Exploring Hybrid Memory for GPU Energy Efficiency through Software-Hardware Co-Design

Bin Wang* Bo Wu† Dong Li‡ Xipeng Shen† Weikuan Yu* Yizheng Jiao* Jeffrey S. Vetter‡

Auburn University* College of William & Mary† Oak Ridge National Lab‡
{bwang,wkyu,yzj0018}@auburn.edu {bwu,xshen}@cs.wm.edu {lid1,vetter}@ornl.gov

***Abstract*—Hybrid memory designs, such as DRAM plus Phase Change Memory (PCM), have shown some promise for alleviating power and density issues faced by traditional memory systems. But previous studies have concentrated on CPU systems with a modest level of parallelism. This work studies the problem in a massively parallel setting. Specifically, it investigates the special implications to hybrid memory imposed by the massive parallelism in GPU. It empirically shows that, contrary to promising results demonstrated for CPU, previous designs of PCM-based hybrid memory result in significant degradation to the energy efficiency of GPU. It reveals that the fundamental reason comes from a multi-facet mismatch between those designs and the massive parallelism in GPU. It presents a solution that centers around a close cooperation between compiler-directed data placement and hardware-assisted runtime adaptation. The co-design approach helps tap into the full potential of hybrid memory for GPU without requiring dramatic hardware changes over previous designs, yielding 6% and 49% energy saving on average compared to pure DRAM and pure PCM respectively, and keeping performance loss less than 2%.**

***Index Terms*—GPU; NVRAM; Co-Design; Energy Efficiency**

I. Introduction

The main memory has been a major efficiency bottleneck for modern computer systems. The power consumption by the main memory in high-end servers is already about 30%-50% of the system power [2], [11], [31]. To support the increasing working sets of many concurrently executing threads, the density scaling (capacity per unit area) of the current dominant main memory technique (i.e., DRAM) is close to a point where the manufacturing costs will be unacceptably high [28]. Therefore researchers are looking for innovations in memory materials and architectures.

An actively pursued approach is to exploit hybrid memory systems [26], [25], [33], [35], [14], [7], [12], [21], [24], [19]. By combining DRAM with non-volatile memory (NVM), hybrid memory can leverage the best characteristics of multiple types of memory. Although NVM is more power-efficient, denser, and cheaper, it is usually characterized with longer write access latency, higher write energy, and limited endurance. This raises a question on how to optimize data placement in hybrid memory to avoid performance loss and improve energy efficiency. Previous work concentrates on CPU that has a modest level of parallelism. Several important questions remain open: What are the implications of massive parallelism to the usage of NVM? How can we reconcile them to maximize energy efficiency? Can the previous findings apply to GPU-like massively parallel architectures?

This paper presents a systematic investigation into these open questions. Using Phase Changing Memory (PCM) as a representative NVM, we conduct a three-fold exploration. In the first part (Section III), we start with an empirical examination of several PCM use cases by following previous studies (on CPU). Contrary to the promising energy efficiency shown before, these designs cause up to 354% energy efficiency degradation compared to traditional DRAM-only memory. We conduct a detailed analysis and find that the main reason for this is the fundamental mismatch between these designs and the massive parallelism in GPU. The mismatch manifests as a large number of misses in the DRAM cache, the loss of effectiveness in using last level cache (LLC) miss rates as memory performance indicators, a large volume of data migrations between DRAM and PCM, and their severe interference to the memory bandwidth of applications.

In the second part of this exploration, we examine whether the issues can be addressed by complementing the previous designs with a set of hardware features for a better match with the massive parallelism. These features include a finer granularity for data management, the adoption of row-buffer misses as a direct indicator of memory performance, the proposal of a batch migration and a bandwidth-aware scheme for mitigating the interference caused by data migrations, and so on. Together they make *parallelism-conscious data migration* possible. When being integrated into a state-of-the-art hybrid memory design [26], these techniques help improve the energy efficiency of several benchmarks substantially. But the overall results are still inferior on some programs compared to traditional DRAM. A further analysis shows a key weakness of the pure hardware support. It always starts with a default data placement because as it has no knowledge of program access patterns and does not know which placement is optimal. For some programs, the initial placement can trigger too much data migration for the hardware mechanism to handle.

Prompted by these observations, in the third part of this study (Section V), we develop a compiler support along with a novel data-placement algorithm. By exploiting the unique features of GPU computing and a GPU energy-performance model, the dynamic programming-based algorithm manages to create a much enhanced initial data placement in memory for a set of GPU programs. Working hand-in-hand with hardware migration support, this new placement strategy saves energy by 6% and 49% on average, respectively, compared to pure DRAM and pure PCM systems. The performance loss is kept at less than 2% (Section VII).

Overall, this study demonstrates the promise of the co-design approach for reconciling hybrid memory with massive parallelism. The approach requires no dramatic changes to existing hybrid memory designs, hence avoiding additional

978 -1-4799-1021-2/13/$31.00 ©2013 IEEE

uncertainties in complexity and feasibility. To the best of our knowledge, this work is the first that exploits systematic compiler-hardware cooperation to match hybrid memory with GPU massive parallelism. As processor-level parallelism keeps increasing, the insights may be applicable to a broad range of studies in the design of future memory systems.

This study makes the following major contributions:

- It examines the mismatch between existing hybrid memory designs and GPU massive parallelism, and introduces a set of hardware features such as *Parallelism-Conscious Migration* to enhance previous designs.
- It introduces a novel representation, *Placement Cost Graph*, to capture placement constraints, and adopts a hardware-software co-design approach with complete compiler and runtime support to assisting data placement on hybrid memory for GPU applications.
- This work provides the first principled understanding in the relation between NVM and GPU massive parallelism, and contributes some key insights for exploiting NVM for GPU computing; the findings may open many new opportunities for bridging NVM and massively parallel architectures.

II. Background

A. GPU Global Memory

Memory in a computer system typically consists of a number of modules. Cell arrays within these modules are organized into rows and columns. Memory reads and writes are performed by selecting the location according to the row and column addresses. Data are then latched into the *row buffer*. The row buffer serves as a fast cache in memory devices. When data are to be evicted from a row buffer, in case of DRAM, they must be written back (via a precharge operation) to the DRAM cells. GPU global memory is typically GDDR3/5 (a type of high performance graphics DRAM) or DDR3 SDRAM. Figure 1 depicts the general architecture of GPU memory. Hundreds of GPU cores share the global memory, the most power-consuming component in GPU (up to 40% of the whole GPU power) [9].

Memory energy can be broken down into dynamic energy and static energy. The former includes activation/precharge, read/write, and termination energy; the latter includes background energy due to the peripheral circuitry, refreshing operations and transistor leakage. The major energy consumption is background and activation/precharge energy.

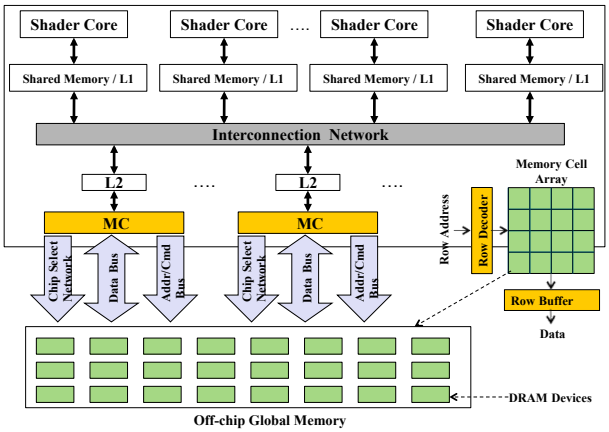

Fig. 1: GPU memory hierarchy

B. Phase Change Memory (PCM)

As an NVM technology, PCM saves energy due to the following reasons. First, due to its non-volatile PCM cells, memory cell leakage is reduced. PCM-based memory does not require refreshing logic and energy. Second, the peripheral circuits (e.g., row and column decoders, input/output network and sense amplifiers) can be powered down without losing content in during idle time [36], saving background energy. Third, with non-volatile PCM cells, only the dirty lines in a row buffer need to be written back to the PCM cell array [15], [36], while in the case of DRAM the entire row buffer needs to be written back to precharge memory cells no matter it is dirty or not. This saves dynamic energy.

PCM write latency and energy consumption are bigger than those of DRAM, because the process of heating the alloy until the change of resistance state is quite slow and energy-consuming. For read operations, PCM access latency is comparable to that of DRAM. For a burst of reads that hit the same bank of the memory, PCM is even slightly better [36] because of its non-destructive nature.

III. Initial Designs and Observations

We start our exploration with applying to GPU two representative hybrid memory designs previously proposed for CPU and a pure PCM design. The first design, named *Dbuff* [25], uses DRAM as a buffer of the PCM. To mitigate the write latency of PCM, it employs a write queue to enable lazy-write. The second design, named *RaPP* [26], organizes DRAM and PCM in a large flat memory and migrates pages between them. The memory controller is used to monitor the misses and write-backs in LLC for each memory frame (or page), ranks frames and dynamically adjusts their placement according to their "popularity". This design represents the state of the art in managing hybrid memory for multicore CPU.

We have made two modifications when applying these designs to GPU. First, GPU has very limited virtual memory support. We implement a page remapping table to facilitate accesses to migrated data in the RaPP design. Second, we find that separating DRAM and PCM in different partitions as done in the previous designs leads to inferior performance and energy efficiency on GPU. The reason is that the GPU massive parallelism and its typically uniform access patterns by all cores cause serious bank conflicts and access bottlenecks on some memory partitions (often the DRAM partitions). We hence adapt these two previous designs by spreading DRAM and PCM into every memory partition to maximize data parallelism. Each partition connects to one memory controller by two memory channels. Each channel consists of memory devices of the same type (DRAM or PCM).

For easy comparison, we put the detailed experiment configurations and measurement results in Sections VI and VII. The overall observation is that the PCM-only design is not appropriate for GPU: the long latency and energy costs of PCM write operations result in 32% lower energy efficiency and about 9% performance loss, compared to the traditional DRAM-only memory. More importantly, contrary to the significant benefits that have been observed by previous studies on multicore CPU, both hybrid memory designs give a much lower energy efficiency and performance on GPU. Dbuff incurs a 29% performance loss and a 87% lower energy

efficiency on average, and RaPP a 5% performance loss and a 11% lower energy efficiency.

Our analysis reveals that the big performance loss in Dbuff comes from the DRAM misses (whose ratio can be as much as 39%). The majority of these misses come from many slow and energy-consuming write accesses. The inherent nature of GPU computing causes such misses with its very high concurrency and data-level parallelism. For example, streaming access pattern of limited temporal data locality is very common in GPU kernels; cold misses at the beginning of kernel executions also affect the performance, especially for small GPU kernels.

The loss in the RaPP design is due to two reasons. First, RaPP uses LLC misses as a locality hint for making data migration decisions. However, unlike multi-core platforms, LLC misses on a GPU fail to serve as a reliable indicator. It shows a poor correlation with row-buffer misses, as Table IV shows. This disparity is due to the fact that GPU has a much smaller LLC, and it is shared by hundreds of cores. As a result, LLC misses largely reflect the conflicts among the accesses triggered by different cores rather than locality patterns. Second, we observe bursty data migrations on 4 out of 9 benchmarks. Such migrations consume up to 50% of the total memory channel bandwidth, causing severe interferences to the regular memory transactions. This is not surprising on a massively parallel platform, considering the concurrent accesses conducted by thousands of GPU threads.

Overall, neither of the two CPU-oriented designs matches well with the massive parallelism in GPU. We introduce new features into the RaPP design, and propose the coordination between hardware and software to solve those problems.

IV. Reconciliation through Hardware Features

In this section, we focus on an enhancement of the RaPP design given its better performance. Our strategy is to add new hardware features that can mitigate the interferences and maximize the benefits of data migration.

New Features: Our features re-examine some key questions—what, how and when to migrate, —in the context of GPU massive parallelism. (1) Compared to the initial design, we use a finer granularity for data migration. By avoiding the unnecessary migration of some data, it reduces the incurred bandwidth consumption. The catch is that it may result in a larger space overhead, which is addressed through a selective treatment as we will show. (2) We make data migration decisions based on row-buffer misses rather than LLC misses. As a direct indicator of each row's access performance, row-buffer misses provide more immediate hints. (3) We propose *batch migration*, which reduces frequency and overhead by aggregating data together for migration. (4) We introduce a bandwidth pressure-aware mechanism, which reduces the interferences caused by data migrations by adaptively controlling the intensity of data migrations. (5) Finally, with a conservative approach for space-efficient access monitoring, the new scheme reduces storage requirement so that the data structure for managing migrations can be put onto the GPU chip, lowering management time overhead. By integrating these new features into the multi-queue migration scheme used in the RaPP algorithm, we have built up a *parallelism-conscious migration scheme*.

Parallelism-Conscious Migration: Figure 2 shows the high-level view of the migration scheme. The scheme monitors and migrates data at the grain of 256-byte segment, rather than row buffer or memory page size as used in previous work. The fine granularity matches the GPU last level cache-line size, and saves migration energy by avoiding waste of memory channel bandwidth. A data segment is associated with a segment descriptor that records the segment index in current channel (*idx*), a position bit (*clr*: 0 for PCM and 1 for DRAM), expiration time (*exptm*), a reference count (*refcnt*), and a row buffer miss count (*rbmcnt*). The row buffer miss is monitored by the MC as done in the previous work [33]. When a data segment without descriptor is referenced, its corresponding descriptor is immediately created.

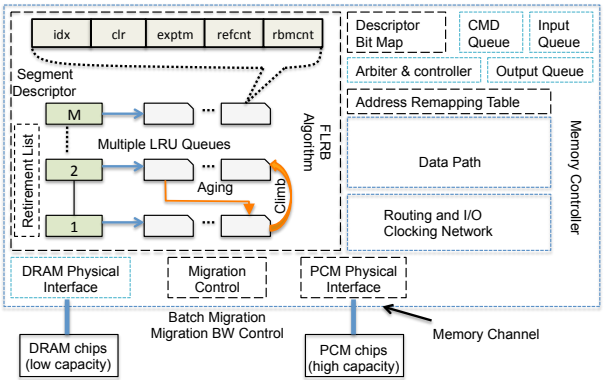

Fig. 2: Memory controller with data migration logics. The dotted black boxes depict added logics.

The kernel of the new migration scheme is an algorithm named *FLRB*, which manages the migration based on a rich set of information, including data access frequency, locality, recency, and bandwidth utilization. The attainment of access frequency and recency comes from the multi-queue organization of data segment descriptors, in a way similar to the previous design [26]. Data segment descriptors fall into M queues ($M = 8$ in our case), each of which is an LRU queue with the least recently used segment at the head position. The queue i ($0 \leq i < M$) stores the descriptors of the segments with reference count $\in \left[2^{i-1}, 2^i\right)$. We call the queue at i=0 the *low-end queue*, and the one at i=M-1 the *high-end queue*. The reference count calculation weighs PCM write access by 3 and other accesses by 1 in order to distinguish the latency difference between write and read operations. At each access, the MC increases the *refcnt* of the segments to be accessed, and promotes it to a higher-level queue if the *refcnt* exceeds the value range of the current queue.

The MC uses the *exptm* field of a descriptor to track the recency of accesses to a data segment. The field is assigned with an expiration time, which is the last accessed time plus a constant (150 in our cases). When expired, the descriptor is moved to the tail of the lower-level queue or evicted from the multi-queue. The MC checks the queues in a round-robin fashion. A data segment is selected as a candidate to migrate from PCM to DRAM, when its descriptor is in queue m and also suffers from n row buffer misses (m=3 and n=2, selected empirically). To manage free space, the MC tracks a pool of consecutive free space through pointers, and uses a buffer to record the indices of 50 recently freed memory segments

as their data have migrated to PCM. When a descriptor of a DRAM segment is evicted from the low-end queue, its corresponding data segment is copied back to the original place in PCM, and its corresponding entry in the address mapping table is cleaned for reuse. If DRAM runs out, the data segment can be proactively migrated to PCM despite its *exptm*. For those PCM data segments that are not migrated into DRAM, their segment descriptors are immediately released for reuse when evicted. We assume that the MC could put data evenly across the PCM area for wear leveling.

To improve row buffer spatial locality, we introduce a migration scheme named *batch migration*. Whenever a data segment is chosen as the migration candidate, the MC checks if there are any neighboring segments (i.e., physically within the same row) in the same queue as candidate segments. If so, they will be migrated together. Furthermore, since the DRAM region for migration is physically located in the reserved rows of each DRAM bank, and the MC could place the migrated segments into physically contiguous destination addresses to maximize data locality, which is similar to the scheme in [29].

In addition, we introduce bandwidth awareness into the migration to reduce the interference of data migration on performance. It limits the bandwidth consumption data migration to the available bandwidth. It calculates the bandwidth consumption (BW_{pre}) by the regular memory transactions in the last time quantum based on Equation 1, and then computes the bandwidth utilization (BW_util in Equation 2), based on which it derives the maximum migration data size ($MigrationDataSize$) in the next time quantum (Equation 3.) The time quantum is empirically selected as 1000 memory cycles.

$$BW_{pre} = NUM_{mem\_trans} * Trans\_Size/time\_quantum \quad (1)$$

$$PeakBW = DataBusWidth * Mem\_Clock * DDR\_multipiler,$$
($DDR\_multipiler$=2 for DDR2 and 8 for DDR3)

$$BW\_util = BW_{pre}/PeakBW \quad (2)$$

$$MigrationDataSize = (1-BW\_util)*PeakBW*time\_quantum \quad (3)$$

Space Overhead: The FLRB algorithm demands on-chip storage for a few data structures. A segment descriptor requires 64 bits (i.e., 22 bits for segment index, 1 bit for segment positioning, 31 bits for expiration time, 8 bits for access counting, and 2 bits for row buffer miss counting). If we maintain the segment descriptor for each data segment, the area size will be unacceptably large. For example, 1GB memory demands 32MB on-chip storage. We cap the descriptor storage to 32KB (i.e., 4K descriptors) so that only the descriptors of the most recently accessed data segments get into the multi-queue. This is based on our empirical observation that more than 60% data migration happens to less than 4KB data segments. Other overhead includes the remapping table (20KB) that stores descriptors in the multi-queue, two row buffer-sized migration buffers (16KB) that decouple data copying and address updating, and a 1KB retirement list for data migration to DRAM. Thus, the total space is 69KB per memory controller. Considering that the L2 cache (256KB per MC) in latest Tesla K20X and GeForce GTX TITAN takes about 1/8 area of the die, 69KB on-chip storage incurs 3.3% area overhead. We expect this overhead could be further reduced by using denser memory techniques (such as STT-RAM) and more efficient algorithms to manage these on-chip data structures.

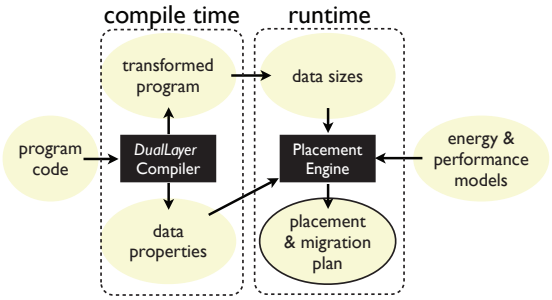

Fig. 3: Overview of the compiler and runtime software support.

Results and Discussions: We evaluate our hardware design, and the results (detailed in Section VII) show that, in comparison to the initial design in Section III, the row buffer miss rate is reduced by 11.8% and the migration rate by 15% because of the batch migration, locality-aware data placement, migration bandwidth control and other features. On the other hand, we observe about 20% a loss of energy efficiency. A detailed analysis reveals that the initial inferior placement of data causes a large volume of data migration. In both our study and previous studies [26], data are blindly placed onto the larger part of memory (PCM) initially. This is because hardware has little knowledge of the program and hence is hard to determine a good initial data placement. Without resorting to more complicated microarchitecture, we use a co-design approach to simplify the hardware design.

V. Compiler-Directed Data Placement

Prompted by the observations in the previous sections, we develop a compiler and runtime software support to help alleviate the initial data placement problem. Figure 3 offers an overview. At compile time, our compiler, named *DualLayer*, extracts some data properties, and puts some special function calls into the program. At runtime, a placement engine computes the appropriate data placement based on compile-time and runtime information. Although the technique is applicable to different data structures, we use array as the unit for placement for its common usage.

A. DualLayer *Compiler*

Our *DualLayer* compiler consists of two parts. The first part is an LLVM-based analyzer. It leverages the inter-procedural program analysis of LLVM (e.g., alias analysis and loop analysis) to derive kernel memory access patterns, including the number of reads and writes to each input array in all kernels. It then estimates the number of coalesced and uncoalesced memory transactions with each memory access instruction based on the extracted access patterns. It uses symbolic representations when the array size is runtime variables, which will be resolved at runtime (Section V-B.) The second part is an PTX code analyzer, which works on one kernel at a time. It extracts some low-level information of the kernels that the high-level compiler is hard to get precisely, including the numbers of total instructions and synchronizations that are needed by the energy and performance models (Section V-D.) The compiler inserts into the program some special function calls to trigger the runtime placement engine and actual data allocations as will be elaborated in Section V-C.

B. Placement Engine

The placement engine, when invoked at runtime, tries to determine a proper data placement and migration plan. It must deal with multi-fold complexities efficiently. First, the size of the two types of memory imposes constraints on the placement. Even if it is common that all data can fit into the combined memory system, they may not fit into one type of memory. Second, the best data placements may differ for different kernels. Data migrations between kernels may accommodate different requirements, but at the expense of extra time and energy costs. Conceptually, the main problem facing the placement engine is to find a suitable representation that captures all the constraints on data placement, and then come up with an efficient algorithm to solve them. We are unaware of any prior solution to such a problem. In fact, this placement problem itself has not been formally defined before.

Definition of Optimal Placement Problem on Hybrid Memory: Suppose that there are K kernels accessing A unique arrays. The sizes of the arrays are known; z_i denotes the size of the ith array. We can use an A-dimensional binary vector, $< v_1, v_2, \cdots, v_A >$, to represent a *data placement scheme* for a kernel, where v_i can have three possible values:

1 : the ith array occupies GPU PCM;
0 : the ith array occupies no GPU memory;
-1 : the ith array occupies GPU DRAM.

Let $V_i =< v_{i1}, v_{i2}, \cdots, v_{iA} >$ be the data placement scheme for the i'th kernel. The goal of the Optimal Placement Problem is to find a value assignment for all v_{ij} ($i = 1, 2, \cdots, K$, $j = 1, 2, \cdots, A$) so that the following two *size conditions* hold:

(1) $\sum_{j=1}^{A} v_{ij} * (1 + v_{ij}) * z_j/2 < C$;
(2) $\sum_{j=1}^{A} v_{ij} * (v_{ij} - 1) * z_j/2 < D$;
and the following *total cost* is minimized:

$$(\sum_{i=1}^{K} S_i(V_i)) + (\sum_{i=2}^{K} S_i(V_{i-1}, V_i)),$$

where C and D represent the sizes of PCM and DRAM respectively, $S_i(V_i)$ is the cost of the i'th kernel when the placement scheme V_i is used for that kernel, and $S_i(V_{i-1}, V_i)$ is the cost of migrations used to transform data placement from V_{i-1} to V_i (it equals zero when $V_{i-1} = V_i$.) The cost can be defined in various ways. We use Energy-Delay Product (EDP) in this work as will be explained in Section VI.

Placement Cost Graph: A key feature of our solution is the *Placement Cost Graph*, a novel representation that captures various constraints on data placement in a hybrid memory system. A Placement Cost Graph is an acyclic directed graph as illustrated in Figure 4. All its nodes, except two special ones (e.g., n_0 and n_9 in Figure 4), fall into K groups (assuming K GPU kernels), with the ith group corresponding to the set of legitimate (i.e., meeting the two size constraints (1) and (2)) data placements for the ith kernel ($i = 1, 2, \cdots, K$), one node per legitimate placement. In Figure 4, the placement scheme for each node is labeled in parentheses, and zero values in the vectors are not shown.

The nodes in the ith group are fully connected with all the nodes in the $(i-1)$th group (if $i > 0$) and with all the nodes in the $(i+1)$th group (if $i \leq K$.) The edges, pointing from the lower-numbered group to the higher-numbered one, represent all possible data migrations between two adjacent

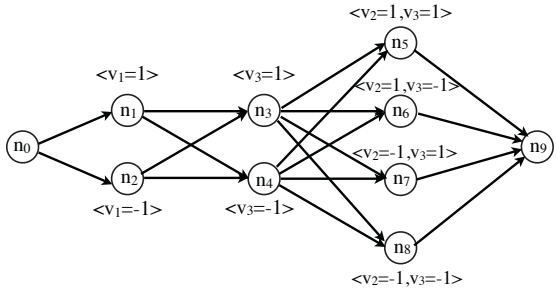

Fig. 4: A Placement Cost Graph of a program with three kernels. Edge weights are not shown for readability.

kernels; obviously, no connections are needed among nodes in the same group. Every edge in the graph carries a weight. The weight on the edge from node$_i$ in Group l to node$_j$ in Group $(l+1)$ (denoted as $w_{i \leftarrow j}$) equals the cost of the $(l+1)$th kernel when it uses node$_j$'s data placement, plus the cost of the data migrations needed for changing node$_i$'s data placement to node$_j$'s. In the notations given in the problem definition, the weight equals to $S_{l+1}(V_j) + S_{l+1}(V_i, V_j)$. For instance, the weight $w_{3 \rightarrow 6}$ is $S_3(< v_2 = 1, v_3 = -1 >) + S_3(< v_3 = 1 >, < v_2 = 1, v_3 = -1 >)$. The cost is estimated by putting the data access patterns into a prebuilt energy model (explained in Section V-D.)

Starting and ending nodes are two special nodes in a Placement Cost Graph. The former connects with all the nodes in the first group, with the edge that points to node$_x$ carrying the cost of the first kernel when it uses node$_x$'s data placement. For example, the weight $w_{0 \rightarrow 1}$ is $S_1(< v_1 = 1 >)$. The ending node is where all the nodes in the Kth group connect to; these connecting edges carry zero weight.

Dynamic Programming: By capturing both size constraints and the migration overhead, the Placement Cost Graph lays the foundation for solving the Optimal Cost Graph, and simplifies the placement problem into a shortest path problem. A path here refers to a route from the starting node to the ending node in the graph. The length of a path is the sum of the weights of all edges on the path. It is easy to see that the length equals the *total cost* when the data placements represented by the nodes on the path are used for the kernels. A shortest path in a Placement Cost Graph hence gives an optimal solution to the corresponding Optimal Placement Problem. We use the standard dynamic programming to solve the shortest path problem. Its time complexity is $O(mn)$, where m is the number of nodes and n is the number of edges in the graph. The running time is negligible for most GPU programs as they typically use less than ten arrays per kernel, which is confirmed by results in Section VII. In some extreme cases where the program contains many arrays and kernels, the algorithm can be applied offline based on profiling information (with some runtime backup plans as mentioned next).

C. Runtime Support

The runtime support is enabled by the compiler, which inserts the following special function calls into the application.

- void __getSize(int s1, int s2, ...);
- void __calPlace();
- void __mallocHyb(int kID, int aID, void **arr, int bytes);
- void __dataMigrate(int kID);
- void __finalize(void *array);

A call to "__getSize" is put before the first kernel call. It uses its parameters to pass the sizes of input arrays to an internal data structure "__dataSizes[]". Each parameter is either a variable in the application or a constant "-1", which means that the size is still unknown at the function invocation. A call to "__calPlace" invokes the placement engine to solve the shortest path problem by using the compiler's output on data properties, the "__dataSizes" array, and the energy model. If the size of any array is unknown (-1), it uses the largest known size of all arrays for it. It uses K A-bit vectors, "__place[][]", to record whether an array should be put on DRAM (bit=0) or PCM (bit=1) at the ith kernel ($i = 1, 2, \cdots, K$). The call to "__mallocHyb" allocates the array "arr" based on "__place[kID][aID]". Upon an allocation failure, if the target device is the small DRAM, it tries PCM before exiting with errors. A call to "__dataMigrate" is inserted right before every kernel call except for the first. It checks the differences between "__place[kID-1]" and "__place[kID]", and migrates an array if its corresponding bits differ in the two bit vectors. A call to "__Finalize" is put at the end of the program to inform the hardware to free the used resources.

D. Energy and Performance Models

This part describes the energy and performance models used by the placement engine to approximate the energy consumption and performance of a kernel to determine a data placement plan. The performance model is an extension of a previous GPU analytical timing model [8]. It estimates the execution time of a kernel based on MWP, CWP, the number of computation cycles and the number of memory waiting cycles. MWP represents the number of memory requests that can be serviced concurrently, and CWP represents the number of warps that can finish one computational period during one memory access period. This model works for a single-typed memory system. The introduction of NVM affects MWP, CWP, and the number of memory waiting cycles. We adjust the calculation of these three terms by taking into account of different access latency to different memory systems.

Equations 4-6 show the model computing energy consumption by a kernel. Table I summarizes the used notations. Our energy model breaks down the whole memory energy into background (E_{bg}), read/write due to accesses to memory arrays ($E_{rd/wr}$), and refresh due to DRAM refreshing (E_{ref}). The background energy depends on time distribution of three background power modes (i.e., active standby, precharge power-down and precharge standby). The time distribution of the three modes is statistically related to data access intensity in each rank. More intensive accesses tend to result in more occurrences of active standby and precharge standby (i.e., high power mode). We hence model the average background power as a correlation function of access intensity (defined as AF in the model). The function is established by using segmented linear regression with AF as the independent variable and with average background power as the dependent variable. To measure access intensity, we calculate the total number of memory accesses to each rank's data, based on the estimation of which memory rank each data object resides in.

Read/write energy is estimated by multiplying the number of read/write operations with read/write energy per operation. The read/write energy per operation is based on existing literature [6], [14], [25]. The refresh energy is based on refresh power obtained from [30] and on data distribution among ranks (to determine N_{rank}). The execution time needed to calculate the background energy and the refreshing energy is obtained from the performance model described earlier in this section.

$$E_{mem} = \sum_{\substack{i=dram,\\ pcm}} (E_{bg\_i} + N_{rd\_i} * E_{rd\_i} + N_{wr\_i} * E_{wr\_i}) + E_{ref} \quad (4)$$

$$E_{bg\_dram/pcm} = \sum_{i=0}^{\#ranks} f(AF_i) * T \quad (5)$$

$$E_{ref} = P_{ref} * N_{rank} * T \quad (6)$$

Discussions: For a program with loops surrounding kernel calls, the algorithm is applied inside-out. It first applies to the innermost loop body only. After finding out the proper placement for kernels in that loop, it works on the outer loop body. In the Placement Cost Graph for this program, two nodes are created for the inner loop, one for the data placement already determined in its first kernel and the other for its final kernel, while no nodes are created for the other kernels in the inner loop because they do not interact directly with the kernels in the outer loop in terms of data migrations. This process continues until reaching the highest level of the program.

TABLE I. Energy model notations

E_{mem}	Energy consumption of the hybrid memory system
E_{bg_i}	Background energy consumption
N_{rd_i}, N_{wr_i}	Number of read/write memory operations
E_{rd_i}, E_{wr_i}	Array read/write energy
E_{ref}	Refresh energy consumption
AF	Activity factor
$f(AF)$	Background energy correlation function
T	Total execution time
P_{ref}	Refresh power per rank
N_{rank}	Number of DRAM ranks involved to store data

VI. Experimental Methodology

In our evaluation we use a GPU simulator [32] (extended from GPGPU-Sim v3.1.0 [1]) connected with a memory simulator, DRAMSim2 [27]. We estimate memory power consumption using a power model developed by Micron Corporation [30] and embedded within DRAMSim2. To have a fair comparison, we include DRAM low-power modes during simulation (particularly, precharge power down and precharge standby). The baseline Streaming Multiprocessor (SM) is based on the NVIDIA's Fermi Architecture [22]. Main characteristics of the simulated PCM are based on [20], [6], [25]. Similar to [15], [36], we enhance PCM by writing back dirty row buffers only. Main architectural characteristics of our simulated GPU are summarized in Table II.

Our default hybrid memory system consists of 256 MB DRAM and 1 GB PCM on each memory partition, resulting in 7.5 GB global memory (comparable to main stream GPUs). For fair comparison, in our pure DRAM and PCM configurations, the total memory size on each partition is the same as that of the hybrid memory (i.e., 1.25GB). For the tests without compiler-directed data placement and migration, data arrays are initially placed on PCM to leverage relatively low background power and large size of PCM for fair comparison. Similar to many prior studies [26], [13], [18], we use Energy-Delay Product (EDP) to evaluate energy efficiency. We use CUDA benchmarks listed in Table III. All of the benchmarks are selected from CUDA SDK [23] and Rodinia [5]. We are interested in these benchmarks because they are memory-intensive, and thus suitable to evaluate a memory system.

TABLE III. Benchmark Characteristics (cs: Convolution-Separable, mt: MersenneTwister, sn:SortingNetworks, bs:BlackScholes, sp: scalarProd, pf:Pathfinder, sc:Streamcluster, lava:lavaMD); Access patterns (I:Blocking-regular, II:Blocking-irregular, III:Streaming, IV: Irregular.)

Name	Description	# kernel	#arrays	Classification
cs	a separable 2D convolution	2	3	I&III
mt	random number generator	2	1	III
sn	bitonic sort	4	4	I&III
bs	Black-Scholes formula	1	5	III
sp	scalar product	1	3	I&III
pf	ghost zone optimization	1	3	I&III&IV
sc	number finding	1	5	I&III&IV
lava	particle potential and relocation	1	4	I&III&IV
nn	nearest neighbor	1	2	III

Unless otherwise indicated, the pure DRAM system is used as the baseline for all comparisons. The notation "pc" refers to our parallelism-conscious design, including both hardware and software support; "pc-hw" refers to our design with hardware support only; "pc-sw" refers to our design with software-directed placement and migration only. We use "dbuff" and "RaPP" for the two initial designs of hybrid memory in Section III. "pcmOnly" refers to the pure PCM-based system.

VII. Results and Analysis

A. GPU Workload Characteristics

We use 9 CUDA benchmarks, selected from the CUDA SDK and Rodinia suite [5] for their intensive memory accesses. Based on memory access patterns, the benchmarks fall into four categories as shown in Table III. A benchmark belongs to multiple categories if the accesses to its arrays display multiple access patterns. Six of our benchmarks heavily use shared global memory or display coalesced accesses, which are amenable to GPU architecture, while three benchmarks have dynamic irregular access patterns. Because PCM is highly sensitive to write operations, we further look into the write intensity of these benchmarks. We calculate the write ratio for each data element for each array, and then classify them into five write intensity categories. The category 0 refers to read-only, while the category (0.6, 1] refers to the highly write-intensive pattern. Figure 5 summarizes the data size for each category and displays their percentages in terms of the data size. We have found a large amount of read-only/read-intensive data. For example, most of data in pf, sp and sc are read-only. Contrary to those results in CPU which show much higher write intensity [17], these results reveal the unique opportunities for using PCM on GPU.

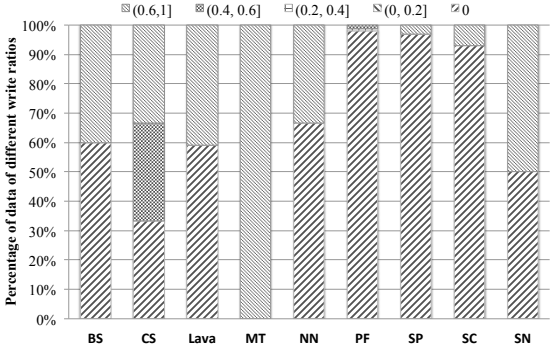

Fig. 5: Write Intensity Distribution of Working Set Data

TABLE IV. Miss Rates (%)
(dcm: DRAM miss rates; llcm: LLC miss rates; rbm: row-buffer miss rates)

	bs	cs	lava	mt	nn	pf	sp	sc	sn
dcm	.11	.17	.0	.12	3.9	39	30	3.6	10
llcm	74	33	56	19	50	41	94	46	56
rbm	11	19	1.1	8.4	18	56	84	2.8	2

B. Dbuff and RaPP

We experiment with the designs of Dbuff and RaPP. Our implementation follows the previous studies [25], [26] with the two modifications mentioned in Section III. In addition, we have tried a spectrum of configurations for these two designs. For instance, we experiment with 4 different block sizes (512B, 1K, 2K, and 4K) and associativity (4, 8, 16) for Dbuff. For lack of space, we report the results of the best configuration (1K; 8-way) in terms of the average EDP.

Figures 6 and 7 report performance and EDP. The results show that the pcm-only design results in 43% energy efficiency degradation and 9% performance loss on average, which demonstrates the necessity of using a hybrid design. However both hybrid memory designs appear to give results even worse than the pcm-only design. The Dbuff causes on average 29% performance loss and 87% lower energy efficiency. The main reason is that the programs show substantial misses in the DRAM as shown in Table IV. The benchmark sp is such an extreme case (13X larger in EDP). RaPP causes 5% performance loss and 11% energy efficiency loss on average. Our analysis shows that one of the reasons is that the design uses LLC miss rates as the hint on the memory frame's temporal locality and makes migration decisions accordingly. As shown in Table IV, the LLC misses and row-buffer misses (a direct indicator of the memory-level data locality) have a poor correlation. This disparity suggests that using LLC as the locality hint is misleading. We also observe bursty data migrations on 4 out of 9 benchmarks (bs, lava, sc, sn). Such migrations consume up to 50% of the total memory channel bandwidth, causing severe interferences to the regular memory transactions.

C. Performance Implications

Figure 6 reports the performance for the three migration strategies. The pc-hw performs the worst for all tests, because the pure hardware-based approach is a slow refining process. It takes time to capture the memory access patterns of data segments with poor locality and expensive memory operations, besides migrating them to DRAM. On the other hand, with the compiler support, we can significantly avoid this situation, demonstrated by the much better performance of pc-sw and pc. We notice that the performance of pc-sw and pc is pretty close while the energy consumption of pc is much lower (shown in the next subsection). This means that, with the support of compiler, the hardware support mainly targets at improving energy efficiency with fine data adjustment. In general, with our data migration method, the performance loss is less than 2%, compared with the DRAM-only system.

D. Energy Implications

Figure 7 reports the normalized EDP savings from all three migration strategies. The pc strategy performs the best with 6% EDP reduction on average. If we purely rely on the compiler-guided placement, the saving is around 4%. There are mainly two reasons. First, our compiler-directed placement suffers from some noise in program analysis (the

TABLE II. Core Parameters

Architecture Configuration	15 SM clusters (30 SMs), Butterfly network (1400Mhz), 6 Memory Partitions
SM Pipeline	1400Mhz, Pipeline Width:16, threads per Warp:32, Maximum threads per SM:1024
Memory Subsystem (per SM)	Constant: 8KB/64B-line/24-way, Texture: 12KB/128B-line/2-way, DL1: 16KB/128B-line/8-way Registers: 32768, Shared Memory: 48KB
L2 Unified Cache	768KB, 256B line, 8-way
Global Memory	DDR3-800Mhz, x8, 1.5V, Cacheline interleaving
Memory Controller	2-channel, 64-entry Transaction Queue, 16-entry Command Queue, FR-FCFS, Open-page
Timing (cycles) [6], [25]	DRAM - tRCD: 11, tRP: 11, tRRDact: 5, tRRDpre: 5, Refresh time: 64ms, tRFC/tREFI: 64/7.8μs PCM - tRCD: 34, tRP: 138, tRRDact: 3, tRRDpre: 18, Refresh time: N/A, tRFC/tREFI: N/A

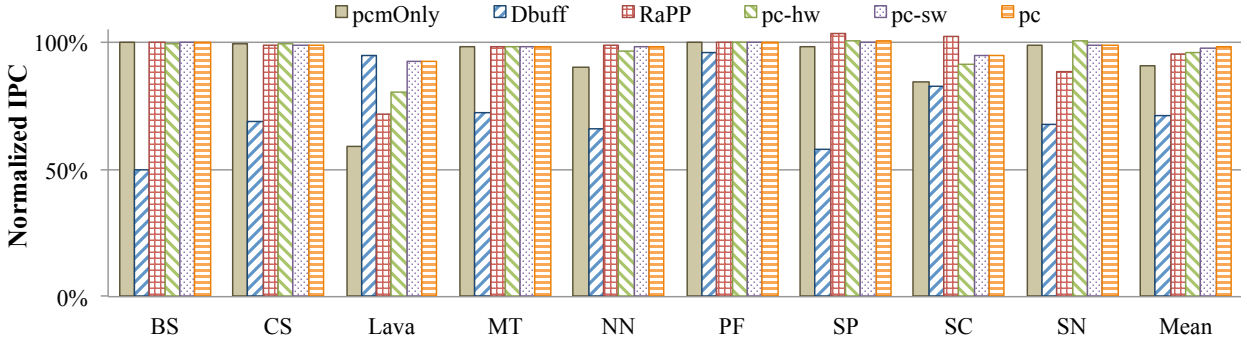

Fig. 6: IPC normalized to those of pure DRAM for various memory systems and migration strategies.

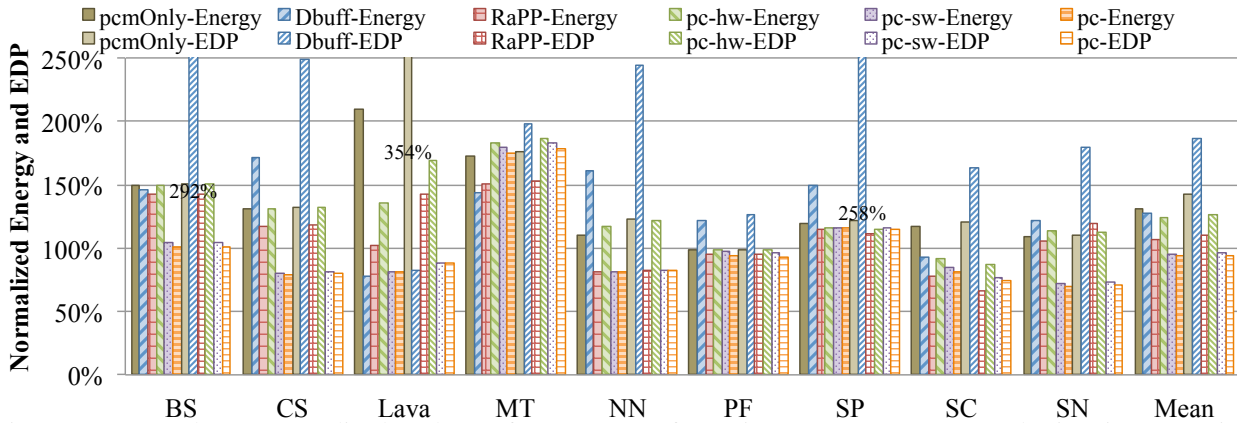

Fig. 7: Energy and EDP normalized to those of pure DRAM for various memory systems and migration strategies.

energy model, the branches and loops in a program). Second, the analysis considers one array as a unit for placement decision at the beginning of the program execution, and thus lacks the ability to do fine-grained tuning. However, the big gap of 33% for EDP between pc and pc-hw argues for the importance of compiler-directed initial placement, which benefits the program from the beginning and avoids memory bandwidth contention from unnecessary data migration. The bottom row of Table V reports the reduction percentage of memory migrations provided by the better initial placements, compared to the pure hardware-based scheme.

We notice that for some particular benchmarks, such as sn and nn, pc-sw outperforms pc-hw significantly, because the initial placement is already close to the optimal but pc-hw still suffers from the naive placement policy. For the two outlier benchmarks, pf and mt, all three strategies perform worse than the baseline. One potential reason is that the execution time of these two benchmarks is too short to benefit from the hardware migration unit.

E. Impact on Row Buffer Misses

Figure 8 shows the row buffer miss rate. The row buffer miss rate refers to the total number of row buffer misses in all memory banks divided by the total number of memory references. The row buffer miss rate reflects row buffer locality. The lower is the miss rate, the better is the row buffer locality. We first notice that pc achieves the lowest row buffer miss rate in 4 cases and achieves the second lowest row buffer miss rate (less than 4% difference in the miss rate) in 2 cases, which demonstrates the effectiveness of locality-aware hardware-based migration. We also notice that pc does not necessarily result in better row buffer locality than pc-hw, shown in the benchmarks bs, lava, and sc. This is because the compiler directs data placement not solely based on locality and the hardware migration can only refine the locality of migrated data. On the other hand, with the direction of compiler, we effectively avoid expensive PCM writes and the performance loss as shown in Figure 6, even though the row buffer misses may be slightly worse than the pure hardware-based approach. Furthermore, we notice that the hardware migration helps the pure software-based approach reduce the row buffer miss rate, demonstrated by the fact that pc achieves lower row buffer miss rate than pc-sw.

F. Cycles Per Memory Access

The relatively long memory access latency is one of the major obstacles for embracing NVM. We calculate the average memory access latency for all memory transactions, and show

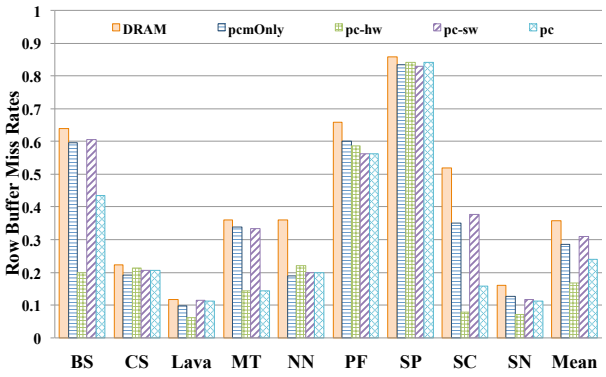

Fig. 8: Normalized row buffer miss rates for various memory systems and migration strategies.

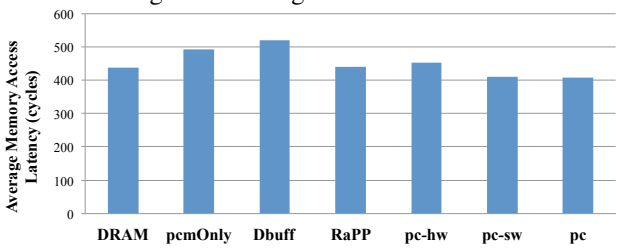

Fig. 9: Cycles per memory access for various memory systems and migration strategies.

the average value for all benchmarks in Figure 9. pc has the lowest value among all cases, even lower than the DRAM case, because of improved row buffer locality. Without the coordination of hardware and software, both pc-hw and pc-sw incur longer latency than pc. pc also beats the best previous work (i.e., RaPP) due to compiler assists.

G. Memory Bandwidth Consumption and Migration Rate

Figure 10 presents average memory channel bandwidth consumption of all benchmarks. Compared to Dbuff and RaPP, pc reduces bandwidth consumption because of the reduction in unnecessary migration. pc also consumes smaller bandwidth than pc-hw because of compiler assistance. Figure 11 presents the migration rate. The migration rate is calculated as the percentage of migrated data (weighted by the corresponding times of migrations) in terms of total data size. Among the three hardware-based migration, pc has the lowest rate because of compiler assistance. pc-hw has higher migration rate than RaPP, because of the smaller migration granularity, but pc-hw has better energy efficiency.

H. Overhead of the Computation of the Initial Placements

Table V reports the time overhead from the computation of the initial data placements using the algorithm in Section V. The overhead is less than 1% for all programs except for three programs *nn*, *sn*, and *sp*, which have the shortest duration of

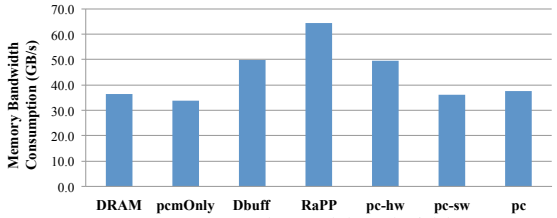

Fig. 10: Average memory channel bandwidth consumption

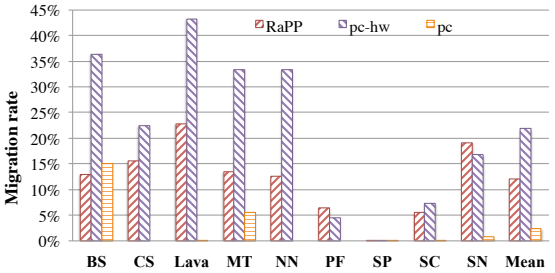

Fig. 11: Migration rate (weighted by migration times) for hardware migration strategies

TABLE V. Overhead of Initial Placement Engine and Migration Reduction (oh: overhead; rd: reduction of migration)

	bs	cs	lava	mt	nn	pf	sp	sc	sn
oh	.15	.16	.2	.09	2.4	.13	.04	4.1	2.8
rd	41	100	63	0	100	100	0	16	51

all programs. The largest overhead is 4.1% on *sp*, still much smaller than the EDP savings.

I. Impact on PCM Write Endurance

Another major concern for applying PCM is its limited write endurance. In order to analyze the lifetime of PCM, we use the endurance analytical model from [25], $Y = W_{max} * S/(F * B * 2^{25})$, where W_{max} is the maximum number of writes that a PCM cell can take before being worn out, S is the total size of PCM in bytes, F is the processor frequency, B is the write traffic in bytes per cycle and a year roughly has 2^{25} seconds. Similar to [15], we assume W_{max} as 10^8 in this evaluation.

Figure 12 shows that a pure PCM configuration without any optimization has an average lifetime of 0.8 year ($B = 19.19$), because of intensive data accesses. Dbuff and RaPP improve the lifetime to 1.7 years ($B = 7.16$) and 2.1 years ($B = 8.59$), respectively, by directing memory writes to DRAM. However, they are much less efficient than our co-design approach (pc with 5.4 years life time and $B = 2.29$). Further analysis reveals that pc-hw redirects 29% write traffic to DRAM, and pc-sw redirects 86.4%, while co-design redirects 98.4%. Our co-design approach efficiently places a large extent of write-intensive data in the DRAM memory.

VIII. Related Work

Data placement in memory systems: Some studies have considered data placement for the hybrid memory system on multicore CPU. Qureshi et al. [25] and Bivens et al. [4] use DRAM as a set-associative cache that is logically placed between processor and PCM. PCM is accessed when DRAM buffer eviction or buffer miss happens. Ramos et al. [26] rely on MC to monitor popularity and write intensity of memory pages, based on which to migrate pages between DRAM and PCM. Yoon et al. [33] place data based on row buffer locality. These studies have all focused on multicore CPU. None of

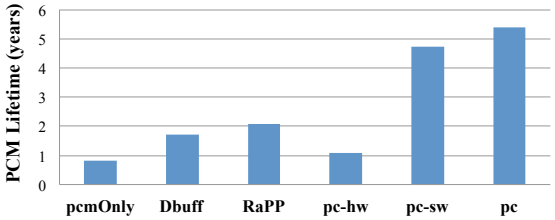

Fig. 12: Estimated PCM endurance

them has studied the implications of GPU massive parallelism to hybrid memory design. As Section III shows, when applying to massive parallel platforms, they yield unsatisfying performance and energy efficiency. We are not aware of previous proposals of the set of new hardware features in Section IV for reconciling the design with GPU parallelism. Furthermore, the previous studies have used software mechanism mostly as a way to maintain the memory management scheme (e.g., paging, virtual pages to physical frames mapping, and wear leveling), rather than to guide data placement.

Compiler-Assisted GPU Memory Performance Optimization: Recent years have seen a number of studies on enhancing GPU memory performance through compiler-assisted techniques. Hormati et al. present a compiler framework to support stream programming on GPU, including some memory optimizations [10]. Zhang et al. propose an online adaptive scheme to enable runtime thread-data remapping to minimize uncoalesced memory accesses [34]. Lee et al. [16] describe some techniques to optimize memory references during openMP-to-CUDA translation. Baskaran et al. [3] use a polyhedral analysis to optimize affine memory references in regular loops. All these studies are about matching software with the special properties of GPU DRAM accesses, rather than finding the best placement of data on a hybrid memory system. We are not aware of any previous proposal of the Placement Cost Graph for modeling constraints in data placement on a hybrid memory. To our best knowledge, this work is the first that uses compiler support to find an appropriate data placement on a hybrid memory.

IX. Conclusion

We have explored the use of hybrid memory as the GPU global memory, and investigated how to match memory design with massive parallelism of GPU devices and improve energy efficiency. By exploiting the synergism of compiler and hardware, our hybrid memory architecture leads to an average energy efficiency improvement of 6% and 49%, respectively, compared to pure DRAM and pure PCM. The performance loss is less than 2%.

Acknowledgments

Bin Wang and Bo Wu have made significant contributions on the hardware side and the compiler side, respectively. This work is funded in part by an Alabama Innovation Award, by an NSF award CNS-1059376 and by DOE Early Career Award. We are very thankful for GPU equipment donated from NVIDIA to Auburn University.

References

[1] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009.

[2] L. A. Barroso and U. Holzle, *The Datacenter as a Computer: An Introduction to the Design of WarehouseScale Machines*. Morgan and Claypool, 2009.

[3] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for GPGPUs," in *ICS*, 2008.

[4] A. Bivens, P. Dube, M. Franceschini, J. Karidis, L. Lastras, and M. Tsao, "Architectural Design for Next Generation Heterogeneous Memory Systems," in *International Memory Workshop*, 2010.

[5] 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.

[6] J. Chen, R. C. Chiang, H. H. Huang, and G. Venkataramani, "Energy-Aware Writes to Non-Volatile Main Memory," in *Workshop on Power-Aware Computing and Systems*, 2011.

[7] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A Hybrid PRAM and DRAM Main Memory System," in *DAC*, 2009.

[8] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *ISCA*, 2009.

[9] ——, "An integrated gpu power and performance model," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010.

[10] A. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke, "Sponge: portable stream programming on graphics engines," in *ASPLOS*, 2011.

[11] HP, "Server Power Calculators," http://h30099.www3.hp.comconfigurator/powercalcs.asp.

[12] W. kei S. Yu, R. Huang, S. Q. Xu, S.-E. Wang, E. Kan, and G. E. Suh, "SRAM-DRAM hybrid memory with applications to efficient register files in fine-grained multi-threading," in *ISCA*, 2011.

[13] A. Lebeck, X. Fan, H. Zeng, and C. Ellis, "Power Aware Page Allocation," in *ASPLOS*, 2000.

[14] B. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecturing Phase Change Mmemorhy as a Scalable DRAM Architecture," in *Int. Symp. Computer Architecture*, 2009.

[15] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *ISCA*, 2009.

[16] S. Lee, S. Min, and R. Eigenmann, "Openmp to gpgpu: A compiler framework for automatic translation and optimization," in *PPoPP*, 2009.

[17] D. Li, J. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu, "Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications," in *IPDPS*, 2012.

[18] J. Li and J. F. Martinez, "Power-Performance Considerations of Parallel Computing on CMP," *ACM Trans. Arch Code Optim*, 2005.

[19] 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.

[20] Micron, "MT41J128M8," http://download.micron.com/pdf/datasheets/dram/ddr3/1Gb_DDR3_SDRAM.pdf.

[21] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi, "Operating system support for NVM+DRAM hybrid main memory," in *In Proc. of the conference on Hot topics in operating systems*, 2009.

[22] NVIDIA, "Fermi: NVIDIA's Next Generation CUDA Compute Architecture," http://www.nvidia.com/fermi.

[23] NVIDIA, "GPU computing SDK code samples," http://developer.nvidia.com/cuda-toolkit-31-downloads.

[24] H. Park, S. Yoo, and S. Lee, "Power Management of Hybrid DRAM/PRAM-Based Main Memory," in *DAC*, 2011.

[25] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High-Performance Main Memory System Using Phase-Change Memory Technology," in *ISCA*, 2009.

[26] L. Ramos, E. Gorbatov, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in *ICS*, 2011.

[27] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16 –19, jan.-june 2011.

[28] Semiconductor Industry Association, "The International Technology Roadmap for Semiconductors. Process integration, devices, and structures," http://www.itrs.net/Links/2010ITRS/2010Update/ToPost/2010Tables_LITHO_FOCUS_D_ITRS.xls, 2010.

[29] 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.

[30] M. Technology, "Calculating memory system power for ddr3," Technical Report TN-41-01, Tech. Rep., 2007.

[31] M. U. Nawathe, L. Warriner, K. Yen, B. Upputuri, D.Greenhill, A.Kumar, and H. Park, "An 8-core, 64-thread, 64-bit, power efficient SPARC SoC," in *ISSCC*, 2007.

[32] B. Wang and W. Yu, "Performance and Power Simulation for Versatile GPGPU Global Memory," in *IPDPS (PhD forum)*, 2013.

[33] H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu, "Dynrbla: A high-performance and energy-efficient row buffer locality-aware caching policy for hybrid memories," Carnegie Mellon University, Tech. Rep., 2011.

[34] E. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for gpu computing," in *ASPLOS*, 2011.

[35] W. Zhang and T. Li, "Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architecture," in *PACT*, 2009.

[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.