An Analog Preconditioner for Solving Linear Systems

Ben Feinberg*, Ryan Wong†, T. Patrick Xiao *, Christopher H. Bennett *, Jacob N. Rohan ‡,
Erik G. Boman *, Matthew J. Marinella *, Sapan Agarwal *, Engin Ipek §
*Sandia National Laboratories
†University of Rochester
‡University of Texas at Austin
§Qualcomm Inc
*{bfeinbe, txiao, cbennet, egboman, mmarine, sagarwa}@sandia.gov, †rwong5@ece.rochester.edu, ‡jacobnrohan@utexas.edu

***Abstract*—Over the past decade as Moore's Law has slowed, the need for new forms of computation that can provide sustainable performance improvements has risen. A new method, called in situ computing, has shown great potential to accelerate matrix vector multiplication (MVM), an important kernel for a diverse range of applications from neural networks to scientific computing. Existing in situ accelerators for scientific computing, however, have a significant limitation: these accelerators provide no acceleration for preconditioning—a key bottleneck in linear solvers and in scientific computing workflows.**

This paper enables in situ acceleration for state-of-the-art linear solvers by demonstrating how to use a new in situ matrix inversion accelerator for analog preconditioning. As existing techniques that enable high precision and scalability for in situ MVM are inapplicable to in situ matrix inversion, new techniques to compensate for circuit non-idealities are proposed. Additionally, a new approach to bit slicing that enables splitting operands across multiple devices without external digital logic is proposed. For scalability, this paper demonstrates how in situ matrix inversion kernels can work in tandem with existing domain decomposition techniques to accelerate the solutions of arbitrarily large linear systems. The analog kernel can be directly integrated into existing preconditioning workflows, leveraging several well-optimized numerical linear algebra tools to improve the behavior of the circuit. The result is an analog preconditioner that is more effective (up to 50% fewer iterations) than the widely used incomplete LU factorization preconditioner, ILU(0), while also reducing the energy and execution time of each approximate solve operation by 1025x and 105x respectively.

***Index Terms*—Accelerator architectures, Analog computers, Linear systems**

I. Introduction

Scientific computing has been one of the most important drivers of innovation since the advent of digital computing. Advances in computing have enabled increasingly high-fidelity simulations, often of phenomena that would be too expensive or otherwise infeasible to investigate in the real world. To continue the pace of advancement as technology scaling slows requires new computing approaches and paradigms. One promising new paradigm is *in situ* computing where physical phenomena within memory arrays are exploited to perform computation more efficiently than using conventional digital systems [5]. In particular, *in situ* matrix-vector multiplication (MVM) shows significant potential due to the broad applicability of MVM.

Although many *in situ* MVM accelerators are optimized for the reduced precision requirements of machine learning [7], [31]–[33], recently an *in situ* MVM accelerator for scientific computing was proposed with significant potential benefits over a GPU-based system [11]. Unfortunately, accelerating MVM alone is insufficient for addressing state-of-the-art scientific computing problems. To efficiently solve large linear systems, where matrices can have billions of rows, iterative solvers are required. In these solvers, the rate of convergence is paramount, as reducing the number of iterations to convergence directly reduces both the latency of computation and the energy consumption. Moreover, for many challenging problems, widely used iterative solvers may not converge at all. To improve the rate of convergence, modern solvers use *preconditioners* that transform linear systems into problems that are easier to solve. Without support for preconditioning, *in situ* MVM acceleration for iterative solvers cannot provide the order-of-magnitude improvements implied by prior work [11].

Alternatively, recent work has demonstrated an *in situ* method for matrix inversion using analog crossbars [36]. However, these analog inversion circuits cannot handle matrices that are larger than a single memory array and cannot solve high-precision problems. Together, these limitations make them inadequate to solve the large, computationally intensive problems in scientific computing. Instead of using these circuit primitives to solve matrices directly, this work demonstrates that approximate analog matrix inversion is well-suited to preconditioning as part of a larger iterative solver. As a preconditioner, *in situ* matrix inversion only needs to provide an approximate solution to a linear system. Furthermore, domain decomposition—a widely used technique for preconditioning large systems—provides a framework for splitting the preconditioning problem into finite sized blocks which

Supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

are solved individually. The proposed analog preconditioner is more effective than the widely used software RAS+ILU(0) preconditioner, enabling convergence in up to 50% fewer iterations when used as part of the GMRES iterative solver.

To improve the accuracy of computation sufficiently for *in situ* matrix inversion to work as an effective preconditioner, a compensation scheme for circuit non-idealities arising from parasitic effects, amplifier impedances, and finite amplifier gain is developed. Additionally, to increase precision,a new modification of bit slicing is proposed to enable splitting operands across multiple devices as with *in situ* MVM.

To simplify the integration of the proposed *in situ* matrix inversion hardware with existing numerical linear algebra workflows, this paper demonstrates how the analog processing step can replace the commonly used incomplete LU decomposition (ILU) preconditioner with minimal algorithmic changes. Furthermore, this paper analyzes how existing optimizations in preconditioning workflows can augment the capabilities of the proposed hardware without significant changes to existing algorithms.

When compared to a digital system using ILU(0), the proposed *in situ* preconditioner reduces the energy consumption and execution time of an approximate solve by 1025× and 105×, respectively. The proposed system can be integrated with prior work on *in situ* accelerators for iterative solvers [11] with low overhead, maximizing the system-level benefits of preconditioning.

II. Background

This work builds on prior work that includes *in situ* computing and high performance linear algebra.

A. In Situ Computing

Over the past quarter century, there has been significant interest in processing in memory (PIM) accelerators to help combat the high cost of data movement energy. At the same time, developments in resistive memory have enabled numerous advances for *in situ* accelerators, with the potential to further improve conventional PIM systems. Unlike PIM systems where computing is performed close to the memory arrays, *in situ* computing performs computation within a memory array itself, and the result of the computation is read directly from the array. For instance, when performing a dot-product operation in a PIM system, the values of both vectors must be individually read out and multiplied. By contrast, in an *in situ* system, the multiplication and accumulation operations would be performed entirely within the memory array, and the sum read out directly.

1) In Situ Matrix Vector Multiplication: One *in situ* kernel that has been widely explored is matrix vector multiplication (MVM) due to the ease of mapping the kernel to resistive crossbars, and the broad applicability of MVM [5], [31]. Figure 1 shows a conceptual example of an *in situ* MVM accelerator performing the computation $\mathbf{A}\vec{x} = \vec{y}$ on a 3×3 resistive crossbar. The resistive elements are programmed such that the resistance $\mathbf{R}_{i,j}$ of each element in the matrix is

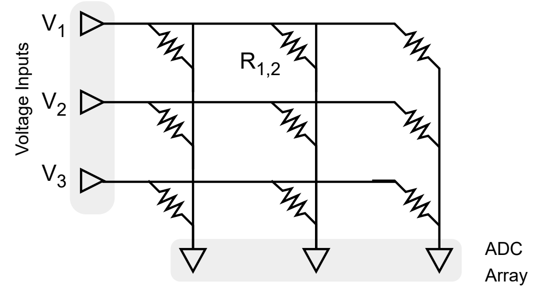


Fig. 1: Conceptual example of an *in situ* MVM crossbar.

proportional to the values $1/\mathbf{A}_{i,j}$, and voltages are applied to each row proportional to the values of $\vec{x}$, noted as V_{1-3} in Figure 1. Therefore, the current through an element at coordinate *i,j* in the matrix is equal to $V_i/\mathbf{R}_{i,j}$ where $\mathbf{R}$ is the resistance matrix, and $\vec{V}$ is the vector of applied currents. The currents through each column of the crossbar sum through Kirchoff's law, such that the current flowing into each ADC is proportional to y_j.

In situ MVM accelerators broadly use two different approaches for mapping a matrix to a crossbar array depending on the precision requirements of the application. One technique, *bit slicing*, enables *in situ* MVM with operand precision greater than can be reliably programmed into a single resistive device [5]. When using bit slicing, the matrix is divided into multiple N-bit matrices with the partial results from each crossbar aggregated through a shift-and-add reduction network [31]. Equation 1 shows an example of bit slicing using two bits per cell with a 4-bit fixed point matrix.

$$\begin{bmatrix} 7 & 9 \\ 12 & 1 \end{bmatrix} = 2^2 \begin{bmatrix} 1 & 2 \\ 3 & 0 \end{bmatrix} + 2^0 \begin{bmatrix} 3 & 1 \\ 0 & 1 \end{bmatrix} \quad (1)$$

Bit slicing also enables *in situ* accelerators to achieve the same theoretical precision as digital systems with sufficient ADC precision. Alternatively, several *in situ* accelerators have proposed using a single crossbar for the entire matrix, bounding the precision of the computation to the precision of multi-level memory elements; however, for machine learning applications this limitation is often tolerable to achieve higher energy efficiency and lower latency.

Using this *in situ* MVM primitive, many accelerators have been proposed for a wide range of applications. Neural networks are of particular interest for *in situ* MVM acceleration due to their tolerance of low precision, and MVM dominance. Although most of these accelerators focus purely on neural network inference to amortize the high cost of writing the devices, several accelerators for training specifically, or both training and inference have also been proposed [40]. Even with the reduced precision requirements of neural networks, many *in situ* neural network accelerators still rely on bit slicing to provide the same accuracy guarantees as a conventional digital system.

Beyond neural networks, *in situ* MVM acceleration has also been proposed for other MVM-dominated applications, including combinatorial optimization [5], graph analytics [33],

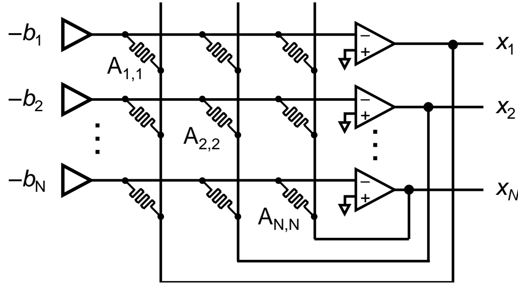


Fig. 2: Conceptual example of an *in situ* matrix inversion array.

and scientific computing [11]. These applications use bit slicing to achieve higher operand widths, from 32 bit fixed point in the Memristive Boltzmann Machine [5] to 128 bits to perform double precision floating point computation for scientific computing. Another approach by Le Gallo *et al.* [21] proposes using a two level solver for scientific computing, where a resistive crossbar MVM is used for a low precision solve, and the result is refined by a conventional system.

2) In Situ Linear Algebra Beyond MVM: Beyond *in situ* MVM, other *in situ* linear algebra accelerators have been proposed. These accelerators exploit the duality between an analog circuit, and a linear algebra kernel. For instance, Huang *et al.* [17] demonstrated an accelerator that uses analog integrators and multipliers to accelerate ordinary and partial differential equations (PDEs). The proposed accelerator builds on techniques proposed in the 1960s for analog computation and provides significant programmability for handling a wide variety of one and two dimensional PDEs. However, the authors note that for three dimensional PDEs the proposed accelerator runs into significant scalability limits when compared to digital systems.

A different analog accelerator design for solving linear systems using resistive crossbars that was initially proposed by Richter *et al.* [28], and recently experimentally demonstrated by Sun *et al.* [36], is shown in Figure 2. Similar to *in situ* MVM, a resistive crossbar is programmed such that the conductance of each device is proportional to the corresponding value in a matrix $\mathbf{A}$. Additionally, the voltages applied to the columns of the matrix, x_i, are analogous to the voltage inputs in Figure 1 such that a current proportional to the MVM result is flowing through each row. However, unlike *in situ* MVM, the voltages are not externally applied, but rather are determined by feedback provided by the amplifiers. The amplifier at the end of each row has a high input impedance. Consequently, with the current sources on the left-hand side of the figure disabled, the MVM must satisfy the equation $\mathbf{A}\vec{x} = 0$, where $\vec{x}$ is proportional to the voltages applied to each column. When the current sources are enabled the circuit must now satisfy the equation $\mathbf{A}\vec{x} - \vec{b} = 0$, or alternatively $\mathbf{A}\vec{x} = \vec{b}$. Therefore, the voltages at the output node of the amplifiers must be proportional to the solution vector $\vec{x}$ in the system of linear equations $\mathbf{A}\vec{x} = \vec{b}$. Using this circuit and similar circuit primitives, Sun *et al.* [34], [36] have demonstrated the potential to accelerate many applications that can be computed by solving a system of linear equations, including linear regression [37] and probabilistic eigenvalue computation, such as PageRank [35].

B. High Performance Linear Solvers

One of the most important kernels in scientific computing is solving a system of linear equations $\mathbf{A}\vec{x} = \vec{b}$. These systems of linear equations are often derived from discretizing PDEs, where a continuous PDE is transformed into a mesh that approximates the real geometry. Since most PDEs of interest do not have a closed-form solution, this is the primary method of modeling physical phenomena that are described by PDEs [30]. The systems of linear equations resulting from this discretization are typically large and sparse, meaning the majority of values in the matrix are zeros. The sparsity of these matrices enables significantly larger matrices to fit in memory than if the matrices were represented using a dense matrix format.

Solvers for sparse systems of linear equations can be subdivided into two categories: direct and iterative methods. Direct methods use techniques such as LU or QR factorization to solve a matrix, similar to the techniques used for solving dense matrices. However, direct solvers for sparse matrices suffer from *fill-in* where zero entries become nonzero, increasing the memory footprint of the computation and limiting scalability. For larger sparse matrices, iterative solvers such as Conjugate Gradient (CG) [14] or Generalized Minimal Residual (GMRES) [29] are typically used. In an iterative method, each iteration refines the estimate of $\vec{x}$ until the estimate reaches a specified tolerance, or a maximum iteration count is reached. One challenge with iterative methods is the rate of convergence; some matrices require tens or hundreds of thousands of iterations to reach the specified tolerance, and some matrices may not converge at all. To remedy this, and improve the rate of convergence, a *preconditioner* is typically applied during each iteration of the solver.

1) Preconditioning Iterative Methods: A preconditioner is a matrix $\mathbf{M}$ that approximates $\mathbf{A}$ such that solving for $\vec{x}$ converges faster with $\mathbf{M}^{-1}\mathbf{A}$ than with $\mathbf{A}$ alone [30]. Thus, rather than solving for $\vec{x}$ in $\mathbf{A}\vec{x} = \vec{b}$, a preconditioned iterative solver solves for $\vec{x}$ in $\mathbf{M}^{-1}\mathbf{A}\vec{x} = \mathbf{M}^{-1}\vec{b}$. In practice, preconditioning is implemented as a solve with matrix $\mathbf{M}$ during each iteration of the solver; therefore, solves with $\mathbf{M}$ should be significantly easier than solves with $\mathbf{A}$. For instance, a simple form of preconditioning is *Jacobi* (or diagonal) preconditioning. In Jacobi preconditioning, the matrix that approximates $\mathbf{A}$ is $\mathbf{M} = \text{diag}(\mathbf{A})$, and by extension $\mathbf{M}^{-1} = 1/\text{diag}(\mathbf{A})$. In this case solving $\mathbf{M}\vec{x} = \vec{b}$ is an element-wise multiplication between two vectors; however, the approximation is poor unless $\mathbf{A}$ is diagonally dominant. This highlights a common trade-off in preconditioners: better approximations of $\mathbf{A}$ generally speed preconditioning more efficiently, but they are more complex to compute in each iteration.

Another common form of preconditioner is incomplete LU (ILU) factorization, based on the LU factorization direct

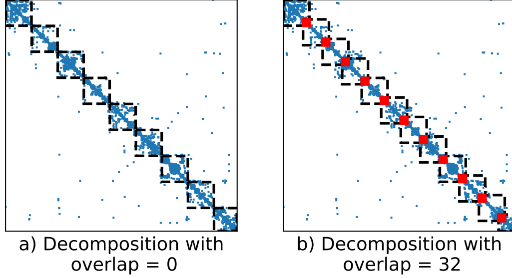


Fig. 3: Example of domain decomposition a) without overlap; b) with overlap

solver [30]. In both LU and ILU, $\mathbf{A}$ is factored into lower and upper triangular matrices $\mathbf{L}$ and $\mathbf{U}$. These matrices are subsequently used to compute or approximate $\mathbf{A}^{-1}$ in every iteration using forward and backward substitution. ILU is incomplete because values in $\mathbf{L}$ and $\mathbf{U}$ are ignored to reduce fill-in, creating an approximate solve with a significantly lower memory footprint. Different levels of fill-in are also commonly used to exploit the trade-off mentioned above between preconditioner efficiency and approximation quality. For instance, ILU(0) has the same sparsity pattern in the inverse as the original sparse matrix, such that there is no fill-in, and ILU(1) has the sparsity pattern of the matrix squared.

2) Domain Decomposition: One challenge with ILU approximations for preconditioning is the decreasing effectiveness of the approximation with larger matrices. To improve scalability, domain decomposition is a widely used technique that splits a matrix into blocks lying along the diagonal, and each block is solved [30]. The block solves can be performed using ILU for approximate solutions, or LU for full solves. Subsequently, block solutions are combined into a solution for the full matrix. Notably, even when each block is solved exactly, the combined solution from each matrix block is still an approximation as it does not include any values outside of the diagonal blocks. This is shown in Figure 3a, where nonzero matrix elements—represented by blue dots—appear outside of the diagonal blocks, and are not included in the full approximation.

To improve the approximation, the blocks can be overlapped such that the relationship between elements that would otherwise be in different blocks are considered; however, this come at the cost of additional blocks to cover the full diagonal. Figure 3b shows a set of overlapping blocks with the overlapping regions included in both blocks in red. When blocks are not overlapped, the resultant vector from each block solution can be concatenated to form a full solution vector: this is referred to as the Block Jacobi approach. When blocks are overlapped, techniques such as Restricted Additive Schwarz (RAS) are needed to combine the block solutions without double counting vector elements from the overlapping regions [6].

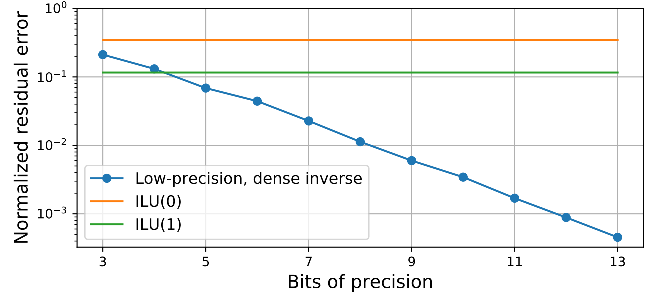


Fig. 4: Comparison of approximation accuracy between ILU(0), ILU(1) and low-precision circuit approximations.

III. Key Idea: An Analog Domain Decomposition Preconditioner

The analog MVM and matrix solver circuits discussed above are inherently limited to finite size memory arrays, and by extension finite problem sizes. Therefore, realistic problems must be split into subproblems that can fit onto a single crossbar and combined externally. Unlike MVM, however, the inherent nonlinearity of matrix inversion requires algorithmic support to split the matrix in a computationally useful way. Nonlinearity also precludes analog MVM techniques such as bit slicing, which are used to boost the precision of the computation. Together, these two requirements—tolerance for low precision, and algorithmic support for splitting the matrix into finite-sized subproblems—lead to the target application of domain decomposition preconditioners.

A purely analog approach without digital bit slicing raises potential concerns about the accuracy of computation, especially when used for scientific computing where double-precision floating point values are the norm. However, as discussed in the previous section, preconditioning leverages approximate solutions, which can be computed efficiently. Therefore, we propose that an analog circuit approximation of the full matrix $\mathbf{A}$ can itself serve as the preconditioner for the iterative solve.

To demonstrate the suitability of analog matrix inversion for preconditioning, Figure 4 compares the normalized residual error of matrix solutions between ILU(0) and an idealized circuit solution with finite bit resolution; the same level of precision is assumed on the inputs, programmed matrix elements, and outputs. This analysis uses a 1044×1044 sparse Wathen matrix [39], and does not consider circuit non-idealities, which will be addressed in the following section. Figure 4 shows that even when the approximation uses only three bits of precision, the ideal circuit approximation has lower error than ILU(0).

The difference in error between the two systems is caused by the difference in the approximation methods. Recall from the previous section that ILU approximates the matrix inverse by using only a subset of elements in the matrix inverse. Therefore, in a 1.5% dense matrix, such as the Wathen matrix used in Figure 4, 98.5% of the inverse elements are dropped, and the remaining 1.5% are represented with double precision floating point. *In situ* inversion by contrast represents the approximate

value of every element in the inverse; however, each value is only represented with only a few bits of precision.

As discussed in the previous section, in a domain decomposition framework each subblock is solved individually and the resultant vectors are combined to approximate the solution of the full matrix. These approaches can be applied directly to the analog matrix inversion, enabling solves from many crossbars to be efficiently combined into a single usable solution. This combination is still performed digitally and at high precision; therefore, the error of the full matrix solution is based on the approximation error of each block solve. Thus, the lower approximation error from the analog solver, shown in Figure 4, will also reduce the approximation error of the domain decomposition, creating a more effective preconditioner.

IV. Improving Precision

In the previous section, we noted that preconditioning does not require perfect accuracy. Nonetheless, there are still several sources of error introduced during the analog computation that can degrade the accuracy of a solution to an unacceptable degree, and need to be addressed.

A. Analog Bit Slicing

Representing the elements of $\mathbf{A}$ with just eight bits of precision is challenging for many resistive memory technologies. This necessitates a technique to split a high-precision computation over multiple low-precision devices. As mentioned previously, because matrix inversion is a nonlinear operation, this cannot be accomplished using conventional digital bit slicing methods.

Analog bit slicing enables splitting a matrix over multiple arrays by connecting multiple memory arrays together through amplifiers. This technique is an extension of an idea proposed by Sun *et al.* [34], [36] to handle positive and negative values by connecting two arrays through analog inverters. Figure 5 shows how both techniques can be combined to form an analog solver with twice the precision of a single device, representing both positive and negative values, using three arrays. Using b-bit cells, the full matrix can be decomposed as $\mathbf{A} = \mathbf{A}_{\text{low}} + 2^b\,\mathbf{A}_{\text{high}} - 2^b\,\mathbf{A}_{\text{neg}}$, where a single low-order bits matrix $\mathbf{A}_{\text{low}}$ is used to adjust both the positive and negative values of the matrix. Since each of the three matrices is strictly non-negative, the matrix $\mathbf{A}_{\text{neg}}$ is discretized to 2^b+1 levels (2^b positive values and zero) such that the largest negative value that can be represented is equal to the largest positive value.

Importantly, unlike digital bit slicing, analog bit slicing cannot be used to achieve arbitrary precision through increasing numbers of arrays. As the number of arrays is increased, the column voltages must pass through multiple gain stages—each de-amplifying the signal by 2^b—to reach the lowest-bit array. Eventually, these voltages will approach the noise floor of the system, limiting the possible precision advantages. Based on the observation in Section III, we focus on the three-array circuit in Figure 5, as it provides sufficient precision for effective preconditioning. We find that splitting each element in the matrix into a sum of conductances lying in separate arrays reduces the current flowing through each individual array, slightly mitigating the accuracy loss induced by parasitic resistances.

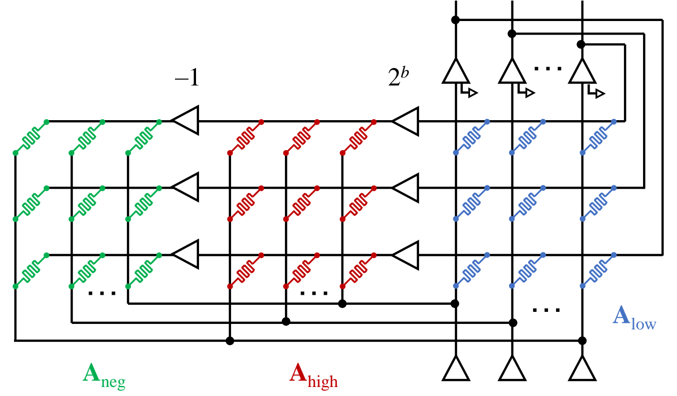


Fig. 5: Analog bit slicing using a three matrix system.

B. Compensating for Circuit Non-Idealities

Although the analog solver circuit in Figure 2 finds a precise matrix solution in the ideal case, circuit non-idealities can create differences between the target matrix $\mathbf{A}$ and the matrix that is actually being inverted by the physical system: $\mathbf{A}^*$. The latter is computed from $\mathbf{A}$ using a circuit simulation. Figure 6 shows the element-wise difference between $\mathbf{A}$ and $\mathbf{A}^*$ for a dense $200{\times}200$ matrix, where Figure 6c shows the matrix with no compensation for these non-idealities. In this work we focus on the impact of two non-idealities: finite gain of the row amplifiers and parasitic resistances, as shown in Figure 6a.

1) Compensating for Amplifier Non-Idealities: As described in Section II-A2, a virtual ground at the input of the row amplifier is essential to enforce the needed equality between the input currents $\vec{b}$ and the currents $\mathbf{A}\vec{x}$ that flow through the crossbar memory elements. In reality, this condition cannot perfectly hold due to the non-ideal properties of the amplifier, namely a finite gain $-\alpha$ (inverting), input resistance R_{in}, and output resistance R_{out}. When the rows are no longer at virtual ground, the feedback connection from the output to the input of the jth amplifier crosses not only the resistor R_{jj} but also through additional sneak paths within the array. Provided that the output resistance is sufficiently small, this effect can be well approximated as an error along the diagonal of the matrix and thus can be corrected by programming a compensated matrix onto the crossbar that differs from $\mathbf{A}$ only along the diagonal elements. For the case of a single positive array without bit slicing, as in Figure 2, the correction for amplifier non-idealities is given by:

$$R_{ij} = \begin{cases} R_{\min} A_{ij}^{-1} & i \neq j \\ \dfrac{\alpha' R_{\min}}{\alpha' A_{ij} - \sum_k A_{ik} - R_{\min} R_{\text{in}}^{-1}} & i = j \end{cases} \tag{2}$$

where $R_{\min}$ is the minimum resistance of an array element, which is used to represent the maximum matrix element value

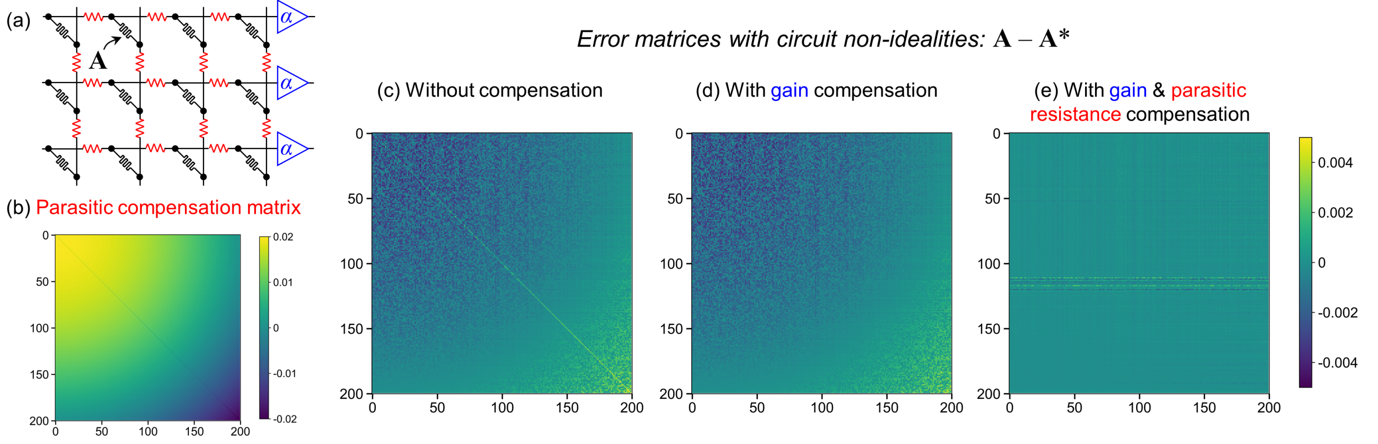


Fig. 6: (a) Finite amplifier gain and parasitic array resistance can induce errors in the analog solve; (b) Compensation matrix $\mathbf{P}$ for parasitic resistance; (c)-(e) show element-wise error patterns for a random 200×200 dense matrix with: (c) no compensation for circuit non-idealities; (d) gain compensation; (e) gain and parasitic resistance compensation. A gain of $\alpha = 10^5$ and a resistance ratio of 10^6 between the resistive memory and the parasitic resistance is assumed.

of unity after scaling, and

$$\alpha' = 1 + \frac{\alpha}{1 + R_{\text{out}} R_{\text{min}}^{-1} \sum_j A_{ij}} \tag{3}$$

The result of applying this correction can be seen in Figure 6 for a 200×200 matrix and a gain of $\alpha = 10^5$. The finite gain introduces a pronounced error along the diagonal in the uncompensated matrix (Figure 6c), which is eliminated upon applying the correction above (Figure 6d). This correction remains accurate so long as the output resistance is small enough to satisfy $R_{\text{out}} \ll \alpha / \sum_j \frac{1}{R_{ij}}$ for all columns i. When this no longer holds, the output resistance introduces an even greater degree of coupling among all the array currents, and a static correction to the programmed matrix will not guarantee an effective compensation for non-ideal amplifier properties. Nonetheless, this condition is readily satisfied by a well-designed operational amplifier at practical array sizes, especially if the matrix is sparse.

For the case of the three-array system in Figure 5 that implements negative weights and analog bit slicing, the correction needs only to be applied on one of the three arrays. Let $\mathbf{R}_{\text{low}}$, $\mathbf{R}_{\text{high}}$, and $\mathbf{R}_{\text{neg}}$ denote the resistance matrices that implement the decomposed matrices $\mathbf{A}_{\text{low}}$, $\mathbf{A}_{\text{high}}$, and $\mathbf{A}_{\text{neg}}$, respectively. For the two arrays that are not directly connected to the row amplifiers, the resistive elements can be set directly to their corresponding matrix values, both on and off the diagonal: $R_{\text{high},ij} = R_{\text{min}} / A_{\text{high},ij}$ and $R_{\text{neg},ij} = R_{\text{min}} / A_{\text{neg},ij}$ for all i and j. Additionally, $R_{\text{low},ij} = R_{\text{min}} / A_{\text{low},ij}$ for $i \neq j$. Zero values are represented by the maximum resistance of the array element R_{max}.

The on-diagonal resistances in $\mathbf{R}_{\text{low}}$ are corrected for amplifier non-idealities:

$$R_{\text{low},ii} = \frac{\alpha' R_{\text{min}}}{\alpha' A'_{\text{low},ii} - \sum_k \left(A'_{\text{low}} + A'_{\text{high}} + A'_{\text{neg}}\right)_{ik} - R_{\text{min}} R_{\text{in}}^{-1}} \tag{4}$$

where α' has the same definition as in Equation (3), and the modified matrix elements for each of the three decomposed matrices are:

$$A'_{\text{X},ij} = \begin{cases} A_{\text{X},ij} & A_{\text{X},ij} > 0 \\ R_{\text{min}}/R_{\text{max}} & A_{\text{X},ij} = 0 \end{cases} \tag{5}$$

Notably, for a given choice of the matrix $\mathbf{A}$, the amplifier gain α, and on/off ratio $R_{\text{max}}/R_{\text{min}}$ of the resistive memory element, there is no guarantee that the gain compensation above can be applied; for this compensation to be valid for a particular element $R_{\text{low},ii}$, the denominator of Equation (4) must remain positive. Consider a matrix $\mathbf{A}$, which represents an $N \times N$ domain of a larger problem matrix, that has a sparsity of S (fraction of zeros). Assuming a large amplifier input resistance $R_{\text{in}} \gg R_{\text{min}}$, we can write this requirement approximately as:

$$\alpha' - \frac{3NSR_{\text{min}}}{R_{\text{max}}} - \gamma N(1-S) > 0 \tag{6}$$

The first term assumes that the diagonal elements $A_{\text{low},ii}$ have maximal values; using the MC64 scaling algorithm (see Section V-B), this is guaranteed. The second term represents the zero elements along the ith row, for which $A'_{\text{low}} = A'_{\text{high}} = A'_{\text{neg}} = R_{\text{min}}/R_{\text{max}}$. The third term represents the nonzero elements along the ith row, where γ is the expected value $E[A'_{\text{low}} + A'_{\text{high}} + A'_{\text{neg}}]$. For a given matrix element A_{ij}, one of A_{high} or A_{neg} will always be zero, and we make the worst-case assumption that the other element is maximal. Therefore, $E[A'_{\text{high}} + A'_{\text{neg}}] = R_{\text{min}}/R_{\text{max}} + 1 \approx 1$. We assume that the off-diagonal low-order bits will be randomly distributed and thus, $E[A'_{\text{low}}] = 0.5$ and $\gamma = 1.5$.

Rearranging the above equation and using our estimate of $\gamma = 1.5$, we obtain the following requirement for gain compensation:

$$\frac{R_{\text{max}}}{R_{\text{min}}} > \frac{3NS}{\alpha' - 1.5N(1-S)} \tag{7}$$

This relationship connects the device on/off ratio to the amplifier gain, domain size, and matrix sparsity. There are two possible ways that this condition may fail to be satisfied: (1) If the right-hand side diverges, no gain compensation is possible regardless of the device on/off ratio. In this case, the amplifier gain is insufficient to overcome the effect of sneak currents through the array elements that correspond to the nonzero values of $\mathbf{A}$. This is addressed by increasing the amplifier gain, using a smaller array, or by constructing a sparser domain. Roughly speaking, the gain needs to exceed the number of nonzero elements in a row; the more sparse the matrix, the smaller the gain required, which reduces the amplifier power consumption. The required gain is larger in the presence of input or output resistances. (2) The on/off ratio does not exceed the value on the right-hand side. In this case, the amplifier has sufficient gain to overcome the sneak currents through the nonzero elements but not through all the elements (zero and nonzero). The zero elements are implemented using the resistance $R_{\max}$, and thus draw a small but non-negligible sneak current.

In this work, we design an operational amplifier (Section VI-A) whose gain is sufficient to handle the level of sparsity that is present in a suite of representative problem matrices for scientific computing. For the resistive devices, we assume a resistive memory technology with an on/off ratio of 300, which has been previously demonstrated [16]. Gain compensation is not applied to any element that fails to satisfy Equation (7).

In many cases, a large proportion of the above amplifier compensation term may be truncated due to finite operand precision. However, even in these cases, accurately computing this compensation can improve the accuracy by altering the rounded values of each matrix. For instance, when the gain compensated matrix is truncated to 4 bits, gain compensation has the overall effect of zero-centering the diagonal errors although it does not eliminate them.

2) Parasitic Compensation: A larger source of error is the parasitic interconnect resistance between the cells in the crossbar, shown in Figure. 6(a). As a result of voltage drops along the rows and columns, the desired voltage fails to appear across the memory elements, distorting the product $\mathbf{A}\vec{x}$. For *in situ* MVM, several compensations schemes have been proposed that exploit the feed-forward behavior of the circuit to balance the parasitic resistance seen across different paths [2], [18]. These techniques are not directly applicable to *in situ* matrix inversion because of the feedback paths in the circuit. Nevertheless, a similar approach to parasitic compensation can be applied in this case by determining a general error pattern $\mathbf{P}$, which can then be used to apply a correction for all matrices of a certain size and parasitic resistance.

We generate the error matrix from the observation that due to the way that IR drops are accumulated in the memory array, parasitic resistance affects all matrices of a given size similarly, with a comparatively small variation in its effect from problem to problem. We therefore compute $\mathbf{P}$ by starting with $\mathbf{A_0}$, which is a uniform matrix of ones with gain compensation

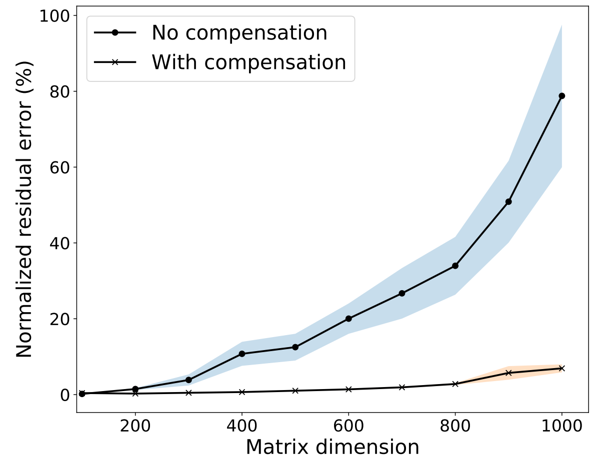


Fig. 7: The accuracy of the analog solve versus matrix size (fully dense) with and without parasitic compensation, averaged over 10 matrices per size. An array with a device resistance of $R_{low} = 1M\Omega$ and parasitic resistance of 0.1 Ω is assumed.

applied along the diagonal. This matrix, which represents the case where all cells draw the same current, is used as an average case for the purposes of compensation. We then compute $\mathbf{A}_0^*$, which accounts for the effect of parasitic resistance. The error matrix is given by $\mathbf{P} = \mathbf{A_0} - \mathbf{A}_0^*$, which is shown in Figure 6(b) for a 200×200 matrix. The same general error pattern appears at all matrix sizes. Notably, the uniform matrix $\mathbf{A}_0$ is singular, and therefore cannot be inverted; however, the corresponding matrix with circuit effects $\mathbf{A}_0^*$ is invertible.

For any problem matrix $\mathbf{A}$, the matrix that will be programmed into the crossbar is given by: $\mathbf{A}_{\text{final}} = \text{GC}(\mathbf{A}) + (\mathbf{P} * \text{GC}(\mathbf{A}))$, where GC refers to the gain compensation equations in the previous section. The error pattern after applying both gain and parasitic resistance compensation is shown in Figure 6e. $\mathbf{P}$ is scaled by the value of $\mathbf{A}$ to avoid the compensation term overwhelming small values in the matrix. We note while the parasitic correction $\mathbf{P}$—which is derived from our choice of $\mathbf{A}_0$—reveals the underlying parasitic error pattern common to all matrices, it does not fully eliminate the errors in $\mathbf{A}^*$ that are specific to the problem $\mathbf{A}$, which appear as perturbations on this general pattern that can be seen in Figure 6d.

Figure 7 shows how the accuracy of the analog solve – expressed as the normalized residual $(\vec{b} - \mathbf{A}\vec{x})/||\vec{x}||$, where $\vec{x}$ is the output of the circuit – degrades with increasing array size due to parasitic resistance. We assume a minimum device resistance of 1 MΩ (achievable with various non-volatile memories such as flash memory and resistive RAM) and 0.1 Ω parasitic resistance per cell. Parasitic compensation greatly mitigates these errors, enabling effective preconditioning even for large matrices. We note that these results are for a fully dense matrix; for sparse matrices, the effect of parasitic resistance is reduced as many of the devices in the array no longer draw any current, and even larger matrices can be

accelerated.

Although the analysis above focuses on a single uniform parasitic value across the array, the proposed parasitic compensation technique should work with randomly variable parasitic values. The parasitic compensation matrix is based on the average parasitic effect over many matrices, across which the parasitic voltage drops appear in different ways. If there is a random spread in the value of the parasitic resistance, a compensation based on the average parasitic resistance value will still produce a benefit for the same reason that it remains effective for different randomly chosen matrices and inputs. Additionally, the observation that an effective parasitic compensation matrix can be found as a function of only the circuit parameters suggests that tailored parasitic compensation matrices could be extracted post-fabrication and distributed with the accelerator based on the specific process variation.

V. Integration with Linear Algebra Tools

Section III discusses how domain decomposition techniques from numerical linear algebra provide algorithmic separability to enable a scalable preconditioner system using the solver. Similarly, other widely used numerical linear algebra techniques are equally applicable to the proposed *in situ* preconditioner. The setup process for both the digital and *in situ* preconditioners are discussed below. The setup process is done once for a solve and the same setup is used for all iterations in a solver. Both preconditioners use the same basic process, allowing the analog solver hardware to be directly integrated into widely used tools without substantial overhead. This ensures that the analog solver can integrate with, benefit from, and accelerate state-of-the-art linear algebra techniques.

A. Matrix Partitioning

Prior work on *in situ* acceleration of sparse matrix operations proposed a matrix partitioning method based on recursively splitting blocks. Although this technique was effective on the matrices analyzed by Feinberg *et al.* [11], a more advanced tool for matrix partitioning is widely used in numerical linear algebra. Metis [19] is a matrix partitioning tool for unstructured graphs that can produce balanced partitions consisting of non-contiguous sets of rows by leveraging the duality between sparse matrices and graphs.

Both the baseline digital system and the proposed analog preconditioner use Metis to split the sparse matrix into multiple domains. One important difference between the two systems is how to handle differing sizes within domains. This is especially important for overlapping domains, where even if the initial domains are identical in size, there can be significant differences in the size of the overlap regions for each domain. Conventional digital systems can easily handle imbalances between domain sizes. However, a hardware solver requires the domains to remain at a fixed size that corresponds to the size of an array. To remedy this, we developed a new variable-overlap, fixed-size domain method where each domain is greedily expanded until it includes exactly enough elements for a given array size. For instance, if a domain with no overlap that contains 1013 entires is mapped to an array of size 1024×1024, the remaining 11 entries will be filled with random neighboring entries.

B. Matrix Scaling

Matrix scaling is commonly used to reduce the condition number of a matrix, and to improve the stability of LU decomposition by maximizing the matrix values on the diagonal. We assume all matrices are structurally non-singular, which is reasonable as singular matrices are not invertible. In this case, there exists a permutation P and diagonal scalings D_1 and D_2 such that PD_1AD_2 has ones on the diagonal and the off-diagonal entries are all less than one [27][Thm. 2.8]. In the sparse case, the MC64 routine in HSL finds this permutation and scaling efficiently [10]. For symmetric positive definite (SPD) matrices, no permutation is needed and the scaling is trivial as the larges entries are always on the diagonal. Therefore, we will hereafter assume matrices have been permuted/scaled to have unit diagonal.

In addition to improving condition number, MC64 scales the weights such that they are ideal for mapping to a physical system. The finite bounded range both maximizes the precision of representation for each value in the matrix by optimally using the available physical dynamic range and simplifies the mapping of each value across the analog bit slices.

C. Preconditioner Setup

Once each domain has been appropriately scaled, the approximate solution method—either ILU or an *in situ* matrix solution—is initialized. For the digital system, an ILU decomposition is computed and the factorized values of L and U are stored. For the *in situ* matrix solver, the gain and parasitic compensation methods discussed in Section IV are applied, the matrix elements are mapped to the appropriate resistances, and programmed into the arrays.

D. Interfacing with the Analog Solver

Interfacing to the accelerator is similar to interfaces previously proposed for *in situ* MVM [5], [7], [11]. The preprocessing steps described above are performed on the host, and the processed matrices are written to the accelerator controller which handles the specific programming operations for each array. Unlike prior work, analog preconditioning does not require data exchange between multiple arrays; therefore, the data path configuration steps of prior work are not needed. Instead, the results of each domain solve can be read out by the host after a fixed latency. The proposed accelerator can also integrate with prior work on *in situ* iterative solver acceleration [11]. In this case, preconditioner arrays are added to each accelerator bank, and an additional preconditioner kernel is added to the controller program to be executed once per iteration. In either case, the approximate solve used as part of preconditioning is typically implemented through an optimized library routine rather than directly implemented by the programmer. Similarly, the analog preconditioner can be

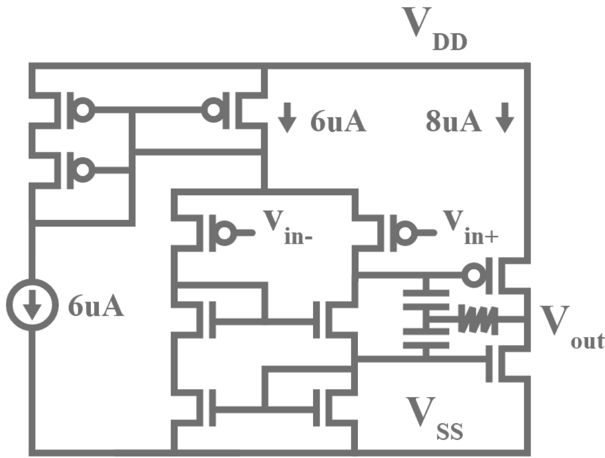


Fig. 8: Operational amplifier circuit with class AB output for low static power dissipation

exposed to the programmer primarily through an optimized API.

VI. Hardware Design

We focus primarily on the design of the feedback amplifiers on each row, the analog to digital converter (ADC) and the digital to analog converters (DAC) as the components that most directly impact system accuracy, and are optimized specifically for the proposed accelerator.

A. Operational Amplifier Design

Given the importance of the operational amplifier (OA) to the performance of the circuit, we design a custom OA using a commercial 14 nm PDK. The OA shown in Figure 8 is designed using 800 mV supply voltage and achieves 12 μW static power dissipation with an estimated area footprint of $< 50\,\mu\text{m}^2$. The design features a pFET input differential pair and dual nFET current mirrors. The two common-source output transistors form a class AB output stage and are biased in the subthreshold regime to provide a large dynamic output current swing. The open-loop OA achieves 36 dB differential mode (DM) gain, 28 dB common mode rejection (CMR) and 500 MHz unity gain frequency. Miller compensation is used to adjust the open-loop phase margin to 80°, which is intended to provide considerable stability headroom in the closed-loop system. For dynamic circuit simulations, we model the parasitic circuit elements within a memory cell using the technology specifications; these include parasitic capacitance to ground, capacitance between neighboring word lines and bit/word-line wire resistance.

To validate the functionality, we perform transient circuit simulations of a 64×64 resistive element array using the above OA design and parasitic components based on a commercial 14 nm process. All input current sources are set to 100 ns pulse width. For each test case, all input current sources are identical and adjusted as a whole between 250 nA and 1100 nA so that the expected solution falls within 50 mV of the rail voltage. Figure 9 demonstrates the output waveform for a

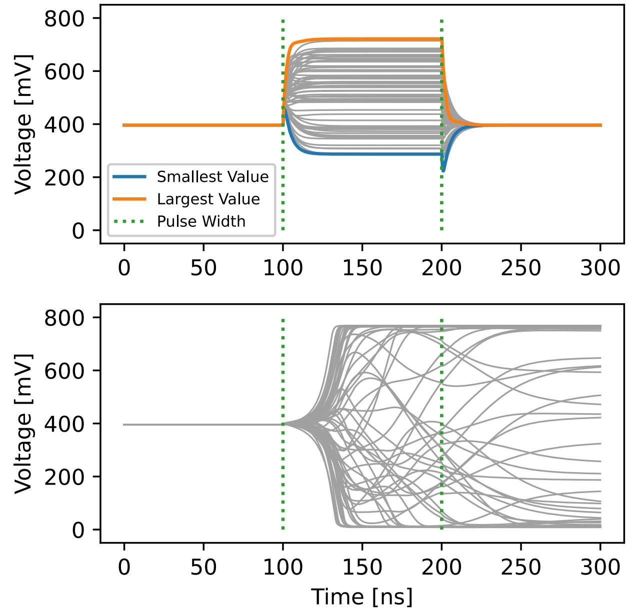


Fig. 9: Top) output voltage waveforms using text matrix which met stability conditions Bottom) example of diverging solution by intentionally violating stability criterion

stable solution and for a test matrix that intentionally did not meet the stability criterion discussed below.

For the circuit to converge, all OAs using negative feedback must be stable. A negative feedback loop is created when a resistive element is connected from the OA output to its inverting input. Problematically, additional inverting sneak paths within the array can cause the system to become unstable depending on the $\mathbf{A}$ matrix. This issue was previously identified by [36] which showed that the loop gain of every amplifier is stable when the OA gain is sufficiently large and the diagonal of $\mathbf{A}^{-1}$ is strictly positive. In practice, this stability condition is not a major limitation for the proposed system. This criterion is guaranteed for positive definite matrices [15], and is also met by many, but not all, matrices that are not positive definite. Importantly, the criterion must be met by each 1024×1024 domain, rather than over the full matrix.

B. Ramp Analog to Digital Converter

Prior work on *in situ* MVM has often opted for successive approximation register (SAR) ADCs, where a single high-precision ADC is multiplexed over each output, and a full MVM quantization latency is defined as the sampling period of the ADC multiplied by the number of outputs [11]. By producing outputs one at a time, SAR ADCs complement bit-sliced architectures where each output requires reduction using digital logic, and replicating the digital logic per output would be prohibitive. Since *in situ* matrix inversion is incompatible with digital bit slicing we use a different ADC design; a parallel ramp ADC, similar to the system proposed by Marinella *et al.* [25].

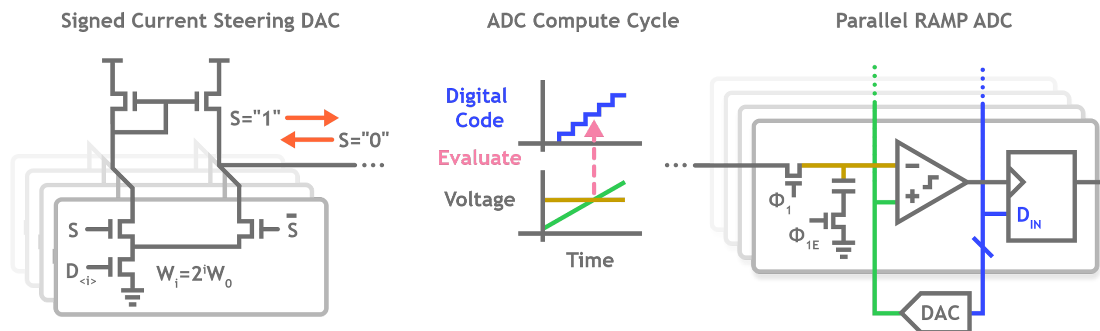


Fig. 10: Operation of the parallel current steering DAC (left), and parallel ramp ADC (right).

A parallel ramp ADC (Figure 10) consists of a single shared DAC connected to a digital counter that generates a voltage ramp through each possible ADC output. Each column's voltage is sampled onto a capacitor using an integrator, and passed into a comparator that triggers when the ramp signal exceeds the output value of the array and latches the counter value into an output register. The advantage of a Ramp ADC for *in situ* matrix inversion is the relatively low target precision and large array size. The Ramp ADC is particularly advantageous for this application since our *in situ* matrix inversion operation requires relatively low target precision and is intended to scale to a very large array size. By assuming a SAR ADC has the same sample rate as the clock frequency of the ramp counter, a ramp ADC will offer a lower full-array quantization latency if the number of levels to quantize is less than the number of bits. The proposed accelerator targets eight-bit outputs with 1024×1024 arrays; therefore, a ramp ADC will quantize the full set of array outputs 4× faster than a single SAR ADC.

C. Digital to Analog Current Converter Array

Unlike *in situ* MVM accelerators that can use bit slicing or similar techniques for the input vector, the nonlinearity of matrix inversion necessitates a current based DAC array for *in situ* matrix inversion. The input vector $\vec{b}$ is scaled and mapped to an array of 7 bit (6 + sign) fixed point current sources.

A signed current steering DAC is realized by a set of differential pairs with a shared pFET current mirror (Figure 10). The nFET devices are sized appropriately for each bit to provide appropriately scaled current. When the sign bit is low, the output of the device sinks current. When the sign bit is high, the device uses the pFET current mirror to source current. Notably, the output voltage is ensured by feedback within the later OA, increasing the effective output resistance.

VII. Experimental Setup

We develop a detailed system model to evaluate the latency, energy efficiency, area, and accuracy of the analog preconditioner. Key parameters for the analog preconditioner are listed in Table I.

A. Circuits and Devices

To characterize the accuracy of the matrix inversion including parasitic effects, we develop a specialized circuit analysis tool that computes the matrix $\mathbf{A}^*$ for a given input matrix, device resistance range, parasitic resistance, and amplifier gain. This tool constructs a linear conductance matrix describing the circuit with parasitic resistances and extracts the inverse values corresponding to each element of the original matrix. We validated the tool against DC operating point simulations in LTSpice [1] on 50 arrays with sizes ranging from 20×20 to 45×45. Overall, this tool has an average residual error of 0.12% and a maximum error of 2.7%. Using this tool, we can characterize the parasitic effects on each 1024×1024 domain significantly more efficiently than with SPICE. To further improve simulation runtime, after confirming that the three-array system had comparable errors to the single array system, all simulations use a single array system, and zero values are treated as open connections. However, gain compensation is disabled for rows that do not satisfy the condition in Equation 7.

TABLE I: System Parameters

Cells	$R_{\text{low}} = 1\text{M}\Omega$, $R_{\text{high}} = 300\text{M}\Omega$, 4 bits per cell
Arrays	3 array stack, ±8 effective bits per matrix element, 1T1R
ADC	8-bit ramp
DAC	6 + 1 (sign) bit

The evaluation uses process parameters from a commercial 14 nm PDK and key peripheral circuits are designed and simulated using the same process. We use the circuits implemented in Kull *et al.* [20] and Baert *et al.* [4] to model the area and energy of the capacitive DAC and the asynchronous counter, respectively, within the ramp ADC. The remaining peripheral circuits are modeled using parameters from Marinella *et al.* [25], which are also based on a 14 nm commercial process. We model a resistive memory with a low resistance state of 1MΩ and an On/Off ratio of $R_{\text{high}}/R_{\text{low}} = 300$, similar to the devices used by Hu *et al.* [16]. For the energy estimates, we make the worst-case assumption that every device has a resistance equal to $R_{low} = 1\text{M}\Omega$. We also draw from the circuit designs in that work for the current conveyor integrator and the comparators, which have been optimized for low power and small footprint. To enable programming of individual memory devices, we assume a 1T1R array in which each unit cell contains an access transistor that remains on during a solve operation. Based on a 1T1R cell layout, we extract the parasitic interconnect resistance between memory cells as R_p=8Ω.

Based on prior work [34], we assume the matrix inversion settling time scales as row_{nnz}/λ_{min} where row_{nnz} is the maximum number of nonzeros in a domain row, and λ_{min} is the smallest eigenvalue of the matrix. To establish a baseline settling time for the system we compute row_{nnz}/λ_{min} for the matrix simulated in Figure 9 and measure how long this matrix takes to settle to a value within $1/512$ of the final value such that the ADC would observe no further changes. Since the settling time is dependent on the matrix, and is not known initially, we assume each column's voltage is sampled onto a load capacitor using a current conveyor every 256 ns to align with the 8 bit ADC quantization interval. The computation halts when two subsequent quantizations are identical. The resistive array, its peripheral amplifiers, and the current DACs must all remain turned on during this time. For subsequent solves of the same matrix, the overall solve time is known, so the quantization can be performed a full quantization interval earlier, and the amplifiers and DACs can be powered off to save energy.

B. Architecture

The analog preconditioner is compared against two conventional systems, a CPU system with dual socket Intel Xeon-E5 2620v3s, and a GPU system with an Nvidia Volta V100 SXM2. We measure the execution time of the LU solve function in each domain using precomputed $\mathbf{L}$ and $\mathbf{U}$. For the CPU, each domain is timed individually on a single core, which represents the best case scenario for the solve as there is no interference from other cores. Energy is measured using the Running Average Power Limit (RAPL) interface [8] summing both the reported processor and memory energies. For the GPU, the solver kernels equal to the total number of domains are launched together to increase GPU occupancy, and GPU power consumption is measured using the Nvidia Management Library (NVML). For both energy measurements, we subtract the energy consumption reported by RAPL/NVML from the background energy consumption measured when the system is idle.

TABLE II: Evaluated Matrices

Name	**Size**	**Number of Nonzeros**	**Mean Domain Condition Number**
bodyy4	17,546	121,550	189.2
t2d_q4	9,801	87,025	195.2
poisson3Da	13,514	352,762	87.7
Dubcova1	16,129	253,009	84.3

C. Algorithms

We evaluate the preconditioner using GMRES(20) on four matrices from the SuiteSparse matrix collection (Table II) [9]. We selected matrices with between 8K and 20K rows so that the $\mathbf{A}^*$ matrices could be realistically computed. Additionally, we select matrices that converge to a tolerance of 10^{-10} using unpreconditioned GMRES(20) in more than 200 iterations. This criterion ensures that the matrices being evaluated have a sufficient number of iterations to distinguish between the different preconditioning approaches.

The baseline preconditioner in our analysis is domain decomposition (RAS) with ILU(0), meaning ILU where the decomposition has the same number of nonzeros as the original sparse matrix. Although more advanced ILU techniques exist, such as ILUTP, tuning the parameters in these systems is a significant challenge even for domain experts. We instead focus on ILU(0) for each domain solve due to its widespread use and general effectiveness. Incomplete LU decomposition was performed using the SuperLU [23], and cuSPARSE libraries.

VIII. Evaluation

We evaluate the effectiveness, energy, execution time, and area of a CPU based preconditioner compared to the proposed analog preconditioner.

A. Preconditioner Effectiveness

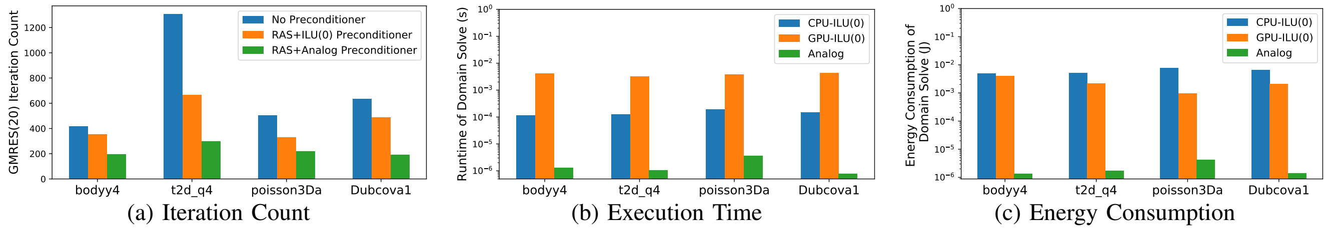
Figure 11a shows the GMRES(20) iteration counts required to achieve convergence. These results demonstrate the importance of preconditioning to iterative linear solvers, as applying even a simple preconditioner can reduce the total iteration count by nearly 50%. This also shows the limitation of prior work on *in situ* MVM for iterative solvers [11], as without support for preconditioning, the benefits of *in situ* MVM acceleration is bounded by the increased iteration count.

Consistent with the idealized results shown in Figure 4, the analog inversion better approximates the true inverse, resulting in a significant reduction in iteration count. However, unlike Figure 4, this analysis includes additional sources of error due to circuit non-idealities, which can be compensated using the techniques discussed in Section IV.

Importantly, some matrices that satisfy the stability condition noted in Section VI-A can fail to be effective preconditioners, increasing the total iteration count. However, the matrices which are difficult for the proposed accelerator often require more complex preconditioners than ILU(0), thereby increasing the cost of preconditioning. For instance, the *memplus* matrix from the SuiteSparse matrix collection [9] requires over 4000 iterations with an ILU(0) preconditioner. Concretely identifying the requirements for effective preconditioning of a broader range of matrices, and implementing more complex analog preconditioning schemes is an important question for future work.

B. Execution Time

Figure 11b compares the execution time of each domain solve with the CPU and GPU based conventional systems. The purely analog nature of the solver circuit enables each domain to be solved several orders of magnitude faster than conventional digital systems. The overall execution time of the preconditioner ranges from 768 ns to 3.58 μs based on the settling time of the slowest domain. To place these matrix solve times in the context of prior work on *in situ* acceleration of iterative linear solvers, Feinberg *et al.* [11] report a single 512×512 *in situ* crossbar operation requires 427 ns, and due to the bit slicing approach, each *in situ* crossbar operations must be performed many times. Assuming



(a) Iteration Count (b) Execution Time (c) Energy Consumption

Fig. 11: Summary of results comparing the analog preconditioner to conventional digital systems.

in situ floating point emulation where each MVM requires 64 vector bit slices, the total *in situ* MVM requires 28 μs, 7.6 to 36$\times$ slower than a 1024$\times$1024 approximate solve. Therefore, analog preconditioning can be added to previously proposed *in situ* accelerators for iterative solvers with a small increase in per-iteration runtime, maximizing the benefits that can be realized from preconditioning.

Figure 11b also shows the CPU-based solve out-performing the GPU-based solve. This is due to the low number of domains in the analysis, which leaves the GPU heavily underutilized. Additionally, the triangular back-substitutions used for solve with **L** and **U** are heavily sequential and difficult to efficiently implement on GPUs [3], [22], [26]. To make this computation more amenable to a GPU, the number of domains could be significantly increased. In this case, the CPU-based system would suffer as the total number of domains would prevent a simple mapping of one domain per CPU core. However, the analog solver can effectively scale to many nodes since each analog solver array is significantly smaller than either of the CPU or GPU baselines.

TABLE III: Solver Array Energy Breakdown

Component	Energy (μJ)	Energy (%)
Total	1.377	100.00
DAC	1.132	82.83
ADC	0.010	0.071
Array	0.173	12.57
Amplifiers	0.061	4.46

C. Energy Consumption

Figure 11c compares the energy consumed when performing an approximate solve on a single domain between the analog preconditioner and the conventional digital systems. Similar to the observation above, the energy consumption of a 1024$\times$1024 *in situ* approximate solve is comparable to the energy consumption of a full 512$\times$512 *in situ* MVM with double precision floating point emulation [11]. Therefore, the energy savings from fewer required iterations can be achieved with low preconditioning overhead.

A breakdown of the energy consumption of the solver where the circuit settles within 500 ns is shown in Table III, where the parallel DACs are the dominant factor. This differs significantly from *in situ* MVM where low-precision inputs are typically used, and several techniques exist to handle multi-bit inputs [40]. However, the nonlinearity of matrix inversion prevents the application of those techniques. Through judicious design of the other components, the energy consumption of the high precision DAC is tolerable. One important part of the low energy consumption is the OA discussed in Section VI-A. Notably, this OA only has a 36 dB differential gain, significantly lower than is potentially required for a 1024$\times$1024 array (Section IV). However, due to the sparsity of the matrices of interest, and the matrix scaling to create a unit diagonal, a low gain amplifier can be used with significant energy savings.

TABLE IV: Solver Array Area Breakdown

Component	Area (mm^2)	Area (%)
Total	0.543	100.00
DAC	0.1131	20.84
ADC	0.0223	4.11
Array	0.2538	46.75
Amplifiers	0.1537	28.30

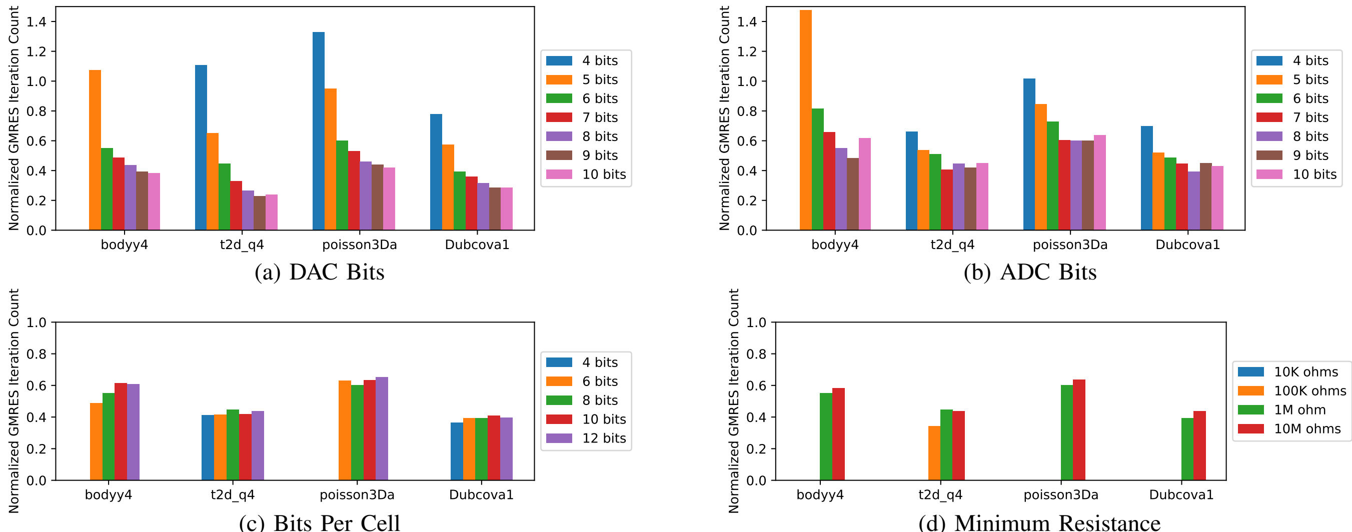
D. Area

An area breakdown of an analog domain solver is shown in Table IV. The DAC array is quite small despite being power-hungry due to the small footprint of the differential pairs.

The overall area of 0.543 mm^2 for each solver is small enough that *in situ* preconditioners could be readily integrated with previously proposed accelerators for iterative solvers. For instance, the accelerator proposed by Feinberg *et al.* has a total area of 539 mm^2. Adding an analog preconditioner circuit to each bank would incur only a 13% overhead, while providing over 50% speedups as discussed in Section VIII-B.

E. Initialization Overheads

In order to conduct the analog solve operation, a matrix must be written into the array. We estimate the cost of writing and fine-tuning a 1024$\times$1024 crossbar of ReRAM devices, stored in a 1T1R array with 4 bits per cell. SET and RESET operations require $I_{\text{form}} = 20\mu\text{A}$, $I_{\text{SET}} = 20\mu\text{A}$, and $I_{\text{RESET}} = 40\mu\text{A}$, similar to values given in prior work [24], [38]. The proposed system employs low-power writes to drive down total system cost; word-lines charged to low values ($V_g < 0.7\text{V}$) provide sufficient current to write devices in a commercial 14 nm process. Given a sparsity of 5% nonzeros, we assume a column-wise write-verify scheme can perform parallel writes [12], [13]; therefore, we estimate the per-column write budget is 920 nJ over 30 iterations. Assuming all three arrays shown in Figure 5 must be written, this yields 3.9 mJ per initial write operation. These iterations are also the dominant latency for initialization. Assuming a 20 ns write pulse per column per iteration, the worst case latency for writing a 1024$\times$1024 array is 1.229 ms. Including read operations, the total latency for a



(a) DAC Bits (b) ADC Bits

(c) Bits Per Cell (d) Minimum Resistance

Fig. 12: Sensitivity of GMRES(20) iteration counts, as a function of a) DAC bits; b) ADC bits; c) Bits per cell; and d) the minimum device resistance with R_p=8Ω. No results are shown for cases where the system did not converge within 2000 iterations.

full array write is 1.686 ms, or 5.058 ms if all three arrays are programmed sequentially.

However, iterative solvers are often used as part of a sequence of solves where the underlying matrix is updated between solves. In this case, when fine tuning the nonzeros from a previous solve, only 10 iterations are required, and updates are only needed in two of the three arrays. Therefore, when fine tuning the energy cost falls to 867μJ, and latency drops proportionately to 512μs per array. Notably, the initial ILU(0) factorization on the baseline system can take several milliseconds; therefore, even in the initial write case, the setup costs of the analog preconditioner are comparable to digital systems, and the analog solvers are more efficient once initialized.

F. Sensitivity to Circuit Parameters

Figures 12 shows the effect of different types of precision on the efficacy of the preconditioner. In general increasing the number of DAC bits is the most effective way to improve the preconditioner, but as discussed above it is also typically the most expensive. By selecting a 6 bit DAC, the proposed system can maximize the effectiveness of the preconditioner within a tight energy budget, as there are significant diminishing returns on adding a 7th DAC bit.

Decreasing the minimum resistance below 1MΩ makes three of the four test matrices non-convergent, although above that threshold increasing the resistance has little effect. This suggests that the parasitic compensation technique discussed in Section IV may introduce, rather than eliminate noise when the parasitic resistance effects are small. Additionally, the lack of convergence for these matrices below 6 bits per cell suggests that analog bit slicing is currently necessary to realize analog preconditioning; however, stacking additional arrays does not appear necessary. Therefore, the potential noise limits of analog bit slicing are unlikely to be a significant problem.

IX. Conclusion

This work demonstrates the potential of a new *in situ* kernel for matrix inversion that utilizes analog emerging devices and circuit optimizations. Unlike prior work on *in situ* MVM, the non-linearity of matrix inversion creates several challenges requiring careful co-design of the hardware and algorithm to develop an accelerator for useful problems. Nevertheless, an appropriate co-design approach enables an accelerator that requires minimal support from inefficient special-purpose digital logic. The proposed accelerator design leverages existing tools and techniques from numerical linear algebra, simplifying the integration of the accelerator into existing systems. The combination of co-design and detailed circuit analysis to compensate for non-idealities enables a preconditioning accelerator that is both up to 50% more effective than a commonly used preconditioner, and improves the runtime and energy consumption of the preconditioner by 105$\times$ and 1025$\times$ respectively. Although the proposed accelerator is not currently applicable to all matrices, and focuses on a single preconditioning technique, these results suggest that analog preconditioning has significant potential for numerical linear algebra.

References

[1] "LTspice," https://www.analog.com/en/design-center/design-tools-and-calculators/ltspice-simulator.html, 2020.

[2] S. Agarwal, R. L. Schiek, and M. J. Marinella, "Compensating for parasitic voltage drops in resistive memory arrays," in *2017 IEEE Intl. Memory Workshop (IMW)*, 2017, pp. 1–4.

[3] H. Anzt, E. Chow, and J. Dongarra, "Iterative sparse triangular solves for preconditioning," in *Euro-Par 2015: Parallel Processing*, 2015, pp. 650–661.

[4] M. Baert and W. Dehaene, "20.1 a 5GS/s 7.2 ENOB time-interleaved VCO-based ADC achieving 30.5fj/conv-step," in *2019 IEEE Intl. Solid-State Circuits Conference - (ISSCC)*, 2019, pp. 328–330.

[5] M. N. Bojnordi and E. Ipek, "Memristive Boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, March 2016.

[6] X. Cai and M. Sarkis, "A restricted additive Schwarz preconditioner for general sparse linear systems," *SIAM Journal on Scientific Computing*, vol. 21, no. 2, pp. 792–797, 1999.

[7] P. Chi, S. Li, S. Li, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *Intl. Symp. on Computer Architecture (ISCA)*, June 2016.

[8] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: memory power estimation and capping," in *Intl. Symp. on Low-Power Electronics and Design (ISLPED)*, 2010, pp. 189–194.

[9] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[10] I. S. Duff and J. Koster, "The design and use of algorithms for permuting large entries to the diagonal of sparse matrices," *SIAM Journal on Matrix Analysis and Applications (SIMAX)*, vol. 20, no. 4, pp. 889–901, 1999.

[11] B. Feinberg, U. K. R. Vengalam, N. Whitehair, S. Wang, and E. Ipek, "Enabling scientific computing on memristive accelerators," in *Intl. Symp. on Computer Architecture (ISCA)*, 2018, pp. 367–382.

[12] E. J. Fuller, S. T. Keene, A. Melianas, Z. Wang, S. Agarwal, Y. Li, Y. Tuchman, C. D. James, M. J. Marinella, J. J. Yang *et al.*, "Parallel programming of an ionic floating-gate memory array for scalable neuromorphic computing," *Science*, vol. 364, no. 6440, pp. 570–574, 2019.

[13] D. C. Guterman, N. Mokhlesi, and Y. Fong, "Charge packet metering for coarse/fine programming of non-volatile memory," 27 2006, US Patent 7,068,539.

[14] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *Journal of Research of the National Bureau of Standards*, vol. 49, no. 6, pp. 409–436, Dec 1952.

[15] R. A. Horn and C. R. Johnson, *Matrix Analysis*, 2nd ed. Cambridge, UK: Cambridge Univ. Press, 2013.

[16] M. Hu, J. P. Strachan, Z. Li, E. M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, J. J. Yang, and R. S. Williams, "Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication," in *Design Automation Conference (DAC)*, June 2016.

[17] Y. Huang, N. Guo, M. Seok, Y. Tsividis, and S. Sethumadhavan, "Evaluation of an analog accelerator for linear algebra," in *Intl. Symp. on Computer Architecture (ISCA)*, 2016, p. 570–582.

[18] Y. Jeong, M. A. Zidan, and W. D. Lu, "Parasitic effect analysis in memristor-array-based neuromorphic systems," *IEEE Trans. on Nanotechnology (TNANO)*, vol. 17, no. 1, pp. 184–193, 2018.

[19] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing (SISC)*, vol. 20, no. 1, pp. 359–392, 1998.

[20] L. Kull, T. Toifl, M. Schmatz, P. A. Francese, C. Menolfi, M. Brändli, M. Kossel, T. Morf, T. M. Andersen, and Y. Leblebici, "A 3.1 mW 8b 1.2 GS/s single-channel asynchronous SAR ADC with alternate comparators for enhanced speed in 32 nm digital SOI CMOS," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 48, no. 12, pp. 3049–3058, 2013.

[21] M. Le Gallo, A. Sebastian, R. Mathis, M. Manica, H. Giefers, T. Tuma, C. Bekas, A. Curioni, and E. Eleftheriou, "Mixed-precision in-memory computing," *Nature Electronics*, vol. 1, no. 4, pp. 246–253, 2018.

[22] R. Li and Y. Saad, "GPU-accelerated preconditioned iterative linear solvers," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 443–466, 2013.

[23] X. S. Li and M. Shao, "A supernodal approach to incomplete LU factorization with partial pivoting," *ACM Trans. Mathematical Software (TOMS)*, vol. 37, no. 4, 2010.

[24] M. Mao, Y. Cao, S. Yu, and C. Chakrabarti, "Programming strategies to improve energy efficiency and reliability of ReRAM memory systems," in *2015 IEEE Workshop on Signal Processing Systems (SiPS)*, 2015, pp. 1–6.

[25] M. J. Marinella, S. Agarwal, A. Hsia, I. Richter, R. Jacobs-Gedrim, J. Niroula, S. J. Plimpton, E. Ipek, and C. D. James, "Multiscale co-design analysis of energy, latency, area, and accuracy of a ReRAM analog neural training accelerator," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems (JETCAS)*, vol. 8, no. 1, pp. 86–101, 2018.

[26] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU," *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011*, vol. 1, 2011.

[27] M. Olschowka and A. Neumaier, "A new pivoting strategy for Gaussian elimination," *Linear Algebra Appl.*, no. 240, pp. 131–151, 1996.

[28] I. Richter, K. Pas, X. Guo, R. G. Patel, J. Liu, E. Ipek, and E. G. Friedman, "Memristive accelerator for extreme scale linear solvers," in *Gov. Microcircuit Applications & Critical Technology Conference (GOMACTech)*, 2015.

[29] Y. Saad and M. H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on Scientific and Statistical Computing (SISC)*, vol. 7, no. 3, pp. 856–869, 1986.

[30] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 3rd ed. SIAM, 2003.

[31] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Intl. Symp. on Computer Architecture (ISCA)*, June 2016.

[32] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A pipelined ReRAM-based accelerator for deep learning," in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2017, pp. 541–552.

[33] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating graph processing using ReRAM," in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2018, pp. 531–543.

[34] Z. Sun, G. Pedretti, and D. Ielmini, "Fast solution of linear systems with analog resistive switching memory (RRAM)," in *Intl. Conf. on Rebooting Computing (ICRC)*, 2019, pp. 1–5.

[35] Z. Sun, E. Ambrosi, G. Pedretti, A. Bricalli, and D. Ielmini, "In-memory PageRank accelerator with a cross-point array of resistive memories," *IEEE Trans. on Electron Devices (T-ED)*, vol. PP, pp. 1–5, 02 2020.

[36] Z. Sun, G. Pedretti, E. Ambrosi, A. Bricalli, W. Wang, and D. Ielmini, "Solving matrix equations in one step with cross-point resistive arrays," *Proceedings of the National Academy of Sciences (PNAS)*, vol. 116, no. 10, pp. 4123–4128, 2019.

[37] Z. Sun, G. Pedretti, A. Bricalli, and D. Ielmini, "One-step regression and classification with cross-point resistive memory arrays," *Science Advances*, vol. 6, no. 5, 2020.

[38] H. Van Tran, A. Ly, T. Vu, S. Hong, F. Zhou, X. Liu, and N. Do, "Methods for writing to an array of resistive random access memory cells," 14 2019, US Patent App. 16/119,416.

[39] A. J. Wathen, "Realistic eigenvalue bounds for the Galerkin mass matrix," *IMA Journal of Numerical Analysis*, vol. 7, no. 4, pp. 449–457, 1987.

[40] T. P. Xiao, C. H. Bennett, B. Feinberg, S. Agarwal, and M. J. Marinella, "Analog architectures for neural network acceleration based on non-volatile memory," *Applied Physics Reviews*, vol. 7, no. 3, p. 031301, 2020.