Efficient Kernel Management on GPUs

Xiuhong Li and Yun Liang
Center for Energy-Efficient Computing and Applications
School of EECS, Peking University, China
{lixiuhong,ericlyun}@pku.edu.cn

***Abstract*—As the complexity of applications continues to grow, each new generation of GPUs has been equipped with advanced architectural features and more resources to sustain its performance acceleration capability. Recent GPUs have been featured with concurrent kernel execution, which is designed to improve the resource utilization by executing multiple kernels simultaneously. However, prior systems only achieve limited performance improvement as they do not optimize the thread-level parallelism (TLP) and model the resource contention for the concurrently executing kernels. In this paper, we design a framework that optimizes the performance and energy-efficiency for multiple kernel execution on GPUs. It employs two key techniques. First, we develop an algorithm to adjust the TLP for the concurrently executing kernels. Second, we employ cache bypassing to mitigate the cache contention. Experiments indicate that our framework can improve performance by 1.42X on average (energy-efficiency by 1.33X on average), compared with the default concurrent kernel execution framework.**

I. Introduction

Heterogeneous system that couples CPUs and GPUs together is the growing trend for energy-efficient computing. For example, the system-on-chip (SoC) used by NVIDIA's Tegra, Qualcomm's Snapdragon, and Samsung's Exynos series all integrate GPUs with CPUs. Each new generation of GPUs ushers in new architectural features and more computing resources to sustain its leading role in high performance and energy-efficient computing. As GPUs have become increasingly powerful, more and more general purpose applications especially those with unstructured design and irregular behaviors are enabled for them. However, the diversity in program behaviors of these general applications presents challenges in designing GPU architecture.

The hardware resources on GPUs include (i) registers (ii) shared memory (iii) threads and thread blocks. A GPU kernel will launch as many thread blocks concurrently as possible until one or more dimension of resources are exhausted. However, we observe that general purpose kernels tend to exhibit great diversity in resource utilization as shown in Table II. Different kernels may exhaust different resources, leaving different resources under-utilized. Therefore, concurrent kernel execution with complementary resource usage has the potential to improve the resource utilization and performance.

More recently, GPU vendors have enabled concurrent kernel execution to improve the resource utilization. For example, NVIDIA Fermi architecture supports concurrent kernel execution from the same application; NVIDIA Kepler architecture improves Fermi by introducing the Hyper-Q feature, which maintains multiple independent kernel queues to concurrently execute independently kernels. Recent studies have also demonstrated that certain applications can benefit from concurrent kernel execution [1]. Nevertheless, several problems still remain. First, none of the present systems and prior techniques have the flexility to optimize the TLP for the concurrently executing kernel. Second, prior studies primarily focus on resource utilization, but ignore the resource contention [2]. Recent studies also show that running with the maximum number of threads does not always give the best performance due to the contention in caches, network, etc., and subsequent pipeline and memory stall [3]. Therefore, concurrent kernel execution has to strike the right balance between the resource utilization and contention.

In this work, we propose an optimization framework that manages the multiple kernel execution on GPUs. The framework involves in two key techniques. First, we design a TLP modulation algorithm that adjusts the TLP for the concurrently executing kernel. We define the TLP as the number of simultaneously executing thread blocks. Second, we develop a cache bypassing technique that can adaptively adjust the number of thread blocks that use the cache to alleviate the cache contention. We conduct a systematic evaluation using 25 two-kernel workloads. Experiments indicate that our framework can achieve 1.42X performance speedup and 1.33X energy-efficiency improvement, compared with the default concurrent kernel execution framework.

II. Background and Motivation

A. Baseline GPU Architecture

We use the architecture in the GPGPU-Sim [4] (version 3.2.2) shown in Figure 1. It is a Fermi-like architecture, where the detailed setting is shown in Table I. A GPU is composed of multiple Streaming Multiprocessors (SMs), and all the SMs share the same interconnection network. In each SM, there are many SIMD computing resources, such as streaming processors (SPs), load store units (LD/ST), and special function units (SFUs). Besides the computing resources, there are large amounts of memory resources including instruction cache, register file, L1 data cache, and shared memory, etc.

The threads of a GPU application are first organized into groups called thread blocks. When a kernel is launched, a thread block as a whole is assigned to one SM for execution. The number of threads that can simultaneously execute on one SM is limited by the available resources of an SM.

978-3-9815370-7-9/DATE16/

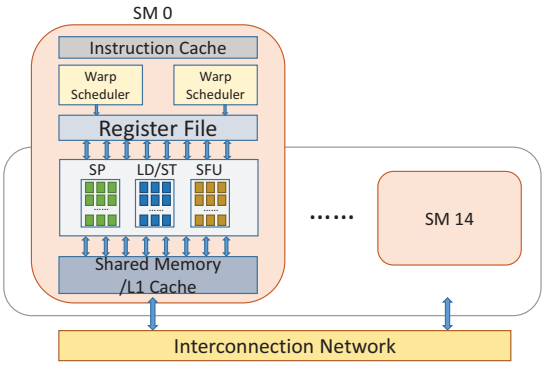


Fig. 1: GPU architecture.

TABLE I: GPGPU-Sim configuration.

# Compute Units (SM)	15
SM configuration	32 cores, 700MHz
Thread Limits per SM	1536 threads and 8 thread blocks
Register Limits per SM	32768 registers
Shared memory Limits per SM	48KB
L1 Data Cache	16KB
L2 Unified Cache	768KB
Warp Scheduler	2 warp schedulers per SM, GTO policy

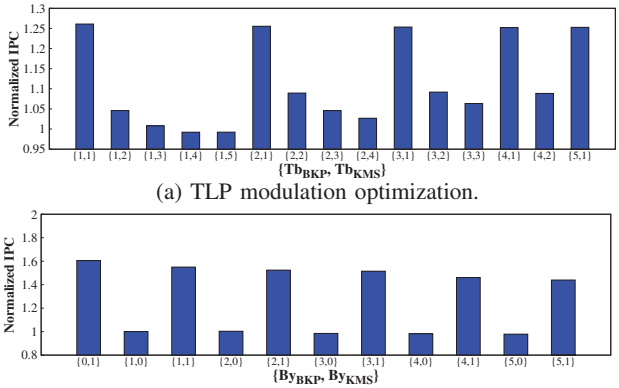
B. Concurrent Kernel Execution

The current generations of GPUs (e.g., NVIDIA Fermi, Kepler and Maxwell) support concurrent kernel execution from the same application using *stream* interface in CUDA programming model. A *stream* is a sequence of commands that execute in order. But different *streams* can execute their commands concurrently. In this work, similar to prior studies [1], [2], [5], we consider the concurrent execution of independent kernels from multi-programm workloads. In particular, we concentrate on two-kernel workloads.

C. Motivational Study

Heterogeneous kernels tend to use different resources, leaving different resources under-utilized as shown by Table II. By executing different kernels together, we have the opportunities to enable resource sharing. Though concurrent kernel execution allows multiple kernels execute simultaneously, the total number of threads/thread blocks is still bounded by the hardware limits. Thus, we first need to determine the TLP for each concurrently executing kernel. We define the TLP of two-kernel set $\{A, B\}$ as $\{Tb_A, Tb_B\}$, where Tb_A and Tb_B represents the number of thread blocks that execute concurrently for kernel *A* and *B*, respectively. Thus, there are totally $Tb_A + Tb_B$ thread blocks that execute concurrently for the two kernel set $\{A, B\}$.

We create a two-kernel set with kernels {*BKP*, *KMS*}. Figure 2a explores the design space of the TLP when executing these two kernels concurrently. The horizontal axis represents different TLP for {*BKP*, *KMS*}. There are totally 15 points in the design space. Performance is normalized to the default concurrent kernel execution mechanism. We notice that the performance depends on the TLP of the two kernels. By exploring this design space, we can improve the performance



(a) TLP modulation optimization.

(b) Cache bypassing optimization.

Fig. 2: Motivation study.

by up to 25%. We also notice that running with the maximal TLP (e.g., $\{1, 5\}$, $\{2, 4\}$, $\{3, 3\}$, $\{4, 2\}$) does not always ensure the best performance.

By exploring the TLP, we can effectively improve the performance. However, concurrency may lead to resource contention especially the L1 cache contention due to its limited size. For example, using the best TLP setting $\{5, 1\}$ in Figure 2a, we notice that the L1 data cache hit rate drops from 64.28% to 49.61%. To solve this problem, we propose cache bypassing technique for concurrent kernel execution scenario. Our cache bypassing is performed at thread block level. More clearly, we will let a subset of concurrently executing thread blocks bypass the cache for each kernel. If a thread block chooses to bypass the cache, then all the memory requests from all the threads in the thread block will bypass the L1 cache. For the two-kernel set $\{A, B\}$, we define its bypassing solution as $\{By_A, By_B\}$, where By_A and By_B represents the number of thread blocks that bypass the cache from kernel *A* and *B*, respectively. Obviously, $By_A \leq Tb_A$ and $By_B \leq Tb_B$. Figure 2b shows the results of cache bypassing optimization. In Figure 2b, the TLP is set to {5,1} for $\{Tb_{BKP}, Tb_{KMS}\}$. There are totally L1 bypassing candidates. The performance is normalized to cache-all, where none of the thread blocks bypass the cache. Through cache bypassing, we can further improve the performance by 60%.

The optimal performance results for two-kernel set {*BKP*, *KMS*} is achieved with the setting $\{Tb_A, Tb_B\}$ = $\{5, 1\}$ and $\{By_A, By_B\}$ = $\{0, 1\}$. Exploring TLP with cache bypassing optimization can be an effective strategy in improving overall performance significantly.

III. Kernel Management Framework

Our multiple kernel management framework is shown in Figure 3. It leverages on two components: *TLP modulation* and *cache bypassing*. *TLP modulation* component determines the TLP for each concurrent executing kernel. *Cache bypassing* component adjusts the number of thread blocks that bypass the cache to reduce the cache contention. Next, we will present the details of each component.

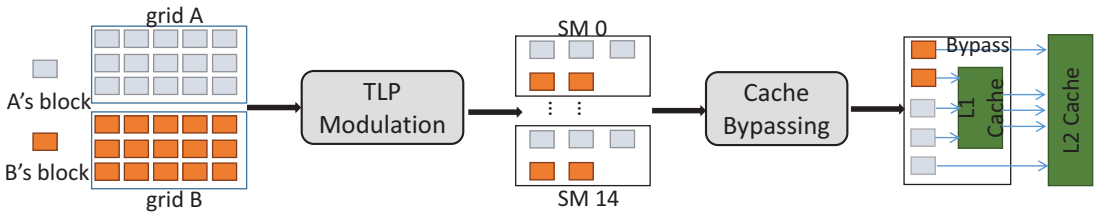


Fig. 3: Overview of our framework.

A. TLP Modulation

The massive threading is a strength of GPU but a challenge for concurrent kernel execution. Recent studies demonstrated that for a single kernel execution, running with the maximum number of threads does not always ensure the best performance due to resource contention [3], [2], [6]. In our concurrent kernel execution, multiple kernels race for the resources on the GPU, leading to high contention. Therefore, we need to determine the TLP for the concurrently executing kernels. We first characterize the TLP of single kernel and classify the kernels into different types through profiling. Then, we design a static TLP modulation algorithm based on the characterization.

1) Kernel Categorization: We propose to categorize the kernels based on how the performance varies as the TLP increases. For a single kernel, we define its TLP as the number of thread blocks that concurrently execute. The maximal TLP on our platform is 8. We categorize the kernels into three categories as follows,

- *Up.* The performance of the kernel increases as the TLP increases.
- *Optimal.* The performance of the kernel first increases then decreases as the TLP increases.
- *Down.* The performance of the kernel decreases as the TLP increases.

Figure 4a, 4b and 4c illustrate kernels in different categories. Given a kernel k, we use $opt(k)$ to represent its TLP that gives the best performance. If kernel k is type *Up*, then $opt(k)$ is the maximum TLP; if kernel k is type *Optimal*, then $opt(k)$ is somewhere between the minimal and maximum TLP; if kernel k is type *Down*, then $opt(k)$ is the minimum TLP.

Then, we analyze the *structural stall* and *memory stall* that critically influence the performance. Prior work [2] categorizes kernels in a similar way, but we provide in-depth analysis of stall and identify the relation between kernel categorization and stall. The stall will also be used in Section III-B as a metric to guide cache bypassing.

- Structural Stall. It refers to the stall caused by lack of execution units. In this case, the pipeline has to be stalled and no warps can be issued until the execution units are available. Large TLP could aggravate the contention of execution units and cause structural stall.
- Memory Stall. It refers to the stall caused by long memory latency. In this case, no warps could be issued until the data is available. The *memory stall* is due to poor data locality and read-after-write (RAW) hazards. Memory stall could be hidden by large TLP.

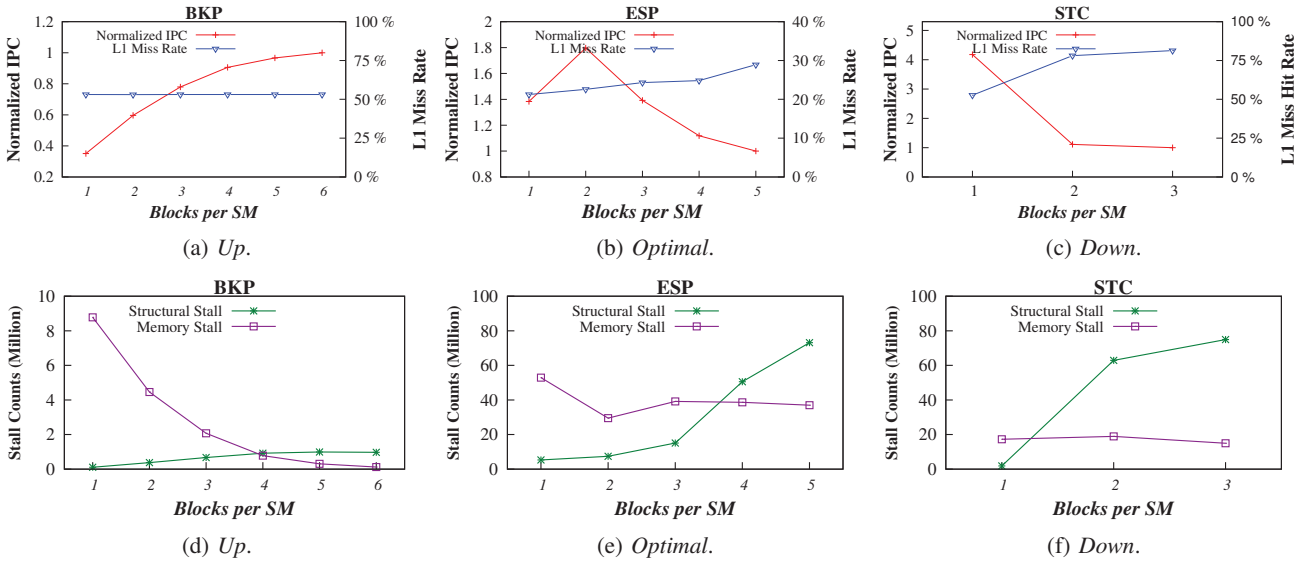
Figure 4d, 4e and 4f depict how the *structural stall* and *memory stall* vary with the TLP for different types of kernels (e.g., *BKP*, *ESP* and *STC*), respectively. For type *Up*, the *memory stall* decreases dramatically as the TLP increases, while the *structural stall* has very small variation. For type *Up* kernels, their behaviors mainly depend on *memory stall*; more TLP helps to improve the performance as it helps to hide the lengthy memory operations. For type *Optimal* and *Down* kernels, the performance does not show obvious improvement from more TLP . On the contrary, more TLP may hurt the performance as it can increase the *memory stall* and *structural stall* due to resource contention. Table II gives the type for each kernel.

2) Static TLP modulation Algorithm: For a two-kernel set $\{A, B\}$, we determine the $\{Tb_A, Tb_B\}$ based on their types. We consider all the combinations of two kernels except for both kernels are type *Up*, because in this case both two kernels require high TLP and TLP modulation has no benefits. For this case, we use the default concurrency. We combine type *Down* or *Optimal* kernel with other kernels. Because running these two types of kernels individually, the optimal block number issued to SM is less than the capacity of SM, which gives space for concurrency on the same SM.

Our algorithm attempts to determine Tb_A for type *Down* or *Optimal* kernel (A) based on optimal block number (i.e. *opt(A)*) derived from off-line profiling and then determine Tb_B for the other kernel (B) using the remaining resources. Tb_B is the maximum number of thread blocks of kernel B on an SM using the remaining resources after launching Tb_A thread blocks of kernel A. Our algorithm is symmetric. When the two kernels belong to the same type, we will arbitrarily choose one as kernel A and the other as kernel B.

Our framework can adapt to the more than two kernels scenario. First, we do off-line profiling to learn the categorization information of each kernel. Then, we select two kernels from the kernel pool using the kernel categorization information (i.e. kernel A is type *Down* or *Optimal*) and run them concurrently. After one of the two kernels finishes, we select another kernel from the kernel pool.

Although straight forward brute force search can get the optimal TLP configuration, as the number of concurrent kernels increases, the search is prohibitively time consuming. By contrast, except for the off-line profiling, the cost of our



(a) *Up*. (b) *Optimal*. (c) *Down*.

(d) *Up*. (e) *Optimal*. (f) *Down*.

Fig. 4: Kernel characterization.

algorithm does not increase with problem scale. In general, our algorithm's result is always close to the optimal TLP configuration.

B. Cache Bypassing

Concurrent kernel execution helps to improve the resource utilization, but introduces a new challenge in the form of resource contention. The primary resource contention is the L1 cache contention due to its limited size. Typically, a GPU is equipped with 16 or 32 KB cache per SM. As each SM can execute thousands of threads, this leads to only a few bytes cache capacity per thread [7], [8]. Concurrent kernel execution makes this even worse as the useful data of one kernel might be evicted from the cache by the other kernel.

We employ cache bypassing to mitigate the cache contention by selectively bypassing the cache for a portion of threads in a kernel. For two concurrently executing kernels, we propose to bypass one kernel and let the other kernel use the cache. In general, we apply cache bypassing for the kernel that is more tolerant to memory latency and let the kernel with good locality use the cache. Suppose we choose to bypass kernel *A*, then we only bypass a subset of thread blocks (By_A) for it. That is, among its concurrently executing Tb_A thread blocks, By_A thread blocks will bypass the cache and the rest $Tb_A - By_A$ thread blocks will use the cache. Our framework also has the flexibility to dynamically adjust By_A at runtime.

For a two-kernel set $\{A, B\}$, we use By_A and By_B to represent the number of thread blocks that bypass the cache for kernel A and kernel B, respectively. Since we only choose one kernel to bypass, then either By_A or By_B is 0. Then, we use $Stall^A_{By_A}$ ($Stall^B_{By_B}$) to represent the incurred stall (structural stall and memory stall) when By_A (By_B) thread blocks from kernel A (B) bypass the cache during a sampling period. Finally, we use $Stall_{none}$ to represent the stall for the case where no thread blocks bypass the cache for either kernel. We use on-line learning to determine the kernel to bypass and its number of thread blocks that bypass the cache, which consists of five steps as follows:

- *Step 1.* Initially, we set $By_A = By_B = 0$. We collect $Stall_{none}$ after a sampling period.
- *Step 2.* We tentatively choose kernel A to bypass. We set $By_A = 1$, $By_B = 0$. We collect the $Stall^A_{By_A}$ after a sampling period.
- *Step 3.* We tentatively choose kernel B to bypass. We set $By_B = 1$, $By_A = 0$. We collect the $Stall^B_{By_B}$ after a sampling period. Then we compare $Stall^A_{By_A}$, $Stall^B_{By_B}$, and $Stall_{none}$. If $Stall_{none}$ is the minimum, then we will not bypass any thread block for either kernel and return; If $Stall^A_{By_A}$ is smaller, then we will choose kernel A to bypass and set $By_A = 1$ and vice versa.
- *Step 4.* Suppose we choose kernel A as the bypassing kernel. Then, we will collect $Stall^A_{By_A+1}$ after a sampling period. If $Stall^A_{By_A+1}$ is smaller than $Stall^A_{By_A}$, then we will increment By_A by 1 and continue Step 4; otherwise By_A is not changed. Finally, if By_A reaches its upper limit Tb_A, we will stop updating By_A.

Each thread block is associated with a 1-bit tag to distinguish cache or bypass. If a thread block is tagged with 1 (0), then it will bypass (use) the cache. We use By^{cur}_A to represent the number of active thread blocks that are tagged with 1 for kernel A. Note that By^{cur}_A may be different from By_A as the thread blocks are dispatched and committed dynamically. When a thread block from kernel A is dispatched, we compare the current number of thread blocks that bypass (By^{cur}_A) with the target number (By_A). If $By^{cur}_A < By_A$, then we will tag the new thread block with 1; otherwise, we will tag it with 0. We define a sampling period as the lifetime of Tb_A thread blocks of kernel A and Tb_B thread blocks of kernel B that concurrently execute.

TABLE II: Kernel description.

Abbr.	Kernel Name	Benchmark Suite	grid size	block size	Opt.	Max.	thread utilization	shared memory utilization	register utilization	Type
BKP	bpnn_layerforward	Rodinia [9]	4096	256	6	6	100%	13.28%	75%	*Up*
HST	calculate	Rodinia	1849	256	3	3	50%	18.75%	84.38%	*Up*
SRD	extract	Rodinia	450	512	3	3	100%	0	56.25%	*Up*
SPM	spmv_jds	Parboil [10]	374	192	2	8	100%	0	75%	*Optimal*
BLK	blackschole	CUDA SDK	480	128	6	8	66.7%	0	100%	*Optimal*
LBM	performStream	Parboil	18000	120	3	6	50%	0	93.75%	*Optimal*
KMS	invert_mapping	Rodinia	1936	256	1	6	100%	0	56.25%	*Down*
STC	kernel_compute	Rodinia	128	512	1	3	100%	0	92.75%	*Down*

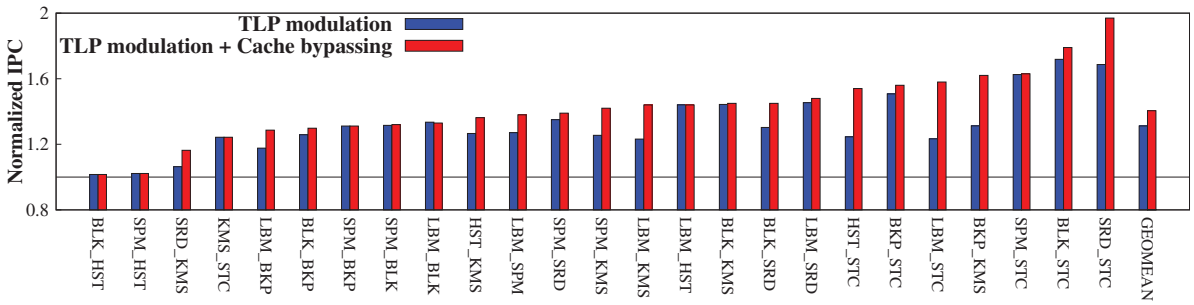


Fig. 5: Performance Results

C. Hardware Implementation Details

Our framework requires very small area for hardware implementation. The maximum number of concurrently executing thread blocks on an SM is 8. Thus, we only need two 3-bit registers for Tb_A and Tb_B. Similarly, we need four 3-bit registers (e.g., By_A, By_B, By_A^{cur}, By_B^{cur}) to count the number of thread blocks that bypass for each kernel on each SM. Finally, we need a table to keep $Stall^A_{By_A}$ and $Stall^B_{By_B}$ on each SM. Since the $By_A \leq 8$. The size of table is 16 12-bit registers totally for two kernels.

IV. Experiments

We implement our concurrent kernel execution framework based on GPGPU-sim [4] (version 3.2.2). The Fermi-like architecture setting is shown in Table II. We evaluate our technique using 8 kernels as shown in Table II. Using 8 kernels, we can create 28 two-kernel workloads. Among the 28 workloads, there are 3 workloads, where two kernels both belong to *Up* type. For these three workloads, we use default concurrency. In the following, we show results for the other 25 workloads.

A. Performance Result

Figure 5 shows the performance results for all the 25 two-kernel workloads. On average, our framework achieves 1.42X performance speedup. For workloads such as {*BLK*, *HST*} and {*SPM*,*HST*}, it shows marginal performance improvement. Because *HST* has much longer execution time than *BLK* and *SPM*. Different kernels have different execution time. The discrepancy in execution time will dilute the benefit from concurrency. The larger the discrepancy of execution time, the less the performance improvement. For example, the execution time of *HST* is 4X longer than *BLK*.

We observe that both *TLP modulation* and *cache bypassing* are important for our concurrent kernel execution. *TLP modulation* alone is an effective optimization technique for most of the workloads. The performance gain is 1.30X on average. However, for certain workload such as {*BKP*, *KMS*}, *TLP modulation* alone does not give much performance improvement due to cache contention. With the help of cache bypassing, *cache bypassing* increases the performance result from 1.25X to 1.62X for workload {*BKP*,*KMS*}.

One of the reasons that our framework improves the performance is from reduction of structural stall and memory stall.On average, it reduces memory stall and structural stall by 38.4% and 12.1%, respectively. The other reason is from alleviation of cache contention. For two-kernel workloads{*KMS*, *BKP*}, {*HST*, *KMS*}, and {*LBM*, *STC*} cache bypassing shows remarkable additional improvements. Among these workloads, kernels *STC*, *KMS*, belong to type *Down*. Type *Down* kernels are cache-sensitive and when we increase the number of blocks on each SM, the L1 data cache miss increases rapidly and the performance is decreased as shown by Figure 4. Hence, they will benefit from cache bypassing. Our cache bypassing helps to reduce L1 cache miss rate by about 30% for these three two-kernel workloads. Figure 6 shows the energy-efficiency result. Our framework improves the resource utilization by concurrently executing multiple kernels. On average, our framework can improve energy-efficiency about 1.33X.

B. Comparison with the state-of-the-art techniques

We compare our proposed framework with two the state-of-the-art concurrent kernel execution techniques.

Comparison with *Smart-Even*. Adriaens et al. [5] propose *Smart-Even* technique to support spatial multitasking by partitioning SMs to different kernels. *Smart-Even* is a coarse-grained concurrency model, which cannot improve resource utilization within an SM. Figure 7 compares the performance

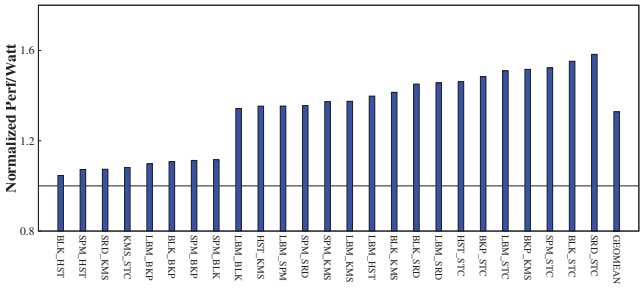


Fig. 6: Energy-efficiency result.

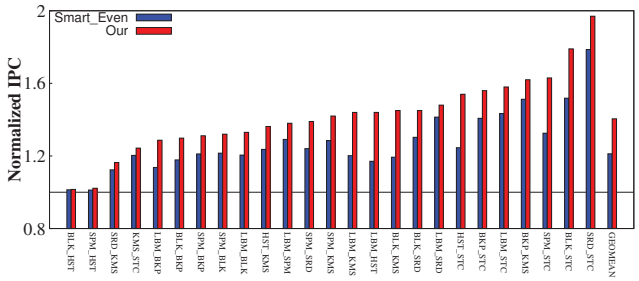


Fig. 7: Comparison results with *Smart-Even*.

of *Smart-Even* and our framework. On average, our framework achieves 1.42X speedup, while *Smart-Even* just gives 1.21X speedup.

Comparison with *mCKE*. Lee et al. [2] discuss mixed concurrent kernel execution technique to dispatches two concurrent kernels to the same SM. They only give the TLP setting for a few workloads through empirically study. For a fair comparison, Figure 8 shows the comparison results using the same workloads in [2]. On average, our framework achieves 1.38X speedup, while *mCKE* only gives 1.12X speedup. Because our framework employs TLP modulation technique to determine the TLP for each kernel. Moreover, cache bypassing technique can also alleviate resource contention incurred by concurrency.

V. Related Work

On-chip memory optimization. To better utilize computation resources on GPUs, different optimization techniques are proposed including data placement [11], register allocation [12] and cache optimization. Xie et al. [7], [8] propose a systematic framework for cache bypassing on GPUs. However, the above works are all designed for the case of single kernel.

Concurrency or multitasking. Concurrent kernel execution for GPUs becomes increasingly important. In general, there are two techniques employed on concurrency or multitasking, one is coarse-grained concurrency [5], [13], and the other is fine-grained concurrency [1], [2]. Adriaens et al. [5] first observe that many GPGPU applications fail to fully utilize all the available GPU resources and suggest to use spatial multitasking to improve resource utilization. Liang et al. [13] propose an efcient heuristic algorithm and a software emulation framework for temporal and spatial multitasking. Pai

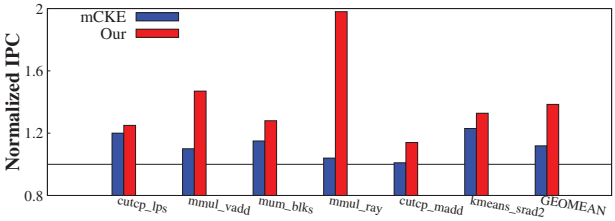


Fig. 8: Comparison results with *mCKE*.

et al. [1] identify that CUDA programs do not scale to utilize all the available resources on GPUs, where blocks from different kernels can execute in the same SM. Similarly, Lee et al. [2] propose another fine-grained concurrency called *mCKE*. However, all the above techniques ignore incurred resource contention and the energy-efficiency.

VI. Conclusion

GPUs are ubiquitous as computing platforms for high performance and energy-efficient computing. In this paper, we implement a fine-grained concurrent kernel execution mechanism, which employs a TLP modulation algorithm to determine the TLP and a cache bypassing technique to mitigate cache contention caused by concurrent kernel execution. We conduct systematic measurement of concurrent kernel execution on GPUs using representative workloads and demonstrate that concurrent kernel execution can achieve substantial performance and energy-efficiency improvement by 1.42X and 1.33X on average, respectively.

Acknowledgment

This work was supported by the National Science Foundation China (No. 61300005).

References

[1] P. Sreepathi, T. M. J., and G. R., "Improving gpgpu concurrency with elastic kernels," in ASPLOS 2013.
[2] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving gpgpu resource utilization through alternative thread block scheduling," in HPCA 2014.
[3] K. Onur, J. Adwait, K. M. Taylan, and D. C. Ranjan, "Neither more nor less: Optimizing thread-level parallelism for gpgpus," in PACT 2013.
[4] B. A., Y. G.L., F. W.W.L., W. H., and A. T.M., "Analyzing cuda workloads using a detailed gpu simulator," in ISPASS 2009.
[5] A. J. T., C. Katherine, K. N. Sung, and S. M. J., "The case for gpgpu spatial multitasking," in HPCA 2012.
[6] R. T. G., O. Mike, and A. T. M., "Cache-conscious wavefront scheduling," in MICRO-45.
[7] X. Xie, Y. Liang, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on gpus," in ICCAD 2013.
[8] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated static and dynamic cache bypassing for gpus," in HPCA 2015.
[9] "Rodinia Benchmark Suite." http://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/.
[10] "Parboil Benchmark Suite." http://impact.crhc.illinois.edu/Parboil/parboil.aspx.
[11] C. Li, Y. Yang, Z. Lin, and H. Zhou, "Automatic data placement into gpu on-chip memory resources," in CGO 2015.
[12] X. Xie, Y. Liang, X. Li, Y. Wu, G. Sun, T. Wang, and D. Fan, "Enabling coordinated register allocation and thread-level parallelism optimization for gpus," in MICRO-48.
[13] Y. Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, and D. Chen, "Efficient gpu spatial-temporal multitasking," *Parallel and Distributed Systems, IEEE Transactions on*, 2015.