# Simultaneous Multikernel GPU: Multi-tasking Throughput Processors via Fine-Grained Sharing

Zhenning Wang*, Jun Yang†, Rami Melhem,
Bruce Childers, Youtao Zhang, and Minyi Guo‡
Department of Computer Science, †Electrical and Computer Engineering Department
*‡Shanghai Jiao Tong University, P. R. China, University of Pittsburgh, U. S. A.
*znwang@sjtu.edu.cn, †juy9@pitt.edu, {melhem,childers,zhangyt}@cs.pitt.edu,‡guo-my@cs.sjtu.edu.cn

## ABSTRACT

Studies show that non-graphics programs can be less optimized for the GPU hardware, leading to significant resource under-utilization. Sharing the GPU among multiple programs can effectively improve utilization, which is particularly attractive to systems where many applications require access to the GPU (e.g., cloud computing). However, current GPUs lack proper architecture features to support sharing. Initial attempts are preliminary: They either provide only static sharing, which requires recompilation or code transformation, or they do not effectively improve GPU resource utilization. We propose Simultaneous Multikernel (SMK), a fine-grain dynamic sharing mechanism, that fully utilizes resources within a streaming multiprocessor by exploiting heterogeneity of different kernels. We propose several resource allocation strategies to improve system throughput while maintaining fairness. Our evaluation shows that for shared workloads with complementary resource occupancy, SMK improves GPU throughput by 52% over non-shared execution and 17% over a state-of-the-art design.

## 1. INTRODUCTION

The growth of general-purpose GPU computing has led to its wide adoption in cloud computing, data centers, and even mobile/embedded computers. A GPU typically has many streaming multiprocessors (SM), each containing many simple execution cores [1]. GPUs provide a massive number of compute cores to exploit thread-level parallelism for hiding memory latency through heavy multi-threading. Many applications have been ported to GPUs to leverage the enormous computing power these architectures offer for significant speedup [2][3][4].

However, non-graphics applications may be less optimized for GPUs, causing on-chip resource under-utilization. One kernel[1] may use minimal scratchpad memory while accessing the L1 cache heavily. Another kernel may have the opposite usage. This situation can arise because a user may not be experienced enough to write high quality code to fully utilize available resources. Prior work has reported a similar observation [5]. In addition, even though GPUs are heavily multi-threaded, there are still abundant core idle cycles because current GPUs are memory bandwidth bound. Memory requests that cannot be served promptly may introduce more stalls than the available multi-threading capability can hide. This situation is especially common for memory intensive applications, indicating that dynamic core computing cycles are often under-utilized. We have observed 52.5%-98.1% core idle time for ten Parboil [6] benchmarks.

Furthermore, data centers or clouds are typically exposed as services to users. Consolidation and virtualization are used to share hardware resources among several applications for cost-effectiveness and energy efficiency. Yet, even the latest GPUs have no or minimal support for shared execution of multiple applications. Once a kernel is launched onto the GPU, it cannot be easily interrupted and kernels of other applications have to wait for the GPU to become available.

There are software attempts to enable sharing the GPU. These techniques mainly rely on users or code transformation to statically define or fuse parallelizable kernels [5][7]. These solutions can be appropriate for embedded systems where the execution of kernels is determined statically. However, the approaches cannot easily accommodate general systems having unknown jobs arriving dynamically. Hence, static techniques do not provide sharing among dynamically arriving GPU jobs. Also, once launched, kernels cannot be preempted and resumed later.

Kernel preemption was recently proposed with architectural extensions [8][9]. The main mechanism is to swap the context of a kernel on one SM with the context of a new kernel. A context switch is achieved by hot-swapping kernel contexts between one SM and main memory, which generates high memory traffic and incurs large performance overhead. Alternatively, one SM can drain all active TBs before receiving new TBs from the preempting kernel. This approach may lead to long turn-around times due to the potentially long time an SM needs to completely drain all active TBs. Chimera [8] introduces a third option, termed "flushing", to simply drop the execution of idempotent TBs if preempted. This approach also integrates hot-swapping, draining and flushing into a hybrid mechanism to achieve low preemption overhead. These proposals allow different application kernels to execute concurrently on disjoint sets of SMs, achieving spatially-partitioned multitasking (Spart) of a GPU [9][10]. Unfortunately, resource under-utilization occurs mostly within an SM. Hence, context switching all the

[1]A GPU application consists of multiple kernels, each capable of spawning many threads that are grouped into thread blocks (TB).

978-1-4673-9211-2/16/$31 

TBs running on an SM cannot effectively improve GPU resource utilization.

In this paper, we introduce a new notion of a multi-tasking GPU that (1) significantly improves resource utilization (both static and dynamic) to boost overall system throughput, (2) provides fair sharing among concurrent kernels, and (3) improves turn-around time for concurrent jobs. We propose **Simultaneous Multikernel (SMK)**, drawing an analogy from simultaneous multithreading for CPUs, to increase thread-level parallelism (TLP) of a GPU. SMK exploits kernel heterogeneity to allow fine-grain sharing by multiple kernels *within each SM*. SMK is enabled by a fine-grain context switch mechanism on per TB basis for low preemption overhead. Moreover, new TB dispatch strategies and a new warp scheduling strategy are proposed to maintain resource fairness among sharing kernels.

The fundamental principle is to co-execute kernels with compensating resource usage in the same SM to achieve high utilization and efficiency. For example, a memory-intensive kernel can co-execute with a compute-intensive kernel. The former may not use scratchpad memory but incur a large number of memory stall cycles. The latter may fully use scratchpad memory (to speedup execution), and thus, use a large number of compute cycles. Pairing these kernels in the same SM can greatly improve utilization of both scratchpad and compute cycles, achieving higher overall efficiency and GPU throughput. We make the following technical contributions in this paper:

- *A fine-grain context switch mechanism to support SMK with very low preemption overhead.* We propose to perform a context switch only on a per TB basis. When a preempting kernel arrives, we swap out just enough TBs on a SM to make enough "room" for a new TB of the preempting kernel to start execution. This design greatly reduces context switch overhead and the lead time before a preempting kernel begins execution, which results in better response time.
- *A TB dispatch mechanism to take into account static resource usage and fairness among different kernels.* When dispatching a TB onto an SM, we integrate fairness to ensure that static resource allocation is fair. This TB dispatch can be geared to generate SMK sharing with resource partitioning. Hence, our design naturally subsumes Spart with lower context switch overhead.
- *A warp scheduling algorithm that manages the dynamic core compute cycles of concurrent kernels.* We design a runtime mechanism to guide warp scheduling of each SM in allocating cycles to different kernels to minimize idle time. The warp scheduler allocates cycles to kernels in proportion to an individual kernel's dynamic cycle utilization when run independently (obtained through online profiling of SMs).

An evaluation on 45 pairs of Parboil kernels show improvement in system throughput of up to 61.2% over Spart can be achieved by SMK, with an average of 17% for pairs with complementary resource usage, and an average of 12.7% for all 45 pairs. Fairness is improved by 5.7% over Spart. The average turn-around time is greatly reduced as well, with an average of 19.0% for pairs with complementary resource usage, and an average of 10.6% for all 45 pairs. SMK has higher throughput as more kernels run concurrently, with an average of 14.8% and 14.3% improvement over Spart for 3 and 4 kernels respectively.

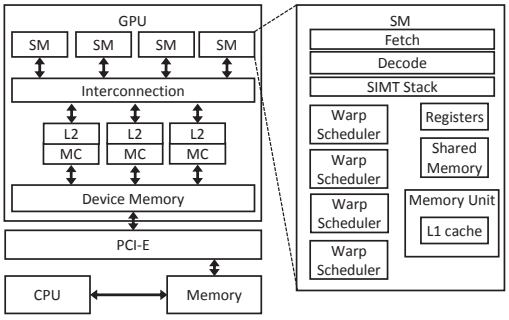



Figure 1: System overview.

## 2. BACKGROUND

In this section, we describe the GPU execution model and the baseline architecture. The baseline is the one supported by GPGPU-Sim [11], which models an Nvidia discrete GPU. We do not describe integrated GPUs because the only difference is the connection between the CPUs and the GPU. We use Nvidia/CUDA terminology, although the description applies to GPUs from other vendors as well.

### 2.1 GPU Execution Model

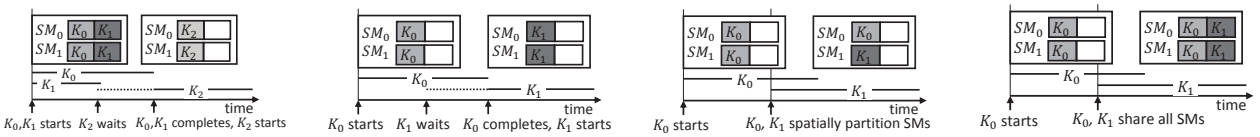
Usually, a GPU program has two parts: host code and device code (GPU kernel). Kernels are SIMT (Single Instruction Multiple Threads) programs for GPUs. The programmer writes code for one thread, and multiple threads execute the same code on the GPU. Threads are grouped into thread blocks (TB). The number of threads and the TB size are set by the programmer.

The number of concurrent TBs is limited by GPU resources (registers, scratchpad memory, and thread number). If the resources are not enough to dispatch all TBs in a kernel, the remaining TBs wait for the executing ones to finish. A TB is the minimal unit of dispatch. For example, if there are enough resources to hold 384 threads but the size of a TB is 256, then only one TB is dispatched and the remaining resources are wasted. The resources that a thread needs are determined at compile time.

### 2.2 GPU Architecture

Figure 1 shows an overview of a GPU system, where discrete devices are connected through the PCI-E bus. A GPU has multiple Streaming Multiprocessors (SMs). SMs share GDDR5 memory through an interconnect network as device memory. Memory requests are distributed to multiple memory controllers according to address. A GPU also has a unified L2 data cache for all SMs.

The SM is the main execution unit. Threads from one TB are grouped into warps in hardware. Each warp has 32 threads, which is the SIMD width of the GPU. A SM can




(a) Current software's solution

(b) Current OS's solution

(c) Current architecture solution with preemption and spatial partitioning

(d) SMK solution with preemption and per SM sharing

Figure 2: Evolution of multitasking GPU.

hold only a certain number of warps due to resource limitations. It has one or more warp schedulers to fetch, decode and issue instructions. Each warp scheduler governs 32 ALUs, which is the size of a warp. Warps are equally assigned to different schedulers that decide how to schedule the warps. Registers, shared memory and caches are shared by the warps being scheduled.

SMs can switch to different warps without any overhead because they do not need to switch warp context. Other warps can be executed while one warp is stalled by memory or other operations. As a result, the latency of the stalled warp is hidden and does not have impact on throughput.

## 3. RELATED WORK

Initial support for kernel concurrency relies on the programmer to define parallelizable *streams* of kernels, each stream being a sequence of kernels with dependencies. Hyper-Q [12] improves that approach by providing hardware queues for streams so they can launch from different queues and co-run on the same GPU. User-defined static parallelism is suitable for a single application with multiple kernels, but it is not appropriate for multiple applications. MPS [13] is a runtime mechanism to insert kernels from different processes into Hyper-Qs. However, once kernels are in the Hyper-Qs, MPS has no control over how the TBs are dispatched. As pointed out previously [14], the hardware would mostly serialize two parallelizable kernels since there is no control of TB dispatch. Moreover, MPS does not implement resource management. Co-running kernels compete for resources in an uncontrolled manner, which harms fairness. Additionally, these approaches do not enable sharing for dynamically arriving GPU jobs, as once launched, kernels cannot be preempted and resumed later.

More sophisticated software-based schemes workaround hardware limitations by merging two kernels into one with compiler techniques [5][7][14]. The execution path is controlled by conditional statements [15][16]. Although parallelizing different applications is possible, such static approaches do not facilitate dynamic sharing, as illustrated in Fig. 2a. Further, since kernels from different applications are fused together, the hardware sees only one kernel, and hence, the approach could produce unfair scheduling among different kernel threads. Also, kernel source code must be available prior to execution which can be problematic. Hence, these schemes only work when the kernels to execute are known in advance (e.g., embedded systems).

There have been many efforts in OS or hypervisor of a virtual machine to enhance multitasking in a GPU [17][18][19]. The typical approach is to intercept kernel launch requests and change to another kernel as demanded [20]. The main limitation is an already launched kernel cannot be preempted, so a new kernel must wait for the completion of the executing one, as illustrated in Fig. 2b. Hence, no sharing is supported and resource utilization is not improved.

Recently, architectural extensions have been proposed for sharing the GPU with kernel preemption. A context switch is achieved by hot-swapping kernel contexts in one SM with a new kernel via the main memory, or draining all running TBs on one SM and then loading in a new context [9]. Chimera [8] reduces swap overhead by dropping running TBs of one SM if the kernel is idempotent [21]. These techniques allow multiple kernels to share the GPU via spatially partitioning SMs (Spart), as illustrated in Fig. 2c. However, each SM in Spart still executes one kernel at a time while resource under-utilization mainly occurs *within* an SM. Our proposed SMK on the contrary enables multiple kernels to share each SM (Fig. 2d), and hence, can achieve better resource utilization and higher GPU throughput.

## 4. MOTIVATION

In this section, we first give a brief introduction of spatial partitioning, and discuss its limitation. Then, we discuss the heterogeneity of kernels, which motivates our work.

### 4.1 Sharing Granularity

As discussed above, state-of-the-art kernel preemption swaps context at the granularity of an entire SM [8][9]. This preemption scheme has two drawbacks. First, preemption overhead is high. Each context includes hundreds of kilobytes of registers, scratchpad memory and other execution status, which generates high memory traffic volume during context switch. This traffic not only blocks the current kernel being swapped, but may also stall other concurrent kernels that are not swapped, because the memory bandwidth is saturated by the context switch.

Second, preempting at the granularity of a whole SM results in a sharing mode of spatially partitioning SMs among different kernels. This sharing is limited by the number of SMs in a GPU, and works well only when SMs are abundant. However, the number of SMs has been relatively constant for recent GPU generations. Table 1 shows the number of SMs and the resources in each SM for recent GPU generations. The number of SMs started from 30, went down to 16 in Fermi and maintained that level in following generations. Instead of increasing the number of SMs, each SM is indeed becoming more powerful. For example, the number of warp

Table 1: The resources of the GPU of ascending generations. The resources are for each SM.

| Generation | # of SMs | # of ALU in SM | Registers | Shared Memory | Thread Number Limit | TB Limit |
|---|---|---|---|---|---|---|
| Tesla(GTX280) | 30 | 8 | 64KB | 16KB | 1024 | 8 |
| Fermi(GTX580) | 16 | 32 | 128KB | 48KB | 1536 | 16 |
| Kepler(GTX780 Ti) | 15 | 192 | 256KB | 48KB | 2048 | 16 |
| Maxwell(GTX980) | 16 | 128 | 256KB | 96KB | 2048 | 32 |

Table 2: Resource usage on Nvidia GTX980. The limiting resource is in **bold**.

| Kernel | Registers | Shared Memory | Thread Number | Type | Stall Cycles |
|---|---|---|---|---|---|
| lattice6overlap | 87.5% | 67.1% | **100%** | Compute Intensive | 54.5% |
| StreamCollide | **92.3%** | 0% | 52.7% | Memory Intensive | 89.9% |
| mysgemmNT | **94.5%** | 5.7% | 68.8% | Compute Intensive | 52.5% |
| spmvjds | 46.9% | 0% | **93.8%** | Memory Intensive | 91.2% |
| block2Dregtiling | 75.0% | 0% | **100%** | Memory Intensive | 91.2% |
| genhists | 76.6% | **94.8%** | 87.5% | Compute Intensive | 60.2% |

instructions that can be issued per cycle increased from 2 (at width of 16 threads) to 4 (at width of 32) for GTX580 and GTX980 respectively. Hence, one SM in GTX980 has similar computing capability as four SMs in GTX580. One possible reason for decreasing the number of SMs is that the on-chip interconnection is not yet ready for a large number of SMs. Currently, SMs are connected through a crossbar, which faces scalability challenges. Hence, Spart is restricted by the fact that the number of SMs is non-increasing. Moreover, Spart cannot address low utilization, which will be exacerbated with more powerful SMs, as discussed next.

## 4.2 Kernel Heterogeneity

Resource usage and runtime behavior change from kernel to kernel. We examined all benchmarks from Parboil and Rodinia and observed low resource utilization across nearly all of them for a recent GPU generation, Nvidia GTX980. Table 2 reports resource usage and runtime behavior of a few representative cases.

The table shows that resource usage of different kernels is dramatically different, partly due to different thread organizations decided by the programmer. *StreamCollide* and *mysgemmNT* fully use registers but leave shared memory almost entirely unused. On the contrary, *block2Dregtiling* and *lattice6overlap* are limited by hardware constraints on the number of threads, which causes registers and shared memory to be under-utilized. This problem cannot be solved by partitioning SMs among kernels because each SM can only be assigned to one kernel at a time. Resources are still underutilized within the partition of SMs.

Runtime behavior of the kernels is also different. *lattice6overlap*, *mysgemmNT* and *genhists* are compute intensive with high IPC, while the other kernels are memory intensive. Kernels with high IPC utilize shared memory to cache frequently accessed data, reducing stall cycles. Memory-intensive kernels use thread concurrency to hide long memory access latency, and stall cycles are much larger than in compute-intensive kernels. However, overlapping compute cycles with memory stall cycles happens only among threads within one SM.

If multiple kernels can be assigned to one SM, kernels with different resource demands or runtime behavior can be placed in the same SM. This strategy increases utilization and may improve performance. Assume we have two kernels ($K_1$ and $K_2$) for a SM with 3K registers and 3KB shared memory. $K_1$ demands 1K registers and 2KB shared memory per TB, while $K_2$ demands 2K registers and 0B shared memory per TB. If we use spatial partitioning, each SM can hold only one TB from either kernel. However, if we could dispatch two kernels to one SM, then each SM can hold one TB from $K_1$ and one TB from $K_2$. Resource utilization and TLP increases in this circumstance. More importantly, the total throughput of the GPU may be improved if $K_1$ and $K_2$ can hide each other's stall cycles. For example, suppose $K_1$ is compute intensive and $K_2$ is memory intensive. Then $K_1$ may be scheduled to execute while $K_2$ is stalled on memory, reducing the overall stall cycles.

In conclusion, the current sharing mechanism, Spart, is increasingly limited with the advancement of GPU architectures. The inherit heterogeneity of kernels offers an opportunity to design a new sharing mechanism truly suitable for modern GPUs to improve resource utilization and total GPU throughput.

# 5. SIMULTANEOUS MULTIKERNEL

To enable sharing, SMK dynamically considers a general scenario where the GPU is currently executing a kernel $K$ which has exhausted at least one type of GPU resource. To allow a pending kernel, *newK*, to co-execute with $K$ on each SM, preemption must be supported to swap *a portion* of $K$'s context on each SM with a portion of the new context of *newK* so that the aggregated resources of both still fit in the SM. This approach is in contrast to Spart where *all* context of $K$ on a SM is swapped out so that the SM will host only *newK*. Hence, the first issue we consider is the design of *partial context switching*. This is immediately followed by the question of how much of $K$'s context should be swapped with *newK*, or how to allocate resources of an SM between $K$ and *newK*. This decision is critical to achieving high overall GPU throughput, while being fair to co-running kernels. Furthermore, the execution of warps *within* one SM can be imbalanced because the warp schedulers are not aware of multiple kernels. To this end, we discuss the proposed design of partial context switching, resource allocation with fairness, and warp scheduling with multiple kernels.

## 5.1 Partial Context Switching

In the base GPU, a centralized SM driver is in charge of

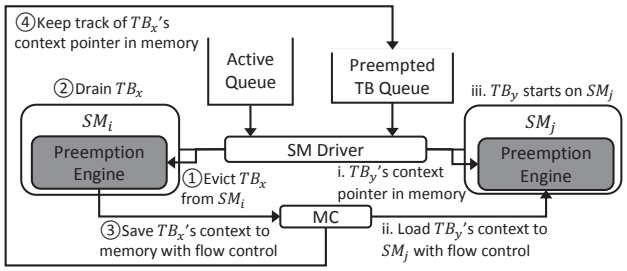



Figure 3: SMK-supported SM. Added components are shaded. SM Driver controls the SMs and the Preemption Engine. It can issue commands to Preemption Engine to swap TBs into a SM or swap TBs out of a SM.

receiving commands, such as launching kernels and memory operations, from the CPU; initializing the SM for kernel execution; and dispatching TBs of a kernel onto different SMs. In each cycle, the SM driver searches an Active Queue containing a sequence of TBs belonging to the kernel to find a candidate TB for dispatch.

We assume a general execution model where applications do not necessarily arrive at the GPU at the same time. The kernels of applications are issued to the GPU on a first-come-first-serve basis. When *newK* arrives, *K* (of different applications) is partially preempted such that *newK* can co-execute with *K*. This depends on how many resources are needed to dispatch TBs of *newK*. We choose to swap contexts of kernels in TB units in line with what the driver uses when dispatching *newK*. Hence, the context of *K* is saved to memory one TB at a time, until enough resources are released to host one TB of *newK*. For example, if only 10% of static resources are vacant in the SM before starting *newK* and one TB of *newK* requires 15%, then SMK swaps out several TBs of *K* to free at least 5% of static resources. Hence, we term our preemption mechanism *Partial Context Switching* (PCS).

The main distinctive feature of PCS is that preemption takes place without blocking the SM, i.e., the SM continues executing the remaining TBs of *K* while switching contexts. In Spart, an SM does not make progress during a context switch. PCS leads to not only forward progress in kernel execution during a context switch, but also less overhead. We next describe the steps of PCS.

We use the same base GPU execution engine as previous work [8][9], which covers the basic preemption and multitasking support. We discuss only the essential components relevant to our design. As shown in Figure 3, we add a Preemption Engine (PreEng) per SM to perform PCS. When the SM driver decides to swap $TB_x$ from $SM_i$, it sends this information to PreEng in $SM_i$ (①). Next, PreEng stops fetching new instructions for $TB_x$, and drains all currently executing instructions from its pipeline, including pending memory requests (②). Once draining is complete, PreEng initiates PCS by sending memory store requests directly to the memory controller for the context of $TB_x$, namely for register contents, shared memory values, barrier information and SIMT stack (③). Once PCS is complete, PreEng sends the memory address, a pointer, of $TB_x$'s context to the Preempted TB Queue (④), which maintains the TBs that the SM driver may select from and dispatch in the near future.

Swapping in a TB's context is symmetric to swapping out context, as illustrated by swapping in a $TB_y$ to $SM_j$ in Figure 3. The SM driver first reads memory locations of $TB_y$'s context from the Preempted TB Queue and sends it to PreEng in $SM_j$ (i). This PreEng then issues memory load requests to the memory controller with the pointer to $TB_y$'s context (ii). Once the context is fully loaded, $TB_y$ starts executing on $SM_j$ (iii). Because all executing instructions of a TB are drained before context switching and the resources of TBs are statically isolated, there will be no data hazard in context switching.

In summary, the goal of PCS is to use the least amount of context switch overhead to maximize resource utilization of each SM, while ensuring preempted kernels progress.

## 5.2 Resource Usage

With the capability of putting multiple kernels in a single SM, the next issue is to determine which kernel's TB should be dispatched to which SM to increase resource utilization. This process is done in the SM driver by examining resource usage of kernels and SMs. The driver makes dispatch decisions with the objective of improving overall throughput while being fair to all kernels. We note that these two goals can lead to contradictory decisions. For example, to achieve high GPU throughput, the SM driver could continuously dispatch a compute intensive kernel (IPC is high) while starving a memory intensive one. Hence, it is critical to enforce a fair-share policy during dispatch for SMK to be valuable.

In Spart, kernels are allocated to an integer number of SMs. Fairness is ensured by allocating an equal number of SMs to kernels. In SMK, however, there are multiple static resources to allocate, and different kernels stress resources differently. Some kernels may use registers more heavily than shared memory, and other kernels may do the opposite. Hence, it is not straightforward to "equally" divide resources.

We adopt Dominant Resource Fairness (DRF), a generalized metric for multiple resource usage [22]. It was originally proposed for job scheduling in clusters. The intuition behind DRF is that multi-resource allocation should be determined by a kernel's dominant resource share, i.e., the maximum share that a kernel requires of any resource. In GPUs, there are four kinds of resources allocated during TB dispatch of a kernel: registers, shared memory, number of active threads and number of TBs [1]. These resources limit the total number of TBs that can be dispatched. If kernel A mostly uses registers and kernel B mostly uses shared memory, then DRF tries to equalize A's share of registers with B's share of shared memory. The dominant resource share of the kernels ($rK$) and SMs ($rSM$) is computed as:

$$rK\ (rSM) = \max\{r(Register), r(Threads), r(SharedMemory), r(TB)\},$$
$$where \quad r(x) = \frac{x_{taken}}{x_{limit}}$$

Consider an example. Suppose one TB of kernel A uses

10% of the registers, 20% of the shared memory, 30% of the thread count limit and 3% of the TB count limit of one SM. The number of threads is the dominant resource for A. Every time kernel A has a TB to dispatch, it takes 30% of the thread count resource of the SM. The resource usage of a SM is calculated similarly. If 30% of the registers, 15% of the shared memory, 20% of the thread count limit and 10% of the TB count limit of the SM are taken, the resource usage of the SM is 30%. To calculate resource usage of a kernel, we use the amount of resources it has occupied on the GPU. Keeping track of the kernel and SM resource usage is trivial as the resource demand of each TB in the kernel is determined at compile time. The SM driver is also fully aware of the TB status in each SM. Hence, the SM driver can use this resource information to determine a fair allocation.

## 5.3 Fair Resource Allocation

Once kernel and SM resource usage are defined, the next issue is resource allocation with fairness. The objective is to equalize resource usage of each kernel (denoted as *rK*) as much as possible. This can be quantified as minimizing the difference between the maximum and minimum of *rK* of all kernels, which is termed the *range of rK*. This criteria is used by the SM driver when dispatching a TB onto an SM.

We first develop a naïve dispatch algorithm, termed *on-demand resource allocation*, which is a simple heuristic of the Knapsack problem. This algorithm forms the basis of the second algorithm, *resource partitioned algorithm*, which applies the same heuristic within a partition of resources.

### 5.3.1 On-Demand Resource Allocation

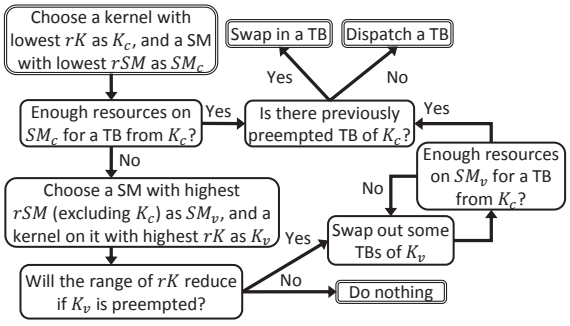



Figure 4: On-demand Resource Allocation.

On-Demand Resource Allocation allocates resources for one TB during dispatch. The approach identifies a victim TB on a SM and checks whether applying PCS to that TB would reduce the range of *rK*. The algorithm's flow chart is shown in Figure 4. The SM driver first selects a kernel with the lowest *rK* as a candidate kernel, $K_c$. Then, the driver selects an SM with the lowest resource usage as a candidate SM, $SM_c$. Next, the driver checks whether there are enough resources in $SM_c$ to receive a TB from $K_c$. If yes, the driver swaps in a TB from the Preempted Thread Block Queue, if there are any, or dispatches a new TB of $K_c$. The SM driver prioritizes previously preempted TBs over new ones to reduce the memory footprint of storing preempted TBs.

If there are not enough resources in $SM_c$, the SM driver continues to search for a victim SM, $SM_v$, with the highest *rSM* of all SMs. When calculating *rSM*, if kernel $K_c$ has TBs on $SM_v$, then this resource usage is excluded from *rSM*, such that a victim kernel other than $K_c$ can be found. A victim kernel $K_v$ is the one with the highest *rK* in $SM_v$. Then, the range of *rK* on $SM_v$ is calculated to check if preempting $K_v$'s TB would reduce the range. If so (implying preemption will improve fairness), a preemption is performed. TBs of $K_v$ are swapped out and one TB of $K_c$ is swapped in. Otherwise, no preemption is done.

Using the range of *rK* to determine when to do preemption is critical to the performance of the algorithm. Note that once the range is relatively small, indicating that the resource allocation between different kernels is fairly balanced, then preemption will not be performed. This criteria helps to stabilize the algorithm and throttles overly frequent preemption, as observed in our study.

### 5.3.2 Allocation with Resource Partitioning

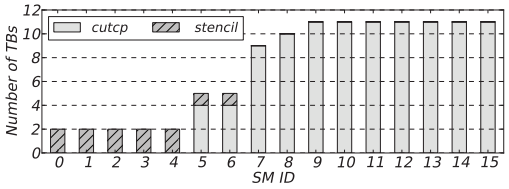



Figure 5: The TB distribution of *cutcp+stencil* with On-Demand Allocation.

On-demand Resource Allocation aims to achieve *global resource* balance between kernels. For example, if register usage is the dominant resource of a kernel, then that kernel's *rK* is calculated by summing register usage across all SMs divided by the total register capacity of all SMs. We have observed that the algorithm cannot control the local resource allocation within each SM. To realize SMK, kernels of complementary resource usage should share resources within one SM. However, the on-demand algorithm cannot enforce such sharing. Consequently, the algorithm often generates an allocation with many SMs, each running only one kernel, close to what Spart would produce. Figure 5 shows an example of running *cutcp+stencil* with On-Demand Resource Allocation. As shown in the figure, SM 0-4 run *stencil* exclusively, and SM 7-11 run *cutcp* exclusively. There are only 2 SMs sharing two kernels, limiting the effect of SMK. Note that the allocation scheme generated by the algorithm may not be unique. To achieve fairness in SMK, we propose to allocate the resources for each kernel *before* their TBs are dispatched. We term this strategy as resource partitioning, and the allocated resources for one kernel is called a resource partition.

To create resource partitions on one SM, the SM Driver applies the DRF policy [22]. It aims to equalize *rK* for all kernels on one SM. As depicted in Figure 6, partitioning starts with an empty SM where *rK* of each kernel is 0. Then the driver picks an initial kernel, say $K_1$, and updates $rK_1$ assuming that a TB of $K_1$ was dispatched. Next, a kernel with the lowest *rK*, say $K_2$, is picked and its *rK* is updated assuming that one more TB of $K_2$ was dispatched to the SM. This procedure iterates until no more TBs can be dispatched

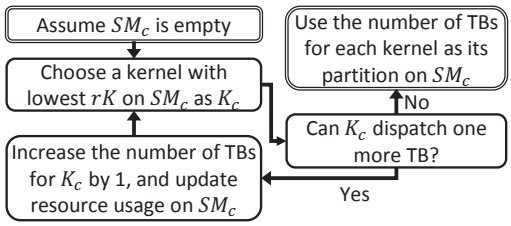



Figure 6: Calculating resource partitions on one SM.

to this SM. That is, at least one type of resource in the SM has been exhausted. At this time, the resource partitions of all kernels have naturally been created – the resources occupied by each kernel through this iterative procedure form the resource partitions for those kernels. The procedure always attempts to reduce the range of $rK$: On every iteration, a TB from the kernel with the lowest $rK$ is dispatched so its $rK$ moves toward the highest $rK$, contracting the range.

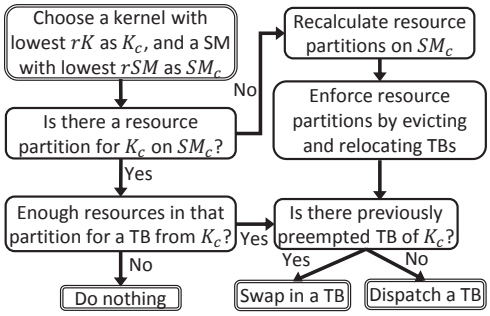



Figure 7: TB dispatch with resource partitioning.

The TB dispatch algorithm, which uses resource partitioning, is shown in Figure 7. The SM driver first identifies $K_c$ and $SM_c$ similar to On-demand Resource Allocation. Then, the SM Driver checks whether there is already a resource partition for $K_c$ on $SM_c$. If there is one and there is room in that partition, dispatch (swap in) a TB. If there is a resource partition for $K_c$ on $SM_c$ but there is no room in that partition, do nothing.

If there is no resource partition for $K_c$ on $SM_c$, the driver initiates resource partitioning on $SM_c$. The new partitions are enforced by evicting and relocating existing TBs on $SM_c$ to make TBs within one resource partition aligned. Relocation is done by swapping TBs out and swapping them in. Even with this overhead, we still observe good speedup over Spart. This preemption overhead can be mitigated with extra storage within a SM.

Figure 8 illustrates an example of resource partitioning. Assume the SM Driver needs to create resource partitions for two kernels on one SM, $K_1$ and $K_2$. The $rK$ of one TB of $K_1$ and $K_2$ are 10% and 6% respectively. Iteration 0 is the initial state when the SM is empty. In iteration 1, one TB from $K_1$ is selected to run in the SM, increasing $rK_{K1}$ to 10%. At this time, $K_2$ has lower $rK$, so its TB is chosen next to add to the SM in iteration 2, which increases $rK_{K2}$ to 6%. The dominant resource of the SM is the register file, and $rSM$ is increased to 13% (sum of register usage). Now $K_2$

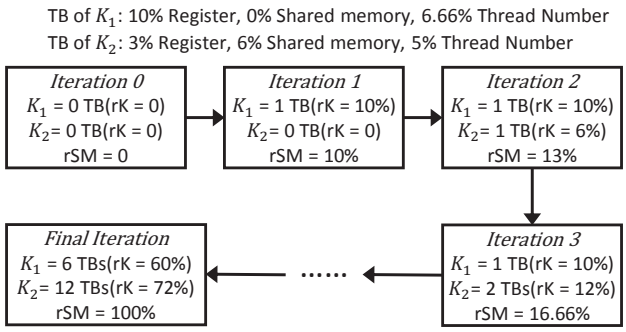



Figure 8: An example of resource partitioning for 2 kernels.

Table 3: Resource and performance fairness with fair resource allocation (higher is better).

| Pair Type | Resource | Performance |
|---|---|---|
| Compute+Memory | 0.94 | 0.49 |
| Compute+Compute | 0.94 | 0.64 |
| Memory+Memory | 0.88 | 0.55 |
| All | 0.92 | 0.54 |

still has the lowest $rK$, so in iteration 3, one TB from $K_2$ is added to the SM, and $rK_{K2}$ becomes 12%. At this iteration, the dominant resource of the SM is the number of threads, and $rSM$ is increased to 16.66%. Although not shown in iteration 4, one TB from $K_1$ will be added to the SM. This process iterates until $rSM$ is 100%, i.e., when SM resource, active thread count in this case, is fully saturated. At this time, resource partitions for $K_1$ and $K_2$ are formed: $K_1$ can dispatch up to 6 TBs and $K_2$ can dispatch up to 12 TBs.

With partitioned resource allocation, every SM has a dedicated partition for each kernel, guaranteeing that multiple kernels can co-execute within one SM. Our experimental results show that this allocation greatly improves GPU throughput, demonstrating the benefit of SMK.

## 5.4 Fair Allocation on Dynamic Resource via Warp-Scheduling

Both on-demand and partitioned resource allocation target static resources, such as registers. At runtime, warps of TBs from different kernels are scheduled by the warp scheduler to use computing cycles. There are many warp scheduling policies [23][24]. However, these policies are all designed to improve performance of a single kernel. When multiple kernels co-execute, we have observed that the kernels can incur severe contention for computing cycles, even though the kernels have a fair share of static resources. For example, a compute-intensive kernel may monopolize computing cycles, slashing execution opportunities for other concurrent kernels. A memory-intensive kernel may fill the miss status holding register (MSHR), blocking requests from other kernels that might be memory non-intensive but stalled on memory. Warps of the stalled kernels cannot resume execution to hide latencies of other warps. Hence, fair share of static resources may not result in fairness in performance achieved by different kernels.

To quantify fairness, we adopt the metric defined in [25], which is the smallest normalized IPC (normalized to iso-

lated execution) divided by the largest normalized IPC of all kernels. We extend this metric to static resource fairness, which is the smallest $rK$ divided by the largest $rK$. Table 3 shows the averages of resource and performance fairness using the resource partitioning allocation, over different pairings of kernels (e.g., compute- and memory-intensive kernels in the first row). A value of 1 indicates perfect fairness, and a value of 0 means complete starvation of one kernel. As we can see, the algorithm can achieve good fairness for static resource allocation, but the performance fairness values are in general quite low and do not even have a similar trend as resource fairness. Hence, we need to further enhance the algorithm with allocating the dynamic resource, i.e., the computing cycles.

To achieve performance fairness, we define a fair allocation of computing cycles as one where a kernel has a share of cycles that is proportional to the amount required when the kernel is executed exclusively on one SM. The proportion is determined by the ratio of resources allocated in SMK (i.e., number of TBs) to resources used in isolated execution. Intuitively, if a kernel has $T$ TBs per SM when run in isolation, and $x\%$ of the cycles are used to issue instructions, then the percentage of cycles the kernel should be allocated in SMK, denoted as $C_k$, is $x\% \times \frac{S}{T}$, where $S$ is the number of TBs allocated in SMK using resource partitioning. The sum of $C_k$ for all kernels should not exceed 100%. Hence, each kernel should get its own share proportionally, which is defined as $Quota_k$:

$$Quota_k = \frac{C_k}{\sum_{for\ all\ k} C_k}$$

To implement this allocation, we need to obtain $x$ and $T$ from isolated execution of a kernel. This can be done by assigning a dedicated SM to each kernel for profiling. The benefit of this profiling outweighs the loss from having fewer SMs for SMK, as indicated by our experiments. Tracking computing cycles of kernels directly is impractical because the execution of kernels are overlapped. Instead, we use Warp Instructions per Cycle (WIPC) per warp scheduler over a period of time to approximate computing cycles. Then, during each epoch of execution, each warp scheduler allocates $Quota_k \times Epoch\_length$ number of instructions for kernel $k$. This allocation is distributed evenly among all warp schedulers since TBs are evenly distributed across warp schedulers when they are dispatched to an SM. When a warp scheduler issues an instruction from a kernel, its quota is decremented by 1. If one kernel's quota reaches zero, the warp scheduler will stop issuing new instructions from that kernel. If all quotas reach zero, new quotas will be calculated using the most recent WIPC, and assigned to each kernel.

For example, suppose there are two kernels, $K_1$ and $K_2$. Both can run 8 TBs in isolated execution. In SMK, suppose $K_1$ has 2 TBs and $K_2$ has 4 TBs in one SM. WIPCs per warp scheduler for $K_1$ and $K_2$ are 0.4 and 0.5 respectively. Then, $C_1 = 0.1$, and $C_2 = 0.25$. And so $Quota_1 = \frac{0.1}{0.35}$ and $Quota_2 = \frac{0.25}{0.35}$.

The proposed allocation of cycles ensures that the number of issued instructions is related to the number of TBs of the kernel present in an SM. As a result, warps of kernels can be relatively fairly scheduled by warp schedulers. We use a linear estimation of execution time in this design for simplicity, although this may overestimate kernel performance. However, it does not impact the allocation much because all kernels are overestimated. The objective here is to achieve a balance of core compute cycles among kernels, instead of exact measurement.

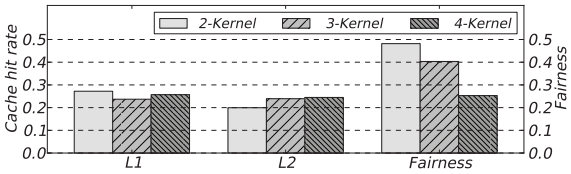



Figure 9: L1/L2 cache hit rate and fairness for different number of kernels coexisting within one SM. The 4 kernels used are *lbm*, *sgemm*, *stencil*, and *tpacf*.

## 5.5 Increasing Number of Concurrent Kernels

The algorithms described in Sections 5.3 and 5.4 are generally applicable to arbitrary number of concurrent kernels per SM. However, there could be potential issues with more kernels sharing an SM. The first issue is the possibility of higher contention on the L1/L2 cache. We observed that the L1/L2 cache hit rate does not always decrease with more sharing kernels, as shown in Figure 9 for one example kernel mix. This situation happens because the warp schedulers can capture the intra-kernel locality and keep their corresponding working set in L1/L2 cache. Hence, the overall L1 hit rate does not necessarily decrease. The second issue is the possibility of resource fragmentation in registers and shared memory due to the disparity in kernel resource demand. Having more kernels may lead to more internal fragmentation of resources. Fortunately, our Resource Partitioning Allocation guarantees that there is no fragmentation within one kernel's resource partition by relocating non-aligned TBs. Vacancies occur only between adjacent kernel allocations. Hence, in our evaluation, we did not observe significant fragmentation that would threaten the performance of kernels.

However, we observe a drop in fairness as the number of concurrent kernels per SM increases. This drop is due to the performance estimation by profiling SMs, and linear scaling has some inaccuracies. This could cause over-estimation or under-estimation of the *Quota* for each kernel. If the warp scheduling policy is not fair for warps (in this case, the greedy-then-oldest policy), it may favor some kernels over others, resulting in over-performing kernels and under-performing kernels. As the number of concurrent kernels per SM increases, the performance of the kernel with least share of execution becomes worse because there are more kernels taking its share of execution, resulting in worse fairness. A more accurate performance estimation, e.g., offline profiling, or a fair warp scheduling policy, e.g. round-robin, can be used to mitigate this problem. Our study shows that 2 kernels per SM gives the best fairness and turnaround time.

If we limit the number of kernels within one SM to two,

*rK* can be extended to include the number of running SMs for a kernel. This will produce equal number of SMs among sharing kernels because the number of SMs are also taken into account when considering the fairness. For example, for 3 kernels, $K_1$, $K_2$ and $K_3$, running on 3 SMs, the generated sharing scheme may be ($K_1$, $K_2$) on $SM_1$, ($K_2$, $K_3$) on $SM_2$, and ($K_1$, $K_3$) on $SM_3$.

Table 4: Simulation parameters for GPGPU-Sim.

| GPU Parameter | Value | SM Parameter | Value |
|---|---|---|---|
| Core Clock | 1216MHz | Registers | 256KB |
| Memory Clock | 7GHz | Shared Memory | 96KB |
| Number of SMs | 16 | Threads | 2048 |
| MC | 4 | TB limit | 32 |
| Sched. Policy | GTO | Warp Scheduler | 4 |

# 6. EXPERIMENTAL EVALUATION

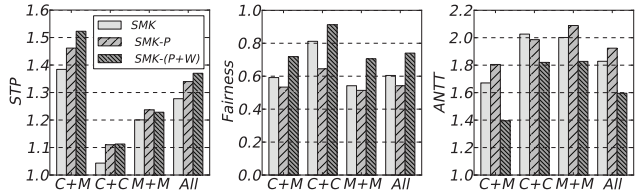
## 6.1 Methodology

We evaluate our design using the latest version of GPGPU-Sim [11], a widely adopted GPU simulator. The simulation parameters in Table 4 are from the specifications of Nvidia Geforce GTX980 [26] to reflect that each SM is larger in recent architectures. We used 10 benchmarks from Parboil [6] and we enumerate all pairs, a total of 45, to simulate multi-programmed workloads. *bfs* is excluded because it uses only a small portion of GPU resources, and can coexist with other kernels with little interference of performance. We ran the benchmarks with the largest datasets in Parboil, except for the experiments that evaluate preemption overhead. When measuring this overhead, we used small data sets to extenuate the effect of preemption. We modified GPGPU-Sim to support running multiple kernels on the same SM. We assume that the registers and the shared memory in each SM are linearly addressed as described in [27].

We first evaluate the different variations of SMK: On-demand Resource Allocation (**SMK**) and Allocation with Resource Partitioning (**SMK-P**). We apply Dynamic Resource Allocation via Warp-Scheduling on top of SMK-P(**SMK-(P+W)**). The epoch length for SMK-(P+W) is 10k cycles. Then, we compare the most efficient approach, SMK-(P+W), with spatial partitioning (**Spart**), which we implemented according to the description in [9]. In the evaluation, we used the three metrics proposed in [25]: system throughput (STP), fairness, and average normalized turnaround time (ANTT). System throughput for a pair of benchmarks is the sum of the normalized IPCs of the benchmarks (normalized to the IPC of isolated execution). Similarly, turnaround time is the arithmetic average of normalized turnaround time. Fairness (a value between 0 and 1) is the smallest normalized IPC divided by the largest normalized IPC of all kernels.

We ran 2M cycles for each pair in our evaluation, since according to [10], the results are accurate when the simulation is longer than 1M cycles. If one program ends before 2M cycles, it is re-executed. If the benchmark has multiple kernels or the kernel is executed multiple times, we add the instructions and cycles of all kernels to calculate the IPC.

For comparison, the 45 pairs of benchmarks are divided into three groups. We report the average in each group: group "C+C" has pairs where both benchmarks are compute intensive (such as *cutcp*, *mri-q*, *sgemm* and *tpacf*), group "M+M" has pairs where both benchmarks are memory intensive (such as *histo*, *lbm*, *mri-g*, *sad*, *spmv* and *stencil*), and group "C+M" has pairs where one benchmark is compute intensive and the other is memory intensive. We only show the details of STP for the "C+M" group, as the results are most interesting for this group.




(a) STP (b) Fairness (c) ANTT

Figure 10: Comparison of the SMK designs.

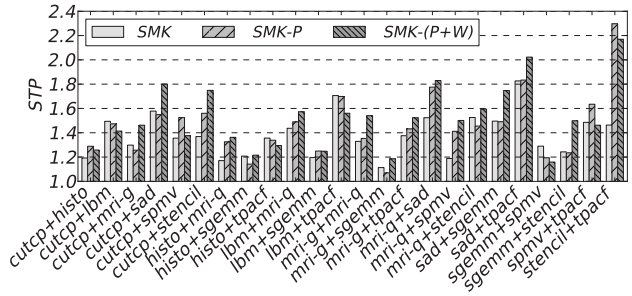



Figure 11: STP for kernel pairs in the "C+M" group.

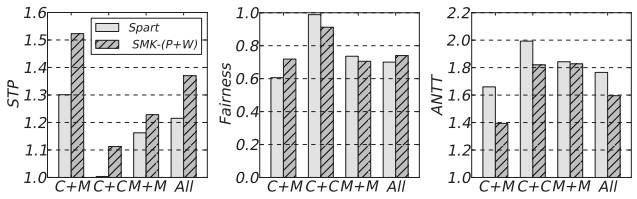
## 6.2 Results of SMK Designs

Figures 10 and 11 show the results for the different variations of SMK. As SMK is designed for sharing kernels of complementary resource usage, it is most effective for the "C+M" group. On average, SMK, SMK-P, SMK-(P+W) improves throughput over isolated kernel execution by 38%,46%, and 52%, respectively. The overall average of the three designs are 28%, 34%, and 37%. SMK-P improves over SMK because more within-SM sharing is enforced through resource partitioning. In most cases, SMK-(P+W) creates more overlapped execution by balancing the execution of kernels, further improving the throughput. However, there are cases that the opportunities of overlapping are limited because SMK-(P+W) may limit the execution of one kernel to achieve fairness.

From the fairness standpoint (Figure 10b), SMK-(P+W) is also the best because it ensures that each kernel in one SM receives a fair chance of executing. In the figure, the closer the bars are to 1.0, the better the fairness, indicating that the performance changes with and without SMK across different kernels are roughly the same. On average, SMK, SMK-P and SMK-(P+W) achieve fairness of 0.6, 0.54 and 0.74. SMK-(P+W) is very effective in improving fairness among co-running kernels. For example, we observed that

fairness values of SMK-P in *mri-g+sgemm* and *sad+sgemm* (not shown in the figure) are extremely low – 0.121 and 0.124 – but the values are improved significantly to 0.81 and 0.56 with SMK-(P+W).

For ANTT, a value close to 1 is better, meaning that the response time in SMK is closer to isolated execution. If throughput and fairness are both good, then ANTT (Figure 10c) should also be good. If one of them is weak, then ANTT will be weak too. Since SMK-(P+W) is best for both throughput and fairness, its ANTT value is also the lowest among the three. SMK-P, on the other hand, is weak in fairness, and thus, also weak in ANTT.




(a) STP (b) Fairness (c) ANTT

Figure 12: Comparison of SMK-(P+W) and Spart.

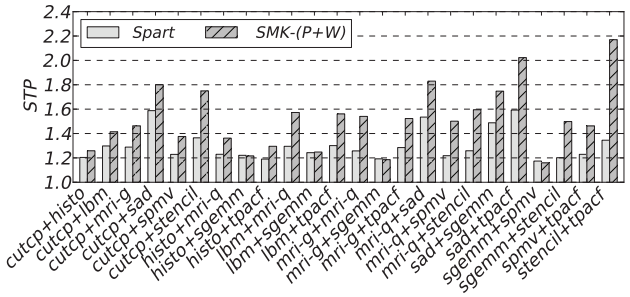



Figure 13: STP of SMK-(P+W) vs. Spart for the "C+M" group.

## 6.3 Comparison with Spatial Partitioning

Figures 12 and 13 compare SMK-(P+W) with Spart [9]. In general, SMK-(P+W) outperforms Spart for most pairs, with the highest improvement being 61.3%. In the cases where Spart outperforms SMK-(P+W), the throughput difference is less than 5%.

SMK-(P+W) does best in the "C+M" group since compute-intensive and memory-intensive kernels naturally compensate each other especially in execution cycles. For example, in *lbm+tpacf*, the system throughput of SMK-(P+W) is larger than Spart by 20% (see Figure 13). This improvement comes from two factors. First, memory utilization is better with SMK-(P+W) because the TBs of the memory-intensive kernel, *lbm*, are distributed evenly to SMs. This distribution fully utilizes the SM memory access components (like MSHR). Second, the compute-intensive kernel, *tpacf*, has many low-latency instructions that hide the long memory latency of the memory-intensive kernel. On average, the "C+M" group improves throughput by 52% over the isolated execution, in contrast to 30% achieved by Spart.

The improvement of SMK-(P+W) over Spart can be even larger when one kernel of a pair is cache-sensitive. Specifically, in Spart, each SM is assigned with one kernel, and the maximum number of TBs of that kernel. However, this behavior may cause thrashing in the cache when TBs compete for cache space. For a compute-intensive kernel, the cache resource is wasted because it utilizes the shared memory to cache data. With SMK, cache-sensitive kernels (such as *sad* and *stencil*) and compute-intensive kernels can be effectively paired so that the cache-sensitive kernel occupies most of the cache (achieving a high hit rate) without affecting the other kernel. The most significant improvements in our evaluation come from this effect of complementary pairing. For example, *stencil* is cache sensitive, and it has very good performance when paired with a cache-insensitive kernel.

Kernels in the "C+C" group also benefit from SMK (not shown due to space limits). The thread-level parallelism is increased by fully utilizing GPU resources because there are more opportunities to interleave memory access and other instructions. The pair *cutcp+mri-q* is such an example that has a 15% improvement over Spart.

However, for kernels in the "M+M" group, sharing an SM does not increase thread-level parallelism and the memory subsystem remains a bottleneck. SMK may degrade throughput slightly in a few cases, but the overall average remains higher than Spart by 5.1%, as shown in Figure 12a.

Figure 12b and 12c show that our design also improves fairness and turnaround time over spatial partitioning. SMK-(P+W) does best due to fine-grained control over warp execution; turnaround time is much improved and fairness is preserved.

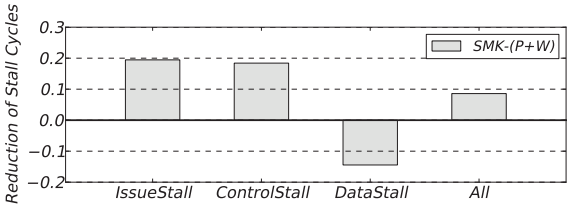
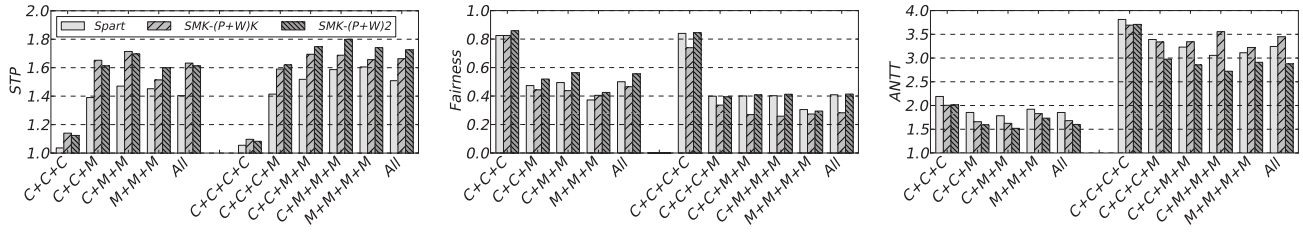



Figure 14: Reduction of stall cycles over Spart.

## 6.4 Stall Cycles

Figure 14 shows the percentage of different types of stall cycles for "C+M" pairs. Stalls can occur under three conditions: the instruction pipeline is full, there are control hazards or there are data hazards. As we can see, SMK significantly reduces the stalls caused by instruction pipeline. This is because SMK utilizes the instructions from different kernels to have more opportunities to issue different kinds of instructions. For example, if the SM is executing a compute intensive kernel, the instruction pipeline will be busy most of the time, leaving the load/store unit idle. In the case of SMK, the instructions of another kernel can be issued to utilize the load/store unit, reducing pipeline stalls. SMK also reduces control stalls by interleaving fetching and executing. However, SMK has more data hazard stalls than Spart because it has more pending instructions when fully utilizing the pipeline. With more pending instructions, there are more




(a) System Throughput(STP) (b) Fairness (c) ANTT

Figure 15: SMK-(P+W) vs. Spart for three and four kernels.

entries in the scoreboard, and more chances for data hazard stalls. Overall, SMK-(P+W) reduces issue stalls by 19.5%, control stalls by 18.4%, and increases data stalls by 14.5%, resulting in an overall stall reduction of 8.6%.

## 6.5 Preemption Overhead

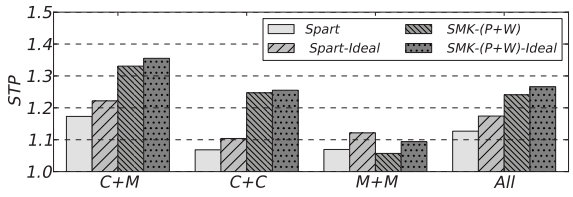



Figure 16: Throughput without preemption overhead.

We also examined the overhead of preemption in SMK-(P+W) compared with Spart. Preemption not only takes time to save context to memory, but also puts pressure on the memory subsystem. We evaluated this overhead by running an ideal simulation, where the cost of data transfer to store/retrieve the context is set to zero (no cost). We denote the no-cost preemption case as Spart-Ideal and SMK-(P+W)-Ideal, as shown in Figure 16. We collected these results for short running programs (using small data sets) which are more sensitive to preemption overhead than longer running ones.

As the figure shows, preemption has less impact in SMK than Spart. The difference between Spart and Spart-Ideal is 5%, while the difference between SMK-(P+W) and SMK-(P+W)-Ideal is 2%. SMK-(P+W) has less overhead because it interleaves preemption with normal execution. TBs that are preempted behave similarly to memory-intensive kernels. Moreover, the results also show that, on average, SMK-(P+W) with overhead indeed performs better than Spart-Ideal, demonstrating the effectiveness of our proposed design.

## 6.6 Three or Four Kernels

To study the scalability of our schemes, we evaluate the performance of running three or four kernels. We randomly select 56 combinations of 3-kernel workloads and 70 combinations of 4-kernel workloads. The selected combinations cover all possible mixes of compute and memory intensive kernels. We experiment with two schemes, **SMK-(P+W)K**, which puts as many kernels as possible into one SM, and **SMK-(P+W)2**, which limits the number of kernels in one SM to 2. As shown in Figure 15a, STP scales well for 3 and 4 kernels. The overall improvements are 14.8% and 14.3% over Spart for 3 and 4 kernels respectively. SMK-(P+W)2 has similar STP to SMK-(P+W)K for 3 kernels, but scales better for 4 kernels. Hence, for throughput, it is better to have more kernels running concurrently.

Fairness is a bit lower when more than 2 kernels are allocated to each SM, as shown in Figure 15b. It is more difficult to control the progress of multiple kernels than two kernels. And we have discussed the reason in Section 5.5. The fairness of SMK-(P+W)K is the worst among all schemes.

Figure 15c shows the ANTT results. As the number of kernels increases, ANTT increases sublinearly to the number of kernels. SMK-(P+W)2 has the best turnaround time among all schemes.

In summary, SMK has similar STP improvement over Spart and good fairness in the 2-kernel, 3-kernel and 4-kernel cases. We conclude that SMK is a scalable design.

## 6.7 Hardware Overhead

To implement SMK, the SM driver needs to be extended with new control logic to implement the following: (1) The TB dispatch algorithm; (2) Signals to control the preemption engine. Finally, the warp scheduler needs to be extended to implement the cycle quotas for sharing kernels.

Partial Context Switching has similar overhead to full context switch [9], which requires hardware such as preempted TB queue etc. to implement preemption. For On-demand Resource Allocation, the overhead is mostly logic, and the information that the algorithm needs is already present in current GPUs. For Resource Partition and Warp Scheduling, several counters are needed to store the resource partition and quota information.

Specifically, Resource Partition needs 7 counters to record the resource partition range of registers, shared memory and threads, and the number of TBs for each kernel on each SM. For a GPU with 16 SMs, each supporting 4 kernels, for example, the total overhead is 448 ($7 \times 4 \times 16$) counters. Similarly, the Warp Scheduling scheme needs 4 counters (1 for profiling, 1 for tracking epoch, 2 for storing quota) to track profiling and quota information. Hence the total overhead is 256 counters. Overall, the majority of overhead is in logic, and the storage overhead is only about 2.8KB if the counters are 32-bit.

# 7. CONCLUSIONS

State-of-the-art GPUs do not yet support preemptive scheduling, which is essential for modern systems. Previous work on this issue provides preemption mechanisms and strategies to share GPUs, but still leaves GPUs under-utilized. We propose a novel design to enable fine-grain sharing of multiple co-executing kernels for GPUs. Moreover, we propose several strategies to fully exploit the potential of this mechanism. We not only maintain resource fairness among kernels, but also ensure that kernel execution is done in a fair manner. Our results show that our design significantly improves system throughput with good fairness.

## 8. ACKNOWLEDGEMENTS

We thank the anonymous reviewers. This work is partially sponsored by the National Basic Research 973 Program of China (No. 2015CB352403), the National Natural Science Foundation of China (NSFC) (No. 61261160502, No. 61272099), the Program for Changjiang Scholars and Innovative Research Team in University (IRT1158, PCSIRT), the Scientific Innovation Act of STCSM (No. 13511504200), and the EU FP7 CLIMBER project (No. PIRSES-GA-2012-318939). This work is supported in part by NSF grants CNS-1012070, CNS-1305220, CCF-1422331 and CCF-1535755. This work was carried out while Zhenning Wang visited the University of Pittsburgh on a CSC scholarship.

## 9. REFERENCES

[1] Nvidia, "Programming Guide," 2014.

[2] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," in *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pp. 917–924, ACM, 2003.

[3] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU Cluster for High Performance Computing," in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, IEEE Computer Society, 2004.

[4] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce Framework on Graphics Processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pp. 260–269, ACM, 2008.

[5] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU Concurrency with Elastic Kernels," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pp. 407–418, ACM, 2013.

[6] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[7] 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*, pp. 260–271, Feb 2014.

[8] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative Preemption for Multitasking on a Shared GPU," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 593–606, ACM, 2015.

[9] 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, IEEE Press, 2014.

[10] J. Adriaens, K. Compton, N. S. Kim, and M. Schulte, "The case for GPGPU spatial multitasking," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pp. 1–12, Feb 2012.

[11] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 163–174, April 2009.

[12] T. Bradley, "Hyper-Q example," 2012.

[13] NVIDIA, "Sharing a GPU between MPI processes: multi-process service(MPS)," 2012.

[14] 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 *ICS' 15*, 2015.

[15] G. Wang, Y. Lin, and W. Yi, "Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU," in *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pp. 344–350, Dec 2010.

[16] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, "Fine-grained resource sharing for concurrent GPGPU kernels," in *4th USENIX Workshop on Hot Topics in Parallelism (HotPar). Berkeley, CA*, 2012.

[17] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating System Abstractions to Manage GPUs As Compute Devices," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 233–248, ACM, 2011.

[18] K. Menychtas, K. Shen, and M. L. Scott, "Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pp. 301–316, ACM, 2014.

[19] Kato, Shinpei and Lakshmanan, Karthik and Rajkumar, Raj and Ishikawa, Yutaka, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *Proc. USENIX ATC*, pp. 17–30, 2011.

[20] K. Menychtas, K. Shen, and M. L. Scott, "Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack.," in *USENIX Annual Technical Conference*, pp. 291–296, 2013.

[21] S. W. Kim, C.-l. Ooi, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar, "Reference idempotency analysis: A framework for optimizing speculative execution," in *PPoPP '01*, pp. 2–11, ACM, 2001.

[22] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types.," in *NSDI*, vol. 11, pp. 24–24, 2011.

[23] 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, ACM, 2011.

[24] 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, IEEE Computer Society, 2012.

[25] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE micro*, vol. 28, no. 3, pp. 42–53, 2008.

[26] NVIDIA, "NVIDIA Geforce GTX980 Whitepaper," 2014.

[27] 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, IEEE Computer Society, 2007.