`

Towards Memristor based Accelerator for Sparse Matrix Vector Multiplication

Jianwei Cui, Qinru Qiu

Dept. of Electrical Engineering & Computer Science

Syracuse University, Syracuse, NY, 13244, USA

{jcui, qiqiu}@syr.edu

Abstract — **In the last few years, memristor crossbar array is drawing increasing attention from the research community as a promising neuromorphic computing accelerator. In this work, we investigate the hardware acceleration of a sparse matrix vector (SpMV) multiplication engine based on memristor crossbar array. We demonstrate that naive matrix coefficient mapping is infeasible and unpractical if the matrix has large dimensions. To combat this problem, we extend the traditional Cuthill-McKee algorithm used for matrix restructuring, and propose a generalized sparse matrix reordering (GSMR) technique, which leverages linear transformation to effectively break down any rectangular unsymmetrical matrices into minimum number of sub-blocks that fit into the reasonably sized crossbar array. Simulated results show that our proposed design achieves appealing performances in terms of speed and energy efficiency compared to both CPU and GPU platforms. In addition, a memristor crossbar array utilizing GSMR outperforms its counterpart with no-GSMR by 90% performance improvements and 44% energy reduction.**

I. INTRODUCTION

In recent years, great progresses have been made in fabricating high-speed memristors with appealing physical features [14][15][16]. In [1], crossbar array was proposed as a leading candidate for future memory and logic applications. Because of memristor crossbar's analog nature and small footprint on silicon, it can be configured as highly parallel and low power matrix vector multiplication computing engine [2][3][4]. Many researchers have proposed using memristor crossbar as computing accelerator, especially in applications such as neuromorphic computing [5][6][17].

Naive mapping from a matrix to a memristor crossbar array is usually wasteful and sometimes not feasible. First of all, the dimension of matrices and vectors varies from application to application. For many scientific computing and big data applications, matrices of interest can be extremely large. Design for the worst-case scenario would yield a huge crossbar. Although the memristor device allows high-density integration, the auxiliary circuit of the memristor crossbar array (e.g. D/A and A/D converters, op-amp circuits, etc.) grows with the crossbar size, it is apparent that a crossbar array with extremely large number of rows and columns would be impractical to fabricate [7]. Furthermore, in applications where matrices are sparse, direct mapping of a sparse matrix leads to very low hardware utilization. Finally, memristor has bounded maximum resistance, it is difficult to program them to zero conductance which corresponds to infinitely large resistance. Hence a sparse matrix that has large number of zero entries may lead to large approximation error when implemented using memristor crossbar.

To address the aforementioned issues, in this work, we propose to decompose the matrix into sub-blocks, which can be mapped to a set of fixed sized crossbar arrays, and the partial results are combined in the end. Those sub-blocks that contain all zeros will be omitted and not consume any computing resources. For example, a 1024×1024 sparse matrix can be decomposed into 1024 32×32 sub-blocks. Only those sub-blocks that have non-zero entries need to be mapped to memristor crossbar. The actual number of sub-blocks needed may be much less than 1024 due to the sparsity.

A question naturally arising is how to manipulate the sparse matrix to maximize the number of all-zero sub-blocks; which consequently minimizes the number of required memristor crossbar arrays. There has been a variety of literature devoted to sparse matrix partition for parallel computing using multicore CPU or GPGPU [8][9]. However, the main objective of these methods is to minimize inter-core communication, as the bandwidth and performance of bus and memory subsystem are the bottleneck of high performance computing. In a memristor crossbar based SpMV multiplication engine, reducing communication volume is no longer the goal. Instead, we propose to "group" the sparse matrix's non-zero entries as aggressively as possible. The contribution of this paper can be summarized as the following:

1. The hardware optimization of memristor based sparse matrix vector multiplication is formulated as to find the minimum non-zero block cover of a permutation of the given sparse matrix. To the best of our knowledge, this is the first work that associates minimum non-zero block cover with the matrix bandwidth, and applies matrix bandwidth reduction technique to optimize the memristor based hardware accelerator.
2. A generalized sparse matrix reordering (GSMR) technique is proposed to facilitate sparse matrix partition. Its benefit is twofold. First, it creates more all-zero sub-blocks, which do not require hardware resource; second, by removing as many zeros from the matrix as possible, it reduces the accumulated error.
3. The GSMR algorithm is applied to sparse matrices both from real life applications and random generation. Significant performance and energy improvements are achieved.

II. GENERALIZED SPARSE MATRIX REORDERING FOR CROSSBAR HARDWARE REDUCTION

A. Background

Fig. 1 [7] shows a memristor crossbar used as matrix vector multiplication engine, which computes $\boldsymbol{y} = \boldsymbol{Ax}$ in the analog domain, where A is an M by N matrix**.** It is made of two layers of metal wires, with N wires on the top, each corresponding to an entry in the input vector, and M wires at the bottom, each corresponding to an entry in the output vector. Between each overlap of a top wire and a bottom wire there is a memristor as the connector. It is easy to see that if the conductance of the memristor at coordinate i at the bottom and j on the top is programmed to have value $\boldsymbol{A}_{i,j}$, and $\boldsymbol{x}$ is applied as input voltage vector, $\boldsymbol{y}$ will be produced as the output current vector on the bottom wires. To incorporate the memristor crossbar into digital computing framework, extra circuitry is required. For example, A/D and D/A converters are needed at the input and output as the

communication interface. The bottom wires must be held at ground potential in order to make the crossbar function correctly. In addition, to support negative entries in $\boldsymbol{A}$, another crossbars and subtraction circuit is needed [10].

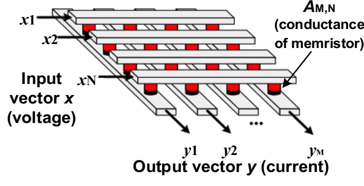

Fig. 1. Memristor crossbar for matrix vector multiplication.

B. Motivations

Consider a matrix vector multiplication kernel, $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$, where $\boldsymbol{A}$ is a large sparse matrix with m rows and n columns, $\boldsymbol{x}$ is a column vector of length n, and $\boldsymbol{y}$ is a column vector of length m. Given a set of square sized memristor crossbar arrays, whose dimension is $k\times k$, where k is much smaller than m and n. We divide the matrix A into $\left\lceil\frac{m}{k}\right\rceil \times \left\lceil\frac{n}{k}\right\rceil$ equal sized grids and map each grid block to a memristor crossbar. We use I and J to represent index of a grid and i and j to the index of a matrix element. A direct mapping decomposes the large matrix vector multiplication into $\left\lceil\frac{m}{k}\right\rceil \times \left\lceil\frac{n}{k}\right\rceil$ smaller matrix vector multiplications, represented as $\boldsymbol{y}_{I,J} = \boldsymbol{A}_{I,J}\boldsymbol{x}_J$. Here $\boldsymbol{x}_J$, $0 \le J \le \left\lceil\frac{n}{k}\right\rceil$, is a partition of vector $\boldsymbol{x}$, i.e. $\boldsymbol{x} = \bigcup_J \boldsymbol{x}_J$, and all grids $\boldsymbol{A}_{I,J}$, $0 \le J \le \left\lceil\frac{n}{k}\right\rceil, 0 \le I \le \left\lceil\frac{m}{k}\right\rceil$ form a partition of matrix $\boldsymbol{A}$, i.e. $\boldsymbol{A} = \bigcup_{I,J} \boldsymbol{A}_{I,J}$. The sub-vector $\boldsymbol{x}_J$ contains k elements copied from the original vector $\boldsymbol{x}$ as $\{x_{J\cdot k}, x_{J\cdot k+1}, \dots, x_{J\cdot k+k-1}\}$. The sub-matrix $\boldsymbol{A}_{I,J}$ contains matrix elements $\{a_{i,j}\}$ where $a_{i,j} \in \boldsymbol{A}$ and $I\cdot k \le i \le I\cdot k + k - 1, J\cdot k \le j \le J\cdot k + k - 1$. The final result $\boldsymbol{y}$ is partitioned into $\left\lceil\frac{m}{k}\right\rceil$ sub-vectors, $\boldsymbol{y} = \left[\boldsymbol{y}_0{}^T,\ \boldsymbol{y}_1{}^T, \dots \boldsymbol{y}_{\left\lceil\frac{m}{k}\right\rceil}{}^T\right]^T$, and its I-th entry is calculated as $\boldsymbol{y}_I = \sum_{J=1}^{\left\lceil\frac{n}{k}\right\rceil} \boldsymbol{y}_{IJ}$.

When $\boldsymbol{A}$ is sparse, some grids contain all zeros, hence do not need to map to any crossbar. In contrast to "direct mapping", a "compact mapping" finds the minimum set of grids $\boldsymbol{C} \subseteq \{\boldsymbol{A}_{I,J}\}$ that is a cover of all non-zero entries in $\boldsymbol{A}$. The goal of the proposed generalized sparse matrix reordering technique is to re-arrange the rows and columns of the matrix $\boldsymbol{A}$ to cluster the zeros and non-zeros, such that $|\boldsymbol{C}|$ (i.e. size of $\boldsymbol{C}$) is minimized. It aims at finding a row-wise permutation matrix $\boldsymbol{P}$ and a column-wise permutation matrix $\boldsymbol{Q}$, to obtain $\boldsymbol{A}' = \boldsymbol{PAQ}$ where $\boldsymbol{A}'$ is essentially a reorganized version of $\boldsymbol{A}$ with the same number of non-zero entries, but different sparsity distribution. The non-zero entries in $\boldsymbol{A}'$ are clustered so that a minimum set of non-zero grid blocks can be found. For simplicity, in the rest of the paper, we refer to a "grid block" simply as a "block".

In order to apply compact mapping with generalized matrix reordering, the similar permutation needs to be applied on the input vector as well. The transformed input vector is denoted as $\boldsymbol{x}'$, $\boldsymbol{x}' = \boldsymbol{Q^T x}$, which is realized by reordering $\boldsymbol{x}$ according to $\boldsymbol{Q^T}$. And the transformed output vector is denoted as $\boldsymbol{y}'$, $\boldsymbol{y}' = \boldsymbol{A}'\boldsymbol{x}'$. The expected output $\boldsymbol{y}$ can be obtained from $\boldsymbol{y}'$ by permutation: $\boldsymbol{y} = \boldsymbol{P^T}\boldsymbol{y}'$.

$$\mathrm{P} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \qquad \mathrm{Q} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} \qquad (1)$$

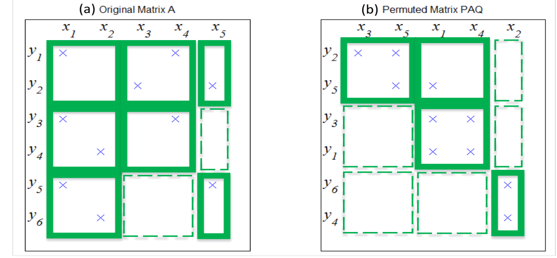

Fig. 2. A motivational example. (a) The original matrix A. (b) Permuted matrix A′.

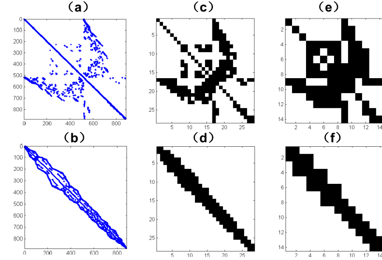

Fig. 3. Applying GSMR on sparse matrix *qh882*. Blue dots represent non-zero entries, and black squares represent non-zero sub-blocks. (a) Original matrix. (b) GSMR reordered matrix. (c) and (d) 32x32 non-zero block cover of original and reduced matrices. (e) and (f) 64x64 non-zero block cover of original and reduced matrices.

As an example that demonstrates how permutation can affect the non-zero entry distributions in a matrix, Fig. 2(a) gives the original 6×5 matrix denoted as $\boldsymbol{A}$, with each "×" denoting a non-zero entry. Notice that in the original matrix the non-zeros are scattered across the matrix. By applying $\boldsymbol{P}$ as the row permutation and $\boldsymbol{Q}$ as column permutation, shown in Eq. 1, we obtain a new matrix $\boldsymbol{A}' = \boldsymbol{PAQ}$ which has its non-zeros located closer to each other, as shown in Fig. 2(b). The permutated column and row indices are labeled in the figure. The solid green boxes in the figure are 2×2 blocks used to cover the non-zero entries. As we can see, the minimum size non-zero block cover is 7 and 4 for matrices $\boldsymbol{A}$ and $\boldsymbol{A}'$ respectively.

In the above procedure, the key is to find effective permutation matrix $\boldsymbol{P}$ and $\boldsymbol{Q}$ to map the original matrix $\boldsymbol{A}$ to $\boldsymbol{A}'$, which has minimum non-zero block cover. Traditionally there are a variety of algorithms that transforms sparse matrices into other forms to improve computation efficiency [8][9]. However, all of these algorithms aim at CPU or GPU based parallel computing platforms. Their objective is to reduce the communication between computing cores that work in parallel. We refer to such partition method as *communication optimal*. Two computing cores need to communicate with each other only when they are processing entries located in the same row. Hence the goal of "communication optimal" matrix reordering and partition is not to minimize the number of non-zero blocks, but to minimize the number of non-zero blocks located in the same row. Another method that is recently proposed to group non-zeros in a matrix is spectral clustering. Spectral clustering also seeks to group non-zeros entries in a sparse matrix by calculating eigenvalues of matrix derived from a similarity graph [7]. However, our experiments show that its performance is not as promising as our proposed GSMR technique.

C. Generalized Sparse Matrix Reordering

Given a $m\times n$ matrix $\boldsymbol{A}$ and its block partition Π, which partitions $\boldsymbol{A}$ into equal sized submatrices with dimension $k\times k$. The set of all submatrices that contains non-zero entries is referred to as the *non-zero block cover* of $\boldsymbol{A}$. We also define the *bandwidth* (B_A) of matrix $\boldsymbol{A}$ as $B_{\boldsymbol{A}} = \max_{i,j}|i-j| + 1$, $\forall i,j$, where a_{ij} is a non-zero

entry in matrix $\boldsymbol{A}$. It is easy to see that a diagonal matrix has bandwidth 1, and a matrix with small bandwidth is a matrix whose non-zero entries are clustered near the diagonal.

Without loss of generality, we constrain matrix $\boldsymbol{A}$ so that none of its rows or columns are all zeros. Otherwise, we simply remove that row or column. We refer to this type of matrices as weakly irreducible matrices, since they are a super-set of irreducible matrices. Intuitively we can see that, for the weakly irreducible matrix $\boldsymbol{A}$, its permutation $\boldsymbol{A'}$ will have smaller non-zero block coverage, if it has smaller bandwidth. For the extreme case when $\boldsymbol{A}$ is a square matrix that has only one non-zero entry in each row and column, this property can be strictly proved as in Theorem 1. The proof of the theorem is omitted due to space limitation.

Theorem 1. If $\boldsymbol{A}$ is a square matrix that has only one non-zero entry in each row and column, the minimum non-zero block cover occurs when $\boldsymbol{A}$ is permutated to a diagonal matrix.

The Cuthill-McKee algorithm is traditionally used to permute a symmetric sparse matrix into a band matrix [11]. Such permutation usually reduces the matrix bandwidth. However, the algorithm can only be applied to square symmetric matrix. To overcome the limitation, for a general matrix A, we construct matrix $\boldsymbol{B}$ as the following:

$$\boldsymbol{B} = \begin{pmatrix} \boldsymbol{0} & \boldsymbol{A}^{\mathrm{T}} \\ \boldsymbol{A} & \boldsymbol{0} \end{pmatrix} \quad (2)$$

Obviously when $\boldsymbol{A}$ is a matrix with m rows and n columns, $\boldsymbol{B}$ will be a symmetric matrix with $m+n$ rows and $m+n$ columns. If we deem $\boldsymbol{A}$ as a representation of a bipartite graph $G_{\boldsymbol{A}}$ in which there are m "row bank" vertices and n "column bank" vertices, $\boldsymbol{B}$ can be deemed as a representation of ordinary undirected graph $G_{\boldsymbol{B}}$, which has the same topological structure of $\boldsymbol{A}$. The relationship between $G_{\boldsymbol{A}}$ and $G_{\boldsymbol{B}}$ is vertices labeled 1 to n in $G_{\boldsymbol{B}}$ correspond to the "column bank" vertices in $G_{\boldsymbol{A}}$, and vertices labelled $n+1$ to $m+n$ in $G_{\boldsymbol{B}}$ correspond to the "row bank" vertices in $G_{\boldsymbol{A}}$. By applying Cuthill-McKee algorithm to $\boldsymbol{B}$, we obtain a permutation matrix $\boldsymbol{V}$ such that $\boldsymbol{VBV^T}$ is a band matrix. If we denote $\boldsymbol{\pi}$ as the permutation corresponding to $\boldsymbol{V}$, then we can obtain a row permutation $\boldsymbol{\alpha}$ by extracting entries that represents "row bank" vertices in $\boldsymbol{\pi}$ and maintain their order. Likewise, we can obtain column permutation $\boldsymbol{\beta}$ by extracting entries that represents "column bank" vertices in $\boldsymbol{\pi}$ and maintain their order. By applying $\boldsymbol{\alpha}$ as the row permutation and $\boldsymbol{\beta}$ as the column permutation on $\boldsymbol{A}$**,** a bandwidth reduced $\boldsymbol{A'}$ can be obtained. We refer to this extended Cuthill-McKee method as *generalized sparse matrix reordering* (*GSMR*) as it can be applied to general matrix that is rectangular and non-symmetrical. The GSMR procedure is summarized in Algorithm 1.

ALGORITHM 1. GENERALIZED SPARSE MATRIX REORDERING

	Input: rectangular, unsymmetrical matrix $\boldsymbol{A}$
	Output: matrix $\boldsymbol{A'}$ ($\boldsymbol{A}$ after reordering), row permutation $\boldsymbol{\alpha}$ and column permutation $\boldsymbol{\beta}$
1	Construct matrix $\boldsymbol{B}$ from $\boldsymbol{A}$ as in Eq. 2
2	View $\boldsymbol{B}$ as a connection graph G_B
3	Choose in G_B a vertex **x** with the lowest degree and let **R** := ({**x**}); permutation vector $\boldsymbol{\pi}$ = ø
4	**while** |**R**| < the number of all vertices in G_B
5	Construct the adjacency set $\mathbf{P}_i$ of $\mathbf{R}_i$ ($\mathbf{R}_i$ denoting i-th component of **R**) and exclude the vertices already in **R**
6	Sort $\mathbf{P}_i$ ascendingly according to vertex degree
7	Append $\mathbf{P}_i$ to the result set **R**
8	Append index of $\mathbf{R}_i$ to the end of $\boldsymbol{\pi}$
9	**endwhile**
10	Extract $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ from $\boldsymbol{\pi}$ respectively
11	Obtain $\boldsymbol{A'}$ by apply $\boldsymbol{\alpha}$ on rows of $\boldsymbol{A}$ and $\boldsymbol{\beta}$ on columns of $\boldsymbol{A}$

An example of how GSMR help to reduce the non-zero block cover is given in Fig. 3. Here GSMR is applied on a sparse matrix called *qh882*. Using 32×32 sub-blocks, the size of non-zero block cover goes down from 199 to 134 after GSMR, which corresponds to a 33% reduction, as shown in Fig. 3(c) and Fig. 3(d). Using 64×64 sub-blocks, the number goes down from 82 to 46, or a 44% reduction, as shown in Fig. 3(e) and Fig. 3(f).

III. Experimental Results and Evaluation

To evaluate the performance of the proposed framework employing generalized sparse matrix reordering (GSMR) algorithm, we compare its performance with three other baseline platforms: CPU platform, GPU platform, and memristor crossbar based platform with naïve "compact mapping" (i.e. without GSMR but omit all-zero sub-blocks). The CPU based baseline uses Intel i7-3770K CPU running at 3.5GHz with power consumption of 70W. On this platform, SpMV multiplication is realized with compiled code written in C++, and takes matrices in coordinate list (COO) format as input. Also we use CUSP, a C++ sparse matrix library running with an Nvidia Tesla C2070 GPU to test its SpMV multiplication performances. The GPU has a peak power of 215W.

The parameters of the memristor crossbar and details of its associated circuitry [12] are illustrated in Table I. The clock speed is set to 1GHz and the power is estimated with Design Compiler. The memristor crossbar size we use in the experiment is set to 32×32 and 64×64 respectively. A portion of sparse matrix data that we use is obtained from [13], which contains examples from a variety of research realms. We also randomly generated sparse matrices of different sizes and sparsities (rand1 to rand4). Details of these sparse matrices can be found in Table II. The input vectors are dense and randomly generated. Each sparse matrix is multiplied with 100 randomly generated vectors and the average performance is reported.

TABLE I. PARAMETERS USED BY THE SIMULATION

Data Converter	ADC Power	ADC latency	DAC Power	DAC latency
	290.59mW/GHz	1ns	7.32mW/GHz	1ns
Sensing Circuit	Op Amp Power		Op Amp Delay	
	100μW		0.6ns	
Control	Program Pulse (T_{prog})		Evaluation Pulse (T_{eval})	
	10ns		1ns	
Other	V_{dd}		MCC power (P_{MCC})	
	1.0V		15.7mW	

TABLE II. SPARSE MATRICES USED IN THE EXPERIMENT

Matrix Name	# of Rows	# of Columns	# of Non-zeros	Sparsity
illc1033	1033	320	4732	0.0143
illc1850	1850	712	8758	0.0066
qh1484	1484	1484	6110	0.0028
qh882	882	882	3354	0.0043
rand1	1000	1100	110000	0.1000
rand2	1100	1000	110000	0.1000
rand3	1000	1100	500	0.0010
rand4	1100	1000	500	0.0010
zenios	2873	2873	15032	0.0018

Table III shows the reduction of non-zero block cover size of those sparse matrices after GSMR reordering. The block dimension is set to either 32 or 64. We can observe that for matrices that are relatively sparser (rand3, rand4 and zenios), the reduction in the size of non-zero block cover is more significant than relatively denser ones (rand1 and rand2). This indicates that better performance can be

`

obtained for sparser matrices. On average using GSMR reduces the number of sub-blocks by 45.9% and 45.7% respectively. The percentage reduction after using spectral clustering is also given. Note that spectral clustering performs worse than GSMR in all cases, and in some cases it even increases the size of non-zero block cover. This is because for those sparse matrices, the non-zero entries are already moderately grouped toward the diagonal line, spectral clustering tends to break them apart and make them scatter around. We also tested the "communication optimal" partition traditionally used by SpMV in multicore systems. Since they target at different objective, our experiment shows a unified deterioration compared to the original matrix in terms of number of non-zero blocks. Due to the space limitation, their results are not listed here.

TABLE III. PERCENTAGE REDUCTION IN THE SIZE OF NON-ZERO BLOCK COVER

Matrix Name	$k = 32$		$k = 64$	
	GSMR (%)	SC (%)	GSMR (%)	SC (%)
illc1033	43.9	20.6	31.1	2.7
illc1850	37.0	-0.9	42.9	-2.5
qh1484	47.3	31.3	59.1	28.2
qh882	32.7	15.1	43.9	1.2
rand1	2.7	0.0	2.1	0.0
rand2	1.6	1.0	3.5	0.0
rand3	91.4	-56.2	89.3	-1.8
rand4	91.9	-64.5	89.5	61.7
zenios	66.5	18.2	64.8	24.6
Ave.	**45.9**	**-2.1**	**45.7**	**11.4**

Table IV summarizes the performance and energy performance of different platforms. For the CPU based implementation, the performance is measured as the application CPU time that is used only to calculate the matrix vector operation with exclusion of the I/O time. The energy consumption is the accumulated CPU power consumption during that time. It is worth to note that for relatively small sized matrices, GPU is even slower than CPU, which is caused by the overhead of transferring data from host memory to device memory.

TABLE IV. PERFORMANCE AND ENERGY IMPACT OF USING GSMR

Reference Implementation	GSMR ($k = 32$)		GSMR ($k = 64$)	
	Speed	Energy	Speed	Energy
CPU based	2.7×	171.8×	3.3×	121.3×
GPU based	2.6×	514.7×	3.0×	398.9×
Crossbar w/o GSMR	1.9×	1.8×	1.9×	2.0×

For memristor crossbar based framework, the performance is measured as the time spent on memristor programming, crossbar results evaluation and partial results merging. The energy is measured as total energy dissipation on controller, data converters, op-amps and other auxiliary components. For crossbar size of 32, we can see that compared to calculating the sparse matrix operation on CPU (GPU) platforms, the crossbar based hardware platform with GSMR optimization achieves on average 2.7× (2.6×) improvement in computation time and 171.8× (514.7×) improvement in energy consumption. Compared to the crossbar based hardware platform with without GSMR optimization, the one with GSMR optimization gives 1.9× improvements in computation time and 1.8× improvements in energy consumption. Using a larger (64×64) crossbar, the improvements in performance is even more significant, but the energy improvement is reduced. The results indicate that k is a parameter that controls the trade-off in performance and energy dissipation. Exploring the impact of k will be one of the directions of our future work.

IV. CONCLUSIONS

In this paper, we present a highly efficient sparse matrix vector (SpMV) multiplication framework featuring memristor crossbar accelerator. We develop a technique called Generalized Sparse Matrix Reordering (GSMR) by leveraging linear transformation to break down rectangular matrices into sub-blocks to make them fit into reasonably sized crossbar, and reduce the number of sub-blocks. Experimental results show that compared to CPU, GPU and no-GSMR platforms, our GSMR based platform achieves great reduction in both computation time and energy consumption.

ACKNOWLEDGEMENT

This work is partially supported by the National Science Foundation under Grants CCF-1337300.

REFERENCES

[1] Kuk-Hwan Kim et al, "A Functional Hybrid Memristor Crossbar-Array/CMOS System for Data Storage and Neuromorphic Applications", *Nano Letters* 2012 12 (1), 389-395.

[2] Miao Hu; Hai Li; Qing Wu; Rose, G.S.; Yiran Chen, "Memristor crossbar based hardware realization of BSB recall function," *Neural Networks (IJCNN), The 2012 International Joint Conference on*, vol., no., pp.1,7, 10-15 June 2012.

[3] Miao Hu; Hai Li; Yiran Chen; Qing Wu; Rose, G.S.; Linderman, R.W., "Memristor Crossbar-Based Neuromorphic Computing System: A Case Study," *Neural Networks and Learning Systems, IEEE Transactions on* , vol.25, no.10, pp.1864,1878, Oct. 2014.

[4] Boxun Li; Yi Shan; Miao Hu; Yu Wang; Yiran Chen; Huazhong Yang, "Memristor-based approximated computation", *ISLPED 2013 IEEE International Symposium on*, pp.242,247, 4-6 Sept. 2013.

[5] Beiye Liu; Miao Hu; Hai Li; Zhi-Hong Mao; Yiran Chen; Tingwen Huang; Wei Zhang, "Digital-assisted noise-eliminating training for memristor crossbar-based analog neuromorphic computing engine," *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE* , vol., no., pp.1,6, May 29 2013-June 7 2013.

[6] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive Devices for Computing," *Nat. Nanotechnol*., vol. 8, no. 1, pp. 13–24, 2013.

[7] Wei Wen; *et al*., "An EDA Framework for Large Scale Hybrid Neuromorphic Computing Systems." *Proc. 52th Annual. Design. Automation Conference, (DAC '15)*, June 2015.

[8] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," *Proc. 27th Int. ACM Conf. Int. Conf. Supercomput. - ICS '13*, p. 273, 2013.

[9] N. Bell and M. Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA," Nvidia, 2008.

[10] R. E. Pino, H. (Helen) Li, Y. Chen, M. Hu, and B. Liu, "Statistical memristor modeling and case study in neuromorphic computing," *Proc. 49th Annu. Des. Autom. Conf. - DAC '12*, p. 585, 2012.

[11] E. Cuthill and J. McKee. 1969. "Reducing the bandwidth of sparse symmetric matrices," *Proceedings of the 1969 24th national conference (ACM '69)*. ACM, New York, NY, USA, 157-172.

[12] X. Liu, C. Engr, and J. J. Yang, "A Heterogeneous Computing System with Memristor- Based Neuromorphic Accelerators," *Proceedings of 18th IEEE High Performance Extreme Computing Conference*, 2014.

[13] http://math.nist.gov/MatrixMarket/data/SPARSKIT/

[14] Chua, L.O., "Memristor-The missing circuit element," *Circuit Theory, IEEE Transactions on*, vol.18, no.5, pp.507,519, Sep 1971.

[15] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S.Williams, "The missing memristor found," *Nature*, vol. 453, pp. 80–83, 2008.

[16] J. Joshua Yang, M. X. Zhang, M. D. Pickett, F. Miao, J. Paul Strachan, W. Di Li, W. Yi, D. A. A. Ohlberg, B. Joon Choi, W. Wu, J. H. Nickel, G. Medeiros-Ribeiro, and R. Stanley Williams, "Engineering nonlinearity into memristors for passive crossbar applications," *Appl. Phys. Lett*., vol. 100, no. 2012, pp. 98–102, 2012.

[17] R. E. Pino, H. (Helen) Li, Y. Chen, M. Hu, and B. Liu, "Statistical memristor modeling and case study in neuromorphic computing," *Proc. 49th Annu. Des. Autom. Conf. - DAC '12*, p. 585, 2012.