2014 47th Annual IEEE/ACM International Symposium on Microarchitecture

Adaptive Cache Management for Energy-efficient GPU Computing

Xuhao Chen*‡†, Li-Wen Chang†, Christopher I. Rodrigues†, Jie Lv†, Zhiying Wang*‡ and Wen-Mei Hwu†

*State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China
‡School of Computer, National University of Defense Technology, Changsha, China
†Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, USA
{cxh,lchang20,cirodrig,jielv1}@illinois.edu, zywang@nudt.edu.cn, w-hwu@illinois.edu

***Abstract*—With the SIMT execution model, GPUs can hide memory latency through massive multithreading for many applications that have regular memory access patterns. To support applications with irregular memory access patterns, cache hierarchies have been introduced to GPU architectures to capture temporal and spatial locality and mitigate the effect of irregular accesses. However, GPU caches exhibit poor efficiency due to the mismatch of the throughput-oriented execution model and its cache hierarchy design, which limits system performance and energy-efficiency.**

The massive amount of memory requests generated by GPUs cause cache contention and resource congestion. Existing CPU cache management policies that are designed for multicore systems, can be suboptimal when directly applied to GPU caches. We propose a specialized cache management policy for GPGPUs. The cache hierarchy is protected from contention by the bypass policy based on reuse distance. Contention and resource congestion are detected at runtime. To avoid oversaturating on-chip resources, the bypass policy is coordinated with warp throttling to dynamically control the active number of warps. We also propose a simple predictor to dynamically estimate the optimal number of active warps that can take full advantage of the cache space and on-chip resources. Experimental results show that cache efficiency is significantly improved and on-chip resources are better utilized for cache-sensitive benchmarks. This results in a harmonic mean IPC improvement of 74% and 17% (maximum 661% and 44% IPC improvement), compared to the baseline GPU architecture and optimal static warp throttling, respectively.

***Keywords*-GPGPU; cache management; bypass; warp throttling**

I. Introduction

Energy-efficiency has become a critical design requirement as the semiconductor industry moves from multicore to manycore processors. Heterogeneous architectures have been proposed to improve energy-efficiency, because different types of processing engines can be specialized for different types of computation patterns. For example, using CUDA [36] or OpenCL [24] for general-purpose computing on graphics processing units (GPGPUs) has become pervasive in the high performance computing community, due to its improved energy-efficiency over conventional multicore CPUs when dealing with data-parallel kernels. Manufacturers are incorporating hardware and software features into these throughput-oriented accelerators to better support unstructured applications with various memory access patterns. Caches have been included in GPUs to leverage on-chip data reuse, and can provide significant speedup for irregular applications written in a straightforward manner. However, current GPU cache design and its management schemes are inefficient when running memory intensive applications, which hampers system performance and energy-efficiency.

Typical CPU cache architecture is optimized for memory latency, which does not necessarily benefit throughput-oriented processors like GPUs, because massive multithreading makes cache locality difficult to capture [18], [10], [38]. Like CPU caches, GPU caches performance is hampered by *thrashing*, particularly inter-warp contention [39], [19], which is much more common in GPUs due to massive multithreading. GPUs usually have hundreds or thousands of threads running simultaneously. Thus, they exhibit much smaller cache capacity per thread and much shorter cache line lifetimes than CPUs, with many cache lines being replaced before they are reused. Inter-warp contention occurs among warps that are scheduled to the same SIMT core. It leads to poor temporal locality and thereby degrades performance. Since the application working set is usually much larger than the cache size, advanced replacement policies cannot solve the contention problem in GPUs [39].

Cache bypassing [11] has been proposed in CPUs as a thrashing-resistant technique to protect cache lines from early eviction. It selectively bypasses some memory requests to save cache space for other requests and thereby alleviates contention. When applied to GPUs, pure cache bypass policies [11] designed for last level caches (LLCs) in CPUs may not achieve the expected improvement. This is because the massive amount of miss (including bypass) requests may lead to on-chip resource (either MSHR or on-chip network) congestion and thus limit the performance benefit [19].

Thread throttling techniques [43], [9] can effectively alleviate resource congestion in CPUs. *Cache-conscious wavefront scheduling* (CCWS) [39] leverages thread/warp throttling to alleviate inter-warp contention and improve the L1 cache hit rate in GPUs. Two schemes have been proposed: static wavefront limiting (SWL) using statically determined *maximum active warps* (MAW) on each warp scheduler, and dynamaic CCWS that leverages run-time in-

1072-4451/14 $31.00 © 2014 IEEE
DOI 10.1109/MICRO.2014.11

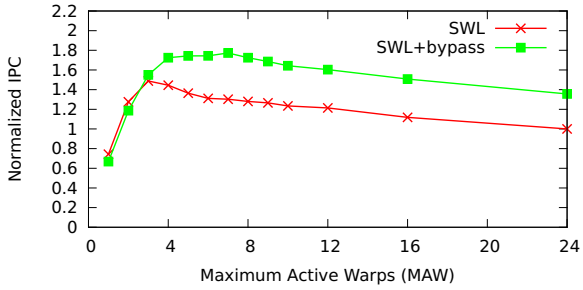

Figure 1. BFS performance of *SWL with bypass-disabled* (`SWL`) and *SWL with bypass-enabled* (`SWL+bypass`), simulated on a Fermi-like GPU architecture (Table II), and all normalized to SWL-24 (static warp limiting with MAW of 24). The bypass policy used here is the state-of-the-art CPU bypass policy [11] that is ported by us to GPUs (described in Section IV).

formation to estimate the optimal MAW. However, pure warp throttling reduces the number of warps and therefore loses the benefits of massive multithreading, e.g. hiding memory latency. It also causes the Network-on-Chip (NoC) and DRAM bandwidth to be underutilized, artificially limiting opportunities to further improve performance.

Both pure bypassing and pure warp throttling have performance limitations. Fig. 1 shows the performance improvement of `BFS` (Breadth First Search) under two schemes: SWL with bypass-disabled (`SWL`) and bypass-enabled (`SWL+bypass`). We change the MAW of SWL from 1 to 24 in our experiment. `BFS` has memory divergence because its access behavior depends on input data, and can not be well coalesced at programming or compile time. The figure illustrates that bypassing improves system performance over `SWL` for any given MAW that is larger than 2. In particular, limiting MAW from 24 to 7 can improve the normalized IPC of `SWL+bypass` from 1.36× to 1.77×. The figure also shows that the optimal MAWs that maximize the performance of `SWL` and `SWL+bypass` are different (3 and 7 respectively). The optimal MAW is increased when enabling bypassing because bypassing allows more warps to issue memory requests without degrading cache performance (since cache lines are protected by the thrashing-resistant bypass policy), and the spare bandwidth is utilized to send requests to the next level of memory hierarchy. This implies that *coordinated bypassing and warp throttling* (CBWT) can further improve performance over pure cache bypassing or pure warp throttling. The same trend is observed for other cache sensitive benchmarks (Fig. 8).

To overcome the limitations of pure bypassing and pure warp throttling, we present a specialized cache management policy for GPGPUs. Cache contention and resource congestion caused by massive multithreading is detected at runtime, and the detector notifies the controller to enable cache bypassing dynamically. The reuse distance-based bypass policy protects hot cache lines from early eviction, and is coordinated with warp throttling to avoid over-saturating on-chip resources. To estimate the optimal number of active warps, we propose a simple predictor using a gradient method based on dynamic sampling of contention and congestion intensity. All these features in our policy are designed on top of a cost-effective cache structure with simple sampling modules. This paper makes the following contributions:

- We characterize cache behavior of massively multi-threaded applications on GPUs, and evaluate existing CPU cache management policies ported to GPU. Experimental results show the performance opportunities as well as limitations.
- We present an adaptive cache management policy, namely CBWT, for GPGPUs. Cache contention and on-chip resource congestion caused by massive multithreading are detected dynamically, based on which the bypass policy is coordinated with warp throttling to take full advantage of the GPU cache capacity and other on-chip resources.
- We propose a simple dynamic predictor for CBWT and implement it in a cycle-accurate simulator. Experimental results demonstrate that it can improve cache performance and better utilize on-chip resources, and therefore improve IPC by 1.74× and 1.17×, compared to the baseline GPU architecture and optimal static warp throttling respectively.

The rest of the paper is organized as follows: the baseline GPU architecture is introduced in Section II. Benchmark characterization and motivation is presented in Section III. Section IV evaluates the state-of-the-art bypass policy on GPU. The proposed management scheme is described in Section V. Section VI presents the experimental results. Section VII discusses related works, and Section VIII concludes.

II. Baseline GPU Architecture

Fig. 2 shows the organization of our baseline GPU architecture, which is similar to a typical modern GPU design as found in NVIDIA or AMD GPUs. Our work is also applicable to the GPU part of HSA-like heterogeneous architectures [25], which usually have smaller GPU cache sizes than discrete GPUs. Note that although we use NVIDIA CUDA terminology in the following sections, our design is also applicable to OpenCL.

Execution Model. In this paper, we assume that kernels execute sequentially, i.e. only one kernel is executed at a time. Each kernel includes groups of threads called *cooperative thread arrays* (CTAs), i.e. *thread blocks* or *work groups*. Within each CTA, subgroups of threads called *warps* are executed in *lockstep* fashion. We call warps running on the same SIMT core *sibling warps*. Warp schedulers are responsible for scheduling sibling warps onto execution units. Many scheduling policies have been proposed, such as loose round-robin (LRR), greedy-then-oldest (GTO) [39], two-level scheduling [32], CTA-aware scheduling [20], cache-conscious scheduling (CCWS) [39], and divergence-aware scheduling (DAWS) [40].

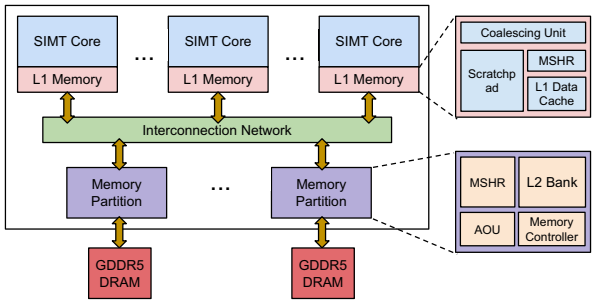

Figure 2. Baseline GPU Architecture

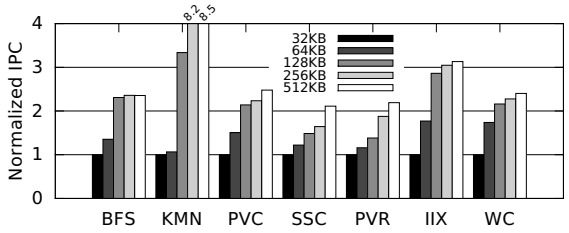

Figure 3. Speedups over 32KB L1 cache as the L1 cache size increases

Benchmarks	Description	Suite
Highly Cache Sensitive (HCS)		
BFS	Breadth First Search	[7]
KMN	K-means Clustering	[7]
PVC	Page View Count	[15]
SSC	Similarity Score	[15]
PVR	Page View Rank	[15]
IIX	Inverted Index	[15]
WC	Word Count	[15]
Moderately Cache Sensitive (MCS)		
SM	String Match	[15]
SPMV	Sparse Matrix Vector Multiply	[42]
FFT	Fast Fourier Transform	[42]
CFD	CFD Solver	[7]
NW	Needleman-Wunsch	[7]
Cache Insensitive (CI)		
SD1	Graphic Diffusion	[7]
BP	Back Propagation	[7]
STL	Stencil	[42]
WP	Weather Prediction	[34]
FWT	Fast Walsh Transform	[34]

Table I
GPGPU BENCHMARKS, CATEGORIZED BY THEIR CACHE SENSITIVITY THAT IS MEASURED BY CHANGING L1 CACHE SIZE (FIG. 3)

Memory Hierarchy. The GPU memory hierarchy consists of register files, L1 memories (scratchpad and L1 data cache), shared L2 cache, and off-chip GDDR DRAM [33], [35], [1]. For simplicity, we do not discuss texture caches or other special-purpose caches in detail. L1 memory is private per-core and shared by sibling warps. Scratchpad memory is programmer visible and is used for explicit intra-CTA communication. The L2 cache works as the central point of coherency, and is shared across all threads of the entire kernel. It is connected to the SIMT cores through an interconnection network, and partitioned into multiple banks that are connected to each memory channel. Memory controllers schedule memory requests that miss in the L2 cache to the DRAM. In both levels, missed requests are recorded by the Miss Status Holding Registers (MSHRs), and then sent to the next level of the memory hierarchy. Atomic operations are performed at each memory partition by the Atomic Operation Unit (AOU).

Throughput-oriented Cache Design. GPU caches are designed for maximizing throughput in terms of memory coalescing, write policy, coherency, inclusion property, etc. GPUs conserve memory bandwidth by grouping requests issued simultaneously from the same warp. All requests are merged, if possible, by a hardware *coalescing unit* before being sent to L1 caches. Thus the spatial locality benefits provided by CPU caches can be largely captured by the coalescing unit in GPUs when programmers are careful with the memory access patterns in their programs. GPU L1 caches typically feature a write-through policy, with [1] or without [33], [35] write-allocation. This policy saves bandwidth compared to a write-back policy [41], [16], since GPU applications have very little reuse on written data. The L2 cache is write-back with write-allocation, which is the same design choice as a conventional CPU LLC. Modern GPUs typically do not provide hardware support for L1 cache coherence to avoid the overhead that coherence messages add to NoC traffic and memory access latency [41], [16]. Current GPU L2 caches do not enforce inclusion. NVIDIA GPU caches are non-inclusive non-exclusive caches [2], [41], meaning cache lines that are brought into L1 caches are also brought into the L2 cache, but an L2 cache line is evicted silently (without recalling L1 caches) when replacement happens. This design reduces the number of redundant data copies than inclusive caches, but still allows L1 caches to locally provide shared input values. Hardware prefetching and victim cache are ruled out according to the micro-benchmarking experiments [3], which are reasonable hardware tradeoffs for GPUs.

III. UNDERSTANDING GPU CACHE INEFFICIENCY

In this section, we evaluate GPU cache performance to understand application behaviors on a cache hierarchy similar to those in current GPUs. We use memory intensive GPU benchmarks selected from Parboil [42], Rodinia [7], MapReduce [15] and the CUDA SDK [34] (listed in Table I). We categorize the benchmarks into highly cache sensitive (HCS), moderately cache sensitive (MCS) and cache insensitive (CI) ones. Note that if an application mostly uses the programmable scratchpad, it is generally not sensitive to L1 data cache size. Although our work mostly focuses on HCS benchmarks, CI benchmarks are included to show the robustness of our design.

Cache contention. Previous CPU cache management studies [17] have highlighted some problems that limit CPU

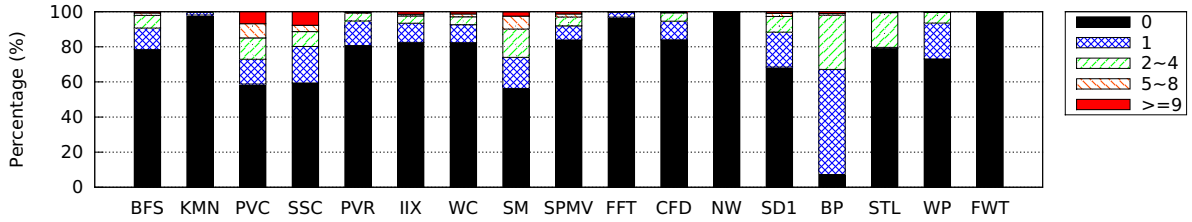

Figure 4. L1 cache reuse count distribution. It shows the number of repeated accesses to cache blocks in L1, after the block is filled in.

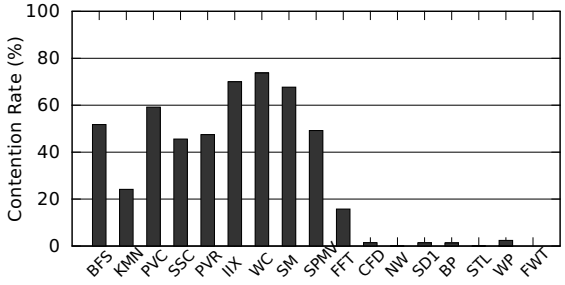

Figure 5. The L1 cache (32KB) contention ratio detected by the L2 cache. A repeated memory request sent from the same L1 cache to the L2 cache is counted as a victim request. The contention ratio is calculated by total number of victim requests over total number of requests to the L2 cache.

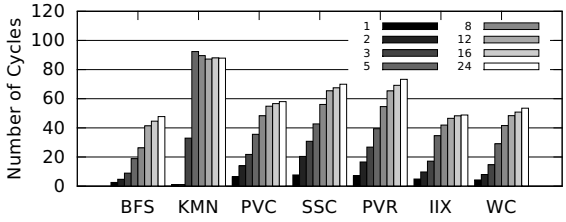

Figure 6. Average NoC latency (number of cycles that a packet takes to walk through the network) changes as MAW increases from 1 to 24.

cache efficiency. One of these problems is thrashing [37], i.e. the working set is larger than the cache size. This becomes worse for GPU caches as their effective size per thread diminishes with a large number of concurrent threads, e.g. Fermi supports a maximum of 48 warps on each SIMT core, and each warp contains 32 SIMT threads, and these up to 1536 threads share 16KB or 48 KB L1 data cache. In GPUs, thrashing can be caused by intra- or inter-warp contention [39], [19]. Inter-warp contention happens when sibling warps contend for L1 cache space that they share and continually replace the data of each other in the cache. Unfortunately, Rogers et al. [39] show that the Belady-optimal policy [5], which always chooses the furthest reused candidate to replace, cannot address the contention problem because the application working set is usually much larger than the cache size, which causes frequent early eviction.

To illustrate how contention significantly impacts HCS applications, Fig. 3 shows the speedup of HCS benchmarks with increasing L1 cache size over the baseline (32KB L1). For these benchmarks, increasing L1 cache size reduces contention and improves performance. Fig. 4 shows the reuse count distribution for the 32KB L1 cache. For most of the benchmarks, a large percentage of the cache lines inserted into the L1 caches are never reused (the zero reuse count). The zero-reuse cache lines are either streaming accesses or victims of early eviction. Fig. 5 presents the L1 contention ratio detected by the L2 cache. This indicates that early eviction happens very frequently for HCS benchmarks. Note that since the contention ratio is detected by the L2 cache, with limited L2 cache size, it might be inaccurate if the L2 cache also has severe contention. KMN is such a benchmark that generates a lot of accesses and causes severe contention in both L1 and L2 caches. However, the measured L1 contention ratio is not very high, because the L2 cache lines are also frequently evicted and only able to accumulate some of the contention requests.

Resource congestion. Massive multithreading can cause *resource congestion* that is not typically seen in CPU caches. This is one of the reasons why cache efficiency does not directly correlate positively with system performance in GPUs. When congestion occurs, memory requests can neither be serviced nor be sent to the next level of memory hierarchy, wihch leads to severe memory stalls and causes computation units under-utilized. In this paper, we focus on NoC bandwidth that may become the bottleneck when bypassing is enabled. When a massive number of requests are generated and miss in L1 cache, they are sent to the L2 cache via NoC and can quickly saturate NoC bandwidth. And then the packet transfer latency becomes dramatically high. This not only hampers performance, but also degrades power-efficiency, because the program gains no speedup (even slows down if considering cache behavior) but NoC is quite power-consuming (close to 10% of the entire system power) [28]. Fig. 6 presents the average NoC latencies of HCS benchmarks change as their MAWs increase from 1 to 24. The NoC latency becomes dramatically high when MAW is larger than 5 for most of the benchmarks, because more concurrent warps generate more requests and quickly saturate the NoC.

IV. Cache Bypassing on GPUs

In CPUs there are several ways to deal with cache contention. Increasing cache size could help, but cannot improve cache efficiency if the application working set

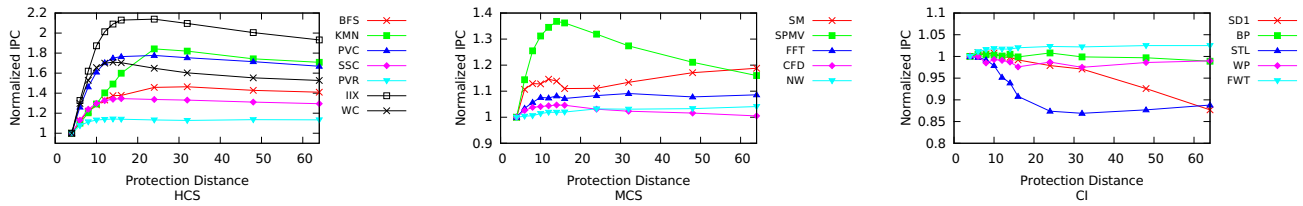

Figure 7. Performance of SPDP-B changes as the protection distance increases, normalized to baseline.

is even larger, and it increases access latency, area, and power consumption. Cache management policies have been investigated to improve cache efficiency [17], [11]. However, advanced replacement policy can not solve GPU cache contention problems, because the effective cache size is too small in the massive multithreading environment [3]. To make good use of the cache space, hot cache lines should be protected, and requests to conflicting addresses have to be bypassed when contention happens. Thus, existing CPU bypass policies [12], [11] can be applied to GPU to avoid pollution and thrashing access patterns.

The state-of-the-art bypass policy, protection distance prediction (PDP) [11], introduces *protection distance* (PD) to protect cache lines. Each line has a remaining PD (RPD) to indicate how many accesses this line will be protected from. When a line is inserted or promoted, its RPD is set to the PD. The RPD is then decremented for each access to the target set. A line is protected only if its RPD is larger than 0. An unprotected line can be replaced by the incoming requests. The cache is bypassed if no unprotected lines are found. Two schemes are introduced: static PDP bypass (SPDP-B) and dynamic PDP. SPDP-B uses a statically determined PD. Dynamic PDP predicts reuse distance (RD) by hardware sampling, and periodically estimates the optimal PD (which maximizes the hit rate) based on RD sampling.

We first implement SPDP-B for GPU L1 caches to show how the benchmarks behave in response to different PDs. We enumerate the PD from associativity to maximum PD (4 to 64). Fig. 7 illustrates the effect on performance. For HCS benchmarks, the optimal PD can lead to significant speedups. However, the curves are flatter than those observed in CPUs [11]. This is because GPU warp interleaving makes the memory behavior much less predictable, and thus the RD is relatively unstable in such a massive multithreading environment. But the RD-based protection mechanism is still beneficial for GPUs, because statistically, GPU memory behavior is still predictable [3]. The relatively flat curve implies that a suboptimal prediction may not lose too much benefit, which is an opportunity to reduce hardware overhead. Note that `STL` and `SD1` have short RDs, and thus need small PDs. A large PD leads to poor performance for them because it protects some cache lines beyond their lifetime, delaying their replacement with useful data.

We also implement dynamic PDP for GPU L1 caches. There are three design choices, PDP-private (PDP-P), PDP-global (PDP-G) and PDP-shared (PDP-S). PDP-P enhances each L1 cache with a PDP module, and each L1 cache predicts PD locally using its private PD predictor. PDP-G is a centralized design, which collects access sequences from all the SIMT cores, and makes a centralized prediction for all L1 caches. PDP-S chooses only one (or several) SIMT core as a sampler. The sampler makes local prediction and applies its PD to all the other SIMT cores.

We evaluate the performance improvement of the three designs. As the most expensive design, PDP-P achieves the best performance (on average 50.6% IPC improvement over baseline on HCS benchmarks), because each L1 cache has its own predictor, which can make a more accurate prediction when SIMT cores have different memory behaviors. PDP-G and PDP-S achieve average IPC improvement of 44.6% and 45.4% respectively, which are close to that of PDP-P. PDP-S performs very similarly to PDP-P for most of the benchmarks. This is because GPU programs usually have similar behavior for all the threads [27], and the optimal PD estimated by one of the SIMT cores is probably also the optimal PD for the rest. In the following sections, we adopt the PDP-S design since it is the cheapest one. Note that load imbalance may lead to different reuse distance for different SIMT cores, and therefore reduce the performance benefit for PDP-S. However, even with suboptimal PD, PDP-S is able to protect hot cache lines and improve cache efficiency, given that GPU applications generally benefit over a broad range of PD values, as shown in Fig. 7.

Despite the potential performance benefit, the PDP bypass policy is less effective in GPUs because of the following limitations. First, the PD prediction, which is intuitively designed for CPU LLCs, is based on a hit rate model [11]. However it is not robust in GPUs due to warp interleaving [8]. As was discussed before, GPU cache performance does not always correlate positively with system performance due to resource congestion and warp interleaving. Therefore a pure hit rate model that leads to better cache performance for single threaded processors may not result in system performance improvement in a massively parallel environment. Second, a massive amount of memory requests sent to the cache subsystem may saturate on-chip resources. When memory divergence occurs, MSHRs can easily become full, and the incoming requests are congested until an MSHR is available, which leads to severe memory stalls [19]. If the requests that bypass cache blocks also

bypass MSHRs, this may reduce the burden of MSHRs, but read accesses to the same cache line would be sent multiple times to the next level of memory hierarchy, which loses locality and wastes NoC or DRAM bandwidth. This is especially serious when memory divergence results in a huge working set and the reuse distance is so large that PDP would bypass a large portion of the memory requests to keep the cache efficient. Then the massive amount of bypassed accesses would be sent through the NoC and DRAM channel, causing heavy traffic and thereby degrading performance.

For the first limitation, since an inaccurate PD prediction usually will not lose too much performance (as shown in Fig. 7), trying to improve prediction accuracy with extra hardware is not cost-effective. However, for the second limitation, we intend to introduce warp throttling to control the amount of requests that are allowed to be issued to the memory subsystem. This control is not possible with pure bypassing mechanism.

V. Adaptive Cache Management Scheme

As we described in Section III, massive parallelism causes cache contention and congestion, leading to inefficient cache usage and unsatisfactory performance on GPUs. This motivates specialized management to handle different access behaviors in GPUs. If contention occurs, the cache controller needs to trigger bypassing to protect hot cache lines long enough to avoid early eviction. However, when bypassing is hampered by resource congestion, the warp scheduler should be notified to throttle multithreading. We first present the bypass policy coupled with the static warp throttling technique, to demonstrate the potential performance benefit of our approach. Then the hardware enhancement needed to detect access patterns is introduced. Finally, we propose the runtime management scheme *coordinated bypassing and warp throttling* (CBWT) which combines bypassing with warp scheduling to dynamically control parallelism and take full advantage of on-chip resources.

A. Bypassing with Static Warp Throttling

Warp throttling alleviates contention by suspending some of the active warps. In SWL scheme, we can find the optimal MAW by static brute-force search. The optimal MAW is different for different benchmarks and changes for each benchmark when its input data is changed [39]. While SWL can reduce the degree of multithreading, bypassing tries to selectively accommodate the cache space with hot cache lines and forward the others to the lower level of the memory hierarchy. When bypassing is enabled together with SWL, there is an opportunity to further improve performance because pure warp throttling may lead to under-utilization of on-chip resources, but cache bypassing can enable more multithreading without degrading cache performance.

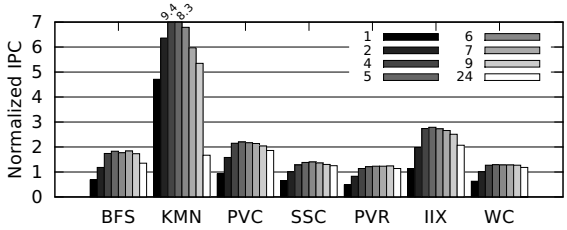

Figure 8. Performance of SWL at various multithreading limits, i.e. MAWs, when bypassing is enabled. Normalized to SWL-24 with bypassing disabled.

Fig. 8 shows the speedups over baseline on HCS benchmarks as the MAW increases, when PDP bypassing is enabled for L1 caches. The scheme that picks the optimal MAW is called Best-CBWT in Section VI. In the figure, we observe that the optimal MAWs for HCS benchmarks are always smaller than 24, because too many concurrent warps cause contention and congestion. However, the performance is also not satisfactory when MAW is too small, due to resource under-utilization. Compared to SWL with bypassing disabled (not shown in the figure), for most of the HCS benchmarks, we find that the optimal MAW is increased when bypassing is enabled. This is because when bypassing is enabled, it is possible to issue more warps without degrading cache performance, since caches are protected by the bypass policy. The extra warps can take advantage of the underutilized NoC and/or DRAM bandwidth, which can potentially improve system performance. Likewise, the new optimal MAW is different for different benchmarks and inputs. This motivates us to find the optimal MAW dynamically.

B. Hareware Extension

To be able to dynamically manage the cache system in response to different access behaviors, the controller should be aware of the feedback information from caches and other on-chip resources. We here introduce the hardware extension that enables access behavior detection. Our detection mechanism tries to leverage existing hardware and thus has a very low hardware cost to implement.

Contention Detection. Detecting thrashing access patterns is essentially detecting early eviction. When an evicted line is reaccessed it is interpreted as an early eviction. The number of early evictions detected in a given period is called the *lost locality score* (LLS) [39]. A high LLS indicates that the hardware should throttle parallelism to reduce thrashing. To dynamically adjust MAW, CCWS introduces the *victim tag array* (VTA) to collect the *lost locality* as feedback. CCWS adds a VTA for each L1 cache to detect prematurely evicted lines. The VTA holds the tags of recently evicted lines. If one of these lines is reloaded into the cache, it indicates that an early eviction occurred.

To reduce hardware cost, we extend the tag array of L2 cache with extra bits. As shown in Fig. 9, an L2 cache entry

consists of these fields: state bits, PD (protection distance), tag, data and *victim bits*. The state bits, PD, tag, and data are functionally the same as those in PDP caches. The victim bits are added to record the lost locality. They are bit masks associated with a cache line where each bit records the access history from a particular L1 cache before the line's eviction. The bit is set when the L2 cache fulfills a request from the corresponding L1 cache, and reset when the line is evicted from L2 cache. Using the victim bits, early eviction in an L1 cache can be detected when the L1 cache sends a second request for a cache line that was requested recently. In this case, the LLS is incremented to indicate that a reuse opportunity is lost. When the LLS rate (LLS over total number of accesses) is high enough, the L2 cache notifies the L1 cache indicating that contention is detected. The hardware overhead can be reduced by sampling, i.e. some (e.g. one in every 64 sets) of the L2 cache sets are selected as samplers and enhanced with the extra bits, while other sets remain the same as normal ones. It is worth pointing out that our design is cost-effective because the existing tag array in the L2 cache is leveraged to collect information for L1 caches instead of adding an extra VTA to each L1 cache. The overhead can be further reduced by letting multiple SIMT-cores share the same victim bit.

Congestion Detection. Jia et al. [19] observed cache resource (cache block, miss queue or MSHR) congestion in some applications and proposed to check resource reservation failures and trigger bypassing. In this paper, however, we focus on congestion caused by bandwidth (NoC or DRAM) saturation. This can be detected by monitoring NoC and the memory controllers. It provides the feedback information to control bypassing and multithreading. When cache congestion is detected, the cache controller enables bypassing. If NoC or DRAM congestion is also detected, the warp schedulers are notified and they may try to reduce the number of active warps according to the statistical information related to congestion and bypassing. In our implementation, we focus on L1 cache bypassing and only detect NoC congestion. This information is used to estimate the optimal MAW in Section V-C. Some NoC packets are randomly selected as samplers, and the NoC latencies (number of cycles used to travel through the network) of these samplers are recorded by the hardware counters. The average NoC latency is calculated after each sampling period.

C. Coordinated Bypassing and Warp Throttling

Our proposed method for dynamic CBWT uses the run-time observed L1 cache bypass rate, NoC congestion, and

Figure 9. Hardware Extensions on the L2 Cache

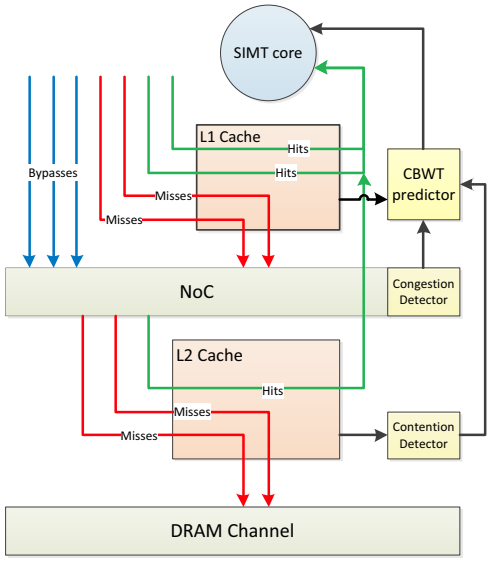

Figure 10. Cache hierarchy overview of CBWT. Memory requests consist of hits, misses and bypasses. Caches are protected by the PDP bypass policy. Extra sampling modules (yellow blocks) are added to monitor contention and congestion. To overcome the limitations of bypassing, CBWT adaptively controls the number of active warps to take full advantage of the cache capacity and other on-chip resources.

L1 contention to adjust the number of active warps to avoid overburdening the memory system. The feedback paths are illustrated in Fig. 10. Note that in this paper we only implement CBWT in L1 caches, and we leave the extension to L2 cache as future work. Sampled statistics are accumulated over periods of 16k accesses to the *sampler L1 cache* (one of the existing L1 caches that is selected for sampling reuse information for PDP-S). The hardware attempts to adjust the MAW to keep the network in a busy but low-congestion range.

The MAW is adjusted in multiple steps, illustrated in Fig. 11. A kernel begins executing using the maximum possible number of warps, n_{max}. Then, the hardware sets the MAW to an estimated optimum value based on the observed rate of bypassing (❶). When the MAW drops below a threshold T_w, it is adjusted in steps of ± 1 to bring the network latency into the target range. When the NoC latency is larger than a threshold (T_{NoC_H}) *or* the change of NoC latency is larger than another threshold (T_{NoC_G}), the MAW is decremented (❷). Similarly, when the NoC latency is smaller than a threshold (T_{NoC_L}), the MAW is incremented (❸).

The initial MAW estimate (❶) is based on a simple model of memory behavior. The model does not need to be very accurate since the MAW is subsequently adjusted with hardware feedback. Suppose that each thread in a SIMT-core repeatedly accesses a single cache line, there is no inter-thread sharing, and warps are scheduled in round-robin order. Then the working set size is proportional to the number of active warps, and locality can be maximized by

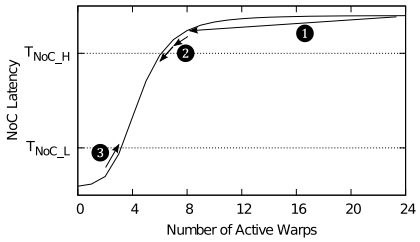

Figure 11. As the number of active warps increases, NoC latency tends to grow up to a saturation point. Coordinated bypassing and warp throttling uses feedback to keep the network in the busy but low-congestion range between T_{NoC_L} and T_{NoC_H}.

scaling down the MAW so that the working set size matches the cache size. During a sampling period, the number of bypassed memory requests m_b (representing possibly-reused data that does not fit in the cache) and the total number of memory requests m are recorded. The ratio m_b/m is the fraction of the working set that does not fit in cache. The MAW is scaled down to

$$n_t = (1 - \frac{m_b}{m}) \times n_{max}, \quad (1)$$

where n_t is the estimated optimal MAW with only warp throttling, to match the working set to the cache size.

Fig. 12 shows the summarized detail of the proposed MAW search algorithm. During each sampling period, the contention and congestion information is collected. The *direction* is a status variable, which holds the change (increasing or decreasing) of MAW in the previous sampling period. A coarse and aggressive search (❶) based on Eq. 1 happens when contention and congestion are detected (by monitoring the LLS ratio and NoC latency). When the MAW is smaller than T_w (set to 10 in our implementation), a fine-grained and conservative search (❷ and ❸) is performed. If the NoC latency is large or the change of NoC latency is significant (e.g. decreased by more than 5%), the MAW is decreased (❷). On the other hand, if the NoC latency is small, the MAW should be increased to exploit more parallelism (❸). In this conservative search, the MAW can also be rolled back one step (❹) when the previous update is out of the sweet spot. The search stops when congestion falls in the desired range (❺ and ❻). By making an initial estimate of the MAW and then incrementally adjusting it using hardware feedback, our method dynamically selects a good MAW quickly.

D. Hardware Cost and Complexity

CBWT has comparatively low hardware overhead. The additional costs in both storage area and logic complexity are reasonably low, so that the proposed memory hierarchy can be easily produced with the current manufacturing process. The bypass policy overhead is the same as PDP cache [11], but since we apply it in L1 caches, the PDP sampling overhead is lower than that in LLCs, because L1 caches have smaller sizes and require fewer sampling sets. Moreover

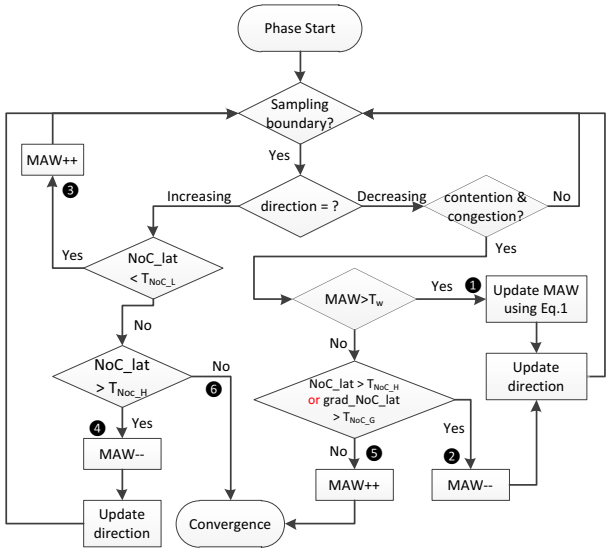

Figure 12. Flow chart of the proposed algorithm. NoC_lat=NoC latency.

since we choose the PDP-S design, the bypass policy only requires one predictor and sampling module, the bypass policy overhead is negligible. As for CBWT prediction, the storage overhead has three parts. The first one is the congestion detector which requires counters to calculate the NoC latency of the sampler packets. The second part is the contention detector which includes victim bits as well as two saturating counters to accumulate LLS and the number of accesses to the sampler sets in the L2 cache. Assume the storage overhead for victim bits is represented by O_v. and the L2 cache has N sampler sets and M ways, and the number of L1 caches is P. Then the storage overhead is: $O_v = P \times N \times M$ bits. For a 16-core GPU with a 16-way 1M L2 cache and the sampling ratio is $\frac{1}{64}$), 8 sets are selected as samplers (512 sets in total), then $O_v = 256B$. The third part is the predictor, and since the search algorithm is relatively simple, the predictor is also cheap to implement.

VI. Evaluation

We first evaluate CBWT and pure bypassing design by considering the improvements on overall performance, their impact on cache efficiency, DRAM traffic and power efficiency. Then CBWT is compared with the pure warp throttling scheme, i.e. CCWS [39]. We also compare CBWT with MRPB [19], wihch employs request buffers to reoder memory requests to preserve intra-warp locality before they are sent to L1 caches. We use GPGPU-Sim v3.2.0 [4] to model the baseline architecture which mimics a generic NVIDIA Fermi GPU [33]. The baseline architecture uses a detailed GDDR5 DRAM model. NoC traffic in each direction between the SIMT cores and the memory partitions are serviced by two separate networks which can transfer a 32-byte flit per interconnect cycle to/from each memory partition. GPUWattch [28] is used to estimate the power consumption. Table II lists the major configuration parameters.

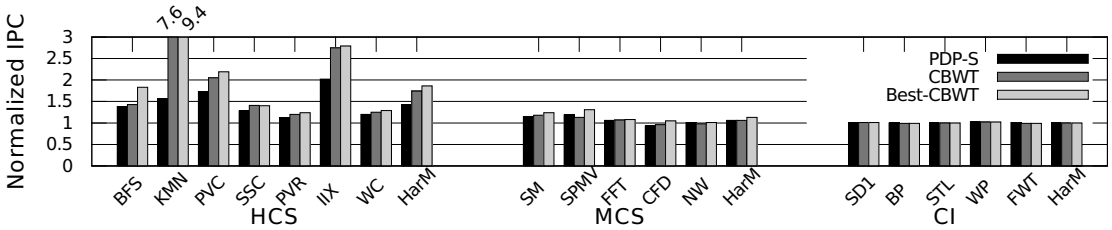

Figure 13. Performance improvement of PDP-S and CBWT over baseline on all benchmarks.

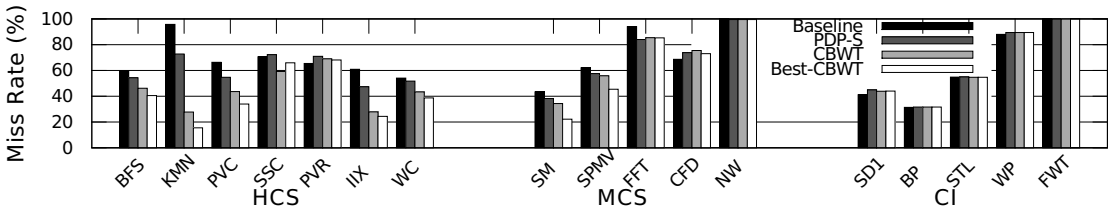

Figure 14. L1 cache miss rate on all benchmarks.

The benchmarks in Table I are used for evaluation.

A. Comparison with Pure Bypassing

Performance. Compared to pure bypassing, CBWT is able to control the number of active warps, and thus reduces reuse distance as well as alleviates resource congestion. This is especially critical for those applications whose reuse distances are long and a huge number of memory requests make the NoC highly congested. Fig. 13 illustrates normalized performance (IPC) improvement of PDP-S, CBWT and Best-CBWT on all benchmarks, with respect to baseline. Best-CBWT is the optimal SWL with bypassing enabled, which is described in Section V-A. It shows that CBWT achieves an average of 74% IPC improvement on HCS benchmarks over baseline, which significantly outperforms PDP-S (42% improvement over baseline). This is mainly because CBWT reduces MAWs when contention and congestion are detected. Although PDP-S can alleviate contention, it can not handle NoC over-saturation caused by a massive amount of requests. For HCS benchmarks, CBWT reduces 18.3% of the average NoC latency compared to PDP-S. With the bypass ratio CBWT can quickly find the optimal or near-optimal MAWs, and throttle the memory request flow to fully utilize on-chip resources. Among the HCS benchmarks, `KMN` and `IIX` exhibit tremendous speedups over baseline, 7.6× and 2.7× respectively, by alleviating intensive cache contention as well as severe NoC congestion.

SIMT Core	16 cores, SIMT width=32, 5-Stage Pipeline, 1.4GHz
Per-core Limit	48KB scratchpad, 32768 registers, 32 MSHRs, 1536 threads, 48 warps
L1 Cache	32KB/core, 4-way, 128B line, coalescing enabled
L2 Cache	8 banks, 128KB/bank, 16-way, 128B line
Scheduling	LRR warp scheduling, round-robin CTA scheduling
Interconnect	32B channel width, 1.4GHz, BW=350GB/s per direction
DRAM Model	Out-of-order (FR-FCFS), 924 MHz, 8MCs, channel BW=8Bytes/Cycle
GDDR5 Timing	tCL=12, tRP=12, tRC=40, tRAS=28, tRCD=12, tRRD=6

Table II
SIMULATION CONFIGURATION

Compared to Best-CBWT, CBWT only loses 8.6% performance for HCS benchmarks on average. For `SSC`, it even slightly outperforms Best-CBWT because this benchmark has a phase change property and a fixed MAW cannot get optimal performance. CBWT cannot constantly outperform Best-CBWT because the prediction has a start-up cost associated with the contention and congestion detection. For example, for `BFS` which consists of multiple small kernels, CBWT is much worse than Best-CBWT. And also since the prediction is not always 100% accurate, CBWT may get sub-optimal performance. However, since CBWT precisely controls the intensity of contention and congestion, it predicts well and overall archives competitive performance compared to Best-CBWT. Fig. 13 also shows normalized performance on MCS and CI benchmarks. For most of them, the bypass policies make little difference to performance since they are not very sensitive to cache behavior. For all of the benchmarks, CBWT does not obviously degrade performance compared to the baseline, especially for HCS benchmarks.

Cache Efficiency. Fig. 14 illustrates that the reason for the performance improvement for PDP-S and CBWT on HCS benchmarks is a sharp reduction in cache misses. For example, more than a 50% reduction is observed under CBWT for `KMN` and `IIX` compared to baseline. The miss rate of HCS benchmarks declines dramatically for PDP-S because bypassing can effectively alleviate cache contention. The

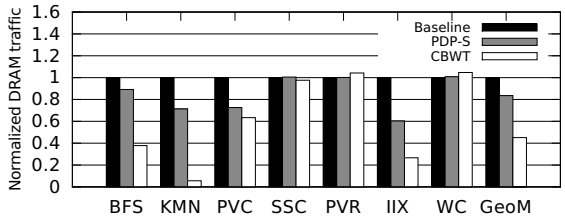

Figure 15. DRAM traffic on HCS benchmarks, normalized to baseline.

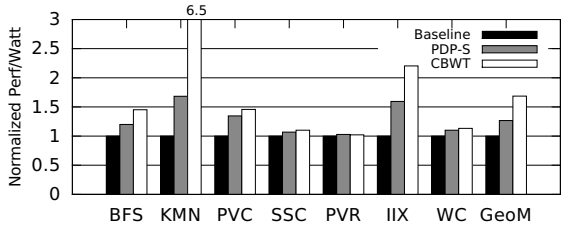

Figure 16. Energy efficiency on HCS benchmarks, normalized to baseline.

cache lines are protected from early eviction when thrashing happens. CBWT further decreases miss rate over PDP-S because it limits the number of active warps that can issue memory requests to the cache system, and thus preserves inter-warp locality by delaying some of the requests. This is especially important when the application working set is too large and a large percentage of requests are bypassed and congest the network. For the MCS and CI benchmarks, miss rates are relatively similar for different policies. These are consistent with the performance results.

DRAM Traffic and Energy Efficiency. Fig. 15 shows DRAM requests generated by HCS benchmarks. On average, PDP-S can reduce 16.5% of DRAM traffic. CBWT reduces DRAM traffic by more than 50% on average, which is due to the combined effects of bypass policy and warp throttling. The dramatic reduction of DRAM traffic leads to an average DRAM power reduction of 14.4%. Fig. 16 shows the system energy efficiency (Perf/Watt) for HCS benchmarks. Overall, CBWT outperforms the baseline GPU with an average of 58.6% system energy efficiency improvement in terms of Perf/Watt. For all evaluated benchmarks, including HCS, MCS and CI benchmarks, CBWT has an average of 25.4% system energy efficiency improvement over the baseline. Considering only the effect on DRAM energy efficiency, although not shown in the figure, CBWT provides an average of 111% (2.11×) Perf/Watt improvement over the baseline. Overall, CBWT gives an average of 76.7% DRAM energy efficiency improvement.

B. Comparison with Other Schemes

Perfomance. Fig. 17 shows normalized performance (IPC) improvement of Best-SWL (static optimal CCWS), MRPB, and CBWT on HCS benchmarks, normalized to the baseline. CBWT shows an average IPC improvement of 1.74×, which outperforms Best-SWL and MRPB (1.52× and 1.57× respectively). Compared to CCWS, however,

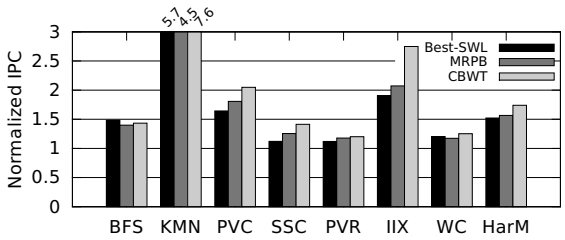

Figure 17. Performance improvement of Best-SWL, MRPB, and CBWT on HCS benchmarks.

CBWT allows more active warps to run concurrently. More threading not only speeds up computation, but also more effectively hides memory latency. This is because CBWT can better take advantage of the spare bandwidth and service more requests earlier, which leads to fewer memory stalls. Compared to MRPB which has no control on the NoC flow, CBWT is also able to better utilize NoC bandwidth. Thus performance improvements are observed for the benchmarks that generate a massive amount of requests. For example, the speedups of `KMN` under Best-SWL and MRPB are significantly smaller (5.7× and 4.5× respectively) than that under CBWT (7.6×). We also observe dramatic speedups when applying Best-SWL and MRPB for `IIX` (1.9× and 2.1× respectively), but CBWT is still able to further improve performance (2.7×).

Interaction with Warp Schedulers. The LRR (Loose Round Robin) [39] warp scheduling policy used in the baseline has been shown to be inefficient compared to advanced scheduling policies [20], [39], [40]. We change the policy to GTO (Greedy Then Old) which runs a single warp until it stalls then picks the oldest ready warp, and evaluate the performance of Best-SWL, MRPB and CBWT. Fig. 18 illustrates the average IPC of Best-SWL, MRPB and CBWT on HCS benchmarks, normalized to baseline. It shows that CBWT outperforms Best-SWL and MRPB, no matter which policy the scheduler uses. If normalized to Best-SWL, MRPB achieves almost the same performance, while CBWT achieves 17.3% and 37.8% performance improvement with LRR and GTO scheduler respectively.

CBWT provides more improvement when GTO is applied because GTO shrinks the performance benefit of Best-SWL and MRPB, since they both only focus on cache performance. However CBWT monitors NoC congestion and properly controls the request flow through bypassing and throttling, which is still beneficial even if GTO improves the cache behavior. Note that Li [29] proposed a GPU cache management scheme called PCAL, which is based on CCWS and activates more warps when NoC is underutilized. As reported, for 12 cache sensitive benchmarks, under GTO scheduling policy, PCAL provides 2.8% IPC improvement over best-SWL on average. With extra traffic optimization, it acheives 6.4% IPC improvement on average. The main difference of the PCAL design compared to CBWT is the bypassing scheme. PCAL bypasses all the requests issued

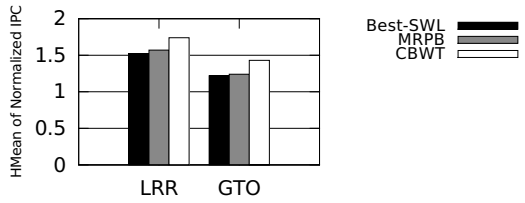

Figure 18. Average performance improvement of Best-SWL, MRPB, and CBWT over baseline on HCS benchmarks, using LRR and GTO scheduling policies.

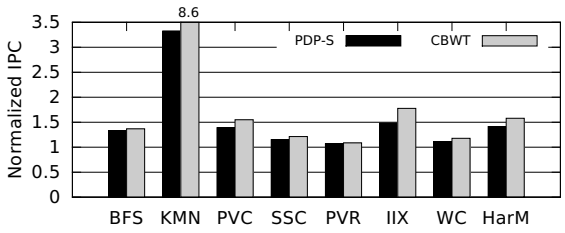

Figure 19. Performance of PDP-S and CBWT normalized to baseline on HCS benchmarks. The baseline, PDP-S and CBWT all have 64KB L1 cache per SIMT core.

by the warps with lower priority. However, CBWT bypasses requests based on reuse distance and protects hot cache lines to take full advantage of the cache capacity.

C. Sensitivity Study

In the future, manufactures may enlarge cache size to satisfy performance requirements of emerging applications. We also evaluate our design with 64KB L1 caches to give some insight into the effects on larger cache sizes. Fig. 19 presents the normalized IPC of PDP-S and CBWT over the baseline, all with 64KB L1 cache per SIMT core. On average, PDP-S and CBWT significantly improve system performance by 41.3% and 51.9% respectively. This implies that even if larger L1 caches might be available in the future, it is still beneficial to apply advanced cache management schemes. For some benchmarks, the improvement of CBWT over PDP-S shrinks because larger L1 caches absorb many more requests and eliminate a lot of NoC traffic. However, CBWT still achieves significant improvement for `KMN` compared to PDP-S, because `KMN` generates a huge amount of requests which still cause NoC congestion even if the L1 cache size is doubled. The effects of larger MSHR size are also evaluated (not shown in the figure). Going to 128 MSHRs per core increases NoC latency, since more requests can be injected into the network. However, CBWT can adaptively control NoC flow and its performance is insensitive to the changes of MSHR size: the average normalized IPC of HCS benchmarks changes from 1.74 to 1.70.

VII. Related Work

A. CPU Cache Management

Cache management is a well-explored research area for CPUs. Replacement and bypass policies are related to our work. Re-reference interval prediction (RRIP) [17] modifies the LRU and NRU policies to treat misses and hits differently so that the reused cache lines are protected from being replaced by a burst of requests with a very large reuse interval. Cache bypassing [21], [12], [23] has been investigated to selectively bypass data in the on-chip caches. Gaur et al. [12] presented a selective bypass algorithm based on trip counts and use counts for exclusive LLCs. Kharbutli et al. [23] introduced a counter-based LLC bypass algorithm that leverages a prediction table. Dead block prediction techniques [22], [26], [30] are utilized to guide replacement and bypass decisions. They generally predict a cache line is dead and avoid caching it by selecting it as replacement candidate or bypassing it. PDP cache [11] introduced *protection distance* (PD) to protect cache lines from being replaced. If no unprotected line is found, the incoming request is bypassed. Dynamic PDP introduces a sampling module to collect reuse information, and uses a dedicated pipeline to compute the PD at runtime. Our work is built on the PDP bypass policy and adapted to GPGPUs.

B. GPU Cache Management

Warp scheduling policies [39], [20] have been investigated as methods to improve cache efficiency in GPUs. CCWS [39] heuristically schedules warps to alleviate L1 cache contention. The *victim tag array* is introduced to collect the *lost locality score* which indicates how serious the inter-warp contention is. DAWS [40] refines CCWS using cache footprint prediction. When severe contention is detected (some warps lose too much locality) the scheduler suspends some of the warps (not allowed to issue requests and stalled). CBWT instead reduces contention through cache bypassing and requires cheaper hardware. Compared to pure warp throttling in CCWS, CBWT coordinates bypassing and warp throttling together and opens up a new opportunity to improve performance.

MRPB [19] uses FIFO buffers to reorder memory requests, shortening the reuse distance of memory requests before they are sent to L1 caches. Although CBWT combines bypassing and warp throttling to preserve locality, it can work together with MRPB to further improve performance and energy efficiency. MRPB also employs bypassing to reduce intra-warp contention in GPU L1 cache. Bypassing is triggered when resource unavailability stalls happen, i.e. a burst of requests access the same cache set in a short period of time, and the target set runs out of available block to service incoming requests. This bypassing scheme only works for this special case and it can also be incorporated into our bypassing framework.

Li [29] proposed a cache bypassing scheme on top of CCWS, called Priority-based Cache Allocation (PCAL). PCAL starts from an optimal number of active warps when bypassing is disabled. It then adds or removes warps according to resource utilization. Extra warps are given lower priority to allocate in the cache to alleviate contention. As

reported, PCAL achieves limited performance improvement over CCWS. Compared to its warp-level bypass policy, our bypass policy is a finer-grained scheme at the request-level. It is more efficient to fill the cache with the hottest cache lines from all warps, instead of all cache lines from some prioritized warps. Besides, our design requires lower hardware overhead, since PCAL is built based on CCWS which requires a VTA in each L1 cache.

Choi et al. [10] proposed a GPU read-bypassing scheme which prevents the LLC from being polluted by streaming data that is consumed only within a CTA and thereby preserves the LLC for inter-CTA communication. Our work mainly considers L1 cache and our bypass policy is based on reuse distance prediction. A unified GPU on-chip memory design is proposed by Gebhart et al. [14] to satisfy varying capacity needs across different applications. LLC management policies for 3D scene rendering workloads on GPUs are explored by Gaur et al. [13], while our work focuses on general purpose applications. Some other work studied cache management schemes for heterogeneous systems [27], [31]. Although our work focuses on GPGPUs, it is also applicable to fused CPU-GPU systems like APUs. It is even more important for APUs to include contention-resistant or congestion-resistant techniques because their GPU private caches are relatively smaller than those of discrete cards. Compiler or programming techniques for improving GPU cache performance are also investigated [18], [44], [6]. Although static compiler-directed bypassing could be effective for regular applications, we provide a hardware dynamic solution that adapts to different runtime behaviors. This allows for dynamic adjustment in response to different input data and application phase behaviors.

C. Thread Throttling

Thread throttling techniques have also been proposed in multi-threaded CPU systems. Suleman et al. [43] adjust the number of concurrent threads when the application is limited by data synchronization or memory bandwidth. In this framework, a loop starts with some training iterations. During the training phase, only a single thread is allowed to run, and the amount of synchronization and bandwidth utilization are monitored. The framework then calculates the optimal number of threads according to an analytical model, and enables multithreading of this number. We do not choose to restrict single-threaded execution to estimate resource requirements of a single warp, since it may significantly hurt performance in a massive-multithreading environment. Cheng et al. [9] use a thread throttling technique to reduce memory latency in multicore systems. They also build an analytical model and limit the number of concurrent memory tasks to avoid the interference among memory requests. These are software techniques that make decisions in the runtime system. Our work also considers memory interference, but we leverage bypass information and utilize a simple predictor to make the hardware easy to implement.

VIII. Conclusion

GPGPUs are throughput-oriented processors that can hide memory latency by massive multithreading. However, GPU caches are not designed with enough awareness of massive multithreading, leading to poor efficiency. In this paper we rethink the cache hierarchy and its management trade-offs in GPGPUs, and show that a specialized cache management design for GPU computing is important to improve performance and energy-efficiency. Cache bypassing is adopted to enable protection on hot cache lines and alleviate cache contention. To overcome the limitations of pure bypassing, our bypass policy is coordinated with warp throttling and adaptively controls the parallelism when contention or congestion happens. This management scheme is built on top of a cost-effective hardware design and is simple enough to implement in real hardware. Experimental results demonstrate that our design significantly outperforms the baseline architecture and state-of-the-art management schemes for cache-sensitive benchmarks.

Acknowledgement

We thank the entire IMPACT Research Group for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments and suggestions, and Wenhao Jia from Princeton University for generously sharing his source code. This work is partly supported by the 863 Program of China (2012AA010905), the Starnet Center for Future Architecture Research (C-FAR), the Defense Advanced Research Projects Agency under award HR0011-13-2-0014, the DoE Vancouver Project (DE-FC02-10ER26004/DE-SC0005515), the UIUC CUDA Center of Excellence, the NSFC (61433019, 61272144), the HPCL of NUDT (201302-02), the China Scholarship Council, and the NUDT Graduate Innovation Fund.

References

[1] *AMD Graphics Cores Next (GCN) Architecture white paper*, AMD, 2012.

[2] N. Anssari, "Using hybrid shared and distributed caching for mixed-coherency GPU workloads," Master's thesis, University of Illinois at Urbana-Champaign, May 2012.

[3] S. S. Baghsorkhi *et al.*, "Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2012.

[4] A. Bakhoda *et al.*, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[5] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, June 1966.

[6] L.-W. Chang *et al.*, "A scalable, numerically stable, high-performance tridiagonal solver using GPUs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2012.

[7] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2009.

[8] X. Chen *et al.*, "Adaptive cache bypass and insertion for many-core accelerators," in *Proceedings of the 2nd International Workshop on Many-core Embedded Systems*, 2014.

[9] H.-Y. Cheng *et al.*, "Memory latency reduction via thread throttling," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[10] H. Choi, J. Ahn, and W. Sung, "Reducing off-chip memory traffic by selective cache management scheme in GPGPUs," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, 2012.

[11] N. Duong *et al.*, "Improving cache management policies using dynamic reuse distances," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[12] J. Gaur, M. Chaudhuri, and S. Subramoney, "Bypass and insertion algorithms for exclusive last-level caches," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011.

[13] J. Gaur *et al.*, "Efficient management of last-level caches in graphics processors for 3D scene rendering workloads," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[14] M. Gebhart *et al.*, "Unifying primary cache, scratch, and register file memories in a throughput processor," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[15] B. He *et al.*, "Mars: A mapreduce framework on graphics processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[16] B. A. Hechtman *et al.*, "QuickRelease: A throughput-oriented approach to release consistency on GPUs," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, 2014.

[17] A. Jaleel *et al.*, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010.

[18] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and improving the use of demand-fetched caches in GPUs," in *Proceedings of the 26th ACM International Conference on Supercomputing*, 2012.

[19] W. Jia, K. A. Shaw, and M. Martonosi, "MRPB: Memory request prioritization for massively parallel processors," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, 2014.

[20] A. Jog *et al.*, "OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[21] T. L. Johnson and W.-m. W. Hwu, "Run-time adaptive cache hierarchy management via reference analysis," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[22] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[23] M. Kharbutli and D. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 433–447, 2008.

[24] *The OpenCL C Specification Version: 2.0*, Khronos Group, 2013.

[25] G. Kyriazis, "Heterogeneous system architecture: A technical review," *AMD Fusion Developer Summit*, 2012.

[26] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[27] J. Lee and H. Kim, "TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture," in *Proceedings of the IEEE 18th International Symposium on High-Performance Computer Architecture*, 2012.

[28] J. Leng *et al.*, "GPUWattch: Enabling energy optimizations in GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[29] D. Li, "Orchestrating thread scheduling and cache management to improve memory system throughput in throughput processors," Ph.D. dissertation, University of Texas at Austin, May 2014.

[30] H. Liu *et al.*, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, 2008.

[31] V. Mekkat *et al.*, "Managing shared last-level cache in a heterogeneous multicore processor," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013.

[32] V. Narasiman *et al.*, "Improving GPU performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[33] *NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™*, NVIDIA, 2009.

[34] NVIDIA, "CUDA C/C++ SDK code samples," 2011. [Online]. Available: http://developer.nvidia.com/cuda-cc-sdk-code-samples

[35] *NVIDIA's Next Generation CUDA™ Compute Architecture: Kepler™ GK110*, NVIDIA, 2012.

[36] *CUDA C Programming Guide v5.5*, NVIDIA, 2013.

[37] M. K. Qureshi *et al.*, "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[38] M. Rhu *et al.*, "A locality-aware memory hierarchy for energy-efficient GPU architectures," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[39] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[40] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware warp scheduling," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[41] I. Singh *et al.*, "Cache coherence for GPU architectures," in *IEEE 19th International Symposium on High Performance Computer Architecture*, 2013.

[42] J. A. Stratton *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," UIUC, Tech. Rep. IMPACT-12-01, March 2012.

[43] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on CMPs," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[44] X. Xie *et al.*, "An efficient compiler framework for cache bypassing on GPUs," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2013.