Improving GPU Pipeline Resource Utilization through Simultaneous Multitasking

ABSTRACT

Graphics Processing Units (GPUs) are being increasingly used for high performance and energy efficient computing. Further, with the rise of data-center and cloud-computing environments, GPUs are starting to host multiple tasks from multiple users simultaneously. Though recent proposals such as spatial multitasking have improved the concurrency support, several problems still remain. First, spatial multitasking techniques improve the off-chip bandwidth utilization, but fail to improve the resource utilization within SM. Second, state-of-the-art simultaneous techniques primarily focus on memory resources like registers, shared memory and threads, but ignore the pipeline utilization.

In this paper, we show that pipeline under-utilization is the major performance bottleneck and that heterogeneous applications are imbalanced in utilization of different pipeline resources. Fortunately, such imbalance opens up the opportunities for simultaneously executing tasks with different pipeline resource requirements. Hence, we propose a simultaneous multitasking framework involving two key techniques. First, we employ a block dispatcher to adaptively adjust the kernels' thread-level parallelism for pipeline utilization improvement. Second, we propose a novel warp scheduling policy that supports execution of warps from two kernels simultaneously. We conduct systematic evaluation using a variety of applications. Experiments indicate that our simultaneous multitasking framework (*SMK*) achieves 1.51X (average) speedup for 28 two-kernel workloads.

1. INTRODUCTION

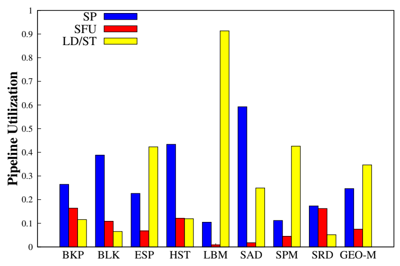
Over the past few years, GPUs have emerged as a powerful computing platform for general purpose computing. It seems clear that more and more users will enjoy the power of GPU computing by using the GPU-accelerated tasks. Further, GPUs are starting to host multiple tasks simultaneously. For example, one user may request to concurrently execute more than one task on the GPU integrated in his/her mobile SoC (System-on-Chip). More importantly, the rise of data-center and cloud-computing environments has led to an even larger scale of multitasking — many applications from multiple users compete for access to GPU resources simultaneously. Hence, the need for efficient multitasking support on GPUs is increasingly urgent.

Multitasking is particularly useful for improving the resource utilization of GPUs. Using NVIDIA's terminology, a GPU is first 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 (SPUs), and LD/ST units. Table 1 presents the resources in detail for each generation of NVIDIA GPUs from Fermi, Kepler to Maxwell. As shown, the resources are growing with each new generation. However, GPU applications especially the irregular and general purpose applications are often unable to effectively utilize all the resources on the GPUs [1, 2, 3, 4]. These applications tend to use only a portion of SMs or a portion of memory and computation resources within an SM [1, 5].

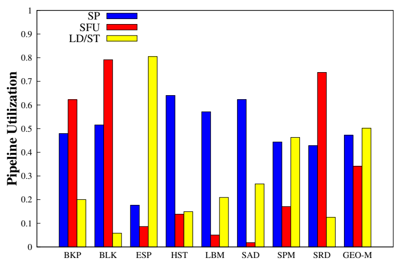
	GTX 480 Fermi	GTX 680 Kepler	GTX 980 Maxwell
SMs	15	8	16
SPs	32*15=480	192*8=1536	128*16=2048
LD/ST Units	16*15=240	32*8=256	32*16=512
SFUs	4*15=60	32*8=256	32*16=512
Threads/SM	1536	2048	2048
Warps/SM	48	64	64
Thread Blocks/SM	8	16	32
32-bit Registers/SM	32768	65536	65536
Shared Memory/SM	48KB	48KB	96KB

Table 1: Equipped hardware resources for different NVIDIA GPU generations.

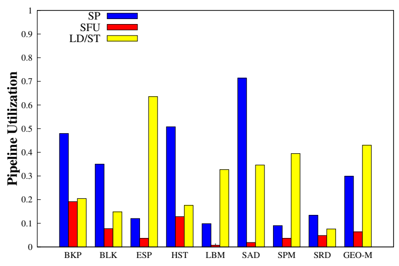
In response, NVIDIA architectures have included multitasking support since the Fermi architecture. The latest Kepler and Maxwell architectures feature with Hyper-Q mechanism [6], which allows kernels from the same process executing concurrently using different hardware queues. However, in practice, this 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 spatial and simultaneous multitasking. Spatial multitasking [7, 8] improves the SM utilization by assigning disjoint sets of SMs to different kernels. Although spatial 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. Simultaneous multitasking [9, 1] improves the spatial multitasking by assigning heterogeneous kernels onto the same SM. However, state-of-



(a) Baseline pipeline utilization.

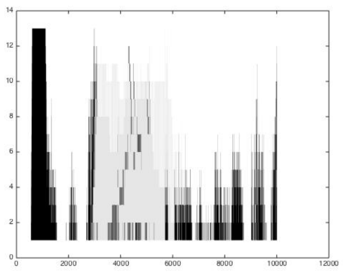


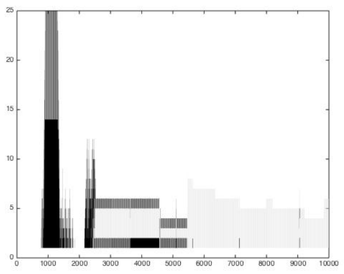
(b) Pipeline utilization with perfect memory.



(c) Pipeline utilization with max. TLP.

Figure 1: Pipeline utilization.


(a) Phase of HST.


(b) Phase of LBM.

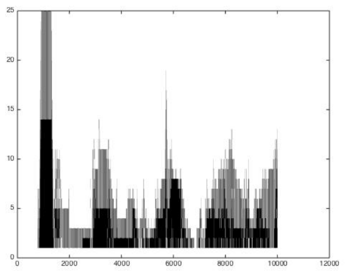

(c) Phase of HST and LBM.

Figure 2: Phase behaviors of heterogeneous kernels.

the-art simultaneous multitasking [9] primarily focuses on the memory resources, but ignores pipeline resources.

We identify that pipeline under-utilization is the major resource waste. Figure 1a depicts the *pipeline utilization* for the kernels in Table 2 on a Fermi-like architecture. Each pipeline resouce (SP, SFU, LD/ST) contains several SIMD units as shown in Table 1. Given a kernel, for each SIMD unit, we define its *pipeline utilization* as the ratio of cycles that the pipeline is active. The average *pipeline utilization* indirectly models both the horizonal and vertical pipeline waste(Section 2.2), in a similar way with the superscalar on the CPUs [10]. Overall, the *pipeline utilization* is very low. For Fermi architecture, the average pipeline utilization of SP, SFU, and LD/ST is 24%, 3%, and 33%, respectively. We also conduct the same experiments on a Kepler-like architecture, the average *pipeline utilization* of SP, SFU, and LD/ST is 29%, 5%, and 37%, respectively.

To demystify the reasons behind the low pipeline utilization, we study the impacts of cache and thread-level parallelism. On one hand, prior studies [11, 12] have demonstrated that due to serious cache contention, L1 cache can become the performance bottleneck for certain applications. If all the warps are waiting for data, then the pipeline has to be stalled. On the other hand, the thread-level parallelism (TLP) is determined by the application and the on-chip resources including registers and shared memory. If the application does not have enough TLP to drive the pipeline, then the pipeline has to be stalled, too [11]. Figure 1b shows the pipeline utilization with perfect memory system, which means zero access latency and no cache misses. Figure 1c shows the pipeline utilization with maximum TLP by enlarging the register and shared memory size. Compared with Figure 1a, pipeline utilization has been improved with perfect memory system and maximum TLP. However, there is still a large room to improve.

We argue that the low pipeline utilization is mainly attributed to the inherent imbalance in utilization of the pipeline resources for GPU applications. For example, application *LBM* almost fully utilizes LD/ST, but has very low utilization for SP. More importantly, each application exhibits variation in pipeline utilization during execution. Figure 2a and Figure 2b gives the detailed pipeline utilization during execution of application *HST* and *LBM*, respectively. The X-axis represents the clock cycle and Y-axis represents warp id. At cycle x, if warp y is executed on a SP, then we set the color of (x, y) as black. If warp x is executed on a LD/ST, then the color of that point is gray. The color is white if warp y is idle at time x. In this experiment, we ignore SFU. We find that different resources are demanded at different phases during execution. For example, during clock period 0-1000, SP is mainly occupied. However, during clock period 3000-6000, LD/ST is mainly used. During clock period 2200-2500, there are nearly no active warps and both SP and LD/ST units are idle.

Motivational Example. The under-utilization of pipeline resources and the imbalances in pipeline utilization of heterogeneous kernels offer opportunities to simultaneosly executing kernels with complementary resource demands to improve the overall resource utilization and thus performance. Figure 3 shows the pipeline utilization and performance by simultaneous executing *LBM* and *HST* together. As shown, performance has been increased with the pipeline utilization. The detailed pipeline utilization of the two kernels during

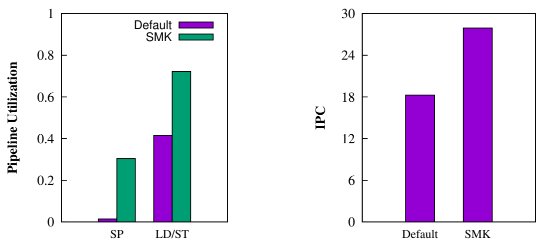


Figure 3: Motivational example by executing *LBM* and *HST* simultaneously.

runtime is shown in Figure 2c. Clearly, the pipeline idle time has been significantly reduced.

In this paper, we develop a simultaneous multitasking framework (SMK) involving two key techniques. First, we employ a block dispatcher to adaptively adjust the kernels' TLP for pipeline utilization improvement. Second, we propose a novel warp scheduling policy that supports execution of warps from two kernels simultaneously. We contribute the state-of-the-art GPU computing in the following aspects,

- We identify pipeline under-utilization as the major performance bottleneck and heterogeneous kernels exhibit imbalance in utilization of pipeline resources.
- We develop a simultaneous multitasking framework on GPUs (*SMK*) to improve performance by executing heterogeneous kernels with complementary resource demands on the same SM. *SMK* first employs a thread block dispatcher to dynamically adjust the TLP for each simultaneously executing kernel. *SMK* also uses a warp scheduler that simultaneously issues warps from multiple kernels.

We conduct systematic evaluation using a variety of applications. Experiments indicate that our simultaneous multitasking framework achieves average 1.51X speedup for 28 two-kernel workloads.

The rest of this paper is orgranized as follows. Section 2 describes the background of programming model and baseline GPU architecture. Section 3 presents the framework and details of our proposed simultaneous multitasking technique. Section 4 shows the evaluation results. Section 5 and Section 6 describe the related work and conclusion.

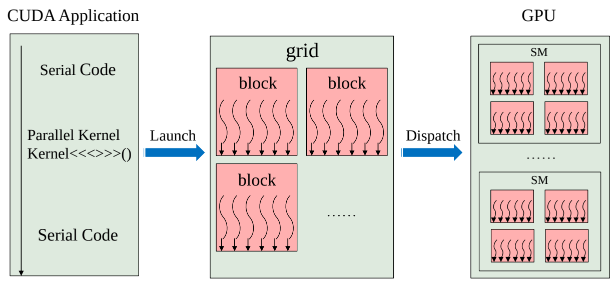


Figure 4: GPU programming model.

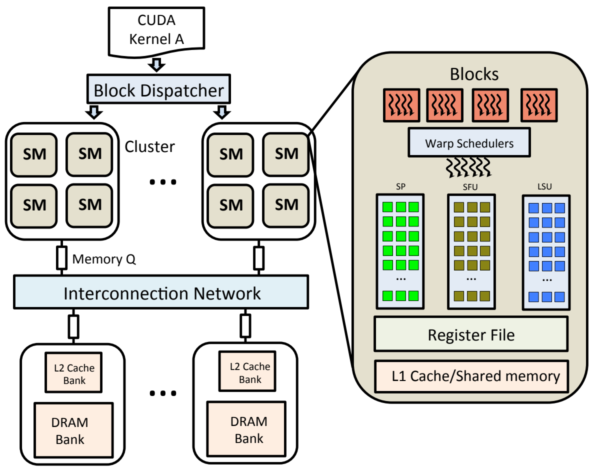


Figure 5: Baseline GPU architecture.

2. BACKGROUND

2.1 GPU Architecture

In GPU applications, the computing task that is offloaded to GPU for acceleration is written as a special function, called *kernel* (task). As is shown in Figure 4, 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.

A GPU is composed of several streaming multiprocessors (SMs) and SMs are connected with interconnection network and off-chip memory as shown in Figure 5. Each SM contains both on-chip memory and computation resources. Memory resources include L1 cache, register and shared memory. Each SM is also equipped with three types of SIMD pipelines for computation including streaming processor (SP), special function unit (SFU), and load/store unit(LD/ST). These SIMD pipelines are used for executing different types of instructions. The resource details of each generation of NVIDIA GPU can be found in Table 1.

Block Dispatcher. When a kernel is launched to GPUs, block dispatcher will dispatch threads to SMs at the unit of thread blocks. The maximum number of thread blocks on an SM is determined by the resource usage (register, shared memory, and threads) per block and the resource budget of each SM [1]. When a thread block finishes execution, its occupied resources will be released.

Warp Scheduler. After the thread blocks are dispatched to SMs, the blocks will be further divided into warps. The number of threads within one warp is a hardware limit(32 in NVIDIA GPUs). Threads in one warp are executed in a lockstep style and they share a dedicated storage for the context. At each clock cycle, the warp scheduler chooses one ready warp and issues it to a dedicated pipeline. Pipeline will be idle if there are no ready warps for it. Contemporary GPUs use multiple warp schedulers per SM for high throughput. For example, Fermi and Kepler is equipped with 2 and 4

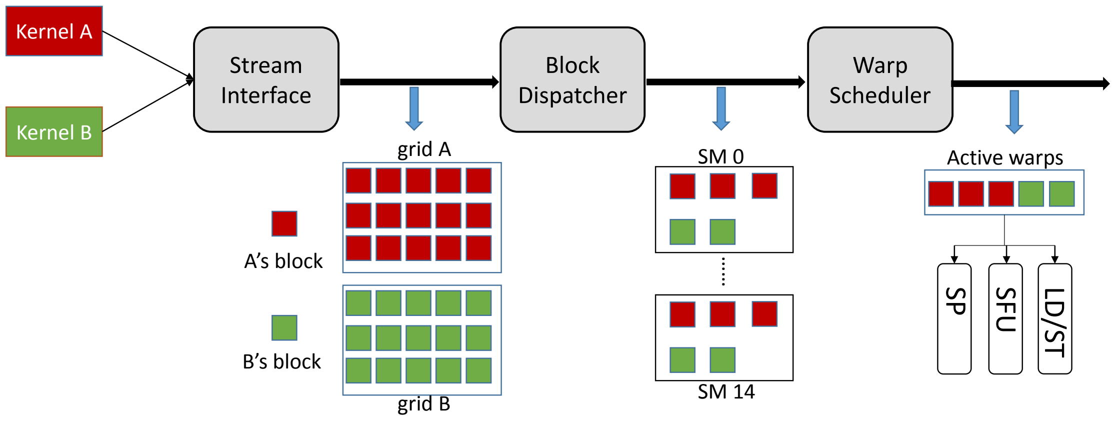


Figure 6: *SMK* framework overview.

warp schedulers per SM, respectively.

Instruction Fetch. Each warp has a dedicated instruction buffer to store fetched instructions. In Fermi, Kepler, and Maxwell, they all support up to two consecutive instructions from a warp issued at a cycle. Thus, each dedicated instruction buffer has two entries for the two instructions. When two entries are both empty, the warp is eligible for instruction fetch. The eligible warps are scheduled to access the instruction cache in a round robin manner. The selected warp sends a read request to instruction cache, and if the cache hits, two consecutive instructions are fetched; if the cache misses, instruction requests to lower-level cache is issued.

2.2 Pipeline Utilization

As previously discussed, each SM on a GPU contains three types of SIMD pipelines: *SP*, *LD/ST Unit*, and *SFU*. The SIMD width of these SIMD pipeline is 16 on Fermi and kepler, and 32 on Maxwell. The key to high performance for GPU applications lies in massive thread to drive the SIMD computation resources. However, as shown in Figure 1, the pipeline utilization is low due to the inherent imbalance in pipeline utilization.

Here, we illustrate how SIMD pipelines are wasted. Figure 7 shows an example for SPs. We assume each SM is equipped with 4 SP pipelines. During execution, every SP can be either idle or occupied at each cycle. Thus, similar to superscalar in CPU [10], the pipeline under-utilization is attributed to both vertical and horizontal waste. Horizontal waste occurs when only a portion of the SPs are occupied (cycle 1, 2, 7, and 8). Vertical waste occurs when all of the SPs are idle (cycle 3, 4, and 5).

2.3 Multitasking Support

Default Concurrency. The latest Kepler and Maxwell architectures feature with Hyper-Q mechanism [6], which allows tasks from the same process executing concurrently. Using stream software interface, programmers can push independent kernels into different streams so that they can be executed concurrently. However, the default concurrency support is still primitive. Concurrent kernel execution only occurs during the period when the first kernel is about to finish and the second kernel just gets started [1].

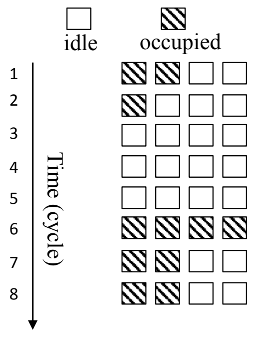


Figure 7: The illustration of pipeline under-utilization.

Spatial Multitasking. Spatial multitasking on GPUs is first proposed by Adriaens et al. [7]. Recently, Wu et al. [8] proposes a new spatial multitasking *SMC*. Spatial multitasking 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. Spatial multitasking 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.

Simultaneous multitasking. To fully utilize the resources equipped on GPUs, we propose to use simultaneous multitasking. Simultaneous multitasking allows thread blocks from different kernels executed onto the same SMs. Compared to spatial multitasking, simultaneous multitasking is a fine-grained approach. Prior study [9, 1] attempts to employ simultaneous multitasking for performance improvement. However, they all ignore pipeline utilization. In this paper, we use the pipeline utilization as guidance to tune the parameters of simultaneous multitasking and exploit the imbalance of pipeline utilization among heterogenous kernels.

3. SMK FRAMEWORK OVERVIEW

Figure 6 presents the overview of *SMK* framework. By default, the current generations of GPUs (Fermi, Kepler, and Maxwell) support concurrent kernel execution 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, similar to the prior studies [1, 8], we consider the concurrent execution of independent kernels from multiprogrammed workloads. We use the *stream* interface for simultaneous multitasking as shown in Figure 6. Independent kernels can be pushed to different streams for concurrent execution. In the following, we concentrate on the two-kernel workloads. But our framework can be applied to more than two kernels, too (Section 3.4).

When *SMK* receives two independent kernels from stream interface, it leverages on two components to mix them and simultaneously execute them as shown by Figure 6. First, it relies on the block dispatcher to determine the number of thread blocks for each kernel. Then, the warp scheduler will exploit the imbalance in pipeline utilization to schedule warps from two kernels simultaneously. In the following, we provide implementation details for block dispatcher and warp scheduler. The kernles shown in Figure 1a have consistently low pipeline utilization for SFU, so we will not model SFU in our framework.

3.1 Block Dispatcher

SMK 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. Given two kernels $\{A, B\}$, we use Tb_A and Tb_B to denote the number of thread blocks that simultaneously execute on each SM. Obviously, Tb_A and Tb_B are limited by the resource constraints,

$$Resource_A * Tb_A + Resource_B * Tb_B \leq Resource_{SM} \quad (1)$$

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.

As shown in Figure 1, heterogeneous kernels tend to use different pipeline resources. 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. More importantly, as shown in Figure 1, kernels exhibit large variations in pipeline utilization during execution. Hence, our block dispatcher is designed to adaptively adjust Tb_A and Tb_B.

To achieve this goal, our block dispatcher leverages online 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.* Set an 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 set $Tb_A = Max_A/2$ and $Tb_B = Max_B/2$, where Max_A (Max_B) is the maximum number of thread blocks that an SM can accommodate for kernel A (B). Then, we iteratively execute step 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.

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,

$$PUCT_{SP} = \frac{SP_{cur}}{SP_{his}} \quad (2)$$

$$PUCT_{LDST} = \frac{LDST_{cur}}{LDST_{his}} \quad (3)$$

where SP_{cur} and SP_{his} represents SP pipeline utilization in the current sampling period and historical sampling periods, respectively. Similarly, $LDST_{cur}$ and $LDST_{his}$ represents 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, 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,

$$PUCT = \frac{SP_{his}}{LDST_{his}} \quad (4)$$

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 detailed flow of 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.

3.2 Warp Scheduling

Warp is the thread scheduling unit on GPUs. On current GPUs. each SM is equipped with multiple warp schedulers. Each scheduler selects a ready warp to SIMD pipelines in a certain policy. Our *SMK* framework employs a two-level warp scheduler. At first-level, it determines the kernel from which it selects warps; at the second-level, it determines the

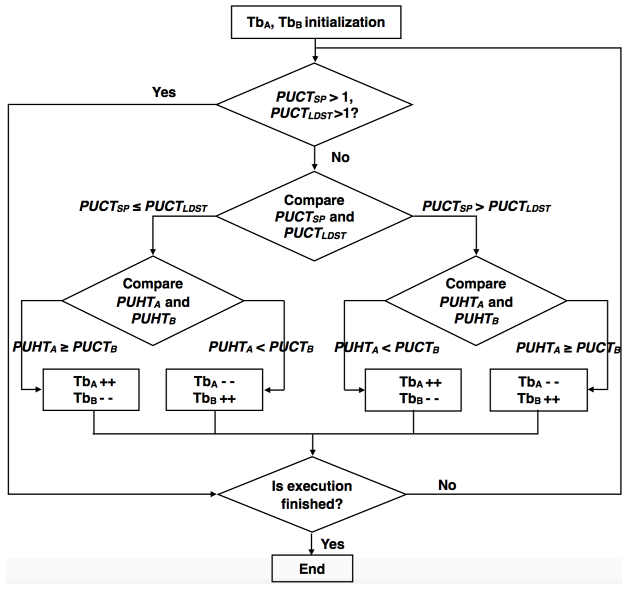


Figure 8: Flow chart of the block dispatching algorithm.

warps from the selected kernel. The state-of-the-art warp scheduling techniques [13, 14, 15, 12, 16] can be used for the second-level design. In this work, we mainly focus on the design of first-level. We choose Greedy-Then-Oldest (*GTO*) as the second level policy. For the first-level scheduling, we compare two policies: *Alternative* and *Simultaneous*.

Alternative. In *Alternative*, only one kernel is scheduled at the same time. Only when all the warps of the current kernel are stalled, the scheduler switch to warps from another kernel. Figure 9a shows the behavior of *Alternative* warp scheduler. Lee et al [9] first propose *Alternative*.

Simultaneous. Heterogeneous kernels tend to use different pipelines. Thus, intuitively, we propose to *Simultaneous* policy that schedules warps from different kernels at each cycle. Latest GPUs all feature more than one warp schedulers: Fermi has 2 and Kepler has 4. We propose to use half of the schedulers for each kernel.

Figure 9b compares the two warp schedulers. For *Alternative*, blocks from different kernels are allowed to dispatch to the same SM. The scheduler can hide the long latency by switching the warps from the other kernel to reduce vertical pipeline waste efficiently. But *Alternative* has no effect on horizontal pipeline waste. For *Simultaneous*, the scheduler is aware of phase behavior of kernel and attempts to schedule warps from different kernels in each cycle. So it can effectively reduce both vertical and horizontal pipeline waste.

3.3 Fetching Policy

For *Simultaneous* warp scheduler, in each cycle, warps from both kernels will be issued. The warp scheduling stage is dependent on the fetch stage. By default, the fetch unit chooses a warp in a round-robin fashion and then fetches two instructions for this warp. If the fetch unit can not provide sufficient instructions for the issue stage, the potential benefits of block dispatcher and warp scheduler will be diminished. Thus, we evaluate different fetching policy,

In *SMK* framework, the fetch policy has two levels. At the first-level, it determines which kernel to fetch; at the second-level, it determines which warps to fetch. We evaluate several fetching schemes, which are labeled *alg.num*, where *alg* is the first level fetch policy, and *num* is the number of warps that can fetch in a cycle. After the kernel is selected, we choose warps in a round-robin fashion and for each warp, we fetch as many as two consecutive instructions. We compare different fetch policies as follows,

1. *Null.1*. This is the default scheme, where the warps from different kernels are considered equally. Each cycle one warp fetches as many as 2 instructions. The instruction fetch bandwidth is 2.

2. *RR.2*. Each cycle a kernel will be selected in a round-robin manner. Then, two warps from the selected kernel are chosen, and each of the two warps fetches as many as 2 instructions. The instruction fetch bandwidth is 4.

3. *Both.2*. In each cycle, instructions are fetched from both kernels. For each kernel, it fetches as many as two instructions from one warp. The instruction fetch bandwidth is 4.

3.4 Extension to more than two-kernel

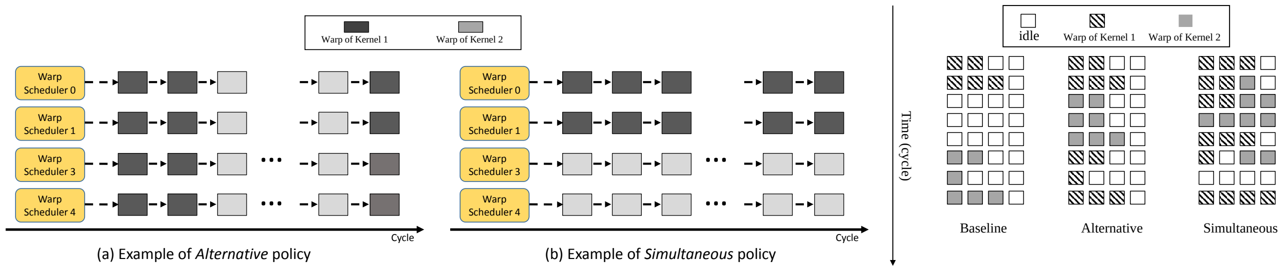
In the above discussion, we focus on the two-kernel workloads. However, our framework can also be extended for more than two-kernel workloads. More clearly, given a set of kernels stored in a pool, we can select two kernels for concurrent kernel execution first. When one of the kernel finishes, we will select another kernel from the pool and execute it concurrently with the left kernel. In practise, we find that executing more than three kernels simultaneously will aggravate resource contention such as L1 cache, resulting in worse performance. Hence, we only allow two kernels executing simultaneously.

3.5 Hardware Implementation Overhead

SMK framework requires small hardware changes. First, block dispatcher requires several hardware tables for storing the *PUCT* and *PUHP*. For each table, it has only two entries. For *PUCT*, one entry is for SP and another entry is for LD/ST spe. For *PUHP*, two entries are for two kernels separately. Second, GPUs are already equipped with multiple warp schedulers, we only need to assign the warp scheduler to the corresponding kernel. Last, if we use the default fetch policy (*Null.1*), then we do not need change fetch unit. However, if we adopt *RR.2* and *Both.2*, the total bandwidth of the output buses is 4 instructions. To support the policies, we design a 2-banked instruction cache as shown in Figure 10. Before the two PC addresses are sent to each bank, a bank conflict logic is needed to ensure that each address is sent to a separate bank. After the data is read, an output selection logic is needed in the result drive. In addition, multiple address buses are also needed. In Figure 10, the additional hardware is marked as blue color. The changes have a negligible impact on area and cache access time.

4. EXPERIMENTS

Our Simultaneous Multitasking framework (*SMK*)is implemented based on GPGPU-sim (version 3.2.2) using the



(a) An illustration of warp scheduling. (b) An illustration of pipeline utilization.

Figure 9: Comparison of different warp scheduling policy

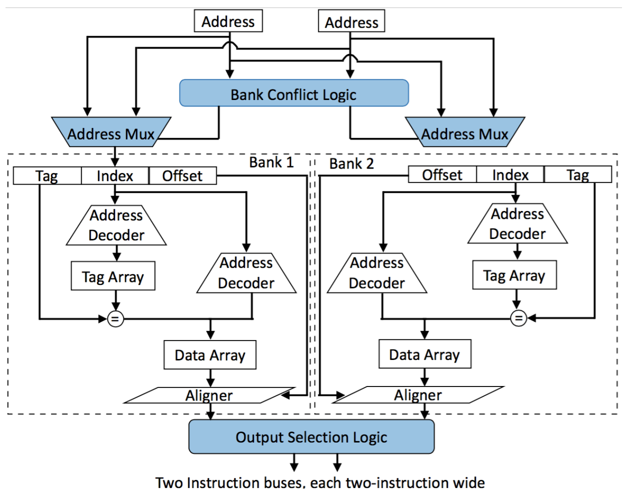


Figure 10: Design modifications on instruction cache

Table 2: Kernel Description

Abbr.	Kernel Name	Benchmark Suite	grid size	block size
BKP	bpnn_layerforward	Rodinia	4096	256
HST	calculate	Rodinia	1849	256
SAD	mb_sad	Parboil	1584	61
SRD	extract	Rodinia	450	512
SPM	spmv_jds	Parboil	374	192
BLK	blackschole	CUDA SDK	480	128
LBM	performStream	Parboil	18000	120
ESP	initRNGp	CUDA SDK	970	256

Table 3: GPGPU-Sim Configuration

# Compute Units (SM)	15
SM configuration	192 cores, 700MHz
Thread Limits per SM	2048 threads and 16 thread blocks
Register Limits per SM	65536 registers
Shared memory Limits per SM	48KB
Warp Scheduler	4 warp schedulers per SM, GTO policy
L1 Data Cache	16KB
L2 Unified Cache	1536KB

Kepler-like configurations in Table 3. We extend the *stream* interface to support our concurrency model. We evaluate our technique using 8 kernels from Rodinia [17], Parboil [18] and CUDA SDK [19] benchmark suites. Using 8 kernels, we can create 28 two-kernel workloads.

In the following, we perform five sets of experiments to evaluate *SMK*. First, we show the overall performance of *SMK*. Second, we evaluate different warp scheduling policies. Third, we compare our work with the state-of-the-art multitasking techniques [7, 8, 9]. Fourth, we discuss the sensitivity of *SMK* to different fetch policies. Finally, we show the results for more than two-kernel scenarios. In Section 3, we discuss two warp scheduling policies and three fetch policies. Here, we use simultaneous warp scheduler with the default instruction fetch policy as the default configuration of *SMK* and further discussions are presented in Section 4.2 and Section 4.5.

4.1 Performance Results

Figure 11 shows the performance speedup of our *SMK* framework compared with the default concurrency policy. The geometric mean of the achieved performance speedup is 1.51X.

Because different kernels have different execution time, there is always one kernel that finishes earlier than the other one. We split the execution time of the two concurrently executing kernels into two phases, *Concurrency Phase* and *Sequential Phase*. *Concurrency Phase* refers to the interval from when the two kernels start to when the first kernel finishes. *Sequential Phase* refers to the interval from when the first kernel finishes to the end. Obviously, only the *Concurrency Phase* benefits from our concurrent kernel execution. If *Sequential Phase* is much longer than the *Concurrency Phase*, then the performance improvement of our concurrent kernel execution will be diluted. For workloads *LBM_SPM*, *BLK_SRD*, and *HST_SAD*, our *SMK* show marginal performance improvement due to the discrepancy in their lifetime. For example, the running time of *ESP* is 5X longer than *SPM*.

Figure 13 shows the improvement of SP and LD/ST units utilizations compared with the default concurrency policy. On average, the utilizations of SP and LD/ST are improved by 18% and 12%, repectively. Performance speedup and pipeline utilization improvement are highly correlated. For workloads *LBM_BKP*, *ESP_BKP* and *BLK_ESP*, our *SMK* shows very high performance speedup. Because for these workloads, the two kernels have very complementary pipeline requirement as shown in Figure 1a. *LBM* and *ESP* have 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 signifi-

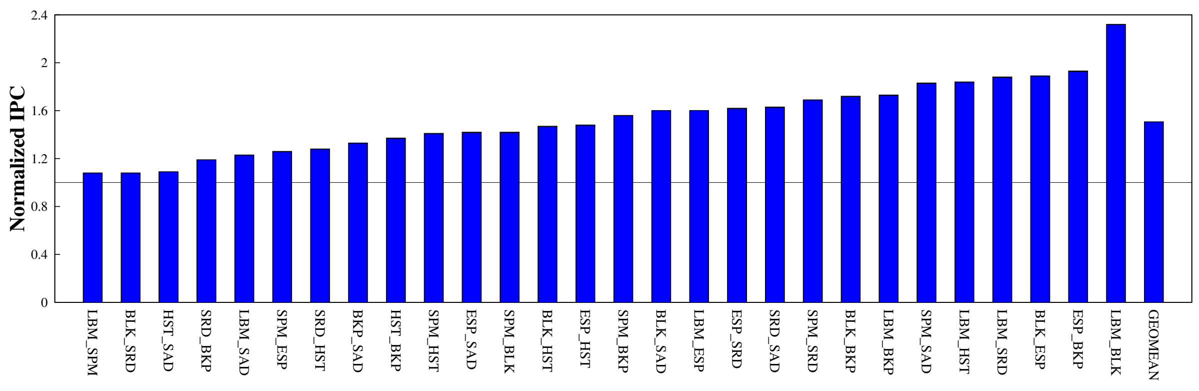


Figure 11: Performance results of SMK

cant performance improvement.

For workloads *BLK_BKP*, *HST_BKP*, and *BLK_HST*, our *SMK* can still achieve significant pipeline utilization improvement and performance speedup. However, the overall pipeline utilization shown in Figure 1a seems that the two kernels are not complementary. To find out the reason of performance benefit, we record the dynamic block dispatching process of *HST_BKP* in Figure 12. 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.

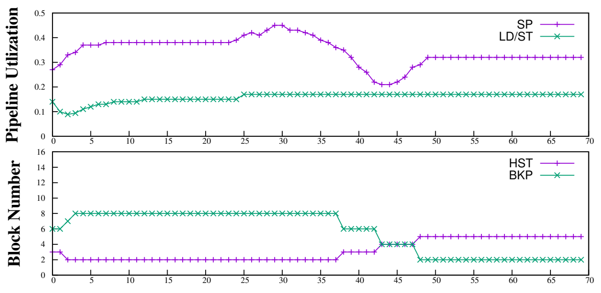


Figure 12: An illustration of block modulation process

Figure 14 shows the L1 cache miss rate. For most two-kernel workloads, our SMK has minor impact on the L1 cache performance. For some workloads, such as *LBM_BLK* and *ESP_BKP*, the miss rate is even reduced. This is because block dispatcher can dynamically adjust the TLP. High TLP may result in high cache miss rate and pipeline waste [11]. For these workloads, block dispatcher choose to decrease TLP to reduce the pipeline waste. As a result, cache miss is reduced.

4.2 Evaluation of different warp schedulers

Figure 15 shows the performance results of our *SMK* framework using *Alternative* warp schedulers and *Simultaneous* warp scheduler. On average, the achieved speedup s of SMK using *Alternative* and *Simultaneous* are 1.26X and 1.51X, repectively. Overall, our proposed *Simultaneous* warp schduler performs consistenly better for all workloads.The *Alternative* warp scheduler only schedule warps from one kernel in each cycle. Only there are no active warps of this kernel, *Alternative* would schedule warps from another kernel. *Alternative* warp scheduler mainly reduce vertical waste. In contrast, *SMK* can reduce both vertical and horizontal waste.

4.3 Comparison with the state-of-the-art techniques

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

Spatial Multitasking. Adriaens et al. [7] 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 which executes computing-intensive on idel 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 by Adriaens et al. In addition, Wu et al. [8] propose a 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 *SMK*. The performance is normalized to the baseline concurrency. On average, the performance speedups of *Smart-Even*, *SMC*, *SMK* are 1.21X, 1.26X and 1.51X, 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 *SMK* is fine-grained. Both *Smart-even* and *SMC* do not consider the pipeline utilization. Overall, *SMK* outperforms those two schemes for all the two-kernel workloads.

Simultaneous Multitasking. Lee et al. in [9] discuss the mixed concurrent kernel execution technique. Similar to our *SMK*, *mCKE* is also a fine-grained concurrency mechanism,

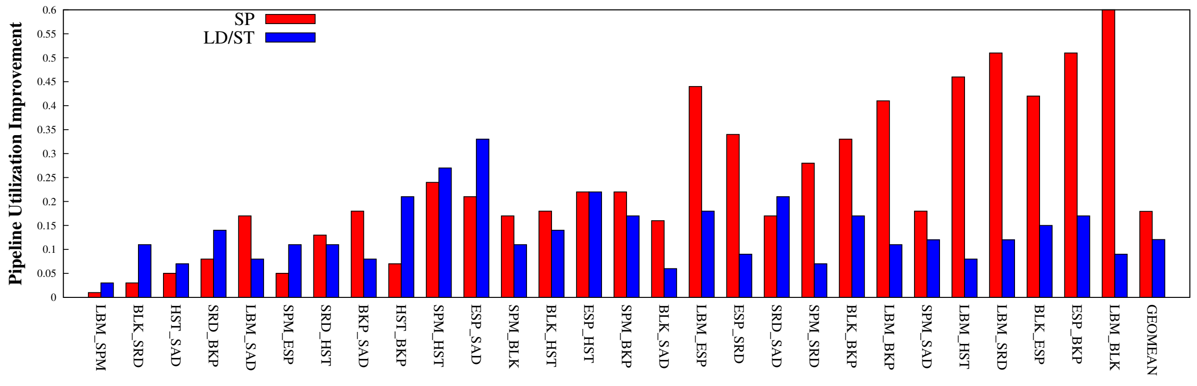


Figure 13: Pipeline utilization

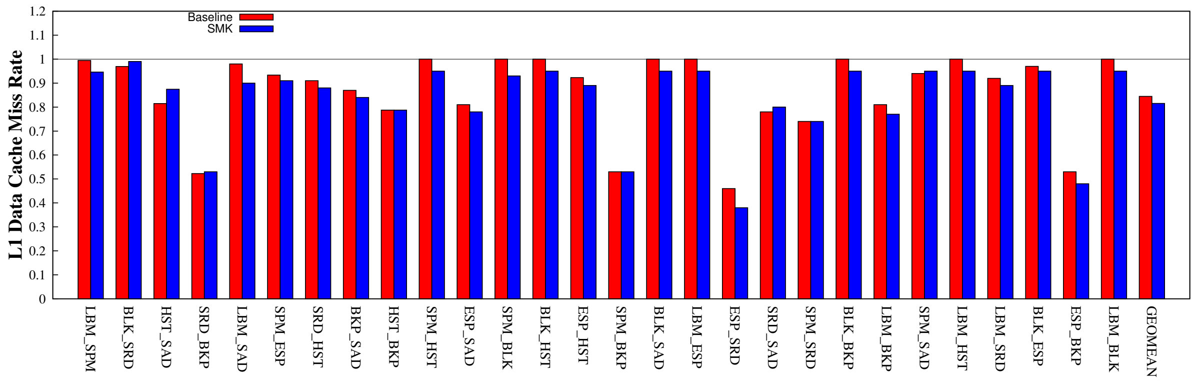


Figure 14: L1 data cache miss rate

which dispatches two concurrent kernels to the same SM. *mCKE* only gives the block configuration for several workloads. Hence, we can not compare it with *SMK* systematically. For a fair comparison, we only *SMK* with *mCKE* using the workloads used in [9].

Figure 18a shows the performance result. The performance is normalized to the baseline concurrency. Our *SMK* achieves better performance for all the evaluated workloads. On average, *our work* achieves 1.41X speedup, while *mCKE* achieves 1.12X speedup. The reason is two-folded. The first is, our *SMK* 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. The second is, *mCKE* applies *Alternative* warp scheduler, which is not as efficient as *Simultaneous* warp scheduler as we discuss in Section 3.2. Figure 18b shows that all of these workloads, *SMK* achieves get obvious pipeline improvement.

4.4 Evaluation of more than two-kernel scenarios

We also conduct experiments using three-kernel, four-kernel, and five-kernel scenarios. Due to space limit, we only perform experiments for a few workloads. Figure 19 shows the evluation results of extension. We select 3 three-kernel workloads, 3 four-kernel workloads, and 2 five-kernel workloads. We achieve 1.44X speedup on average. This demonstrates that *SMK* can scale to more than two-kernel scenarios.

4.5 Sensitivity to different fetch policie

We study three different fetch policies in Section 3.3. *Null.1*, *RR.2*, and *Both.2*. In the above subsections and our overall performance shown in Figure 11, we apply *Null.1* to conduct fair evaluation and comparison. In this subsection, we will discuss sensitivity of our proposed framework to different fetch polices. Figure 17 shows the performance speedup of our proposed framework using different fetch policies.

We can observe that the improvement gained by doubling fetch bandwidth is limited. For example, the *RR.2* and *Both.2* outperform *Null.1* by 21% and 26%, respectively. For *ESP_SPM* and *LBM_SPM*, both *RR.2* and *Both.2* can not help performance. For the three workloads, the bottlneck is imbalanced usage of pipeline, but not the lack of instrucions. Similarly, for *SRD_SPM*, *BLK_SPM*, and *LBM_BLK*, the pipeline utilization is significantly improved by our proposed framework. So though the instruction bandwidth is double, the performance benefit is little.

Overall, the *Both.2* scheme provides better performance than *RR.2*. Because *Both.2* fetches instructions from two kernels, which makes it more flexible to balance pipeline utilization in warp issue stage. Moreover, the fetch mode matches well with simultaneous warp scheduling policy. They both process warps from diffrent kernels.

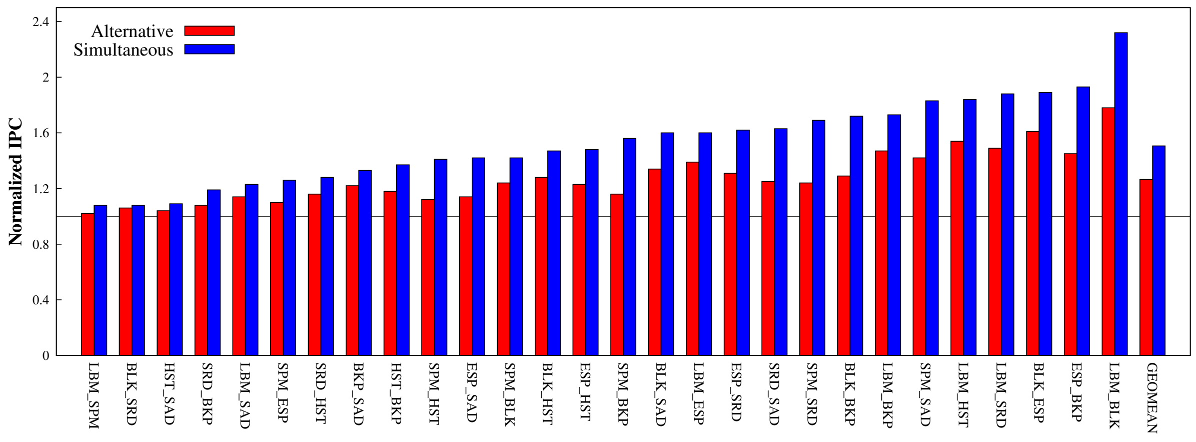


Figure 15: The performance results of SMK framework using different warp schedulers.

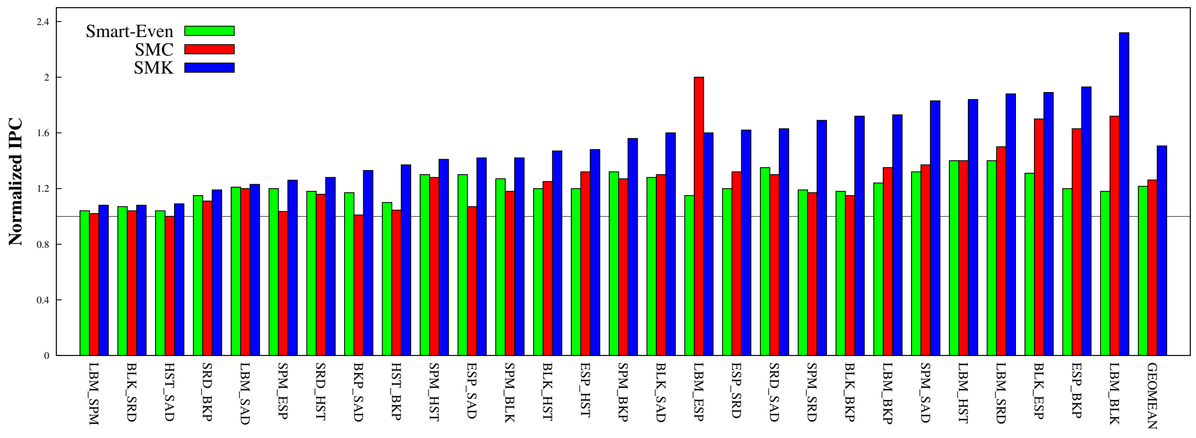


Figure 16: Comparison with coarse-grained multitasking.

5. 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 [20, 21, 3, 4, 22, 2], warp schedulers [14, 16, 15, 20, 12, 14, 13], on-chip memory optimizations [23, 24, 25, 26, 27, 28], and concurrent kernel executions [29, 1, 9, 7, 30, 8].

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 [7, 29, 8], they partition the SMs among kernels to enable concurrent execution. Adriaens et al. 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 [7]. They discuss serveal SM partitioning heuristics and evaluate their performance. Wu et al [8] 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. [29] propose an efficient heuristic algorithm and a software emulation framework for botm temporal and spatial concurrency. However, all those techniques do not allow different kernels executing in one SM, their performance impropments come from the improved the off-chip memory bandwidth utilization and they are oblivious to the SIMD pipeline utilization inner SMs. Lee et al. [9] propose a fine-grained concurrency policy, however, their technique is still primitive. Pai et al. identify that CUDA programs do not scale to utilize all the available resources on GPUs [1]. In order to improve resource utilization, they propose kernel transformation techniques that convert CUDA kernels into elastic kernels which ebables fine-grained control over their resource usage. By merge multiple kernels, they enable fine-grained concurrent kernel execution. This is an software approach while our *SMK* is hardware appraoch. Tanasic et al [30] and Pichai et al asplos2014Pichai consider memory isolation and provide architectural support for multiprogramming on GPUs.

Block dispatcher and warp scheduler is also hot research topics on GPUs. The block dispatcher directly determines the thread level parallelim on each SM and further affects warp scheduler. Rogers et al. [12] propose to detect the inter-

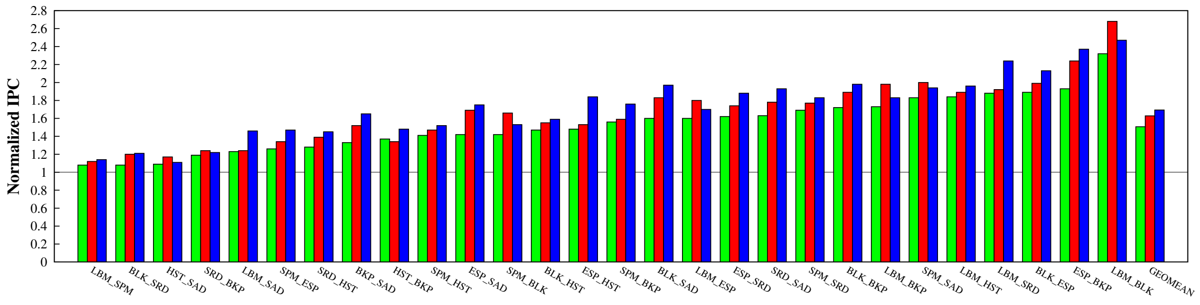
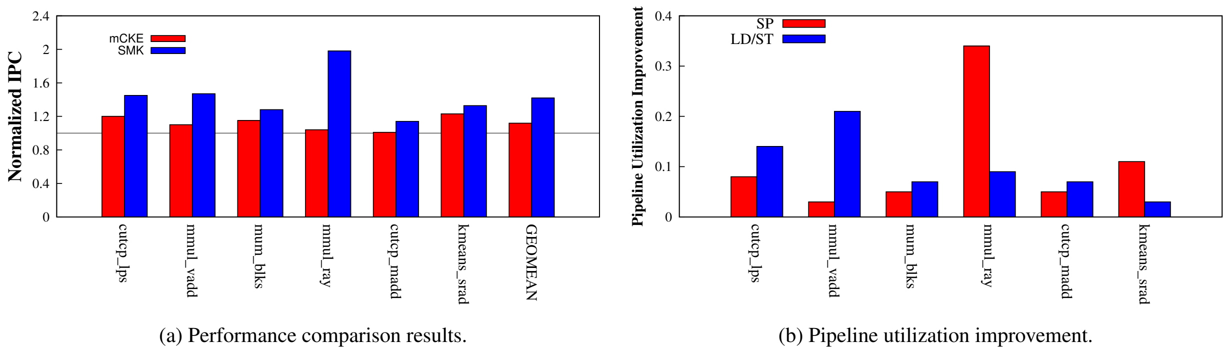


Figure 17: Performance Result of different fetch policies



(a) Performance comparison results.

(b) Pipeline utilization improvement.

Figure 18: Comparison with mCKE

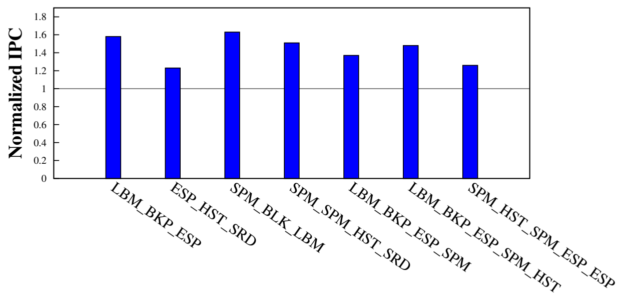


Figure 19: Evaluation of extension

warp cache contention at run-time and limit the number of active warps when the cache contention is detected. Kayiran et al. demonstrate that executing the maximum possible number of thread blocks on a core is not always the optimal choice from the performance perspective due to the cache contention problem [11]. Lee et al. propose a thread block scheduler to determine an optimal number of thread block issued to a core [9]. However, throttling the number of blocks assigned to a core results in static resource under-utilization on GPUs. Our framework also propose a block dispatcher, which determine block numbers of multiple concurrent kernels and does not limit the number of blocks per SM.

Veynu et al [15] and Rogers et al [20] propose two-level scheduling and cache-conscious warp scheduling. They focus on the optimization in sigle kernel scenario. In contrast, our *Simultaneous* warp scheduler is design to improve the pipeline utilization in concurrent kernel executions. Our warp scheduler has two-level, kernel-level and warp-level. In this work, we focus on the kernel-level. In fact, these works are all warp-level, and can be easily extended in our framework.

6. CONCLUSIONS

Improving resource utilization on GPUs is primary to fully exploit the GPUs' computing ability. Prior works mainly focus on optimization on the static resouces utilization and on-chip memory. However, in this work, we first identify that if only considering static resource and l1 cache, the improvement space is limited. We find that a kernel may show imbalanced requirement for pipelines and the requirement for pipelines is always changing during run-time. Thus, we suggest dispatching different kernels to an SM to adaptively get the pipeline utilization balanced and further improve performance.

In this paper, we propose a simultaneous multitasking framework which consists of a block dispatcher and a warp scheduler. The block dispatcher can adaptively modulate the number of blocks concurrently executing on an SM. The warp scheduler can support warps from both kernels can be issued simultaneously. We conduct systematic evaluation on our *SMK* framework using 28 representative two-kernel workloads and demonstrate that concurrent kernel execution can achieve substantial performance improvement (1.51X on average). Compared with the state-of-the-art multitasking techniques on GPUs, *Smart-Even* from [7], *SMC* from [8], and *mCKE* from [9] our technique increase the performance speedup from 1.21X, 1.29X, and 1.12X to 1.51X.

7. REFERENCES

[1] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU Concurrency with Elastic Kernels," *SIGPLAN Not.*, vol. 48, pp. 407–418, March 2013.

[2] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, IISWC '12, pp. 141–151, 2012.

[3] W. W. L. Fung and T. M. Aamodt, "Thread block compaction for efficient simt control flow," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pp. 25–36, 2011.

[4] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-40, pp. 407–420, 2007.

[5] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, "Fine-grained resource sharing for concurrent gpgpu kernels," in *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, HotPar '12, pp. 10–10, 2012.

[6] "CUDA HyperQ." `http://docs.nvidia.com/cuda/samples/6_Advanced/simpleHyperQ/doc/HyperQ.pdf`.

[7] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for gpgpu spatial multitasking," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pp. 1–12, 2012.

[8] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, "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, pp. 119–130, 2015.

[9] 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 *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, HPCA '14, pp. 260–271, 2014.

[10] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ISCA '98, pp. 533–544, 1998.

[11] O. Kayiran, A. Jog, M. Kandemir, and C. Das, "Neither more nor less: Optimizing thread-level parallelism for gpgpus," in *2013 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pp. 157–166, 2013.

[12] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pp. 72–83, 2012.

[13] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "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, pp. 395–406, 2013.

[14] S.-Y. Lee and C.-J. Wu, "Caws: Criticality-aware warp scheduling for gpgpu workloads," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pp. 175–186, 2014.

[15] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "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, pp. 308–317, 2011.

[16] J. A. Jablin, T. B. Jablin, O. Mutlu, and M. Herlihy, "Warp-aware trace scheduling for gpus," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pp. 163–174, 2014.

[17] "Rodinia Benchmark Suite.." `http://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Downloads`.

[18] "Parboil Benchmark Suite.." `http://impact.crhc.illinois.edu/Parboil/parboil.aspx`.

[19] "CUDA SDK.." `https://developer.nvidia.com/cuda-code-samples`.

[20] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware warp scheduling," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pp. 99–110, 2013.

[21] Z. Cui, Y. Liang, K. Rupnow, and D. Chen, "An accurate gpu performance model for effective control flow divergence optimization," in *2012 IEEE 26th International Parallel Distributed Processing Symposium*, ITPDPS '12, pp. 83–94, May 2012.

[22] T. G. Rogers, D. R. Johnson, M. O'Connor, and S. W. Keckler, "A variable warp size architecture," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pp. 489–501, 2015.

[23] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu, "Adaptive cache management for energy-efficient gpu computing," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pp. 343–355, 2014.

[24] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and improving the use of demand-fetched caches in gpus," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pp. 15–24, 2012.

[25] J. Lee, N. Lakshminarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching mechanisms for gpgpu applications," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, MICRO-43, pp. 213–224, 2010.

[26] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated scheduling and prefetching for gpgpus," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pp. 332–343, 2013.

[27] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "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, pp. 96–106, 2012.

[28] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated static and dynamic cache bypassing for gpus," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, HPCA '15, pp. 76–88, 2015.

[29] Y. Liang, H. Huynh, K. Rupnow, R. Goh, and D. Chen, "Efficient gpu spatial-temporal multitasking," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, pp. 748–760, March 2015.

[30] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pp. 193–204, 2014.