IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems

Preserving Row Buffer Locality for PCM Wear-Leveling Under Massive Parallelism

Xinning Wang* Bin Wang* Zhuo Liu* Weikuan Yu‡

Auburn University*
{xzw0033,bwang,zhuoliu}@auburn.edu

Florida State University‡
{yuw}@cs.fsu.edu

***Abstract*—Phase Change Memory (PCM) is a promising alternative technology for DRAM because of its advantages in terms of transistor density and energy consumption. It has been exploited to work in concert or alone inside various memory systems to meet the growing bandwidth needs of massive parallelism. PCM memory cells, however, have a common problem of limited write endurance. Various wear-leaving techniques have been employed for uniform distribution of memory writes, typically through address transformation schemes such as randomization to avoid hot writes. Unfortunately, such address transformation can have the undesirable consequence of disrupting the row buffer locality in sequential memory accesses, resulting in the loss of memory performance. Our analysis reveals that this situation is particularly severe under massive parallelism of manycore processors such as GPUs. In this paper, we introduce a combination of two techniques, matrix-based partial randomization and row-buffer locality-aware rotation, to alleviate the locality disruption of address transformation and preserve the row buffer locality of PCM-based global memory in GPU. Our evaluation results show that, compared to existing techniques, our techniques can adequately preserve the row buffer locality and minimize the loss of memory performance, while achieving similar endurance and better energy efficiency for a variety of GPGPU applications.**

I. Introduction

Graphic Processing Units (GPUs) have become a compelling solution due to their massive parallelism and high energy efficiency. However, massive parallelism with hundreds of thousands of threads places enormous pressure on memory capacity and bandwidth. One strategy is to leverage the Non-Volatile Memory (NVM), such as Phase-Change Memory (PCM), Resistive Random Access Memory (ReRAM) or Spin-Transfer Torque RAM (STT-RAM or MRAM). These memory devices have superior features, such as higher performance, greater energy efficiency, and higher cell density. Nevertheless, a main hurdle for these memory devices is their limited lifetime. For example, compared to DRAM (10^{15} writes), the lifetime of PCM (10^8) is orders of magnitude shorter.

A lot of wear-leveling techniques [22, 23, 26, 14, 2] have been proposed to address such endurance issue. Among them, Start-Gap [22] becomes quite popular, because it can uniformly spread memory accesses to all memory sub-regions with small overhead. In general, Start-Gap first uses a low-cost rotation scheme to relocate heavily written cache lines to neighboring positions, and then randomizes the memory address space to reduce the clustering of heavily written lines.

Normally, memory devices use row buffers to facilitate memory accesses. Row buffer is an external circuitry to temporarily hold a row of data loaded from memory chips. Consecutive memory requests that hit in row buffer, which is referred to as Row Buffer Locality (RBL), can be serviced immediately. Otherwise, a new row will be loaded into row buffer through a series of memory commands, which can be 10X slower than direct row buffer accesses. Considering the long latency and high energy cost of PCM writes, preserving RBL can have triple impacts on latency, throughput, and energy efficiency [9, 19, 27, 35, 11].

For GPU programming, streaming access pattern is common and critical to gain high performance and reduce the programming complexity. This pattern generates bursty sequential accesses to logically contiguous memory regions and can greatly benefit from preserved RBL. However, randomizing memory address space due to endurance concerns distributes logically contiguous memory blocks into arbitrary regions, destroying potential RBL for both reads and writes inside GPU memory access streams. Similar issues also exist in other wear leveling techniques [23, 26]. Thus, RBL-oblivious wear-leveling techniques can incur significant penalty to the performance of GPU memory sub-systems.

These observations have motivated us to investigate and explore a RBL-aware wear leveling solution. In this paper, we examine several address mapping schemes and empirically analyze their impacts on RBL, performance and endurance. We present a novel bijective matrix-based partial random scheme and a RBL-aware rotation algorithm. They reduce the RBL losses caused by the existing wear-leveling schemes and hence improve PCM-based GPU performance. Our evaluation results show that the integrated RBL preservation scheme can minimize the loss of RBL, while achieving similar endurance and better energy efficiency for PCM devices than the Start-Gap for a variety of GPU applications.

In summary, our study makes the following contributions:

1) We have observed that randomization-based wear leveling for non-volatile memories can destroy the RBL inherently embedded in GPGPU applications, degrading the performance of memory systems;
2) We propose a partial randomization method to constrain the degradation of RBL via bijective matrices, prove the prerequisite of matrices locality, analyze and compare the changes of variances in linear method;
3) We design a RBL-aware Rotation (RAR) policy to

1526-7539/15 $31.00 © 2015 IEEE
DOI 10.1109/MASCOTS.2015.39

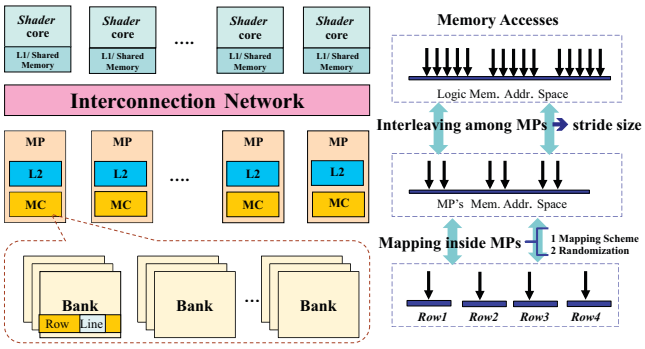


Fig. 1: GPU architecture and hierarchical address mapping

further preserve RBL in the operations of relocating heavily written blocks;

4) We implement the two techniques in a cycle-accurate simulator. Our evaluations show that, compared with the state-of-the-art wear leveling technique (Start-Gap), our techniques on average achieve 9% reduction of memory service latency, 8% saving of power consumption, 8% of IPC performance improvement, and comparable endurance for a variety of GPGPU applications.

In the rest of this paper, we first introduce our baseline GPU architecture in Section II and describe our motivations in Section III. The two key techniques are elaborated in Sections IV and V. Experimental results are presented in Section VI. Finally, we review the related work in Section VII and conclude the paper in Section VIII.

II. Baseline GPU Architecture

We study a baseline GPU that is similar to the NVIDIA's Fermi Architecture [20]. As shown on the left of Fig. 1, there are several Streaming Multiprocessors (SMs or shade cores) in our baseline GPU, each of which has on-chip L1 caches and shared memory. Multiple Memory Partitions (MPs) are equipped to collectively sustain the memory bandwidth requirement of massive threads in the SMs. Each MP consists of a L2 data cache and a Memory Controller (MC) to manage the high-bandwidth but long-latency off-chip memory devices. SMs and MPs are connected through an interconnection network. The right part of Fig. 1 illustrates the steps of transforming the addresses of individual memory requests in the global memory address space into physical addresses in memory chips. First, global addresses are striped into local addresses in each MP in an interleaved manner. Our baseline GPU has a default stride length of 256B. Then local addresses are mapped into physical addresses in the memory chips, i.e., banks, rows, and columns.

Our baseline architecture is simulated on GPGPU-Sim (version 3.2.1), a cycle-accurate GPGPU Simulator [3]. The characteristics of our baseline GPU [1] and parameters of PCM [11] are summarized in Table I. This baseline GPU is also studied in [32, 29, 31, 33, 30]. tCCD, tRRD, and tRCD specify the delays in column-to-column, row-to-row, and row-to-column activations. tRAS is the time interval between row access and data restoration in PCM. We use tRP to simulate

TABLE I: Baseline GPU Configuration

# SM	30
SM Config.	1.4GHz, Pipeline width:32, Branch divergence handling: immediate post-dominator, Warp scheduling: RR
Resources/SM	#Reg: 32 KB, Shared Memory: 48KB, 32 threads/warp, 48 warps/SM, 8 blocks/SM
L1 Private Data$	32KB, 128B line, 4-way assoc., 32 MSHRs
L2 Cache/MP	128KB, 128B line, 16-way assoc., 32 MSHRs, 700 MHz, Minimum latency: 120 cycles
Interconnect	1400 MHz, 32B channel width, 1 stage butterfly network
# Mem. Partitions	6
Mem. Channel	FR-FCFS, 8 bytes/cycle, Minimum Latency: 220 cycles
Mem. Model	1848 MHz, 16 Banks/channel, 4KB row size
Mem. Timing	nbk =16, nbkgrp=4, tCCD=7, tRRD=18, tRCD=37, tRAS=46, tRP=100, tRC=146, tCL=5, tWL=4, tWTR=10
Mem. Energy (pJ/b)	Array read: 2.47, Array write: 16.82, Row buffer read: 0.92, Row buffer write: 1.02, background: 0.08

TABLE II: Access Patterns of Benchmarks

#	Benchmark	Description	MPKI	WPKI	write%
1	SP	ScalarProd	2.68	0.001	0.038
2	NN	Nearest Neighbor	6.25	2.5	40
3	KM	Kmeans	2.15	0.027	1.3
4	FWT	FastWalshTransform	2.67	1.293	48.5
5	SN	SortingNetworks	1.12	0.534	47.5
6	CS	ConvolutionSeparable	2.05	0.994	48.4
7	BFS	Breadth First Search	23.4	6.74	28.8
8	BS	BlackScholes	2.21	0.633	28.6
9	SC	Streamcluster	4.31	0.38	8.8
10	MT	MersenneTwister	1.49	0.746	50.1

the Write Recovery time (tWR) of PCM. tRC indicates the time of row accesses and row buffer precharge. tCL and tWL restrict the frequency of the buffer commands, and tWTR shows the delay time from write to read. We consider a wide range of memory intensive GPGPU applications from CUDA SDK [21] and Rodinia [4]. Table II lists major characteristics of the benchmarks used in this study. MPKI denotes Misses Per Kilo Instructions; WPKI denotes memory Writes Per Kilo Instructions; write% represents the percentage of writes in each benchmark.

III. Analysis of Address Mapping Schemes

In this section, we motivate this paper by examining three typical address mapping schemes and analyzing their distinct impacts on endurance, performance, and energy efficiency for a PCM-based GPU global memory sub-system.

A. Address Mapping Schemes

We use a 30-bit virtual address space to illustrate the mapping from local addresses to physical addresses. In our baseline, local addresses are mapped onto banks, rows, and columns in the unit of 256B memory segment. Fig. 2 shows how bank (*Bank_Index*), row (*Row_Index*), and column (*Column_Index*) addresses are calculated in three baseline mapping schemes. The lowest several bits represent the offset within each memory segment. These schemes are further described below in more details.

- Basic Address Mapping scheme (BAM) preserves RBL by mapping 2^c logically contiguous memory segments into the row that is located via *Row_Index* and *Bank_Index*. When the current row is full, the row with the same *Row_Index* but different *Bank_Index* is selected.

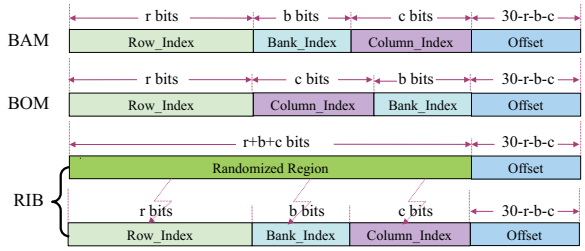


Fig. 2: Address mapping in BAM, BOM and RIB

TABLE III: Bank-level skews in three schemes

	SP	NN	KM	FWT	SN	CS	BFS	BS	SC	MT	GMean
BAM	1.03	1.03	1.14	1.15	1.18	1.06	2.23	1.03	3.67	1.04	1.31
BOM	1.99	1.94	1.78	2.04	2.04	1.91	1.96	2.00	1.98	1.98	1.96
RIB	1.01	1.01	1.02	1.03	1.05	1.01	2.69	1.04	3.27	1.03	1.26

- Bank-Oriented Mapping scheme (BOM) achieves high bank level parallelism (BLP) by constantly mapping contiguous memory segments into neighboring banks.
- Random Invertible Binary matrix based scheme (RIB) is used in conjunction with the Start-Gap wear leveling algorithm [22]. In this illustration, RIB first uses an invertible matrix to randomize the local address space and then takes the BAM scheme for a final mapping of addresses. RIB can better normalize accesses to different memory sub-regions of a PCM device, which is the key to improve its lifetime.

Note that *Row_Index* in the three schemes are always in the most significant bits of the local addresses to maximize either RBL (in BAM) or BLP (in BOM).

B. Wear Leveling Capabilities

Address mapping schemes determine the distribution of memory segments in the memory devices, thus they can affect the endurance of PCM chips. We use bank-level write skew [3, 15] to quantify such impacts. Bank-level write skew is calculated as the ratio of the maximal write counts to average write counts for all banks, as shown below:

$$BankSkew = W_{max}/E\{W_i\}, \tag{3.1}$$

where W_i is the number of writes to the i^{th} memory bank, W_{max} means the maximal value among W_i, and $E\{W_i\}$ is the average (or expected) write counts of all banks. $BankSkew$ indicates the write skew among different banks of PCM chips, thus it is directly correlated with the lifetime of a PCM device. A minimum value of 1 for $BankSkew$ indicates a perfect wear-leveling, while other larger values represent non-uniform writes to PCM chips. Substantial $BankSkew$ can quickly wear out a PCM chip. Table III lists the bank-level write skews in the three mapping schemes. As we can see, on average, RIB has the lowest bank-level skews, i.e., the best wear leveling potentials.

C. Row Buffer Locality and Performance

Fig. 3 shows the impacts of the three address mapping schemes on the performance of the evaluated GPU benchmarks in terms of Instructions Per Cycle (IPC). The results are normalized to BAM. Since the evaluated benchmarks

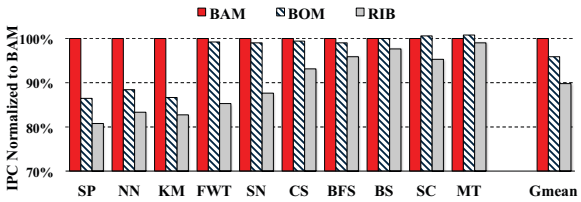


Fig. 3: IPC of RIB, BOM and BAM

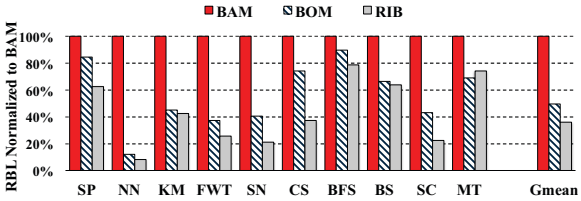


Fig. 4: Row Buffer Locality of RIB, BOM and BAM

have streaming access patterns with high RBL for inter-warp memory accesses, BAM has the best performance among the three schemes. By contrast, BOM and RIB on average (geometric mean) cause 4% and 10% IPC loss, respectively. The performance degradation is especially significant for SP, NN, KM and FWT, whose memory accesses are dominated by streaming patterns. BOM can maximize BLP, which is important to sustain memory throughput. The performance loss of BOM reflects that the GPGPU applications are more sensitive to RBL. Meanwhile, BAM can help preserve BLP to certain extent.

We define the following metric to quantify RBL:

$$RBL_{avg} = \frac{\sum_{i=1}^{s} ACC_i}{\sum_{i=1}^{s} ACT_i}, \tag{3.2}$$

where RBL_{avg} stands for the average RBL among different banks, ACC_i is the total number of memory accesses to the i^{th} bank, ACT_i is the total number of row activations in the i^{th} bank, and s is the number of banks. Fig. 4 shows the RBL_{avg} of the evaluated benchmarks in the three mapping schemes. The results are normalized to BAM, which achieves the maximal RBL. Compared to BAM, BOM and RIB on average conserve only 50% and 36% of RBL, respectively. BOM degrades RBL but achieves higher BLP. However, RIB weakens both RBL and BLP due to its address randomization component that aims at uniformly spreading accesses. Such performance disadvantage of RIB is essentially caused by its focus on wear leveling, as shown in Table III. Lack of RBL can incur frequent long-latency row activation commands to memory chips, degrading the throughput of GPU memory subsystems. Thus, we can conclude that a RBL-aware address mapping scheme is crucial to sustain the performance of GPGPU applications on GPUs.

D. Row Buffer Locality and Energy Efficiency

Besides aforementioned performance implications, preserving RBL can merge consecutive write requests and then reduce write traffic to memory chips. Because of the asymmetric energy consumption of PCM read/write requests, exploiting RBL is also critical to achieve high energy efficiency of leveraging

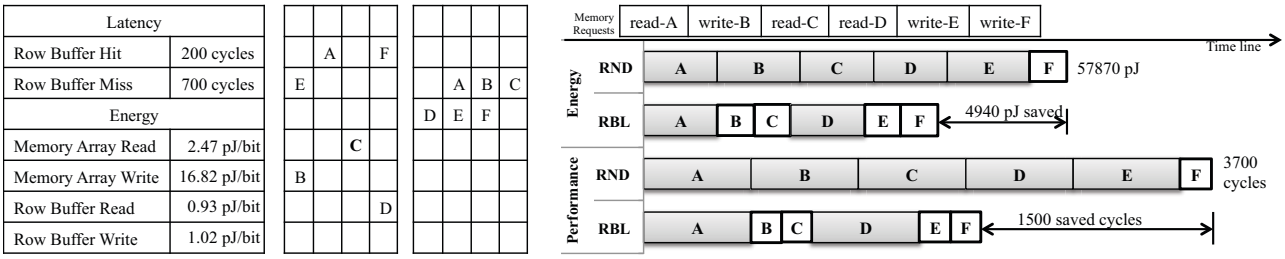


Fig. 5: Conceptual comparison between randomization-based and RBL-aware address mapping schemes. (a) Timing and energy parameters of PCM devices; (b) Data block locations using randomization-based wear leveling; (c) Data block locations using RBL-aware wear leveling; and (d) Performance and energy consumption comparison.

PCM devices in GPGPUs. Fig. 5 illustrates how RBL-aware and randomization-based address mapping schemes differ in maintaining performance and energy efficiency.

Fig. 5(a) lists the key PCM parameters, which are based on the parameters reported in [11]. In order to illustrate this problem, we assume six memory requests to different memory blocks, A, B, C, D, E, and F. Randomization-based and RBL-aware address mapping schemes result in distinct distributions for the six data blocks, shown in Fig. 5(b) and Fig. 5(c), respectively. In this example, only A and F are mapped into the same row in the randomization-based address mapping; while A, B, and C (similarly, D, E, and F) are mapped in the same row in RBL-aware address mapping. Due to the different capabilities in preserving RBL, randomization-based address mapping (*RND*) consumes 57,870 pJ energy, while RBL-aware address mapping (RBL) consumes 52,930 pJ energy, saving 4,940 pJ. Meanwhile, RBL-aware address mapping also saves 1500 cycles for the memory sub-system to service the six requests. Note that this energy saving is mainly from the reduced dynamic energy consumption, and this can be more significant if the static power consumption is also considered.

In this paper, we aim to design an address mapping scheme that can achieve comparable endurance potential of RIB and high performance of BAM. To this end, we propose a partial randomization method using bijective matrices to constrain randomization into smaller blocks rather than the whole memory address space. On top of that, we propose a RBL-aware rotation mechanism to further improve the endurance of PCM-based global memory in GPUs. These techniques are elaborated in detail in the next two sections.

IV. Partial Randomization Through Bijective Matrices

In this section, we propose an address transformation scheme that can achieve partial randomization through bijective matrices. Then, we prove its partial randomization ability and the prerequisite of invertibility, including how to set up the matrix. Furthermore, we analyze the variance of address randomization and show a strategy for the hardware implementation of bijective matrices.

A. Partial Randomization

The main idea of partial randomization is to construct a partitioned matrix which can partially randomize the memory addresses so that bank-level write skews can be minimized while retaining RBL in streaming accesses.

Suppose we have a square matrix M with dimension $(r+b+c)\times(r+b+c)$, where r, b and c represent the width of Row_Index, $Bank\_index$, and $Column\_Index$ as shown in Fig. 2, respectively. Since the $offset$ bits do not affect RBL, we can ignore them to simplify the problem. Although the column bits can not destroy RBL directly, randomizing them conduces to shuffling the entire addresses and simplifying the implementation of hardware. The address space of bank, row, and column can be formulated as single square bijective matrices [6]. Such matrices can be partitioned into nine submatrix blocks as the matrix M shown in (4.1):

$$M = \begin{pmatrix} A & B & C \\ D & E & F \\ G & H & K \end{pmatrix} \tag{4.1}$$

The dimensions of A, B, C, D, E, F, G, H, and K are r×r, r×b, r×c, b×r, b×b, b×c, c×r, c×b, and c×c, respectively. Note that A, E, and K in the cross line are square matrices.

Theorem 1: If A and $E - DA^{-1}B$ are invertible, then M of (4.1) is invertible, $\Leftrightarrow K - GA^{-1}C - (H - GA^{-1}B)(E - DA^{-1}B)^{-1}(F - DA^{-1}C)$ is invertible.

Proof: We elaborate the deduction process of Theorem 1 in two steps. First we describe the required condition in Theorem 1. The first line of matrix M, multiplies matrix $-DA^{-1}$, where each line of block matrix undergoes the Element Row Operation (ERO), then is added to the second line. Similarly, we multiply the first line with $-GA^{-1}$, and add the result to the third line. The resulting matrix is then M_1 as shown below:

$$M_1 = \begin{pmatrix} A & B & C \\ 0 & E - DA^{-1}B & F - DA^{-1}C \\ 0 & H - GA^{-1}B & K - GA^{-1}C \end{pmatrix} \tag{4.2}$$

$\because E - DA^{-1}B$ is invertible $\therefore (E - DA^{-1}B)^{-1}$ exists.
We then multiply the second line of M_1 with $(H - GA^{-1}B)(E - DA^{-1}B)^{-1}$, and add the product to the third line. Let us consider $T = K - GA^{-1}C - (H - GA^{-1}B)(E - DA^{-1}B)^{-1}(F - DA^{-1}C)$. We will have a new matrix, M_2 as shown below:

$$M_2 = \begin{pmatrix} A & B & C \\ 0 & E - DA^{-1}B & F - DA^{-1}C \\ 0 & 0 & T \end{pmatrix} \tag{4.3}$$

Using similar ERO, M_2 is further transformed into matrix M_3 as shown below:

$$M_3 = \begin{pmatrix} A & 0 & 0 \\ 0 & E - DA^{-1}B & 0 \\ 0 & 0 & T \end{pmatrix} \tag{4.4}$$

$\because \left|M\right| = \left|A\right| \cdot \left|E - DA^{-1}B\right| \cdot \left|T\right| \neq 0 \therefore \left|T\right| \neq 0$.
$\therefore$ If A and $E - DA^{-1}B$ are invertible, then M is invertible.
$\Rightarrow (K - GA^{-1}C - (H - GA^{-1}B)(E - DA^{-1}B)^{-1}(F - DA^{-1}C))$ is invertible.

Next we prove the necessary condition of Theorem 1 by constructing an extended matrix $M^{'}$ to calculate matrix M^{-1}.

$$M^{'} = \begin{pmatrix} A & B & C & I_n & 0 & 0 \\ D & E & F & 0 & I_m & 0 \\ G & H & K & 0 & 0 & I_s \end{pmatrix} \tag{4.5}$$

Using the same ERO, we can multiply the first line with matrix $-DA^{-1}$, then add the product to the second line. In addition, we multiply the first line with matrix $-GA^{-1}$, then add the product to the third line. Matrix $M^{'}$ is then transformed to matrix M^* as shown below:

$$M^* = \begin{pmatrix} A & B & C & I_n & 0 & 0 \\ 0 & E - DA^{-1}B & F - DA^{-1}C & -DA^{-1} & I_m & 0 \\ 0 & H - GA^{-1}B & K - GA^{-1}C & -GA^{-1} & 0 & I_s \end{pmatrix} \tag{4.6}$$

Let us consider matrix $T_1 = -(H - GA^{-1}B)(E - DA^{-1}B)^{-1}$, and matrix $T_2 = (H - GA^{-1}B)(E - DA^{-1}B)^{-1}DA^{-1} - GA^{-1}$. We then multiply the second line with matrix T_1, and add T_2 to the third line. Matrix M^* is then transformed to matrix M^{**} as shown below:

$$M^{**} = \begin{pmatrix} A & B & C & I_n & 0 & 0 \\ 0 & E - DA^{-1}B & F - DA^{-1}C & -DA^{-1} & I_m & 0 \\ 0 & 0 & T & T_2 & T_1 & I_s \end{pmatrix} \tag{4.7}$$

We further multiply each line with the invertible matrices of the first non-zero block matrices. Through these procedures, the extended matrix $M^{'}$ is eventually transformed into the Row Echelon Form (REF) which is shown as below:

$$\begin{pmatrix} I_n & . & . & . & . & . \\ 0 & I_m & . & . & . & . \\ 0 & 0 & I_s & . & . & . \end{pmatrix} \tag{4.8}$$

Matrices of Row Echelon Form (REF) can be transformed into the Reduced Row Echelon Form (RREF) through ERO. Therefore, the extended matrix can be transformed into a form of matrices as shown below:

$$\begin{pmatrix} I_n & 0 & 0 & . & . & . \\ 0 & I_m & 0 & . & . & . \\ 0 & 0 & I_s & . & . & . \end{pmatrix} \tag{4.9}$$

Hence, Theorem 1 is correct. ■

Invertibility of matrix M is the prerequisite of randomization in Start-Gap and its variants. Therefore, we raise Theorem 1 satisfying the basic requirement of full randomization. However, based on our aforementioned analysis, we need to restrict the randomization to maintain RBL and performance. From blocked matrix M, it is obvious that the sub-matrix E influences the randomization of Row_Index. To maintain RBL, we constrain the variations of row indexes. The form of matrix M can not be directly utilized to preserve RBL. We narrow down the range of M and derive Theorem 2.

Theorem 2: Matrix A and E are invertible, then matrix P in (4.10) is invertible, $\Leftrightarrow K - GA^{-1}C$ is invertible.

$$P = \begin{pmatrix} A & 0 & C \\ 0 & E & 0 \\ G & 0 & K \end{pmatrix} \tag{4.10}$$

where C and G are r×c and c×r matrices, respectively. A, E, and K are r×r, b×b, and c×c matrices, and are either identity matrices or matrices of the form as shown by (4.11)

$$\begin{pmatrix} 0 & 1 & 0 & . & . & . & . & 0 \\ 0 & 0 & 1 & . & . & . & . & 0 \\ 0 & . & . & . & 1 & . & . & 0 \\ 0 & . & . & . & 0 & . & 1 & 0 \\ \alpha_1 & \alpha_2 & . & . & . & . & \alpha_{n-1} & \alpha_n \end{pmatrix} \tag{4.11}$$

The diagonal elements of (4.11) are all zeros except the last one. The cross line behind it has all ones. The last row (α_1, α_2, ..., α_{n-1}, α_n) has all non-zero values. Therefore, all the row and column vectors are linearly independent of each other. Thus, this form of matrices is guaranteed to be invertible.

Theorem 2 offers a distinct strategy of preserving RBL and constraining randomized addresses by adding four zero matrices. We replace matrices B, D, and F with zero matrices, then matrix M becomes matrix P. When A, E, and K are set to identity matrices and C and G are set to zero matrices, P does not take any effect in randomizing memory addresses. A, E, and K can not be set to zero matrices, only to the form of equation (4.11). C and G randomize the address spaces of banks and columns.

We can then use this bijective matrix P for partial randomization of memory addresses. Suppose one set of consecutive bits needs to be excluded from randomization, the corresponding submatrix can be set to the identity matrix. Otherwise, we use square matrices as shown by (4.11). Suppose the local memory address from the r^{th} bit to the $(r+b)^{th}$ bit in address space is required to keep unchanged, and all the other bits need to be randomized. In this case, matrix E is a $b \times b$ identity matrix or form of (4.11). A and K also have the form of (4.11). C and G can be set according to Theorem 2 to ensure the whole matrix is invertible.

B. Variance Analysis

We further analyze the variation of memory addresses after the partial randomization by the bijective matrix. We define a variance metric to measure the dispersion degree of random addresses. Suppose a random variable X representing a random memory address, we can construct a vector series $\{x_i; i \in \mathbb{N}\}$ as a distribution of its transformed addresses, where $\mathbb{N}$ denotes a natural number. The expected value of X is $\mu = E[X]$. We then need to formulate a divergent and unstationary process. Hereby, we denote $x_i = \det X_i$. The randomized address can use the above block matrix such that $Y = P \times X$, where Y is the randomized address vector after using matrix (4.10), P is one instance of (4.10) and X is the unchanged vector.

Upon a large number of memory writes, the total write counts of a cache line can be approximated as a Gaussian Distribution. By the Central Limit Theorem [25], we can denote $X \sim N(\mu, \sigma^2)$. After a linear multiplication of matrices, the mean has changed to $E[Y]$.

Let $X = \left(x_1, x_2, \cdots, x_n\right)^T, Y = \left(y_1, y_2, \cdots, y_n\right)^T$, a_{ij} is an element of matrix $P_{n\times n}$. After multiplying with matrix $P_{n\times n}$, compute the mean and variance of Y,

$$E(y_i) = \sum_{j=0}^{n} a_{ij} E(x_j) \tag{4.12}$$

Similarly, we have

$$Var(y_i) = \sum_{j=0}^{n} a_{ij} Var(x_j) \tag{4.13}$$

We simplify the procedure of computation of address values, using the above pre-conditions in (4.12) and (4.13), we define the metric of transformation of address variables as below (4.14)

$$Y_{10} = \sum_{i=1}^{n} y_i 2^{i-1} \tag{4.14}$$

where Y_{10} represents the decimal value of address variable. Since random variable X follows a Gaussian Distribution approximately with large writes, $x_1, x_2, \cdots, x_n$ are independent normal random variables, then the sum of them also follows Gaussian Distribution. Therefore, the variance of decimal value of (4.14) can be calculated as below,

$$Var\left(Y\right) = \sum_{i=1}^{n} \left[\left(2^{i-1}\right) \sum_{j=0}^{n} a_{ij} Var\left(x_j\right) \right]^2 \tag{4.15}$$

Similarly, we have

$$Var\left(X\right) = \sum_{i=1}^{n} \left[\left(2^{i-1}\right) Var\left(x_i\right) \right]^2 \tag{4.16}$$

Because of invertible matrix P, there must be none-zero, i.e., at least one a_{ij} either in each row or in each column. From (4.15) and (4.16),

$$\begin{aligned} & Var\left(Y\right) - Var\left(X\right) \\ &= \sum_{i=1}^{n} \left[\left(2^{i-1}\right) \sum_{j=0}^{n} a_{ij} Var\left(x_j\right) \right]^2 - \sum_{i=1}^{n} \left[\left(2^{i-1}\right) Var\left(x_i\right) \right]^2 \\ &= \sum_{i=1}^{n} \left\{ \left(2^{i-1}\right)^2 \left(\sum_{j=0}^{n} a_{ij} Var\left(x_j\right) + Var\left(x_i\right) \right) \right. \\ & \left. \left(\sum_{j=0}^{n} a_{ij} Var\left(x_j\right) - Var\left(x_i\right) \right) \right\} \geq 0 \end{aligned} \tag{4.17}$$

Therefore, we get $Var\left(Y\right) \geq Var\left(X\right)$.

The variance of random variable X shows the address variation after using partial randomization. Furthermore, different partial matrices will reflect different degree of how far a set of addresses is spread out. Comparing two invertible matrices, we define variable $Y^{'}, Y^{''}$ as randomized address variables by multiplying matrices B and C, respectively. Δ denotes the difference of their variances:

$$\begin{aligned} \Delta &= Var\left(Y^{'}\right) - Var\left(Y^{''}\right) \\ &= \sum_{i=1}^{n} \left[\left(2^{i-1}\right) \sum_{j=0}^{n} b_{ij} Var\left(x_j\right) \right]^2 - \sum_{i=1}^{n} \left[\left(2^{i-1}\right) \sum_{j=0}^{n} c_{ij} Var\left(x_j\right) \right]^2 \\ &= \sum_{i=1}^{n} \left\{ \left(2^{i-1}\right)^2 \left(\sum_{j=0}^{n} b_{ij} Var\left(x_j\right) + \sum_{j=0}^{n} c_{ij} Var\left(x_j\right) \right) \right. \\ & \left. \left(\sum_{j=0}^{n} b_{ij} Var\left(x_j\right) - \sum_{j=0}^{n} c_{ij} Var\left(x_j\right) \right) \right\} \\ &= \sum_{i=1}^{n} \left(2^{i-1}\right)^2 \left(\sum_{j=0}^{n} \left(b_{ij} + c_{ij}\right) \left(b_{ij} - c_{ij}\right) Var\left(x_j\right)^2 \right) \end{aligned} \tag{4.18}$$

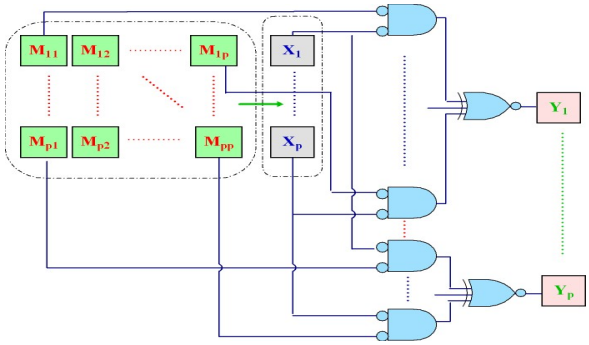


Fig. 6: Implementation for the Partial Randomization

When multiplying different invertible matrices to the address vectors, we need to discuss whether Δ is positive or negative due to the uncertainty of differences between two matrices. We can reasonably assume that there are some zero blocked matrices in a bijective matrix C, the number of ones in matrix B is more than or equal to that of C. Hence $\Delta \geq 0$. This means that the address variable is not only sporadic but also very well centralized.

C. Hardware Implementation

In practice, this transformation scheme could be done using logic circuits. Multiplication and addition operations can be substituted by an AND gate and a NOR gate. Fig. 6 gives a demonstration of a possible implementation. M is the operation matrix, X is the input address and Y is the mapped address. Y_1 is the first bit of the output address, which equals to the inner product of X and first row of M; Y_2 is equal to the inner product of X and the second row of M, etc. The elements in M can be either constant digital logic values or programmable values that have associated registers operated by a microcontroller. Using this solution, the proposed mapping and randomization can be implemented by hardware.

For a memory system with $r + b + c$ bits address space, computing each bit requires $2(r+b+c)$ AND gates and two-input NOR gates. Thus, the total storage overhead is $(r+b+c)^2$ and approximately $2(r + b + c)^2$ gates for logic. The extra latency is less than one cycle.

V. Row Buffer Locality-Aware Rotation

Section IV has introduced an address transform scheme that preserves RBL. In this section, we further introduce a novel RBL-aware Rotation (RAR) algorithm to minimize the performance impact of rotation-based wear leveling algorithms.

The randomization used by Start-Gap can destroy RBL while our bijective matrix based partial randomization solves this issue. Nevertheless, the overhead caused by rotation is non-trivial. As evaluated in [34], the cache line movement of Start-Gap can degrade the memory performance by 2% on average and up to 7%. Furthermore, the data invalidation scheme introduced in [34] is not applicable to GPU device memory because such data cannot be discarded like cache blocks. Therefore, we propose a novel rotation algorithm based on several observations: 1) rotation requests are not as urgent

Algorithm 1 RBL-Aware Opportunistic Rotation

```
1: while TRUE do
2:   Randomize(Addr);
3:   if WriteRequest then
4:     Writecounts ← Writecounts + 1;
5:   end if
6:   if Writecounts ≥ Interval then
7:     if Mem_busy then
8:       T_batch ← T_batch + 1;
9:       RTQ ← Push(Addr);
10:    else
11:      Rotate Current Line;
12:    end if
13:    Writecounts ← 0;
14:  end if
15:  if (RTQ.length ≥ RTTh ∧ !Mem_busy) ∨
     RTQ.isFull then
16:    //Rotate the bank by T_batch lines
17:    while T_batch ≥ 1 do
18:      Rotate one line;
19:      T_batch ← T_batch − 1;
20:    end while
21:  end if
22: end while
```

as front-end memory requests, which means that they can be delayed; 2) rotation requests may destroy the RBL of ongoing sequential requests if they are not well scheduled; and 3) by aggregating several discrete rotation requests together, such rotation requests can show good RBL, thereby reducing the degradation to system performance.

To mitigate the performance overhead caused by rotations in wear-leveling, we introduce an opportunistic and batched rotation algorithm. As shown in Algorithm 1, the addresses of memory accesses are randomized by partial randomization function, which converts virtual addresses to physical addresses. First, we activate rotation only when the rotation queue is full or the memory is idle, in which way the cache line movement will be delayed and the RBL of normal accesses will not be interrupted by rotations. Second, we enable opportunistic row rotation, which means that rotation is started only when the memory is not very busy, i.e., when the number of entries in the queue is smaller than a threshold. Third, we enable batched rotation to accumulate good RBL. With these three features, the overhead of wear-leveling rotations will be mostly hidden, thus not degrading the system performance.

In Algorithm 1, $Interval$ is a configurable parameter, and we adopt a commonly used value of 100 writes. $RTTh$ denotes the threshold of rotation request number in the rotation queue. Memory status (Mem_busy) is determined by counting the number of pending requests in the memory request queue. If the memory is currently busy, $Addr$ is put into the rotation queue, which means delaying the rotation instruction and continues the execution of normal requests (Lines 7-9). When the memory is idle and there is a rotation operation that needs to be executed, the rotation operation can be issued

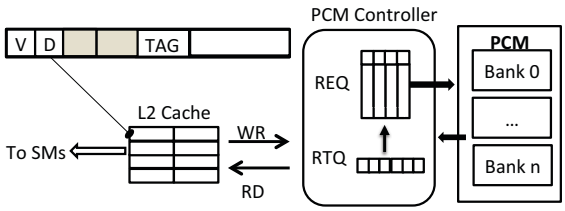


Fig. 7: Implementation of RBL-Aware Rotation

immediately (Line 11). When the rotation queue has enough pending rotation requests (i.e., larger than $RTTh$) and the memory is idle, the memory scheduler can execute these delayed rotation operations in a batch manner (Lines 15-21). In case the rotation queue is full, rotation requests are prioritized.

Fig. 7 depicts the implementation of our RAR technique. In the memory controller, one new component - RoTation Queue (RTQ) is introduced to manage the scheduling of rotation requests. To identify the normal requests and rotation requests in the Request Queue (REQ), we added a 1-bit flag to each request entry of the request queue. To implement RAR, adding several registers to monitor the status of memory systems is necessary. For each bank, a start register, a gap register and a counter register are needed. In addition, another counter Mem_busy is needed for monitoring the realtime RBL of current accesses. In total, we have 16 banks per chip and each bank needs around 5 bytes, including registers and monitor flags. Therefore, our algorithm will have a storage overhead of less than 100 bytes. In addition, the RTQ has a write port and a read port, and is implemented by a 256B SRAM. Each of the 16 banks has 4 entries, which can support up to 400 writes per bank. The GPU memory address space has a width of 40 bits. We ignore its 8 burst bits and accept the 32-bit useful address. By using the Cacti tool [7] to estimate the hardware cost (using 45nm technology), we find that the overall area usage is 0.023 mm^2. In sum, RAR incurs overhead at an acceptable level.

VI. Performance Evaluation

In this section, we evaluate the effectiveness of our bijective matrix based locality preserving policy and RBL-aware wear-leveling technique. We denote Start-Gap (including both RIB and rotation) as SG [22], the bijective matrix partial randomization as BJM, and BJM with RBL-aware rotation as BJM+RAR. In the rest of this section, we first introduce the endurance results and then the performance results including IPC, RBL and memory access latency. Then, we analyze variations of RBL along with data size and performance. Finally, we demonstrate the energy consumption results.

A. Endurance

Table IV shows the bank-level write skew for BAM, SG, BJM and BJM+RAR for different benchmarks. We use the bank-level metric as shown by (3.1). The worn-out banks can impact the lifetime of the entire GPU memory. Due to its RBL preservation, BJM+RAR achieves better write uniformity (lower bank-level write skews) than BAM and SG. The improvement is especially significant for two benchmarks BFS and SC. While SG is able to balance most bank writes,

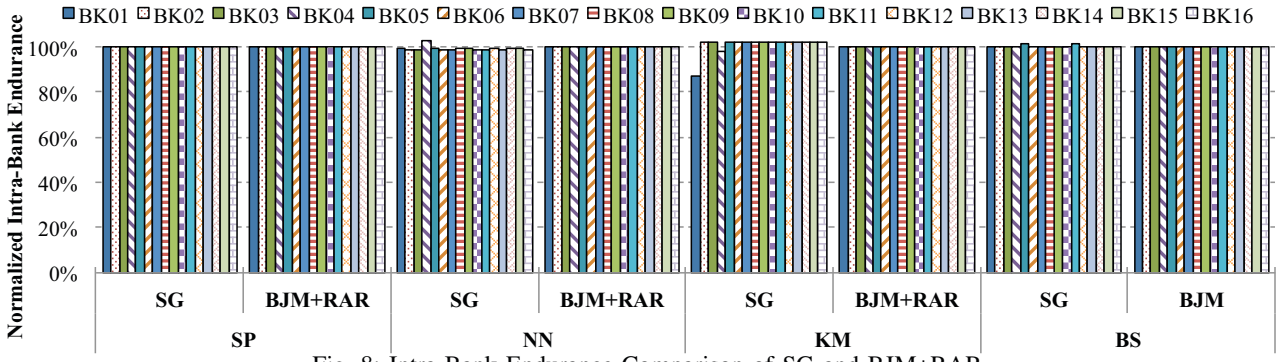


Fig. 8: Intra-Bank Endurance Comparison of SG and BJM+RAR

TABLE IV: Bank-level Write Skews of Different Schemes.

	SP	NN	KM	FWT	SN	CS	BFS	BS	SC	MT	GMean
BAM	1.03	1.03	1.14	1.15	1.18	1.06	2.23	1.03	3.67	1.04	1.31
SG	1.01	1.01	1.02	1.03	1.05	1.01	2.69	1.04	3.27	1.03	1.26
BJM	1.01	1.01	1.02	1.04	1.04	1.02	1.64	1.00	1.70	1.03	1.12
BJM+RAR	1.01	1.01	1.01	1.02	1.02	1.01	1.71	1.01	1.45	1.02	1.10

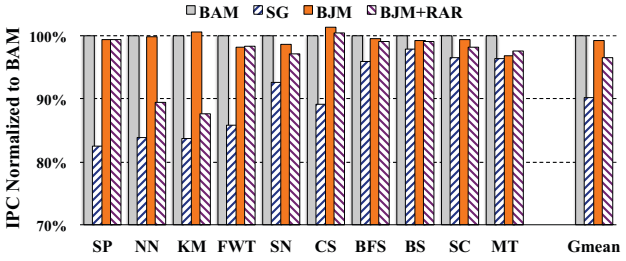


Fig. 9: IPC comparisons of different schemes

our techniques further decrease the inter-bank skews for prolonging the whole memory's lifetime.

Fig. 8 further shows the variation of intra-bank endurance for 4 different benchmarks: SP, NN, KM, and BS. The calculation of intra-bank endurance metric has a similar format as (3.1), but at the finer-grained cache line level. As we can see, for the 16 memory banks, SG incurs intra-bank endurance variations for NN and BS. However, our technique BJM+RAR achieves uniform bank endurance for all benchmarks as well.

B. Performance

Fig. 9 shows the IPC performance of BAM, SG, BJM and BJM+RAR for different benchmarks (normalized to BAM). On average (Geometric Means), SG, BJM and BJM+RAR achieve 90%, 99% and 97% of the IPC performance of BAM, respectively. We observe that BJM leads to better write endurance through partial randomization while causing only 1% performance loss to BAM. Note that both SG and BJM+RAR have rotation enabled for wear leveling. We can see that BJM+RAR leads to 8% performance improvement compared to SG due to the combined benefit of RBL preservation from BJM and the batched rotation from RAR.

The IPC improvement mainly attributes to the better memory access performance from improved RBL. Fig. 10 shows RBL variations when choosing different schemes. We can see that SG causes a dramatic drop of RBL (66% lower than BAM) because both the rotation and randomization reduce the

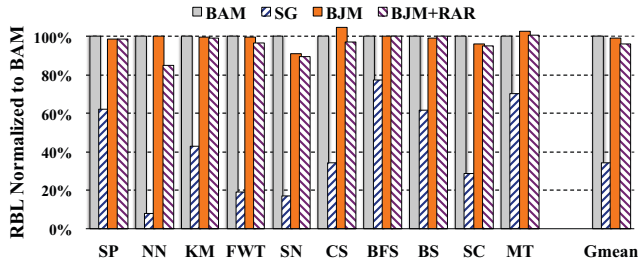


Fig. 10: Row buffer locality comparisons of different schemes

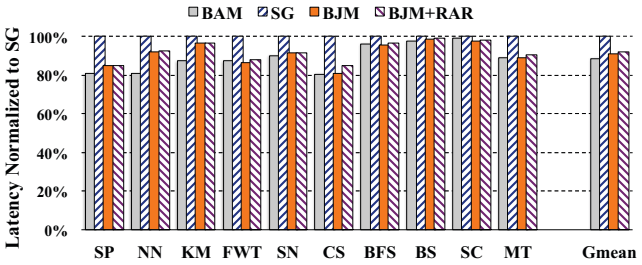


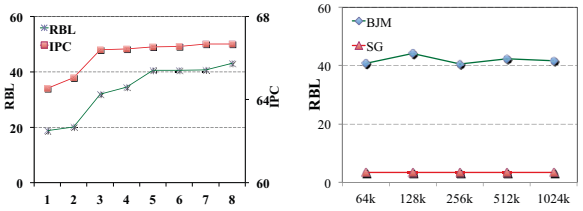
Fig. 11: Memory access latencies in different schemes

memory access sequentiality. However, BJM mitigates such problem using bijective matrix based partial randomization. It conserves 99% of RBL compared to BAM. BJM+RAR preserves as much as 96% of RBL because of RAR algorithm.

Fig. 11 shows memory access latency for BAM, SG, BJM and BJM+RAR (normalized to SG). BAM has the lowest access latency because no rotation is enabled. SG demonstrates the highest latency because of the lost RBL and wear leveling rotation. In contrast, BJM and BJM+RAR cause minimal performance loss compared to BAM and achieve 8%-9% faster memory access compared to SG. This is due to locality preservation features in our techniques.

C. RBL analysis

To gain more insights into the importance of selecting a proper matrix P as defined in (4.10) for BJM, we use the NN benchmark as an example and show the impacts of different matrices P on the RBL and IPC of NN in Fig. 12(a). Specifically, we chose eight matrices according to the theory introduced in (4.10). The x-axis denotes the eight different matrices used for BJM and we can observe how they affect the RBL (the left y-axis) and IPC performance (right y-axis)



(a) RBL and IPC variations when different matrices P are used

(b) RBL variation of BJM and SG with different data sizes

Fig. 12: IPC and RBL variations of NN benchmark

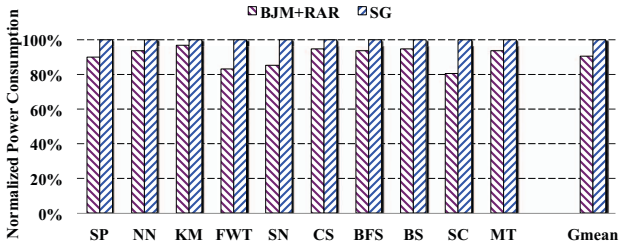


Fig. 13: Power consumption compared with SG

of the NN benchmark, respectively. In general, a matrix P can achieve good performance when the distribution of address variables is controlled in a reasonable range and measurably dispersed. Furthermore, other empirical experiments are conducted to find a matrix among the eight that can achieve the best overall performance for all the benchmarks. In addition, we also observe that the enhancement on RBL directly leads to performance improvement on IPC.

To investigate how RBL varies with an application's input data sizes, we configure the NN benchmark with five different input sizes (64KB, 128KB, 256KB, 512KB, 1024KB) and evaluate for two cases: SG and BJM in Fig. 12(b). As we can see, for both schemes, RBL does not vary much with the increasing of input data sizes. We observe similar trends for other benchmarks and omit the discussion for succinctness. This figure demonstrates that for a benchmark with different input sizes, BJM shows consistent RBL improvement over SG.

D. Energy Consumption

Our BJM+RAR scheme not only improves the performance and balances the inter-bank lifetime, but also reduces the energy consumption compared to the baseline technique. We calculate the dynamic power consumption as the operation energy plus background energy divided by execution time [13]. Fig. 13 shows that BJM+RAR can averagely save 8% power consumption compared to SG. To be specific, BJM+RAR brings different improvements over SG on power consumption for these benchmarks with diversified access patterns. The main reason for such advantage is that BJM+RAR improves the access locality through efficient bijective matrix-based address mapping and reduces rotation overheads by a RBL-aware rotation scheme, which together contribute to the energy consumption savings.

VII. Related Work

A. Wear Leveling for PCM

Qureshi et al. [23] proposed to uniformly distribute writes in the average case by organizing data as rotating lines in a page. Seong et al. [26] used a dynamically randomized address mapping scheme that swaps data using random keys to prevent adversaries. Zhou et al. [36] proposed a wear-leveling mechanism that integrates two techniques at different granularity: a fine-grained row shifting mechanism and a coarse-grained segment swapping mechanism. Their work suffers from the storage overhead of address mapping units and the time overhead of periodical sorting to pick up appropriate segments for swapping. Ipek et al. [8] proposed a solution to improve the lifetime of PCM by replicating a single physical memory page over two faulty, otherwise unusable PCM pages.

Ferreira et al. [5] proposed three schemes to manage PCM-based main memory: minimizing writeback traffic via cache replacement policies, avoiding unnecessary writes of clean bits, and swapping logical pages with idle regions. Kong and Zhou [10] argued that using non-volatile technology as main memory imposed the need to encrypt the data on the memory devices since the data would be resilient on the device for many years without power. Wang et al. [34] studied the wear leveling techniques for PCM-based caches. In their work, the authors proposed an invalidation based scheme that discards dirty cache blocks to avoid unbalanced write traffic. Our work is distinguished from these prior efforts in that we have identified the locality issue for both performance and endurance improvement.

B. Row Buffer Locality Aware Techniques

RBL has been widely investigated in memory request scheduling algorithms, due to its triple impacts on reducing memory service latency, increasing throughput, and saving energy costs. FR-FCFS algorithm [24] pioneered to prioritize the requests that could hit in row buffer, and later this concept has been extensively exploited in [9, 12, 18, 19] to improve both performance and fairness in multi-threaded environments. Stuecheli et al. [27] proposed to speculatively schedule write-backs from last level cache to form longer bursts of writes. Sudan et al. [28] proposed to co-locate hot data blocks into the same row via virtual address mapping of micro-pages. Muralidhara et al. [17] proposed to disperse a large number of data into different memory channels based on memory intensity and dynamically observed RBL.

Yoon et al. [35] proposed to map data that might incur frequent row buffer misses into DRAM while mapping data that might exhibit high RBL into PCM in a hybrid memory setting. Wang et al. [32, 29, 31] proposed a hybrid GPU global memory design that can leverage RBL to guide dynamic data migrations. The PCM device architecture could be optimized to have separate sensing and buffering circuitry, thus providing the flexibility to build multiple smaller row buffers in the same memory bank [11, 16] with limited area overhead. In general, better RBL could be achieved via multiple row buffers, which

have larger potential to selectively buffer hot data chunks at finer granularity. To the best of our knowledge, we are among the first to explore the RBL issue in PCM wear leveling.

VIII. Conclusion

In this paper, we have revealed the issue of row buffer locality loss in wear-leveling techniques for PCM devices, examined several address transformation schemes, and empirically analyzed their impacts on performance and row buffer locality. We propose several row buffer locality preserving techniques for PCM wear-leveling under massive parallelism so as to balance the memory access performance and the endurance of PCM-based GPU global memory. Accordingly, we have introduced a bijective matrix based locality preservation policy and an RBL-aware rotation algorithm. We have measured the performance of our techniques using a GPGPU simulator for a diverse set of GPU benchmarks. Our experimental results demonstrate that, while providing comparable write endurance, our techniques can preserve row buffer locality and improve the IPC performance of benchmarks by as much as 8% compared to existing wear-leveling techniques such as Start-Gap. We also achieve steady energy consumption savings compared to Start-Gap.

Acknowledgments
This research was supported in part by an Alabama Innovation Award and the National Science Foundation awards 1059376, 1320016, 1340947 and 1432892.

References

[1] GPGPU-Sim GTX480 Configuration. https://github.com/gpgpu-sim/gpgpusim_distribution/tree/master/configs/GTX480.
[2] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: A Prototype Phase Change Memory Storage Array. In *HotStorage*, 2011.
[3] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *ISPASS*, 2009.
[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.
[5] A. P. Ferreira, M. Zhou, S. Bock, B. R. Childers, R. G. Melhem, and D. Mossé. Increasing PCM Main Memory Lifetime. In *DATE*, 2010.
[6] D. Hankerson, G. Hoffman, D. Leonard, C. C. Lindner, K. Phelps, C. Rodger, and J. Wall. *Coding Theory and Cryptography: The Essentials*. CRC Press, 2000.
[7] HP. Cacti. http://www.hpl.hp.com/research/cacti.
[8] E. Ipek, J. Condit, E. B. Nightingale, D. Burger, and T. Moscibroda. Dynamically Replicated Memory: Building Reliable Systems from Nanoscale Resistive Memories. In *ASPLOS*, 2010.
[9] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers. In *HPCA*, 2010.
[10] J. Kong and H. Zhou. Improving Privacy and Lifetime of PCM-based Main Memory. In *DSN*, 2010.
[11] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *ISCA*, 2009.
[12] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-aware DRAM controllers. In *MICRO*, 2008.
[13] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *ISCA*, 2013.
[14] D. Liu, T. Wang, Y. Wang, Z. Shao, Q. Zhuge, and E. Sha. Curling-PCM: Application-Specific Wear Leveling for Phase Change Memory based Embedded Systems. In *ASP-DAC*, 2013.
[15] Z. Liu, B. Wang, P. Carpenter, D. Li, J. S. Vetter, and W. Yu. PCM-Based Durable Write Cache for Fast Disk I/O. In *MASCOTS*, 2012.
[16] J. Meza, J. Li, and O. Mutlu. Evaluating Row Buffer Locality in Future Non-Volatile Main Memories. Technical report, Carnegie Mellon University, 2012.
[17] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. T. Kandemir, and T. Moscibroda. Reducing Memory Interference in Multicore Systems via Application-aware Memory Channel Partitioning. In *MICRO*, 2011.
[18] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO*, 2007.
[19] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA*, 2008.
[20] NVIDIA. Fermi: NVIDIA's Next Generation CUDA Compute Architecture. http://www.nvidia.com/fermi.
[21] NVIDIA. CUDA C/C++ Code Samples, 2011.
[22] M. K. Qureshi, J. P. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing Lifetime and Security of PCM-Based Main Memory with Start-Gap Wear Leveling. In *MICRO*, 2009.
[23] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable High-Performance Main Memory System Using Phase-Change Memory Technology. In *ISCA*, 2009.
[24] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. ISCA, 2000.
[25] S. Ross. *A First Course in Probability*. Pearson Prentice Hall, eighth edition, 2009.
[26] N. H. Seong, D. H. Woo, and H.-H. S. Lee. Security Refresh: Prevent Malicious Wear-out and Increase Durability for Phase-Change Memory with Dynamically Randomized Address Mapping. In *ISCA*, 2010.
[27] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John. The Virtual Write Queue: Coordinating DRAM and Last-Level Cache Policies. In *ISCA*, 2010.
[28] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis. Micro-Pages: Increasing DRAM Efficiency with Locality-Aware Data Placement. In *ASPLOS*, 2010.
[29] B. Wang, Y. Jiao, W. Yu, X. Shen, D. Li, and J. S. Vetter. A Versatile Performance and Energy Simulation Tool for Composite GPU Global Memory. In *MASCOTS*, 2013.
[30] B. Wang, Z. Liu, X. Wang, and W. Yu. Eliminating intra-warp conflict misses in GPU. In *DATE*, 2015.
[31] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. S. Vetter. Exploring Hybrid Memory for GPU Energy Efficiency Through Software-hardware Co-design. In *PACT*, 2013.
[32] B. Wang and W. Yu. Performance and Power Simulation for Versatile GPGPU Global Memory. In *IPDPS Workshops*, 2013.
[33] B. Wang, W. Yu, X.-H. Sun, and X. Wang. DaCache: Memory Divergence-Aware GPU Cache Management. In *ICS*, 2015.
[34] J. Wang, X. Dong, Y. Xie, and N. P. Jouppi. i^2WAP: Improving Non-Volatile Cache Lifetime by Reducing Inter- and Intra-Set Write Variations. In *HPCA*, 2013.
[35] H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu. Row Buffer Locality Aware Caching Policies for Hybrid Memories. In *ICCD*, 2012.
[36] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *ISCA*, 2009.