IEEE Access
Multidisciplinary ⁝ Rapid Review ⁝ Open Access Journal

Received 17 July 2024, accepted 26 August 2024, date of publication 2 September 2024, date of current version 10 September 2024.

*Digital Object Identifier 10.1109/ACCESS.2024.3452762*

RESEARCH ARTICLE

# A Genetic Algorithm Accelerator Based on Memristive Crossbar Array for Massively Parallel Computation

**MOHAMMADHADI BAGHBANMANESH[1] AND BAI-SUN KONG[1,2], (Member, IEEE)**
[1]Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon 16419, South Korea
[2]Department of Artificial Intelligence, Sungkyunkwan University, Suwon 16419, South Korea

Corresponding author: Bai-Sun Kong (bskong@skku.edu)

This work was supported in part by NRF under Grant RS-2024-00354034 and Grant 2020M3H2A1076786, in part by IITP under Grant RS-2024-00399394 and Grant N. 2019-0-00421, in part by KIAT under Grant P0023704 and Grant P0012451. EDA tools were supported by the IC Design Education Center (IDEC).

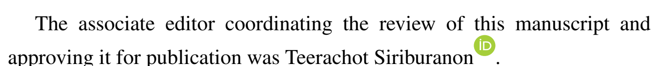
**ABSTRACT** Genetic algorithm (GA) has been extensively used for solving complex problems. Due to a high computational burden of finding solutions using GA, acceleration with hardware support has been a choice. In this paper, a GA accelerator based on the processing-in-memory (PIM) methodology to address the computational issue of GA is proposed. The proposed GA accelerator has a memristive crossbar array that can support parallelism with memory and computation combined. For letting the crossover operation for GA exploit massive parallelism provided by the array, a novel crossover scheme called aligned hybrid crossover is proposed, in which multiple multi-point crossovers coexist whose crossover bit positions are aligned. By using the memristive array, the mutation operation can also be done simultaneously for all required chromosome bits. Moreover, the fitness for weighted-sum computation-based *0-1 knapsack* and *subset-sum* problems is shown to be evaluated in full parallel for the entire chromosomes in a population. The effects of memristance variation in the array on the fitness evaluation and the read margin are investigated. According to performance evaluation, the proposed GA accelerator having a $64 \times 64$ memristive crossbar array is found to reduce the clock cycles significantly for performing operations like crossover, mutation, selection, and fitness evaluation. Specifically, for executing the generational GA with a chromosome population size of 64 with each chromosome having 64 bits, the total number of clock cycles required per generation is at least 10 times reduced as compared to conventional designs.

**INDEX TERMS** Genetic algorithm, crossbar array, memristor, processing-in-memory.

## I. INTRODUCTION

Genetic algorithm (GA) [1] is an algorithm for stochastic optimization that originates from the natural evolution theory of animals. Since Holland first proposed [2], the GA has been used successfully to address a wide range of problems. For example, GA is used for feature selection in machine learning [3], hybrid deep learning modeling [4], the optimization of the multi-channel convolutional neural network [5], and other problems having lots of parameters. GA has been considered a versatile search algorithm that can be used at the cost of runtime to simultaneously investigate several regions of a solution space.

Implementation of the GA in hardware is a good approach to speed up the process. One of the obvious issues in this algorithm is to fast find the global optimum guided by a fitness function, which can be running on hardware such as the von Neumann machine. Many hardware-specific designs for GA have been developed [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25] to address the issue. Main bottlenecks in a GA hardware implementation are computation and communication.


The associate editor coordinating the review of this manuscript and approving it for publication was Teerachot Siriburanon.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

They can happen in both the crossover and mutation operations. In the crossover operation, many bits of chromosomes need to be transferred from memory, shuffled by a processor, and written back into the memory. For a mutation operation, some bits of each chromosome must be read from the memory and flipped by the processor, and the resulting bits must be written into the memory. In a conventional architecture based on a von Neumann machine, these procedures can be performed per each chromosome requiring many clock cycles. Moving data between processor and memory can require many clock cycles and much energy consumption. Because the processor and memory are separate from each other and data must be frequently accessed, the throughput can be limited due to insufficient data transfer rate between them. A memristor is a nonlinear two-terminal device referring to the relationship between electric charge and magnetic flux, which was postulated and named by Chua in 1971 [26]. Memristors can be configured as a crossbar array for in-memory computation with massive parallelism. Memristive crossbar arrays are usually considered to be computation- and energy-efficient, because they can perform both storage and computation in-situ, potentially eliminating the need for data transfer between memory and processor.

In this paper, a memristive crossbar array-based hardware accelerator for providing massively parallel processing of GA is presented. The contributions of this paper are summarized as follows:

1) We pursue removing the distance between the processor and memory by using a memristive crossbar array for processing GA performing crossover, mutation, and fitness evaluation. We reduce the execution time for GA processing in our processing-in-memory (PIM) methodology, which can be implemented with simple memristive circuits.

2) To fully exploit the inherent massive parallelism of the array, crossover operations are done in fully parallel for all chromosome bits in a segment using a new approach called aligned hybrid crossover. Mutation operations in each generation are also done in parallel only in two clock cycles.

3) An efficient way of evaluating fitness for weighted-sum computation-based *0-1 knapsack* and *subset-sum* problems using our crossbar array is also proposed, allowing it to be completed in one or two cycles.

4) The effects of memristance variation in the array on the fitness evaluation and the read margin are also investigated.

5) For executing the generational GA with a chromosome population size of 64 with each chromosome having 64 bits using the proposed crossbar array-based hardware accelerator, the total number of clock cycles required per generation is substantially reduced as compared to conventional designs.

The remaining sections are organized as follows. Section II introduces basic concepts of the GA algorithm and previous works on hardware implementation. In Section III, the architecture and operation of the proposed GA accelerator based on a memristive crossbar array are presented. Section IV presents the simulation results of our design using a well-known HP memristor model and describes the comparison results with conventional designs. The conclusions are given in Section V.

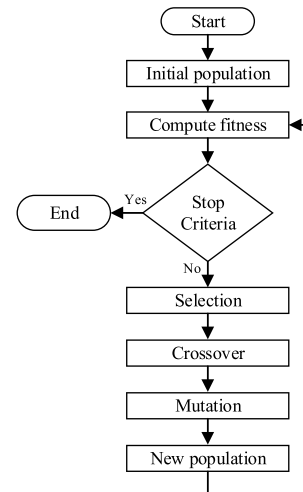



**FIGURE 1.** **Flowchart of genetic algorithm.**

## II. BACKGROUND AND OVERVIEW

### A. GENETIC ALGORITHM

Figure 1 shows a flowchart representing the execution procedure of a genetic algorithm. The initial population is first prepared, and the fitness of each chromosome is calculated. After criteria are examined, the selection operator picks the best chromosomes as parents. Then, to generate a new population, child chromosomes are produced by crossover and mutation. The crossover combines the genetic information of parents by concatenating partial bit strings of both parents. The mutation selects chromosomes randomly and flips some bits in a predefined probability. The worst part of the population is replaced by newly created offspring in each generation. The termination criterion might be the maximum number of generations or a satisfactory fitness value. In genetic algorithms, two population control methods are popular: generational and steady-state [27]. In a generational GA, the entire population gets replaced each generation. In a steady-state GA, only a few individuals are replaced. In our architecture, the generational GA has been used. Compared to the steady-state GA, the generational GA has an advantage of reducing the number of generations required to find a solution. On the other hand, it can have a disadvantage of taking longer time and consuming more energy to prepare child chromosomes. In our design, these long time and large energy issues are addressed by using a memristive crossbar array having massive parallelism.

### B. MEMRISTIVE CROSSBAR ARRAY

A variety of applications can be found for use of memristive crossbar array architecture. For instance, the array architecture can be used for computations related to the feature selection in image processing [28], multiply-and-accumulation [29], artificial intelligence and neuromorphic computing [30], and many others. It is possible to construct

memristive interconnections in a nano-scaled space for configuring a large-scale crossbar array, which can combine the memory power of memristors and the parallel processing of the array. The electrical resistance of a memristor is not constant but depends on the history of the current that has been flowing through the device. After the titanium dioxide ($TiO_2$) memristor device was developed by HP in 2008 [31], the models are being improved by the invention of various device fabrication technologies. The array composed of these memristors can be considered a promising solution for next-generation computing systems because of their non-volatility, fast programming, low power consumption, nano-scale device dimension, and high on/off ratio.

### C. RELATED WORKS

Since the amount of computations for performing GA is huge as mentioned earlier, hardware-specific designs can be used for speeding up the process. During the last decades, hardware architectures for accelerating these operations on FPGA [6], [8], [11], [15], [16], [17], [18], [21], [22] and ASIC [7], [9], [10], [12], [13], [14], [19], [20] have been introduced. Recently, architectures utilizing memristive arrays [23], [24], [25] have also been proposed. ASIC, FPGA, and memristive implementations for this purpose have benefits and drawbacks. ASIC implementations are more efficient in terms of power consumption because they can use fewer chip resources and occupy less silicon area but have less flexibility in terms of customizing parameters. FPGA implementations are more flexible to customize GA parameters but use more resources and consume more power. Memristive implementations can take advantage of inherent massive parallelism in a crossbar array, speeding up computation in an energy-efficient way. One demerit of this approach is a limited selection of memristive devices. Since many new devices are being developed recently, faster and more energy-efficient GA accelerations are in vision.

The first GA implementation on the hardware used Xilinx FPGA with a modular design approach by using behavioral VHDL [6]. In the implementation, the chromosome bit length and population size were limited to 5 and 16, respectively, making it unsuitable for solving real-world problems. Another GA implantation used pipelining to improve the performance on six FPGA chips for performing selection, crossover, mutation, and fitness evaluation [11]. The population in each generation was generated by the steady-state GA, and all modules were described by VHDL. In this implementation, when the chromosome bit length increases, the number of clock cycles required for performing the crossover operation increases linearly, causing unsuitability for handling GAs having chromosomes with a large number of bits. In [15], users can generate a GA IP core for an Altera FPGA chip by special software called SmartGA. It is flexible to change parameter values such as chromosome bit length and population size. For implementing a fitness function, it uses a lookup table but consumes a lot of resources and chip area that can cause much power consumption.

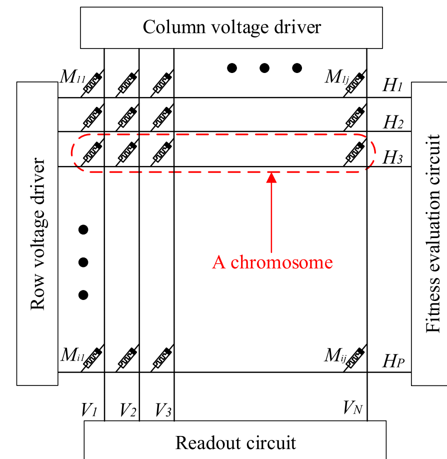



**FIGURE 2.** GA accelerator with memristive crossbar array.

A VLSI hardware design was proposed in [9], which uses a multi-processor technique for executing GA operations in parallel and distributed modes. However, due to the communication overhead between processors, convergence takes a long time. The design in [17] provided a GA IP core that allowed for more flexibility in setting parameters. However, because the fitness function module is external and requires communication between the GA core and the fitness module, the number of clock cycles required in each generation is large. Reference [19] designed a chip having chromosomes whose bit length is 32 and used steady-state GA in a foundry CMOS technology. The system is also capable of connecting chips in a chain to expand the chromosome bit length, which is called genetic algorithm processors (GAPs). In this architecture, when the chromosome bit length is greater than 32, the communication between GAPs is required, resulting in a considerable increase on the number of clock cycle. Recently, a few research results handling computations related to GA using memristive arrays have been published [23], [24], [25]. In [23], a memristive network was used for detecting the edge of an image. The genetic algorithm was utilized to iteratively detect edges, in which memristors with a low resistance value were considered as edge pixels in the image. In [24], a crossbar array with memristors having analog multi-level conductance was used for an image classification task. The research investigated the effect of the failure rate of memristors in the crossbar array by comparing two training algorithms (GA and Local Update). The design in [25] performed GA operations including selection, crossover, and mutation by using matrix calculations based on a matrix-friendly genetic algorithm (MGA). The matrix calculations were performed in a set of memristive vectors and arrays in a pipelined manner to process the genetic algorithm. No fitness evaluation was supported in this method.

## III. GA ACCELERATOR BASED ON MEMRISTIVE CROSSBAR ARRAY

### A. OVERALL ARCHITECTURE

Figure 2 shows the overall architecture of the proposed GA accelerator consisting of the memristive crossbar array and

peripheral circuits including row and column voltage drivers, a readout circuit, and a fitness evaluation circuit. The crossbar array is composed of memristors at the cross points of horizontal and vertical lines. Memristive devices at cross points establish a required connectivity. In the array, each row corresponds to a chromosome and each column corresponds to a collection of bits at the same position in all chromosomes. Respective rows are labeled as horizontal lines ($H_1 \sim H_P$) and respective columns as vertical lines ($V_1 \sim V_N$), where subscripts $P$ and $N$ indicate the population size and chromosome bit length, respectively. Each memristor in the array is indexed by $M_{ij}$, where $i$ and $j$ denote the $i^{th}$ horizontal and $j^{th}$ vertical lines, respectively. For GA operations, the row and column voltage drivers generate and drive required voltage levels like $V_W$, $V_R$, or $GND$ to row and column lines, respectively. The fitness values of chromosomes are evaluated, and the best chromosomes are selected by the fitness evaluation circuit. If required, chromosome bits can be read out from the array by the readout circuit. Table 1 summarizes a list of parameters defined for the accelerator.

Figure 3 shows the schematic symbol of a memristor used in our array, in which the black square represents the positive terminal. As shown in the left drawing in Figure 3 when the programming voltage ($V_W$) is applied to the positive terminal, the resistance switches to $R_{ON}$ (*SET* operation). When $V_W$ is applied to the negative terminal, the resistance switches to $R_{OFF}$ (*RESET* operation), as seen by the drawing in the middle. To read a memristive resistance, the read voltage ($V_R$) can be applied to either the positive or negative terminal with the other terminal connected to GND through a resistor ($R_S$), by which the memristive current can be converted into a voltage. When the memristor to read is in the $R_{OFF}$ state the voltage across $R_S$ will be very small, otherwise, it will be quite large. Using the $R_S$ is a simple method for reading the state of the memristor. There are a variety of methods for reading the memristor state. In our case, the virtual ground method is used to reduce the amount of sneak path current, which will be elaborated more in Section IV. In our implementation, the $R_{OFF}$ state is considered as logic '0' and the $R_{ON}$ state as logic '1'.

**TABLE 1.** Parameters for memristive crossbar array-based GA accelerator.

| Parameter | DESCRIPTION | Parameter | Description |
|---|---|---|---|
| $H_i$ | $i^{th}$ horizontal line | *Pr1* | Chromosome having *Parent1* from previous generation |
| $V_j$ | $j^{th}$ vertical line | *Pr2* | Chromosome having *Parent2* from previous generation |
| $M_{ij}$ | Memristor at $i^{th}$ horizontal and $j^{th}$ vertical lines | *Chi* | Chromosome having $i^{th}$ child |
| $N$ | Chromosome bit length | $FV_i$ | Fitness value of $i^{th}$ chromosome |
| $P$ | Population size | *MU* | Mutation operation |
| $N_{CK}$ | Number of clock cycles | *CR* | Crossover operation |
| $N_{CH}$ | Number of children | *SE* | Selection operation |
| $N_{CP}$ | Number of cut points | - | - |

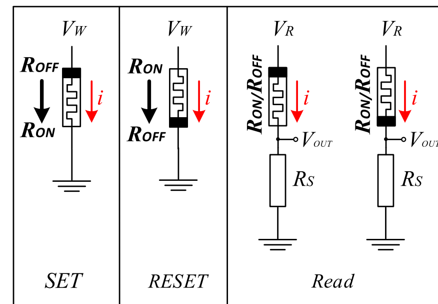



**FIGURE 3.** Schematic symbol of the memristor.

## B. PARALLEL ALIGNED HYBRID CROSSOVER

In our GA accelerator, the crossover operation is done in parallel in the memristive crossbar array. In general, a single- or multi-point crossover scheme can be utilized, in which single or multiple crossover points are used, respectively. Since these conventional crossover schemes cannot allow enough parallelism in our array, a novel crossover scheme called *aligned hybrid crossover* is used in our design. Figure 4 shows the procedure for generating six child chromosomes from two parent chromosomes based on our crossover scheme. In the figure, each row represents a chromosome, whose bit values indicate the memristive states. The first two rows (*Pr1* and *Pr2*) are the chromosomes to store parents (*Parent1* and *Parent2*) selected as the best in the previous generation. The other six rows (*Ch1*∼*Ch6*) are the chromosomes to store new children to be generated. *Parent1* (*Parent2*) bit values to be written into *Pr1* (*Pr2*) are shown in the dotted box above the first row in Figure 4(a) (Figure 4(b)). Before starting the crossover operation, all chromosome bits are assumed to be at $R_{OFF}$. Crossover points are randomly selected at two different bit positions, as seen in Figure 4(a). Then, in the first clock cycle (cycle #1), logic '1' bits in the first segment (left eight bits) of *Parent1* are copied into *Pr1,* and a selected half of child chromosomes (*Ch1*, *Ch3*, and *Ch5*) simultaneously. In the second clock cycle (cycle #2), logic '1' bits in the second segment (middle ten bits) are copied into *Pr1* and another selected half of the child chromosomes (*Ch2*, *Ch3*, and *Ch6*). Similarly, in the third clock cycle (cycle #3), the third segment (right twelve bits) is copied into corresponding chromosomes in the array. Then, all the bits in *Parent1* are copied into *Pr1* and into the required bit positions of *Ch1*∼*Ch6* in just three clock cycles, whose resulting memristive states are indicated by red-colored regions in Figure 4(a). Figure 4(b) shows the memristive states (indicated by green-colored regions) after the next three cycles (clock #4∼clock #6) to copy *Parent2* into *Pr2* and *Ch1*∼*Ch6*.

From the description above, it can be recognized that some chromosomes are generated by the single-point crossover

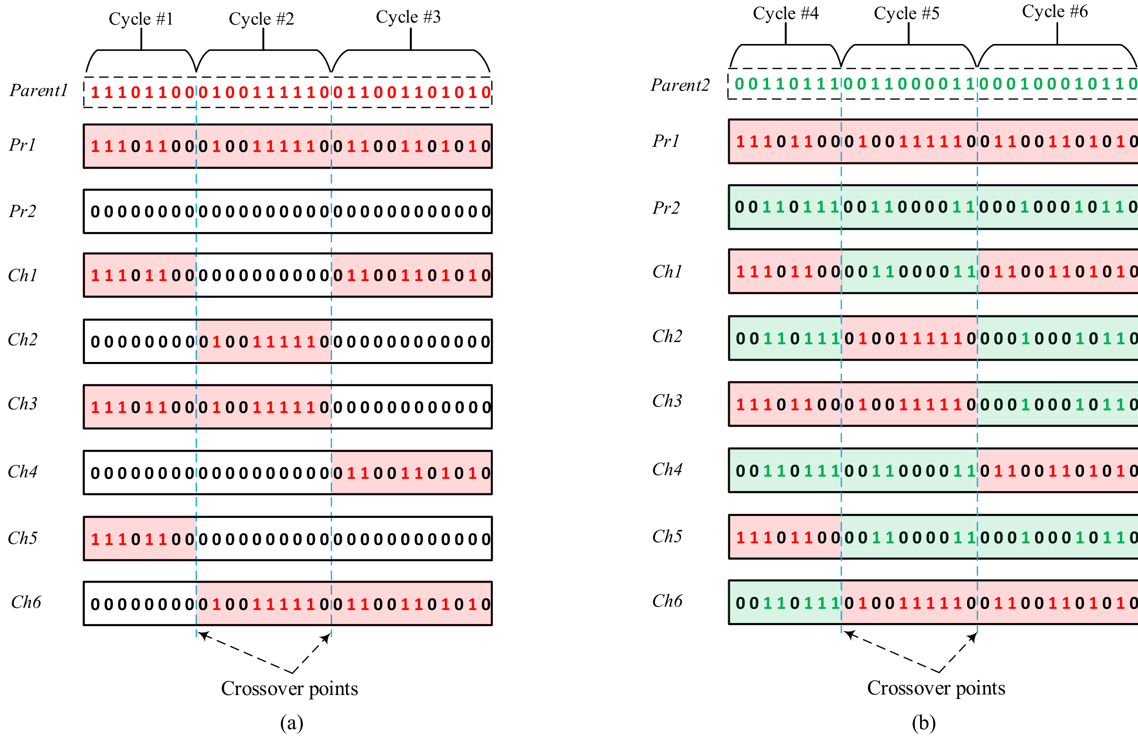



**FIGURE 4. Conceptual procedure for array-based crossover operation: (a) copying *parent1* into array, (b) copying *parent2* into array.**

while others are generated by the two-point crossover. Specifically, as shown in Figure 4(b) depicting the final chromosome bit patterns, child chromosomes *Ch3* ∼ *Ch6* are obtained by the single-point crossover at one of the crossover points, whereas *Ch1* and *Ch2* are generated by the two-point crossover at two different crossover points. In a situation where we have more than three crossover-points, three- or four-point crossover can also be possible. Hence, our crossover scheme is a combination of the single- and multi-point crossovers for the given crossover bit positions. That's why the proposed crossover scheme is *hybrid*, implying that multiple multi-point crossovers coexist. On top of this, note that all crossover points in Figure 4 are *aligned* among chromosomes. Specifically, the crossover point of child chromosomes *Ch3* and *Ch4* generated by the single-point crossover is aligned. The crossover point of *Ch5* ∼ *Ch6* is also aligned at a different position from that of *Ch3* and *Ch4*. Moreover, *Ch1* and *Ch2* generated by the two-point crossover again have their crossover points aligned to each other and to those in the single-point crossover. Therefore, the proposed crossover scheme supports various multi-point crossovers with its crossover bit positions aligned, thereby the *aligned hybridcrossover*. By so doing, the total number of clock cycles for completing the crossover operation can be reduced by exploiting more parallelism provided by the array. When the number of chromosomes increases, the number of crossover points will also increase, which means that the proposed crossover scheme will be a combination of many crossover schemes having single-, two-, and three-point, etc.

The number of crossover points ($N_{CP}$) required in this scheme for a given population size ($P$) can be written as

$$\boldsymbol{N_{CP} \geq log_2 (P) - 1} \tag{1}$$

$$N_{CK} = (N_{CP} + 1) \times 2 \tag{2}$$

The minimum number of crossover points for a given population size can be chosen by the condition in (1). The number of clock cycles ($N_{CK}$) required to complete all crossover operations in a generation for a given number of crossover points can be found using (2). It is important to note that the number of clock cycles required does not increase as the number of bits in a chromosome increases since all the memristors in a segment update their states simultaneously. Note that this interesting feature originates from a combination of the proposed aligned hybrid crossover and an inherent parallelism provided by the crossbar array.

Figure 5 depicts the memristive crossbar array and peripheral circuits to explain the crossover operation in more detail. To generate four children from two parents, two equidistant crossover points is used for simplicity in this example (in Figure 4 the crossover points were chosen at random bit positions). The timing diagrams on the top and left show what voltage levels are to be driven to the vertical and horizontal lines in each clock cycle, respectively. As shown in the upper left corner in Figure 5(a), *Parent1* (6 bits) stored in

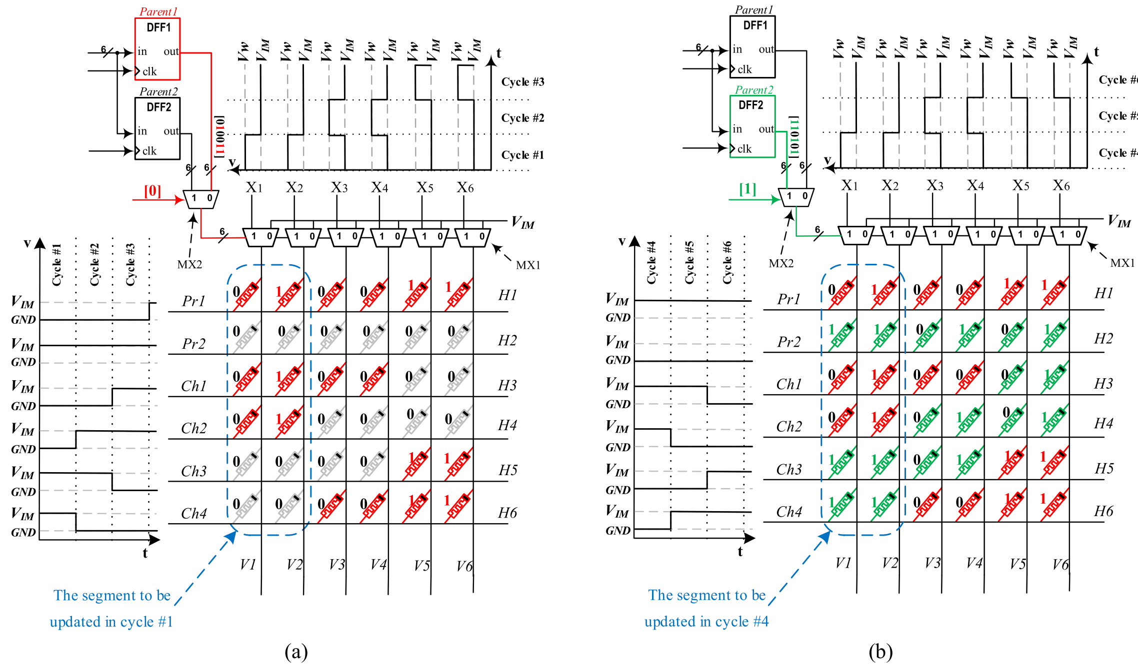



**FIGURE 5.** Hardware structure and operation for crossover: (a) copy *Parent1* into array chromosomes, and (b) copy *Parent2* into array chromosome.

*DFF1* is selected by *MX2*. (Relevant signals and circuits are highlighted in red). A group of multiplexers (*MX1*) driving the vertical lines from the top, whose selection signals come from *MX2*, choose *Xi* for logic '1' bits in *Parent1* and $V_{IM}$ for logic '0' bits as the line voltages. Intermediate voltage $V_{IM}$ is used to avoid unintended updates of neighboring memristors, whose voltage level can be appropriately chosen for minimizing the current through half-selected memristors. The timing diagram at the top depicts cycle-based voltage levels of *Xi* to be used as vertical line voltages. With this configuration, in the first clock cycle (cycle #1), left two bits of *Parent1* are to be copied into the same positions of *Pr1* and two selected child chromosomes (*Ch1* and *Ch2*). For this, $X_1/X_2$ and $X_3/X_4/X_5/X_6$ (inputs to mux *MX1*) are driven with $V_W$ and $V_{IM}$, respectively. Then, considering the left two bits of *Parent1* are '01' (see the bit pattern at the output of *DFF1*), only the second vertical line ($V_2$) is driven with $V_W$, and all other vertical lines are driven with $V_{IM}$. At the same time, horizontal lines, $H_1$, $H_3$, and $H_4$ are driven to *GND* whereas $H_2$, $H_5$, and $H_6$ are driven to $V_{IM}$, as seen by the timing diagram on the left. Then, only the second bit of *Parent1* is copied into the same bit positions of *Pr1*, *Ch1*, and *Ch2*, letting memristors $M_{12}$, $M_{32}$, and $M_{42}$ change their states to $R_{ON}$. In the second clock cycle (cycle #2), only $X_3$ and $X_4$ are driven to $V_W$, and $H_1$, $H_3$, and $H_6$ are driven to *GND* to copy the middle two bits in *Parent1* into the corresponding columns of *Pr1*, *Ch1*, and *Ch4*. In this case, considering that the corresponding *Parent1* bits are '00', no memristors will change their states. In the third clock cycle (cycle #3), by the same procedure, memristors $M_{15}$, $M_{16}$, $M_{55}$, $M_{56}$, $M_{65}$, and $M_{66}$ change their states to $R_{ON}$. In Figure 5(a), the resulting memristor bit values copied from *Parent1* are highlighted in red. Figure 5(b) shows the procedure for copying *Parent2* into *Pr2* and the corresponding bit positions in child chromosomes during the next three clock cycles (cycles #4 to #6), whose results in the array are highlighted green. In summary, all chromosomes in a segment in the array update their bit values simultaneously in a single clock cycle, resulting in the crossover operation completed only in six clock cycles in this example. Table 2 shows a list of cycle-based control voltages to the horizontal and vertical lines to prepare six chromosomes as a new population.

**TABLE 2.** Cycle-based control voltages for horizontal and vertical lines to generate a new population by crossover.

| Cycle | Operation | Horizontal line voltage | Vertical line voltage |
|---|---|---|---|
| 1 | *Copy logic '1' bits of Parent1 (1st part) into the array* | *$H_1$,$H_3$,$H_4$: GND; $H_2$,$H_5$,$H_6$: $V_{IM}$;* | *$V_2$:$V_W$; $V_1$,$V_3$~$V_6$: $V_{IM}$;* |
| 2 | *Copy logic '1' bits of Parent1 (2nd part) into the array* | *$H_1$,$H_3$,$H_6$: GND; $H_2$,$H_4$,$H_5$: $V_{IM}$;* | *$V_1$~$V_6$: $V_{IM}$;* |
| 3 | *Copy logic '1' bits of Parent1 (3rd part) into the array* | *$H_1$,$H_5$,$H_6$: GND; $H_2$,$H_3$,$H_4$: $V_{IM}$;* | *$V_5$,$V_6$: $V_W$; $V_1$~$V_4$: $V_{IM}$;* |
| 4 | *Copy logic '1' bits of Parent2 (1st part) into the array* | *$H_2$,$H_5$,$H_6$: GND; $H_1$,$H_3$,$H_4$: $V_{IM}$;* | *$V_1$,$V_2$: $V_W$; $V_3$~$V_6$: $V_{IM}$;* |
| 5 | *Copy logic '1' bits of Parent2 (2nd part) into the array* | *$H_2$,$H_4$,$H_5$: GND; $H_1$,$H_3$,$H_6$: $V_{IM}$;* | *$V_4$: $V_W$; $V_1$,$V_2$, $V_3$,$V_5$,$V_6$: $V_{IM}$;* |
| 6 | *Copy logic '1' bits of Parent2 (3rd part) into the array* | *$H_2$,$H_3$,$H_4$: GND; $H_1$,$H_5$,$H_6$: $V_{IM}$;* | *$V_6$: $V_W$; $V_1$~$V_5$: $V_{IM}$;* |

## *C. PARALLEL MULTI-BIT MUTATION*

As mentioned earlier, the mutation operation allows some randomly selected chromosome bits to be flipped in each

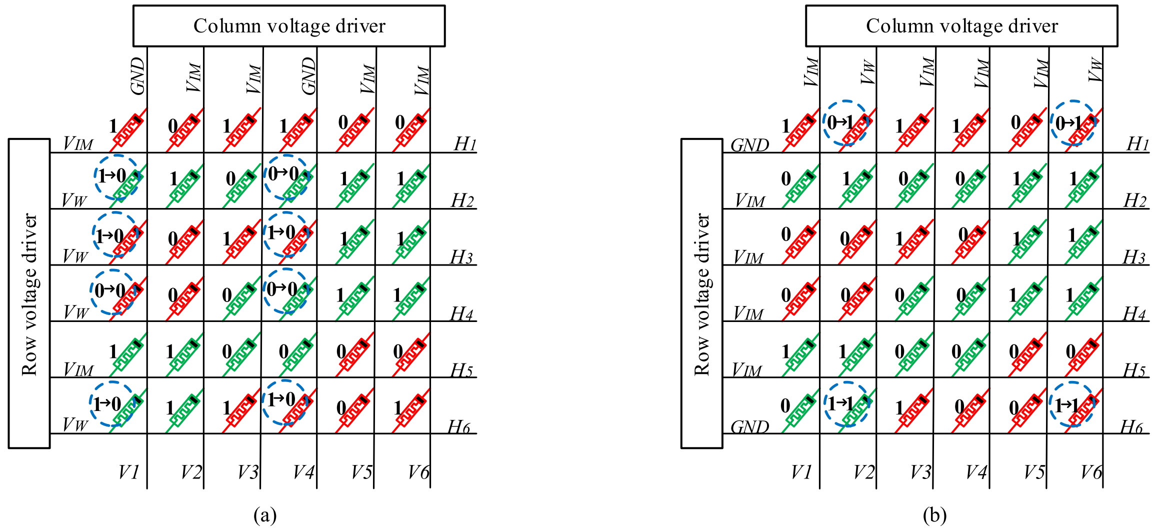



**FIGURE 6.** Parallel multi-bit mutation: (a) resetting, and (b) setting chromosome bits.

generation to increase population diversity. In general, many clock cycles are required for performing mutation operations to a group of chromosomes stored in memory because chromosome bits to be mutated must be read to identify their state values, and the flipped values should be written back into the chromosomes in memory. In our design, the timing overhead for doing these operations can be avoided by letting mutation operations be done on site where chromosomes are stored in a full-parallel manner. Figure 6 shows an example of how it can be done in our memristive array. In the first clock cycle (Figure 6(a)), some horizontal ($H_2$, $H_3$, $H_4$, and $H_6$) and vertical ($V_1$ and $V_4$) lines randomly selected according to a given mutation rate are driven to $V_W$ and $GND$, respectively. To avoid unintended state changes for other memristors and to minimize the sneak current, all unselected horizontal and vertical lines are driven to $V_{IM}$. Then, the states of the selected memristors will stay at or switch to $R_{OFF}$ (RESET operation). As shown in Figure 6(a), eight memristors in dotted blue circles are selected for mutation. Memristors $M_{21}$, $M_{31}$, $M_{34}$, $M_{61}$, and $M_{64}$ are flipped to $R_{OFF}$ since they were in $R_{ON}$, and memristors $M_{24}$, $M_{41}$, and $M_{44}$ are not flipped since they were already in $R_{OFF}$. In the second clock cycle (Figure 6(b)), horizontal ($H_1$ and $H_6$) and vertical ($V_2$ and $V_6$) lines are selected and driven to $GND$ and $V_W$, respectively. As above, all unselected horizontal and vertical lines are again set to $V_{IM}$. Then, memristors $M_{12}$ and $M_{16}$ change their states to $R_{ON}$ (SET operation) with the state of memristors $M_{62}$ and $M_{66}$ unchanged. In summary, by letting some selected chromosome bits be reset to $R_{OFF}$ in the first clock cycle, and some other selected chromosome bits to be set to $R_{ON}$ in the second clock cycle, the whole mutation operations in a generation can be done in just two clock cycles. Considering that not all selected memristors are flipped in each clock cycle, it would be better to use a higher mutation rate than in an ordinary case. Table 3 explains cycle-based control voltages driven to horizontal and vertical lines for mutation in a generation.

**TABLE 3.** Control voltages to perform mutation operation.

| Cycle | Operation | Horizontal line voltage | Vertical line voltage |
|---|---|---|---|
| 1 | *RESET* | *Hx*: $V_W$; (*x*: randomly selected horizontal lines. Remaining lines are driven with $V_{IM}$) | *Vy*: *GND*; (*y*: randomly selected vertical lines. Remaining lines are driven with $V_{IM}$) |
| 2 | *SET* | *Hx*: *GND*; (*x*: randomly selected horizontal lines. Remaining lines are driven with $V_{IM}$) | *Vy*: $V_W$; (*y*: randomly selected vertical lines. Remaining lines are driven with $V_{IM}$) |

### D. PARALLEL FITNESS EVALUATION AND SELECTION

As mentioned earlier, the fitness evaluation is a computationally intensive function in a genetic algorithm. To show the computational capability of our memristive crossbar array, two well-known weighted sum computation-based problems (*0-1 knapsack* and *subset-sum* problems [32]) are employed as fitness functions. The *0-1 knapsack* problem is a combinatorial (discrete) optimization, in which the sum of the values (*Vi*'s) of items in a knapsack is maximized without letting the sum of the weights (*Wi*'s) for the items exceed the capacity ($C$). The *0-1 knapsack* problem can be described as

$$\begin{aligned} &Maximize \sum_{i=1}^{N} Vi \times Xi; \\ &Subject\ to \sum_{i=1}^{N} Wi \times Xi \leq C; \\ &Xi \in \{0, 1\}; i = 1, \ldots, N \end{aligned} \tag{3}$$

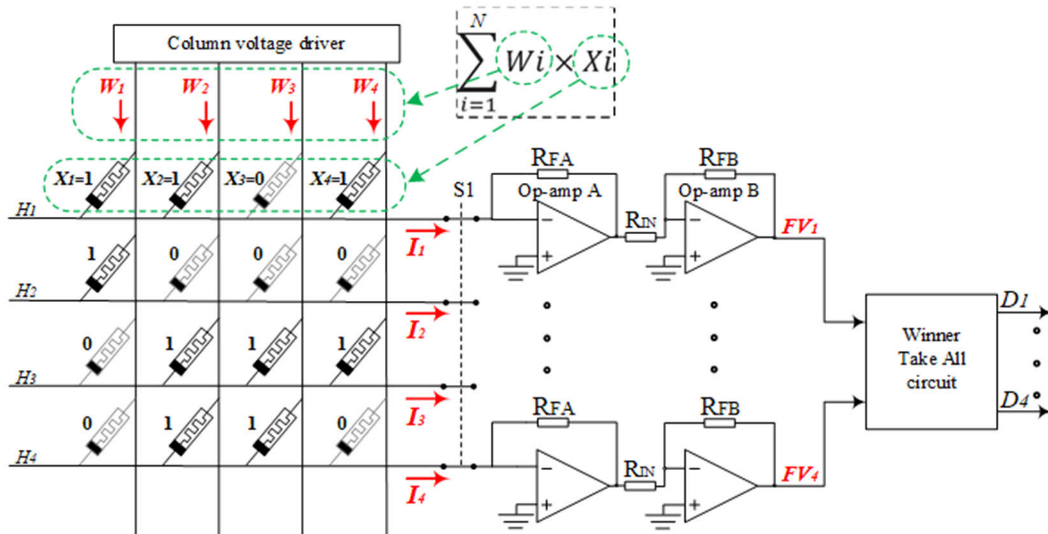



**FIGURE 7.** Memristive array-based fitness evaluation and parent selection.

where $N$ is the number of single-bit items whose $i^{th}$ item is indexed by $Xi$. $Vi$ and $Wi$ are the value and weight of $Xi$, respectively. When $Xi$=1, the $i^{th}$ item is in the knapsack. The *0-1 knapsack* problem is to find the best way to pack a knapsack with a subset of a given collection of items. The best packing then is one in which the weight-sum of the items in the knapsack is less than or equal to a given capacity and the value-sum of the items in the knapsack is maximized. One variation of this optimization is the *subset-sum* problem, in which the weight equals the value [32]. The expressions for the *subset-sum* problem can be written as

$$Max\left(\sum_{i=1}^{N} Wi \times Xi\right) \le C;$$
$$Xi \in \{0, 1\}; i = 1, \ldots, N \quad (4)$$

Given $N$ single-bit items ($Xi$*'s*) with positive integer weights ($Wi$*'s*) and a capacity of $C$, the problem finds a subset of items whose total weight can be maximized without exceeding the capacity. Both problems are NP-complete in combinatorial optimization, and finding the optimum value through an exhaustive search will take exponential time ($O(k^n)$), where $k$ is constant and $n$ is the input size of the problem.

Eq's. (3) and (4) above indicate that, for calculating the fitness for the *0-1 knapsack* and *subset-sum* problems, a lot of multiplication and addition operations are required to obtain products between items and weights and the sum of these products, respectively. In our design, evaluating the fitness values of all chromosomes and selecting the best two chromosomes among them can be done in parallel in two or three clock cycles. Detailed procedures for doing these operations are explained below.

### 1) *SUBSET-SUM* FITNESS EVALUATION AND SELECTION

Figure 7 shows how the fitness values for the *subset-sum* problem can be evaluated in the crossbar array. Four memristors in each chromosome are considered as four items ($Xi$'s), where the memristive state $R_{ON}$ is regarded as the corresponding item in the knapsack ($Xi=1$). The array has a set of op-amps as the peripheral circuit. Horizontal lines ($H_1 \sim H_4$) behave as virtual grounds since through switches *S1* they are connected to the inverting inputs to op-amps (op-amps A) having negative feedback. In the first cycle, the vertical lines are driven with weight voltages ($Wi$'s) by the column voltage driver at the top. Each horizontal line then has a current proportional to the weighted sum of vertical line voltages (as weights) and memristor states (as items in the knapsack). Assuming negligible currents through memristors having the $R_{OFF}$ state ($Xi$=0), only the memristors in the $R_{ON}$ state ($Xi$=1) are considered in the summation. The weight-sum current of each chromosome on its horizontal line and the resulting fitness voltage at the output of each op-amp (*op-amp B*) can be written as

$$I_j = \sum_{i=1}^{4} W_i \times X_i \quad (5)$$

$$FV_j = \left(-R_{FA} \times I_j\right) \times \left(-\frac{R_{FB}}{R_{IN}}\right) \quad (6)$$

where $i$ is chromosome bit index, and $j$ is chromosome index. Note that calculating the fitness this way has a time complexity of *O(1)*. Then, as the selection operation the index of the first-best chromosome (*Pr1*) is identified using the winner-take-all (WTA) circuit [33] after the check whether $FV_j$ voltages exceed the capacity. That is, one among four WTA outputs (*S1*~*S4*) is activated, which is associated with the best chromosome. In the second cycle, the index of the second-best chromosome (*Pr2*) is detected similarly.

### 2) *0-1 KNAPSACK* FITNESS EVALUATION AND SELECTION

By the comparison between (3) and (4), the difference between the *subset-sum* and *0-1 knapsack* problems is that

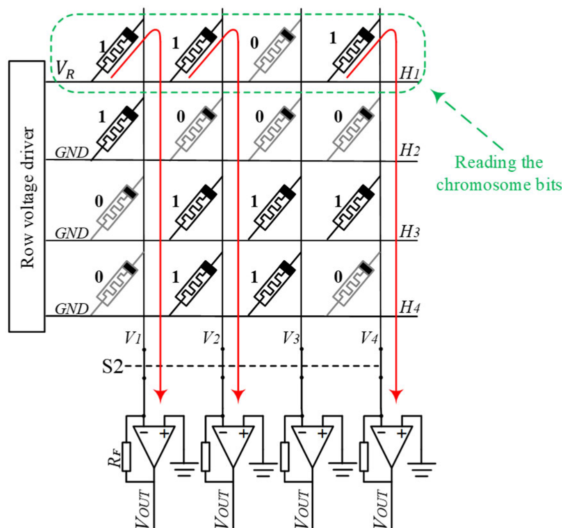



**FIGURE 8.** Reading the selected chromosome.

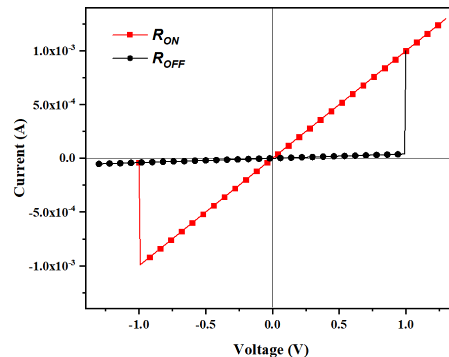



**FIGURE 9.** The I-V curve of the HP memristive device [31].

the former uses one parameter (*weight*) whereas the latter uses two parameters (*weight* and *value*). To handle this difference and find the fitness value for the *0-1 knapsack* problem, an additional clock cycle is used as explained below. The operation in the first clock cycle is almost the same as that of the *subset-sum* problem. The voltages corresponding to the weights are provided as the vertical line voltages, and the weighted-sum voltages are evaluated at the outputs of the op-amps as before. The difference from the operation of the *subset-sum* problem in the first clock cycle is that the WTA operation is not done because the weighted-sum voltages are not the fitness as seen in (4). Whether these voltages exceed the capacity is only checked and saved. In the second cycle, the fitness values to be used by the WTA circuit are evaluated. The difference of the second cycle from the first is that the voltages corresponding to the values (not the weights) of items are driven to the vertical lines. The resulting value-sum current on each horizontal line and the fitness voltage can be written as

$$I_j = \sum_{i=1}^{4} V_i \times X_i \tag{7}$$

$$FV_j = \left(-R_{FA} \times I_j\right) \times \left(-\frac{R_{FB}}{R_{IN}}\right) \tag{8}$$

where $i$ is chromosome bit index, and $j$ is chromosome index as before. Using the fitness voltages in (8), the first best chromosome is selected in the same way as in the *subset-sum* problem. In the third cycle, the second-best chromosome is selected.

### 3) STORING BEST CHROMOSOMES

After the fitness evaluation and selection identify two best chromosomes, bit patterns of these chromosomes are read from the array and stored in memory for letting them be used as parents in the next generation. Figure 8 shows how to read these bit values. By closing switches S2, the vertical lines are connected to the inverting inputs of the op-amps having a negative feedback, which is treated as the virtual ground. In the first clock cycle, assuming the chromosomes on the first row is read, $V_R$ is applied to the associated horizontal line ($H_1$) and *GND* to all other horizontal lines. Memristive currents (arrows in red) are then converted into voltages by the op-amp circuits and stored in *DFF1* as *parent1* (Figure 5) for use in the next generation. The same procedure is done in the second clock cycle to read the second-best chromosome bit values to be stored in *DFF2* as *parent2*.

## IV. SIMULATION AND EVALUATION

For assessing performance, the proposed GA accelerator with a 64 × 64 memristive crossbar array was designed, which means that there are 64 chromosomes each having 64 bits. CADENCE was used for designing the memristive crossbar array, and Verilog-A for designing other parts. The HP memristor [31] whose electrical behavior was modeled in Verilog-A was used as each memristor in the array. HSPICE simulator was used for timing simulation. For the *subset-sum* and *0-1 knapsack* problems, the weight voltages ranging from

**TABLE 4.** Summary of simulation parameters.

| Name | Meaning | Value |
|---|---|---|
| $V_W$ | Write voltage | 1.1 V |
| $V_R$ | Read voltage | 0.1 V |
| $V_{IM}$ | Intermediate voltage | 0.5 V |
| GND | Ground voltage | 0 V |
| $Wi_{(ssp)}{}^*$ | SSP weights | 0.1 V ~ 0.9 V |
| $Wi_{(Knapsack)}{}^*$ | Knapsack weights | 0.1 V ~ 0.9 V |
| $Vi_{(Knapsack)}{}^*$ | Knapsack profits | 0.1 V ~ 0.9 V |
| MR | Mutation rate | 0.05 |
| $R_{ON}$ | Low resistive value | 1KΩ |
| $R_{OFF}$ | High resistive value | $1000R_{ON}$ |

**i is the index for weight (W) and value (V)*

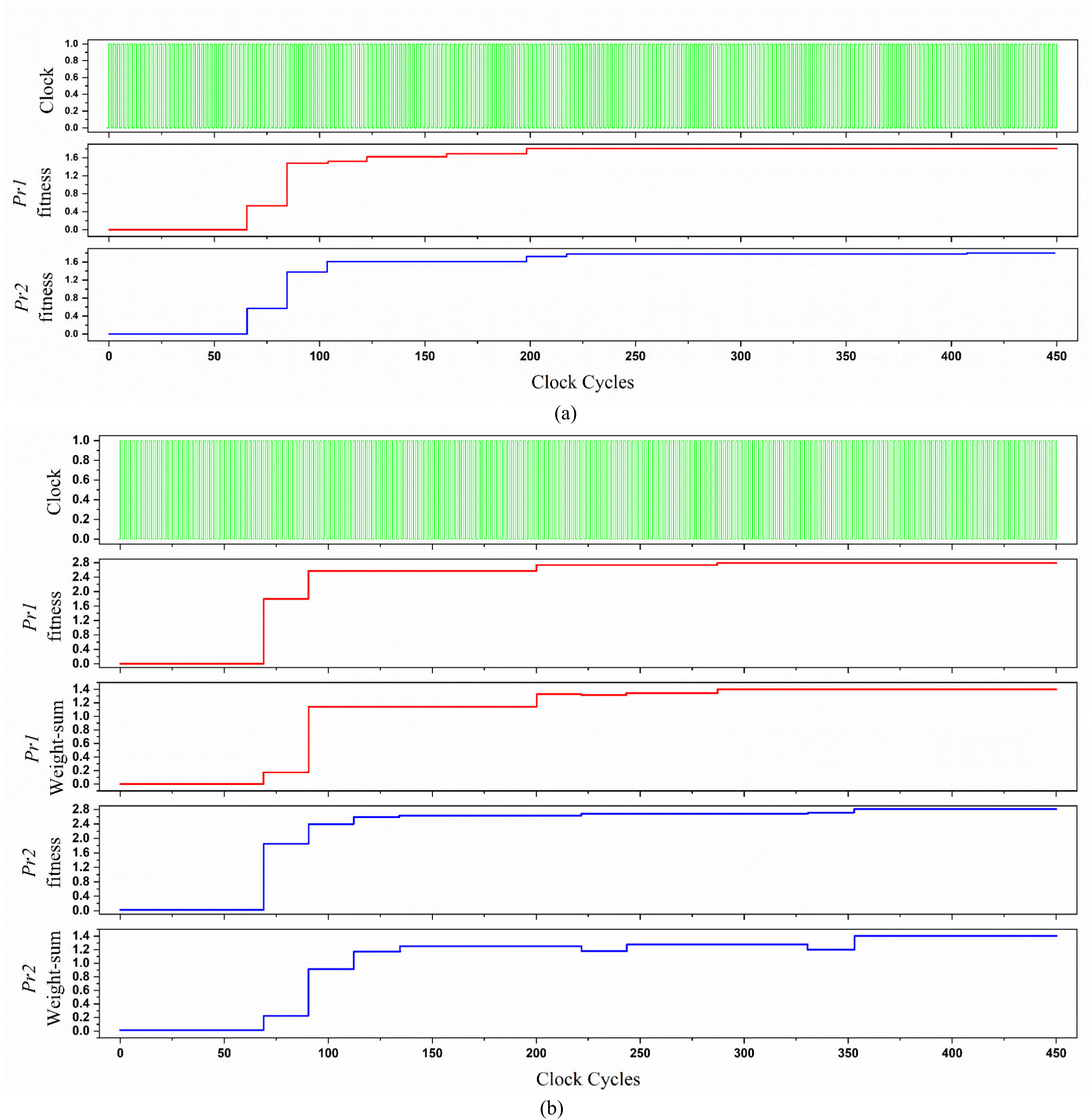



**FIGURE 10.** Simulated waveforms of running GA with 64 × 64 memristive crossbar array for (a) ***subset-sum*** and (b) ***0-1 knapsack*** problems.

0.1 V to 0.9 V and the capacities of 1.8 V and 1.4 V, were used, respectively.

Figure 9 depicts the I-V curve of the HP memristor [31] that can be modeled as

$$v(t) = \left(R_{ON}\frac{w(t)}{D} + R_{OFF}\left(1 - \frac{w(t)}{D}\right)\right).i(t) \quad (9)$$

$$f(w)\frac{dw}{dt} = \mu_v \frac{R_{ON}}{D} i(t) \quad (10)$$

where $D$ is the device length and $w(t)$ is the state variable indicating the length of the doped region. $R_{ON}$ is the resistance when $w(t)=D$ and indicates a low resistive state, $R_{OFF}$ is the resistance when $w(t)=0$ and indicates a high resistive state, and $\mu_v$ is the ion mobility of the device. In order to avoid $w(t)$ growing beyond the physical size of the device, the derivative of it is multiplied by window function $f(w)$ defined as

$$f(w) = 1 - \left(2\left(\frac{w}{D}\right) - 1\right)^{2P_w} \quad (11)$$

where $P_w$ is a positive integer. The parameters of the memristor model used in our simulation are set to be $P_w=2$, $R_{ON}=1K\Omega$, and $R_{OFF}=1M\Omega$. Table 4 shows a list of parameter values used in our simulation.

Figure 10(a) shows the simulated waveforms for solving the *subset-sum* problem in 20 generations. The waveforms depicted from the top are clock input and fitness voltages of the first and second best chromosomes (chromosome indexes in each generation can change), respectively. Figure 10(b) shows simulated waveforms for the weight-sum and value-sum (fitness) voltages of the two best chromosomes of the

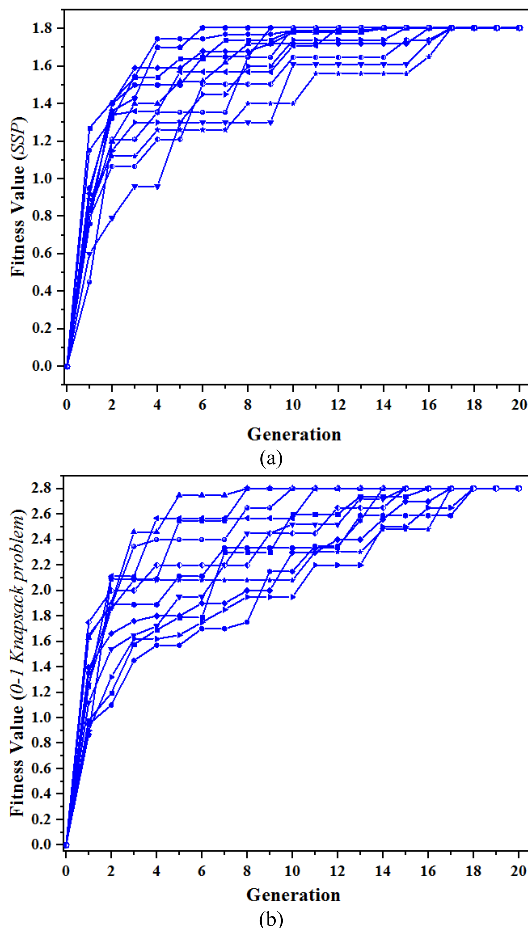



**FIGURE 11.** Fitness convergence for (a) ***subset-sum*** and (b) ***0-1 Knapsack problems*** with nominal resistance of $R_{ON}$ and $R_{OFF}$.

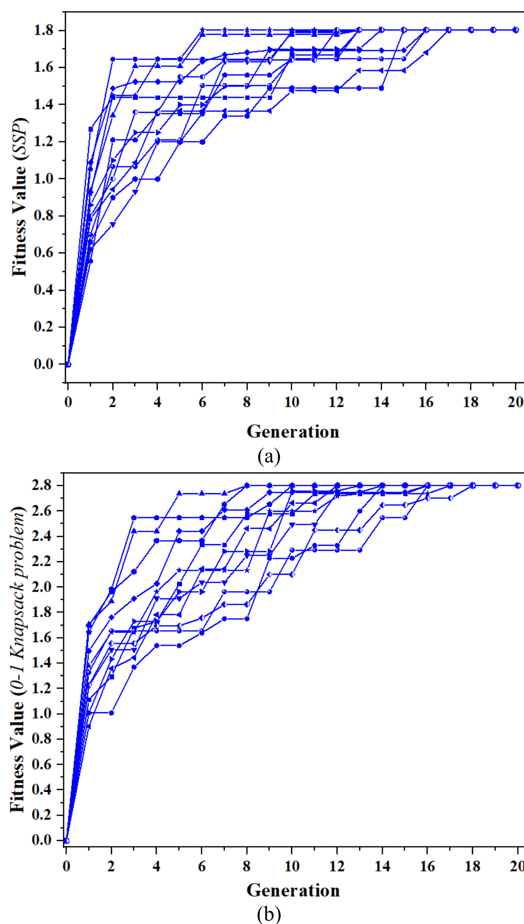



**FIGURE 12.** Fitness convergence for (a) ***subset-sum*** and (b ***) 0-1 Knapsack problems*** with 20% memristance variation.

*0-1 knapsack problem* for 20 generations. In these waveforms, the fitness voltage of parents increases generation after generation and eventually reaches the global optimum. Since the best chromosomes are always kept across generations, each fitness voltage can never go down during running. The weight-sum voltages for the *0-1 knapsack problem* (Figure 10(b)) can sometimes go down when new chromosomes selected as the best have a lower weight-sum value. Figure 11(a) and Figure 11(b) depict the fitness convergence with random initial chromosome bit values for solving *the subset-sum* and *0-1 knapsack problems*, respectively. In all trials for each problem, the fitness converges to the same optimal level regardless of the initial condition. For the subset-sum problem, the fitness voltage converges to 1.8 V as the capacity of the knapsack. For the *0-1 knapsack problem*, it converges to 2.8 V.

To see the effect of memristance variation on fitness convergence, the migration of fitness values was obtained by a crossbar array in which each memristance has up to 20% variation. For using a realistic memristance variation, the lognormal probability density function [34] is used, in which the distributions of $R_{ON}$ and $R_{OFF}$ values are represented by

$$R_{ON} = Lognormal\left(\mu_{R_{ON}}, \sigma_{R_{ON}}\right)$$

$$R_{OFF} = Lognormal\left(\mu_{R_{OFF}}, \sigma_{R_{OFF}}\right) \tag{12}$$

where the *lognormal* function is defined as

$$f_x\left(x; \mu, \sigma\right) = \frac{1}{x\sigma\sqrt{2\pi}} e^{\left(-\frac{(\ln x/\mu)^2}{2\sigma^2}\right)}, x > 0 \tag{13}$$

$x$ is a random variable sampled from the resistance of the memristor for both $R_{ON}$ and $R_{OFF}$ states. $\mu_{\mathrm{ON}}$ and $\sigma_{\mathrm{ON}}$ ($\mu_{\mathrm{OFF}}$ and $\sigma_{\mathrm{OFF}}$) are the mean and standard deviation of the distribution of $R_{ON}$ ($R_{OFF}$), respectively. Figure 12(a) and (b) show the fitness convergence for solving the *subset-sum* and *0-1 knapsack* problems, respectively, where the memristors used have a distribution of variations governed by (12). To see the effect of the memristance variation only, identical initial values of memristors are used in all trials. In both cases, fitness values converge well to target values with their shapes similar to those in Figure 11(a) and (b) having no memristance variation, which indicates that the variation in memristance values gives a negligible difference in fitness convergence. This result can be attributed to the fact that GA inherently has a large randomness during crossover and mutation operations so additional randomness by memristance variation plays almost no additional role.

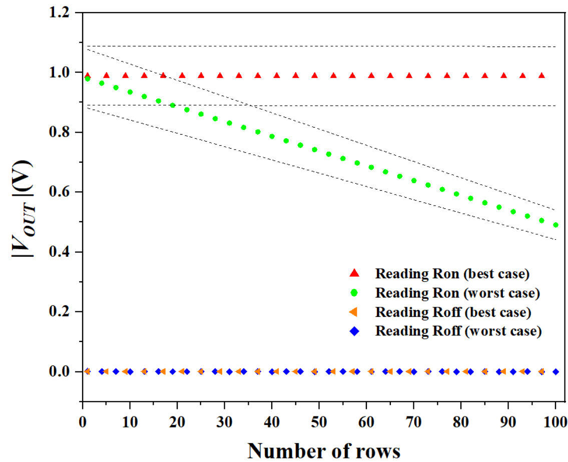



**FIGURE 13.** Read output voltage versus the number of rows or the number of chromosomes.

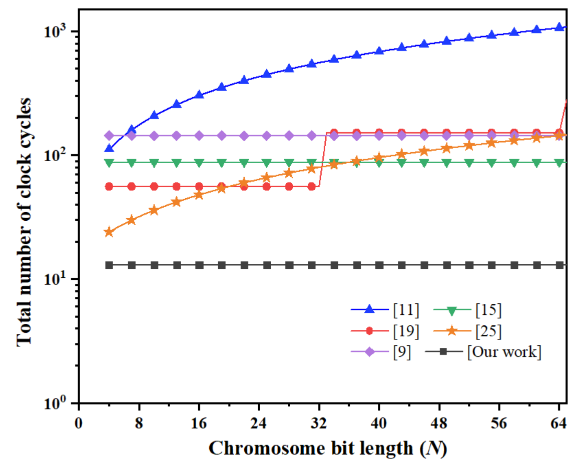



**FIGURE 14.** Number of clock cycles vs. chromosome bit length (*N*) for completing selection, crossover, and mutation operations in a generation for a fixed population (*P*) of 16.

Figure 13 shows the simulated output voltage versus the number of rows or the number of chromosomes when the circuit in Figure 8 reads the bit values of a chromosome. Here, each bit value in the chromosome to be read is at either $R_{ON}$ or $R_{OFF}$ with all other unselected memristors on the same vertical line being at $R_{ON}$ (the worst case) or $R_{OFF}$ (the best case). For all curves, dashed lines indicate the output voltages for the extreme memristance values ($\pm 10\%$ from the nominal value). The gain of the op-amps in Figure 8 is assumed to be 1000, and the feedback resistor ($R_F$) has ten times larger resistance value than the nominal $R_{ON}$ of the memristor. When the number of rows are 100, the output voltage for reading the $R_{ON}$ state in the worst case where all other memristors are at the $R_{ON}$ state is 441 mV as seen in Figure 13. The worst-case read margin will be almost the same as this value because the output voltage for reading the $R_{OFF}$ state is around 1000 times smaller. Note that, considering the fact that a read voltage as low as 300 mV can be easily detected using an ordinary sense amplifier, the size of the memristive crossbar array can be made too large even with memristance variations.

The sneak path current referring to unintended current through unselected memristors can be an issue in the memristive crossbar array. In our case, there is no sneak current issue during the fitness evaluation in Figure 7 because all the memristive currents to be summed are valid. There is also no sneak path current during the read for the selected chromosome because the virtual ground method is used to read the states of memristors, as seen in Figure 8. There can be sneak currents during the crossover and mutation operations shown in Figure 5 and 6. Since these are write operations, the sneak path current causes no functional issue and just increases the power overhead. Using a selector transistor in each memristor cell can eliminate the sneak path current although the approach will increase the array size, resulting in a trade-off between power and area [35].

Figure 14 compares in a log scale the total number of clock cycles for completing the operations of crossover, mutation, and selection in a generation as chromosome bit length ($N$) increases for constant population size ($P = 16$) with conventional ASIC-based [9], [19], FPGA-based [11], [15], memristor-based [25], and proposed GA accelerators. All curves start from $N = 4$ because it is not possible to generate 16 chromosomes having different bit patterns if the chromosome bit length is less than four. For the ASIC-based accelerator in [19], four clock cycles are needed for crossover per a pair of chromosomes when the chromosome bit length is less than 32, requiring 32 ($= 4 \times (16/2)$) total clock cycles. When the chromosome bit length is larger than 32, multi-genetic algorithm processors (multi-GAP) whose number is equal to $\lceil N/32 \rceil$ should be used, which are connected in a chain via SPI ports, resulting in 16 clock cycles for generating a pair of chromosomes. So, the total number of clock cycles increases step-wise ($16 \times (P/2) \times (\lceil N/32 \rceil - 1)$) for $N > 32$. For the selection and replacement, it takes $7\times (P/2)$ additional clock cycles per two chromosomes. For the ASIC design in [9], three clock cycles are needed to do crossover per chromosome whose bit length is up to 64, resulting in 48 clock cycles ($=3 \times 16$) for generating 16 chromosomes, and a total of $11\times P$ clock cycles are required to perform all GA operations. In [15], three clock cycles are required for generating a pair of chromosomes by crossover, requiring 24 ($= 3\times(16/2)$) total clock cycles. For mutation, selection, and replacement, require additional $5\times P$ clock cycles. In [11], the number of clock cycles for completing crossover per chromosome has a linear relationship with the bit length ($2+N$) because multiplexers selecting parent bits are controlled by a serial shift register, resulting in 144 ($= (2+N)\times(P/2)$) clock cycles when the chromosome bit length is 16 and the population size is 16. The number of clock cycles required for the mutation is equal to the number of clock cycles needed for the crossover, and additional $2\times P$ clock cycles are required for selection and replacement. In [25], before a GA generation starts, whole selection matrix values should be written into its memristive array that takes $P$ cycles. The number of forward passes in a generation is equal to the number of columns

**TABLE 5. Performance comparison of GA accelerators.**

| | Crossover | Mutation | Selection | Fitness evaluation | Replacement | Equation for total number of clock cycles per generation | Total number of clock cycles per generation ($P$=$N$=64) | Hardware Platform | Publication Year |
|---|---|---|---|---|---|---|---|---|---|
| [19] | $\left(\max\left\{16\times\left(\left\lceil\frac{N}{32}\right\rceil^{*}-1\right),4\right\}\right)\left(\frac{P}{2}\right)$ | | $3\left(\frac{P}{2}\right)$ | External | $4\left(\frac{P}{2}\right)$ | $\left(\max\left\{16\times\left(\left\lceil\frac{N}{32}\right\rceil^{*}-1\right),4\right\}+7\right)\left(\frac{P}{2}\right)$ $N$: unlimited, $32 \le P \le 256$ (excluding fitness evaluation) | 736 + clock_cycle_fitness* | ASIC | 2015 |
| [7] | $149\left(\frac{P}{2}\right)$ | | NA | External | NA | NA | 4800 + clock_cycle_fitness* | ASIC | 1998 |
| [9] | $3P$ | | $6P$ | External | $2P$ | $11P$ (excluding fitness evaluation) | 704 + clock_cycle_fitness* | ASIC | 1999 |
| [15] | $3\left(\frac{P}{2}\right)$ | $2P$ | $2P$ | LUT or User-defined circuit | $P$ | $6.5P$ $8 \le N \le 1024$; $32 \le P \le 16384$, (excluding fitness evaluation) | 416 + clock_cycle_fitness* | FPGA | 2008 |
| [11] | $(2+N)\left(\frac{P}{2}\right)$ | $(2+N)\left(\frac{P}{2}\right)$ | $P$ | Pipelined with multiple FPGAs | $P$ | $4P$+$NP$ $N = 70$, $P = 256$, (excluding fitness evaluation) | 4352 + clock_cycle_fitness* | FPGA | 2001 |
| [21] | NA | NA | NA | External | NA | NA | 572 + clock_cycle_fitness* | FPGA | 2018 |
| [18] | NA | NA | NA | Use IP core as an external module | NA | NA | 2190 + clock_cycle_fitness* | FPGA | 2009 |
| [25] | $P + (2\times N)$ | | | NA | 0 | $P + (2\times N)$ (excluding fitness evaluation) | 192 + clock_cycle_fitness* | Memristor | 2022 |
| Our Method | $(log_2(P)\times 2) + 1$** | 2 | 2**** | *Knapsack*: 2*** *SSP*: 1*** | 0 | $\left(\left(log_2(P)\right)\times 2\right)+7$ (excluding fitness evaluation) | 17 + 2 (with *0-1 knapsack* fitness) | Memristor | - |

$\lceil x \rceil$ *denotes the ceiling function that gives the largest integer greater than or equal to x*
* *The number of clock cycles for fitness evaluation in each implementation*
** *For resetting all chromosomes before starting the crossover operation*
*** *For calculating the fitness values for all chromosomes and selecting two best chromosomes*
**** *For reading out the best chromosomes from the array and storing them into DFFs*

in the population matrix. Before each forward pass starts, a single column from the mutation and crossover matrices should be written into respective memristive vectors. So, $N$ cycles are required to process all the columns in crossover and mutation matrices per generation. Because, in each forward pass, a single column in the population matrix is applied to the memristive network as the input vector, the number of clock cycles proportional to $N$ is needed to update the whole population matrix. Therefore, preparing a new population in each generation requires a total of $P$+$(2\times N)$ clock cycles. Meanwhile, in our design, the number of clock cycles required for completing the crossover in a generation is as low as eight ($log_2(16)\times 2$ as seen in (1) and (2)). The number of clock cycles here does not increase with increase of the chromosome bit length for a fixed population size, which is because child chromosomes in the array update their bit values segment by segment in parallel as explained earlier. Considering additional 6 clock cycles to do mutation and selection operations and reading the selected chromosomes, 15 clock cycles are required independent of chromosome bit length, as seen in Figure 14.

Figure 15 compares in a log scale the total number of clock cycles needed to complete the crossover, mutation, and selection operations in a generation as the population size ($P$) increases when the chromosome bit length is constant ($N = 16$). The clock cycles required to do these operations have a linear relationship to $P$ with different slopes for ASIC-based [9], [19] and FPGA-based [11], [15] designs. For the memristive array-based design in [25], the total clock cycles for completing these operations in a generation are $P$+$(2\times N)$ as seen previously. For the proposed accelerator, the number of clock cycles for crossover has a logarithmic relationship to the population size ($log_2(P)\times 2$). Considering four additional cycles for mutation and selection, two cycles for reading two best chromosomes, and one cycle to reset the memristors, a total of 19 clock cycles (without considering fitness evaluation) are required for a population of 64 chromosomes, which is increasing logarithmically, as seen in Figure 15.

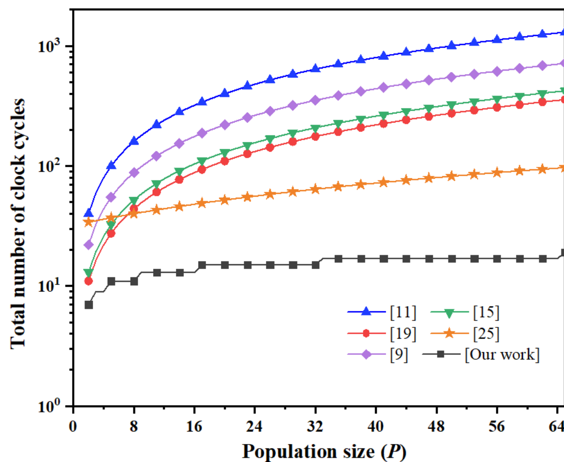



**FIGURE 15.** Number of clock cycles vs. population size ($P$) for completing selection, crossover, and mutation operations in a generation for a fixed chromosome bit length ($N$) of 16.

Table 5 compares the performance of various GA accelerators in terms of clock cycles required for completing operations such as crossover, mutation, selection, replacement, and fitness evaluation in a generation. The first two columns indicate the number of clock cycles for the crossover and mutation operations, respectively. In [27], they designed a custom hardware in which both crossover and mutation are done in one module. According to relevant equations in the paper, four clock cycles are needed for both crossover and mutation when the chromosome bit length is less than 32. For chromosomes whose bit lengths are longer than 32, multiple GAPs are required so the number of clock cycles increases due to the communication overhead between GAPs. In other conventional designs, the number of clock cycles for performing crossover and mutation are proportional either only to population size ($P$) or both to population size and chromosome bit length ($N$). Meanwhile, in our design, the number of clock cycles required for crossover has no relationship with chromosome bit length and a logarithmic relationship with population size. Moreover, the number of clock cycles for mutation is constant regardless of population size and chromosome bit length. For fitness evaluation, all conventional works were reported to rely on either external hardware or custom IP core with no mention of detail. It can be expected that they will require multiple clock cycles. Meanwhile, as explained earlier, the proposed accelerator can finish the *subset-sum* and *0-1 knapsack* fitness evaluation in just one and two clock cycles, respectively, taking advantage of the inherently massive parallelism of the memristive crossbar array. The number of clock cycles required to do the selection for almost all conventional works depends on population size ($P$), whereas in our design it can be done in just up to four clock cycles depending on the type of fitness function. In the proposed GA accelerator, the memristance values change as desired during the crossover operation, so no explicit replacement operation is needed. Table 5 also shows the expressions for total clock cycles required for doing whole operations in a generation as a function of population size ($P$) and chromosome bit length ($N$). Note that, for each conventional work, the number of clock cycles for fitness evaluation is not included since it is not available. As an example, comparison, the total number of clock cycles required for operations with a chromosome bit length of 64 and a population size of either 32 or 64 are listed, in which our design takes only 19 clock cycles including the *0-1 knapsack* fitness evaluation. For conventional designs, the clock cycles required are in a range between 416 and 4800 without considering the fitness evaluation.

## V. CONCLUSION

This paper proposes a memristive crossbar array-based hardware accelerator for fast processing the generic GA operations like crossover, mutation, selection, and fitness evaluation. The proposed architecture takes advantage of the inherent massive parallelism provided by the crossbar array to surpass conventional von Neumann-based GA accelerators. The crossover and mutation in GA are known to be computationally intensive due to a lot of computations relevant to chromosomes and many data movements between memory and processor. To address the issue, the crossover operation is done in parallel for the chromosome bits in a segment in the array. Specifically, we used a new method for the GA crossover operation called *aligned hybrid crossover*. We also performed the mutation per generation in only two clock cycles. Since evaluating the fitness values of chromosomes is another computationally intensive task, an efficient way of evaluating the fitness functions using our memristive crossbar array is proposed, allowing the fitness function to be evaluated inside the memristor crossbar array in only one cycle. The effects of memristance variation in the array on the fitness evaluation and the read margin are also investigated. The performance of our GA accelerator based on a 64 × 64 memristive crossbar array indicating that there are 64 chromosomes having 64 bits each was evaluated. It showed that over 10 times reduced number of clock cycles for completing the entire GA operations in a generation was achieved, which indicates that the proposed GA accelerator is considerably more efficient than previous designs.

## REFERENCES

[1] M. D. Vose, *The Simple Genetic Algorithm: Foundations and Theory*. Cambridge, MA, USA: MIT Press, 1999.

[2] J. H. Holland, "Genetic algorithms and the optimal allocation of trials," *SIAM J. Comput.*, vol. 2, no. 2, pp. 88–105, Jun. 1973.

[3] P. Z. Lappas and A. N. Yannacopoulos, "A machine learning approach combining expert knowledge with genetic algorithms in feature selection for credit risk assessment," *Appl. Soft Comput.*, vol. 107, Aug. 2021, Art. no. 107391.

[4] A. A. Mamun, M. Hoq, E. Hossain, and R. Bayindir, "A hybrid deep learning model with evolutionary algorithm for short-term load forecasting," in *Proc. 8th Int. Conf. Renew. Energy Res. Appl. (ICRERA)*, Nov. 2019, pp. 886–891.

[5] H. Chung and K.-S. Shin, "Genetic algorithm-optimized multi-channel convolutional neural network for stock market prediction," *Neural Comput. Appl.*, vol. 32, no. 12, pp. 7897–7914, Jun. 2020.

[6] S. D. Scott, A. Samal, and S. Seth, "HGA: A hardware-based genetic algorithm," in *Proc. 3rd Int. ACM Symp. Field-Programmable Gate Arrays*, Feb. 1995, pp. 53–59.

[7] S. Wakabayashi, T. Koide, K. Hatta, Y. Nakayama, M. Goto, and N. Toshine, "GAA: A VLSI genetic algorithm accelerator with on-the-fly adaptation of crossover operators," in *ISCAS 98. Proc. IEEE Int. Symp. Circuits Syst.*, vol. 2, Jul. 1998, pp. 268–271.

[8] J. Jung Kim and D. Jin Chung, "Implementation of genetic algorithm based on hardware optimization," in *Proc. IEEE IEEE Region 10 Conf. TENCON 99. Multimedia Technol. Asia–Pacific Inf. Infrastruct.*, vol. 2, May 1999, pp. 1490–1493.

[9] N. Yoshida and T. Yasuoka, "Multi-GAP: Parallel and distributed genetic algorithms in VLSI," in *Proc. IEEE SMC99 Conf. Proc. IEEE Int. Conf. Syst. Man, Cybern.*, vol. 5, Jun. 1999, pp. 571–576.

[10] S. Koizumi, S. I. Wakabayashi, T. Koide, K. Fujiwara, and N. Imura, "A RISC processor for high-speed execution of genetic algorithms," in *Proc. 3rd Annu. Conf. Genetic Evol. Comput.*, 2001, pp. 1338–1345.

[11] B. Shackleford, "A high-performance, pipelined, FPGA-based genetic algorithm machine," *Genetic Program. Evolvable Mach.*, vol. 2, no. 1, pp. 33–60, 2001.

[12] N. Yoshida, T. Yasuoka, T. Moriki, and T. Shimokawa, "VLSI hardware design for genetic algorithms and its parallel and distributed extensions," *Int. J. Knowl. BASED Intell. Eng. Syst.*, vol. 5, no. 1, pp. 14–22, 2001.

[13] S.-D. Chen, P.-Y. Chen, and Y.-M. Wang, "A flexible genetic algorithm chip," in *Proc. Nat. Comp. Symp.*, vol. 2003, pp. 253–257.

[14] T. Imai, M. Yoshikawa, H. Terai, and H. Yamauchi, "VLSI processor architecture for real-time GA processing and PE-VLSI design," in *Proc. IEEE Int. Symp. Circuits Syst.*, Oct. 2004, p. 625.

[15] P.-Y. Chen, R.-D. Chen, Y.-P. Chang, L.-S. Shieh, and H. A. Malki, "Hardware implementation for a genetic algorithm," *IEEE Trans. Instrum. Meas.*, vol. 57, no. 4, pp. 699–705, 2008.

[16] K. M. Deliparaschos, G. C. Doyamis, and S. G. Tzafestas, "A parameterised genetic algorithm IP core: FPGA design, implementation and performance evaluation," *Int. J. Electron.*, vol. 95, no. 11, pp. 1149–1166, Nov. 2008.

[17] P. R. Fernando, S. Katkoori, D. Keymeulen, R. Zebulum, and A. Stoica, "Customizable FPGA IP core implementation of a general-purpose genetic algorithm engine," *IEEE Trans. Evol. Comput.*, vol. 14, no. 1, pp. 133–149, Feb. 2010.

[18] M. Vavouras, K. Papadimitriou, and I. Papaefstathiou, "High-speed FPGA-based implementations of a genetic algorithm," in *Proc. Int. Symp. Syst., Archit., Model., Simul.*, Jul. 2009, pp. 9–16.

[19] S. P. Hoseini Alinodehi, S. Moshfe, M. Saber Zaeimian, A. Khoei, and K. Hadidi, "High-speed general purpose genetic algorithm processor," *IEEE Trans. Cybern.*, vol. 46, no. 7, pp. 1551–1565, Jul. 2016.

[20] S. P. H. Alinodehi, S. J. Louis, S. Moshfe, and M. Nicolescu, "A modified steady state genetic algorithm suitable for fast pipelined hardware," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Jun. 2017, pp. 2526–2533.

[21] M. Peker, "A fully customizable hardware implementation for general purpose genetic algorithms," *Appl. Soft Comput.*, vol. 62, pp. 1066–1076, Jan. 2018.

[22] M. F. Torquato and M. A. C. Fernandes, "High-performance parallel implementation of genetic algorithm on FPGA," *Circuits, Syst., Signal Process.*, vol. 38, no. 9, pp. 4014–4039, Sep. 2019.

[23] Y. Yongbin, Y. Chenyu, D. Quanxin, N. Tashi, L. Shouyi, and Z. Chen, "Memristive network-based genetic algorithm and its application to image edge detection," *J. Syst. Eng. Electron.*, vol. 32, no. 5, pp. 1062–1070, Oct. 2021.

[24] K. N. Edwards and X. Shen, "Comparison of update and genetic training algorithms in a memristor crossbar perceptron," *AIP Adv.*, vol. 12, no. 2, Feb. 2022, Art. no. 025327.

[25] Y. Yu, J. Mo, Q. Deng, C. Zhou, B. Li, X. Wang, N. Yang, Q. Tang, and X. Feng, "Memristor parallel computing for a matrix-friendly genetic algorithm," *IEEE Trans. Evol. Comput.*, vol. 26, no. 5, pp. 901–910, Oct. 2022.

[26] L. Chua, "Memristor–The missing circuit element," *IEEE Trans. Circuit Theory*, vols. CT-18, no. 5, pp. 507–519, May 1971.

[27] G. Syswerda, "A study of reproduction in generational and steady-state genetic algorithms," *Foundations of Genetic Algorithms*, vol. 1, pp. 94–101, Jan. 1991.

[28] K. D. Muleta and B.-S. Kong, "RRAM-based spiking neural network with target-modulated spike-timing-dependent plasticity," *IEEE Trans. Biomed. Circuits Syst.*, early access, Aug. 19, 2024, doi: 10.1109/TBCAS.2024.3446177.

[29] F. Cai, J. M. Correll, S. H. Lee, Y. Lim, V. Bothra, Z. Zhang, M. P. Flynn, and W. D. Lu, "A fully integrated reprogrammable memristor–CMOS system for efficient multiply–accumulate operations," *Nature Electron.*, vol. 2, no. 7, pp. 290–299, Jul. 2019.

[30] Y. Li and K.-W. Ang, "Hardware implementation of neuromorphic computing using large-scale memristor crossbar arrays," *Adv. Intell. Syst.*, vol. 3, no. 1, Jan. 2021, Art. no. 2000137.

[31] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, May 2008.

[32] S. Martello, *Knapsack Problems: Algorithms and Computer Implementations*. Hoboken, NJ, USA: Wiley, 1990.

[33] M.-S. Seol and B.-S. Kong, "A compact low-power time-domain winner-take-all circuit," in *Proc. 30th IEEE Int. Conf. Electron., Circuits Syst. (ICECS)*, Dec. 2023, pp. 1–16.

[34] M. Hu, Y. Wang, Q. Qiu, Y. Chen, and H. Li, "The stochastic modeling of $TiO_2$ memristor and its usage in neuromorphic system design," in *Proc. 19th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2014, pp. 831–836.

[35] H. Li, S. Wang, X. Zhang, W. Wang, R. Yang, Z. Sun, W. Feng, P. Lin, Z. Wang, L. Sun, and Y. Yao, "Memristive crossbar arrays for storage and computing applications," *Adv. Intell. Syst.*, vol. 3, no. 9, Sep. 2021, Art. no. 2100017.

**MOHAMMADHADI BAGHBANMANESH** received the B.S. degree in electronics engineering from Shiraz University, Shiraz, Iran, in 2012. He is currently pursuing the master's degree with the Department of Electrical and Electronic Engineering, Sungkyunkwan University, Suwon, South Korea. His research interests include the design of mixed-signal ultra-low-power neuromorphic chips, with a focus on memristors and high-performance computing (HPC).

**BAI-SUN KONG** (Member, IEEE) received the B.S. degree in electronics engineering from Yonsei University, Seoul, South Korea, in 1990, and the M.S. and Ph.D. degrees in electrical engineering from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 1996.

In 1996, he was with LG Semicon Company (currently SK Hynix Corporate), Seoul, as a Senior Design Engineer, where he worked on the design of high-density and high-bandwidth DRAMs. In 2000, he joined the Faculty of Korea Aerospace University, Goyang, South Korea, as an Assistant Professor with the School of Electronics Telecommunication and Computer Engineering. In 2005, he moved to Sungkyunkwan University, Suwon, South Korea, where he is currently a Professor with the College of Information and Communication Engineering. From 2018 to 2019, he was with the Center for Nanotechnology, NASA Ames Research Center, CA, U.S., and the Nanoelectronic Integrated Systems Laboratory, University of California, Santa Cruz, CA, U.S., where he was involved in collaborative research on neuromorphic circuit and system design. His research interests include high-speed low-power microprocessor and memory design, high-speed wireline transceiver design, fast-transient high-efficiency dc-dc converter design, and neuromorphic IC design for bio-inspired applications. He has received the Most Excellent Design Paper Award from Asia and South Pacific Design Automation Conference (ASP-DAC) and the 2023 Darlington Best Paper Award from the IEEE Circuits and Systems (CAS) Society.