Dynamic Resource Management for Efficient Utilization of Multitasking GPUs

Jason Jong Kyu Park
University of Michigan 密歇根大学
Ann Arbor, MI
jasonjk@umich.edu

Yongjun Park
Hongik University 弘益大学
Seoul, Korea
yongjun.park@hongik.ac.kr

Scott Mahlke
University of Michigan
Ann Arbor, MI
mahlke@umich.edu

Abstract

As graphics processing units (GPUs) are broadly adopted, running multiple applications on a GPU at the same time is beginning to attract wide attention. Recent proposals on multitasking GPUs have focused on either spatial multitasking, which partitions GPU resource at a streaming multiprocessor (SM) granularity, or simultaneous multikernel (SMK), which runs multiple kernels on the same SM. However, multitasking performance varies heavily depending on the resource partitions within each scheme, and the application mixes. In this paper, we propose GPU Maestro that performs dynamic resource management for efficient utilization of multitasking GPUs. GPU Maestro can discover the best performing GPU resource partition exploiting both spatial multitasking and SMK. Furthermore, dynamism within a kernel and interference between the kernels are automatically considered because GPU Maestro finds the best performing partition through direct measurements. Evaluations show that GPU Maestro can improve average system throughput by 20.2% and 13.9% over the baseline spatial multitasking and SMK, respectively.

两种形式的 multitasking

问题所在

CCS Concepts **•Computer systems organization → Single instruction, multiple data; •Hardware → On-chip resource management; •Software and its engineering → Multiprocessing / multiprogramming / multitasking**

Keywords Graphics Processing Unit; Multitasking; Resource Management

1. Introduction

The single instruction multiple thread (SIMT) programming model used by CUDA [18] and OpenCL [12] unlocked the computing capability of GPUs for general-purpose applications. Many supercomputers in the TOP500 List [2] and the Green500 List [1] are already composed of GPUs due to their high performance for data-parallel applications and energy efficiency. GPUs are also readily available in cloud computing services like Amazon Web Services [4]. Supercomputers, cloud services, and data centers with GPUs will benefit significantly with shared GPUs because resource sharing is critical to efficient resource utilization in these environments [29].

To meet such demand, multitasking on GPUs began to receive a wide attention from both academia and industry. Earlier attempts [6, 11, 23] were made on the software level to provide a notion of fairness when multiple processes are trying to share a GPU. More recent studies from academia have explored the possibility of alternative preemption techniques specific to the GPUs to enable low overhead multitasking [21, 26]. Industry is also moving in a similar direction. Hyper-Q [15] in Nvidia's Kepler architecture enables concurrent execution of independent kernels on a single GPU with multiple independent queues. However, it employs leftover policy, where only remaining resources from running kernels are assigned to newly launched kernels. It mainly benefits kernels that do not fully utilize GPUs. Nvidia also introduced Multi-Process Service [16], which is a software support for MPI applications to multitask on a single GPU.

Spatial multitasking [3], which divides resources at the streaming multiprocessor (SM) granularity, was first studied to partition shared GPUs among multiple kernels. Recently, simultaneous multikernel (SMK) [30] or intra-SM slicing [32] have been proposed, which shares an SM between multiple kernels similar to simultaneous multithreading (SMT) on CPUs. However, neither is superior over the other because their performance varies depending on the resource partitions within each scheme and the application mixes.

ASPLOS '17 April 8–12, 2017, Xi'an, China.
© 2017 ACM. ISBN 978-1-4503-4465-4/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3037697.3037707

两种模式在什么情况下效率更好

SMK performs better than spatial multitasking when kernels running on the same SM have different characteristics: (1) when resource requirements of kernels are different so that more threads can be launched on an SM with SMK, or (2) when highly utilized functional units are different so that SMK can interleave instructions from the kernels with low contention. On the other hand, spatial multitasking has advantage over SMK when the co-running kernels interfere with each other significantly especially due to load-store units or L1 cache. In such cases, running these kernels separately on different SMs can be more effective by avoiding contention. Because spatial multitasking and SMK have their own advantages and disadvantages, a resource partitioning scheme for multitasking GPUs should be able to exploit both.

Determining the best performing resource partition is further complicated because many resource partitions are available for each scheme. Moreover, multikernel execution entails more difficulties in predicting the performance because of the interference between kernels. Without addressing these problems, the benefits from multitasking GPUs are limited.

To that end, we propose GPU Maestro, which addresses these questions to implement a dynamic resource management to efficiently utilize multitasking GPUs. We first propose a lightweight dynamic scheduling mechanism, which utilizes a direct measurement of existing performance counters on GPUs. Because complex interactions between kernels are incorporated in the direct measurement, GPU Maestro avoids the performance prediction errors when extrapolating single kernel execution profiles. The key idea of GPU Maestro is that GPUs are composed of multiple SMs, where each SM can be allocated differently and monitored for performance. In each epoch, GPU Maestro tests the performance of different configurations, whose results are used to find the best performing resource partition. The selected resource partition will be used in the next epoch, and a small subset of SMs are used to test different partitions again. After a few epochs, GPU Maestro converges to the best performing resource partition for the given kernel combination.

另外两个需要关注的问题：

While resource partitioning is an important problem to solve, two additional challenges arise when sharing an SM between kernels: (1) a resource fragmentation problem, where a thread block cannot be scheduled because resources are available in smaller chunks although there are enough resources on an SM in total, and (2) a starvation problem from the greedy-then-oldest (GTO) warp scheduler, where a newly launched kernel has lower probability of issuing instructions.

以上两个问题的解决方法

GPU Maestro solves the resource fragmentation problem on SMK GPUs using 2-way resource allocation, which forces thread blocks from the same kernel to allocate resources consecutively in the opposite directions similar to how a stack and heap grow in the opposite directions in a process's virtual memory for CPUs. GPU Maestro solves the starvation problem by adopting a kernel-aware scheduling. We show that a simple loose round-robin kernel-aware scheduling performs the best among multiple possible candidates.

We make following contributions in this paper:

- We show that system performance of multitasking GPUs can vary depending on the application mixes. We illustrate when SMK performs better than spatial multitasking, and vice versa. Furthermore, we show the difficulties in predicting multitasking performance because of dynamism within a kernel and interference between kernels.
- We propose GPU Maestro, which dynamically manages resource partitioning on multitasking GPUs to maximize the system performance. We show a lightweight implementation for GPU Maestro, which considers both dynamism and interference by monitoring existing performance counters for different allocations with a subset of SMs.
- We present how the resource fragmentation problem can manifest on SMK GPUs. GPU Maestro solves resource fragmentation by using 2-way resource allocation, which restricts how a kernel allocates and releases resources when launching or preempting a thread block.
- We study the interaction between warp scheduling and SMK GPUs, and show that starvation can negatively impact the system. We show that kernel-aware warp scheduling is critical to avoiding starvation, and a simple round robin scheduling of kernels improves system performance better than other complex scheduling methods.

2. Background

2.1 GPU Terminology

Programmers rely on a single instruction multiple thread (SIMT) model to write the *kernel* code, which is a parallel code section that runs on GPUs. In the SIMT programming model, a *thread* is the basic unit of execution. Threads are hierarchically grouped into a *thread block*, and further into a *grid*. The notion of a thread block is important in the SIMT programming model because threads within a thread block can share a fast, on-chip scratchpad memory called *shared memory*. Moreover, threads within a thread block can use explicit barrier operations for synchronization. The thread block is also a basic scheduling unit in GPU hardware. Communications between thread blocks can be done using atomic operations, however, having too many atomic operations can degrade performance as they are serialized by the GPU hardware. In the Nvidia's GPU hardware, there is another group of threads called a *warp*. Threads in a warp execute concurrently similar to SIMD. While a warp is not an exposed concept to the GPU programming model, many GPU kernels exploit the warp size to further optimize the

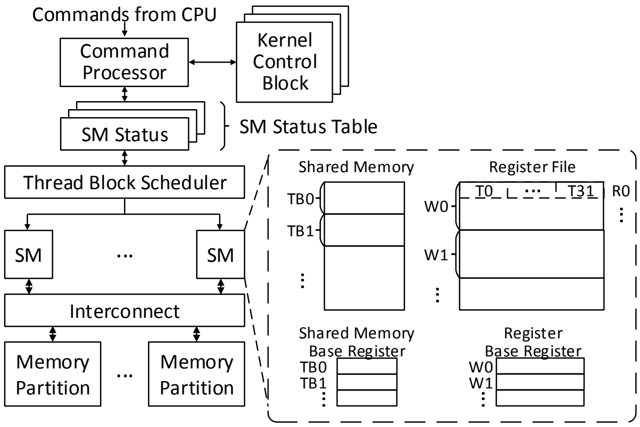


Figure 1: A baseline GPU architecture with multitasking support. TB denotes a thread block, and W denotes a warp.

code. Typically, a warp consists of 32 threads in Nvidia GPUs.

SMs are similar to CPU cores as warps from the scheduled thread blocks fetch, decode, and execute the instructions. In the Maxwell architecture [17], there are two private L1 caches, namely a unified L1/texture data cache, and a read-only constant cache. Starting from the Kepler architecture [15], global memory accesses bypass the L1D cache by default, and programmers have to explicitly use LDG instructions to cache global memory accesses. An SM also has one scratchpad memory called shared memory as previously discussed.

2.2 GPU Architecture

Figure 1 illustrates the baseline GPU architecture with multitasking support. At the top, a command processor is shown, which receives the commands from the CPU and controls the GPU accordingly, and updates memory-mapped registers when the command is done. In the middle, the thread block scheduler is depicted with control information structures. At the bottom left, SMs and memory partitions are shown with an interconnect. At the bottom right, shared memory and register file allocation within an SM is shown. Both shared memory and register file are multi-banked structures although they are shown as one structure for simplicity. The thread block scheduler determines how many thread blocks from each kernel will be launched on each SM. A thread block is preempted if the thread block scheduler decides to reduce the number of thread blocks running from one kernel to allow thread blocks from other kernels to run. The thread block scheduler also launches a thread block with the chosen partition when an SM has enough resources to accept a new thread block. It has to remember the preempted thread blocks so that it can relaunch those thread blocks later.

When a kernel is launched, the related information will be stored at a kernel control block (KCB), whose name is borrowed from process control block (PCB) in the operating system [24]. Similar to PCB, the KCB stores the kernel-specific information, which is necessary to launch and run a kernel on the GPU. The KCB contains which CPU process launched the kernel, information on memory management structures (e.g. page table), the grid size, the number of executed thread blocks, the resource requirement of a single thread block, and runtime information such as instructions per cycle (IPC) and average thread block execution cycles. The runtime information can be used to enable low overhead preemption [21]. To support multitasking on GPUs, the KCB is extended to multiple entries. Note that current generation GPUs are likely to have similar structures already if they support Hyper-Q, which allows multiple independent kernels from the same process to concurrently run on the GPU.

An SM status table (SMST) stores the current status of SMs. An SM status includes kernels executing on the SM, the state of each thread block (free, running, or preempting), the mapping between thread blocks and kernel, and remaining resources on the SM such as registers, shared memory, threads, and thread blocks.

Because there are multiple SMs on a GPU with no shared state among them, the easiest way for a multitasking GPU is to run each kernel on different subsets of SMs. Spatial multitasking [3] noticed this property, and explored possible SM partitioning policies. Another way to schedule multiple kernels on a single GPU is to do SMK execution [30, 32] similar to the SMT on CPUs [27]. In SMK, an SM is shared among multiple kernels at finer granularity in the order of few cycles, which can improve resource utilization further.

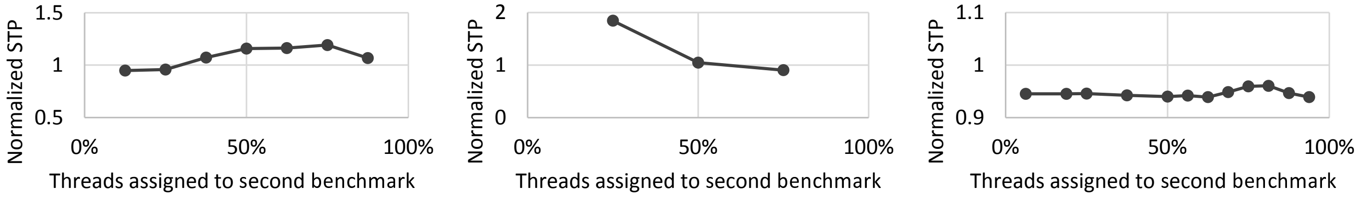
2.3 Multikernel Metrics

Throughout the paper, we measure the multikernel performance using the metrics suggested by Eyerman et al. [8]; average normalized turnaround time (ANTT), and system throughput (STP). ANTT represents the user-perceived response time, while STP portrays the overall progress of the system. The following equations are used to calculate ANTT and STP:

$$ANTT = \frac{1}{N}\sum_{i=1}^{N}\frac{CPI_i^{MK}}{CPI_i^{SK}} \tag{1}$$

$$STP = \sum_{i=1}^{N}\frac{CPI_i^{SK}}{CPI_i^{MK}} \tag{2}$$

where N denotes the number of benchmarks, CPI_i^{MK} is the CPI when a benchmark is executed in the multiprogrammed workload, and CPI_i^{SK} is the CPI when the benchmark is executed alone. ANTT is a lower-is-better metric and STP is a higher-is-better metric. ANTT and STP may disagree on which multikernel scheduling is better because each metric reflects a separate aspect of the whole system.



(a) COM/COM

(b) COM/MEM

(c) MEM/MEM

Figure 2: Normalized STP of SMK when fixed number of threads are assigned to each kernel within an SM for representative examples. STP of spatial multitasking is used as 1. (a) Even SMK is as good as the best performing SMK, (b) large performance gap exists between the best performing SMK and even SMK, and (c) spatial multitasking performs better than any SMK.

3. Motivation and Challenges

3.1 Spatial vs. Simultaneous Multikernel

Table 1 shows the trends in the four generations of Nvidia GPUs, where the resources on a single SM have increased as well as the number of SMs. This trend of having more concurrency supports within an SM and across SMs indicates that multitasking GPUs are promising. Concurrency across SMs favors spatial multitasking, while concurrency within an SM favors SMK. To fully understand the benefit of multitasking GPUs, we study the performance of different resource partitions for varying application mixes.

Figure 2 depicts the normalized STP of SMK on a GTX980 when fixed numbers of threads are assigned to each kernel within an SM. All the STP values are normalized to that of spatial multitasking. Using compute-intensive (COM) and memory-intensive (MEM) benchmarks in Table 3, we study three representative benchmark pairs to discuss opportunities: LC/BP for COM/COM, HS/SC for COM/MEM, and LMD/ST for MEM/MEM. Figure 2 (a) depicts a case where SMK with even partitioning, which gives equal number of resources to each kernel resulting in 50% on the x-axis, performs better than or similar to other SMK resource partitions as well as spatial multitasking when both benchmarks are COM. Figure 2 (b) illustrates a case where SMK with uneven partitioning benefits much larger than SMK with even partitioning and spatial multitasking. This commonly occurs if one of the benchmark is MEM and the other is COM. By giving more thread blocks to COM, they can utilize the idle cycles from MEM. Figure 2 (c) shows a case where spatial multitasking performs better than any SMK. This commonly happens if both benchmarks are MEM and the contention for the load/store unit or cache is high.

In general, SMK performs better than spatial multitasking when (1) SMK launches more threads, or (2) co-running kernels have different execution unit utilizations or small interference. For example, it is known that memory-intensive kernels cannot hide the memory latency even with thousands of threads because all the threads are likely to be waiting for the memory as they run the same code [20]. In such cases, SMK can issue instructions from compute-intensive kernels to improve compute resource utilization. On the other hand, spatial multitasking can provide better performance when the interference between kernels within an SM is large enough to degrade the system performance for SMK.

	Fermi	Kepler	Maxwell	Pascal
Threads	1536	2048	2048	2048
Thread Blocks	8	16	32	32
Registers	128kB	256kB	256kB	256kB
Shared Memory	48kB	48kB	64kB	64kB
# of SMs (Tesla Model)	14 (M2050)	15 (K40)	24 (M40)	56 (P100)

Table 1: Resource trends in an SM on GPUs.

3.2 Multitasking GPU Performance

The system performance of a multitasking GPU is greatly impacted by how GPU resources are partitioned among kernels because spatial multitasking and SMK have their own advantages and disadvantages depending on the application mixes. A practical implementation of multitasking GPUs has to address how to find the best performing resource partition exploiting both spatial multitasking and SMK. However, predicting multitasking performance is difficult especially for SMK because of the complex interactions between kernels.

To find the best performing resource partition, performance should be estimated for different resource partitions. A straightforward way would be to extrapolate the multikernel performance from single kernel execution performance. However, we argue that extrapolation ignores the complex interaction when running multiple kernels on the same SM, and cannot capture the dynamism.

3.2.1 Interference

Figure 3 shows an IPC trace within an SM using a 50k instruction window. From 16M to 80M instruction interval, Figure 3 (Top) shows the IPC trace of SRAD and BFS when they are running together on an SM with SMK. We assume even resource partitioning between two kernels. Figure 3

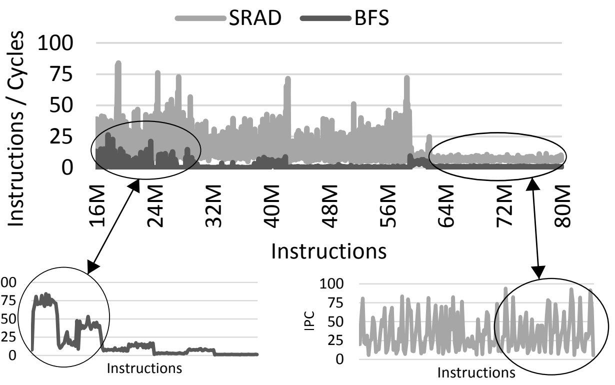


Figure 3: A trace of the instructions per cycle (IPC) within an SM using a 50k instruction window when running SRAD and BFS together with SMK (top), and when running them independently for the same interval (bottom). Circles indicate the same interval, where the IPC trace shows different behavior between running alone, and SMK.

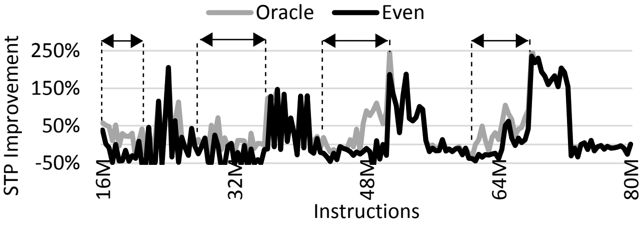


Figure 4: A trace of STP improvement over non-shared execution for Oracle and Even resource partition within an SM using 50k instruction window when running SRAD and BFS together. Indicated sub-intervals are when Oracle chooses different partition from Even.

(Bottom) illustrates the IPC trace for BFS and SRAD when they are executed in isolation. To indicate interference, we circled two sub-intervals, where the IPC trace is substantially different between running alone, and with SMK. In the left circle, BFS has a program phase with high IPC when executed alone, which is less distinguishable with low IPC when running together with SRAD through SMK. On the right circle, the IPC of SRAD is less than half of independent execution due to memory and cache contention with BFS. We should not estimate SMK performance based on single kernel performance because interference can change the execution behavior significantly as shown in the figure.

3.2.2 Dynamism

Figure 4 depicts a trace of STP improvement within an SM for Oracle and Even over non-shared execution using 50k instruction window when running SRAD and BFS together with SMK. Oracle uses the best performing resource partition for each instruction window, and instantly switches between different thread block partitions with zero overhead.

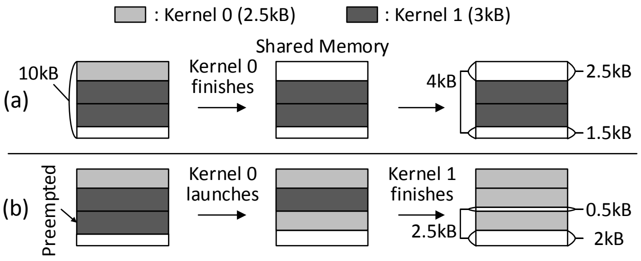


Figure 5: An illustration of resource fragmentation problem for shared memory when (a) a kernel terminates, and (b) a kernel is preempted.

Even always distributes threads evenly among SRAD and BFS. Oracle can improve STP by 30.0% over Even across the shown trace. Sub-intervals drawn on the figure illustrate a period when Oracle chooses different thread block partition from Even. Due to the dynamism, the best performing partition can continuously change during the execution.

3.2.3 Summary

To capture interferences and dynamism, a dynamic resource management framework that continuously monitors the multitasking performance using a direct measurement is necessary to maximize the performance benefit from multitasking GPUs.

3.3 SMK Challenges

The implementation of SMK GPUs itself has two additional challenges to be addressed: (1) it has to solve resource fragmentation, which can become worse due to more preemptions with dynamic scheduling, and (2) it should avoid the starvation problem, which comes from the interaction between warp scheduling and multikernel execution.

3.3.1 Challenge 1: Resource Fragmentation

Figure 1 illustrated that shared memory and register file are shared among the thread blocks within an SM. Because shared memory is shared at a thread block granularity, each thread block allocates a consecutive region in the shared memory. Because thread blocks execute the same code to compute the shared memory address, physical address for shared memory has to be differentiated between thread blocks. The shared memory base register (SBR) contains the base shared memory address for each thread block. When a thread block accesses shared memory at runtime, the address is computed by adding the virtual address with the SBR. Similarly, consecutive registers are allocated to each warp. Note that a physical register actually contains the same register index for 32 threads to reduce the number of ports because a warp consists of 32 threads. The physical register index is computed by adding the register index with a warp's register base register (RBR).

Figure 5 illustrates the resource fragmentation problem in SMK by showing an example with shared memory. A

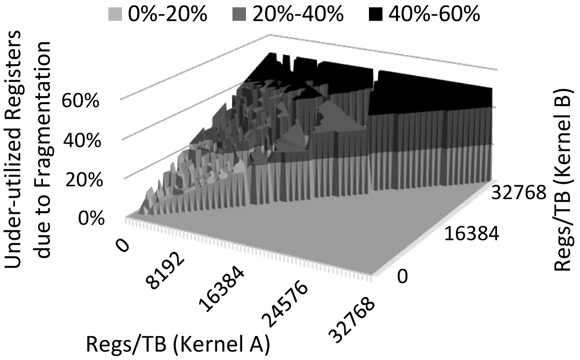


Figure 6: A synthetic study of register fragmentation for kernels with various register usages. Regs/TB refers to the registers per thread block. Kernel A and B are assumed to be running together using even SMK initially. Register fragmentation is measured assuming that kernel B launches another thread block due to resource repartitioning, and the thread blocks are preempted from kernel A accordingly.

thread block from kernel 0 uses 2.5kB of shared memory, while a thread block from kernel 1 uses 3kB. We assume that the total size of the shared memory is 10kB. A resource fragmentation occurs because a thread block has to allocate shared memory and registers consecutively. When resources are only available in smaller chunks although there is enough free resource in total, resources are fragmented and the new thread block cannot be launched. The fragmentation can occur during resource allocation or release: when a new kernel is launched, a kernel is terminated, or a preemption occurs due to dynamic resource repartitioning. We illustrate two examples in detail.

Figure 5 (a) shows how resource fragmentation occurs when a kernel terminates. When kernel 0 finishes all of its thread blocks, 2.5kB of shared memory is released for other kernels to use leaving a total of 4kB. In the given scenario, the free shared memory cannot be utilized by kernel 1 because kernel 1 requires 3kB of consecutive shared memory, while 4kB is in chunks of 2.5kB and 1.5kB. Figure 5 (b) depicts how shared memory is fragmented when a kernel is preempted partially. During the execution, the bottom thread block from kernel 1 is preempted due to repartitioning. A thread block from kernel 0 is launched instead. Later, kernel 1 eventually finishes, and thread blocks from kernel 0 will fill in. However, it can issue only one thread block rather than two because 2.5kB is fragmented into chunks of 0.5kB and 2kB. Similar resource fragmentation problems can occur for registers as well.

To quantify the amount of fragmentation, Figure 6 performs a synthetic study of register fragmentation for kernels with various register usages. Kernel A and B are assumed to be running together using even SMK. Shared memory is ignored. 32 concurrent thread blocks and 65536 registers are available within an SM as the GTX980 does. Note that z-axis excludes under-utilized registers due to lack of resources, e.g., when 1536 free registers are under-utilized because thread blocks from both kernels require 16000 registers per thread block. Similar to Figure 5 (b), thread blocks from kernel A consume consecutive registers first, then thread blocks from kernel B consume next consecutive registers. In this study, register fragmentation is measured assuming that kernel B is launching a new thread block due to resource repartitioning, and thread blocks from kernel A are preempted accordingly. The choice of preempted thread blocks is random.

Register fragmentation can reach upto 50% in the worst case when kernel A consumes fewer than 32768 registers and kernel B requires close to 32768 registers per thread block. In such cases, free registers after thread blocks of kernel B, which are denoted by a white box in Figure 5 (b) at the bottom, and the released registers from kernel A together provide the space for the newly launched thread block of kernel B in total, however, they are fragmented. A half of the graph does not have register fragmentation: when kernel A requires a larger number of registers per thread block than kernel B, kernel B can launch a new thread block without fragmentation. However, multitasking performance is irrelevant to the register usage hence we cannot force these scenarios. Moreover, this is an artifact of launching one more thread block from kernel B: fragmentation can still occur when kernel B tries to launch multiple new thread blocks.

The duration of fragmentation is related to the remaining thread block execution time of kernel A and B. To compute the average duration of fragmentation, an uniform random distribution of thread block execution progress at the beginning of fragmentation is assumed, and the thread block progress is assumed to be in sync for multiple thread blocks. 1.3ms is the average duration of fragmentation across the evaluated benchmarks. Note that this is an optimistic measure because thread blocks finish out-of-order in reality. In general, register fragmentation can be large, and take a long time to be resolved.

To avoid resource fragmentation, we have to make sure that free resources are adjacent to released resources when a kernel finishes execution, and a thread block to preempt should be chosen with released resources in mind.

3.3.2 Challenge 2: Starvation

The choice of warp scheduling can greatly impact SMK performance. For example, GTO gives higher probability of issuing an instruction to the kernel that issued thread blocks earlier. This can lead to a starvation period for the other kernel. The starvation problem can become worse with unequal resource partitioning because a kernel with fewer warps will have even lower probability of issuing. Figure 7 shows an IPC trace within an SM using a 50k instruction window when running LC/BS, where LC is launched before BS. We use even thread block partitioning for SMK, and GTO for warp scheduling. Note that both LC and BS do not show

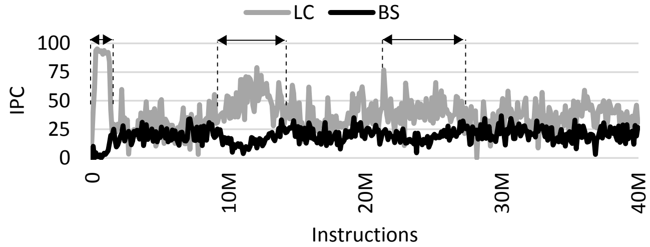


Figure 7: An IPC trace within an SM using a 50k instruction window when running LC/BS. LC is launched before BS. Sub-intervals, where BS starves, are shown.

significant phase changes during isolated execution. Because there is no phases nor repartitioning, we expect to see stable IPC for both benchmarks. However, as shown in the figure, starvation periods exist, where LC issues more instructions than its average while BS starves with lower IPC than its average. Although the performance gap during the starvation periods becomes smaller as the system progresses, they can take a large portion of execution, e.g., BS progresses by 50% near 30M instructions.

On the other hand, round-robin (RR) scheduling gives equal probability of issuing an instruction to each warp. We studied the performance of SMK under RR and found out that STP is improved by 2.5%, on average, over SMK under GTO when the proposed resource partitioning in Section 4.1 is used. While RR performs better than GTO for SMK in general, we noticed that SMK under GTO may perform better than SMK under RR for individual workloads because GTO performs better than RR in most cases for single kernel execution [22]. To maximally benefit from SMK, we need to retain the benefits of GTO within a single kernel execution, while avoiding starvation problem by allowing the other kernel to make progress.

4. GPU Maestro Design

In this section, we introduce GPU Maestro, a dynamic resource management for efficient utilization of multitasking GPUs. Figure 8 illustrates an architectural overview of the overall system. On the top of the figure, we illustrate 2-way resource allocation, where resources are provided to kernels from opposing ends of the resource pool. On the bottom of the figure, we show that the dynamic resource management framework collects runtime performance statistics from each SM using existing performance counters, and tells a thread block scheduler to repartition for multitasking GPUs. The dynamic resource management framework is implemented in software, and runs on the command processor. The thread block scheduler preempts thread blocks to achieve the desired resource partition similar to the prior works [21, 26]. GPU Maestro is composed of three components: dynamic resource management framework, 2-way resource allocation, and kernel-aware warp scheduling mechanism.

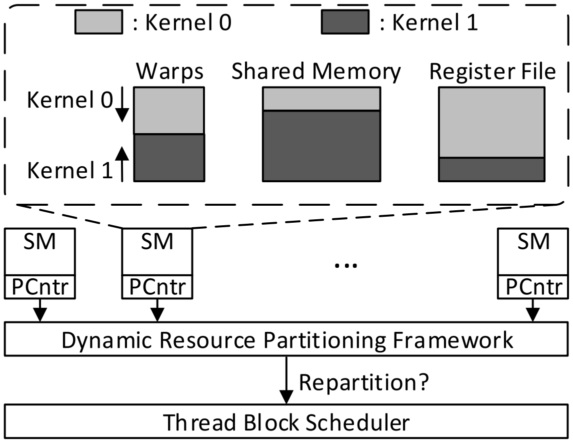


Figure 8: An architectural overview of GPU Maestro. PCntr refers to existing performance counters on an SM.

4.1 Dynamic Resource Partitioning

Figure 9 illustrates how GPU Maestro performs dynamic resource partitioning when two kernels are running on the GPU. At the top, we show the timeline, where repartitioning decisions occur at the end of every epoch using the monitored performance. GPU Maestro uses 50k cycles for each epoch, which is long enough to minimize the repartitioning overhead. Note that after repartitioning decisions, the next epoch is not started until the preemptions take place and the desired thread block partition is achieved. On the bottom of the figure, we show how resource partitioning decisions are made. SMs are divided into three groups: dedicated, trial, and follower. Dedicated SMs run each kernel without SMK, and are used to measure the single kernel performance as well as the performance of spatial multitasking. Trial SMs test resource repartitioning possibilities for SMK GPUs. Follower SMs are partitioned with a resource partitioning that is measured to provide the best performance from the previous epoch.

GPU Maestro assigns two SMs to trial SMs. To determine the thread block partition for trial SMs, GPU Maestro defines a trial kernel, which launches fewer thread blocks when having an SM entirely. Because resource requirements are known at compile time, the trial kernel can be determined at the kernel launch time. The trial kernel assigns one more thread block for one trial SM, and one fewer thread block for the other compared to followers. The other kernel fills up the rest of the resources in the trial SMs. The initial partitioning state for follower SMs is SMK with even partitioning by default. GPU Maestro uses a history-based partition prediction from the previous SMK for the initial state in followers when a running kernel was assigned fewer thread blocks than even partition during previous SMK. A regression model may also be used to predict the performance of a kernel [10, 33] and determine the initial partitioning state, but we leave it as a future work.

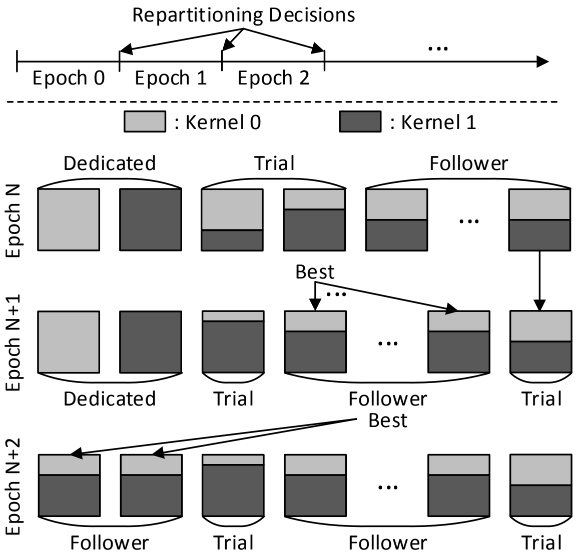


Figure 9: An illustration of dynamic resource partitioning process in GPU Maestro when two kernels are running on the GPU. GPU Maestro makes repartitioning decisions at the end of every epoch, which may not result in repartitioning if the previous preferred partition is the best. Follower SMs follows the repartitioning decision including spatial multitasking, which gives the best performance. Note that trial and follower SMs can change to minimize the preemption overhead from repartitioning. Dedicated SMs turn into follower SMs when a steady state is reached.

When making repartitioning decisions, GPU Maestro tries to minimize the preemption overhead. In the figure, one of the trial SMs becomes a follower and vice versa from epoch N to epoch N+1. When dedicated SMs show stable performance as the SMK partition reaches a steady state, GPU Maestro stores the performance of dedicated SMs to be used in upcoming epochs, and turns dedicated SMs into follower SMs to maximize system performance. Dedicated SMs are reassigned when either a new kernel is launched, thread blocks are repartitioned, or once every hundred epochs. The last case ensures that the stored performance is indeed stable.

To determine which resource partition performs the best, GPU Maestro first defines the performance objective. As discussed in Section 2.3, there could be multiple metrics for multikernel execution. GPU Maestro can set one of these metrics for the performance objective. For example, when ANTT is set as the performance objective, GPU Maestro computes ANTT for spatial multitasking using dedicated SMs and SMK with different resource partition using trial and follower SMs. Whichever partition with the lowest ANTT will be selected because ANTT is a lower-is-better metric. To compute these metrics, we both need CPI for multikernel execution as well as single kernel execution. GPU Maestro uses the CPI of dedicated SMs to estimate CPI for the single kernel execution.

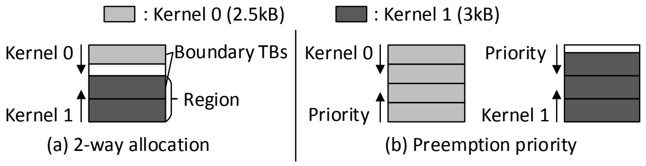


Figure 10: (a) The proposed 2-way allocation, where a kernel schedules thread blocks in the opposite direction, and (b) the fixed preemption priority order imposed by the 2-way allocation.

When spatial multitasking performs better than SMK, GPU Maestro adds one SM per kernel from followers for spatial multitasking at each repartitioning epoch rather than turning all followers for spatial multitasking because additional SMs may not provide scalable performance as the memory-intensive benchmarks in Table 3 show. This brings additional benefit to GPU Maestro by allowing trial and follower SMs to continuously provide SMK performance with uneven partitioning, which may perform better than spatial multitasking.

The proposed dynamic resource management framework requires 2-bits per SM to indicate whether it is dedicated, trial, or follower. Although existing performance counters are utilized, extra storage is also required to store dedicated SM's performance from previous epoch when dedicated SMs are turned into followers. Comparing and deciding the best performing partition are done in software running on the command processor.

4.2 2-Way Resource Allocation

By restricting the direction of allocation, kernels allocate resources consecutively in the steady state with free space between the kernels, which avoids the resource fragmentation when one of the kernel is finished. Figure 10 (a) shows the proposed 2-way allocation, where one-dimensional resources (RF, shared memory, and etc.) are allocated either from top to bottom or from bottom to top. We define boundary thread blocks as the thread blocks, where resource allocation meets the other kernel or the free space. We also define a region, where kernels can safely launch its new thread blocks.

We also impose a fixed preemption priority order for each kernel. Figure 10 (b) illustrates the preemption priority, which is the reverse direction of the resource allocation direction. By forcing the preemption priority, the already launched kernel can release resources adjacent to the free space, which will allow the other kernel to start allocating resources in the opposite direction. Thus, the resource fragmentation coming from both Figure 5 (a) and (b) can be avoided, and the other kernel can maximally utilize the released resources. The preemption overhead from the

System	Parameters
SM	16 SMs, 1126 MHz, 4 warp schedulers 2K threads, 32 thread blocks 64K registers, 96kB shared memory 2kB L1I, 48kB L1D
Memory Subsystem	2MB L2, 4 memory partitions 224 GB/s bandwidth

Table 2: System configuration.

fixed preemption priority can be minimized by using prior work [21].

There is one limitation to 2-way allocation: the number of kernels running concurrently on an SM is limited to two. However, this does not force the shared GPU to run only two kernels because spatial multitasking can be mixed with 2-way allocation SMK, or 2-way allocation can be extended into 2n-way allocation to run more than two kernels. When spatial multitasking is combined with 2-way allocation to run more than two kernels, one SM may run first two kernels in SMK while other SMs can run other combinations of kernels. Combining spatial multitasking with SMK can be a better solution than having SMK to support more than two kernels because the cost of avoiding resource fragmentation completely can be significant.

2n-way allocation performs 2-way allocation recursively to run *2n* kernels simultaneously on an SM. 2n-way allocation first divides resources equally among *n* pairs of kernels. Between the pair of kernels, 2-way allocation is used. 2n-way allocation can resolve the fragmentation within the pair of kernels due to 2-way allocation, however, fragmentation may exist between the pairs of kernels.

4.3 Kernel-aware Warp Scheduling

To avoid the starvation problem, choosing which kernel to issue instructions becomes an important problem in SMK GPUs similar to choosing which thread to fetch in SMT [28]. Conceptually similar to two-level warp scheduling [13], kernel-aware scheduling still utilizes GTO warp scheduling within each kernel for better single kernel performance, while providing kernels enough issue slots to avoid the starvation problem by changing kernel priorities. In this paper, we propose to use loose round-robin (LRR) for kernel-aware scheduling. LRR flips kernel priority whenever a lower priority kernel has a ready instruction to issue, thus, provides almost equal opportunity for each kernel to progress. While LRR is the simplest mechanism, it achieves the best performance compared to other more intelligent schemes as shown in Section 5.2.

In SMK, there are situations where warp scheduling priority should diverge from the underlying warp scheduling policy. For example, when some of the warps are being preempted with draining [21, 26], these warps should be given higher priority to issue instructions to reduce repartitioning time. Another example is when a kernel has no more thread blocks to issue because it is near the end of execution. Again, the warps within the boundary thread block should be given higher priority to issue instructions because resources only become available to the other kernel when resources are released from the boundary thread block.

Benchmark	Source	Type
Breadth First Search (BFS)	Rodinia [7]	MEM
Back Propagation (BP)	Rodinia [7]	COM
BlackScholes (BS)	Nvidia SDK [14]	MEM
B+ Tree (BT)	Rodinia [7]	MEM
Coulombic Potential (CP)	Parboil [25]	COM
FDTD	Nvidia SDK [14]	MEM
Fast Walsh Transform (FWT)	Nvidia SDK [14]	MEM
HotSpot (HS)	Rodinia [7]	COM
Heart Wall (HW)	Rodinia [7]	COM
Kmeans (KM)	Rodinia [7]	MEM
Laplace-Boltzmann Method (LBM)	Parboil [25]	MEM
Leukocyte Tracking (LC)	Rodinia [7]	COM
LavaMD (LMD)	Rodinia [7]	MEM
LU Decomposition (LUD)	Rodinia [7]	MEM
Magnetic Resonance Imaging (MRIQ)	Parboil [25]	COM
MUMmerGPU (MUM)	Rodinia [7]	MEM
Needleman Wunsch (NW)	Rodinia [7]	MEM
SAD	Parboil [25]	COM
Streamcluster (SC)	Rodinia [7]	MEM
SRAD	Rodinia [7]	MEM
Stencil (ST)	Parboil [25]	MEM
TPACF	Parboil [25]	COM

Table 3: Benchmark specification.

5. Results

We extend the GPGPU-Sim v3.2.2 [5] to simulate GPU Maestro. The Nvidia GTX980 based on the Maxwell architecture [17] is modeled as shown in Table 2. We use a wide range of GPGPU applications from various benchmark suites including Nvidia Computing SDK [14], Rodinia [7], and Parboil [25] for evaluation. Table 3 lists all the evaluated benchmarks, and their types. The types of the benchmarks are statically defined using their profiled performance with different numbers of SMs. COM, which is compute-intensive benchmarks, is defined as the benchmarks with more than 12x speedup when the number of SMs is changed from one to sixteen. MEM, which is memory-intensive benchmarks, is defined as the benchmarks that show less than 12x speedup. Although the GTX980 does not cache global memory accesses in the L1D by default, we assumed that they are cached at L1D using LDG intrinsic because many of the simulated GPGPU benchmarks have intra-warp locality and benefit from the L1D [22]. The performance objective of GPU Maestro is set to ANTT, and Chimera [21] is used for preemption.

Multi-programmed workloads are constructed using all possible pairs of GPGPU benchmarks from Table 3. A typical evaluation method is adopted from prior GPU multitasking works [3, 21, 26]. Each multi-programmed workload

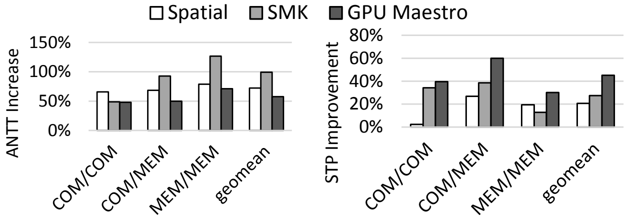


Figure 11: ANTT increase and STP improvement of Spatial, SMK, and GPU Maestro over non-shared execution. Spatial partitions resources evenly at the SM granularity. SMK partitions resources evenly within the SMs. A geometric mean of all combinations of co-running kernels from each category is shown. ANTT is a lower-is-better metric, and STP is a higher-is-better metric.

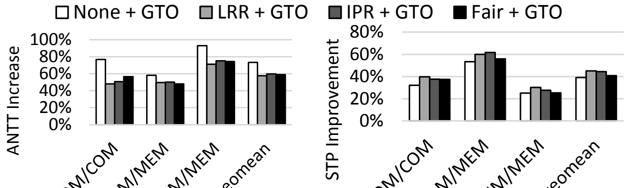


Figure 12: ANTT increase and STP improvement of various kernel-aware scheduling techniques on GPU Maestro over non-shared execution. A geometric mean of all combinations of co-running kernels from each category is shown. All the kernel-aware scheduling techniques are implemented on top of GTO warp scheduling. None does not do any kernel-aware scheduling. ANTT is a lower-is-better metric, and STP is a higher-is-better metric.

starts all the benchmarks simultaneously in the beginning. Each benchmark is run until it finishes its execution or 1 billion instructions. When one benchmark completes earlier than the other, it restarts from the beginning to continuously stress the system. The reported results are gathered only for the first round of the execution, and the restarted executions are ignored.

5.1 Resource Partitioning Performance

We first compare GPU Maestro with the baseline Spatial and SMK. Spatial partitions resources evenly among the kernels at the SM granularity, while SMK partitions resources evenly within the SMs. Both fill up the remaining resources with the other kernel if resources are released from one kernel.

Figure 11 shows ANTT increase and STP improvement of Spatial, SMK, and GPU Maestro over non-shared execution. When sharing the GPU, ANTT, which measures response time, is increased compared to running it in isolation on the GPU. On average, Spatial, SMK, and GPU Maestro increase ANTT by 72.2%, 99.1%, and 57.6%, respectively, while improving STP by 20.7%, 27.3%, and 45.0%, respectively. In general, SMK is better than Spatial for COM/COM, but Spatial is better than SMK for MEM/MEM. SMK is better for STP in COM/MEM, but Spatial is better for ANTT. GPU Maestro outperforms both Spatial and SMK because it can exploit the best performing partition among both schemes.

In COM/COM pairs, SMK with even partitioning performs similar to any other SMK partitionings as discussed in Section 3.1. As a result, the performance gap between SMK and GPU Maestro is small for COM/COM. In MEM/MEM pairs, the performance gap between Spatial and GPU Maestro is smaller than the other pairs because Spatial is often better than SMK when running MEM/MEM pairs. In general, the system performance is improved consistently with GPU Maestro over Spatial and SMK.

5.2 Kernel-aware Scheduling Performance

To compare kernel-aware scheduling techniques, we consider three techniques: LRR, issued-per-ready (IPR), and Fair [30]. All the techniques are implemented on top of GTO warp scheduling. To show the benefit of kernel-aware scheduling, we also show None, which does not do any kernel-aware scheduling. All the techniques use GPU Maestro's dynamic resource management scheme.

IPR: IPR mimics ICOUNT [28] for GPUs by using following metric for each warp scheduler:

$$IPR = \frac{\sum_{cycles} \#\ of\ issued\ warps}{\sum_{cycles} \#\ of\ ready\ warps} \tag{3}$$

IPR is cumulated over every cycle. A kernel with lower IPR has higher priority to issue an instruction. This metric avoids starvation because IPR becomes smaller as a kernel waits and does not issue instructions. IPR also favors compute-intensive kernels because they have more ready warps compared to memory-intensive kernels, which makes IPR lower.

Fair: Fair uses following equation to compute quota for each kernel:

$$Quota_k = \frac{C_k}{\sum C_k}, \quad C_k = x\% \times \frac{S_k}{T_k} \tag{4}$$

where $x\%$ is the percentage of issued cycles, which is profiled from single kernel execution, and T_k is the number of thread blocks in an SM when the kernel is run in isolation, and S_k is the number of thread blocks allocated in SMK for the kernel. This is equivalent to SMK-W in [30] except for the thread block partitioning algorithm.

Figure 12 illustrates ANTT increase and STP improvement of None, LRR, IPR, and Fair over non-shared execution. On average, None, LRR, IPR, and Fair increase ANTT by 73.4%, 57.6%, 59.6%, and 58.9%, respectively, while improving STP by 39.0%, 45.0%, 44.4%, and 40.8%, respectively. In general, all kernel-aware scheduling techniques

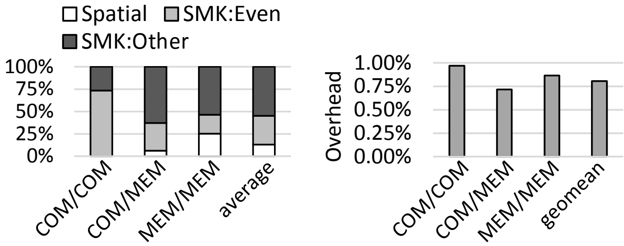


Figure 13: (Left) The percentage of repartitioning decisions to use spatial multitasking, SMK with even partitioning, and SMK with other non-even partitioning, and (Right) repartitioning overhead in GPU Maestro. An average of all combinations of co-running kernels from each category is shown.

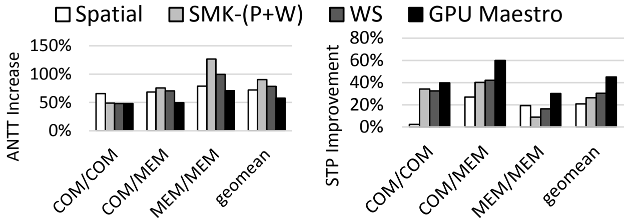


Figure 14: ANTT increase and STP improvement of Spatial, SMK-(P+W), WS, and GPU Maestro over non-shared execution. A geometric mean of all combinations of co-running kernels from each category is shown. ANTT is a lower-is-better metric, and STP is a higher-is-better metric.

reduce ANTT and improve STP compared to not having kernel-aware scheduling.

Among LRR, IPR, and Fair, LRR performs the best for both ANTT, and STP. Interestingly, the trend of improvement is the opposite of the implementation complexity. For LRR, a single bit for each kernel is enough to notify whether it has a ready warp to issue. For IPR, we need counters to record the number of issued warps as well as the number of ready warps, and a comparison logic. For Fair, we need extra logic to compute quota, and the percentage of issued cycles has to be profiled or measured directly on the GPU for each kernel. The main reason for the performance difference between these kernel-aware scheduling techniques goes back to the starvation problem. For example, GPU Maestro assigns fewer thread blocks to MEM application when running COM/MEM combinations. When running COM/MEM combinations in Fair, the quota for MEM application will be small because S_k becomes smaller due to GPU Maestro, and $x\%$ is already smaller than the COM application. While Fair still allocates more issue slots to the MEM application compared to not having kernel-aware scheduling, it still suffers from starvation compared to LRR and IPR. We conclude that kernel-aware scheduling is necessary in SMK, and the simplest LRR kernel-aware scheduling in fact gives the best performance both in terms of ANTT and STP.

5.3 Repartitioning Analysis

In this section, we analyze the dynamic thread block repartitioning in detail. Figure 13 (Left) illustrates the distribution of repartitioning decisions in GPU Maestro. Spatial is when GPU Maestro chooses to use spatial multitasking instead of SMK. SMK:Even denotes when GPU Maestro chooses Even as the best performing thread block partition on SMK, and SMK:Other refers to the decisions when GPU Maestro assigns unequal resources to the kernels on SMK. On average, GPU Maestro decides to utilize spatial multitasking for 13.0% of the time, SMK:Even for 32.2%, and SMK:Other for 54.8%. Among SMK:Other, 29.4% of the time corresponds to having one thread block difference from SMK:Even, and the remaining 70.6% comes from having more unequal resource assignments. Analyzing each category, SMK:Even is utilized the most in COM/COM, while SMK:Other is used the most in other categories. Also, spatial multitasking is used often for MEM/MEM. These results are consistent to the study of motivating examples in Section 3.1.

Figure 13 (Right) depicts the repartitioning overhead in GPU Maestro. To measure this overhead, we ran an ideal case, where repartitioning takes place with zero overhead, and compared the STP. The overhead can be thought of as the frequency of thread block repartitioning multiplied by how much system throughput is wasted during the repartitioning. On average, GPU Maestro has 0.8% overhead from dynamic thread block repartitioning, which is small. Also, the absolute difference of the overhead is small across the categories.

5.4 Comparison to Prior Works

We also compare GPU Maestro with three prior works: spatial multitasking [3], SMK-(P+W) [30], and Warped-Slicer (WS) [32]. For spatial multitasking at the SM granularity, we use Smart as the SM partitioning heuristic, which partitions SMs evenly among applications. However, it is guaranteed that no application is given more SMs than it can fill up with thread blocks. SMK-(P+W) and WS partition resources within the SM granularity. SMK-(P+W) uses the dominant resource share to partition resources within an SM, and applies fair warp scheduling. WS uses single kernel execution profiles to partition resources within an SM.

Figure 14 illustrates ANTT increase and STP improvement of the prior works and GPU Maestro over non-shared execution. On average, Spatial, SMK-(P+W), WS, and GPU Maestro increases ANTT by 72.2%, 90.3%, 78.4%, and 57.6%, respectively, while improving STP by 20.7%, 26.2%, 30.3%, and 45.0%, respectively. GPU Maestro outperforms Spatial, SMK-(P+W), and WS in both ANTT and STP.

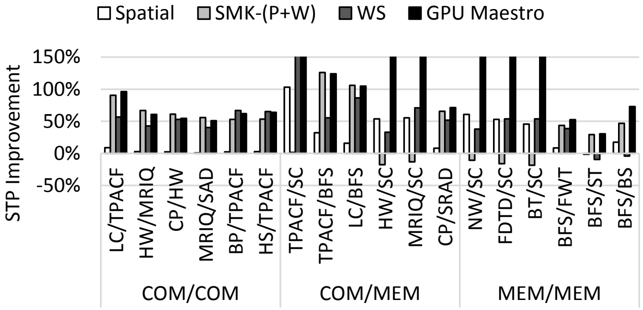


Figure 15: STP improvement of Spatial, SMK-(P+W), WS, and GPU Maestro over non-shared execution for selected benchmark pairs.

For a further in-depth analysis, Figure 15 shows STP improvement of the techniques over non-shared execution for selected benchmark pairs. From each category, we selected three pairs, where the performance improvement of SMK with even thread block partitioning over Spatial is the largest, and three pairs with opposite characteristics. For the COM/COM category, SMK always performs better than Spatial hence we picked six pairs with the former characteristic. As shown by the figure, GPU Maestro performs similar or better than any prior works. Because GPU Maestro can select the best performing resource partition regardless of whether Spatial or SMK is favored, it achieves better performance compared to prior works that focus on one of the schemes. Moreover, GPU Maestro can capture dynamism within a kernel and interference between kernels, which is not taken into account for WS. WS only utilizes single kernel execution profiles. For example, benchmark pairs including BFS, which had a large interference with the other kernel by slowing it down, do not perform well for WS.

6. Related Work

The first attempts on GPU multitasking came from software approaches by merging kernels at compile time [19] or providing abstractions at the operating system level [23]. These approaches often require modifications to the source code, which may not be applicable in general cases. More recently, hardware preemption mechanisms were studied. Tanasic et al. [26] studied two preemption techniques: traditional context switching, and SM draining. In SM draining, an SM is no longer scheduled with new thread blocks. When the running thread blocks are finished, the SM can be preempted. Chimera [21] further enabled fast preemption with SM flushing, and demonstrated that more efficient preemptive multitasking is possible by using different preemption techniques for each thread block. By having hardware supports for multitasking GPUs, existing GPU kernels can be seamlessly take advantage of them.

Elastic kernel [19] controls the resource usage of kernels in more fine-grained manner to improve resource utilization on SMs. However, their evaluation of multitasking is limited to the timeslice of a kernel rather than preemptive multitasking. Persistent threads [9, 31] are software approaches to enable spatial multitasking. However, persistent threads require the programmers to explicitly change the kernels to fit in the framework with increased difficulty for debugging. Moreover, extra registers required by the framework may not be acceptable for register-constrained kernels.

Spatial multitasking [3] runs multiple kernels on shared GPUs at the SM granularity. While spatial multitasking improved system throughput over single kernel execution, it also showed that the performance difference between various SM partitioning schemes were relatively small. The idea of SMK [30] has been explored, however, it only addressed DRF thread block partitioning, which focuses on fair resource allocation. WS [32] utilizes single kernel execution profiles to consider uneven partitioning within an SMK, however, it did not address the interference between kernels. Moreover, prior works on SMK did not study resource fragmentation problem in depth. GPU Maestro addresses these problems, and achieves better performance than these prior works by utilizing both spatial multitasking and SMK.

7. Conclusion

In this paper, we proposed GPU Maestro, a dynamic resource management for efficient utilization of multitasking GPUs. GPU Maestro identified that spatial multitasking and SMK have their own advantages and disadvantages depending on the resource partitions and the application mixes. GPU Maestro explores which resource partition provides the best performance by directly measuring the performance of a subset of SMs with different partitions. GPU Maestro also identified two challenges in implementing SMK GPUs. First, it showed the existence of resource fragmentation problem, and proposed 2-way resource allocation, which forces kernels to allocate and release resources consecutively in opposing directions. Second, GPU Maestro also demonstrated that kernel-aware warp scheduling is critical to fully benefit from SMK, and suggested that simple LRR kernel-aware scheduling performs the best. Evaluations have shown that GPU Maestro improves the STP by an average of 45.0% while increasing the ANTT by an average of 57.6% over non-shared execution.

Acknowledgments

We would like to thank the anonymous reviewers as well as the fellow members of CCCP research group for their valuable comments and feedbacks. This work is supported in part by the National Science Foundation under grant SHF-1217917 and XPS 1628991, by Samsung GRO Award, by the National Research Foundation of Korea under grant NRF-2015R1C1A1A01053844, and by the Korea Institute for Advancement of Technology grant (No. N0001883) funded by the Korea government (MSIP, MOTIE).

References

[1] Green500 list, 2016. https://www.top500.org/green500/lists/2016/11/.

[2] Top500 list, 2016. http://www.top500.org/lists/2016/11/.

[3] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The case for GPGPU spatial multitasking. In *Proc. of the 18th International Symposium on High-Performance Computer Architecture*, pages 1–12, 2012.

[4] Amazon. Amazon web services. https://aws.amazon.com/ec2/.

[5] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Proc. of the 2009 IEEE Symposium on Performance Analysis of Systems and Software*, pages 163–174, Apr. 2009.

[6] C. Basaran and K.-D. Kang. Supporting preemptive task executions and memory copies in GPGPUs. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 287–296, 2012.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, , J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the IEEE Symposium on Workload Characterization*, pages 44–54, 2009.

[8] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, 2008.

[9] K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style GPU programming for GPGPU workloads. *Innovative Parallel Computing*, pages 1–14, 2012.

[10] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, pages 152–163, 2009.

[11] S. Kato, K. Lakshmanan, R. R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. pages 17–30, 2011.

[12] KHRONOS Group. OpenCL - the open standard for parallel programming of heterogeneous systems, 2010. URL `http://www.khronos.org`.

[13] 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 *Proc. of the 44th Annual International Symposium on Microarchitecture*, pages 308–317, 2011.

[14] NVIDIA. GPU Computing SDK. http://developer.nvidia.com/gpu-computing-sdk.

[15] NVIDIA. NVIDIA's next generation CUDA compute architecture: Kepler GK110, 2012. www.nvidia.com/content/PDF /NVIDIA_Kepler_GK110_Architecture_Whitepaper.pdf.

[16] NVIDIA. Sharing a GPU between MPI processes: Multi-process service (MPS) overview, 2014. http://docs.nvidia.com/deploy/mps /index.html.

[17] NVIDIA. NVIDIA GeForce GTX 980: Featuring Maxwell, the most advanced GPU ever made, 2014. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF.

[18] NVIDIA. NVIDIA CUDA C Programming Guide, version 7.5, 2015.

[19] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 407–418, Mar. 2013.

[20] J. J. K. Park, Y. Park, and S. Mahlke. ELF: Maximizing memory-level parallelism for GPUs with coordinated warp and fetch scheduling. In *Proceedings of SC15: the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2015.

[21] J. J. K. Park, Y. Park, and S. Mahlke. Chimera: Collaborative preemption for multitasking on a shared GPU. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 593–606, Mar. 2015.

[22] T. G. Rogers, M. OĆonnor, and T. M. Aamodt. Cache-conscious wavefront scheduling. In *Proc. of the 45th Annual International Symposium on Microarchitecture*, pages 72–83, 2012.

[23] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating system abstractions to manage GPUs as compute devices. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles*, pages 233–248, 2011.

[24] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley and Sons, Inc., 8th edition, 2013.

[25] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Mar. 2012.

[26] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on GPUs. In *Proc. of the 41st Annual International Symposium on Computer Architecture*, pages 193–204, 2014.

[27] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[28] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[29] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proc. of the 10th European Conference on Computer Systems*, 2015.

[30] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *Proc. of the 22nd International Symposium on High-Performance Computer Architecture*, pages 358–369, Mar. 2016.

[31] 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 *Proc. of the 2015 International Conference on Supercomputing*, pages 119–130, June 2015.

[32] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram. Warped-slicer: Efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming. In *Proc. of the 43rd Annual International Symposium on Computer Architecture*, 2016.

[33] Y. Zhang and J. D. Owens. A quantitative performance analysis model for GPU architectures. In *Proc. of the 17th International Symposium on High-Performance Computer Architecture*, pages 382–393, Feb. 2011.