Adaptive Cache Bypass and Insertion for Many-core Accelerators

Xuhao Chen[1,2], Shengzhao Wu[2], Li-Wen Chang[2], Wei-Sheng Huang[2], Carl Pearson[2], Zhiying Wang[1], and Wen-Mei W. Hwu[2]

[1]School of Computer, National University of Defense Technology
[2]Electrical and Computer Engineering, University of Illinois at Urbana-Champaign
{*cxh, wu14, lchang20, whuang47, pearson, w-hwu*}*@illinois.edu*, {*zywang*}*@nudt.edu.cn*

ABSTRACT

Many-core accelerators, e.g. GPUs, are widely used for accelerating general-purpose compute kernels. With the SIMT execution model, GPUs can hide memory latency through massive multithreading for many regular applications. To support more applications with irregular memory access pattern, cache hierarchy is introduced to GPU architecture to capture input data sharing and mitigate the effect of irregular accesses. However, GPU caches suffer from poor efficiency due to severe contention, which makes it difficult to adopt heuristic management policies, and also limits system performance and energy-efficiency.

We propose an adaptive cache management policy specifically for many-core accelerators. The tag array of L2 cache is enhanced with extra bits to track memory access history, an thus the locality information is captured and provided to L1 cache as heuristics to guide its run-time bypass and insertion decisions. By preventing un-reused data from polluting the cache and alleviating contention, cache efficiency is significantly improved. As a result, the system performance is improved by 31% on average for cache sensitive benchmarks, compared to the baseline GPU architecture.

Categories and Subject Descriptors

C.1.4 [**Computer Systems Organization**]: Processor Architectures—*Parallel Architectures*; B.3.2 [**Memory Structures**]: Design Style—*Cache memories*

General Terms

Design, performance

Keywords

GPGPU, cache management, bypass, insertion

1. INTRODUCTION

MES '14, June 15 2014, Minneapolis, MN, USA
Copyright 2014 ACM 978-1-4503-2822-7/14/06...$15.00.
http://dx.doi.org/10.1145/2613908.2613909

Using CUDA [25] or OpenCL [17] for general-purpose computing on graphics processing units (GPGPUs) has become pervasive in High Performance Computing (HPC). Manufacturers are incorporating hardware and software features to these throughput-oriented accelerators to better support unstructured applications with unpredictable memory access patterns. Caches are one of the important hardware features that have been included in GPUs to leverage on-chip data reuse, which can provide significant speedup for irregular applications written in a straightforward fashion. To extend the range of applications that benefit from caches, GPUs need to have specialized cache designs due to their massively multithreaded execution model.

Massive multithreading makes GPU cache locality difficult to capture [12, 5, 26], and conventional CPU cache management policies are not suitable for GPUs. A GPU usually has hundreds or thousands of threads running simultaneously. Thus, its cache capacity per thread is much smaller and cache line lifetime is much shorter than that is typically experienced in CPU. This poor temporal locality reduces the cache efficiency. Additionally, the spatial locality benefits provided by CPU caches are largely captured by the *coalescing unit* in GPU when the same warp accesses consecutive memory locations, before the requests are sent to the memory system. Recent studies [27, 14] have shown that GPU L1 caches suffer from high contention among concurrent warps that are scheduled to the same SIMT core. Poor locality and lack of intelligent management lead to unsatisfactory performance impact on applications [12, 26].

To understand GPU cache inefficiency, we conduct detailed characterization on massively multithreaded applications. We find that cache contention due to massive multithreading, as well as streaming and thrashing, are major sources of cache inefficiency. Even optimal replacement policy can not solve this problem [27]. Thus it is critical to extend cache line lifetime, so that cache lines can get reused before its eviction, and useful locality information can be collected and used to guide management. To extend cache line lifetime, cache lines should be protected by some mechanism, and some memory requests needs to be bypassed, i.e. sent to the next level of memory hierarchy without inserting it into cache. *Cache bypassing* can extend lifetime, avoid early eviction, and alleviate cache contention. However, CPU bypass policies which are usually designed for last level cache

(LLC) in CPU, can not make robust decisions when applied to GPU, because poor locality make it difficult to get useful information. To address this problem, we propose an adaptive bypass policy specifically for GPGPU, which is aware of massive multithreading. The basic observations that inspire our design are twofold: first, cache contention caused by massive multithreading significantly reduces the chance of cache line reuse, and thus *hot* (i.e., those are reused many times) cache lines should be protected; second, for streaming accesses, cache lines are never reused, and therefore should be unprotected or bypassed. Contention is detected at runtime which triggers bypass dynamically to protect hot lines. To prevent streaming accesses polluting the cache, the insertion policy is modified to treat *hot* and *cold* lines differently. Our management policies are designed on top of a cost-effective cache structure, but significantly improve cache efficiency and system performance. This paper makes the following contributions:

- We conduct detailed characterization and analyses on the parallel patterns of massively multithreaded applications, and indicate the necessity of a specialized management for GPU caches.
- We present an adaptive cache management policy for GPGPUs. Hardware extension is introduced to capture locality information. The proposed bypass and insertion policy can reduce contention and preserve space for hot cache lines.
- We implement our design in a cycle-accurate simulator, and the experimental results demonstrate that it can better exploit temporal locality than current designs, and thereby improve GPU system performance.

The rest of the paper is organized as follows: the massively parallel programming model and the baseline GPU architecture are introduced in Section 2. benchmark characterization is done in Section 3. The proposed management policy are described in Section 4. Section 5 presents the experimental results. Related works are discussed in Section 6. Finally Section 7 concludes.

2. BACKGROUND

In this paper, we focus on massively parallel programs (MPPs) written in OpenCL or CUDA. Although we use NVIDIA CUDA terminology in the following sections, our design is also applicable to OpenCL.

2.1 Massively Parallel Programming Model

Individual MPP functions written in single-program multiple-data (SPMD) form [25] and executed on the GPU device are called *kernel* functions. An MPP begins execution on a CPU and launches kernels onto a GPU. In this paper, we assume that kernels execute sequentially, i.e. only one kernel is executed at a time. Each instance of the SPMD function is executed by a GPU *thread*. Groups of such threads called *cooperative thread arrays* (CTAs), a.k.a *thread blocks* or *work groups*, are guaranteed to execute concurrently on the same SIMT core. Within each group, subgroups of threads called *warps* are executed in lockstep fashion, evaluating one instruction for all threads in the warp at once.

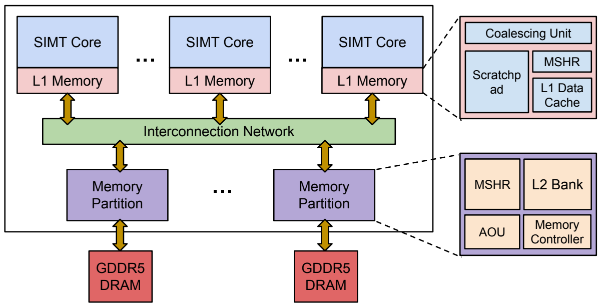

Figure 1: Baseline GPU Architecture

2.2 Baseline GPU Architecture

Figure 1 shows the organization of our baseline GPU architecture. The memory system consists of register files, L1 memory (scratchpad and L1 data cache), L2 cache, and off-chip GDDR DRAM [23, 24, 2]. L1 memory is private per-core and shared by warps running on the same SIMT core. Note that scratchpad memory is programmer visible and is used for explicit intra-CTA communication. The L2 cache is shared across all threads and is divided into multiple banks that are connected to the SIMT cores through an interconnection network. Each memory partition contains one L2 cache bank. All requests are grouped by a *coalescing unit* [23, 25] before sent to L1 cache. Read-modify-write atomic operations are performed at each memory partition by an *Atomic Operation Unit* (AOU) [23].

GPU caches usually adopt a write-through, write no-allocate policy for the L1 cache. This policy has been evaluated [28] to be more efficient than a write-back policy for GPUs. The L2 caches are write-back with write-allocation, which is the same design choice as conventional CPU LLC. Compared to a write-through design, this reduces the total number of DRAM accesses. Since the aggregate L1 cache size is close to L2 cache size [23], current GPU LLC is not inclusive which holds fewer redundant data copies than an inclusive cache.

After a kernel is launched the CTA scheduler schedules available CTAs associated with the kernel in a round-robin and load balanced fashion on all the SIMT cores. A warp scheduler is also responsible to schedule warps onto execution units. Many policies have been proposed, e.g. loose round-robin (LRR) policy, greedy-then-oldest (GTO) [27], two-level scheduling [22], cache-conscious scheduling (CCWS) [27], and CTA-aware scheduling [14]. CCWS shows that a good scheduling policy can improve cache performance when severe contention occurs in the L1 cache.

3. MOTIVATION

In this section, we evaluate GPU cache performance to expose why current GPU cache hierarchy is inefficient. An efficient cache should contain *hot* cache lines that are reused multiple times during their lifetimes. In this case, the cache would show high hit rate and reuse count. These qualities will reduce memory latency as well as DRAM traffic, which save bandwidth and energy consumption, and thereby improve system performance and energy efficiency. We also discuss the GPU cache access patterns that motivate our de-

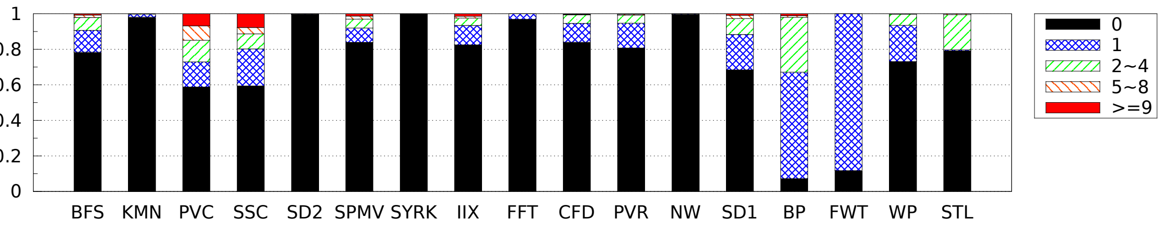

Figure 2: L1 Cache Reuse Count Distribution. It shows the number of times cache lines are reused throughout the entire program execution. Whenever a cache line is never reused it is effectively wasting cache space.

sign. We use memory bound GPU benchmarks that Jog et al. [14] selected from Parboil [29], Rodinia [4], MapReduce [10] and CUDA SDK [1] (listed in Table 1). Most of them are cache sensitive because we focus on improving cache efficiency. We also include some cache insensitive benchmarks to show that they are not hampered by the proposed design.

3.1 GPU Cache Inefficiency

Figure 2 shows the reuse count distribution for L1 cache. For most of the benchmarks, a large portion of the cache lines brought into the L1 caches are never reused, i.e. the reuse count is zero. For example, BFS has nearly 80% cache lines that are never reused. The non-reused cache lines could either be streaming accesses or victims of early eviction. The early eviction happens even more commonly in L1 cache than L2 cache since the L1 cache is relatively small and shared by tens of warps (e.g. Fermi supports a maximum of 48 warps, and each warp contains 32 SIMT threads). The warps contend for L1 cache space and continually replace the data of each other in the cache. Note that even if GPU caches are not efficient, there is not usually a significant performance penalty for GPU-friendly applications because massive multithreading can effectively hide memory latency. Cache performance only becomes critical when an application is bounded by memory bandwidth and the memory latency can no longer be hidden due to application characteristics. Besides, since cache line lifetime is very short [27], even the optimal replacement policy (i.e. always choose the furthest reused candidate to replace) shows very limited improvement due to frequent early eviction.

3.2 Cache Access Patterns

Streaming is one of the common access patterns for regular GPGPU applications. If this pattern dominates the entire kernel, cache efficiency has a limited impact on performance. Streaming often leads to very high miss rate, especially when the memory accesses are well coalesced. High miss rate can also be caused by *thrashing* or *cache contention*. Thrashing accesses are also common because L1 caches are relatively small. Since warps that are scheduled onto the same SIMT core share L1 cache, they may contend for limited L1 cache space, and requests from different warps cause cache lines continually replace each other, so no warp is able to use the L1 cache effectively. In order to illustrate how the cache contention affects L1 cache behavior, Figures 3 and 4 show the miss rate and speedup when varying the L1 cache size for cache sensitive benchmarks. These benchmarks significantly benefit from larger L1 cache size, because less contention

Benchmarks	Description	Suite
	Cache Sensitive	
BFS	Breadth First Search	[4]
KMN	K-means Clustering	[4]
PVC	Page View Count	[10]
SSC	Similarity Score	[10]
SD2	Graphic Diffusion	[4]
SPMV	Sparse Matrix Vector Multiply	[29]
SYRK	Symmetric Rank-K	[9]
IIX	Inverted Index	[10]
	Moderately Sensitive	
FFT	Fast Fourier Transform	[29]
CFD	CFD Solver	[4]
PVR	Page View Rank	[10]
NW	Needleman-Wunsch	[4]
	Cache Insensitive	
SD1	Graphic Diffusion	[4]
BP	Back Propagation	[4]
STL	Stencil	[29]
WP	Weather Prediction	[1]
FWT	Fast Walsh Transform	[1]

Table 1: Benchmarks

can lead to the capture of the temporary locality within the warp or between warps on the same SIMT core. Thus, in case of mixed access pattern in real application, this contention pattern should be distinguished from streaming pattern, because there is a possibility to improve the behavior of contention by employing better management policy, but no such possibility exists for streaming pattern. In order to fundamentally address these problems, GPU cache management policy should be aware of massive multithreading, and be adaptive to different cache access behaviors.

4. G-CACHE ORGANIZATION AND MANAGEMENT

In this section, we describe our proposed design, the *G-Cache*. The difficulty of managing GPU cache is that the massive multithreading makes it difficult to get useful cache locality information [11] which is collected by a conventional CPU cache to guide management. Based on the analyses in Section 3, we found that cache contention leads to severe cache inefficiency in L1 cache which results in unsatisfying performance. Early eviction happens so frequently that the cache controller cannot tell which lines are really the candidates for future reuse. Thus, GPUs need some specialized mechanism to avoid caching streaming accesses, so that the lifetime of cache lines can be extended enough to collect reuse information. Even if there is no streaming accesses,

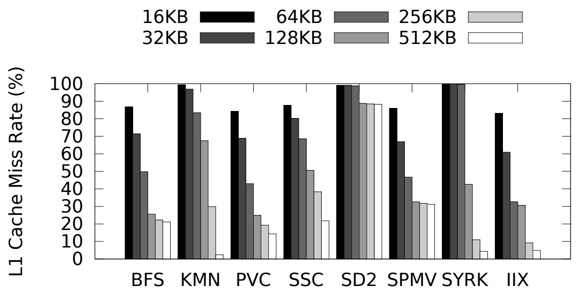

Figure 3: L1 Cache Size Sensitivity: Miss Rate

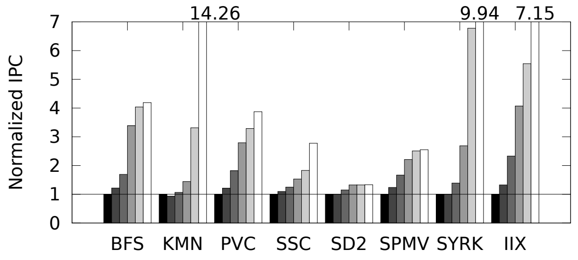

Figure 4: L1 Cache Size Sensitivity: Speedup

the controller has to be able to protect hot cache lines long enough and forward some requests to the lower level cache or DRAM. Cache bypassing can be used to alleviate the contention if we can detect it at runtime. However, a bypass policy should be designed carefully to avoid bypassing cache lines that will be frequently reused in the future.

4.1 Hardware Extension

We first introduce the hardware extension to enable heuristic management. Each L1 cache set adds one bit "bypass switch" (Figure 5) to control bypass, i.e. "0" means bypass disabled, and "1" means bypass enabled. We also extend the tag array of L2 cache with extra bits to store memory access history. As shown in Figure 6, an L2 cache entry consists of these fields: state bits, RRPV (Re-Referencing Prediction Value [11]), tag, data and "victim bits". The state bits, RRPV, tag and data are functionally the same as they do in traditional caches. The victim bits are added to control bypass decisions. They are bitmasks 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 L2 cache fulfills the request from the corresponding L1 cache, and reset when the line is evicted from L2 cache. Using the victim bits, contentions in L1 caches can be detected when the L1 cache sends a second request to the cache line that was requested recently. When contention is detected, L1 cache can enable bypass to mitigate it. Note that the G-Cache is a cost-effective design because the existing tag array in L2 cache are reused to collect locality information for L1 cache, instead of adding an extra victim tag array [27] or memory address FIFO [6]. The hardware overhead can be further reduced by letting multiple SIMT-cores share the same victim bit, or even all cores share 1 bit. This may lead to inaccurate bypass decision, thus is a tradeoff between overhead and performance gain.

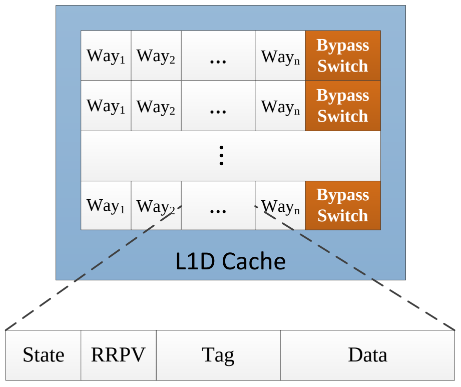

Figure 5: Hardware Extension on L1 Data Caches

4.2 Adaptive Bypass and Insertion

We adopt cache bypassing to extend the lifetime of cache lines. CPU bypass policies are not effective when applied to GPU, because they are intuitively designed for CPU LLC, and is not aware of GPU massive multithreading. We present an adaptive bypass policy for GPGPU which can deal with cache contention caused by massive multithreading. In our design, L2 cache is responsible to detect contention and notify the L1 cache. It interprets two memory requests from the same L1 cache to the same memory location as contention that causes early evictions in the L1 caches. Along with the requested data, L2 cache sends the value of the corresponding victim bit to the requester. This value notifies the L1 cache that this line was referenced before and became a victim of early eviction. With the victim bit set, L1 cache controller opens the bypass switch of the target set in L1 cache. When bypass switch is on, and all the existing lines in the target set are hot, a incoming block will be bypassed, since we want to protect hot lines. The RRPV of every existing line in the target set is checked to make this decision. If the RRPV is smaller than a threshold TH_{hot}, the line is considered as hot. If the attached victim bit is set, meaning the incoming block has lost locality, TH_{hot} will be lower to make it easier to replace one of the existing lines. However, when a line is bypassed, RRPVs of all lines in the target set are incremented. This is because the bypass victim could be a hot line in the future. To make sure it can be inserted into L1 cache when it becomes hot, the "hotness" of the existing lines is reduced whenever bypass happens. Note that the bypass switch can be shut down periodically to reduce side effect of bypassing. Overall, the proposed bypassing is heuristically controlled and adaptive to the cache contention. We adopt bypass on fill (whether or not to fill data into L1 cache) since it can further extend lifetime.

Figure 7 shows how our proposed bypassing works. We

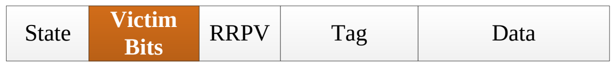

Figure 6: Hardware Extension on L2 Cache

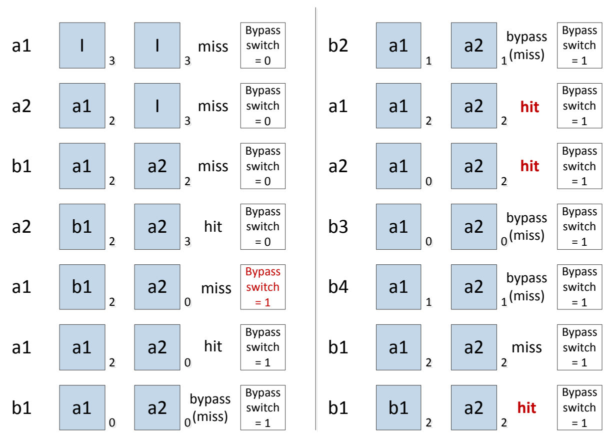
a1 I 3 I 3 miss Bypass switch = 0
a2 a1 2 I 3 miss Bypass switch = 0
b1 a1 2 a2 2 miss Bypass switch = 0
a2 b1 2 a2 3 hit Bypass switch = 0
a1 b1 2 a2 0 miss Bypass switch = 1
a1 a1 2 a2 0 hit Bypass switch = 1
b1 a1 0 a2 0 bypass (miss) Bypass switch = 1
b2 a1 1 a2 1 bypass (miss) Bypass switch = 1
a1 a1 2 a2 2 hit Bypass switch = 1
a2 a1 0 a2 2 hit Bypass switch = 1
b3 a1 0 a2 0 bypass (miss) Bypass switch = 1
b4 a1 1 a2 1 bypass (miss) Bypass switch = 1
b1 a1 2 a2 2 miss Bypass switch = 1
b1 b1 2 a2 2 hit Bypass switch = 1

Figure 7: Bypass example with a 2-way associative cache. The left column is the access sequence, i.e. $\{a_1, a_2, b_1, ..., b_1\}$. Blue boxes are cache lines. "I" denotes "Invalid". The number in the bottom right side is the corresponding RRPV.

present this example in the way that is similar to Figure 3 in [11]. Assume that the access stream is $\{a_1, a_2, b_1, a_2, a_1, a_1, b_1, b_2, a_1, a_2, b_1, b_1\}$, which is a stream mixed with hot accesses (a_x) and streaming accesses (b_x). This kind of access stream can be found in SPMV, where the matrix is streaming, but the vector is reused multiple times. In the beginning, the bypass switch of this set is off, indicating bypass of this set is disabled. When the request a_1 gets missed for the second time, L2 cache detects this as contention, and sends a notification to L1 cache, which opens the bypass switch. After that point, when b_1 and b_2 come in, they are bypassed, because both a_1 and a_2 are hot (with RRPVs less than 2), and the bypass switch is on. Thus, for the following requests a_1 and a_2, they both hit in cache. In case b_1 is becoming hot in the future, the RRPVs of a_1 and a_2 are incremented each time a request is bypassed. When b_1 keeps coming in, a_1 and a_2 become cold, and they eventually get replaced if b_1 is accessed enough times.

4.3 Hardware Cost and Complexity

G-Cache 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 manufacture process. The storage overhead for G-Cache is in victim bits, represented by O_v. Assume L2 cache has N sets and M ways, and the number of L1 caches is P, the storage overhead is: $O_v = P \times N \times M$ bits. For a 16-core GPU with a 512-set 16-way associative L2 cache (1MB in size), $O_v = 16KB$, essentially 1KB for each L1 cache on average. To further reduce hardware overhead, we can reduce the length of victim bits L_v by letting multiple SIMT-cores share the same victim bit. Assume S_v SIMT-cores share one bit, then $L_v = \frac{P}{S_v}$. The storage overhead in L1 cache is negligible and the logic complexity for bypass policy is close to state-of-the-art cache replacement policy (RRIP). In terms of interconnect traffic, because the victim bits information collected by L2 cache is sent together with the data response from L2 cache, there are no extra requests or responses being generated.

SIMT Core	16 cores, 1.4GHz, 5-Stage Pipeline, SIMT width = 32
Resources / Core	48KB Scratchpad, 32768 Registers, 1536 Threads, 48 warps
L1 Data Caches / Core	32KB, 4-way, 128B line size
L2 Cache Bank	128KB, 16-way, 128B line size, 700MHz
Features	Coalescing enabled, 32 MSHRs/core
Scheduling	LRR warp scheduling, round-robin CTA scheduling
Interconnect	2D Mesh, 1.4 GHz, 32B channel width
DRAM Model	FR-FCFS, 8MCs, 4 DRAM banks/MC, 2KB row size
GDDR5 Timing	1.4 GHz, tCL=12, tRP=12, tRC=40, tRAS=28, tRCD=12, tRRD=6

Table 2: Simulation Configurations

4.4 Comparison of Alternative Methods

We consider several alternative strategies for improving cache efficiency. First, increasing cache size could reduce contention and extend cache line lifetime, but also increases cache access latency and costs more area and power consumption [14]. Even though larger caches are being included in GPUs, a management policy that is aware of massive multithreading is still beneficial. Second, cache-aware scheduling policies can also alleviate contention [27, 14]. While CCWS tries to reduce multithreading, bypass tries to saturate L1 cache with hot cache lines and forward the others to the lower level of the memory hierarchy. The L2 cache provides locality information to L1 cache instead of requiring victim tag array [27] to collect it. Moreover, bypass can also cooperate with the scheduler to further improve cache efficiency. Third, existing CPU bypass policies [7, 6] which are intuitively designed for LLC, can be applied to GPU, but limited L1 cache size and cache inefficiency makes them less effective. In PDP cache [6], the amount of hits that fall into the samplers are too small, which makes it unstable to estimate the near-optimal PD, and thus leads to suboptimal performance. Instead G-Cache does not try to find the optimal PD, but to keep the hot lines in the cache. and avoid streaming accesses staying in cache too long. Thereby cache space is saved to mitigate cache contention, which sometimes leads to even better performance than SPDP-B policy. Besides, G-Cache is more cost-effective because no sampling logic, dedicated pipeline or harsh table is required.

5. EVALUATION

We evaluate G-Cache by considering the improvements on overall performance and its impact on cache efficiency. G-Cache (GC) is compared with several designs including the baseline (BS) and the baseline with 3-bit SRRIP policy (BS-S), as well as PDP-3, PDP-8 and SPDP-B which are described in [6]. The only difference between BS and BS-S is the L1 cache replacement policy. BS uses LRU policy, while BS-S uses 3-bit SRRIP policy. Note that we use 32 FIFOs per-set for PDP-3 and PDP-8, and 256 FIFOs per-set for SPDP-B. For all of them, 256 RDD counters are used to make PD estimation as accurate as possible, although it is expensive to implement. Although SPDP-B is not practical and PDP-8 is too expensive, they are presented to show the near-optimal performances. We show that GC achieves almost the same performance as SPDP-B, and outperforms

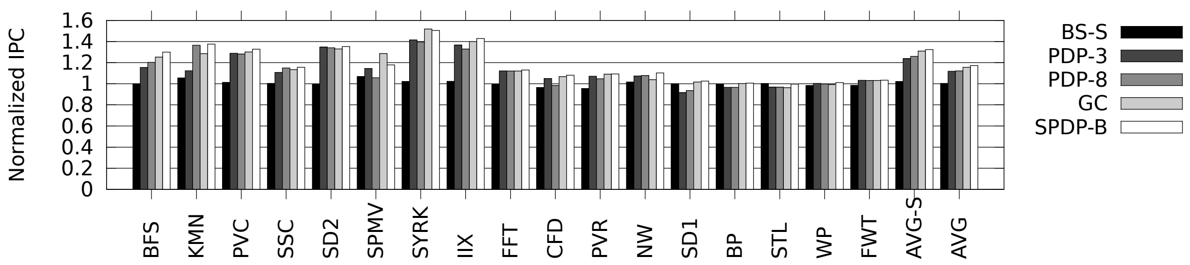

Figure 8: Performance speedup of our proposed design normalized to baseline GPU architecture

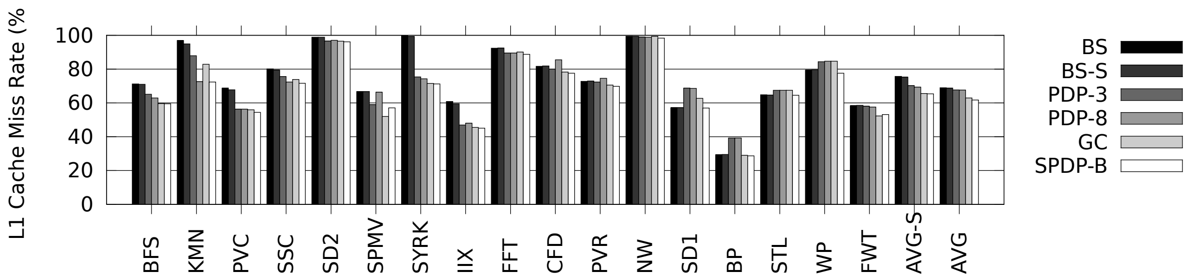

Figure 9: L1 miss rate of all benchmarks

dynamic PDP with cheaper and simpler hardware. We use GPGPU-Sim v3.1.2 [3] to model the baseline architecture which mimics a generic NVIDIA Fermi GPU [23]. The baseline architecture uses a detailed GDDR5 DRAM model. Table 2 lists the major configuration parameters. We use the benchmarks in Table 1 for evaluation.

5.1 Performance

Figure 8 shows performance (IPC) speedup of all different designs, normalized to BS. We use geometric mean for speedup. For cache sensitive benchmarks, GC gets reasonable speedup over BS, from 13.4% up to 51.8%, and 30.9% on average, while PDP-3 outperforms BS by 23.8% on average. For most of them, GC achieves competitive performance than SPDP-B, but for `KMN` and `NW` it works worse, because the reuse distances of these two benchmarks are too large that GC can not provide enough protection for these thrashing behaviors. While for `SPMV`, GC works better than SPDP-B because GC can tell the difference between streaming accesses and hot accesses, but PDP cannot. Thus the streaming accesses are evicted earlier which save space for useful cache lines. Another reason is, PDP gets unstable PDs due to lack of locality information, but GC essentially enforces its RRPVs to be less than 8 (2^3), which makes its RRPVs more stable than the PDs of PDP policies, and thus works better than PDP for benchmarks whose optimal PD is close to or smaller than 8. To further improve GC to deal with very large PDs, we can increment RRPVs on every M_{th} bypass to the set, with a counter counting up to M. M is set to 1 initially, and can be adjusted at runtime according to the contention information collected by L2 cache.

We can also find that on average PDP-3 archives performance close to PDP-8, (23.8% and 26% for cache sensitive benchmarks), and works even better than PDP-8 for some of the benchmarks. After all GC outperforms dynamic PDP policies and needs simpler hardware extension. For moderately sensitive benchmarks, even if they are not quite sensitive to L1 cache size, they still can slightly benefit from bypassing. For cache insensitive benchmarks, they show no speedup or very limited speedup as expected. For all benchmarks, GC and PDP-3 outperforms BS by 15.6% and 11.8% on average respectively. By comparing BS-S with BS, we find out that without bypass, 3-bit SRRIP policy almost has no impact on the performance, which demonstrates the performance benefit comes from bypass instead of replacement policy. The reason why replacement policy hardly affect performance is, although 3-bit SRRIP policy can potentially extend cache line lifetime, early eviction still happens frequently without bypassing.

5.2 Cache Efficiency

Figure 9 illustrates that the performance improvement comes from the decline of L1 cache miss rate. For most of the cache sensitive benchmarks, the cache miss rate drops significantly. For `SPMV` we can see that GC reduces more miss rate than SPDP-B, which leads to better performance. On the other hand, GC works worse than SPDP-B for `KMN` due to the short protection distance. The miss rate of `SD2` goes from 98.8% to 96.6%, but its performance improves 33%. This is because `SD2` can benefit from longer lifetime extended by the "bypass on fill" policy. Figure 9 also shows that 3-bit SRRIP gets the same miss rate as baseline and thus is almost helpless when severe contention happens. `SD1`, `STL` and `WP` show a little bit increase on the miss rate with GC, because their access patterns do not benefit from bypass, but bypass still happens because of detected contention. For cache insensitive benchmarks, they barely show any difference. Table 3 lists the bypass ratio (cache bypass as a fraction of accesses) of GC and SPDP-B. GC bypassed more accesses than SPDP-B for SPMV, while KMN and NW have much more accesses bypassed in SPDP-B than GC. This is consistent with the performance and miss rate results.

5.3 Scalability Study

In the future, manufactures may enlarge L1 cache size to satisfy performance requirement of more emerging applications. We also evaluate our design with larger L1 caches. Figure 10 shows G-Cache can improve system performance even when 64 KB L1 caches are applied to baseline and G-Cache. Even if larger caches are applied, the contention cannot be eliminated. Thus, the bypass policy can still help alleviate contention and improve performance. On average, G-Cache can provide 35.7% performance improvement for cache sensitive benchmarks and 16.1% overall, which is also very close to those of SPDP-B (40.1% and 19.5%).

6. RELATED WORKS

6.1 CPU Cache Management

Cache management is a well-explored research area for chip multiprocessors (CMPs). Re-reference interval prediction (RRIP) [11] modifies the LRU and the NRU policies to treat misses and hits differently so that the reused cache lines are protected from being replaced by a burst of requests whose reuse interval is in the distant future. We build our bypass policy on top of RRIP, and specialize it for GPGPU.

Cache bypassing has been investigated to selectively bypass data in the on-chip caches. Jayesh et al. presented a selective bypass algorithm based on trip counts and use counts for exclusive LLC [7]. Mazen et al. introduced a counter-based LLC bypass algorithm [16] leveraging a prediction table. Choi et al. [5] proposed a GPU read-bypassing scheme which prevents the shared cache from being polluted by streamed data that are consumed only within a CTA, thereby protests shared data for inter-CTA communication. Dead block prediction techniques [15, 18, 20] 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 [6] introduced "protection distance" to protect cache lines from being replaced. If no line is not protected, the incoming request is bypassed. Dynamic PDP requires a sampling module with per-set FIFOs and counter array, and a dedicated pipeline to compute the protection distance (PD) at runtime.

6.2 GPU Cache Management

LLC management policy for 3D scene rendering workloads on graphics processor are explored by Gaur et al. [8], while our work focuses on general purpose applications on GPU. Some other work studied cache management schemes for CPU-GPU heterogeneous systems [19, 21]. MRPB [13] employed L1 cache bypassing to reduce intra-warp contention in GPU. The bypass is triggered when resource unavailability stalls happen, i.e. a burst of requests access the same cache set in a short period of time. While MRPB bypassing works for a special case, G-Cache is designed for more general cases, which supports much more applications.

Benchmarks	G-Cache Bypass Ratio	SPDP-B Bypass Ratio	Optimal PD of SPDP-B
BFS	30.2%	33.5%	14
KMN	56.1%	70.6%	24
PVC	37.8%	35.8%	10
SSC	45.0%	55.6%	20
SD2	37.9%	45.6%	16
SPMV	37.2%	18.1%	6
SYRK	43.0%	36.2%	9
IIX	34.7%	25.3%	12
FFT	8.5%	25.3%	32
CFD	44.3%	29.2%	7
PVR	39.9%	0%	4
NW	5.1%	59.0%	68
SD1	2.7%	0%	4
BP	0.2%	0.1%	5
STL	11.3%	0%	4
WP	31.9%	27.5%	9
FWT	0%	3.8%	32

Table 3: Comparison on bypass control of G-Cache and SPDP-B with 32KB 4-way associative L1 cache

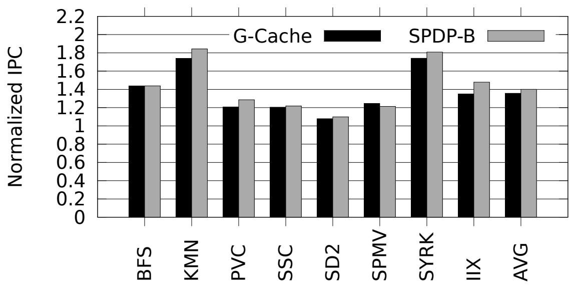

Figure 10: Performance speedup over baseline architecture. All designs have 64KB L1 caches.

Other works investigated warp scheduling policies [27, 14] to improve cache efficiency. CCWS [27] leverages warp scheduler to alleviate L1 cache contention in GPUs. A victim tag array called "lost locality detector" is proposed in each L1 cache to gather replacement information. When contention is detected, i.e. some warps lost a lot of locality, the scheduler then suspends some of the other warps (not allowed to issue instructions). However, G-Cache reduces contention through cache bypassing and requires simpler and cheaper hardware. MRPB [13] uses FIFO buffers to reorder memory requests before they are sent to L1 cache. Reordering can preserve intra-warp locality and reduce inter-warp contention. While CCWS and MRPB can both improve cache efficiency, G-cache focuses on cache management, and it can work along with CCWS and MRPB to further improve performance and energy efficiency.

Compiler techniques for improving GPU cache performance are also investigated [12, 30]. Although static compiler-directed bypassing could be effective for regular applications, we provide a hardware dynamic solution that adapts to different runtime behavior of applications. This allows for dynamic adjustment in response to different input data and application lifetime behaviors.

7. CONCLUSION

Many-core accelerators e.g. GPUs are evolving to support more applications. However, GPU caches is not designed with awareness of massive multithreading. In this paper we rethink the cache organization and management in GPGPUs. We observe that the current GPU cache performs poorly when massive multithreading makes it difficult to capture locality. Based on observations and analyses, we propose an adaptive management policy to achieve better performance. We employ runtime bypass to alleviate cache contention in L1 cache. The bypass policy is designed to be aware of massive multithreading and aggressively extend cache line lifetime when severe contention happens. The management policy is built on top of a cost-effective hardware design. Experimental results demonstrate that our design significantly outperforms baseline architecture for cache sensitive benchmarks.

8. ACKNOWLEDGEMENTS

The work is partly supported by the Starnet C-FAR Center, the DoE Vancouver Project (DE-FC02-10ER26004/DE-SC0005515) and the University of Illinois CUDA Center of Excellence. Xuhao Chen is supported by the State Scholarship Study Abroad Program for Graduate Studies at the China National University of Construction by Chinese Scholarship Council and the NUDT Graduate Innovation Fund.

9. REFERENCES

[1] "NVIDIA CUDA SDK code samples."

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

[3] A. Bakhoda, G. Yuan *et al.*, "Analyzing cuda workloads using a detailed gpu simulator," In ISPASS '09, Boston, MA, 2009.

[4] S. Che, M. Boyer *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," In IISWC '09, 2009.

[5] H. Choi, J. Ahn *et al.*, "Reducing off-chip memory traffic by selective cache management scheme in gpgpus," In GPGPU-5. New York, NY, USA: ACM, 2012.

[6] N. Duong, D. Zhao *et al.*, "Improving cache management policies using dynamic reuse distances," In MICRO-45 '12. Washington, DC, USA: IEEE Computer Society, 2012.

[7] J. Gaur, M. Chaudhuri *et al.*, "Bypass and insertion algorithms for exclusive last-level caches," In ISCA '11. New York, NY, USA: ACM, 2011.

[8] J. Gaur, R. Srinivasan *et al.*, "Efficient management of last-level caches in graphics processors for 3d scene rendering workloads," In MICRO '13. Davis, CA, USA: IEEE Computer Society, 2013.

[9] S. Grauer-Gray, L. Xu *et al.*, "Auto-tuning a high-level language targeted to gpu codes," In InPar '12, May 2012.

[10] B. He, W. Fang *et al.*, "Mars: A MapReduce framework on graphics processors," In PACT '08. New York, NY, USA: ACM, 2008.

[11] A. Jaleel, K. B. Theobald *et al.*, "High performance cache replacement using re-reference interval prediction (RRIP)," In ISCA '10. New York, NY, USA: ACM, 2010.

[12] W. Jia, K. A. Shaw *et al.*, "Characterizing and improving the use of demand-fetched caches in GPUs," In ICS '12. New York, NY, USA: ACM, 2012.

[13] W. Jia, K. A. Shaw *et al.*, "MRPB: Memory request prioritization for massively parallel processors," In HPCA-20 '14, 2014.

[14] A. Jog, O. Kayiran *et al.*, "OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance," In ASPLOS '13. New York, NY, USA: ACM, 2013.

[15] S. M. Khan, Y. Tian *et al.*, "Sampling dead block prediction for last-level caches," In MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010.

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

[17] *The OpenCL C SpecificationVersion: 2.0*, Khronos OpenCL Working Group, July 2013.

[18] A.-C. Lai, C. Fide *et al.*, "Dead-block prediction & dead-block correlating prefetchers," In ISCA '01. New York, NY, USA: ACM, 2001.

[19] J. Lee and H. Kim, "TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture," In HPCA '12. Washington, DC, USA: IEEE Computer Society, 2012.

[20] H. Liu, M. Ferdman *et al.*, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," In MICRO '41. Washington, DC, USA: IEEE Computer Society, 2008.

[21] V. Mekkat, A. Holey *et al.*, "Managing shared last-level cache in a heterogeneous multicore processor," In PACT '13. Piscataway, NJ, USA: IEEE Press, 2013.

[22] V. Narasiman, M. Shebanow *et al.*, "Improving GPU performance via large warps and two-level warp scheduling," In MICRO-44 '11. New York, NY, USA: ACM, 2011.

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

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

[25] *CUDA C Programming Guide v5.5*, NVIDIA Corporation, July 2013.

[26] M. Rhu, "A locality-aware memory hierarchy for energy-efficient GPU architectures," In MICRO '13. Davis, CA, USA: IEEE Computer Society, 2013.

[27] T. G. Rogers, M. O'Connor *et al.*, "Cache-conscious wavefront scheduling," In MICRO '12. Washington, DC, USA: IEEE Computer Society, 2012.

[28] I. Singh, A. Shriraman *et al.*, "Cache coherence for GPU architectures," In HPCA '13, 2013.

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

[30] X. Xie, Y. Liang *et al.*, "An efficient compiler framework for cache bypassing on GPUs," In ICCAD '13, 2013.