared data induced dependency exists because of the partitioned input feature map for each kernel (Section II-B). We observe that enforcing all the kernels of the same layer to simultaneously compute over continuous shared data, can eliminate the shared data induced dependency and only keep the data dependency between layers. We refer this mechanism as *kernel batching*. When all the layers in the network apply kernel batching, all the kernels belonging to the same layer behave uniformly - either pending for input or computing output.

Kernel batching requires several rules to guarantee the computations follow the network configuration without being repeated on different kernels. First, the kernel batching places all the kernels together with one stride size between any adjacent two kernels. This makes sure the generated output data are continuous within each computation iteration. Second, the duplicate kernels shift in the same direction with the same gap between each computation. This makes sure the generated output data between computation iterations are continuous. Third, the duplicate kernels shifts may shift multiple strides between computations. Depending on how the kernels are batched and the direction kernels are moving, each kernel may take multiple strides to avoid recomputating the data.

We propose two types of kernel batching: linear batching and square batching. The linear batching places all the kernels in a row (or a column). Figure 6 (a) shows an example of the linear batching in a convolutional layer. The kernel size is $2\times2\times64$, and the stride size is 1. The same kernel is duplicated into four replicas in a row with indices A, B, C, and D. The batched kernels move in a vertical direction with a single stride gap. The linear batched kernels rely on rectangular-shaped input to generate a one-line output. The square batching places the same amount of kernels in consecutive rows and columns. Any two adjacent kernels still share only one stride gap. Figure 6 (b) shows an example of the square batching. In this example, we use the same kernel size and stride as the linear batching. The square batched kernels rely on the square-shaped input to generate a square-shaped output.

Two types of kernel batching have different advantages. The squared batching fit with the convolutional layer connects to the pooling layer. Because pooling layer relies on square-shaped input, if the squared batched kernels produce exactly the same sized output data for every computation iteration, the idleness between the convolutional layer and the pooling layer is minimized. However, the square batching requires the kernel duplication number to be square, which is much less feasible to the available hardware resources. The linear batching, which can be achieved with any number of duplicate kernels, is more feasible. FPRA uses linear batching and square batching simultaneously. We explore the performance impact of different strategies of batching type in Section IV.

Data sharing aware memory. The kernel batching helps reduce the shared data induced idleness in the pipeline. However, as the batched kernels share more data, the amount of input data used by all the kernels is significantly increased. For example, in a convolutional layer with 64 64×64-sized input feature maps and four kernel replicas, using linear batching can increase the amount of input data by $1.6\times$ compared to no kernel batching. Because the duplicate kernel is not aware of the shared data, the amount of data forwarding between layers can significantly increase. The increased data movement exhausts the bandwidth of NoC, and leads to limited performance increase (Section IV).

To solve the issue, we propose *data sharing aware memory* to efficiently forward data between layers, and dispatch data to duplicate kernels. The data sharing aware memory utilizes the eDRAM of each tile. The data sharing aware memory buffers a part of the input and output feature maps of the layer. As the Figure 7 shows, the feature maps data is transferred via inter-tile data synchronization through NoC. The output data of four linearly batched kernels in layer i stores in the data sharing aware memory, and transfer to the memory of layer $i+1$ together. The data sharing aware memory stores the input feature map for all the kernels as an entirety. The memory dispatches the input data to each kernel's crossbar arrays. The dispatch only happens inside the tile (e.g., intra-tile data synchronization), which costs no NoC bandwidth. This is because each kernel is loaded across multiple crossbar arrays, where crossbar arrays belong to different tiles. In this example, the data sharing aware memory dispatches data $a-d$ to the kernel A's crossbar arrays, $c-f$ to the kernel B's crossbar arrays, $e-h$ to the kernel C's crossbar arrays, and $g-j$ to the kernel D's crossbar arrays.

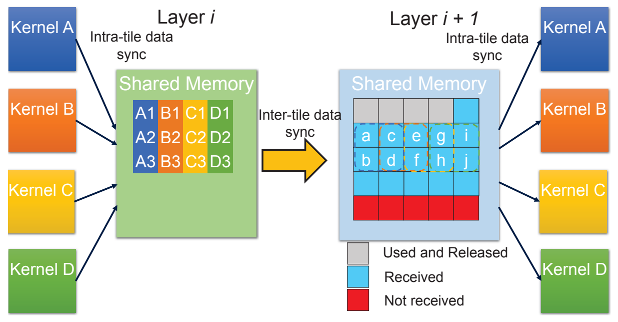

Fig. 7. Data sharing aware memory in two layers of a DNN.

Data sharing aware memory also manages the execution of batched kernels. Since data sharing is invisible to duplicate kernels, the memory utilizes *intra-tile data synchronization* to start computation of each kernel. The data sharing aware memory keeps track of the kernel's location and progress, as the duplicate kernels metadata are stored inside the memory. This provides corresponding input data for each layer upon receiving these data. As batched kernels move with the same stride and direction, the data sharing aware memory uses the *batched window* to uniformly keep track of the input and output data. Figure 8 shows an example of the batched window shifting over the input and output shared memories.

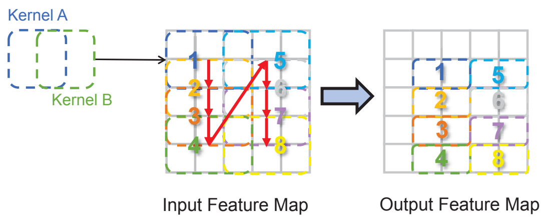

Fig. 8. Batched window shifting in the shared input/output feature maps of a convolutional layer. The layer uses two batched $2\times2\times64$-sized kernels with one stride size. When two kernels are batched together, the shared input forms a $2\times3\times64$-sized window. The numbers indicate the steps of the window movement. The red arrow represents the direction of the window movement.

Intra-tile data synchronization. As shown on Figure 9, the intra-tile data synchronization consists of the following steps. At the beginning, MACs remain idle when the input data within the batched window is still pending (❶). When all the data in the batched window is received, the tile controller notifies all the MACs, and forwards the corresponding data to the MAC belonging to each kernel (❷). The MACs start computation upon receiving their input data (❸). When computation finishes, the output data is forwarded to the shared memory (❹). Once the output data within the output batched window is received, the tile controller sends data to the other tiles (e.g., next layer's tiles) through inter-tile data synchronization. To start the next iteration of computation, the controller also shifts the batched windows for both output and input (❺). The tile is reset to pending, and the input memory re-evaluates the accessibility of data within the batched window. All MACs resume to the idle state and wait for the input data (❶).

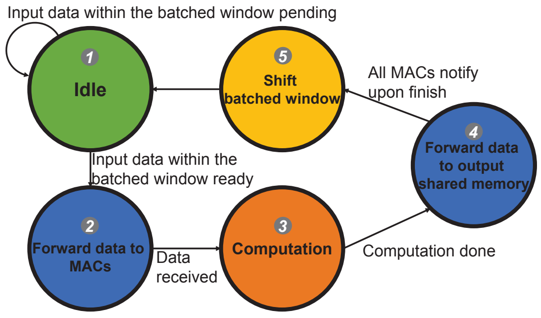
Input data within the batched window pending
1 Idle
5 Shift batched window
All MACs notify upon finish
4 Forward data to output shared memory
Input data within the batched window ready
2 Forward data to MACs
Data received
3 Computation
Computation done

Fig. 9. State diagram of the intra-tile data synchronization.

Tile Architecture. To support kernel batching and memory sharing, we design the tile-level architecture for FPRA (as shown in Figure 10). This architecture design that consists of I/O buffer, shared memory, tile controller and MACs. The I/O buffer handles the data transfer between tiles. The buffer connects to the router to handle communications with other tiles. The buffer uses two circular FIFO queues to send data and receive data separately.

The shared memory stores input/output feature maps and kernel-related metadata. The buffered input/output feature maps are stored separately and shared among all the MAC units. Kernel-related metadata contains hardware assignment

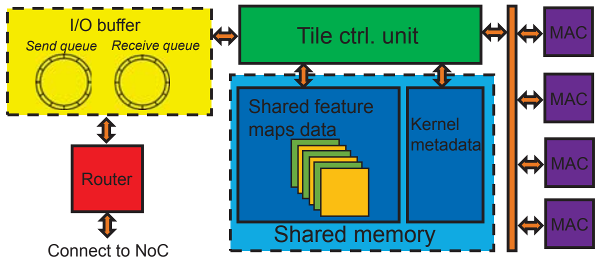

Fig. 10. Tile architecture.

data and the execution model. Unlike feature maps data, metadata are written into the memory during the accelerator's preconfiguration, and cannot be modified during the execution. Metadata is used by the tile controller to perform kernel batching and manage data sharing aware memory.

The tile controller manages the data forwarding between the rest of the components. The preconfigured metadata provides the controller information on how MACs are assigned to different kernels and how data is shared between MACs. The tile controller uses the *intra-tile data synchronization* to not only transfer the data between shared memory and MACs, but also manage the execution of MACs in the correct order.

IV. Evaluations

FPRA simulation. We build our in-house cycle-accurate simulator in Python. Our simulator implements the entire architecture of the RRAM-based accelerator, and simulates computational and data movement operations. The simulator models the pipeline between layers, such that hardware belonging to each layer has its own status (e.g., cycles and power) and uses message passing for data movements and communication. We adopt and modify the DRAM timing model in Ramulator [14] to simulate the eDRAM. For the operations in the crossbar, we include the cycle parameters of matrix multiplication, ADC, and S+A based on [8], [23]. We use the statistics from [10] to model the power consumption of crossbar array. The power and area of the tile controller are validated with System Verilog through Synopsys Design Compiler and Synopsys 32nm PDK [26]. The power and area of ADC and DAC is based on [15] and [21], respectively. We use CACTI tool [17] to model the rest of the on-chip components including eDRAM and buses under 32nm. The detailed parameters of the simulations are listed in Table I.

Baselines. We include two other baselines for our evaluation: pipeline enabled RRAM-based accelerator without kernel duplication (**w/o kernel duplication**), and pipeline enabled RRAM-based accelerator with kernel duplication (**w/ kernel duplication opt**). The implementation of these two baselines are based on the architecture described in [23] and [25].

Benchmarks. We select eight representative DNN models for ImageNet classification [7]. These networks include GoogleNet [28], MobileNet [22], ShuffleNet [16], InceptionV3 [29], ResNet [9], and VGG [24]. As FPRA and baselines only accelerate the inference operation, we use the pre-trained weights in the torchvision package [30] for the experiments. Because FPRA avoids any modification on the network models and the input data, results show no compromise over the inference accuracies (both top-one and top-five).

To utilize existing DNN models [1], [18], the simulator uses the inference trace generated by the Glow compiler [20].

TABLE I
Hardware configuration of the FPRA.

Component	Parameter	Spec.	Power
Controller	Frequency	1GHz	0.3mW
Tile	Number	168	312mW
Shared eDRAM	Size Bus-width T_{read}/T_{write}	64KB 256b 12ns	17.66mW
I/O buffer	Size	256 entries 2 queues	5.3mW
MAC	No. per tile	12	24mW
Crossbar array	No. per MAC Size Bit-width $T_{mul}/T_{ADC}/T_{S+A}$	8 128×128 2 1 cycle	2.4mW

We modify the compiler to generates a trace of the network inference. The trace includes the number of computation operations, input and output data sizes, the sequence of executions.

Performance. Figure 11 shows performance results under three metrics. Besides the FPRA configuration, we add the FPRA configuration without data sharing aware memory (referred to as **FPRA w/o data sharing aware memory**). Figure 11 (a) is the speedup over baseline w/o kernel duplication. The accelerator idle ratio in Figure 11 (b) is measured with the *geometric mean of each layer's idle ratio.* The throughput result in Figure 11 (c) is measured in giga 16-bit operations per second (GOP/s). We normalize the result to the w/ kernel duplication baseline. The result is generated with a 16-chip setup. We exclude the w/o kernel duplication, due to its underutilization of the hardware and low throughput.

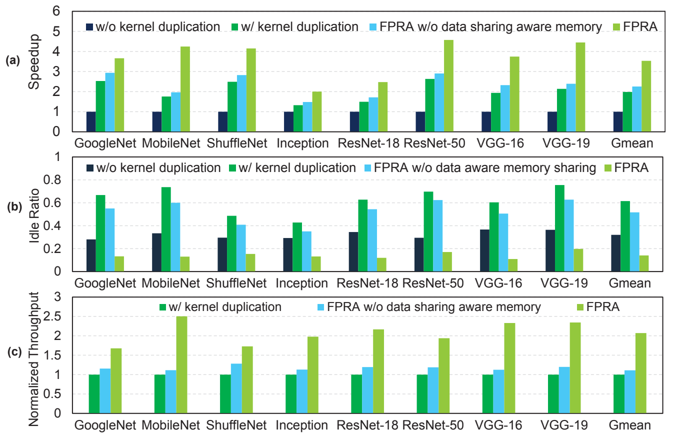

Fig. 11. Comparison of results over (a) speedup of inference latency, (b) idle ratio, and (c) throughput under different configurations.

We make two observations from the result. First, FPRA has a clear advantages over performance. We observe FPRA has the highest speedup and throughput among all configuration. FPRA has a 2.0× speedup and 2.1× throughput increase against w/ kernel duplication on average. Beside, FPRA obtains a higher speedup on the networks with more layers. For instance, compared to the w/ kernel duplication, FPRA has a 2.1× speedup on the VGG-19, and a 1.9× speedup on the VGG-16. We observe the same trend on the ResNet that the speedup is higher on ResNet-50 than on ResNet-18. This indicates FPRA brings more benefit with deeper networks, as they tend to have a more severe data dependency. We identify that FPRA achieves the lowest idle ratio among configurations. FPRA is able decrease the shared data induced idleness by 3.5× compared to w/ kernel duplication. Second, data sharing aware memory is essential to the FPRA. The FPRA w/o data sharing aware memory configuration suffers from high idle ratio. We discover that the kernel batching increases the traffic to throttle the NoC bandwidth. The FPRA with data sharing aware memory manages to reduce idle ratio and further increases over the speedup and throughput.

Batching strategies. Figure 12 compares three strategies for using two types of kernel batching. *All-linear* utilizes linear batching on all the layers. *Square-max* attempts to use square batching wherever possible. This strategy batches the kernel to the maximum possible square, and places squares next to each other. For example, for 2048 kernels, it batches them to two 32×32 square batches, and then places them side by side. The *pooling-square* utilizes square batching on the layer before the pooling layer whenever possible. This strategy first batches the kernels to the size of pooling filter, and then linearly places them in a row. For instance, for 32 kernels on a layer before 2×2 max-pooling, it batches them into eight 2×2-sized batches and places them in a row. We observe that pooling-square achieves the best performance on average. Hence, FPRA uses the pooling-square by default.

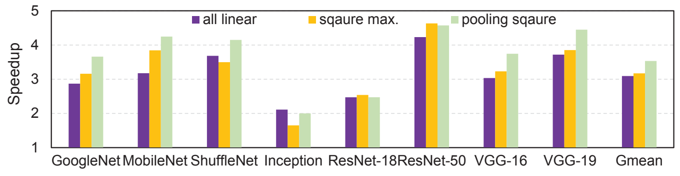

Fig. 12. Speedup of three batching strategies with FPRA.

Power efficiency. Figure 13 shows the power efficiency result of two FPRA configurations and baseline. The power efficiency is measured by giga 16-bit operations per second per watt (GOPS/W). We normalized the results of two FPRA configurations to the w/ kernel duplication opt baseline. We make two observations from the result. First, the FPRA without data sharing aware memory configuration only has a 24% increase compared to the baseline on the power efficiency. The kernel batching brings more data movements between tiles. The heavy memory traffic delays the communication between tiles. Second, the FPRA manages to achieve 2.4× power efficiency of the baseline. With kernel batching and data sharing aware memory, FPRA is able to reduce idleness during the inference, which leads to more operations with a single unit of power.

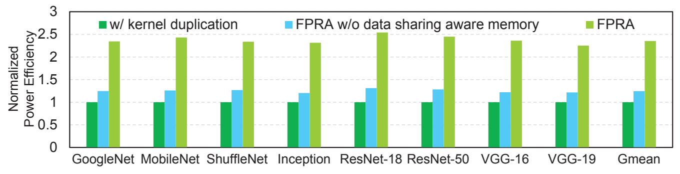

Fig. 13. Power efficiency of three configurations.

V. Conclusion

We propose FPRA, a highly-parallel RRAM-based accelerator design. FPRA applies the *kernel batching* and *data sharing aware memory* to reduce the data sharing induced idle in the pipeline. FPRA's kernel batching positions the kernel closer to each other to reduce the data sharing induced data dependencies between kernels while FPRA's data sharing aware memory efficiently manages data forwarding between layers and duplication kernels. The evaluation shows FPRA achieves 2.0× speedup improvement, 2.1× throughput increase, and 2.4× power efficiency increase compared to the state-of-the-art baseline.

Acknowledgements

We thank the anonymous reviewers from ISLPED 2021. This research is partially supported by the SRC/DARPA Center for Research on IntelligentStorage and Processing-in-memory, and the Sirindhorn International Thai-German Graduate School of Engineering (TGGS) New Researcher Grant.

References

[1] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *OSDI*, 2016.
[2] A. Ankit *et al.*, "PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *ASPLOS*, 2019.
[3] A. Boroumand *et al.*, "Google workloads for consumer devices: Mitigating data movement bottlenecks," in *ASPLOS*, 2018.
[4] Y. Chen *et al.*, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *JSSC*, 2017.
[5] P. Chi *et al.*, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *ISCA*, 2016.
[6] T. Chou *et al.*, "CASCADE: Connecting rrams to extend analog dataflow in an end-to-end in-memory processing paradigm," in *MICRO*, 2019.
[7] J. Deng *et al.*, "ImageNet: A large-scale hierarchical image database," in *CVPR*, 2018.
[8] D. Fujiki *et al.*, "In-memory data parallel processor," in *ASPLOS*, 2018.
[9] K. He *et al.*, "Deep residual learning for image recognition," in *CVPR*, 2016.
[10] M. Hu *et al.*, "Dot-product engine for neuromorphic computing: Programming 1t1m crossbar to accelerate matrix-vector multiplication," in *DAC*, 2016.
[11] Y. Ji *et al.*, "Bridge the gap between neural networks and neuromorphic hardware with a neural network compiler," in *ASPLOS*, 2018.
[12] Y. Ji *et al.*, "FPSA: A full system stack solution for reconfigurable ReRAM-Based NN accelerator architecture," in *ASPLOS*, 2019.
[13] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017.
[14] Y. Kim *et al.*, "Ramulator: A fast and extensible dram simulator," *CAL*, 2016.
[15] L. Kull *et al.*, "A 3.1mW 8b 1.2GS/s single-channel asynchronous SAR ADC with alternate comparators for enhanced speed in 32nm digital SOI CMOS," in *ISSCC*, 2013.
[16] N. Ma *et al.*, "ShuffleNet V2: Practical guidelines for efficient CNN architecture design," in *ECCV*, 2018.
[17] N. Muralimanohar *et al.*, "CACTI 6.0: A tool to model large caches," *HP laboratories*, 2009.
[18] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *NIPS*, 2019.
[19] X. Qiao *et al.*, "Atomlayer: A universal ReRAM-Based CNN accelerator with atomic layer computation," in *DAC*, 2018.
[20] N. Rotem *et al.*, "Glow: Graph lowering compiler techniques for neural networks," 2018.
[21] M. Saberi *et al.*, "Analysis of power consumption and linearity in capacitive digital-to-analog converters used in successive approximation adcs," *TCAS-I*, 2011.
[22] M. Sandler *et al.*, "MobileNetV2: Inverted residuals and linear bottlenecks," in *CVPR*, 2018.
[23] A. Shafiee *et al.*, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *ISCA*, 2016.
[24] K. Simonyan *et al.*, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
[25] L. Song *et al.*, "PipeLayer: A pipelined ReRAM-Based accelerator for deep learning," in *HPCA*, 2017.
[26] Synopsys, "Teaching resources for ic design," 2020, https://www.synopsys.com/community/university-program/teaching-resources.html.
[27] V. Sze *et al.*, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, 2017.
[28] C. Szegedy *et al.*, "Going deeper with convolutions," in *CVPR*, 2015.
[29] C. Szegedy *et al.*, "Rethinking the inception architecture for computer vision," in *CVPR*, 2016.
[30] Torch Contributors, "torchvision – pytorch master documentation," 2019, https://pytorch.org/docs/stable/torchvision/index.html#torchvision.
[31] S. Yu, "Neuro-inspired computing with emerging nonvolatile memorys," *Proceedings of the IEEE*, 2018.