Eliminating Intra-Warp Conflict Misses in GPU

Bin Wang Zhuo Liu Xinning Wang Weikuan Yu

Dept. of Computer Science, Auburn University
{*bwang,zzl0014,xzw0033,wkyu*}*@auburn.edu*

***Abstract*—Cache indexing functions play a key role in reducing conflict misses by spreading accesses evenly among all sets of cache blocks. Although various methods have been proposed, no significant effort has been expended on the behavior of conflict misses in GPU where threads are organized into warps and execute in lock-step. When intra-warp accesses could not be coalesced into one or two cache blocks, which is often referred to as memory divergence, a warp incurs up to SIMD-width (e.g., 32) independent cache accesses. Such a burst of divergent accesses not only increases contention on cache capacity, but also incurs intra-warp associativity conflicts when they are pathologically concentrated in a few cache sets. Due to the lock-step execution, the GPU Load/Store units would be stalled when intra-warp concentration exceeds available cache associativity. Through an in-depth analysis of GPU access patterns, we find that column-majored strided accesses are likely to incur high intra-warp concentration. Based on the analysis, we propose a *Full Permutation (FUP)* based indexing method that adapts to both large and medium strides in this pattern. Across the 10 highly cache-sensitive GPU applications we have evaluated, FUP eliminates intra-warp associativity conflicts and outperforms two state-of-the-art indexing methods by 22% and 15%, respectively.**

I. Introduction

Graphics Processing Units (GPUs) have proven as a viable technology for a wide variety of applications to exploit the massive computing capability. In a GPU, threads are organized into warps and execute in lock-step. This execution model generally faces two challenges, i.e., control divergence and memory divergence. Control divergence occurs when threads in a warp take different code path for execution, while memory divergence refers to the case where intra-warp accesses cannot be coalesced into one or two cache blocks.

Recently, GPUs have employed a hierarchy of caches, which can reduce the latency of memory operations and save the on-chip network and off-chip memory bandwidth when there is locality within the accesses. However, the contention from massive parallelism often makes the caching performance unpredictable. Notably, the bursty divergent accesses can cause associativity stalls when they are pathologically concentrated into a few cache sets. To tackle this problem, MRPB [14] aggressively bypasses L1D whenever associativity stall occurs, but the cache is still underutilized. Two recent works [24], [25] have reported that throttling the number of warps that can be actively scheduled is able to reduce the accumulated working set so that the contention on cache capacity is alleviated and locality is preserved. However, they are mainly designed to alleviate capacity misses, having little control over intra-warp associativity conflicts. Without spreading bursty intra-warp accesses evenly into all cache sets, associativity conflicts inevitably undercut the potential performance benefits that other optimizations can bring. For example, the concurrency throttling in [24], [25] becomes futile.

Pseudo-random cache indexing methods have been extensively studied to reduce conflict misses within CPU systems. However, no prior indexing method has exploited the pathological behaviors of GPU cache indexing. In CPU systems, parallelism is supported at a moderate level, in which memory accesses are often dispersed over time. However for GPUs, high thread counts are common, intra-warp accesses often come in long bursts and memory bandwidth plays a critical role in sustaining high computation throughput. These distinctive features pose challenges in designing a GPU-specific indexing method.

Based on these observations, we study how to design a GPU data cache indexing method to eliminate intra-warp associativity conflicts. Our contributions from this study include:

- We present the problem of intra-warp conflict misses in GPU from the aspect of pathological behaviors of current cache indexing method.
- We propose a new metric, *intra-warp concentration*, to evaluate GPU cache indexing methods. This metric quantifies the dynamic concentration of divergent intra-warp accesses into cache sets and is more correlated with intra-warp conflicts.
- We design a *Full-Permutation (FUP)* based GPU cache indexing method, which achieves perfect intra-warp concentration for strided access patterns among GPU benchmarks and significantly reduces the conflict misses due to intra-warp contention.

Our experiment results show that FUP improves the performance of 10 highly cache-sensitive GPU benchmarks by $2.46\times$ (Geometric Mean), and outperforms two state-of-the-art cache indexing methods by 22% and 15%, respectively. Besides, the metric of intra-warp concentration is also proved to be more closely correlated with the quality of GPU cache indexing methods than other GPU-oblivious static metrics.

II. Background and Motivation

A. Baseline GPU Architecture

Fig. 1 depicts the GPU architecture we study in this work. In this Fermi-like GPU, each Streaming Multiprocessor (SM) contains multiple warp slots that are managed by a hardware

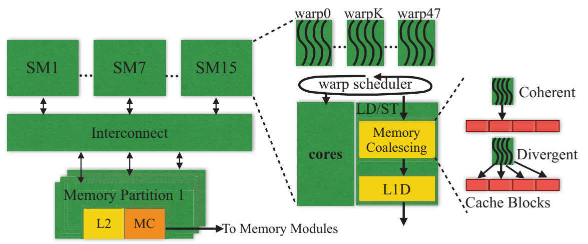
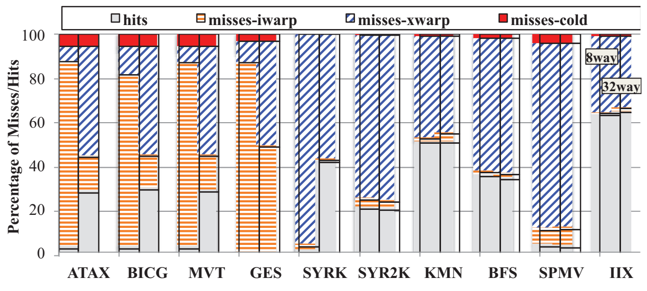
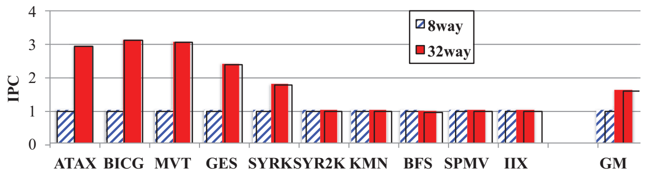


Fig. 1: Baseline GPU Architecture.



(a) Cache miss/hit rates



(b) IPC

Fig. 2: The impacts of associativity on highly cache-sensitive benchmarks. Both caches have 32KB capacity and 128B lines.

scheduler. In each cycle, the warp scheduler issues the warp with highest priority among all ready ones to execute in cores or load/store units (LD/ST) [17], [19], [20], depending on the type of pending instructions. The warp scheduler is capable of switching ready warps at zero overhead so that GPU pipeline could remain busy. Once a memory instruction to global memory is issued, it has to go through the Memory Coalescing Unit (MCU) to check whether the intra-warp accesses fall into one or two 128B cache blocks; if not, multiple accesses are generated. The resultant memory accesses are sequentially sent to a single 128B port to the L1 cache. Without coherence support for global data, L1D writes through dirty data and evicts cache blocks on write hits. The GPGPU-Sim 3.x Manual [1] describes other components in more details.

B. Types of Cache Misses in GPU

Because GPUs normally have very small L1D, any potential locality could be easily thrashed by the massive parallelism, especially when memory divergence boosts the per-warp cache footprint. However, memory divergence incurs not only inter-warp capacity misses, but also high intra-warp conflict misses when the divergent intra-warp accesses are pathologically mapped in the same cache set. Because of the discrepancy between cache associativity and the number of divergent accesses per warp, a warp can have intra-warp cache conflicts [14], resulting in stalled execution. Such intra-warp conflicts can cause execution stalls to more warps, destroying the capability of overlapping memory and computation for high throughput.

We use 10 highly cache sensitive benchmarks to illustrate the problem of intra-warp associativity conflicts and motivate this work. The details of the benchmarks are presented in Section IV. We categorize conflict misses into intra- and inter-warp misses. An intra-warp miss refers to the case where a thread's data is evicted by other threads within the same warp; otherwise a conflict miss is referred to as inter-warp miss. As shown in Fig. 2a, after increasing the associativity from 8 to 32, the intra-warp misses (*misses-iwarp*) in ATAX, BICG, MVT, and GES are significantly reduced, even though 49% of misses in GES are still intra-warp misses. Meanwhile, larger associativity reduces the inter-warp misses (*misses-xwarp*) in SYRK. Fig. 2b presents the IPC improvement of a larger associativity. For ATAX, BICG, MVT, GES, and SYRK, a 32-way 32KB L1D improves performance by 2.6×.

C. Associativity-Sensitive Access Patterns

With multidimensional data arrays, **column-major strided** access pattern creates high intra-warp contention on associativity. The most common example of this pattern is A[tid*STRIDE+offset], where *tid* is the unique thread ID and *STRIDE* is user-defined stride size. By using this pattern, each thread iterates a stride of data independently. In a conventional cache indexing function, the target set is computed as $set = (addr/blkSz) \ mod \ (n_{set} - 1)$, where $addr$ is the memory address, $blkSz$ is the length of cache block and n_{set} is the number of cache sets. When the address stride between two consecutive threads is equal to a multiple of $blkSz \times n_{set}$, all blocks needed by a single warp are mapped into the same cache set. Since cache associativity is often smaller than warp size (32), conflict misses occur within each single divergent load and then the memory pipeline is congested by the serialized per-warp accesses [14]. We investigate the cache indexing method to dispense the bursty intra-warp access into all cache sets so that intra-warp contention is reduced and cache capacity can be better utilized.

III. Full-Permutation Based GPU Cache Indexing

In this section, we first elaborate a new metric for quantifying the distribution of intra-warp accesses and then propose our full-permutation indexing method.

A. A Metric for GPU Cache Indexing Method

We use **Intra-warp Concentration** to quantify the distribution uniformity of intra-warp accesses into the cache sets. It is measured by:

$$Intra\text{-}warp\ concentration = \frac{N_{acc}}{N_{cache\_sets\_touched}} \tag{1}$$

where N_{acc} is the number of accesses in a load instruction and $N_{cache\_sets\_touched}$ is the number of sets that are caching the data of the load instruction. A value of 1 indicates an ideal distribution of intra-warp accesses, i.e., intra-warp contention on cache associativity is eliminated. Any value larger than 1 indicates the existence of intra-warp concentration. Note that coherent loads often lead to an ideal intra-warp concentration.

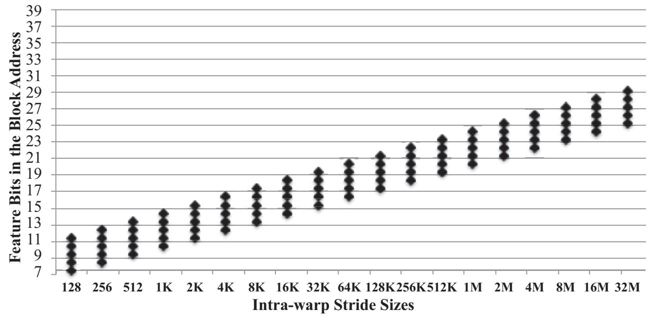


Fig. 3: The feature bits in the block addresses when various strides are used in the strided access of *tid*STRIDE*. Here $tid \in [0, 31]$ and the bits of block offsets ($0 \sim 6^{th}$) are omitted.

Different from the static metrics used in [15] to quantify the pathological behaviors of cache indexing methods, intra-warp concentration describes dynamic contention among bursty intra-warp accesses. Because avoiding long memory accesses caused by intra-warp contention is very critical, it is beneficial to introduce the intra-warp concentration metric for measuring the effectiveness of conflict management by GPU cache indexing methods.

B. Feature Bits of Intra- and Inter-Warp Addresses

In order to disperse intra-warp accesses into all cache sets, it is necessary to understand how one address is different from the others. In the column-major strided access pattern, the majority of the bits in intra-warp addresses are the same, i.e., having no variability, while a small amount of bits are altered. We name those altered positions as *Feature Bits*. The length of feature bits depends on SIMD-width. For example, when SIMD-width is 32, a warp would generate up to 32 accesses; consequently, $log_2(SIMD\text{-}width)$ bits in the block addresses become feature bits to distinguish intra-warp accesses.

Fig. 3 shows the positions of feature bits in the intra-warp block addresses when the stride sizes range from 128 to 32M for $Warp_0$ (threads in this warp have global indexes between 0 and 31). Assuming a 40-bit virtual address space, the feature bits of $Warp_0$'s accesses spread in the range from 7^{th} to 29^{th} bit. Because a GPU kernel is often launched with a numerous number of threads, this upper bound would be theoretically extended close to the most significant bit of the block address. Given that the global memory size is no larger than 16GB among contemporary GPGPU cards, the range from 7^{th} to 34^{th} bit covers any legitimate stride size. Thus we use bits in this range ($num\_feature\_bits$) to search feature bits for GPU cache indexing. We will show how feature bits can be used in a GPU cache indexing method to spread intra-warp accesses evenly into cache sets. Compared to the designated feature bits, if the invariable bits are used to form the set index, identical bits exist in the resultant set indexes of intra-warp accesses, causing intra-warp concentration.

C. Full-Permutation for GPU Cache Indexing

In order to cover all feature bits in memory addresses, we propose the *Full-Permutation (FUP)* based cache indexing

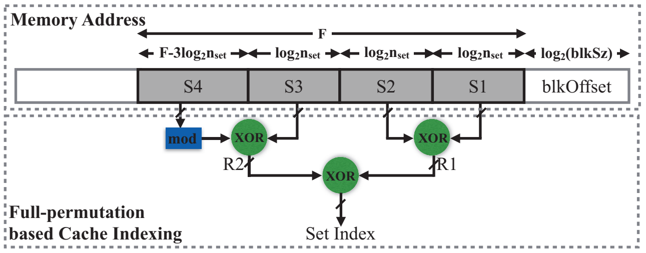


Fig. 4: Illustration of full-permutation cache indexing.

method. In general, FUP uses F bits in the middle of block addresses, where $F = max(num\_feature\_bits, 4 \times log_2 n_{set})$. As shown in Fig. 4, the F bits are divided into four groups, i.e., *S1, S2, S3,* and *S4*. We now discuss how L1D impacts the implementation of FUP:

1) When $num\_feature\_bits$ is equal to $4 \times log_2 n_{set}$, each group has $log_2 n_{set}$. Thus the four groups of bits are paired, and then XORed in parallel at the first level, and then the two intermediate indexes, *R1* and *R2*, are further XORed to generate the final index.
2) When $num\_feature\_bits$ is smaller than $4 \times log_2 n_{set}$, FUP could still be implemented as case 1. Oversubscribing the bits of block addresses for cache indexing simplifies the hardware implementation and can disperse inter-warp accesses into all sets.
3) When $num\_feature\_bits$ is larger than $4 \times log_2 n_{set}$, *S4* has more bits than the other three groups. We use a prime number based modulo operation (*mod*) to convert *S4* into $log_2 n_{set}$ bits so that the remaining logic could be unchanged. This modulo operation could be implemented using a set of narrow add operations [15] and causes limited intra-warp concentration for very large strides.

In total, this scheme requires $3 \times log_2 n_{set}$ two-input XOR gates for the logic implementation. The delay of two-level XOR gates is less than 1 cycle even for a very aggressively pipelined processor. Even when the *mod* operation is needed, the delay of FUP could still be implemented in no more than 2 cycles. Because GPU is not highly sensitive to L1D cache latency, this delay is easily compensated by reduced intra-warp concentration.

FUP naturally adapts to streaming-like coherent loads where inter-warp feature bits tend to be concentrated in the lower bits of block addresses.

D. Comparison of Various Indexing Methods

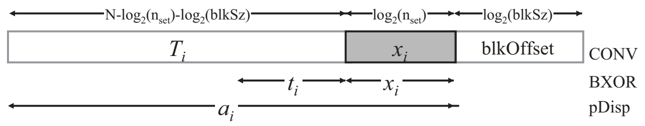


Fig. 5: Decomposition of a N-bit memory address in various indexing methods. x_i and t_i represent partial index bits of block address a_i.

We elaborate the comparison among several indexing methods. Given a N-bit memory address, Fig. 5 illustrates the

TABLE I: Baseline GPGPU-Sim Configuration

# of SMs	30 (15 clusters of 2)
SM Configuration	1400Mhz, Reg #: 32K, Shared Memory: 48KB, SIMD Width: 16, warp: 32 threads, max threads per SM: 1024
Caches / SM	Data: 32KB/128B-line/8-way, Constant: 8KB/64B-line/24-way, Texture: 12KB/128B-line/2-way
Branching Handling	PDOM based method [8]
Warp Scheduling	GTO
Interconnect	Butterfly, 1400Mhz, 32B channel width
L2 Unified Cache	768KB, 128B line, 16-way
Min. L2 Latency	120 cycles (compute core clock)
# Memory Partitions	6
# Memory Banks	16 per memory partition
Memory Controller	Out-of-Order (FR-FCFS), max request queue length: 32
GDDR5 Timing	$t_{CL} = 12$, $t_{RP} = 12$, $t_{RC} = 40$, $t_{RAS} = 28$, $t_{RCD} = 12, t_{RRD} = 6, t_{CDLR} = 5, t_{WR} = 12$

TABLE II: Highly Cache-Sensitive CUDA Benchmarks

#	Abbr.	Application	Sensitivity	Input
1	ATAX [11]	matrix-transpose and vector multip.	Associativity	$8K \times 8K$
2	BICG [11]	kernel of BiCGStab linear solver	Associativity	$8K \times 8K$
3	MVT [11]	Matrix-vector-product transpose	Associativity	8K
4	GES [11]	Scalar-vector-matrix multiplication	Associativity	4K
5	SYRK [11]	Symmetric rank-K operations	Associativity	512×512
6	SYR2K [11]	Symmetric rank-2K operations	Capacity	256×256
7	KMN [5]	Kmeans Clustering	Capacity	28k 4x features
8	BFS [5]	Breadth-First-Search	Capacity	5M edges
9	SPMV [6]	Sparse matrix multiplication	Capacity	default
10	IIX [13]	Inverted Index	Capacity	6.8M

decomposition of address bits in cache indexing methods that we will compare with. In the conventional method (*CONV*), the lowest $log_2(n_{set})$ bits of block address, i.e., (x_i) bits, are selected as the set index.

Pseudo-random indexing methods are often used to randomize accesses to cache sets. Among them, XOR-based indexing methods are by far the most extensively studied. We choose the bitwise-XOR (*BXOR*) [10] as a representative of pseudo-random indexing methods. BXOR extends the bits for set index calculation via $x_i \oplus t_i$, where t_i also has $log_2(n_{set})$ bits. As we can see from Fig. 5, both *CONV* and *BXOR* are incapable of covering all feature bits of the intra-warp addresses. The $2 \times log_2(n_{set})$ bits used by *BXOR* only cover small strides. For large strides, *BXOR* includes invariable bits for index calculation, leading to high intra-warp concentration. Because of the large range of feature bits, simply altering the bits for *BXOR*, such as the scheme reported in [18], is still unlikely to make *BXOR* adapt to all kinds of strides.

Kharbutli et al. [15] propose two prime numbers based indexing methods, prime modulo (*pMOD*) and prime displacement (*pDisp*). pMOD uses a prime number of sets in the cache. pDisp calculates the set index as follows: $index = (p \times T_i + x_i)$ $\textbf{mod}$ n'_{set}, where tag T_i has $N - log_2(n_{set}) - log_2(blkSz)$, p is a prime number and n'_{set} is the largest prime number that is smaller than n_{set}. The introduction of prime numbers based modulo operation disturbs the high regularity in column-major strided access pattern, dispersing the bursty intra-warp accesses into the majority of available cache sets ($n'_{set} < n_{set}$). However, this causes under-utilization of cache capacity, and causes intra-warp concentration at the degree of n_{set}/n'_{set}.

IV. Experimental Evaluation

A. Experimental Methodology

We use GPGPU-Sim [3] (version 3.2.1), a cycle-accurate simulator, for the performance evaluation of FUP cache indexing method, as discussed in section III. The main characteristics of our baseline GPU architecture are summarized in Table I. This baseline is also studied in [24], [25], [30]. The highly cache-sensitive benchmarks we study are from Rodinia [5], SHOC [6], PolyBench/GPU [11], and MapReduce [13]. Table II lists a brief description of each benchmark and the input sizes that we use for performance evaluation. All of the benchmarks are run to completion which takes between 70 million and 1.5 billion instructions.

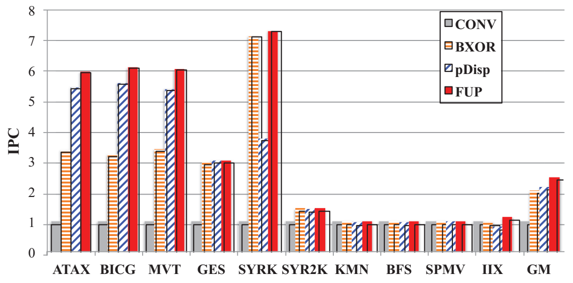


Fig. 6: The impacts of cache indexing methods on IPC.

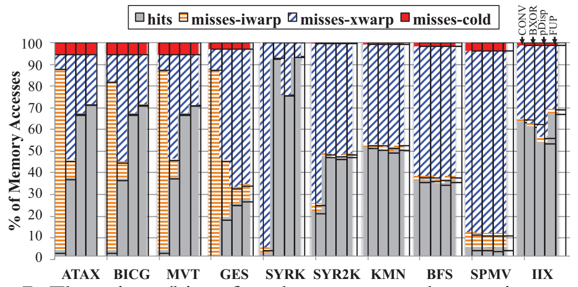


Fig. 7: The misses/hits of cache accesses when various cache indexing methods are applied.

B. Performance of FUP Cache Indexing

In this section, we present and discuss the performance impacts of cache indexing methods on highly cache-sensitive GPGPU applications. Among all of the evaluations, *CONV* is taken as the baseline for comparison.

1) IPC: Fig. 6 shows the IPC results. On average, *BXOR* [10], *pDisp* [15] and FUP outperform *CONV* by 2.01, 2.14, and 2.46, respectively. For associativity-sensitive benchmarks, i.e., ATAX, BICG, MVT, GES, SYRK and SYR2K, the three methods achieve a performance improvement of 3.21, 3.70 and 4.36, respectively. Across all the benchmarks, *BXOR* and *FUP* do not hurt performance, but *pDisp* downgrades the performance of IIX by 18.4%. This is mainly because *pDisp* underutilizes cache capacity and constantly disturbs the uniform set accesses that are achievable in *CONV*.

2) Cache Hits/Misses: Fig. 7 shows the results of cache hits/misses. In general, the IPC performance directly comes from the reduced cache conflict misses. In the baseline, *CONV* constantly incurs high intra-warp misses in ATAX, BICG, MVT, and GES. By converting associativity contention into capacity contention, *BXOR* successfully reduces intra-warp misses so that cache hit rate increases. Meanwhile, *pDisp* and *FUP* eliminate the intra-warp misses in these benchmarks

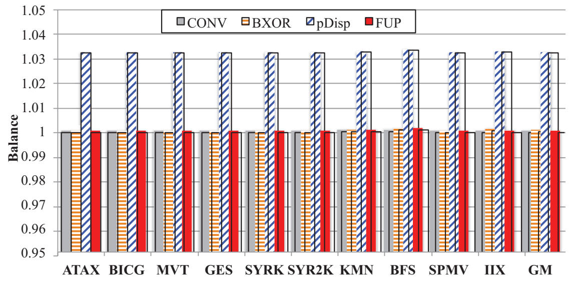


Fig. 8: The balance of cache access distribution in different cache indexing methods.

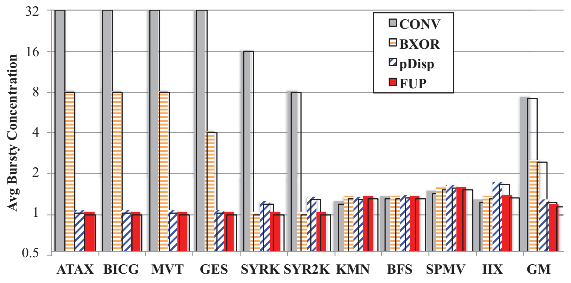


Fig. 9: Average intra-warp concentration in different cache indexing methods. The y-axis is in logarithmic scale.

and cache hit rates further increase, demonstrating their superior performance over *BXOR*. *FUP* outperforms *pDisp* in preserving cache locality because of the full utilization of all available cache sets. In SYRK and SYR2K, *CONV* maps inter-warp accesses into the same cache sets, while *BXOR, pDisp,* and *FUP* reduce the inter-warp concentration by widely spreading these inter-warp accesses. For other benchmarks, cache indexing methods introduce small disturbance into cache hit/miss rates that are reflected in their IPC variance.

3) Balance: **Balance** quantifies the accumulated uniformity of distributing the addresses over all the sets in the cache and can be measured using following equation [15]:

$$balance = \frac{\sum_{j=1}^{n_{set}} \frac{b_j \times (b_j+1)}{2}}{\frac{m}{2 \times n_{set}} \times (m + 2 \times n_{set} - 1)} \quad (2)$$

where b_j is the total accesses to j^{th} set and m is the total cache accesses. The per-set accesses are weighted by $\frac{b_j \times (b_j+1)}{2}$, and the denominator of the equation gives the sum of the weights of all sets in a perfectly random address distribution. A lower balance value indicates a better address distribution over all sets, and a value of 1 indicates an ideal distribution. As shown in Fig. 8, all methods except *pDisp* eventually accumulate an even distribution across all benchmarks. The 3.2% average change of balance in *pDisp* comes from the fact that it uses only 31 out of 32 sets in the baseline L1D. More importantly, this metric is disconnected with the actual performance improvement as shown in Fig. 6, because it quantifies the final distribution of cache accesses and thus is incapable of describing the dynamic intra-warp contention on cache associativity when intra-warp accesses are pathologically concentrated on a few sets.

4) Average Intra-warp Concentration: Fig. 9 presents average bursty concentration. For the associativity-sensitive benchmarks, *CONV* constantly maps the intra-warp accesses of ATAX, BICG, MVT and GES into a single set (an average bursty concentration of 32) and that of SYRK and SYR2K into 2 and 4 sets, respectively. Such degree of intra-warp concentration causes high intra-warp misses in ATAX, BICG, MVT and GES, and inter-warp misses in SYRK and SYR2K. By taking more bits of the memory address into consideration, *BXOR* achieves ideal bursty concentration in SYRK and SYR2K that have medium strides, but still causes high bursty concentration in ATAX, BICG, MVT and GES that have large strides. By keeping the intra-warp concentration within cache associativity, *BXOR* converts the majority of intra-warp misses in *CONV* into inter-warp misses, which is the key to improve performance. Meanwhile, *pDisp* incurs an average bursty concentration of 1.09 for the associativity-sensitive benchmarks, slightly drifting away from its ideal intra-warp concentration of 1.03 (=32/31). Besides, by spreading intra-warp accesses into most of the addressable sets, *pDisp* not only avoids intra-warp contention, but also provides better potential in L1D locality preservation. This advantage explains 15.3% IPC improvement of *pDisp* over *BXOR* in associativity-sensitive benchmarks, as shown in Fig. 6. Besides IIX, *pDisp* also incurs higher intra-warp concentration in SYRK and SYR2K that have small strides. For the other benchmarks, *BXOR* and *pDisp* have similar results. By taking all feature bits into cache index calculation, *FUP* constantly adapts to both large and small strides and therefore achieves a perfect intra-warp concentration in all associativity-sensitive benchmarks. For all the other benchmarks, *FUP* incurs larger intra-warp concentration than *CONV*, which is absorbed by the massive parallelism so that IPC is barely impacted. Besides, the variance of intra-warp concentrations in *BXOR, pDisp,* and *FUP*, closely matches their IPC improvement, which suggests the effectiveness of intra-warp concentration as a metric to evaluate GPU cache indexing methods.

V. Related Work

Since we have compared bitwise-XOR and prime displacement methods in Section III-D in detail, we briefly review other related work in this section. Pseudo-random cache indexing methods have been extensively studied to reduce conflict misses. Topham et al. [29] used XOR to build a conflict-avoiding cache; Seznec [27], [4], [28] combined XOR indexing and circular shift in a skewed associative cache to form a perfect shuffle across all cache banks. XOR is also widely used for memory indexing [22], [16], [23], [12], [7], [31]; however, no work has exploited the pathological behaviors of cache indexing methods in GPUs.

Another common approach to reduce conflict misses is to use a secondary indexing method for alternative cache sets when conflicts happen. This category of work includes skewed-associative cache [27], column-associative cache [2], and v-way cache [21]. We believe *FUP* is orthogonal to these cache architectures and could be combined with them into GPUs to further reduce conflict misses.

Some works have also noticed that certain bits in the address are more critical to reduce cache miss rate. Givargis [9] used offline profiling to detect feature bits for embedded systems. This scheme is only applicable for embedded systems where workloads are often known prior to execution. Ros et al. [26] proposed ASCIB, a three-phase algorithm, to track the changes in address bits at runtime and dynamically discard the invariable bits for cache indexing. ASCIB needs to flush certain cache sets whenever the cache indexing method changes, thus it is best suitable to direct-mapped cache. ASCIB also needs extra storage to track the changes in the address bits. *FUP* proactively covers all the feature bits in the intra- and inter-warp addresses to realize perfect intra-warp concentration in strided access patterns and incurs no storage overhead.

Regarding the problem of intra-warp conflict misses in GPU architecture, MRPB [14] is the most related work. Instead of increasing the utilization of L1D, MRPB attempts to reorder/prioritize per-warp accesses and aggressively bypass L1D when intra-warp conflicts stall the LD/ST unit. *FUP* solves the problem by spreading intra-warp accesses over all cache sets. Without any storage overhead or complicated logic for request reordering and prioritization, *FUP* opens another avenue to solve the problem of intra-warp conflicts in GPUs.

VI. Conclusion

The inclusion of on-chip data caches into GPU was intended to reduce memory operation latency and save bandwidth. But the high thread counts often destroy the locality in L1D and cause high intra-warp associativity conflicts upon memory divergence. One existing work uses aggressive L1D bypassing to alleviate associativity conflicts, but still leaves L1D underutilized. Without spreading intra-warp accesses into all available sets, associativity conflicts serialize the LD/ST unit and undercut the potential of applying other optimization. Thus, it is desirable to investigate how GPU cache indexing method should be tailored to disperse bursty intra-warp accesses and eliminate conflicts among them.

In this work, we first defined a metric called intra-warp concentration and a type of characteristics called feature bits to guide the design of GPU cache indexing method. In addition to the metric and feature bits, we proposed a *Full-Permutation (FUP)* based GPU cache indexing method that uses all feature bits to calculate the set index via two-level XOR gates. By adopting FUP, 10 highly cache-sensitive benchmarks experience an average $2.46\times$ performance improvement, outperforming the two state-of-the-art methods, *BXOR* and *pDisp*, by 22% and 15%, respectively. Meanwhile, intra-warp concentration is also proven to be an effective metric to quantify the dynamic uniformity of intra-warp accesses over all cache sets. With the help of FUP, GPUs can adapt to any stride size that does not overflow device memory capacity.

Acknowledgments

This work is funded in part by an Alabama Innovation Award, and by National Science Foundation awards 1059376, 1320016, 1340947, and 1432892.

References

[1] T. M. Aamodt and W. W. Fung. GPGPUSim 3.x Manual. http://gpgpu-sim.org/manual/index.php/GPGPU-Sim_3.x_Manual, 2014.

[2] A. Agarwal and S. Pudar. Column-associative Caches: a Technique for Reducing the Miss Rate for Direct-Mapped Caches. In *ISCA*, 1993.

[3] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, 2009.

[4] F. Bodin and A. Seznec. Skewed associativity improves program performance and enhances predictability. *IEEE Trans. Computers*, 46(5):530–544, 1997.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[6] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable Heterogeneous Computing (SHOC) benchmark suite. In *GPGPU*, 2010.

[7] Frailong, Jalby, and Lenfant. XOR-schemes: A flexible data organization in parallel memories. In *ICPP*, 1985.

[8] W. W. L. Fung, I. Sham, G. L. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *MICRO*, 2007.

[9] T. Givargis. Improved indexing for cache miss reduction in embedded systems. In *DAC*, 2003.

[10] A. González, M. Valero, N. Topham, and J. M. Parcerisa. Eliminating Cache Conflict Misses Through XOR-based Placement Functions. In *ICS*, 1997.

[11] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a High-Level Language Targeted to GPU Codes. In *Innovative Parallel Computing*, 2012.

[12] D. Harper and J. Jump. Vector access performance in parallel memories using A skewed storage scheme. *IEEE Transactions on Computers (TOC)*, C-36(12):1440–1449, Dec. 1987.

[13] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A MapReduce Framework on Graphics Processors. In *PACT*, 2008.

[14] W. Jia, K. A. Shaw, and M. Martonosi. MRPB: Memory Request Prioritization for Massively Parallel Processors. In *HPCA*, 2014.

[15] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using Prime Numbers for Cache Indexing to Eliminate Conflict Misses. In *HPCA*, 2004.

[16] D. H. Lawrie and C. R. Vora. The prime memory system for array access. *IEEE Trans. Computers*, 31(5):435–442, 1982.

[17] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, Mar. 2008.

[18] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal. A detailed GPU cache model based on reuse distance theory. In *HPCA*, 2014.

[19] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2009.

[20] NVIDIA. NVIDIAs Next Generation CUDA Compute Architecture: Kepler GK110, 2012.

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

[22] R. Raghavan and J. P. Hayes. On randomly interleaved memories. In *SC*, 1990.

[23] B. R. Rau. Pseudo-Randomly Interleaved Memory. In *ISCA*, 1991.

[24] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-Conscious Wavefront Scheduling. In *MICRO*, 2012.

[25] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-aware Warp Scheduling. In *MICRO*, 2013.

[26] A. Ros, P. Xekalakis, M. Cintra, M. E. Acacio, and J. M. García. ASCIB: Adaptive Selection of Cache Indexing Bits for Removing Conflict Misses. In *ISLPED*, 2012.

[27] A. Seznec. A Case for Two-Way Skewed-Associative Caches. In *ISCA*. ACM SIGARCH and IEEE Computer Society TCCA, 1993.

[28] A. Seznec. A new case for skewed associativity. Technical report, IRISA Technical Report 1114, 1997.

[29] N. P. Topham, A. González, and J. González. The Design and Performance of a Conflict-Avoiding Cache. In *MICRO*, 1997.

[30] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. S. Vetter. Exploring Hybrid Memory for GPU Energy Efficiency Through Software-hardware Co-design. In *PACT*, 2013.

[31] Z. Zhang, Z. Zhu, and X. Z. 0001. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *MICRO*, 2000.