The Demand for a Sound Baseline in GPU Memory Architecture Research

Hongwen Dai, Chao Li, Zhen Lin, Huiyang Zhou
North Carolina State University
Raleigh, NC
{hdai3, cli17, zlin4, hzhou}@ncsu.edu

***Abstract*-Modern GPUs adopt massive multithreading and multi-level cache hierarchies to hide long operation latencies, especially off-chip memory access latencies. However, poor cache indexing and cache line allocation policy as well as a small number of miss-status handling registers (MSHRs) can exacerbate the problem of cache thrashing and cache-miss-related resource congestion. Besides, modulo address mapping among memory partitions may cause severe partition camping, resulting in underutilization of DRAM bandwidth and capacity of banked L2 cache. Furthermore, prior GPU cache bypassing studies unrealistically assume there is no limit on the number of in-flight bypassed requests, which may lead to pathological experimental results in simulation.**

In this work, we investigate the performance impact of the aforementioned factors and demonstrate the necessity for a sound baseline in GPU memory architecture research. Our results show that advanced cache indexing functions can greatly reduce conflict misses and improve cache efficiency; the allocation-on-fill policy brings a better performance than allocation-on-miss. Besides, the performance does not consistently improve with more MSHRs. Instead, there can be performance degradation in certain scenarios. In addition, Xor mapping can greatly mitigate the problem of memory partition camping. Furthermore, the fact that a limited number of in-flight bypassed requests can be supported should be taken into account in GPU cache bypassing studies, for more reliable results and conclusions.

I. INTRODUCTION

General purpose computation on graphics processing units (GPGPU) has become prevalent in high performance computing. Modern GPUs consist of multiple Streaming Multiprocessors (SMs), each of which containing 32 to 192 CUDA cores and 2 to 4 warp schedulers [20][21][22][24]. A GPU kernel is launched with a grid of thread blocks (TBs). Threads within a TB form multiple warps and all threads in a warp execute in a lock step manner.

Besides massive multithreading, GPUs have adopted multi-level cache hierarchies to mitigate long off-chip memory access latencies. However, cache thrashing is severe on GPUs due to the small cache capacity per thread and the short cache-line lifetime. Moreover, since miss status handling registers (MSHRs) and miss queue entries need to be allocated for outstanding misses, massive multithreading also causes significant memory pipeline stalls when such resources are fully occupied. Simply enlarging cache capacity and/or adding more cache-miss-related resources is costly in terms of area and power. Therefore, there have been significant research works on GPU memory architecture. In this work, we highlight several often-overlooked aspects of GPU cache design as well as request distribution among memory partitions and demonstrate the necessity for a sound baseline in GPU memory architecture research.

First, although cache indexing methods have been well studied to reduce conflict misses in CPUs [6][11][16], no prior works have thoroughly studied the performance impact of various advanced cache indexing functions on GPUs.

Second, for a request sent to the L1 D-cache, if it is a hit, the required data is returned immediately; if it is a miss, cache-miss-related resources are allocated and the request is forwarded to the L2 cache. Allocate-on-miss and allocate-on-fill are two cache line allocation policies. With allocate-on-miss, a cache line slot, a MSHR, and a miss queue entry need to be allocated for an outstanding miss. In contrast, with allocate-on-fill, a MSHR and a miss queue entry need to be allocated when an outstanding miss occurs but the victim cache line slot is chosen when the required data has returned from lower memory levels. In both policies, if any of the required resources is not available, a reservation failure occurs and the memory pipeline is stalled. The allocated MSHR is reserved until the data is fetched from the L2 cache/off-chip memory while the miss queue entry is released once the miss request is forwarded to the L2 cache. Since allocate-on-fill preserves the victim cache line longer in the cache before eviction and reserves fewer resources for an outstanding miss, it tends to enjoy more cache hits and fewer reservation failures, and in turn better performance than allocate-on-miss. Although allocate-on-fill requires extra buffering and flow-control logic to fill data to the cache in-order, the in-order execution model and the write-evict policy make the GPU L1 D-cache friendly to allocate-on-fill as there is no dirty data to write to L2 when a victim cache is to be evicted at the fill time. Therefore, it is intriguing to investigate how well allocate-on-fill performs for GPGPU applications and whether it is cost-effective.

Third, since the allocated MSHRs are reserved until the required data come back from lower memory levels, it is intuitive to boost performance by deploying more MSHRs to reduce reservation failures and thus memory pipeline stalls, and in the meanwhile increase memory-level-parallelism (MLP). However, more MSHRs may lead to more warps scheduled to access the L1 D-cache in a short interval and increase the possibility of cache thrashing. So it

is useful if we can better understand the performance impact of the MSHR size.

Fourth, the memory partition mapping function plays a critical role in distributing requests among multiple memory partitions. Although the Modulo address mapping is simple to implement and effective for some applications, it may result in severe memory partition camping and requests are disproportionately handled by one or a small subset of memory partitions on a GPU, leading to the underutilization of DRAM bandwidth and capacity of banked L2 cache. Therefore, it is important to check how memory request distribution among partitions as well as performance will be affected if a different address mapping function is adopted.

Fifth, prior works [3][5][8][13][14][32] on GPU cache bypassing assume there are always adequate hardware resources to store the relevant information of bypassed requests and thus unlimited number of in-flight bypassed requests can be supported. However, such an assumption is overly optimistic in practice.

MRPB [8] is one of the pioneering works on GPU cache management, inspiring research works on GPU cache bypassing [5][13][14][32] and mitigation of memory pipeline stalls [27][31]. In this work, we investigate how it performs with an altered cache indexing function, cache line allocation policy, MSHR size and memory partition mapping functions. Besides MRPB, we also examine the effectiveness of the GPU cache bypassing scheme MDB[5] with the constraint that only a finite number of in-flight bypassed requests can be supported. Overall, we justify the necessity for a sound baseline in GPU memory architecture research.

Overall, this paper makes the following contributions:

- We show that cache indexing functions have remarkable impact on the overall performance and allocate-on-fill brings significantly higher performance for GPGPU applications, than allocate-on-miss;
- We demonstrate that while more MSHRs can provide a higher MLP, performance is not necessarily improved due to the impact on L1 D-cache performance;
- We present that a well-performing memory partition mapping function should be adopted for more balanced request distribution among memory partitions;
- We illustrate that the effectiveness of prior schemes is reduced with the enhanced baseline and the limitation on the number of in-flight bypassed requests imposes non-trivial impact on GPU cache bypassing schemes.

II. BACKGROUND

As shown in Figure 1, multiple warp schedulers can reside in each SM of a GPU and each scheduler supervises multiple warps. And in each SM, there are on-chip memory resources including a L1 read-only texture cache, a L1 read-only constant cache, a L1 data cache (D-cache), and shared memory. A unified L2 cache is shared among multiple SMs. Typically, the L1 D-cache uses the write-evict with either write-allocate [1] or write-no-allocate [20][22] policies, and

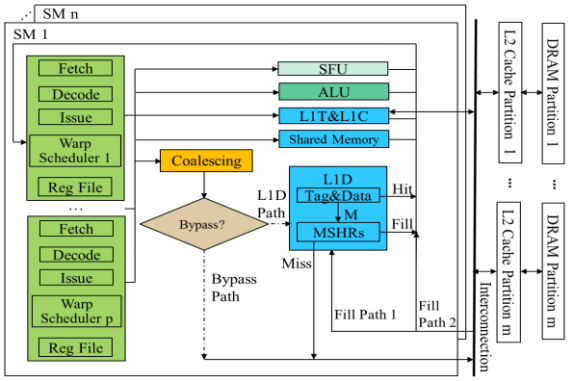


Figure 1. Baseline GPU.

Table 1. Baseline architecture configuration

# of SMs	16, SIMD width=32, 1.4GHz
Per-SM warp schedulers	4 Greedy-Then-Oldest schedulers
Per-SM limit	3072 threads, 96 warps, thread blocks, 64 MSHRs
Per-SM L1D-cache	16KB, 128B line, 4-way associativity
Per-SM shared memory	96KB, 32 banks
Unified L2 cache	2048 KB, 128KB/partition, 128B line, 16-way associativity, 128 MSHRs
L1D/L2 policies	alloc-on-miss, LRU, L1D:WEWN, L2: WBWA
Interconnect	16*16 crossbar, 32B flit size, 1.4GHz
DRAM	16 memory partitions, Modulo mapping, FR-FCFS scheduler, 924MHz

the L2 cache uses the write-back write-allocate policy to save NoC and DRAM bandwidth [28]. Moreover, requests sent to the lower level memory hierarchy (L2 cache and DRAM) are distributed among memory partitions based on address mapping function.

On GPUs, global and local memory requests from threads in a warp are coalesced into as few transactions as possible before being sent to the memory hierarchy. The cached or bypassed information is typically encoded in instruction opcodes [23], indicating whether a request is sent to the L1 D-cache through the 'L1D path' or directly to the L2 cache through the 'Bypass Path', as shown in Figure 1.

For a request sent to the L1 D-cache, the cache indexing function is applied to determine which set to search for the required data and to insert/evict a cache line if it is a miss. Thus, the cache indexing function is crucial to balance requests among cache sets.

Then, for a cache miss, the cache-miss-related resources are allocated before sending the miss request to the L2 cache. For allocate-on-miss, the allocated resources include a cache line slot, a MSHR and a miss queue entry while for allocate-on-fill, just a MSHR and a miss queue entry are allocated. If any of the required resources is not available, a *reservation failure* occurs and the memory pipeline is stalled. Since a MSHR entry is reserved until the data is fetched from lower memory levels, the MSHR size determines how many in-

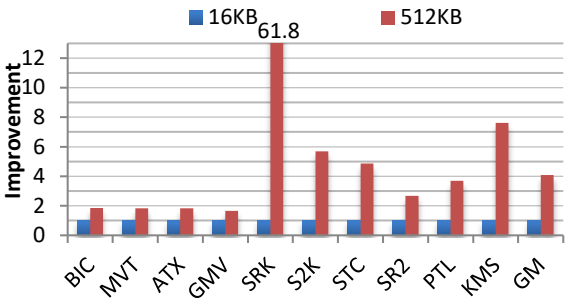


Figure 2. HCC (High Cache Contention) benchmarks from Polybench and Rodinia.

Table 2. Benchmarks

Abbreviation (Description)	Type	Source
BIC (BiCGStab linear solver subkernel)	HCC	[7]
MVT (Matrix-vector-product and transpose)	HCC	[7]
ATX (Matrix-transpose-vector multiply)	HCC	[7]
GMV (Scalar-vector-matrix multiply)	HCC	[7]
SRK (Symmetric rank-2k operations)	HCC	[7]
S2K (Symmetric rank-2k operations)	HCC	[7]
STC (StreamCluster)	HCC	[4]
SRD2 (Srad_v2)	HCC	[4]
PTL (Particle Filter)	HCC	[4]
KMS (K-means clustering)	HCC	[4]

flight outstanding misses can be supported, i.e., the upper bound of memory-level parallelism.

Although a request sent to the L1 D-cache enjoys low access latency if it hits in the L1 D-cache, the massive requests on GPUs easily cause cache thrashing and cache-miss-related resource congestion, degrading the overall performance. GPU cache bypassing has been proposed to effectively mitigate these problems. And similar to the fact that MSHRs are used to record relevant information of outstanding misses, some hardware structure should be deployed to store information for bypassed requests, such as which threads in which warp ask for the data as well as the destination register.

III. Experimental Methodology

Simulation Environment: we use GPGPUsim V3.2.2 [2], a cycle-accurate GPU microarchitecture simulator, to evaluate various design choices. Table 1 shows the baseline Maxwell-like [21] configuration that has been widely used in GPU architecture studies. **Benchmarks:** we evaluate two entire benchmark suites, Rodinia [4] and Polybench [7], including both regular and irregular applications.

Before we start our investigation, we first examine the impact of cache by checking the performance from a small 16KB 4-way set-associative L1 D-cache and a large 512KB full-associative L1 D-cache. Based on the performance improvement from the 512KB L1 D-cache, we classify benchmarks with more than 50% improvement as High Cache Contention (HCC) and others as Low Cache Contention (LCC). Our study focuses on HCC benchmarks

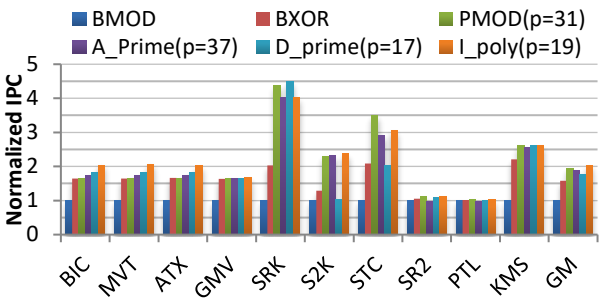


Figure 3. Performance impact of cache indexing functions with baseline cache management.

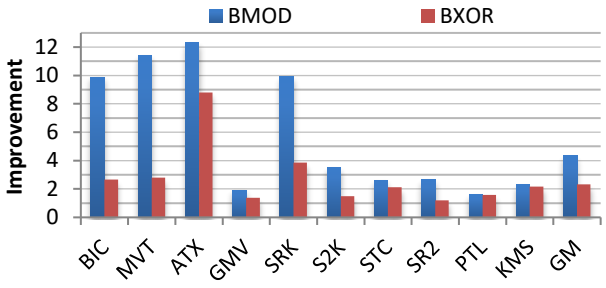


Figure 4. Performance improvement from MRPB with cache indexing functions BMOD and BXOR.

as the is not much variance for LCC ones, the same as prior GPU memory architecture studies [3][5][8][14][26][30].

All HCC benchmarks are shown in Table 2 and Figure 2, where performances are normalized to that from the 16KB L1 D-cache and the average performance from the 512 KB L1 D-cache is 4.09x. The significant performance improvement indicates it is crucial to optimize memory architecture for high performance.

IV. GPU Cache Indexing

In this section, we illustrate the performance impact of cache indexing functions and show that a well performing cache indexing function should be deployed in the first place.

A. Performance impact

With limited cache capacity per thread, GPU caches suffer from severe capacity contention. Furthermore, the capacity may not be well utilized due to a large number of conflict misses, which are resulted from column-majored stride accesses in a warp of threads and thus a high number of un-coalesced requests [30]. Besides, it has been observed that the baseline-Modulo mapping used by default in GPGPUsim may cause pathological performance results [12][14]. Hereby, we thoroughly study the impact of several advanced cache indexing functions to identify the simple and effective one to mitigate conflict misses for GPGPU applications.

Figure 3 shows performance from different GPU cache indexing functions used upon a 16KB L1 D-cache, including BMOD (Baseline Modulo), BXOR (Bitwise XOR[6]), PMOD (Prime Modulo[11]), A_Prime (Another Prime

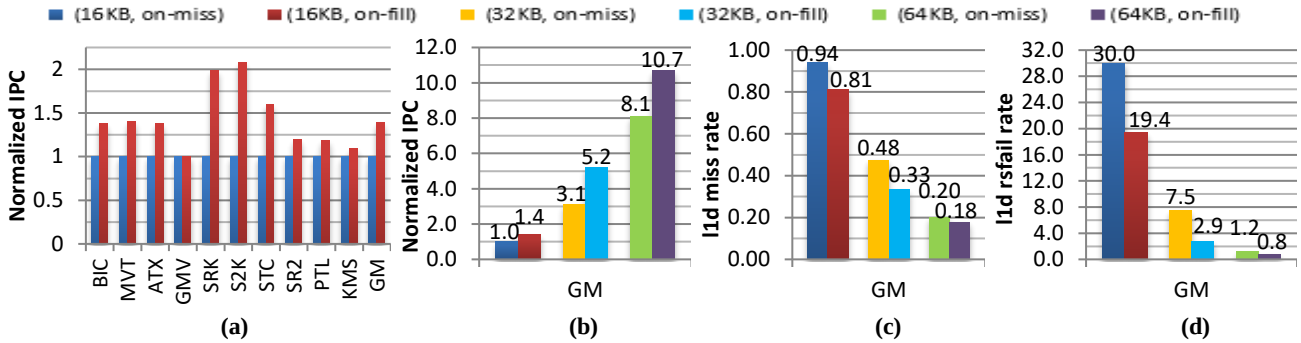


Figure 6. Performance impact of cache line allocation policy with baseline cache management: (a)16KB L1 D-cache: normalized IPC; (b) normalized IPC; (c) L1 D-cache miss rate; (d) L1 D-cache rsfail rate (reservation failures per access).

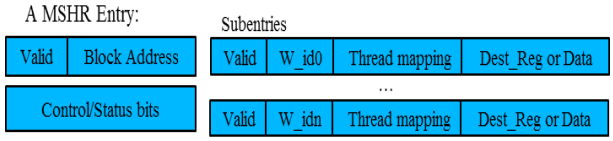


Figure 5. A MSHR Entry on GPUs.

Modulo[16]), D_Prime (Prime Displacement[11]) and I-Poly (Irreducible Polynomial[25]). First, despite the variations, there are significant performance improvements from advanced cache indexing functions, and it is 1.58x, 1.95x, 1.88x, 1.75x and 2.04x for BXOR, PMOD, A_Prime, D_Prime and I_poly, respectively. While other cache indexing functions either lose their effectiveness in certain cases (like D_prime for S2K and STC) or require more complex computation (like I-poly) which may add latency onto the critical path, BXOR is effective and simple to implement. Therefore, BXOR is a good choice for cache set indexing for GPUs, matching the finding in the work [17], which identified that BXOR is used to map addresses to cache sets on GPUs, through micro-benchmarking.

B. Effectiveness of MRPB with different cache indexings

On GPU cache management, Jia et al. proposed MRPB [8] which deploys memory request prioritization buffers to reduce the effective working set and bypasses L1 D-cache when a request encounters a reservation failure. It is shown that MRPB significantly improves the performance of HCC benchmarks but it does not mention how cache sets are indexed in their experiments. Thus, it remains interesting to check the effectiveness of MRPB under BMOD and BXOR.

As shown in Figure 4, MRPB significantly improves the performance of HCC benchmarks when BMOD is used, matching the experimental results in the work [8]. However, the performance improvement of MRPB over the baseline is greatly reduced when BXOR is adopted. On average, the normalized performance over the baseline drops from 4.35x with BMOD to 2.32x with BXOR, respectively, confirming that pathological result may occur when BMOD is used.

Given the remarkable impact of cache indexing on HCC applications, we believe that a well-performing cache indexing function should be deployed in the first place and

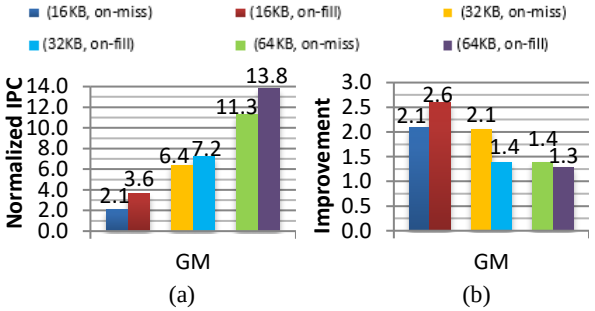


Figure 7. Effectiveness of MRPB with varied cache line allocation policies: (a) normalized IPC; (b) performance improvement over the baseline cache management.

BXOR is good to use in terms of cost-effectiveness. Please note: BOXR cache indexing is used in the following discussion.

V. CACHE LINE ALLOCATION

In this section, we dissect the performance impact of the two cache line allocation policies, namely allocate-on-miss and allocate-on-fill.

A. Performance impact

As described in Section I, when there is an outstanding miss, allocate-on-miss allocates a cache line in addition to a MSHR and a miss queue entry while allocate-on-fill just allocates a MSHR and a miss queue entry. Allocation-on-fill brings in two performance benefits. First, since allocate-on-fill does not evict the victim cache line until the requested data come back from L2 cache/off-chip memory, cache lines have longer lifetime to capture temporal reuses. This is particularly the case for GPUs as the L2 cache latency is much higher than that of CPU L2 caches since multiple SMs share the L2 cache on a GPU. Besides, the in-order execution model and the write-evict policy make the GPU L1 D-cache friendly to allocate-on-fill as there is no dirty data to write to L2 when a victim cache line is to be evicted at the fill time. Furthermore, as allocate-on-fill does not reserve a cache line slot, cache-miss-related resource congestion is lighter.

Figure 5 shows a MSHR entry on GPUs. The basic structure is similar to the *simple organization* proposed by

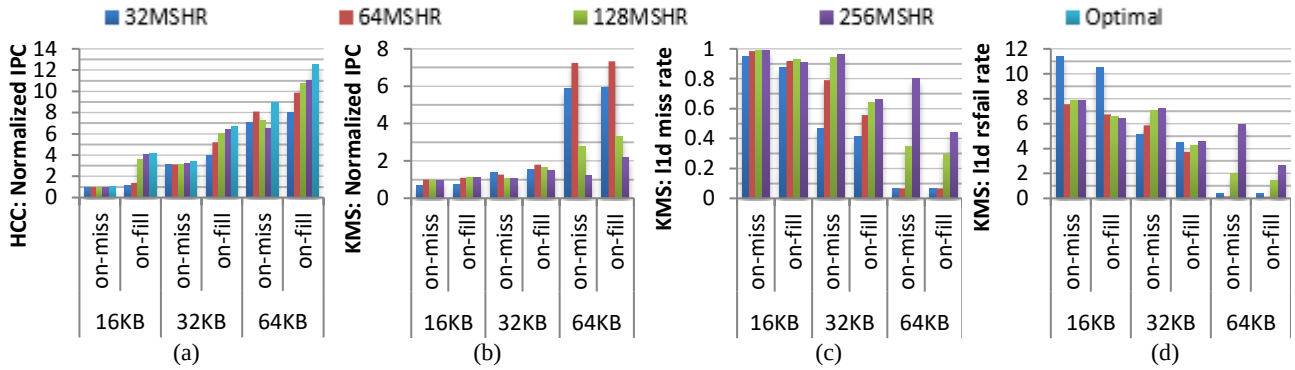


Figure 8. Performance impact of MSHR size: (a) average normalized IPC; (b) KMS: normalized IPC; (c) KMS: L1 D-cache miss rate; (d) KMS: L1 D-cache rsfail rate (reservation failures per access).

Tuck et al [29]. The fields of a single MSHR entry include valid bit, block address, control/status bits (prefetch, which subblocks have arrived, etc.). Due to the nature of gather operation on GPUs, thread mapping in subentries tracks which words in the requested cache line map to which threads, as described in WarpPool [10].

Figure 6(b) shows the performance of allocate-on-miss and allocate-on-fill on various cache capacities, normalized to that of a 16KB L1 D-cache with allocate-on-miss. Individual kernels' performances are also shown for cache capacity 16KB, in Figure 6(a). As demonstrated, allocate-on-fill consistently outperforms allocate-on-miss. On average, the performance from allocate-on-fill (allocate-on-miss) is 1.4x(1.0x), 3.1x(5.2x) and 8.1x(10.7x) with a 16KB, 32KB and 64KB L1 D-cache, respectively.

As discussed, the better performance of allocate-on-fill comes from a higher L1 D-cache efficiency and relieved cache-miss-related resource congestion. Figure 6 (c) and (d) show *L1 D-cache miss rate* and *L1 D-cache rsfail rate* (reservation failures per access), respectively. And take a 32KB L1 D-cache as the example, the L1 D-cache miss rate (rsfail rate) is reduced from 0.48(7.5) with allocate-on-miss to 0.33(2.9) with allocate-on-fill.

B. Effectiveness of MRPB with different cache line allocation policies

Given the significant performance impact of cache line allocation policies, we also check the effectiveness of MRPB when varying this factor. As shown in Figure 7(a), where performances are normalized to that from the baseline cache management on a 16KB L1 D-cache with allocate-on-miss, allocate-on-fill also benefits MRPB with fewer misses and reservation failures, outperforming allocate-on-miss. For example, on a 32KB L1 D-cache, the performance increases from 6.4x with allocate-on-miss to 7.2x with allocate-on-fill.

However, as the performance is already boosted for the baseline cache management with allocate-on-fill, generally the effectiveness of MRPB is reduced. For instance, the performance improvement from MRPB over the baseline on a 32KB L1 D-cache is 110% with allocate-on-miss and it is reduced to 40% with allocate-on-fill, shown in Figure 7(b).

To summarize, considering the non-trivial performance impact, we argue that allocate-on-fill should be examined in the evaluation of GPU cache management schemes.

VI. MSHR SIZES

In this section, we study the impact of MSHR size. Since a MSHR entry is reserved until the required data is returned from lower memory hierarchies, the number of MSHRs determines the number of outstanding misses which can be served in parallel, i.e., the upper bound of MLP. Besides, a scheduled warp will be stalled in the memory pipeline if all MSHRs are reserved by previous outstanding misses and thus TLP may be reduced with a small number of MSHRs.

A. Performance impact

First, we show that the MSHR size has a high impact on the overall performance. Figure 8(a) shows the average performance with different MSHR sizes, namely 32, 64, 128 and 256 MSHRs and when the optimal MSHR size is applied to each benchmark, indicated by 'Optimal'. First, while the average performance increases with more MSHRs in most scenarios, an up-then-down performance trend shows up on a 64KB L1 D-cache with allocate-on-miss, indicating more MSHRs do not necessarily bring a better performance, confirming the observations in the work [31]. Second, the best performing MSHR size varies for different benchmarks and no single MSHR size can hold the advantage consistently. Thus the 'Optimal' performance can be much better than that from any fixed MSHR size. For example, the normalized IPC of 'Optimal' is 8.94x (12.54x) while the best performance among the fixed MSHR sizes is 8.11x (11.0x) from 64MSHR (256MSHR) for allocate-on-miss (allocate-on-fill) with a 64KB L1 D-cache. Fourth, as shown, allocate-on-fill is more immune to the potential adverse effect of more MSHRs and thus consistently obtains a better performance with more MSHRs, on average.

B. The impact of MSHR sizes on L1 D-cache performance

To further investigate the impact of MSHR size, we use KMS as a case study to demonstrate that more MSHRs may hurt the overall performance.

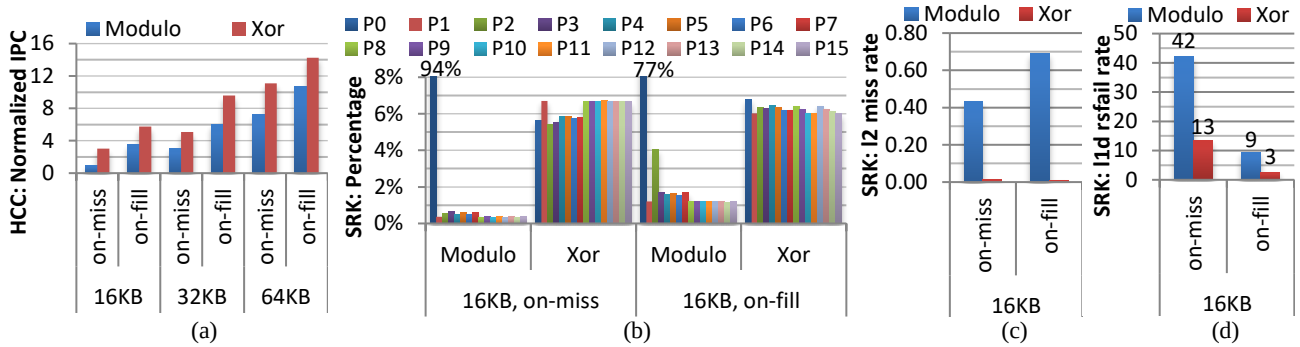


Figure 10. Performance impact of memory partition mapping: (a) average normalized IPC; (b) SRK: percentage of requests across 16 memory partitions; (c) SRK: L2 cache miss rate; (d) L1 D-cache rsfail rate (reservation failures per access).

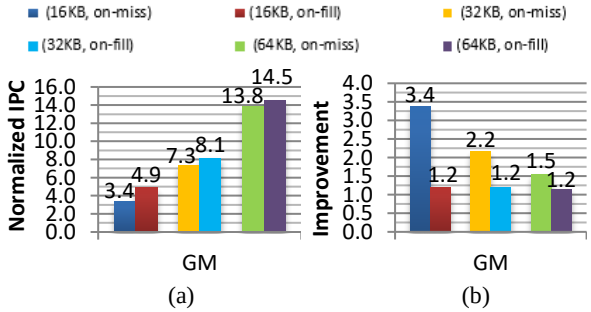


Figure 9. Effectiveness of MRPB with the optimal MSHR size: (a) normalized IPC; (b) performance improvement over the baseline cache management.

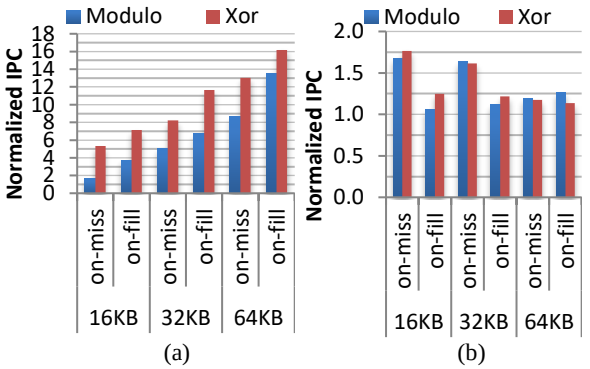


Figure 11. Effectiveness of MRPB with Modulo and Xor memory partition mapping: (a) normalized IPC; (b) improvement over the baseline cache management.

Figure 8(b) presents the performance for KMS from different MSHR sizes, normalized to the performance form a 16 L1 D-cache with allocate-on-miss and 64 MSHRs. For a small 16KB L1 D-cache, where the miss rate remains low, more MSHRs lead to relieved cache-miss-related resource congestion and increased MLP (memory level parallelism), resulting in a better performance. However, for a larger cache, especially, 64KB L1 D-cache, the performance first increases and then decreases with more MSHRs. Specifically, under allocate-on-miss, the performance increases from 5.9x with 32MSHR to 7.3x with 64MSHR because the miss rate remains low (around 0.065) and the number of reservation failures is reduced. Then the performance drops to 2.8x with 128MSHR and to 1.2 with 256MSHR, because both the miss rate and the number of reservation failures significantly increase. And the same variation can also be observed for allocate-on-fill.

To further figure out why the miss rate increases with more MSHRs, we looked into the cycle-by-cycle L1 D-cache accesses and found that due to the multithreading execution model of GPUs, in which a new warp will be scheduled if the current one is waiting for results of its previous instructions, when there are more MSHRs, more warps are actively scheduled to access L1 D-cache, causing cache thrashing; on the other hand, when there are fewer MSHRs, fewer warps can allocate a MSHR for their cache misses and when a request is fulfilled and a reserved MSHR is released, based on the warp scheduling policy, it is highly possible that one of the previously scheduled warps can be scheduled to issue a memory request again and thus has a bigger chance to get a hit in the cache.

To summarize, on GPUs, although more MSHRs can bring a higher MLP, they also enable more requests into caches in a short interval and increase the probability of cache thrashing, confirming the finding in works [17][31] that fewer MSHRs yield better cache behavior for some benchmarks. Nevertheless, since 128 MSHRs can bring a good performance on average, we suggest using 128 MSHRs in the configuration, considering the cost-effectiveness.

C. Effectiveness of MRPB with the optimal MSHR size

Given the significant performance impact of MSHRs, we also check the effectiveness of MRPB when the optimal MSHR size is applied for each benchmark.

On one hand, the performance of MRPB is further improved with the optimal MSHR size, as shown in Figure 9 (a). For instance, the average performance from MRPB is 6.4x (7.2x) on a 32KB L1 D-cache under allocate-on-miss (allocate-on-fill) with 64MSHR (Figure 7(a)) and it is improved to 7.3x (8.1x) when the optimal MSHR size is applied for each benchmark. On the other hand, the performance improvements from MRPB may decrease when the optimal MSHR size is applied to it and the baseline cache management. For example, while the improvement from

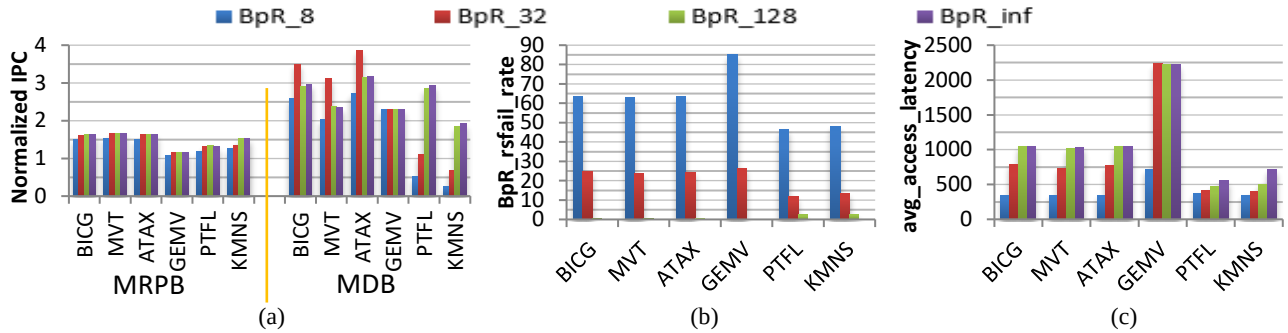


Figure 12. Effectiveness of MRPB and MDB when different numbers of in-flight bypassed requests can be supported: (a) normalized IPC; (b)MDB: reservation failures due to the constraint of BpR_#; (c)MDB: average memory access latency.

MRPB over the baseline cache management is 40% on a 32KB L1 D-cache with 64 MSHRs and allocate-on-fill, it is reduced to 20% when considering the optimal performance.

VII. Request Distribution among Memory Partitions

In this section, we investigate the impact of memory request distribution among partitions. By default, addresses are linearly distributed among memory channels/partitions (*Modulo mapping*), in GPGPUsim. However, with *Modulo mapping*, column-majored stride accesses may cause severe partition camping and requests are disproportionately handled by a small subset of memory partitions on a GPU, leading to performance degradation. In the meanwhile, *Xor mapping* is simple to implement and can overcome the problems with *Modulo mapping*.

Figure 10 demonstrates the impact from the two memory partition mapping functions. Figure 10(a) shows that on average, *Xor mapping* consistently outperforms *Modulo mapping*. For example, the normalized IPC is 3.1x (6.0x) for *Modulo mapping* and 5.1x (9.6x) for *Xor mapping* on a 32KB L1 D-cache with allocate-on-miss (allocate-on-fill).

To further investigate how *Xor mapping* outperforms *Modulo mapping*, we use the benchmark SRK for case study and examine memory request distribution among partitions, L2 cache efficiency and L1 D-cache miss-related resources congestion. First, Figure 10(b) shows that a majority of requests are mapped to memory partition 0 with *Modulo mapping*, leading to extremely low DRAM bandwidth utilization. And *Xor mapping* overcomes the problem of memory partition camping by evenly distributing requests among all 16 partitions. The more balanced memory request distribution not only greatly improves DRAM bandwidth utilization, but also improves L2 cache efficiency. As shown in Figure 10(c), L2 cache miss rate significantly decreases with *Xor mapping*, due to more balanced accesses to L2 banks and thus better L2 capacity utilization. Furthermore, the improved efficiency/performance at the lower level memory hierarchy also benefits L1 D-cache accesses. Figure 10(d) shows that L1 D-cache rsfail rate (reservation failures per access) drops from 42(9) to 13(2) for allocate-on-miss (allocate-on-fill). This is because almost all requests sent to the lower level memory hierarchy can be absorbed by L2 cache under *Xor mapping* (Figure 10(c)). And in turn round trip latency for L1 D-cache misses is significantly reduced, leading to sooner release of MSHRs occupied by those misses and thus relieved congestion in MSHR allocation as well as fewer reservation failures (memory pipeline stalls).

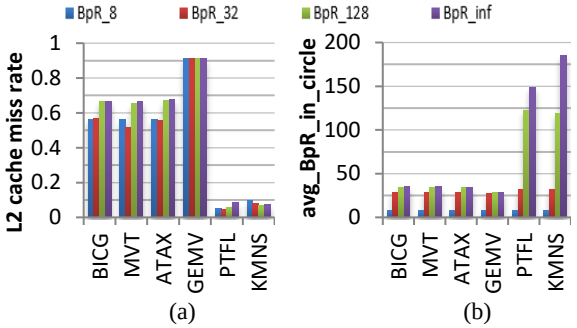
Given the significant performance impact of memory partition mapping, we also check the effectiveness of MRPB when *Xor mapping* is adopted. Figure 11(a) shows that the performance of MRPB is also improved with *Xor mapping*. For example, the normalized IPC increases from 8.6x (13.5x) with *Modulo mapping* to 13.0x (16.2x) with *Xor mapping* on a 64KB L1 D-cache with allocate-on-miss (allocate-on-fill). Regarding the improvement over the baseline (Figure 11(b)), it decreases in some scenarios and increases in others since different benchmarks may react differently when memory partition mapping alters. Nevertheless, it is crucial to ensure balanced request distribution among memory partitions and L2 banks.

VIII. Modelling Realistic GPU Cache Bypassing

In this section, we investigate how bypassing schemes perform when dedicated hardware structures are allocated to record the relevant information of bypassed requests and thus just a finite number of in-flight bypassed requests can be supported. Works [3][5][8][12][13][14][32] on GPU cache management have demonstrated that intelligent cache bypassing can significantly improve the overall performance but failed to discuss the constraint from hardware structures used to keep the relevant information of bypassed requests. However, it is unrealistic to assume that an unlimited number of in-flight bypassed requests can be supported.

Similar to prior GPU cache bypassing studies, allocate-on-miss is used and 128 MSHRs are deployed for a 16KB L1 D-cache in this study. #_BpR is used to denote the maximum number of in-flight bypassed requests that can be supported. And hardware structures similar to but simpler than regular MSHRs (Figure 5) can be used to keep the relevant information of bypassed requests such as which threads ask for the data and the destination register.

Figure 12 (a) shows the performance (normalized to the baseline cache management without bypass) of MRPB [8] and MDB [5] when different numbers of in-flight bypassed



(a) (b)

Figure 13. (a)L2 cache efficiency and (b) the average number of inflight bypassed requests in MDB with different BpR_#.

requests can be served in parallel on a 16 KB L1 D-cache. MDB is a model-drive approach for GPU cache bypassing and it bypasses a certain number of warps or thread blocks based on the combined impact of cache contention and cache-miss-related resource congestion. Here we only show representative benchmarks with diverse and significant performance variance when BpR_# changes. BpR_inf is used by default in prior GPU cache bypassing works, in which any determined bypassing request can be sent to lower memory levels since there is no limitation from hardware to store the relevant information of bypassed requests. As shown, MDB outperforms MRPB, because it can improve L1 D-cache efficiency more effectively. Since benchmarks show more significant performance improvement and also more observable diversity with different values of BpR_# with MDB, we use MDB to further investigate the impact of BpR_# in the following discussion.

It is intuitive to think that the higher BpR_#, the better the performance since the constraint from such a factor is relieved. However, it is not always the case and the examined benchmarks show diverse behaviors, as in Figure 12(a). First, as expected, the performance increases from BpR_8 to BpR_32 for most of the examined benchmarks. Then, from BpR_32 to BpR_128, there is significant performance degradation for benchmarks BICG, MVT and ATAX while the performance of benchmarks GEMV remains relatively stables and benchmarks PTFL and KMNS continuously obtain performance improvement. Finally, there is not much variation between BpR_128 and BpR_inf across all the examined benchmarks.

To better understand the impact of BpR_#, we studied the following two metrics: L1 D-cache BpR_rsfail_rate and average memory access latency. The former one denotes the number of reservation failures per bypassed request due to the constraint from BpR_# and such a reservation failure occurs when a new request is determined to bypass the L1 D-cache but the BpR_# has already been reached by prior bypassed requests. And the metric, average memory access latency, represents the time interval between when a request is sent to the memory hierarchy and when the required data comes back to the requesting SM.

Figure 12(b) shows L1 D-cache BpR_rsfail_rate with BpR_8, BpR_32, BpR_128 and BpR_inf. And we have the following observations. First, with BpR_8 where only 8 in-flight bypassed requests can be supported in maximum, there are a large number of reservation failures due to the constraint of BpR_# and in turn many unsuccessful bypass attempts, resulting in severe memory pipeline stalls. In other words, the effectiveness of GPU cache bypassing may be undermined if just a small number of in-flight bypassed request can be supported. Second, L1 D-cache BpR_rsfail_rate significantly drops from BpR_8 to BpR_32 and this leads to the performance improvement from BpR_8 to BpR_32. Third, L1 D-cache BpR_rsfail_rate continues to drop from BpR_32 to BpR_128 and BpR_inf and there is almost no reservation failures due to the limitation of BpR_# for BpR_128 and BpR_inf.

However, although L1 D-cache BpR_rsfail_rate is near-zero for BpR_128 and BpR_inf, the performance is not necessarily better compared to that when fewer inflight bypassed requests can be supported. For instance, the normalized IPC drops from 3.49x with BpR_32 to 2.90x with BpR_128 for benchmarks BICG. Such performance degradation occurs due to the lengthened memory access latency, as shown in Figure 12(c). Specifically, the average memory access latency increases from 793 cycles with BpR_32 to 1051 with BpR_128 for BICG.

Despite that benchmarks BICG, MVT and ATAX show performance degradation from BpR_32 to BpR_128, benchmarks PTFL and KMNS obtain continuous performance improvement with a larger BpR_#. Similar to other benchmarks, PTFL and KMNS encounter fewer reservation failures and lengthened memory access latency when more inflight bypassed values of memory access latency with a larger BpR_#, as shown in Figure 12 (b) and (c). However, the increment of memory access latency is minor for PTFL and KMNS. Specifically, from BpR_32 to BpR_128, the average memory access latency just increases from 413 to 469 for PTFL and from 405 to 507 for KMNS. Thus although the memory access latency is lengthened for PTFL and KMNS, it still has a relatively low value and does not offset the benefits brought by fewer reservation failures due to the constraint of BpR_#. In contrast, since the average memory access latency of GEMV is more than 2200 cycles starting from BpR_32 and the benefits from fewer reservation failures are offset and the performance of GEMV remains relatively stable across all examined BpR_#.

A request, which bypasses or encounters a miss at L1 D-cache, goes through the interconnect network and then gets served by either L2 cache or DRAM. Therefore, the access latency of such a request has two major parts, one is to go through the interconnect network and the other is to be accommodated by L2 cache or DRAM.

Thus to further investigate the impact of BpR_#, we check L2 cache miss rate and avg_BpR_in_circle, as shown in Figure 13. L2 cache miss rate indicates the L2 cache efficiency and the higher L2 cache miss rate, the more

requests are sent to DRAM and the larger average latency for a request to be served. The metric avg_BpR_in_circle denotes the average number of inflight bypassed requests during execution and it reflects the extent of interconnect congestion. Basically, the larger avg_BpR_in_circle, the higher degree of interconnect congestion and the longer latency for a request to go through the interconnect network. First, Figure 13(a) shows that for benchmarks BICG, MVT and ATAX, there is non-trivia L2 cache miss rate increase from BpR_32 to BpR_128 because more warps are actively scheduled to send requests to the memory subsystem. In the meanwhile, since there are more inflight bypassed requests, as shown in Figure (b), the latency to go through the interconnect network also increases. The combined effect of the two factors leads to the significantly lengthened memory access latency and performance degradation for the three benchmarks. Then for the benchmark PTFL and KMNS, L2 cache miss rate remains lower than 0.1 for various BpR_#, indicating that more inflight bypassed requests do not thrash L2 cache and almost all L1 D-cache misses can be absorbed by it. On the other hand, unlike other benchmarks, PTFL and KMNS have a large number of bypassed requests and in turn a high degree of memory-level-parallelism (MLP). For example, with BpR_128, the value of avg_BpR_in_circle for BICG is 35, and it is as high as 123 for PTFL and 118 for KMNS. Since PTFL and KMNS experience a much shorter memory access latency and a higher MLP, they can have more requests served and in turn execute more data-dependent instructions per cycle, and therefore obtains continuous performance improvement with a larger BpR_#.

As demonstrated, the number of in-flight bypassed requests can significantly affect the performance of GPU cache bypassing schemes. Therefore, it is not realistic to assume there are unlimited hardware resources to store the relevant information of bypassed requests. On the other hand, the higher number of in-flight bypassed requests to be supported does not necessarily brings higher performance due to the congestion in interconnect network and conflicts at lower memory levels. Besides, a limitation on the number of in-flight bypassed requests can also achieve bypass throttling, which is targeted by some prior works [3][14]. So, we believe the fact that only a limited number of in-flight bypassed requests can be supported should be taken into account, to get more realistic results and conclusions in GPU cache bypassing studies.

IX. Concluded Sound Baseline Configuration

In this part, we give out the suggested sound baseline configuration. Based on our study, we argue for the following methodology to be used in GPU memory architecture research: (1) an indexing function such as BXOR to reduce conflict misses in the caches; (2) allocation-on-fill policy in the GPU caches to improve cache utilization; (3) for studies on memory-level parallelism, the number of MSHRs needs to be explored as an important design space parameter; (4) a memory partition mapping

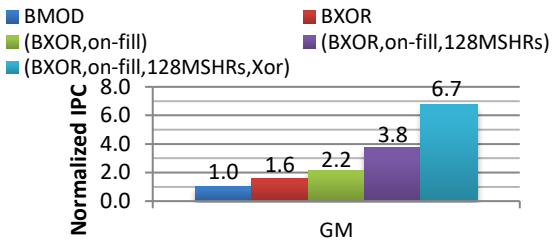


Figure 14. Accumulated performance improvement from an enhanced baseline with a 16KB L1 D-cache.

function such as Xor to mitigate the problem of memory partition camping; (5) studies on cache bypassing should not assume unlimited number of bypasses. Instead, the bypass slots (i.e., the maximal number of in-flight bypasses) is an important design space parameter to be explored.

If a single baseline is desired (i.e., no design space exploration), the sound one from our results is:

BXOR + allocate-on-fill + 128 MSHRs + 32/128 bypassing slots, with Xor mapping used to distribute requests among memory partitions.

The sound baseline is open sourced at:

https://github.com/ShadowArray/WDDD-Sound-Baseline

Regarding the performance of the enhanced baseline, we show the accumulated performance improvement with a 16KB L1 D-cache, in Figure 14. Without specific description in the legend, the default configuration is (BMOD, on-miss, 64MSHRs, Modulo) which uses BMOD for cache set indexing and allocate-on-miss as the cache line allocation policy, deploys 64MSHRs and distributes requests among memory partitions with *Modulo mapping*. As shown, the performance continuously increases when the baseline is enhanced. And on average, the accumulated performance is as high as 6.7x with (BXOR, on-fill, 128MSHRs, Xor), compared to the default configuration in GPGPUsim.

For the indexing function, it may not be the best performance-wise as some other hashing functions may distribute the accesses more evenly than BXOR. But considering the hardware complexity, our results suggest that BXOR is good to use. For allocation-on-fill vs. allocation-on-miss, allocation-on-fill extends the life-time of the cached data. Therefore, it is better than allocation-on-miss in general. And 128 MSHRs can greatly mitigate reservation failures. Finally, as illustrated in Section VIII, applications show diverse behaviors when more inflight bypassed requests can be supported. Some applications show an up-then-down performance trend, some present a relatively stable performance and others continuously reap performance improvement. As such, we suggest that two points, 32 and 128 bypassing slots should be studied. On one hand, the configuration of 32 bypassing slots can achieve bypass throttling which is targeted by some prior works [3][14]. On the other hand, 128 bypassing slots can achieve performance close to that when there is no constraint on the number of inflight bypassed requests and since it leads to a

higher memory-level parallelism, potentially it can benefit applications for which L2 cache has a high efficiency and can effectively filter requests sent to it.

Although not shown here, in addition to Greedy-Then-Oldest (GTO) used so far, we have also experimented Loose-Round-Robin (LRR) warp scheduling policy and found that the overall performance is also boosted with the enhanced baseline as the memory subsystem efficiency is improved. Thus the suggested sound baseline shall be used despite that a different warp scheduling policy may be adopted.

X. Related Work

Although cache indexing functions have been well studied on CPUs [6][11][16], previous works on GPU cache management did not elaborate on this issue in detail. On one hand, some works did not mention the underlying cache indexing function, like MRPB[8] and WarpPool[10]. On the other hand, although some other works pointed out that the BMOD mapping used by default in GPGPUsim might cause pathological results [12][14][17], they did not thoroughly study the impact of various advanced indexing functions.

Cache line allocation policy determines what cache-miss-related resources are allocated for an outstanding miss. For allocate-on-miss, those resources include a cache line [9], a MSHR and miss queue entry while allocate-on-fill [2] does not reserve a cache line. Therefore, allocate-on-fill tends to incur fewer reservation failures and enjoy more hits. Besides, although some works [17][27][31] have mentioned the potential performance impact of MSHR size on GPUs, they did not study nor examine the impact with varying other factors. In contrast, we studied the impact of MSHR size with different cache sizes and cache line allocation policies.

Although many prior GPU cache bypassing works have shown significant performance improvements [3][5][8][13] [14][32] from their schemes, they did not mention the constraint from the hardware structures used to store the relevant information of bypassed requests. Since only a finite number of in-flight bypassed requests can be supported in reality, we demonstrate that it should be taken into account in GPU cache bypassing studies.

XI. Conclusions

As throughput oriented processors, GPUs leverage massive multithreading to hide long operation latencies. However, the massive memory requests in GPGPU applications lead to fewer cache lines per thread and shorter cache line lifetime on GPUs than CPUs. In this work, we comprehensively investigated the performance impact of cache set indexing, cache line allocation policy, the number of MSHRs, and request distribution among memory partitions on GPUs as well as more realistic GPU cache bypassing.

Our studies show that advanced cache indexing functions should be deployed in the first place to reduce the severe conflict misses; allocate-on-fill should be used to increase cache hits and reduce memory pipeline stalls; the number of MSHRs plays an important role in affecting the cache efficiency besides supporting MLP/TLP. Furthermore, we show that a good memory partition mapping function, such as Xor, should be deployed to mitigate the problem of memory camping. And while previous GPU cache bypassing works unrealistically assume an unlimited number of in-flight bypassed requests can be supported, we demonstrate such a constraint can significantly affect the performance of a GPU cache bypassing scheme and this factor should be taken into account in GPU cache bypassing studies. Finally, we propose the sound baseline configuration for future GPU memory architecture studies and open source it.

References

[1] AMD GCN Architecture White paper, 2012.

[2] A. Bakhoda et al. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of ISPASS*, 2009.

[3] X. Chen et al. Adaptive cache management for energy-efficient GPU computing. In *Proceedings of MICRO*, 2014.

[4] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of IISWC*, 2009.

[5] H. Dai, et al. A Model-Driven Approach to Warp/Thread-Block Level GPU Cache Bypassing. In *Proceedings of DAC*, 2016.

[6] A. González et al. Eliminating cache conflict misses through XOR-based placement functions. In *Proceedings of ICS*, 1997.

[7] S. Grauer-Gray et al. Auto-tuning a high-level language targeted to GPU codes. In *Proceedings of InPar*, 2012.

[8] W. Jia et al. MRPB: Memory request prioritization for massively parallel processors. In *Proceedings of HPCA*, 2014.

[9] D. Kroft, et al. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of ISCA*, 1981.

[10] J. Kloosterman, et al. WarpPool: sharing requests with inter-warp coalescing for throughput processors. In *Proceedings of MICRO*, 2015.

[11] M. Kharbutli et al. Using prime numbers for cache indexing to eliminate conflict misses. In *Software, IEE Proceedings*, 2004.

[12] A. Li et al. Adaptive and transparent cache bypassing for GPUs. In *Proceedings of SC*, 2015.

[13] C. Li et al. Locality-Driven Dynamic GPU Cache Bypassing. In *Proceedings of ICS*, 2015.

[14] D. Li et al. Priority-based cache allocation in throughput processors. In *Proceedings of HPCA*, 2015.

[15] J. Liu, et al. SAWS: synchronization aware GPGPU warp scheduling for multiple independent warp schedulers." In *MICRO*, 2015.

[16] Y. Ma, et al. Using indexing functions to reduce conflict aliasing in branch prediction tables. IEEE Transactions on Computers 8 (2006).

[17] C. Nugteren, et al. A Detailed GPU Cache Model Based on Reuse Distance Theory. In *Proceedings of HPCA*, 2014.

[18] V. Narasiman et al. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of* MICRO, 2011.

[19] NVIDIA, "CUDA C/C++ SDK code samples,"2011.

[20] NVIDIA Kepler GK110 Architecture Whitepaper, 2012.

[21] NVIDIA GeForce GTX 980 Whitepaper, 2014.

[22] NVIDIA's CUDA compute architecture: Fermi. 2009.

[23] NVIDIA Parallel Thread Execution ISA Version 4.2.

[24] NVIDIA Pascal GP100 Architecture, GTC, 2016.

[25] B. R. Rau et al. Pseudo-randomly interleaved memory." *ISCA*, 1991.

[26] T. Rogers et al. Cache-conscious wavefront scheduling." In *Proceedings* MICRO, 2012.

[27] A. Sethia et al. Mascar: Speeding up gpu warps by reducing memory pitstops. In *proceedings of HPCA*, 2015.

[28] I. Singh et al. Cache coherence for GPU architectures. In *Proceedings of HPCA*, 2013.

[29] J. Tuck et al. Scalable cache miss handling for high memory-level parallelism. In *Proceedings of MICRO*, 2006.

[30] B. Wang et al. Eliminating intra-warp conflict misses in GPU. In *Proceedings* of DATE, 2015.

[31] B. Wang et al. "OAWS: Memory Occlusion Aware Warp Scheduling." In *Proceedings* of PACT, 2016.

[32] X. Xie et al. Coordinated static and dynamic cache bypassing for GPUs. In *Proceedings of HPCA*, 2015.