ReCG: ReRAM-Accelerated Sparse Conjugate Gradient

Mingjia Fan
SSSLab, China University of Petroleum-Beijing

Xiaoming Chen
Institute of Computing Technology, Chinese Academy of Sciences

Dechuang Yang
SSSLab, China University of Petroleum-Beijing

Zhou Jin
SSSLab, China University of Petroleum-Beijing

Weifeng Liu
SSSLab, China University of Petroleum-Beijing

ABSTRACT

Solving sparse linear systems is crucial in scientific computing. Sparse Conjugate Gradient (CG) is one of the most well-known iterative solvers with high efficiency and low storage requirements. However, the performance of sparse CG solvers implemented on storage-compute separated architectures is greatly limited by the irregular memory access and the large amount of data transmission.

In this paper, we propose a processing-in-memory (PIM) architecture, ReCG, based on the resistive random-access memory (ReRAM) to accelerate sparse CG solvers. The design of ReCG faces three major challenges: (1) how to make complex sparse CG more suitable for acceleration with ReRAM-based architecture, (2) how to map sparse and irregular operations to regular crossbars that are more suitable for dense operations, and (3) how to coordinate the dataflow among hardware units to minimize the impact of the poor write endurance of ReRAMs on CG acceleration. To address these challenges, we (1) classify the kernels of sparse CG by exploring the commonality of operations and design a flexible and dedicated architecture, (2) efficiently implement the sparse and irregular operations by utilizing both content-addressable memory (CAM) and multiply-and-accumulate (MAC) crossbars, and (3) develop a novel scheduling strategy for the dataflow. The experimental results show that ReCG improves the performance by up to three, one and one order of magnitude compared with PETSc on CPU and GPU and CALLIPEPLA on FPGA, respectively, and the energy consumption is reduced by up to two, two and one order of magnitude.

KEYWORDS

Sparse Conjugate Gradient, Processing-in-memory, ReRAM, Iterative Solver, Sparse Linear Algebra

1 INTRODUCTION

Solving sparse linear systems is essential in scientific and engineering fields, such as fluid dynamics simulation and optimization problems, etc., and it usually dominates the total runtime [1]. Since the performance of sparse linear solvers is usually limited by the irregular memory access and the large amount of data transmission, improving the efficiency has become a key challenge. Conjugate Gradient (CG) [2] is an effective iterative method for solving sparse linear systems due to its efficient convergence speed and low memory requirement. Coupled with the Jacobi preconditioner, JPCG (Jacobi Preconditioned Conjugate Gradient) is efficient for solving symmetric positive definite linear systems [3].

Over the past decades, researchers have made a lot of efforts to accelerate CG on CPU, GPU, and FPGA platforms, and have achieved performance improvements [4, 5]. However, due to the limitations of the architectures, data transmission between memories and computing units requires considerable time and energy consumption, which significantly restricts the overall performance, and has become a bottleneck to accelerate sparse CG. In contrast, processing-in-memory (PIM) breaks through architectural constraints and allows in-situ computation with memory [6]. The resistive random access-memory (ReRAM) [7], an emerging non-volatile memory technology, is considered as a promising candidate for realizing PIM architectures. Using ReRAMs enables us to customize PIM accelerators for the sparse CG solver, thus overcoming the limitations of storage-compute separated architectures.

However, it is challenging to implement the sparse CG algorithm on ReRAM-based PIM architectures. Firstly, the sparse CG algorithm itself is complex. There are more than 10 kernels, which involve matrices, vectors and scalars, and there are also irregular operations such as Reduction, which makes it difficult to realize the CG algorithm efficiently on PIM architectures. Secondly, sparse operations in CG, e.g., Sparse matrix-vector multiplication (SpMV), are difficult to be efficiently mapped onto the multiply-and-accumulate (MAC) crossbars, which are more suitable for dense operations. Finally, the write endurance of ReRAM is poor, and there are complex dataflow and data handling processes in the CG algorithm, which makes it difficult to reduce the write cost. These challenges must be well addressed to realize a high-performance sparse CG solver on ReRAM-based PIM accelerators.

This paper proposes ReCG, a ReRAM-based PIM architecture for accelerating sparse CG solvers. For the challenges mentioned above, we propose corresponding solutions. Firstly, we analyze the sparse CG algorithm and find that the kernels can be classified into three categories: sparse matrix operations, vector operations and scalar operations. Therefore, we make full use of the commonality of the operations to design a flexible and dedicated architecture, and design a hardware structure composed of tree accumulators and MAC crossbars to implement the irregular operations like Reduction. Secondly, we propose a new sparse matrix compression storage format, which efficiently computes the results of the entire SpMV within a four-phase workflow. Finally, we comprehensively analyze the complex dataflow and data dependencies in CG and

DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3656515

propose a new dataflow scheduling strategy. With this strategy, we successfully reduce the data transmission amount by almost half and parallelize most of the operations in CG. At the same time, this strategy also lowers the write cost on ReRAMs. The contributions of this work are summarized as follows:

- To the authors' knowledge, this is the first ReRAM-based PIM JPCG accelerator, ReCG.
- We design a flexible architecture and formulate a hardware structure to implement irregular operations like Reduction in parallel using adders and MAC crossbars.
- We propose a new sparse matrix compression storage format and a four-phase workflow that efficiently utilizes half of the indexes to parallelly compute the entire SpMV.
- We present a novel dataflow scheduling strategy that parallelizes the entire algorithm and reduces the effect of poor write endurance of ReRAM on accelerating JPCG.

2 BACKGROUND

2.1 Jacobi Preconditioned Conjugate Gradient

For solving the linear system $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is sparse, there are two kinds of methods: direct methods and iterative methods. Direct methods suffer from the huge memory consumption and extremely complex algorithm implementations, so for large-scale problems, iterative methods are more preferred. CG [2] is a widely-used iterative method. To improve the convergence, preconditioners are usually needed. The Jacobi preconditioner [3] approximates matrix $\mathbf{A}$ with its diagonal and can help reduce the number of iterations and speed up the convergence. JPCG is an important solver that is extensively used in practice.

Algorithm 1 JPCG solver for solving linear system $\mathbf{Ax} = \mathbf{b}$

```
Input: $\mathbf{A}$, $\mathbf{b}$, $\mathbf{M}^{-1}$ ($\mathbf{M}$ is the Jacobi preconditioner), $\mathbf{x_0}$ (initial solution vector), $\tau$ (convergence threshold), $N$ (maximum number of iterations)
Output: $\mathbf{x}$
1:  $\mathbf{r} \leftarrow \mathbf{b} - \mathbf{A} \cdot \mathbf{x}$                {Initialize residual vector r}
2:  $\mathbf{z} \leftarrow \mathbf{M}^{-1} \cdot \mathbf{r}$                {Preprocess residual vector r}
3:  $\mathbf{p} \leftarrow \mathbf{z}$                {Initialize search direction}
4:  $rz \leftarrow \mathbf{r}^{\mathrm{T}} \cdot \mathbf{z}$                {Initialize $rz$}
5:  $rr \leftarrow \mathbf{r}^{\mathrm{T}} \cdot \mathbf{r}$                {Initialize $rr$}
6:  for ($i = 0$; $i < N$ and $rr > \tau$; $i++$)
7:      $\mathbf{ap} \leftarrow \mathbf{A} \cdot \mathbf{p}$                {Compute ap}
8:      $\alpha \leftarrow rz/(\mathbf{p}^{\mathrm{T}} \cdot \mathbf{ap})$                {Compute step size $\alpha$}
9:      $\mathbf{x} \leftarrow \mathbf{x} + \alpha \cdot \mathbf{p}$                {Update solution vector x}
10:     $\mathbf{r} \leftarrow \mathbf{r} + \alpha \cdot \mathbf{ap}$                {Update residual vector r}
11:     $\mathbf{z} \leftarrow \mathbf{M}^{-1} \cdot \mathbf{r}$                {Update preconditioning vector z}
12:     $rz_{new} \leftarrow \mathbf{r}^{\mathrm{T}} \cdot \mathbf{z}$                {Compute $rz_{new}$}
13:     $\mathbf{p} \leftarrow \mathbf{z} + (rz_{new}/rz) \cdot \mathbf{p}$                {Update search direction}
14:     $rz \leftarrow rz_{new}$                {Update $rz$}
15:     $rr \leftarrow \mathbf{r}^{\mathrm{T}} \cdot \mathbf{r}$                {Update $rr$}
```

Algorithm 1 demonstrates the flow of JPCG. Lines 1 to 3 initialize the residual vector $\mathbf{r}$, using the preconditioning matrix $\mathbf{M}$ to preprocess the residual vector $\mathbf{r}$, and determining the search direction according to the preconditioning vector. Line 4 initializes the rz, which can be used for updating the search direction. Line 5 is used to compute the square of the norm of the residual vector $\mathbf{r}$, which can measure the size of the residuals, thus evaluating the convergence of the algorithm. Lines 7 to 15 constitute a loop that updates the intermediate vectors and scalars. As can be seen, JPCG involves the processing of many different kernels. Its core operations include SpMV, Reduction operation, and operations between matrices and vectors. The SpMV plays a crucial role in the JPCG algorithm, which accounts for more than 70% of the overall algorithm's computation time, as shown in Fig. 1. This work also focuses on solving the challenges of such sparse operations to better utilize the ReRAM-based PIM technique for JPCG acceleration.

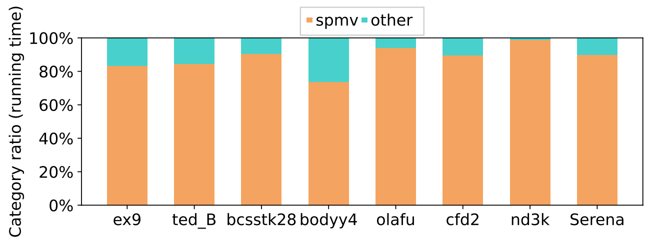


Figure 1: The running time of SpMV in the JPCG.

2.2 Resistive Random-Access Memory

The use of new non-volatile memory devices in PIM technology has become a new hot research topic in recent years. The most important feature of PIM is the ability to perform in-situ computation within memory, which fundamentally avoids data movement. Among emerging non-volatile memory devices, one of the most widely used is ReRAM. ReRAM has the characteristics of high density, fast read and low leakage power. ReRAM-based crossbar arrays have the remarkable capability to perform MAC operations [8–10] through MAC arrays and search operations through CAM arrays.

ReRAM-based MAC Crossbar. Fig. 2(a) illustrates the ReRAM-based crossbar array structure, which boasts the capability of conducting in-situ matrix-vector multiplication (MVM) operations [11]. Voltages are input through the wordlines, and currents are output from the bitlines. An input voltage applied to the crossbar cells is converted into currents through the utilization of appropriately configured cell conductances. Furthermore, in accordance with Kirchhoff's law, the currents from the cells in the same column are accumulated along the bitline, thus performing a MAC operation.

ReRAM-based CAM Crossbar. The content addressable memory (CAM) is capable of performing data search functions, while the ternary content addressable memory (TCAM) introduces a matching function that allows for the ignoring of specific bits, thereby achieving a more flexible and efficient data search process. Fig. 2(b) shows the ReRAM-based TCAM structure [12, 13], where each cell is used to represent a bit of data. The TCAM implementation utilizes XNOR logic on each cell to perform bitwise search operations, enabling parallel processing, promoting fast associative matching, and improving the efficiency of data search.

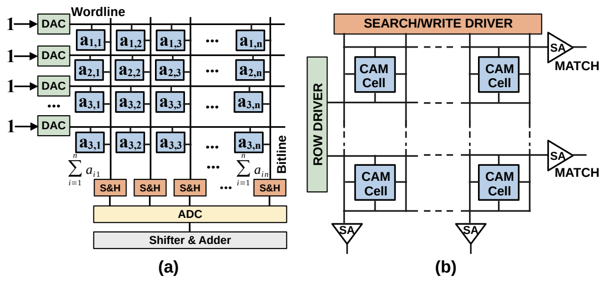
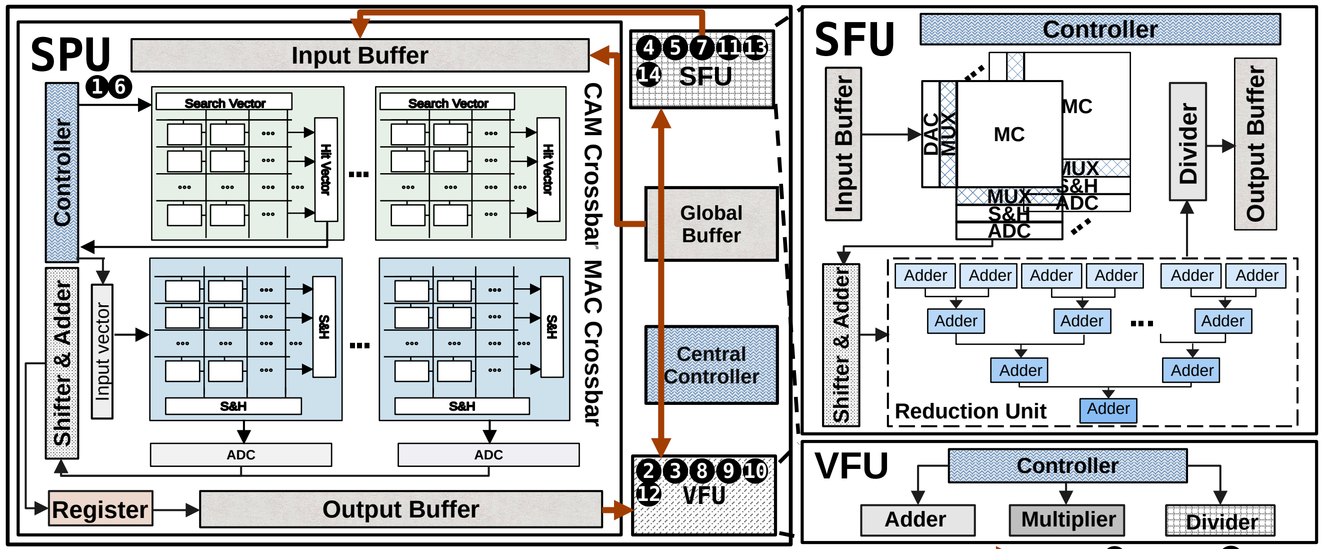


Figure 2: (a) The MAC array. (b) The CAM array.



VFU:Vector Function Unit **SFU**:Scalar Function Unit **SPU**:Sparse Function Unit **MC**:MAC Crossbar **S&H**:Sample and Hold ➡Data Flow ❶ Compute r ❷ Compute z ❸Compute p ❹Compute rz ❺Compute rr ❻Compute ap ❼Dot Product Alpha ❽Update x ❾Update r ❿ Update z ⓫Dot Product rz ⓬Update p ⓭ Update rz ⓮Update rr

Figure 3: The ReCG Architecture.

3 RECG ACCELERATOR

ReCG is a ReRAM-based PIM accelerator for sparse CG. ReCG uses the CAM and in-situ MAC capabilities of the crossbar array structures to efficiently map sparse CG. It supports parallel implementations of irregular operations, such as Reduction, as well as sparse representation and implementation of the SpMV operation.

In this section, we will discuss the design specifics of ReCG in detail. Firstly, we will introduce the proposed architecture, which mainly includes a sparse functional unit (SPU), a scalar functional unit (SFU), and a vector functional unit (VFU), and introduce how to implement the Reduction operation in parallel. Secondly, we will elaborate on the proposed sparse representation and the four-phase workflow to implement the SpMV operation in the SPU. Finally, we will describe in detail how to coordinate the dataflow across the architecture, reducing the data movement of the entire algorithm to nearly half of the original, so as to improve performance.

3.1 Architecture

As shown in Algorithm 1, the whole CG algorithm is complex, involving a variety of operations composed of scalar, vector, matrix and sparse type operators, and there are more than 10 kinds of operations in total. A straightforward implementation of CG on ReRAM-based hardware needs to implement all the operations. In other words, a hardware unit is specifically designed for each operation. However, this approach requires building more than 10 different hardware units and designing separate components within each unit for different types of operators. For problems with a larger matrix size, this architecture can become complex and bulky, resulting in significant hardware costs. In view of this, we notice the commonality of the operations in the whole CG algorithm, and classify the operations into three types: (1) sparse computation, that is, SpMV; (2) scalar computation; (3) vector computation. This helps to reduce the complexity of the whole algorithm. In this work, we design a dedicated architecture based on these three simplified types of operations to minimize unnecessary overhead.

The architecture comprises five primary components: (1) SFU, (2) VFU, (3) SPU, (4) Central Controller, and (5) Global Buffer. Fig. 3 provides an overview of the architecture and the organization of several components. The Central Controller is responsible for loading data into units and buffers, and generating the necessary control logic to execute the CG algorithm. The Global Buffer is used to provide the matrix $\mathbf{A}$, $\mathbf{M}^{-1}$, the vector $\mathbf{b}$, the initial solution vector $\mathbf{x}_0$, the convergence threshold τ, the maximum number of iterations N, and the intermediate solution vector $\mathbf{x}$ computed in each iteration. At the same time, it also stores the data that needs to be transferred between the computing units.

The ***SFU*** is mainly responsible for the scalar computations in the CG algorithm, i.e., it computes the result as a scalar. Six of the fourteen steps in the CG algorithm belong to this computation. These steps include initializing rz and rr, which are respectively in lines 4 and 5 of Algorithm 1, and need to be completed before entering the iterative loop. There are four more steps in the iterative loop to compute α, rz_{new}, rz, and rr, which correspond to lines 8, 12, 14, and 15 of Algorithm 1, respectively. Initially, we consider using MAC crossbars and dividers to implement these steps, but we found that besides simple additions, multiplications, and divisions, there is also the Reduction operation. If only MAC crossbars and dividers are used, it will lead to a significant amount of unnecessary hardware overhead, while the computation process will become serial and slow. In order to reduce the hardware cost and to parallelize the whole computational process, we use a combination of MAC crossbars, tree adders and dividers. In this design, MAC crossbars are mainly responsible for the multiplications and partial accumulations in these steps, while tree adders handle the Reduction operation on the output of MAC crossbars, and the dividers deal with division computations. The use of a tree structure reduces the number of adders and improves the parallelism.

The ***VFU*** is responsible for handling vector computations, which is the process of generating vectors as results. In the CG algorithm, there are six steps that belong to this type of computation. These

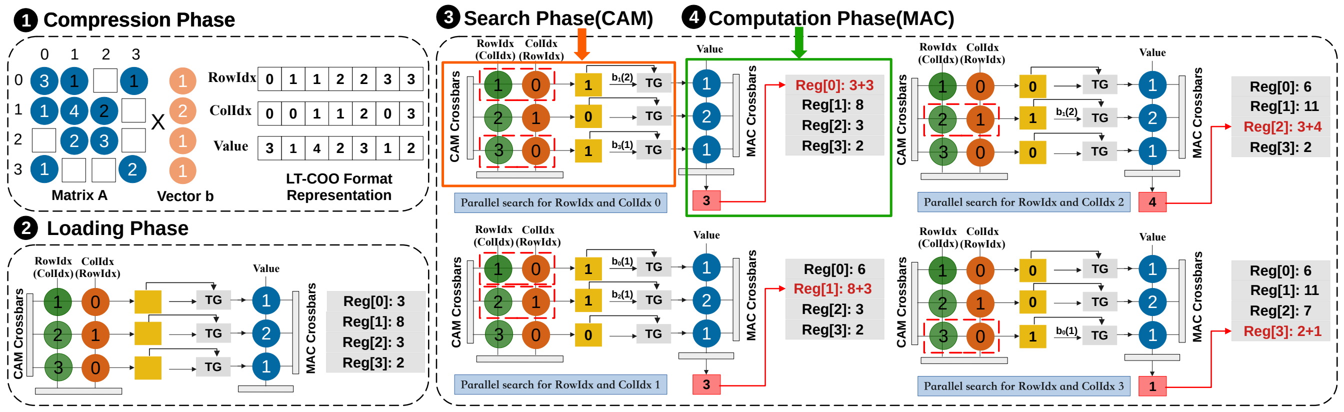


Figure 4: The SpMV Process. There is a 4-phase workflow that works together to complete the SpMV process: Compression Phase, Loading Phase, Search Phase and Computation Phase.

steps include initializing the vectors **z** and **p**, which are in lines 2 and 3 of Algorithm 1, respectively. In addition, it involves updating vectors **x**, **r**, **z**, and **p** in the iterative loop, in lines 9, 10, 11, and 13 of Algorithm 1, respectively. By analyzing these steps, we find that they mainly involve addition, multiplication and division operations between vectors and scalars. In order to handle these operations, we design the VFU, which consists of adders, multipliers and dividers.

The ***SPU*** is used to handle the SpMV operation, which involve initializing **r** and **ap** in lines 1 and 7 of Algorithm 1, respectively, which is the basis of subsequent computations. SpMV plays a key role in accelerating the CG process, which is the core step of the algorithm. Therefore, we specially design the SPU to realize the SpMV. The SPU includes multiple components, such as MAC crossbars, CAM crossbars, registers, etc., which work together to complete the SpMV. The details will be described in the next subsection.

3.2 SpMV

We use the example in Fig. 4 to describe the workflow of implementing SpMV with the SPU. We divide the implementation of SpMV into a four-phase workflow: *compression phase*, *loading phase*, *search phase*, and *computation phase* [14]. The four phases are executed sequentially, but multiple sets of data can be executed in parallel in the same phase.

The initial idea is to use the COO format to store sparse matrices and perform the SpMV operation. However, we realize that the sparse matrices in linear systems solved by CG have symmetric positive definite properties. Therefore, we make full use of this feature to compress the data in COO format and propose a new matrix storage method called LT-COO format. Specifically, we store the lower triangular part of the sparse matrix in COO format according to the row-major order. In the ***compression phase***, we store the sparse matrix in the LT-COO format to reduce the cost of data storage and transmission.

After storing the sparse matrix in the LT-COO format, we write the data stored in the LT-COO format onto the MAC and CAM crossbar arrays. Specifically, as shown in Fig. 4, in the ***loading phase***, in order to reduce the write cost, we only write the indexes of RowIndex and ColIndex in the LT-COO format except diagonal elements to the CAM crossbar arrays, and write their corresponding values to the MAC crossbar arrays. And the corresponding diagonal elements are multiplied with the corresponding elements of vector b and written to the corresponding positions in the register. Next, in the ***search phase***, the controller sends the indexes to the CAM crossbar arrays. Since the matrix is symmetric, the row indexes and column indexes are searched in parallel in the CAM crossbar arrays. Finally, in the ***computation phase***, based on the search results, data 1 or 0 for controlling the transmission gate is generated, and the corresponding vector **b** data is input into the transmission gate. If the data generated by the CAM crossbar arrays is 1, the corresponding vector **b** data is passed through the transmission gate and computed with the data on the MAC crossbar arrays; If the data is 0, the vector **b** cannot be passed. Then, the output result of the MAC crossbar arrays is added with the data at the corresponding position of the register to obtain the final result.

3.3 Scheduling Strategy

We analyze in detail the data dependencies of the scalars, vectors and matrices in each step of the algorithm, and formulate a new dataflow scheduling strategy, which divides the fourteen steps in the whole algorithm into eight phases, as shown in Fig. 5. The strategy realizes the parallel execution within the phases, and enables the units in the whole architecture to have the capability of parallel execution, thus improving the parallelism and accelerating the execution speed of the algorithm. At the same time, the number of data handling and writes are greatly reduced with this strategy.

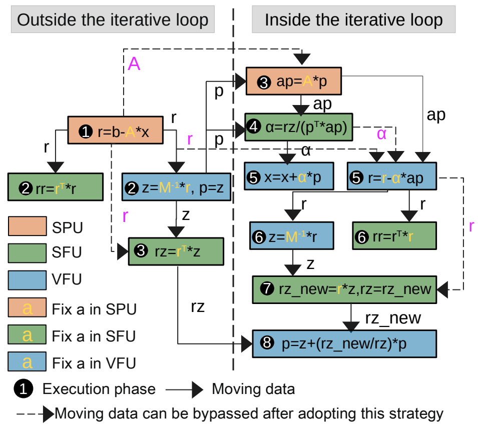


Figure 5: Scheduling strategy.

Outside the iterative loop, we perform dataflow scheduling for the original five computational steps, dividing them into three phases. In phase two, SFU and VFU can be executed in parallel, and steps 2 and 3 can be realized in the VFU at the same time. In addition, we fix the sparse matrix **A** to the SPU, **r** to the SFU and VFU, and $\mathbf{M}^{-1}$ to the VFU. As can be seen from the Fig. 5, the proposed scheduling strategy not only improves the parallelism, but also can only transport **r** and **z** outside the iterative loop, and effectively reduces the amount of transport of **A** and **r**. ***Inside the iterative loop***, we reschedule the original nine computational steps, dividing them into six phases based on operator dependencies. While the sparse matrix computation is performed in the SPU inside the iterative loop, the operations outside the iterative loop are performed in parallel in the SFU. In phases five and seven, the computational steps are executed in parallel in the VFU and SFU, respectively. And in phase six, the operations are performed in parallel in the VFU and SFU to further improve the computational efficiency. In addition, we fix α on VFU and **r** on SFU to avoid frequent data movement and write operations, which effectively reduces the number of data movements and writes for α and **r**.

4 EVALUATION

In this section, we evaluate the ReCG design. We first introduce the experimental setup, and then compare the time and energy consumption of accelerating CG on CPU, GPU and FPGA. Finally, we verify the effectiveness of the scheduling strategy.

4.1 Experimental Setup

Benchmark. We evaluate 36 real-world sparse matrices from the SuiteSparse Matrix Collection [15], which come from different fields such as computational fluid dynamics problem, power network problem, structural problems, etc. Table 1 provides the name, row/column number and number of non-zeros (NNZ) for each matrix. The row/column numbers of the matrices span from 0.003M to 1.565M, and the NNZ ranges from 0.099M to 114.165M.

Table 1: Benchmark information.

Matrix	#Row	#NNZ	Matrix	#Row	#NNZ
ex9	0.003M	0.099M	bcsstk15	0.004M	0.118M
bodyy4	0.018M	0.122M	ted_B	0.011M	0.145M
ted_B_unscaled	0.011M	0.145M	bcsstk24	0.004M	0.160M
nasa2910	0.003M	0.174M	s3rmt3m3	0.005M	0.207M
bcsstk28	0.004M	0.219M	s2rmq4m1	0.005M	0.263M
cbuckle	0.014M	0.677M	olafu	0.016M	1.015M
gyro_k	0.017M	1.021M	bcsstk36	0.023M	1.143M
msc10848	0.011M	1.230M	raefsky4	0.020M	1.317M
nd3k	0.009M	3.280M	nd6k	0.018M	6.897M
2cubes_sphere	0.101M	1.647M	cfd2	0.123M	3.085M
Dubcova3	0.147M	3.637M	ship_003	0.122M	3.777M
offshore	0.260M	4.243M	shipsec5	0.180M	4.599M
ecology2	1.000M	4.996M	tmt_sym	0.727M	5.081M
boneS01	0.127M	5.517M	hood	0.221M	9.895M
bmwcra_1	0.149M	10.642M	af_shell3	0.505M	17.562M
Fault_639	0.639M	27.246M	Emilia_923	0.923M	40.374M
Geo_1438	1.438M	60.236M	Serena	1.391M	64.132M
audikw_1	0.944M	77.652M	Flan_1565	1.565M	114.165M

ReCG Configurations. There is an SPU unit, an SFU unit, and a VFU unit. The SPU unit consists of a controller, MAC crossbars, CAM crossbars, buffers, etc. The SFU unit consists of a controller, MAC crossbars, a Reduction unit, buffers, etc. And the VFU unit consists of adders, multipliers and dividers. The size of each MAC crossbar and CAM crossbar is 128×128, and MAC crossbars are connected to peripheral circuits such as ADC, S&H, etc. Each cell of MAC crossbars and CAM crossbars represents four bits and one bit of data, respectively. In the entire process, we use 64 bits to represent non-zero values and perform 64-bit fixed-point arithmetic.

Methodology. Since there is no published PIM accelerator that can implement JPCG for comparison, we compare CALLIPEPLA[5] on the FPGA, PETSc [4] on the CPU and GPU, which are the most widely used JPCG on the corresponding platforms. [5] is a prototype JPCG accelerator on a Xilinx Alveo U280 FPGA. We run open-source PETSc on an AMD 2nd EPYC 7702 CPU and a NVIDIA Tesla A100 GPU. We use NVSim [16] to obtain the ReRAM write energy and delay. And we utilize NeuroSim [17] to obtain other data. NeuroSim can simulate architecture- and application-level read operations (similar to "inference" of neural networks) to get the energy consumption and execution delay of an entire algorithm. The read energy consumption and latency of the CAM crossbars are 1.08pj and 29.31ns, respectively [13, 18].

4.2 Performance Results

We compare the solving time of the 36 evaluated matrices on the four accelerators/platforms (CPU, GPU, FPGA and ReRAM). Fig. 6 shows the comparison of solving time. For the first 11 smaller-sized matrices (i.e., from matrix ex9 to matrix cbuckle), ReCG is on average an order of magnitude faster compared with PETSc on CPU, and on average twice faster than GPU, but consumes more time than CALLIPEPLA on FPGA.

For the last 25 larger-scale matrices (i.e., from matrix olafu to matrix Flan_1565), we note that ReCG achieves better acceleration compared with the solving time of 11 smaller-scale matrices. Compared with CPU, GPU and FPGA, ReCG can achieve the highest acceleration levels of three orders of magnitude, one order of magnitude and one order of magnitude, respectively. In addition, as shown in Fig. 6, the acceleration of ReCG is more significant as the matrix size increases, showing its good scalability. Compared with the other three platforms, ReCG can more effectively support larger-scale problems.

4.3 Energy Results

Fig. 7 shows the energy consumption of 36 matrices from SuiteSparse on different platforms (CPU, GPU, FPGA and ReRAM). Since ReCG accelerates JPCG on ReRAM by using the PIM technique, and adopts the corresponding dataflow execution strategy, its energy consumption is minimized as much as possible. For sparse matrices of different sizes, ReCG has the lowest energy consumption compared with accelerating JPCG on CPU, GPU and FPGA, with the highest reduction of two orders of magnitude, two orders of magnitude, and one order of magnitude, respectively.

4.4 Scheduling Strategy Evaluation

To minimize the impact of poor write endurance of ReRAM on accelerating JPCG, we analyze the frequency of operator usage and data dependencies in the entire JPCG. we formulate a new dataflow scheduling strategy, which is described in 3.3. With the scheduling strategy, we significantly reduce the write time on ReRAM. As shown in Fig. 8, we demonstrate the percentage of write time for the first 18 matrices. After adopting the proposed scheduling

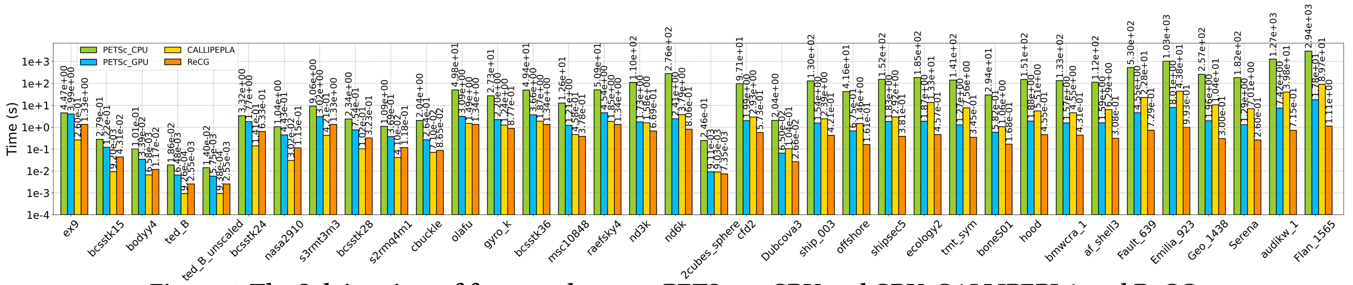


Figure 6: The Solving time of four accelerators: PETSc on CPU and GPU, CALLIPEPLA and ReCG.

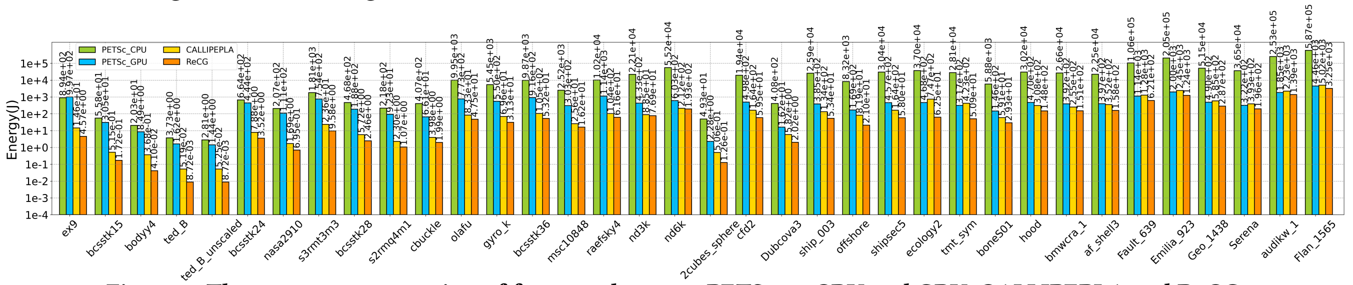


Figure 7: The energy consumption of four accelerators: PETSc on CPU and GPU, CALLIPEPLA and ReCG.

strategy, the write time is reduced by about 50%, which verifies the effectiveness of the scheduling strategy.

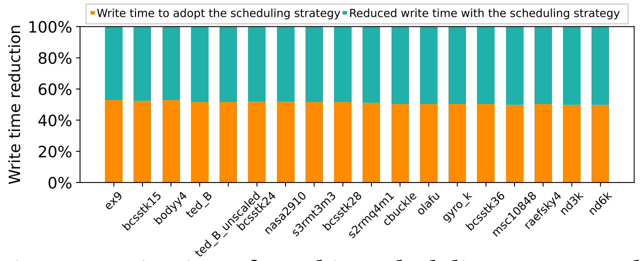


Figure 8: Write time after taking scheduling strategy and reduced write time.

5 CONCLUSION

The feature of in-situ matrix-vector multiplication supported by ReRAM crossbars opens a new direction for accelerating numerical computing applications by PIM hardware. However, when regular crossbars meet irregular sparse matrices, key challenges such as workload mapping and dataflow scheduling must be addressed to efficiently run irregular matrix operations on regular ReRAM crossbars. In this work, we propose ReCG, a ReRAM-based architecture that can efficiently accelerate JPCG. For JPCG, we design multiple units to achieve various kernels of JPCG. We also propose a new dataflow execution strategy to reduce data handling. The experimental results show that the performance of ReCG is improved by up to three, one and one order of magnitude compared with PETSc on CPU, GPU and CALLIPEPLA on FPGA, respectively, and the energy consumption is reduced by up to two, two and one order of magnitude.

ACKNOWLEDGMENT

This work was supported by NSFC (Grant No. U23A20301, 62204265, 62234010, 62122076), the State Key Laboratory of Computer Architecture (ICT, CAS) (No. CARCHA202115), Strategic Priority Research Program of CAS (No. XDB44000000), Key Research Program of Frontier Sciences, CAS (No. ZDBS-LY-JSC012), Innovation Funding of ICT, CAS (No. E261040). Zhou Jin and Weifeng Liu are the corresponding authors.

REFERENCES

[1] C. D. Meyer and I. Stewart. *Matrix analysis and applied linear algebra*. SIAM, 2023.
[2] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 1952.
[3] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.
[4] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In *Modern Software Tools in Scientific Computing*, 1997.
[5] L. Song, L. Guo, S. Basalama, Y. Chi, R. F. Lucas, and J. Cong. Callipepla: Stream centric instruction set and mixed precision for accelerating conjugate gradient solver. In *FPGA*, 2023.
[6] M. Fan, X. Tian, Y. He, J. Li, Y. Duan, X. Hu, Y. Wang, Z. Jin, and W. Liu. Amgr: Algebraic multigrid accelerated on reram. In *DAC*, 2023.
[7] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai. Metal–oxide rram. *Proceedings of the IEEE*, 2012.
[8] P. Chi, S. Li, C. Xu, 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 *ISCA*, 2016.
[9] L. Song, X. Qian, H. Li, and Y. Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In *HPCA*, 2017.
[10] X. Qiao, X. Cao, H. Yang, L. Song, and H. Li. Atomlayer: A universal reram-based cnn accelerator with atomic layer computation. In *DAC*, 2018.
[11] M. Prezioso, F. Merrikh-Bayat, B. D. Hoskins, G. C. Adam, K. K. Likharev, and D. B. Strukov. Training and operation of an integrated neuromorphic network based on metal-oxide memristors. *Nature*, 2015.
[12] N. Jao, A. K. Ramanathan, A. Sengupta, J. Sampson, and V. Narayanan. Programmable non-volatile memory design featuring reconfigurable in-memory operations. In *ISCAS*, 2019.
[13] N. Challapalle, S. Rampalli, L. Song, N. Chandramoorthy, K. Swaminathan, J. Sampson, Y. Chen, and V. Narayanan. Gaas-x: Graph analytics accelerator supporting sparse data representation using crossbar architectures. In *ISCA*, 2020.
[14] X. Zhang, Z. Li, R. Liu, X. Chen, and Y. Han. Fspa: An fefet-based sparse matrix-dense vector multiplication accelerator. In *DAC*, 2023.
[15] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *TOMS*, 2011.
[16] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi. Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *TCAD*, 2012.
[17] P.-Y. Chen, X. Peng, and S. Yu. Neurosim: A circuit-level macro model for benchmarking neuro-inspired architectures in online learning. *TCAD*, 2018.
[18] F. Liu, W. Zhao, Y. Chen, Z. Wang, Z. He, R. Yang, Q. Tang, T. Yang, C. Zhuo, and L. Jiang. Pim-dh: Reram-based processing-in-memory architecture for deep hashing acceleration. In *DAC*, 2022.