A Decomposition-Based Memristive Crossbar Solver and FPGA-Accelerated Hardware Implementation

Suyash Vardhan Singh
Computer Science and Engineering
University of South Carolina
Columbia, SC, USA
ss121@email.sc.edu

Anzhelika Kolinko
Computer Science and Engineering
University of South Carolina
Columbia, SC, USA
akolinko@email.sc.edu

Md Hasibul Amin
Computer Science and Engineering
University of South Carolina
Columbia, SC, USA
ma77@email.sc.edu

Ramtin Zand
Computer Science and Engineering
University of South Carolina
Columbia, SC, USA
ramtin@cse.sc.edu

Jason D. Bakos
Computer Science and Engineering
University of South Carolina
Columbia, SC, USA
jbakos@cse.sc.edu

Abstract

Memristive crossbar-based analog processor-in-memory (PIM) architectures have the potential to deliver substantially higher energy efficiency for machine learning workloads than traditional architectures. The availability of a fast and accurate circuit-level simulation framework could enhance research and development efforts in this field. This paper introduces XbarSim, a domain-specific circuit-level solver designed to generate and solve the nodal equations of memristive crossbars including the effects of bitline and wordline resistance, and deploy the solver onto an FPGA emulator. XbarSim also supports partitioning larger arrays horizontally and vertically in order to subdivide the solver workload to manage memory locality and limit the resource requirement when deployed on an FPGA. The solver uses LU decomposition to pre-process the conductance matrix for each partition and solves for a batch of inputs to achieve high solver throughput. We demonstrate that XbarSim can achieve orders of magnitude speedup compared to Hspice across various sizes of memristive crossbars, and the XbarSim FPGA emulator can further achieve a 2.10X to 5.59X speedup over our software version built on Matlab.

ACM Reference Format:
Suyash Vardhan Singh, Anzhelika Kolinko, Md Hasibul Amin, Ramtin Zand, and Jason D. Bakos. 2025. A Decomposition-Based Memristive Crossbar Solver and FPGA-Accelerated Hardware Implementation. In *Great Lakes Symposium on VLSI 2025 (GLSVLSI '25), June 30–July 02, 2025, New Orleans, LA, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3716368.3735282

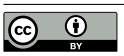
1 Introduction

Memristive crossbar arrays serve as vital components in analog processing-in-memory (PIM) and in-memory computing (IMC) architectures [3, 5–7, 12]. They offer significant acceleration for matrix-vector multiplication operations in machine learning (ML) models by leveraging massive parallelism, analog computation, and minimizing data transfer overheads between memory and processor. As interest in IMC architectures grow, there remains a lack of fast and accurate circuit-level solvers able to support hardware designers in implementing and validating their designs. Moreover, within the crossbar setting of IMC architectures, there is a wide array of design choices and hyperparameters that can be adjusted to fulfill particular design objectives. Hence, developing a crossbar solver tailored to assist in the early-stage design decisions for crossbar-based circuits and architectures can prove highly advantageous.

Memristive crossbar circuits have different non-ideal factors such as parasitics, noise, process variation, sneak paths, and stuck-at faults, which need to be taken into account while predicting their performance across various applications. To address these challenges, a wide range of crossbar solvers has been proposed in the literature. These can be broadly classified into two categories: analytical simulators [4, 10, 13] and circuit-level simulators [2, 9, 14]. MNSim [13] is an analytical simulator employing analytical models to assess the performance of various elements within the crossbar. However, its validation reveals over 5% error in power, energy, and latency calculations compared to full SPICE-level simulations for a 3-layer fully connected neural network (NN).

NeuroSim [4] also offers analytical models for evaluating the power, area, and latency of crossbars. Unlike MNSim, NeuroSim integrates with NN simulators to obtain learning and classification accuracies, but it still lacks an accurate circuit-level prediction model for the analog behavior of crossbar arrays. RxNN [10] provides a faster model for crossbar simulation with non-idealities, but still relies on some abstractions of non-ideal behavior, lacking a full-circuit simulation.

On the other hand, circuit-level approaches, such as those based on SPICE or customized nodal analysis, provide high fidelity by capturing detailed device and interconnect behavior, but suffer from scalability issues when dealing with large arrays. Badcrossbar [11] is a Python-based crossbar simulation framework, which plots voltages and currents for crossbar circuits, but it lacks accurate modeling of non-idealities due to parasitics, noise or process variation. Moreover, it directly solves the full nodal equations using linear algebra e.g., via NumPy and SciPy, and simulation becomes computationally expensive for large arrays. DPE [9], a MATLAB-based simulator, focuses on finding an optimized mapping scheme



GLSVLSI '25, New Orleans, LA, USA

ACM ISBN 979-8-4007-1496-2/25/06
https://doi.org/10.1145/3716368.3735282

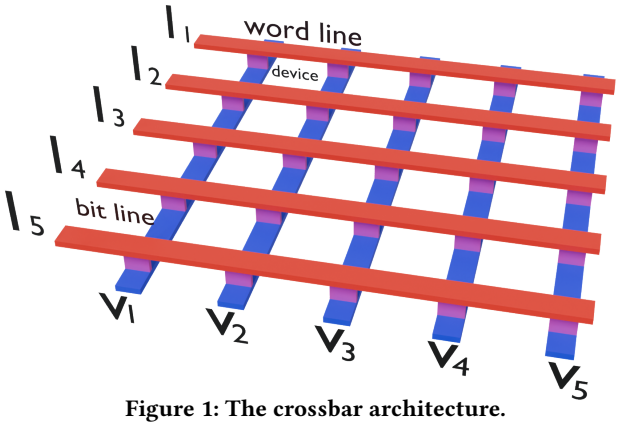


Figure 1: The crossbar architecture.

for memristive crossbars considering non-ideal effects, but lacks accuracy in interconnect resistance calculation. The CCCS [14], another MATLAB-based solver, incorporates techniques for simplifying the evaluation of parasitic effects on the crossbars to speed up the simulation. However, it suffers from high error due to inaccurate estimation of voltage drops caused by parasitic resistance. IMAC-Sim [2] can automatically handle the SPICE simulations of crossbar circuits along with an accurate calculation of parasitics, yet the simulation time is high because it requires a full-circuit SPICE simulation. CrossSim [8] is another circuit-level crossbar simulator capable of handling various non-ideal factors including parasitics and noise, but it can become slow for very large networks if the parasitic-aware mode is enabled across all layers of the DNN model.

In this work, we propose XbarSim, a novel crossbar solver framework including an FPGA emulator, that leverages the lower-upper (LU) decomposition technique to solve the nodal equations in a crossbar. XbarSim includes the effects of parasitic resistances within crossbars and supports batch processing of the inputs. XbarSim can be executed in software using Matlab but also generates an FPGA-based solver through the Vitis HLS design flow, which provides a 3X speedup over Matlab. The FPGA emulator performs the solve in double precision floating-point, stores the matrix data for each partition in on-chip RAM, and uses a single 64-bit memory port for reading input current and outputting output voltages.

2 Methodology

In this work, we consider the crossbar model shown in Fig. 1, comprised of horizontal word lines (WL) and vertical bit lines (BL), with a memristive device at the intersection between each WL and BL and a resistor between each pair of adjacent memristive devices to model the resistance of the word lines and bit lines. Each WL is driven with a current source to model the input vector, and the voltage at the termination of each BL corresponds to the output vector.

2.1 Circuit Model

Consider a crossbar of m wordlines (WLs) and n bitlines (BLs). When including both memristors and a wire loss resistor between each pair of elements on the WLs and BLs, the circuit contains $3 \times m \times n$ resistors and $2 \times m \times n$ voltage nodes. The conductances of the resistors may be stored in a symmetric $2 \times m \times n$ by $2 \times m \times n$ matrix $\boldsymbol{G}$ with $(4(n-2)+6) \times (4(m-2)+6)$ nonzero entries (i.e. the matrix is sparse).

The currents flowing into each node can be stored in a $2 \times m \times n$ vector I (all but n of which are zero, except when using partitioning to solve, explained below). The vector of $2 \times n \times m$ node voltages, V can be computed by solving the equation $\boldsymbol{G}V = I$. For constructing the matrix, assume the following notation:

(1) V_i: the voltage source at WL i,
(2) $D_{(i,j)}$: the memristor that is connected between WL i and BL j,
(3) $G_{i,j}$: its conductance of $D_{(i,j)}$,
(4) G_{BL}: conductance of bit line (wire resistance),
(5) G_{WL}: conductance of word line (wire resistance),
(6) $V_{WL(i,j)}$: the voltage node of the WL-side of the memeristor,
(7) $V_{BL(i,j)}$: voltage node of the BL-side of the memristor,
(8) $I_{D(i,j)}$: current from $V_{BL(i,j)}$ to $V_{WL(i,j)}$,
(9) $I_{WL(i,j)}$: current from $V_{WL(i,j)}$ to $V_{WL(i,j-1)}$, and
(10) $I_{BL(i,j)}$: current from $BL_{i+1,j}$ to $V_{BL(i,j)}$.

These notations are depicted in Fig. 2, which shows three sections of the crossbar array, the upper left, upper right, and lower right, showing how the nodal equations are computed differently for the first row, first column, last row, and last column.

Kirchhoff's current law gives the following equations [11]:

$$I_{WL(i,j)} - I_{WL(i,j+1)} - I_{D(i,j)} = 0 \; for \; j < n \tag{1}$$

$$I_{WL(i,j)} - I_{D(i,j)} = 0 \; for \; j = n \tag{2}$$

$$I_{D(i,j)} - I_{BL(i,j)} = 0 \; for \; i = 1 \tag{3}$$

$$I_{D(i,j)} + I_{BL(i-1,j)} - I_{BL(i,j)} = 0 \; for \; i > 1 \tag{4}$$

When $j = 1$ voltages at WLs are just input voltages:$V_{WL(i,j)} = V_{WL(i,1)} = V_i$ which is also shown in Eq.5.

$$\begin{aligned}(V_{WL(i,j)} - V_i) \times G_{WL} = (V_{BL(i,j)} - V_{WL(i,j)}) \times G_{(i,j)} + \\ + (V_{WL(i,j+1)} - V_{WL(i,j)}) \times G_{WL}; \Rightarrow \\ V_{WL(i,j)} \times (2 \times G_{WL} + G_{(i,j)}) + V_{WL(i,j+1)} \times (-G_{WL}) + \\ + V_{BL(i,j)} \times (-G_{(i,j)}) = V_i \times G_{WL}; \\ for \; 1 \le i \le m, \; j = 1\end{aligned} \tag{5}$$

Nodes corresponding to internal columns ($1 < j < n$) and internal rows ($1 < i < m$) can be computed as in Eqs. 6 and 7. Other equations for node voltages can be derived similarly.

$$\begin{aligned}(V_{WL(i,j)} - V_{WL(i,j-1)}) \times G_{WL} = (V_{BL(i,j)} - V_{WL(i,j)}) \times \\ \times G_{(i,j)} + (V_{WL(i,j+1)} - V_{WL(i,j)}) \times G_{WL}; \Rightarrow \\ V_{WL(i,j)} \times (2 \times G_{WL} + G_{(i,j)}) + V_{WL(i,j-1)} \times (-G_{WL}) + \\ + V_{WL(i,j+1)} \times (-G_{WL}) + V_{BL(i,j)} \times s(-G_{D(i,j)}) = 0; \\ for \; 1 < i < m, \; 1 < j < n\end{aligned} \tag{6}$$

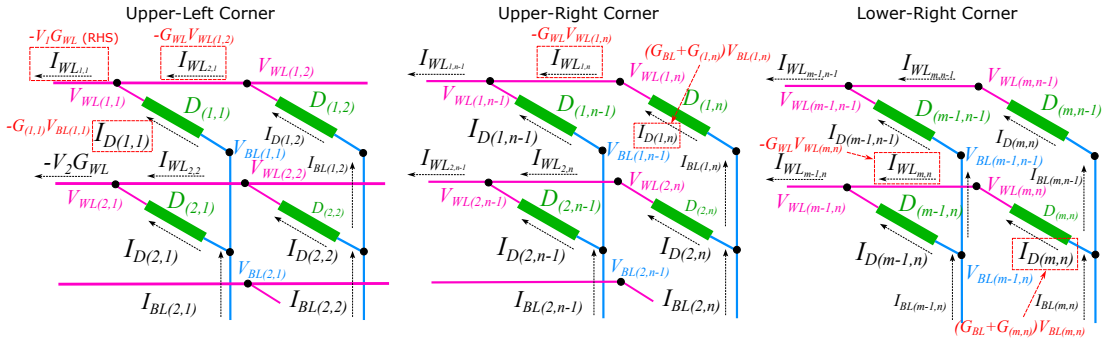


Figure 2: Notion used in crossbar matrix.

$$
\begin{aligned}
(V_{BL(i,j)} - V_{WL(i,j)}) \times G_{(i,j)} + (V_{BL(i,j)} - V_{BL(i-1,j)}) \times G_{BL} = \\
= (V_{BL(i+1,j)} - V_{BL(i,j)}) \times G_{BL}; \Rightarrow \\
V_{BL(i,j)} \times (2 \times G_{BL} + G_{(i,j)}) + V_{BL(i+1,j)} \times (-G_{BL}) + \\
+ V_{BL(i-1,j)} \times (-G_{BL}) + V_{WL(i,j)} \times (-G_{(i,j)}); \\
for\ 1 < i < m,\ 1 < j < m \quad (7)
\end{aligned}
$$

2.2 Matrix Formulation

The resulting equations form a sparse system of linear equations. This system is equivalent for row and column permutations, since row exchanges correspond to a change in the order of equations, while column exchanges correspond to a change in the order of summands in each equation, assuming that corresponding permutations are performed on the vector of unknowns and the right-hand side vector of known values.

One can use a permutation of columns $\{V_{WL(1,1)}, V_{BL(1,1)}, V_{WL(1,2)}, V_{BL(1,2)}, ..., V_{WL(m,n-1)}, V_{BL(m,n-1)}, V_{WL(m,n)}, V_{BL(m,n)}\}$ that corresponds to the vector of unknown nodal voltages. To make the resulting matrix banded diagonal, one can use such a permutation of rows that the matrix value $M_{i,j}$ at the diagonal would be the corresponding unknown coefficient on the right-hand side of the system of linear equations. Hence, for selected column permutation, corresponding row permutation would be nodal equations corresponding to equations $\{1\ (i = 1, j = 1), 3\ (i = 1, j = 1), 1\ (i = 1, j = 2), 4\ (i = 1, j = 2), ..., 1\ (i = m, j = n-1), 4\ (i = m, j = n-1), 2\ (i = m, j = n), 4\}$. An example of a matrix formed this way for $m = 4, n = 3$, can be seen in Fig. 3. Note that Gwl and Gbl represent the wordline and bitline parasitics resistance values, while Gij represent the conductance of crossbar memristor i, j.

There are numerous potential methods for solving the resultant system, such as iterative and decomposition methods. Iterative methods are sensitive to the initial guess, which makes it less practical for this application. On the other hand, decomposition methods can also diverge in cases where several rows (or, equivalently, columns) are close to linearly dependent. However, since we are modeling a real physical system, the resulting matrix should have a solution and is expected to have linearly independent rows (or, equivalently, linearly independent columns).

For this reason, we use an LU decomposition-based solver, in which $\boldsymbol{G}V = I$ is transformed into $\boldsymbol{LU}V = I$, where $\boldsymbol{L}$ is a lower triangular matrix with ones on its diagonal and $\boldsymbol{U}$ is an upper triangular matrix. Substituting $y = \boldsymbol{U}V$ into the equation gives $\boldsymbol{L}y = I$. The solution for the vector y can be calculated with no more than L_{nnz} multiplies and $2nm$ subtracts, where nnz is the number of non-zero entries. The solution for the vector V can be calculated by substituting y into $\boldsymbol{U}V = y$ and solving for V, which requires no more than U_{nnz} multiplies, $2mn$ subtracts, and $2mn$ divides.

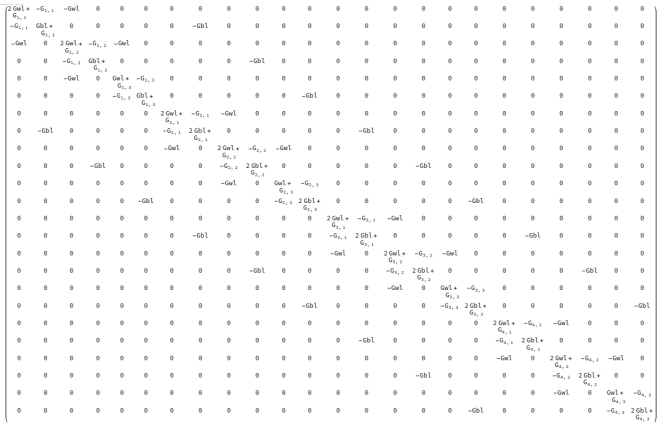


Figure 3: Example of a matrix formed from nodal equations for a 4×3 crossbar.

2.3 Solving via LU Decomposition

In order to quickly evaluate batches of inputs against a simulated crossbar, we use LU decomposition. The solver is designed to solve the $\boldsymbol{G}V = I$ system of equations for a known set of input currents I and an unknown set of output voltages V and assuming the $\boldsymbol{G}$ matrix has already been pre-decomposed. This gives the system $\boldsymbol{LU}V = I$ and is solved in two steps: (1) setting $y = \boldsymbol{U}V$ and solving for y in $\boldsymbol{L}y = I$ (forward substitution) and (2) using the resulting y to solve $\boldsymbol{U}V = y$ (backward substitution).

2.4 HSPICE Baseline

In order to validate the correctness of our proposed solver and to compare its performance against an industry standard circuit simulator, our software generates an HSPICE circuit corresponding

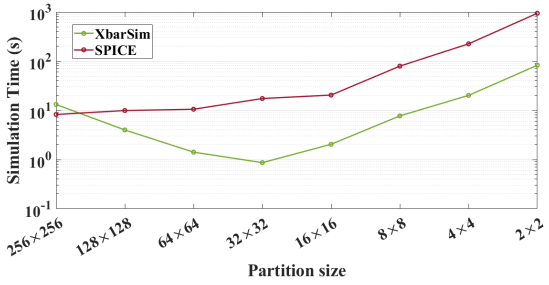


Figure 4: Simulation time for 256×256 crossbar partitioned with different crossbar sizes.

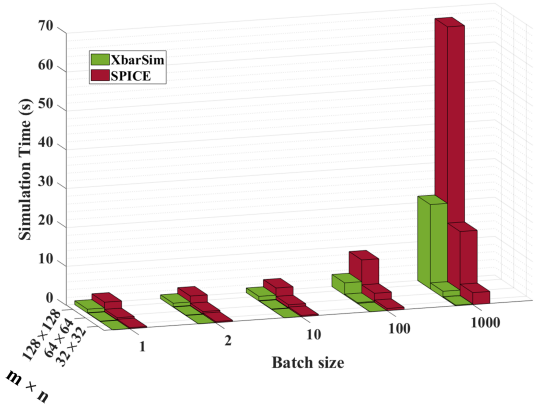


Figure 5: Batched simulation time for various crossbars.

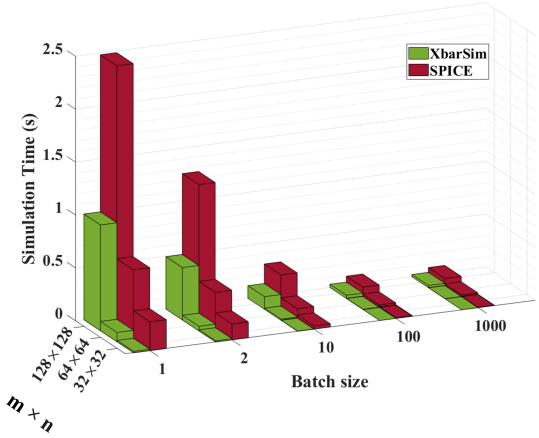


Figure 6: Simulation time per input sample for various crossbars with different batch sizes.

to a crossbar of a given size. In the generated file, the memristor conductances and the input voltages are randomly generated, and the wordline and bitline parasitic resistors are set at 5 Ohms each [1]. We perform a DC steady-state analysis of the circuit using HSPICE 2019.06-2 with the option ".option post=2 POST_VERSION=2001" to store the output nodal voltages in an ASCII output file. Batch runs are carried out by associating ".DATA" values with the input voltage sources, allowing HSPICE to perform a DC analysis for each of these input voltage values.

2.5 Partitioning

A crossbar can be simulated by sequential simulation of its smaller partitions. This divide-and-conquer strategy can lead to increased memory locality and–in the case of a hardware-based solver–fewer resources. The crossbar may be partitioned horizontally, vertically, or both. When partitioning, the crossbar is solved in row-wise order from upper-left to lower-right to allow for the incorporation of previously-solved word line and bit line currents into the I vector [1]. Additionally, the conductance matrix for each partition must incorporate the equivalent word line and bit line conductance of any partitions to the right and/or below.

Fig. 4 shows the simulation time for a 256×256 crossbar partitioned into various smaller crossbar sizes. The results show that there is an optimal partition size where the simulation time is minimized. In this case we obtain the minimum simulation time of 870 ms when we partition it into 32×32 crossbars, while the simulation time is 13.19 s for an equivalent unpartitioned simulation.

2.6 Batch Processing

Decomposing a matrix requires substantially longer time than solving the decomposed matrix, and the most common use case for simulating a crossbar is for a training of a validation set when performing neural architecture search. As such, simulating batches of inputs allows for the decomposition time to be amortized across a batch of solves. Hence, we decompose the matrix once and run the solver for multiple input samples.

Fig. 5 shows the results of batching multiple input data. For a 128×128 crossbar, the simulation time is 1 s without batching. With a batch size of 1000, the simulation time is 21.71 s, resulting in a per-solve speedup of $\frac{1000}{21.71} = 46X$. Fig. 6 shows the simulation time per input sample for various crossbar sizes. The simulation time per sample improves with increasing batch size. For example, with the 128×128 crossbar, the per-solve time reduces to 500 ms for a batch size of 2 and further decreases to 22 ms for a batch size of 1000.

3 FPGA Emulation

For FPGA deployment, the solver combines the forward substitution and the backward substitution required when solving an LU-decomposed matrix. The resulting sequence of operations is sequential, since the solution to each element of vector y, y_i, depends on the solution to y_{i-1} for all $i > 1$, and the solution to each element of the V vector, V_j, depends on V_{j+1} for all $j < 2 \times m \times n$, and the solution to $V_{2 \times m \times n}$ depends on the solution to $y_{2 \times m \times n}$. In essence, the elements of both vectors to be solved form a dependency chain. As such, there is little exploitable parallelism within an individual solve.

Fig. 7 shows the dataflow graph for solving a dense 4x4 system of equations given by a matrix that has already been decomposed into an $\boldsymbol{L}$ and $\boldsymbol{U}$ matrix. In the figure, the elements of the input I vector are read from off-chip memory, and the elements of the output V vector are written to off-chip memory. The elements of

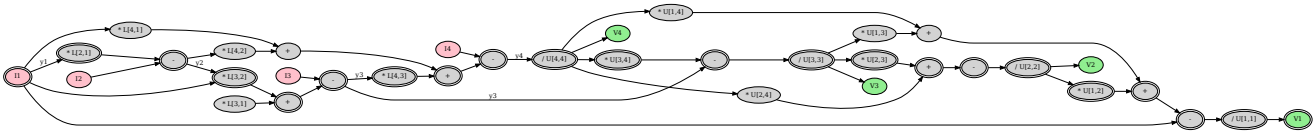


Figure 7: The dataflow graph for solving a dense 4×4 system of equations that is represented as a decomposed product of an L and a U matrix. The nodes containing the I vector are shown in pink and the node containing the V vector are shown in green. The nodes comprising the critical path are shown with a double border.

the $\boldsymbol{L}$ and $\boldsymbol{U}$ matrices are stored in banked on-chip RAM. In this case, most of the operations are part of the critical path, indicating that there are few opportunities for exploiting parallelism when solving a single system.

Despite the lack of parallelism inherent within a single cross solve, there is potential to exploit parallelism for a batch of solves. This is possible by constructing a solver pipeline that performs a complete solve in each pipeline traversal, where each solve corresponds to the crossbar input state.

Since the sparsity structure of the $\boldsymbol{L}$ and $\boldsymbol{U}$ matrices is fixed for a crossbar of a given size, we generate unrolled, straight-line code to implement the solver pipeline and allow the matrix structure to be hard-coded to avoid the overhead of accessing a generalized sparse matrix. Instead, the solver accesses the matrix values from predefined locations in the on-chip matrix RAM. Since both the $\boldsymbol{L}$ and $\boldsymbol{U}$ matrices are sparse, this also allows the code to omit any multiplies against a value of zero. The resulting loop body is synthesized to a pipeline.

When the matrix data is stored in a highly-parallel on-chip RAM, each pipeline traversal requires reading the I vector and writing the V vector, thus the minimum pipeline throughput is the inverse of the number of cycles required to read and write these vectors. Our design assumes a single 64-bit bidirectional memory port, giving a throughput of $\frac{1}{2 \times m \times n}$, or–stated another way–a pipeline initiation interval (II) of $2 \times m \times n$. The latency of the pipeline–or iteration latency (IL)–depends on the critical path of the solver's data flow graph. For a double precision pipeline, the IL can grow to thousands of cycles. For a solver batch size of B, the total execution time is $B \times II + IL$, and the time for a single solve is $II + \frac{IL}{B}$.

Table 3 shows the pipeline II, IL, and the number of double precision operations performed in each pipeline traversal for crossbar solver sizes of 2×2, 4×4, and 8×8. Additionally, the table lists the execution time for batch sizes of 1,000 and 10,000 for each solver and the resultant operations per cycle.

In practice, the Vitis HLS compiler is limited in the depth of the pipeline it can generate, and can only generate a pipeline having a maximum I vector size of 128, corresponding to a crossbar size of 8x8. For this reason, the solver must solve larger crossbars using partitioning as described below, storing one copy of the non-zero entries of the $\boldsymbol{L}$ and $\boldsymbol{U}$ matrices in an on-chip RAM. Additionally, since the number of partitions required for solving a given crossbar scales down at the same rate at which the execution time of solving a larger partition scales up, there is no benefit to deploying a partition solver of size greater than 2×2, corresponding to an 8×8 matrix.

Table 1: Vitis HLS compile time for LU decomposition across matrix sizes.

Xbar Size	Matrix Size	L Matrix nnz	U Matrix nnz	HLS Compile Time (sec)
2×2	8×8	21	21	16.87
4×4	32×32	147	147	141.76
8×8	128×128	1095	1095	19507.3

Table 2: Resource utilization and Fmax for different crossbar sizes.

Xbar Size	LUTs	FFs	DSPs	Fmax (MHz)
2×2	9997	18234	40	356
4×4	24790	62218	80	356
8×8	98222	243221	160	356

Table 1 shows the crossbar size, matrix sizes, number of non-zero entries in the corresponding $\boldsymbol{L}$ and $\boldsymbol{U}$ matrices, and HLS compile time for crossbar solvers of 2×2, 4×4, and 8×8. Table 2 shows the FPGA resource requirements for each solver, in LUTs, flip-flops (FFs), and DSPs, as well as the maximum clock frequency as reported by the HLS compiler.

4 Results

In this section we compare our LU-based crossbar solver running in Matlab to our FPGA-based LU solver. In both cases, we evaluate the performance impact of partitioning the crossbar solve.

Table 4 compares the performance of our proposed solver running on Matlab 2024a on an Intel Xeon Gold 5220R CPU against our proposed FPGA-based solver. Both were evaluated for crossbar sizes 4×4 to 128×128 and for batch sizes of 1,000 and 10,000. All reported execution times are normalized to a single solve (i.e., the total execution time is divided by the batch size). The CPU results reported are an average of 10 runs.

For the Matlab solver we chose the partition size that gave the best performance for each crossbar size and for each batch size. For the FPGA solver we used the 2×2 partition size. Since each partition requires a different $\boldsymbol{L}$ and $\boldsymbol{U}$ matrix, we also list the memory required to store all the matrices for a given crossbar size. The largest is 172,032 double precision values, or approximately 1344 KiB. Note that the embedded-class AMD Zynq Ultrascale+ FPGAs contain 0.8 to 10 MiB of on-chip memory depending on the device, with the

Table 3: Detailed HLS performance metrics across batch sizes and operations

Xbar Size	II	IL	DP mults	DP subs/ adds	DP divides	Batch size 1000			Batch size 10000		
						cycles	time(us)	ops/cycle	cycles	time(us)	ops/cycle
2×2	8	1041	26	29	8	9041	25.40	6.97	81041	227.64	7.77
4×4	32	5502	296	235	32	37502	105.34	15.01	325502	914.33	17.30
8×8	128	32889	1934	1943	128	160889	451.94	24.89	1312889	3687.89	30.51

Table 4: Comparison of Matlab- and FPGA-based Solver

Xbar size	Matlab				FPGA 2×2 partition					
	Batch size 1000		Batch size 10000		Batch size 1000		Batch size 10000		Number of partitions	BRAM requirement (fp64 values)
	Best partition size	Solve time	Best partition size	Solve time	Solve time	FPGA vs Matlab speedup	Solve time	FPGA vs Matlab speedup		
4×4	4×4	545 ns	4×4	331 ns	102 ns	5.34	91.1 ns	3.64	4	168
8×8	8×8	2.27 µs	8×8	782 ns	406 ns	5.59	364 ns	2.15	16	672
16×16	16×16	4.56 µs	16×16	3.06 µs	1.63 µs	2.80	1.46 µs	2.10	64	2688
32×32	32×32	14.1 µs	32×32	12.4 µs	6.50 µs	2.17	5.83 µs	2.13	256	10752
64×64	64×4	55.7 µs	64×4	49.7 µs	26.0 µs	2.14	23.3 µs	2.13	1024	43008
128×128	64×4	223 µs	32×4	198 µs	104 µs	2.14	93.2 µs	2.13	4096	172032

device on the most common development board, the ZCU104 board, containing 5.5 MiB of on-chip RAM. As shown, the FPGA-based solver with a 2×2 partition size delivers a 2.5X to 11.6X speedup over Matlab with its best partition size.

5 Conclusion

In this paper, we introduced XbarSim, a specialized simulation framework tailored for solving matrices associated with memristive crossbar nodal equations. To solve the matrices, we employ LU decomposition and crossbar partitioning. Our Matlab-based solver achieves significant performance improvements as compared to Hspice, and our FPGA-based solver achieved further improvement.

The speedups achieved by XbarSim provide several opportunities for future work, including extending XbarSim to support complete neural network simulations and integrating it with multi-objective optimization algorithms for optimizing ML workload mapping on crossbar-based IMC architectures.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 2409697.

References

[1] Md Hasibul Amin, Mohammed E Elbtity, and Ramtin Zand. 2022. Xbar-partitioning: a practical way for parasitics and noise tolerance in analog imc circuits. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 12, 4 (2022), 867–877.

[2] Md Hasibul Amin, Mohammed E. Elbtity, and Ramtin Zand. 2023. IMAC-Sim: A Circuit-level Simulator For In-Memory Analog Computing Architectures. In *Proceedings of the Great Lakes Symposium on VLSI 2023* (Knoxville, TN, USA) *(GLSVLSI '23)*. Association for Computing Machinery, New York, NY, USA, 659–664. doi:10.1145/3583781.3590264

[3] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R Stanley Williams, Paolo Faraboschi, Wen-mei W Hwu, John Paul Strachan, Kaushik Roy, et al. 2019. PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*. 715–731.

[4] Pai-Yu Chen, Xiaochen Peng, and Shimeng Yu. 2018. NeuroSim: A Circuit-Level Macro Model for Benchmarking Neuro-Inspired Architectures in Online Learning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 12 (2018), 3067–3080. doi:10.1109/TCAD.2018.2789723

[5] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 27–39.

[6] Mohammed Elbtity, Abhishek Singh, Brendan Reidy, Xiaochen Guo, and Ramtin Zand. 2021. An In-Memory Analog Computing Co-Processor for Energy-Efficient CNN Inference on Mobile Devices. *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2021* (2021).

[7] Mohammed E Elbtity, Brendan Reidy, Md Hasibul Amin, and Ramtin Zand. 2023. Heterogeneous integration of in-memory analog computing architectures with tensor processing units. In *Proceedings of the Great Lakes Symposium on VLSI 2023*. 607–612.

[8] Ben Feinberg, T. Patrick Xiao, Curtis J. Brinker, Christopher H. Bennett, Matthew J. Marinella, and Sapan Agarwal. [n. d.]. CrossSim: accuracy simulation of analog in-memory computing. ([n. d.]). https://github.com/sandialabs/cross-sim

[9] Miao Hu, John Paul Strachan, Zhiyong Li, Emmanuelle M. Grafals, Noraica Davila, Catherine Graves, Sity Lam, Ning Ge, Jianhua Joshua Yang, and R. Stanley Williams. 2016. Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. doi:10.1145/2897937.2898010

[10] Shubham Jain, Abhronil Sengupta, Kaushik Roy, and Anand Raghunathan. 2021. RxNN: A Framework for Evaluating Deep Neural Networks on Resistive Crossbars. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40, 2 (2021), 326–338. doi:10.1109/TCAD.2020.3000185

[11] Dovydas Joksas and Adnan Mehonic. 2020. badcrossbar: A Python tool for computing and plotting currents and voltages in passive crossbar arrays. *SoftwareX* 12 (2020), 100617. doi:10.1016/j.softx.2020.100617

[12] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. 2017. PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 541–552. doi:10.1109/HPCA.2017.55

[13] Lixue Xia, Boxun Li, Tianqi Tang, Peng Gu, Xiling Yin, Wenqin Huangfu, Pai-Yu Chen, Shimeng Yu, Yu Cao, Yu Wang, Yuan Xie, and Huazhong Yang. 2016. MNSIM: Simulation platform for memristor-based neuromorphic computing system. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. 469–474.

[14] Fan Zhang and Miao Hu. 2020. CCCS: Customized SPICE-level Crossbar-array Circuit Simulator for In-Memory Computing. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–8.