ReFloat: Low-Cost Floating-Point Processing in ReRAM for Accelerating Iterative Linear Solvers

Linghao Song
University of California, Los Angeles
linghaosong@cs.ucla.edu

Fan Chen
Indiana University Bloomington
fc7@iu.edu

Hai Li
Duke University
hai.li@duke.edu

Yiran Chen
Duke University
yiran.chen@duke.edu

ABSTRACT

Resistive random access memory (ReRAM) is a promising technology that can perform low-cost and in-situ matrix-vector multiplication (MVM) in analog domain. Scientific computing requires high-precision floating-point (FP) processing. However, performing floating-point computation in ReRAM is challenging because of high hardware cost and execution time due to the large FP value range. In this work we present ReFloat, a data format and an accelerator architecture, for low-cost and high-performance floating-point processing in ReRAM for iterative linear solvers. ReFloat matches the ReRAM crossbar hardware and represents a block of FP values with reduced bits and an optimized exponent base for a high range of dynamic representation. Thus, ReFloat achieves less ReRAM crossbar consumption and fewer processing cycles and overcomes the noncovergence issue in a prior work. The evaluation on the SuiteSparse matrices shows ReFloat achieves 5.02× to 84.28× improvement in terms of solver time compared to a state-of-the-art ReRAM based accelerator.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Hardware** → **Emerging architectures**; **Hardware accelerators**.

KEYWORDS

Processing-in-memory, Accelerator, ReRAM, Floating-point.

ACM Reference Format:
Linghao Song, Fan Chen, Hai Li, and Yiran Chen. 2023. ReFloat: Low-Cost Floating-Point Processing in ReRAM for Accelerating Iterative Linear Solvers . In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3581784.3607077

1 INTRODUCTION

With the diminishing gain of Moore's Law [92] and the end of Dennard scaling [38], general-purpose computing platforms such as CPUs and GPUs will no longer benefit from shrinking transistor size or integrating more cores [30]. Thus, domain-specific architectures are critical for improving the performance and energy efficiency of various applications. Rather than relying on conventional CMOS technology, the emerging non-volatile memory technology such as resistive random access memory (ReRAM) is considered as a promising candidate for implementing processing-in-memory (PIM) accelerators [6, 11, 16, 32, 47, 49, 54, 63, 81, 88, 89, 104] that can provide orders of magnitude improvement of computing efficiency. Specifically, ReRAM can store data and perform in-situ matrix-vector multiplication (MVM) operations in the analog domain. Most current ReRAM-based accelerators focus on machine learning applications, which can accept a low precision, e.g., less than 16-bit fixed-point, thanks to the quantization in deep learning [21, 42, 44, 48, 64].

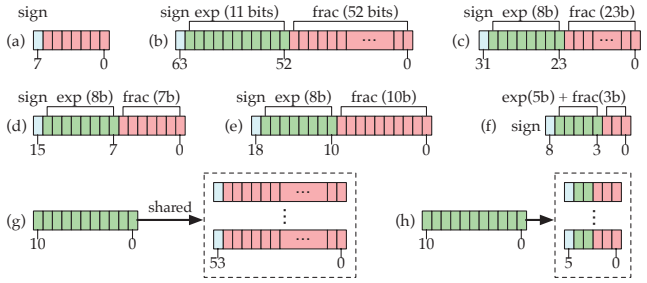


Figure 1: The bit layout of (a) an 8-bit signed integer, (b) a 64-bit double-precision floating-point number, (c) a 32-bit single-precision floating-point number, (d) a Google bfloat16 number, (e) an Nvidia TensorFloat32 number, (f) a Microsoft ms-fp9 number, (g) a block of numbers in block floating point, and (h) a block of numbers in `ReFloat`(x,2,3).

Scientific computing is a collection of tools, techniques, and theories for solving science and engineering problems modeled in mathematical systems [40]. The underlying variables in scientific computing are continuous in nature, such as time, temperature, distance, and density. One of the essential aspects of scientific computing is modeling a complex system with partial differential equations (PDEs) to understand the natural phenomena in science [45, 52], or the design and decision-making of engineered systems [14, 75]. Most problems in continuous mathematics modeled by PDEs cannot be solved directly. In practice, the PDEs are converted to a linear system $A\mathbf{x} = \mathbf{b}$, and then solved through an iterative solver that ultimately converges to a numerical solution [8, 80]. To obtain an acceptable answer where the residual is less than a desired threshold, intensive computing power [31, 84] is required

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11.
https://doi.org/10.1145/3581784.3607077

to perform the floating-point sparse matrix-vector multiplication (SpMV), the critical computation kernel.

Because of the prevalent floating-point operations in scientific computing, it is desirable to leverage ReRAM to achieve parallel in-situ floating-point SpMV. When using the ReRAM crossbar to perform SpMV, we partition the matrix into blocks, encode each matrix element as the ReRAM cell conductance, and convert the input vector to wordline voltage through Digital-to-Analog Converters (DACs). Thus, the bitline will output the results of the dot-product between the current input vector bits and matrix elements mapped in the same crossbar column. Each bitline in the output is connected to a sample and hold (S/H) unit. After all input bits are processed, the results of the SpMV are available at the output of S/H unit, which is converted to multi-bit digital values by Analog-to-Digital Converters (ADCs). In general, the number of bits in the input vector and the matrix determine the number of cycles for performing an SpMV. In contrast, the number of bits representing matrix elements determines the number of crossbars.

We examine mapping the floating-point SpMV by leveraging the same principle used in MVM. Take 64-bit double-precision number as an example: each floating-point number consists of a 1-bit sign (s), an 11-bit exponent (e), and a 52-bit fraction (f). The value is interpreted as $(-1)^s \times (1.b_{51}b_{50}...b_0) \times 2^{(e-1023)}$, yielding a dynamic data range from $\pm 2.2 \times 10^{-308}$ to $\pm 1.8 \times 10^{308}$. The number of crossbars for a matrix M increases *exponentially* with the bits number of the exponent (e_M) and linearly with the bits number of the fraction (f_M). Thus, directly representing floating-point values with a large number of crossbars incurs prohibitive costs.

To reduce the overhead, Feinberg *et al.*[32] propose to truncate the higher bits in exponents, e.g., using the low 6 bits or module 64 of the exponent (the 64 paddings in [32]) to represent each original value, while keeping the number of fraction bits the unchanged (52 bits). However, this ad-hoc solution *does not ensure the convergence* of iterative solves (see Table 1 and Section 6.2). In general, to ensure convergence, we need two requirements. (1) correct matrix values, which are ensured by [32] with the aid of floating-point units (FPUs) when the exponent range of a submatrix falls outside the 6 bits mapped to ReRAMs. (2) correct vector values, which is not considered by [32]. In the computation, matrix value does not change, but vector values change every iteration. Thus, vector values in [32] fall out of range (i.e., the 64 padding). As a result, the solvers do not converge. In addition, the hardware cost increases exponentially with the exponent bits. [32] used 6 bits for the exponent, however, we can further reduce the exponent bits. Thus, [32] did not fully reduce the overhead.

We propose ReFloat, a principled approach based on a flexible and fine-grained floating-point number representation. The key insight of our solution is the *exponent value locality* among the elements in a matrix block, which is the granularity of computation in ReRAM. If we consider the whole matrix, the exponent values can span a wide range, e.g., up to 11 for a matrix, but the range within a block is smaller, e.g., at most 7 for the same matrix. It naturally motivates the idea of choosing an *exponent base* e_b for all exponents in a block and storing only the *offsets* from e_b. For a matrix block, although the absolute exponent values may be large, the variation is not. For most blocks, by choosing a proper e_b, the offset values are much smaller than the absolute exponent values, thereby reducing the number of bits required.

Instead of simply using the offset as a lossless compression method, ReFloat aggressively uses fewer bits for exponent offsets (e) than the required number of bits to represent them. The error is bounded by the existence of value locality in real-world matrices. Moreover, the error is refined due to the nature of the iterative solver. Starting from an all-zero vector, an increasingly accurate solution is produced in each iteration. The iterative solver stops when the defined convergence criteria are satisfied. Because the vector from each iteration is not accurate anyway, the computation has certain resilience to the inaccuracy due to floating-point data representation. It is why [32] can work in certain cases. In ReFloat, when an offset is larger (smaller) than the largest (smallest) offset represented by e bits, the largest (smallest) value of e bits is used for the offset. With e-bit exponent offset, the range of exponent values is $[e_b - 2^{(e-1)} + 1, e_b + 2^{(e-1)} - 1]$. Selecting e_b becomes an optimization problem that minimizes the difference between the exponents of the original matrix block and the exponents with e_b and e-bit offsets.

To facilitate the proposed ideas in a concrete architecture, we define the ReFloat format as $\texttt{ReFloat}(b, e, f)(e_v, f_v)$, where b denotes the matrix block size—the length and width of a square matrix block is 2^b, e and f respectively denote the exponent and fraction bit numbers for the matrix, and (e_v, f_v) denotes the exponent and fraction bit numbers for the vector. With e_b for each block, we are able to represent all matrix elements in the block. Then, we develop the conversion scheme from default double-precision floating-point format to ReFloat format and the computation procedure. Based on ReFloat format, we design the low-cost high-performance floating-point processing architecture in ReRAM. Our results show that for 12 matrices evaluated on iterative solvers (CG and BiCGSTAB), only 3 bits for exponent and 8 or 16 bits for fraction are sufficient to ensure convergence. In comparison, [32] uses 6 bits for exponent and 51 bits for fraction without guaranteeing convergence. It translates to a speedup of 5.02× to 84.28× compared with a state-of-the-art ReRAM-based accelerator [32] for scientific computing even with the assumption that the accelerator [32] functions the same as FP64 solvers. We released the source code at https://github.com/linghaosong/ReFloat.

2 BACKGROUND

2.1 In-situ MVM Acceleration in ReRAM

ReRAM [4, 100] has recently demonstrated tremendous potential to efficiently accelerate the computing kernels in machine learning. Conceptually, each element in a matrix M is mapped to the conductance state of a ReRAM cell. At the same time, the input vector $\mathbf{x}$ is encoded as voltage levels that are applied on the wordlines of the ReRAM crossbar. In this way, the current accumulation on bitlines is proportional to the dot-product of the stored conductance and voltages on the wordlines, representing the result $\mathbf{y} = M \times \mathbf{x}$. Such *in-situ* computation significantly reduces the expensive memory access in MVM processing engines [47], and most importantly, provide massive opportunities to exploit the inherent parallelism in an $N \times N$ ReRAM crossbar.

ReRAM-based MVM processing engines are fixed-point hardware in nature since the matrix and the vector are respectively represented in *discrete* conductance states and voltage levels [100]. If ReRAM is used to support floating-point MVM operation, many crossbars will be provisioned for fraction alignment, resulting in very high hardware costs. We will illustrate the problem in Section 3 to motivate REFLOAT design. Nevertheless, the fixed-point precision requirement is acceptable for machine learning applications thanks to the low-precision and quantized neural network algorithms[22, 42, 48, 51, 58, 106]. Many fixed-point based accelerators [6, 11, 16, 39, 54, 81, 88, 104] are built with the ReRAM MVM processing engine and achieve reasonable classification accuracy.

2.2 Iterative Linear Solvers

```
1  initiate x = x0
2  while (not converge) do
3      //Step 1: compute the residual
4      r = b - A * x
5      //Step 2: compute the correction
6      compute p
7      //Step 3: update the current solution
8      x = x + p
9  end while
```

Code 1: The iterative linear solver.

Scientific computing is an interdisciplinary science that solves computational problems in a wide range of disciplines, including physics, mathematics, chemistry, biology, engineering, and other natural sciences subjects [7, 36, 41]. Systems of large-scale PDEs typically model those complex computing problems. Since it is almost impossible to obtain the analytical solution of those PDEs directly, a common practice is to discretize continuous PDEs into a linear system $A\mathbf{x} = \mathbf{b}$ [8, 80] to be solved by numerical methods. The numerical solution of this linear system is usually obtained by an iterative solver [25, 72, 99].

Code 1 illustrates a typical computing process in iterative methods. The vector $\mathbf{x}$ to be solved is typically initialized to an all-zero vector $\mathbf{x}_0$, followed by three steps in the main body: (1) the residual (error) of the current solution vector is calculated as $\mathbf{r} = \mathbf{b} - A\mathbf{x}$; (2) to improve the performance of the estimated solution, a correction vector $\mathbf{p}$ is computed based on the current residual $\mathbf{r}$; and (3) the current solution vector is improved by adding the correction vector as $\mathbf{x} = \mathbf{x} + \mathbf{p}$, aiming to reduce the possible residuals produced in the next calculation iteration. The iterative solver stops when the defined convergence criteria are satisfied. Two widely used convergence criteria are (1) that the iteration index is less than a preset threshold K, or (2) that the L-2 norm of the residual ($\text{res} = ||\mathbf{b} - A\mathbf{x}||^2$) is less than a preset threshold τ. Notably, all the values involved in Code 1 are implemented as double-precision floating-point numbers to meet the high-precision requirement of mainstream scientific applications.

The various iterative methods follow the above computational steps and differ only in calculating the correction vectors. Among all candidate solutions, Krylov subspace approach is the standard method nowadays. In this paper, we focus on two representative Krylov subspace solvers – Conjugate Gradient (CG) [46] and Stabilized BiConjugate Gradient (BiCGSTAB) [91]. The computational kernels of these two methods are large-scale sparse floating-point matrix-vector multiplication $\mathbf{y} = A\mathbf{x}$, which requires the support of floating-point computation in ReRAM and imposes significant challenges to the underlying computing hardware.

2.3 Fixed-Point and Floating-Point Representations

We use the 8-bit signed integer and the IEEE 754-2008 standard [19] 64-bit double-precision floating-point number as examples to compare the difference between fixed-point and floating-point numbers. They refer to the format used to store and manipulate the digital representation of data. As shown in Figure 1 (a), fixed-point numbers represent integers—positive and negative whole numbers—via a sign bit followed by multiple (*e.g., i*-bit) value bits, yielding a value range of -2^i to $2^i - 1$. IEEE 754 double-precision floating-point numbers shown in Figure 1 (b) are designed to represent and manipulate rational numbers, where a number is represented with a sign bit (s), an 11-bit exponent (e), and a 52-bit fraction ($b_{51}b_{50}...b_0$). The value of a double-precision floating-point is interpreted as $(-1)^s \times (1.b_{51}b_{50}...b_0) \times 2^{(e-1023)}$, yielding a dynamic data range from $\pm 2.2 \times 10^{-308}$ to $\pm 1.8 \times 10^{308}$.

Many efficient floating point formats shown in Figure 1 have been proposed because the default format incurs a high cost for conventional digital systems. However, the applications such as deep learning do not require a very wide data range. The representative examples include IEEE 32-bit single-precision floating point (FP32), Google bfloat16 [95], Nvidia TensorFloat32 [57], Microsoft ms-fp9 [1] [18], and block floating point (BFP) [12, 59]. Accordingly, specialized hardware designs or/and systems are also proposed to amplify the benefits of efficient data formats. For example, Google bfloat is associated with TPU [1, 2, 56], Nvidia TensorFloat is associated with tensor core GPUs, Microsoft floating-point formats are associated with Project Brainwave [18], and BFP are favorable for signal processing on DSPs [29] and FPGAs [20].

However, the floating-point representations favored by deep learning may not benefit scientific computing. For deep learning, weights can be retrained to a narrowed/shrunk space, even without floating-point [21, 48, 66, 79, 107]. In scientific computing, data cannot be retrained, and the shrunk formats can not capture all values. For example, 1.0×10^{-40} falls out of range for FP32, bfloat16, TensorFloat32, and ms-fp9 because of narrow range representation. Two values 1.0×10^{-40} and 1.0×10^{-30} can not be captured by a BFP block because of non-dynamic range representation within a block. The narrow or non-dynamic range may lead to non-convergence in scientific computing.

In general, double-precision floating-point is a norm for high-precision scientific computations because it can support a wide range of data values with high precision. However, the processing demands low hardware costs and high performance.

3 MOTIVATION AND REFLOAT IDEAS

3.1 Fixed-Point MVM processing in ReRAM

The processing of SpMV on ReRAM-based accelerators utilizes matrix blocking on a large matrix to perform MVM on matrix blocks with ReRAM crossbars [32, 89]. The floating-point MVM is

[1] We infer the layout from the description in [18]. No public specifications on ms-fp are available.

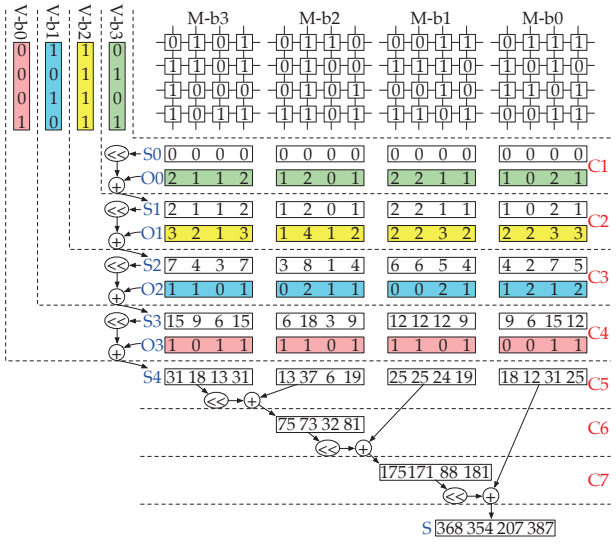


Figure 2: Fixed-point (integer) MVM in ReRAM.

built on fixed-point MVM. To understand the cycle numbers and ReRAM crossbar numbers in ReRAM-based fixed-point MVM, we use Figure 2 as an example.

$$\begin{bmatrix} 368 \\ 354 \\ 207 \\ 387 \end{bmatrix}_d = \begin{bmatrix} 0 & 13 & 7 & 11 \\ 11 & 14 & 3 & 8 \\ 9 & 5 & 2 & 5 \\ 14 & 6 & 9 & 15 \end{bmatrix}_d^T \times \begin{bmatrix} 6 \\ 12 \\ 6 \\ 13 \end{bmatrix}_d = \begin{bmatrix} 0000 & 1101 & 0111 & 1011 \\ 1011 & 1110 & 0011 & 1000 \\ 1001 & 0101 & 0010 & 0101 \\ 1110 & 0110 & 1001 & 1111 \end{bmatrix}_b^T \times \begin{bmatrix} 0110 \\ 1100 \\ 0110 \\ 1101 \end{bmatrix}_b . \quad (1)$$

Figure 2 shows the processing of fixed-point MVM in ReRAM, which represents the computation of an example Eq. (1) by utilizing ReRAM-based MVM engines with single-bit precision. Before computation, we convert the decimal integers in both the matrix and the vector to binary bits. We set the precision for the matrix and input vector to 4-bit. The matrix is bit-sliced into four 1-bit matrices and then mapped to four crossbars, i.e., M-b3, M-b2, M-b1, and M-b0. The input vector is bit-sliced into 4 one-bit vectors, i.e., V-b3, V-b2, V-b1, and V-b0. The multiplication is performed in pipeline. Each crossbar has a zero initial vector S0. In the first cycle C1, the most significant bit (MSB) vector V-b3 is applied on wordlines of the four crossbars, and the multiplication results of V-b3 with M-b3, M-b2, M-b1, and M-b0 are obtained in parallel, denoted by O0. In cycle C2, S0 is right-shifted by 1 bit to get S1, and V-b2 is input to the crossbars to get the multiplication results O1. Similar operations are performed in C3 and C4. After C4, we get S4, the multiplication results of the input vector with four bit-slices of the matrix. In the following threes cycles C5 to C7, we shift and add S4 from the four crossbars to get the final multiplication result. For the fixed-point multiplication of an N_M-bit matrix with an N_v-bit vector, the processing cycle count is $C_{\text{int}} = N_v + (N_M - 1)$.

3.2 Hardware Cost and Performance Analysis of Floating-Point MVM in ReRAM

In this section, we explain in detail the hardware cost, i.e., the *crossbar number* C, and the performance, i.e., the *cycle number* T, of ReRAM-based floating-point MVM. Note that C correlates with the ability to execute floating-point MVMs in parallel with a given number of on-chip ReRAMs [32, 81, 89]: the smaller C, the more parallelism can be explored. A smaller T directly reflects a higher performance of one ReRAM-based MVM on a matrix block. A smaller T and a smaller C reflects a higher performance of one SpMV on a whole matrix.

Crossbar number. Suppose we compute the multiplication of a matrix block M and a vector segment v. In the matrix block M, the number of fraction bits is f_M and the number of exponent bits is e_M. In the vector segment v, the number of fraction bits is f_v and the number of exponent bits is e_v. To map the matrix fraction to ReRAM crossbars, we need (f_M+1) ReRAM crossbars because the fraction is normalized to a value with a leading 1. For example, (52+1) crossbars are needed to represent the 52-bit fraction in double floating-point precision in [32]. To map the matrix exponent to ReRAM crossbars, we need 2^{e_M} ReRAM crossbars for e_M-bit exponent states, which is called padding in [32] where 64-bit paddings are needed for an $e_M = 6$. Thus, C is calculated as

$$C = 4 \times (2^{e_M} + f_M + 1), \quad (2)$$

where the leading multiplier 4 is contributed from sign bits of the matrix block and the vector segment.

Cycle number. We conservatively suppose the precision of digital-analog converters is 1-bit as that in [32, 81]. The number of value states in a vector segment is $(2^{e_v} + f_v + 1)$. For each input state, we need $(2^{e_M} + f_M + 1)$ to perform shift-and-add to reduce the partial results from the ReRAM crossbars. To achieve higher computation efficiency, a pipelined input and reduce scheme [81] can be used. Thus, T is calculated as

$$T = (2^{e_v} + f_v + 1) + (2^{e_M} + f_M + 1) - 1. \quad (3)$$

High hardware cost and low performance in default double precision. In double-precision floating-point (FP64), one MVM in ReRAM consumes 8404 crossbars and 4201 cycles. To understand how bit number affects the hardware cost and performance, we explore the effect of exponent and fraction bit number of matrix and vector on the cycle number and the effect of exponent and fraction bit number of matrix on the crossbar number, illustrated in Figure 3. The crossbar number increases *exponentially* with e_M while linearly with f_M. Furthermore, the cycle number increases *exponentially* with both e_v and e_M, while the latency increases linearly with f_v and f_M.

3.3 Non-Convergence in [32]

The above analysis makes it possible to reduce the number of digits by reducing the number of bits of the exponent and fraction, thereby reducing hardware costs, i.e., fewer cycles and crossbars. However, the accuracy of the solvers may be significantly degraded or even cause non-convergence.

The design of the state-of-the-art ReRAM-based accelerator [32] for floating-point SpMV is driven by the goal of reducing the number of bits for exponent. However, this solution adopts an ad-hoc

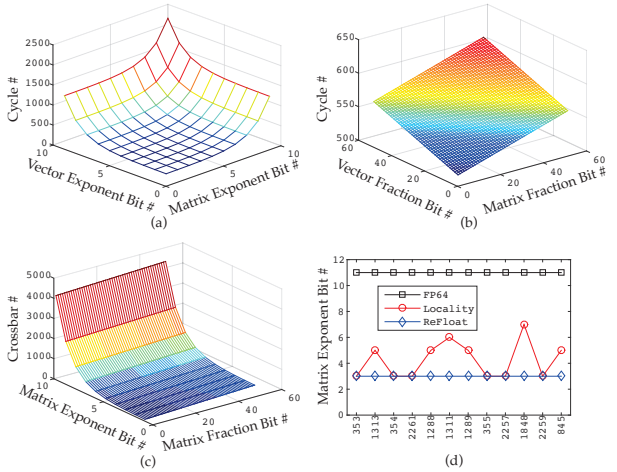


Figure 3: (a) The cycle number of various exponent bit number for vector segment and matrix block, (b) the cycle number of various fraction bit number for vector segment and matrix block, (c) the crossbar number of various fraction and fraction bit number for matrix block, and (d) matrix exponent bit number of double-precision floating point(FP64), the locality in 12 matrices, and ReFloat.

Table 1: The iteration numbers for convergence under various exp(onent) and fra(ction) bit configurations for matrix `crystm03`. NC indicates non-convergence.

exp	**11**	11	11	11	11	11
frac	**52**	30	29	28	27	26
#ite	**80**	82(+2)	82(+2)	83(+3)	83(+3)	84(+4)
exp	11	11	11	11	11	11
frac	25	24	23	22	21	20
#ite	90(+10)	93(+13)	93(+13)	95(+15)	107(+27)	**NC**

exp	10	9	8	7	6
frac	52	52	52	52	52
#ite	80	80	80	20620(+256×)	**NC**

approach that simply truncates a number of high order bits in exponent. Specifically, with the lower 6 bits of exponent, [32] uses module 64 of the exponent to represent each original value and map the matrix to ReRAM. For the matrix values out of the range of 6 bits, [32] uses FPUs to compute. For the computation of $A\mathbf{x}$, the matrix A can be accurately processed in [32]. However, the values of vector $\mathbf{x}$ change at every iteration but [32] does not provide any solution for processing correct vector values. Thus, the vector $\mathbf{x}$ values can fall out of the ranges of 64 paddings (6 bits), and *non-convergence* happens in [32].

Table 1 shows the number of iterations for convergence under various exponent and fraction bit configurations. In default double-precision, it takes 80 iterations to convergence. If we fix the exponent bits and truncate fraction bits, a 21-bit fraction takes 27 additional iterations, and a fraction less than 21 bits leads to non-convergence. If we fix the fraction bits and truncate exponent bits like [32], 7-bit exponent increases the iteration number from 80 to 20620, and an exponent less than 7 bits leads to non-convergence. Thus, the solution proposed in [32] may break the correctness of the iterative solver. In contrast, the number of bits in fraction has less impact on the number of iterations to converge. For example, Table 1 shows that drastically reducing fraction bits from 52 to 30 only increases the number of iterations by 2×. However, [32] kept the number of bits in fraction unchanged and lost the opportunity to reduce hardware cost and improve performance. Thus, we are convinced that we need to develop a more principled approach to find a better solution to the problem.

3.4 Value Locality & Bit Compression

We leverage an intuitive observation of matrix element values—*exponent value locality*—to reduce the number of exponents bits while keeping enough accuracy. We define the locality as the maximum number of required bits to cover the exponent in all matrix blocks of a large matrix. We illustrate the locality of matrices from SuiteSparse [24] in Figure 3(d). As discussed before, ReRAM performs MVM at the granularity of matrix block, whose size is determined by the size of ReRAM crossbar, e.g., 128×128. While exponent values of the whole matrix can span a wide range, e.g., up to 11 for a matrix, but the range is smaller within a block, e.g., at most 7 for the same matrix. Therefore, the default locality, i.e., 11, is redundant. Naturally, it motivates the idea of using an *exponent base* e_b for all exponents in a block and storing only the *offsets* from e_b. For most blocks, by choosing a proper e_b, the offset values are much smaller than the absolute exponent values, thereby reducing the number of bits required.

It is important to note that we do not simply use the offset as a lossless compression method. While exponent value locality exists for most of the blocks, it is possible that for a small number of blocks, the exponent values are scattered across a wide range. If we include enough bits for all offsets, the benefits for the majority of blocks will be diminished. Moreover, it is not necessary due to the nature of iterative solvers.

We can naturally tune the accuracy by the number of bits e allocated for the offsets, which is less than the number of exponent bits necessary to represent the offsets precisely. When an offset is larger (smaller) than the largest (smallest) offset representable by e bits, the largest (smallest) value of e bits is used accordingly. With e-bit exponent offset, the range of exponent values is $[e_b - 2^{(e-1)}+1, e_b+2^{(e-1)}-1]$. Intuitively, given e and e_b, this system can precisely represent the exponent values that fall into a "window" around e_b, while the "size of the window" is determined by $2^{(e-1)}$. Then, selecting e_b becomes an optimization problem that minimizes the difference between the exponents of the original matrix block and the exponents with e_b and e-bit offsets. Thus, we achieve a wide data range but a low hardware cost simultaneously.

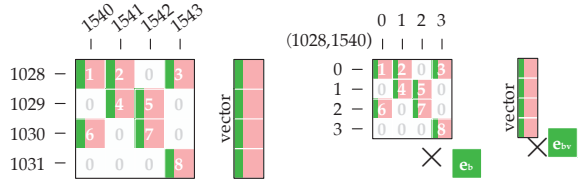
4 REFLOAT DATA FORMAT

4.1 ReFloat Format

We define ReFloat format as $\texttt{ReFloat}(b, e, f)(e_v, f_v)$, where b determines the matrix block size 2^b (the length and width of a square matrix block), e and f respectively denote the exponent and fraction bit numbers for the matrix, and (e_v, f_v) denotes the bit numbers for the vector. Table 2 lists the symbols and corresponding descriptions in ReFloat.

Table 2: List of symbols and descriptions.

`ReFloat`$(b,e,f)(e_v,f_v)$: `ReFloat` format notation.	
Symbol	**Description**
2^b	The size of a square block.
e	The number of exponent bits for a matrix block.
f	The number of fraction bits for a matrix block.
A	A sparse matrix.
$\mathbf{b}$	The bias vector for a linear system.
$\mathbf{x}$	The solution vector for a linear system.
$\mathbf{r}$	The residual vector for a linear system.
a	A scalar of A.
$(a)_e$	The exponent of a, $(a)_e \in \{0,1,2,...\}$.
$(a)_f$	The fraction of a, $(a)_f \in (1,2)$.
A_c	A block of the sparse matrix A.
(i,j)	The index for the block A_c.
(ii,jj)	The index for the scalar a in the block A_c.
(iii,jjj)	The index for the scalar a in the matrix A.
e_b	The base for exponents of elements in a block.
e_{bv}	The base for exponents a vector segment.
e_v	The number of exponent bits for a vector segment.
f_v	The number of fraction bits for a vector segment.



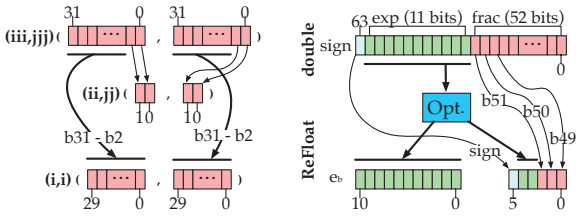
(a) A block in full precision. (b) A block in ReFloat.

Figure 4: Comparison of a matrix block (a) in original full precision format and (b) in ReFloat format.

Figure 4 intuitively illustrates the idea of ReFloat. In Figure 4 (a), each scalar is in a 64-bit floating-point format. It requires a 32-bit integer for row index and a 32-bit integer for column index to locate each element in the matrix block. Therefore, we need $8 \times (32+32+64) = 1024$ bits for storing the eight scalars. With ReFloat, assuming we use `ReFloat`$(2,2,3)$ format as depicted in Figure 4 (b), we see that: (1) each scalar in the block can be indexed by two 2-bit integers; (2) the element value is represented by a $1+2+3=6$-bit floating point number [2]; (3) the block is indexed by two 30-bit integers and (4) an 11-bit exponent base e_b is also recorded. Therefore, we only use $8\times(2+2+6)+2\times30+11 = 151$ bits to store the entire matrix block, which reduces the memory requirement by approximately 10× (151 vs. 1024). This reduction in bit representation is also beneficial for reducing the number of ReRAM crossbars for computation in hardware implementation. Specifically, the full precision format consumes 118 crossbars, as illustrated in [32], our design only requires 16 crossbars with `ReFloat`$(2,2,3)$ format. Thus, given the same chip area, our design is able to process more matrix blocks in parallel.

4.2 Conversion to ReFloat Format

In order to convert the original matrix to a `ReFloat`(b,e,f) format, three hyperparameters need to be determined in advance. The b defines how the indices of input data are converted and determined by the physical size of ReRAM crossbars, i.e., a crossbar with 2^b wordlines and 2^b bitlines. As demonstrated in Figure 5 (a), the leading 30 bits—b_{31} to b_2 of the index (iii,jjj) for a scalar in the matrix A—are copied to the same bits in the index (i,j) for the block A_c. For each scalar in the block A_c, the index (ii,jj) for that scalar inside the block A_c is copied from the last two bits of the index (iii,jjj). The scalars in the same block share the block index (ii,jj), and each scalar uses fewer bits for the index inside that block. Thus, we also save memory space for indices.



(a) Index conversion. (b) Value conversion.

Figure 5: The conversion of index and value in floating-point format to ReFloat format.

Table 3: Various formats represented by ReFloat.

Int8	`ReFloat`$(0,0,7)$	bfloat16 [95]	`ReFloat`$(0,8,7)$
Int16	`ReFloat`$(0,0,15)$	ms-fp9 [18]	`ReFloat`$(0,5,3)$
FP32(float)	`ReFloat`$(0,8,23)$	TensorFloat32 [57]	`ReFloat`$(0,8,10)$
FP64(double)	`ReFloat`$(0,11,52)$	BFP64	`ReFloat`$(6,0,52)$

The hyper-parameters e and f determine the accuracy of floating-point values. A floating-point number consists of three parts: (1) the sign bit, (2) the exponent bits, and (3) the fraction bits. When converted to ReFloat, the sign bit remains unchanged. For the fraction, we only keep the leading f bits from the original fraction bits and remove the rest bits in the fraction, as shown in Figure 5 (b). For the exponent bits, we need to first determine the base value e_b for the exponent. As e means the number of bits for the "swing" range, we need to find an optimal base value e_b to utilize the e bits fully. We formalize the problem as an optimization for find the e to minimize a loss target L, defined as

$$\min_{e_b} L,\ L = \sum_{a\in A_c}\left(\log_2\left(\frac{a}{(a)_f\times 2^{e_b}}\right)\right)^2 = \sum_{a\in A_c}((a)_e - e_b)^2. \quad (4)$$

Let $\partial L/\partial e_b = 0$, we can get

$$e_b = \left\lceil \frac{1}{|A_c|}\sum_{a\in A_c}(a)_e \right\rceil. \quad (5)$$

Thus, we use the original exponent to minus the optimal e_b to get an e-bit signed integer in the conversion. The e-bit signed integer is the exponent in ReFloat.

We use an example to illustrate the format conversion intuitively. The original floating-point values in Eq. (6) are converted to `ReFloat`(x,2,2) format in Eq. (7),

$$\begin{bmatrix} (-1)\times 1.1111\times 2^7 & 1.0101\times 2^8 \\ (-1)\times 1.0000\times 2^9 & 1.0001\times 2^7 \end{bmatrix} = \begin{bmatrix} -248.0 & 336.0 \\ -512.0 & 136.0 \end{bmatrix}, \quad (6)$$

$$2^8\times\begin{bmatrix} (-1)\times 1.11\times 2^{-1} & 1.01\times 2^0 \\ (-1)\times 1.00\times 2^1 & 1.00\times 2^{-1} \end{bmatrix} = \begin{bmatrix} -224.0 & 320.0 \\ -512.0 & 128.0 \end{bmatrix}, \quad (7)$$

where $e_b = 8$. Here, we see that ReFloat incurs conversion loss for the conversion of floating-point values from the original. However, for scientific computing, the errors in the iterative solver are gradually corrected. Thus, the errors introduced by the conversion will

[2]The elements inside a ReFloat block are floating-point, while the elements inside a BFP block are fixed-point.

also be corrected in the iteration. From an application/algorithm perspective, REFLOAT format is versatile, and the popular formats in Figure 1 can all be represented by REFLOAT as listed in TABLE 3. The low hardware cost and format versatility benefit the high performance and fast convergence of REFLOAT in solving PDEs. We will show the performance and convergence of the iterative solver in REFLOAT format in Section 6.

4.3 Computation in ReFloat Format

The matrix A is partitioned into blocks. To compute the matrix-vector multiplication $\mathbf{y} = A\mathbf{x}$, the input vector $\mathbf{x}$ and the output vector $\mathbf{y}$ are correspondingly partitioned into vector segments $\mathbf{x}_c$ and $\mathbf{y}_c$. The size of the vector segments is $(2^b \times 1)$.

For the p-th output vector segment $\mathbf{y}_c(p)$, the computation in the default full precision will be

$$\mathbf{y}_c(p) = \sum_i A_c(i,p)\mathbf{x}_c(i), \tag{8}$$

where $A_c(i,p)$ is the matrix block indexed by (i,p) and $\mathbf{x}_c(i)$ is the input vector segment indexed by i. The matrix blocks at the p-th block column are multiplied with the input vector segments for partial sums and then they are accumulated. In the computation for each matrix block, because the original matrix block $A_c(i,p)$ is converted to $A_c(i,p) \simeq 2^{e_b(i,p)}\tilde{A}_c(i,p)$, the original vector segment $\mathbf{x}_c(i)$ is converted to $\mathbf{x}_c(i) \simeq 2^{e_{bv}(i)}\tilde{\mathbf{x}}_c(i)$, and we encode $2^{e_b(i,p)}\tilde{A}_c(i,p)$ and $2^{e_{bv}(i)}\tilde{\mathbf{x}}_c(i)$ by ReFloat. Thus, the multiplication for the matrix block $A_c(i,p)$ and the vector segment $\mathbf{x}_c(i)$ is computed as $A_c(i,p)\mathbf{x}_c(i) = 2^{e_b(i,p)+e_{bv}(i)}\tilde{A}_c(i,p)\tilde{\mathbf{x}}_c(i)$. The matrix-vector multiplication for the p-th output vector segment in the default format Eq. (8) is then computed as

$$\mathbf{y}_c(p) = \sum_i 2^{e_b(i,p)+e_{bv}(i)}\tilde{A}_c(i,p)\tilde{\mathbf{x}}_c(i). \tag{9}$$

Here we see that with REFLOAT format, the block matrix multiplication in the default format is preserved. In the hardware processing, we perform the fixed-point MVM $\tilde{A}_c\tilde{\mathbf{x}}_c$ by the ReRAM crossbars as shown in Figure 6(c) and multiply the vector exponent and the block exponent in a processing engine as shown in Figure 6(b). Thus, the original high-cost multiplication in full precision $A_c\mathbf{x}_c$ is replaced by a low-cost multiplication.

5 REFLOAT ACCELERATOR ARCHITECTURE

5.1 Accelerator Overview

Figure 6(a) shows the overall architecture of the proposed accelerator for floating-point scientific computing in ReRAM with REFLOAT. We organize the accelerator into multiple banks. Within each bank, ReRAM crossbars are deployed for processing matrix blocks of floating-point MVM. The Input Vector (IV) and Output Vector (OV) buffer are used for buffering the input and output vectors and matrix blocks. The Multiply-and-Accumulate (MAC) units are used to update the vectors. The scheduler is responsible for the coordination of the processing.

5.2 Processing Engine

The most critical component in the accelerator is the processing engine for floating-point SpMV in REFLOAT format. The processing engine consists of a few ReRAM crossbars and several peripheral functional units. We show the architecture of the processing engine in Figure 6(b), assuming we are performing the floating-point SpMV on a matrix block with the format $\texttt{ReFloat}(b,e,f)$.

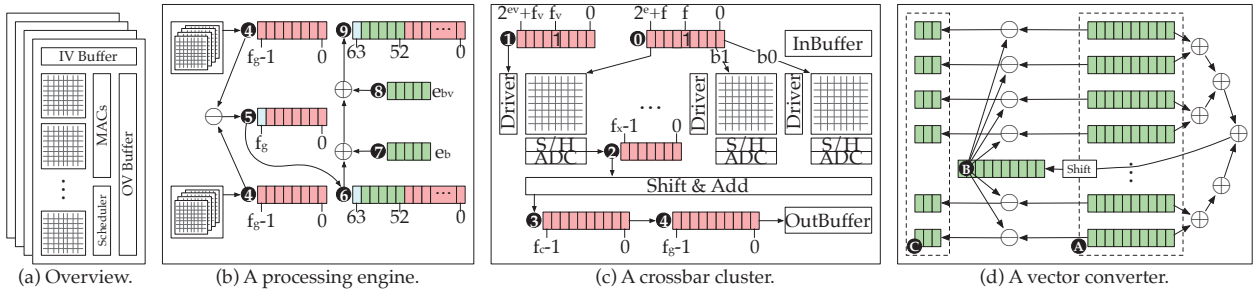
The inputs to the processing engine are: (1) a matrix block in $\texttt{ReFloat}(b,e,f)$ format; (2) an input vector segment in floating-point with e_v exponent bits and f_v fraction bits and the vector length is 2^b; and (3) the exponent base bits e_b for each matrix block. The output of a processing engine is a vector segment for SpMV on the matrix block, which is a double-precision floating-point number.

Before the computation, the matrix block is mapped to the ReRAM crossbars as detailed in Figure 6(c). The fraction part of the matrix block in $\texttt{ReFloat}(b,e,f)$ represents a number of $1.b_{f-1}...b_0$, then we have $(f+1)$ bits for mapping. The e-bit exponent of the matrix block contributes to 2^e padding bits for alignment, then we have another 2^e bits for mapping. Thus, we map the total (2^e+f+1) bits ❼ to (2^e+f+1) ReRAM crossbars, where the i-th bits of the matrix block is mapped to the i-th crossbar [3]. For the input vector segments with e_v exponent bits and f_v exponent bits, a total number of $(2^{e_v}+f_v+1)$ bits ❶ are applied to the driver.

During processing, a cluster of crossbars are deployed to perform the fixed-point MVM for the fraction part of the input vector segment with the fraction part of the matrix block using the shift-and-add method, as the example in Figure 2. The input bits are applied to the crossbars by the driver and the output from the crossbar is buffed by a Sample/Hold (S/H) unit and then converted to digital by a shared Analog/Digital Converter (ADC). For each input bit to the driver (we assume an 1-bit DAC), as the crossbar size is 2^b, the ADC conversion precision is $f_x = b$ bits. Then we need to shift-and-add the results ❷ from all (2^e+f+1) crossbars to get the results ❸ for the 1-bit multiplication of the vector with the matrix fraction. Thus, the bits number of ❸ is $f_c = 2^e+f+1+b$. Next, we sequentially input the bits in ❶ to the crossbars and shift-and-add the collected ❸ for each of the $(2^{e_v}+f_v+1)$ bits to get ❹, which is the result for the multiplication of the matrix block with the input ❶. The bits number of ❹ is $f_g = f_c + 2^{e_v} + f_v + 1 + b$. As shown in Figure 6(b), each matrix block has a sign bit, therefore, it requires two crossbar clusters in a processing engine for the signed multiplication. Each element in the input vector segment also has a sign bit. Thus, we need four ❹ and subtract them to get ❺, which is the multiplication results between the matrix block and the vector segment. The number of bits for ❺ is (f_g+1), and ❺ is a signed number due to the subtraction. Next, we convert the ❺ to a double-precision floating-point ❻. e_b ❼ is the exponent base for the matrix block and e_v ❽ is exponent for the vector segment. We add ❼ and ❽ to the exponent of ❾ to get the ❾— the final results for the multiplication of the matrix block with the vector segment in 64-bit double-precision floating-point.

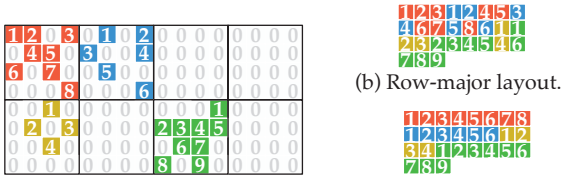
The vector converter is responsible for converting a vector segment in default floating-point precision to REFLOAT for processing in next iteration. Ⓐ the exponents of elements in a vector segment is accumulated by an adder tree and shifted following Eq. (5) to get Ⓑ the vector exponent base e_{bv}. An element-wise subtraction is

[3] Here, we assume that the cell precision for the ReRAM crossbars is 1-bit. For 2-bit cells, two consecutive bits are mapped to a crossbar.



(a) Overview. (b) A processing engine. (c) A crossbar cluster. (d) A vector converter.

Figure 6: (a) the accelerator architecture overview. Architectures of (b) a processing engine for floating-point MVM on a matrix block, (c) a crossbar cluster for fixed-point MVM, and (d) a vector converter.



(a) A sparse matrix. (b) Row-major layout. (c) Block-major layout.

Figure 7: The row-major layout and block-major layout of a sparse matrix.

performed on Ⓐ to get Ⓒ the exponents of the elements in the vector segment.

5.3 Streaming and Scheduling

For the original large-scale sparse matrix, the non-zero elements are stored in either row-major or column-major order. However, the computation in ReRAM crossbars requires accessing elements in a matrix block, i.e., elements indexed by the same window of rows and columns. Thus, there is a mismatch between the data storage format in the original application, e.g., Matrix Market File Format [10], and the most suitable format for ReFloat accelerator. Direct access to the elements in each matrix block will result in random access and wasted memory bandwidth. We propose a block-major layout to overcome this problem, which ensures that most matrix block elements can be read sequentially. Specifically, the non-zeros of each $2^b \times 2^b$ block are stored consecutively, and the non-zeros of every P blocks among the same set of rows are stored linearly before moving to a different set of rows, as shown in Figure 7. Here, P is the number of blocks that can be processed in parallel, which is determined by the hyper-parameters b, e, and f for a given number of available ReRAM crossbars.

6 EVALUATION

6.1 Evaluation Setup

We list the configurations for the baseline GPU platform, the state-of-the-art ReRAM accelerator [32] for scientific computing (Feinberg) and our ReFloat in Table 4. We use an NVIDIA Tesla V100 GPU, which has 5120 Cuda cores and a 32GB HBM2 memory. We use CUDA version 11.7 and cuSPARSE routines in the iterative solvers for the processing on sparse matrices. We measure the running time for the solvers on the GPU. For the two ReRAM accelerators, i.e. Feinberg [32] and ReFloat, we use the parameters in Table 4 for simulation. Both the two ReRAM accelerators have 128 Banks and the crossbar size is 128 × 128. In Feinberg [32], we configure 64 clusters for each bank, which is slightly larger than that (56) in Feinberg [32]. There are 128 crossbars in each cluster. The precision in Feinberg [32] is double floating-point. In ReFloat, we configure 128 banks, 128 subbanks per bank, and 64 crossbars per subbank. The precision in ReFloat is **refloat** with a default setting that $e = 3$, $f = 3$, $e_v = 3$ and $f_v = 8$. For the two ReRAM accelerators, the equivalent computing ReRAM is 17.1Gb. The ADC and ReRAM cells for the two accelerators are of the same configuration. We use

Table 4: Platform Configuration.

GPU (Tesla V100 SXM2)			
Architecture	Volta	CUDA Cores	5120
Memory	32GB HBM2	CUDA Version	11.7
Feinberg [32]			
Bank	128	Crossbar Size	128 × 128
Clusters/Bank	64	Precision	**double**
Xbars/Cluster	128	Comp. ReRAM	17.1Gb
ReFloat			
Bank	128	Crossbar Size	128 × 128
Subbank	128	Precision	**refloat**
Xbars/Subbank	64	Comp. ReRAM	17.1Gb
ADC			
10-bit pipelined SAR ADC @ 1.5GS/s			
ReRAM Cells			
1-bit SLC, T_w = 50.88ns, Comp. Latency=107ns @ (128×128).			

Table 5: Matrices in the evaluation.

ID	Name	#Rows	NNZ	NNZ/R	κ
353	crystm01	4,875	105,339	21.6	4.21e+2
1313	minsurfo	40,806	203,622	5.0	8.11e+1
354	crystm02	13,965	322,905	23.1	4.49e+2
2261	shallow_water1	81,920	327,680	4.0	3.63e+0
1288	wathen100	30,401	471,601	15.5	8.24e+3
1311	gridgena	48,962	512,084	10.5	5.74e+5
1289	wathen120	36,441	565,761	15.5	4.05e+3
355	crystm03	24,696	583,770	23.6	4.68e+2
2257	thermomech_TC	102,158	711,558	6.9	1.23e+2
1848	Dubcova2	65,025	1,030,225	15.84	1.04e+4
2259	thermomech_dM	204,316	1,423,116	6.9	1.24e+2
845	qa8fm	66,127	1,660,579	25.1	1.10e+2

353 1313 354 2261 1288 1311 1289 355 2257 1848 2259 845

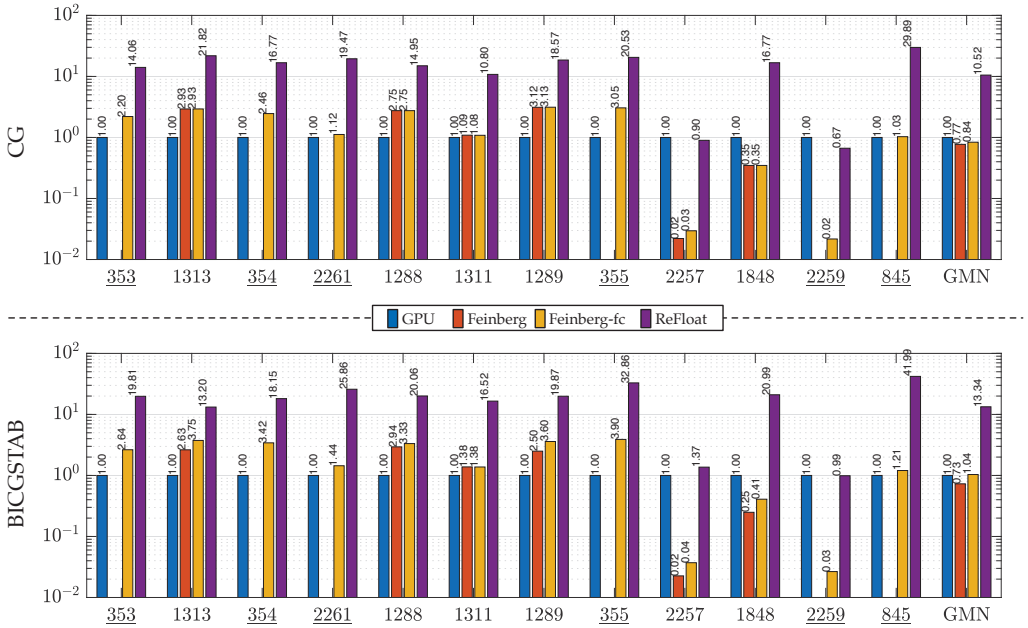


Figure 8: The performance of GPU, Feinberg [32], Feinberg-fc and ReFloat for CG and BiCGSTAB solvers.

a 1.5GS/s 10-bit pipelined SAR ADC [60] for conversion. The DAC is 1-bit, which is implemented by wordline activation. We use 1-bit SLC [74] and the write latency is 50.88ns. The computing latency for one crossbar, including the ADC conversion, is 107ns [32].

Table 5 lists the matrices used in the evaluation. We evaluate on 12 solvable matrices from the SuiteSparse Matrix Collection (formerly UF Sparse Matrix Collection) [24]. The matrices' size (number of rows) ranges from 4,875 to 204,316 and the Number of Non-Zero entries (NNZ) of the matrices ranges from 105,339 for 1,660,579. NNZ/Row is a metric for sparsity. A smaller NNZ/Row indicates a sparser matrix. NNZ/Row ranges from 4.0 to 27.7. The condition number κ ranges widely from 3.6 to 5.74e+5. We also visualize the matrices in Table 5. We apply the iterative solvers CG and BiCGSTAB on the matrices. The convergence criteria for the solvers is that the L-2 norm of the residual vector (we use the term "residual" denoted by R^2 for simplicity to call the L-2 norm of the residual vector in this section) is less than 10^{-8}.

6.2 Performance

We show the performance of the GPU, a state-of-the-art ReRAM accelerator Feinberg [32] and ReFloat for CG and BiCGSTAB solvers in Figure 8. We evaluate the processing time t for the iterative solver to satisfy that the residual is less than 10^{-8}. The performance p is defined as $p = t_{\text{GPU}}/t_x$, x = Feinberg [32], Feinberg-fc or ReFloat. For Feinberg [32], we evaluate both function (convergence) and hardware performance. Note that as we discussed in Sec. 3.3, the vector issue in [32] may lead to non-convergence on most matrices. Feinberg-fc is a strong baseline where we assume the function is correctly the same as that of the default **`double`**. Specifically, we assume Feinberg-fc converges and takes the same iteration number to convergence as that in **`double`** and evaluate the hardware performance of Feinberg-fc.

CG solver. Overall, the geometric-mean(GMN) performance of Feinberg [32]-fc and ReFloat are 0.8362× and 12.59×(up to 29.89×) respectively. GPU and ReFloat converge on all matrices while Feinberg [32] does not converge on 6 out of 12 matrices and the IDs of not converged matrices are 353, 354, 2261, 355, 2259, and 845. The GMN of ReFloat compared to Feinberg [32] on the six converged matrices is 12.94×. For most of the matrices, ReFloat performs better than the baseline GPU. For matrix 2257, 1848 and 2259, the performance of ReFloat is 0.8973×, 16.77× and 0.6660× respectively. However, the performance of Feinberg [32] is even lower, and it is 2.21E-2×, 3.48E-1× and NC respectively. The slow down is because the required number of clusters for SpMV is larger than the number available on the accelerators. If the number of clusters for SpMV on one matrix is fewer than the available clusters on an accelerator, the deployed clusters will be only invoked once to perform the SpMV. But, if the number of clusters for SpMV on one matrix is larger than the available clusters on an accelerator, (1) cell writing for mapping new matrix blocks to clusters and (2) cluster invoking to perform part of SpMV will happen multiple times, thus more time is consumed for one SpMV on the whole matrix. In Feinberg [32], with the default floating-point mapping, i.e., 118 crossbars for a cluster, there are only 2221 clusters available. However, to perform one SpMV on the whole matrix, 209263, 15797, and 381321 clusters are required respectively for matrix 2257, 1848, and 2259. The required cluster number for the two matrix is far larger than the available number in Feinberg [32], resulting in cell writing and cluster invoking 103, 8, and 187 times respectively for the three matrices. So the performance of Feinberg [32] is lower than the baseline GPU on the two matrices. In ReFloat, to perform one SpMV on the whole matrix, the same numbers as that in Feinberg [32] of clusters are required for matrix 2257 and matrix 2259. We configure $e = 3$, $f = 3$ for ReFloat, so the available clusters for matrix 2257 and matrix 2259 are 21845. The cell writing and

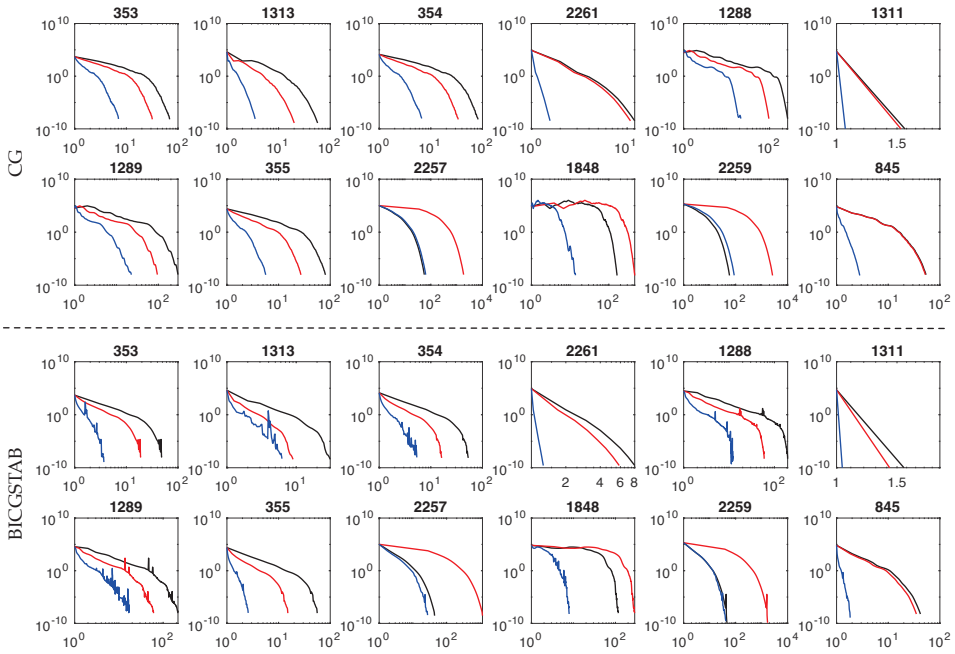


Figure 9: Convergence traces of CG and BiCGSTAB solvers of GPU (black line), Feinberg [32]-fc (red line) and REFLOAT (blue line). The Y axis is the residual and the X axis is the normalized (to GPU) iteration number.

Table 6: Absolute iteration number to reaching convergence.

ID	CG			BiCGSTAB		
	double	refloat	+/-	double	refloat	+/-
353	68	85	+17	49	51	+2
1313	52	55	+3	34	69	+35
354	81	95	+14	58	79	+21
2261	11	11	0	7	7	0
1288	262	305	+43	195	205	+10
1311	1	1	0	1	1	0
1289	294	401	+107	211	317	+106
355	80	95	+15	59	52	-7
2257	55	56	+1	43	36	-7
1848	162	214	+52	118	145	+27
2259	57	58	+1	45	36	-9
845	53	54	+1	41	35	-6

cluster invoking times for matrix 2257 and matrix 2259 are 10 and 18 respectively, which are less than the cell writing and cluster invoking times in Feinberg [32].

Another reason leading to higher performance of REFLOAT compared with Feinberg [32] is that fewer cycles are consumed within a cluster. In Feinberg [32], 233 cycles are consumed for the multiplication even with the assumption that 6 bits are enough for the exponent [32]. In REFLOAT, 28 cycles are consumed for the multiplication. Notice that with a fewer number of exponent bits and fraction bits, we can get (a) a fewer number of clusters required for a whole matrix, (b) a fewer number of cycles consumed for one matrix block floating-point multiplication within a cluster. The two effects (a) and (b) can lead to higher performance, but we also have a third effect (c) larger number of iterations to reaching convergence, which leads to lower performance. However, effects (a) and (b) is stronger than effect (c), so the performance of REFLOAT is higher. The number of iterations for the evaluated matrices to reach convergence is listed in Table 6.

Table 7: Bit number for exponent and fraction of matrix block and vector segment in REFLOAT.

CG				BiCGSTAB			
e	f	e_v	f_v	e	f	e_v	f_v
3	3	3	8	3	3	3	8

BiCGSTAB solver. The geometric-mean(GMN) performance of Feinberg [32]-fc and REFLOAT are 1.036× and 13.34× (up to 41.99×) respectively. The GPU and REFLOAT converge on all matrices while Feinberg [32] does not converge on 6 out of 12 matrices and the IDs of not converged matrices are 353, 354, 2261, 355, 2259, and 845. The GMN of REFLOAT compared to Feinberg [32] on the four converged matrices is 15.98×. The trend for the three platforms on the evaluated matrices are similar to that for CG solver. In each iteration, for CiCGSTAB solver, there are two SpMV on the whole matrix, while for CG solver, there is one SpMV on the whole matrix. From Table 6 we can see, the difference of (+/-) number of iterations to get converge in BiCGSTAB solver is smaller than the gap in CG solver for most matrices. For matrix 355, 2257, 2259 and 845, the difference is negative, which means it takes fewer iterations in **refloat** compared with that in **double**.

6.3 Accuracy

We show the convergence traces (the residual over each iteration) of GPU, Feinberg-fc, and REFLOAT for CG and BiCGSTAB solvers in Figure 9. The iteration number is normalized by the consumed time for the GPU baseline. Table 7 lists the configurations of bit number for matrix block and vector segment in **refloat** for all matrices

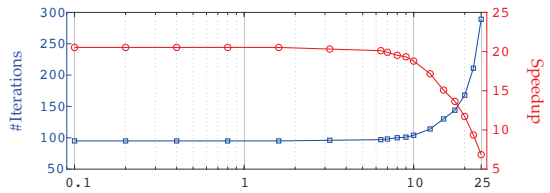


Figure 10: The iteration number and speedup of ReFloat on `crystm03` v.s. noise.

Table 8: Memory overhead of ReFloat v.s. Feinberg [32].

ID	353	1313	354	2261	1288	1311	1289	355	2257	1848	2259	845
	.173	.176	.173	.176	.173	.174	.173	.173	.312	.179	.300	.173

except 1288 and 1828. For 1288 and 1828, the only difference is the $f_v = 16$. The absolute (non-normalized) iteration number to reach convergence is listed in Table 6.

For CG solver, from Table 6 we can see, **`refloat`** leads to more number of iterations to get converged when we do not consider the time consumption for each iteration. From Figure 9 we can see, with the low bit representation, the residual curves are almost the same trend as the residual curves of GPU and Feinberg-fc in default **`double`**. Most importantly, all the traces in **`refloat`** get converged faster than the traces of GPU and Feinberg-fc. For matrix 1288 and matrix 1848, the bit number for fraction of vector segment is 16 because the default 8 leads to non-convergence. For BiCGSTAB solver, from Table 6 we can see, while **`refloat`** leads to more number of iterations to reaching convergence for 5 matrices, the number of iterations to reaching convergence for 4 matrices are even fewer than those in **`double`**. We infer that is because lower bit representation helps to enlarge the changes in the correction term, thus leads to fewer iterations. We also notice there are spikes in the residual curves in **`refloat`** more frequently than spikes in **`double`**, but they finally reach convergence.

6.4 Robustness to Noise

To study the robustness to noise of ReFloat, we disable the error correction. We model the random telegraph noise (RTN) [17] which is widely adopted in ReRAM accelerator noise modeling [3, 32, 47]. We use `crystm03` with CG solver for a case study and show the speedup (compared to GPU) and iteration number v.s. noise deviation σ from 0.1% to 25% in Figure 10. Within 10% noise, the speedup degrades very little and at 25% noise, ReFloat still maintains a 6.85× speedup. As we discussed before, the iterative solvers naturally tolerate noise and deviation.

6.5 Memory Overhead

In Table 8, we compare the memory overhead for the matrix in **`refloat`** normalized to that in **`double`** (used in Feinberg [32]). On average, **`refloat`** consumes 0.192× memory compared with **`double`**. For matrices except 2257 and 2259, **`refloat`** consumes less than 0.2× memory compared with **`double`**. For matrix 2257 and matrix 2259, the average density within a matrix is relatively lower, thus more memory is consumed for the matrix block index and the exponent base.

7 RELATED WORKS

ReRAM-based accelerators. In recent years, the architecture design of ReRAM-based accelerators have been developed for various applications, including deep learning [6, 11, 16, 33, 49, 50, 54, 81, 88, 104], graph processing [13, 89] and scientific computing [32]. The noise and reliability issues in ReRAM-based computing are significantly alleviated by coding techniques and architectural optimizations [33, 70, 96–98]. ReRAM-based accelerators are demonstrated on silicon by [15, 69, 76, 93, 101–103]. Most ReRAM-based accelerators are designed for fixed-point processing, especially for deep learning. Besides [32], [34] applied preconditioner and FloatPIM [49] accelerated floating-point multiplication in ReRAM, but FloatPIM is designed for deep learning in full-precision floating point.

Scientific computing acceleration. Computing routines on general-purpose platforms CPUs and GPUs have been developed for scientific computing, such as CuSPARSE [73], MKL [94], and LAPACK [5]. Architectural and architecture-related optimizations [23, 35, 55, 61, 68, 82, 105] on CPUs/GPUs are explored for accelerating scientific computing. [26–28] leveraged machine learning for the acceleration of scientific computing and [85–87] accelerated sparse linear algebra and solvers on FPGAs. Scientific computing is a major application in high performance computing and heavily relies on general-purpose platforms, but it is a new application domain for emerging PIM architectures and it is challenging because of high cost and low performance of floating-point processing.

Data format. Data formats for efficient computing are explored for CPUs/GPUs [9, 53, 65, 67, 71, 83]. Format and architecture co-optimization includes [37, 43, 90] on CMOS platforms but they are not for emerging PIM architectures and not for scientific computing. Data compression are explored on DRAM systems [62, 77, 78].

8 CONCLUSION

ReRAM has been proved promising for accelerating fixed-point applications such as machine learning, while scientific computing is an application domain that requires floating-point processing. The main challenge for efficiently accelerating scientific computing in ReRAM is how to support low-cost floating-point SpMV in ReRAM. In this work, we address this challenge by proposing ReFloat, a data format, and a supporting accelerator architecture. ReFloat is tailored for processing on ReRAM crossbars. The number of effective bits is significantly reduced to reduce the crossbar cost and cycle cost for the floating-point multiplication on a matrix block. The evaluation results across a variety of benchmarks reveal that the ReFloat accelerator delivers a speedup of 5.02× to 84.28× compared with a state-of-the-art ReRAM-based accelerator [32] for scientific computing even with the assumption that the accelerator [32] functions the same as FP64 solvers. We released the source code at https://github.com/linghaosong/ReFloat.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedbacks. This work is supported by NSF EPMD-1955246, CNS-2112562, and ARO W911NF-19-2-0107.

REFERENCES

[1] [n. d.]. Build and train machine learning models on our new Google Cloud TPUs. https://www.blog.google/topics/google-cloud/google-cloud-offer-tpus-machine-learning/.

[2] [n. d.]. Google supercharges machine learning tasks with TPU custom chip. https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html.

[3] Sapan Agarwal, Steven J Plimpton, David R Hughart, Alexander H Hsia, Isaac Richter, Jonathan A Cox, Conrad D James, and Matthew J Marinella. 2016. Resistive memory device requirements for a neural algorithm accelerator. In *2016 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 929–938.

[4] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive random access memory (ReRAM) based on metal oxides. *Proc. IEEE* 98, 12 (2010), 2237–2251.

[5] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. 1999. *LAPACK users' guide*. SIAM.

[6] 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*. ACM, 715–731.

[7] Athanasios C Antoulas. 2005. *Approximation of large-scale dynamical systems*. Vol. 6. Siam.

[8] Mario Arioli, James W Demmel, and Iain S Duff. 1989. Solving sparse linear systems with sparse backward error. *SIAM J. Matrix Anal. Appl.* 10, 2 (1989), 165–190.

[9] Brett W Bader and Tamara G Kolda. 2008. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing* 30, 1 (2008), 205–231.

[10] Ronald F Boisvert, Roldan Pozo, Karin Remington, Richard F Barrett, and Jack J Dongarra. 1997. Matrix market: a web resource for test matrix collections. In *Quality of Numerical Software*. Springer, 125–137.

[11] Mahdi Nazm Bojnordi and Engin Ipek. 2016. Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1–13.

[12] AJ Bower. 1990. NICAM 728: Digital two-channel sound for terrestrial television. *STIN* 91 (1990), 15460.

[13] Nagadastagiri Challapalle, Sahithi Rampalli, Linghao Song, Nandhini Chandramoorthy, Karthik Swaminathan, John Sampson, Yiran Chen, and Vijaykrishnan Narayanan. 2020. GaaS-X: graph analytics accelerator supporting sparse data representation using crossbar architectures. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 433–445.

[14] Todd Chapman, Philip Avery, Pat Collins, and Charbel Farhat. 2017. Accelerated mesh sampling for the hyper reduction of nonlinear computational models. *Internat. J. Numer. Methods Engrg.* 109, 12 (2017), 1623–1654.

[15] Wei-Hao Chen, Kai-Xiang Li, Wei-Yu Lin, Kuo-Hsiang Hsu, Pin-Yi Li, Cheng-Han Yang, Cheng-Xin Xue, En-Yu Yang, Yen-Kai Chen, Yun-Sheng Chang, et al. 2018. A 65nm 1Mb nonvolatile computing-in-memory ReRAM macro with sub-16ns multiply-and-accumulate for binary DNN AI edge processors. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 494–496.

[16] 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. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. 27–39.

[17] Shinhyun Choi, Yuchao Yang, and Wei Lu. 2014. Random telegraph noise and resistance switching analysis of oxide based resistive memory. *Nanoscale* 6, 1 (2014), 400–404.

[18] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. 2018. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro* 38, 2 (2018), 8–20.

[19] IEEE Standards Committee et al. 2008. 754-2008 IEEE standard for floating-point arithmetic. *IEEE Computer Society Std* 2008 (2008), 517.

[20] Altera Corporation. 2005. TFFT/IFFT Block Floating Point Scaling. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an404.pdf.

[21] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*. 3123–3131.

[22] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).

[23] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. 2019. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing*. 46–57.

[24] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1.

[25] James W Demmel. 1997. *Applied numerical linear algebra*. Vol. 56. Siam.

[26] Wenqian Dong, Gokcen Kestor, and Dong Li. 2023. Auto-HPCnet: An Automatic Framework to Build Neural Network-based Surrogate for High-Performance Computing Applications. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*. 31–44.

[27] Wenqian Dong, Jie Liu, Zhen Xie, and Dong Li. 2019. Adaptive neural network-based approximation to accelerate eulerian fluid simulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–22.

[28] Wenqian Dong, Zhen Xie, Gokcen Kestor, and Dong Li. 2020. Smart-PGSim: Using neural network to accelerate AC-OPF power grid simulation. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[29] David Elam and Cesar Lovescu. 2003. A block floating point implementation for an N-point FFT on the TMS320C55X DSP. *Texas Instruments Application Report* (2003).

[30] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. 2011. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 365–376.

[31] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. 2004. GPU Cluster for High Performance Computing. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, USA, 47. https://doi.org/10.1109/SC.2004.26

[32] Ben Feinberg, Uday Kumar Reddy Vengalam, Nathan Whitehair, Shibo Wang, and Engin Ipek. 2018. Enabling scientific computing on memristive accelerators. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 367–382.

[33] Ben Feinberg, Shibo Wang, and Engin Ipek. 2018. Making memristive neural network accelerators reliable. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 52–65.

[34] Ben Feinberg, Ryan Wong, T Patrick Xiao, Christopher H Bennett, Jacob N Rohan, Erik G Boman, Matthew J Marinella, Sapan Agarwal, and Engin Ipek. [n. d.]. An Analog Preconditioner for Solving Linear Systems. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 761–774.

[35] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. 2021. EGEMM-TC: accelerating scientific computing on tensor cores with extended precision. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 278–291.

[36] Joel H Ferziger and Milovan Perić. 2002. *Computational methods for fluid dynamics*. Vol. 3. Springer.

[37] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S Chung, and Greg Stitt. 2014. A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 36–43.

[38] David J Frank, Robert H Dennard, Edward Nowak, Paul M Solomon, Yuan Taur, and Hon-Sum Philip Wong. 2001. Device scaling limits of Si MOSFETs and their application dependencies. *Proc. IEEE* 89, 3 (2001), 259–288.

[39] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. 2018. In-Memory Data Parallel Processor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. ACM, 1–14.

[40] Gene H Golub and James M Ortega. 2014. *Scientific computing: an introduction with parallel computing*. Elsevier.

[41] Andreas Griewank and Andrea Walther. 2008. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Vol. 105. Siam.

[42] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *International Conference on Machine Learning*. 1737–1746.

[43] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 243–254.

[44] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[45] Paul Harrison and Alex Valavanis. 2016. *Quantum wells, wires and dots: theoretical and computational physics of semiconductor nanostructures*. John Wiley & Sons.

[46] Magnus Rudolph Hestenes and Eduard Stiefel. 1952. *Methods of conjugate gradients for solving linear systems*. Vol. 49. NBS Washington, DC.

[47] 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 *Proceedings of the 53rd annual design automation conference*. ACM, 19.

[48] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research* 18, 1 (2017), 6869–6898.

[49] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. Floatpim: In-memory acceleration of deep neural network training with high precision. In *Proceedings of the 46th International Symposium on Computer Architecture*. 802–815.

[50] Mohsen Imani, Mohammad Samragh Razlighi, Yeseong Kim, Saransh Gupta, Farinaz Koushanfar, and Tajana Rosing. 2020. Deep learning acceleration with neuron-to-memory transformation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1–14.

[51] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE, 2704–2713.

[52] Frank Jensen. 2017. *Introduction to computational chemistry*. John wiley & sons.

[53] Inah Jeon, Evangelos E Papalexakis, Uksong Kang, and Christos Faloutsos. 2015. Haten2: Billion-scale tensor decompositions. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 1047–1058.

[54] Yu Ji, Youyang Zhang, Xinfeng Xie, Shuangchen Li, Peiqi Wang, Xing Hu, Youhui Zhang, and Yuan Xie. 2019. FPSA: A Full System Stack Solution for Reconfigurable ReRAM-based NN Accelerator Architecture. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 733–747.

[55] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. 2018. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826* (2018).

[56] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 1–12.

[57] Paresh Kharya. [n. d.]. TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x. https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/.

[58] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. 2015. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530* (2015).

[59] O Klank and D Rottmann. 1989. DSR-receiver for the digital sound broadcasting via the European satellites TV-SAT/TDF. *IEEE Transactions on Consumer Electronics* 35, 3 (1989), 504–511.

[60] Lukas Kull, Danny Luu, Christian Menolfi, Matthias Braendli, Pier Andrea Francese, Thomas Morf, Marcel Kossel, Hazar Yueksel, Alessandro Cevrero, Ilter Ozkaya, et al. 2017. 28.5 A 10b 1.5 GS/s pipelined-SAR ADC with background second-stage common-mode regulation and offset calibration in 14nm CMOS FinFET. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 474–475.

[61] Junjie Lai and André Seznec. 2013. Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 1–10.

[62] Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Won Woo Ro, and Murali Annavaram. 2015. Warped-compression: Enabling power efficient GPUs through register compression. *ACM SIGARCH Computer Architecture News* 43, 3S (2015), 502–514.

[63] Bing Li, Linghao Song, Fan Chen, Xuehai Qian, Yiran Chen, and Hai Helen Li. 2018. Reram-based accelerator for deep learning. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 815–820.

[64] Fengfu Li, Bo Zhang, and Bin Liu. 2016. Ternary weight networks. *arXiv preprint arXiv:1605.04711* (2016).

[65] Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: Hierarchical storage of sparse tensors. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 238–252.

[66] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. 2015. Neural networks with few multiplications. *arXiv preprint arXiv:1510.03009* (2015).

[67] Bangtian Liu, Chengyao Wen, Anand D Sarwate, and Maryam Mehri Dehnavi. 2017. A unified optimization approach for sparse tensor operations on gpus. In *2017 IEEE international conference on cluster computing (CLUSTER)*. IEEE, 47–57.

[68] Changxi Liu, Biwei Xie, Xin Liu, Wei Xue, Hailong Yang, and Xu Liu. 2018. Towards efficient SpMV on sunway manycore architectures. In *Proceedings of the 2018 International Conference on Supercomputing*. 363–373.

[69] Qi Liu, Bin Gao, Peng Yao, Dong Wu, Junren Chen, Yachuan Pang, Wenqiang Zhang, Yan Liao, Cheng-Xin Xue, Wei-Hao Chen, et al. 2020. 33.2 A fully integrated analog ReRAM based 78.4 TOPS/W compute-in-memory chip with fully parallel MAC computing. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 500–502.

[70] Tao Liu, Wujie Wen, Lei Jiang, Yanzhi Wang, Chengmo Yang, and Gang Quan. 2019. A fault-tolerant neural network architecture. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[71] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 339–350.

[72] Cleve B Moler. 1967. Iterative refinement in floating point. *Journal of the ACM (JACM)* 14, 2 (1967), 316–321.

[73] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. 2010. Cusparse library. In *GPU Technology Conference*.

[74] Dimin Niu, Cong Xu, Naveen Muralimanohar, Norman P Jouppi, and Yuan Xie. 2013. Design of cross-point metal-oxide ReRAM emphasizing reliability and cost. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 17–23.

[75] Marco S Nobile, Paolo Cazzaniga, Andrea Tangherloni, and Daniela Besozzi. 2017. Graphics processing units in bioinformatics, computational biology and systems biology. *Briefings in bioinformatics* 18, 5 (2017), 870–885.

[76] Yachuan Pang, Bin Gao, Dong Wu, Shengyu Yi, Qi Liu, Wei-Hao Chen, Ting-Wei Chang, Wei-En Lin, Xiaoyu Sun, Shimeng Yu, et al. 2019. 25.2 A reconfigurable RRAM physically unclonable function utilizing post-process randomness source with< 6× 10- 6 native bit error rate. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 402–404.

[77] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. 2013. Linearly compressed pages: A low-complexity, low-latency main memory compression framework. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 172–184.

[78] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Michael A Kozuch, Phillip B Gibbons, and Todd C Mowry. 2012. Base-delta-immediate compression: Practical data compression for on-chip caches. In *2012 21st international conference on parallel architectures and compilation techniques (PACT)*. IEEE, 377–388.

[79] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, 525–542.

[80] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. Vol. 82. siam.

[81] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 14–26.

[82] Du Shen, Shuaiwen Leon Song, Ang Li, and Xu Liu. 2018. Cudaadvisor: Llvm-based runtime profiling for modern gpus. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 214–227.

[83] Shaden Smith and George Karypis. 2015. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. 1–7.

[84] Fengguang Song, Stanimire Tomov, and Jack Dongarra. 2012. Enabling and Scaling Matrix Computations on Heterogeneous Multi-Core and Multi-GPU Systems. In *Proceedings of the 26th ACM International Conference on Supercomputing* (San Servolo Island, Venice, Italy) *(ICS '12)*. Association for Computing Machinery, New York, NY, USA, 365–376. https://doi.org/10.1145/2304576.2304625

[85] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. 2022. Serpens: A high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 211–216.

[86] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. 2022. Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 65–77.

[87] Linghao Song, Licheng Guo, Suhail Basalama, Yuze Chi, Robert F Lucas, and Jason Cong. 2023. Callipepla: Stream Centric Instruction Set and Mixed Precision for Accelerating Conjugate Gradient Solver. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 247–258.

[88] 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)*. IEEE, 541–552.

[89] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating graph processing using ReRAM. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 531–543.

[90] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 689–702.

[91] Henk A Van der Vorst. 1992. 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* 13, 2 (1992), 631–644.

[92] M Mitchell Waldrop. 2016. The chips are down for Moore's law. *Nature News* 530, 7589 (2016), 144.

[93] Weier Wan, Rajkumar Kubendran, S Burc Eryilmaz, Wenqiang Zhang, Yan Liao, Dabin Wu, Stephen Deiss, Bin Gao, Priyanka Raina, Siddharth Joshi, et al. 2020. 33.1 a 74 tmacs/w cmos-rram neurosynaptic core with dynamically reconfigurable dataflow and in-situ transposable weights for probabilistic graphical models. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 498–500.

[94] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.

[95] Shibo Wang and Pankaj Kanwar. 2019. BFloat16: the secret to high performance on cloud TPUs. *Google Cloud Blog* (2019).

[96] Wen Wen, Youtao Zhang, and Jun Yang. 2018. Wear leveling for crossbar resistive memory. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[97] Wen Wen, Youtao Zhang, and Jun Yang. 2019. Renew: Enhancing lifetime for reram crossbar based neural network accelerators. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*. IEEE, 487–496.

[98] Wen Wen, Youtao Zhang, and Jun Yang. 2020. Accelerating 3D vertical resistive memories with opportunistic write latency reduction. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–8.

[99] James Hardy Wilkinson. 1994. *Rounding errors in algebraic processes*. Courier Corporation.

[100] H-S Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T Chen, and Ming-Jinn Tsai. 2012. Metal–oxide RRAM. *Proc. IEEE* 100, 6 (2012), 1951–1970.

[101] Tony F Wu, Haitong Li, Ping-Chen Huang, Abbas Rahimi, Jan M Rabaey, H-S Philip Wong, Max M Shulaker, and Subhasish Mitra. 2018. Brain-inspired computing exploiting carbon nanotube FETs and resistive RAM: Hyperdimensional computing case study. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 492–494.

[102] Cheng-Xin Xue, Wei-Hao Chen, Je-Syu Liu, Jia-Fang Li, Wei-Yu Lin, Wei-En Lin, Jing-Hong Wang, Wei-Chen Wei, Ting-Wei Chang, Tung-Cheng Chang, et al. 2019. 24.1 a 1Mb multibit ReRAM computing-in-memory macro with 14.6 ns parallel MAC computing time for CNN based AI edge processors. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 388–390.

[103] Cheng-Xin Xue, Tsung-Yuan Huang, Je-Syu Liu, Ting-Wei Chang, Hui-Yao Kao, Jing-Hong Wang, Ta-Wei Liu, Shih-Ying Wei, Sheng-Po Huang, Wei-Chen Wei, et al. 2020. 15.4 A 22nm 2Mb ReRAM Compute-in-Memory Macro with 121-28TOPS/W for Multibit MAC Computing for Tiny AI Edge Devices. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 244–246.

[104] Tzu-Hsien Yang, Hsiang-Yun Cheng, Chia-Lin Yang, I Tseng, Han-Wen Hu, Hung-Sheng Chang, Hsiang-Pang Li, et al. 2019. Sparse ReRAM engine: joint exploration of activation and weight sparsity in compressed neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 236–249.

[105] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the gpu microarchitecture to achieve bare-metal performance tuning. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 31–43.

[106] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044* (2017).

[107] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2016. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160* (2016).

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT DOI

10.5281/zenodo.8126949

ARTIFACT IDENTIFICATION

We present ReFloat in this work. ReFloat is a ReRAM based accelerator architecture for linear solvers. The computational artifacts include a GPU implementation of conjugate gradient (CG) and biconjugate gradient stabilized method (BiCGSTAB) for solving linear systems, an implementation of a C++ based simulator for the ReFloat accelerator architecture. The GPU implementation serves as the baseline for the evaluation, the C++ simulator simulates the performance of the ReRAM accelerators.

REPRODUCIBILITY OF EXPERIMENTS

The workflow will be: (1) download matrices from SuiteSparse, (2) compile the GPU code and the simulators' code, (3) run a script to report the evaluation results. The estimated execution time is 2 hours. The expected results are the speedups reported in Figure 8, the iteration numbers reported in Table 6. The evaluation results show that our accelerator ReFloat performs better than the GPU and the accelerator presented by Feinberg et al (ISCA'18), and the accelerator presented by Feinberg et al does not converge on many matrices, but our accelerator does.

The evaluation requires (1) an NVIDIA Tesla V100 GPU, or a similar NVIDIA GPU, (2) CUDA version 11.7, and (3) Ubuntu 20.04. The datasets are matrices from SuiteSparse. We provided a script to download the matrices. The source code is available on https://github.com/linghaosong/ReFloat. We provided detailed steps about how to compile, how to download the input matrices, and how to run the evaluation.

ARTIFACT DEPENDENCIES REQUIREMENTS

1. An NVIDIA Tesla V100 GPU, or a similar NVIDIA GPU. 2. Cuda version 11.7. 3. Ubuntu 20.04. 4. The datasets are matrices from SuiteSparse. We provided a script to download the matrices. 5. The source code is available on https://github.com/linghaosong/ReFloat. We provided detailed steps about how to compile, how to download the input matrices, and how to run the evaluation.

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

To obtain a copy of the source code

git clone https://github.com/linghaosong/ReFloat.git

To download the matrices

cd ReFloat/matrices sh download.sh

We have tested the GPU implemneation on a Nvidia V100 GPU with CCuSparse (CUDA version 11.7). To compile the GPU code,

cd ReFloat/gpu make

We provided a script to run the evaluations,

sh run_gpu.sh

We provided the CPU implemneation and the simulation code under the src directory. We suggest that your CPU paltform has OpenMP installed. To compile,

cd src make

To run the CPU implementation,

cd run/cpu sh run_cpu.sh

To run the ReFloat, go to

cd ReFloat/src/run/refloat sh run_refloat.sh