Memristor-Based Spectral Decomposition of Matrices and Its Applications

Zeinab S. Jalali, Chenghong Wang, Griffin Kearney, Geng Yuan, Caiwen Ding, Yinan Zhou, Yanzhi Wang, *Member, IEEE*, and Sucheta Soundarajan

Abstract—The recently developed memristor technology allows for extremely fast implementation of a number of important matrix operations and algorithms. Moreover, the existence of fast matrix-vector operations offers the opportunity to design new matrix algorithms that exploit these operations. Here, we focus on the spectral decomposition of matrices, a task that plays an important role in a wide variety of applications from different engineering and scientific fields, including network science, control theory, advanced dynamics, and quantum mechanics. While there are a number of algorithms designed to find eigenvalues and eigenvectors of a matrix, these methods often suffer from poor running time performance. In this work, we present an algorithm for finding eigenvalues and eigenvectors that is designed to be used on memristor crossbar arrays. Although this algorithm can be implemented in a non-memristive system, its fast running time relies on the availability of extremely fast matrix-vector multiplication, as is offered by a memristor crossbar array. In this paper, we (1) show the running time improvements of existing eigendecomposition algorithms when matrix-vector multiplications are performed on a memristor crossbar array, and (2) present `EigSweep`, a novel, fully-parallel, fast and flexible eigendecomposition algorithm that gives an improvement in running time over traditional eigendecomposition algorithms when all are accelerated by a memristor crossbar. We discuss algorithmic aspects as well as hardware-related aspects of the implementation of `EigSweep`, and perform an extensive experimental analysis on real-world and synthetic matrices.

Index Terms—Spectral decomposition, memristors, eigen values and eigen vectors, symmetric matrices

1 Introduction

In this paper, we explore the use of *memristor* hardware to efficiently and accurately perform the spectral decomposition of real matrices. The problem of finding eigenpairs of matrices is central to a variety of machine learning tasks and other applications, including PCA, spectral clustering, PageRank, community detection, and many more [37]. However, on large matrices, traditional eigendecomposition methods can require many iterations, and can be prohibitively slow. It is thus of great interest to develop faster techniques for identifying the eigenpairs of large matrices. The development of memristor hardware technology makes available extremely fast, hardware-level matrix-vector operations, and thus affords an opportunity to fundamentally redesign eigendecomposition algorithms.

The memristor is a hardware component that functions similarly to a resistor, but with the ability to 'remember' the amount of current that last passed through it, and so has the property that its resistance can effectively be set in real-time [45]. Memristors can be wired together into a matrix known as a *crossbar array*, and by modifying the voltage across the rows and observing the output current, can be used to perform matrix-vector multiplication and solve a system of linear equations in O(1) time [26] (see Section 5 for details). Memristors have found application in areas such as deep learning, which require repeated fast matrix-vector multiplication [45].

However, although memristors offer the chance to design algorithms that would be impractical when implemented without memristor acceleration, there are a number of important considerations. Most importantly, memristors are subject to a number of accuracy-related errors, such as variations generated by the manufacturing process and hardware failures. Thus, any algorithm using memristor crossbars should be robust against such errors. The literature contains a number of techniques for modeling such faults in memristor crossbars [43], which can be used in the evaluation of algorithms using memristors.

In this paper, we first demonstrate that for traditional eigen-decomposition algorithms, on average, use of memristors decreases running time by 0.63x, while decreasing power and energy dissipation by 100x and 5x, respectively (Section 5). Next, we propose `EigSweep`, an eigen-decomposition technique for finding eigenpairs of real, symmetric matrices. `EigSweep` is based on our earlier work in [41],

- *Zeinab S. Jalali, Griffin Kearney, Yinan Zhou, and Sucheta Soundarajan are with the Department of Electrical Engineering & Computer Science, Syracuse University, Syracuse, NY 13244 USA. E-mail: {zsaghati, gmkearne, yzhou109, susounda}@syr.edu.*
- *Chenghong Wang is with the Department of Computer Science, Duke University, Durham, NC 27708 USA. E-mail: chenghong.wang552@duke.edu.*
- *Geng Yuan and Yanzhi Wang are with the Department of Electrical & Computer Engineering, Northeastern University, Boston, MA 02115 USA. E-mail: gmkearne@syr.edu, yanz.wang@northeastern.edu.*
- *Caiwen Ding is with the Department of Computer Science & Engineering, University of Connecticut, Storrs, CT 06269 USA. E-mail: caiwen.ding@uconn.edu.*

Manuscript received 17 August 2021; revised 25 July 2022; accepted 20 August 2022. Date of publication 29 August 2022; date of current version 7 April 2023. This work was supported by NSF under Grant #1637559.
(Corresponding author: Zeinab S. Jalali.)
Recommended for acceptance by M. T. Kandemir.
This article has supplementary downloadable material available at https://doi.org/10.1109/TC.2022.3202746, provided by the authors.
Digital Object Identifier no. 10.1109/TC.2022.3202746

which proposed a basic method for finding eigenvalues using memristors. Our contributions here are as follows:

1) We perform an extensive experimental evaluation of the performance of existing eigendecomposition algorithms when implemented on memristor crossbar arrays. This evaluation considers running time, power, energy, and error robustness. We show that memristor crossbars can be used to improve the performance of existing eigendecomposition algorithms. In our experiments, we obtain an average of 0.63x, 5x and 100x improvement in running time, power and energy consumption, respectively (Section 5), with less than a 5% drop in accuracy when accounting for variations and hardware failures. Our earlier work did not evaluate the use of memristor hardware for existing algorithms.
2) We present `EigSweep`, an approximate eigendecomposition algorithm for finding all (or a set of) eigenpairs. `EigSweep` takes advantage of the memristor crossbar array's ability to quickly solve systems of linear equations; and although it can be implemented without memristor acceleration, doing so would be impractical. On a memristor crossbar, this method has a time complexity of $O(k+n^2)$, where k is the number of iterations required by the algorithm and n is the size of the matrix. On certain important classes of matrices, such as adjacency matrices for unweighted graphs, k is bounded by $O(\frac{n}{\Delta})$, where Δ is a user-specified parameter governing accuracy (Section 6). `EigSweep` improves over our earlier work in the following ways:
   a) `EigSweep` simultaneously finds eigenvectors and eigenvalues, allowing for much improved efficiency.
   b) The basic sweeping method in [41] iterates over a search space using a fixed step size Δ. In contrast, `EigSweep` defines a narrower search space and uses a technique for dynamically updating Δ. This makes it at least 3 times faster in practice.
   c) `EigSweep` achieves a much higher accuracy by adding a verification step to reduce the probability of detecting false eigenvalues.
   d) While the basic sweeping algorithm from [41] was presented as a heuristic, we prove the correctness of `EigSweep` (Section 6).
3) We demonstrate how `EigSweep` can be used in a divide-and-conquer framework to find eigenpairs for matrices too large to fit on a memristor crossbar (Section 6).
4) We perform a comprehensive experimental analysis and demonstrate that `EigSweep` consistently achieves at least a 2x, 70x and 120x better performance in terms of running time, power and energy, respectively, as compared to the next spectral decomposition method, when both are implemented using memristor hardware (Section 7).
5) We analyze the performance of `EigSweep` when accounting for process variations and other errors associated with memristor hardware. We demonstrate that `EigSweep` finds 92% of eigenvalues with a relative error of less than 0.1, using a simulated device variation with 0 mean and 0.03 standard deviation; and finds 93% of eigenvalues with a relative error of less than 0.1 when accounting for a failure rate with probability 0.1. We have updated our simulation environment as compared to our earlier work. Moreover, our earlier work did not account for process variations and other errors associated with memristor hardware.
6) We compare `EigSweep` to existing eigendecomposition techniques on various important applications (Section 7), and show that the output of the PCA, Spectral Clustering, and SVD algorithms can be accurately computed by using `EigSweep`. Our earlier work did not consider applications of spectral decomposition.

The rest of this paper is organized as follows. In Section 2, a summary of related works is provided. In Section 3, we provide a discussion of the background on memristor hardware. In Section 4, we describe our simulation platform. Next, because most existing methods (including the power iteration, Lanczos, and QR decomposition techniques) consist of several iterations, each performing one or more matrix-vector multiplications, we show how these methods can be accelerated using memristor crossbar arrays. In Section 5, we show the improvements in running time of these methods obtained when implemented on a memristor crossbar array. In Section 6, we propose `EigSweep`, which is based on solving systems of linear equations. In each iteration, `EigSweep` solves a system of linear equations on a memristor crossbar. We perform an experimental analysis of `EigSweep`, including accounting for process variations and other errors associated with memristor hardware, in Section 7. Section 8 concludes the paper.

2 Related Work

This paper addresses the problem of finding eigenvalues and eigenvectors using memristor hardware. Matrix-vector multiplications are the core of most spectral decomposition algorithms, and while memristor crossbars can perform these very quickly (see Section 3), there has been very little work which examines the use of memristor hardware for this task.

Memristor crossbars have been used for other types of matrix operations. For example, memristor-based neural networks are now an important research area within computer science: Yuan et al. proposed a memristor-based framework by incorporating alternating direction method of multipliers into DNN training [45], Hu et al. introduced a Dot-Product Engine using memristors [18], and a memristor-based random number generator was proposed by Jiang et al. in [19].

However, few works use memristors to address the spectral decomposition problem. One of the few existing works is that by Liu et al. which introduced a generalization of the power iteration method on memristors [26]. In our earlier work, we proposed a novel memristor-based method for finding eigenpairs [41]. To the best of our knowledge, this paper together with [41] is the first work to propose a novel memristor-based eigenfinding method.

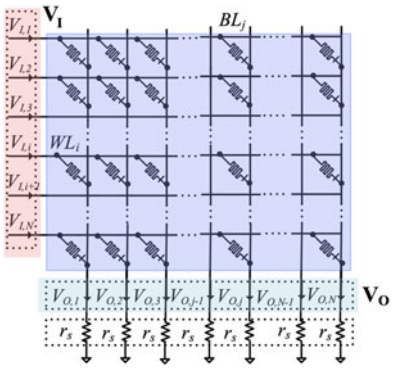

Fig. 1. Crossbar structure and matrix mapping [41].

While there are very few memristor-based techniques for finding eigenpairs, there exists a huge assortment of software-based methods. Some of these methods focus on finding the leading eigenpair of a matrix (e.g., the famous power iteration method and its variations [29]). Among methods to find all eigenpairs simultaneously, most notable are the Jacobi method, Rayleigh quotient, QR decomposition, and their variations [29]. As befits this problem's importance, there are also a number of algorithms that use hardware accelerators to improve running time. For example, Shi et al. and Li et al. use a FPGA accelerator for parallel computation of Jacobi eigensolving methods [25], and Sajjad et al. use a VLSI Architecture and Independent Component Analysis (ICA) technique to find eigendecompositions using the Jacobi method [35].

Most of the algorithms that exist for large symmetric matrices use the core of the QR decomposition technique: they reduce the symmetric matrix into a symmetric tridiagonal matrix, and then iteratively compute the eigenvalues of the tridiagonal matrix [39]. Different implementations of these methods, e.g., using GPUs or out-of-core methods like block algorithms have been proposed [9]. These techniques either parallelize the method, convert the problem to smaller sub-problems, or store the data structures on disks. Well-known methods that compute eigenpairs of a symmetric tridiagonal matrix include the QR algorithm and divide-and-conquer methods. For large matrices, divide-and-conquer methods are the fastest [28]. While various solvers can perform on distributed memory systems, only a few of them can fully use the potential of the modern accelerators like GPU, and so require additional performance tuning [9].

3 Background

Proposed by Leon Chua in 1971 and created by HP Labs in 2008, memristors are a type of electrical device that function similarly to resistors, but with the ability to 'remember' past voltage across their terminals [45]. Memristors possess a variety of important features, including scalability, non-volatility, high density and low power requirements. In general, a memristor functions like a resistor unless its terminals receive a voltage higher than some threshold: i.e., $|V_m| > |V_{th}|$. In this case, the state of the device changes, allowing, in effect, the resistance of the memristor to be modified in real time [45].

Memristor Crossbar Initialization. Of importance to this work is the observation that memristors can be connected together into a crossbar array, as shown in Fig. 1, which can then be used for fast matrix-vector multiplication and solving systems of linear equations. This figure illustrates a typical $N \times N$ memristor crossbar where each pair of vertical *bit-lines* (BL) and horizontal *word-lines* (WL) are connected. Implementing this structure is straightforward, and after making the crossbar, it can be programmed by applying biasing voltages at its two terminals to set it to different resistance. A memristor can be re-programmed at any time via this process [20].

By updating the conductance of the entire array fully in parallel, initialization of the crossbar (i.e., writing the resistances) can be done in $O(1)$ time [20]. Once the resistances have been set, the crossbar array can be designed to exhibit a unique type of parallelism in which matrix-vector multiplication and solving systems of linear equations can also be performed in $O(1)$ time complexity [45]. Thus, the overall time complexity for matrix-vector multiplication and linear equation solving has a running time complexity of $O(1)$.

Matrix Vector Multiplication. For performing a matrix-vector multiplication $\boldsymbol{y} = \boldsymbol{Ax}$, the matrix $\boldsymbol{A}$ is first mapped to the crossbar array. For each element in $\boldsymbol{A}$, we need to determine a corresponding conductance in the conductance array $\boldsymbol{G}$, which is then mapped onto the memristor crossbar array. To do so, all elements in $\boldsymbol{A}$ are divided by the maximum conductance value possible for the crossbar array (g_{max}) so that the elements of the matrix fall into the memristors' conductance range. Then, the values of $\boldsymbol{G}$ are determined according to the bias resistor's conductance g_s and $\boldsymbol{A}$ so that the following relationship holds:

$$\frac{\boldsymbol{A}}{g_{max}} = diag(d_1, \ldots, d_N) \cdot \boldsymbol{G^T}, \tag{1}$$

where $d_i = 1/(g_s + \sum_{k=1}^{N} g_{k,i})$ and each element g_{ij} of the conductance array $\boldsymbol{G}$ is the conductance between WL_i and BL_j. This mapping is done using accurate closed-loop tuning and utilizing access memresistances. r_{ij} is the memresistance for row i and column j, and $r_{ij} = \frac{1}{g_{ij}}$. Next, vector $\boldsymbol{x}$ is mapped to the vector of input voltages $\boldsymbol{V_I}$, and $\boldsymbol{V_I}$ is applied on the WLs. Finally, the output voltage on $\boldsymbol{V_O}$ is given by $\boldsymbol{V_O} = \frac{\boldsymbol{A}}{\boldsymbol{g_{max}}} \cdot \boldsymbol{V_I}$. Thus, matrix-vector multiplication can be performed by measuring the vector of output currents $\boldsymbol{V_O}$ from the bitlines. This can be done by measuring the voltage across resistor r_s and setting $\boldsymbol{y} = g_{max}\boldsymbol{V}_O$, where r_s is the sensing resistors used on all the BLs [26], [40], [41].

Solving Systems of Linear Equations. The memristor crossbar structure can also work in the opposite direction to solve a system of linear equations, as required by `EigSweep` [31]. This can be done by applying a voltage vector $\boldsymbol{V_O}$ to approximate the current flowing through each BL as $I_{o,j} = g_s V_{o,j}$ and calculating the current $I_{o,j}$ through BL_j as $I_{o,j} = \sum_j V_{I,i} g_{i,j}$. So, to find the solution $\boldsymbol{V_I}$, the equation $\frac{1}{g_s}\sum_j V_{I,i} g_{i,j} = V_{o,j}$ is mapped onto the crossbar for each BL_j and the voltages on the WLs are measured.

4 Experimental Platform

Due to immature manufacturing technology and the high cost of the memristor device, access to such devices is currently limited to major industry and academic labs. Because

TABLE 1
Algorithm Comparison

Method	Size	Time (nS)		Power (mW)		Energy (nJ)	
		SPMA	BSP	SPMA	BSP	SPMA	BSP
QR	2000	9.9 E+07	8.6 E+07	2.1E+08	1.6E+14	4.0E+07	4.6E+11
	5000	2.7 E+08	4.9 E+9	7.8E+09	4.0E+14	1.5E+09	1.2E+12
	10000	6.5 E+09	1.5 E+11	1.2E+11	8.1E+14	2.5E+10	2.3E+12
Shifted Power	2000	2.4 E+07	1.6 E+07	6.3E+07	1.9E+10	9.2E+06	7.3E+07
	5000	7.3 E+07	2.4 E+08	1.7E+09	3.5E+11	4.8E+07	1.8E+08
	10000	1.8 E+08	3.4 E+09	2.7E+10	5.7E+12	9.7E+07	3.6E+08
Lanczos	2000	2.6E+08	2.6E+08	4.3E+07	1.4E+10	5.1E+05	5.3E+07
	5000	1.5E+09	1.7E+09	9.7E+08	3.4E+10	1.8E+07	1.4E+08
	10000	6.3E+09	8.8E+09	1.5E+10	7.1E+10	2.9E+08	2.6E+08

of this, much research in this area is evaluated by emulation or simulation. For instance, experiments on a memristor-based neural network framework [45], a memristor-based dot-product engine [18], and a memristor-based random number generator [19] have been done on simulators.

The current state-of-the-art in memristor simulations is a collection of SPICE methods [5]. These simulations model the physics of complex circuits and, as such, are closer to realistic circuit conditions than other behavior simulation toolkits, such as on Matlab.

Simulation Platform With Memristor Accelerator (SPMA). In our experimental analysis, we use the NVSIM simulator [12] and the VTEAM memristor model [23] to model the power, area of the memristor crossbars, energy use, process variations, and other sources of errors. We use a crossbar size of 128×128 with 2-bit multi-level memristor cells. The hardware implementation is based on a 32-bit fixed-point representation. For the peripheral circuit, we follow the design in [36], which uses a 1-bit DAC and 8-bit ADC. We use Verilog HDL for modeling CMOS circuitry, including the controller and circuitry for arithmetic and logical computation units, and the Synopsys Design Compiler to synthesize them at 40 nm technology under a conservative 100 MHz clock frequency.

Baseline Simulation Platform (BSP). To compare the results of experiments when the algorithms are not accelerated by memristor hardware, we use the same Verilog HDL for modeling CMOS circuitry as used in the *SPMA*. In our baseline experiments, solving systems of linear equations and matrix-vector multiplications are not accelerated using memristor.

5 Accelerating Standard Spectral Decomposition Algorithms With Memristor Crossbars

In this section, we experimentally analyze the performance of several well-known eigendecomposition algorithms when implemented on a memristor crossbar array. As matrix-vector multiplication is the basis of many important eigendecomposition methods, a memristor crossbar is a natural complement for such algorithms. For instance, on an n-by-n matrix, the QR decomposition method has a software running time complexity bounded within $O(n^3)$, but can be accelerated to $O(n^2)$ with the help of a memristor crossbar device; and the original Lanczos algorithm has a software time complexity $O(dn^2)$ for finding d eigenpairs ($O(n^3)$ for finding all eigenpairs), which can be reduced to $O(dn)$ ($O(n^2)$) using a memristor crossbar device. Similarly, a major part of the power iteration algorithm is matrix-vector multiplication, and so as with the other methods, the method can be accelerated by implementing it on a memristor crossbar array [26].

To observe improvement in running times, power and energy obtained when implementing these eigendecomposition methods using a memristor crossbar, we simulate the QR decomposition, Lanczos, and shifted power iteration methods, both with and without memristive acceleration. To simulate memristive acceleration, we use the *SPMA* and *BSP* platforms as explained in Section 4. We apply these methods on random square matrices with positive and negative elements between $[-1, 1]$ ranging from sizes $n = 2000$ to 10,000. At each step of these algorithms, whenever a matrix-vector multiplication $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$ is needed, matrix $\boldsymbol{A}$ is mapped to the crossbar, and vector $\boldsymbol{x}$ is mapped as input. Matrix-vector multiplication is performed as explained in Section 3. Table 1 shows running time, power and energy results. The memristor has a tremendous effect on accelerating computing performance. On average, running time decreases by a factor of $0.63\times$ as compared to the case where memristors are not used, while power and energy decrease by $100\times$ and $5\times$, respectively.

When mapping elements to the memristor crossbar array, due to imperfections in fabrication technology, the mapped elements might slightly deviate from the original elements. We tested the accuracy degradation from variations caused by these imperfections. We introduced a device variation with a mean of 0 and standard deviation of 0.05. On average, the accuracy dropped by only 1%, 1% and 3% for Lanczos, shifted power iteration, and QR decomposition methods, respectively, as compared to the case where degradation is not considered. Moreover, we tested accuracy degradation caused by hardware failure with overall 5% failure rate. On average, the accuracy dropped just by 1%, 2% and 4% for Lanczos, shifted power iteration, and QR decomposition methods, respectively, as compared to a perfect memristor without such failures. Details of our accuracy degradation analysis are provided in Section 7.1. Note that the QR decomposition is the most accurate of these methods, and even with a 4% drop, it outperforms the others.

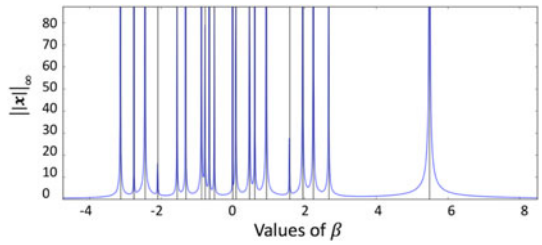

Fig. 2. Peak curve of the adjacency matrix for an Erdos-Renyi graph. Peaks correspond to eigenvalues. The x-axis represents values of β, and the y-axis shows $\|\boldsymbol{x}\|_\infty$, where $\boldsymbol{x}$ is the solution to $(\boldsymbol{A}-\beta\boldsymbol{I})\boldsymbol{x}=\boldsymbol{b}$ [41].

6 Proposed Method: EigSweep

While the previous section demonstrates that memristor hardware can be used to obtain significant speed-ups for standard spectral decomposition algorithm, further improvements can be obtained by designing algorithms specifically for such hardware: in particular, by exploiting a memristor crossbar array's ability to solve systems of linear equations in $O(1)$ time (including the time required to map the necessary values onto the crossbar array, using parallel writing), one can obtain even better performance. In this section, we present `EigSweep`, an algorithm that 'sweeps' across a range of values to identify eigenvalues and their associated eigenvectors using memristor.

6.1 Background

As background, we first summarize the basic sweeping method proposed in our earlier work [41]. This earlier algorithm is based on the following observation: Suppose that $\boldsymbol{A}$ is a non-negative, symmetric matrix, $\boldsymbol{b}$ is a vector of random numbers, and λ is an eigenvalue of $\boldsymbol{A}$. Then, as β approaches λ, with high probability, the absolute value of the maximum element in the solution $\boldsymbol{x}$ to $(\boldsymbol{A}-\beta\boldsymbol{I})\boldsymbol{x}=\boldsymbol{b}$ approaches ∞.

The described relationship is illustrated in Fig. 2, which shows a 'peak curve' for the adjacency matrix of an Erdos-Renyi graph with 20 nodes, where p (the probability that two nodes are connected) is set to 0.05. The x-axis represents values of β, and the y-axis shows $\|\boldsymbol{x}\|_\infty$, where $\boldsymbol{x}$ is the solution to $(\boldsymbol{A}-\beta\boldsymbol{I})\boldsymbol{x}=\boldsymbol{b}$. Vertical lines represent the locations of the eigenvalues of $\boldsymbol{A}$. (A theoretical explanation of this phenomenon can be found in Section 6.5.) This observation leads to the following algorithm for finding the eigenpairs of non-negative symmetric matrix $\boldsymbol{A}$ [41]:

1) Use Gershgorin's Circle Theorem to determine upper and lower bounds for $\boldsymbol{A}$'s eigenvalues [4].
2) Vary β from the upper bound to the lower bound using step size Δ. For each value of β, use the crossbar array to solve $(\boldsymbol{A}-\beta\boldsymbol{I})\boldsymbol{x}=\boldsymbol{b}$ (as explained in Section 3), where $\boldsymbol{b}$ is a vector of random positive values.
3) Identify those locations on the peak curve where $\|\boldsymbol{x}\|_\infty$ approaches an asymptote (i.e., a 'peak'), and set these β values aside as eigenvalues.
4) Use the inverse power method to find eigenvectors associated with eigenvalues. This method applies the power method to matrix $\mathbf{A}-\beta\mathbf{I}$ (where β is the estimated eigenvalue), to find the eigenvector of the dominant eigenvalue of the matrix [1].

6.2 Improvements Over Earlier Version

While the basic sweeping algorithm described was sufficient to explore a number of challenges associated with eigendecomposition using memristor crossbars, it suffered from several algorithmic shortcomings. We discuss `EigSweep` comprehensively in the next section, but first briefly describe improvements over the previous method.

6.2.1 Upper and Lower Bounds

The basic method uses the Gershgorin circle theorem for identifying the search space, which often gives a much larger space than is actually needed.

`EigSweep`, on the other hand, finds a narrower search space by using the power method to approximate the largest and smallest eigenvalues of $\boldsymbol{A}$. It estimates the largest eigenvalue β_{upper} of $\boldsymbol{A}$ by running the power method for a few iterations on matrix $\boldsymbol{A}$. It then uses the power method for a few iterations on matrix $\boldsymbol{A}^{-1}$ to approximate the largest eigenvalue of $\boldsymbol{A}^{-1}$ as λ_1'. The smallest eigenvalue of $\boldsymbol{A}$ is then estimated as $\beta_{lower}=\frac{1}{\lambda_1'}$.[1] `EigSweep` sets the search space as $[\beta_{upper}+\epsilon, \beta_{lower}-\epsilon]$. Padding these values by ϵ (which is a small number larger than Δ_{upper}, the maximum value for step size) ensures that it is not missing the largest or smallest eigenvalue.

For estimating the largest eigenvalue, `EigSweep` uses the crossbar to perform the matrix-vector multiplication required by the power iteration method. First, matrix $\boldsymbol{A}$ is mapped onto the crossbar and an initial vector $\boldsymbol{v}$ is applied as the input voltage. During the process, the crossbar remains the same, and the input vector changes based on the output of the matrix-vector multiplication done on the crossbar in the previous step. After a few steps, the upper-bound is estimated using the Rayleigh quotient. As `EigSweep` only needs an upper bound, it stops after a couple of iterations and does not continue until convergence. `EigSweep` estimates the smallest eigenvalue similarly by mapping $\boldsymbol{B}=\boldsymbol{A}^{-1}$ to the crossbar.

6.2.2 Step Size

The basic method sweeps between the upper bound and lower bound using a fixed step size Δ. The value of Δ is one of the biggest factors that influences running time, as it governs the number of iterations and it influences accuracy. A fixed step size is problematic because it must be small enough to detect nearby eigenvalues; but then this results in unnecessary computational effort in regions of the search space that are sparse in eigenvalues. In contrast, `EigSweep` uses a dynamic step size Δ_t, the Δ at step t, resulting in much faster running times while preserving accuracy. Informally, `EigSweep` sweeps across the specified interval with step size Δ_t to detect local maxima as eigenvalues. Ideally, Δ_t should be small in regions that are dense in eigenvalues, but large in regions with few eigenvalues.

While there is a substantial body of work examining the spacing between the top few eigenvalues of various types of matrices, there are few results regarding spacing between lower eigenvalues. To solve this problem, we use a technique

1. Note that if $\boldsymbol{A}$ is not invertible, adding a small amount of noise will make it invertible. While this might make the eigenvalue slightly inaccurate, as we only require an estimate, this approximation suffices.

for dynamically adjusting step size. The intuition behind this technique is based on the derivative test for peak finding, which locates the critical points of a function. For our method, it is sufficient to approximate whether we are close to a peak, as indicated by the rate of change of the slope of the curve as computed by the first derivative. To do that, at each step $t+1$, as we get closer to the peak, EigSweep sets Δ_{t+1} to a smaller value as compared to Δ_t. Conversely, as it goes farther from a peak, EigSweep sets Δ_{t+1} to a larger value as compared to Δ_t. Because $||\mathbf{x}_t||_\infty$ increases as it gets closer to the peak, we expect $||\mathbf{x}_{t+1}||_\infty - ||\mathbf{x}_t||_\infty$ to be larger than $||\mathbf{x}_t||_\infty - ||\mathbf{x}_{t-1}||_\infty$. Thus, EigSweep sets $\Delta_{t+1} = \Delta_t \cdot \frac{||\mathbf{x}_{t+1}||_\infty - ||\mathbf{x}_t||_\infty}{||\mathbf{x}_t||_\infty - ||\mathbf{x}_{t-1}||_\infty}$. If desired, one can bound Δ_{t+1} values to be in an interval $[\Delta_{upper}, \Delta_{lower}]$.

Lines 21 and 22 in Algorithm 1 describe the process of setting the dynamic step size. Note that this process does not take into account the sign of the slope: this is because it is positive on one side of the peak, and negative on the other, but both sides give an equally good approximation. If desired, the user can specify a minimum and maximum Δ_t value.

6.2.3 Detecting False Peaks

The basic sweeping method sets $\boldsymbol{b}$ to a vector of random positive numbers that might lead to detection of false peaks. To reduce the probability of detecting false peaks, $\boldsymbol{b}$ should be set more carefully. In particular, $\boldsymbol{b}$ should not be nearly-orthogonal to any column in matrix $\boldsymbol{A} - \beta_t \boldsymbol{I}$. If vector $\boldsymbol{b}$ is selected in a way so that for any column vector $\boldsymbol{a}_i^T$ in the matrix, $\boldsymbol{b} \cdot \boldsymbol{a}_i^T \neq 0$, then $\boldsymbol{b}$ is not orthogonal to the matrix. Thus, at each step, $\boldsymbol{b}$ can be selected in a way to satisfy this property. To avoid re-computing $\boldsymbol{b}$ as the matrix $\boldsymbol{A} - \beta_t \boldsymbol{I}$ changes, EigSweep sets $\boldsymbol{b}$ to a vector of co-prime numbers. In the case where $\boldsymbol{A}$ consists of integers (as is often common in real applications), this makes it highly unlikely that b is nearly-orthogonal to any column in $\boldsymbol{A} - \beta_t \boldsymbol{I}$. To further reduce the probability of this occurring, EigSweep verifies the existence of a peak by another vector orthogonal to $\boldsymbol{b}$. To do this, if EigSweep selects some β_t to be an eigenvalue candidate, it verifies it by generating a new vector $\boldsymbol{b}'$ orthogonal to $\boldsymbol{b}$, and repeats the process by solving $(\boldsymbol{A} - \beta_t \boldsymbol{I})\boldsymbol{x}'_t = \boldsymbol{b}'$.[2]

EigSweep achieves a much higher accuracy than the basic method by setting $\boldsymbol{b}$ to be a vector of co-prime numbers (instead of random numbers) and adding the verification step to reduce the probability of detecting false peaks.

6.2.4 Detecting Eigenvectors Separately

The basic sweeping method uses the inverse power method to find eigenvectors corresponding to detected eigenvalues. EigSweep eliminates this step by finding eigenvectors and eigenvalues at the same time. When $\boldsymbol{b}$ is set correctly, if β is detected as an eigenvalue, then the normalized vector x in the linear equation $(\boldsymbol{A} - \beta \boldsymbol{I})\boldsymbol{x} = \boldsymbol{b}$ is the corresponding eigenvector. By computing eigenvectors at the same time as eigenvalues, EigSweep eliminates the need to run inverse power method to find eigenvectors, reducing running time, and increasing accuracy.

2. As $\boldsymbol{b}$ and is a random co-prime vector and $\boldsymbol{b}'$ is orthogonal to $\boldsymbol{b}$, it is extremely unlikely that both will be nearly-orthogonal to some vector in $\boldsymbol{A} - \beta \boldsymbol{I}$, but to further reduce this possibility, one can repeat the process.

6.3 Proposed Method: EigSweep

Based on the improvements described in Section 6.2 we present an overview of EigSweep and show how EigSweep can be used to find all or the desired number of eigenpairs.

The input to the algorithm is: (1) A real, symmetric matrix $\boldsymbol{A}$, (2) The desired number of eigenvalues d (optional) and (3) The desired eigenvalue interval $[\beta_{upper}, \beta_{lower}]$ (optional). The output of the algorithm is the set of all eigenpairs, the desired number of eigenvalues, or the set of eigenpairs from the specified interval.

Algorithm 1 describes EigSweep in pseudocode. An overview of the algorithm is as follows:

EigSweep first defines $\boldsymbol{b}$ and $\boldsymbol{b}'$, where $\boldsymbol{b}$ is a vector of co-prime numbers of length equal to the number of rows of $\boldsymbol{A}$, and $\boldsymbol{b}'$ is a vector orthogonal to $\boldsymbol{b}$, of the same length. If a search interval is not given, EigSweep sets the search space $[\beta_{lower}, \beta_{upper}]$ as the interval between the largest and smallest eigenvalues of $\boldsymbol{A}$, as explained in Section 6.2.1, and $d = |\boldsymbol{A}|$ to find all eigenpairs (Line 1 in Algorithm 1). EigSweep sweeps across different β_t values, beginning with β_{upper} and ending at β_{lower}, with step size Δ_t. β_0 is equal to β_{upper}. By beginning at the upper bound, the algorithm will find the largest eigenvalues first, which are often the most important, allowing the user to terminate the method early if desired. The process consists of several iterations:

- In each iteration, EigSweep solves the linear equation $(\boldsymbol{A} - \beta_{t+1}\boldsymbol{I})\boldsymbol{x}_{t+1} = \boldsymbol{b}$ using the memristor crossbar array, where $(\boldsymbol{A} - \beta_{t+1}\boldsymbol{I})$ is mapped to the crossbar and $\boldsymbol{x}$ is set to the input voltage, as described in Section 3. The algorithm detects peaks by selecting those β_t values such that $\|\boldsymbol{x}_t\|_\infty$ is greater than both $\|\boldsymbol{x}_{t-1}\|_\infty$ and $\|\boldsymbol{x}_{t+1}\|_\infty$. These are noted as potential eigenvalues (Lines 7 to 9 in Algorithm 1).
- EigSweep confirms that a detected peak is an eigenvalue by solving three systems of linear equations using the memristor crossbar: (1) $(\boldsymbol{A} - \beta_{t-1}\boldsymbol{I})\boldsymbol{x}_{t-1} = \boldsymbol{b}'$, (2) $(\boldsymbol{A} - \beta_t \boldsymbol{I})\boldsymbol{x}_t = \boldsymbol{b}'$, and (3) $(\boldsymbol{A} - \beta_{t+1}\boldsymbol{I})\boldsymbol{x}_{t+1} = \boldsymbol{b}'$. EigSweep then checks whether $\|\boldsymbol{x}_t\|_\infty$ is greater than both $\|\boldsymbol{x}_{t-1}\|_\infty$ and $\|\boldsymbol{x}_{t+1}\|_\infty$. If β_t is still a local maxima, β_t is added to the set of eigenvalues (Lines 11 -12 in Algorithm 1).
- If β_t is identified as an eigenvalue, then both the normalization of $\boldsymbol{x}_t$ and $\boldsymbol{x}'_t$, computed from $\boldsymbol{b}$ and $\boldsymbol{b}'$, are approximate eigenvectors. To increase accuracy at minimal computational cost, $err = \boldsymbol{A}\boldsymbol{x}_t - \beta \boldsymbol{x}_t$ and $err' = \boldsymbol{A}\boldsymbol{x}'_t - \beta \boldsymbol{x}'_t$ are computed, and the vector with lower error is selected as an eigenvector, and added to the set of eigenvectors (Lines 13-18 in Algorithm 1).
- In each iteration, β_{t+1} is set to $\beta_t - \Delta_t$, where Δ_t is computed dynamically as explained in Section 6.2.2 (Lines 21-23 in Algorithm 1).
- EigSweep stops when d eigenpairs are computed or when β_t reaches the lower bound of the search space.

6.3.1 Implementation on Memristor Crossbar Array

In each iteration, EigSweep solves the linear equation $(\boldsymbol{A} - \beta \boldsymbol{I})\boldsymbol{x} = \boldsymbol{b}$. To implement this on the memristor crossbar array, EigSweep maps vector $\boldsymbol{b}$ and matrix $(\boldsymbol{A} - \beta \boldsymbol{I})$ onto the crossbar and uses the memristor crossbar to solve the linear

equation. For the first operation, EigSweep uses the parallel writing scheme described in [20] to map the matrix and vector in linear time, and then the linear equation is solved using the crossbar in $O(1)$ time, as described in Section 3.

The matrix $\boldsymbol{A} - \beta_t \boldsymbol{I}$ may contain negative elements, while the memristances on the crossbar array must be non-negative. To deal with this, the technique proposed in [20] is used to transform the system of linear equations (which possibly contains negative numbers) into a non-negative system.

Algorithm 1. EigSweep

Input:
Matrix $\boldsymbol{A}$, Number of desired eigen_pairs d (optional), Eigen space interval $[\beta_{upper}, \beta_{lower}]$ (optional)

Set $(\boldsymbol{b}, \boldsymbol{b'})$, $(\Delta_{upper}, \Delta_{lower})$, $(\beta_{upper}, \beta_{lower})$ (if needed)
$\boldsymbol{E_{val}} = \boldsymbol{E_{vec}} = Null$, $v_{-1} = d' = t = 0$, $d = |A|$
$\beta_0 = \beta_{upper}$, $\Delta_0 = \Delta_{lower}$, $\beta_1 = \beta_0 - \Delta_0$
Solve linear equation $(\boldsymbol{A} - \beta_0 \boldsymbol{I})\boldsymbol{x}_0 = \boldsymbol{b}$ using memristor
$v_0 = \|\boldsymbol{x}_0\|_\infty$
while $d' < d$ or $\beta_t > \beta_{lower}$ **do**
  Solve linear equation $(\boldsymbol{A} - (\beta_{t+1})\boldsymbol{I})\boldsymbol{x}_{t+1} = \boldsymbol{b}$ using memristor
  $v_{t+1} = \|\boldsymbol{x}_{t+1}\|_\infty$
  if $v_{t-1} < v_t > v_{t+1}$ **then**
    Solve 3 linear equations using memristor:
    $(\boldsymbol{A} - (\beta_{t-1})\boldsymbol{I})\boldsymbol{x'}_{t-1} = \boldsymbol{b}$,
    $(\boldsymbol{A} - \beta_t \boldsymbol{I})\boldsymbol{x'}_t = \boldsymbol{b}$,
    $(\boldsymbol{A} - (\beta_{t+1})\boldsymbol{I})\boldsymbol{x'}_{t+1} = \boldsymbol{b}$
    if $\|\boldsymbol{x'}_{t-1}\|_\infty \leq \|\boldsymbol{x'}_t\|_\infty \geq \|\boldsymbol{x'}_{t+1}\|_\infty$ **then**
      $\boldsymbol{E_{val}} = \boldsymbol{E_{val}} \cup \beta_t$, $d' = d' + 1$
      $err = \boldsymbol{A}\boldsymbol{x}_t - (\beta_t)\boldsymbol{x_t}$, $err' = \boldsymbol{A}\boldsymbol{x'}_t - (\beta_t)\boldsymbol{x'}_t$
      if $err > err'$ **then**
        $\boldsymbol{E_{vec}} = \boldsymbol{E_{vec}} \cup \{\frac{\boldsymbol{x'}}{\|\boldsymbol{x'}\|_2}\}$, $v_{t+1} = \|\boldsymbol{x'}\|_\infty$
      else
        $\boldsymbol{E_{vec}} = \boldsymbol{E_{vec}} \cup \{\frac{\boldsymbol{x}}{\|\boldsymbol{x}\|_2}\}$
      end if
    end if
  end if
  $\Delta = \Delta_t \cdot \frac{v_t - v_{t-1}}{v_{t+1} - v_t}$.
  $\Delta_{t+1} = min(max(\Delta, \Delta_{lower}), \Delta_{upper})$
  $\beta_{t+2} = \beta_{t+1} - \Delta_{t+1}$, $t = t + 1$
end while
return $\boldsymbol{E_{val}} \& \boldsymbol{E_{vec}}$

6.3.2 Approximating a Desired Number of Eigenpairs

By setting d to the desired number of eigenpairs and setting the search interval to the desired search space, one can detect the desired number of eigenpairs in *any* given interval. For finding a set of leading eigenpairs, one can begin the search at the upper bound of the identified search space, and stop sweeping after finding the desired number of eigenpairs. For finding the smallest eigenvalues, one can use the same process, except reversing the direction of the sweep.

6.3.3 Detecting Repeating Eigenvalues

EigSweep computes a list of distinct eigenvalues of matrix $\boldsymbol{A}$, but will not know which have multiplicity greater than 1 (have multiple associated eigenvectors). If one is interested in finding all eigenvalues, the perturbation based technique proposed in [41] can be used.

This technique slightly perturbs matrix $\boldsymbol{A}$ and 'splits' those eigenvalues into multiple eigenvalues. To do so, a random matrix $\boldsymbol{B}$, the same size as $\boldsymbol{A}$, with small entries selected uniformly at random from the range $[-\Delta_{max}, \Delta_{max}]$ is generated, and a perturbation matrix $\boldsymbol{P} = \boldsymbol{A} + \boldsymbol{B}$ is computed. Denote the set of calculated eigenvalues of $\boldsymbol{A}$ as $\boldsymbol{s}_1$ and the set of calculated eigenvalues of $\boldsymbol{P}$ as $\boldsymbol{s}_2$. The algorithm maps each element in $\boldsymbol{s}_1$ to the closest element in $\boldsymbol{s}_2$. After mapping all elements in $\boldsymbol{s}_1$, there may exist some elements in $\boldsymbol{s}_2$ that remain unmatched. These elements correspond to eigenvalues of $\boldsymbol{A}$ with multiplicity greater than 1. Thus, at the end, for each of these unmatched elements, the algorithm finds the element in $\boldsymbol{s}_1$ closest to that unmatched element and considers the multiplicity of that eigenvalue to be more than 1.

6.4 Algorithmic Features

EigSweep has a number of appealing algorithmic features:

- Accuracy: by specifying the minimum search increment Δ, the user can directly control the tradeoff between accuracy and running time. If an application does not require high accuracy, results can be estimated very quickly.
- Number of eigenpairs: the user can specify the exact number of desired eigenpairs.
- Eigenpairs corresponding to an interval: the user can set the upper and lower bound of the search space and find only eigenpairs within that interval. This is important for applications such as estimating the rank of a matrix [11]. Using this property, one can parallelize the EigSweep computation process using k crossbars, where the search interval is divided into k sub-intervals and eigenpairs of each sub-intervals are detected separately on a crossbar.
- Find eigenvector(s) corresponding to a given eigenvalue: Given a particular eigenvalue, EigSweep can very quickly compute the corresponding eigenvector. Most methods cannot do this.
- Small eigenpair(s): By reversing the direction of the sweep, EigSweep can easily find the smallest eigenpairs of a matrix. Finding the small eigenvalues of a graph has applications in physics, mathematics, and computer science [21] including finding the matrix condition number [15], and obtaining a bipartite subgraph [2]. Finding small eigenvalues of a matrix using standard techniques is more time consuming than finding the large eigenvalues. EigSweep, by contrast, is capable of finding the small eigenvalues and their corresponding eigenvectors as quickly as the largest ones.
- Parallel Computation: Given k memristor arrays, one can simply divide the search interval into k sub-intervals, find eigenpairs for each sub-interval independently, and then merge the results (padding the sub-interval values slightly ensures that eigenvalues at the boundaries are not missed).

6.5 Correctness Analysis

In this section, we provide a formal analysis on the correctness of EigSweep. Theorem 1 shows that the method correctly

finds distinct approximate eigenvalues, and Theorem 2 shows that the proposed method finds distinct approximate eigenvectors. Complete proofs are provided in the Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2022.3202746.

Theorem 1. *Let $\boldsymbol{A}$ be a real symmetric $n \times n$ matrix, and let $\boldsymbol{b} \in \mathbb{R}^n$. Assume that $\{\beta_i\}$ is an infinite scalar sequence with $\lim_{i\to\infty} \beta_i = \lambda$. Assume that $\boldsymbol{A}_i = \boldsymbol{A} - \beta_i \boldsymbol{I}$ is invertible for all i, with corresponding sequence $\{\boldsymbol{x}_i\}$ defined such that $\boldsymbol{A}_i\boldsymbol{x}_i = \boldsymbol{b}$, and $\boldsymbol{b}$ is not orthogonal to the non-zero members of the null space of $\boldsymbol{A} - \lambda\boldsymbol{I}$. Then λ is an eigenvalue of $\boldsymbol{A}$ if and only if $\|\boldsymbol{x}_i\|_\infty \to \infty$.*

For our next theorem, recall that to find eigenvalues, `EigSweep` sweeps over the search space (β_i) and solves the linear equation $(\boldsymbol{A} - \beta_{t+1}\boldsymbol{I})\boldsymbol{x}_{t+1} = \boldsymbol{b}$. It detects peaks by checking $\|\boldsymbol{x}_t\|_\infty$ value. Detection of a peak is an indicator that β_i is approaching a point where $\|\boldsymbol{x}_i\|_\infty \to \infty$. `EigSweep` verifies the existence of a peak using another vector orthogonal to $\boldsymbol{b}$. If β_i is identified as an eigenvalue, `EigSweep` sets the normalization of $\boldsymbol{x}_i = (\boldsymbol{A} - \beta I)^{-1}\boldsymbol{b}$ as approximate eigenvector. This leads us to Theorem 2.

Theorem 2. *Let $\boldsymbol{A}$ be a real, symmetric $n \times n$ matrix, and let $b \in \mathbb{R}^n$. Assume that λ is an eigenvalue of $\boldsymbol{A}$. If b is not orthogonal to the eigenspace corresponding to λ ($E(\lambda)$), then*

$$\lim_{\beta\to\lambda} \frac{(\boldsymbol{A} - \beta I)^{-1}\boldsymbol{b}}{||(\boldsymbol{A} - \beta\boldsymbol{I})^{-1}\boldsymbol{b}||_2} \in E(\lambda). \tag{2}$$

6.6 Complexity Analysis

In this work, we assume the use of the fully-parallel writing scheme for writing or updating a crossbar in $O(1)$ time, as was proposed in 2014 and which has been used by many works in this area [20]. We also assume that linear equation solving has a time complexity of O(1), as explained in Section 3.

`EigSweep` consists of the following actions:(1) Map matrix $\boldsymbol{A}$ to the crossbar array. This takes $O(1)$ time using the parallel writing scheme. (2) Calculate the upper bound (β_{upper}) and lower bound (β_{lower}) for performing eigenvalue searching, using a few iterations of the power method. This takes $O(1)$ time, as the algorithm performs only a few (2-3) matrix-vector multiplications on the crossbar array by setting the input voltage, each taking $O(1)$ time. (3) Search the interval $[\beta_{upper}, \beta_{lower}]$ with step size Δ_i, which varies at each step i. Suppose that the algorithm performs k increments. In each iteration, the algorithm updates the diagonal elements of the matrix on the crossbar, which takes $O(1)$ time using the parallel writing scheme. The algorithm uses the memristor crossbar to solve the system of linear equations and compute vector x, which takes $O(1)$ time. Then, the algorithm uses the crossbar array to multiply x to x, with $O(1)$ time required to find the second norm of x. Moreover, the verification step and computation of step size also take $O(1)$ time. If we define the size of the search range to be $l = |\beta_{upper}, \beta_{lower}|$, then in the worst case, the number of iterations in this step is $k = O(\frac{l}{\Delta_{min}})$. Thus, the overall time complexity for 3 is $O(k + n^2)$. (4) Compute eigenvalues with multiplicity greater than one by computing the perturbed matrix $\boldsymbol{P}_{n\times n} = \boldsymbol{A}_{n\times n} + \boldsymbol{B}_{n\times n}$, where matrix $\boldsymbol{B}$ is a random matrix, which takes $O(n^2)$ time. `EigSweep` then uses the technique in Section 6.3.3 to identify eigenvalues corresponding to $\boldsymbol{P}$, incurring an additional cost of $O(k + n^2)$. Thus, the complexity for the whole process is $O(k + n^2)$.

6.7 Divide-and-Conquer for Large Matrices

Our discussion thus far has assumed that the input matrix $\boldsymbol{A}$ will fit onto a memristor crossbar array. While memristor technology is improving rapidly, and the sizes of feasible crossbar arrays are growing, this is still a major limitation. To overcome this problem, we use an existing divide-and-conquer (D & C) algorithm. The first D & C eigen-solver was proposed in [17], and this class of algorithm has become popular. Many existing libraries, including Elemental [30], LAPACK [33], and other libraries that use LAPACK like NumPy and SciPy [3], use D & C methods to perform eigen-solving tasks on large matrices. These techniques are competitive with the fastest non-D & C eigen-solving methods, like QR decomposition, in terms of efficiency and stability, with the obvious advantage of scalability. The most widely used D & C methods first convert the symmetric input matrix to a tridiagonal matrix [39]. In the D & C method used here, we first convert the input matrix $\boldsymbol{A}$ to a tridiagonal matrix $\boldsymbol{T}$ with the Householder Reduction method [14]. The Householder Reduction consists of $n - 2$ transformation steps. If we set $\mathbf{A}_0 = \mathbf{A}$, at each step i, Householder matrix $\mathbf{H}_i$ is computed using the standard Householder matrix computation process [14] and is used for transformation of the matrix to a tridiagonal matrix using the $\mathbf{A}_i = \mathbf{A}_{i-1}\mathbf{H}_{i-1}\mathbf{A}_{i-1}$ formula. Matrix $\mathbf{A}_{n-2}$ is a tridiagonal matrix. Thus, the whole process needs $2 * (n - 2)$ matrix-matrix multiplications. As matrix-matrix multiplications can be treated as n matrix-vector multiplications, the transformation can be accelerated using the memristor crossbar. For that, using a divide-and-conquer technique, first matrix $\boldsymbol{A_i}$ and $\boldsymbol{H}_i$ are divided into sub-matrices that can be fit onto the crossbar, then each sub-matrix $\mathbf{A_i}$ is multiplied into sub-matrices of $\mathbf{H_i}$. We then use the ADC method proposed in [17] to convert the matrix $\boldsymbol{T}$ into an arrowhead matrix $\boldsymbol{A}'$ using D&C. This method recursively divides the input matrix into two sub-matrices until the sub-matrices are the desired size s (the size of the memristor crossbar array), and use `EigSweep` to find eigenpairs of these sub-matrices. We then merge the results by finding the eigenpairs of the arrow-head matrix $\boldsymbol{A}'$, using the Aheig method proposed in [38].

6.8 Limitations on Use

Memristors are new and extremely promising, but currently have important limitations that should be taken into consideration when designing algorithms:

First, there may be errors introduced by crossbars due to manufacturing variations, current sneak paths, or degrading memristances. Thus, when designing an algorithm that uses memristors, it is essential to evaluate its robustness against such noise. In Section 7.1, we analyse the degradation in accuracy while accounting for such errors.

Second, memristor crossbar sizes are quite limited. Current research on memristor crossbars assume sizes from

100×100 to 1000×1000 [22]. However, increasing these sizes is an active area of research. For example, [13] shows the trade off between increasing the size of the crossbar array up to 4 k and latency and energy. In Section 6.7 we showed that EigSweep can be used in a divide-and-conquer framework to find eigenpairs for matrices too large to fit on the crossbar.

Third, memristors suffer from a low switching endurance (lifetime number of cycles) [34]. Increasing this value is also an important area of research. For instance, the authors in [24] achieved over 10^{12} cycles using ta-oxide based devices, and the authors of [34] proposed a technique to make memristors more durable by providing the necessary fault tolerance.

7 Experiments

To evaluate the performance of EigSweep, we conduct an extensive analysis on a variety of matrices, including real-world graph adjacency matrices, a power network matrix, Laplacian matrices, and covariance matrices, as well as synthetic matrices. We perform a series of experimental studies and demonstrate that our solution achieves accuracies comparable to existing methods, while obtaining large improvements in running time.

Note that we do not consider cross-platform evaluations with other accelerating hardware such as GPUs. This is because recent studies have demonstrated that memristor crossbars are significantly more efficient than CPU or GPU-based multi-processing solutions in computing matrix-vector multiplications [8]. While one could use other forms of hardware accelerators for the other operations, these other operations form a very small part of the overall running time, and so the overall comparisons would not be affected.

In our experiments, we use real and synthetic random matrices. Our real matrices are derived from five datasets: US-Politics book network,[3] Squeak- Foundation mailing list network [27], Soc-Wiki-Vote social network [32], HB/bcspwr03 test case matrix [10] and a Power Grid infrastructure network [42]. The matrices are the binary, non-negative adjacency matrices of the listed networks. Statistics of the datasets are shown in Table 8 in the Appendix, available in the online supplemental material. For the PCA, spectral clustering and PageRank applications, we consider the covariance matrix, Laplacian matrix and transmission probability matrix of these networks respectively. Additionally, we generate random symmetric matrices with positive and negative elements of sizes 1,000 to 10,000, where each element of the matrix is a random value in the range $[-1, 1]$.

We conduct experiments in four phases. First, we test the accuracy of EigSweep when accounting for process variations and other errors associated with memristor hardware. Second, we compare the performance of EigSweep to three baseline methods in terms of running time, power and energy consumption. Third, we evaluate the performance of the D & C version of EigSweep and finally, we evaluate EigSweep with respect to four high-level spectral-related applications: PCA, spectral clustering, and condition number.

We set EigSweep's minimum step size (Δ) to 0.0001 and the maximum to 0.1. For the comparison baseline methods (such as QR decomposition), we set the number of iterations to be the smallest possible (i.e., fastest running time) while achieving the same accuracy as EigSweep.

7.1 Accuracy Analysis

In this section, we evaluate the accuracy of EigSweep both with and without the presence of accuracy-related errors and hardware failures in the memristor hardware, including variations generated by the manufacturing process.

First, to evaluate the accuracy of EigSweep without hardware errors, we use the built-in eigenpair computing functions in the NumPy library as the correct result. We calculate the relative error between the detected and correct eigenvalues (VAL_{err}) & eigenvectors (VAL_{vec}). First, note that EigSweep produces sets of eigenpairs with high accuracy. The average error is less than 0.01% for both eigenvalues and eigenvectors. Second, the accuracy increases as we focus our attention only on leading eigenpairs, which are more important for many applications (e.g., clustering, PCA). Table 7 in the Appendix, available in the online supplemental material, shows the full accuracy results on real matrices, with similar results for random matrices.

Next, we analyze the accuracy degradation caused by hardware variations and failures.

7.1.1 Variation Analysis

A main challenge in building accelerators is accounting for hardware variations [43]. Here, we evaluate the impact of variations caused by imperfections in fabrication technology and the noise of analog computing. We follow a widely used method to model the variation as a log-normal distribution [6]. In this experiment, we generate a log-normal distribution of size N^2 (the number of elements in matrix $\mathbf{A}$). This distribution has positive and negative numbers with mean 0. We use these values to represent the errors in mapping $\mathbf{A}$ to the crossbar array in our simulation. Table 2 shows the accuracy results on real matrices when introducing a device variation with mean 0 and standard deviations from 0.01 to 0.05. Naturally, the accuracy degrades compared to a perfect memristor (0 error). However, on average, 99%, 92% and 85% of the eigenvalues detected have relative error less than 0.1 for these variations, respectively. For random matrices, on average, 96% of the eigenvalues detected have relative error less than 0.1 for 0 mean and 0.05 standard deviation.

7.1.2 Failure Analysis

Stuck-at defects are a common hardware failure that can significantly degrade the accuracy of the applications of crossbars [44]. Therefore, we investigate the impact of stuck-at defects (i.e., stuck-on and stuck-off defects) on our design. Table 3 shows accuracy results on real matrices when introducing different failure rates. We adopt the widely used memristor failure model [7] with a 1:5.2 ratio of stuck-off and stuck-on defects. "prob" in the Table represents the overall failure rate (the percent of these BLs on the crossbar stuck on either 0 and 1, which can't be changed). On average, 98%, 95% and 93% of the eigenvalues detected

3. http://www-personal.umich.edu/~mejn/netdata/

TABLE 2
Percentage of Accurate Eigenvalues (With VAL_{err} Less Than Error Threshold) on Device Variation Under Lognormal Distribution With 0 Mean and Different Standard Deviations

Dataset	stdev	Error Threshold			
		0.1	0.2	0.3	0.4
US-Politics	**0.01**	99%	100%	100%	100%
	0.03	85%	91%	99%	100%
	0.05	77%	86%	92%	98%
Squeak	**0.01**	100%	100%	100%	100%
	0.03	98%	99%	100%	100%
	0.05	94%	97%	98%	100%
Power-Grid	**0.01**	98%	100%	100%	100%
	0.03	97%	100%	100%	100%
	0.05	93%	97%	98%	98%
Soc-Wiki-Vote	**0.01**	99%	100%	100%	100%
	0.03	95%	99%	100%	100%
	0.05	85%	94%	100%	100%
HB/bcspwr03	**0.01**	98%	100%	100%	100%
	0.03	84%	90%	94%	97%
	0.05	74%	80%	84%	88%

have relative error less than 0.1 for failure probabilities equal to 0.01, 0.05 and 0.10 respectively. For random matrices, on average, 96% of the eigenvalues detected have relative error less than 0.1 for 0.10 failure probability.

7.1.3 Perturbation Analysis

As discussed earlier, `EigSweep` uses a memristor crossbar to solve systems of linear equations at each step t, $((\boldsymbol{A} - (\beta_{t+1})\boldsymbol{I})\boldsymbol{x}_{t+1} = \boldsymbol{b})$. However, the memristor array only approximates the vector x. To evaluate the sensitivity of `EigSweep` to potential imprecision that may occur, in each step, a random vector (of the same size as x_t) with positive and negative elements with mean 0 and different standard deviation is added to x. Table 4 shows results of the perturbation analysis for different standard deviations. The results show that the accuracy does not degrade dramatically. On average, 99% of the eigenvalues detected have relative error less than 0.1 for variation under a log-normal distribution with 0 mean and various standard deviations. For random matrices, on average, 99% of the eigenvalues detected have relative error less than 0.1 for 0 mean and 0.05 standard deviation.

TABLE 3
Percentage of Accurate Eigenvalues (VAL_{err} Less Than Error Threshold) Under Different Failure Probabilities

Dataset	prob	Error Threshold			
		0.1	0.2	0.3	0.4
US-Politics	**0.01**	96%	99%	99%	100%
	0.05	88%	96%	98%	99%
	0.10	80%	92%	96%	98%
Squeak	**0.01**	100%	100%	100%	100%
	0.05	99%	100%	100%	100%
	0.10	99%	100%	100%	100%
Power-Grid	**0.01**	98%	99%	100%	100%
	0.05	98%	99%	100%	100%
	0.10	98%	99%	100%	100%
Soc-Wiki-Vote	**0.01**	99%	100%	100%	100%
	0.05	98%	100%	100%	100%
	0.10	96%	98%	99%	100%
HB/bcspwr03	**0.01**	97%	99%	100%	100%
	0.05	93%	98%	99%	100%
	0.10	90%	97%	99%	99%

TABLE 4
Percentage of Accurate Eigenvalues (With VAL_{err} Less Than Error Threshold) Under Different Perturbation

Dataset	stdev	Error Threshold			
		0.1	0.2	0.3	0.4
US-Politics	**0.01**	100%	100%	100%	100%
	0.03	100%	100%	100%	100%
	0.05	98%	99%	100%	100%
Squeak	**0.01**	100%	100%	100%	100%
	0.03	100%	100%	100%	100%
	0.05	100%	100%	100%	100%
Power-Grid	**0.01**	100%	100%	100%	100%
	0.03	100%	100%	100%	100%
	0.05	100%	100%	100%	100%
Soc-Wiki-Vote	**0.01**	99%	100%	100%	100%
	0.03	98%	100%	100%	100%
	0.05	98%	100%	100%	100%
HB/bcspwr03	**0.01**	97%	98%	100%	100%
	0.03	97%	98%	100%	100%
	0.05	97%	98%	99%	100%

7.2 Running Time Analysis

Next, we compare the running time, power, and energy of `EigSweep` and three popular eigen-finding algorithms: the QR decomposition (QR) method, Lanczos (LS) method, and shifted power iteration (SP) method, where all algorithms have been accelerated using the memristor crossbar array (that is, matrix-vector multiplications and solving of systems of linear equations are implemented on the crossbar using the *SPMA* platform). Fig. 3 shows results on random matrices of sizes up to 10000. Results for real datasets are similar. Across matrix sizes, `EigSweep` is consistently the fastest, and its power and energy consumption the lowest, even when all methods are accelerated by memristor crossbar. The power and energy results include the controller and circuitry for arithmetic and logical computation units, but exclude the DAC and ADC, since different methods share the same design. Across these matrices, `EigSweep` is at least 2x faster than the next method while its energy and power consumption are at least 120x and 70x better. Note also that the accuracy of LS and SP are not nearly as good as QR and `EigSweep`.

7.3 Divide-and-Conquer Evaluation

We compare the results of the D & C method using a crossbar of size 100×100 to those of the QR algorithm, which is

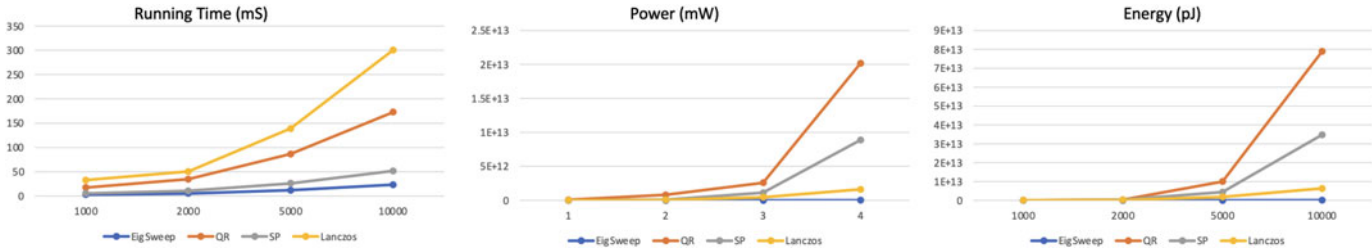

Fig. 3. Running time comparisons of EigSweep with other methods. EigSweep is consistently the fastest.

the most accurate among baseline methods. Because the QR method requires the user to specify the number of iterations (governing the accuracy and running time), for the sake of fairness, we perform two comparisons, one matching the running time of the D & C method, and the other matching its accuracy. In these experiments, both the D & C and QR methods are accelerated by memristor hardware. Table 5 shows the results. The D & C method is able to quickly compute all eigen-pairs, with less than 1% and 4% error for eigenvalues and eigenvectors respectively. The QR method takes a much longer time to achieve this accuracy, and when run for the same amount of time, achieves a much lower accuracy. When at the same accuracy, the D &C is at least $20x$ faster.

7.4 Applications

In the next experiments, we consider important data analysis algorithms that use matrix eigendecompositions: PCA, spectral clustering, and SVD. We again compare the results of EigSweep to those obtained by NumPy's built-in functions.

PCA. PCA is a technique that extracts the dominant information and patterns from a matrix in terms of a complementary set of scores, known as the principal components, which are used to reduce the dimensions of the dataset with minimal loss of information. PCA is the basis of many applications, including dimensionality reduction, image processing, object recognition, and Big Data classification. To implement PCA, we first compute a d-dimensional mean vector and the covariance matrix. Next, PCA uses the set of top-d eigenpairs to transform the samples onto the new subspace. Table 6 shows the $mean_{err}$ results of the PCA experiments. We set k to be 2%, 5% and $fe10\%$ of the total size of the matrices.

TABLE 5
Accuracy of D & C versus QR

Size	Method	$\mathbf{VAL_{err}}$	$\mathbf{VEC_{err}}$	Time
1000×1000	$D\&C$	0.01	1.23	t
	QR_{TM}	0.03	3.74	t
	QR_{AM}	0.02	1.24	60t
2000×2000	$D\&C$	0.02	3.14	t
	QR_{TM}	0.02	5.80	t
	QR_{AM}	0.01	3.36	100t
5000×5000	$D\&C$	0.29	3.71	t
	QR_{TM}	1.21	6.46	t
	QR_{AM}	0.21	3.13	20t

QR_{TM}: QR with the same running time as D&C, QR_{AM}: QR with the same accuracy as D & C. D & C with EigSweep achieves more accurate results with the same running time; or runs faster with the same accuracy.

$mean_{err}$ denotes the mean error between the elements of PCA using EigSweep and the corresponding PCA using NumPy's eigendecomposition results. The results show that the average error is less than 0.001%.

Spectral Clustering. Spectral clustering is a popular data clustering technique, used for applications such as image compression, graph partitioning, maximum cut, and many others. To implement spectral clustering for an $n \times n$ matrix, first the Laplacian matrix is computed, and the first d eigenvectors of the Laplacian matrix are computed. Next, the matrix containing these d eigenvectors is clustered into d clusters using k-means. By computing the top-d eigenvalues using EigSweep, we can implement spectral clustering on memristors. We compared the clusters found by EigSweep to those found by NumPy, setting the number of clusters to be 2%, 5%, and 10% of the number of rows, for the matrices in Table 8. We observe that on these matrices, EigSweep identified clusters identical to those found by NumPy.

SVD and Feature Selection. SVD is one of the most useful matrix factorizations and has many applications including feature selection. To select a set of k features among m input features from a dataset with n points, we first generate a graph where features are nodes and edges are weighted by similarity between features using a kernelized adjacency matrix [16]. After generating this graph, we compute the SVD of the adjacency matrix of the graph using the method described in [28] and use both EigSweep and Numpy to find eigenpairs as part of the process. We then compare the results. We compute the 2, 5, 10, and 20 top singular value decomposition features for each dataset and compute the cosine similarity between the singular values computed using EigSweep and the singular values computed using NumPy. The singular values computed from our method are identical to the baseline.

TABLE 6
$Mean_{err}$ Results of Comparing PCA as Performed by EigSweep versus NumPy Standard Library

Dataset	Number of components (as a % of matrix size)		
	1%	2%	3%
US-Politics	3.1E-5	2.4E-4	3.1E-4
Squeak	4.0E-4	5.1E-4	6.0E-4
Power-Grid	2.5E-4	4.8E-4	6.6E-4
Soc-Wiki-Vote	1.7E-4	3.7E-4	5.4E-4
HB/bcspwr03	3.7E-7	3.9E-6	4.3E-6

The number of components is considered to be 1-3 % of the number of rows (n). For all datasets, the $Mean_{err}$ is less than 0.001%.

8 Conclusion

In this paper, we first showed how memristor can be used to accelerate existing eigen-finding techniques. Use of memristors, on average, decreased running time, power, and energy consumption by 0.63x, 100x, and 5x, respectively with less than a 5% drop in accuracy. We then proposed `EigSweep`, a memristor-based algorithm for approximating eigenpairs of real symmetric matrices and showed how `EigSweep` can be used as part of a divide-and-conquer method for large matrices. We analyzed the robustness of `EigSweep` against device variations, hardware failures, and approximations, and showed that, on average, `EigSweep` can find 92% of eigenvalues with a relative error less than 0.1.

For the applications of PCA, feature detection and spectral clustering, we showed that PCA as computed by `EigSweep` achieves an error of less than 0.001%, and the clusters and features as computed by `EigSweep` are exactly the same as those computed by NumPy.

References

[1] E. Allen and R. Berry, "The inverse power method for calculation of multiplication factors," *Ann. Nucl. Energy*, vol. 29, no. 8, pp. 929–935, 2002.

[2] N. Alon and B. Sudakov, "Bipartite subgraphs and the smallest eigenvalue," *Combinatorics, Probability Comput.*, vol. 9, no. 1, pp. 1–12, 2000.

[3] C. Bauckhage, "NumPy/SciPy recipes for data science: Ordinary least squares optimization," Researchgate.Net, Mar. 2015.

[4] H. E. Bell, "Gershgorin's theorem and the zeros of polynomials," *Amer. Math. Monthly*, vol. 72, no. 3, pp. 292–295, 1965.

[5] D. Biolek, M. Di Ventra, and Y. V. Pershim, "Reliable spice simulations of memristors, memcapacitors and meminductors," *Radioengineering*, vol. 22, no. 4, p. 945, 2013.

[6] G. Charan et al., "Accurate inference with inaccurate RRAM devices: Statistical data, model transfer, and on-line adaptation," in *Proc. 57th ACM/IEEE Des. Automat. Conf.*, 2020, pp. 1–6.

[7] C. Chen et al., "RRAM defect modeling and failure analysis based on march test and a novel squeeze-search scheme," *IEEE Trans. Comput.*, vol. 64, no. 1, pp. 180–190, Jan. 2015.

[8] J. Cui and Q. Qiu, "Towards memristor based accelerator for sparse matrix vector multiplication," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2016, pp. 121–124.

[9] D. Davidović, "An overview of dense eigenvalue solvers for distributed memory systems," in *Proc. IEEE 44th Int. Conv. Inf., Commun. Electron. Technol.*, 2021, pp. 265–271.

[10] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, 2011, Art. no. 1.

[11] E. Di Napoli, E. Polizzi, and Y. Saad, "Efficient estimation of eigenvalue counts in an interval," *Numer. Linear Algebra Appl.*, vol. 23, no. 4, pp. 674–692, 2016.

[12] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 31, no. 7, pp. 994–1007, Jul. 2012.

[13] A. Dozortsev, I. Goldshtein, and S. Kvatinsky, "Analysis of the row grounding technique in a memristor-based crossbar array," *Int. J. Circuit Theory Appl.*, vol. 46, no. 1, pp. 122–137, 2018.

[14] A. A. Dubrulle, "Householder transformations revisited," *SIAM J. Matrix Anal. Appl.*, vol. 22, no. 1, pp. 33–40, 2000.

[15] A. Edelman, "Eigenvalues and condition numbers of random matrices," *SIAM J. Matrix Anal. Appl.*, vol. 9, no. 4, pp. 543–560, 1988.

[16] F. Fallucchi and F. M. Zanzottto, "Singular value decomposition for feature selection in taxonomy learning," in *Proc. Int. Conf. RANLP*, 2009, pp. 82–87.

[17] M. Gu and S. C. Eisenstat, "A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem," *SIAM J. Matrix Anal. Appl.*, vol. 16, no. 1, pp. 172–191, 1995.

[18] M. Hu et al., "Dot-product engine for neuromorphic computing: Programming 1t1m crossbar to accelerate matrix-vector multiplication," in *Proc. 53nd ACM/EDAC/IEEE Des. Autom. Conf.*, 2016, Art. no. 19.

[19] H. Jiang et al., "A novel true random number generator based on a stochastic diffusive memristor," *Nat. Commun.*, vol. 8, no. 1, 2017, Art. no. 882.

[20] D. Kadetotad et al., "Neurophysics-inspired parallel architecture with resistive crosspoint array for dictionary learning," in *Proc. IEEE Biomed. Circuits Syst. Conf.*, 2014, pp. 536–539.

[21] E. Katzav and I. P. Castillo, "Large deviations of the smallest eigenvalue of the wishart-laguerre ensemble," *Phys. Rev. E*, vol. 82, no. 4, 2010, Art. no. 040104.

[22] O. Krestinskaya, A. Irmanova, and A. P. James, "Memristive non-idealities: Is there any practical implications for designing neural network chips?," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2019, pp. 1–5.

[23] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny, "VTEAM: A general model for voltage-controlled memristors," *IEEE Trans. Circuits Syst. II: Exp. Briefs*, vol. 62, no. 8, pp. 786–790, Aug. 2015.

[24] M.-J. Lee et al., "A fast, high-endurance and scalable non-volatile memory device made from asymmetric ta 2 o 5- x/tao 2- x bilayer structures," *Nat. Mater.*, vol. 10, no. 8, 2011, Art. no. 625.

[25] Z. Li, W. Wang, R. Jiang, S. Ren, X. Wang, and C. Xue, "Hardware acceleration of MUSIC algorithm for sparse arrays and uniform linear arrays," *IEEE Trans. Circuits Syst. I: Reg. Papers*, vol. 69, no. 7, pp. 2941–2954, Jul. 2022.

[26] S. Liu, Y. Wang, M. Fardad, and P. K. Varshney, "A memristor-based optimization framework for artificial intelligence applications," *IEEE Circuits Syst. Mag.*, vol. 18, no. 1, pp. 29–44, Jan.–Mar. 2018.

[27] E. R. Murphy-Hill and A. P. Black, "Traits: Experience with a language feature," in *Proc. 19th Annu. ACM SIGPLAN Conf. Object-Oriented Program. Syst. Lang. Appl.*, 2004, pp. 275–282.

[28] Y. Nakatsukasa and N. J. Higham, "Stable and efficient spectral divide and conquer algorithms for the symmetric eigenvalue decomposition and the SVD," *SIAM J. Sci. Comput.*, vol. 35, no. 3, pp. A1325–A1349, 2013.

[29] M. Panju, "Iterative methods for computing eigenvalues and eigenvectors," 2011, *Waterloo Math. Rev.*, vol. 1, no. 1, pp. 9–18, 2011.

[30] J. Poulson, B. Marker, R. A. Van de Geijn, J. R. Hammond, and N. A. Romero, "Elemental: A new framework for distributed memory dense matrix computations," *ACM Trans. Math. Softw.*, vol. 39, no. 2, 2013, Art. no. 13.

[31] I. Richter et al., "Memristive accelerator for extreme scale linear solvers," in *Proc. GOMACTech*, 2015, pp. 1–4.

[32] R. Rossi and N. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proc. 29th AAAI Conf. Artif. Intell.*, 2015, pp. 4292–4293.

[33] Y. Saad, *Numerical Methods for Large Eigenvalue Problems: Revised Edition*, vol. 66. Philadelphia, PA, USA: SIAM, 2011.

[34] H. Saadeldeen et al., "Memristors for neural branch prediction: A case study in strict latency and write endurance challenges," in *Proc. ACM Int. Conf. Comput. Front.*, 2013, Art. no. 26.

[35] M. Sajjad, M. Z. Yusoff, N. Yahya, and A. S. Haider, "An efficient VLSI architecture for FastICA by using the algebraic jacobi method for EVD," *IEEE Access*, vol. 9, pp. 58 287–58 305, 2021.

[36] A. Shafiee et al., "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Architecture*, 2016, pp. 14–26.

[37] S. Si, D. Shin, I. S. Dhillon, and B. N. Parlett, "Multi-scale spectral decomposition of massive graphs," in *Proc. 27th Int. Conf. Neural Inf. Process. Syst.*, 2014, pp. 2798–2806.

[38] N. J. Stor, I. Slapničar, and J. L. Barlow, "Accurate eigenvalue decomposition of real symmetric arrowhead matrices and applications," *Linear Algebra Appl.*, vol. 464, pp. 62–89, 2015.

[39] S. Toledo, "A survey of out-of-core algorithms in numerical linear algebra," *External Memory Algorithms Vis.*, vol. 50, pp. 161–179, 1999.

[40] N. Uysal, B. Zhang, S. K. Jha, and R. Ewetz, "XMAP: Programming memristor crossbars for analog matrix–vector multiplication: Toward high precision using representable matrices," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 41, no. 6, pp. 1827–1841, Jun. 2022.

[41] C. Wang, Z. S. Jalali, C. Ding, Y. Wang, and S. Soundarajan, "A fast and effective memristor-based method for finding approximate eigenvalues and eigenvectors of non-negative matrices," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, 2018, pp. 563–568.

[42] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world'networks," *Nature*, vol. 393, no. 6684, 1998, Art. no. 440.
[43] G. Yuan et al., "Forms: Fine-grained polarized ReRAM-based in-situ computation for mixed-signal DNN accelerator," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Architecture*, 2021, pp. 265–278.
[44] G. Yuan et al., "Improving DNN fault tolerance using weight pruning and differential crossbar mapping for ReRAM-based edge AI," in *Proc. IEEE 22nd Int. Symp. Qual. Electron. Des.*, 2021, pp. 135–141.
[45] G. Yuan et al., "An ultra-efficient memristor-based DNN framework with structured weight pruning and quantization using ADMM," in *Proc. IEEE/ACM Int. Symp. Low Power Electron. Des.*, 2019, pp. 1–6.

Zeinab S. Jalali recieved the MS degree of computer engineering from the Amirkabir University of Technology, in 2015. She is currently working toward the PhD degree with the Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, New York supervised by Dr. Sucheta Soundarajan. Her research interest include algorithmic design, social network analysis, fairness in artificial intelligence, and machine learning.

Chenghong Wang received the BS degree in information security from Harbin Engineering University, and the MS degree in computer science from Syracuse University. He is currently working toward the fourth-year PhD degree in computer science with Duke University under the supervision of Dr. Ashwin Machanavajjhala and Dr. Kartik Nayak. His primary research interests lie in the area of differential privacy, applied cryptography, secure computing, and database security.

Griffin Kearney works as a professional systems engineer while continuing progress towards the completion of his doctoral thesis in Electrical and Computer Engineering part time with Syracuse University. His research and professional focus are on optimal control systems, modern radar signal processing, and network system design, analysis, and optimization. He utilizes a strong foundation in mathematics including optimization theory, modern algebra, functional analysis, and linear systems theory to guide his approach to solving the novel challenges he is concerned with in both his academic and professional endeavors.

Geng Yuan received the MS degree of electrical engineering from Syracuse University, in 2016. He is currently working toward the PhD degree with the Department of Electrical and Computer Engineering, Northeastern University supervised by Dr. Yanzhi Wang. His research interests include high-performance and energy-efficient deep learning system design, computer vision, deep learning architecture design with emerging technologies, etc.

Caiwen Ding received the PhD degree in computer science and engineering from the Northeastern University, Boston, MA, in 2019. He is currently an assistant professor with the Department of Computer Science and Engineering, University of Connecticut, Storrs, CT. His research interets include machine learning and deep neural network systems, computer vision and natural language processing. He is the recipient of the Best Paper Award Nomination with DATE 2018 and DATE 2021.

Yinan Zhou received the bachelor's degree in computer science from Syracuse University, in 2019. He is a software engineer with Google working on user identity infrastructure. His research interest includes computer network security and distributed computing.

Yanzhi Wang (Member IEEE) received the BS degree from Tsinghua University, in 2009, and the PhD degree from the University of Southern California, in 2014. He is currently an assistant professor with the Department of ECE, Northeastern University. His research interests focus on model compression and platform-specific acceleration of deep learning applications. His work has been published broadly in top conference and journal venues , and has been cited above 9,700 times. He has received six best paper and top paper awards, one Communications of ACM Featured Article, another 11 Best Paper Nominations and four popular paper awards. He has received the U.S. Army Young Investigator Program Award (YIP), Massachusetts Acorn Innovation Award, IEEE TSCDM Early Career Award, Ming Hsieh Scholar Award, a number of design contest awards at premier conferences, and other research awards.

Sucheta Soundarajan received the BS degree in mathematics and computer/information science from The Ohio State University in Columbus, Ohio, in 2005, and the PhD degree in computer science from Cornell University in Ithaca, New York, in 2013. She was a postdoctoral associate with the Department of Computer Science, Rutgers University in New Brunswick, New Jersey from 2013 to 2015. She is currently an associate with the Department of Electrical Engineering and Computer Science, Syracuse University in Syracuse, New York. Her research interests include algorithm development and data mining, with a focus on complex networks and large matrices.