MRPB: Memory Request Prioritization for Massively Parallel Processors

Wenhao Jia
Princeton University
wjia@princeton.edu

Kelly A. Shaw
University of Richmond
kshaw@richmond.edu

Margaret Martonosi
Princeton University
mrm@princeton.edu

Abstract

Massively parallel, throughput-oriented systems such as graphics processing units (GPUs) offer high performance for a broad range of programs. They are, however, complex to program, especially because of their intricate memory hierarchies with multiple address spaces. In response, modern GPUs have widely adopted caches, hoping to providing smoother reductions in memory access traffic and latency. Unfortunately, GPU caches often have mixed or unpredictable performance impact due to cache contention that results from the high thread counts in GPUs.

We propose the memory request prioritization buffer (MRPB) to ease GPU programming and improve GPU performance. This hardware structure improves caching efficiency of massively parallel workloads by applying two prioritization methods—request reordering and cache bypassing—to memory requests before they access a cache. MRPB then releases requests into the cache in a more cache-friendly order. The result is drastically reduced cache contention and improved use of the limited per-thread cache capacity. For a simulated 16KB L1 cache, MRPB improves the average performance of the entire PolyBench and Rodinia suites by 2.65× and 1.27× respectively, outperforming a state-of-the-art GPU cache management technique.

1. Introduction

Massively parallel, throughput-oriented systems such as graphics processing units (GPUs) successfully accelerate many general-purpose programs. However, programming GPUs is still difficult, particularly due to their complex memory hierarchies. These hierarchies contain many conceptually separate memory address spaces, including some that require explicit programmer management.

Originally, GPUs had little or no general-purpose caching because conventional GPU workloads such as graphics applications interleave many independent threads to hide memory latency and use small read-only texture caches to save memory bandwidth [5, 6, 11]. Recently, as GPU application domains have broadened, general-purpose read-write caches have been introduced on modern GPUs both to service the increasingly diverse and irregular access patterns [20] and to service regular access patterns without using programmer-managed scratchpads (Section 5.6).

Unfortunately, despite continual industry efforts at improving GPU cache design (Section 5.8), current GPU caches often have mixed or unpredictable performance impact [13, 21, 26]. Their design is still very similar to the latency-optimized CPU caches; their low per-thread capacity, associativity, and other resources are easily overwhelmed by the large number of incoming requests issued by GPU applications, making GPU caches a system bottleneck and causing performance unpredictability. For example, for a typical 16KB GPU L1 data cache, the average cache capacity per thread is only a few dozen *bytes*, as compared to several *kilobytes* per thread in a CPU. Furthermore, typical 4–16 way set-associativity and a few dozen MSHRs might be enough for CPUs, but they struggle to cope with the high thread counts and SIMD widths in GPUs. With many active GPU threads, a conventionally-designed GPU cache, which attempts to service all threads equally and as fast as possible, is highly susceptible to thrashing.

Our work proposes hardware modifications tailored for caches in massively parallel, throughput-oriented systems. Our overarching design goals are: 1) increase the *effective* per-thread cache capacity, 2) reduce contention for limited cache resources such as associativity, 3) exploit the fact that overall memory request processing throughput is more important than latency of individual requests, and 4) exploit the weak GPU memory consistency model.

Due to the large number of active threads in GPUs, significantly increasing per-thread capacity by increasing the physical size of a cache is difficult. If physical capacity remains constrained, then memory request prioritization must be our core approach. Allowing all of a GPU's many threads equal access to caches often causes severe cache contention (Section 2), slowing down every thread. To remedy this, we judiciously prioritize a subset of the active threads to fit their working set into the cache and speed up their execution. Once these threads are finished, other threads are given priority to the cache. This prioritized cache use reduces contention, increases per-thread cache utility, and improves overall system throughput.

We propose a hardware structure called a memory request prioritization buffer (MRPB), which employs two request prioritization techniques: *request reordering* and *cache bypassing*. Request reordering allows MRPB to rearrange a memory reference stream so that requests from

related threads are grouped and sent to the cache together. This significantly enhances memory access locality. Furthermore, by letting the cache focus on a subset of threads at a time, the effective working sets become smaller and more likely to fit in the cache. MRPB's second component, cache bypassing, selectively allows certain requests to bypass the cache, when servicing these requests is predicted to lead to thrashing or stalls. When bursts of conflicting memory requests occur, cache bypassing increases request processing throughput and prevents caches from becoming congested. Together, reordering and bypassing significantly reduce cache contention and improve system throughput.

This paper makes the following contributions:

1) We identify a key weakness in conventional GPU cache designs: inadequate tailoring for high thread counts leads to thrashing- and stall-prone GPU cache designs. We present a taxonomy to characterize this inefficiency.

2) We propose a hardware structure, MRPB, which is light on resources and energy-efficient, requires no change to the rest of the system, and has no dynamic learning cost. *For a 16KB L1 cache, MRPB improves the geometric mean IPC of PolyBench and Rodinia programs by 2.65× and 1.27× respectively. Many memory-sensitive workloads see more than 10× and as much as 17.2× IPC improvement.*

3) MRPB has simpler hardware and better performance than a state-of-the-art technique, cache-conscious wavefront scheduling (CCWS) [24]. In particular, while CCWS only targets cross-warp cache contention, MRPB targets both cross-warp and intra-warp cache contention, improving performance for a broader set of programs.

In the remainder of the paper, Section 2 characterizes GPU cache contention. MRPB design issues are discussed in Section 3. Using Section 4's methodology, Section 5 explores the MRPB design space and presents performance results. Section 6 details implementation issues. Finally, Section 7 describes related work, and Section 8 concludes.

2. Characterization of GPU Cache Contention

2.1. Baseline GPU Cache Request Handling

We first describe how a typical GPU cache handles memory requests. This is critical for understanding sources of cache contention and our work. Figure 1 illustrates a typical GPU design used as the baseline in our study[1]. A GPU has multiple SIMD cores called *streaming multiprocessors* (SMs). Inside each SM, a scalar front end fetches and decodes instructions for groups of threads called *warps*. All threads in a warp have consecutive thread IDs and execute concurrently on the SIMD backend. Multiple warps form a *(thread) block*. During the issue pipeline stage, a *warp scheduler* selects one of the ready warps to issue to the execution/memory pipeline stage. There are several memory

[1]Throughout this paper, we use NVIDIA GPU terminology, but the ideas apply to a broad range of massively parallel systems.

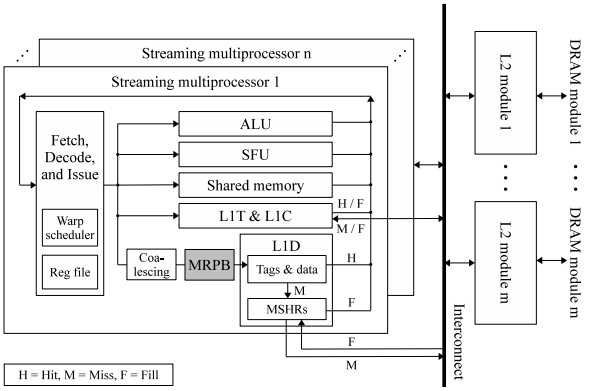

Figure 1. Baseline GPU with MRPB added.

storage units. *Shared memory* is a local, program-managed scratchpad memory. Constant and texture caches (L1C and L1T) are used for special read-only constant and texture data. The remaining memory operations transfer content to and from the DRAM called *global memory*.

All global memory requests from all threads in the same warp are *coalesced* into as few L1 cache line-sized, non-redundant requests as possible, before accessing the L1 data cache (L1D). On a cache hit, data are immediately sent to registers. On a miss, the *miss status holding registers* (MSHRs) are checked to see if the same request is currently pending for another warp. If not, a cache line and an MSHR entry are allocated for this request (potentially evicting an existing cache line) before the request is sent to DRAM via the interconnect; otherwise, a new request need not be sent, and MSHRs ensure the returned request (a cache *fill* event) services both warps in addition to filling in the allocated cache line. A cache miss handler may fail due to several resource unavailability events: all lines in a cache set are allocated and not filled yet, no MSHR space available, miss queue full, etc. The cache-set-full problem is an issue only for allocate-on-miss caches, while *the other problems occur for all caches*, including both allocate-on-miss and allocate-on-fill caches. In such events, the cache cannot immediately process the request and the memory stage is stalled. This request will retry every cycle until needed resources free up. *One goal of our work is to reduce memory stalls by reducing such resource unavailability events.*

A global interconnect connects all SMs to statically partitioned DRAM modules, each with its own L2 cache. GPUs have a weak memory consistency model [20]: requests from the same thread must be seen in order, but requests from different threads can be arbitrarily interleaved between synchronization barriers.

2.2. Terminology and Taxonomy

To motivate MRPB design choices, we first characterize GPU cache contention and thread behavior. Figure 2 illustrates our terminology using a very simple code example, the `atax` kernel in PolyBench (Section 4). While seem-

```
// Thread blocks are one-dimensional with 256 threads.
// A[], temp[], x[] and y[] are of type float.
// A[] is a 2048-by-2048 2D matrix stored as a 1D array.
__global__ kernel_1 {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    for (int i = 0; i < NY; i++) // NY = 2048
        temp[tid] += A[tid * NY + i] * x[i]; }
__global__ kernel_2 {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    for (int i = 0; i < NX; i++) // NX = 2048
        y[tid] += A[i * NY + tid] * temp[i]; }
```

Figure 2. `atax`'s two kernels access `A` in orthogonal directions, causing conflict misses and memory stalls.

```
// Thread blocks are of size 16-by-16.
// J[N][N] is of type float.
// N is a large multiple-of-two number.
__shared__ float temp[16][16];
// All 16 target rows in J map to the same cache set.
temp[threadIdx.y][threadIdx.x] = J[N * 16 * blockIdx.y +
    16 * blockIdx.x + N * threadIdx.y + threadIdx.x];
```

Figure 3. `SRAD` shows a common way to preload shared memory that causes 16-way cache-set contention.

ingly pathological, our results show similar scenarios occur frequently in GPUs, where thread counts are very high relative to associativity. (For example, Figure 3 shows a common way of loading data into shared memory that results in many memory pipeline stalls in a cold allocate-on-miss cache.) In Figure 2, on each loop iteration, the combined threads in a warp load a 32-element column vector from matrix A. These 32 elements have the same row offset i, spreading over 32 consecutive rows of A. Recall that in a 4-way set-associative 16KB L1 cache, memory addresses that are 4KB apart map to the same cache set. Because A's width (NY $\times$ 4B = 8KB, assuming 4B floats) is a multiple of 4KB, all 32 elements loaded by a warp will map to the same cache set, causing many conflict misses. Finally we note that because a warp's threads must operate in lockstep, such conflicts can be recurring and difficult to avoid. Software techniques such as padding can sometimes but not always eliminate these conflicts (Section 2.3). We categorize GPU contention as follows.

Intra-warp (IW) Contention: In GPU kernels that exhibit intra-warp contention, contending cache requests are from the 32 threads in the same warp. Because the overall access footprint of 32 threads does not usually exceed cache capacity, intra-warp contention primarily causes conflict misses. In Figure 2, the conflict misses caused by threads in the same warp accessing the same cache set are considered intra-warp contention. Compared to prior work, MRPB is unique in tackling intra-warp contention. In particular, *no warp schedulers by nature can resolve this contention.*

Cross-warp (XW) Contention: In GPU kernels that exhibit cross-warp contention, contending cache requests are from threads in different warps. Because multiple warps are involved, their total access footprint can easily exceed cache capacity. Therefore, misses caused by cross-warp contention are frequently capacity misses, though conflict misses are a secondary concern. For those workloads that can run multiple blocks concurrently on an SM, warps in these workloads may contend both within and across blocks. Thus, we split cross-warp contention into two types: *cross-warp, intra-block* and *cross-warp, cross-block*.

2.3 Limitation of Software Techniques

Veteran GPU programmers often painstakingly tailor a program's memory accesses to a particular GPU. However, compared to an automated hardware solution such as MRPB, software techniques have limitations.

First, software techniques are applied on a case-by-case basis and may not work for specific algorithms. For example, neither transposing A nor padding A—two common optimizations—eliminates the unintended conflict misses in Figure 2. The former fails because the two kernels in `atax` use A in conflicting ways, so transposing A would help one but hurt the other. Padding fails because even though cache set conflicts no longer occur, other finite resources such as MSHRs and miss queues still limit how fast requests can be processed, causing stalls. In fact, on a real GPU, an NVIDIA Tesla C2070, we verified that no amount of padding to A can noticeably improve `atax`'s performance.

Second, even when software techniques work, applying them imposes extra programmer effort and may introduce new issues. For example, because transposing and padding are ruled out for Figure 2, a programmer may try to load A into shared memory. However, it is easy to create the situation shown in Figure 3, i.e. shared memory loads conflicting with each other, especially if the programmer is not highly experienced with GPU programming. In other words, a hardware solution has high ease-of-programming benefits.

Finally, software techniques often introduce platform-specific code tailoring that is detrimental to cross-platform performance portability. For example, Che et al. showed that different OpenCL platforms often prefer vastly different and even conflicting code optimizations, such as column-major vs. row-major storage order [3]. This makes a case for platform-specific hardware solutions like ours that automatically speed up generic cross-platform codes.

2.4. MRPB Performance Potential

Based on the preceding definitions, we quantify contention's impact in order to motivate MRPB's performance potential. When a cache miss occurs (after the initial cache fill), a resident cache line, often originally requested by a different thread, must be evicted. Depending on the relationship between these two request/eviction threads, each cache miss can be categorized as one of the three types described in Section 2.2. Figure 4 shows the results of this categorization for PolyBench applications on Base-S. (See Table 1 for our two baseline caches. Experiments on Base-L yield similar results.) Six applications (`2dconv`, `2mm`, `3dconv`, `3mm`, `fdtd`, and `gemm`) show low MPKI; we categorize them as cache-insensitive (CI). The next four applications (`atax`, `bicg`, `gesummv`, and `mvt`) experience

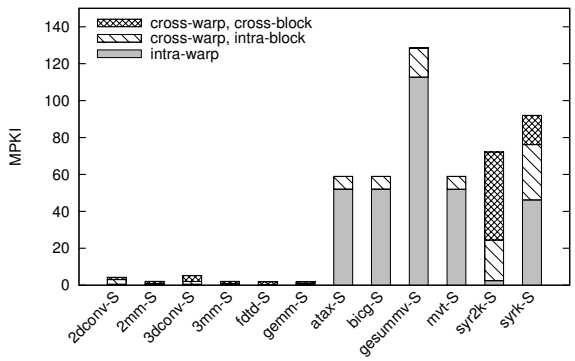

Figure 4. Types of cache contention for each application, in misses per thousand instructions (MPKI).

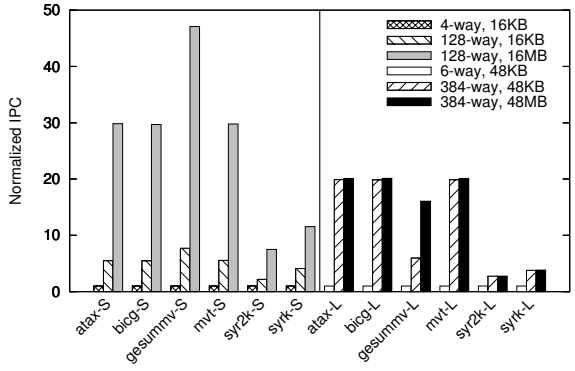

Figure 5. Fully associative or larger caches reduce conflict and capacity misses and improve IPC.

primarily intra-warp (IW) contention. Because these four applications cannot run more than one block concurrently on an SM due to input size limits, their weak cross-warp contention is of the intra-block type. For the remaining two benchmarks, `syr2k` and `syrk`, more than half of their cache misses are due to cross-warp (XW) contention.

We next focus on the cache-sensitive (IW and XW) applications. Our goal is to determine how their cache performance would be improved by increases in either cache associativity or capacity, to further confirm the role of conflict and capacity misses in IW and XW application performance. Figure 5 shows normalized IPC for three variations of each of the two baseline configurations (Base-S and Base-L). In addition to the baseline configuration, the second bar in each three-bar group holds cache capacity constant but makes the cache fully associative, i.e. 128-way for Base-S and 384-way for Base-L. Then, relative to full associativity, the third bar makes the capacity 1024× larger. The second bar is difficult to implement, but shows the severity of conflict misses by eliminating them. The third bar (even more difficult to implement) should further eliminate most if not all capacity misses relative to the second.

Intra-warp (IW) Contention: The IW applications (`atax`, `bicg`, `gesummv`, and `mvt`) show substantial IPC improvements when the cache becomes fully associative,

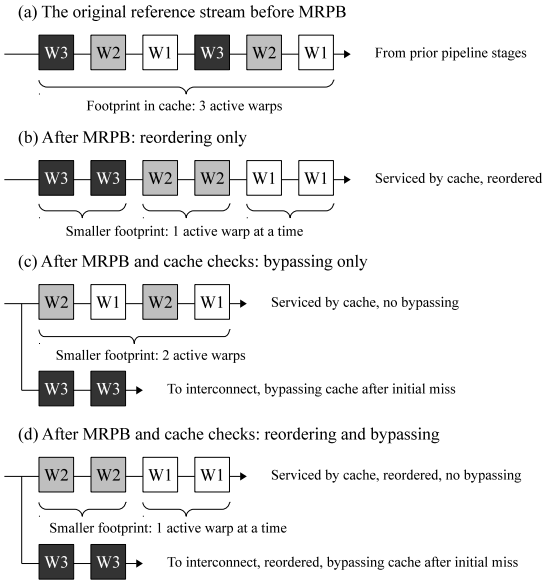

Figure 6. Reordering and bypassing are two request prioritization techniques that reduce cache footprints of active warps. A box's color and the number after W both indicate which warp a request comes from.

even without increasing its size (first bar vs. second bar in Figure 5). This is particularly true for Base-L. The intra-warp contention in these applications is effectively eliminated by full associativity. However, building fully-associative L1 caches is difficult, and MRPB offers similar results with simpler hardware. No prior work has attacked these intra-warp contention effects.

Cross-warp (XW) Contention: For `syr2k` and `syrk`, full associativity alleviates cache demands, but a larger cache improves performance even more for Base-S by eliminating all XW contention in the form of capacity misses. Because increasing cache associativity or capacity is expensive, MRPB uses a more cost-effective method, request prioritization, to reduce XW contention.

3. MRPB: Design Issues

Our goal is to achieve higher computational throughput by minimizing traffic between caches and memory, which is achieved through lower cache contention and higher cache hit rates. Memory request prioritization is at the core of our approach. MRPB employs two different prioritization techniques: request reordering and cache bypassing.

Request reordering, shown in Figure 6(b), rearranges memory requests (boxes in the figure) from a thrashing-prone reference stream into a more cache-friendly order. Requests from the same source (e.g. warp, block, etc.) are grouped together. These requests are likely to access the same cache lines, resulting in better cache hit rates and less contention. Our approach conforms to the GPU memory consistency model by ensuring that requests are reordered only between warps and never across barriers.

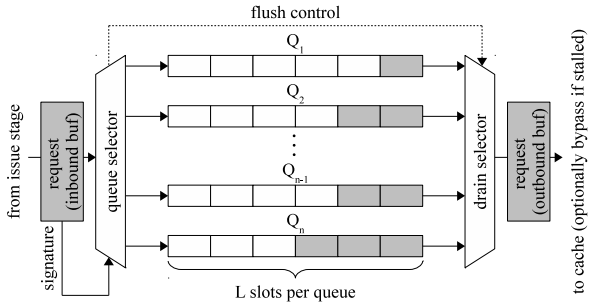

Figure 7. System diagram of MRPB with shading indicating occupied queue entries.

Cache bypassing, shown in Figure 6(c), lets the most severe contention-inducing requests forego filling cache entries on cache misses, to reduce harmful cache pollution. Unlike reordering, bypassing does not change the order of requests in the original reference stream. Because GPU L1 caches are not write-back and not coherent between SMs, a memory request can safely bypass a cache on a cache miss. From the cache's perspective, the warps issuing the non-bypassed requests have priority to use the cache.

As shown later in the paper, reordering is more effective at reducing cross-warp contention and associated capacity misses, and bypassing is more effective at reducing intra-warp contention and associated conflict misses. The approaches can be used together, shown in Figure 6(d). We now detail their individual designs and their combined use.

3.1. System Diagram

As shown in Figure 7, MRPB is composed of a number of first in, first out (FIFO) queues. Each queue holds outstanding memory requests. Before entering the MRPB, these requests have left the issue stage of the SIMD pipeline and have been coalesced, but they have not yet accessed the L1 cache. Requests are placed into a particular queue based on a *signature* value. Signatures can take forms such as block ID, warp ID, etc. Each queue is assigned a unique signature value. A queue selector computes the signature of each incoming request and enqueues the request at the queue of that signature. If that queue is full, the memory pipeline stage is stalled. In each cycle, a designated *drain policy* is used to select one request from one particular queue to pass to the cache. These queues, together with the signature and the drain policies, accomplish the reordering part of prioritization. Bypassing is implemented independent of reordering. After a request misses in the cache, an optional cache bypass action can be taken.

3.2. Reordering: Signatures

MRPB places requests into different queues using a signature-based mapping function. The selected signature should satisfy several conditions. First, every request must have one and only one signature value. Second, requests with the same signature should be more likely to access the same memory content. Third, signatures should never allow for the potential reordering of requests from the same thread. Finally, signatures should be simple to compute.

Because memory requests from the same warp or the same block might possess more locality than requests from different warps or different blocks, it is reasonable to use a request's warp ID or block ID as its signature. These signatures are easy to compute (from bits in the request header) and have clear meanings. However, entirely different classes of signatures, such as target memory addresses or program counter values, are possible. We have experimented with several signatures, but due to space constraints, present results for three simple, effective signatures: *block ID*, *warp ID*, and *in-block warp ID*.

Using block ID or warp ID as the request signature is self-explanatory: take the block ID or warp ID of the thread that issued the request. To compute a request's in-block warp ID, we first determine which warp issued the request. The in-block warp ID of the request is the ordinal position of this warp *within* its parent thread block. If multiple thread blocks are running on an SM, the i-th warp of each of these blocks has an identical in-block warp ID of i.

Because in-block warp ID does not distinguish between different thread blocks, it does not handle cross-block contention. However, a proper drain policy can enable in-block warp ID to prioritize some warps within a thread block over other warps from the same block, handling cross-warp, intra-block contention. In comparison, block ID alone handles cross-block contention but does not handle cross-warp, intra-block contention, since all of a block's warps have the same ID. Warp ID combines the attributes of both signatures: it handles both cross-warp, intra-block and cross-block contention by giving each warp a unique ID.

3.3. Reordering: Drain Policies

After memory requests are enqueued based on their signature values, a drain policy imposes a service priority between these queues. One straightforward drain policy is to apply a *fixed-order* priority, i.e. an item from queue n can be drained only if queues 1 to $n-1$ have been emptied. At the other extreme, a *round-robin* policy selects items from queues in turn, e.g. one item per non-empty queue. Another policy is *longest-first*, which always selects an item from the longest non-empty queue, breaking ties with queue ID. Longest-first is more fair than fixed, because it prevents queues from becoming too long, and prioritizes more than round-robin, because a burst of requests into the same queue would get serviced sooner by longest-first.

To investigate *how many items* should be drained from a queue before moving to a different queue, we consider greedy variations of these policies. A greedy variation must empty a queue before it can use the underlying basic policy to select the next queue. For example, a greedy-then-longest-first policy must drain all items from a queue, after

which it selects the current longest queue and repeats the process. In comparison, our non-greedy policies can switch to a different queue after just one item is drained. (Other variations draining N items at a time are also possible.)

An additional option on the drain policy is the *flush* option. It deals with the finite queue length and reduces storage needs by letting the buffer store only read requests. The flush logic has three parts. 1) When a new read request cannot enter a full queue, the drain policy controller prioritizes the full queue and drains one item from that queue to make space for this request. This ensures that when a burst of requests attempts to enter a low-priority queue with a finite length, they can be handled without having to wait for high-priority queues to be processed first. 2) A flush is triggered whenever an incoming request is a write request, regardless of how many items are in the target queue. In this situation, the controller empties the target queue and then services the incoming write request, rather than enqueuing it. This is because write requests are large. (After coalescing, write requests are up to 136 bytes—128B data plus an 8B header—compared to read requests which only have an 8B header.) Flush-before-write reduces queue size requirements because writes never need to be stored, and it ensures requests in the same queue are not reordered relative to the write. Because only one request is drained per cycle, a flush operation may take multiple cycles to complete depending on how full the target queue is. 3) If neither of the flush conditions above is met, the normal drain policy is consulted to select the next queue to drain.

3.4. Bypassing

While a signature and a drain policy together fulfill the reordering part of prioritization, bypassing is an independent technique that achieves a different form of prioritization. In Section 2.1's baseline architecture, a missed request must allocate resources before the miss can be processed: a cache line, an MSHR entry, a miss queue entry, etc. If *any* of these resources are unavailable, the missed request cannot be processed, stalling the pipeline. This request must be retried in subsequent cycles until resources are available to process it. With cache bypassing turned on, such read requests can be immediately sent to main memory through the interconnect, avoiding congesting the pipeline. Correspondingly, when the responses return, the data are written directly to the registers without filling the cache.

An important design decision concerns *when* to bypass. Among resource unavailability stalls, the cache *associativity stall* has particularly severe impact. On a cache miss, an associativity stall occurs when all cache lines in the target set have been allocated and are waiting to be filled by outstanding requests to main memory, making it impossible to allocate a cache line and causing the memory stage to stall. These stalls happen when a burst of requests access the same cache set in a short period of time, as in Figure 2. After responses for the first few requests return and fill the cache, they are immediately evicted by other stalled requests, while the rest wait. As described in Section 2.1, even in allocate-on-fill caches these associativity stalls can still turn into other types of stalls due to other resource limits such as MSHRs and miss queues. Worse still, if all the requests are from the same warp, they must remain in lockstep, so this thrashing can recur often, congesting the pipeline and causing severe slowdowns even for unrelated threads. Because threads within a warp operate in lockstep, request *reordering* is often not helpful; bypassing mitigates this intra-warp contention.

	Base-S	Base-L
# of SMs	14	
# of threads / warp	32	
Warp schedulers	2 round-robin warp schedulers / SM	
Per-SM limit	1536 threads, 48 warps, 8 blocks, 32 MSHRs	
Per-block limit	1024 threads, 32 warps	
L1D / SM	16KB, 4-way	48KB, 6-way
Shared memory / SM	48KB	16KB
Unified L2	6 × 128KB, 16-way	
DRAM	FR-FCFS scheduler, GDDR5	
L1D/L2 parameters	128B line size, LRU	
L1D/L2 policies	alloc-on-miss, write-evict L1D, write-back L2	
L1D/L2 request size	8-byte reads, 136-byte writes	
SM/DRAM clock	1150/750 MHz	

Table 1. Baseline configurations of simulated GPU.

3.5. Design Summary

Table 4 summarizes the design options explored in Section 5. The entries per queue and buffer delay options are explained in Section 5.3. Overall, MRPB leverages several key GPU attributes to work around the low per-thread cache capacities, associativities, and other resources. In particular, request reordering and prioritization are motivated in part by the GPU attribute that throughput is far more important than latency. Our MRPB design has the same peak request processing throughput as the baseline design: in steady state, MRPB can enqueue and service the same number of requests per cycle. In addition, MRPB abides by GPU memory coherence and consistency models, but exploits their relative weakness to improve cache request footprints and conflict characteristics.

4. Experimental Methodology

Simulation environment: We use a cycle-level simulator, GPGPU-Sim 3.2.0 [1], to evaluate our MRPB designs. An NVIDIA Tesla C2050 GPU is simulated with two baseline configurations (key parameters in Table 1, remainder in GPGPU-Sim release). Baseline configuration S (Base-S) models a 16KB 4-way set-associative L1 cache per SM. Baseline configuration L (Base-L) models a 48KB 6-way set-associative L1 cache per SM. In both cases, because the L1 cache and the shared memory in the same SM share 64KB storage in total, the shared memory capacity is set accordingly. These two configurations are taken from current NVIDIA GPUs and represent distinct design points in terms of resources dedicated to caches. With industry trend-

Name	Program	Type	SF
2dconv	2D convolution	CI	1
2mm	2-matrix multiply	CI	1/4
3dconv	3D convolution	CI	1
3mm	3-matrix multiply	CI	1
fdtd	2D finite different time domain	CI	1/8
gemm	General matrix multiply	CI	1
atax	Matrix-transpose-vector multiply	IW	1/2
bicg	BiCGStab linear solver subkernel	IW	1/2
gesummv	Scalar-vector-matrix multiply	IW	1/4
mvt	Matrix-vector-product transpose	IW	1/2
syr2k	Symmetric rank-2k operations	XW	1/16
syrk	Symmetric rank-k operations	XW	1/4

Table 2. PolyBench benchmarks. SF column shows scaling factors of default inputs.

Name	Program	Input
BP, BP/U	Backpropagation	65536 nodes
BFS	Breadth-first search	65536 nodes
BT	B+ tree	1M nodes
GM	Gaussian matrix	208×208 elements
HS, HS/U	Hotspot	512×512 elements
KMS	K-means clustering	204800 34D points
LUD, LUD/U	LU-decomposition	512×512 elements
NW, NW/U	Needleman-Wunsch	2048×2048 elements
PF, PF/U	Pathfinder	$100K \times 100$ elements
SRAD, SRAD/U	Graphic diffusion	1024×1024 pixels

Table 3. Rodinia benchmarks. All inputs are chosen from default inputs. "/U" represents unshared versions.

ing towards smaller *per-thread* cache capacity, we evaluate MRPB against both baselines to show broad applicability.

Benchmarks (PolyBench and Rodinia): Tables 2 and 3 summarize the benchmarks used in this study, from PolyBench [9] and Rodinia [2]. We simulate all GPGPU-Sim-compatible PolyBench and Rodinia applications using default grid and block dimensions as well as default inputs, scaled if necessary. Other work has used similar input scaling [14, 19]. As in [24], we simulate 17M–3.7B GPU instructions (to program completion).

PolyBench includes GPU ports of CPU multi-threaded programs. These programs have highly diverse and very intensive memory access patterns, resulting in relatively high performance sensitivity to caching. In Section 2.4, we grouped these applications into three types based on our characterization of their cache behavior: cache insensitive (CI), primarily intra-warp contention (IW), and primarily cross-warp contention (XW) (shown in Table 2).

Rodinia's programs have been carefully hand-tuned—often extensively using shared memory—and are thus less sensitive to caching. For some Rodinia programs that use shared memory, we create a separate "*unshared*" version, which uses global memory instead. These unshared programs represent a potential use of MRPB-like structures—developers may spend less time optimizing programs to use shared memory if caches can achieve high efficiency automatically (Section 5.6).

5. Experimental Results

In this section, we first explore MRPB's design space using PolyBench, making sure we do not *harm* the performance of CI applications while improving the performance of both IW and XW applications. (PolyBench applications exhibit cache-sensitivity expected in future cross-platform heterogeneous code without much platform-specific tuning and in code written by non-GPU specialists.) Due to space limits we omit 10 out of 12 -L PolyBench applications because they show similar behavior as their -S counterparts.

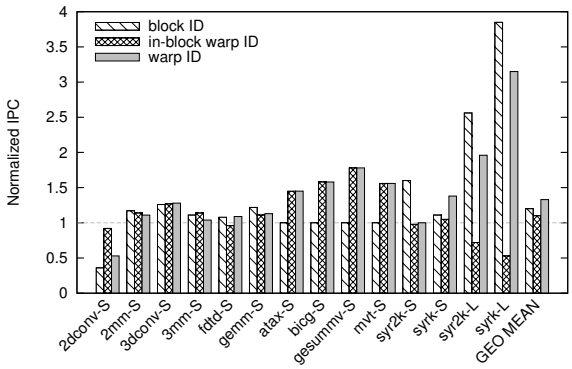

Figure 8. Block ID and in-block warp ID signatures reduce cross-warp, cross-block and cross-warp, intra-block contention respectively. Warp ID reduces both.

We then evaluate the best MRPB design (found from PolyBench) on Rodinia applications, which are overall better optimized and less cache-sensitive than PolyBench applications. The results show MRPB's ability to improve the performance of even relatively well-optimized GPU code.

Benchmarks are ordered as CI (left), IW (middle), and XW (right). All IPC values in this section are normalized against Base-S for -S applications and Base-L for -L applications. All average numbers are geometric means.

5.1. Signatures

The purpose of a signature is to place memory requests into different queues, such that requests in the same queue have improved access locality and reduced cache contention. We evaluate three signatures: block ID, in-block warp ID, and warp ID. As in Table 1, a block can have at most 32 warps, and thus we use 32 queues in MRPB when in-block warp ID is used as a signature. Likewise, block ID and warp ID use 8 and 48 queues respectively. To evaluate these three signatures, we use a fixed-order drain policy (Section 5.2 shows this is the best), unlimited queue sizes and a 3-cycle buffer latency (studied in Section 5.3), and no bypassing (evaluated in Section 5.4).

Figure 8 shows the effectiveness of these signatures on improving IPC. All three signatures work well on the types of contention they target and have minimal impact on most cache insensitive applications. Because the IW applications experience some cross-warp, intra-block contention but no cross-warp, cross-block contention, their IPC improves only when in-block warp ID and warp ID signatures are used. Because XW applications exhibit cross-warp, cross-block contention, their IPC improves when block ID

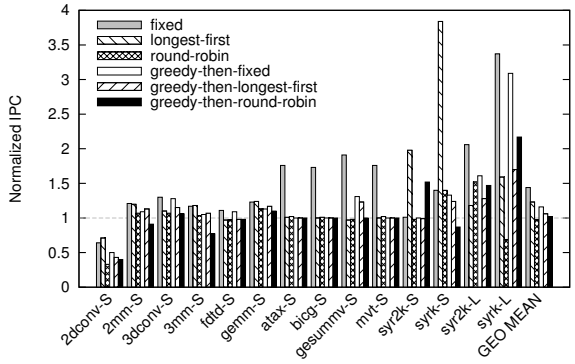

Figure 9. A fixed-order drain policy provides the best overall performance improvement.

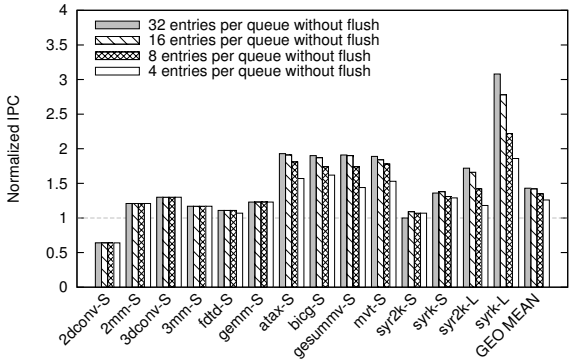

Figure 10. With the flush option off, most programs are insensitive to the number of entries per queue.

and warp ID signatures are used. At smaller cache sizes, these applications experience some cross-warp, intra-block contention, which the in-block warp ID signature can impact slightly, but the lack of this type of contention at larger capacities makes the in-block warp ID signature ineffective.

Over all applications, warp ID performs the best, but uses moderately more queues. In fact, it is a hybrid whose performance is usually close to the better of the other two, particularly true for IW and XW-L applications. *Thus the rest of the paper uses warp ID as MRPB's request signature.* One notable exception is `2dconv-S` which prefers in-block warp ID instead of warp ID as a signature, but its performance is saved by bypassing in Section 5.4 because bypassing serves a similar purpose as in-block warp ID.

5.2. Drain Policy

Figure 9 shows the performance of all 6 policies assuming unlimited queue lengths and no flushing. In these experiments, warp ID is used as the signature. Notably, longest-first performs very well for XW because warps in these applications frequently issue intense bursts of requests, which are easily captured by a longest-first drain policy. However, longest-first's advantage on XW applications is diminished when the buffer size takes a more realistic value in Section 5.3. Overall, the fixed-order policy performs the best on average and for most programs. This is because *the fixed-order policy introduces the most queue prioritization*, which is beneficial for reducing cache footprints and improving performance. We select it as MRPB's drain policy.

5.3. Buffer Size, Flush, and Buffer Latency

Thus far, we have selected a signature (warp ID) and a drain policy (fixed-priority-order) while assuming queues in the buffer have unbounded capacity in order to hold as many read and write requests as needed. Here we explore how sizing and resource choices impact MRPB performance.

With the flush option still off and a 3-cycle buffer latency, we let the queues have a finite length, though each entry is still wide enough to hold either a read or a write request. Figure 10 shows that for most applications short queue

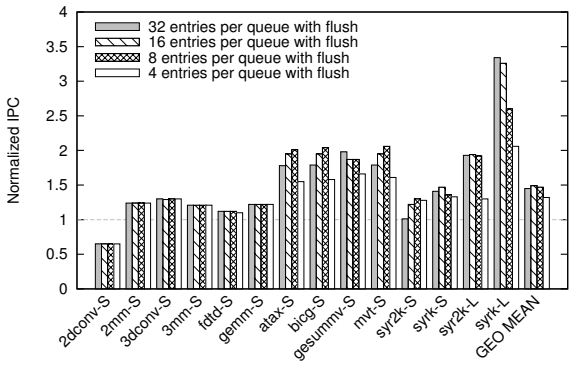

Figure 11. Smaller buffers benefit from the flush option.

lengths suffice, because real applications achieve some degree of coalescing and, hence, a warp usually issues fewer than 32 requests. However, a few applications in IW and XW-L do benefit from longer queue lengths. To balance resource requirements and performance improvements, we select 8 entries per queue as the buffer size.

When the flush option is used, writes no longer enter the queues, and each queue entry can be made much smaller to hold just reads. Figure 11 shows the flush option slightly improves application performance on buffers with smaller queue lengths. For example, `atax`, `bicg`, `mvt`, and `syr2k-L` have improved IPC values at 8 entries in Figure 11 compared to Figure 10. Because the flush option is very helpful in reducing buffer entry width without hurting performance, the final MRPB design uses this option.

Figure 12 (with 8 entries per queue and flush on) explores MRPB's processing latency; MRPB is highly latency-insensitive. Buffer delays (from enqueuing to dequeuing the same request) of 3 to 18 cycles can all be tolerated with little performance impact. Thus, while we have explored fairly simple enqueue signatures and drain selectors, one could implement more sophisticated policies within these ample cycle counts. For the rest of this paper, our models conservatively assume a 5-cycle MRPB enqueue-to-dequeue latency, but faster enqueue/dequeue options are available.

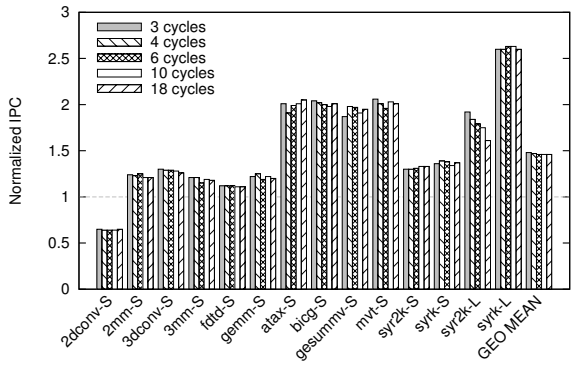

Figure 12. IPC is largely insensitive to buffer latency.

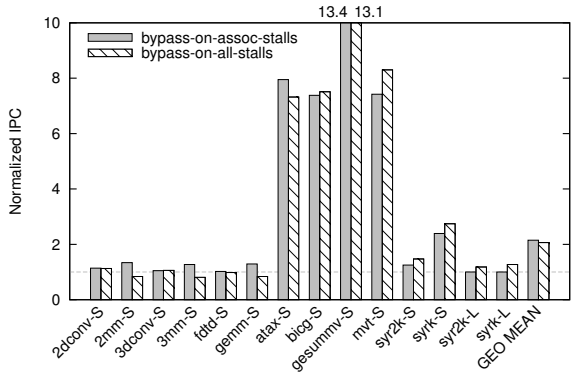

Figure 13. Bypassing reduces impact of IW contention.

5.4. Bypass Option

Prior sections have explored MRPB's reordering, and this section now *explores the impact of bypassing with no reordering used*. Section 5.5 then combines the two approaches. Here, MRPB simply has a single slot that accepts one incoming read or write request and sends it to the cache every cycle. If the cache rejects this request due to resource limits, MRPB can choose to let this request bypass the cache. Otherwise, this request stalls the memory stage and retries every cycle until the cache accepts it.

Figure 13 shows the IPC effects of two different bypassing options: bypass-on-all-stalls and bypass-on-associativity-stalls. We make three observations. 1) Bypassing works well for IW applications. IW applications have strong intra-warp contention; bypassing is the primary way to resolve such contention. 2) Bypassing is ineffective at dealing with cross-warp contention. Bypassing too aggressively denies warps from accruing caching benefits and the bypassed requests can cause congestion in the lower-level memory subsystem. In particular, XW-L applications see little performance improvement from bypassing as they do not have much intra-warp contention and suffer from resource contention in the L2 cache. 3) Bypassing can degrade the performance of the CI applications. Of the two bypassing options, bypass-on-associativity-stalls is less aggressive and causes less performance degradation on CI applications. We use it in the final design.

Parameters	Values
Signature	block-ID (resulting in 8 queues), in-block ID (resulting in 32 queues), **warp ID (resulting in 48 queues)**
Drain policy	**(non-greedy-)fixed-order**, greedy-fixed-order, (non-greedy-)longest-first, greedy-longest-first, (non-greedy-)round-robin, greedy-round-robin
Flush option	off, **on**
Buffer size	4–32 entries per queue, with **8** finally chosen
Buffer latency	3–18 cycles, with **5** finally chosen
Bypass option	off, bypass-on-all-stalls, **bypass-on-assoc-stalls**

Table 4. MRPB design space. Final choices in bold.

5.5. Overall Performance

The final MRPB design has the bold parameter values in Table 4. Figure 14 shows results for this design over all PolyBench and Rodinia applications. Because bypassing reduces the impact of intra-warp contention while reordering handles both types of cross-warp contention, the combined techniques obtain larger performance improvements than either alone. Figure 14 shows our final MRPB design improves the geometric mean IPC of PolyBench by 2.65× over Base-S and by 2.23× over Base-L. It also improves the geometric mean IPC of Rodinia by 1.27× over Base-S and by 1.15× over Base-L. Some applications see an IPC improvement of more than 10× and up to 17.2× in one case.

While MRPB offers impressive speedups against both baselines, Base-S benefits more from MRPB than Base-L. This is mainly because Base-S's smaller (16KB) and less associative (4-way) L1 cache is more likely to experience contention than Base-L's larger (48KB) and more associative (6-way) counterpart. More cache contention gives MRPB more opportunities to reduce conflict and improve IPC.

The performance improvements for Rodinia are significant though smaller than those for PolyBench because Rodinia programs are more thoroughly optimized. In particular, many Rodinia programs are hand-tuned so well that the memory subsystem's efficiency has little performance impact. Even so, MRPB still improves the IPC of a few original and modified Rodinia programs by more than 2×.

Further analysis shows that MRPB achieves such effective IPC improvements by significantly reducing memory traffic and consequently increasing system throughput. For PolyBench-S and Rodinia-S programs, using MRPB results in 26.7% and 15.4% L2-to-L1 memory traffic reductions (in terms of the number of interconnect packets) respectively. Meanwhile, the average cache miss counts are reduced by 54.6% and 37.9% for PolyBench-S and Rodinia-S respectively. -L benchmarks show similar results.

Figure 14 also shows the IPC values achieved by a state-of-the-art GPU cache management technique, CCWS's static wavefront limiting (CCWS-SWL) [24]. MRPB performs better over all four scenarios in terms of geometric mean IPC improvements. It also has a simpler design with lower hardware requirements. One factor in achieving these gains is MRPB's ability to reduce the impact of intra-warp contention. SWL does not consider intra-warp contention.

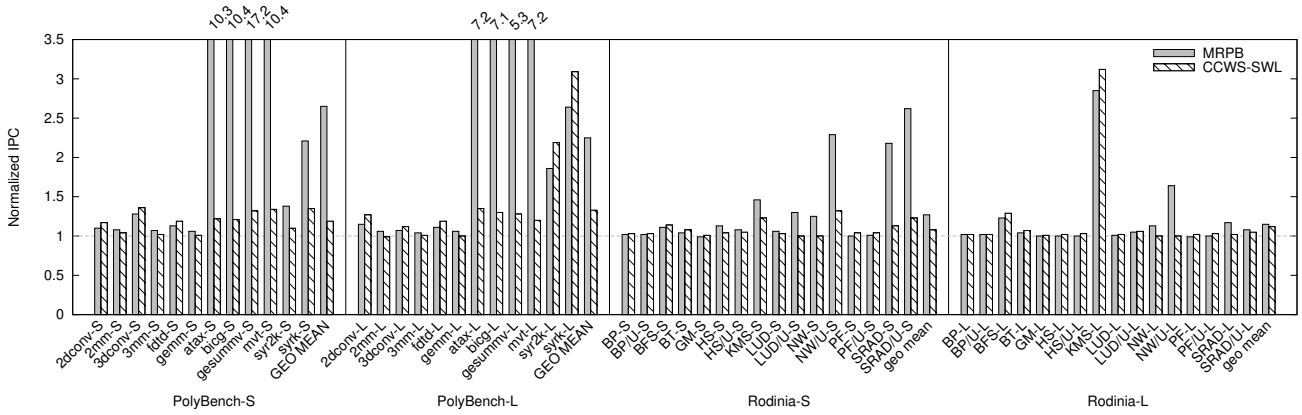

Figure 14. MRPB improves PolyBench and Rodinia IPC substantially over two baselines.

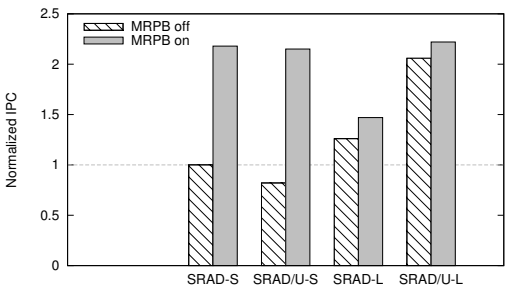

Figure 15. MRPB enables simpler code with better performance. (Normalized to `SRAD-S` with no MRPB.)

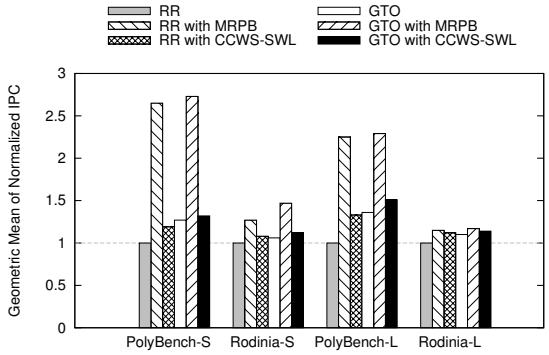

Figure 16. MRPB improves application performance over different warp schedulers. Results are normalized within each of the four groups of bars.

Consequently, MRPB often outperforms SWL by several times for applications with significant intra-warp contention (e.g. PolyBench's IW and Rodinia's Base-S programs).

Finally, MRPB never degrades performance below baseline in any tested scenario, particularly for CI applications.

5.6. Improving Ease of Programming

Rodinia benchmarks are heavily tuned to use programmer-managed shared memory, requiring significant programmer effort. In current GPUs, codes using shared memory usually perform better than those that do not, but MRPB allows the latter to approach or outperform the former. Figure 15 shows the performance of the original `SRAD` along with `SRAD/U` which does not use shared memory. On Base-S with no MRPB, there is a trade-off between programming difficulty and performance: the unshared version is easier to write but performs 18% worse. However, on Base-S with MRPB, both versions perform much better. On Base-L without MRPB, `SRAD/U` performs better than `SRAD`, because `SRAD`'s shared memory usage limits its concurrent thread block count. Turning MRPB on further improves performance for both cases.

Across all 8 scenarios, the *best* performance is achieved with the *unshared* `SRAD/U` running on a 48KB L1 cache *with MRPB*. Similar results are seen for `HS`. In fact, while the six unshared programs run 37% slower than the baseline on average, turning on MRPB reduces the performance gap to only 9%. This makes a case for using MRPB-enabled caches instead of shared memory, to ease programming without significant performance sacrifices.

5.7. Interaction with Warp Schedulers

Prior sections used the default round-robin (RR) warp scheduler; here we consider other warp schedulers. To our knowledge, the cache-conscious warp scheduler (CCWS) is the only warp scheduler that specifically targets GPU caches [24]. It preserves intra-block access locality by reducing cross-warp cache contention. The authors propose two schedulers: a simpler static wavefront limiting (SWL) scheduler and a more complex victim cache-based scheduler. We compare MRPB against SWL because it is the better performing of the two schedulers. Both CCWS schedulers use a base scheduler, and [24] finds a greedy-then-oldest (GTO) warp scheduler to be better than a RR scheduler because GTO preserves intra-warp access locality. We evaluate *both* MRPB and SWL on top of RR and GTO. Figure 16 shows the performance of these four combinations plus the two baselines. While GTO alone significantly outperforms RR, MRPB improves IPC substantially over GTO and RR. Notably, MRPB outperforms SWL on top of both GTO and RR. The intra-warp contention targeted by MRPB cannot be eliminated by any warp scheduler; warp schedulers by nature only target cross-warp contention.

5.8. Comparison with High Associativity

General-purpose caches are a relatively new addition to GPUs, and they continue to see ongoing design changes. One design trend is the attempt to mitigate associativity contention. For example, AMD's Graphics Core Next (GCN) GPUs use 64-way associativity for their 16KB L1 caches. We have simulated a 64-way set-associative 16KB L1 cache with a fairly idealistic 1-cycle hit latency, and MRPB still outperforms this cache by a geometric mean of 4% for all -S benchmarks. Thus, MRPB is an effective alternative to high-associativity caches, especially considering a straightforward associativity increase has high latency and energy costs while MRPB can more cost-effectively achieve high *effective* associativity.

6. Hardware Implementation Details

Figure 7 illustrates a possible MRPB design. While we refer to MRPB in terms of logical FIFO queues, such FIFOs are usually implemented as SRAM [22], with head and tail "pointers" addressing a queue like a circular buffer. The FIFO queues (one per warp) are 48 logical rows in SRAM (although their physical layout would be more square). To select a queue to write to, the warp ID (or other signature) is used as a logical row address. For addressing within these areas, each queue has its own read (dequeue from head) and write (enqueue at tail) addresses. Enqueue/dequeue pointers are small, well-studied shift registers [22].

The largest MRPB we consider is less than 3KB of SRAM, and it considerably outperforms caches simply enlarged by that amount. For example, increasing Base-S to 20KB capacity (5-way set associative) only improves IPC by 1.2× for PolyBench and 1.06× for Rodinia. In addition, MRPB also outperforms large increases in associativity as described in Section 5.8.

To dequeue a request, the drain selector unit selects a queue to read from based on its policy, and applies that as a logical row address. The drain selector internally keeps queue lengths as a set of counters, in order to generate queue full/empty signals. A round-robin drain selector can simply rotate between the row addresses, while the fixed-order policy can be implemented as a 48-to-6 priority encoder with full/empty signals as input. MRPB can operate in a split-cycle fashion, with each cycle supporting first an enqueue (write) operation followed by a dequeue (read) operation. (Dual-porting is another option, but the split-cycle approach uses less area.) Drain latency may vary with the implemented drain policy, but the throughput is not sensitive to this. As Figure 12 shows, MRPB performance is insensitive to enqueue/dequeue latency. Finally, requests in the outbound buffer are considered for bypassing based on the resource-conflict stall signal from the cache. Future work can consider richer bypassing policies using MRPB internal states such as request signatures.

We estimate MRPB's timing and area with Cacti 6.5 [18]. The proposed design can operate at 2.6GHz, more than twice the GPU's 1.15GHz core frequency and enough to support a split-cycle operation. The circuit uses 0.0126 mm^2 per core at 45nm or 0.04% of the simulated Tesla GPU's die area. Relative to the area cost of a 16KB L1 cache with an area-efficient design (e.g. a 4- rather than 1-cycle hit latency), MRPB only adds 10.5% to area. It adds even less relative to a larger (e.g. 48KB) or faster cache.

MRPB also compares favorably against CCWS, outperforming it despite using less area (0.026 mm^2 per core in [24] or at least 0.0141 mm^2 per core with Cacti 6.5 at 45nm). Likewise, MRPB significantly outperforms caches with more capacity, even when those options are conservatively afforded more space than MRPB.

Finally, we consider MRPB's energy efficiency. Although SRAM-based, MRPB does not need a large part of conventional cache logic circuitry such as tag computation and comparison. Thus MRPB consumes less power than a similarly-sized cache. Furthermore, as stated in [24], cache management techniques such as CCWS and MRPB have important power saving leverage because they eliminate many power-hungry cache misses. Since MRPB reduces cache misses by up to 55%, it offers substantial power improvements for the GPU memory subsystem.

7. Related Work

Warp Scheduling: Some related work improves warp schedulers to optimize GPU memory access efficiency [19, 24]. Other work improves warp schedulers mainly targeting branch divergence instead of memory accesses [7, 8, 17]. Among the former, a two-level warp scheduler [19] runs a subset of warps (called a fetch group) as long as possible until they all hit long-latency operations. CCWS [24] improves upon this by using a victim cache tag array to infer cross-warp cache contention. The warp scheduler periodically uses this information to dynamically throttle the active warp count. In contrast, MRPB attacks intra-warp (in addition to cross-warp) contention, which *cannot* be addressed by warp schedulers. Further, because MRPB attacks cache contention in the cache hierarchy itself, it is highly-effective, despite requiring simpler hardware. As Section 5.5 showed, MRPB outperforms CCWS, which outperforms the two-level warp scheduler.

Software Techniques: Several software studies have characterized or optimized GPU cache performance. [26] notes that blindly turning on L1 caches on Fermi GPUs may occasionally harm instead of improve application performance. Some other studies have proposed analytical models for estimating optimal active thread counts [4, 10]. Our prior work proposed a GPU memory access locality taxonomy and a compile-time algorithm to improve caching utility [13]. Some techniques adjust GPU program data layout

to reduce main-memory-to-core traffic [3, 25]. In contrast, MRPB needs no profiling or access to program source code.

Other Cache Management: Our work also relates to memory request prioritization outside the GPU domain itself. In CMP and CPU-GPU heterogeneous systems, prior work includes sophisticated replacement algorithms, bypassing techniques, and cache placement optimization to improve cache hit rates of multi-threaded CPU workloads [12, 16]. TAP modulates cache usage when GPU programs share part of the hierarchy with CPU programs, but its goal is to protect CPU runtimes, not optimize GPU performance [15]. Other work prefetches into the GPU hierarchy [14], which does not help memory bandwidth-limited applications MRPB targets. MRPB also relates to traditional DRAM scheduling algorithms [23]. Compared to prior work, our work mainly aims to improve the caching and runtime efficiency of a GPU program itself, rather than coordinate its interactions with CPUs.

8. Conclusion

This paper proposes the memory request prioritization buffer, a hardware structure that can effectively improve the utility of cache for a massively-parallel, throughput-oriented program. We first characterize GPU cache contention to motivate MRPB. We then describe the various design issues of MRPB and numerically demonstrate the implication of different design choices. Our final MRPB design, chosen based on a systematic design exploration, improves the geometric mean IPC of PolyBench and Rodinia suites by 2.65× and 1.27× respectively over two evaluated baseline configurations.

GPU cache trends have proven difficult to predict, but all indications are that increases in thread counts and computational density will likely outpace increases in cache capacity, leading to ongoing decreases in per-thread cache capacity. Consequently, we believe that techniques such as MRPB that effectively prioritize cache usage and mitigate cache performance unpredictability will continue to grow in importance.

Acknowledgements: This work was supported in part by funding and donations from NSF, DARPA, the C-FAR STARNet center, and equipment donations from NVIDIA and AMD. We thank Kevin Skadron, Daniel Lustig, and the anonymous reviewers for their feedback.

References

[1] A. Bakhoda et al. Analyzing CUDA workloads using a detailed GPU simulator. In *Intl. Symp. on Performance Analysis of Systems and Software*, 2009.

[2] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *Intl. Symp. Workload Characterization*, 2009.

[3] S. Che, J. W. Sheaffer, and K. Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *Intl. Conf. High Performance Computing, Networking, Storage and Analysis*, 2011.

[4] H.-Y. Cheng et al. Memory latency reduction via thread throttling. In *Intl. Symp. Microarchitecture*, 2010.

[5] W. J. Dally et al. Merrimac: Supercomputing with streams. In *ACM/IEEE Conf. Supercomputing*, 2003.

[6] K. Fatahalian and M. Houston. A closer look at GPUs. *Communications of the ACM*, 51(10):50–57, October 2008.

[7] W. W. L. Fung and T. M. Aamodt. Thread block compaction for efficient SIMD control flow. In *Intl. Symp. on High Performance Computer Architecture*, 2011.

[8] M. Gebhart et al. Energy-efficient mechanisms for managing thread context in throughput processors. In *Intl. Symp. Computer Architecture*, 2011.

[9] S. Grauer-Gray et al. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing*, 2012.

[10] Z. Guz et al. Many-core vs. many-thread machines: Stay away from the valley. *IEEE Computer Architecture Letters*, 8(1):25–28, January–June 2009.

[11] Z. S. Hakura and A. Gupta. The design and analysis of a cache architecture for texture mapping. In *Intl. Symp. Computer Architecture*, 1997.

[12] A. Jaleel et al. High performance cache replacement using re-reference interval prediction (RRIP). In *Intl. Symp. on Computer Architecture*, 2010.

[13] W. Jia, K. A. Shaw, and M. Martonosi. Characterizing and improving the use of demand-fetched caches in GPUs. In *Intl. Conf. on Supercomputing*, 2012.

[14] J. Lee et al. Many-thread aware prefetching mechanisms for GPGPU applications. In *Intl. Symp. Microarchitecture*, 2010.

[15] J. Lee and H. Kim. Tap: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *Intl. Symp. on High Performance Computer Architecture*, 2012.

[16] J. Meng and K. Skadron. Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling. In *Intl. Conf. Computer Design*, 2009.

[17] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Intl. Symp. Computer Architecture*, 2010.

[18] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Cacti 6.0: A tool to model large caches. Technical report, HP Laboratories, 2009.

[19] V. Narasiman et al. Improving GPU performance via large warps and two-level warp scheduling. In *Intl. Symp. Microarchitecture*, 2011.

[20] NVIDIA Corp. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 2009. v. 1.1.

[21] NVIDIA Corp. *Tuning CUDA Applications for Fermi*, 2011. v. 1.5.

[22] J. M. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits: A Design Perspective*, chapter 12. Prentice Hall, 2nd edition, 2003.

[23] S. Rixner et al. Memory access scheduling. In *Intl. Symp. Computer Architecture*, 2000.

[24] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-conscious wavefront scheduling. In *Intl. Symp. Microarchitecture*, 2012.

[25] I.-J. Sung, D. Liu, and W.-M. Hwu. DL: A data layout transformation system for heterogeneous computing. In *Innovative Parallel Computing*, 2012.

[26] Y. Torres and A. Gonzales-Escribano. Understanding the impact of CUDA tuning techniques for Fermi. In *Intl. Conf. on High Performance Computing and Simulation*, 2011.