AmgR: Algebraic Multigrid Accelerated on ReRAM

Mingjia Fan[1], Xiaotian Tian[1], Yintao He[2,3], Junxian Li[1], Yiru Duan[1],
Xiaozhe Hu[5], Ying Wang[2,3,4], Zhou Jin[1] and Weifeng Liu[1]
1. Super Scientific Software Laboratory, China University of Petroleum-Beijing, 2. Institute of Computing Technology, Chinese Academy of Sciences, 3. SKLP, Institute of Computing Technology, CAS, 4. Research Center for Intelligent Computing Systems, 5. Tufts University, USA
Email: {mingjia.fan,xiaotian.tian,junxian.li,yiru.duan}@student.cup.edu.cn,
{heyintao19z,wangying2009}@ict.ac.cn, Xiaozhe.Hu@tufts.edu, {jinzhou,weifeng.liu}@cup.edu.cn

***Abstract*—Solving systems of linear equations is a fundamental problem in scientific computing, which has been extensively researched for decades. One of the most well-known solvers is Algebraic Multigrid (AMG), which is widely used in high performance computing due to its good scalability. But currently accelerating AMG relies on the traditional von Neumann architecture of storage and computation separation, which leads to a large data transmission overhead. In this work, we propose a ReRAM-based processing-in-memory (PIM) architecture named AmgR, which overcomes the limitations of the traditional von Neumann architecture for AMG acceleration.**

However, accelerating AMG on ReRAM is non-trivial, because (1) AMG has many computing kernels of various types; (2) there are irregular operations that cannot be directly performed using matrix-vector multiplication suitable for ReRAM, i.e., aggregation operation; (3) ReRAM has poor write endurance, and a lot of data during AMG acceleration needs to be rewritten into ReRAM, resulting in high write cost. To address these issues, firstly, we propose a flexible architecture, which can realize each kernel of AMG and is reused by many kernels to improve resource utilization. Secondly, we propose a dedicated unit to realize the aggregation operation. Finally, we present a new mapping strategy to greatly reduce the number of data handling and writes. The experimental results show that the performance of AmgR is improved by an average of one and two orders of magnitude compared to HYPRE on the CPU and AmgX on the GPU, respectively, while the energy consumption is reduced by an average of two and three orders of magnitude.

***Index Terms*—AMG, Processing-in-memory, ReRAM, Linear algebra**

I. Introduction

Solving systems of linear equations is a core component of many practical applications and one of the most common problems in science and engineering, such as quantum mechanics, statistical analysis and machine learning [1]. Therefore, improving the efficiency of solving linear equations has become one of the key concerns, especially how to improve the time and energy efficiency. But the efficiency of solving linear equations is limited by computational complexity. The Algebraic Multigrid (AMG) can solve linear equations efficiently due to its better computational complexity and high scalability [2]. Thus, it has been widely used in numerical simulation of practical problems in science and engineering.

Researchers have put a lot of effort into parallelizing and optimizing AMG on CPU clusters [3] and using GPUs [4] to accelerate it, achieving good performance improvements. However, due to the limitations of CPU and GPU of the traditional von Neumann architecture, the overhead of data transmission between processor and memory is quite large, which has become a bottleneck to accelerate AMG. In contrast, processing-in-memory (PIM) [5] breaks through architectural limitations to quickly read and access data. Resistive random access memory (ReRAM), as an emerging non-volatile memory technology, is considered as a promising candidate for PIM accelerators. Specifically, it can store data and perform in-situ matrix-vector multiplication (MVM) in the analog domain. Thus, the PIM architecture based on ReRAM provides a new opportunity for accelerating AMG.

However, accelerating AMG on ReRAM poses certain challenges. Firstly, AMG is very complex and has many computing kernels. It is divided into *Setup* phase (use four kernels to construction of the grid hierarchy and the two important components of AMG, interpolation and restriction operators) and *Solve* phase (use the grid hierarchy and components generated in the *Setup* phase to perform multiple grid loop iterations to solve linear equations, with six kernels in one iteration). If each kernel is implemented by a single hardware unit, it is difficult to improve hardware utilization. In addition, AMG has many irregular operations that cannot be directly executed using MVM suitable for ReRAM, i.e., aggregation operation. Finally, ReRAM has poor write endurance, and AMG has many data movements and complex data dependencies, which makes it hard to reduce the number of data handling and writes. Therefore, there are still some difficulties in implementing AMG on ReRAM.

To address challenges mentioned above, firstly, we propose a flexible architecture AmgR containing multiple units, which be reused by many kernels to improve resource utilization. Secondly, we use analog content-addressable memory (CAM) to design AGG unit for aggregation operation. Finally, based on the frequency of operators uesd in the flow, we present a new mapping strategy to reduce the number of data handling and writes. And we compress operators to facilitate storage. Specifically, our work has the following contributions:

This work was supported by National Key R&D Program of China (Grant No. 2021YFB0300600), the State Key Laboratory of Computer Architecture (ICT,CAS) (Grant No. CARCH CARCHA202115), and the Key Program of the National Natural Science Foundation of China (Grant No.62204265, 62234010, 61972415, 62222411).

979-8-3503-2348-1/23/$31.00 © 2023 IEEE

(1) To the authors' knowledge, this is the first ReRAM-based PIM AMG accelerator, AmgR.

(2) We propose a flexible architecture, which can realize each kernel of AMG and reused by many kernels.

(3) We design a dedicated unit to perform the aggregation operation that is difficult to realize with ReRAM.

(4) We present a new mapping strategy to reduce the effect of poor write endurance of ReRAM on AMG acceleration.

(5) AmgR achieves an average performance improvement of one and two orders of magnitude compared to HYPRE and AmgX, respectively, and energy consumption is reduced by two and three orders of magnitude.

II. BACKGROUND

A. Algebraic Multigrid

Consider the linear equation $Ax = b$, where A is a matrix with elements a_{ij} ($A \in R^{n \times n}$), x is an unknown vector to be calculated ($x \in R^n$), and b is the right hand side of the equation ($b \in R^n$). Usually the set of unknowns $\omega = \{x1, x2, ...\}$ is directly recorded as a grid. There are *Setup* phase and *Solve* phase to solve the linear equation with AMG, which have ten kernels.

Algorithm 1 *Setup* Phase

Input: $A_1 \leftarrow A$, I(identity matrix), D^{-1}(D is diagonal matrix of A)
Output: A_{l+1}, P_l, R_l, S_l ($l = 1, 2, ...$)
for $l = 1, 2, ...$
1: $T_l \leftarrow Aggregation(A_l)$ {Aggregation Operation}
2: $S_l \leftarrow I - D^{-1} * A_l$ {Smoother}
3: $P_l \leftarrow S_l * T_l$ {Interpolation/Restriction Operator}
4: $R_l \leftarrow P_l^T$
5: $A_{l+1} \leftarrow R_l * A_l * P_l$ {Triple Matrix Multiplication}

The *Setup* phase focuses on the construction of the grid hierarchy and two important components of the AMG, interpolation operator P and restriction operator R. Algorithm 1 shows the detailed process. First, the aggregation and Jacobi operations are performed based on matrix A to construct the tentative prolongator T and smoother S required for P, respectively (line 1-2). The aggregation operation involves picking the top k maximums for each column of matrix A (corresponding to a point on the adjacency graph) in turn, and aggregating the k points with the largest weights on the edges connected to that point to form a coarse point. Then, P is calculated using S and T (line 3), and R is constructed from the transpose of P (line 4). Finally, the coarse grid matrix is calculated using R, A and P (line 5).

Algorithm 2 *Solve* Phase

Input: $A_1 \leftarrow A, b, x_0, S_l, I, D^{-1}$
Output: x
for $l = 1, 2, ...$
1: $x_l \leftarrow S_l * x_0 + D^{-1} * b$ {Pre-smoothing}
2: $r_l \leftarrow b - A_l x_l$ {Residual Calculation}
3: $r_{l+1} \leftarrow R_l r_l$ {Restriction}
4: $e_{l+1} \leftarrow A_{l+1}^{-1} * r_{l+1}$ {Solve}
5: $x_l \leftarrow x_l + P_l e_{l+1}$ {Interpolation}
6: $x_l' \leftarrow S_l * x_l + D^{-1} * b$ {Post-smoothing}

The *Solve* phase uses the grid hierarchy, P and R constructed in the *Setup* phase to solve the linear equations. As shown in Algorithm 2, the process begins with pre-smoothing to obtain a more accurate initial solution (line 1), followed by the calculation of residual r (line 2). Then, the residual is restricted to the coarse grid using R (line 3), and the coarse grid equation is solved (line 4). Next, the solution on the coarse grid is interpolated back to the fine grid using P to obtain an updated solution (line 5). Finally, post-smoothing is performed to obtain a more accurate solution (line 6).

B. ReRAM-based PIM Designs

ReRAM is an emerging non-volatile memory with advantages such as high density, fast read access and low leakage power [6]. The crossbar arrays composed of ReRAM enable to perform Multiply-Accumulate (MAC) operations and support search operations through analog CAM.

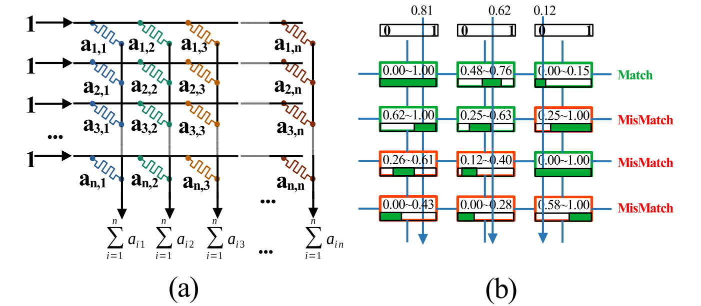

Fig. 1: (a) MAC Crossbar. (b) Analog CAM [7].

ReRAM-based MAC Crossbar. Fig. 1(a) shows the crossbar array structure of the ReRAM, which has the ability to perform in-situ MVM operations. Connecting ReRAM cells to the same row is called the wordline and to the same column is called the bitline. Voltage is input from the wordline and current is output from the bitline. The values of the matrix are programmed to the conductance $G_{i,j}$ (i and j denote the number of rows and columns, respectively) in each ReRAM cell, and the vector is input as a voltage. ReRAM cells connected to the same wordline share the same input voltage V_i. According to Ohm's Law and Kirchhoff's Current Law, the current flowing through each ReRAM cell can be obtained as $I_j = \sum_i V_i * G_{i,j}$, and the sum of the currents output by the ReRAM cells connected to the same bitline is a result of MVM.

ReRAM-based analog CAM Crossbar. The analog CAM supports search in a continuous analog interval [7]. The analog CAM crossbar is shown in Fig. 1(b), the analog CAM searches and stores analog data, where the input data can be continuous values, the stored data is a continuous interval, and the lower and upper bounds represent the acceptable range of matching. If the input data is within this matching range, the match is successful; otherwise, the match fails.

III. AMGR FRAMEWORK

In this section, we discuss the design details of the AmgR. Firstly, we introduce the proposed architecture, which includes reusable computing unit, aggregation unit and operational amplifiers (OAs) unit. Secondly, we elaborate a new mapping strategy to greatly reduce the impact of poor write endurance

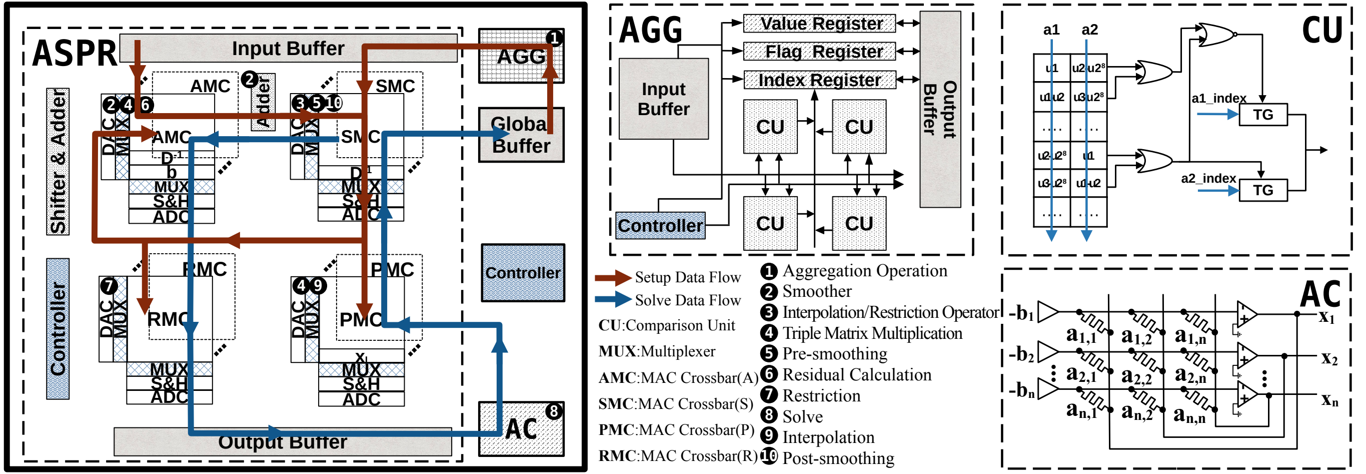

Fig. 2: The AmgR Architecture.

of ReRAM on accelerating AMG. Finally, we present the data flow in AMG based on the architecture and mapping strategy.

A. Architecture

One of the major challenges of implementing AMG on ReRAM is the ten kernels that need to be executed. One solution is to design a hardware unit specifically for each kernel and connect the intermediate buffers in a pipeline manner. In this way, we need to build ten hardware units, and the hardware overhead is large when the matrix size is slightly larger. But we find that computational patterns of many kernels are similar. In this work, we take full advantage of the similarity of computing patterns to design a flexible architecture that can be reused by many kernels in AMG, thereby reusing hardware and reducing unnecessary overhead.

In addition, we can observe that there are many operations in AMG, which can be categorized into three types: 1) aggregation operation, 2) multiplication of the inverse matrix by a vector, 3) multiplication, addition and subtraction between vectors and matrices. We can find that most of them involve various operations related to matrices, but the grid coarsening through aggregation operation does not involve any matrix operations. In order to realize this kind of irregular operation, we propose a dedicated unit.

The overall architecture is shown in Fig. 2, which includes five components: AGG unit, ASPR unit, AC unit, Controller and Global Buffer. The Global Buffer is used to provide the matrix A, D^{-1}, the RHS vector b and to accept the result x calculated each time during the iteration, and the Controller is used to control the whole data flow. Next, we will introduce the three units in the architecture in detail.

1) AGG unit: This unit realizes the aggregation operation in AMG. According to the weights of the undirected graph edges shown in Fig. 3(a), the points connected by the top k edges with larger weights are aggregated into a coarse point each time, and eventually all the points in the graph are aggregated into coarse points, as shown in Fig. 3(c). The core idea of AMG is to solve linear equations in coarse and fine grid layers, which are used to eliminate low-frequency and high-frequency errors, respectively. The aggregation operation is to aggregate fine points into coarse points, so that the coarse grid layer can be constructed according to these aggregated coarse points. It is clear that this operation is very important for AMG, but its implementation on ReRAM is a huge challenge, which is difficult to achieve with MAC crossbar. To achieve this irregular operation, we effectively utilize CAM to propose a dedicated unit, thus constructing the T based on the aggregation operation. This unit consists of four comparison units (CUs), Input Buffer, Output Buffer and three sets of Registers: Index, Value and Flag. The lengths of three Registers are k, k and n, respectively (k is the aggregation size, and n is the matrix order). The Index and Value Registers are used to save the index and value of the aggregated points, respectively. And the Flag Register is used to flag whether a point has been aggregated or not to prevent repeated aggregation.

The specific process is as follows. We need to operate on the adjacency matrix depicted in Fig. 3(b) of the undirected graph shown in Fig. 3(a). For each aggregation, the Index and Value Registers are first initialized to 0. Then, we judge the points that have not been aggregated in turn, selecting the k points with the largest undirected edge weight. These selected points be marked in the Flag Register, and their corresponding indices and values are stored in the Index and Value Registers, respectively. As shown in Fig. 3(b) step1, when k is 3, points 1, 2 and 3 after the first aggregation constitute the first coarse point. And rows 1, 2 and 3 of the first column of the T are set to 1, the remaining rows are set to 0. Next, the above-mentioned aggregation operation is continued on the points that are not aggregated, as shown in Fig. 3(b) step2, that is, points 4, 5 and 8 collectively form the second coarse point. And the entries corresponding to rows 4, 5, and 8 in the second column of the T are set to 1, while the entries in all other rows are set to 0. Finally, points 6 and 7 become the last aggregation, that is, the last coarse point, as shown in Fig. 3(b) step3. In this way, all points have been aggregated, as shown in Fig. 3(c), and the aggregation operation ends. At the same time, the last column of the T is then similarly constructed.

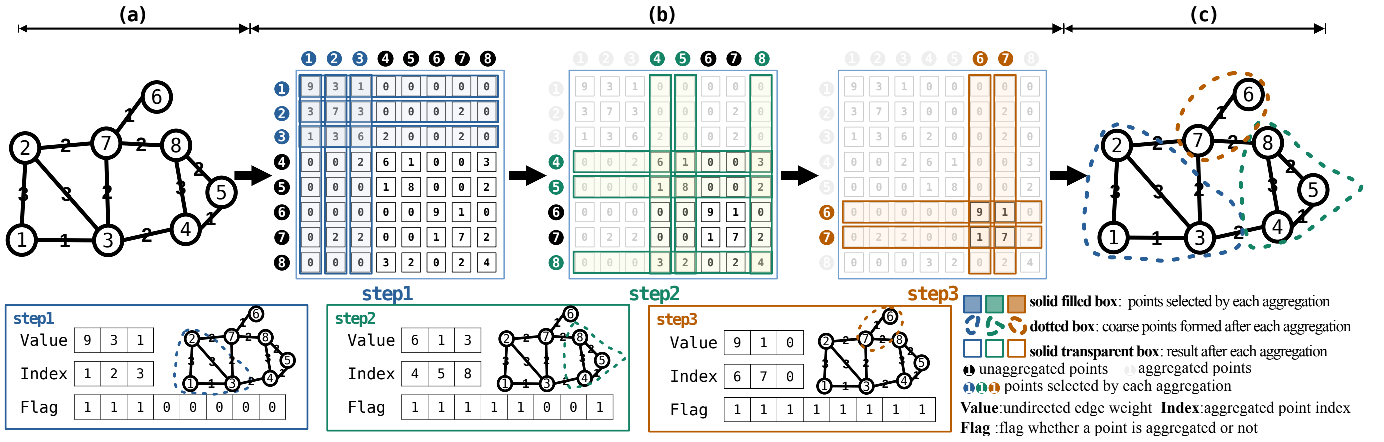

Fig. 3: Aggregation: Aggregate undirected graph in (a). The adjacency matrix (b) is operated by selecting k points with the largest edge weight to aggregate into one coarse point at a time, until all points in the undirected graph are aggregated (c).

2) ASPR unit: This unit includes four MAC Crossbars, Input Buffer, Output Buffer and Controller. In AMG, the functions of constructing smoother S, interpolation operator P, restriction operator R, coarse grid matrix Ac, pre-smoothing, calculating residual and restricting to coarse grid, interpolating and correcting approximate solution of fine grid, and post-smoothing can all be implemented by this unit. However, performing these operations usually requires a large number of data handling. Therefore, in order to reduce the number of data handling and writes, we need to design a mapping strategy that reuses hardware and data, and the details will be mentioned in the following section.

3) AC unit: This unit consists of resistive crossbar and OAs. In AMG, the fine grid layer can only be used to eliminate high-frequency errors, but the operation of solving coarse grid equations implemented by the AC unit can be used to eliminate low-frequency errors. This operation involves solving the inverse of a matrix, which is an expensive operation. Inspired by the work of [8], we leverage the circuit to solve this challenge through the AC unit of Fig. 2. In the cross-point circuit, the input voltages is b, each entry a_{ij} of matrix A is coded as the analog conductance G_{ij} of the resistive memory, and the output voltages of the OAs provide the solution $x = A^{-1}b$.

B. Mapping Methodology

Although we realize the reuse of unit functions with the flexible architecture, the process of the AMG is still extremely complicated. First, the matrix A is used to generate T, S, P, R and Ac in the *Setup* phase to construct the grid hierarchy. In the *Solve* phase, the matrix A, vector b, and these operators S, P, R, Ac generated in the *Setup* phase are used to perform the V-cycle process, including pre-smoothing, calculating the residual, restricting residual to the coarse grid, solving coarse grid equation, interpolating and correcting the approximate solution of the fine grid and post-smoothing. If the whole AMG process is implemented step by step in pipelined form, it will certainly bring huge overhead.

During the construction of the grid hierarchy, the first step is to transport A in order to construct T and S. Then, we transport S and T to construct P and R. Finally, we transport R, A, and P to construct Ac. During the execution of the V-cycle, S, x_0, b and D^{-1} are transported to obtain approximate solution x_1. This is followed by the transport of A, x_1, and b to calculate residual r_1, which is then transported with R to calculate r_2. Then e_2 is calculated by transporting r_2 with Ac. The next step involves transporting x_1, P, and e_2 to update x_1, and finally transporting S, x_1, b, and D^{-1} to construct the final solution x. We can see that a lot of data is handled many times, and executing the V-cycle requires multiple iterations. Therefore, the pipelining entails significant handling costs.

In order to solve this problem, in this work, we propose a new mapping strategy, which effectively exploits the feature that operators are used with different frequencies in the flow. We fix the A, S, P, and R on the four MAC crossbars of ASPR unit, respectively. In order to further reduce overhead, we also compress D^{-1} and fix b, D^{-1} to the MAC Crossbar. In the *Setup* phase, T and S are constructed by transporting A, then A, S, b and D^{-1} are fixed. P and R can be constructed by transporting T, then P and R are fixed. Next, transporting R gets Ac. In the *Solve* phase, in order to get x from x_0, we only need to transport some vectors. Specifically, x_0 and b are transported to calculate x_1, which is then further transported to calculate r_1. Subsequently, r_1 is transported to calculate r_2, which is further transported to calculate e_2. Finally, e_2 and x_1 are transported to update x_1, which is ultimately transported to get x. As shown in Fig. 4, the numbers represent the ten kernels in AMG, and the arrows represent data handling. Obviously, we can see that the number of data handling and writes is greatly reduced by using the mapping strategy.

C. Dataflow

Based on our flexible architecture and mapping strategy, we will describe the dataflow of AMG on the AmgR architecture from the *Setup* phase and *Solve* phase in this section.

(1) *Setup* phase: There are four steps in this phase. Firstly, **tentative prolongator T construction**, matrix A is sent from the Global Buffer to Input Buffer of the AGG unit. The data in each column of matrix A is taken out from Input Buffer

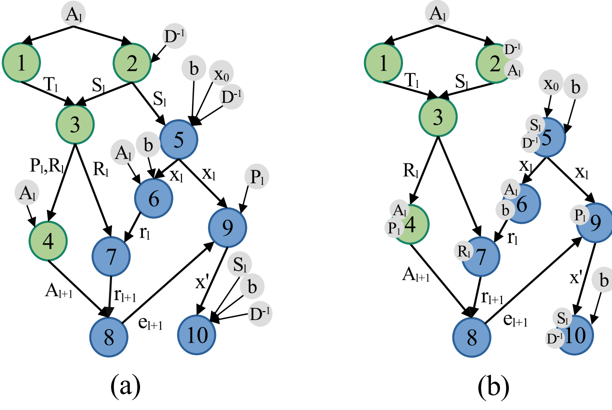

Fig. 4: Mapping. (a) Data handling before using the mapping strategy. (b) Data handling after using the mapping strategy.

in turn, and the top k values are selected by using CU. The top k values and their indices are stored and marked with Registers, and the results of Index Register are stored in the Output Buffer. According to the data in the Output Buffer, the T is constructed. Simultaneously, **smoother** S **construction**, the matrix A and D^{-1} are written from the Global Buffer to the AMC of the ASPR unit, and the diagonal position elements of the intermediate result output from AMC are added by 1 with the Adder, so as to construct S, and it is written to SMC through Buffer. Then, **interpolation and restriction operators construction**, the T is taken from the AGG unit and is input to SMC to construct P and R, which are written to PMC and RMC through Buffer, respectively. Finally, **coarse grid matrix** Ac **construction**, the R is input into AMC through Buffer to calculate $Z = RA$. The intermediate result Z is input into PMC through Buffer to calculate Ac. The Ac is then written to the AC unit through the Global Buffer.

(2) *Solve* phase: There are six steps in this phase. First, **pre-smoothing**, we input x_0 and b from Input Buffer to SMC of ASPR unit to obtain the approximate solution x_1. Then, x_1 is sent to AMC through Buffer and is written to the last row of PMC. The x_1 is multiplied by the matrix A on AMC, and activate the row b, thus obtaining the **residual** r_1. Next, we send r_1 to RMC through Buffer, that is, **the residual is restricted to the coarse grid**. Its result r_2 is then sent to the AC unit through the Global Buffer to **solve the coarse grid equation**. The result e_2 is input to the PMC of ASPR unit through the Global Buffer, and the row x_1 is activated to complete the **interpolation and correct the fine grid approximation solution**, thus updating x_1. Finally, the result x_1 is input into SMC and the D^{-1} row is activated, then the **post-smoothing** is completed and the result x is obtained.

IV. EVALUATION

In this section, we evaluate our AmgR design. We first introduce the experimental setup, and then give a comparison of performance and energy consumption with HYPRE [9] on CPU and AmgX [4] on GPU.

A. Experimental Setup

Benchmark. We select more than 800 matrices from the SuiteSparse Matrix Collection [10] as benchmarks, which come from different fields such as computational fluid dynamics problem, power network problem, etc. As shown in Table 1, we list the information for six benchmarks among them.

TABLE I: BENCHMARK INFORMATION

Matrix	#Rows	#Nonzeros	Field
1138_bus	1,138	4,054	Power Network
adder_dcop_69	1,813	11,246	Circuit Simulation
bayer05	3,268	20,712	Chemical Process Simulation
bcsstk28	4,410	219,024	Structural Problem
ex15	6,867	98,671	Computational Fluid Dynamics
GT01R	7,980	430,909	Computational Fluid Dynamics

AmgR Configurations. There are ASPR unit, AGG unit and AC unit. The ASPR unit consists of Controller, four MAC crossbars, Buffer, etc. Each MAC crossbar is composed of multiple crossbars of 128×128 size, and the crossbars are connected to peripheral circuits such as ADC, S&H, etc. The AGG unit has four CUs, Registers, etc. Each CU consists of CAM array, TG, etc. And the AC unit is composed of multiple crossbars of 128×128 size and OAs.

Methodology. Since there are no PIM accelerators that can implement complex algorithms like AMG for comparison, we compare HYPRE on the CPU and AmgX on the GPU, which are the most widely used AMG software packages on corresponding platforms. We run open-source HYPRE 2.22.1 on an AMD 2nd EPYC 7702, and open-source AmgX 2.2.0 on a NVIDIA Tesla A100. We adopt NeuroSim [11] as the simulator for most of the hardware component models (including ADC, MAC crossbar, DAC, etc.) and NVSim [12] to model the ReRAM writing energy and latency in 32 nm technology node.

B. Performance Results

We use AmgX's performance as the baseline. Fig. 5 shows the performance of more than 800 matrices from SuiteSparse on different platforms (CPU, GPU and ReRAM). We find that AmgR is improved by an average of one and two orders of magnitude compared to HYPRE and AmgX, respectively.

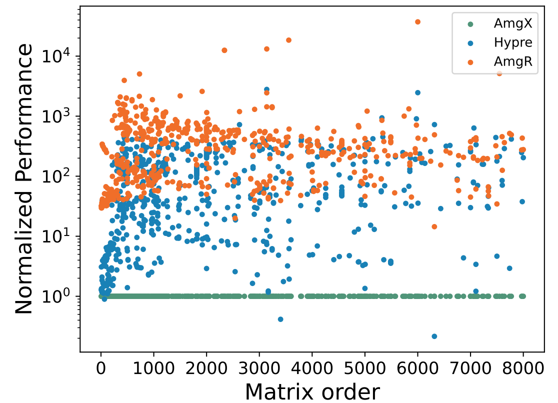

Fig. 5: Normalized performance of AmgR with respect to HYPRE and AmgX.

We select six matrices (as shown in Table 1) from benchmarks to do further performance demonstration. As shown in Fig. 6, due to the different distribution and characteristics of these matrices, there will be some differences in the performance improvement. But overall, AmgR reduces the overhead of memory access time compared to HYPRE and AmgX, and finally achieves certain performance improvement.

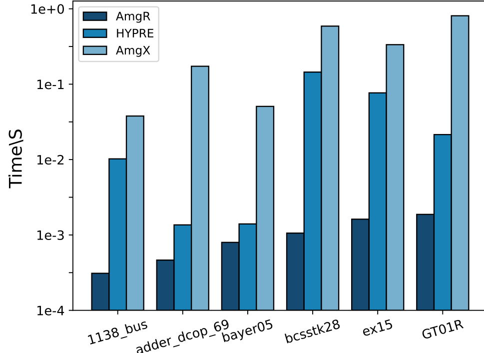

Fig. 6: Time comparison with HYPRE and AmgX.

C. Energy Results

Fig. 7 shows the energy consumption of AmgR compared with HYPRE and AmgX. Our accelerator reduces the energy consumption by two and three orders of magnitude on average compared with the HYPRE and AmgX. In addition, we select six matrices from benchmarks (as shown in Table 1). As shown in Fig. 8, we can know that AmgR has the lowest energy consumption compared with HYPRE and AmgX regardless of the size and characteristics of the matrices.

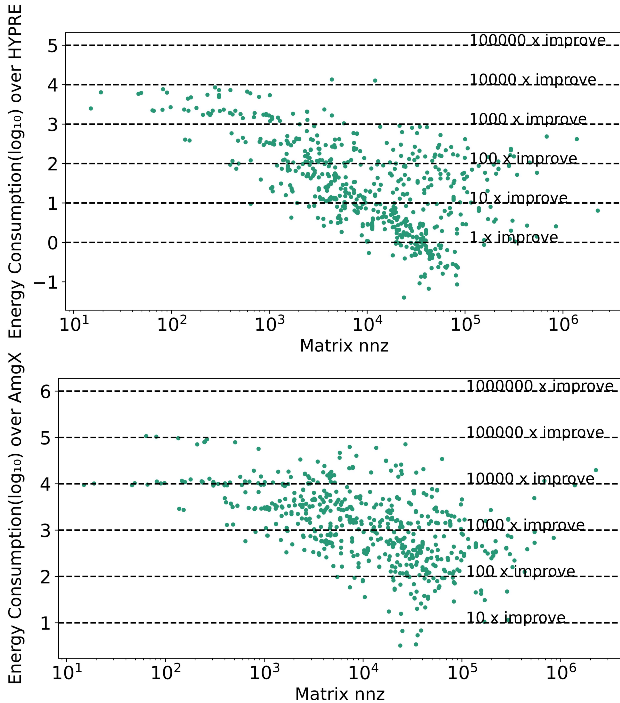

Fig. 7: Energy consumption comparison of AmgR over HYPRE (top) and AmgX (bottom).

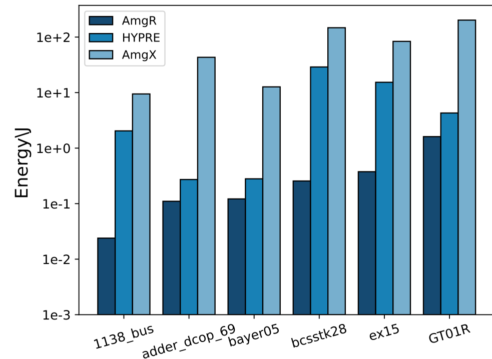

Fig. 8: Energy comparison with HYPRE and AmgX.

V. Conclusion

In this work, we propose AmgR, a flexible ReRAM-based accelerator that can efficiently accelerate AMG. For AmgR, we design multiple units to achieve various kernels of AMG. And we also propose a new mapping strategy to reduce data handling. The experimental results show that the performance of AmgR is improved by an average of one and two orders of magnitude compared to HYPRE on the CPU and AmgX on the GPU, respectively, and the energy consumption is reduced by an average of two and three orders of magnitude.

Acknowledgment

We deeply appreciate the invaluable comments from the reviewers and the helpful discussion with Dr. Guohao Dai. Zhou Jin and Weifeng Liu are the corresponding authors.

References

[1] C. D. Meyer, *Matrix analysis and applied linear algebra*. SIAM, 2000.

[2] A. Brandt, S. MacCormick, and J. Ruge, "Algebraic multigrid (amg) for automatic multigrid solution with application to geodetic computations," *SIAM Journal on Scientific and Statistical Computing*, 1983.

[3] U. M. Yang *et al.*, "Boomeramg: A parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, 2002.

[4] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh *et al.*, "Amgx: A library for gpu accelerated algebraic multigrid and preconditioned iterative methods," *SIAM Journal on Scientific Computing*, 2015.

[5] 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," *ACM SIGARCH Computer Architecture News*, 2016.

[6] T. Chou, W. Tang, J. Botimer, and Z. Zhang, "Cascade: Connecting rrams to extend analog dataflow in an end-to-end in-memory processing paradigm," in *MICRO*, 2019.

[7] C. Li, C. E. Graves, X. Sheng, D. Miller, M. Foltin, G. Pedretti, and J. P. Strachan, "Analog content-addressable memories with memristors," *Nature communications*, 2020.

[8] B. Feinberg, R. Wong, T. P. Xiao, C. H. Bennett, J. N. Rohan, E. G. Boman, M. J. Marinella, S. Agarwal, and E. Ipek, "An analog preconditioner for solving linear systems," in *HPCA*, 2021.

[9] R. D. Falgout and U. M. Yang, "hypre: A library of high performance preconditioners," in *Computational Science — ICCS 2002*, 2002.

[10] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *TOMS*, 2011.

[11] P.-Y. Chen, X. Peng, and S. Yu, "Neurosim: A circuit-level macro model for benchmarking neuro-inspired architectures in online learning," *TCAD*, 2018.

[12] 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.