An Efficient Programming Framework for Memristor-based Neuromorphic Computing

[1]Grace Li Zhang, [1]Bing Li, [1]Xing Huang, [1]Chen Shen, [1]Shuhang Zhang, [1]Florin Burcea, [1]Helmut Graeb, [2]Tsung-Yi Ho, [3]Hai (Helen) Li, and [1]Ulf Schlichtmann
[1]*Technical University of Munich,* [2]*National Tsing Hua University,* [3]*Duke University*
{grace-li.zhang, b.li, xing.huang, chen.shen, shuhang.zhang, florin.burcea, helmut.graeb, ulf.schlichtmann}@tum.de, tyho@cs.nthu.edu.tw, hai.li@duke.edu

***Abstract*—Memristor-based crossbars are considered to be promising candidates to accelerate vector-matrix computation in deep neural networks. Before being applied for inference, memristors in the crossbars should be programmed to conductances corresponding to the network weights after software training. Existing programming methods, however, adjust conductances of memristors individually with many programming-reading cycles. In this paper, we propose an efficient programming framework for memristor crossbars, where the programming process is partitioned into the predictive phase and the fine-tuning phase. In the predictive phase, multiple memristors are programmed simultaneously with a memristor programming model and IR-drop estimation. To deal with the programming inaccuracy resulting from process variations, noise and IR-drop and move conductances to target values, memristors are fine-tuned afterwards to reach a specified programming accuracy. Simulation results demonstrate that the proposed method can reduce the number of programming-reading cycles by up to 94.77% and 90.61% compared to existing one-by-one and row-by-row programming methods, respectively.**

***Index Terms*—memristor, neuromorphic computing, programming, IR-drop, process variations, noise**

I. Introduction

To accelerate computations in deep neural networks, various hardware platforms with emerging devices, e.g, memristor [1]–[6], Mach Zehnder Interferometer [7]–[10], spintronics device [11], [12] and Ferroelectric Field-Effect Transistor [13]–[16] have been introduced. Among these platforms, memristor-based crossbar has the advantages of high computing efficiency and low power consumption. The structure of such a memristor crossbar is shown in Fig. 1, where memristors sit at the crossing points of horizontal wordlines and vertical bitlines. In this crossbar, transistors are used as current limiters to avoid permanent dielectric breakdown and sneak paths as well as for selecting memristors during programming [17].

To deploy memristor crossbars for neuromorphic computing, the conductances of memristors should be programmed to the values corresponding to weights in a neural network after software training to perform vector-matrix computation. The vector can be represented by input voltages applied onto the horizontal wordlines. When a voltage is applied onto a memristor, the resulting current is the multiplication of the voltage and the conductance of the memristor. The currents through memristors in a column are summed up naturally following Kirchoff's Law. Consequently, the result of the vector-matrix multiplication is represented by the currents on the vertical bitlines [2]. With this analog computing style, memristor-based crossbars are highly efficient in terms of computation and power consumption [18].

To apply memristor-based crossbars in practice, the challenge is that a huge number of memristors need to be programmed to represent the corresponding weights in a neural network, e.g., 60 million weights in ResNet-152, leading to a time-consuming initialization of the crossbars before they are ready for computing tasks. To overcome the challenge, several methods have been proposed. In [19], a fully parallel programming scheme is demonstrated with a 2×2 crossbar to accelerate the weight update of neural networks, whose training is implemented on memristor crossbars. In [20], a feedback algorithm is introduced to tune the conductances of memristors with iterative programming and reading pulses, while device variations are determined by reading operations. In [21], the algorithm in [20] is improved with respect to initial voltages, voltage steps and pulse widths.

Most implementations of methods described above program and read memristors on crossbars individually, leading to a large number of programming-reading cycles. To accelerate the application of memristor crossbars in neuromorphic computing, we propose an efficient programming framework. The contributions of this paper are summarized as follows. 1) The programming process is partitioned into a predictive phase and a fine-tuning phase to program memristors from coarse to fine. 2) In the predictive phase, with a memristor programming model, an offline IR-drop-aware grouping algorithm is proposed to select memristors in multiple rows for programming. 3) In the fine-tuning phase, the memristor programming model is calibrated and memristors are fine-tuned further based on the calibrated model. In addition, the sensitivities of weights to the accuracy of neural networks are exploited to assign different tolerance accuracy ranges for target conductances of memristors.

The remainder of this paper is organized as follows. The background of programming memristors in crossbars is explained in Section II. In Section III, we explain the details of the proposed method. Simulation results are shown in Section IV and conclusions are drawn in Section V.

II. Background

A. Programming Characteristics of Memristors

Memristors are a type of resistive memory cells made out of solid-state materials. Inside the oxide layer of a memristor, oxygen vacancies generated during manufacturing can form a conductive filament. The length of this filament can be modulated by a voltage, and the conductance of the memristor

978-3-9819263-5-4/DATE21/©2021 EDAA

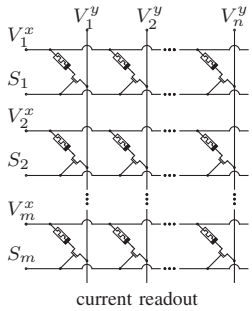


Fig. 1: Memristor-based crossbar with control lines $S_1, S_2, \ldots S_m$.

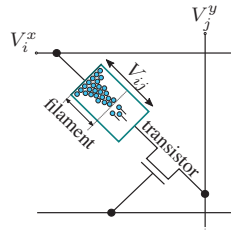
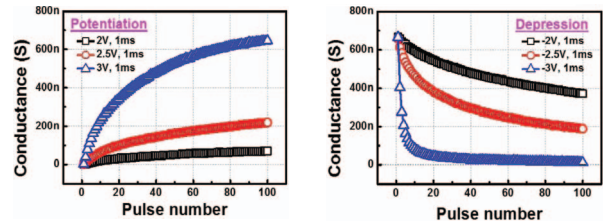


Fig. 2: Programming of memristor by modulating filament.



(a) (b)

Fig. 3: Non-linear programming characteristics of memristors in [24]. (a) The conductance is set from low to high. (b) The conductance is reset from high to low.

can thus be changed accordingly. The process to modulate the conductance of a memristor is called programming, as illustrated in Fig. 2. To simulate the behavior of memristors, memristor programming models have been proposed in [22], [23].

In programming memristors, the relation between the applied voltage and the change of conductance is non-linear, as shown in Fig. 3, where potentiation sets the conductance from low to high and depression resets the conductance from high to low [24]. When a positive voltage is applied onto the memristor, as shown in Fig. 3(a), the conductance is gradually increased with the number of voltage pulses. Although a larger positive voltage increases the conductance by a larger amount, the conductance change and programming voltage do not have a linear relationship. Similarly, a negative voltage decreases the conductance of the memristor, but not linearly either, as shown in Fig. 3(b).

Another programming characteristic shown in Fig. 3 is that the set process where conductances change from low to high and the reset process where conductances change from high to low are not symmetrical. If a positive voltage is used to set a conductance to a certain value, then applying a negative voltage of the same magnitude does not reset the conductance to its previous value. For example, the reset process with the voltage -3V in Fig. 3(b) changes the conductance of the memristor more abruptly compared with the set process with the voltage 3V in Fig. 3(a).

B. Programming Memristor Crossbars

Due to the complex programming characteristics of memristors described above, programming-reading on individual memristors has been widely applied to program memristors on crossbars. In this method, memristors are activated individually for programming, by enabling the transistor connected to the memristor shown in Fig. 1. Afterwards, the voltages on the corresponding wordline and bitline create a relative voltage upon the memristor to change its conductance. For example, to program the memristor in the second row and the second column in Fig. 1, S_2 can be set to a high voltage to enable the transistor and V_2^y can be set to the programming pulse with an appropriate voltage while the second horizontal bitline V_2^x can be connected to ground. In addition, other wordlines and bitlines should be connected to ground to guarantee only the selected memristor is programmed. The application of such a programming voltage pulse on a memristor modulates its conductance and is called a programming cycle.

After each programming cycle, the current conductance of the memristor is read and a new voltage is determined accordingly to program the memristor further, until the conductance falls within the given range around the target value. This programming-reading mechanism can deal with process variations, noise and IR-drop effectively, since the consequences of these factors are already reflected in the conductance that is read out. This one-by-one programming scheme, however, leads to a huge number programming-reading cycles to update all the memristors in a crossbar, especially for large neural networks.

To improve the programming efficiency, a row-by-row programming scheme can be adopted [17]. This method selects the ith row by setting the control signal S_i to high. Afterwards, the memristors in this row are programmed together by setting the required programming voltages in the corresponding columns. In the reading cycle, the conductances of these memristors can be read out within one cycle simultaneously.

C. Challenges in Programming Memristors in Crossbars

Process variations are deviations of process parameters from their nominal specifications after manufacturing. These deviations cause variability in electrical properties of memristors. Consequently, when a voltage is applied to a memristor, the corresponding conductance change under process variations deviates from its nominal conductance change. Noise, also called temporal variations of memristors, is caused by the stochastic formation and rupture of filaments within programming cycles. It affects the programming process of memristors unexpectedly and can only be verified with reading operations.

Besides process variations and noise, IR-drop, resulting from current flowing through wire resistances of memristor crossbars, also affects the effectiveness of parallel programming. IR-drop leads to a voltage drop applied on a memristor. For example, in Fig. 1, the actual voltage V_{ij} applied on a memristor on the ith row and jth column degrades from the voltage difference V_i^x-V_j^y between the two terminals due to the current flowing through wire resistances. When the size of a memristor crossbar becomes large, the effect of IR-drop becomes considerable.

III. Proposed Programming Framework

To improve the programming efficiency of memristor crossbars, we propose an efficient programming framework, where

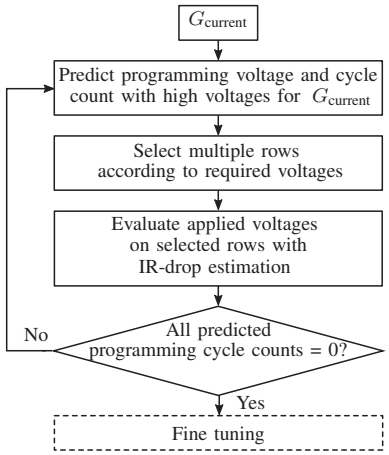


Fig. 4: Predictive phase of the proposed programming framework performed offline.

the programming process is partitioned into a predictive phase and a fine-tuning phase.

A. Predictive Programming Phase

In the predictive programming phase, multiple rows of memristors in the crossbar are selected to be programmed simultaneously. The work flow of the predictive phase is illustrated in Fig. 4.

1) Prediction with the Largest Voltage Amplitude V^{max}

Before programming, the initial conductances $G_{current}$ of memristors might be far from their target conductances. To move the conductances of memristors to their target values quickly, in the predictive phase, we only use the available voltage with the largest amplitude, denoted as V^{max}, to program memristors, since a finer granularity of programming voltages does not program memristors accurately due to process variations and noise. The polarity of this voltage for a memristor is determined according to its programming direction. For example, if the initial conductance of a memristor R_{ij} at the ith row and the jth column on the crossbar is much smaller than its target conductance, the programming voltage is required to be $+V^{max}$. We also call this voltage the required voltage for the memristor R_{ij}, denoted as $V_{ij}^{required}$, so that $V_{ij}^{required} = +V^{max}$ in this case. On the contrary, if the initial conductance of a memristor R_{ij} is much larger than its target conductance, its required voltage is $-V^{max}$, also $V_{ij}^{required} = -V^{max}$. If the initial conductance of the memristor is close to its target conductance, its required voltage is 0.

In each programming cycle, a memristor is programmed with its required voltage, so that its initial conductance approximates its target conductance gradually. This programming process continues until the resulting conductance cannot approximate the target conductance any more. The number of programming cycles is thus called the ***predicted programming cycle count***, denoted as C_{ij} for the memristor R_{ij}. After each programming cycle, C_{ij} may change, so that it should be reevaluated.

Fig. 5 illustrates an example, where memristors on a row of a crossbar are programmed with $\pm 1.5V$. For example,

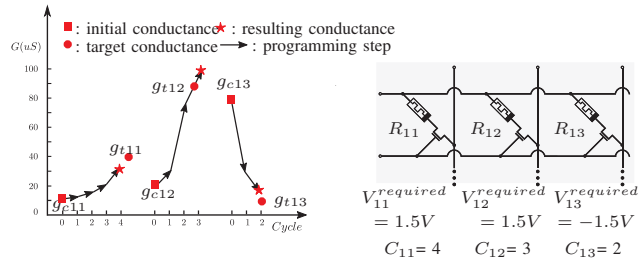


Fig. 5: The predicted programming cycle counts with the largest voltage amplitude for memristors on a row of a crossbar.

3 predicted programming cycles with 1.5V are required to program R_{12} roughly from the initial conductance g_{c12} to the target conductance g_{t12}. Similarly, predicted programming cycle counts of 4 and 2 are required for memristors R_{11} and R_{13}, respectively. After being applied with a series of voltages, the resulting conductances shown as stars are close to the target conductances shown as dots. But usually they are not identical to the target values, due to the coarse granularity of conductance changes generated by high voltages.

2) Grouping Multiple Rows According to Required Voltages of Memristors

When selecting multiple rows to be programmed simultaneously, the memristors on the selected rows should be programmed to their target conductances as much as possible. To achieve this goal, we consider the memristors that require the largest predicted programming cycle count on a row as ***dominant memristors*** of the row, whose number might be more than 1. In a row-wise programming scheme, the predicted programming cycle counts of dominant memristors determine the overall predicted programming cycle count of the whole row. Therefore, we reduce the predicted programming cycle counts of the dominant memristors in a programming cycle to improve the programming efficiency. To guarantee the reduction of the overall predicted programming cycle count of the whole row, the reevaluated predicted programming cycle counts of nondominant memristors should not exceed the previous programming cycle counts of the dominant memristors, after a programming cycle is finished.

To reduce the predicted programming cycle count of a row, we first search the dominant memristors on this row. Afterwards, we assign the required voltage to be the programming voltage of the dominant memristor to guarantee that its predicted programming cycle count can be reduced by 1. Since the voltages on the horizontal wordlines are connected to ground, the voltage V_{ij} imposed on R_{ij} is equal to the negative of the voltage imposed on the bitline, written as $-V_j^y$. The constraint described above can be written as follows

$$V_{ij} = -V_j^y = V_{ij}^{required}, \text{for dominant memristor } R_{ij} \quad (1)$$

where V_{ij} and V_j^y are variables we need to determine. $V_{ij}^{required}$ is a constant computed previously.

To avoid that the reevaluated predicted programming cycle counts of nondominant memristors exceed those of the dominant memristors on the same row after a programming cycle, we reevaluate the predicted programming cycle counts of nondonimant memristors with $+V^{max}$ and $-V^{max}$ in

this programming cycle, separately. If $+V^{max}$ applied on a nondominant memristor R_{ij} makes its reevaluated predicted programming cycle count C_{ij}^{+} exceed the programming cycle count C_i^d of the dominant memristors on the same row, we constrain the voltage applied on this memristor to exclude $+V^{max}$. Similarly, if $-V^{max}$ applied on a nondominant memristor R_{ij} makes its reevaluated predicted programming cycle count C_{ij}^{-} exceed C_i^d, we constrain the voltage V_{ij} to exclude $-V^{max}$. The constraints described above can be written as follows

$$V_{ij} = -V_j^y < +V^{max},\ C_{ij}^{+} > C_i^d \tag{2}$$

$$V_{ij} = -V_j^y > -V^{max},\ C_{ij}^{-} > C_i^d. \tag{3}$$

To determine which rows in a large crossbar can be grouped for programming in parallel, a variable S_i is assigned for the ith row to indicate whether this row is enabled to be programmed or not, with $S_i = 1$ representing that this row is enabled and vice versa. If $S_i = 1$, the dominant memristors on this row should be programmed with required voltages. If $S_i = 0$, the voltages applied on vertical bitlines do not affect the memristors on this row.

According to the analysis above, we can formulate the multi-row programming problem as follows

$$\text{Maximize} \quad \sum_{i=1}^{m} S_i \tag{4}$$

$$\text{subject to} \quad (1)\text{–}(3),\ \text{if } S_i = 1 \tag{5}$$

where m is the total number of rows. The conditional constraint (5) can be converted into linear constraints as described in [25]. After solving the formulation above, the enabled rows and the corresponding voltages on the vertical lines can be obtained.

3) Reevaluation of Applied Voltages with IR-drop Estimation

In the formulation (4)–(5), the voltage V_{ij} applied on a memristor R_{ij} is assumed to be the difference between the voltage on the horizontal wordline and the voltage on the vertical bitline. However, due to wire resistance on the crossbar, the real voltage applied on this memristor deviates from this value. To consider the effects of IR-drop on the selected rows, we adopt the Modified Nodal Analysis (MNA) [26] with the wire resistances set according to [27] to establish the relation between the real applied voltages on memristors and voltage sources on vertical bitlines. This relation can be expressed as follows

$$V_{ij} = \sum_{j=1}^{n} \beta_j \times V_j^y \tag{6}$$

where n is the total number of columns of a crossbar and β_j is the parameter to indicate the contribution by the voltage V_j^y on the jth column. Consequently, the voltage on a memristor is actually affected by all the voltages on vertical bitlines.

Considering IR-drop, the formulation (4)–(5) should be revised by replacing the $V_{ij} = -V_j^y$ in (1)–(3) with (6). For example, (2)–(3) becomes

$$V_{ij} = \sum_{j=1}^{n} \beta_j \times V_j^y < +V^{max},\ C_{ij}^{+} > C_i^d \tag{7}$$

$$V_{ij} = \sum_{j=1}^{n} \beta_j \times V_j^y > -V^{max},\ C_{ij}^{-} > C_i^d. \tag{8}$$

However, this replacement makes the formulation very complicated and time-consuming to be solved. To simplify the formulation considering IR-drop, in the modified formulation described as follows, we only enable the rows obtained from the formulation (4)–(5) in Section III-A2, while other rows are forced to be disabled.

Another problem is that providing dominant memristors with exact required voltages becomes difficult, since the real voltages applied on memristors are determined by all voltage sources according to (6). Therefore, we relax the condition in (1) by allowing the programming voltage of a dominant memristor to be smaller than the required voltage and we minimize the difference between the actually applied voltage and the required voltage. The constraint described above can be written as

$$V_{ij} = \sum_{j=1}^{n} \beta_j \times V_j^y < V_{ij}^{required}, \text{for dominant memristor } R_{ij} \tag{9}$$

and the formulation to determine the voltages on vertical lines considering IR-drop becomes

$$\text{Minimize} \quad \sum_{R_{ij} \in \mathbf{R^d}} |V_{ij}^{required} - V_{ij}| \tag{10}$$

$$\text{subject to} \quad (7)\text{–}(9),\ \text{for } S_i = 1 \text{ obtained from } (4)\text{–}(5) \tag{11}$$

where $\mathbf{R_d}$ is the set of dominant memristors.

The multi-row programming with IR-drop estimation described above is deployed in each programming cycle repeatedly. The predictive phase is finished when the predicted programming cycle count of each memristor is reduced to 0. Due to the unknown process variations and noise, the predictive phase only programs memristor crossbars with a coarse granularity. Therefore, memristors should be fine-tuned further to approximate the target conductances.

B. The Fine-tuning Phase

At the beginning of the fine-tuning phase, the current conductances of the memristors are read once and used as the new initial values for programming. Thereafter, to gather the information of variations, a certain number of memristors, 80% set in the experiments, spread across the crossbar are selected and programmed with a given voltage. The resulting conductances of these memristors are read again and the real changes of these conductances are compared with the changes estimated with the memristor programming model. The average of the ratios of real and estimated conductance changes is used to calibrate the memristor programming model, so that it reflects the real properties of the memristors more accurately.

Since the memristor programming model is calibrated after reading operations, the fine-tuning is performed online. To avoid much computation in this phase, we deploy the row-by-row programming scheme and the one-by-one programming scheme instead of selecting multiple rows.

The row-by-row fine-tuning above terminates when the conductances of all memristors predicted with the calibrated model fall into their corresponding tolerance accuracy ranges of the target conductances. Since the importance of weights to the accuracy of neural networks is not the same, memristors can be programmed into different tolerance ranges. To determine such

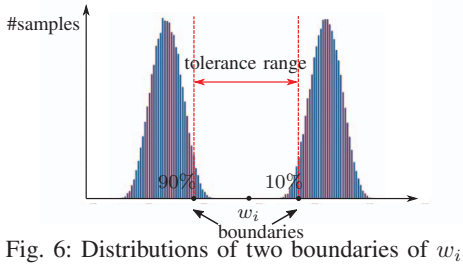


Fig. 6: Distributions of two boundaries of w_i.

ranges, we search the boundaries of the weight tolerance range by maintaining a given inference accuracy. We first sample a given number search directions N for the weights after software training and search continuously in the sampled directions to find the set of weights that form the boundary of the accuracy tolerance. The search magnitudes of weights are set to be the reverse of their sensitivities to the cost function of the neural network. Afterwards, the boundary distributions of weights can be obtained. Fig. 6 illustrates the boundary distribution of one weight in N search samples, where the x-axis represents the values of the weight and the y-axis represents the number of samples for each value of the weight. The two distributions are the results of changing the weights in the positive and the negative directions, respectively. To maximize the possibility of achieving an acceptable accuracy, we set the tolerance boundary of w_i to the 90% and 10% probabilities of the two distributions.

After the row-by-row fine-tuning terminates, reading operations are performed individually to verify the programming accuracy of memristors. If a given number of memristors, set to 80% in the experiments, are programmed within the corresponding tolerance ranges, inference is tested on memristor crossbars. If the inference accuracy satisfies a specified threshold, set to 90% of the accuracy after software training, the memristor crossbar is considered to be programmed successfully. If either the programming accuracy or the inference accuracy cannot satisfy the requirements, memristors whose conductances are not within the corresponding tolerance ranges after reading are further programmed with one-by-one fine-tuning until the inference accuracy achieves the specified value.

IV. Simulation Results

To evaluate the efficiency of the proposed programming method, 128 × 128 memristor crossbars [28] were used to implement three neural networks, a single-layer fully-connected neural network (FCNN), LeNet5 and ResNet20 and tested with MNIST and Cifar10. The proposed programming framework and the neural networks were implemented using Gurobi [25] and Tensorflow [29] respectively, and tested with an Intel 2.67 GHz CPU and a GTX1080Ti GPU. The memristor programming model in [30] was used. However, the proposed method can work with any other memristor programming models. In the predictive programming phase in Section III-A1, V^{max} was set to 1.5V [30]. In addition, ±1V to ±3V in a step of 0.05V were provided for programming to achieve ±1.5V under IR-drop and for fine-tuning and the width of the voltage pulses was set to 10ns [30]. To avoid significant IR-drop, we restrict the number of rows for parallel programming to be smaller than a given number, 20. The number of sampled search directions in Section III-B is set to 100000.

To verify the efficiency of the proposed programming method, the initial conductances of memristors on crossbars were randomly generated. The target conductances were derived according to weights after software training with linear mapping. We emulated the programming process of 100 types of memristor crossbars by sampling process variations and noise for each test combination, e.g., FCNN+MNIST. We assumed that global variations cause that the conductance changes deviate from the nominal conductance changes by up to 30% and local variations cause the deviation of conductance changes by up to 6%, leading to a total of 36% deviations, when a voltage is applied on a memristor. The noise causes the deviation of the conductance changes by 3% of the nominal conductance changes for memristors [31].

For comparison, we implemented the one-by-one programming mechanism which programs memristors individually. In addition, we implemented a row-by-row mechanism, which processes all the memristors in a row simultaneously. The voltage selection method in [20] was used in the one-by-one and row-by-row mechanisms. The programming tolerance of memristors in the two mechanisms was set to 0.5% of the target conductances. The inference accuracy with the two mechanisms is almost the same with the accuracy with weights after software training.

The comparison of the two methods above and the proposed method is shown in Table I. The inference accuracy after software training Acc^t is shown in the third column. The numbers of 128 × 128 crossbars required to implement the three neural networks are shown in the fourth column.

The proposed method consists of the predictive programming phase and the fine-tuning phase. C_{pre} in Table I is the average number of the programming cycles of 100 types of emulated memristor crossbars in the predictive phase. Since only fixed high voltages generating large conductance changes were used for programming, the number of programming cycles in the predictive phase is not very large. C_{fr} and C_{fo} represent the average number of programming-reading cycles of the row-by-row fine-tuning and one-by-one fine-tuning.

The total number of programming-reading cycles in the proposed programming method is presented in C_{total}, with $C_{total} = C_{pre} + C_{fr} + C_{fo}$. The inference accuracy with the memristor crossbars programmed with the proposed method is shown as Acc^I. This accuracy is slightly lower than that at the software level, since the programming process terminates when the inference accuracy with memristor crossbars achieves 90% of the original value.

The average numbers of programming-reading cycles with the existing one-by-one [20] and row-by-row [17] methods are shown as the column C_{one} and C_{row}, respectively. The reduction of the programming-reading cycles with the proposed method from the results of one-by-one and row-by-row programming methods are shown in the columns r_{one} and r_{row}, respectively. These results demonstrate that our method can reduce the programming-reading cycles by up to 94.77% and 90.61%, respectively, and thus improve the programming

TABLE I: Comparisons in Programming Efficiency

Testcases				Our Method					Comparison				W/O Pre	Runtime
$Network$	$Dataset$	Acc^t	$\#Cro$	C_{pre}	C_{fr}	C_{fo}	C_{total}	Acc^I	C_{one}	C_{row}	r_{one}	r_{row}	C_{oft}	$T(s)$
FCNN	MNIST	92.44%	7	148	17927	0	18075	90.39%	324724	193406	94.43%	90.61%	21378	17
LeNet5	Cifar10	76.82%	9	444	133829	51402	185675	72.93%	2534413	1344836	92.67%	86.19%	211114	1846
ResNet20	Cifar10	91.29%	59	2675	578210	0	580885	87.79%	11117100	5940846	94.77%	90.22%	745623	3106

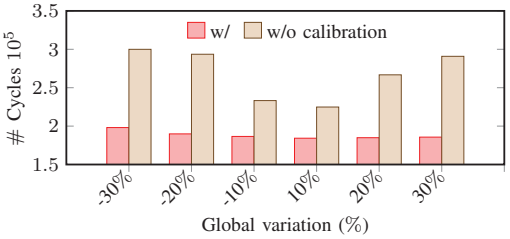


Fig. 7: Comparison between the total number of programming-reading cycles of the implementations of LeNet5 with and without calibration of the memristor model.

efficiency significantly.

To demonstrate the effectiveness of the fine-tuning programming phase, we implemented the proposed method without the predictive phase. The total numbers of programming-reading cycles with the pure fine tuning of the proposed method is shown as the column C_{oft}. C_{oft} is still significantly smaller than those of row-by-row and one-by-one methods, thanks to the calibration of the memristor programming model and assignments of the different tolerance accuracy ranges for memristors.

The last column in Table I shows the runtime to calculate the voltage series in programming cycles in the predictive phase. This step is executed off-line so that the execution is not significantly relevant.

To demonstrate the effectiveness of the calibration of memristor models, we implemented the proposed programming method without the calibration when global variations contribute to the deviations of the conductance changes with different ratios. The comparisons between the total number of programming-reading cycles of the proposed method with and without the calibration of the predictive model are shown in Fig. 7. It can be seen clearly that the calibration of the predictive model improves the programming efficiency significantly. In addition, the larger the global process variations are, the more effective the calibration technique becomes.

In the proposed method, we assign different tolerance accuracy ranges to the target conductances of memristors according to the importance of the corresponding weights, as described in Section III-B. Our experimental results show that a huge number of memristors can be programmed within 2%–4% of their target conductances instead of the strict 0.5% range, while still maintaining a specified inference accuracy.

V. Conclusions

In this paper, we have proposed an efficient programming framework for memristor crossbars. Instead of programming memristors individually, programming parallelism is maximally explored by deploying multi-row and row-by-row programming considering the effects of IR-drop. Consequently, the number of programming-reading cycles can be reduced significantly.

References

[1] M. Hu, H. Li, Y. Chen, Q. Wu, and G. S. Rose, "BSB training scheme implementation on memristor-based circuit," in *Computational Intelligence for Security and Defense Applications*, 2013, pp. 80–87.

[2] C. Liu, M. Hu, J. P. Strachan, and H. Li, "Rescuing memristor-based neuromorphic design with high defects," in *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE, 2017, pp. 1–6.

[3] S. Zhang, G. L. Zhang, B. Li, H. H. Li, and U. Schlichtmann, "Lifetime enhancement for rram-based computing-in-memory engine considering aging and thermal effects," in *IEEE International Conference on Artificial Intelligence Circuits and Systems*, 2020, pp. 11–15.

[4] Y. Zhu, G. L. Zhang, T. Wang, B. Li, Y. Shi, T.-Y. Ho, and U. Schlichtmann, "Statistical training for neuromorphic computing using memristor-based crossbars considering process variations and noise," in *Proc. Design, Autom., and Test Europe Conf.*, 2020, pp. 1590–1593.

[5] S. Zhang, G. L. Zhang, B. Li, H. H. Li, and U. Schlichtmann, "Aging-aware lifetime enhancement for memristor-based neuromorphic computing," in *Proc. Design, Autom., and Test Europe Conf.*, 2019, pp. 1751–1756.

[6] S. Zhang, B. Li, H. H. Li, and U. Schlichtmann, "A pulse-width modulation neuron with continuous activation for processing-in-memory engines," in *Proc. Design, Autom., and Test Europe Conf.*, 2020, pp. 1426–1431.

[7] Y. Shen, N. C. Harris, and S. Skirlo, "Deep learning with coherent nanophotonic circuits," *naturephotonics*, vol. 11, pp. 441–446, 2017.

[8] Z. Zhao, D. Liu, M. Li, Z. Ying, L. Zhang, B. Xu, B. Yu, R. T. Chen, and D. Z. Pan, "Hardware-software co-design of slimmed optical neural networks," in *Proc. Asia and South Pacific Des. Autom. Conf.*, 2019, pp. 705–710.

[9] M. Y.-S. Fang, S. Manipatruni, C. Wierzynski, A. Khosrowshahi, and M. R. DeWeese, "Design of optical neural networks with component imprecisions," *Opt. Express*, vol. 27, pp. 14 009–14 029, 2019.

[10] Y. Zhu, G. L. Zhang, B. Li, X. Yin, C. Zhuo, H. Gu, T.-Y. Ho, and U. Schlichtmann, "Countering variations and thermal effects for accurate optical neural networks," in *Proc. Int. Conf. Comput.-Aided Des.*, 2020.

[11] J. Grollier, D. Querlioz, K.Y.Camsari, K.Everschor-Sitte, S. Fukami, and M. D. Stiles, "Neuromorphic spintronics," *IEEE Transactions on Electron Devices*, 2020.

[12] A. Sengupta, K. Yogendra, and K. Roy, "CMOS spintronic devices for ultra-low power neuromorphic computation," in *IEEE international symposium on Circuits and Systems (ISCAS)*, 2016, pp. 922–925.

[13] M. Li, X. Yin, X. S. Hu, and C. Zhuo, "Nonvolatile and energy-efficient feFET-based multiplier for energy-harvesting devices," in *Proc. Asia and South Pacific Des. Autom. Conf.*, 2019, pp. 526–567.

[14] X. Yin, C. Li, Q. Huang, L. Zhang, M. Niemier, X. S. Hu, C. Zhuo, and K. Ni, "FeCAM: A universal compact digital and analog content addressable memory using ferroelectric," *IEEE Transactions on Electron Devices*, vol. 67, no. 7, pp. 2785–2792, 2020.

[15] K. Ni, X. Yin, A. F. Laguna, S. Joshi, S. Dünkel, M. Trentzsch, J. Müller, S. Beyer, M. Niemier, X. S. Hu, and S. Datta, "Ferroelectric ternary content-addressable memory for one-shot learning," *Nature Electronics*, vol. 2, pp. 521–529, 2019.

[16] X. Yin, K. Ni, D. Reis, S. Datta, M. Niemier, and X. S. Hu, "An ultra-dense 2fefet tcam design based on a multi-domain feFET model," *IEEE Transactions on Circuits and Systems II: Express Briefs (TCAS II)*, 2018.

[17] E. J. Merced-Grafals, N. Dávila, N. Ge, R. S. Williams, and J. P. Strachan, "Repeatable, accurate, and high speed multi-level programming of memristor 1T1R arrays for power efficient analog computing applications," *Nanotechnology*, vol. 27, no. 36, pp. 365 202:1–9, 2016.

[18] M. Hu, J. P. Strachan, Z. Li, E. M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, J. J. Yang, and R. S. Williams, "Dot-product engine for neuromorphic computing: programming 1t1m crossbar to accelerate matrix-vector multiplication," in *Proc. Design Autom. Conf.*, 2016, pp. 1–6.

[19] L. Gao, I.-T. Wang, P.-Y. Chen, S. Vrudhula, J.-s. Seo, Y. Cao, T.-H. Hou, and S. Yu, "Fully parallel write/read in resistive synaptic array for accelerating on-chip learning," *Nanotechnology*, vol. 26, no. 45, pp. 455 204:1–9, 2015.

[20] F. Alibart, L. Gao, B. D. Hoskins, and D. B. Strukov, "High precision tuning of state for memristive devices by adaptable variation-tolerant algorithm," *Nanotechnology*, vol. 23, no. 7, pp. 075 201:1–9, 2012.

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

[22] P. Yao, H. Wu, B. Gao, J. Tang, Q. Zhang, W. Zhang, J. J. Yang, and H. Qian, "Fully hardware-implemented memristor convolutional neural network," *Nature*, vol. 577, pp. 641–646, 2020.

[23] C. Li, D. Belkin, Y. Li, P. Yana, M. Hu, N. Ge, H. Jiang, E. Montgomery, P. Lin, Z. Wang, W. Song, J. P. Strachan, M. Barnell, Q. Wu, R. S. Williams, J. J. Yang, and Q. Xia, "Efficient and self-adaptive in-situ learning in multilayer memristor neural networks," *Nature Communications*, vol. 9, pp. 1–8, 2018.

[24] J. Park, M. Kwak, K. Moon, J. Woo, D. Lee, and H. Hwang, "TiOx-based RRAM synapse with 64-levels of conductance and symmetric conductance change by adopting a hybrid pulse scheme for neuromorphic computing," *IEEE Electron Device Lett.*, vol. 37, no. 12, pp. 1559–1562, 2016.

[25] Gurobi Optimization, Inc., "Gurobi optimizer reference manual," 2013. [Online]. Available: http://www.gurobi.com

[26] L. Pillage, *Electronic circuit and system simulation methods*. McGraw-Hill, 1999.

[27] A. Chen, "A comprehensive crossbar array model with solutions for line resistance and nonlinear device characteristics," *IEEE Trans. Electron Devices*, vol. 60, no. 4, pp. 1318–1326, 2013.

[28] L. Xia, M. Liu, X. Ning, K. Chakrabarty, and Y. Wang, "Fault-tolerant training with on-line fault detection for rram-based neural computing systems," in *Proc. Design Autom. Conf.*, 2017, pp. 33:1–6.

[29] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *Symp. Oper. Syst. Des. & Impl.*, vol. 16, 2016, pp. 265–283.

[30] Z. Jiang, Y. Wu, S. Yu, Y. Lin, K. Song, Z. Karim, and W. H.-S. Philip, "A compact model for metal-oxide resistive random access memory with experiment verification," *IEEE Trans. Electron Devices*, vol. 63, no. 5, pp. 1884–1892, 2016.

[31] S. Yu, B. Gao, Z. Fang, H. Yu, J. Kang, and H.-S. P. Wong, "A neuromorphic visual system using RRAM synaptic devices with sub-pj energy and tolerance to variability: Experimental characterization and large-scale modeling," in *Proc. Int. Electron Dev. Meeting*, 2012, pp. 10.4.1 – 10.4.4.