Locality-Driven Dynamic GPU Cache Bypassing

Chao Li *
North Carolina State University
cli17@ncsu.edu

Shuaiwen Leon Song
Pacific Northwest National Lab
Shuaiwen.Song@pnnl.gov

Hongwen Dai
North Carolina State University
hdai3@ncsu.edu

Albert Sidelnik
NVIDIA Research
asidelnik@nvidia.com

Siva Kumar Sastry Hari
NVIDIA Research
shari@nvidia.com

Huiyang Zhou
North Carolina State University
hzhou@ncsu.edu

ABSTRACT

This paper presents novel cache optimizations for massively parallel, throughput-oriented architectures like GPUs. L1 data caches (L1 D-caches) are critical resources for providing high-bandwidth and low-latency data accesses. However, the high number of simultaneous requests from single-instruction multiple-thread (SIMT) cores makes the limited capacity of L1 D-caches a performance and energy bottleneck, especially for memory-intensive applications. We observe that the memory access streams to L1 D-caches for many applications contain a significant amount of requests with low reuse, which greatly reduce the cache efficacy. Existing GPU cache management schemes are either based on conditional/reactive solutions or hit-rate based designs specifically developed for CPU last level caches, which can limit overall performance.

To overcome these challenges, we propose an efficient locality monitoring mechanism to dynamically filter the access stream on cache insertion such that only the data with high reuse and short reuse distances are stored in the L1 D-cache. Specifically, we present a design that integrates locality filtering based on reuse characteristics of GPU workloads into the decoupled tag store of the existing L1 D-cache through simple and cost-effective hardware extensions. Results show that our proposed design can dramatically reduce cache contention and achieve up to 56.8% and an average of 30.3% performance improvement over the baseline architecture, for a range of highly-optimized cache-unfriendly applications with minor area overhead and better energy efficiency. Our design also significantly outperforms the state-of-the-art CPU and GPU bypassing schemes (especially for irregular applications), without generating extra L2 and DRAM level contention.

*A large portion of this work was done when Chao Li was an intern at PNNL. This work is supported by NSF grant #1216569 and DOE ASCR Beyond Standard Modeling (BSM) project. The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830.

©2015 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the United States Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

ICS '2015 Newport Beach, California USA

Copyright © 2015 ACM 978-1-4503-3559-1/15/06 ...$15.00.

http://dx.doi.org/10.1145/2751205.2751237 .

Categories and Subject Descriptors

C.1.2 [**Multiple Data Stream Architecture**]: Single-Instruction, Multiple-Data processors (SIMD)

General Terms

Design, Experimentation, Performance, Energy Efficiency

Keywords

GPU architecture Optimization, Locality, Cache Bypassing

1. INTRODUCTION

Massively parallel, throughput-oriented processors such as graphics processing units (GPUs) leverage high thread-level parallelism to overlap long latency memory accesses with computation. On-chip caches, especially L1 data caches (L1 D-caches) which were designed to improve the performance of irregular workloads without programming scratchpads [2, 5, 4], remain critical to provide high-bandwidth low-latency data accesses. However, the limited L1 D-cache capacity becomes a performance bottleneck as the working set of the massively threaded applications often exceeds the L1 D-cache capacity, causing severe *thrashing* [24, 15]. More importantly, large number of the incoming memory requests with no or low reuse may evict cache lines with high reuse, resulting in *cache pollution*. In this case, even advanced cache replacement policies (e.g. RRIP [14] and SHiP [29]) are ineffective to address such contention problems on GPUs [24]. Furthermore, the massive multithreading on GPUs can cause the unpredictability of cache locality and various resource congestion (e.g. MSHR allocation failures) [5, 16].

To alleviate contention and avoid early eviction, cache bypassing schemes have been proposed for CPUs [19, 12, 13, 18] and GPUs [15, 30, 31, 27]. The CPU-based approaches are usually designed for last level caches (LLCs), where data locality is already filtered by previous level(s) of caches. But the poor locality of GPU workloads and resource congestion impose difficulty for them to make robust predictions and they often increase L2 and DRAM level traffic [11] (Section 6.1(a)). GPU-based bypassing schemes are generally conditional/reactive bypassing (e.g., bypass upon unavailable resources [15] or coarse-grained bypassing on warps or thread-blocks [30, 31, 27]) which can incorrectly bypass accesses with good reuse and cause memory pipeline stalls (Section 6.1(a)). None of the above approaches is a *preventive* scheme considering the uniqueness of the *data locality* of GPU access streams, which often contain a non-trivial number of requests with no/low reuse and/or distant re-reference intervals caused by frequent bursts of references (Section 3.2).

Moreover, a fully-adaptive bypassing scheme is required to maintain the efficiency of workloads with good caching behavior, which is often neglected by previous approaches [24, 15, 12, 11] (Section 6.1(b)).

In this paper, we propose a locality-driven dynamic bypassing design that automatically filters the access stream on cache insertions based on reuse frequency of accesses, so that only the data with high reuse and short reuse distances [12] are stored in the L1 D-cache. For area and energy efficiency, we propose to *decouple the tag and data stores* of the existing L1 D-cache and integrate the locality filtering capability into the tag store through simple and cost-effective hardware extensions. Our design uses separate replacement policies to manage the decoupled tag and data store, such that the reuse information of the program and temporal locality of the data lines can be preserved. Overall, this paper makes the following contributions:

(1) Through a detailed analysis on the reuse characteristics of GPU workloads, we identify the key inefficiency of the conventional thrashing and stall-prone GPU cache design: *irregular cache-unfriendly* memory accesses resulting in contention at various cache levels.

(2) We propose a locality-driven dynamic bypassing solution that is cost-effective and requires no profiling or runtime prediction. We demonstrate that our design can significantly improve the performance and energy efficiency of *irregular cache-unfriendly* workloads, while maintaining the efficiency for regular workloads with favorable caching behavior.

(3) Our design achieves significant performance improvements over the baseline caches and outperforms the state-of-the-art CPU (*PDP-best* [12]) and GPU (*MRPB* [15]) cache bypassing schemes, by dramatically reducing various types of cache contention without generating extra L2 and DRAM traffic.

The remainder of the paper is organized as follows: Section 2 discusses the background, the GPU memory hierarchy in particular. Section 3 dissects the effectiveness of L1 D-caches for various workloads. Section 4 details our new design. Section 5 presents the experimental methodology. The evaluation of our design is shown in Section 6. Section 7 discusses the related work and Section 8 concludes.

2. BACKGROUND

2.1 Baseline Architecture

This work proposes microarchitectural improvements to a massively parallel processor such as GPU architectures. Such processor consists of multiple SIMD cores, also known as streaming multiprocessors[1] (SMs) in NVIDIA GPUs or Computing Units in AMD GPUs. As shown in Figure 1, each SM follows the single instruction-multiple threads (SIMT) execution model by fetching and decoding each instruction for a group of threads called a warp. All threads in a warp execute in lock-step in the SIMT backend. In the issue stage, a warp scheduler will select one of the ready warps to issue into the computing/memory pipeline stage. GPUs provide multiple types of memory units to improve the memory bandwidth such as L1 D-caches and shared memory (or scratchpads). Global memory and scratchpad accesses are served through L1 D-cache and shared memory respectively. Both L1 D-cache and shared memory utilize the same hardware structure and the capacity can be configured through a run-time API. While shared memory can be explicitly managed by programmers, L1 D-cache is implicitly controlled by

[1]Without specified mention, NVIDIA terminology will be used throughout the paper to illustrate our work. However, the idea applies to a wide range of massively parallel architectures.

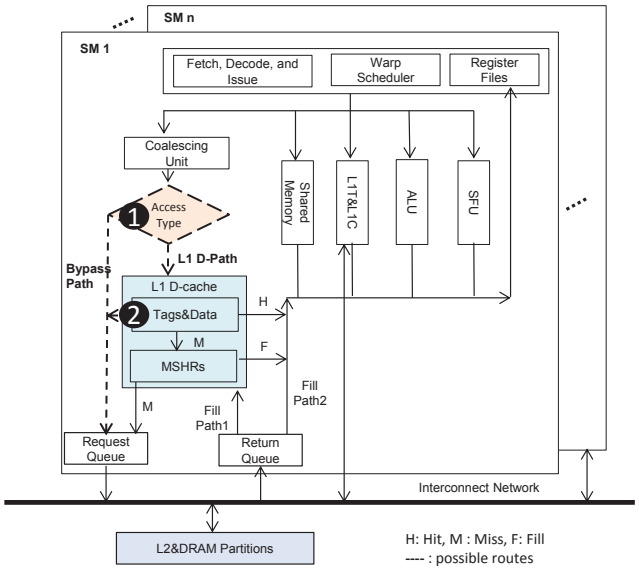


Figure 1: Memory request handling in the baseline architecture.

hardware to exploit data locality. As GPU workloads with irregular memory access patterns are becoming prevalent, effective utilization of explicitly managed memory becomes difficult. This in turn increases the importance of efficient L1 D-cache designs. All the SMs are connected by an interconnected network to the partitioned memory module, each with its own L2 data cache and DRAM partition. To save bandwidth [25], L1 is typically write through, with either write-allocate [2] or write-no-allocate [5, 4], while L2 is write back with write-allocate. Victim caches and hardware prefetching are traditionally not enabled in GPUs [6].

2.2 Baseline Memory Request Handling

When a memory instruction is dispatched into the memory pipeline, the load_store (LD_ST) units will identify the memory access type and dispatch it into different memory units (shared memory, L1 D-cache, etc.). The requests for global and local memory data from threads in the same warp will go through the coalescing unit to generate as few L1 D-cache line-sized requests as possible. Then for these requests, there are two possible paths depending on whether an access is cacheable, as shown in Figure 1. For cacheable accesses, the first path, which sends the memory requests into L1 D-cache and is labeled as '**L1 D-path**', is used. On a cache hit, a request will be served by sending data to the register file immediately. On a cache miss, the miss handling logic will first check the miss status holding register (MSHR) to see if the same request is currently pending from prior ones. If so, this request will be merged into the same entry and no new data request needs to be issued. Otherwise, a new MSHR entry and cache line will be reserved for this data request. A cache status handler may fail on resource unavailability events such as when there are no free MSHR entries, all cache blocks in that set have been reserved but still haven't been filled, the miss queue is full, etc. If any of these events occurs, the memory pipeline will stall and this request will retry every cycle until needed resources are freed. Considering the small number of cache lines and MSHR entries, these resources can be quickly occupied if all memory requests are diverted into **L1 D-Path**.

The second path is for un-cacheable accesses, such as global memory accesses in NVIDIA's Kepler architecture [4]

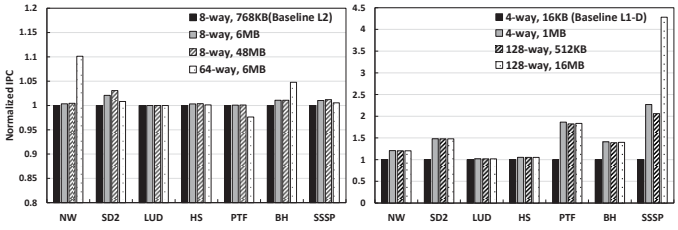


Figure 2: L1 (right) and L2 (left) level contention study by increasing their associativity and capacity. It clearly shows that the major performance bottleneck resides at the L1 D-cache level for CNF workloads.

(not cached in Kepler's L1). It diverts the memory requests to bypass the L1 D-cache (labeled as '**Bypass Path**' in Figure 1) and directly sends requests through an interconnect into the next level memory hierarchy. On the response fill from the return queue, there are two paths to fill the data correspondingly. If the original memory request follows the **L1 D-Path**, then the response data will be filled into the reserved cache line in L1 D-cache and the corresponding MSHR entry is marked as filled ('**Fill Path 1**'). Otherwise, the data will directly write back into the register file ('**Fill Path 2**').

Compared to the **Bypass Path**, the **L1 D-Path** lowers access latencies if requested data already resides in the cache. However, memory accesses following this path may incur high stall cycles for resource contention due to massive parallelism, and also thrash the cache by evicting out the data lines that may be reused shortly. Ideally, an efficient locality monitoring mechanism should exist to divert memory requests with high data reuse into the L1 D-cache, while other requests that have no or low data reuse will be directed to the **Bypass Path** instead of contending on the cache resources.

3. GPU CACHE INEFFICIENCY AND WORKLOAD ANALYSIS

App Name	N-IPC	Num Dynamic Insts	Type	Suite
PTF	1.36	7022693	CNF	[9]
SD2	1.36	740966400	CNF	[9]
NW	1.24	1617408	CNF	[9]
SSSP	1.21	385881828	CNF	[8]
LUD	1.17	5875200	CNF	[9]
HS	1.05	109686484	CNF	[9]
BH	1.04	7909748	CNF	[8]
CFD	1.00	9308160	CI	[9]
LFK	1.00	7516393326	CI	[9]
GS	1.00	12648	CI	[9]
FFT	1.00	104005632	CI	[33]
MYC	1.00	26986736	CI	[9]
PF	1.00	649231040	CI	[9]
SD1	0.97	8281816	CF	[9]
HT	0.55	8811310884	CF	[9]
MM	0.54	58851328	CF	[21]
BT	0.41	542310058	CF	[9]
BP	0.23	190054784	CF	[9]

Table 1: Application categorization based on cache bypassing impact. CNF: Cache Unfriendly, CI: Cache Insensitive, CF: Cache Friendly. IPC is normalized to the case of all taking L1 D-Path (N-IPC). The selected applications include Particular Filter (PTF), Srad2 (SD2), Needleman-Wunsch (NW), Single-Source Shortest Path (SSSP), LU Decomposition (LUD), Hotspot (HS), Barnes-Hut (BH), CFD Solver (CFD), Leukocyte (LFK), Gaussian Elimination (GS), Fourier Transformation (FFT), Myocyte (MYC), PathFinder (PF), Srad1 (SD1), Heartwall (HT), Matrix Multiplication (MM), B+Tree (BT), and Back Propagation (BP).

To evaluate the efficacy of the GPU caches, we first characterize a wide range of GPU applications (shown in Table 1) covering various cache sensitivity and memory access patterns (e.g., applications with *highly irregular access patterns* such as BH and SSSP). These applications are selected from multiple widely used benchmark suites (including GPGPUsim benchmarks [7], CUDA SDK [3], AMD SDK [1], Rodinia [9], and Lonestar suites [8], etc) and represent production GPU codes which are optimized and hand-tuned using explicitly managed scratchpad memory extensively. We quantify how much performance improvement, which is measured using normalized instruction per cycle (N-IPC), each application can gain if all memory requests have taken the **Bypass Path** compared to all taking the **L1 D-Path** in Figure 1. This IPC improvement is shown in Table 1, where the eighteen applications are sorted in descending order. Applications whose IPC improvement is greater than 1 are classified as *Cache Unfriendly* (**CNF**), indicating the current L1 D-cache management has detrimental impact on their performance. Applications whose IPC values are not impacted are classified as *Cache Insensitive* (**CI**), as whether having a L1 D-cache or not has negligible effects on their performance. The remaining applications whose IPC improvement is less than 1 are *Cache Friendly* (**CF**): workloads that perform better if all accesses go through L1 D-cache. Section 5 provides a detailed description of these applications and the baseline architecture the results from this section were collected from.

3.1 GPU Cache Inefficiency

The memory access stream of GPU workloads has two characteristics: a mix of data requests with different reuse frequencies and different reuse distances [12]. For instance, in the Fermi architecture [5], incoming accesses will enter into the L1 D-cache without being checked on whether they have future reuse. All data accesses contend with each other for limited cache resources, resulting in memory pipeline stalls. To the other extreme, similar to the bypassing approach used in Table 1, Kepler [4] will bypass all global memory accesses from L1, only handling register spills to the local memory.

In general, there are three types of contention at the L1 D-cache level: **(a) Inter-warp contention: capacity misses**. Current GPU architectures (e.g. Fermi and Kepler) commonly have 16∼48 KB capacity of L1 D-cache [5, 4], but the memory footprint of the applications is typically one to two orders of magnitude larger, which causes severe thrashing. Data blocks get evicted out frequently before any reuse happens, especially when the reuse distance is long. More importantly, memory requests with no reuse may evict cache lines that have high reuse, resulting in cache pollution. **(b) Intra-warp contention: conflict misses**. This is caused by the concurrent threads within the same warp (32 threads) accessing to the same cache set (current GPU L1 has much lower set-associativity than 32). **(c) Other resource congestion**. They include resource allocation failures from limited MSHRs and miss queue entires. These resources can be quickly occupied and cause stalls if data requests come into L1 without being filtered based on their reuse patterns.

To reduce the contention described above, we apply the most direct optimization, which is increasing the capacity and associativity of the L1 and L2 caches, and observe how it affects overall performance. We intentionally increase the cache associativity and capacity to a large value, which is expensive and impractical to implement in real GPU cache designs due to increased access latency, area, and power consumption. Figure 2 shows several observations for the CNF

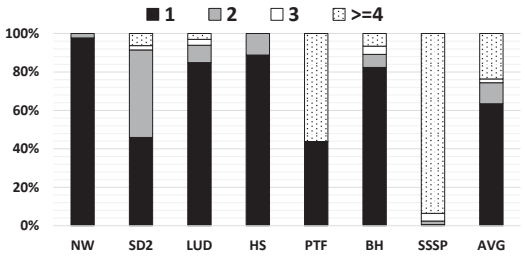


Figure 3: Reuse breakdown for CNF applications. It shows the percentage of addresses in L1 D-cache have been referenced by m number of times. The values of m are shown in the legend.

applications, which encountered the most cache contention: (1) Compared to the L1 D-cache, CNF's performance is insensitive to the capacity and associativity increase of L2, which indicates that L2 is not the major performance bottleneck. (2) CNF's performance can be improved by increasing L1 D-cache's size and associativity (to full associativity), but it is mainly bounded by cache capacity. (3) Even with these impractical cache configurations, the performance improvements for some CNF applications are still not significant, including NW, LUD, HS, and BH. These findings motivate us to develop a cost-effective cache bypassing mechanism for GPUs to maximally reduce the contention by only letting the most useful data into the L1 D-cache.

The state-of-the-art bypass policy, *protection distance prediction (PDP)* [12], has been proposed for CPU LLCs. This technique applies a *protection distance (PD)* counter to each cache line and uses it to time how many accesses are left for this line to be protected. If $PD = 0$, this line is marked as *unprotected* and can be replaced by the incoming accesses. If there are no *unprotected* lines left, cache is bypassed. However, applying PDP-based approaches (static or dynamic) to a GPU L1 D-cache can encounter several problems [11]. First, due to warp interleaving and resource congestion in GPUs, as a hit-rate based scheme designed for single thread processors, PDP may not improve the performance of CNF applications because the hit rate does not directly correlate to performance in GPUs (see Section 6.1(c)). Second, by augmenting a cache line with a protecting distance, PDP is susceptible to bypass the blocks with frequent data reuse and short reuse distances, but keeps the staled blocks with very long reuse distances for a protection interval (see Section 6.1(d)), causing cache pollution. Third, many of the CF applications shown in Table 1 have been optimized for cache performance. Therefore, any request bypassing caused by inaccurate PDP prediction can result in a significant performance degradation (see Section 6.1(b)). Finally, it is difficult for PDP to predict applications with irregular memory accesses, such as BH and SSSP (see Section 5 and 6.1(a)).

3.2 Impact of Applications On Performance

An application's intrinsic memory access and reuse pattern can also affect its performance. In this section, we use the application characterizations from Table 1 to provide insights in making design decisions for our proposed approach.

In Table 1, the CI Category contains of six applications where enabling the L1 D-cache has no performance impact. We categorize the reasons behind such insensitivity into four typical scenarios: (1) workloads have no read access from the global memory, such as LEK and CFD; (2) workloads have the characteristics of intensive branching and very little memory access, such as MYC and GS, which also have very low IPC due to branch divergences; (3) streaming workloads, such as FFT, in which the global memory access happens only for loading the input signal sets into the shared memory; (4) workloads have high memory level parallelism, such as PF, where the critical path is determined by the first memory access miss. Because of their performance insensitivity to the L1 D-cache, the CI applications are not the focus of our study. However, we include them to prove the robustness of our proposed design.

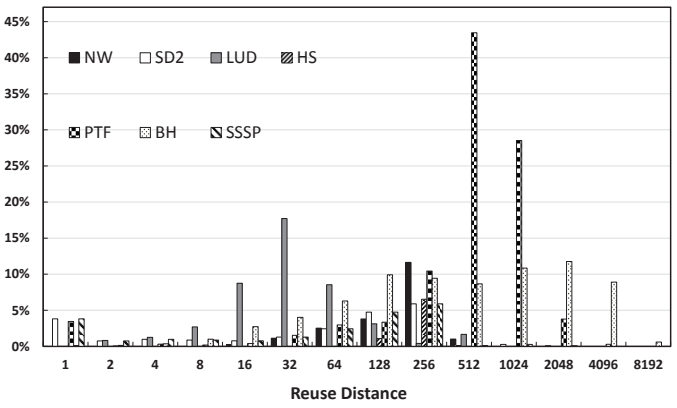


Figure 4: The reuse distance histograms of L1 D-cache access stream of the CNF applications. The reuse distance is computed at cache access level instead of cache structure level (e.g. the reuse distance for access pattern ABCA is 2).

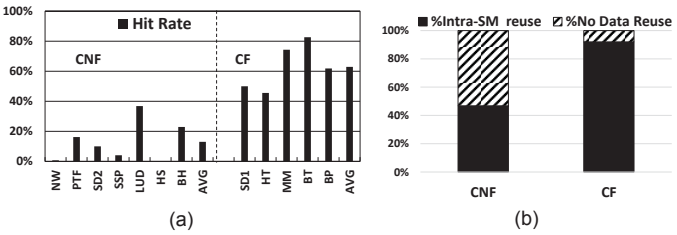


Figure 5: Hit rates and data reuse characteristics of GPU workloads: (a) Hit rates of CNF and CF applications; (b) The percentage of data accesses with and without data reuse.

The cache sensitive workloads include both CNF and CF applications. For CNF workloads, bypassing all memory requests from L1 D-cache improves the performance of NW, PTF, and SD2 by 24.2%, 36.8% and 36.6%, respectively. This is a direct result of the low cache performance across the CNF workloads, as shown in Figure 5(a). Compared with the CF applications, the overall cache hit rate in CNF is much worse. For example, the hit rates for NW and HS are smaller than 1%. LUD has the highest hit rate in CNF (36.8%). However, 62.8% of its hits are write hits, which are useless due to the write evict policy in baseline GPUs [5, 4]. On average, the cache hit rate for CNF workloads is only 8.9%, while the average for the CF workloads is as high as 55.4%. To understand why CNF workloads have such low cache performance, we evaluate the data locality of each application and quantify the data reuse rate of their memory stream in each SM. As shown in Figure 5(b), CNF workloads have a larger portion of data requests without any reuse, 53.2% on average compared to only 7.9% in the CF category. To analyze the characteristics of the reuse in the CNF workloads, we present the reuse distance histograms of these workloads in Figure 4. As shown in Figure 4, the reuse distances are relatively high when the cache block size is 128 bytes (baseline architecture). Such large reuse distances (>128) present a challenge for the limited number of blocks

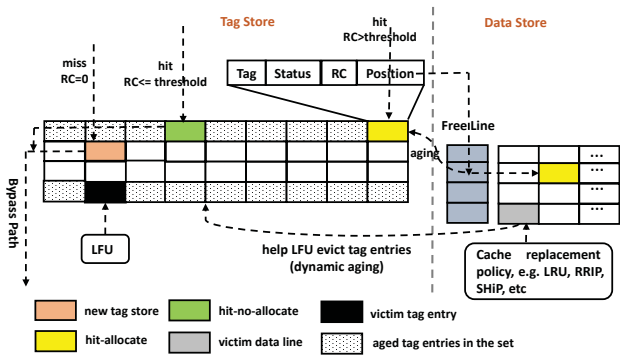


Figure 6: Decoupled L1D cache design diagram.

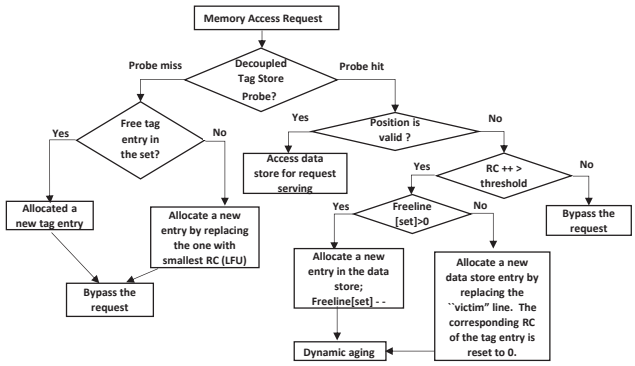


Figure 7: Operation flow chart of the Decoupled L1D.

(i.e., limited capacity) in the L1 D-cache. Also, Figure 3 shows that memory requests for the majority of CNF applications (other than PTF and SSSP) are only reused less than three times. These data accesses with no or low reuse cause two problems: (1) they will contend with accesses with high reuse for resources and resulting in the average stall time for CNF workloads as high as 48.04% of the total execution time; (2) such accesses without any reuse are scattered inside the memory access stream, which indirectly increases the reuse distance of accesses with reuse. Finally, without accurate locality information, deciding which memory requests should enter L1 D-cache is difficult for applications with irregular access patterns (e.g. SSSP), which has been neglected by the previous studies [11, 15, 24, 30].

4. DESIGN METHODOLOGY

The focus of this work is to design efficient bypassing for a GPU L1 D-cache that can dynamically divert memory requests based on reuse patterns. As shown in Figure 1, there are two design points available to implement such mechanism: an independent hardware component between the coalescing unit and L1 D-cache (❶ in Figure 1), and the existing L1 D-cache integrated with the locality filtering capability (❷ in Figure 1). We choose ❷ over ❶ because both structures consist of entries that can be identified as tags (e.g. L1 tag store) so such redundancy in ❶ will increase access latency by performing additional tag checks per access. Also, ❷ will likely have smaller area and higher energy efficiency. Accordingly, we propose to decouple the tag and data stores of the existing L1 D-cache and integrate the locality filtering capability into the tag store through minor hardware expansions. In this way, we can leverage the management of the independent tag store to control which memory requests can allocate data lines in the data store. By taking advantage of the tag store's smaller entry size (8∼9 bytes) compared to that of the data store (128 bytes per line), additional entries in each set can be added in the tag store with lower overhead such that it can capture the locality information of a working set larger than the data store size. We name our new design "**Decoupled L1D**" and its diagram is shown in **Figure 6**.

4.1 Decoupled L1D: Structure

As shown in Figure 6, the tag and data store in the Decoupled L1D cache are independent structures although we choose to let them have the same number of sets to simplify the management. The decoupled tag store has expanded the original tag store with more entries in each set, and each tag entry has also been padded with more fields. In addition to the original fields, including the address tag, a status field, and a LRU counter, the new tag store entry now contains a *Reference Count (RC)* field and a *Position* field. The *RC* field (6-bit) holds the reference frequency (reuse) accumulated for the address. The *Position* field (2-bit) connects a tag store entry with a data line in the data store using a pointer to record the data line's position. Once a data line in the data store is allocated, the corresponding tag store entry needs to set the *Position* field accordingly. The new data store in our Decoupled L1D has the same structure as the original one, except that each set has added one field named *Free_Line*, indicating how many free lines are available for use in the set. This field will be updated upon a new data line allocation or an eviction.

4.2 Decoupled L1D: Operations

Instead of having uniform operations in the current L1 D-cache design, our Decoupled L1D enables separate sets of update and replacement logic for the tag and data stores. Also, we add a new cache request status *bypass* into the existing four statuses, i.e., *hit, hit pending, miss, and reservation failed*, of a L1 D-cache request. Figure 6 and 7 describe how the Decoupled L1D operates.

In the initial architectural status, each tag-store entry and data line is cleared (*Position* field is set to invalid). The *Free_Line* field is initialized with the associativity size of each set in the data store. **On a probe miss of the tag store**, a new tag entry is allocated, and a bypass status is returned, as shown in Figure 7. A bypass will not trigger the cache miss handler but instead it will directly take the **Bypass Path** (Figure 6) without allocating a line in the data store. If there are no free tag entries in that set, the entry with the smallest *RC* is taken as the replacement victim. In other words, the LFU (Least Frequently Used) replacement policy is used for the decoupled tag store. Since the way to update *RC* can significantly impact the lifetime of a tag entry, we apply a customized dynamic aging[2] scheme to the *RC* fields of the tag entries in the following scenarios: upon an allocation or eviction in the **data store**, the *RC* values of all the corresponding **tag entries** in that set (not including the allocated or evicted tag entry) are reduced by 1. Thus, the *RC* values of **stale entries** (i.e. entries with no future reference or long re-reference interval that still remain in the tag store) are reduced and eventually become small enough to be evicted out of the tag store.

On a probe hit in the decoupled tag store, there are two different operations depending on whether the *Position*

[2]We have explored several aging schemes (e.g. bit-shifting RC only upon evictions in the data store) but do not report them here due to space limitations.

field has pointed to an allocated data line. If the field has been set with a data-store line position, this memory request will proceed as a cache hit or cache hit-pending, similar to the original L1 D-cache. Otherwise, it will increase the *RC* value of the tag entry and then compare it with a *locality threshold*, which is pre-defined based on workload characteristics so that the data blocks with no or low reuse will not be inserted into the L1 D-cache. The subsequent program flow after the comparison is shown in Figure 7. If there are no free data lines in a selected set upon a data-line allocation, the data store's replacement policy (e.g. LRU, SHiP, RRIP, etc) will be triggered to find the victim line for eviction. Then, the RC value of the corresponding tag entry is set to 0 and the rest of the tag entries in the set will be aged. This ensures that the temporal locality of the data lines is preserved in our design by using the replacement policy from the original L1 D-cache. The associated parameters introduced in this section (including the number of tag entries, the set associativity, and the locality threshold) have direct impact on the area, power and performance of our design. We explore them in Section 6.2.

Fairness: For a fair system, the decision on allocating and replacing tag entries must **closely match the data reuse frequency and distance**. In our design, the locality of the program is preserved. If a data block has high reuse, its *RC* will continue to accumulate and eventually pass the threshold to allocate a new line in the data store for capturing subsequent reuses. Also, the state-of-the-art cache replacement policies (e.g. LRU, RRIP, etc) are used in the data store to help evict the tag entries (dynamic aging) that exhibit a distant re-reference interval caused by frequent bursts of references. In this way, the tag and data stores are circulated in a sustainable way with minimum number of stale tag entires generated.

4.3 SM Dueling

Our proposed design aims to improve performance of CNF applications. For CF applications, whose performance benefits greatly from utilizing the L1 D-cache, a locality threshold on *RC* may delay the data to be stored in the cache for reuse. Therefore, we can disable the *RC* checking for CF workloads by setting the locality threshold to zero, i.e., all accesses go to the L1 D-cache. For our decoupled tag store, the newly added tag entries can be power gated to reduce the energy overhead. To decide which mechanism to use for an application during execution, we employ a simple, but effective SM dueling technique [23] based on the insights from Figure 5a. Assume that we have a GPU with N SMs (N≥2). Initially, we assign *SM_0* to use our new design and *SM_1* to use the original L1 D-cache. The remaining SMs can choose to use either type. At a pre-defined time interval (see Table 3), *SM_0* and *SM_1* will compare their cache miss rates: if *SM_0* has a higher cache miss rate than *SM_1*, then all the SMs will start using the original L1 D-cache; otherwise all the SMs will enable the new design.

5. EXPERIMENTAL METHODOLOGY

Simulation Environment: Our proposed design is evaluated using GPGPUsim V3.2.2 [7], which is a widely used cycle-accurate simulator for GPU architecture research. The baseline architecture is modeled based on a generic NVIDIA Fermi GPU [5] and its configuration is shown in Table 2. Our new design can be similarly applied to other recent GPU architectures as well, such as NVIDIA Kepler and Maxwell [4]. The design choices for our Decoupled L1D (based on a 16KB L1 D-cache) is shown in Table 3, which will be validated in Section 6.2 (design space exploration).

SIMT Core (SM)	15 cores, SIMD width=32, 1.4GHz, 5-stage pipeline
Max/SM	1536 threads, 32768 registers, 48 warps, 32 MSHRs with 256 entries
L1 Cache/SM	16KB/core, 128B line, 4-way assoc, 1-cycle hit latency
Shared Mem/SM	48 KB; 32 banks; 3-cycle latency; 1 access per cycle
Unified L2 Cache	768 KB, 128KB/bank, 6 banks, 128B line, 16-way assoc
DRAM	6 memory channels, BW: 48 bytes/cycle, 1.4 GHz
DRAM Schedule Queue	Size = 16 and Out of order (FR-RCFS)
Warp Scheduling Policy	Greedy then oldest (GTO)

Table 2: Baseline architecture configuration.

Decoupled L1D	Tag Store		Data Store	
	#entries	256	#lines	128
	structure	32 set, 8-way	32 set, 4-way	
	replacement policy	LFU dynamic aging	LRU, SHiP, RRIP, etc.	
	threshold	2	N/A	
SM Dueling	Initially, SM0– new design; SM1–baseline cache; the rest SMs follow SM1. Time interval: 500 cycles. Miss rate difference threshold: 10%.			

Table 3: Configurations of our proposed Decoupled L1D.

Benchmarks: All 18 applications used in this study are shown in Table 1, which represents a wide range of optimized real-world GPU applications. Since shared memory is extensively used, the performance tends to be less sensitive to the L1 D-cache, which makes them the perfect candidates to show whether our proposed design can still improve performance over optimized codes. We evaluate 14 workloads from Rodinia Benchmark [9] with their default inputs. We include two additional applications: Matrix Multiplication (MM), a highly efficient version using tiled cache [21]; and Fast Fourier Transformation (FFT), an optimized version fully utilizing on-chip memory [33]. We also include two widely-used workloads: Barnes Hut N_body Simulation (BH) and Single-Source Shortest Paths (SSSP) from Lonestar GPU suite [8], both of which exhibit irregular memory access patterns. BH exhibits memory irregularity from data-dependent memory accesses, and SSSP has access irregularity from the memory traversing in a recursive iteration. In our experiments, all workloads run to completion on the simulator.

6. RESULTS AND ANALYSIS

6.1 Overall Performance Evaluation

To evaluate the performance of our proposed Decoupled L1D cache design, we compare it with several related schemes and show the results and detailed analysis. As discussed previously, Decoupled L1D uses state-of-the-art replacement policies (e.g. LRU, RRIP [14], etc.) in the data store to preserve data lines' temporal locality. To make a fair comparison, we use the same cache replacement policy (a version of LRU implemented in GPGPUsim) as the basis for all the following schemes:

BL: Baseline Architecture. This configuration is shown in Table 2. All the global/local memory accesses will be inserted into the L1 D-cache. If resources are unavailable, the memory pipeline is stalled. This architecture favors CF workloads (Table 1).

BALL: Bypass all memory accesses from the L1 D-cache. All the global/local memory accesses will be diverted into

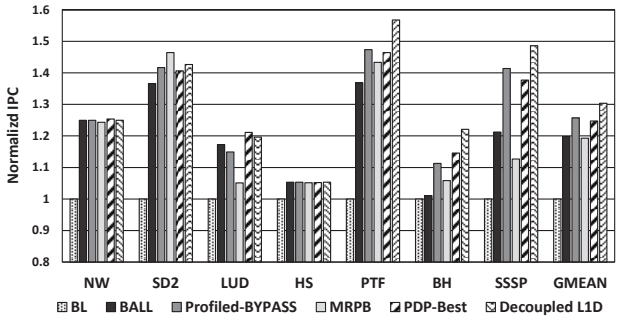


Figure 8: Performance comparisons of various cache management and bypassing schemes for the CNF workloads.

the Bypass Path (Figure 1). L1 D-cache is disabled. *BALL* favors CNF workloads (Table 1).

Profiled-BYPASS: During a profiling run, we record the reference frequency for each data block by going through every coalesced memory request before they hit L1 D-cache. This profiled information is then used to direct fine-grained L1 bypassing (acted as a software filter) in a new run. This static approach is used to evaluate performance benefits that can be achieved when the reuse information of a workload is already available. Applying such an approach for bypassing has three major downsides: (1) it is dependent on the input and thread-configuration; (2) memory traces could be non-deterministic for each run; and (3) applying bypassing in a new run based on the previous profiling information may change the completion time of different thread blocks (TBs) and their associated dispatch order.

We implemented three types of profiling-based filters: (1) a per SM local filter recording the intra-SM data reuse (local), (2) a kernel-level global filter recording the inter-SM data reuse (global), and (3) a combined local+global filter. All three filters have unlimited entries to store workloads' memory addresses and their reference counts. Different locality thresholds from 1 to 16 were tested for the best performance. Among them, the global+local filter achieved the best average performance over BL. The reason is that with bypassing enabled, the thread block execution order is altered from the profiling run and the inter-SM reuses may become intra-SM reuse. From this point on, we will only use the global+local filter (named "**Profiled-BYPASS**") for comparisons with other mechanisms. The profiling costs are ***not*** included in the figures of this section.

MRPB: The *state-of-the-art* GPU dynamic bypassing approach [15]. When any of the resource unavailable events happens that may lead to a pipeline stall, the memory requests are bypassed from the L1 D-cache until resources are available. MRPB includes memory request reordering queues, which are also modeled in our experiment.

PDP-best: The *state-of-the-art* CPU bypassing approach for LLCs, as discussed in Section 3.1. Both static and dynamic PDP approaches have been proposed in [12], in which the performance upper bound of PDP is achieved by the static PDP approach. Thus we compare our design to PDP-best, which is the best performed static PDP-bypass with the optimal PD specific to each workload (by exhaustively searching all possible PDs).

Decoupled L1D: Our design as described in Section 4.

Based on the experimental results, we make the following observations and analysis:

(a) Performance Comparisons For CNF Workloads: Figure 8 shows that for CNF workloads our proposed Decoupled L1D achieves the highest performance, up to 56.8% and an average of 30.3% performance improvement (using the geometric mean) over *BL*. The performance comparisons among the different schemes highlights the importance of cache bypassing for GPUs. *BL* (all memory accesses go into the L1 D-cache) performs the worst because without any cache insertion management, memory requests with little to no reuse are diverted into the L1 D-cache, causing cache pollution and congesting the memory pipeline. *BALL* (all memory accesses bypass L1 D-cache) improves the performance for CNF workloads due to fewer memory pipeline stalls, however this mechanism completely disables the cache and overlooks data reuse in the memory access stream. As shown in Figure 5b, there are still over 40% of the data accesses that have reuse for the CNF workloads. *MRPB* improves CNF's performance over *BL* by a geometric mean of 19.2%, only bypassing when a resource (cache lines, MSHR, miss queue entry, etc.) unavailable event occurs, in order to reduce resource contention. The disadvantage of *MRPB* is that it does not consider any data reuse pattern in the access stream. In other words, *MRPB* could still let data with no or low reuse be cached and bypass data with high reuse. In comparison, our proposed *Decoupled L1D* enables a fine-grained and effective bypassing based on reference count, which can better capture the data reuse in an access stream. Figure 8 shows that our design outperforms *MRPB* by up to 36% and an average of 11% for CNF workloads. On average, our dynamic approach also achieves better performance than the *Profiled-BYPASS*. For some CNF workloads like BH, we even observe an up to 11% performance improvement, without even counting the profiling overhead in *Profiled-BYPASS*. This proves that our dynamic method is more flexible and practical than the profiling-based bypass, without encountering profiling overheads, inaccurate requests diverting, and the potential nondeterministic effects.

Comparing with PDP-best, for the irregular CNF workloads like BH and SSSP, our *Decoupled L1D* outperforms *PDP-best* by 8% and 11%. For PTF, which has long reuse distances and burst access patterns, *Decoupled L1D* outperforms *PDP-best* by 10.33%. For the regular CNF applications with very low reuse such as HS, NW, and LUD (Figure 3), our design performs as well as *PDP-best*. Based on [5, 4], the primary reason for creating a GPU L1 D-cache is for improving the efficiency of irregular workloads processing, since shared memory cannot address the irregularities at runtime. We argue that the **essential goal** of our design is to explore the potential efficiency improvement for irregular workloads using bypassing. Unlike the hit-rate based PDP approaches constrained by the protection distance (discussed in Section 3.1), our *Decoupled L1D* monitors the dynamic reuse frequency and reuse distances using the decoupled tag store, which better captures the reuse patterns for irregular applications. Additionally, with the customized dynamic aging, stale tag entries in the tag store can be evicted out timely to avoid cache pollution.

(b) Performance Comparisons For CF and CI Workloads: CF applications, which have favorable caching behavior and regular control, can suffer greatly from inaccurate bypassing due to disrupted data locality. In Figure 9, significant performance loss compared to *BL* has been observed for *BALL* (up to 77%), *MRPB* (up to 19%), and *PDP-best* (up to 30%). On the contrary, we do not observe performance loss for our design because *SM Dueling* determines if the new management scheme fits well with the workload pattern and dynamically turns bypassing on or off in all SMs (see Section 4.3). For CI workloads, as expected, our design

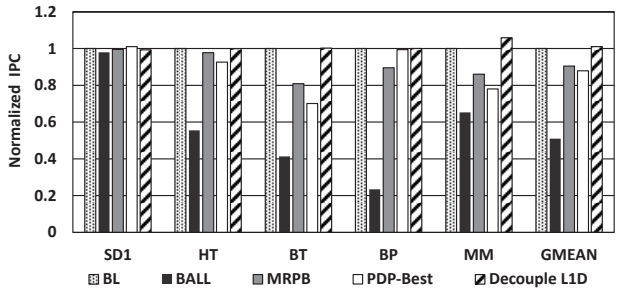


Figure 9: Performance comparisons of various cache management and bypassing schemes for the CF workloads.

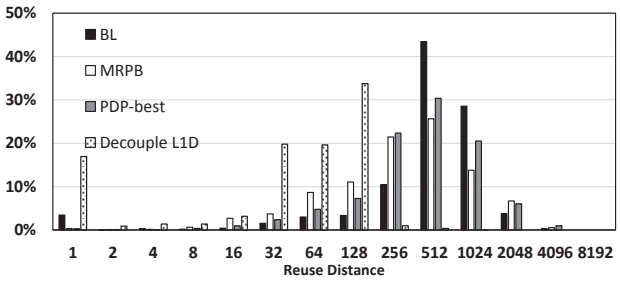


Figure 11: The reuse distance histogram of L1 access stream of PTF (a case with long reuse distance shown in Figure 4) shaped by different designs. The reuse distance is computed at the cache access level instead of cache structure level (e.g., the reuse distance for access pattern ABCA is 2).

along with other L1 D-cache management schemes has no effect on performance (with an average variance of 0.03% over the baseline performance).

(c) Hit Rate Improvement: Figure 10a shows that *Decoupled L1D* can significantly improve the L1 D-cache hit rate for CNF workloads over *BL* by an average of 38.5%. Our design also outperforms the hit-rate based scheme *PDP-best*, with significant improvements for SD2 (71%), LUD (24%) and SSP (13%). This also confirms that hit rate is not directly correlated to performance (Figure 8) on GPUs due to warp interleaving and resource congestion. Meanwhile, *MRPB* only achieves a 17.4% average hit rate due to its coarse-grained bypassing and lack of consideration for data locality. Furthermore, both NW and HS have negligible hit rates (<1%) across different mechanisms, caused by low reference counts (*RC*) for their memory requests (either 1 or 2 according to Figure 3).

(d) Shaping Cache-Friendly Memory Access: Using the CNF workload PTF as an example, Figure 11 shows that our design can shape a more cache-friendly memory access stream than other approaches. First, the data accesses that have a long reuse distance and therefore low reuse frequency will not be inserted into the L1 D-cache. As shown in Figure 11, the accesses with reuse distances of 512, 1024, and 2048 (such reuses require a cache with the capacity of 2048∗the 128 cache line size = 256kB) will be bypassed in our design to avoid cache pollution. Second, our design can reduce the reuse distance of data accesses by filtering out unfriendly memory accesses. For instance, the percentage of the reuse distance '1' has been significantly increased by our design and the overall reuse distance range is shrunk to better fit the limited cache capacity.

(e) Alleviating Various Contention: Through our *Decoupled L1D*, the footprint of the filtered L1 access stream for CNF applications is dramatically reduced by an average of 63.4% compared to the baseline. This in turn reduces

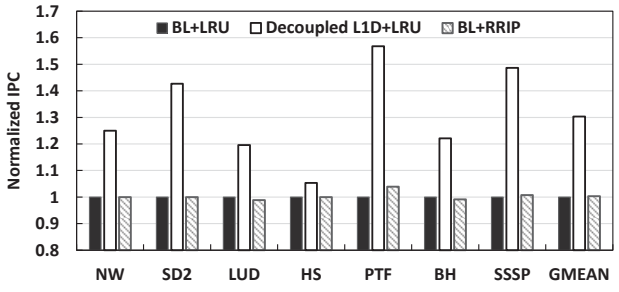


Figure 12: Impact of cache replacement policies on performance of CNF workloads on GPUs.

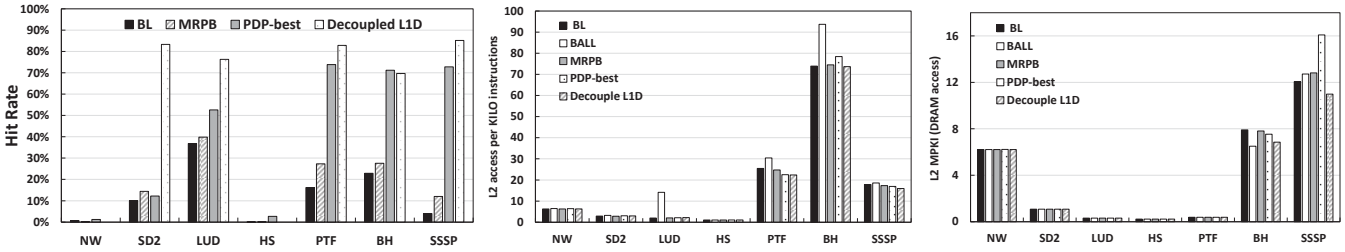
L1 capacity and conflict contention. Since the bypassed requests do not compete for L1 MSHRs and miss queues (Figure 1), the stall cycles due to reservation fails have also been significantly reduced by an average of 84.5% for CNF workloads.

(f) DRAM And L2 Level Traffic After Bypassing: Figure 10b and 10c show the L2 and DRAM traffic after using various bypassing mechanisms. In Figure 10b, our dynamic bypassing mechanism encounters the least amount of traffic to L2 (least pressure to NoC). The same observation can be made for DRAM traffic, shown in Figure 10c. Also, for the two irregular applications BH and SSSP, DRAM traffic from PDP-best is much higher than our design, which can result in resource congestion. This makes applying PDP approaches on GPUs less attractive because it requires warp throttling techniques to accompany PDP bypassing (e.g., the design in [11]) to be effective for irregular workloads. However, warp throttling can significantly reduce parallelism and subsequently reduce overall performance, which our design does not suffer from.

(g) Impact of Cache Replacement Policies On Performance: Advanced replacement policies such as RRIP [14] and SHiP [29] have been proposed to amortize cache contention in CPU LLCs. Although RRIP and SHiP can reduce cache contention, the blocks with long reuse distances are still brought into the cache albeit in the LRU position. This is less a problem in CPU LLCs but is problematic for GPU L1 D-cache. Figure 12 shows the performance comparison of CNF workloads running on Baseline+LRU, Baseline+RRIP, and Decoupled L1D+ LRU. We can easily observe that the replacement policies have little performance impact on CNF workloads. This is because unlike the CPU LLCs, GPU L1 D-cache has a much smaller cache associativity and capacity. Bringing in data with long reuse distances may evict more data with good locality. More importantly, without bypassing, all the L1 D-cache misses compete for limited cache resources (e.g. MSHRs and miss queue), resulting in serious resource congestion. Therefore, an efficient bypassing strategy such as ours is very necessary for GPUs, no matter which cache replacement policy is applied.

6.2 Design Space Exploration

Decoupled L1D tag store's entry size: In the decoupled tag store, there are two types of entries based on their *Position* field (Figure 6 in Section 4). The entries with their *Position* set to *valid* are those that have already been connected with a data-store line position (named committed entries) and the rest are those that are competing for allocating a data line in the data store (named candidate entries). The lower bound number of the tag entries should be larger than the number of cache lines. Otherwise, some data cache lines are wasted. A larger size of tag entries can

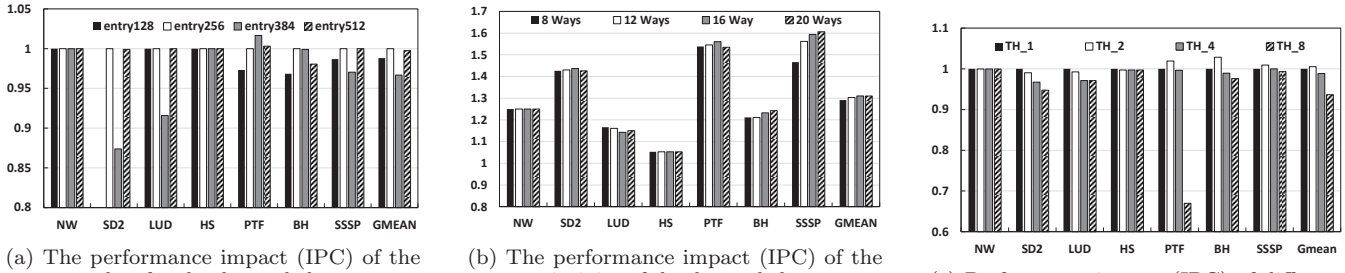


(a) L1 D-cache performance (based on hit rate) for various designs. This is the average hit rate for the overall kernel lifetime.

(b) Off-chip L2 accesses per thousand instructions for various designs, representing L2 level traffic.

(c) L2 misses per thousand instructions (MPKI) for various designs, representing DRAM level traffic.

Figure 10: Cache performance (hit rate) and memory traffic caused by bypassing in various designs.



(a) The performance impact (IPC) of the entry number for the decoupled tag stores. All are normalized to the one with 256 entries. All are fully associative.

(b) The performance impact (IPC) of the set associativity of the decoupled tag store. The performance is normalized to the baseline 4-way associativity cache.

(c) Performance impact (IPC) of different locality thresholds, normalized to threshold=1.

Figure 13: Design space exploration for our proposed decoupled L1 D-cache for GPUs. The design choices match Table 3.

accommodate more candidate entries with a longer reuse distance. However, if the size of tag entries is too large, it will lose the sensitivity on reuse distance and cause cache pollution. For example, at the entry size 384, there are at least 256 entries that can be allocated for candidate entries since there are 128 cache lines (Table 2). If an entry has burst access pattern with reuse distance longer than 128, it can still stay in the tag store and may evict out other entries with short reuse distance by allocating lines in data store, causing cache pollution. Base on the results shown in Figure 13a, 256 entries should be used as the entry size in the tag store.

Decoupled L1D tag store's associativity: Figure 13b presents the effect of varying the tag store's associativity of the decoupled L1D on performance. The expanded tag store will record the reuse frequency of the incoming memory accesses and the size of way per set will affect how long an entry can reside in the tag store. When the number of ways is small, it may thrash the tag entries (that compete for allocating data lines) frequently due to high contention. When the number of ways is large, it can preserve the entries with a longer reuse distance. Our simulation results show that the 16-way design point can achieve only a 2% performance improvement over the 8-way with doubled size of the tag entries. Therefore, we select 8-way because it provides the highest performance per unit area.

Reference Count (RC)'s locality threshold selection: The RC field and locality threshold are discussed in Section 4.2 as part of our new tag store design. Figure 13c shows the effect of varying the values of the locality threshold on performance. The majority of our CNF benchmarks achieve the highest performance at the threshold value of 2 (or near the highest) due to their low data reuse pattern shown in Figure 3. Thus, we set the locality threshold value to 2 in our design. In this work, we did not implement an online system for determining the value of the threshold for individual workloads because cache unfriendly workloads do not exhibit high variation for this design point.

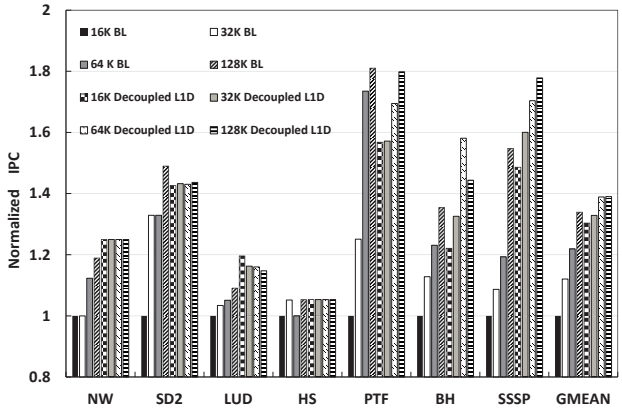


Figure 14: The effectiveness of our design on the baseline L1 D-cache with different capacities. All data is normalized to a 16KB baseline cache.

6.3 Sensitivity to L1 D-cache Sizes

Figure 14 shows the sensitivity of our design on various L1 D-cache sizes. From the performance comparisons between the baseline caches and our approach, we have two observations: (1) For every cache size we studied, our design has shown performance improvements over the baseline, even at a larger cache size (128KB). With the increasing cache size, the performance improvements of our design over the baseline do not increase as rapidly. This is because smaller cache sizes are more sensitive to contention caused by large cache footprint while bigger caches can accommodate larger working sets and accesses with longer reuse distance. Also, for a workload, if the working set fits in the cache already, fur-

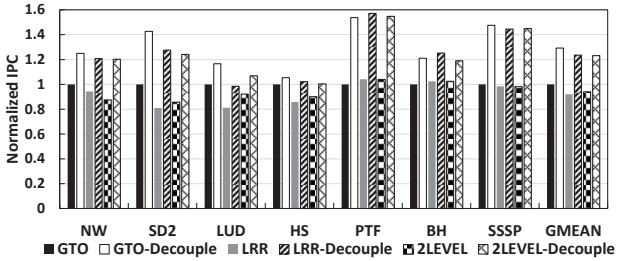


Figure 15: Performance of our design with various warp scheduling policies, normalized to pure GTO performance.

ther increasing the cache size has no performance benefit. (2) Even a 16KB Decoupled L1D can achieve the average performance close to a 128KB baseline L1 D-cache, which indicates that our design can significantly improve GPU performance without incurring higher area overhead.

6.4 Sensitivity to Warp Scheduling Policies

Prior sections use the Greedy-Then-Oldest (GTO) warp scheduling policy for experiments. Figure 15 shows the sensitivity of our design to various warp scheduling policies. Loosely Round Robin Scheduler (LRR) is oblivious to the access locality while GTO outperforms it by preserving the intra-warp data locality. The two-level scheduler (2LEVEL) proposed by Narasiman et al.[22] exploits inter-warp data locality by scheduling groups of warps (called FG) together to ensure different groups touching long-latency memory accesses at different times. However, 2LEVEL still performs worse than GTO. Rogers et al.[24] found that performing inter-FG in round-robin would destroy intra-warp locality of old warps, which however can be captured by GTO. Figure 15 highlights that improving the scheduling policy can increase performance by reducing the inter-warp contention. Our design complements it by further reducing the intra-warp contention (e.g., conflict misses & MSHR reservation fails), which cannot be addressed by any warp scheduler [15]. Therefore, our design can achieve further performance gains with different warp scheduling policies.

6.5 Hardware Cost And Energy Efficiency

We evaluate the hardware cost and energy efficiency of our Decoupled L1D design using CACTI 6.5 [28].

Hardware Cost: The major source of area overhead for supporting the SRAM-based Decoupled L1D comes from the expanded tag store. We didn't change the bandwidth and number of ports in the Decoupled L1D and we assume 1 additional cycle for the extended tag-store probe. However, this additional latency is hidden by the thread-level parallelism, similar to the additional buffer latency described in [15]. Each tag has two additional fields, one for RC (6 bit) and the other for Position field (2 bit). With the expanded tag entries (256), the total area of Decoupled L1D is estimated as 0.244 mm^2 for the entire 15-SM system using 45nm technology, which is 0.008 mm^2 more than the original L1 D-cache. Such additional area represents approximately 0.02% of the GTX480 area [5], which is also a 15-SM system implemented in a 40nm technology. Other miscellaneous overheads such as the 2-bit per set 'Free_Lines' counter in Decoupled L1D are negligible in comparison.

Energy Efficiency: Since our design is a simple tag store extension to the existing L1 D-cache, it does not require a large amount of energy-consuming cache logic circuitry. From the baseline L1 D-cache to Decoupled L1D, the *dynamic read energy per access* and *total leakage power* increases by 0.002 nJ and 1.7 mW respectively. Such small energy overhead can be disregarded compared to the substantial energy improvements from our design by significantly reducing L1 cache misses (by up to 80% for SSSP) and off-chip (L2+DRAM) memory accesses. Specifically, the three CNF workloads with the most energy reduction are PTF (10.4%), BH (14.7%) and SSSP (11.4%).

7. RELATED WORK

CPU Cache Management: Prior studies have looked into optimizing LRU-based cache replacement policies for better performance, such as PiPP[32], RRIP[14] and SHiP[29]. However, due to the small cache capacity and massive parallelism in GPUs, even the most advanced replacement policies cannot address their cache contention problems [24]. Our locality-driven dynamic bypassing design aims to decide if a memory access should even be inserted into the cache, in order to reduce cache contention. Our approach is complementary to these cache replacement optimizations and in fact they are applied to preserve the temporal locality of the cache lines in our design. There also have been some work [19, 12, 13, 18] proposed for CPUs on cache bypassing, which are usually designed for LLCs, where data locality is already filtered by previous level(s). But the poor locality of GPU workloads imposes difficulty for these approaches to make robust predictions for L1 D-cache (where locality is unfiltered). Our approach is specifically designed for throughput-oriented architectures like GPUs and it outperforms the state-of-the-art CPU bypassing approach PDP-best [12] for both CNF and CF workloads, especially for irregular applications. V-Way cache [23] manipulates data store to enable global data replacement. On the contrary, our design focuses on integrating locality filter into the L1 D-cache for bypassing by using an expanded tag store to trace the reuse frequency of accesses, with significantly different structures and operations.

GPU cache management: Previous work [22, 24, 17, 20] has made efforts on improving cache performance by changing the warp scheduling policies. For instance, CCWS [24] dynamically throttles active warps to avoid cross-warp contention by using a victim cache tag array. However, warp schedulers can neither address intra-warp contention nor eliminate cache pollution that is extensively caused by caching "unfriendly" data. Furthermore, warp throttling methods can reduce parallelism for processing memory requests and subsequently limit the resource utilization. MRPB [15] designs a FIFO queue for reordering memory requests in order to reduce intra- and inter- warp contention. It also uses a simple bypassing mechanism, which only bypasses when intra-warp contention occurs upon unavailable resources (e.g. MSHR reservation fails). More importantly, it does not consider any data locality of the access stream, which results in incorrectly bypassing accesses with good reuse. In contrast, our design aims to only store the data with high re-references and short reuse distance to maximally reduce cache pollution and resource contention. As shown in Section 6.1, for both CNF and CF workloads, our design outperforms MRPB significantly, which outperforms CCWS [24] and [22]. Work [11] directly applies a dynamic PDP approach (runtime best-effort prediction using hardware sampling) on GPUs for L1 bypassing and uses warp throttling if contention at L2 and DRAM level are generated by PDP. Our efficient design outperforms PDP-best (exhaustively search for the best PD) for both CF and irregular CNF applications without increasing L2 and DRAM level traffic. Xie et al. [31] analyzed which global-memory

loads should be cached at compile time and chose a subset of thread blocks to use the cache at runtime. In [27], the program counters (PCs) of memory instructions are used to make a cache-bypassing decision. In comparison, we rely on memory address to drive our dynamic cache bypassing. As GPU kernels are relatively short, the working set of a memory access instruction can be relatively large, especially considering the same PC in all the thread blocks. Therefore, the instruction-level bypassing [31, 27] is more coarse-grain and may lose the opportunity of exploiting data reuse at memory address granularity.

Compiler [30, 16] and software level [10, 26] techniques were also proposed to improve GPU cache performance. Compiler-driven mechanisms [30, 16] depend heavily on inputs and profiling runs. Although they can be effective to optimize cache performance for regular applications, they cannot predict the runtime behaviors of irregular applications. Unlike our dynamic hardware approach, software techniques [10, 26] need to directly interact with applications for code optimization or data transformation, and can be detrimental to cross-platform performance portability [15].

8. CONCLUSIONS

In this paper, we analyze the GPU workloads to reveal the data reuse characteristics of different types of applications. For cache unfriendly (CNF) applications, their memory access stream feature accesses with low reuse and/or long reuse distances, causing severe cache contention and resource congestion. To address this challenge, we propose a locality-driven dynamic bypassing solution that integrates locality filtering functionality into the decoupled tag store of the current GPU L1 D-cache through simple and cost-effective hardware extensions. Experiment results show that our design achieves significant performance and energy improvements over the baseline caches and outperforms the state-of-the-art CPU and GPU cache bypassing schemes. It can significantly reduce various levels of contention without generating extra memory traffic and remain effective for various cache capacities and warp scheduling policies.

9. REFERENCES

[1] AMD APP SDK: http://developer.amd.com, 2015.

[2] AMD Graphics Cores Next (GCN) Architecture White paper, 2012.

[3] NVIDIA CUDA SDK: https://developer.nvidia.com/cuda-downloads. 2015.

[4] NVIDIA Kepler GK110 white paper. 2012.

[5] NVIDIA's next generation CUDA compute architecture: Fermi. 2009.

[6] S. S. Baghsorkhi et al. Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors. In *PPoPP '12*. ACM, 2012.

[7] A. Bakhoda et al. Analyzing cuda workloads using a detailed gpu simulator. In *ISPASS'09*, April 2009.

[8] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. In *IISWC'12*, Nov 2012.

[9] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC'09*, Oct 2009.

[10] S. Che et al. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *SC '11*. ACM, 2011.

[11] X. Chen et al. Adaptive cache management for energy-efficient gpu computing. In *MICRO-47*. ACM, 2014.

[12] N. Duong et al. Improving cache management policies using dynamic reuse distances. In *MICRO-45*, 2012.

[13] J. Gaur et al. Bypass and insertion algorithms for exclusive last-level caches. In *ISCA '11*. ACM, 2011.

[14] A. Jaleel et al. High performance cache replacement using re-reference interval prediction (RRIP). In *Proc of ISCA '10*. ACM, 2010.

[15] W. Jia, K. Shaw, and M. Martonosi. MRPB: Memory request prioritization for massively parallel processors. In *HPCA'14*.

[16] W. Jia, K. A. Shaw, and M. Martonosi. Characterizing and improving the use of demand-fetched caches in gpus. In *ICS '12*. ACM, 2012.

[17] A. Jog et al. OWL: Cooperative thread array aware scheduling techniques for improving gpgpu performance. In *ASPLOS '13*. ACM, 2013.

[18] T. L. Johnson and W.-m. W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *ISCA '97*. ACM, 1997.

[19] M. Kharbutli and D. Solihin. Counter-based cache replacement and bypassing algorithms. *Computers, IEEE Transactions on*, April 2008.

[20] S.-Y. Lee and C.-J. Wu. CAWS: Criticality-aware warp scheduling for gpgpu workloads. In *PACT '14*. ACM, 2014.

[21] C. Li et al. Understanding the tradeoffs between software-managed vs. hardware-managed caches in gpus. In *ISPASS'14*, March 2014.

[22] V. Narasiman et al. Improving gpu performance via large warps and two-level warp scheduling. In *MICRO-44*, 2011.

[23] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-Way cache: Demand Based Associativity via Global Replacement. In *ISCA '05*, 2005.

[24] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-conscious wavefront scheduling. In *Proc of IEEE MICRO-45*, 2012.

[25] I. Singh et al. Cache coherence for GPU architectures. In *HPCA '13*. ACM, 2013.

[26] I.-J. Sung, G. Liu, and W.-M. Hwu. DL: A data layout transformation system for heterogeneous computing. In *InPar'12*, May 2012.

[27] Y. Tian et al. Adaptive gpu cache bypassing. In *GPGPU'15 workshop*. ACM, 2015.

[28] S. Wilton and N. Jouppi. CACTI: an enhanced cache access and cycle time model. *Solid-State Circuits, IEEE Journal of*, 31(5), May 1996.

[29] C.-J. Wu et al. SHiP: Signature-based hit predictor for high performance caching. In *MICRO-44*. ACM, 2011.

[30] X. Xie et al. An efficient compiler framework for cache bypassing on gpus. In *ICCAD'13*.

[31] X. Xie et al. Coordinated static and dynamic cache bypassing for gpus. In *Proc of HPCA'15*, pages 76–88. IEEE, Feb 2015.

[32] Y. Xie and G. H. Loh. PiPP: Promotion/Insertion pseudo-partitioning of multi-core shared caches. In *Proc of ISCA'09*. ACM, 2009.

[33] Y. Yang et al. Shared memory multiplexing: A novel way to improve gpgpu throughput. In *PACT '12*. ACM, 2012.