CART: Cache Access Reordering Tree for Efficient Cache and Memory Accesses in GPUs

Yongbin Gu, Lizhong Chen
School of Electrical Engineering and Computer Science
Oregon State University, Corvallis, USA
{guyo, chenliz}@oregonstate.edu

***Abstract*—Graphics processing units (GPUs) have been increasingly used to accelerate general purpose computing. Thousands of concurrently running threads in a GPU demand a highly efficient memory subsystem for data supply. A key factor that affects the memory subsystem is the order of memory accesses. While reordering memory accesses at L2 cache has large potential benefits to both cache and DRAM, little work has been conducted to exploit this. In this paper, we investigate the largely unexplored opportunity of L2 cache access reordering. We propose *Cache Access Reordering Tree (CART)*, a novel architecture that can improve memory subsystem efficiency by actively reordering memory accesses at L2 cache to be cache-friendly and DRAM-friendly. Evaluation results using a wide range of benchmarks show that, the proposed CART is able to improve the average IPC of memory intensive benchmarks by 34.2% with only 1.7% area overhead.**

I. Introduction

With massive parallel computing ability, graphics processing units (GPUs) are being increasingly used to accelerate numerous scientific, economic and general purpose computing applications. GPUs employ single instruction, multiple thread (SIMT) architecture, which allows thousands of threads running simultaneously (e.g. up to 3584 threads in NVidia GTX1080 Ti). These concurrent threads generate a large number of memory requests that put high pressure on the memory subsystem (e.g., cache, on-chip network, DRAM) [1]. If not designed with care, the memory subsystem can easily become a serious factor that prevents GPUs from achieving peak performance. With the current technology and application trends, the issue of memory subsystem will likely worsen in the near future. On the technology side, the development of memory technology have been lagging behind processing, e.g., from NVidia GTX480 to GTX1080 Ti, the core count increases by more than 7.4X, but the DRAM bandwidth increases only by about 1.7X. On the application side, irregular memory access patterns have been exhibited in more and more GPU workloads (such as trees, priority queues, key-value storage [2], [3]), which often have poor cache locality and greatly exacerbate the memory stress. Thus, it is imperative to explore new opportunities in the memory subsystem, particularly at the architecture level, to bridge the gap between technology and application demands.

A key factor that determines the efficacy of memory subsystem at all levels of the memory hierarchy is the order of memory accesses. The order affects not only the hit/miss of the current level, but also determines which accesses are exposed to the next level. While prior research has investigated the access reordering benefits in L1 cache and in DRAM (More details in Related Work), the reordering opportunity at L2 cache has largely been unexplored. Nevertheless, the access order to L2 can have a large impact on both L2 cache and DRAM. On the one hand, the access order can be utilized to extract potential data locality to increase cache hit, as well as to reduce avoidable head-of-line blocking in the request buffer of L2 cache. On the other hand, the access order also determines the request order to DRAM. A benign request sequence to DRAM offered by L2 can greatly facilitate memory controllers to improve row-buffer hit and bank-level parallelism (BLP), both of which are critical to DRAM performance. Substantial research is needed on how memory accesses can be reordered to achieve a cache-friendly and DRAM-friendly order.

In this work, we explore the opportunity of reordering memory accesses at L2 cache. We conduct an in-depth analysis on when and why access reordering at L2 can be beneficial to both cache and memory. The challenge, however, is to design a well-rounded reordering architecture that addresses data locality, row-buffer hit, bank-level parallelism and low design cost at the same time.

To address this challenge, we propose *Cache Access Reordering Tree (CART)*, a novel yet effective architecture to reorder memory accesses at L2 cache. The main idea is to classify and group memory accesses by passing the accesses through a reordering tree. The reordering tree takes into account data locality in cache lines to increase cache hit, as well as the bank, row and column information of the accesses to increase DRAM efficiency in case of cache misses. We propose a way to use a very small number of leaf queues to mimic the effects of having a large number of queues to reduce hardware cost. A fill policy and a drain policy for memory requests are carefully designed to make full use of the reordering tree. Cycle-accurate simulations based on a wide range of benchmarks show that, the proposed CART is able to improve the average IPC (geometric mean) of memory intensive benchmarks by 34.2% with only 1.7% area overhead, compared with the conventional design. Furthermore, CART is able to complement other state-of-the-art techniques on GPU caches to achieve higher performance. For example, when combined with MRPB (Memory Request Prioritization Buffer) [4] and RACB (Resource Aware Cache Bypass) [5], the two combinations can achieve a total improvement of average IPC by 38.6% and 41.5%, respectively.

II. Background and Motivation

A. Memory Subsystem in GPUs

Figure 1 depicts a typical GPU architecture and where the proposed CART fits. A GPU mainly consists of streaming

multiprocessors (SMs), interconnect network, L2 cache, and DRAM. An SM has a number of SIMT cores (e.g. 128 cores per SM in NVidia GTX1080 Ti) to execute multiple threads in parallel. For the memory subsystem, L1 cache(s) exists inside each SM and handles requests from multiple SIMT cores within the SM; whereas L2 handles memory requests that are coming from the SMs through the interconnect network. The logically unified L2 cache is split into several partitions and each partition is associated with a DRAM partition. To track multiple outstanding misses to the DRAM, miss status handling registers (MSHRs) are employed to keep track of the needed information for each DRAM request, such as the requester core ID, cache block address, returned data destination, new data for write-back (in case of writing). For a primary cache miss that requests a new cache block, one MSHR entry is allocated. For a secondary cache miss (that requests data in the same cache block that has been allocated an MSHR entry and is currently pending), one slot in the MSHR entry is allocated, provided that an empty slot is available in that entry. A typical MSHR may have 32 or 64 entries, with each entry having 4 slots.

B. Impact of Access Order on L2 Cache

The order of memory accesses to L2 cache plays a significant role in determining memory access latency. The access order not only affects the locality of data which in turn influences cache misses, but also has a large impact on the blocking time of memory accesses in the cache. The latter is due to the FIFO nature of the incoming buffers in L2 cache. In conventional GPUs, memory requests that come out of the interconnection networks are enqueued in the incoming buffer of the corresponding L2 cache partition (Figure 1 and Figure 2(a). When a request moves to the head of the buffer, L2 checks if the request is a hit in the cache; if not, the request needs to be issued to DRAM by allocating an MSHR entry or slot. However, a reservation fail (RF) may happen when no entry/slot is available in the MSHR or when the miss queue to DRAM is full. As a result, the request has to stay in the incoming buffer and retries later. This blocks other memory requests in the FIFO buffer, even though some of the requests could hit in the L2 cache (no need for MSHR) or use MSHR in other ways (more analysis in Section 3.1). This head-of-line blocking is more pronounced for irregular memory accesses that have burst patterns. One of our goals is to reduce the occurrence of head-of-line blocking without affecting data locality through a better cache-friendly reordering scheme.

C. Impact of Access Order on DRAM

The order of memory accesses also has a large impact on the efficiency of DRAM because of row-buffer conflicts and bank-level parallelism (BLP). DRAM has a three-level structure, namely banks, rows and columns [6]. For example, a DRAM chip may consist of 16 banks, with each bank having thousands of rows and tens of columns in each row. The size of each column in a row is usually the size of a cache line (e.g., 128 bytes). Therefore, upon a cache miss, the memory address is decoded to locate the correct bank, row and column to fetch an entire cache line (i.e., a column). Modern DRAMs employ a row buffer in each bank that serves as a "cache" function for temporarily storing the contents of one row, so as to accelerate future accesses of columns in the same row. A *row buffer conflict* occurs if the column from a different row is requested, in which the row buffer is flushed and refilled by the contents of the newly requested row. This results in additional access latency. Several memory schedulers (e.g. [7], [8]) try to reduce row buffer conflicts by reordering memory accesses on the DRAM side. However, due to the above-mentioned blocking issue in L2, many memory requests are congested at L2. This leaves a limited number of requests at the front of DRAM for reordering. Hence, it is important to create a benign order of memory accesses early on at the L2 cache. Similarly, as banks in a DRAM chip can work in parallel, it is also beneficial to reorder memory accesses at L2 in a DRAM-friendly way to help distributing memory requests more evenly among different banks to increase parallelism.

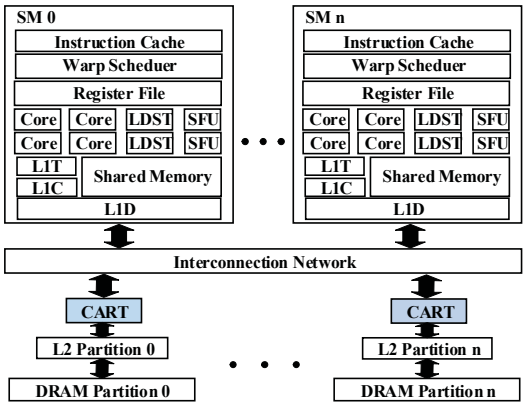


Fig. 1: A typical GPU architecture (MSHRs in L1 are omitted for clarity). The proposed CART is added before L2.

D. Need for More Research on Reordering

To improve the effectiveness of the memory subsystem, several optimization approaches have been proposed, but the opportunity of reordering memory access order at L2 cache has largely been unexplored. One approach is to increase MSHR sizes to reduce reservation fails. However, enlarging MSHR is often prohibitively costly due to its content-addressable memory (CAM) circuitry [9], [10], and not all blocking cases are caused by MSHR size limitation. Additionally, increasing MSHR size does not improve DRAM efficiency as it could not reorder memory requests to lower row buffer conflicts or increase BLP.

In terms of reordering, reordering memory requests at L1D in a cache-friendly order has been proposed to increase cache hits and overall performance [4]. Cache bypassing is used to reduce the penalty of reservation fails [5], [11], [12]. Researchers also propose to reorder through memory schedulers at memory controllers to reduce memory accessing latency and increase DRAM working parallelism [13]. While more related works are discussed in Section VII, existing approaches have not explored the reordering at L2 cache, which has large impact on both cache and DRAM as analyzed in the above two subsections. In following sections, we present how a reordering architecture and strategy can be designed at L2 cache to address data locality, head-of-line blocking, row buffer conflict, and bank-level parallelism at the same time.

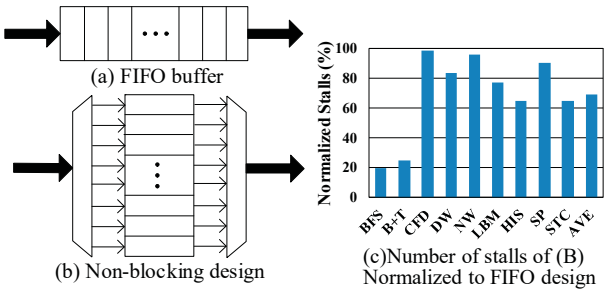


Fig. 2: Blocking vs. non-blocking request buffers.

III. Exploring Access Reordering at L2

A. Blocking of Memory Requests at L2

The root cause of the blocking issue of memory requests at L2 is the FIFO structure of the incoming buffer. Under such design, if the memory request at the head of the incoming buffer (head request) is stalled, all the subsequent requests are blocked in the buffer. Specifically, there are three cases where removing such blocking may lead to performance benefits: (1) the head memory request is a primary cache miss and is stalled due to the lack of available entry in MSHR; however, a currently blocked subsequent request could have been merged into an existing MSHR entry (i.e., a secondary miss). (2) the head memory request is a secondary cache miss and is stalled due to the lack of available slot in the matching MSHR entry (i.e., needs to be merged with the primary miss); however, empty MSHR entries are available and could have been allocated to currently blocked subsequent requests. (3) the head memory request is stalled due to reservation fail or DRAM saturation (or any other reasons), but the blocked subsequent requests could have hit in L2 cache and should have proceeded.

B. A Straightforward Non-blocking Design

To reap the above benefits, we start by considering a simple but non-blocking incoming buffer design that supports any access order. As illustrated in Figure Figure 2(b), the incoming buffer is restructured to enable parallel selection of any memory request using a giant multiplexer. When a request encounters a stall, a selection policy (e.g., round-robin) is employed to select the next request that is qualified for draining from the buffer structure. The selected request must not be stalled by the same resource as the previously stalled request. Although being straightforward, this design can significantly reduce the number of stalls at L2 by 68.8% on average, as shown in Figure 2(c). Nevertheless, this design has two major drawbacks:

- It only solves the blocking that is local to L2 cache, while neglecting other opportunities in DRAM down the line, such as row buffer hit and bank-level parallelism.
- The arbitration can be quite complex, as the multiplexer and control logic need to scan through all the requests in the buffer to identify a qualified draining candidate.

C. An Improved Design for Access Reordering

To tackle these problems, we examine an improved design that takes into account bank-level parallelism and arbitration. As shown in Figure 3, in this design, the FIFO incoming buffer remains the same, but B FIFO queues are added to classify memory requests that come out of the incoming buffer. A simple address extractor extracts the bank address from a given memory request, and directs the request to one of the FIFO queues by calculating (bank_address mod B).

Note that if B equals the number of banks, memory requests are essentially queued by their bank addresses. However, B can be less than the number of banks, in which case memory requests destined to different banks may share a queue. Finally, for draining, a round-robin policy is used to select a non-empty queue among the B queues in each cycle.

Compared with the parallel design in Figure 2(b), DRAM bank-level parallelism is improved because every time a memory request is selected to drain, its bank address is guaranteed to be different from the last time, thus helping to have multiple banks to work concurrently. Furthermore, arbitration complexity is also reduced as the arbitrator only needs to select among B choices. Simulation results show 10.8% improvement in IPC and 21.0% improvement in DRAM efficiency (defined as DRAM active cycles over total DRAM cycles) of this design, with arbitration time appropriately accounted for. The improvement is greater for larger B due to the higher degree of BLP.

Although this design addresses incoming buffer blocking, arbitration, and BLP issues, it still has two drawbacks:

- While draining from different queues increases BLP, it destroys the data locality in the original program. This significantly increases miss rate (20.8% more on average).
- Memory requests that go into the same queue may have mixed (random) row and column address, thus susceptible to row buffer conflicts.

To address these issues, we need a more comprehensive, yet low-cost, reordering scheme, as proposed next.

IV. CART: Cache Access Reordering Tree

A. Overview

Our objective is to reorder memory accesses at L2 cache in a cache-friendly and DRAM-friendly way. To achieve this, in addition to classifying memory requests based on bank addresses, the requests need to be further classified by row and column addresses. Ideally, requests with the same bank, row and column addresses should be grouped together, because they access the same row buffer in the DRAM and belong to the same cache line (i.e., same column). However, this grouping method is highly impractical as there are thousands of different rows in a bank and tens of columns in a row. It is impossible to provide a separate queue for each combination of (bank, row, column). Therefore, we need a way to mimic the effects of having a large number of queues but using a limited number of physical queues. To realize this, we propose *Cache Access Reordering Tree (CART)*.

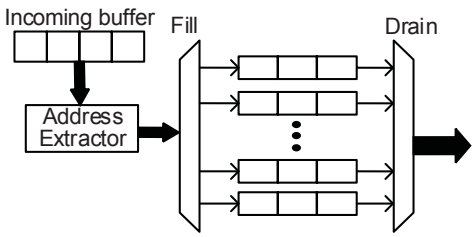


Fig. 3: Reordering based on bank information.

As shown in Figure 1, CART is positioned right before L2 cache to actively reorder memory requests. Figure 4 illustrates the structure within CART. Every memory request that pops out from the incoming buffer will go through a reordering tree to reach one of the FIFO *leaf queues*. To achieve a high degree of BLP, CART provides a tree *branch* for each bank (e.g., 16 branches if a DRAM chip has 16 banks). Within a branch/bank, instead of having a leaf queue for each pair of (row, column), there is a small pool of leaf queues (e.g., 8 queues). A leaf queue can be dynamically assigned to any (row, column) pair to buffer memory requests that have the matching row and column addresses. The *fill* policy determines if a memory request should be put into an existing leaf queue or be assigned a new leaf queue. The *drain* policy determines which leaf queue to output a memory request. A leaf queue is de-assigned when it is empty. A tag is attached to each leaf queue to indicate the current (row, column) assignment of the queue. As the bank address of a branch is implicitly known, the tag includes only the information of row and column, where R_x represents the row address and C_x represents the column address. To provide fairness and avoid the cases where one row uses up all the leaf queues in a branch, each row has a fixed number of assigned leaf queues. For example, if this number is 2 and the branch size is 8 leaf queues, then there are four rows in a branch, with each row being capable of buffering memory requests for two different columns. Each leaf queue can be very small, with only a few entries per queue.

With this structure, requests are naturally grouped by rows and columns, whereas accesses to different banks are separated in different branches. These properties make it possible to address data locality, row buffer hit, and BLP issues at the same time. The carefully-designed fill and drain policies (described in following subsections) utilize the CART structure to achieve these objectives, while simplifying arbitration efforts.

Figure 5 exemplifies what can be achieved by the proposed CART. MRn denotes memory request n, and Bi, Rj and Ck represent the bank address, row address and column address of this request, respectively. Figure 5(a) shows the original order of a sequence of memory accesses (note: leftmost request occurs first in time). With the FIFO incoming buffer in conventional L2 cache designs, there are a number of places where data locality and BLP are lost. For instance, MR0 and MR4 belong to the same row in the DRAM bank. However, by the time that MR4 arrives at the DRAM, the row buffer may have been replaced by MR1's row, causing an extra row buffer conflict. Also, MR0 and MR2 belong to the same cache line and MR2 could hit in L2 without going to DRAM. However, due to MR1 that takes place between MR0 and MR2, MR0's cache line could be replaced by MR1 if they are mapped to the same position in L2. This disrupts data locality and causes MR2 to miss in the cache.

Figure 5(b) illustrates the access order after performing cache-friendly reordering. By switching the order of MR1 and MR2 (e.g., by filling MR1 and MR2 into different leaf queues and then drain MR0 and MR2 consecutively), MR2 can result in a cache hit without fetching from DRAM. Additionally, DRAM-friendly reordering as illustrated in Figure 5(c) can improve BLP and reduce row buffer conflicts. For example, MR3 has a bank address that is different from that of MR2. If MR3 and MR1 switch order (e.g., by draining different branches in CART), Bank 0 and Bank 1 can fetch the required data in parallel, increasing DRAM BLP. Furthermore, MR4

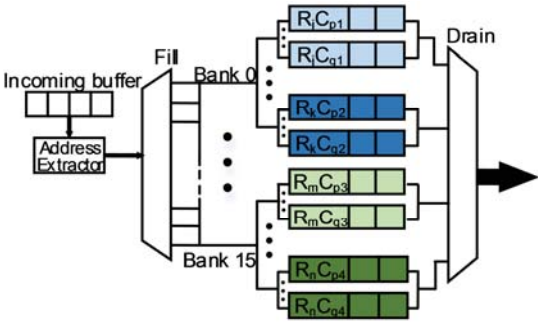


Fig. 4: Diagram of the proposed CART.

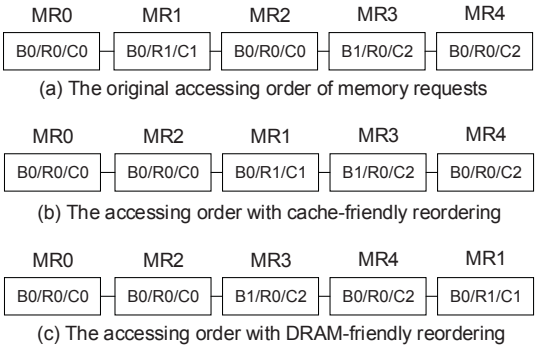


Fig. 5: An example of the effects of CART. *Bi*/*Rj*/*Ck* denotes that the address of the memory request (MR) is in bank *Bi*, row *Rj* and column *Ck*.

can reuse the data in the row buffer of B0/R0 if MR4 is put into one of the leaf queues that belong to B0/R0. This avoids a potential row buffer conflict.

B. Leaf Queue Allocation

A key aspect of CART is to use a small number of leaf queues to approximate the effect of having a large number of queues in a branch. The rationale behind this that, despite the tens of thousands of (row, column) address combination for a bank, there are only a limited number of (e.g., tens of) outstanding memory requests per bank. Moreover, some of the outstanding requests share the same row and column addresses, and can be merged in one leaf queue. This indicates that it is possible to use a small number of leaf queues to buffer all the outstanding memory requests, as long as there is a leaf queue or an entry in a leaf queue available by the time a new request comes.

With a given number of leaf queues in a branch, there are several queue allocation strategies. We can allocate more leaf queues to a row, so as to accommodate more memory requests for different column addresses in the row, at the cost of fewer rows. Or we can allocate fewer leaf queues to a row, which increases the number and diversity of rows in a branch, but at the cost of fewer columns per row. Additionally, under the same total buffer space, the number of entries in a leaf queue can be reduced to increase the number of leaf queues. To this end, we conducted an extensive experimental study to identify the best trade-off configuration. While details are presented in Section VI-A, we observed that the performance of providing two 2-entry leaf queues per row and four rows per branch is within 3% of the performance of a configuration that has 32

times as many buffer resources. This demonstrates the viability of using limited queues to reorder memory requests.

C. Fill Policy

The fill policy determines which leaf queue an incoming memory request should be buffered. Since CART provides one branch per bank, the fill policy only needs to select among the leaf queues in a branch. The fill policy works in a straightforward way: if one of the leaf queues contains the same row and column information as the new request, the request is merged into this queue by occupying an empty entry (in the FIFO order). If there is no empty entry in the matching queue or if there is no matching queue, a new available leaf queue is allocated to store the request, with the tag information set to the row and column addresses of the new request. Here, "available" means that an empty leaf queue is available among the leaf queues that are assigned to that row. Lastly, if no such leaf queue is available, the new memory request is stalled in the incoming buffer and retries later. Note that the probability for stalling can be kept very low with a sufficient number of leaf queues, and our study shows that the above configuration with only 8 leaf queues per branch can already achieve a near-zero probability in most cases.

D. Drain Policy

The drain policy is inherently more difficult to design than the fill policy, as the effect of a not-so-good fill decision may be delayed and partially compensated by the buffering effect of the tree, but a drain decision directly affects which requests are issued to the cache. Unfortunately, commonly-used general drain policies such as greedy, longest-first, and round-robin do not work well with CART. For example, if the longest-first policy is applied to CART, the longest leaf queue is selected to drain in every cycle. This increases locality as all the requests in a queue share the same row and column, but squanders the opportunity for bank-level parallelism. Figure 6 illustrates an example. When the longest-first drain policy is applied, the first several selected requests would be in this order is Q1_MR7, Q1_MR6, Q2_MR11, Q0_MR2, Q1_MR5, Q2_MR10 (queue ID is used for tie-breaker). As can be seen, no requests are selected for Bank 2 and Bank 3, and they are not working during all this time. Similarly, round-robin policy does not work well either as it actively selects requests across different queues, thus destroying the data locality.

To address these issues, we propose a "rotating banks and same-or-longest row" drain policy to achieve both cache-friendly and DRAM-friendly order. The drain policy includes two aspects. In the first aspect, the policy selects one and only one request from a branch (i.e., leaf queues belonging to the same bank), and then immediately rotates to the next non-empty branch (bank) in the next cycle. In the second aspect, when the policy rotates back the same branch, the same leaf queue that was selected last time is selected this time, or if that leaf queue has already been fully drained, the longest leaf queue with the same row in the branch is selected (if all the leaf queues of that row are drained, simply select the longest leaf queue in the branch). Essentially, the first aspect increases BLP; whereas the second aspect increases data locality and row buffer hit, as the requests in the same leaf queue access the same cache line (if cache hit) or the same DRAM row (if cache miss). Take the example in Figure 6 again. Assuming the proposed policy starts with Bank 1,

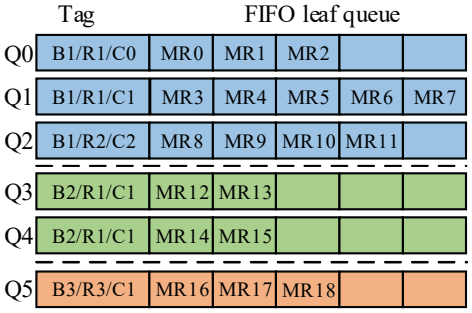


Fig. 6: Illustration of drain policies for a given tree status.

since this is the first time with no prior history, the longest leaf queue, Q1_MR7, is selected. Then the policy rotates to Bank 2 and selects Q3_MR13. This is followed by Q5_MR18 in Bank 3. When the policy rotates back to Bank 1, the same leaf queue as the last time, Q1_MR6, is selected to maintain the locality, regardless of whether Q1 is currently the longest. Similar process continues, with the draining order of Q3_MR12, Q5_MR17, Q1_MR5, Q4_MR15, Q5_MR16, Q1_MR4, Q4_MR14, Q1_MR3, Q0_MR2 (because of same row with Q1), Q0_MR1, Q0_MR0, Q2_MR11, Q2_MR10, Q2_MR9, Q2_MR8. Compared with longest-first and round-robin, this order achieves substantially reduced cache misses and row-buffer conflicts while utilizing multiple banks effectively.

TABLE I: CART design configuration.

# of leaf queue	Bank: 16; Row: 4; Column: 2
Queue size	2 entry per queue
Tag	Row + column addresses
Drain policy	Rotating banks and same-or-longest row

TABLE II: Simulation configuration.

# of SMs	28
Warp Scheduler	GTO
Per-SM limit	48 warps, 8 CTAs
# of Memory Partitions	8
L1D cache	16 KB, 32-set, 4-way, 32 MSHR, Allocate on Miss, Local write-back, global write-through
L2 cache	8x128 KB, 64-set, 16-way, 32 MSHR, Allocate on Miss, write-back
DRAM	FR-FCFS scheduler, GDDR5, 16 banks
SM/L2/DRAM clock	1400/700/1150 MHz

V. Evaluation Methodology

We implement the proposed schemes in a cycle-accurate simulator, GPGPU-Sim 3.2.2 [14]. GPUWatch [15] is employed to evaluate the energy consumption of our proposed scheme and baseline architecture. Table II lists the configuration used in the simulator. The benchmarks from Rodinia [16], Parboil [17], and Nvidia GPU Computing SDK are evaluated. Table III lists more details of the benchmarks. The second and fifth columns in the table illustrate the total number of instructions executed by the entire SMs over the number of L2 cache miss. These values reflect the extent to which the performance of the benchmarks depends on cache performance [18], [19]. The benchmarks whose total executed instructions per L2 miss are less than 1500 are considered as the memory intensive benchmarks and are marked *M* in the "Type" column. Other benchmarks whose values are over 1500 are compute intensive benchmarks and are marked *C* type.

TABLE III: Evaluated benchmarks.

Benchmarks	Abbr.	Inst./L2 Miss	Type	Benchmarks	Abbr.	Inst./L2 Miss	Type	Benchmarks	Abbr.	Inst./L2 Miss	Type
Backprop	BP	1718	C	Tpacf	TP	842408	C	Spmv	SP	568	M
Bfs	BFS	62	M	Aligned Types	AT	238	M	Stencil	STC	614	M
B+tree	B+T	1497	M	AsyncAPI	AA	416	M	Sgemm	SG	9957	C
Cfd	CFD	404	M	Black Schole	BS	476	M	Transpose	TRP	400	M
Dwt2d	DW	530	M	Convolution Separable	CS	701	M	Mri-q	MRI	120455	C
Heartwall	HW	5189	C	Convolution Texture	CT	3104	C	Scan	SCN	233	M
Nw	NW	198	M	Fast Walsh Transform	FWT	641	M	Lbm	LBM	176	M
Kmeans	KMS	19236	C	QuasirRandom Generator	QG	2798	C	Merge Sort	MS	1436	M
Cutcp	CUT	400509	C	Radix Sort Thrust	RST	425	M	Histo	HIS	751	M
Sad	SAD	2733	C	Sorting Network	SN	1160	M	SobolQRNG	SQ	1247	M

We compare CART against the baseline architecture, as well as two state-of-the-art techniques, MRPB [4] and RACB [5]. MRPB uses memory request prioritization buffer to reorder memory requests in the L1D cache and bypass selected requests. RACB uses bypassing technique in both L1D and L2 cache according to the resource availability in these caches. Note that all the schemes employs the widely used FR-FCFC scheduling [8] at the memory controllers. Therefore, memory request reordering opportunity before DRAM is exploited in all the schemes.

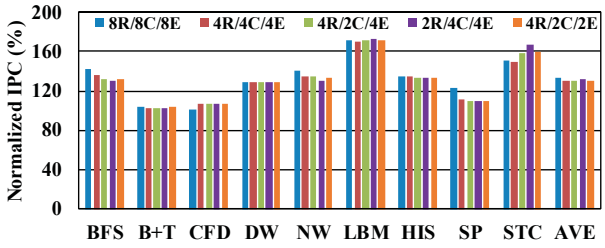


Fig. 7: Finding good performance-cost tradeoff for CART.

VI. Results and Analysis

A. Exploring CART Design Space

While CART works better with more leaf queues, it may not necessary to provide a large number of queues, particularly with cost consideration. To gain insight on how to identify a good trade-off design between the resource consumption and performance improvement, we have examined the impact of different leaf queue numbers and queue sizes on performance. Since the number of branch is fixed to one branch per bank, we only need to explore the design space of row numbers, column numbers and queue sizes (entry numbers). Figure 7 plots the impact on performance improvement over baseline architecture for several memory-intensive benchmarks by using various configurations. nR denotes that there are at most n different row addresses in the leaf queues belonging to each bank. nC means that for the leaf queues belonging to a specific bank and row address, there are at most n different column addresses of memory requests; the nE represents the entry numbers in each FIFO queue.

It can be seen that, more resource leads to higher performance improvement for CART. However, the difference is not very large, demonstrating that CART does not need an impractical number of queues for each row and column combination. Figure 7 shows a diminishing return when adding more resources. In fact, the average performance improvement drops only by 2.0% when the resource for CART is reduced from 8R/8C/8E to 4R/2C/2E, which is a resource reduction of 96.9% (512 entries vs. 16 entries). While not shown in the figure, providing an extremely large number of leaf queues and entries (approaching ideal performance) has a performance improvement within 3% of the 4R/2C/2E configuration. Based on this study, we select 4R/2C/2E as the current configuration of CART in our further evaluation. The table I summarizes the configuration of CART.

B. Performance Comparisons

Our performance comparison consists of two main parts. The first part is to evaluate how effective the proposed CART is when applied alone, compared with other schemes (baseline, MRPB and RACB) applied alone. The second part is to evaluate if CART can complement other schemes to improve the efficiency of memory subsystem. As with other works in the area [18], we plot the results of memory-intensive and compute-intensive benchmarks separately.

First of all, Figure 8 shows the overall performance improvement of different schemes for the memory-intensive benchmarks. Compared with the baseline, the proposed CART alone can achieve 34.2% average IPC improvement (geometric mean). For some benchmarks, such as Transpose (TRP), CART even achieves 2.26X IPC improvement. This shows that actively reordering the incoming requests to a cache-friendly and DRAM-friendly order can help relieve the pressure on memory subsystem. MRPB mainly works on the L1D cache efficiency by reordering the cache in a cache-friendly order and bypassing upon associativity-stalled requests. MRPB improves the average IPC of memory-intensive benchmarks by 23.4%. In addition, RACB focuses on the resource-aware L1D and L2 cache bypassing. When the resource in L1D or L2 cache is no longer available, the bypassing is activated. The average IPC in RACB increases by 12.6% over the baseline. Therefore, the proposed CART performs better than the other two schemes by a large margin.

Second, as the proposed CART improves the efficiency of both L2 cache and DRAM, the combination of CART and MRPB or RACB can explore the benefits across L1D, L2 cache and DRAM. Figure 8 also shows the performance improvement of combined schemes. The combination of CART and MRPB can improve the average IPC by 38.6%, and the combination of CART and RACB can achieve average IPC improvement by 41.5%. Those two improvements are much higher than applying MRPB and RACB alone. This illustrates that the proposed scheme exploits an new opportunity that is largely complementary to existing ones.

We also examine the effect on compute-intensive benchmarks by using different schemes. The results are shown

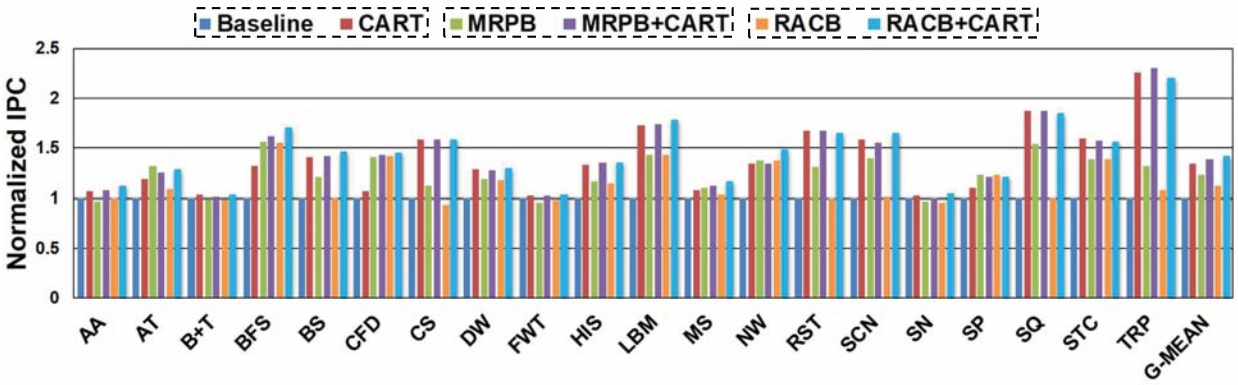


Fig. 8: Performance comparison of different schemes for memory-intensive benchmarks.

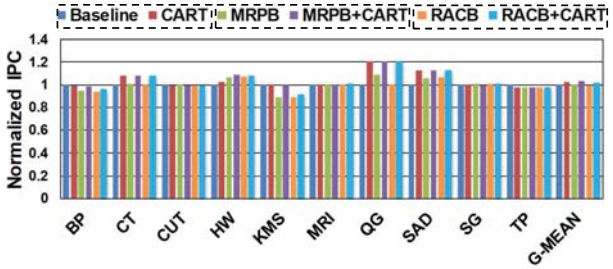


Fig. 9: Perf. comparison for compute-intensive benchmarks.

in Figure 9. The performance improvement of MRPB and RACB are both within the 1.0%. CART is slightly better with an average improvement of around 2.5%, although some benchmarks have observed larger improvement (e.g., 20.8% in QG and 12.7% in SAD). These results on compute-intensive benchmarks are understandable as they are insensitive to memory. For a fair comparison, when considering all the memory-intensive and compute-intensive benchmarks together, CART is still able to achieve an average improvement of 26.5%.

C. Insight of Performance Improvement

Several factors may contribute to the performance improvement of CART: reduction of row buffer conflicts, improvement in L2 hits, and increase of bank utilization. The proposed CART pursues the benefits of more row buffer hits by giving a high priority in the drain policy to the memory requests that have the same row addresses as the previously accessed row. Figure 10a compares the number of row buffer conflicts in the baseline architecture and CART. Many memory intensive benchmarks are observed to have a reduction of row buffer conflicts. On average, the benchmarks with CART have 12.3% decrease in row buffer conflicts. This decrease helps to reduce the time in replacing the content of row buffers, thereby increasing DRAM efficiency. Similarly, Figure 10b and Figure 10c show the impact of CART on L2 cache hit and by bank utilization, respectively. Note that, while not all the benchmarks have improvement in all the three aspects, we have observed and verified that each benchmark has large improvement in at least one of the aspects.

D. Hardware Implementation

We use Cacti 6.5 [20] and RTL implementation to estimate the area and timing of CART. All the key components of CART are implemented and evaluated for hardware cost, including the SRAM-based leaf FIFO queues, the address extractor, the comparators in leaf queues (eight parallel comparators to allow an incoming request to be selected and written into a leaf queue in each cycle), the request selector to reflect the drain policy, etc. With all the components together, CART incurs 0.016 mm^2 per L2 cache partition under 45 nm, which is only 1.7% relative to each L2 cache partition. In comparison, MRPB adds 10.5% hardware overhead (also relative to L2).

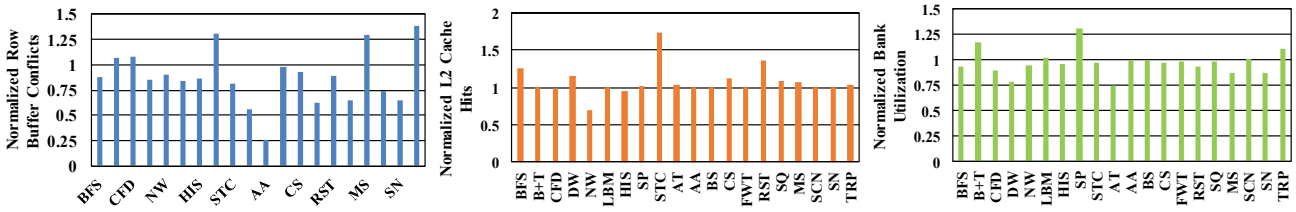
E. Energy efficiency

Due to the small hardware overhead, the proposed CART has very limited overhead on power consumption. As a result, the overall energy consumption of the GPU is reduced due to the shortened execution time. We lumped the CART power overhead in GPUWattch, and simulation results show that CART reduces the energy consumption by 18.9%, compared with the conventional design.

VII. Related Work

Cache bypassing: Several studies have focused on cache bypassing to alleviate cache pressure for GPUs. Xie et al. [21] use compilers to analyze cache utilization of the code based on specific metric and select related instructions for cache access and bypass. Dynamic cache bypassing is also proposed [4], [5], [11], [12]. Besides using compilers, Xie et al. [11] propose to bypass memory requests in a thread block based on the ratio of thread blocks that cache or bypass at runtime. A data locality monitoring mechanism is developed by Li et al. [12] to select highly reusable data to be stored in L1D cache. Jia et al. [4] and Dai et al. [5] both use resource aware technique to bypass memory requests that are stalled in cache to increase memory efficiency. However, all those cache bypassing schemes do not consider the possible impact on DRAM, whereas our work increases DRAM working efficiency by improving DRAM BLP and reducing row buffer conflicts during filling and draining.

Warp scheduling: The memory efficiency can be also improved by optimizing the warp scheduler (e.g., [18], [22], [23]). We evaluated several warp schedulers (GTO, LRR, two-level). The proposed CART achieves similar relative improvement for different warp schedulers, indicating that CART is not sensitive to warp scheduling. It might be that the effects of warp scheduling on the access order has been degraded (filtered) by the L1 cache before L2.



(a) Reduction of Row Buffer Conflicts. (b) Improvement of L2 Cache Hits. (c) Increase of Bank Utilization.

Fig. 10: More details on Performance Improvement.

Data-locality optimizing: Besides passively bypassing memory requests, an active optimization, MRPB (memory request prioritization buffer [4]), is proposed that actively reorders the memory requests in the L1D cache to a cache friendly order to increase cache hits. Also, a memory access scheduling policy is proposed [13] to reduce the negative impact brought by inter-thread interference. This improves the throughput of DRAM. As our proposed CART aims to change the memory requests to a cache-friendly and DRAM-friendly order before entering L2 cache, CART can be used to complement those schemes, as exemplified in evaluation.

Software-level: Software-level schemes can also be used to improve GPU memory subsystem. Streamline is proposed [24] to resolve irregular memory references and control flows through data reordering and job swapping in software. Another work [25] proposes a new algorithm to minimize non-coalesced memory accesses. However, requests to L2 may come from different SMs, thus revealing new reordering opportunity. This is exploited in our scheme that reorders memory requests before entering L2 cache. It is completely done in hardware without the need for application profiling.

VIII. Conclusion

The order of memory accesses plays a significant role in determining the efficacy of memory subsystem. In this work, we propose Cache Access Reordering Tree (CART), a novel architecture that actively reorders memory requests in L2 cache to relieve the congestion of L2 and to increase DRAM working efficiency. The proposed CART is able to improve the IPC of a wide range of memory-intensive workloads by 34.2%. The results also show that the proposed scheme can complement other memory subsystem optimization techniques to further improve system performance.

Acknowledgments: We sincerely thank the reviewers for their helpful comments and suggestions. This research was supported, in part, by the National Science Foundation grants #1566637, #1619456, #1619472 and #1750047.

References

[1] O. Kayran, A. Jog, and M. T. K. C. R. Das, "Neither more nor less: optimizing thread-level parallelism for gpgpus," in *PACT*, 2013.
[2] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *IISWC*, 2012.
[3] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt, "Characterizing and evaluating a key-value store application on heterogeneous cpu-gpu systems," in *ISPASS*, 2012.
[4] W. Jia, K. A. Shaw, and M. Martonosi, "Mrpb: Memory request prioritization for massively parallel processors," in *High Performance Computer Architecture (HPCA)*, 2014.
[5] H. Dai, C. Kartsaklis, C. Li, T. Janjusic, and H. Zhou, "Racb: Resource aware cache bypass on gpus," in *SBAC-PADW*, 2014.
[6] K. K. Chang and et al., "Understanding latency variation in modern dram chips: Experimental characterization, analysis, and optimization," in *SIGMETRICS*, 2016.
[7] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems," in *International Symposium on Microarchitecture(MICRO)*, 2006.
[8] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *ISCA*, 2000.
[9] J. Tuck, L. Ceze, and J. Torrellas, "Scalable cache miss handling for high memory-level parallelism," in *MICRO*, 2006.
[10] M. Jahre and L. Natvig, "Performance effects of a cache miss handling architecture in a multi-core processor," 2007.
[11] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated static and dynamic cache bypassing for gpus," in *High Performance Computer Architecture (HPCA)*, 2015.
[12] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou, "Locality-driven dynamic gpu cache bypassing," in *International Conference on Supercomputing(ICS)*, 2015.
[13] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *International Symposium on Microarchitecture(MICRO)*, 2007.
[14] 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.
[15] 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.
[16] 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.
[17] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.
[18] A. Sethia, D. A. Jamshidi, and S. Mahlke, "Mascar: Speeding up gpu warps by reducing memory pitstops," in *High Performance Computer Architecture (HPCA)*, 2015.
[19] J. Lee and H. Kim, "Tap: A tlp-aware cache management policy for a cpu-gpu heterogeneous architecture," in *High Performance Computer Architecture (HPCA)*, 2012.
[20] S. J. Wilton and N. P. Jouppi, "Cacti: An enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits (JSSC)*.
[21] X. Xie, Y. Liang, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on gpus," in *ICCAD*, 2013.
[22] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *MICRO*, 2011.
[23] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *International Symposium on Microarchitecture(MICRO)*, 2012.
[24] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for gpu computing," in *ASPLOS*, 2011.
[25] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, "Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu," in *PPoPP*, 2013.