Enabling Scientific Computing on Memristive Accelerators

Ben Feinberg*, Uday Kumar Reddy Vengalam*, Nathan Whitehair*, Shibo Wang† and Engin Ipek*†
*Department of Electrical and Computer Engineering
†Department of Computer Science
University of Rochester
Rochester, NY 14627 USA
*{bfeinber, uvengala, nwhiteha}@ece.rochester.edu †{swang, ipek}@cs.rochester.edu

***Abstract*—Linear algebra is ubiquitous across virtually every field of science and engineering, from climate modeling to macroeconomics. This ubiquity makes linear algebra a prime candidate for hardware acceleration, which can improve both the run time and the energy efficiency of a wide range of scientific applications. Recent work on memristive hardware accelerators shows significant potential to speed up matrix-vector multiplication (MVM), a critical linear algebra kernel at the heart of neural network inference tasks. Regrettably, the proposed hardware is constrained to a narrow range of workloads: although the eight- to 16-bit computations afforded by memristive MVM accelerators are acceptable for machine learning, they are insufficient for scientific computing where high-precision floating point is the norm.**

This paper presents the first proposal to enable scientific computing on memristive crossbars. Three techniques are explored—reducing overheads by exploiting exponent range locality, early termination of fixed-point computation, and static operation scheduling—that together enable a fixed-point memristive accelerator to perform high-precision floating point without the exorbitant cost of naïve floating-point emulation on fixed-point hardware. A heterogeneous collection of crossbars with varying sizes is proposed to efficiently handle sparse matrices, and an algorithm for mapping the dense subblocks of a sparse matrix to an appropriate set of crossbars is investigated. The accelerator can be combined with existing GPU-based systems to handle datasets that cannot be efficiently handled by the memristive accelerator alone. The proposed optimizations permit the memristive MVM concept to be applied to a wide range of problem domains, respectively improving the execution time and energy dissipation of sparse linear solvers by 10.3x and 10.9x over a purely GPU-based system.

***Keywords*-Accelerator Architectures; Resistive RAM.**

I. Introduction

Computational models of dynamic systems are essential to our understanding of the world across a diverse set of problem domains, including high-energy physics [1], weather and climate modeling [2], biology [3], and even macroeconomics [4]. Achieving high fidelity using these models often demands immense computing power, potentially requiring thousands of nodes running over periods of weeks or months [5]. Enabling future advances in these problem domains requires significant improvements to computing efficiency on frequently used kernels. One important source of such kernels is linear algebra, which applies to nearly every domain of science and engineering [6].

Recently, a new class of accelerators based on memristive *in-situ* computation has been proposed [7]–[11]. These accelerators show significant potential for speeding up matrix-vector multiplication (MVM) in the context of connectionist machine learning models, such as deep neural networks and the Boltzmann Machine. Not only does *in-situ* MVM reduce the rapidly increasing cost of data movement [12], but it also allows for substantial parallelism by exploiting analog computation. Although promising, existing memristive accelerator proposals rely on the lax precision requirements of machine learning workloads to side-step the issue of precise computation, which makes these accelerators inapplicable to scientific computing.

This paper shows that a memristive accelerator can be architected to arbitrary precision requirements, preserving significant performance and efficiency gains over a GPU baseline while also meeting the particular needs of scientific workloads. Three techniques are proposed to perform floating-point computation with the same precision as IEEE-754 on the fundamentally fixed-point hardware of a memristive crossbar: 1) the overhead of floating- to fixed-point conversion is reduced by exploiting dynamic range locality; 2) each fixed-point computation is terminated as soon as the precision requirements of IEEE-754 have been satisfied; and 3) operations are scheduled statically to compute only the necessary bits. Although fixed-point emulation of floating point is not a new concept, and is often used on systems lacking floating-point hardware support [13], without the proposed optimizations it imposes a prohibitive throughput penalty on *in-situ* MVM.

This paper also addresses the problem of efficiently computing *sparse* MVM on memristive crossbars. The matrices generated by scientific applications are typically sparse [14]. Software optimizations that target sparse matrices have been widely explored, and the literature contains numerous solutions depending on the behavior of the underlying hardware [15]. These techniques, however, are not applicable to MVM performed using a memristive crossbar, because these techniques rely on multiple levels of indirection that have no direct analog in *in-situ* linear algebra. To meet this chal-

lenge, a novel heterogeneous hardware substrate comprising crossbars with different sizes is proposed. A preprocessing step maps the dense subblocks within a sparse matrix onto the proposed substrate by exploiting the interlocking trade-offs among crossbar size, throughput, matrix density, and blocking efficiency.

The proposed techniques combine with existing GPU-based approaches to accelerating high performance linear algebra. Some matrices—due to their sparsity patterns—cannot be mapped efficiently to even a heterogeneous substrate during the preprocessing step; in those rare cases, the computation is performed on a GPU instead. Notably, the choice between the accelerator and the GPU can be made quickly, based on the output of the preprocessing step.

The proposed system is evaluated on two high-performance iterative solvers with a set of 20 input matrices from the SuiteSparse collection [14], representing problem domains within computational fluid dynamics, structural analysis, circuit analysis, and seven others. Taken together, the proposed techniques improve the execution time by $10.3\times$ and reduce the energy consumption by $10.9\times$ over a conventional GPU platform.

II. Background

The proposed accelerator builds upon prior work on memristive accelerators for machine learning, and linear algebra for scientific computing.

A. Memristive Accelerators for Machine Learning and Graph Processing

The increasing prominence of machine learning workloads in recent years has led to many proposals for dedicated machine learning accelerators, both in academia [16] and industry [17]. These accelerators focus on the MVM operation as the primary kernel in connectionist machine learning models. More recently, a new class of MVM accelerators has been proposed that exploits the inherent parallelism of analog computation to perform MVM with memristive networks [7]–[11], [18]. Conceptually, a matrix is mapped onto a memristive crossbar such that the conductance of each crossbar cell is proportional to the corresponding matrix coefficient. For instance, in a binary matrix, **0**s and **1**s are respectively encoded as the low and high conductance states of the memristive devices. With these values mapped, a dot product operation is performed by applying voltages to each row, and quantizing the output current at the columns[1], as shown in Figure 1. Given a memristor at the intersection of row i and column j with resistance $R_{i,j}$, and row voltage V_i, the current through the memristor is $V_i/R_{i,j}$; the total current flowing through the column is $\sum_i V_i/R_{i,j}$, which is proportional to the dot product of the voltages and the conductances.

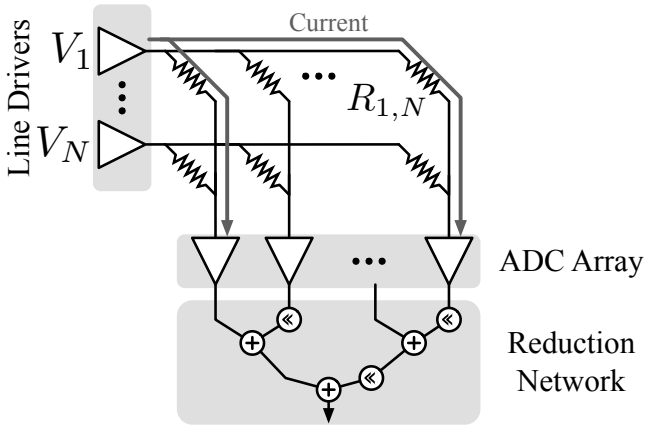


Figure 1. A conceptual example of memristive crossbars for computation.

This conceptual example, however, assumes high-precision analog circuits and many-bit memristive devices, both of which incur substantial overheads. To reduce these overheads, prior work has employed *bit slicing*, a technique to map each element of a matrix to multiple crossbars, reducing the required memristor resolution [7]–[10]. An example of bit slicing is shown in Equation 1, where each 3-bit matrix coefficient is mapped onto three binary crossbars. Note that while this example uses single-bit cells, the technique can be generalized to multi-bit cells.

$$\begin{bmatrix} 3 & 1 \\ 6 & 5 \end{bmatrix} = 2^2 \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} + 2^1 \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} + 2^0 \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad (1)$$

To perform computation on the full operand, the partial dot products from all of the bit slices are combined through a shift-and-add reduction network [7], [8]; an example is shown on the right hand side of Figure 1. The same bit slicing technique is applied to the vector as well as the matrix.

B. High-Performance Linear Algebra for Scientific Computing

Complex physical models are often described by systems of partial differential equations (PDEs). There are no known methods for finding analytical solutions to general systems of PDEs. Thus, these continuous PDEs are typically discretized into a system of linear equations, $A\mathbf{x} = \mathbf{b}$, representing a mesh of points that approximate the original model [19]. The resulting system of linear equations is often sparse; that is, most of the variables have zero contribution to most of the linear equations. Conceptually, this sparsity is a result of the locality inherent in the physical system. Variables in each linear equation represent the relationship between given points in the discretized mesh, and nearby points tend to have stronger relationships.

A sparse linear system can be solved using either direct or iterative methods. Direct methods include factorizations such

[1]When describing hardware, we use the memory systems convention for the terms *column* and *row*, *i.e.*, the rows of a matrix are mapped to the columns of the crossbars. To minimize confusion, we explicitly refer to rows within a matrix as "matrix rows."

as LU or Cholesky, as implemented in the High Performance LINPACK benchmark [20]. These direct methods can result in significant *fill-in*, where zero entries become non-zeroes; this increases the memory footprint and reduces the benefits of storing the matrix in a sparse format. Iterative methods, by contrast, do not involve modifying the underlying matrix. This results in significantly lower memory requirements, enabling the full sparse matrix to fit in memory whereas a sparse version with significant fill-in might not.

Iterative methods are subdivided into stationary and Krylov subspace methods. We focus on the latter since they are the primary methods used today [19]. Krylov subspace methods iteratively improve an estimated solution $\hat{\mathbf{x}}$ to the linear system $A\mathbf{x} = \mathbf{b}$. These methods are typically implemented using a stopping tolerance, ϵ, and terminate when $|\mathbf{b} - A\hat{\mathbf{x}}| < \epsilon$. There are many Krylov subspace solvers with different behaviors depending on the matrix structure, including conjugate gradient (CG) for symmetric positive definite matrices (SPD) [21], as well as BiConjugate Gradient (BiCG), Stabilized BiCG (BiCG-STAB) [22], and Generalized Minimal Residual [23] for non-SPD matrices.

Due to the importance of PDEs and the wide applicability of sparse solvers, a number of accelerator systems leveraging FPGAs and ASICs have been proposed. Kung *et al.* [24] propose an SRAM based processing in memory accelerator for simulating systems of differential equations. The accelerator uses 32-bit fixed-point, and does not meet the high-precision floating-point requirements of mainstream scientific applications (a key focus of our work). Additionally, Kung *et al.* accelerates the cellular nonlinear network model, which differs significantly from mainstream methods for solving PDEs. By contrast, we target widely accepted iterative solvers such as CG. Zhu *et al.* [25] discuss a graph processing accelerator that leverages content addressable memories (CAMs). This accelerator may be applicable to sparse MVM; however, the proposed hardware heavily depends on the sparsity of the input vector (less than 1% non-zeros) to limit CAM size. This vector sparsity is uncommon in iterative solvers: an analysis of the linear systems that we evaluate show vector densities ranging from 30-100%, which would result in prohibitive CAM overheads. Kestur *et al.* [26] and Fowers *et al.* [27] propose FPGA based accelerators that target sparse MVM for iterative solvers; however, the proposed approaches do not outperform a GPU. Dorrance *et al.* [28] propose an FPGA accelerator that achieves an average speedup of $1.28\times$ over a GTX TITAN GPU on single-precision sparse MVM; however, it is not clear how the FPGA would scale to double-precision. By contrast, the proposed memristive accelerator achieves an average speedup of $10.3\times$ over a Tesla P100 GPU on double-precision sparse MVM.

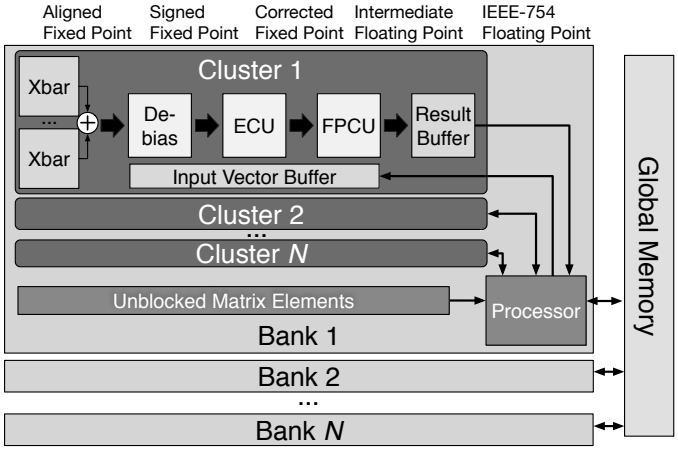


Figure 2. System overview.

III. System Organization

The accelerator is organized in a hierarchy of banks and clusters as shown in Figure 2. Each bank comprises a heterogeneous set of clusters with different crossbar sizes, as well as a local processor that orchestrates computation and performs the operations not handled by the crossbars. Within a cluster, a group of crossbars perform sparse MVM on a fixed-size matrix block.

A. Banks

Unlike prior work on memristive accelerators [7], [8], the clusters within each bank are of different sizes. This allows the system to capture dense patterns within sparse matrices, as discussed in Section V. With sparse matrices, some elements are ill-suited to the memristive crossbars, either because the elements do not block efficiently, or because the computation requires more bits than are provided by a cluster. In either case, the value is not mapped to a cluster and is instead handled by the local processor. Finally, cross-bank communication (for operations that require resources from multiple banks) is implemented through reads and writes to global memory.

To interface with the local processor, every cluster contains two memory-mapped SRAM buffers: an incoming vector buffer, and a result buffer. The incoming vector buffer holds the vector to be multiplied by the matrix in floating-point format, and extracts the bit slices from the vector as needed. The partial result buffer holds the running sum of the partial dot products in an intermediate floating-point format. Once the computation on a block completes, each scalar in the partial result buffer is converted to a final IEEE-754 representation, which is read by the local processor.

B. Clusters

A cluster comprises a set of crossbars—similar to the mats in the Memristive Boltzmann Machine [7], or the *in-situ multiply accumulate* (IMA) units in ISAAC [8]—that perform MVM and reduction across all of the bit slices

within a single matrix block. Unlike prior work, clusters are designed for IEEE-754 compatible double-precision floating-point computation. Each cluster contains 127 crossbars with single-bit cells that are organized in a shift-and-add reduction tree. The double-precision floating-point coefficients of a matrix block are converted to 118-bit fixed-point operands comprising a 53-bit mantissa, one sign bit, and up to 64 bits of padding for mantissa alignment. This 118-bit value is then encoded with a nine-bit error-correcting AN code (as proposed in [29]) for a full operand width of up to 127 bits.

Figure 3 shows the structure of the crossbars in a cluster and how computation proceeds. Initially, a vector bit slice $\mathbf{x}[\mathbf{i}]$ is applied to all of the crossbars within a cluster, and the resulting dot product for each crossbar column is latched by a sample-and-hold circuit (1).[2] A set of ADCs (one per crossbar) then start scanning the values latched by the sample-and-hold units (2). The outputs of the ADCs are propagated through a shift-and-add reduction network (3), producing a single fixed-point value that corresponds to the dot product between a row of the matrix A and a single bit slice of the vector $\mathbf{x}$. The system is pipelined such that every cycle, a single fixed-point value is produced by the reduction network, and after every column has been quantized, a new vector bit slice is applied to all of the crossbars.

Once the fixed-point result has been computed, three additional operations are performed to convert it to an intermediate floating-point representation (Figure 2). First, the bias is removed, converting the value from unsigned to signed fixed point. Second, error correction is applied. Third, the running sum for the relevant vector element is loaded from the partial result buffer, the incoming partial dot product is added to the running sum, and the result is written back. After the running sum is read from the partial result buffer, it is inspected to determine whether the precision requirements of IEEE-754 have been met (in which case, no further vector bit slice computations are needed). If so, all of the crossbars in the cluster are signaled to skip quantizing the corresponding column in the remaining vector bit slice calculations.[3] The criteria for establishing whether the precision requirements of IEEE-754 have been met are discussed in the next section.

IV. Achieving High Precision

Unlike machine learning applications, scientific workloads require high precision [30]. To enable scientific computing with memristive accelerators, the system must meet these precision requirements through floating-point format support and error correction.

[2]Throughout the paper, we use $\mathbf{x}_j[i]$ to refer to the j^{th} coefficient of the i^{th} bit slice of vector $\mathbf{x}$.

[3]The total number vector bit slices is still bounded by the number of vector bit slices required by the worst case column.

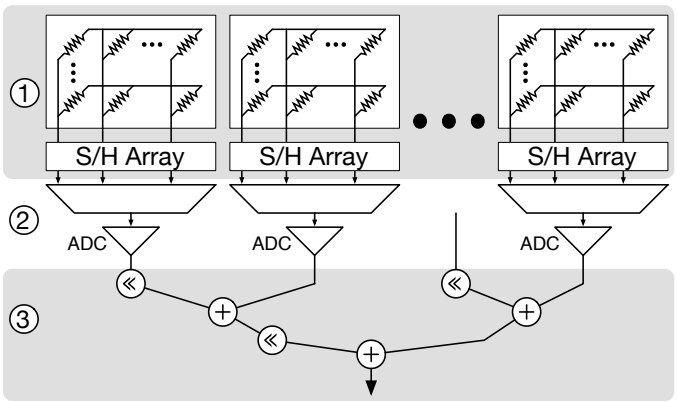


Figure 3. Cluster organization.

A. Floating-Point Computation on Fixed-Point Hardware

In double precision floating point, every number is represented by a 53-bit mantissa (including the implied leading **1**), an 11-bit exponent, and a sign bit [31]. When performing addition in a floating-point representation, the mantissas of the values must be aligned so that the bits of the same magnitude can be added. Consider as an example two base-ten floating-point numbers with two-digit mantissas, on which the computation $1.2 + 0.13$ is to be performed. To properly compute the sum, the values must be aligned based on the decimal point, $1.20+0.13$. In a conventional floating-point unit, this alignment is performed dynamically based on the exponent field of each value. On a memristive crossbar, however, since the addition occurs within the analog domain, the values must be converted to an aligned fixed-point format before they are programmed into the crossbar. This requires padding the floating point mantissas with zeros; in the example above, the addition would be represented as $120 + 013$ in fixed point, with an exponent of 10^{-2}.

Although this simple padding approach allows fixed-point hardware to perform floating-point operations, it comes at a significant cost. In double-precision, for instance, naïve padding requires 2100 bits in the worst case to fully account for the exponent range: 2046 bits for padding, 53 bits for the mantissa itself, and one sign bit. Not only is this requirement prohibitive from a storage perspective, but full fixed-point computation would require multiplying each matrix bit slice by each vector bit slice for a worst case of 4.4 million crossbar operations. Fortunately, two important properties of floating-point numbers can be exploited to reduce the cost of the fixed-point computation. First, while the IEEE-754 standard provides 11 bits for the exponent in double precision, few applications ever approach the full exponent range. Second, the naïve fixed-point computation described above provides more precision than IEEE-754, as it computes bits well beyond the 53-bit mantissa. These extraneous values are subsequently truncated when the result is converted back to floating point.

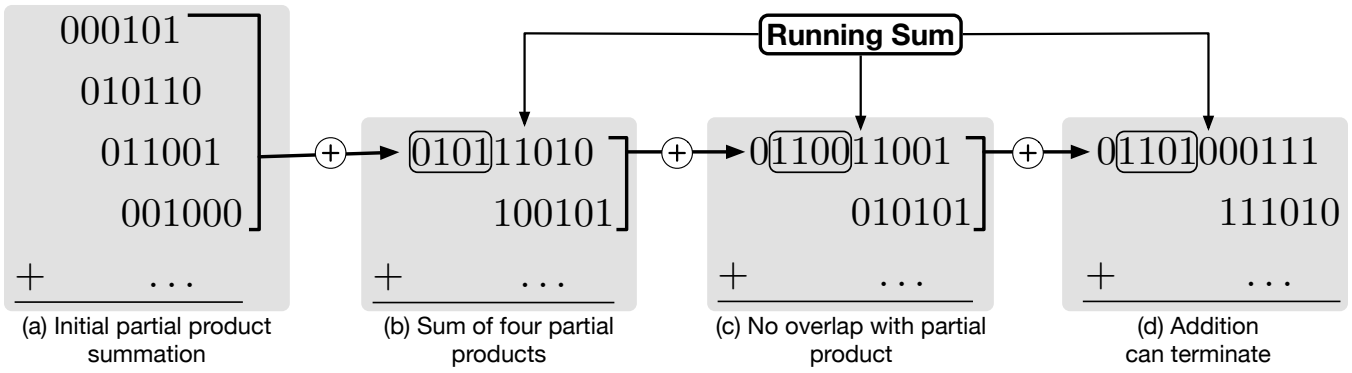


Figure 4. Illustrative example of partial product summation: a) addition of the first four partial products; b) overlap between the running sum and the mantissa; c) carries can still modify the running sum; d) the mantissa bits are settled.

B. Avoiding Extraneous Calculations

Three techniques are proposed to reduce the cost of performing floating-point computation on fixed-point hardware.

Exploiting exponent range locality. The 11-bit exponent field specified by IEEE-754 means that the dynamic range of floating-point values is $2^{2^{11}-2}$ (two exponent fields are reserved for special values), or approximately 10^{616}. (For comparison, the size of the state space for the game of Go is only 10^{172} [32].) IEEE-754 provides this range so that applications from a wide variety of domains can use floating-point values without additional scaling; however, it is highly unlikely that any model of a physical system will have a dynamic range requirement close to 10^{616}. In practice, the number of required pad bits is equal to the difference between the minimum and maximum exponents in the matrix, and can be covered by a few hundred bits rather than 2046. Furthermore, since the fixed-point overhead is required only for numbers that are to be summed in the analog domain, the alignment overhead can be further reduced by exploiting locality. Since the matrix is much larger than a single crossbar, it is necessarily split into blocks (discussed in Section V-B1); only those values within the same block are added in the analog domain and require alignment. The matrices can be expected to exhibit locality because they often come from physical systems where a large dynamic range between neighboring points is unlikely.

The aforementioned alignment procedure is also applied to the vector to be multiplied by the matrix. The alignment of mantissas encodes the exponent of each vector element relative to a single exponent assigned to the vector as a whole. Notably, multiplication does not require alignment since it is sufficient to add the relevant exponents of the matrix block, the vector, and the exponent offset implied by the location of the leading **1** in each dot product.

Early termination. In a conventional double-precision floating point unit (FPU), each arithmetic operation is performed on two operands, after which the resulting mantissa is rounded and truncated to 53 bits. To compute a dot product between two vectors, $\mathbf{a} \cdot \mathbf{x}$, the FPU first calculates the product of two scalars, $a_1 x_1$, rounds and truncates the result to 53 bits, and adds the truncated product to subsequent products: $\mathbf{a} \cdot \mathbf{x} = a_1 x_1 + a_2 x_2 + \cdots$. A memristive MVM accelerator, on the other hand, computes a dot product as an aggregation of partial dot products calculated over multiple bit slices. This necessitates a truncation strategy different from that of a digital FPU.

A naïve approach to truncation on a memristive MVM accelerator would be to perform all of the bit sliced matrix-vector multiplications, aggregate the partial dot products, and truncate the result. This naïve approach would require a total of 127×127 crossbar operations, since every bit slice of $\mathbf{a}$ must be multiplied by every bit slice of $\mathbf{x}$. [4] However, many of these operations would contribute only to the portion of the mantissa beyond the point of truncation, wasting time and energy. By detecting the position of the *leading* **1**—*i.e.*, the **1** at the most significant bit position in the final result—the computation can be safely terminated as soon as the 52 following bits of the mantissa have settled.

Consider the simplified example depicted in Figure 4, in which a series of partial products with different bit-slice weights are to be added (a). To clarify the exposition, the example assumes that the partial products are six bits wide, and that the final sum is to be represented by a four-bit mantissa and an exponent. (The actual crossbars sum 127-bit partial products, encoding the result with a 53-bit mantissa and an exponent.) In the example, the addition of partial products proceeds iteratively from the most significant partial product toward the least significant, accumulating the result in a *running sum*. The goal is to determine whether this iterative process can be terminated early while still producing the same mantissa as the naïve approach.

Figure 4b shows the running sum after the summation of the four most significant partial products. Since the next

[4] Recall from Section III that scalars are encoded with up to 127-bit fixed point. All examples in this section assume a 127-bit representation for simplicity, unless otherwise noted.

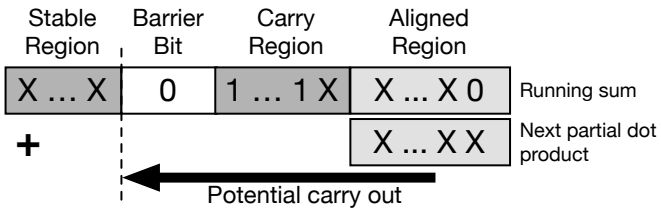


Figure 5. Regions within a running sum.

partial product still overlaps with the four-bit mantissa, it is still possible for the mantissa to be affected by subsequent additions. At a minimum, the computation must proceed until the mantissa in the running sum has cleared the overlap with the next partial product. This occurs in Figure 4c; however, at that point, it is still possible for a carry from the subsequent partial products to propagate into and modify the mantissa. Notably, if the accumulation is halted at any point, the sum of the remaining partial products would produce at most one carry out into the running sum. Hence, in addition to clearing the overlap, safe termination requires that a **0** less significant than the mantissa be generated, such that it will absorb the single potential carry out, preventing any bit flips within the mantissa (Figure 4d).

At any given time, the running sum can be split into four non-overlapping regions, as shown in Figure 5. The *aligned region* is the portion of the running sum that overlaps with the subsequent partial products. A chain of **1**s more significant than the aligned region is called the *carry region* as it can propagate the single generated carry out forward. This carry, if present, is absorbed by a *barrier bit*, protecting the more significant bits in the *stable region* from bit flips. **Thus, the running sum calculation can be terminated without loss of mantissa precision as soon as the full mantissa is contained in the stable region.**

Scheduling array activations. Even with the early termination technique discussed above, there is still a possibility for extraneous crossbar operations to be performed. Although each partial dot product takes up to 127 bits, the final mantissa is truncated to 52 bits following the leading **1**, requiring fewer than 127 bits to be computed in most cases. Therefore, some subset of the 127-bit partial dot product may be wasted depending on the schedule according to which vector bit slices are applied to the crossbars.

Figure 6 shows three approaches to scheduling the crossbar activations. In the figure, rows and columns respectively represent the matrix and vector bit slices, sorted by significance. The numbers at the intersection of each row and column represent the significance of the resulting partial product. Different groupings indicate which bit-sliced matrix-vector multiplications are scheduled to occur simultaneously; the black colored groupings must be performed, whereas the gray colored groupings may be skipped due to early termination. Hence, the number of black groupings

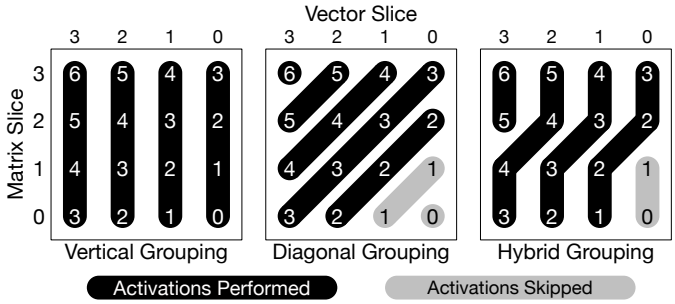


Figure 6. Three policies for scheduling crossbar activations.

correlates with execution time, and the number of crossbars enclosed by the black groupings correlates with crossbar activation energy. The example shown in the figure assumes that the calculation can be terminated early, after the fifth most significant bit of the mantissa is computed. (That is, all groupings computing a partial product with significance of at least **2** must be performed.)

In the leftmost case, all of the crossbars are activated with the same vector bit slice, after which the next significant vector bit slice is applied to the crossbars. This naïve *vertical* grouping requires 16 crossbar activations, performed over four time steps. In the next example, arrays are activated with different vector bit slices, such that each active array is contributing to the same slice of the output vector simultaneously. This *diagonal* grouping results in 13 crossbar activations—the minimum required in the example—over five time steps. In general, this policy takes the minimum number of crossbar activations, at the cost of additional latency.

The *hybrid* grouping shown on the right, which we adopt in the evaluation, strikes a balance between the respective energy and latency advantages of the diagonal and vertical groupings. The result is 14 crossbar activations performed over four time steps. The more closely the hybrid grouping approximates a diagonal grouping, the greater the energy savings at the cost of latency.

Taken together, the above techniques significantly reduce the overhead of performing floating-point computation on fixed-point hardware by restricting computations to only those that are likely to produce a bit needed for the final mantissa. The techniques render the latency and energy consumption strongly data dependent, effectively forming a data-specific subset of the floating point format without any loss in precision.

C. Handling Negative Numbers

The previously proposed ISAAC memristive accelerator [8] uses a biasing scheme to handle negative numbers. The approach adds a bias of 2^{16} to each 16-bit operand, such that the values range from 0 to 2^{16} rather than $\pm 2^{15}$. Results are de-biased by subtracting $2^{16} p_v$, where p_v is the population count (*i.e.*, the number of bits set to **1**) of the relevant bit slice of the vector. We adopt this technique with

one modification: in place of the constant 2^{16}, we use a per-block biasing constant based on the actual floating-point exponent range contained within the block.

D. Floating Point Anomalies and Rounding Modes

The IEEE-754 floating point standard specifies a number of floating point exceptions and rounding modes that must be handled correctly for proper operation [31].

Rounding modes. When the proposed accelerator is operated in its typical configuration, mantissa alignment and leading **1** detection result in a truncation of the result to the required precision. The potential contributions of any uncalculated bits beyond the least-significant bit would only result in a mantissa of greater magnitude than what is ultimately obtained, and since a biasing scheme is used to enable the representation of negative values (Section IV-C), the truncation is equivalent to rounding the result of a dot product toward negative infinity. For applications requiring other rounding modes (*e.g.*, to nearest, toward positive infinity, toward zero), the accelerator can be configured to compute three additional settled bits before truncation, and perform the rounding according to those bits.

Infinities and NaNs. Signed infinities and not-a-numbers (NaNs) are valid floating point values; however, they cannot be mapped to the crossbars in a meaningful way in the proposed accelerator, nor can they be included in any dot product computations without producing a NaN, infinity, or invalid result [31]. Therefore, the accelerator requires all input matrices and vectors to contain no infinities or NaNs, and any infinities or NaNs arising from an intermediate computation in the local processor are handled there to prevent them from propagating to the resulting dot product.

Floating point exceptions. Operations with floating point values may generally lead to underflow, overflow, invalid operation, and inexact exceptions [31]. Since *in-situ* dot products are performed on mantissas aligned with respect to relative exponents, there are no upper or lower bounds on resulting values. (Only the *range* of exponents is bounded.) This precludes the possibility of an overflow or underflow occurring while values are in an intermediate representation. However, when the result of a dot product is converted from the intermediate floating point representation into IEEE-754, it is possible that the required exponent will be outside of the available range. In this case, the exponent field of the resulting value will be respectively set to all **1**s or all **0**s for overflow and underflow conditions. Similarly to CUDA, computations resulting in an overflow, underflow, or inexact arithmetic do not cause a trap [33]. Since the accelerator can only accept matrices and input vectors with finite numerical values, and since the memristive substrate is used only for computing dot products, invalid operations (*e.g.*, 0 / 0, $\sqrt{-|c|}$) and operations resulting in NaNs or infinities can only occur within the local processor, where they are handled through an IEEE-754 compliant floating point unit.

E. Detecting and Correcting Errors

Prior work by Feinberg *et al.* proposes an error-correction scheme for memristive neural network accelerators based on AN codes [29]. We adopt this scheme with three modifications, taking into consideration the differences between neural networks and scientific computing. First, as noted in Section I, scientific computing has higher precision requirements than neural networks; thus, the proposed accelerator uses only 1-bit cells, making it more robust. Second, the matrices for scientific computing are typically sparse, whereas the matrices in neural networks tend to be dense; this leads to lower error rates using the RTN-based error model of Feinberg *et al.*, allowing errors to be corrected with greater than 99.99% accuracy. However, the use of arrays larger than used in prior work introduces new sources of error. If the array size is comparable to the dynamic range of the memristors, the non-zero current in the R_{HI} state may cause errors when multiplying a low density matrix row with a high density vector. To avert this problem, we limit the maximum crossbar block size in a bank (discussed in the next section) to 512×512, with a memristor dynamic range of 1.5×10^3. Third, since the floating-point operands expand to as many as 118 bits during the conversion from floating- to fixed point, we do not use multi-operand coding, but rather apply an $A = 251$ code that protects a 118-bit operand with eight bits for correction and one bit for detection. The correction is applied after the reduction but before the leading-one detection.

V. Handling and Exploiting Sparsity

An *in-situ* MVM system has an inherent advantage when performing computation with large and dense matrices, as the degree of parallelism obtained by computing in the analog domain is proportional to the number of elements in the matrix, up to the size of the matrix, $M \times N$. As noted in Section II-B, however, the majority of the matrices of interest are sparse. This reduces the effective parallelism, making it a challenge to achieve high throughput.

A. Tradeoffs in Crossbar Sizing

The potential performance of a memristive MVM system is constrained by the sizes of its crossbars: as crossbar size increases, so does the peak throughput of the system. This peak is rarely achieved, however, since the sparsity pattern of each matrix also limits the number of elements that may be mapped to a given block.

We define the throughput of the system as the number of effective element-wise operations in a single-cluster MVM computation divided by the latency of that computation,

τ_{MVM}. As the size of a crossbar increases, greater throughput is naïvely implied, since the size of the matrix block representable within the crossbar—and, consequently, the number of sums-of-products resulting from a cluster operation—also increases. However, since only non-zero matrix elements contribute to an MVM operation, effective throughput increases only when the number of non-zero (NNZ) elements increases, where NNZ depends on both block size, $M \times N$, and block density, d_{block}. Actual throughput for a given block is therefore $d_{block} \times M \times N / \tau_{MVM}$. Thus, to achieve high throughput and efficiency, large and dense blocks are generally preferable to small or sparse blocks.

The number of non-zero elements actually mapped to a crossbar is, in practice, far less than the number of cells in the crossbar, and the trend is for large blocks to be sparser than small blocks. This tendency follows intuitively from the notion that as the block size approaches the size of the matrix, the density of the block approaches the density of the matrix—0.37% for the densest matrix in the evaluation.

The latency of a memristive MVM operation is also directly related to the size of a crossbar. The maximum possible crossbar column output occurs when all N bits of a column and all N bits of the vector slice are set to 1, resulting in a sum-of-products of $\sum_1^N 1 \times 1 = N$. Therefore, a sufficient analog to digital converter (ADC) for a crossbar with N rows must generally have a minimum resolution of $\lceil \log_2[N+1] \rceil$ bits (though this can be reduced, as shown in Section V-B2). For the type of ADC used in the proposed system and in prior work [8], conversion time per column is strictly proportional to ADC resolution, which increases with $\log_2 N$ where N is the number of rows, and the number of ADC conversions per binary MVM operation is equal to the number of columns, M. Total conversion time is thus proportional to $M \lceil \log_2[N+1] \rceil$.

1) Energy: Crossbar sizing strongly affects the amount of energy used by both the array and the ADC during an MVM operation.

ADC energy. The average power dissipation of the ADC grows exponentially with resolution [34]–[36]. Since resolution is a logarithmic function of the number of rows, N, but conversion time is proportional to resolution, ADC energy per column conversion is approximately proportional to $N \log_2 N$. The column count, M, determines the number of ADC conversions per MVM operation, so the total ADC energy per operation is proportional to $M \times N \log_2 N$.

Crossbar energy. Crossbar latency is defined as the minimum amount of time during which the crossbars must be actively pulling current in order to ensure that all output nodes are within less than $\frac{\text{LSB}}{2}$ of their final voltages, thereby constraining all column outputs to be within the error tolerance range of the ADC. The worst-case path resistance and capacitance of an $M \times N$ crossbar are both proportional to $M + N$. The number of RC time constants necessary to ensure error-free output is proportional to resolution, or $\log_2 N$. While the crossbar is active, power is proportional to the total conductance of the crossbar, which can be found by multiplying the number of cells, $M \times N$, by the average path conductance, which is proportional to $\frac{1}{M+N}$. Crossbar energy for a single MVM operation grows with the product of crossbar latency and power, and is thus found to be proportional to $(M \times N)(M + N) \log_2 N$.

2) Area: As with ADC power, ADC area also scales exponentially with resolution [34]–[36], making average ADC area approximately proportional to N. Crossbar area is dominated by drivers; consequently, the total crossbar area grows as $M(M + N)$.

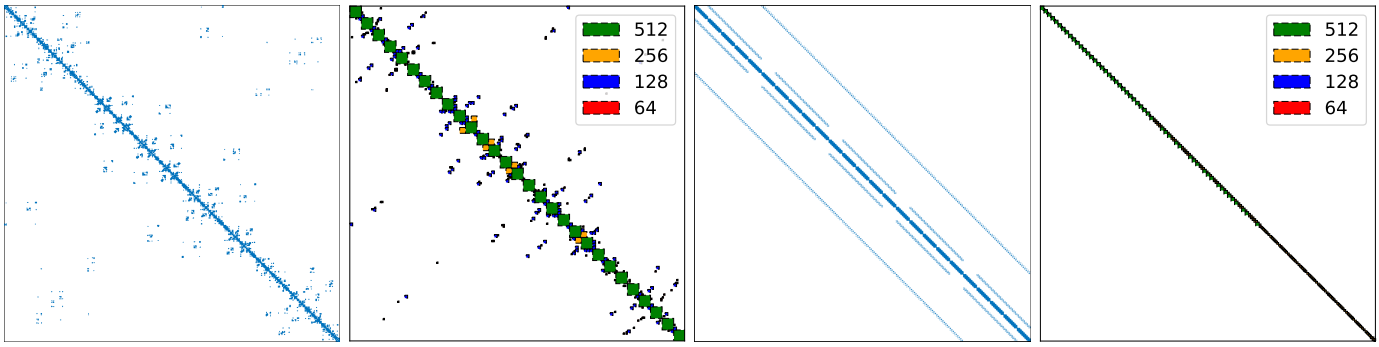
B. A Heterogeneous Crossbar Substrate

Crossbar sizing requires striking a careful balance between parallelism and energy efficiency: a crossbar must be large enough to capture sufficient non-zero elements to exploit parallelism, yet small enough to retain sufficient density for area and energy efficiency. By contrast to the single crossbar size used in prior work [7], [8], [18], the proposed system utilizes crossbars of different sizes, allowing blocks to be mapped to the crossbars best suited to the sparsity pattern of the matrix. This heterogeneous composition makes the system performance robust, yielding high performance and efficiency on most matrices without over-designing for a particular matrix type or sparsity pattern.

The problem of performance-optimal matrix blocking is neither new nor solved in a computationally feasible manner. Prior work seeks to block sparse matrices while maximizing throughput and minimizing fill-in, but focuses on block sizes that are too small to be efficient with memristive crossbars (12×12 at most [15]).

1) Mapping a Matrix to the Heterogeneous Substrate: A fast preprocessing step is proposed that efficiently maps an arbitrary matrix to the heterogeneous collection of clusters. The matrix is partitioned into block candidates, and for each candidate block the number of non-zero elements captured by the block and the range of the exponents within the block are calculated. If the exponent range of the block falls outside of the maximum allowable range (*i.e.*, 117), the elements are selectively removed until an acceptable range is attained. The number of non-zero elements remaining is then compared to a dimension-dependent threshold; if the number of elements exceeds the threshold, the candidate block is accepted. If the candidate block is rejected, a smaller block size is considered and the process repeats.

The preprocessing step considers every block size in the system (512×512, 256×256, 128×128, and 64×64) until each element has either been mapped to a block or found to be unblockable. Elements that cannot be blocked efficiently or that fall outside of the exponent range of an accepted block are completely removed from the mapping, and are instead processed separately by the local processor. Figure



(a) Pres_Poisson sparsity (b) Pres_Poisson blocking (c) Xenon1 sparsity (d) Xenon1 blocking

Figure 7. Sparsity and blocking patterns of two of the evaluated matrices.

7 shows the non-zero elements and the resulting blocking patterns for two sparse matrices.

Since the non-zero elements in the matrix are traversed at most one time for each block size, the worst-case complexity of the preprocessing step is $4 \times NNZ$. In practice, the early discovery of good blocks brings the average-case complexity to $1.8 \times NNZ$.

2) Exploiting Sparsity: Though sparsity introduces challenges in terms of throughput, it also presents some opportunities for optimization.

Computational invert coding. Prior work [8] introduces a technique for statically reducing ADC resolution requirements. If a crossbar column contains all **1**s, the maximum output for the column is equal to the number of rows, N. To handle this extreme case, the ADC would require $\lceil \log_2[N+1] \rceil$ bits of resolution. Eliminating this case reduces the ADC resolution requirement by one bit. This is achieved by inverting the contents of a column if it contains all **1**s. To obtain a result equivalent to a computation with a non-inverted column, the partial dot product is subtracted from the population count of the applied vector bit slice.

We extend this technique to *computational invert coding* (CIC), a computational analog to the widely used bus-invert coding [37]. If the number of ones in a binary matrix row is less than $\frac{N}{2}$, the required resolution of the ADC is again decreased by one bit. By inverting any column mapping that would result in greater than 50% bit density, a static maximum of $\frac{N}{2}$ **1**s per crossbar column is guaranteed. Notably, if the column has exactly $\frac{N}{2}$ **1**s, a resolution of $\log_2 N$ is still required. Since the proposed system allows for removing arbitrary elements from a block for processing on the local processor, this corner case can be eliminated. Thus, for any block, even in matrices approaching 100% density, the ADC resolution can be statically reduced to $\log_2[N]-1$ bits. CIC also lowers the maximum conductance of the crossbars (since at most half of the cells can be in the R_{LO} state), which in turn reduces crossbar energy.

ADC headstart. The ADCs that are commonly used in memristive accelerators [7], [8] perform a binary search over the space of possible results, with the MSb initially set to **1**. However, the maximum output of each column is constrained by the number of ones mapped to that column. Matrix rows tend to produce at most $\lceil \log_2[d_{row} \times 0.5 \times N] \rceil$ bits of output, assuming that **1**s and **0**s are equally likely. Since the actual number of output bits is data dependent, this tendency cannot be leveraged to statically reduce ADC precision beyond what is attainable with CIC. Instead, the ADC is pre-set to the maximum number of bits that can be produced by the subsequent conversion step. This approach allows the ADC to start from the most significant possible output bit position, and reduces the amount of time required to find the correct output value by an amount proportional to the number of bits skipped. Notably, this does not affect operation latency since reduction logic is fully synchronous and assumes a full clock period for conversion; however, by decreasing the time spent on conversion, the energy of the MVM operation is reduced.

VI. Programming the Accelerator

We focus on Krylov subspace solvers for specificity, as they are a widely used application involving sparse MVM; however, the proposed accelerator is not limited to Krylov subspace solvers. These solvers are built from three computational kernels: a sparse-matrix dense-vector multiply, a dense-vector sum or *AXPY* ($\mathbf{y} \leftarrow a\mathbf{x}+\mathbf{y}$), and a dense-vector dot product ($z = \mathbf{x} \cdot \mathbf{y}$). To split the computation between the banks, each bank is given a 1200-element section of the solution vector $\mathbf{x}$; only that bank performs updates on the corresponding section of the solution vector and the portions of the derived vectors. For simplicity, this restriction is not enforced by hardware and must be followed by the programmer for correctness.

On problems that are too large for a single accelerator, the MVM can be split in a manner analogous to the partitioning on GPUs: each accelerator handles a portion of the MVM, and the accelerators synchronize between iterations.

A. Kernel Implementation

To implement Krylov subspace solvers, we implement the three primary kernels needed for the solver.

1) Sparse MVM: The sparse MVM operation begins by reading the vector map for each cluster. The vector map contains a set of three-element tuples, each of which comprises the base address of a cluster input vector, the vector element index corresponding to that base address, and the size of the cluster. Since the vector applied to each cluster is contiguous, the tuple is used to compute and load all of the vector elements into the appropriate vector buffer. Thereafter, a start signal is written to a cluster status register and the computation begins. Since larger clusters tend to have a higher latency, the vector map entries are ordered by cluster size.

Once all cluster operations have started, the processor begins operating on the non-blocked entries. As noted in the previous section, some values within the sparse matrix are not suitable for mapping onto memristive hardware either because they cannot be blocked at a sufficient density, or because they would exceed the exponent range. These remaining values are stored in the compressed sparse row format [15].

When a cluster completes, an interrupt is raised and serviced by an interrupt service routine running on the local processor. Once all clusters signal completion and the local processor finishes processing the non-blocked elements, the bank has completed its portion of the sparse MVM. The local processors for different banks rely on a barrier synchronization scheme to determine when the entire sparse MVM is complete.

2) Vector Dot Product: To perform a vector dot product, each processor computes the relevant dot product from its own vector elements. Notably, due to vector element ownership of the solution vector $\mathbf{x}$ and all of the derived vectors, the dot product is performed using only those values that belong to a local processor. Once the processor computes its local dot product, the value is written back to shared memory for all other banks to see. Each bank computes the final product from the set of bank dot products individually to simplify synchronization.

3) AXPY: The AXPY operation is the simplest kernel since it can be performed purely locally. Each vector element is loaded, modified, and written back. Barrier synchronization is used to determine completion across all banks.

VII. Experimental Setup

We develop a parameterized and modular system model to evaluate throughput, energy efficiency, and area. Key parameters of the simulated system are listed in Table I.

A. Circuits and Devices

We model all of the CMOS based periphery at the 15nm technology node, using process parameters and design rules from FreePDK15 [38] coupled with the ASU Predictive Technology Model (PTM) [39]. TaOx memristor cells based on [40] are used, with linearity and dynamic range as reported by [18]. During computation, cells are modeled as resistors, with resistance determined either by a statistical approach considering block density or actual mapped bit values, depending on whether the model is being evaluated as part of a design-space exploration or to characterize performance with a particular matrix. The interconnect is modeled with a conservative lumped RC approach, using the wire capacitance parameters provided by the ASU PTM, derived from [41]. To reduce path distance and latency, we split the crossbars and place the drivers in the middle, rather than at the periphery, similarly to the double-sided ground biasing technique from [42].

Table I
Accelerator configuration

System	(128) banks, double-precision floating point, $f_{clk} = 1.2$GHz, 15nm process, $V_{DD} = 0.80$V
Bank	$(2) \times 512 \times 512$ clusters, $(4) \times 256 \times 256$ clusters, $(6) \times 128 \times 128$, $(8) \times 64 \times 64$ clusters, 1 LEON core
Cluster	127 bit slice crossbars
Crossbar	$N \times N$ cells, $(\log_2[N] - 1)$-bit pipelined SAR ADC, $2N$ drivers
Cell [18], [40]	TaOx, $R_{on} = 2$kΩ,$R_{off} = 3$MΩ, $V_{read} = 0.2$V, $V_{set} = -2.6$V, $V_{reset} = 2.6$V, $E_{write} = 3.91$nJ, $T_{write} = 50.88$ns

The peripheral circuitry includes $2N$ driver circuits, based on [43] and sized such that they are sufficient to source/sink the maximum current required to program and read the memristors. Also included are N sample-and-hold circuits based on [44], with power and area scaled for the required sampling frequency and precision.

A 1.2 GHz 10-bit pipelined SAR ADC [34] is used as a reference design point. 1.2 GHz is chosen as the operating frequency of the ADC in order to maximize throughput and efficiency while maintaining acceptable ADC signal to noise and distortion ratio (SNDR) at our supply voltage. Area, power, and the internal conversion time of each individual crossbar's ADC are scaled based on resolution, as discussed in Section V-A. Approximately 7% of the reported ADC power scales exponentially with resolution, with the remainder of power either scaling linearly or remaining constant, and 20% of the total power is assumed to be static based on the analysis reported in [34]. Similarly, approximately 23% of the reported area scales exponentially (due to the capacitive DAC and the track-and-hold circuit), and the remainder either scales linearly or remains constant. We hold the conversion time of the ADC constant with respect to resolution, rounding up to the nearest integral clock period. During the slack portion of the clock period (*i.e.*, the time between conversions), we reduce the ADC power dissipation to its static power.

The local processors are implemented with the open-source LEON3 core [45]. Since the open-source version of LEON3 does not include the netlist for the FPU for synthesis, we modify the FPU wrapper to the fused multiply add (FMA) timings for an FMA unit generated by FP-

Table II
EVALUATED MATRICES, SPD MATRICES ON TOP.

Matrix	NNZs	Rows	NNZ/Row	Blocked
2cubes_sphere	1647264	101492	16.2	49.7%
crystm03	583770	24696	23.6	94.7%
finan512	596992	74752	7.9	46.7%
G2_circuit	726674	150102	4.5	60.9%
nasasrb	2677324	54870	49.8	99.1%
Pres_Poisson	715804	14822	48.3	96.4%
qa8fm	1660579	66127	25.1	92.8%
ship_001	3896496	34920	111.6	66.4%
thermomech_TC	711558	102158	6.8	0.8%
Trefethen_20000	554466	20000	27.7	63.3%
ASIC_100K	940621	99340	9.5	60.9%
bcircuit	375558	68902	5.4	64.9%
epb3	463625	84617	5.5	72.2%
GaAsH6	3381809	61349	55.12	69.2%
ns3Da	1679599	20414	82	3.2%
Si34H36	5156379	97569	52.8	53.7%
torso2	1033473	115697	8.9	98.1%
venkat25	1717792	62424	27.5	79.8%
wang3	177168	26064	6.8	64.6%
xenon1	1181120	48600	24.3	81.0%

GEN [46]. The LEON3 core, FMA, and in-cluster reduction network are synthesized using the NanGate 15nm Open Cell Library [47] with the Synopsys Design Compiler [48]. SRAM buffers within each cluster and the eDRAM memory are modeled using CACTI7 [49] using 14nm eDRAM parameters from [50].

B. Architecture

We compare against an nVidia Tesla P100 GPU accelerator modeled using GPGPUSim [51] and GPGPUWattch [52]. To evaluate the performance of the LEON3 based local processors, we implement CG and BiCG-STAB in C and compile the implementations using the LEON3 Bare C Compiler with timers around each kernel call. Since all bank microprocessors must synchronize during each iteration, we evaluate the latency of the bank microprocessor with the largest number of unblocked elements.

In the worst case, the preprocessing step touches every non-zero in the matrix four times, which is comparable to performing four MVM operations. Thus, we conservatively assume that preprocessing takes time equivalent to four MVM operations on the baseline system.

C. Algorithms

We evaluate the proposed accelerator on 20 matrices from the SuiteSparse matrix collection [14] using CG for symmetric positive definite matrices (SPD) and BiCG-STAB for non-SPD matrices. The matrices were selected to showcase a range of matrix structures, sizes, and densities across a variety of applications. When available, we use the $\mathbf{b}$ vector provided by the collection; when $\mathbf{b}$ is unavailable we use a $\mathbf{b}$ vector of all $\mathbf{1}$s as in prior work [53].

The solvers running on the proposed accelerator converge in the same number of iterations as they do when running on the GPU, since both systems perform computation at the same level of precision.

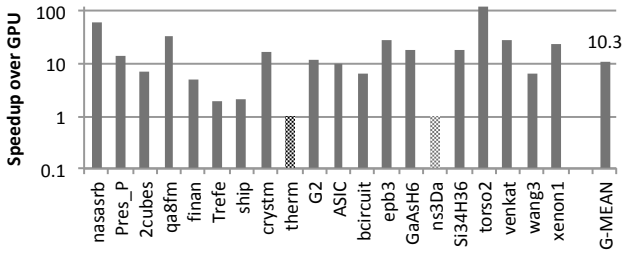


Figure 8. Speedup over the GPU baseline.

VIII. EVALUATION

We evaluate the system performance, energy, area, and robustness to different sources of error.

A. Execution Time

Figure 8 shows that the accelerator achieves an average speedup of $10.3\times$ as compared to the GPU baseline across the set of 20 matrices. One important insight from these results is that the average number of non-zeros per matrix row is a poor proxy for potential speedup. The matrix with the greatest speedup in the dataset (torso2) has just 8.9 NNZ per matrix row and the two quantum chemistry matrices (GaSsH6 and Si34H36) have more than 50 NNZ per row, showing above-average but not exceptional speedup due to their lower blocking efficiencies (*i.e.*, the percent of the non-zeros that are blocked): 69% and 53%, respectively. Since each bank microprocessor is designed to work in conjunction with its clusters, matrices with fewer rows perform worse than larger matrices at the same blocking efficiency. As expected, blocking efficiency is the most reliable predictor of speedup.

Notably in Table II, there are two matrices that are effectively unblocked by the proposed preprocessing step, thermotech_TC and ns3Da, with respective blocking efficiencies of 0.8% and 3.2%. Since the accelerator is optimized for *in-situ* rather than digital MVM, and since the majority of the elements in these two matrices cannot be mapped to the proposed heterogeneous substrate, the accelerator is more than an order of magnitude less efficient than the GPU baseline on these matrices. Rather than attempt to perform MVM on matrices so clearly ill-suited to the accelerator, we assume that the proposed accelerator co-exists with a GPU in the same node, and that the GPU may be used for the rare matrices which do not block effectively. Since the blocking algorithm has a worst-case performance of four MVM operations (Section V-B1), and since the actual complexity reaches the worst case if and when the matrix cannot be blocked, we can choose whether to perform the computation on the accelerator or the GPU after the blocking has completed. This results in a performance loss of less than

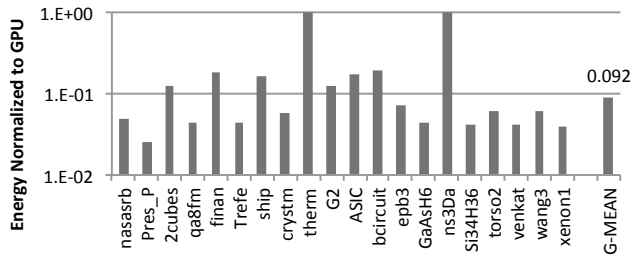


Figure 9. Accelerator energy consumption normalized to the GPU baseline.

3% for both matrices, far better than if the accelerator were used directly.

B. Energy Consumption

Energy consumption results are shown in Figure 9. Total energy consumption is improved by $14.2\times$ on the 18 matrices executed on the accelerator, and by $10.9\times$ over the entire 20-matrix dataset. The impacts of differing exponent ranges can be seen in the energy results. Notably, the nasasrb matrix has a 3% higher blocking efficiency than Pres_Poisson; however, Pres_Poisson shows nearly twice the improvement in energy dissipation; this is due to the much narrower exponent range of Pres_Poisson. Pres_Poisson never requires more than 14 bits of padding for storage, which also indicates a much narrower dynamic range for the computation, and by extension fewer vector bit slices that are needed per cluster. By contrast, nasasrb has multiple blocks with excluded elements (due to those elements requiring greater than 117 bits), and in general the average number of stored bits per cluster is 107 (30 more bits than the worst block of Pres_Poisson). This strong dependence on the exponent range indicates a potential benefit of the fixed-point computation over floating point. As discussed in Section IV-B, floating point is designed for broad compatibility and exceeds the requirements of many applications. By operating on fixed-point representations of data for much of the computation, the accelerator implicitly creates a problem-specific subset of the floating-point format with significant performance benefits, without discarding any information that would cause the final result to differ from that of an end-to-end floating-point calculation.

C. Area Footprint

The area footprint of the accelerator is computed using the area model described in Section VII-A. The overall system area for the 128-bank system described above is $539mm^2$, which is lower than the $610mm^2$ die size of the baseline Nvidia Tesla P100 GPU [54]. Notably, unlike in prior work on memristive accelerators, the crossbars and peripheral circuitry are the dominant area consumer—rather than the ADCs—with a total of 54.1% of total overall cluster area. This is due to two important differences from prior work: 1) due to computational invert coding (Section V-B2),

Table III
AREA, ENERGY, AND LATENCY OF DIFFERENT CROSSBAR SIZES (INCLUDES THE ADC).

Size	Area [mm2]	Energy [pJ]	Latency [nsec]
64	0.00078	28.0	53.3
128	0.00103	65.2	107
256	0.00162	150	213
512	0.00352	342	427

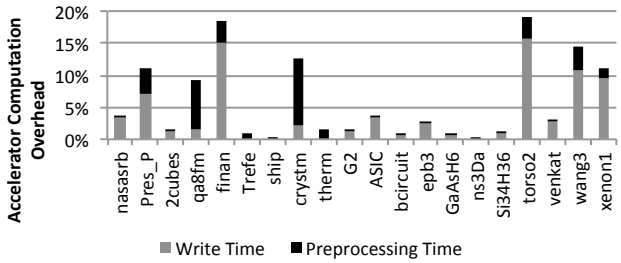


Figure 10. Overhead of preprocessing and write time as a percent of total solve time on the accelerator.

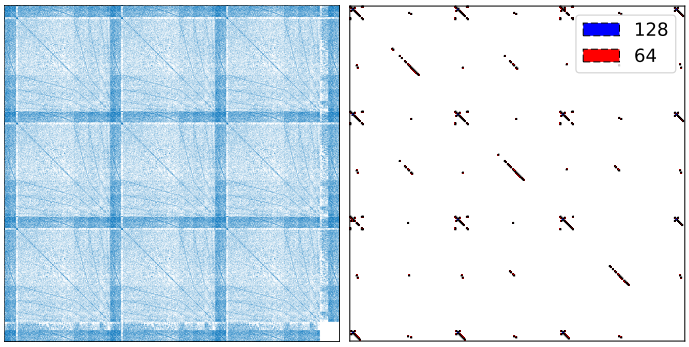
the ADC precision can be reduced by one bit, which has an exponential effect on ADC area; and 2) the wider operands that propagate through the reduction network. The per-bank processors and global memory buffer consume 13.6% of the total system area. This result suggests that a more powerful processor than the LEON3 core may be integrated based on application demands without substantially increasing system area. Table III summarizes the area as well as the latency and energy characteristics for different crossbar sizes used in different clusters.

D. Initialization Overhead

Figure 10 shows the overhead of setting up the iterative computation, including matrix preprocessing time and programming time, normalized to the entire solve time of each linear system on the accelerator. Since iterative algorithms take thousands of iterations to converge, the overhead of writing is effectively amortized, leading to an overhead of less than 20% across the entire matrix set and generally falling as the size of the linear system increases. By extension, the number of iterations required to converge increases faster than the write requirements of the array. For large linear systems (the most likely use case of the proposed accelerator), the overhead is typically less than 4% of the total solver runtime. The overhead may be even less in practice as multiple time steps are generally simulated to examine the time-evolution of the linear system. In these time-stepped computations, only a subset of non-zeros change each step, and the matrix structure is typically preserved, requiring minimal re-processing.

E. System Endurance

Although memristive devices have finite switching endurance, the iterative nature of the algorithms allows a single matrix to be programmed once and reused throughout the computation. Even under a conservative set of assumptions



(a) ns3Da sparsity (b) ns3Da blocking

Figure 11. Sparsity and blocking patterns of ns3Da.

where each array is fully rewritten for each new matrix, ignoring both the sparsity of the blocks and the time step behavior mentioned above, the system lifetime is sufficient: given a memristive device with a conservative 10^9 writes [55]–[57], the total system lifetime is over 100 years, assuming the system runs constantly and a new matrix is fully rewritten between solves.

F. Understanding Difficult-to-Block Matrices

Figure 11 shows the non-zero element distribution and the blocking pattern for ns3Da, a matrix with a particularly poor blocking efficiency. The figure shows that, despite the high relative density of the matrix, the values do not form dense sub-blocks that can be blocked efficiently. Instead, the values are distributed relatively uniformly over the matrix, with the third-highest NNZs per matrix row in the evaluated dataset. Figure 7 shows the non-zero distribution of two matrices that can be blocked effectively, Pres_Poisson and Xenon1; the non-zeros of both matrices are clustered primarily around the diagonal. An analysis of the blocks formed on ns3Da shows that most of the formed blocks capture patterns with non-zeros separated by approximately 30 elements: 2 per matrix block-row in 64 blocks, and 4 per matrix block-row in 128 blocks. This suggests that even if ns3Da were to be blocked more efficiently, the effective speedup would be constrained by the limited current summation from having few blocks per matrix row.

G. Sensitivity to Device Characteristics

Since scientific computing emphasizes high precision, we analyze the effects of device characteristics on system accuracy and convergence behavior.

Dynamic range. We re-evaluate the convergence behavior with various cell ranges (*i.e.*, on/off state ratios) to determine how the system behaves when the states are closer together or farther apart. Figure 12 shows the relative iteration count at various configuration points. The results show effectively no sensitivity to dynamic range for single-bit cells. With two-bit cells, the relatively low dynamic range results in some computational error, as the cell states are not sufficiently spread to tolerate the sources of error in

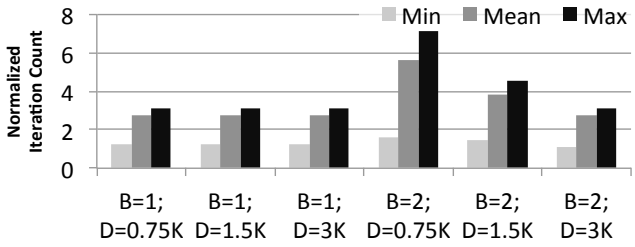


Figure 12. Iteration count as a function of bits per cell and dynamic range, normalized to 1-bit cells with $R_{off}/R_{on} = 1500$. The minimum, maximum, and the mean iteration counts are reported over 100 Monte-Carlo simulation experiments.

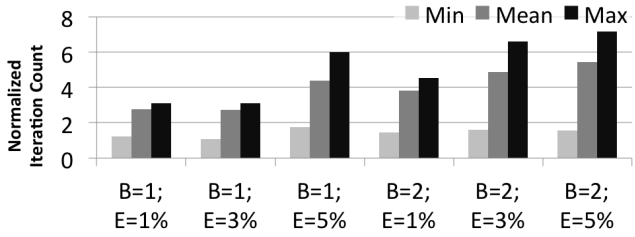


Figure 13. Iteration count as a function of bits per cell and programming error, normalized to 1-bit cells with no programming error. The minimum, maximum, and the mean iteration counts are reported over 100 Monte-Carlo simulation experiments.

current summation. This, in turn, introduces error into the computation, which tends to hinder convergence rates.

Programming precision. We re-evaluate the convergence behavior with and without cell programming errors to determine how results are affected by non-ideal programming. The results show virtually no sensitivity to programming error for single-bit cells until the programming error reaches 5%, well within the achievable programming precision reported in the literature [58]. Again, the general dependence becomes stronger as the number of bits per cell is increased. High programming error with the same tolerance introduces errors into the computation, hindering convergence.

IX. Conclusions

Previous work with memristive systems has been restricted to low-precision, error-prone, fixed-point computation, taking advantage of machine learning workloads that demand fidelity but not precision. We have presented the first correct implementation of a double-precision floating-point compute unit based on a memristive substrate. The innovations presented here take advantage of insights that allow the system to achieve a $10.3\times$ throughput improvement and a $10.9\times$ reduction in energy consumption over a GPU baseline when executing iterative solvers. The potential for such a system in scientific computing, where iterative solvers are ubiquitous, is especially promising.

Acknowledgment

This work was supported in part by NSF grant CCF-1533762.

REFERENCES

[1] F. Pop, "High performance numerical computing for high energy physics: a new challenge for big data science," *AAdv. High Energy Phys.*, vol. 2014, 2014.

[2] R. A. Pielke Sr, *Mesoscale meteorological modeling*, 2nd ed. Academic press, 2013.

[3] B. Schölkopf, K. Tsuda, and J.-P. Vert, *Kernel methods in computational biology*, 1st ed. MIT press, 2004.

[4] L. Maliar and S. Maliar, "Numerical methods for large-scale dynamic economic models," 2014.

[5] M. Vogelsberger, S. Genel, V. Springel, P. Torrey, D. Sijacki, D. Xu, G. Snyder, S. Bird, D. Nelson, and L. Hernquist, "Properties of galaxies reproduced by a hydrodynamic simulation," *Nature*, vol. 509, pp. 177–82, May 2014.

[6] P. Messina. (2017, February) The U.S. D.O.E. exascale computing project—goals and challenges. https://www.nist.gov/sites/default/files/documents/2017/02/21/messina_nist_20170214.final_.pdf. [Online]. Available: "https://www.nist.gov/sites/default/files/documents/2017/02/21/messina_nist_20170214.final_.pdf"

[7] 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.

[8] 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.

[9] 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 High Performance Computer Architecture (HPCA)*, June 2016.

[10] 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)*, Feb. 2017.

[11] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating graph processing using ReRAM," Feb. 2017.

[12] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," in *Intl. Symp. on Workload Characterization (IISWC)*, Sept. 2013.

[13] J. Hauser. (2002) SoftFloat. http://www.jhauser.us/arithmetic/SoftFloat.html. [Online]. Available: "http://www.jhauser.us/arithmetic/SoftFloat.html"

[14] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Math Softwware (TOMS)*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[15] R. Vuduc, "Automatic performance tuning of sparse matrix kernels," Ph.D. dissertation, 2003.

[16] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A machine-learning supercomputer," in *Intl. Symp. on on Microarchitecture (MICRO)*, Dec. 2014.

[17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Intl. Symp. on on Computer Architecture (ISCA)*, June 2017.

[18] 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.

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

[20] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: Past, present, and future." *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 803–820, 2003.

[21] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, 1952, vol. 49, no. 6.

[22] H. A. Van der Vorst, "Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems," *SIAM Journal on scientific and Statistical Computing (SISC)*, vol. 13, no. 2, pp. 631–644, 1992.

[23] 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, July 1986.

[24] J. Kung, Y. Long, D. Kim, and S. Mukhopadhyay, "A programmable hardware accelerator for simulating dynamical systems," in *Intl. Symp. on Computer Architecture (ISCA)*, June 2017.

[25] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti, "Accelerating sparse matrix-matrix multiplication with 3d-stacked logic-in-memory hardware," in *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2013.

[26] S. Kestur, J. D. Davis, and E. S. Chung, "Towards a universal FPGA matrix-vector multiplication architecture," in *Intl. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, May 2012.

[27] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication," in *Intl. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, May 2014.

[28] R. Dorrance, F. Ren, and D. Marković, "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs," in *ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA)*, Feb. 2014.

[29] B. Feinberg, S. Wang, and E. Ipek, "Making memristive neural net accelerators reliable," in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, Feb 2018.

[30] D. H. Bailey, "High-precision floating-point arithmetic in scientific computation," *Computing in science & engineering*, vol. 7, no. 3, pp. 54–61, 2005.

[31] D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo *et al.*, "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[32] L. V. Allis, "Searching for solutions in games and artificial intelligence," 1994.

[33] N. Whitehead and A. Fit-Florea, "Precision & performance: Floating point and IEEE 754 compliance for nvidia gpus," Nvidia, Tech. Rep., 2011.

[34] L. Kull, D. Luu, C. Menolfi, M. Braendli, P. A. Francese, T. Morf, M. Kossel, H. Yueksel, A. Cevrero, I. Ozkaya, and T. Toifl, "28.5 a 10b 1.5gs/s pipelined-SAR ADC with background second-stage common-mode regulation and offset calibration in 14nm CMOS FinFET," in *Intl. Solid-State Circuits Conference (ISSCC)*, Feb 2017, pp. 474–475.

[35] L. Kull, T. Toifl, M. Schmatz, P. A. Francese, C. Menolfi, M. Brandli, 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*, vol. 48, no. 12, pp. 3049–3058, 2013.

[36] M. Saberi, R. Lotfi, K. Mafinezhad, and W. A. Serdijn, "Analysis of power consumption and linearity in capacitive digital-to-analog converters used in successive approximation adcs," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, no. 8, pp. 1736–1748, 2011.

[37] M. R. Stan and W. P. Burleson, "Bus-invert coding for low-power I/O," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 3, no. 1, pp. 49–58, 1995.

[38] K. Bhanushali and W. R. Davis, "FreePDK15: An open-source predictive process design kit for 15nm FinFET technology," in *Intl. Symp. on on Physical Design (ISPD)*, March 2015.

[39] Predictive technology model (PTM). http://ptm.asu.edu/. [Online]. Available: http://ptm.asu.edu/

[40] D. Niu, C. Xu, N. Muralimanohar, N. P. Jouppi, and Y. Xie, "Design of cross-point metal-oxide ReRAM emphasizing reliability and cost," in *Intl Conference on Computer-Aided Design (ICCAD)*, Nov. 2013.

[41] S.-C. Wong, G.-Y. Lee, and D.-J. Ma, "Modeling of interconnect capacitance, delay, and crosstalk in VLSI," *IEEE Transactions on semiconductor manufacturing*, vol. 13, no. 1, pp. 108–111, 2000.

[42] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *"Intl. Symp. on High Performance Computer Architecture (HPCA)"*, Feb 2015.

[43] C. Yakopcic and T. Taha, "Model for maximum crossbar size based on input driver impedance," *Electronics Letters*, vol. 52, no. 1, pp. 25–27, 2015.

[44] M. O'Halloran and R. Sarpeshkar, "A 10-nw 12-bit accurate analog storage cell with 10-aa leakage," *IEEE Journal of Solid-State Circuits*, vol. 39, no. 11, pp. 1985–1996, 2004.

[45] Leon3/GRLIB. http://www.gaisler.com/index.php/downloads/leongrlib. [Online]. Available: http://www.gaisler.com/index.php/downloads/leongrlib

[46] S. Galal, O. Shacham, J. S. B. II, J. Pu, A. Vassiliev, and M. Horowitz, "FPU generator for design space exploration," in *IEEE Symposium on Computer Arithmetic (ARITH)*, April 2013.

[47] M. Martins, J. M. Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen, "Open cell library in 15nm FreePDK technology," in *Intl. Symp. on Physical Design (ISPD)*, March 2015.

[48] Synopsys, "Synopsys Design Compiler User Guide," http://www.synopsys.com/\Tools/Implementation/RTLSynthesis/DCUltra/Pages/.

[49] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 14:1–14:25, June 2017.

[50] G. Fredeman, D. W. Plass, A. Mathews, J. Viraraghavan, K. Reyer, T. J. Knips, T. Miller, E. L. Gerhard, D. Kannambadi, C. Paone, D. Lee, D. J. Rainey, M. Sperling, M. Whalen, S. Burns, R. R. Tummuru, H. Ho, A. Cestero, N. Arnold, B. A. Khan, T. Kirihata, and S. S. Iyer, "A 14 nm 1.1 mb embedded DRAM macro with 1 ns access," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 1, pp. 230–239, Jan 2016.

[51] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," April 2009.

[52] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling energy optimizations in GPGPUs," in *Intl. Symp. on Computer Architecture (ISCA)*, June 2013.

[53] H. Anzt, J. Dongarra, M. Kreutzer, G. Wellein, and M. Khler, "Efficiency of general krylov methods on GPUs – an experimental study," in *Intl. Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016.

[54] "Nvidia tesla P100," NVIDIA Corporation, Tech. Rep. WP-08019-001_v01.1, 2016.

[55] Z. Wei, Y. Kanzawa, K. Arita, Y. Katoh, K. Kawai, S. Muraoka, S. Mitani, S. Fujii, K. Katayama, M. Iijima *et al.*, "Highly reliable TaOx ReRAM and direct evidence of redox reaction mechanism," in *Electron Devices Meeting, 2008. IEDM 2008. IEEE International*. IEEE, 2008, pp. 1–4.

[56] J. J. Yang, M.-X. Zhang, J. P. Strachan, F. Miao, M. D. Pickett, R. D. Kelley, G. Medeiros-Ribeiro, and R. S. Williams, "High switching endurance in TaOx memristive devices," *Applied Physics Letters*, vol. 97, no. 23, p. 232102, 2010.

[57] C.-W. Hsu, I.-T. Wang, C.-L. Lo, M.-C. Chiang, W.-Y. Jang, C.-H. Lin, and T.-H. Hou, "Self-rectifying bipolar TaOx/TiO2 RRAM with superior endurance over 1012 cycles for 3d high-density storage-class memory."

[58] F. Alibart, L. Gao, B. D. Hoskins, and D. B. Strukov, "High precision tuning of state for memristive devices by adaptable variation-tolerant algorithm," *Nanotechnology*, vol. 23, no. 7, Jan. 2012.