A Low-Computation-Complexity, Energy-Efficient, and High-Performance Linear Program Solver Using Memristor Crossbars

Ruizhe Cai, Ao Ren, Yanzhi Wang, Sucheta Soundarajan, Qinru Qiu
Electrical Engineering and Computer Science
Syracuse University
Syracuse, NY, USA
{rcai100, aren, ywang393, susounda, qiqiu}@syr.edu

Bo Yuan
Electrical Engineering
City University of New York
New York, NY, USA
byuan@ccny.cuny.edu

Paul Bogdan
Electrical Engineering
University of Southern California
Los Angeles, CA, USA
pbogdan@usc.edu

***Abstract*—Linear programming is required in a wide variety of application including routing, scheduling, and various optimization problems. The primal-dual interior point (PDIP) method is state-of-the-art algorithm for solving linear programs, and can be decomposed to matrix-vector multiplication and solving systems of linear equations, both of which can be conducted by the emerging memristor crossbar technique in O(1) time complexity in the analog domain. This work is the first to apply memristor crossbar for linear program solving based on the PDIP method, which has been reformulated for memristor crossbars to compute in the analog domain. The proposed linear program solver can overcome limitations of memristor crossbars such as supporting only non-negative coefficients, and has been extended for higher scalability. The proposed solver is iterative and achieves O(*N*) computation complexity in each iteration. Experimental results demonstrate that reliable performance with high accuracy can be achieved under process variations.**

I. Introduction

Linear programs are common in a wide variety of applications, including routing, scheduling, and other optimization problems. Interior point methods are a popular class of algorithms for solving linear programs. Unlike the well-known simplex algorithm, which traverses vertices of the feasible region to find the optimal solution, interior point methods trace a path through the interior of the feasible region. The *primal-dual interior point* (PDIP) method uses the gap between the current solutions of the primal linear program and its dual in order to determine the path to follow within the feasible region. In each iteration, the algorithm involves calculating matrix-vector product and solving systems of linear equations. The emerging memristor crossbar technology can be potentially utilized to achieve significant speed-ups due to its significant benefits in matrix operations.

Memristor was predicted as the fourth circuit element nearly half a century ago [17]. It was not physically created until 2008 by HP lab [18]. Non-volatility, low power consumption, and excellent scalability are some of its promising features. More importantly, its capability to record historical resistance makes it unique, and has resulted in heightened interests over the last several years. A crossbar structure of memristor devices (i.e. a *memristor crossbar*) can be utilized to perform matrix-vector multiplication and solve systems of linear equations in the analog domain in O(1) time complexity [7][8][9]. Such advantages in matrix operations make it ideal candidate for implementing the state-of-the-art PDIP method for solving linear programs given its high usage of matrix-vector multiplication and solving linear systems. Moreover, the effect of process variations of memristor devices can be significantly mitigated (as shown in experimental results) by the inherent noise tolerance of the iterative PDIP algorithm.

Although promising, multiple challenges need to be overcome when applying memristor devices for linear program solving. Since the memristor crossbar performs matrix operations in the analog domain, we need to formulate the whole PDIP algorithm using memristor crossbar in the analog domain in order to avoid the significant overhead of D/A and A/D conversions. Moreover, some limitations of memristor crossbars (e.g., only non-negative matrix coefficients and square matrices when solving a system of linear equations can be supported) need to be properly addressed.

To the best of our knowledge, this paper is the first to provide a comprehensive algorithm-hardware framework on memristor crossbar for linear program solving. The PDIP method is reformulated for memristor crossbar and analog computations. The proposed solver can effectively deal with matrices containing negative numbers, and has been extended for linear program solving with higher scalability that can overcome size limitations of the memristor crossbar structure. The proposed solver achieves pseudo-O(*N*) computation complexity, i.e., O(*N*) complexity in each iteration, which is a significant improvement compared with the software-based PDIP method of O(N^3). Experimental results demonstrate that the performance of proposed implementation is reliable with less than 4% inaccuracy on average under 10% process variations. Based on our estimation, the proposed solver could lead to 7960X improvements in speed and 6.7×10^5 X reduction in energy consumption.

II. Background

A. Linear Program Solving Methods

The simplex method of Dantzig was the first efficient algorithm for solving linear programming problems, and is still popular today [21]. The simplex algorithm considers the feasible region of the linear program (i.e., the space of points satisfying all constraints), which is a polytope. The algorithm begins at one vertex of the polytope, and moves from vertex to vertex in such a way as to increase the value of the objective function. The simplex algorithm is extremely efficient in practice, but has exponential running time in the worst case [20].

Interior point methods for solving linear programs were developed in response to this inefficiency. Unlike the simplex algorithm, which moves from vertex to vertex of the feasible

978-1-5090-1367-8/16/$31.00 ©2016 IEEE

region, interior point methods traverse the interior of this region. Karmarkar's projective method was the first interior point algorithm that was both polynomial time in the worst case as well as fast in practice [20]. This method first begins at an interior point within the feasible region. It next applies a projective transformation so that the current interior point is the center of the projective space, and then moves in the direction of steepest descent. This is repeated until convergence.

The primal-dual interior point method uses the above technique, but incorporates information from the dual of the problem. Every linear program has a dual program, with the property that when the primal linear program has an optimal solution, the dual linear program also has the same optimal solution, and these two solutions are equal. The primal-dual interior point method exploits this property by simultaneously solving both the primal linear program as well as its dual, and steadily decreasing the duality gap (i.e., the difference between the value of the current solution to the primal and the current solution to the dual).

B. Memristor and Memristor Crossbar

Memristor was introduced by L.O. Chua as the fourth element of circuit and was founded by HP labs in 2008 [17][18]. It remembers its most recent resistance, which can be altered by excitation with energy greater than a threshold [3]-[6]. More specifically, the state of a memristor will change when certain voltage higher than the threshold voltage, i.e., $|V_m| > |V_{th}|$, is applied at its two terminals for a small time period. Otherwise, the memristor behaves like a resistor. Such memristive property makes it an ideal candidate for non-volatile memory and matrix computations [7][8].

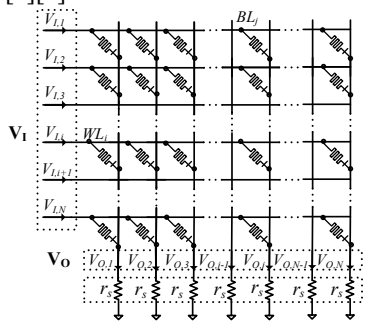


Fig. 1. Structure of Memristor Crossbar

With its high degree of parallelism, the memristor crossbar array is attractive for matrix computations (which can often be performed with O(1) time complexity). A typical structure of an $N \times N$ memristor crossbar is shown in Fig. 1, in which a memristor is connected between each pair of horizontal word-line (**WL**) and vertical bit-line (**BL**). This structure could provide large number of signal connections within a small footprint. In addition, it is capable of reprogramming each memristor to different resistance states by properly applying biasing voltages at its two terminals [1][2][9]. For multiplications, a vector of input voltages $\mathbf{V_I}$ is applied on **WL**s and the current through each **BL** can be collected by measuring the voltage across resistor R_S with conductance of g_S. Assume that the memristor at the connection between WL_i and BL_j has a conductance of $g_{i,j}$. Then the output voltages are represented by $\mathbf{V_O} = \mathbf{C} \cdot \mathbf{V_I}$, where the connection matrix $\mathbf{C}$ is constructed by a programmed crossbar array, which transfers the input vector $\mathbf{V_I}$ to the output vector $\mathbf{V_O}$. $\mathbf{C}$ is determined by the conductance of memristors:

$$\mathbf{C} = \boldsymbol{D} \cdot \boldsymbol{G}^T = diag(d_1, \dots, d_N) \cdot \begin{bmatrix} g_{1,1} & \cdots & g_{1,N} \\ \vdots & \ddots & \vdots \\ g_{N,1} & \cdots & g_{N,N} \end{bmatrix}^T \quad (1)$$

where $d_i = 1/(g_s + \sum_{k=1}^{N} g_{k,i})$.

In reverse, the memristor crossbar structure can also be used to solve a linear system of equations, by mapping the linear equations to the memristor crossbar structure. A voltage vector $\mathbf{V_O}$ is applied on each R_S of **BL**, so the current flowing through each **BL** can be approximated as $I_{o,j} = g_S V_{o,j}$. On the other hand, current $I_{o,j}$ through BL_j can also be calculated as $I_{o,j} = \sum_j V_{I,i} g_{i,j}$. Hence, for each BL_j, equation $\frac{1}{g_s}\sum_j V_{I,i} g_{i,j} = V_{o,j}$ is mapped. Therefore, the system of linear equations $\mathbf{C} \cdot \mathbf{V_I} = \mathbf{V_O}$ is mapped to the memristor crossbar structure, and solution $\mathbf{V_I}$ can be determined by measuring voltages on the **WL**s. Please note that, elements of matrix $\mathbf{C}$ should be non-negative in order to be mapped to memristor crossbar, because resistance cannot reach negative values. It is worth mentioning that the matrix calculation process with the memristor crossbar just has a negligible effect on memristance of each memristor, because the time period that current go through a memristor is short enough during the calculation process.

It is proved in [9] that a fast and simple approximation $g_{i,j} = c_{i,j} \cdot g_{max}$ can be adopted for mapping above matrix onto the memristor crossbar (g_{max} is the largest value in $\boldsymbol{G}$). Therefore for matrix-vector multiplication $\mathbf{A}x = b$, $\mathbf{A} = g_{max} \cdot \mathbf{C}$ and $b = g_s \boldsymbol{V_O}$; for the solution of linear system $\mathbf{A}x = b$, $\mathbf{A} = g_{max} \cdot \mathbf{C}$ and $x = \frac{g_s}{g_{max}} \boldsymbol{V_I}$.

III. Memristor Crossbar-Based Linera Program Solver

We present a *memristor crossbar based linear program solver based on the PDIP algorithm*, which overcomes hardware limitations of memristor crossbar while taking its advantages on matrix operations. The presented solver could handle the vastly used matrix operations in PDIP algorithm efficiently with significantly reduced computation complexity (to pseudo-O(N)), power consumption, and latency. Moreover, the proposed solver can deal with matrices containing negative numbers that cannot be directly mapped on to memristor crossbars. In addition, we introduce an extension for linear program solving with higher scalability that can overcome size limitations of the memristor crossbar structure.

This section is organized in five parts: The PDIP algorithm is discussed in part A; The proposed memristor crossbar-based linear program solver is introduced in part B; Part C discusses writing coefficients in memristor crossbar, and the proposed solutions for representing and computing large-scale matrices are introduced in part D. Part E investigates computation complexity of proposed memristor crossbar-based solvers.

A. The Primal-Dual Interior Point (PDIP) Method for Solving Linear Programs

Linear programs or linear programming problems [15] are problems that can be expressed as:

Maximize $\boldsymbol{c^T x}$ subject to:

$\boldsymbol{Ax} \preccurlyeq \boldsymbol{b}$ ($\boldsymbol{A} \in \mathbb{R}^{m \times n}$) $\quad \boldsymbol{x} \succcurlyeq \mathbf{0}$

where $\boldsymbol{Ax} \preccurlyeq \boldsymbol{b}$ means that every element in vector $\boldsymbol{Ax}$ is smaller than or equal to corresponding element in vector $\boldsymbol{b}$. Every linear program can be converted into a symmetrical dual problem:

Minimize $\boldsymbol{b^T y}$ subject to:

$\boldsymbol{A^T y \geqslant c \quad y \geqslant 0.}$

By Introducing two additional variables, inequality constraints can be transformed into equality constraints. The above problem can be reformulated as follows [16]:

Maximize $\boldsymbol{c^T x}$ subject to:

$$\boldsymbol{Ax + w = b} \qquad \boldsymbol{x, w \geqslant 0} \tag{2a}$$

and its dual:

Minimize $\boldsymbol{b^T y}$ subject to:

$$\boldsymbol{A^T y - z = c} \qquad \boldsymbol{y, z \geqslant 0} \tag{2b}$$

with complementary conditions:

$$\forall i \in \mathbb{N} \cap [1, n] \wedge \forall j \in \mathbb{N} \cap [1, m]: x_i z_i = 0, y_j w_j = 0$$

which can be represented using the following matrix notations:

$$\boldsymbol{XZ_e = 0, YW_e = 0} \tag{2c}$$

In the above equation, uppercase notations are utilized to denote diagonal matrices, e.g.,

$$\boldsymbol{X} = diag(x_1, \dots, x_n),$$

where $\boldsymbol{x} = [x_1, \dots, x_n]^T$, and the subscript e stands for the reverse operation, that is $\boldsymbol{X_e} = [\boldsymbol{X_{11}}, \dots, \boldsymbol{X_{ii}}, \dots, \boldsymbol{X_{nn}}]^T$.

Due to nonlinearity characteristics in (2c), the above problem is difficult to solve directly. The interior point algorithm [14][16] is introduced to solve this problem effectively. In this algorithm, $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{w}$, $\boldsymbol{z}$ are initialized as arbitrary vectors and updated iteratively until Eqns. (2a) – (2c) are (sufficiently) satisfied. In each iteration, a set of vectors $\Delta\boldsymbol{x}$, $\Delta\boldsymbol{y}$, $\Delta\boldsymbol{w}$, $\Delta\boldsymbol{z}$, which are referred to as *step direction vectors*, are derived from solving the following system of equations:

$$\boldsymbol{A(x + \Delta x) + (w + \Delta w) = b} \tag{3a}$$

$$\boldsymbol{A^T(y + \Delta y) - (z + \Delta z) = c} \tag{3b}$$

$$\boldsymbol{(X + \Delta X)(Z + \Delta Z)_e = \mu} \tag{3c}$$

$$\boldsymbol{(Y + \Delta Y)(W + \Delta W)_e = \mu} \tag{3d}$$

where $\boldsymbol{\mu}$ is a small value vector. Since $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{w}$, $\boldsymbol{z}$ are nonnegative vectors, the complementary conditions in (2c) are replaced with $\boldsymbol{\mu}$-complementary conditions (3c) and (3d). Ignoring the second-order elements in (3c) and (3d), the above system of equations can be represented as a system of linear equations of $\Delta\boldsymbol{x}$, $\Delta\boldsymbol{y}$, $\Delta\boldsymbol{w}$, $\Delta\boldsymbol{z}$, denoted by:

$$\boldsymbol{A\Delta x + \Delta w = b - Ax - w} \tag{4a}$$

$$\boldsymbol{A^T\Delta y - \Delta z = c - A^T y + z} \tag{4b}$$

$$\boldsymbol{Z\Delta x + X\Delta z = \mu - XZ_e} \tag{4c}$$

$$\boldsymbol{W\Delta y + Y\Delta w = \mu - YW_e} \tag{4d}$$

The unknown vectors $\Delta\boldsymbol{x}$, $\Delta\boldsymbol{y}$, $\Delta\boldsymbol{w}$, $\Delta\boldsymbol{z}$ can be calculated from solving the system of linear equations (4a)-(4d) and applied to update $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{w}$, $\boldsymbol{z}$. Above steps are repeated until $\boldsymbol{Ax + w - b}$ and $\boldsymbol{A^T y - z - c}$ are small enough.

B. Memristor Crossbar-based Linear Program Solver Using PDIP Algorithm

The memristor crossbar array structure has high potential for implementing PDIP algorithms due to its advantages in matrix operations. However, the memristor crossbar array structure has some limitations, which necessitate the adjustment of PDIP algorithm for effective memristor crossbar based implementations. Since the matrix elements are represented as non-negative memristance values in the memristor crossbar, a novel mechanism is required for representing negative matrix coefficients. In addition, the linear system to be solved should have a square coefficients matrix. Next, we propose a memristor crossbar based linear program solver using PDIP algorithm through effectively resolving the abovementioned issues.

For facilitating memristor-based implementations, linear equations in (4a) – (4b) can be rewritten as a linear system with $2(n+m)$ variables, as shown in (5):

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{A^T} & \boldsymbol{0} & \boldsymbol{-I} \\ \boldsymbol{Z} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{X} \\ \boldsymbol{0} & \boldsymbol{W} & \boldsymbol{Y} & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \Delta\boldsymbol{x} \\ \Delta\boldsymbol{y} \\ \Delta\boldsymbol{w} \\ \Delta\boldsymbol{z} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b - Ax - w} \\ \boldsymbol{c - A^T y + z} \\ \boldsymbol{\mu - XZ_e} \\ \boldsymbol{\mu - YW_e} \end{bmatrix} \tag{5}$$

where $\boldsymbol{I}$ represents the identity matrix with diagonal values equal to 1.

In order to make the matrix representable in memristor crossbar structure, new variables have to be introduced to eliminate negative elements. Consider a linear system $\boldsymbol{Ax = b}$, in which $\boldsymbol{A_{i,j}}$ is negative element. It can be transformed into a nonnegative matrix by introducing a compensation variable $x_c = -x_j$. Hence, the above linear system is equivalent to:

$$\begin{bmatrix} \boldsymbol{A_{1,1}} & \dots & \boldsymbol{A_{1,j}} & \dots & \boldsymbol{A_{1,n}} & \boldsymbol{0} \\ \vdots & \dots & \dots & \dots & \vdots & \boldsymbol{0} \\ \boldsymbol{A_{i,1}} & \dots & \boldsymbol{0} & \dots & \boldsymbol{A_{i,n}} & \boldsymbol{-A_{i,j}} \\ \vdots & \dots & \dots & \dots & \vdots & \boldsymbol{0} \\ \boldsymbol{A_{n,1}} & \dots & \boldsymbol{A_{n,j}} & \dots & \boldsymbol{A_{n,n}} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{1} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{1} \end{bmatrix} \begin{bmatrix} \boldsymbol{x_1} \\ \vdots \\ \boldsymbol{x_j} \\ \vdots \\ \boldsymbol{x_n} \\ \boldsymbol{x_c} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b_1} \\ \vdots \\ \boldsymbol{b_j} \\ \vdots \\ \boldsymbol{b_n} \\ \boldsymbol{0} \end{bmatrix} \tag{6}$$

As shown in Eqn. (5), the matrix on left hand-side consists of a sub-matrix $\boldsymbol{-I}$ introduced by $\Delta\boldsymbol{z}$ in Eqn. (3b). A new variable vector, $\Delta\boldsymbol{v} = -\Delta\boldsymbol{z}$, has to be introduced. Besides, a compensation variable vector $\Delta\boldsymbol{u} = -\Delta\boldsymbol{w}$ is required for maintaining a square matrix. In addition, $\boldsymbol{A}$ and $\boldsymbol{A^T}$ are the only matrices that may contain negative elements. Processes like Eqn. (6) are needed to eliminate all negative elements in $\boldsymbol{A}$ and $\boldsymbol{A^T}$. Therefore we have:

$$\begin{bmatrix} \boldsymbol{A'} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \text{ or } \boldsymbol{A''} \\ \boldsymbol{0} & \boldsymbol{A^{T\prime}} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} \text{ or } \boldsymbol{A^{T\prime\prime}} \\ \boldsymbol{Z} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{X} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{W} & \boldsymbol{Y} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} \text{ or } \boldsymbol{A'} & \boldsymbol{0} \text{ or } \boldsymbol{A^{TI}} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \text{ or } \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \Delta\boldsymbol{x} \\ \Delta\boldsymbol{y} \\ \Delta\boldsymbol{w} \\ \Delta\boldsymbol{z} \\ \Delta\boldsymbol{u} \\ \Delta\boldsymbol{v} \\ \Delta\boldsymbol{p} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b - Ax - w} \\ \boldsymbol{c - A^T y + z} \\ \boldsymbol{\mu - XZ_e} \\ \boldsymbol{\mu - YW_e} \\ \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{0} \end{bmatrix} \tag{7a}$$

where $\Delta\boldsymbol{p}$ comprises $\Delta p_i = \begin{cases} -\Delta x_j \text{ if } A_{\alpha,j} < 0 \text{ for some } \alpha \\ -\Delta y_k \text{ if } A^T_{\beta,k} < 0 \text{ for some } \beta \end{cases}$. $\boldsymbol{A'}$ and $\boldsymbol{A^{T\prime}}$ are matrices that change the negative elements in $\boldsymbol{A}$ and $\boldsymbol{A^T}$ to zero. $\boldsymbol{A''}$ and $\boldsymbol{A^{T\prime\prime}}$ are matrices whose elements are the absolute values of negative elements in $\boldsymbol{A}$ and $\boldsymbol{A^T}$. $\boldsymbol{A^I}$ and $\boldsymbol{A^{TI}}$ are matrices consisting of 1 and 0's. Locations of 1's depend on the locations of negative elements in $\boldsymbol{A}$ and $\boldsymbol{A^T}$ (please refer to Eqn. (6) as an example).

The above equation can be denoted as:

$$\boldsymbol{M\Delta s = r} \tag{7b}$$

where $\boldsymbol{M}$ can be implemented and variable vector $\Delta\boldsymbol{s}$ can be derived using memristor crossbar.

In the PDIP algorithm, once $\Delta\boldsymbol{x}$, $\Delta\boldsymbol{y}$, $\Delta\boldsymbol{w}$, $\Delta\boldsymbol{z}$ (all in the derived vector $\Delta\boldsymbol{s}$) are derived, we will update $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{w}$, $\boldsymbol{z}$, which can be performed using summing amplifiers. We will further

update the left-hand side matrix $\boldsymbol{M}$ and the right hand-side vector $\boldsymbol{r}$ of Eqn. (7b). Updating matrix $\boldsymbol{M}$ is relatively straightforward since we only need to update $\boldsymbol{X}$, $\boldsymbol{Y}$, $\boldsymbol{Z}$, and $\boldsymbol{W}$ in $\boldsymbol{M}$, using the memristor writing technology as shall be discussed in part C. On the other hand, $\boldsymbol{r}$ can be viewed as the difference of two vectors:

$$\boldsymbol{r} = \begin{bmatrix} \boldsymbol{b} - \boldsymbol{Ax} - \boldsymbol{w} \\ \boldsymbol{c} - \boldsymbol{A^T y} + \boldsymbol{z} \\ \boldsymbol{\mu} - \boldsymbol{XZ_e} \\ \boldsymbol{\mu} - \boldsymbol{YW_e} \\ \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{0} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b} \\ \boldsymbol{c} \\ \boldsymbol{\mu} \\ \boldsymbol{\mu} \\ \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{0} \end{bmatrix} - \begin{bmatrix} \boldsymbol{Ax} + \boldsymbol{w} \\ \boldsymbol{A^T y} - \boldsymbol{z} \\ \boldsymbol{XZ_e} \\ \boldsymbol{YW_e} \\ \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{0} \end{bmatrix} \tag{8a}$$

The subtraction could be implemented using summing amplifiers [3]. Next, we will discuss the calculation of the last vector in Eqn. (8a). Note that

$$\boldsymbol{M}\begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ \boldsymbol{w} \\ \boldsymbol{z} \\ \boldsymbol{u} \\ \boldsymbol{v} \\ \boldsymbol{p} \end{bmatrix} = \begin{bmatrix} \boldsymbol{A'} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \text{ or } \boldsymbol{A''} \\ \boldsymbol{0} & \boldsymbol{A^{T\prime}} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} \text{ or } \boldsymbol{A^{T\prime\prime}} \\ \boldsymbol{Z} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{X} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{W} & \boldsymbol{Y} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} \text{ or } \boldsymbol{A^I} & \boldsymbol{0} \text{ or } \boldsymbol{A^{TI}} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \text{ or } \boldsymbol{I} \end{bmatrix}\begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ \boldsymbol{w} \\ \boldsymbol{z} \\ \boldsymbol{u} \\ \boldsymbol{v} \\ \boldsymbol{p} \end{bmatrix} = \begin{bmatrix} \boldsymbol{Ax} + \boldsymbol{w} \\ \boldsymbol{A^T y} - \boldsymbol{z} \\ \boldsymbol{2XZ_e} \\ \boldsymbol{2YW_e} \\ \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{0} \end{bmatrix} \tag{8b}$$

where $\boldsymbol{u} = -\boldsymbol{w}$, $\boldsymbol{v} = -\boldsymbol{z}$, and $\boldsymbol{p}$ consists of elements whose value are negative of some elements of $\boldsymbol{x}$ and $\boldsymbol{y}$, depending on the location of negative elements in $\boldsymbol{A}$ and $\boldsymbol{A^T}$. The result of Eqn. (8b) is only slightly different from the last vector in Eqn. (8a) on the 3rd and 4th elements. Since the matrix-vector product in memristor crossbar is represented as voltage, we can first calculate (8b) by performing matrix-vector multiplication using the updated memristor crossbar $\boldsymbol{M}$, and then acquire the last vector in Eqn. (8a), using a simple dividing-by-2 procedure on corresponding elements. $\boldsymbol{r}$ can be updated accordingly.

Our proposed memristor crossbar-based linear program solver is summarized as follows:

Algorithm 1: Memristor Crossbar-based Linear Program Solver

Input: Matrix $\boldsymbol{A}$, vectors $\boldsymbol{b}$, $\boldsymbol{c}$, constant ε_b, ε_c, μ, θ

Output: Vector $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{w}$, $\boldsymbol{z}$

Initialize $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{w}$, $\boldsymbol{z}$ with an arbitrary guess.

While $\boldsymbol{Ax} + \boldsymbol{w} - \boldsymbol{b} \succ \varepsilon_b$ or $\boldsymbol{A^T y} - \boldsymbol{z} - \boldsymbol{c} \succ \varepsilon_c$:

- Update matrix $\boldsymbol{M}$ in (7b) in memristor crossbar and vectors $\boldsymbol{p}$, $\boldsymbol{u}$, $\boldsymbol{v}$ in (8b) based on $\boldsymbol{A}$, $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{w}$, $\boldsymbol{z}$.
- Derive $\boldsymbol{r}$ based on (8a) and (8b) using memristor crossbar.
- Solve $\boldsymbol{M}\Delta\boldsymbol{s} = \boldsymbol{r}$ using memristor crossbar.
- Update $\boldsymbol{s} = \boldsymbol{s} + \theta\Delta\boldsymbol{s}$.

End

Return $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}, \boldsymbol{z}$.

C. Writing Coefficients in Matrices

The analog computation requires that memristor arrays (e.g., matrix $\boldsymbol{M}$ in (7b)) be programmed prior to execution (solving linear program), and be updated in each iteration during execution. Modifying the resistance of a memristor device can be achieved by applying V_{dd} or $-V_{dd}$ (satisfying $|V_{dd}| > |V_{th}|$) to two terminals of the memristor device [1][2][9]. In a memristor crossbar, the voltage difference V_{dd} is applied on the corresponding **WL** and **BL** that are connected to the target memristor device, whereas other **WL**s and **BL**s are biased by $V_{dd}/2$, which will have negligible effect on other memristor devices since $|V_{dd}/2| < |V_{th}|$ [2]. Programming a memristor device to a specific resistance is achieved by adjusting the amplitude and width of the write pulse (or the total number of write pulse spikes) [2][9]. The writing circuits of memristor crossbars and controlling circuits will be CMOS based.

D. Supporting Large-Scale Matrices

A memristor crossbar has limitation on its size due to manufacturing and performance considerations [13], which can potentially limit its scalability for large-scale and high-data rate applications. In order to overcome this shortcoming, motivated by [12], we adopt analog network-on-chip (NoC) communication structures that effectively coordinate multiple memristor crossbars for supporting large-scale applications. Data transfers within this NoC structure maintain analog form and are managed by the NoC arbiters.

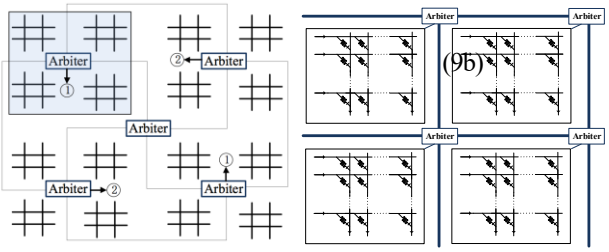


Fig. 2. NOC Structure for Large Scale Computation

Fig. 2 (a) and (b) illustrate two potential analog NoC structures for multiple memristor crossbars. Fig. 2 (a) is a hierarchical structure of memristor crossbars, in which four crossbar arrays are grouped and controlled by one arbiter, and four such groups again form a higher-level group controlled by a higher-level arbiter. Fig. 2 (b) is a mesh network-based structure of memristor crossbars, which resembles the mesh network-based NoC structure in multi-core systems [13]. Analog buffer and switches [10][11] will be utilized (in the arbiters) for the proper operation of this structure. The controller of NoC structure will be implemented in CMOS circuits. The NoC structure in Fig. 2 (a) will adopt a centralized controller whereas that in Fig. 2 (b) could employ a distributed controller similar to mesh network-based NoC in multi-core systems [13].

In addition to the NoC structure, we also present an memristor-based linear program solver with enhanced scalability. They key motivation is to use an iterative process to reduce the required size of matrix $\boldsymbol{M}$ in (7b), thereby improving scalability. More specifically, we treat Eqns. (4a)- (4b) as two systems of linear equations:

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{A^T} \end{bmatrix}\begin{bmatrix} \Delta\boldsymbol{x} \\ \Delta\boldsymbol{y} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b} - \boldsymbol{Ax} - \boldsymbol{w} \\ \boldsymbol{c} - \boldsymbol{A^T y} + \boldsymbol{z} \end{bmatrix} \tag{9a}$$

$$\begin{bmatrix} \boldsymbol{X} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{Y} \end{bmatrix}\begin{bmatrix} \boldsymbol{\Delta z} \\ \boldsymbol{\Delta w} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\mu} - \boldsymbol{XZ_e} \\ \boldsymbol{\mu} - \boldsymbol{YW_e} \end{bmatrix} \tag{9b}$$

Unlike (7a) which solves all step direction vectors (i.e., $\Delta\boldsymbol{x}$, $\Delta\boldsymbol{y}$, $\Delta\boldsymbol{w}$, $\Delta\boldsymbol{z}$) as one linear system, the proposed iterative algorithm for large-scale operations updates the step directions in an iterative approach. While updating step directions for vector $\boldsymbol{x}, \boldsymbol{y}$, vectors $\boldsymbol{w}, \boldsymbol{z}$ are assumed to be fixed so that we only need to solve Eqn. (9a) using memristor crossbar. After updating $\boldsymbol{x}, \boldsymbol{y}$, we derive the step directions for vectors $\boldsymbol{w}, \boldsymbol{z}$ by solving (9b) using memristor crossbar.

However, the coefficient matrix in (9a) is singular if $\boldsymbol{A}$ is not a square matrix. That is, Eqn. (9a) has no solution. In order to make (9a) solvable, part of the zero elements needs to be transformed to nonzero elements while causing limited impact to solution. Hence, following change is made to (9a).

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{R_U} \\ \boldsymbol{R_L} & \boldsymbol{A^T} \end{bmatrix} \begin{bmatrix} \Delta \boldsymbol{x} \\ \Delta \boldsymbol{y} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b - Ax - w} \\ \boldsymbol{c - A^T y + z} \end{bmatrix} \quad (9c)$$

where $\boldsymbol{R_U}$ is a matrix whose upper right m by m sub-matrix is zero matrix and $\boldsymbol{R_L}$ is a matrix whose lower left n by n sub-matrix is zero matrix

If $n > m$, $\boldsymbol{R_L}$ is used to replace lower left zero elements, and if $m > n$, $\boldsymbol{R_U}$ is used to replace upper right zero elements. Random positive numbers that are less than a threshold value are used to construct $\boldsymbol{R_L}$ and $\boldsymbol{R_U}$. Process alike Eqn. (6) is still needed after this step, therefore, Eqn. (9a) is transformed into

$$\begin{bmatrix} \boldsymbol{A'} & \boldsymbol{R_U} & \boldsymbol{0} \text{ or } \boldsymbol{A''} \\ \boldsymbol{R_L} & \boldsymbol{A^{T\prime}} & \boldsymbol{0} \text{ or } \boldsymbol{A^{T\prime\prime}} \\ \boldsymbol{0} \text{ or } \boldsymbol{A^I} & \boldsymbol{0} \text{ or } \boldsymbol{A^{TI}} & \boldsymbol{0} \text{ or } \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \Delta \boldsymbol{x} \\ \Delta \boldsymbol{y} \\ \Delta \boldsymbol{p} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b - Ax - w} \\ \boldsymbol{c - A^T y + z} \\ \boldsymbol{0} \end{bmatrix} \quad (9d)$$

On the other hand, the right hand-side vectors of Eqns. (9a) and (9b) can be calculated as:

$$\begin{bmatrix} \boldsymbol{b - Ax - w} \\ \boldsymbol{c - A^T y + z} \\ \boldsymbol{0} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b - w} \\ \boldsymbol{c + z} \\ \boldsymbol{0} \end{bmatrix} - \begin{bmatrix} \boldsymbol{A'} & \boldsymbol{0} & \boldsymbol{0} \text{ or } \boldsymbol{A''} \\ \boldsymbol{0} & \boldsymbol{A^{T\prime}} & \boldsymbol{0} \text{ or } \boldsymbol{A^{T\prime\prime}} \\ \boldsymbol{0} \text{ or } \boldsymbol{A^I} & \boldsymbol{0} \text{ or } \boldsymbol{A^{TI}} & \boldsymbol{0} \text{ or } \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ \boldsymbol{p} \end{bmatrix} \quad (10a)$$

$$\begin{bmatrix} \boldsymbol{\mu - XZ_e} \\ \boldsymbol{\mu - YW_e} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\mu} \\ \boldsymbol{\mu} \end{bmatrix} - \begin{bmatrix} \boldsymbol{X} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{Y} \end{bmatrix} \begin{bmatrix} \boldsymbol{z} \\ \boldsymbol{w} \end{bmatrix} \quad (10b)$$

Details of the proposed iterative linear program solver for enhancing scalability are described as below:

Algorithm 2: Memristor Crossbar-based Linear Program Solver for Large-Scale Operations

Input: Matrix $\boldsymbol{A}$, vectors $\boldsymbol{b}$, $\boldsymbol{c}$, constant ε_b, ε_c, μ, θ

Output: Vector $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{w}$, z

Initialize $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{w}$, z with an arbitrary guess.

While $\boldsymbol{Ax + w - b} \succ \varepsilon_b$ or $\boldsymbol{A^T y - z - c} \succ \varepsilon_c$:

- Update coefficient matrix $\boldsymbol{M_1}$ in (9d) and vector $\boldsymbol{p}$ in (10a) based on $\boldsymbol{A}, \boldsymbol{x}, \boldsymbol{y}$.
- Calculate vector $\boldsymbol{r_1}$ based on $\boldsymbol{M_1}$ and $\boldsymbol{s_1}$ in (10a) using memristor crossbar, where $\boldsymbol{s_1} = [\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{p}]^T$.
- Solve $\boldsymbol{M_1 \Delta s_1} = \boldsymbol{r_1}$ using memristor crossbar.
- Update $\boldsymbol{s_1} = \boldsymbol{s_1} + \theta \Delta \boldsymbol{s_1}$
- Update coefficient matrix $\boldsymbol{M_2}$ in (9b). based on $\boldsymbol{x}, \boldsymbol{y}$
- Calculate vector $\boldsymbol{r_2}$ based on $\boldsymbol{M_2}$ and $\boldsymbol{s_2}$ in (10b) using memristor crossbar, where $\boldsymbol{s_2} = [\boldsymbol{w}, \boldsymbol{z}]^T$.
- Solve $\boldsymbol{M_2 \Delta s_2} = \boldsymbol{r_2}$ using memristor crossbar.
- Update $\boldsymbol{w}, \boldsymbol{z}$ with $\boldsymbol{s_2} = \boldsymbol{s_2} + \theta \Delta \boldsymbol{s_2}$.

End

Return $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}, \boldsymbol{z}$.

E. Algorithms Complexity Comparisons

Given the fact that iteration-exiting conditions are same in software-based PDIP algorithm and the proposed memristor crossbar-based solver, the difference in iteration times is minimal. For each iteration step in software-based PDIP algorithm, a set of $2(n+m)$ equations needs to be solved. Solving such linear system could require $O(N^3)$ time complexity with direct method such as *Gaussian Elimination method* or *LU-Decomposition*, and $O(N^2)$ for each iteration by using iterative method such as *Gauss-Seidel method* ($N = n + m$). For the proposed solver, complexity for updating $\boldsymbol{X}$, $\boldsymbol{Y}$, $\boldsymbol{W}$, $\boldsymbol{Z}$ in matrix $\boldsymbol{M}$ is $O(N)$ (please note that matrices $\boldsymbol{A}$ and $\boldsymbol{A^T}$ do not need updating), and solving linear system in Eqn. (7a) only costs $O(1)$ time complexity. That is, for each iteration the complexity for memristor crossbar-based linear program solver is $O(N)$, while software-based PDIP algorithm could cost at least pseudo-$O(N^2)$. As for memristor crossbar-based linear program solver for large-scale applications, complexity for updating $\boldsymbol{X}$, $\boldsymbol{Y}$ in matrix Eqn. (9b) is $O(N)$, and complexities for solving (9a) and (9b) on memristor crossbar are both $O(1)$. Hence, the time complexity for memristor crossbar-based linear program solver for large-scale applications is also $O(N)$ for each iteration step, and the overall complexity is pseudo-$O(N)$.

Note that the above analysis only applies for the iterative solution of linear programs. On the other hand, the initialization time complexity is $O(N^2)$ for dense matrices, and will be lower for sparse matrices that are common in linear programs.

IV. Experiments and Results

Our experiments based on memristor model from [22] show significant improvement in speed and energy efficiency of memristor crossbar based implementation. The estimated delay for solving linear programs ranges from 36μs if the number of variables is 100 to 490μs if the number of variables is 1024. This estimation is based on (i) actual simulation results indicating that it generally takes 9–12 iterations for convergence, and (ii) the amount of coefficients updating in each iteration is 4N where N is the number of optimization variables. A maximum of 7,960X estimated improvement in speed is achieved compared with PDIP algorithm implemented in MATLAB executed on an Intel I7 server (when the number of variables is 1024). This significant improvement is because of reduction in complexity and speedup due to dedicated hardware implementation. The maximum amount of energy reduction is 6.7×10^5X in this case, which is even more significant than the speedup because of the low power consumption of memristor crossbar.

Under ideal condition, matrix operations on memristor crossbar-based design should be accurate given Kirchhoff's law [19]. However, due to process variations, the actual memristance matrix of a memristor crossbar may be different from the theoretical values. Because the impact of process variations is too complex to be expressed by a mathematical closed-form solution, we model it as a uniform distribution with a maximum range d_{max}.

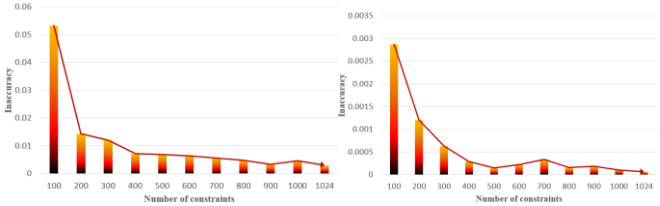
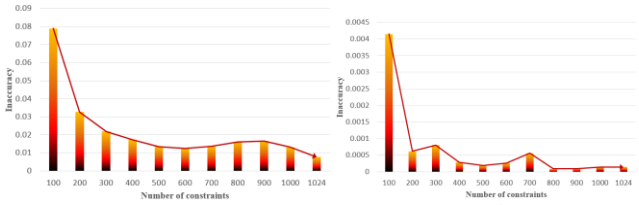


Fig. 3. (a) Accuracy simulation results of Memristor Crossbar-based Linear Program Solver for up to 5% process variations of each cell. (b) Accuracy simulation results of Memristor Crossbar-based Linear Program Solver for Large-Scale Operations for up to 5% process variations of each cell.

Both two algorithms (discussed in Section 3.B and 3.D) are given 1000 sets of tests with two different maximum ranges of process variations (5%, 10%). Each set of tests contains 100

randomly generated tests under the same matrix size and range of process variations. As for matrix size ($\boldsymbol{A} \in \mathbb{R}^{\boldsymbol{m}\times\boldsymbol{n}}$, m = number of constraints, n = number of variables), the number of constraints is three times of the number of variables. The number of constraints varies from 4 to 1024. Results obtained from memristor crossbar-based solver are compared with $\boldsymbol{c^T x}$ obtained from software-based PDIP algorithm, and inaccuracy is measured by difference in percentage. Above experiments are modeled and simulated in Matlab using memristor crossbar model [9]. Experiments results are shown in Fig. 3 and Fig. 4.

For d_{max}= 5%, the inaccuracy range is 0.2% to 5.3% for Memristor Crossbar-based Linear Program Solver and 0.01% to 0.2% for Memristor Crossbar-based Linear Program Solver for Large Scale Operation. For d_{max} = 10%, corresponding inaccuracy ranges are 0.7% to 7.8% and 0.01% to 0.4%. As shown in Fig. 3 and Fig. 4, inaccuracy decreases with increasing of numbers of constraints. Both implementations have shown reliable and accurate performance.



(a) (b)

Fig. 4. (a) Accuracy simulation results of Memristor Crossbar-based Linear Program Solver for up to 10% process variations of each cell. (b) Accuracy simulation results of Memristor Crossbar-based Linear Program Solver for Large Scale Operations for up to 10% process variations of each cell.

It can be observed that, inaccuracy drops significantly from the case where the number of constraints is less than 100 to the case where the number of constraints is less than 200. We believe that some singular matrices induced by process variations in the intermediate steps may cause such steep drop. While memristance is altered under the impact of process variation, its mapping matrix might be changed from a non-singular matrix to closer to a singular matrix (with determinant equal to 0), which could lead to zero solution or less accurate solution for the linear system. Since the coefficient size is relatively small, it could be more easily affected by some elements change and turn into a singular matrix.

Apart from singular matrix, matrix whose determinant is close to zero could be more vulnerable to process variations. Recall that each unknown in the solution of a linear system can be formulated as the division between determinants of a sub-matrix of coefficient matrix and coefficient matrix according to the Crammer's rule; the solution is inversely proportional to the determinant of coefficient matrix. Hence, matrices whose determinant values are close to zero could lead to massive change in values of solution under the impact of process variation. The accuracy for above two circumstances could be easily affected by process variation.

However, based on our randomly generated experiments, the above two circumstances are not common, and are very rare for large-scale matrices. With an average of 2% inaccuracy for 5% process variations and 4% for 10% process variations and an average of less than 0.005% inaccuracy for large-scale operations, memristor crossbar-based linear program solver using PDIP algorithm can provide very high accuracy with high energy/power efficiency.

V. Conclusion

This paper described the design of memristor crossbar-based linear program solver using primal-dual interior point algorithm. Two implementations using memristor crossbar have been presented for effectively trading-off between hardware complexity and computing speed. We also presented extension schemes to large-scale applications. Experimental results demonstrate reliable performance with high accuracy.

References

[1] A. Heittmann and T. G. Noll, "Limits of writing multivalued resistances in passive nano-electronic crossbars used in neuromorphic circuits," ACM Great Lakes Symposium on VLSI (GLSVLSI), 2012, pp. 227–232.

[2] D. Kadetotad, Z. Xu, A. Mohanty, P-Y. Chen, B. Lin, J. Ye, S. Vrudhula, S. Yu, Y. Cao, and J-S. Seo, "Neurophysics-inspired parallel architecture with resistive crosspoint array for dictionary learning," in IEEE Biomedical Circuits and Systems Conference (Bio-CAS) 2014, Lausanne, Switzerland.

[3] B. Liu, Y. Chen, B. Wysocki, and T. Huang, "The circuit realization of a neuromorphic computing system with memristor-based synapse design," in Neural Information Processing, 2012, pp. 357-365.

[4] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," Nature, vol. 453, pp. 80–83, 2008.

[5] Q. Xia, W. Robinett, M. W. Cumbie, N. Banerjee, T. J. Cardinali, J. J. Yang, W. Wu, X. Li, W. M. Tong, D. B. Strukov, G. S. Snider, G. Medeiros-Ribeiro, and R. S. Williams, "Memristor-CMOS hybrid integrated circuits for reconfigurable logic," Nano letters, vol. 9, no. 10, pp. 3640–3645, 2009.

[6] J. Liang, S. Yeh, S. S. Wong, and H.-S. P. Wong, "Effect of wordline/bitline scaling on the performance, energy consumption, and reliability of cross-point memory array," ACM Journal on Emerging Technologies in Computing Systems, vol. 9, no. 1, 2013.

[7] S. H. Jo, T. Chang, I. Ebong, B. B. Bhadviya, P. Mazumder, and W. Lu, "Nanoscale memristor device as synapse in neuromorphic systems," Nano letters, vol. 10, no. 4, pp. 1297–1301, 2010.

[8] M. Di Ventra, Y.V. Pershin and L.O. Chua, "Circuit elements with memory: memristors, memcapacitors, and meminductors," Proceedings of the IEEE, vol. 97, no. 10, pp. 1717–1724, 2009.

[9] M. Hu, H. Li, Y. Chen, G. Rose, and Q. Wu, "BSB Training Scheme Implementation on Memristor-Based Circuit," 2013 Symposium Series on Computational Intelligence, 2013.

[10] Analog Input Buffer Archiecture: https://www.cirrus.com/en/pubs/appNote/an241-1.pdf.

[11] J. Steensgaard, "Bootstrapped low-voltage analog switches," in International Symposium on Circuits and Systems (ISCAS), 1999.

[12] X. Liu, M. Mao, B. Liu, H. Li, Y. Chen, B. Li, Y. Wang, H. Jiang, M. Barnell, Q. Wu, and J. Yang, "RENO: a high-efficient reconfigurable neuromorphic computing accelerator design," in Proc. of Design Automation Conference (DAC), 2015.

[13] S. Vangal, J. Howard, G. Ruhl, and S. Dighe, "An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS," in IEEE International Solid-State Circuits Conference (ISSCC), 2007.

[14] Robere R.; 2012; "Interior Point Methods and Linear Programming"; University of Toronto.

[15] Robert O. Ferguson "Linear Programming," American Machinist, April 11, 1955, pp. 121-136.

[16] Vanderbei, R. J. (2001) Linear Programming, Foundations and Extensions (Kluwer Academic, Boston).

[17] L.Chua,"Memristor the missing circuit element," *IEEE Transactionon Circuit Theory*, vol. 18, pp. 507–519, 1971.

[18] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, pp. 80–83, 2008.

[19] Rak, A.; Cserey, G., "Macromodeling of the Memristor in SPICE," in Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on , vol.29, no.4, pp.632-636, April 2010

[20] Karmarkar N.K., An Interior Point Approach to NPComplete Problems Part I, AMS series on Contemporary Mathematics 114, pp. 297308.

[21] George B. Dantzig and Mukund N. Thapa. 1997. Linear programming 1: Introduction. Springer-Verlag

[22] Yakopcic, C.; Taha, T.M.; Hasan, R., "Hybrid crossbar architecture for a memristor based memory," in *Aerospace and Electronics Conference, NAECON 2014 - IEEE National* , vol., no., pp.237-242, 24-27 June 2014