2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming

Qiumin Xu*, Hyeran Jeon†, Keunsoo Kim‡, Won Woo Ro‡, and Murali Annavaram*
* *Ming Hsieh Department of Electrical Engineering, University of Southern California*
{*qiumin, annavara*}*@usc.edu*
† *Department of Computer Engineering, San Jose State University*
hyeran.jeon@sjsu.edu
‡ *School of Electrical and Electronic Engineering, Yonsei University*
{*keunsoo.kim, wro*}*@yonsei.ac.kr*

***Abstract*—As technology scales, GPUs are forecasted to incorporate an ever-increasing amount of computing resources to support thread-level parallelism. But even with the best effort, exposing massive thread-level parallelism from a single GPU kernel, particularly from general purpose applications, is going to be a difficult challenge. In some cases, even if there is sufficient thread-level parallelism in a kernel, there may not be enough available memory bandwidth to support such massive concurrent thread execution. Hence, GPU resources may be underutilized as more general purpose applications are ported to execute on GPUs. In this paper, we explore multiprogramming GPUs as a way to resolve the resource underutilization issue. There is a growing hardware support for multiprogramming on GPUs. Hyper-Q has been introduced in the Kepler architecture which enables multiple kernels to be invoked via tens of hardware queue streams. Spatial multitasking has been proposed to partition GPU resources across multiple kernels. But the partitioning is done at the coarse granularity of streaming multiprocessors (SMs) where each kernel is assigned to a subset of SMs. In this paper, we advocate for partitioning a single SM across multiple kernels, which we term as intra-SM slicing. We explore various intra-SM slicing strategies that slice resources within each SM to concurrently run multiple kernels on the SM. Our results show that there is not one intra-SM slicing strategy that derives the best performance for all application pairs. We propose *Warped-Slicer*, a dynamic intra-SM slicing strategy that uses an analytical method for calculating the SM resource partitioning across different kernels that maximizes performance. The model relies on a set of short online profile runs to determine how each kernel's performance varies as more thread blocks from each kernel are assigned to an SM. The model takes into account the interference effect of shared resource usage across multiple kernels. The model is also computationally efficient and can determine the resource partitioning quickly to enable dynamic decision making as new kernels enter the system. We demonstrate that the proposed *Warped-Slicer* approach improves performance by 23% over the baseline multiprogramming approach with minimal hardware overhead.**

***Keywords*-GPUs; scheduling; multiprogramming; multi-kernel; resource management;**

I. Introduction

Each new generation of GPUs has delivered more powerful theoretical throughput empowered by ever-increasing amount of execution resources [1]–[4]. Traditional graphics-oriented applications are successful in exploiting the resource availability to improve throughput. With the advent of new programming models, such as OpenCL [5] and CUDA [6], general purpose applications are also relying on GPUs to derive the benefits of power-efficient throughput computing. However, resource demands across general purpose applications can vary significantly, leading to the widely-studied issue of GPU resource underutilization [7]–[17].

One of the promising solutions to resolve the resource underutilization issue is multiprogramming, where kernels from diverse applications can be concurrently executed on a GPU. Several new design features of recent generations of GPUs are encouraging this trend.

The HSA foundation co-led by AMD introduced a queue-based multiprogramming approach for heterogeneous architectures that include GPUs [18], [19]. NVIDIA also introduced *concurrent kernel execution (CKE)* that allows multiple kernels of an application to share a GPU. For instance, *Hyper-Q* was introduced by NVIDIA for the Kepler architecture which enables kernels to be launched to the GPU via 32 parallel hardware queue streams [2]. These hardware mechanisms use a *Left-Over* policy that assigns as many resources as possible for one kernel and then accommodates another kernel if there remain sufficient resources [20]. These simple policies enable concurrent execution of kernels only opportunistically.

Inspired by the support for concurrent execution, several researchers have proposed microarchitectural and software-driven approaches to concurrently execute kernels from different applications more aggressively [7], [8], [10], [12]. These studies showed that concurrent kernel execution can be beneficial for improving resource utilization especially when the kernels have complementary characteristics. For example, when a compute-intensive kernel and a memory-intensive kernel from different applications share a GPU, both pipeline and memory bandwidth are well utilized without compromising either kernel's performance. Several models [7], [10], [13] have been proposed to find the optimal pair of kernels that can be executed concurrently for better performance and energy efficiency. Software-driven approaches have used kernel resizing to maximize the opportunity for concurrent execution, even within the current Left-Over policy. Kernel slicing [10] partitions a big kernel into smaller kernels so that any single kernel does not consume all available resources. Elastic kernel [8] runs a function that dynamically adjusts kernel size by using resource availability information. Refactoring kernels and rewriting application code to improve concurrency have

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.29

shown that there are significant performance advantages with concurrent execution. However, it may not be feasible or desirable to modify and recompile every application for improving concurrency. To reap the benefits of concurrent kernel execution more broadly, in this paper we focus on hardware-driven multiprogramming approaches, where kernels from diverse applications can be automatically launched to the GPU without software modifications.

One hardware-driven multiprogramming approach is spatial multitasking [12], a microarchitectural solution that splits the streaming multiprocessors (SMs) in a GPU into at least two groups and allows each SM group to execute a different kernel. Unlike the Left-Over policy, which allows multiple kernels to run only if there is enough space, spatial multitasking enables at least two different kernels to concurrently run on a GPU without prioritizing just one kernel over the other. As different applications have their own dedicated set of SMs, we refer to this approach as inter-SM slicing. Inter-SM slicing is a simple way to preserve concurrency with minimal design changes. While spatial multitasking enables better utilization of GPU-wide resources such as memory bandwidth, they do not address the resource underutilization issue within an SM. For example, if a concurrent thread array (CTA) within a kernel requires 21% of the shared memory then only four CTAs can be launched on the SM which leaves 16% of the shared memory to be wasted. Thus, if the available resource in an SM is not an integer multiple of the required resource of a CTA, there will be resource fragmentation.

Inspired by these challenges, this paper explores another approach to resolve resource underutilization issue within an SM while preserving concurrency. We propose *Warped-Slicer*, which is a technique that enables efficient sharing of resources within an SM across different kernels. For example, two compute-intensive kernels can be concurrently run on an SM without compromising their performance if each of the kernels has computation intensity in different kinds of instructions such as an ALU-intensive application and an SFU-intensive application. Each kernel may have different resource demand and performance behavior, and we try to minimize the performance impact suffered by each kernel when multiple kernels are assigned to the same SM. Lee et al. [9] discussed the potential benefits if one were to support intra-SM slicing. Since the focus of their paper is to design thread block scheduling policies, they did not explore microarchitectural design challenges of concurrently executing multiple kernels on the SM and what are the best policies for resource assignment between the two kernels.

We first present a scalable intra-SM resource allocation algorithm across any number of kernels. The goal of this algorithm is to allocate resource to kernels so as to maximize resource usage while simultaneously minimizing the performance loss seen by any given kernel due to concurrent execution. This algorithm is similar to the water-filling algorithm [21] that is used in communication systems for equitable distribution of resources. We present the algorithm assuming we have oracle knowledge of each application's performance versus resource demands. We then show that we can approximate the oracle knowledge by doing short on-line profiling runs to collect these statistics. Next, we describe how the profiling can be done efficiently to identify when to use inter-SM slicing and when to activate intra-SM slicing. Through extensive evaluation, we show that the proposed dynamic partitioning technique significantly improves the overall performance by 23%, fairness by 26% and energy by 16% over the baseline Left-Over policy.

II. Methodology and Motivation

Table I
Baseline Configuration

Parameters	Value
Compute Units	16, 1400MHz, SIMT Width = 16x2
Resources / Core	max 1536 Threads, 32768 Registers max 8 CTAs, 48KB Shared Memory
Warp Schedulers	2 per SM, default gto
L1 Data Cache	16KB 4-way 64MSHR
L2 Cache	128KB/Memory Channel, 8-way
Memory Model	6 MCs, FR-FCFS, 924MHz
GDDR5 Timing	t_{CL}=12, t_{RP}=12, t_{RC}=40, t_{RAS}=28, t_{RCD}=12, t_{RRD}=6

A. Multiprogramming Support in GPUs

There is growing support for enabling multiprogramming in GPUs. In the Kepler and Maxwell generation, up to 32 concurrent streams are mapped into multiple hardware work queues, which removes false dependency among concurrent streams; this technology is branded as *Hyper-Q* [2], [3]. Recent GPUs allow grid launch inside a GPU kernel code for reducing CPU-GPU context switching time if the amount of the work should be adjusted dynamically [25]. The HSA foundation, co-led by AMD introduced a queue-base multiprogramming approach for heterogeneous architectures that include GPUs [18]. HSA-compatible GPUs that support TLBs adopt multiprogramming even in the SM level [4], [26]. However, there is no publicly available documentation on how resource partitioning is done among applications inside an SM. As such it is imperative to understand the best resource partitioning approach to maximize resource utilization while at the same time preventing conflicting resource demands.

B. Methodology

In this section, we first show motivational data regarding how resource utilization varies across different application categories and how different applications face different hurdles to reduce their stall times. We used GPGPU-Sim v3.2.2 [24] in our evaluation and our configuration parameters are described in Table I. GPGPU-Sim front-end is extensively modified to allow multiple processes to concurrently share the same execution backend.

We studied a wide range of GPU applications from image processing, math, data mining, scientific computing and finance domains. These applications are from the CUDA SDK [6], Rodinia [22], Parboil [23] and ISPASS [24] benchmark sets, as summarized in Table II. To quantify the resource utilization of each benchmark we ran each benchmark in isolation for two million cycles without any multiprogramming.

Table II
RESOURCE UTILIZATION FLUCTUATES ACROSS 10 GPGPU APPLICATIONS (ARITHMETIC MEAN OF ALL CORES ACROSS TOTAL CYCLES).

Application	Abbr.	Inst.	Reg.	Shm.	ALU	SFU	LS	Griddim	Blkdim	L2 MPKI	Type	Profile%
Blackscholes [6]	BLK	0.9B	95%	0%	48%	73%	84%	480	128	51.3	Memory	0.7%
Breadth First Search [22]	BFS	0.6B	71%	0%	14%	6%	46%	1954	512	84.4	Memory	5%
DXT Compression [6]	DXT	1.2B	56%	33%	47%	11%	21%	10752	64	0.03	Compute	0.25%
Hotspot [22]	HOT	0.7B	84%	19%	41%	22%	75%	7396	256	5.8	Compute	0.36%
Image Denoising [6]	IMG	1.7B	43%	0%	81%	30%	11%	2040	64	0.3	Compute	0.14%
K-Nearest Neighbor [22]	KNN	0.4B	37%	0%	14%	26%	42%	2673	256	100.0	Memory	6%
Lattrice-Boltzmann [23]	LBM	0.2B	98%	0%	7%	1%	100%	18000	120	166.6	Memoy	0.25%
Matrix Multiply [23]	MM	0.6B	86%	5%	52%	1%	34%	528	128	1.7	Compute	0.25%
Matrix Vector Product [23]	MVP	0.2B	74%	0%	9%	7%	96%	765	192	89.7	Cache	0.9%
Neural Network [24]	NN	0.9B	94%	0%	43%	22%	89%	54000	169	3.7	Cache	0.25%

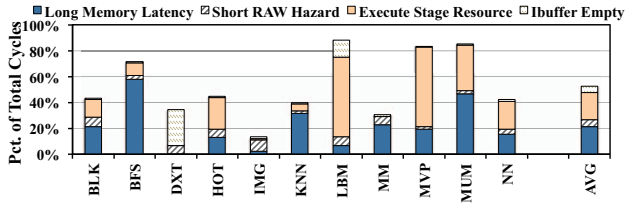


Figure 1. Fraction of total cycles (of all cores) during which warps cannot be issued due to different reasons.

C. Motivational Analysis

Table II shows the resource utilization of the target applications. We chose large input size to avoid GPU resource underutilization due to insufficient input. The grid dimension and block dimensions used in each application are shown in the table, labeled *Griddim* and *Blkdim*, respectively. *Inst*. column shows the total number of instructions executed during the two million cycles for each benchmark. We measured the register (labeled *Reg* in the table) and shared memory (*Shm*) demand of applications. This information can be obtained at compile time without any simulations. We also measured the average utilization of functional units (*ALU*, special function units *SFU*, and load/store units *LS*) while executing each application. As shown, applications have diverse resource usage. KNN uses 37% of registers while LBM utilizes 98%. IMG barely issues LS resources, while MVP consumes 96% of the LS resources. On average, 70% of registers, 6% of shared memory, 31% of ALU units, 11% of SFUs and 49% of LD/ST units are utilized.

The *Type* column classifies the benchmarks into memory or compute intensive based on whether the L2 misses per kilo warp instructions executed (labeled *L2 MPKI*) is high (≥ 30) or low. We chose 30 as the threshold because there's a large gap (10-50) between high *L2 MPKI* benchmarks and low *L2 MPKI* benchmarks. The cache type denotes the L1 Cache Sensitive applications in Section IV. The *Profile%* column will be discussed later.

We measured the fraction of cycles when an application is stalled because no new warps can be issued due to a variety of structural and functional reasons. Figure 1 shows the fraction of cycles during the total execution cycles where no warps are executed due to various stalls. Long memory latency stalls and execution stage resource stalls (the required functional unit is unavailable) in total waste 40% of GPU cycles. There are also short RAW stalls (read-after-write dependencies). The i-buffer empty stalls are when the warps are waiting for next instruction to be fetched. As shown, not all applications suffer the same set of bottlenecks. For instance, *DXT* is mostly waiting for the instruction fetch, while *BFS* is awaiting for response from memory system.

The fact that different applications demand different resources (as shown in Table II) and different applications are stalled by different constraints (as shown in Figure 1) suggests that there is a potential for improving performance by combining different application pairs that have differing resource needs and stall reasons. For instance, we can choose *DXT* and *BFS* to co-locate in the same SM. Furthermore, concurrent kernel execution in the same SM can get around the design imposed limits on how many thread blocks can be launched from a given kernel. For example, the maximum number of CTAs allocated to an SM is limited by the total number of available registers, shared memory size, available warps, maximum CTA count allowed by the GPU. A majority of kernels are limited by only one of these limits [27]. When a given resource (e.g., the register file) is underutilized by one kernel because it has reached the usage limits on a different resource (e.g., the shared memory), it is possible to co-locate another kernel that places very little demand on the shared memory but can make use of the unused registers.

In this paper, we propose to identify the best multiprogramming approach for a given set of kernel types, without any software modifications. Proper resource partitioning across multiple co-located kernels within an SM is a challenging problem. One of the challenges is that application performance does not necessarily improve proportionally to the amount of resources that are assigned to it. The relationship between performance and resource allocation is mostly non-linear and sometimes even non-convex, as shown later in Figure 3a. Therefore designing an efficient algorithm for GPUs to optimize SM resource allocation with various constraints can be a challenging task. To the best of our knowledge, none of these challenges, policies and performance prediction models have been studied in-depth in prior works. There has been a plethora of work in CPU space to enable efficient concurrent application execution by equitable sharing of resources, such as last level cache and memory bandwidth [28]–[36]. But in a GPU the size of the register file far exceeds the size of the cache and the number of execution lanes is at least an order magnitude more than in a CPU. Hence, GPUs present unique challenges for intra-SM sharing, which must be addressed.

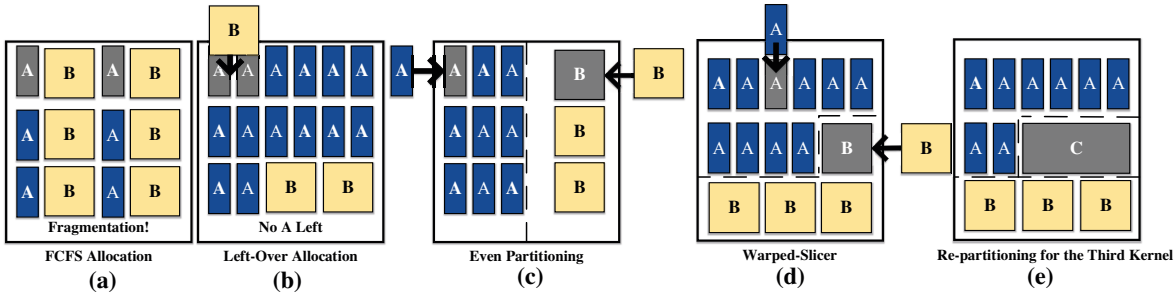


Figure 2. Illustration of proposed storage resource allocation strategies for improving resource fragmentation.

III. Intra-SM Slicing

In this section, we first discuss some intra-SM slicing approaches that we consider for evaluation in this paper. We illustrate these approaches in Figure 2. The resource allocation strategies we discussed here are equally applicable to register file, shared memory and threads; for simplicity, we use shared memory in this section as an example. We assume that two kernels named A and B are running together on the same SM and each of the kernel A's CTAs request only 50% of the shared memory that is required by a CTA of kernel B.

The outermost box graphically represents the total shared memory in an SM. The shared memory required by a CTA from kernel A is represented as a rectangle while shared memory required by a CTA from kernel B is represented as a larger square. In Figure 2a, shared memory is allocated in a First-Come-First-Serve (FCFS) manner. For example, as shown in the figure, if CTAs of kernel A and B are assigned to an SM in a interleaved manner, then kernel A and kernel B's shared memory allocations are interspersed in the shared memory. When a CTA from kernel A finished (a gray-colored rectangle), the deallocated shared memory region is not large enough to fit a CTA from kernel B. As a result, all the shared memory originally assigned to A will be fragmented after kernel A terminates and those regions cannot be used for the newly arriving CTAs of kernel B.

The second strategy to consider is the concurrent kernel execution approach that uses Left-Over allocation strategy. Figure 2b shows the Left-Over strategy. Under Left-Over, Kernel A is given all the shared memory it needs, and only when it does not need any more resources is the remaining memory assigned to kernel B. Only when two adjacent CTAs of kernel A finish, a new CTA from B can take the resources used by the two CTAs of kernel A. Note that when the first CTA from A finishes execution, one has to wait until the second CTA from A to finish before assigning CTA B.

The third strategy to run two kernels is to apply even spatial partitioning [12]. Spatial multitasking was previously proposed for assigning different sets of SMs to different kernels. We use the same approach for intra-SM slicing. We evenly split the resources across the two kernels. As such half of the register file and shared memory are given to each kernel. As shown in Figure 2c, kernel A and B are assigned half of the shared memory region from the beginning of the execution; left half of the shared memory is dedicated to kernel A and the right half is reserved by kernel B. Whenever a CTA from A terminates a new CTA from A is assigned. However, even-split may limit the shared memory usage. For example, although there are remaining resources on the right half that can accommodate CTAs of kernel A, kernel A cannot use those resources.

The last strategy which is proposed in this paper is the *Warped-Slicer*. At the beginning when two kernels start running on the GPU, we determine the best partition of the register and shared memory resources. To maximize the resource utilization, the partition can assign more registers to kernel A and provide more shared memory to kernel B. After the initial partition is done, the CTA from kernel A can only replace another CTA from kernel A.

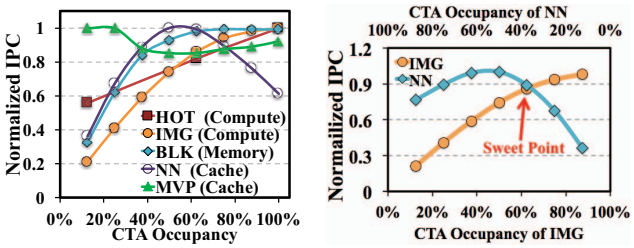
Figure 2e illustrates how the proposed Warped-Slicer policy can be easily extended to more than two kernels. When a third kernel comes, we launch a new resource repartitioning phase for the three kernels. Then the GPU runtime reallocates some of the currently used resources for the third kernel C (the gray-colored rectangle). From that point on, kernel A and kernel B will issue no more CTAs to use the marked resources. Kernel C will then start to execute once the assigned resources are freed from A and B.

In this paper, we evaluated the strategies described in this section in depth. As we show later in our results section, *Warped-Slicer* is significantly better than *even spatial partitioning*. However, Warped-Slicer requires a way to estimate the resource allocations across multiple kernels that maximize the resource usage and improve the cumulative performance. In the next two sections, we will present two performance prediction models for achieving this goal.

IV. Intra-SM Resource Partitioning Using Water-Filling

In this section, we present an analytical method for calculating the resource partitioning that maximizes performance. We present the analytical model assuming we have full knowledge of each application's performance versus resource demands. Such oracle knowledge may be gained for instance by running each application with varying amounts of resources and measuring the performance. In the next section, we show how to realistically collect simple microarchitectural statistics to replace the oracle knowledge.

Before presenting the details of the algorithm, we show an approach for classifying applications based on their performance scalability with thread-level parallelism. This classification is used by the partitioning algorithm later. Figure 3a shows how the application performance varies



(a) (b)

Figure 3. (a) Performance vs. increasing CTA occupancy in one SM, (b) identify the performance sweet spot.

when the number of CTAs assigned to an SM increases. The X-axis shows the number of CTAs allocated as a fraction of the maximum allowed CTAs for a given benchmark. In our experiments, a maximum of eight CTAs can be assigned to an SM. However, some benchmarks may need more resources than are provided in an SM to execute eight CTAs. In this case, the maximum allowed CTAs could be less than eight CTAs. The Y-axis shows the IPC of the application normalized to the best IPC the application achieves. The behavior diverges across different applications [9], [37]–[39]. The resulting graphs can be empirically classified into the following categories.

- Compute Intensive-Non Saturating: The performance continues to increase as more CTAs are assigned to an SM. This behavior is shown by benchmarks like HOT.
- Compute Intensive-Saturating: The performance continues to increase with CTAs but then it saturates. This behavior is shown by benchmarks such as IMG. The performance saturation could be due to pipeline stalls on RAW dependencies.
- Memory Intensive: These benchmarks, such as BLK, also exhibit increasing performance with CTA count but they saturate rather quickly. If an application is memory intensive (number of L2 cache misses as a fraction of total number of instructions executed is large), then the performance saturates much more quickly than the applications in the compute intensive-saturation category.
- L1 Cache Sensitive: L1 Cache sensitive applications continue to increase their performance with CTA count up to the point when the L1 cache is filled up. At that point, adding more CTAs to the SM results in L1 cache thrashing and performance degradation. This is the case with both NN and MVP benchmarks.

The goal is to find the resource distribution across two applications so as to achieve optimal performance when two applications are combined to run on the same SM. Since GPUs allocate resources at the CTA level, resource distribution can be translated into how many CTAs from each application are assigned to an SM. Figure 3b illustrates visually how many CTAs are assigned to each of the two applications. In this illustration, we select two applications: IMG which is a Compute Intensive-Saturating application and NN which is in the Cache Sensitive application category. We plot IMG's resource occupancy versus performance on the primary X-axis. We then plot a mirrored image of NN plot on the secondary X-axis. NN graph shows how its performance varies as resource occupancy decreases from 100% to 0%. By plotting the two graphs in this manner, we see that the total use of resources from two applications is always equal to 100% at any given X-axis point. This figure clearly illustrates why even partitioning of the SM resources to these two applications is sub-optimal; the performance of NN is maximized, but IMG suffers a massive 30% performance loss, compared to the peak achievable IPC. On the other hand, if we select 60% resources for IMG and 40% resources for NN, then IMG and NN each suffers only 10% performance loss compared to the peak performance achieved when each application is executed sequentially. Thus, we can maximize the benefits of running the two applications concurrently.

Based on the intuition provided above we propose an optimization model that relies on the performance and resource utilization data to find the best concurrent execution approach. We find that there exists a **sweet spot**, where the performance degradation of each application is minimized when running both applications concurrently. The sweet spot partitioning is captured by the following optimization function:

$$\text{Max} \ \min_{i} \ P(i,T_i) : \sum_{i=1}^{K} R_{T_i} \leq R_{tot} \tag{1}$$

where $P(i,T_i)$ is the performance of application i normalized to the maximum achievable performance when T_i CTAs are assigned to the application, K is the number of applications sharing the SM. R_{T_i} is the resource requirement of T_i CTAs. The sum of all the resource requirements should be less than the total resources available in an SM (R_{tot}). Thus, the optimization tries to find the minimum performance loss across all the applications assigned to an SM subject to the constraint that the total resource usage does not exceed available SM resources.

The detailed algorithmic implementation is shown in Algorithm 1. We use two vectors, Q_i and M_i. Q_i stores the incremental best performance achieved by running increasing # of CTAs from Kernel i. M_i maintains the # of CTAs that can achieve the performance stored in Q_i. g_i is the index pointing to the current allocation of resources in M_i and Q_i. Initially, each kernel is assigned one CTA. Then in each iteration, we identify a kernel that loses performance the most compared to its peak achievable performance. Then, we assign the minimum number of CTAs that can improve the kernel's performance. We use T_i CTAs for kernel i as the best SM partition strategy. T_i is iteratively updated to find the optimal number of CTAs that minimize the performance loss due to concurrent execution across all applications. K is the number of applications sharing the SM and N is the maximum concurrent number of CTAs restricted by an SM. The time and space complexities of Algorithm 1 are both $O(KN)$, which is superior to a brute-force implementation, where the complexity is $O(N^K)$. Note that the model described above is inspired by the water-filling algorithm [21] which is used extensively in communication systems for distributing resources such as bandwidth across multiple competing users. However, water-filling algorithm [21] uses

Algorithm 1 Water-Filling Partitioning Algorithm

$R_L = R_{tot}$ ▷ R_L represents total resources left
▷ $P_{i,j}$ stores the perf of kernel i with j CTAs
▷ $Q_{i,d}$ stores the max perf with less than or equal to j CTAs, elements of the same value are not stored
▷ M_i stores the associated # of CTAs that lead to Q_i
▷ Initialize vectors for all kernels
for i = 1 ... K **do** ▷ K is the max # of Kernels
  $max = 0, d = 0$ ▷ d is the index of Q and M
  for j = 1 ... N **do** ▷ N is the max # of CTAs
    if $P_{i,j} > max$ **then**
      $max = Q_{i,d} = P_{i,j}$, $M_{i,d} = j$, d++
    end if
  end for
  $T_i = 1$, g_i=1 ▷ T_i is # of CTAs assigned to Kernel i. g_i points to current resource allocation in M and Q. Initially minimum 1 CTA is allocated to each kernel.
  $R_L = R_L - R_i$
end for
while $R_L >= 0$ **do**
  find = false, mp = MAX ▷ mp: Min perf.
  for i= 1 ... K **do**
    if not Full(i) and $Q_{i,g_i} < mp$ **then**
      find = true, mp = Q_{i,g_i}, $S = i$ ▷ S is the selected kernel with min perf. to assign the next CTA
    end if
  end for
  if not find **then** break;
  end if
  $dT = M_{S,g_S+1} - M_{S,g_S}$ ▷ dT: minimum amount of CTAs required to have incremental perf increase
  ▷ R_S represents the resource required to allocate one CTA from the selected kernel
  if $R_L >= R_S \cdot dT$ **then**
    $R_L = R_L - R_S \cdot dT$, g_S++, $T_S = T_S + dT$
  else
    Full(S) = true; ▷ No more resource should be allocated to kernel S
  end if
end while

continuous functions while our proposed solution is discrete.

One potential issue with this algorithm is that it only tries to minimize the performance loss across various CTA combinations. If a performance loss upper-bound is not set, some applications may lose too much performance due to concurrent execution. Therefore, we disband the co-location of multiple kernels in the same SM when the performance loss exceeds a threshold. In such case, we simply fall back on spatial multitasking.

We set the performance loss threshold of any single kernel to $\frac{1}{K} \times 120\%$ if K kernels are concurrently sharing an SM. As we will show later in Section V, only two pairs of applications chose spatial multitasking over intra-SM partitioning. And even with a higher threshold value, the majority of the application pairs gained significant performance benefits from intra-SM partitioning.

A. Profiling Strategy

Recall that the water-filling algorithm's description in the previous section relies on the availability of performance versus the number of CTAs for each kernel. However, in practice, the impact of CTA count on performance is not available a priori. Hence, in this section we present a simple hardware-based dynamic profiling strategy to estimate the performance versus CTA allocation for each kernel. In essence, we compute Q_i and M_i for each application based on a short runtime profile rather than the entire application run. One of the unique aspects of a GPU design is that there are a plethora of identical SMs. We utilize these parallel SMs to measure the performance impact of varying CTA count for each of the K kernels that are being co-located in an SM. We use two kernels as an illustration to describe our profiling approach. However, our profiling technique is applicable to any number (K) of kernels. As shown in Figure 4, we first divide the available SMs equally between the two kernels. We then assign a sequentially increasing number of CTAs from each of the two kernels to its allocated SMs. For example, we assign from kernel 1, 1 CTA to SM0, 2 CTAs to SM1, 3 CTAs to SM2 etc. Similarly, we assign from kernel 2, 1 CTA, 2 CTAs, 3 CTAs each to a different set of SMs. For instance, if a GPU has 16 SMs and each SM can accommodate 8 CTAs then kernel 1 and kernel 2 will each use 8 SMs during the profile phase. Each SM will run anywhere from 1 to 8 CTAs. Note that this approach is scalable to multiple kernels by simply time sharing one SM to run with a different number of CTAs sequentially and then collect the Q_i and M_i for each kernel.

We then employ a sampling phase to measure the IPC of each SM as it executes a given number of CTAs in isolation for 5K cycles. While the L1 cache miss rate will not be impacted by application executions on other cores, the L2 and memory accesses are shared across all SMs. The SM with more CTAs tends to consume higher memory bandwidth. As a result, the sampling phase may not accurately measure the performance of each application when running with a given number of CTAs on a GPU in isolation. To solve this problem, we design a scaling factor, inspired by recent work [40], to offset this memory bandwidth contention penalty. We found the scaling factor to be highly effective in predicting performance while accounting for differing bandwidth demands across the SMs.

Recently, Jog and Kayiran observed that the IPC, DRAM bandwidth and L2 MPKI have the following relationship for memory intensive applications on GPU [40]:

$$IPC \propto \frac{BW}{MPKI} \tag{2}$$

Based on Equation 2, we design a weight factor to offset the imbalance problem as follows:

$$\begin{aligned} IPC_{scaled} &= IPC_{sampled} * factor \\ factor &= 1 + \phi_{mem} * \psi \\ \psi &= \frac{B_{scaled} \times MPKI_{sampled}}{B_{sampled} \times MPKI_{scaled}} - 1 \end{aligned} \tag{3}$$

where the $IPC_{sampled}$, $B_{sampled}$ and $MPKI_{sampled}$ are the IPC, bandwidth, L2 cache miss rate measured during the

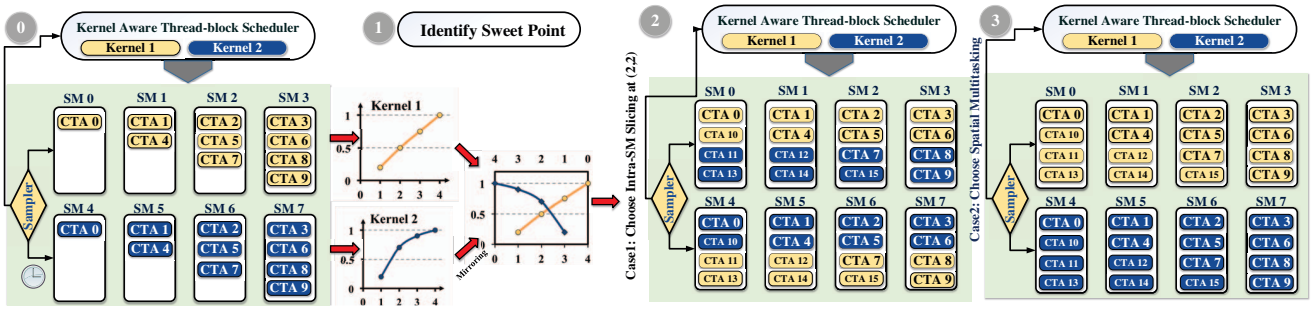


Figure 4. Illustration of the proposed profiling strategy used in Warped-Slicer.

sampling period. The IPC_{scaled} is the projected IPC based on the adjusted new bandwidth B_{scale} and the new L2 cache miss rate $MPKI_{scaled}$. ϕ_{mem} is the portion of pipeline stalls which are caused by long memory latency out of total sampling cycles.

Empirically we observed that L2 MPKI changes minimally with the number of CTAs. Intuitively, MPKI measures misses normalized to thousand instructions. Hence, irrespective of the number of CTAs MPKI seem to fairly stable in our empirical observations. Hence, ψ is directly proportional to the bandwidth usage factors. The amount of bandwidth consumed is proportional to the number of CTAs assigned to an SM. As a result, for each SM we can simplify the ψ computation as follows:

$$\psi \approx \frac{CTA_i}{CTA_{avg}} - 1 \tag{4}$$

The entire process flow for using dynamic profiling information to generate the CTA distributions is shown in Figure 4. Since profiling is done only for a small fraction of cycles over the entire kernel execution window the overhead of profiling is negligible. The Profile% column in Table II shows the profiling overhead. The resource partitioning algorithm is $O(KN)$, which is extremely fast in computing the resource distribution. In all our results we show the net performance after accounting for the profiling overhead.

To quantify the impact of using profiled data, rather than the full application run, we compared the number of CTAs assigned to each of the two kernels using the resource partitioning algorithm using both the IPC_{scaled} obtained from sampling and the true IPC obtained from full application runs in isolation. The number of CTAs that were assigned to each of the two kernels is within, at most, one CTA for more than 90% of the kernel pairs. The detailed distribution of CTA allocations across all possible pairs of kernels that we studied in this paper are shown in Table III.

B. Dealing with Phase Behavior

Note that our proposed approach already handles any phase changes between different kernels by profiling every new kernel at its launch time. One remaining concern with profiling is that there is an implicit assumption that the performance data collected will stay stable for the entire kernel duration for that kernel. This concern can be resolved as

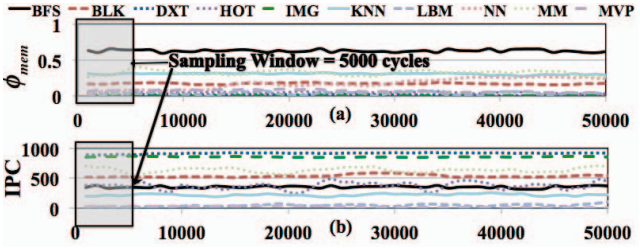


Figure 5. Sampling the program characteristics using a 5K cycles of sampling window.

follows: First, IPC will be monitored during co-execution of a kernel and if the IPC changes significantly and the change is sustained over a long duration (say, at least the length of profile run of 5K cycles), then a new sampling phase may be initiated at that point in execution. During this sampling interval, higher priority is assigned to the sampled kernel while holding back other kernels. While this approach does not provide complete isolation, it may be sufficient to rebuild Q_i and M_i for a kernel with reasonable accuracy. Once the vectors are available we can re-run the resource partitioning algorithm to determine the new resource distribution. But more critically, we looked at significant and sustained IPC changes in kernel execution across several GPU benchmarks. The ϕ_{mem} and average IPC collected per SM during the first 5K cycles is highlighted and compared to a much larger 50K cycle execution window for several benchmarks in Figure 5. Evidently, the sampling window can provide a fairly accurate characterization of the entire kernel execution.

V. Evaluation

In this section, we evaluate our Warped-Slicer (represented as Dynamic in our figures) and compare it with three multiprogramming alternatives that were described earlier: namely, left-over partitioning, even partitioning, and spatial multitasking (Spatial) [12], [13], [41]. Note that spatial multitasking is an inter-SM partitioning scheme while the other schemes are intra-SM partitioning schemes. For generating multi-programmed workloads, we created three categories of benchmarks by pairing compute, cache and memory application types (See Table II). The three categories are Compute + Cache, Compute + Memory, and Compute + Compute. For each of the categories, we generate

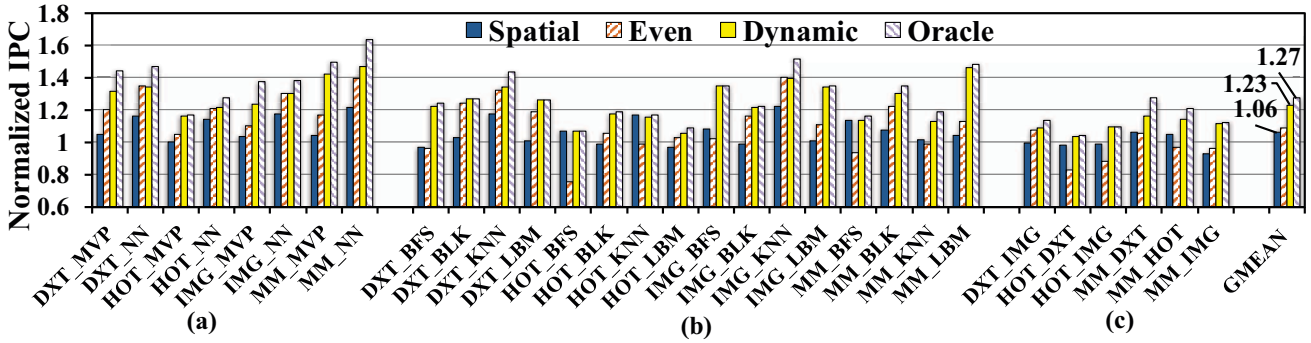


Figure 6. Performance results of all 30 pairs of applications: (a) Compute + Cache. (b) Compute + Memory. (c) Compute + Compute. The results are normalized to baseline Left-Over policy. *GMEAN* shows the overall geometric mean performance across the three workload categories.

Table III
RESOURCE PARTITIONING WHEN WARPED-SLICER (DYN) AND EVEN MULTIPROGRAMMING ALGORITHMS ARE USED.

Compute + Cache			Compute + Memory			Compute + Memory			Compute + Compute		
Workload	Dyn	Even	Workload	Dyn	Even	Workload	Dyn	Even	Workload	Dyn	Even
DXT_MVP	(7,1)	(4,4)	DXT_BFS	(6,2)	(4,1)	IMG_BFS	(6,2)	(4,1)	DXT_IMG	(4,4)	(4,4)
DXT_NN	(4,4)	(4,4)	DXT_BLK	(4,4)	(4,4)	IMG_BLK	(5,3)	(4,4)	HOT_DXT	(2,6)	(1,4)
HOT_MVP	(3,1)	(1,4)	DXT_KNN	(4,4)	(4,3)	IMG_KNN	(4,4)	(4,3)	HOT_IMG	(2,6)	(1,4)
HOT_NN	(2,4)	(1,4)	DXT_LBM	(5,3)	(4,3)	IMG_LBM	(7,1)	(4,3)	MM_DXT	(3,5)	(2,4)
IMG_MVP	(7,1)	(4,4)	HOT_BFS	spatial	(1,1)	MM_BFS	spatial	(2,1)	MM_HOT	(2,2)	(2,1)
IMG_NN	(3,5)	(4,4)	HOT_BLK	(2,4)	(1,4)	MM_BLK	(3,4)	(2,4)	MM_IMG	(3,5)	(2,4)
MM_MVP	(5,1)	(2,4)	HOT_KNN	(2,4)	(1,3)	MM_KNN	(4,4)	(2,3)			
MM_NN	(3,5)	(2,4)	HOT_LBM	(3,1)	(1,3)	MM_LBM	(5,1)	(2,3)			

all combinations of benchmarks from the two categories. Thus, a total of 30 benchmark pairs were generated. While executing these diverse pairs of benchmarks, one challenge is to make sure that any given application pair executes the same amount of work across different multiprogramming approaches evaluated in this paper. To achieve this goal we use the following approach. Recall that for collecting individual application statistics in Table II we ran each benchmark for two million cycles. We recorded the total number of instructions executed during that two million cycles for each benchmark in the *Inst* column. When running the benchmark pair we run each benchmark until it reaches that recorded instruction count. Once a benchmark finishes the target instruction count that benchmark simulation is halted and its assigned GPU resources are released. The slower benchmark may then consume all the available resources to reach its own instruction target. The total simulation time is treated as the execution time for the application pair. This way, each application simulates the same number of instructions under all configurations. For warped-slicer, we set the profiling phase to be 5K cycles long. We wait for 20K cycles for GPU to warm up before starting the first profiling phase. At the end of the profiling phase, the partitioning algorithm reads the profile data and decides the number of CTAs each application is going to get. We compare the profiling cycles over average kernel execution time of each benchmark in the $Profile\%$ column in Table II. As such, the profiling overhead is minimum for most applications.

A. Performance

Figure 6 shows the IPC of various multiprogramming approaches normalized to the IPC of the Left-Over policy. The average IPC of concurrently executed kernels is calculated by dividing the sum of all kernels' instruction count by execution time until all kernels finish. Note that Left-Over policy performs very similar to the sequential execution of the two applications since the second application will not start execution until after the first application is done with issuing all of its CTAs. Overall, the IPC of the Left-Over policy when running the two applications is 3.2% higher than the average IPC of the two applications running sequentially. The Oracle approach is the highest performance we obtained for the application pair among all multiprogramming approaches discussed in this paper (Left-Over, Spatial and Intra-SM Slicing). To identify the best results for intra-SM slicing, we exhaustively ran all possible CTA combinations.

On average, all multiprogramming algorithms derived better performance than the baseline Left-Over policy. The proposed Warped-Slicer approach (Dynamic) outperformed the other algorithms and is close to the oracle results for most applications. Warped-Slicer partitions resources according to the workload's performance and CTA count relationship as measured during the profiling phase. Spatial multitasking achieved only minimal performance improvement over Left-Over. Spatial partitions resources only across the SM boundaries. As explained in the earlier sections, inter-SM slicing can cause resource underutilization within an SM, which cannot be handled by splitting workload across SMs. As expected, the two Intra-SM slicing approaches, Even and Warped-Slicer, derived better performance than Spatial. Warped-Slicer derived an average of 23% performance improvement over Left-Over policy, which is widely used in current GPUs, 14% better than even partitioning of an SM and 17% better than Spatial multitasking.

Table III shows how differently our Warped-Slicer partitions resources than Even approach. When Intra-SM mul-

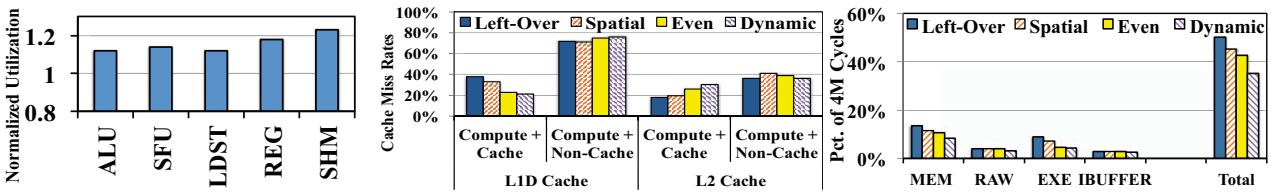


Figure 7. Various performance statistics: (a) assorted resource utilizations of the proposed Warped-Slicer normalized by even partitioning policy, (b) cache miss rates, (c) breakdown of total stall cycles.

tiprogramming is chosen for the workload, the numbers in the parenthesis indicate the number of CTAs run by each of the two applications; each application is assigned *required resource per CTA* $\times$ *# CTAs*. The workloads that run Inter-SM multiprogramming are marked as *Spatial* and each application is equally assigned eight SMs. Each application pair has two numbers listed, the first number corresponds to the number of CTAs assigned to the first application and the second number corresponds to the number of CTAs assigned to the second application. In many cases, Warped-Slicer assigns more CTAs than Even. For example, in HOT_DXT of Compute+Compute category, Warped-Slicer runs two HOT CTAs and six DXT CTAs whereas Even only assigns one HOT CTA and four DXT CTAs. Since each application can use up to 50% of the intra-SM resources in the Even approach, resources may be fragmented if the assigned 50% resource is not an integer-multiple of the required resources of each application. On the other hand, Warped-Slicer finds the optimal resource assignment where performance degradation and resource fragmentation is minimal for both applications.

In some cases, Warped-Slicer assigns the same number of CTAs as the Even partition, as can be seen in MM_MVP. For this workload, the total number of CTAs assigned by Even and Warped-Slicer is the same. However, Warped-Slicer assigns fewer CTAs for MVP because MVP is a cache intensive application. Hence, more CTAs can cause cache trashing, and fewer CTAs improve overall performance. On the other hand, the Even approach assigns the same number of CTAs for both applications, which leads to worse performance than the Warped-Slicer.

B. Resource Utilization

Figure 7a shows how the Warped-Slicer increases ALU and SFU pipeline utilization, register file and shared memory utilization over the Even partitioning approach. The ALU utilization metric is the fraction of cycles when an ALU pipeline is occupied over the total execution cycles. Overall, Warped-Slicer derives over 15% higher utilization of all the GPU resources. This is because the Warped-Slicer chooses the optimal resource partitioning that can minimize the resource underutilization. On the other hand, Even allows each application to use up to half of each resource, regardless each application's resource demand. Therefore, the resource utilization is lower due to resource fragmentation.

C. Cache Misses

When running multiple applications on an SM, the resource utilization can be improved as shown in the previous sections. However, as each application accesses different regions of memory space, cache contention might increase. In Figure 7b, we measured cache miss rates in L1 and L2 caches. We observed different behaviors from different application categories. For Compute + Non-Cache applications, as expected, L1 cache miss rate is the lowest in Left-Over and highest in Even and Warped-Slicer. Interestingly, for Compute + Cache applications, Warped-Slicer achieves the lowest miss rate, which is 2% lower than Even and 17% lower than Left-Over. This is because Warped-Slicer runs fewer cache-intensive CTAs concurrently, which leads to lower L1 cache contention. L2 cache miss rate shows slightly different trend because L2 caches are shared across multiple SMs and hence, data of different applications can contend in the L2 cache even under Spatial approach. Consequently, Spatial derived higher L2 cache miss rate than Left-Over. Warped-Slicer derives highest L2 miss rate in Compute + Cache category because its total L2 accesses reduced significantly by 43% due to lower L1 cache miss rate.

D. Stall Cycles

Multiprogramming not only enhances the pipeline utilization but also reduces stall cycles caused by resource contention. By running compute-intensive and memory-intensive applications together, the memory congestion can be relieved because memory accesses are generated primarily by only half of the concurrently running warps while the other half of the warps stress computational resources.

Figure 7c shows various stall cycles under the multiprogramming approaches. As expected, long latency memory stalls reduced most using the Warped-Slicer. Note that the stall cycles are also reduced with Spatial multitasking that does not share SMs because, when memory-intensive and compute-intensive applications are multiprogrammed, the memory-intensive applications are assigned to only half of the total SMs and hence the memory congestion can be still relieved. Overall, the Warped-Slicer encounters 15% fewer accumulated total stall cycles than Left-Over.

E. Multiple Kernels Sharing SM

In this section, we show that our proposed scheme can work on more than two kernels assigned to concurrently execute on an SM. As described in our algorithmic description, the proposed approach is general and it does

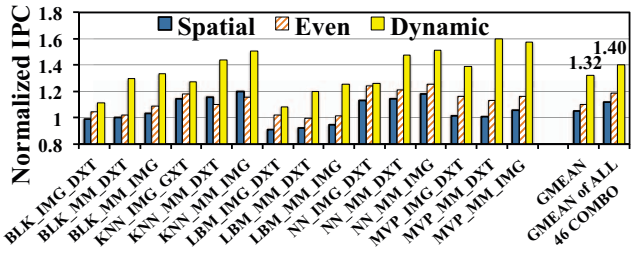
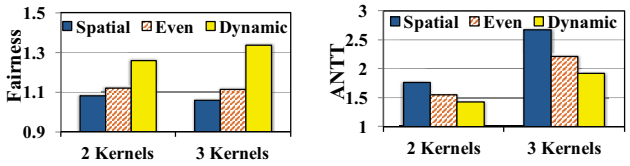


Figure 8. Performance result when combining three applications in an SM.



(a) Fairness(Minimum Speedup) (b) Average Normalized Turn-around Time

Figure 9. Comparison of fairness improvement (Normalized to Left-Over Policy) and ANTT reduction between multiprogramming policies.

not depend on the number of concurrent kernels being considered for execution. We evaluated all the combinations of three applications which contain at least one compute application. BFS and HOT are not included because of their large CTA size, which prevents more than two kernels from being executed. Figure 8 shows all the combinations of memory/cache applications working with two compute applications with the last bar showing the overall performance improvement over all combinations. Warped-Slicer consistently outperforms even partitioning and on average by 21%.

F. Fairness Metrics

Figure 9a shows the minimum speedup across various configurations evaluated. Compared with the Even partitioning, the proposed scheme improves fairness (as measured by the minimum speedup metric) by 14% in 2 Kernels and 23% in 3 Kernels.

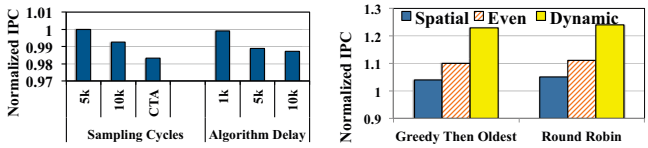
Figure 9b shows the average normalized turnaround time which is another important metric to measure fairness. The Warped-Slicer improves this metric significantly over even-partitioning. It improves by 15% when 3 kernels are running on the SM.

G. Power and Energy Analysis

We use GPUWattch [42] and McPAT [43] for power evaluation. Compared with baseline Left-Over policy, Warped-Slicer increases average dynamic power by 3.1% due to the increased resource utilization. Overall, however, our Warped-Slicer saves the baseline energy consumption by 16% through significantly reduced total execution time.

H. Sensitivity Analysis

We also investigated how the length of the profiling phase influences the prediction correctness. We ran all 30 pairs of



(a) Sensitivity to profiling length and algorithm delay (b) Sensitivity to warp schedulers

Figure 10. Performance sensitivity to profiling parameters and warp schedulers.

applications with the Warped-Slicer while varying profiling length from 5K, 10K cycles, and finally up to the total number of cycles of an entire CTA execution. Figure 10a shows the IPC under various profiling lengths, normalized by the IPC when using 5K cycle for the profiling length. We found that across all the application pairs, the IPC variations are at most 2% with varying profiling lengths.

We then investigated how the algorithm execution delay influences the overall performance. We again ran all 30 pairs of applications with an additional delay varying from 1K cycles, 5K cycles to 10K cycles. between finishing sampling to start the new partition. We found that overall the IPC change is less than 1.5%. The time for calculating the resource-partitioning algorithm does not block the warps from executing on the SM. While the decision is being made, the CTAs already issued in the sampling phase can still execute on the machine. Therefore, even when the additional algorithm delay increases to 10k cycles, the performance loss is still minimal.

Figure 10b studies the performance impact of different warp schedulers. The IPC and speedup of using the Warped-Slicer is not impacted by the underlying warp schedulers that were studied in this paper: greedy then oldest scheduler and the round-robin scheduler.

We also examined the impact of less contended SM resources by evaluating the system with 256KB register file, 96KB shared memory, 32 maximum CTAs and 64 maximum warps. Our Warped-Slicer still significantly improves the performance and fairness of the baseline policy, both by 26%. Since software written for future GPUs will utilize more and more registers and shared memory resources, we believe that our schemes that target resource contention and efficient partitioning will be increasingly important for future generations of GPUs.

I. Implementation Overhead

We synthesized the various profiling counters and the global sampling logic required by Algorithm 1 using NCSU PDK 45nm library [44]. The set of counters for sampling occupies $714um^2$ per SM and the global counters and logic for Algorithm 1 occupies $0.04mm^2$. We also extract the power and area of 16 SMs from GPUWattch [42], which is $704mm^2$ and consumes 37.7W of dynamic power and 34.6W of leakage power. The total area overhead of our proposed approach for 16 SMs is $0.05mm^2$, resulting in only 0.01% area overhead. The total dynamic power is 54mW and the leakage power is 0.27mW, accounting for 0.14% of the dynamic power and 0.001% leakage power overhead.

VI. Related Work

Workload Selection and Task Scheduling in CPUs: Several approaches have addressed the workload selection and assignment problems in CPUs. Snavely et al. [28], [45] first proposed the SOS scheduler which uses profile-based knowledge to select the best symbiosis co-runners. Several other approaches [30], [46]–[50] proposed CPU performance models to construct optimal workload assignments. For optimizing thread scheduling, on-line characterization techniques have been explored for simultaneous multithreaded (SMT) processors. For instance, Choi and Yeung [33] proposed an on-line resource partitioning based on performance monitoring. In SMT enabled CPUs, there are only a few threads. Hence, continuous performance monitoring to decide resource partitioning is acceptable. Our approach monitors the performance versus resource allocation using a small profile run to characterize the full application behavior, which is inspired by existing leader-follower style sampling techniques in CPU domain [32]. Our sampling approach is similar to [33], but different in that we sample with varying number of CTA counts concurrently on different SMs in a GPU thereby collecting the required profiling metrics in a single short profiling phase.

For partitioning on-chip resources, existing works in SMT primarily focused on cache partitioning [29], [31], [32], [35], [36]. In CPUs, the fraction of physical registers and cache can be easily adjusted by controlling the number of in-flight instructions of each thread. However, in GPUs, such dynamic control is not available since each CTA gets all its resources at once. Once a CTA is assigned to an SM, register file and shared memory must be allocated statically and cannot be released until the CTA is completed. Thus, allocation-time resource scheduling is more important in GPUs and hence we focused more on register file and shared memory partitioning rather than caches. In addition, GPUs have a much smaller L1 cache with lower hit rate, since GPU's L1 cache is shared by thousands of threads.

Multiprogramming on GPUs: Several software-centric GPU multiprogramming approaches have been studied. Jiao et al. [7] proposed power-performance models to identify the optimal kernel pairs that can achieve better performance per watt. Their models statically estimate energy efficiency of any pair of applications so that the optimal kernel pair can be multiprogrammed together. Pai et al. [8] proposed elastic kernel design that adjusts each kernel's resource usage dynamically subject to the available hardware resources. They modified application code to run a special function that adjusts resource usage based on current resource utilization. Zhong and He [10] proposed a dynamic kernel slicing that partitions a kernel into several smaller kernels so that multiple kernels can more efficiently share the resources. Adriaens et al. [12] proposed spatial multitasking, which runs multiple applications on different sets of SMs. Ukidave et al. [41] explored several adaptive spatial multiprogramming approaches. Aguilera et al. [51] pointed out the unfairness of the spatial multitasking and examined several task assignment methods for more fair resource usage and better throughput by distributing workloads across SMs. Gregg et al. [11] developed a run-time kernel scheduler that merges two OpenCL kernels so that the kernels can run on a single-kernel-capable GPU.

These software-centric multiprogramming methods improved concurrency and performance significantly by refactoring kernels and rewriting application code. In many situations, it may not be feasible to modify every application for improving concurrency. Also, once a kernel is sliced, the sliced kernel's size cannot be adjusted in the run time, which might cause another inefficiency. Our study proposes a hardware mechanism that does not require any application code modification. We also provide a novel method to determine the best multiprogramming strategy in the run time, which is not applicable to the software-centric approaches.

Recently, several studies show a new trend of hardware support for multiprogramming. The HSA Foundation [18] standardized hardware and software specifications to run multiple applications on heterogeneous system architectures. In the specification, they also cover the execution of multiple applications in the same GPU, which uses multiple simultaneous application queues which are similar in spirit to the NVIDIA's Hyper-Q. Still, there is not a detailed explanation of how the applications are assigned to execution cores.

Wang et al. [52] proposed a dynamic CTA launching method for irregular applications. They observed that CTA-level parallelism is better than kernel-level parallelism for resource utilization, especially in irregular applications. They proposed a runtime platform that supports dynamic CTA invocation. This work is orthogonal to our approach. We focus more on efficient multiprogramming strategy rather than improving application parallelism. We used a performance model to determine the optimal resource partitioning between two different applications and each application's kernel parameters.

Preemptive scheduling on GPUs: Preemptive scheduling essentially context switches one kernel with another kernel to enable multiprogramming.The main challenge of preemptive scheduling is the high overhead of context switching. Tanasic et al. [53] explored classic context switching and draining. Classic context switching stops running kernel to save the current context to the memory and then a new kernel is brought in. Park et al. [54] added another preemptive scheduling algorithm, flushing. Flushing detects idempotent kernels, which generate exactly the same result regardless how many times the kernel is executed. Then, to run an urgent kernel, they drop a running idempotent kernel and yield corresponding resources to the urgent kernel. Later, the dropped idempotent kernel is re-executed from the beginning. Recently, Yang et. al. [55] proposed a partial context switching technique to allow fine-grain sharing by multiple kernels within each SM. This approach tries to resolve the long context switching delay suffered during preemption. These studies are orthogonal with our study because our study focuses more on multiple kernels' concurrent execution rather than temporal GPU sharing.

Dynamic execution parameter adjustment: Kayiran et al. [38] explored dynamic workload characterization to adjust the number of CTAs on the fly, which derives better

overall performance. Lee et al. [9] proposed a profiling-based single kernel execution optimization for GPU. Lee et al. [56] proposed a dynamic voltage scaling for better throughput under a power budget. They periodically check voltage, frequency and SM core count to adjust the three parameters in the next epoch. Sethia and Mahlke [39] proposed a hardware runtime system that dynamically monitors and adjust several parameters, such as the number of CTAs, core frequency, and memory frequency. The proposed runtime system can be configured either to improve energy efficiency by throttling underutilized resources or to improve performance by boosting core frequency and adjusting the number of CTAs. These four studies used single kernel execution environment as their baseline. On the other hand, our approach determines the optimal resource partitioning to enable multiple kernels concurrently run on the same SM.

VII. Conclusion

In this paper, we present a novel approach for efficiently partitioning resources within an SM across multiple kernels in GPU. The algorithm we proposed follows the water-filling algorithm in communication networks, but we apply it to efficient resource sharing problem within an SM. We demonstrate how to implement this algorithm in practice using a short profile run to collect the statistics required for executing the algorithm. We then evaluated our proposed design on a wide range of GPU kernels and show that our proposed approach improves performance by 23% over a baseline Left-Over multiprogramming approach.

Acknowledgment

We thank Francisco Romero and anonymous reviewers for their valuable comments on this work. We thank Congyin Shi for helping with digital synthesis. This work was supported by DARPA-PERFECT-HR0011-12-2-0020, NSF-CAREER-0954211 and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. NRF-2015R1A2A2A01008281).

References

[1] "Whitepaper: NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™," tech. rep., NVIDIA, 2009.

[2] "Whitepaper: NVIDIA's Next Generation CUDA™ Compute Architecture: Kepler™ GK110," tech. rep., NVIDIA, 2012.

[3] "Whitepaper: NVIDIA GeForce GTX980," tech. rep., NVIDIA, 2014.

[4] "AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE," tech. rep., AMD, 2012.

[5] A. Munshi, "The OpenCL specification," in *Khronos OpenCL Working Group*, 2008.

[6] "NVIDIA CUDA compute unified device architecture - programming guide." https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2008.

[7] Q. Jiao, M. Lu, H. P. Huynh, and T. Mitra, "Improving GPGPU energy-efficiency through concurrent kernel execution and DVFS," in *Intl. Symp. on Code Generation and Optimization (CGO)*, Feb. 2015.

[8] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2013.

[9] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving GPGPU resource utilization through alternative thread block scheduling," in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, Feb. 2014.

[10] J. Zhong and B. He, "Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling," in *Tran. on Parallel and Distributed System (TPDS)*, vol. 25, pp. 1522–1532, 2014.

[11] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, "Fine-grained resource sharing for concurrent GPGPU kernels," in *Conf. on Hot Topics in Parallelism (HotPar)*, June 2012.

[12] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for GPGPU spatial multitasking," in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, Feb. 2012.

[13] Y. Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, and D. Chen, "Efficient GPU spatial-temporal multitasking," in *Tran. on Parallel and Distributed Systems (TPDS)*, vol. 26, pp. 748–760, 2014.

[14] P. Xiang, Y. Yang, and H. Zhou, "Warp-level divergence in GPUs: Characterization, impact, and mitigation," in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, Feb. 2014.

[15] Q. Xu and M. Annavaram, "PATS: Pattern aware scheduling and power gating for GPGPUs," in *Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, Aug. 2014.

[16] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on GPUs: Where are the bottlenecks?," in *Intl. Symp. on Workload Characterization (IISWC)*, Oct 2014.

[17] H. Jeon and M. Annavaram, "Warped-DMR: Light-weight error detection for GPGPU," in *Intl. Symp. Microarchitecture (MICRO)*, Dec. 2012.

[18] H. Foundation, "Heterogeneous system architecture (HSA): Architecture and algorithms," in *Intl. Symp. on Computer Architecture tutorial (ISCA)*, June 2014.

[19] M. Schulte, M. Ignatowski, G. Loh, B. Beckmann, W. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S. Reinhardt, and G. Rodgers, "Achieving exascale capabilities through heterogeneous computing," vol. 35, pp. 26–36, July 2015.

[20] B. Coon, J. Nickolls, J. Lindholm, R. Stoll, N. Wang, J. Choquette, and K. Nickolls, "Thread group scheduler for computing on a parallel thread processor," May 2012. US Patent 8732713.

[21] D. Palomar and J. Fonollosa, "Practical algorithms for a family of waterfilling solutions," in *Tran. on Signal Processing (TSP)*, vol. 53, pp. 686–695, Feb. 2005.

[22] 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 *Intl. Symp. on Workload Characterization (IISWC)*, Oct. 2009.

[23] 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.," tech. rep., Mar. 2012.

[24] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009.

[25] S. Jones, "Introduction to dynamic parallelism," in *Nvidia GPU Technology Conference (GTC)*, May 2012.

[26] K. Wilcox, D. Akeson, H. Fair, J. Farrell, D. Johnson, G. Krishnan, H. Mclntyre, E. McLellan, S. Naffziger, R. Schreiber, S. Sundaram, and J. White, "4.8 A 28nm x86 APU optimized for power and area efficiency," in *Intl. Solid- State Circuits Conf. (ISSCC)*, Feb. 2015.

[27] M. K. Yoon, K. Kim, S. Lee, W. W. Ro, and M. Annavaram, "Virtual Thread: Maximizing thread-level parallelism beyond gpu scheduling limit," in *Intl. Symp. on Computer Architecture (ISCA)*, June. 2016.

[28] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Nov. 2000.

[29] F. J. Cazorla, A. Ramirez, M. Valero, and E. Fernandez, "Dynamically controlled resource allocation in SMT processors," in *Intl. Symp. on Microarchitecture (MICRO)*, Dec. 2004.

[30] F. J. Cazorla, P. M. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero, "Predictable performance in SMT processors: Synergy between the OS and SMTs," in *Tran. on Computers (TOC)*, vol. 55, pp. 785–799, 2006.

[31] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2004.

[32] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Intl. Symp. on Microarchitecture (MICRO)*, Dec. 2006.

[33] S. Choi and D. Yeung, "Learning-based SMT processor resource distribution via hill-climbing," in *Intl. Symp. on Computer Architecture (ISCA)*, June 2006.

[34] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *Intl. Symp. on Computer Architecture (ISCA)*, June 2008.

[35] R. Wang and L. Chen, "Futility scaling: High-associativity cache partitioning," in *Intl. Symp. on Microarchitecture (MICRO)*, Dec. 2014.

[36] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family," in *Intl. Symp. on High Performance Computer Architecture (HPCA)*, Mar. 2016.

[37] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser, "Many-core vs. many-thread machines: Stay away from the valley," in *Computer Architecture Letters (CAL)*, vol. 8, pp. 25–28, 2009.

[38] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for GPGPUs," in *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2013.

[39] A. Sethia and S. Mahlke, "Equalizer: Dynamic tuning of GPU resources for efficient execution," in *Intl. Symp. on Microarchitecture (MICRO)*, Dec. 2014.

[40] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, "Anatomy of GPU memory system for multi-application execution," in *Intl. Symp. on Memory Systems (MEMSYS)*, Oct. 2015.

[41] Y. Ukidave, C. Kalra, D. R. Kaeli, P. Mistry, and D. Schaa, "Runtime support for adaptive spatial partitioning and inter-kernel communication on GPUs," in *Intl. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct. 2014.

[42] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling energy optimizations in GPGPUs," in *Intl. Symp. on Computer Architecture (ISCA)*, June 2013.

[43] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Intl. Symp. on Microarchitecture (MICRO)*, Dec. 2009.

[44] "The freepdk process design kit." http://www.eda.ncsu.edu/wiki/FreePDK.

[45] A. Snavely, D. M. Tullsen, and G. Voelker, "Symbiotic jobscheduling with priorities for a simultaneous multithreading processor," in *Intl. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2002.

[46] S. Eyerman and L. Eeckhout, "Probabilistic job symbiosis modeling for SMT processor scheduling," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2010.

[47] P. Radojković, V. Čakarević, M. Moretó, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero, "Optimal task assignment in multithreaded processors: A statistical approach," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2012.

[48] A. El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas, "Compatible phase co-scheduling on a CMP of multi-threaded processors," in *Intl. Symp. on Parallel and Distributed Processing (IPDPS)*, Apr. 2006.

[49] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2005.

[50] A. Settle, J. Kihm, A. Janiszewski, and D. Connors, "Architectural support for enhanced SMT job scheduling," in *Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, Sept. 2004.

[51] P. Aguilera, K. Morrow, and N. Sung Kim, "Fair share: Allocation of GPU resources for both performance and fairness," in *Intl. Conf. of Computer Design (ICCD)*, Oct. 2014.

[52] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on GPUs," in *Intl. Symp. on Computer Architecture (ISCA)*, June 2015.

[53] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on GPUs," in *Intl. Symp. on Computer Architecuture (ISCA)*, June 2014.

[54] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared GPU," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2015.

[55] 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 *Intl. Symp. on High Performance Computer Architecture (HPCA)*, Mar. 2016.

[56] J. Lee, V. Sathisha, M. Schulte, K. Compton, and N. S. Kim, "Improving throughput of power-constrained GPUs using dynamic voltage/frequency and core scaling," in *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2011.