Accelerate GPU Concurrent Kernel Execution by Mitigating Memory Pipeline Stalls

Hongwen Dai[1], Zhen Lin[1], Chao Li[1], Chen Zhao[2], Fei Wang[2], Nanning Zheng[2], Huiyang Zhou[1]
[1]Department of Electrical and Computer Engineering, North Carolina State University
{hdai3, zlin4, cli17, hzhou}@ncsu.edu
[2]School of Electrical and Information Engineering, Xi'an Jiaotong University
{chenzhao, wfx, nnzheng}@mail.xjtu.edu.cn

***Abstract*-**Following the advances in technology scaling, graphics processing units (GPUs) incorporate an increasing amount of computing resources and it becomes difficult for a single GPU kernel to fully utilize the vast GPU resources. One solution to improve resource utilization is concurrent kernel execution (CKE). Early CKE mainly targets the leftover resources. However, it fails to optimize the resource utilization and does not provide fairness among concurrent kernels. Spatial multitasking assigns a subset of streaming multiprocessors (SMs) to each kernel. Although achieving better fairness, the resource underutilization within an SM is not addressed. Thus, intra-SM sharing has been proposed to issue thread blocks from different kernels to each SM. However, as shown in this study, the overall performance may be undermined in the intra-SM sharing schemes due to the severe interference among kernels. Specifically, as concurrent kernels share the memory subsystem, one kernel, even as computing-intensive, may starve from not being able to issue memory instructions in time. Besides, severe L1 D-cache thrashing and memory pipeline stalls caused by one kernel, especially a memory-intensive one, will impact other kernels, further hurting the overall performance.

In this study, we investigate various approaches to overcome the aforementioned problems exposed in intra-SM sharing. We first highlight that cache partitioning techniques proposed for CPUs are not effective for GPUs. Then we propose two approaches to reduce memory pipeline stalls. The first is to balance memory accesses of concurrent kernels. The second is to limit the number of inflight memory instructions issued from individual kernels. Our evaluation shows that the proposed schemes significantly improve the weighted speedup of two state-of-the-art intra-SM sharing schemes, Warped-Slicer and SMK, by 24.6% and 27.2% on average, respectively, with lightweight hardware overhead.

1. Introduction

Following the technology scaling trend, modern GPUs integrate an increasing amount of computing resources [1][28][29][30][31]. Since GPUs have become prevalent in high performance computing, they need to support applications with diverse resource requirements. As a result, GPU resources are typically underutilized by a single kernel.

To solve the problem of GPU resource underutilization, concurrent kernel execution (CKE) [20] has been proposed to support running multiple kernels concurrently on a GPU. One approach to achieve concurrent kernel execution is to apply the left-over policy, in which resources are assigned to one kernel as much as possible and the leftover resources are then used for another kernel. The examples implementing this approach include the queue-based multiprogramming [35][37] introduced by AMD and Hyper-Q by NVIDIA [29]. However, the simple left-over policy fails to optimize resource utilization and does not provide fairness or quality of service (QoS) to concurrent kernels.

Researchers have proposed software and hardware schemes to better exploit CKE. Studies have shown that CKE improves GPU resource utilization especially when kernels with complementary characteristics are running together. Models [17] [26] [48] aim to find the optimal pair of kernels to run concurrently. Kernel slicing [48] partitions a big kernel into smaller ones such that no single kernel consumes all resources. Elastic kernel [32] dynamically adjusts the kernel size based on the resource availability. Those approaches have demonstrated the advantages of CKE. However, it may not be feasible to modify every application. To exploit CKE more broadly, we focus on hardware approaches in this work.

One hardware-based CKE scheme is spatial multitasking [2], which groups streaming multiprocessors (SMs) in a GPU into multiple sets and each set can execute a different kernel. Such SM partition enables better fairness among kernels but does not address resource underutilization within an SM. For instance, the computing resources in an SM, often idle when running a memory-intensive kernel, cannot be utilized for a compute-intensive kernel on other SMs.

One appealing approach to improve resource utilization within an SM is intra-SM sharing, in which thread blocks from different kernels can be dispatched to one SM. The intuitive idea is to run kernels with complementary characteristics concurrently on an SM, such as a compute-intensive kernel and a memory-intensive one.

SMK [45] and Warped-Slicer [46] are two state-of-the-art intra-SM sharing schemes, and they adopt different algorithms to determine Thread-Block (TB) partition among concurrent kernels, i.e., how many TBs can be issued from individual kernels to the same SM. Specifically, SMK uses the metric 'Dominant Resource Fairness (DRF)' to fairly allocate static resources (including registers, shared memory, number of active threads and number of TBs) among kernels. As good fairness in static resource allocation does not

necessarily lead to good performance fairness, SMK also periodically allocates quotas of warp instructions for individual kernels based on profiling each kernel in isolation. On the other hand, Warped-Slicer determines TB partition based on scalability curves (performance vs. the number of TBs from a kernel in an SM), which can be obtained with either offline/static profiling each kernel in isolation or online/dynamic profiling during concurrent execution. The TB partition, for which the performance degradation of each kernel is minimized when running them concurrently, is identified as the sweet point.

Although SMK and Warped-Slicer outperform spatial multitasking, their static approaches profile individual kernels in isolation and the dynamic approaches do not fully address the interference among concurrent kernels within an SM. First, since kernels share the same memory pipeline in intra-SM sharing, one kernel will starve if it continuously loses competition to issue memory instructions. Specifically, as compute-intensive kernels have significantly fewer memory instructions than memory-intensive ones, their memory instructions tend to be delayed, leading to their severe performance loss. Second, as shown in previous works [8] [38], memory pipeline stalls caused by cache-miss-related resource saturation prevent ready warps from issuing new memory instructions. In intra-SM sharing, such memory pipeline stalls and L1 D-cache thrashing caused by one kernel will impose stalls on all other co-running kernels, hurting their performance.

To overcome the aforementioned issues in intra-SM sharing on GPUs, this paper explores the following approaches. First, we investigate the effectiveness of cache partitioning and highlight that cache partitioning cannot effectively reduce memory pipeline stalls on GPUs. Second, we propose to balance memory request issuing such that a compute-intensive kernel does not undergo starvation in accessing the shared memory subsystem. Third, we propose memory instruction limiting to control the number of inflight memory instructions from individual kernels so as to reduce memory pipeline stall and relieve L1 D-cache thrashing. With the proposed schemes, we can reduce interference among concurrent kernels and mitigate memory pipeline stalls, thereby achieving higher computing resource utilization, weighted speedup, and fairness.

Overall, this paper makes the following contributions:

- We demonstrate that while the state-of-the-art intra-SM sharing schemes can identify a good performing TB partition, they do not fully address the interference, especially within an SM, among concurrent kernels.
- We show that while cache partitioning cannot reduce memory pipeline stalls, it is beneficial to balance memory accesses and limit the number of inflight memory instructions from concurrent kernels.
- Our experiments show that compared to the two state-of-art intra-SM sharing schemes: our approaches improve the Weighted Speedup of Warped-Slicer and SMK by 24.6% and 27.2% on average, respectively.

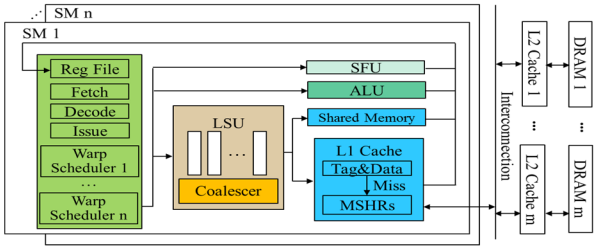


Figure 1. Baseline GPU.

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, 16 thread blocks, 128 MSHRs
Per-SM L1D-cache	24KB, 128B line, 6-way associativity
Per-SM SMEM	96KB, 32 banks
Unified L2 cache	2048 KB, 128KB/partition, 128B line, 16-way associativity, 128 MSHRs
L1D/L2 policies	xor-indexing, allocate-on-miss, LRU, L1D:WEWN, L2: WBWA
Interconnect	16*16 crossbar, 32B flit size, 1.4GHz
DRAM	16 memory channels, FR-FCFS scheduler, 924MHz, BW: 48bytes/cycle

2. Motivation and Methodology

2.1. Baseline Architecture and Memory Request Handling

As shown in Figure 1, a modern GPU consists of multiple streaming multiprocessors (SMs). 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 instructions in a SIMD manner. More than one warp scheduler can reside in one SM.

Besides massive multithreading, GPUs have adopted multi-level cache hierarchies to mitigate long off-chip memory access latencies. Within each SM, the on-chip memory resources include a read-only texture cache and a constant cache, an L1 data cache (D-cache), and shared memory. A unified L2 cache is shared among multiple SMs.

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. 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, including a miss status handling register (MSHR) and a miss queue entry, and then the request is sent to the L2 cache. 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 sent to the L2 cache.

2.2. Multiprogramming Support in GPUs

Since GPUs continue to incorporate an increasing amount of computing resources, CKE has been introduced to improve

Table 2. Benchmarks

Benchmark	RF_oc	SMEM_oc	Thread_oc	TB_occu	Cinst /Minst	Req /Minst	l1d_ miss_rate	l1d_ rsfail_rate	Type
cp(cutcp)[40]	87.5%	67.0%	66.7%	100.0%	4	2	0.45	0.04	C
hs (hotspot) [7]	98.4%	21.9%	58.3%	43.8%	7	3	0.97	1.53	C
dc(dxtc)[27]	56.2%	33.3%	33.3%	100.0%	5	1	0.09	0.17	C
pf (pathfinder) [7]	75.0%	25.0%	100.0%	75.0%	6	2	0.99	0.00	C
bp (backprop) [7]	56.2%	13.3%	100.0%	75.0%	6	2	0.80	0.33	C
bs(bfs)[7]	75.0%	0.0%	100.0%	37.5%	4	1	1.00	0.00	C
st(stencil)[40]	75.0%	0.0%	100.0%	37.5%	4	1	0.67	1.15	C
3m (3mm) [11]	56.2%	0.0%	100.0%	75.0%	2	1	0.63	5.45	M
sv (spmv) [40]	75.0%	0.0%	100.0%	100.0%	3	3	0.78	5.23	M
cd(cfd)[7]	100.0%	0.0%	33.3%	100.0%	9	6	0.96	7.23	M
s2(sad2)[40]	50.0%	0.0%	66.7%	100.0%	2	2	0.92	6.80	M
ks (kmeans) [7]	56.2%	0.0%	100.0%	75.0%	3	17	1.00	7.96	M
ax (ATAX) [11]	56.2%	0.0%	100.0%	75.0%	2	11	0.97	79.70	M

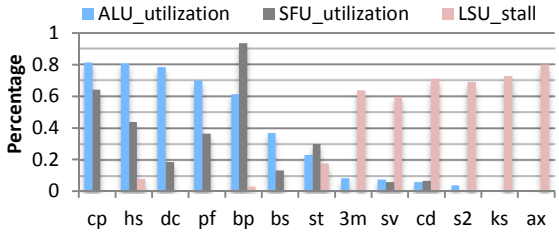


Figure 2. Computing resource utilization and LSU stalls.

resource utilization. With the Hyper-Q architecture [29], kernels are mapped into multiple stream queues. Grid launch inside a GPU kernel has been proposed to reduce costly CPU intervention [19]. The HSA foundation [35] introduced a queue based approach for heterogeneous systems with GPUs. However, none specifies how memory instructions are selected to issue from individual kernels when they run concurrently in the same SM.

2.3. Methodology

We use GPGPUsim V3.2.2 [5], a cycle-accurate GPU microarchitecture simulator, to evaluate different CKE schemes. Table 1 shows the baseline NVIDIA Maxwell-like GPU architecture configuration. We extensively modified GPGPUsim to issue warp instructions from concurrent kernels, which share the same backend execution pipeline.

We have studied various GPU applications from NVIDIA CUDA SDK [27], Rodinia [7], Parboil [40] and Polybench [11]. CKE workloads are constructed by paring different applications. Each workload runs for 2M cycles and a kernel will restart if it completes before 2M cycles, the same as in previous works [2][45]. We mainly report the evaluation using the Weighted Speedup, which is the sum of speedups of co-running kernels, speedup being defined as the normalized IPC in concurrent execution over the IPC in isolated execution, and the average normalized turnaround time (ANTT), which quantifies the average user-perceived slowdown and incorporates fairness [10]. Besides, while it is intuitive to concurrently run kernels with complimentary characteristics, such a practice requires knowledge about the characteristics of kernels to run. Not to lose generality, we also report the experimental results for concurrent kernels of the same type. Also, as different compute-intensive kernels may use different types of computing units and different memory-intensive kernels may stress various parts along the memory access path (e.g. the interconnect, the L2 cache, and memory), there is still potential to increase the overall GPU resource utilization for concurrent kernels of the same type with dedicated management.

2.4. Workload Characterization

In this section, we classify applications into compute-intensive and memory-intensive categories. Then we present motivational data regarding how the utilization of computing units, the percentage of LSU (Load/Store Unit) stall cycles, and the memory access behaviours vary across applications.

Table 2 and Figure 2 show the benchmark characteristics. First, Table 2 presents the occupancy of static resources (including registers, shared memory, the number of threads and the number of TB slots). It is possible to improve the static resource utilization by running benchmarks with complementary requirements concurrently, as discussed in SMK [45]. Second, as shown in Figure 2, where benchmarks are arranged in the decreasing order of ALU utilization, an inverse relationship exists between utilization of computing units and percentage of LSU stalls. Based on the percentage of LSU stalls, we classify benchmarks with more than 20% LSU stalls as *memory-intensive* (**M**), and others as *compute-intensive* (**C**), indicated in the column 'Type' in Table 2. We shall note that LSU stalls reflect memory pipeline stalls resulted from failing to allocate cache-miss-related resources (MSHRs, miss queue entries, etc.) for outstanding misses and since more such resources are provisioned in our modelled architecture (Table 1), certain benchmarks, e.g., bs, may show different characteristics, compared to previous works [6][36].

Table 2 also shows that compute-intensive kernels and memory-intensive ones have different memory/cache access behaviours. First, compute-intensive kernels have more compute instructions per memory one, than memory-

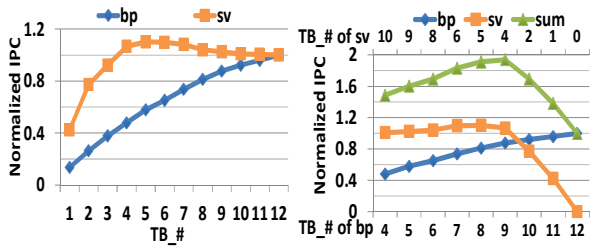


Figure 3. (a) Performance vs. increasing TB occupancy in one SM, (b) identify the performance sweet spot.

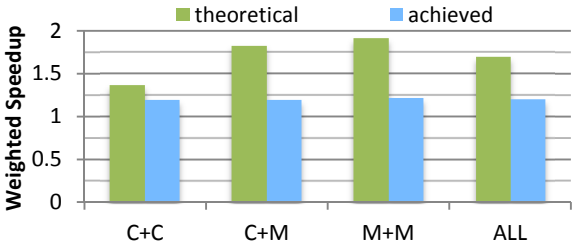


Figure 4. Performance gap of dynamic Warped-Slicer-like approach when adapted for performance prediction.

intensive kernels, as indicated in the column 'Cinst/Minst'. For example, while 'Cinst/Minst' is 7 for hs, it is only 2 for 3m. Second, memory-intensive kernels may have different degrees of memory request coalescing, as indicated in the column 'Req/Minst' which denotes the average number of requests per memory instruction. For instance, while 'Req/Minst' is 3 for sv, it is 17 for ks. And the other two columns 'l1d_miss_rate' and 'l1d_rsfail_rate' in Table 2 denote miss rate and reservation failures per L1 D-cache access, respectively. Although compute-intensive kernels may show a high 'l1d_miss_rate', they still achieve high utilization of compute units, due to the streaming accesses in L1 D-cache and the usage of shared memory. And a high 'l1d_rsfail_rate' of a kernel implies severe cache-miss-related resource congestion and thus memory pipeline stalls

2.5. Motivational Analysis

In this part, we illustrate that while the state-of-the-art SM-sharing scheme, Warped-Slicer, can find a good performing TB partition, there is potential to improve the performance by addressing the intra-SM interference among kernels.

Figure 3 illustrates the workflow of Warped-Slicer with the two-programmed workload bp+sv, where bp is compute-intensive and sv is memory-intensive. Figure 3(a) shows the performance scalability curves of both benchmarks when they run in isolation. While the performance of bp near-linearly increases with more TBs, the performance of sv first increases and then decreases with more TBs launched to an SM. Then as shown in Figure 3(b), the scalability curves of bp and sv are used to identify the sweet point, i.e., the TB combination, where the performance degradation of each kernel is minimized while meeting the resource constraint. For the case of bp+sv, the sweet point is (9, 4), i.e. 9 TBs from bp and 4 TBs from sv, and the theoretical Weighted Speedup, which is computed as the sum of normalized IPCs, for bp+sv is 1.94 at the sweet point.

As introduced in Section 1, while the static Warped-Slicer approach profiles each kernel in isolation, the dynamic approach obtains the performance scalability curves of co-running kernels concurrently by running different numbers of TBs on SMs (1 TB on one SM, 2 TBs on a second SM and so on), where each SM is allocated to execute TBs from one kernel and time sharing of SMs is applied if the total number of possible TB configurations from all co-running kernels is more than the number of SMs. The dynamic approach considers the interference among co-running kernels across SMs and uses the scaling factor and the weight factor to offset the imbalance problem in the L2 cache and memory accesses [46].

Figure 4 shows that the achieved Weighted Speedup is lower than the theoretical value when the optimal TB partitions are selected using dynamic Warped-Slicer. The results are the geometric mean of all the combinations of 2 kernels and different combinations exhibit different features regarding how individual kernels perform. First, for C+C workloads, the achieved Weighted Speedup is close to the theoretical and co-running kernels show similar speedups. Second, for C+M workloads, the memory-intensive kernel may dominate the usage of memory pipeline and also execution in an SM while the compute-intensive one shows significant performance loss because its memory requests suffer delays and cannot be timely served, so as its computation operations. Third, for M+M workloads, one kernel may suffer more than the other, due to their different memory/cache access behaviours. Due to such interference, the achieved Weighted Speedup for C+M and M+M workloads are much lower than the theoretical. Therefore, it is crucial to further reduce the interference among kernels in intra-SM sharing, especially in the memory pipeline and the memory subsystem, so as to improve computing resource utilization and achieve better performance.

3. Overcome the Hurdle of Memory Pipeline Stalls

In Section 2.5, we demonstrated that memory pipeline stalls incurred by one kernel may negatively affect other kernels on the same SM. To overcome this issue, we study three methods to better accommodate memory requests and improve computing unit utilization. The three methods are: 1) cache partitioning; 2) balance memory request issuing to prevent the starvation of any kernel in accessing data; and 3) limit the number of inflight memory instructions to mitigate L1 D-cache thrashing and memory pipeline stalls.

3.1. Cache Partitioning

In this part, we show that simply applying cache partitioning cannot improve the Weighted Speedup for intra-SM sharing.

Figure 5 illustrates the effectiveness of cache partitioning, where 'WS' denotes TB partition using Warped-Slicer and

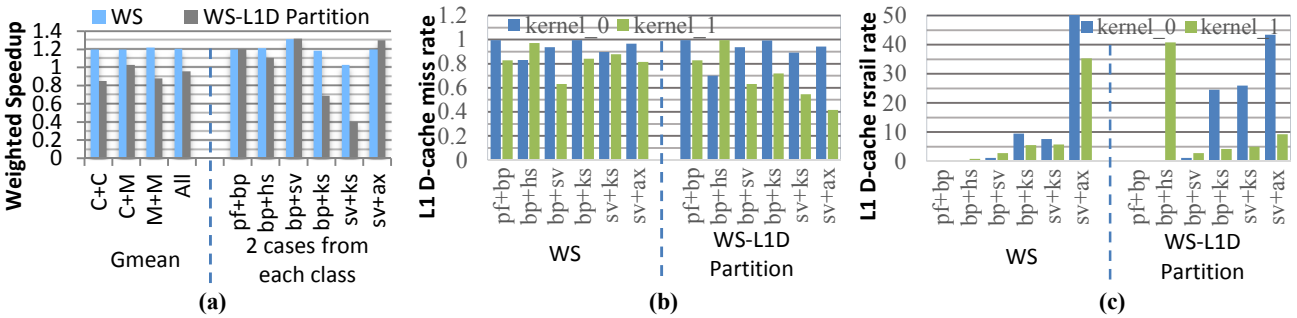


Figure 5. Effectiveness of Cache Partitioning: (a) Weighted Speedup; (b) L1 D-cache miss rate; and (c) L1 D-cache rsfail rate.

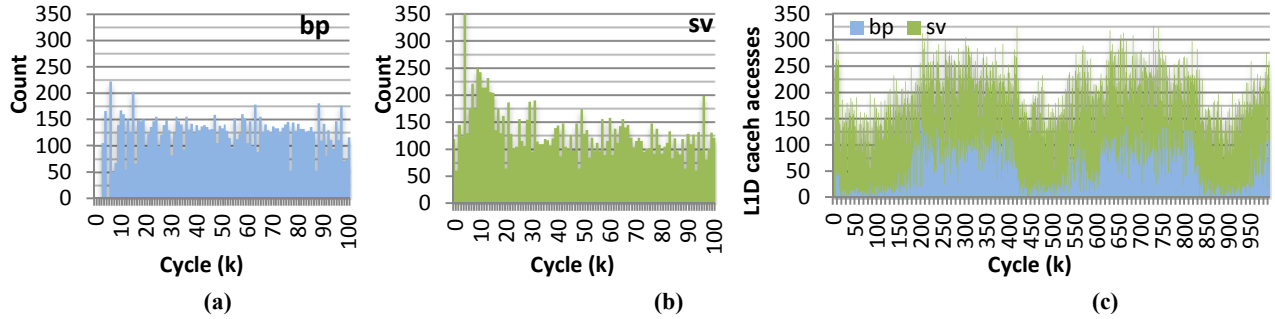


Figure 6. L1 D-cache accesses: (a) bp executes in isolation; (b) sv executes in isolation; (c) bp and sv run concurrently.

'WS-L1D Partition' is for WS with L1 D-cache partitioning, for which UCP (Utility-based Cache Partitioning) [34] is adopted. First, Figure 5(a) shows that, on average, L1 D-cache partitioning fails to improve Weighted Speedup across all three workload classes. Two workloads from each class are selected for further investigation, pf+bp and bp+hs from the C+C class, bp+sv and bp+ks from C+M, and sv+ks and sv+ax from M+M. Among them, there is not much Weighted Speedup variation for pf+bp and bp+sv; bp+hs, bp+ks and sv+ks show Weighted Speedup degradation with L1 D-cache partitioning; and only sv+atx obtains Weighted Speedup improvement.

We checked L1 D-cache efficiency and cache-miss-related resource congestion to better understand how cache partitioning affects performance. Figure 5 (b) and (c) present the L1 D-cache miss rate and rsfail rate (reservation failures per access) of individual kernels in the selected workloads. First, pf+bp and bp+sv do not show much variation in L1 D-cache miss rate and rsfail rate between WS and 'WS-L1D Partition'. Second, although bp in bp+hs, ks in bp+ks and ks in sv+ks has a lower L1 D-cache miss rate in 'WS-L1D Partition' than WS, the other kernel, namely hs in bp+hs, bp in bp+ks and sv in sv+ks, suffers from a much higher L1 D-cache rsfail rate, i.e., more reservation failures per access, because a smaller portion of L1 D-cache is assigned to it according to UCP and a cache slot needs to be allocated for an outstanding miss. The combined effect of the reduced L1 D-cache miss rate of one kernel, a higher rsfail rate of the other and high miss penalty leads to the performance degradation. Third, for sv+ax, ax obtains both lower L1 D-cache miss rate and rsfail rate, and the reduction in reservation failures of ax benefits the co-running kernel sv, resulting in improved performance of sv+ax with 'WS-L1D Partition'.

As discussed, while cache partitioning may help reduce the miss rate of a kernel in intra-SM sharing, it does not necessarily reduce the overall memory pipeline stalls. Although it is possible that the effectiveness of cache partitioning can be improved if reservation failures are taken into account, it is nontrivial to differentiate reservation failures caused by each kernel as the cache-miss-related resources are shared and the occupancy of those resource by one kernel may lead to reservation failures of another and it requires dedicated hardware designs, in addition to shadow tag arrays used to evaluate partitioning configurations [34].

3.2. BMI: Balanced Memory Request Issuing

In this section, we present the idea of balanced memory request issuing in intra-SM sharing.

As shown in Table 2, compute-intensive kernels tend to have more compute instructions per memory one, indicated by 'Cinst/Minst', than memory-intensive kernels. Also, compute-intensive kernels have fewer requests per memory instruction, indicated by 'Req/Minst'. When multiple kernels concurrently run on the same SM, they compete for the same memory pipeline. Since a memory-intensive kernel has more memory instructions in nature, it has a higher probability to access LSU (Load/Store Unit) if there is no dedicated memory instruction issuing management. When the compute-intensive kernel needs to issue a memory instruction, however, it has to wait until the LSU becomes available. Due to the high 'Req/Minst' of memory-intensive kernels, the waiting time can be quite long especially when the memory-intensive kernel has a low hit rate and/or a high

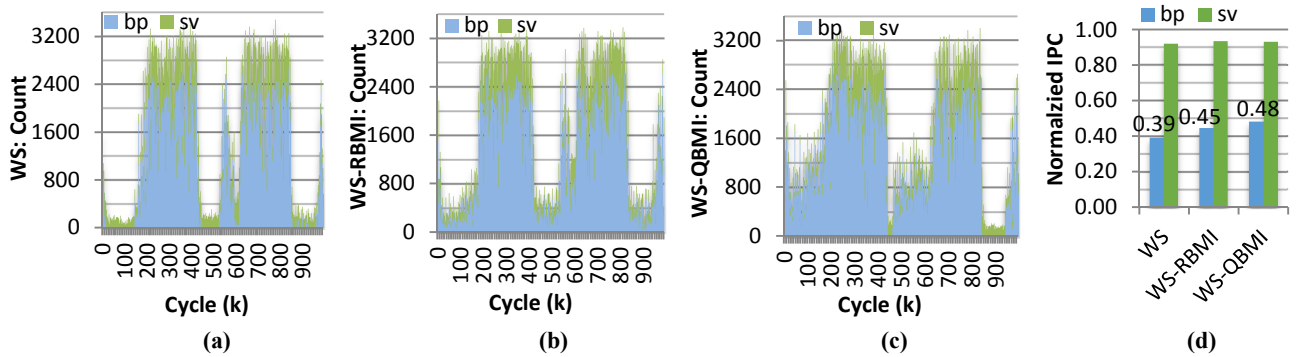


Figure 8. Warp instruction issuing: (a) WS; (b) WS-RBMI; (c) WS-QBMI; (d) Normalized IPC.

rsfail rate. Moreover, the compute-intensive kernel may lose the competition again due to the warp scheduling policy even when the LSU becomes available.

Figure 6 shows the number of L1 D-cache accesses with a sampling interval of 1K cycles for the workload bp+sv, where bp is compute-intensive and sv is memory-intensive. Figure 6 (a) and (b) show that both bp and sv have a considerable amount (around 130) of accesses every 1K cycles when either runs in isolation. However, as shown in Figure 6 (c), when they run concurrently, sv dominates L1 D-cache accesses and bp starves. Even when the underlying GTO warp scheduling policy determines that the warps of bp should be actively scheduled, like for the time windows [200K, 400K] and [600K, 800K], sv still aggressively issues requests to L1 D-cache and bp only achieves around 70 accesses per 1K cycles, much lower compared to that when it runs in isolation. Consequently, bp cannot get its memory requests accommodated timely and therefore not able to execute data-dependent computation instructions, leading to the performance degradation.

To resolve the problem that one kernel starves from failing to access the memory subsystem, we propose balanced memory request issuing (BMI). One way to implement BMI is to issue memory instructions from concurrent kernels in a loose round-robin manner and this approach is referred to as RBMI. However, since one warp memory instruction may result in multiple memory requests and different kernels show different 'Req/Minst' (as seen in Table 2), RBMI cannot ensure balanced memory accessing among concurrent kernels. To overcome this problem, we propose quota-based memory instruction issuing, named QBMI, and the memory instruction quotas of concurrent kernels are calculated with the formula below:

$$Quota_{ki} = \frac{LCM\left(\left(\frac{Req}{Minst}\right)_{k0}, \left(\frac{Req}{Minst}\right)_{k1}, \ldots, \left(\frac{Req}{Minst}\right)_{kn}\right)}{\left(\frac{Req}{Minst}\right)_{ki}}$$

where $Quota_{ki}$ is the quota for kernel i, LCM denotes Least Common Multiple and $\left(\frac{Req}{Minst}\right)_{ki}$ is the average number of requests per memory instruction for kernel i. Therefore, the higher 'Req/Minst' of a kernel, the lower quota will be assigned to it. As defined, QBMI takes multiple accesses from an atomic operation into account.

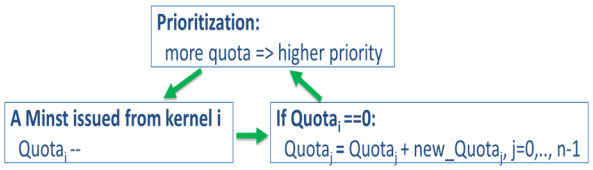
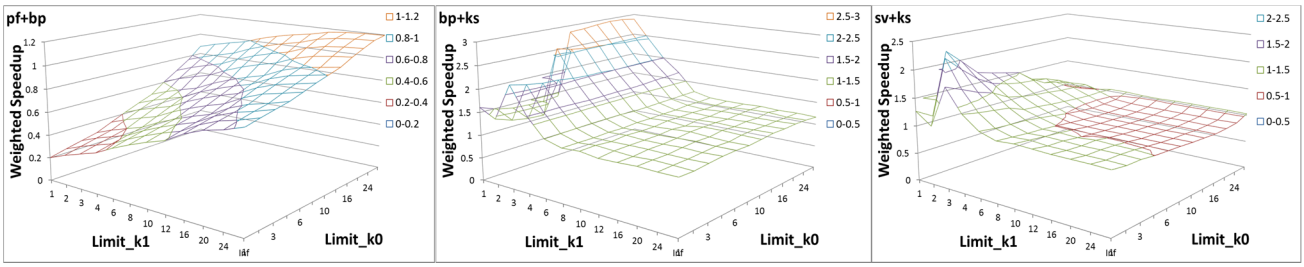


Figure 7. The Workflow of QBMI (Quota Based Memory request Issuing).

The workflow of QBMI is shown in Figure 7. The priority of a kernel to issue a memory instruction is based on its current quota and the more its quota, the higher its priority. Each time a memory instruction is issued from a kernel, its quota is decremented by 1. When the quota of any kernel reaches zero, a new set of quotas, calculated with the most recent values of 'Req/Minst', will be added to the current quota values of concurrent kernels, so as to eliminate the scenario where a kernel with zero quota cannot issue memory instructions even when there is no ready memory instruction from any other co-running kernel. Specifically, 'Req/Minst' of a kernel is updated every 1024 memory requests issued by it. The sampling interval of 1024 accesses works well as the metric 'Req/Minst' is relatively stable throughout the execution of a GPU kernel.

Figure 8 shows the number of warp instructions issued from the two kernels of the workload bp+sv, with a sampling interval of 1K cycles, where WS-RBMI in Figure 8(b) denotes TB partition with Warped-Slicer plus RBMI and WS-QBMI in Figure 8(c) is for Warped-Slicer plus QBMI. As we can see, RBMI and QBMI both enable more warp instructions to be issued from the compute-intensive kernel bp than WS. For example, compared to WS, bp issues more instructions for the time windows [200K, 400K] and [600K, 800K] under WS-RBMI and even more instructions are issued under WS-QBMI. This confirms our hypothesis: in-time memory request accommodation is critical for the execution of compute-intensive kernels in CKE. And more warp instruction issuing translates to a better performance of bp. As shown in Figure 8(d), the normalized IPC (IPC of concurrent execution to that of isolated execution) of bp



(a) pf+bp **(b) bp+ks** **(c) sv+ks**

Figure 9. Weighted Speedup with varied memory instruction limiting numbers on kernel 0 and kernel 1 for workloads from different classes: (a) C+C workload: pf+bp; (b) C+M workload: bp+ks; (c) M+M workload: sv+ks.

increases from 0.39 under WS to 0.45 under WS-RBMI and to 0.48 under WS-QBMI as more cache bandwidth is assigned to bp and in turn more compute instructions can be executed. In the meanwhile, the performance of sv remains relatively stable. Therefore, the overall performance is improved. Since QBMI can better cope with the different degrees of memory coalescing (indicated by 'Req/Minst') and shows a better performance than RBMI, we adopt QBMI in the rest of the paper.

3.3. MIL: Memory Instruction Limiting

Although QBMI is good to use when memory access imbalance (i.e., when one kernel issues too many requests) is the main issue, it does not necessarily reduce memory pipeline stalls. In this section, we highlight that the overall performance can be further improved by explicitly limiting the number of in-flight memory instructions when the memory pipeline stalls often and different kernels are in favour of different memory instruction limiting numbers.

While BMI can help achieve more balanced memory accesses and get a kernel's requests served more timely, it does not necessarily reduce memory pipeline stalls incurred by a memory-intensive kernel and the co-running kernels may still suffer from such penalties. In the meanwhile, limiting the number of in-flight memory instructions of a kernel is an effective way to reduce memory pipeline stalls and improve L1 D-cache efficiency. To exploit this factor, we propose MIL (Memory Instruction Limiting) for intra-SM sharing and we investigate two variants of MIL: SMIL (static MIL) and DMIL (dynamic MIL).

3.3.1 SMIL: Static Memory Instruction Limiting

In SMIL, we run simulations for all combinations regarding the number of in-flight memory instructions that can be issued from individual kernels in intra-SM sharing. Specifically, we vary the in-flight memory instruction limiting number on kernel 0 from 1 to 24, and symmetrically for kernel 1. The simulation point of no such a limitation (Inf) for each kernel is also examined.

We use one representative workload from each class to show how SMIL performs. Figure 9 shows the performance with varied memory instruction limiting numbers on kernel 0 and kernel 1. In the figures, the right horizontal axis (Limit_k0) denotes the memory instruction limiting number on kernel 0 while the left one (Limit_k1) is for kernel 1, and the vertical axis indicates Weighted Speedup.

Figure 9(a) shows that for the C+C workload, pf+bp, with a fixed Limit_k1, the performance increases with a larger Limit_k0 and it is similar when varying Limit_k1 with Limit_k0 fixed. Therefore, there is no need to limit the number of inflight memory instructions for co-running compute-intensive kernels. Figure 9(b) shows the case of a C+M workload, bp+ks, in which the overall performance suffers from a large number of in-flight memory instructions issued by kernel 1, ks. When Limit_k1 is at least 8, the performance is low with varied Limit_k0 and Limit_k1. When Limit_k1 is small (smaller than 8), a better performance can be achieved with a large Limit_k0, due to the improved L1 D-cache locality resulted from the combined effect of in-flight memory instruction limiting and underlying warp scheduling policy GTO. Figure 9(c) presents the case of sv+ks, an M+M workload, in which the performance remains low when Limit_k1 is at least 8. However, different from bp+ks, the overall performance first increases and then decreases with a larger Limit_k0 when Limit_k1 is small (smaller than 8). As a result, an optimal point exists in terms of the peak performance and it is (3, 1) for sv+ks, indicating the highest performance occurs when Limit_k0 is 3 and Limit_k1 is 1.

As illustrated, limiting the number of in-flight memory instructions from the memory-intensive kernel effectively improves the overall performance. The compute-intensive kernel can have a better chance to access memory subsystem and get requests served timely when memory pipeline stalls are reduced with fewer in-flight memory instructions from the co-running memory-intensive kernel. In the meanwhile, the performance of the memory-intensive kernel also increases due to the improved L1 D-cache efficiency. However, since different kernels have different memory access features, indicated by metrics like 'Req/Minst' and 'l1d miss rate' (Table 2), different workloads show different optimal limiting numbers for concurrently running kernels.

3.3.2 DMIL: Dynamic Memory Instruction Limiting

Although SMIL can effectively improve the performance, it requires re-profiling whenever there are updates in the architecture, application optimization and input size. Also, it cannot cope with application phase changing behaviours.

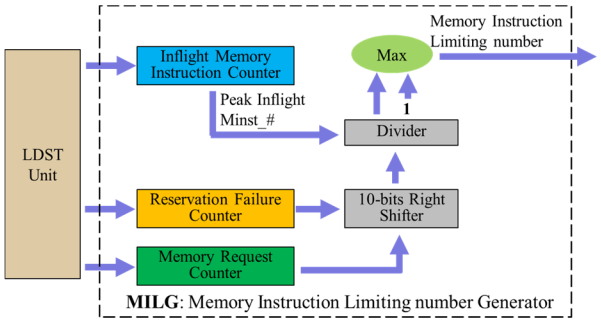


Figure 10. Organization of a Memory Instruction Limiting number Generator (MILG).

Therefore, we propose DMIL (dynamic MIL) to adapt the in-flight memory instruction limiting numbers at run-time.

As discussed in Section 2.1, cache-miss-related resources, including a cache line slot, a MSHR and miss queue entry, are allocated for an outstanding miss. If any of the required resources is unavailable, *reservation failures* occur, resulting in memory pipeline stalls. And the memory pipeline stalls incurred by one kernel will affect other concurrent kernels, further reducing computing resource utilization and degrading performance. Therefore, we use the number of *reservation failures per memory request* as the indicator to check how severe cache contention and cache-miss-related resource congestion are and use the following formula to calculate the memory instruction limiting number:

$$MIL_{ki} = Max\left(\frac{Peak\_Inflight\_Minst_{ki}}{Rev\_Fail_{ki} \gg 10}, 1\right)$$

where MIL_{ki} is the memory instruction limiting number for kernel i and it is generated with a sampling interval of every 1024 memory requests from this kernel. According to our experiments, the selected sampling interval works well in capturing the phase behaviours. $Peak\_Inflight\_Minst_{ki}$ represents the peak number of inflight memory instructions in the last sampling interval. $Rev\_Fail_{ki} \gg 10$ calculates the number of reservation failures per memory request. To avoid the scenario that a kernel is prohibited to issue memory instructions, the policy that at least one inflight memory instruction from a kernel is incorporated. Overall, the allowed number of inflight memory instructions from a kernel is reduced when there are more than one reservation failures per memory request. *The key insight is to achieve at most one reservation failure per memory request* (i.e., a fully utilized/near stall-free memory pipeline).

Figure 10 shows the organization of a memory instruction limiting number generator (MILG), which has one in-flight memory instruction counter to capture the peak number of inflight memory instruction in a sampling interval, one reservation failure counter, one memory request counter and one 10-bit right shifter which is used to calculate reservation failures per memory request.

Since each kernel has its own MILG, for our created 2-kernel workloads, there are two MILGs on each SM. It is flexible to extend to support more kernels on one SM. As

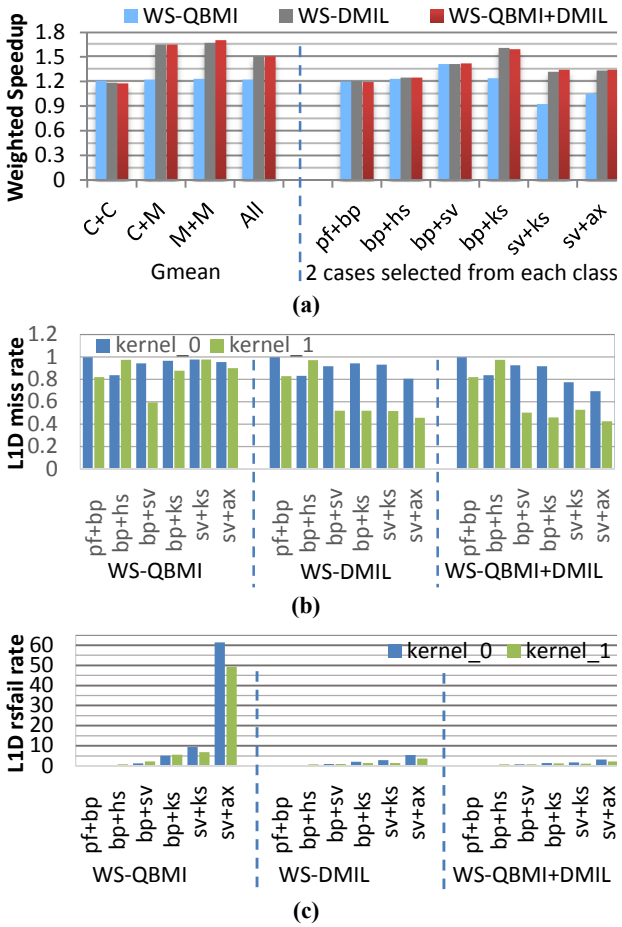


Figure 11. Performance impact of QBMI and DMIL.

there are MILGs in each SM, we refer to this design as local DMIL. Although it is possible to reduce the hardware cost by deploying global DMIL, which monitors concurrent kernel execution on one SM and broadcasts the generated results to others, global DMIL requires all SMs run the same pair of kernels. Due to the inflexibility of global DMIL, we stick to local DMIL in this study.

3.4. QBMI vs. DMIL

As discussed in Section 3.2 and 3.3, QBMI can balance memory accesses of concurrent kernels and DMIL can boost performance by reducing memory pipeline stalls incurred by memory-intensive kernels. In this part, we compare the performance impact of QBMI and DMIL, and investigate the integration of the two.

Figure 11 illustrates how QBMI and DMIL perform when Warped-Slicer is used for TB partition and QBMI+DMIL denotes the combination of the two. Figure 11(a) shows the Weighted Speedup. First, WS-QBMI and WS-DMIL achieve similar performance for C+C workloads since compute-intensive kernels have high 'Cinst/Minst' and low memory pipeline stalls. Second, on average, WS-DMIL outperforms WS-QBMI for C+M and M+M workloads with improved L1

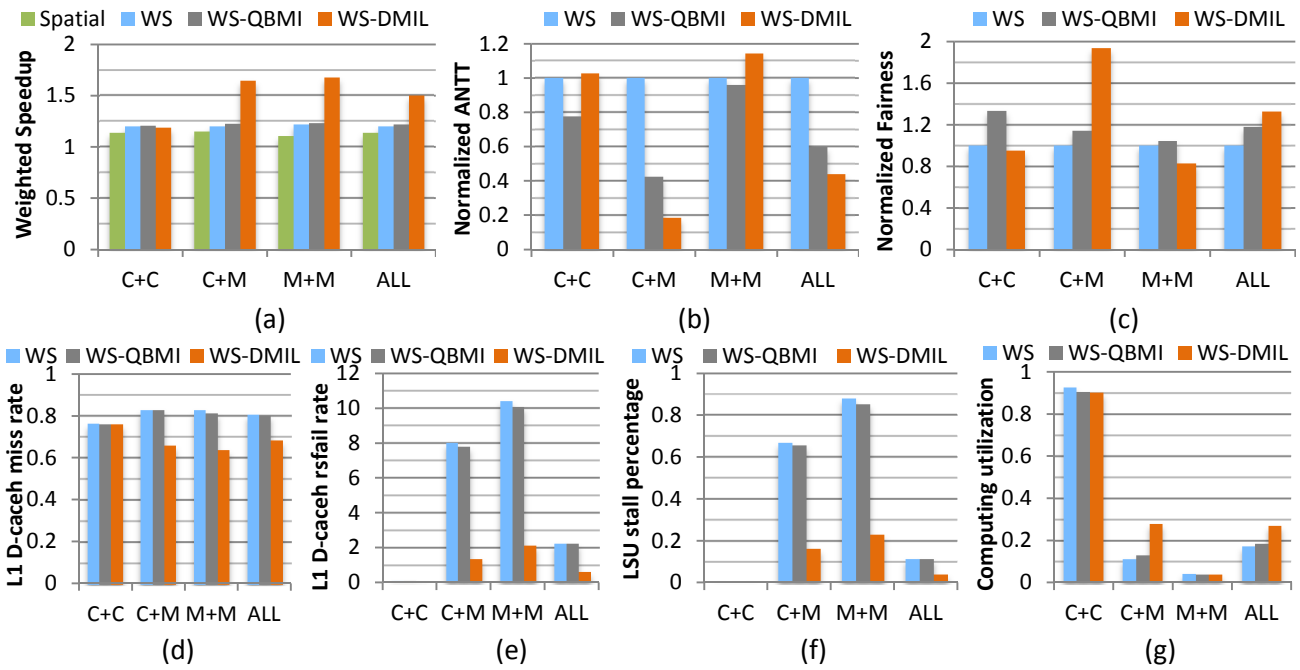


Figure 12. Effectiveness of QBMI and DMIL on top of Warped-Slicer: (a) Weighted Speedup; (b) Normalized ANNT (Average Normalized Turnaround Time); (c) Normalized Fairness; (d) L1 D-cache miss rate; (e) L1 D-cache rsfail rate (reservation failures per access); (f) percentage of LSU stall cycles; (g) computing resource utilization.

D-cache efficiency and further reduced memory pipeline stalls. For instance, compared to WS-QBMI, WS-DMIL effectively reduces L1 D-cache miss rate of ks from 0.88 to 0.52 in the C+M workload bp+ks and from 0.98 to 0.52 in the M+M workload sv+ks (Figure 11(b)). Figure 11(c) shows that WS-DMIL has a lower L1 D-cache rsfail rate, indicating fewer memory pipeline stalls.

While it is tempting to integrate QBMI and DMIL to reap the benefits of both, Figure 11 (b) and (c) show that the improvement from WS-QBMI+DMIL over WS-DMIL is minor regarding L1 D-cache efficiency and memory pipeline stalls, resulting its slightly better performance than WS-DMIL, as shown in Figure 11(a). Therefore, we report how QBMI and DMIL perform separately in the evaluation.

4. Experimental Results and Analysis

In this section, we conduct experimental analysis on our schemes and investigate how they improve the performance of the two state-of-art intra-SM sharing techniques, Warped-Slicer and SMK, both targeting TB (Thread-Block) partition, as described in Section 1.

WS: Warped-Slicer [46], which enforces TB partition using the performance scalability curves generated by dynamically profiling kernels during concurrent execution.

WS-QBMI: Our proposed quota-based balance memory request issuing (QBMI) is applied to WS.

WS-DMIL: Our proposed dynamic memory instruction limiting (DMIL) is applied to WS.

SMK-(P+W): SMK-(P+W) in work [45], which enforces TB partition based on fairness of static resources allocation and periodically allocates warp instruction quotas for concurrent kernels with profiling each one in isolation. In SMK-(P+W), a kernel will stop issuing instructions if it runs out of quota and a new set of quotas will be assigned only when quotas of all kernels equal zero. Since the warp instruction quota allocation in SMK-(P+W) and our proposed QBMI/DMIL are mutually exclusive, we apply our schemes to SMK-P and compare them with SMK-(P+W).

SMK-(P+QBMI): QBMI is applied to SMK-P.

SMK-(P+DMIL): DMIL is applied to SMK-P.

4.1. Performance Evaluation and Analysis

4.1.1 Comparison with Warped-Slicer

In this part, we illustrate how our proposed QBMI and DMIL perform when Warped-Slicer is used for TB partition.

a) Weighted Speedup, ANNT and Fairness

Figure 12(a) show the Weighted Speedup of WS, WS-QBMI and WS-DMIL, and spatial multitasking (*Spatial*) is shown as a reference. We have the following observations. First, WS performs better than Spatial on average with a better resource utilization within an SM, consistent with prior works [45][46] Second, WS, WS-QBMI and WS-DMIL have similar Weighted Speedup for C+C workloads where there are almost no memory pipeline stalls. Third, while WS-QBMI and WS-DMIL outperform WS for C+M and M+M workloads, WS-DMIL has a much higher Weighted Speedup due to further reduced memory pipeline stalls and improved L1 D-cache efficiency. On average, Weighted Speedup is 1.13 from Spatial, 1.20 from WS, 1.22 from WS-QBMI and

1.49 from WS-DMIL. Thus WS-QBMI and WS-DMIL improve the performance of WS by 1.5% and 24.6%.

In addition to Weighted Speedup, we also report ANNT (the lower the better) and Fairness (the lowest normalized IPC over the highest normalized IPC, the higher the better) [10], shown in Figure 12(b) and (c), respectively. First, WS-QBMI improves ANNT and Fairness for C+C workloads. Second, WS-QBMI and WS-DMIL greatly improve ANNT over WS for C+M workloads. Nevertheless, WS-QBMI and WS-DBMIL outperforms WS by 40.5% and 56.1% in terms of ANTT, on average. Regarding Fairness, WS-QBMI and WS-DBMIL outperforms WS by 17.8% and 32.3%.

While WS can limit the memory instructions as well by limiting the number of the thread blocks, our scheme works at a more fine-grained granularity. For instance, WS loses the memory instruction limiting capability when there is only one TB from the memory-intensive kernel (e.g., ax in sv+ax). Besides, our schemes can better cope with the phase changing behaviours of applications. Such advantages lead to better performance of our schemes.

b) L1 D-cache Miss Rate, rsfail Rate and LSU Stalls

In this section, we show that our proposed schemes achieve high L1 D-cache efficiency and significantly relieve cache-miss-related resource congestion. Figure 12 (d) and (e) show that WS suffers from a high L1 D-cache miss rate and rsfail rate in C+M and M+M workloads. WS-QBMI experiences similar L1 D-cache miss rate and rsfail rate to WS. In contrast, WS-DMIL consistently demonstrates lower L1 D-cache miss rates and fewer reservation failures per request than both WS and WS-QBMI.

c) LSU Stalls and Computing Resource Utilization

The relieved L1 D-cache miss-related resource congestion will translate to fewer memory pipeline stalls. As shown in Figure 12(f), the percentage of LSU stall cycles closely correspond to the L1 D-cache rsfail rate. And the reduced memory pipeline stalls can lead to a better computing resource utilization, especially for C+M workloads, as shown in Figure 12(g). One exceptional case is M+M workloads where WS-DMIL has a significantly lower L1 D-cache rsfail rate but it does not deliver a higher computing resource utilization. This is because essentially, both kernels of M+M workloads stress the memory pipeline and intrinsically have lower computing resource usage.

4.1.2 Comparison with SMK

Besides Warped-Slicer, we also investigate how QBMI and DMIL perform when SMK is used for TB partition. Figure 13 shows Weighted Speedup of SMK-(P+W), SMK-(P+QBMI) and SMK-(P+DMIL). We have similar observations to those when Warped-Slicer is used. First, all three schemes have similar Weighted Speedup for C+C workloads. Second, SMK-(P+QBMI) and SMK-(P+DMIL) outperform SMK-(P+W) for C+M workloads, while SMK-(P+DMIL) has a much higher Weighted Speedup. Third, SMK-(P+DMIL) remains effective for M+M workloads. On

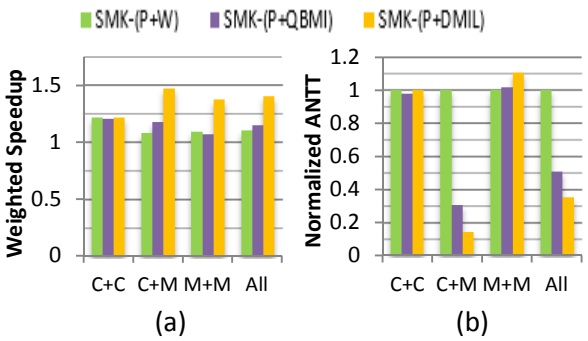
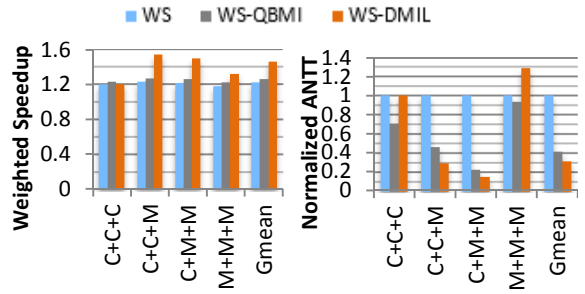


Figure 13. Effectiveness of QBMI and DMIL on top of SMK: (a) Weighted Speedup; (b) Normalized ANNT.



(a) Weighted Speedup (b) Normalized ANTT

Figure 14. Effectiveness of QBMI and DMIL in 3-kernel concurrent execution on top of Warped-Slicer.

average, Weighted Speedup is 1.10 from SMK-(P+W), 1.15 from SMK-(P+QBMI) and 1.40 from SMK-(P+DMIL). Thus SMK-(P+QBMI) and SMK-(P+DMIL) boost the Weighted Speedup of SMK-(P+W) by 4.4% and 27.2%, respectively. Although details not shown here, SMK-(P+QBMI) and SMK-(P+DMIL) outperforms SMK-(P+W) by 49.2% and 64.6% in terms of ANTT. The improvements of SMK-(P+QBMI) and SMK-(P+DMIL) are due to higher L1 D-cache efficiency, reduced LSU stalls and higher computing unit utilization.

4.2. More Kernels in Concurrent Execution

In this part, we demonstrate that our proposed schemes have good scalability and remain effective when more than two kernels concurrently on an SM. As described in Section 3.2 and 3.3.2, the proposed QBMI and DMIL are general and not restrained by the number of concurrent kernels. We evaluate all the combinations of 3-kernel workloads and have similar observations to those on 2-kernel workloads, as shown in Figure 14. First, when all kernels are compute-intensive, indicated by 'C+C+C', WS, WS-QBMI and WS-DMIL have similar Weighted Speedup while WS-QBMI improves ANTT. Second, WS-QBMI and WS-DMIL outperform WS for C+C+M and C+M+M workloads. Third, WS-DMIL continues to improve Weighted Speedup for M+M+M workloads, where all kernels are memory-intensive, but sacrifices fairness for Weighted Speedup. On average, WS-QBMI and WS-DMIL improve Weighted Speedup of WS by

3.2% and 19.4%, respectively. In terms of ANTT, WS-QBMI and WS-DMIL outperform WS by 58.3% and 68.7%.

Besides Warped-Slicer, we also investigate when SMK is used for TB partition in 3-kernel concurrent execution. SMK-(P+QBMI) and SMK-(P+DMIL) improve the average Weighted Speedup of SMK-(P+W) by 5.5% and 21.9%; for ANTT, SMK-(P+QBMI) and SMK-(P+DMIL) outperform SMK-(P+W) by 79.1% and 85.9%, respectively.

4.3. Sensitivity Study

Sensitivity to L1 D-cache Capacity: Although not shown, we examined how our schemes perform with various L1 D-cache capacities. On average, WS-QBMI outperforms WS with 2.1% (1.5%) higher Weighted Speedup and 32.1% (30.8%) better ANTT, on a 48KB (96KB) L1 D-cache; and WS-DMIL outperforms WS with 18.5% (3.5%) higher Weighted Speedup and 22.6% (10.1%) better ANTT, on a 48KB (96KB) L1 D-cache. Furthermore, while reservation failures due to MSHRs are the most common ones in our study, our schemes remain effective with increased MSHR sizes because high queuing delays in the memory subsystem can result in all MSHRs quickly being used up, similar to that when the cache capacity is enlarged to reduce memory pipeline stalls.

Sensitivity to Warp Scheduling Policy: besides the default GTO (Greedy-Then-Oldest) warp scheduling policy used in the prior experiments, we investigate how QBMI and DMIL perform when LRR (Loose Round Robin) is deployed. Our experiments show that on average, WS-QBMI and WS-DMIL boost the average Weighted Speedup of WS by 3.2% and 25.8%, respectively; in terms of ANTT, WS-QBMI and WS-DMIL outperform WS by 16.4% and 34.3%.

4.4. Hardware Overhead

As described in Section 3.3, the hardware cost for a memory instruction limiting number generator (MILG) includes one 7-bit inflight memory instruction counter (maximum 128 instructions can access L1 D-cache concurrently), one 12-bit reservation failure counter, one 10-bit memory request counter, and one 10-bit right shifter (only wires). For QBMI, one more 10-bit memory instruction counter and extra arithmetic logics are required to compute 'Req/Minst' and quotas. Although the amount of these components are proportional to the number of SMs, those overheads are negligible, compared to the area of a GPU [28][29][30][31]. Besides, the calculation and decision signal broadcasting are not on the critical path. So no extra delay is incurred.

4.5. Further Discussion

In this part, we further discuss the inadequacy to partitioning cache-miss-related resources as well as the energy efficiency and applicability of our proposed schemes.

Although it is tempting to partition cache-miss-related resources to prevent any kernel from starvation in allocating them, our experiments show that simply partitioning such resources cannot improve performance. This is because all accesses to LSU are in-order and even when a kernel's assigned portion of resources is not fully occupied, its accesses can be blocked by accesses from other co-running kernels of which the assigned resources already saturate, leading to unrelieved memory pipeline stalls.

Regarding energy efficiency, with our proposed schemes, although the average dynamic power may increase due to the improved computing resource utilization, the overall energy efficiency is improved due to much reduced leakage energy.

Although we only use the metric *reservation failures per memory request* to generate memory instruction limiting numbers, the performance results are highly promising. The underlying idea can be applied to other parts along the memory access path. For example, stalls encountered at the L1-interconnect and/or interconnect-L2 queues, can be incorporated to obtain memory instruction limiting numbers. Also, certain memory access optimization techniques, like cache bypassing, can be first applied to improve the isolated execution, and as a result the TB partition of Warped-Slicer and SMK may change. Although the effectiveness of our schemes in boosting Weighted Speedup may decrease, they shall remain effective in improving fairness, similar to the fact that QBMI enhances the fairness of C+C workloads, as shown in Section 4.1.1. Moreover, while cache bypassing relieves the contention at one level, it offloads transactions to the lower level memory hierarchies and may still stress the interconnect, L2 cache or memory, causing congestion at these parts. This would be the case especially for memory-intensive kernels, if there is no constrain on the number of their bypassed requests. We expect the underlying idea of BMI and MIL can be applied in such scenarios by monitoring/managing the memory access behaviours of different kernels at these parts along the memory access path to reduce congestion. We leave comprehensive investigation on this topic in our future work.

5. Related Work

Resource distribution and request throttling on CPUs: Several studies have addressed resource distribution among threads in simultaneous multithreading on CPUs, with a focus on cache partitioning [14][34][36][44]. Our work targets concurrent kernel execution on GPUs where there are massive memory accesses and cache partitioning is not as effective as for CPUs. Thus, we propose balanced memory request issuing (BMI) and inflight memory instruction limiting (MIL) to prevent one kernel starving from failing to access LSU and reduce LSU stalls. Besides, Ebrahimi et al [9] proposed a shared memory resource management approach, FST (Fairness via Source Throttling), to mitigate inter-core interference and improve system performance for chip-multiprocessor systems. On one hand, FST monitors unfairness and throttles down sources of interfering memory requests, similar to our proposed BMI and MIL. On the other hand, our schemes focus on intra-SM (or intra-core) interference among applications, and we directly monitor/manage memory accesses from within an SM.

Concurrent kernel execution on GPUs: Several software-centric GPU multiprogramming approaches have been studied as discussed in Section 1.

Researchers have also proposed hardware schemes to better exploit concurrent kernel execution on GPUs. Adriaens et al. [2] proposed spatial multitasking to groups SMs into different sets which can run different kernels. Ukidave et al. [42] studied runtime support for adaptive spatial partition on GPUs. Aguilera et al. [3] showed the unfairness of the spatial multitasking and proposed fair resource allocation for both performance and fairness. Tanasic el al. [41] proposed pre-emption mechanisms to allow dynamic spatial sharing of GPU cores across kernels. Gregg et al. [12] proposed a kernel scheduler to increase throughput. Wang et al. [43] proposed dynamic thread block launching to better support irregular applications. Wang et al. [45] and Xu et al.[46] addressed SM resource partition at the granularity of thread block. Park et al [33] improved GPU resource utilization through dynamic resource management but the scalability is limited. In comparison, we focus on mitigating interference among kernels and further improving resource utilization of an SM, with a good scalability to support multiple concurrent kernels.

Thread throttling and cache management on GPUs: Managing accesses to the limited memory resources has been a challenge on GPUs. Guz et al. [13] showed that a performance valley exists with increased number of threads accessing a cache. Bakhoda et al. [5] showed some applications perform better when scheduling fewer TBs. Kayıran et al. [21] and Xie et al. [47] dynamically adjust the number of TBs accessing L1 D-caches. Rogers et al. [36] proposed CCWS to control the number of warps scheduled.

On GPU cache management, Jia et al. [17] used multiple queues to preserve intra-warp locality. Kloosterman et al. [22] proposed WarpPool to exploit inter-warp locality with request queues. Detecting and protecting hot cache lines has been proposed in the work [24].

Researchers have also exploited the combination of thread throttling and cache bypassing. Li et al. [25] proposed priority-based cache allocation on top of CCWS. Chen et al. proposed CBWT [6] to adopt PDP for L1 D-cache bypassing and applies warp throttling. Li et al. [23] propose a compile-time framework for cache bypassing at the warp level.

These approaches mainly target cache locality. However, as Sethia et al. [38] and Dai et al. [8] demonstrated, cache-miss-related resource saturation can cause severe memory pipeline stalls and performance degradation. To address this issue, they proposed Mascar and MDB, respectively.

In comparison, our schemes do not throttle any TBs or warps but limit the number of in-flight memory instructions to reduce memory pipeline stalls and accelerate concurrent kernel execution with one SM shared by multiple kernels. Our approaches are complementary to cache bypassing: if not controlled, bypassing can make a memory-intensive kernel occupy even more memory resources.

6. Conclusions

In this paper, we show that the state-of-the-art intra-SM sharing techniques do not fully address the interference in CKE on GPUs. We argue that dedicated management on memory access is necessary and propose to balance memory request issuing from individual kernels and limit inflight memory instructions to mitigate memory pipeline stalls. We evaluated our proposed schemes on two intra-SM sharing schemes, Warped-Slicer and SMK. The experimental results show that our approaches improve Weighted Speedup by 24.6% and 27.2% on average over Warped-Slicer and SMK, respectively, with minor hardware cost. Our approaches also significantly improve the fairness.

Acknowledgments

We would like to thank the anonymous reviewers and the shepherd for their insightful comments to improve our paper. This work is supported by an NSF grant CCF-1618509, a Chinese research program “introducing talents of discipline to universities B13043”, and an AMD gift fund.

References

[1] AMD GCN Architecture White paper, 2012.

[2] Adriaens, Jacob T., Katherine Compton, Nam Sung Kim, and Michael J. Schulte. "The case for GPGPU spatial multitasking." In IEEE International Symposium on High-Performance Comp Architecture, pp. 1-12. IEEE, 2012.

[3] Aguilera, Paula, Katherine Morrow, and Nam Sung Kim. "Fair share: Allocation of GPU resources for both performance and fairness." In 2014 IEEE 32nd International Conference on Computer Design (ICCD), pp. 440-447. IEEE, 2014.

[4] Awatramani, Mihir, Joseph Zambreno, and Diane Rover. "Increasing GPU throughput using kernel interleaved thread block scheduling." In 2013 IEEE 31st International Conference on Computer Design (ICCD).

[5] Bakhoda, Ali, George L. Yuan, Wilson WL Fung, Henry Wong, and Tor M. Aamodt. "Analyzing CUDA workloads using a detailed GPU simulator." In Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, pp. 163-174.

[6] Chen, Xuhao, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. "Adaptive cache management for energy-efficient gpu computing." In Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 343-355. IEEE Computer Society, 2014.

[7] Che, Shuai, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. "Rodinia: A benchmark suite for heterogeneous computing." In Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, pp. 44-54. IEEE, 2009.

[8] Dai, Hongwen, S. Gupta, C. Li, C. Kartsaklis, M. Mantor, and H. Zhou. "A model-driven approach to warp/thread-block level GPU cache bypassing." InProceedings of the Design Automation Conference (DAC), Austin, TX, USA, pp. 5-9. 2016.

[9] Ebrahimi, Eiman, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. "Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems." In ACM Sigplan Notices, vol. 45, no. 3, pp. 335-346. ACM, 2010.

[10] Eyerman, Stijn, and Lieven Eeckhout. "Restating the case for weighted-ipc metrics to evaluate multiprogram workload performance." IEEE Computer Architecture Letters 13, no. 2 (2014): 93-96.

[11] Grauer-Gray, Scott, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. "Auto-tuning a high-level language targeted to GPU codes." In Innovative Parallel Computing (InPar), 2012, pp. 1-10. IEEE, 2012.

[12] Gregg, Chris, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. "Fine-grained resource sharing for concurrent GPGPU kernels." Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism. 2012.

[13] Guz, Zvika, Evgeny Bolotin, Idit Keidar, Avinoam Kolodny, Avi Mendelson, and Uri C. Weiser. "Many-core vs. many-thread machines: Stay away from the valley." IEEE Computer Architecture Letters 8, no. 1 (2009): 25-28.
[14] Herdrich, Andrew, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. "Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family." In 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 657-668. IEEE, 2016.
[15] Jiao, Qing, Mian Lu, Huynh Phung Huynh, and Tulika Mitra. "Improving GPGPU energy-efficiency through concurrent kernel execution and DVFS." In Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, pp. 1-11. IEEE Computer Society, 2015.
[16] Jia, Wenhao, Kelly A. Shaw, and Margaret Martonosi. "MRPB: Memory request prioritization for massively parallel processors." In 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), pp. 272-283. IEEE, 2014.
[17] Jog, Adwait, Evgeny Bolotin, Zvika Guz, Mike Parker, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. "Application-aware memory system for fair and efficient execution of concurrent GPGPU applications." InProceedings of workshop on general purpose processing using GPUs, p. 1. ACM, 2014.
[18] Jog, Adwait, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Niladrish Chatterjee, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. "Anatomy of GPU Memory System for Multi-Application Execution." In Proceedings of the 2015 International Symposium on Memory Systems, pp. 223-234. ACM, 2015.
[19] Jones, Stephen. "Introduction to dynamic parallelism." In GPU Technology Conference Presentation S, vol. 338, p. 2012. 2012.
[20] Justin Luitjens. “Cuda Streams: Best Practices and Common Pitfalls”, GPU Techonology Conference, 2015.
[21] Kayıran, Onur, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. "Neither more nor less: optimizing thread-level parallelism for GPGPUs." InProceedings of the 22nd international conference on Parallel architectures and compilation techniques, pp. 157-166. IEEE Press, 2013.
[22] Kloosterman, John, Jonathan Beaumont, Mick Wollman, Ankit Sethia, Ron Dreslinski, Trevor Mudge, and Scott Mahlke. "WarpPool: sharing requests with inter-warp coalescing for throughput processors." In Proceedings of the 48th International Symposium on Microarchitecture, pp. 433-444. ACM, 2015.
[23] Li, Ang, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. "Adaptive and transparent cache bypassing for GPUs." In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 17. ACM, 2015.
[24] Li, Chao, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. "Locality-driven dynamic GPU cache bypassing." In Proceedings of the 29th ACM on International Conference on Supercomputing, pp. 67-77. ACM, 2015.
[25] Li, Dong, Minsoo Rhu, Daniel R. Johnson, Mike O'Connor, Mattan Erez, Doug Burger, Donald S. Fussell, and Stephen W. Redder. "Priority-based cache allocation in throughput processors." In 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pp. 89-100. IEEE, 2015.
[26] Liang, Yun, Huynh Phung Huynh, Kyle Rupnow, Rick Siow Mong Goh, and Deming Chen. "Efficient gpu spatial-temporal multitasking." IEEE Transactions on Parallel and Distributed Systems 26, no. 3 (2015): 748-760.
[27] “NVIDIA CUDA compute unified device architecture -programming guide.” https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2008.
[28] “Whitepaper: NVIDIA’s Next Generation CUDA TM Compute Architecture: Fermi TM ,” tech. rep., NVIDIA, 2009.
[29] “Whitepaper: NVIDIA’s Next Generation CUDA TM Compute Architecture: Kepler TM GK110,” tech. rep., NVIDIA, 2012.
[30] “Whitepaper: NVIDIA GeForce GTX980,” tech. rep., NVIDIA, 2014.
[31] “Whitepaper: NVIDIA GeForce GTX1080,” tech. rep., NVIDIA, 2016.
[32] Pai, Sreepathi, Matthew J. Thazhuthaveetil, and Ramaswamy Govindarajan. "Improving GPGPU concurrency with elastic kernels." ACM SIGPLAN Notices 48, no. 4 (2013): 407-418.
[33] Park, Jason Jong Kyu, Yongjun Park, and Scott Mahlke. "Dynamic Resource Management for Efficient Utilization of Multitasking GPUs." In Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, 2017.
[34] Qureshi, Moinuddin K., and Yale N. Patt. "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches." In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 423-432. IEEE Computer Society, 2006.
[35] Rogers, Phil, Ben Sander, Yeh-Ching Chung, B. R. Gaster, H. Persson, and W-M. W. Hwu. "Heterogeneous system architecture (hsa): Architecture and algorithms tutorial." In Proceedings of the 41st Annual International Symposium on Computer Architecture.
[36] Rogers, Timothy G., Mike O'Connor, and Tor M. Aamodt. "Cache-conscious wavefront scheduling." In Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 72-83. IEEE Computer Society, 2012.
[37] Schulte, Michael J., Mike IgnatoWSki, Gabriel H. Loh, Bradford M. Beckmann, William C. Brantley, Sudhanva Gurumurthi, Nuwan Jayasena, Indrani Paul, Steven K. Reinhardt, and Gregory Rodgers. "Achieving exascale capabilities through heterogeneous computing." IEEE Micro 35, no. 4 (2015): 26-36.
[38] Sethia, Ankit, D. Anoushe Jamshidi, and Scott Mahlke. "Mascar: Speeding up gpu warps by reducing memory pitstops." In 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pp. 174-185. IEEE, 2015.
[39] Sethia, Ankit, and Scott Mahlke. "Equalizer: Dynamic tuning of gpu resources for efficient execution." In Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 647-658. IEEE Computer Society, 2014.
[40] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, Li Wen Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. IMPACT Technical Report, 2012.
[41] Tanasic, Ivan, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. "Enabling preemptive multiprogramming on GPUs." In ACM SIGARCH Computer Architecture NeWS, vol. 42, no. 3, pp. 193-204. IEEE Press, 2014.
[42] Ukidave, Yash, Charu Kalra, David Kaeli, Perhaad Mistry, and Dana Schaa. "Runtime support for adaptive spatial partitioning and inter-kernel communication on gpus." In Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on, pp. 168-175. IEEE, 2014.
[43] Wang, Jin, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. "Dynamic thread block launch: a lightweight execution mechanism to support irregular applications on GPUs." In ACM SIGARCH Computer Architecture NeWS, vol. 43, no. 3, pp. 528-540. ACM, 2015.
[44] Wang, Ruisheng, and Lizhong Chen. "Futility scaling: High-associativity cache partitioning." In 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 356-367. IEEE, 2014.
[45] Wang, Zhenning, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. "Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing." In 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 358-369. IEEE, 2016.
[46] Xu, Qiumin, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. "Warped-slicer: efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming." In Proceedings of the 43rd International Symposium on Computer Architecture, pp. 230-242. IEEE Press, 2016.
[47] Xie, Xiaolong, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. "Coordinated static and dynamic cache bypassing for GPUs." In 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pp. 76-88. IEEE, 2015.
[48] Zhong, Jianlong, and Bingsheng He. "Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling." IEEE Transactions on Parallel and Distributed Systems 25, no. 6 (2014): 1522-1532.