Efficient Kernel Management on GPUs

YUN LIANG and XIUHONG LI, Peking University

Graphics Processing Units (GPUs) have been widely adopted as accelerators for compute-intensive applications due to its tremendous computational power and high memory bandwidth. 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, it is still a challenge to find a way to manage the resources on GPUs for concurrent kernel execution. Prior works 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 article, we design an efficient kernel management framework that optimizes the performance for concurrent kernel execution on GPUs. Our kernel management framework contains two key components: TLP modulation and cache bypassing. The TLP modulation is employed to adjust the TLP for the concurrently executing kernels. It consists of three parts: kernel categorization, static TLP modulation, and dynamic TLP modulation. The cache bypassing is proposed to mitigate the cache contention by only allowing a subset of a kernel's blocks to access the L1 data cache. Experiments indicate that our framework can improve the performance by 1.51× on average (energy-efficiency by 1.39× on average), compared with the default concurrent kernel execution framework.

CCS Concepts: • **Computer systems organization → Single instruction, multiple data**; **Multicore architectures;**

Additional Key Words and Phrases: General purpose graphics processing unit (GPGPU), energy-efficiency, kernel management

ACM Reference Format:
Yun Liang and Xiuhong Li. 2017. Efficient kernel management on GPUs. ACM Trans. Embed. Comput. Syst. 16, 4, Article 115 (May 2017), 24 pages.
DOI: http://dx.doi.org/10.1145/3070710

1. INTRODUCTION

Over the past few years, GPUs have emerged as a powerful computing platform for general-purpose computing. Each new generation of GPUs ushers in new architectural features and more computing resources to sustain its leading role in high-performance computing. As GPUs have become increasingly powerful, more and more general-purpose applications, especially those with unstructured design and irregular behaviors, are ported to GPUs for acceleration. GPUs are starting to host multiple tasks simultaneously. For example, one user may request to concurrently execute more than one task on the GPU integrated in the mobile SoC (System-on-Chip). More

This work is supported by the National Science Foundation China (No. 61672048).
Authors' addresses: Y. Liang, Office 518s, Science building No.5, Peking University, China, 100871; email: ericlyun@pku.edu.cn; X. Li, Office 512, Science building No.5, Peking University, China, 100871; email: lixiuhong@pku.edu.cn.

DOI: http://dx.doi.org/10.1145/3070710

Table I. Equipped Hardware Resources for Different NVIDIA GPU Generations

	GTX 480 (Fermi)	GTX 680 (Kepler)	GTX 980 (Maxwell)
SMs	15	8	16
SPs	32*15 = 480	192*8 = 1536	128*16 = 2048
LD/ST Units	16*15 = 240	32*8 = 256	32*16 = 512
SFUs	4*15 = 60	32*8 = 256	32*16 = 512
Threads/SM	1,536	2,048	2,048
Warps/SM	48	64	64
Thread Blocks/SM	8	16	32
32-bit Registers/SM	32,768	65,536	65,536
Shared Memory/SM	48KB	48KB	96KB

importantly, the rise of data-center and cloud-computing environments has led to an even larger scale of multitasking, where many applications from multiple users compete for access to GPU resources simultaneously. The diversity in program behaviors of these concurrently executing applications presents new challenges in kernel management to improve performance and energy efficiency on GPUs.

Using NVIDIA's terminology, a GPU is composed of multiple streaming multiprocessors (SMs). In general, each SM contains both memory and computation resources. In particular, memory resources include registers, shared memory, and the contexts for threads and thread blocks; computation resources include three types of pipeline function units: streaming processors (SPs), special functional units (SFUs), and load store units (LD/ST). Moreover, the resources are growing with each new generation for NVIDIA GPUs from Fermi and Kepler to Maxwell, as shown in Table I. However, GPU applications, especially the irregular and general-purpose applications, are often unable to effectively utilize all the resources on the GPUs [Pai et al. 2013; Fung and Aamodt 2011; Burtscher et al. 2012; Fung et al. 2007]. These applications tend to use only a portion of SMs or a portion of memory and computation resources within an SM [Pai et al. 2013; Gregg et al. 2012]. Hence, the concurrently executing of applications with different resource requirements will have the potential to improve resource utilization and energy efficiency.

Actually, 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. However, in practice, this implementation gives marginal improvement as the concurrency only happens when a task does not use all the SMs and it is about to finish. The need for multitasking support also motivates researchers to investigate new techniques. Recent proposals include coarse-grained and fine-grained multitasking. Coarse-grained multitasking [Adriaens et al. 2012] improves the SM utilization by assigning disjoint sets of SMs to different kernels. Although coarse-grained multitasking helps to improve the resource utilization, the improvement is restrictive. It only improves for the cases where the kernels lack parallelism or saturate with memory bandwidth; it does not improve the resource utilization within an SM. Fine-grained multitasking [Lee et al. 2014; Pai et al. 2013] improves the spatial multitasking by assigning heterogeneous kernels onto the same SM. However, state-of-the-art fine-grained multitasking [Lee et al. 2014] primarily focuses on the memory resources but ignores pipeline resources. Thus, several problems still remain. First, none of the present systems and prior techniques have the flexibility to optimize the thread-level parallelism (TLP) for the concurrently executing kernel. Second, prior studies primarily focus on resource utilization, but ignore the resource contention [Lee et al. 2014]. 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, for example., and subsequent pipeline and memory stall [Kayiran et al. 2013]. 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 identify that different kernels show obvious diversities in resource utilization. The variations mainly lie in two aspects. On one hand, as TLP increases, different kernels have different performance response. On the other hand, different kernels will have imbalanced requirements on computing resources and memory resources. A memory-intensive kernel may leave SP under-utilized, while a compute-intensive kernel may leave LD/ST under-utilized. Concurrent kernel execution with complementary resource usage has the potential to improve the resource utilization and thereby energy- efficiency. Although the TLP management techniques for GPUs have been proposed in the context of single-kernel execution, they do not consider the heterogeneities between different kernels. Thus we design a TLP modulation technique that adjusts the TLP for the concurrently executing kernel, based on the above observations. 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. Our work make the following contributions:

—We develop a TLP modulation technique to adjust the TLP for the concurrently executing kernel. It first employs kernel characterization. Then, based on the characterization, a static TLP modulation algorithm is conducted to determine the initial TLP configuration. During the runtime, on the basis of the initial TLP configuration, a dynamic TLP modulation algorithm will adaptively adjust TLP configuration.
—We design a cache bypassing technique to mitigate the cache contention to further improve the performance.
—Our framework is scalable, and we effectively extend it to multiple kernel scenario.

We conduct a systematic evaluation using 28 two-kernel workloads. Experiments indicate that our framework can achieve 1.51× performance speedup and 1.39× energy-efficiency improvement, compared with the default concurrent kernel execution framework.

The rest of this article is organized as follows. In Section 2, we describe the background, which concretely consists of baseline GPU architecture, concurrent kernel execution, and energy consumption analysis on GPUs. Then Section 3 presents a motivational study and characterizes the effects of TLP and cache bypassing on performance. In Section 4, we give the details of our framework. The evaluational results are presented in Section 5. Section 6 presents the related work and Section 7 concludes the article.

2. BACKGROUND

2.1. Baseline GPU Architecture

In GPU applications, the computing task that is offloaded to GPU for acceleration is written as a special function, called a *kernel*. When the kernel is launched onto GPU, a grid (an instance of the kernel) is instantiated. A grid consists of up to hundreds or thousands of threads. The threads in a grid are organized in a hierarchical manner. Every 32 threads are grouped into a *warp*, and warps are further grouped into *thread block*. The number of blocks in a grid and the number of threads in a block are specified by the programmer.

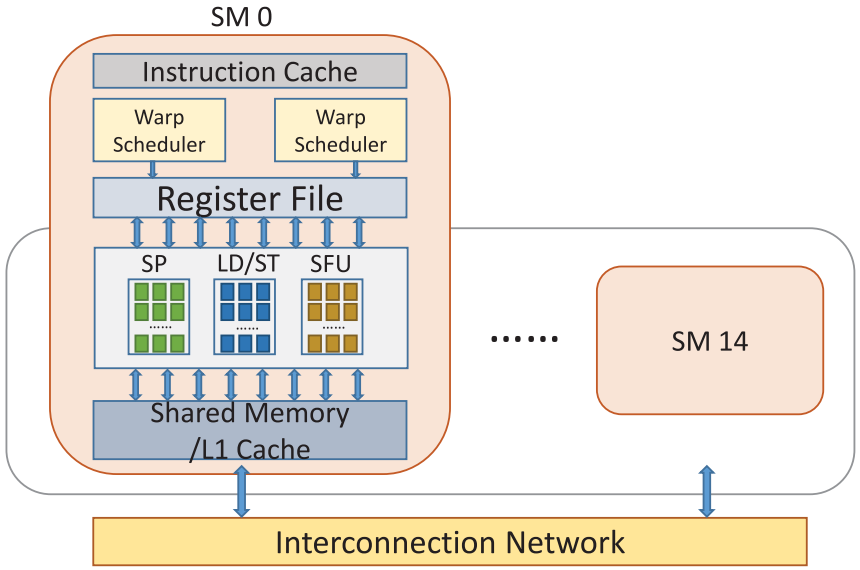


Fig. 1. Baseline GPU architecture.

Table II. GPGPU-Sim Configuration

SM Configuration	
# Compute Units (SM)	15
SM configuration	32 cores, 700MHz
Thread Limits per SM	1,536 threads and 8 thread blocks
Register Limits per SM	32,768 registers
Shared memory Limits per SM	48KB
L1 Data Cache	16KB, 32-set, 4-way, cache line (128B)
Warp Scheduler	2 warp schedulers per SM, GTO policy
Memory Subsystems Configuration	
L2 Cache	768 KB, 700 MHz, 64-set, 8-way
DRAM	924 MHz, latency (100 cycles), nbk=16:CCD=2:RRD=6:RCD=12:RAS=28 RP=12:RC=40:CL=12:WL=4:CDLR=5:WR=12:nbkgrp=4:CCDL=3:RTPL=2

We use the architecture in the GPGPU-Sim(version 3.2.2) shown in Figure 1, which is a widely adopted cycle-accurate GPGPU simulator. Its detailed setting is shown in Table II. 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, for example. When a kernel is launched, a thread block as a whole is assigned to one SM for execution. The number of thread blocks that can execute on one SM is limited by the available resources of an SM.

2.2. Concurrent Kernel Execution

The current generations of GPUs (e.g., NVIDIA Fermi and Kepler) 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* execute their commands concurrently. In this work, similarly to the prior studies [Adriaens et al. 2012; Pai et al. 2013; Lee et al. 2014], we consider the concurrent execution of independent kernels from multiprogrammed workloads. We use the *stream* interface for our concurrent kernel execution. More importantly, our framework can automatically generate *stream* interface for the concurrently executing kernels. Each *stream* corresponds to one kernel. Then, using the *stream* interface, the kernels from different *streams* are executed concurrently. In this work, we concentrate on two-kernel workloads. But our kernel management framework can be applied to more than two kernels, too.

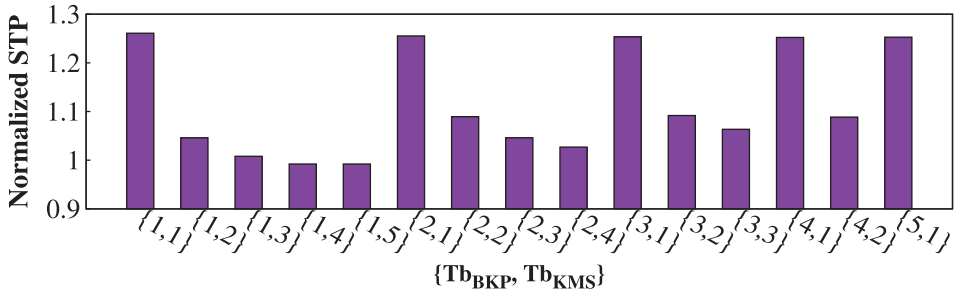
Baseline Concurrency. The concurrency supported on the current generations of GPUs is very rudimentary. The latest Kepler and Maxwell architectures feature the Hyper-Q mechanism, which allows kernels from the same process to execute concurrently. In general, the scheduler employs the *LeftOver* policy to schedule the kernels as indicated by prior study [Pai et al. 2013]. Under the *LeftOver* policy, the scheduler begins issuing thread blocks from the first kernel. When all the thread blocks of the first kernel are dispatched, if there are idle SMs, then the scheduler will issue thread blocks from the second kernel to the idle SMs. Hence, concurrent kernel execution only occurs during the period when the first kernel is about to finish and the second kernel just gets started. So, the performance improvement of the baseline concurrent kernel execution over sequential kernel execution is very minimal. Using stream software interface, programmers can push independent kernels into different streams so they can be executed concurrently.

Coarse-Grained Concurrency. Coarse-grained concurrency on GPUs was first proposed by [Adriaens et al. 2012]. Coarse-grained concurrency divides the SMs into disjoint sets and assigns to different kernels. Thus, the thread blocks from different kernels are executed concurrently on different SMs. When one of the kernels finishes execution, it releases those SMs it occupies exclusively, and then the other kernels will take over all the SMs for its remaining execution. Coarse-grained concurrency is useful for the cases where the tasks lack of parallelism or saturate with memory bandwidth. However, it does not improve the resource utilization within an SM.

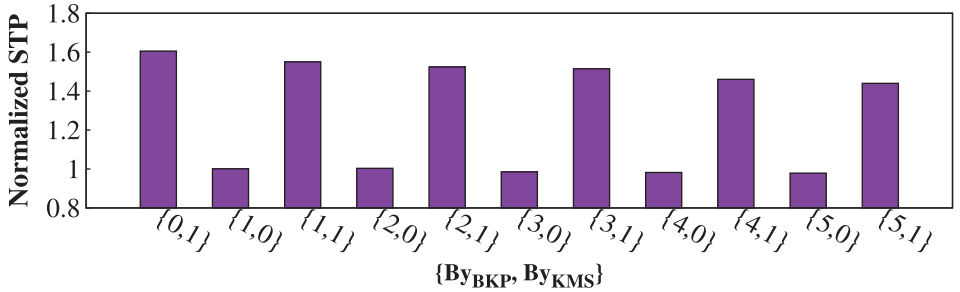
Fine-Grained Concurrency. To fully utilize the resources equipped on GPUs, we propose to use fine-grained concurrency. Fine-grained concurrency allows thread blocks from different kernels executed onto the same SMs. Compared to coarse-grained concurrency, fine-grained concurrency is a fine-grained approach. Prior studies have attempted to employ fine-grained concurrency for performance improvement [Lee et al. 2014; Pai et al. 2013; Li and Liang 2016]. However, they all ignored pipeline utilization. In this article, we use the pipeline utilization as a guide to tune the parameters of fine-grained concurrency and exploit the imbalance of pipeline utilization among heterogeneous kernels.

3. MOTIVATION

Heterogeneous kernels tend to use different resources, leaving different resources under-utilized as shown in Table III. By executing different kernels together, we have the opportunity to enable resource sharing. Though the concurrent kernel execution mechanism allows multiple kernels to execute concurrently, 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 represent 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 $\{A, B\}$.



(a) TLP modulation optimization.



(b) Cache bypassing optimization.

Fig. 2. Motivation study.

We create a two-kernel set with kernels {*BKP*, *KMS*} from the Rodinia benchmark suite [Che et al. 2009] (For details of these two kernels, see Table III). Kernel *BKP* (Back Propagation) is a machine-learning algorithm to train the weights of connecting nodes on a neural network [Che et al. 2009]. Kernel *KMS* (Kmeans) is a clustering algorithm to identify related points by associating each data point with its nearest cluster [Che et al. 2009]. Figure 2(a) 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 baseline concurrency, seen in Section 2.2. We notice that the performance depends on the TLP of the two kernels. By exploring this design space, we can improve the performance 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 2(a), 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 2(b) shows the results of cache bypassing optimization. In Figure 2(b), the TLP is set to {5,1} for $\{Tb_{BKP}, Tb_{KMS}\}$. There are totally L1 bypassing candidates. The

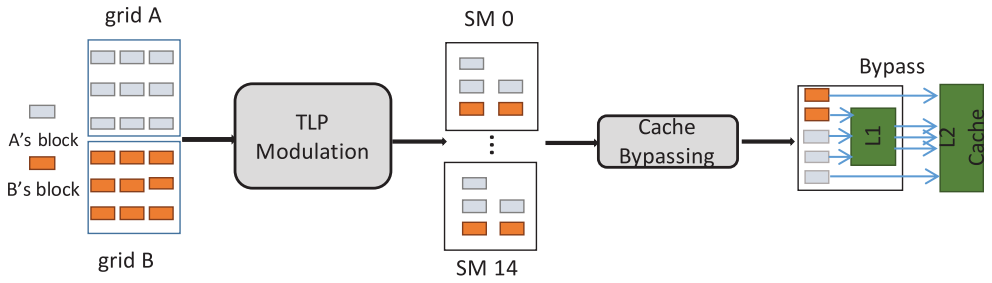


Fig. 3. Framework overview of our framework.

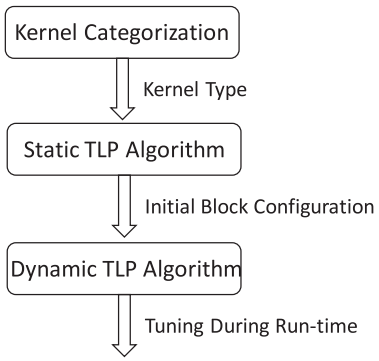


Fig. 4. Illustration of the TLP modulation process.

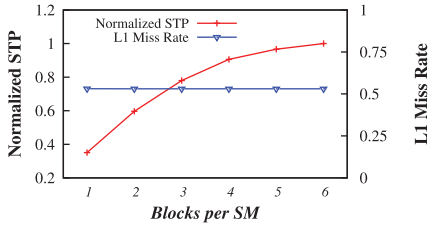
performance is normalized to cache-all, where none of the thread blocks bypass the cache. Through cache bypassing, we can further improve the performance by 60%.

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.

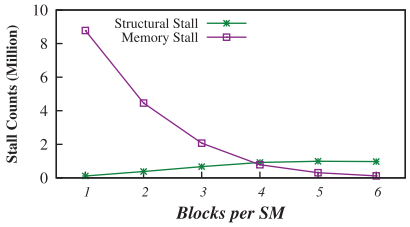
4. KERNEL MANAGEMENT FRAMEWORK

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.

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. In our concurrent kernel execution, multiple kernels race for the resources on the GPU, leading to high contention [Kayiran et al. 2013; Lee et al. 2014; Rogers et al. 2012]. Therefore, we need to determine the TLP for the concurrently executing kernels. We design our TLP modulation algorithm based on the two following observation. First, different kernels show different behaviors as the TLP increases when it executes in single-mode. Second, different kernels have different preferences on computation resources and memory resources. As shown in Figure 4, we first perform kernel categorization and classify the kernels into different types through profiling. Next, we design a static TLP modulation algorithm based on kernel

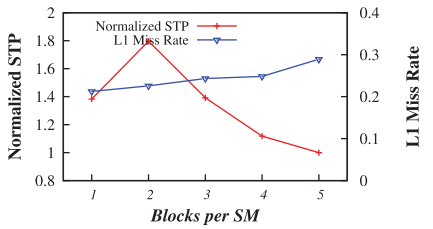


(a) System throughput and L1 data cache miss rate vary with TLP.

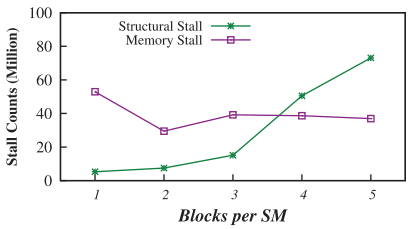


(b) Memory stall and structural stall vary with TLP.

Fig. 5. Characterization of Type Up kernel using kernel BKP as an example.



(a) System throughput and L1 data cache miss rate vary with TLP.



(b) Memory stall and structural stall vary with TLP.

Fig. 6. Characterization of Type Optimal kernel using kernel ESP as an example.

type information, which can provide an initial TLP configuration. Then, on the basis of the initial TLP configuration obtained by static TLP modulation algorithm, we employ a dynamic TLP modulation algorithm, which can tune TLP configuration at runtime according to the requirements on computation resources and memory resources.

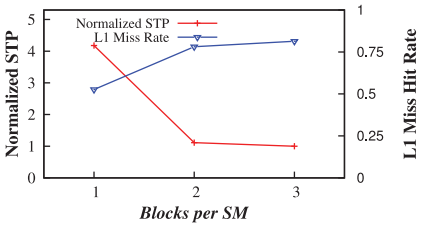
4.1. Kernel Categorization

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 16. We categorize the kernels into three categories as follows,

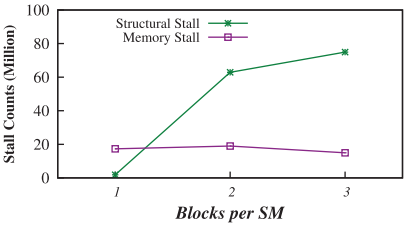
—*Up.* The performance of the kernel increases as the TLP increases.
—*Optimal.* The performance of the kernel first increases then decreases as the TLP increases.
—*Down.* The performance of the kernel decreases as the TLP increases.

Figure 5(a), Figure 6(a), and Figure 7(a) 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 [Lee et al. 2014] 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 4.4 as a metric to guide cache bypassing.



(a) System throughput and L1 data cache miss rate vary with TLP.



(b) Memory stall and structural stall vary with TLP.

Fig. 7. Characterization of Type Down kernel using kernel STC as an example.

—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 are 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 5(b), Figure 6(b), and Figure 7(b) 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 III gives the type for each kernel.

4.2. 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.

As is shown in Algorithm 1, the static TLP modulation 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 a straightforward 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 algorithm does not increase with problem scale. In general, our algorithm's result is always close to the optimal TLP configuration.

ALGORITHM 1: TLP Modulation Algorithm

```
   Input: Kernel A and Kernel B
   Output: Tb_A and Tb_B
 1 if Type_A = Down ∧ Type_B = Up then
 2 |   Tb_A = opt(A);
 3 |   r = Compute_Remain(B);
 4 |   Tb_B = r;
 5 end
 6 else if Type_A = Down ∧ Type_B = Optimal then
 7 |   Tb_A = opt(A);
 8 |   r = Compute_Remain(B);
 9 |   Tb_B = min(r, opt(B));
10 else if Type_A = Down ∧ Type_B = Down then
11 |   Tb_A = opt(A);
12 |   Tb_B = opt(B);
13 else if Type_A = Optimal ∧ Type_B = Optimal then
14 |   Tb_A = opt(A);
15 |   r = Compute_Remain(B);
16 |   Tb_B = min(r, opt(B));
17 else if Type_A = Optimal ∧ Type_B = Up then
18 |   Tb_A = opt(A);
19 |   r = Compute_Remain(B);
20 |   Tb_B = r;
21 end
```

4.3. Dynamic TLP Modulation Algorithm

The static TLP modulation algorithm gives the initial TLP configuration. The initial value obtained by the static TLP modulation can facilitate the convergence of the dynamic TLP modulation algorithm. For some certain kernels, of which the phase behavior is not obvious, the initial value can guide the dynamic TLP modulation and reduce the searching overhead thus improve the final performance. Using the initial TLP configuration, we design a dynamic TLP modulation algorithm. Based on the SIMD pipeline usage at runtime, the dynamic algorithm can adjust TLP to balance computing operation and memory access operation. It dispatches thread blocks from two kernels on one SM. In other words, each SM contains a mix of thread blocks from different kernels for concurrent execution. Obviously, Tb_A and Tb_B are limited by the resource constraints,

$$Resource_A * Tb_A + Resource_B * Tb_B \leq Resource_{SM} \tag{1}$$

where $Resource_A$ ($Resource_B$) denotes the required resource per block for kernel A (B) and $Resource_{SM}$ denotes the resource budget per SM. The total number of thread blocks depends on multiple resources (register, shared memory, threads). For each type of resource, Equation (1) has to be satisfied.

Different kernels tend to prefer different resources. As is shown in Figure 2(a), there are different ways to combine Tb_A and Tb_B, leading to different resource utilization.

The goal of block dispatcher is to determine the Tb_A and Tb_B such that the overall utilization and thus the performance can be improved. Hence, our block dispatcher is designed to adaptively adjust Tb_A and Tb_B.

To achieve this goal, our block dispatcher leverages on-line learning. It uses pipeline utilization as the performance metric to guide the learning process. The learning process consists of three steps as follows:

—*Step 1.* Get the initial value for Tb_A and Tb_B when A and B start execution.
—*Step 2.* Start the timer when there are Tb_A blocks of kernel A and Tb_B blocks of kernel B, and keep Tb_A and Tb_B unchanged for a sampling period.
—*Step 3.* Compare the performance metric of the current sampling period with that of histories and update Tb_A and Tb_B if necessary.

Initially, we get Tb_A and Tb_B from static TLP modulation algorithm. Then, we iteratively execute Steps 2 and 3 until one kernel finishes execution.

In Step 2, we start the timer when the number of thread blocks of kernels A and B equal to $\{Tb_A, Tb_B\}$. We use the lifetime of Tb_A thread blocks of kernel A and Tb_B blocks of kernel B as sampling period. Because after each sampling period, the TLP configuration $\{Tb_A, Tb_B\}$ and the cache bypassing configuration will be updated accordingly. Thus, this sampling period definition is convenient for updating and keeps harmony with our algorithms.

In Step 3, we first define the *Pipeline Utilization Change Trend (PUCT)* as the metric to predict the pipeline utilization. $PUCT_{SP}$ and $PUCT_{LDST}$ are defined as follows:

$$PUCT_{SP} = \frac{SP_{cur}}{SP_{his}}, \tag{2}$$

$$PUCT_{LDST} = \frac{LDST_{cur}}{LDST_{his}}, \tag{3}$$

where SP_{cur} and SP_{his} represent SP pipeline utilization in the current sampling period and historical sampling periods, respectively. Similarly, $LDST_{cur}$ and $LDST_{his}$ represent LD/ST unit pipeline utilization in the current sampling period and historical sampling periods, respectively. At the end of *Step 3*, we update SP_{his} using SP_{his} and SP_{cur} and $LDST_{his}$ using $LDST_{his}$ and $LDST_{cur}$.

If both $PUCT_{SP}$ and $PUCT_{LDST}$ are greater than 1, then this implies that both SP and LD/ST utilization increase in this period. In this case, we will not update $\{Tb_A, Tb_B\}$. Otherwise, we select the SIMD pipeline, which has the minimal *PUCT* value as the *Critical Pipeline*. Then, we will try to increase its pipeline utilization by updating $\{Tb_A, Tb_B\}$.

For each kernel, we characterize its pipeline preferences using Pipeline Utilization Historical Preference (*PUHP*) as follows:

$$PUHP = \frac{SP_{his}}{LDST_{his}}. \tag{4}$$

High *PUCT* implies that the kernel prefers to use SP rather than LDST and vice versa. Then, we select the kernel based on *PUCT* and update its thread block number correspondingly. For example, suppose *Critical Pipeline* is SP and kernel A has higher *PUCT* value; then we will increase Tb_A. But when we increase Tb_A, we might have to decrease Tb_B to meet Equation (1). Figure 8 gives a detailed flow of the proposed block dispatching algorithm.

Finally, when one thread block finishes, another block from the same kernel will be dispatched onto the same SM. When one of the kernels finishes execution, the other kernel will take over all the resources for its remaining execution.

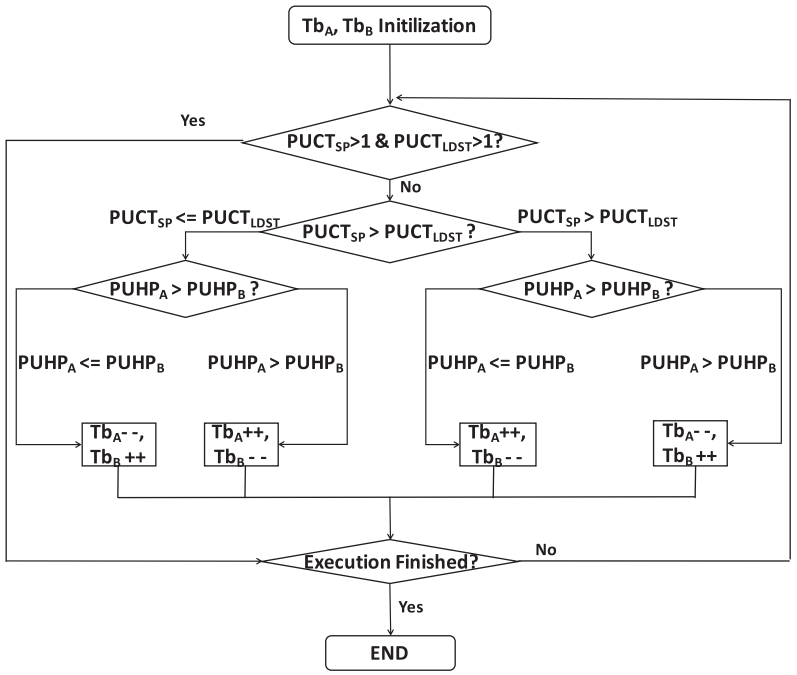


Fig. 8. Flow chart of the block dispatching algorithm.

4.4. Cache Bypassing

Concurrently kernel execution not only helps to improve the resource utilization but also enables computation and memory overlapping between kernels as described by prior subsection. However, it also presents 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 32KB cache per SM (e.g., NVIDIA Fermi and Kepler). As each SM can execute thousands of threads, this leads to only a few bytes cache capacity per thread [Xie et al. 2013, 2015b; Liang et al. 2015b]. Concurrent kernel execution makes this even worse as the useful data of one kernel might be evicted from the cache by the other kernel. The goal of cache bypassing is to mitigate the cache contention by selectively bypassing the cache requests from 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. By doing this, we not only avoid cache contention but also exploit the locality. 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 of the $Tb_A - By_A$ thread blocks will use the cache. We have the flexibility to dynamically adjust By_k at runtime. Next, we present details on how to use on-line learning to find the kernel to bypass and then adjust the number of thread blocks that bypass the cache for it.

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 (pipeline and memory) when By_A (By_B) thread blocks

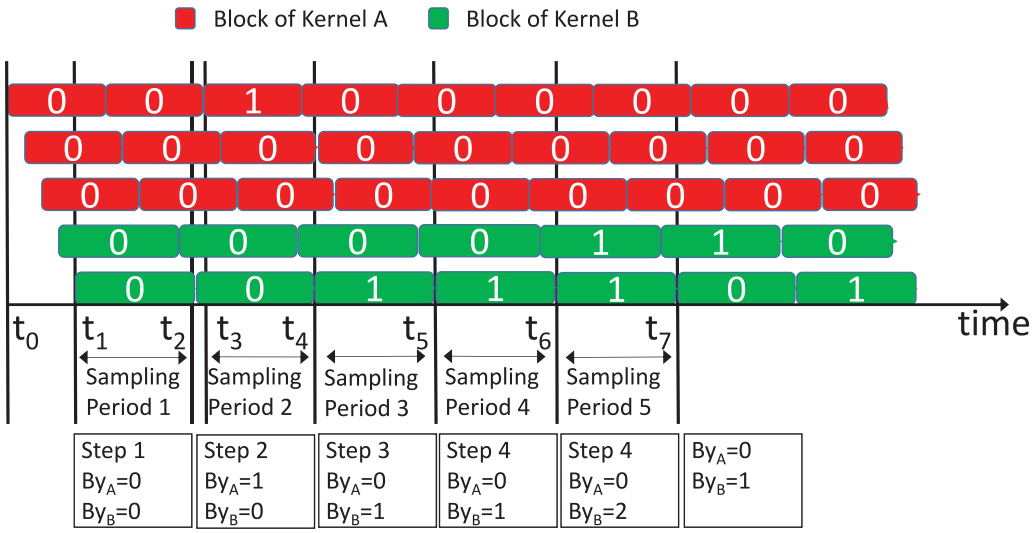


Fig. 9. Illustration of Online Learning; 0 represents using cache, and 1 represents cache bypassing.

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. The on-line learning consists of five steps as follows:

(1) Step 1. Initially, we set $By_A = By_B = 0$. We collect $Stall_{none}$ after a sampling period.
(2) 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.
(3) 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.
(4) Step 4. Suppose we choose kernel A as the bypassing kernel (decided in Step 3). 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 the number of thread blocks that bypass cache for kernel A, $By_A = By_A + 1$ and continue Step 4; otherwise, we will keep By_A thread blocks bypassed. Finally, if By_A reaches its upper limit Tb_A, then 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, then it will bypass the cache; if it is tagged with 0, then it will 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 differ 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. Figure 9 illustrates the sampling period and on-line learning process. In this example, we assume Tb_A is 3 and Tb_B is 2. We start the timer for the first sampling period at t1 as 3 blocks from A and 2 blocks

from B start concurrent execution at t1. At t2, all these five blocks finish execution and this marks the end of Step 1. When we start Step 2 at t_3, we tentatively let kernel A bypass one thread block. In the next sampling period(t_4,t_5), we let kernel B bypass one thread block. In this example, bypassing kernel *B* gives less stall. Thus, we will let kernel B bypass the cache and its bypass number is fixed to 1 after the learning.

4.5. Overhead Discussion

Actually, our algorithm only introduces marginal overhead. For the static algorithm, it only performs some comparison operations. Thus, the complexity of the static algorithm is $O(1)$. For the dynamic algorithm, it is invoked periodically and during each invoking, the algorithm performs three comparison operations and two add operation at most. Thus, the complexity of the dynamic algorithm is also $O(1)$. In terms of performance overhead, we first compare the time of the algorithm with the sampling period. A sampling period is the lifetime of Tb_A thread blocks of kernel A and Tb_B blocks of kernel B. In general, the lifetime of a thread block is in the microsecond level; however, a comparison or add operation is within the nanosecond level. So the performance overhead is negligible. Moreover, in our evaluation, we have included the overhead into the final performance (four clock cycles per single-precision arithmetic operation).

Our concurrency framework requires very small area for hardware implementation. The maximum number of concurrently executing thread blocks on an SM is 16. Thus, we need two 4-bit registers for Tb_A and Tb_B. Similarly, we need four 4-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. To support the dynamic TLP modulation algorithm, we need to record the runtime information (i.e., SP_{cur}, SP_{his}, $LDST_{cur}$, $LDST_{his}$, $PUCT_{SP}$, $PUCT_{LDST}$, $PUHP_A$, and $PUHP_B$). Finally, we need a table to keep $Stall^A_{By_A}$ and $Stall^B_{By_B}$ on each SM. We give a 32-bit register for each metric. Thus, the total size of table is 6 4-bit registers and 10 32-bit registers for two kernels.

Moreover, the runtime overhead is also negligible. For the TLP modulation algorithm and cache bypassing algorithm, the runtime overhead is only to obtain the metrics and conduct some comparisons. This overhead is very small compared to the kernel execution time. For the evaluation, we have included the overhead.

4.6. Extension to Higher Than Two-Kernel

In the above discussion, we focus on the two-kernel workloads. Actually, our framework can also be extended for higher-than two-kernel workloads. In general, we have two options to support higher-than two-kernel fine-grained kernel management. First, all of the kernels are running simultaneously within an SM. Second, each time two kernel are running simultaneously within an SM. When one kernel finishes, another pending kernel is scheduled to run with the remaining kernel. For the first option, there could be some potential issues. The first issue is the possibility of more resource contention within an SM. In practice, we find that executing more than three kernels simultaneously will aggravate resource contention such as L1 cache, resulting in worse performance. Moreover, large number of concurrently executing kernels may also aggravate the pressure of the warp scheduler. The second issue is that the block one SM can accommodate is only 16 for the mainstream GPU nowadays. The number of concurrently executing kernels is at most 16. Thus, the scalability is relatively poor.

So, we choose the second option for our extension. We only allow two kernels executing simultaneously at the same time. In this way, our TLP modulation algorithm and cache bypassing scheme can also easily be extended to highter-than two-kernel scenarios. More clearly, as shown in Figure 10, given a set of kernels stored in a pool, we can select two kernels for concurrent kernel execution first. When one of the kernel finishes, we

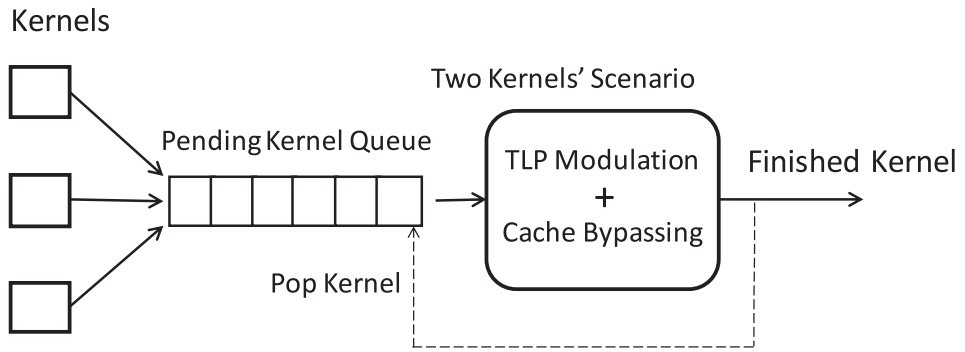


Fig. 10. Extension for higher than two-kernel scenarios.

Table III. Kernel Description

Abbr.	Kernel Name	Benchmark Suite	Opt.	Max.	Thread Utilization	Shared Memory Utilization	Register Utilization	Type
BKP	bpnn_layerforward	Rodinia	6	6	100%	13.28%	75%	*Up*
HST	calculate	Rodinia	3	3	50%	18.75%	84.38%	*Up*
SRD	extract	Rodinia	3	3	100%	0	56.25%	*Up*
SPM	spmv_jds	Parboil	2	8	25%	0	18.75%	*Optimal*
BLK	blackschole	CUDA SDK	6	8	50%	0	75%	*Optimal*
LBM	performStream	Parboil	3	6	25%	0	46.88%	*Optimal*
KMS	invert_mapping	Rodinia	1	6	16.67%	0	9.38%	*Down*
STC	kernel_compute	Rodinia	1	3	33.33%	0	31.40%	*Down*
average					50%	4%	48.25%	

will select another kernel from the pool and execute it concurrently with the left kernel. With the satisfaction of dependency, we give the kernels which have complementary resource preference higher priority. For example, if one kernel finishes and the left kernel is memory intensive, we will choose an independent compute intensive kernel from the pending kernel pool. If all of the kernels in the pending kernel pool are memory intensive, then we will choose the first one according to First Come First Service (FCFS) policy.

5. EXPERIMENTS

We implement our concurrent kernel management framework based on GPGPU-sim (version 3.2.2) [Bakhoda et al. 2009]. We conduct the simulation using the configuration in Table II. We extend the *stream* interface to support our concurrency model. We evaluate our technique using eight kernels as shown in Table III. They are from Rodinia [Che et al. 2013], Parboil [Stratton et al. 2012], and CUDA SDK benchmark suites. Using eight kernels, we can create 28 two-kernel workloads. Note that the resource utilization is obtained when they apply their optimal TLP configuration. In the following, we perform three sets of experiments to evaluate our kernel management framework. First, we show the overall performance, and break down the contributions of TLP modulation and cache bypassing. We present the experiment results from performance to the behind reasons (i.e., TLP modulation process, L1 data cache miss rate, and pipeline stall result). Then, we also show energy-efficiency results. Second, we compare our work with the state-of-the-art concurrent kernel execution frameworks. Finally, we shows the extension results for highter-than two-kernel scenarios.

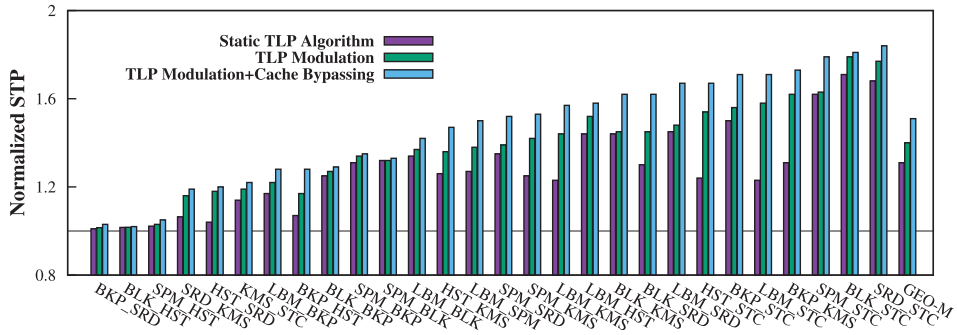


Fig. 11. Performance impact of our framework. Results are normalized to RR.

5.1. Performance Results

Figure 11 shows the performance results for all the 28 two-kernel workloads. 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 times. 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*.

Effect of Static TLP Modulation Algorithm. We discussed in Section 4.1 that different kernels show different behaviors as the TLP increases. For the kernels belonging to type *Optimal* and type *Down*, given the optimal TLP configuration *opt(k)*, they will leave the resources under-utilized. By concurrently executing kernels, the static TLP modulation algorithm helps in improving resource utilization. On average, the static TLP modulation algorithm achieves 1.31× performance speedup. Table IV shows the TLP settings selected by the static TLP modulation algorithm. By the static TLP modulation algorithm, on average, the utilization of thread, shared memory, and register is 72.63%, 5.24%, and 77.50%, respectively. By comparison, there are significant improvement than baseline concurrency (i.e., the utilization of thread, shared memory, and register is 50%, 4%, and 48.25%, respectively).

Effect of Dynamic TLP Modulation Algorithm. For two kernels both belonging to type *Up*, the static TLP modulation algorithm cannot help to concurrently executing them. Moreover, during runtime, the fixed TLP configuration cannot fully exploit the complementation of concurrently executing kernels. The dynamic TLP modulation algorithm can adjust the TLP configuration based on the runtime information. On average, coordinated static and dynamic TLP modulation can achieve 1.40× performance speedup. The primary advantage comes from the pipeline stall reduction. Our framework can reduce of structural stall and memory stall. On average, it reduces memory stall and structural stall by 38.4% and 12.1%, respectively. Figure 12 compares the structural stall and memory stall of the baseline concurrency and our framework. In Figure 12, for each workload, the left column represents the stall of baseline concurrency and the right column represents the stall of our framework normalized to the default concurrency. We find that for some workloads, such as {*HST*, *KMS*}, {*LBM*, *BKP*}, and {*BLK*, *BKP*}, our framework slightly increases the structural stall. This is because for these workloads concurrent kernel execution increases the TLP, leading to more threads racing for the execution units. But our framework still get significant stall reduction by reducing the memory stall. For workloads *LBM_BKP*, *STC_BKP*, and *BLK_STC*,

Table IV. TLP Settings Selected by the Static TLP Modulation Algorithm and Resource Utilization

Two-Kernel	Tb_A	Tb_B	Thread Util	SMem Util	Reg Util
BKP_SRD	6	0	100%	13.28%	75%
BLK_HST	6	1	66.67%	6.25%	99.46%
SPM_HST	2	3	70.83%	18.75%	92.14%
SRD_KMS	2	1	83.33%	0	46.88%
HST_SRD	3	0	50%	18.15%	84.38%
KMS_STC	1	1	50%	0	40.77%
LBM_BKP	3	4	33.59%	0	96.88%
BKP_HST	6	0	100%	13.28%	75%
BLK_BKP	6	2	83.33%	4.42%	100%
SPM_BKP	2	4	91.67%	8.85%	68.75%
SPM_BLK	2	6	75%	0	93.75%
LBM_BLK	3	4	58.33%	0	96.88%
HST_KMS	3	1	66.67%	18.75%	82.76%
LBM_SPM	3	5	87.5%	0	93.75%
SPM_SRD	2	2	91.67%	0	56.25%
SPM_KMS	2	1	41.67%	0	28.13%
LBM_KMS	3	1	41.67%	0	56.25%
LBM_HST	3	2	41.80%	12.50%	95.80%
BLK_KMS	6	1	66.67%	0	84.38%
BLK_SRD	6	1	83.33%	0	93.75%
LBM_SRD	3	2	91.67%	0	84.38%
HST_STC	2	1	66.67%	12.50%	80.32%
BKP_STC	4	1	100%	8.85%	81.40%
LBM_STC	3	1	58.33%	0	78.27%
BKP_KMS	5	1	100%	11.07%	71.88%
SPM_STC	2	1	58.33%	0	50.15%
BLK_STC	5	1	75%	0	93.90%
SRD_STC	2	1	100%	0	68.90%
average			72.63%	5.24%	77.50%

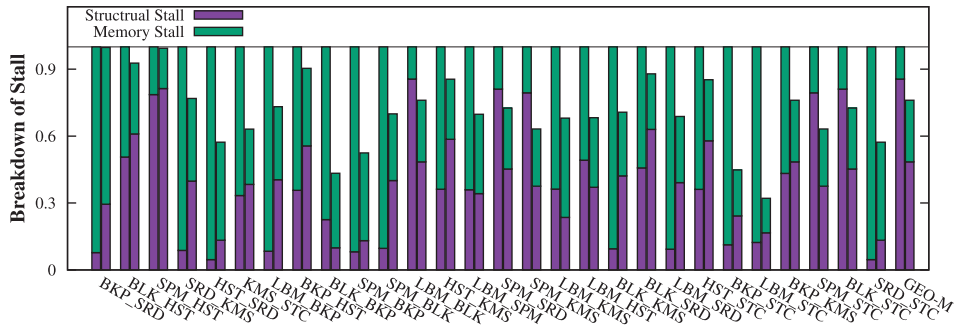


Fig. 12. Structural stall and memory stall breakdown.

our work shows very high performance speedup, because for these workloads, the two kernels have a very complementary pipeline requirement. *LBM* and *STC* have an extremely imbalanced requirement for LD/ST pipelines, while *BLK* and *BKP* prefer SP pipelines. When these kernels are dispatched to the same SM and scheduled simultaneously, they can balance the pipeline utilization and achieve significant performance improvement. To further dig out benefits of TLP modulation on performance

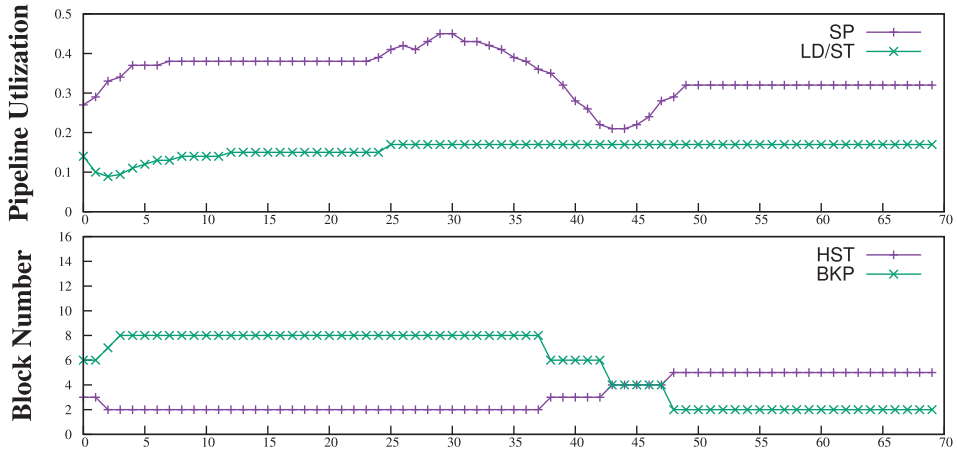


Fig. 13. An illustration of block modulation process

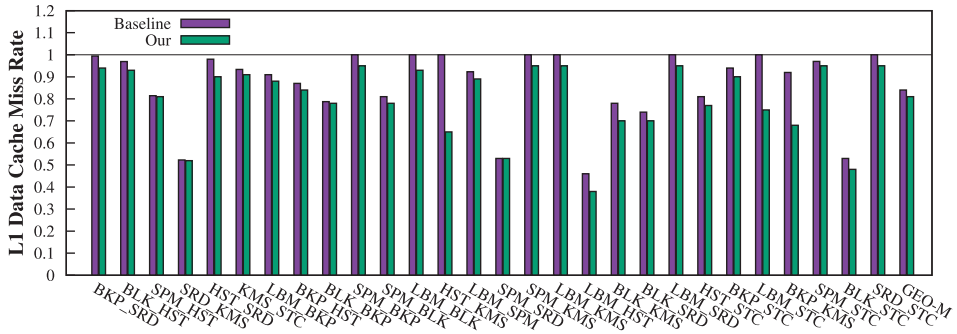


Fig. 14. L1 data cache miss rate.

benefit, we record the dynamic block dispatching process of *HST_BKP* in Figure 13. We find that in the beginning, LD/ST utilization decreases. Our block dispatcher dynamically modulates the block configuration from $\{3, 6\}$ to $\{2, 8\}$ to improve LD/ST utilization. Then, the dispatcher modulate block configuration from $\{2, 8\}$ to $\{4, 2\}$ to improve SP utilization.

Effect of Cache Bypassing. *TLP modulation* alone is an effective optimization technique for most of the workloads. However, for certain workloads, such as {*BKP*, *KMS*}, *TLP modulation* alone does not give much of a performance improvement due to cache contention. On average, by employing cache bypassing, our framework can achieve 1.51× performance speedup. For the two-kernel workloads {*BKP*, *KMS*}, {*HST*, *KMS*}, and {*LBM*, *STC*}, cache bypassing shows remarkable additional improvements. Among these workloads, kernels *STC* and *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 5(b), Figure 6(b), and Figure 7(b). Hence, they will benefit from cache bypassing. Our cache bypassing helps to reduce the L1 cache miss rate by about 30% for these three two-kernel workloads. Figure 14 shows the L1 cache miss rate. For most two-kernel workloads, our work has minor impact on the L1 cache performance. For some workloads, such as *LBM_BLK* and *STC_BKP*, the miss rate is even reduced. This is because the

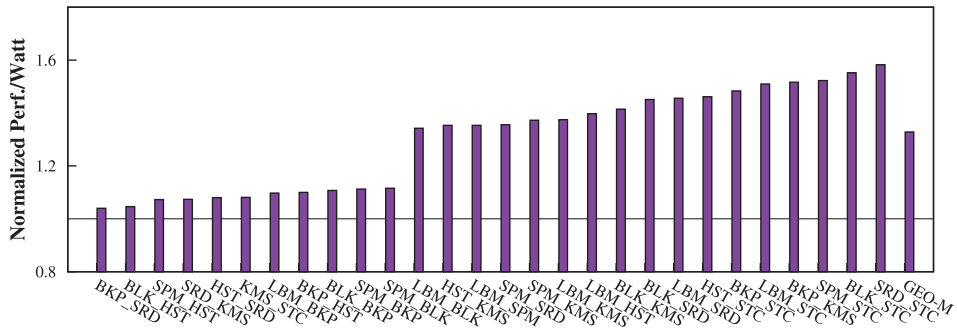


Fig. 15. Energy-efficiency result.

block dispatcher can dynamically adjust the TLP. High TLP may result in a high cache miss rate and pipeline waste. For these workloads, the block dispatcher chooses to decrease TLP to reduce the pipeline waste. As a result, cache miss is reduced.

Energy-Efficiency Result. It is demonstrated that GPU can achieve tremendous horsepower. However, this is in the cost of heavy energy consumption. We analyze the energy consumption of different kernels (in Table III) using the GPGPU Wattch [Leng et al. 2013]. SIMD pipeline and DRAM are the two main energy contributors. Moreover, heterogeneous kernels tend to use different resources, leaving different resources underutilized. By executing different kernels together, we have the opportunities to enable improve energy-efficiency. Figure 15 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 1.39×.

5.2. Comparison with the State-of-the-Art Techniques

We compare our proposed scheme with state-of-the-art GPU multitasking techniques.

Coarse-Grained Concurrency. Adriaens et al. [2012] demonstrate that for memory-intensive kernels, some SMs are idle due to off-chip memory bandwidth saturation. To fully exploit GPU resources, they propose a spatial multitasking that executes computing-intensive on idle SMs. Spatial multitasking partitions those SMs among different kernels. Thus, it is a coarse-grained concurrency mechanism, where each SM only executes one single kernel. We implement the *Smart-Even* policy proposed in [Adriaens et al. 2012]. In addition, [Wu et al. 2015] propose framework spatial multitasking and thread throttling (*SMC*). They implement both SM partition and thread block throttling on each SM. The proposed framework includes two steps. First, they evenly partition SMs to the two concurrent kernels and search for the optimal block number per SM using a hill climbing method. Second, they fix the block number on each SM and search the optimal SM partition configuration.

Figure 16 compares the performance results of *Smart-Even* and *SMC* with our our work. The performance is normalized to the baseline concurrency. On average, the performance speedups of *Smart-Even*, *SMC*, our work are 1.21×, 1.26×, and 1.51×, respectively. *Smart-Even* and *SMC* are both coarse-grained concurrency; they only consider the off-chip memory bandwidth utilization and do not consider the pipeline utilization inner SMs. In contrast, our our work is fine grained. Both *Smart-even* and *SMC* do not consider the pipeline utilization. Overall, our work outperforms those two schemes for all the two-kernel workloads. For workload {LBM, KMS}, SMC gets better performance than our framework. Because *LBM* and *KMS* are both extremely memory intensive and the LD/ST unit utilization for them is very high. In our framework, we

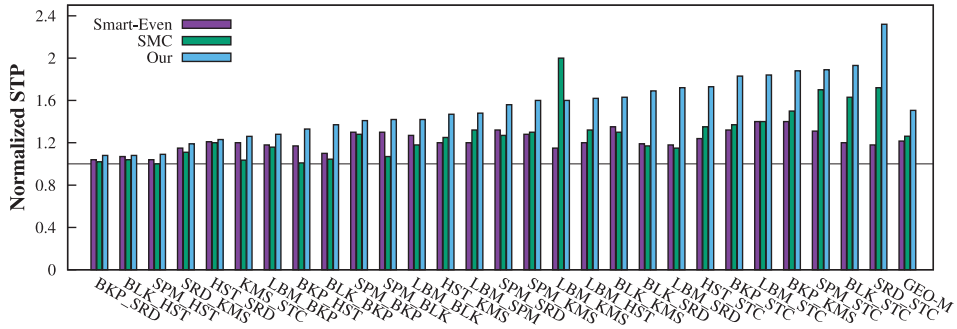
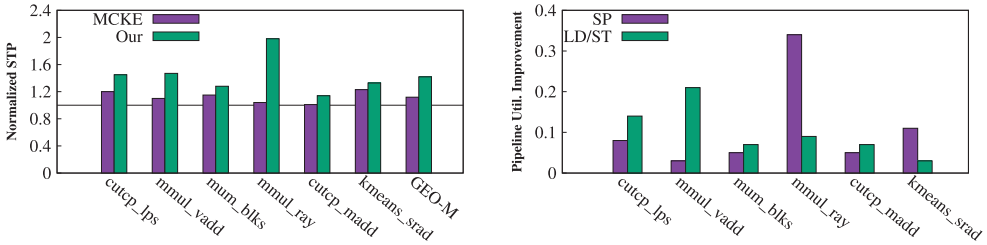


Fig. 16. Comparison with coarse-grained multitasking.



(a) Performance comparison results.

(b) Pipeline utilization improvement.

Fig. 17. Comparison with mCKE.

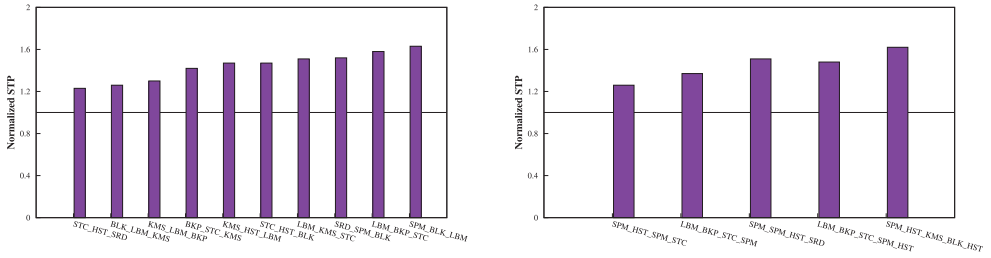
dispatch blocks from LBM and KMS into the same SM, which will introduce inherent contention on LD/ST units. Our framework can adjust the TLP configuration to alleviate the contention, but SMC only performs SM partitioning and thread throttling, which would not aggravate the LD/ST unit contention. Thus, the performance improvement of our framework is not significant. However, on average, due to the contribution of pipeline utilization analysis, our framework can outperform those coarse-grained frameworks.

Fine-Grained Concurrency. Lee et al. [2014] discuss the mixed concurrent kernel execution technique. Similarly to our our work, *mCKE* is also a fine-grained concurrency mechanism that dispatches two concurrent kernels to the same SM. *mCKE* only gives the block configuration for several workloads. Hence, we cannot compare it with our work systematically. For a fair comparison, we only our work with *mCKE* using the workloads used in Lee et al. [2014].

Figure 17(a) shows the performance result. The performance is normalized to the baseline concurrency. Our work achieves better performance for all the evaluated workloads. On average, our work achieves 1.41× speedup, while *mCKE* achieves 1.12× speedup. The reason is twofold. The first is that our our work dynamically adjusts the block number of concurrent kernels on each SM by monitoring the runtime information, while *mCKE* fixes the block number during the execution.

5.3. Evaluation of the More-Than-Two-Kernels Scenarios

In this section, we will demonstrate that our framework can well scale to the more-than-two-kernels scenarios. First, we conduct experiments for a three-kernel scenario. We create three-kernel workloads using the same methodology with two-kernel workloads. Due to space limit, we select 10 representative three-kernel workloads. Figure 18(a) shows the evaluation results. We can find that for three-kernel workloads, we can also



(a) Three-kernel scenario. (b) More than three-kernel scenarios.

Fig. 18. Evaulation of scalability.

achieve similar performance speedup. To further evaluate the scalability, we also select 4 four-kernel workloads and 2 five-kernel workloads in Figure 18(b). We achieve 1.44× speedup on average. This demonstrates that our work can scale to the more-than-two-kernels scenarios.

6. RELATED WORK

As GPUs are widely adopted as accelerators for general-purpose parallel applications, more and more optimization techniques are proposed to fully release the computing horsepower. They mainly focus on control flow divergence optimizations [Rogers et al. 2013; Cui et al. 2012; Fung and Aamodt 2011; Fung et al. 2007; Rogers et al. 2015; Burtscher et al. 2012], warp schedulers [Lee and Wu 2014; Jablin et al. 2014; Narasiman et al. 2011; Rogers et al. 2013, 2012; Jog et al. 2013a], on-chip memory optimizations [Zhang et al.; Chen et al. 2014; Jia et al.; Lee et al. 2010; Jog et al. 2013b; Gebhart et al. 2012; Xie et al. 2015b; Li et al. 2015a], and concurrent kernel executions [Liang et al. 2015a; Pai et al. 2013; Lee et al. 2014; Adriaens et al. 2012; Tanasic et al. 2014; Chen et al. 2017].

On-chip memory optimization. To better utilize computation resources on GPUs, different optimization techniques are proposed, including data placement [Li et al. 2015b], register allocation [Xie et al. 2015a; Hayes and Zhang 2014], and cache optimization [Chen et al. 2014; Xie et al. 2015b]. Xie et al. [2013] propose a systematic framework for cache bypassing on GPUs . A few studies identify that on-chip memory optimizations are also important for general-purpose applications, especially those with unstructured and irregular behaviors. Cache bypassing techniques, which selectively bypass some memory requests to help to alleviate the cache contention and further improve performance, are discussed. [Jia et al.] demonstrate that using cache does not always have a positive performance impact on GPUs. Xie et al. [2015b] propose both static and dynamic techniques to determine the cache bypassing behaviors to mitigate the cache contention problem. Chen et al. [2014] propose a coordinated scheme that not only applies cache bypassing but also throttles the number of active warps. However, they do not exploit the performance benefit of concurrent kernel executions. Moreover, our framework can work synthetically with those cache bypassing works.

Concurrency or multitasking. Concurrent kernel execution for GPUs becomes increasingly important. As the number of applications ported to GPUs continue to increase, concurrent kernel execution or multitasking support for GPUs becomes increasingly important. Coarse-grained concurrency policies are proposed in Adriaens et al. [2012], Liang et al. [2015a], [Wu et al. 2015], and they partition the SMs among kernels to enable concurrent execution. [Adriaens et al. 2012] first observe that many GPGPU applications fail to fully utilize all the available GPU resources and suggest to partition the SMs to improve off-chip memory bandwidth utilization. They discuss

several SM partitioning heuristics and evaluate their performance. [Wu et al. 2015] propose a software program transformation framework to implement SM partitioning. They adopt a hill-climbing heuristic to determine both SM partitioning configuration and thread throttling configuration. Liang et al. [2015a] propose an efficient heuristic algorithm and a software emulation framework for both temporal and spatial concurrency. However, all those techniques do not allow different kernels executing in one SM; their performance improvements come from the improved off-chip memory bandwidth utilization, and they are oblivious to the SIMD pipeline utilization inner SMs. Lee et al. [2014] propose a fine-grained concurrency policy; however, their technique is still primitive. Pai et al. [2013] identify that CUDA programs do not scale to utilize all the available resources on GPUs. To improve resource utilization, they propose kernel transformation techniques that convert CUDA kernels into elastic kernels that enable fine-grained control over their resource usage. By merge multiple kernels, they enable fine-grained concurrent kernel execution. This is a software approach, whereas our frameworks is a hardware approach. [Tanasic et al. 2014] and [Lin et al. 2016] consider preemptive multitasking on GPUs.

7. CONCLUSION

GPUs are ubiquitous as computing platforms for high-performance and energy-efficient computing. In this article, we implement a fine-grained concurrent kernel execution mechanism that employs a TLP modulation technique 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 1.51× and 1.39×, on average, respectively.

REFERENCES

Jacob T. Adriaens, Katherine Compton, Nam Sung Kim, and Michael J. Schulte. 2012. The case for GPGPU spatial multitasking. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA'12)*.

Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*.

Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A quantitative study of irregular programs on GPUs. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC'12)*.

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC'09)*.

Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. 2017. EffiSha: A software framework for enabling effficient preemptive scheduling of GPU. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'17)*.

Xuhao Chen, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. 2014. Adaptive cache management for energy-efficient GPU computing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*.

Zheng Cui, Yun Liang, K. Rupnow, and Deming Chen. 2012. An accurate GPU performance model for effective control flow divergence optimization. In *Proceedings of the 2012 IEEE 26th International Parallel Distributed Processing Symposium (IPDPS'12)*.

Wilson W. L. Fung and Tor M. Aamodt. 2011. Thread block compaction for efficient SIMT control flow. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*.

Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2007. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40)*.

Mark Gebhart, Stephen W. Keckler, Brucek Khailany, Ronny Krashinsky, and William J. Dally. 2012. Unifying primary cache, scratch, and register file memories in a throughput processor. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*.

Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. 2012. Fine-grained resource sharing for concurrent GPGPU kernels. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism (HotPar'12)*.

Ari B. Hayes and Eddy Z. Zhang. 2014. Unified on-chip memory allocation for SIMT architecture. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS'14)*.

James A. Jablin, Thomas B. Jablin, Onur Mutlu, and Maurice Herlihy. 2014. Warp-aware trace scheduling for GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*.

Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. 2012. Characterizing and improving the use of demand-fetched caches in GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*.

Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013a. OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*.

Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013b. Orchestrated scheduling and prefetching for GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*.

Onur Kayiran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. 2013. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*.

Jaekyu Lee, N. B. Lakshminarayana, Hyesoon Kim, and R. Vuduc. 2010. Many-thread aware prefetching mechanisms for GPGPU applications. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-43)*.

M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*.

Shin-Ying Lee and Carole-Jean Wu. 2014. CAWS: Criticality-aware warp scheduling for GPGPU workloads. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*.

Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling energy optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*.

Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. 2015. Locality-driven dynamic GPU cache bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS'15)*.

Chao Li, Yi Yang, Zhen Lin, and Huiyang Zhou. 2015. Automatic data placement into GPU on-chip memory resources. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'15)*.

Xiuhong Li and Yun Liang. 2016. Efficient kernel management on GPUs. In *Proceedings of Design, Automation and Test in Europe (DATE'16)*.

Yun Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, and Deming Chen. 2015a. Efficient GPU spatial-temporal multitasking. *IEEE Trans. Parallel Distrib. Syst.* 26, 3 (Mar. 2015), 748–760.

Yun Liang, Xiaolong Xie, Guangyu Sun, and Chen Deming. 2015b. An efficient framework for cache bypassing on GPUs. *IEEE Trans. Comput.-Aid. Des.* 32, 10 (October 2015), 1677–1690.

Zhen Lin, Lars Nyland, and Huiyang Zhou. 2016. Enabling efficient preemption for SIMT architectures with lightweight context switching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16)*.

Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*.

Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. 2013. Improving GPGPU concurrency with elastic kernels. *SIGPLAN Not.* 48, 4 (Mar. 2013), 407–418.

Timothy G. Rogers, Daniel R. Johnson, Mike O'Connor, and Stephen W. Keckler. 2015. A variable warp size architecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA'15)*.

Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*.

Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2013. Divergence-aware warp scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*.

John A. Stratton, Christopher Rodrigrues, I-Jui Sung, Nady Obeid, Liwen Chang, Geng Liu, and Wen-Mei W. Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Technical Report. University of Illinois at Urbana-Champaign.

Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. 2014. Enabling preemptive multiprogramming on GPUs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*.

Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. 2015. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS'15)*.

Xiaolong Xie, Yun Liang, Xiuhong Li, Yudong Wu, Guangyu Sun, Tao Wang, and Dongrui Fan. 2015a. Enabling coordinated register allocation and thread-level parallelism optimization for GPUs. In *Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-48)*.

Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. 2013. An efficient compiler framework for cache bypassing on GPUs. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'13)*.

Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. 2015b. Coordinated static and dynamic cache bypassing for GPUs. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*.

Hang Zhang, Xuhao Chen, Nong Xiao, and Fang Liu. 2016. Architecting energy-efficient STT-RAM based register file on GPGPUs via delta compression. In *Proceedings of the 53rd Annual Design Automation Conference (DAC'16)*.

Received September 2016; revised January 2017; accepted March 2017