A Model-Based Software Solution for Simultaneous Multiple Kernels on GPUs

HAO WU, WEIZHI LIU, HUANXIN LIN, and CHO-LI WANG, The University of Hong Kong

As a critical computing resource in multiuser systems such as supercomputers, data centers, and cloud services, a GPU contains multiple compute units (CUs). GPU Multitasking is an intuitive solution to underutilization in GPGPU computing. Recently proposed solutions of multitasking GPUs can be classified into two categories: (1) spatially partitioned sharing (SPS), which coexecutes different kernels on disjointed sets of compute units (CU), and (2) simultaneous multikernel (SMK), which runs multiple kernels simultaneously within a CU. Compared to SPS, SMK can improve resource utilization even further due to the interleaving of instructions from kernels with low dynamic resource contentions.

However, it is hard to implement SMK on current GPU architecture, because (1) techniques for applying SMK on top of GPU hardware scheduling policy are scarce and (2) finding an efficient SMK scheme is difficult due to the complex interferences of concurrently executed kernels. In this article, we propose a lightweight and effective performance model to evaluate the complex interferences of SMK. Based on the probability of independent events, our performance model is built from a totally new angle and contains limited parameters. Then, we propose a metric, *symbiotic factor*, which can evaluate an SMK scheme so that kernels with complementary resource utilization can corun within a CU. Also, we analyze the advantages and disadvantages of kernel slicing and kernel stretching techniques and integrate them to apply SMK on GPUs instead of simulators. We validate our model on 18 benchmarks. Compared to the optimized hardware-based concurrent kernel execution whose kernel launching order brings fast execution time, the results of corunning kernel pairs show 11%, 18%, and 12% speedup on AMD R9 290X, RX 480, and Vega 64, respectively, on average. Compared to the Warped-Slicer, the results show 29%, 18%, and 51% speedup on AMD R9 290X, RX 480, and Vega 64, respectively, on average.

CCS Concepts: • **Theory of computation** → **Scheduling algorithms**; *Parallel computing models*; Probabilistic computation; • **Software and its engineering** → *Software performance*;

Additional Key Words and Phrases: GPGPU, concurrent kernel execution

ACM Reference format:

Hao Wu, Weizhi Liu, Huanxin Lin, and Cho-Li Wang. 2020. A Model-Based Software Solution for Simultaneous Multiple Kernels on GPUs. *ACM Trans. Archit. Code Optim.* 17, 1, Article 7 (March 2020), 26 pages. <https://doi.org/10.1145/3377138>

1 INTRODUCTION

Graphic compute units (GPUs) are widely adopted in computers due to their remarkable computing capabilities and energy efficiency. Applications accelerated with GPUs cover various domains, such as data-intensive scientific applications [3], file systems [24], network systems [27], database

Authors' addresses: H. Wu, W. Liu, H. Lin, and C.-L. Wang, The University of Hong Kong, CB414 Department of Computer Science HKU, Hong Kong; emails: wuhaocatherine@gmail.com, {wzliu, hxlin, clwang}@cs.hku.hk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2020/03-ART7

<https://doi.org/10.1145/3377138>

applications [30], and cloud applications [26]. Furthermore, with increasing computing power and new architecture features, new-generation GPUs can support larger and more complex computing tasks. Observation [21, 29] has shown the on-chip resource underutilization of single-kernel execution. Therefore, while GPUs become more general, the underutilization of GPUs is becoming a more critical issue in modern systems. Efficiently sharing GPUs for general-purpose computing on GPU (GPGPU) applications is of great importance. Programmers write a GPGPU program using CUDA [19] or OpenCL [8] programming models and offload computing to GPU as kernels.

Corunning kernels have drawn extensive attention both in industry and academia [1, 6, 10, 13, 14, 16, 17, 20–22, 28, 29, 31, 33]. The resources used by a kernel include both static resources (threads, registers, and shared memory) and dynamic resources (computing cores, memory load/store units, bandwidth, and memory interconnection). Modern GPU architectures, like NVIDIA Kepler [20] and AMD GCN [16], support concurrent kernel execution to solve such a multiresource allocation problem. However, their LEFTOVER scheduling policy of GPU hardware schedulers [1, 21] decreases the concurrency, because the first launched kernel may use up one of the static resources of a GPU and make other kernels unable to be dispatched. Solutions proposed in recent research to improve the concurrency of GPU kernels can be classified into two categories: (1) spatially partitioned sharing (SPS), which coexecutes different kernels on disjointed sets of compute units (CUs) [1, 10, 13, 17], and (2) simultaneous multikernel (SMK), which runs multiple kernels simultaneously within a CU [14, 21, 22, 29, 31, 33]. In general, SMK can improve the resource utilization even more than SPS, because SMK can launch more threads on a CU by corunning kernels with both complementary static resource requirements and interleaving instructions from kernels with low dynamic resource contentions, while SPS only allows different kernels to corun on disjointed sets of CUs [22].

However, there is a lack of software solutions for applying SMK to GPUs. To implement SMK on GPUs, we need to answer two questions: (1) how to schedule a specified number of workgroups for each kernel to a CU and (2) how to get the most efficient intra-CU sharing. A GPU consists of several CUs or streaming multiprocessors (SMs). Threads of a kernel are organized in a hierarchy: 32 (in NVIDIA GPUs) or 64 (in AMD GPUs) threads are grouped into a warp or a wavefront, and several warps or wavefronts compose a workgroup (in OpenCL). In general, a GPU executes a kernel with a two-level scheduling policy. The first level is allocating workgroups to a specific CU when there are enough free static resources for these workgroups. To support fast context switching between wavefronts or warps, the static resources of a workgroup cannot be preempted by other workgroups until the workgroup finishes its execution. A workgroup can be dispatched to a CU only if its required static resources are available. If one kernel cannot use up all static resources of a CU, other kernels have the opportunity to dispatch workgroups to the CU. That is to say, in order to schedule different kernels to a CU, we can reduce the number of workgroups of a kernel. If a kernel with a few workgroups has been dispatched to a GPU, the remaining static resources of the GPU can satisfy new workgroups from another kernel.

An SMK scheme shows how corunning kernels are mixed within a CU, i.e., the corresponding number of active wavefronts for each kernel in a CU. The second question is to find the best SMK scheme. However, no performance model for SMK is proposed to ensure that a GPU is shared efficiently. Recently proposed performance models on GPUs always focus on the single-kernel executions on GPUs. The slowdown estimation model for SPS proposed by Hu et. al. [10] shows the interference of the memory system for corunning kernels on a GPU. Their performance model cannot be directly applied to SMK. Also, their work is done on simulators, so that it is difficult to collect some parameters of their model on GPUs, e.g., the average number of banks of a memory request and the number of the extra last-level cache hit ratio of a memory request. The interferences of concurrent kernels within a CU are complicated to evaluate. The Wrap-Slicer [31] uses

performance vs. increasing CTA occupancy curves of each individual kernel to determine the best SMK scheme, while it fails to address the interferences between corunning kernels. Maestro [22] points out that the trace of instruction per cycle (IPC) is substantially different for corunning kernels from the case of running alone. A computing instruction will be postponed because the computing resource is being used by another kernel. The competition for dynamic resources among concurrent kernels within a CU has a great influence on performance. Work [14] shows that the bandwidth contentions of corunning kernels, including DRAM bandwidth and L2-to-L1 Network-on-Chip (NoC) bandwidth, play an important role in GPU multitasking. However, Maestro [22] and other work [14] both utilize architecture techniques to support dynamic resource management, which are impossible or generate great overheads by software solutions.

In this article, we propose a software solution for SMK on GPUs. With a probability model based on the multiplication rule of independent events, our performance model for SMK offers a new angle to understanding the interferences of an SMK scheme. The input of our model is the utilization of functional units and the bandwidth utilization. These parameters reflect how a kernel uses these dynamic resources and the memory system. We then propose a metric, *symbiotic factor*, to evaluate an SMK scheme. Also, to reduce the number of workgroups of a kernel, we apply two techniques: kernel slicing [33] and kernel stretching [21]. In this work, we analyze the advantages and disadvantages of these two techniques and integrate them to apply SMK on GPUs. We validate our model on 18 benchmarks from PolyBench [7] and Rodinia [5]. Compared to the optimized hardware-based concurrent kernel execution, whose kernel launching order brings fast execution time, the results of corunning kernel pairs show 11%, 18%, and 12% speedup on AMD R9 290X, RX 480, and Vega 64, respectively, on average. Compared to the Warped-Slicer [31], the results show 29%, 18%, and 51% speedup on AMD R9 290X, RX 480, and Vega 64, respectively, on average. Our contributions are:

- A lightweight performance model for SMK. Building on a probability model of independent events, our model estimates the effects of the resource contentions and only uses a few parameters that can be collected by profiling tools on GPUs.
- We propose a metric, *symbiotic factor* (SF), to evaluate the performance of an SMK scheme. This metric can guide the scheduler to choose the best SMK scheme and exploit effective resource sharing. We also show how to extend the SF to SF_{HS} and SF_{WS} according to the harmonic speedup and weighted speedup.
- We investigate the overhead of *kernel slicing* and *kernel stretching*. We observe that only using one of these two techniques will entail huge overhead for SMK schemes. Then, we propose a solution to implement SMK schemes on GPUs by adopting a hybrid technique.

The remainder of the article is organized as follows. Section 2 introduces the background of the concurrent kernel execution and our motivation and challenges. In Section 3, we present the details of our performance model. Section 4 elaborates our implementation of SMK schemes on GPUs. Section 5 reports the results of the experiments. Section 6 reviews related work. Section 7 concludes the article.

2 BACKGROUND AND MOTIVATION

2.1 Concurrent Execution of Multiple GPGPU Kernels

In general, a GPU executes a kernel with a two-level scheduling policy. The first level is allocating workgroups to a specific CU when there are enough free static resources for these workgroups. To support fast context switching between wavefronts or warps, the static resources of a workgroup cannot be preempted by other workgroups. A workgroup can be dispatched to a CU only if its

Fig. 1. Utilization of static resources. SGPRs: scalar registers; VGPRs: vector registers.

required static resources are available. A workgroup is *active* when it has been dispatched to a CU and does not complete its execution. The second level is dispatching wavefronts within a CU to execute. Threads within the same wavefront must execute the same instruction on different data (SIMD) at a time. A CU always contains multiple SIMD units, and these units can execute different instructions from different wavefronts. A fast context switching is a fundamental execution feature of GPUs to hide memory latency. For example, when a wavefront is stalled by memory load/store operations, other wavefronts on that CU can do computing.

Modern GPUs provide basic multitasking mechanisms to run multiple kernels concurrently. For Nvidia GPUs, the GigaThread Engine, a global work distribution engine at the chip level, schedules workgroups to various CUs with a round-robin and LEFTOVER policy [21]. For AMD GPUs, according to the official documents [16], we assume that the scheduler engines, eight independent ACEs (Asynchronous Compute Engines), can dispatch independent kernels with the same priority in an equivalent way, which means workgroups from independent kernels are dispatched simultaneously.

Due to the inefficient resource sharing of the hardware-based scheduling policy, recent works [1, 10, 13, 17] propose spatially partitioned sharing (SPS) to solve the problem. It coexecutes different kernels on disjointed sets of CUs. However, SPS fails to coexecute heterogeneous workloads within a CU, so that we cannot benefit from kernels with low static and dynamic resource contentions. Prior works [21, 22, 29, 33] give us another opportunity to share a GPU with SMK and point out that SMK can improve the resource utilization more than SPS. However, none of them has proposed a performance model to evaluate the interferences for concurrent kernels on a CU level.

2.2 The Underutilization of GPU Resources

The issue of GPU underutilization is well known. Figure 1 shows the utilization of four static resources for running the single kernel on two generations of AMD GPUs. The utilization of static resources changes slightly on three generations of AMD GPUs, so we just show two GPUs here. Table 1 shows the basic information of our benchmarks. We describe the details of these benchmarks in Section 5. None of the benchmarks can fully utilize all three different dynamic resources. Each CU contains a fixed amount of static resources, such as threads, registers, and local memory. It is evident that while one of the static resources gets saturated, the others are underutilized. The utilization of threads is nearly 100% for most benchmarks, except HYB, LMD, LEG, and NNE, while the utilization of the other resources is low. HYB is limited by local memory, while LMD and LEG are limited by VGPRs. For NNE, because of its small workgroup size, though all four resources are underutilized, its active workgroup number is limited by the maximum workgroup number supported within a CU. Since the static resources within a CU are not changed much for three generations (see Table 7), the utilization of static resources is almost the same. Also, the static

Table 1. Basic Information of Benchmarks on AMD GPUs

Abbr.	Benchmark	Kernel	N_{WG}	WG_{size}	WG_{active}	
					RX 480	Vega
FD1	FDTD [7]	fdtd_kernel1	16384=(64,256)	(32,8)	10	10
FD3	FDTD [7]	fdtd_kernel3	16384=(64,256)	(32,8)	10	10
SYR	SYR2K [7]	syr2k_kernel	16384=(64,256)	(32,8)	10	10
CDI	cfld [5]	initialize_variables	1212=(1212,1)	(256)	10	10
CDT	cfld [5]	time_step	1212=(1212,1)	(256)	10	10
GAU	gaussian [5]	Fan2	495616=(704,704)	(16,16)	10	10
HOT	hotspot [5]	hotspot	1849=(43,43)	(16,16)	10	10
HYB	hybridsort [5]	bucketsort	24415=(24415)	(32,1,1)	16	16
KMS	kmeans [5]	kmeans_swap	1930=(1930)	(256)	10	10
KMC	kmeans [5]	kmeans_kernel_c	1930=(1930)	(256)	10	10
LMD	lavaMD [5]	kernel_gpu_openc1	1000=(1000)	(128)	8	8
LEG	leukocyte [5]	GICOV_kernel	408=(408)	(256)	8	4
LED	leukocyte [5]	dilate_kernel	548=(548)	(256)	10	10
NNE	nn [5]	NearestNeighbor	16384=(16384)	(64)	16	16
PFK	particalfilter [5]	particle_single.cl/ likelihood_kernel	1563=(1563)	(256)	10	10
PFF	particalfilter [5]	particle_single.cl/ find_index_kernel	1563=(1563)	(256)	10	10
PAT	pathfinder [5]	dynproc_kernel	256000=(256000)	(256)	10	10
SRS	srad [5]	srad_kernel	899=(899)	(256)	10	10

N_{WG} : the number of workgroups. WG_{size} : the size of workgroup. WG_{active} : the maximum number of workgroups of a CU that can be active.

resources required by kernels are heterogeneous. Papers and articles [1, 10, 13, 17] propose SPS to improve the static resource utilization and ensure fairness among concurrently executed kernels. The main disadvantage of SPS is that it only allows different kernels to execute concurrently on disjointed sets of CUs, so that although it can lessen the underutilization of static resources on the whole GPU, for each set of CUs, such underutilization persists.

The underutilization of GPUs is reflected not only in static resources but also in dynamic resources such as SALUs (scalar units), VALUs (vector units), and memory units (Figure 2). The y-axis of Figure 2 shows the percentage of the corresponding dynamic resource that is busy during the execution time of a kernel on AMD GPUs. The MEMS bar is the ratio of the stall time to the busy time on memory units. The higher value on the MEMS bar means that the interconnection of the GPU memory system is busy so that many memory operations are stalled. All data are collected by CodeXL [2]. Because the active workgroup number is limited by the maximum workgroup number supported within a CU, NNE shows underutilization of all dynamic resources. For some benchmarks (HOT, LED, PFF, and PAT), the utilization of SALUs exceeds 100%. Because the SALUs are shared by all SIMD units on a CU, the profiling tool counts some reused calculations twice or more. Even as one of the dynamic resources gets saturated, the others remain highly underutilized. The SALUs and VALUs are used less than 2% of the time for benchmark SYR, while the memory units are busy nearly all the time. The SALUs are used less than 11% of the time for benchmark LMD, whereas the VALUs are used more than 82% of the time. In addition, we can see that the differences in dynamic resource utilization among three GPUs are larger than those of static resource utilization. For example, the utilization of memory units for benchmark LMD is lower than 7% on

Fig. 2. Utilization of dynamic resources on three generations of AMD GPUs. SALU: scalar unit; VALU: vector unit; MEM: memory unit; MEMS: ratio of the stall time to the busy time on the memory unit; Bandwidth: utilization of memory bandwidth (compared to the theoretical bandwidth).

R9 290x and RX 480, while it is over 95% on Vega. Therefore, one fails to explore the differences between GPU architectures if only optimizing the sharing of static resources.

Recently, researchers have proposed SMK to improve the utilization through the modification of GPU architectures [22]. Because of the heterogeneity of the runtime behaviors of kernels, corunning kernels with low resource contentions can launch more threads on a CU and interleave instructions with different types. Additionally, the performance of SMK schemes is various, especially when the number of corunning kernels is large. In Section 5.2.2, our result shows that the average number of SMK schemes is about 36 for each kernel pair. Even after removing the SMK schemes whose static resources are underutilized, the number is over 6.5. In that experiment, the maximum variation of performance of each pair with different SMK schemes is 74%, 193%, and 92% on AMD R9 290x, RX 480, and Vega respectively (see Figure 12). The average is over 14% on three GPUs. The average variation in terms of harmonic speedup or weighted speedup reaches about 30% on all three GPUs. Therefore, it is necessary and significant to evaluate the interferences of SMK and guide the scheduler, to improve resource utilization. However, because of the extreme complexity of kernel runtime behaviors, there is a lack of performance models to extrapolate interferences of corunning kernels within a CU, and no software SMK solution is proposed.

2.3 Challenges

Our goal is to propose a software solution of SMK to explore the maximum efficiency of GPU sharing. This involves tackling at least three challenges.

- Efficiently scheduling a specified number of workgroups to a CU on top of the hardware scheduler is not easy. We should minimize the overhead of *kernel slicing* and *kernel stretching* techniques we adopted in order to get the benefit from an SMK scheme.
- Interferences of SMK are too complicated to quantify, and their impact on performance is hard to evaluate accurately for GPUs. Interference of a dynamic resource may delay the instruction execution using other dynamic resources because of the instruction dependency, while the delay can also be hidden because of the fast context-switching feature of the GPU.

Table 2. The Notions of a Kernel and SMK Schemes

Symbol	Definition
$N_{wg}(k)$	The number of workgroups of kernel K
$N_{wf_per_wg}(k)$	The number of wavefronts in a workgroup for kernel k
N_{simd}	The number of SIMD units in a GPU
(k, s)	A kernel whose number of wavefronts is $s \times N_{simd}$, i.e., the number of dispatched wavefronts to a SIMD unit is s
S	The SALU function unit
V	The VALU function unit
M	The memory function unit
M_{std}	The memory function unit is active but stalled

- Our performance model of SMK should be lightweight, which can help schedulers coschedule kernels with complementary resource utilization to a CU. We need to reveal the correlations among tens of available performance counters and select a few of them.

3 PERFORMANCE MODELING FOR SMK SCHEME

From the previous section, we know that a CU contains multiple SIMD units (four for AMD GPUs) and that each wavefront can only run on a specified SIMD unit once dispatched. Therefore, our SMK scheme is represented as a mixture of wavefronts of kernels on a SIMD unit. To clarify our model, we define some symbols for a kernel and an SMK scheme in Table 2.

Kernel: A kernel k contains N_{wg} workgroups, and the number of wavefronts of a workgroup of kernel k is $N_{wf_per_wg}(k)$.

(k, s): Let N_{simd} be the number of SIMD units in a GPU. Then, kernel k contains $s \times N_{simd}$ wavefronts. The GPU scheduler dispatches s wavefronts to a SIMD unit for kernel k .

An SMK Scheme: Given a set of n kernels $\{k_1, k_2, \dots, k_n\}$, an SMK scheme $\mathcal{SS} = \{(k_i, s_i) | i \in [1, n]\}$ represents the number of active wavefronts on each SIMD unit and is s_i for kernel k_i .

In this section, we first introduce how we evaluate conflicts for concurrently running kernels. Then, we get the delay time caused by these conflicts for each dynamic resource and investigate how these delay times affects the overall execution time. We find the boundary of the execution time by considering the latency hiding feature of the GPU architecture and then let the mean value to be the execution time. Finally, we propose a metric, *sybiotic factor*, to evaluate an SMK scheme.

3.1 Conflicts Evaluated Based on the Probability of Independent Events

We use two concurrently executed kernels as an example to illustrate our idea, and it can apply to multikernels easily. For two concurrent kernels, we first assume that these two kernels are ideally overlapped and then model the slowdown caused by conflicts of each dynamic resource. Within a CU, if two wavefronts from different independent kernels require the same dynamic resource, a conflict happens. If they use different dynamic resources, their execution is in parallel. Our performance model is based on the probability of independent events. Assume that for a period $T_{overlap}$, k_1 and k_2 corun in a CU and they overlap with each other ideally (no conflicts happen). The key idea of our performance model is that we first calculate the slowdown caused by contentions of each dynamic resource and then discuss how they affect the execution time based on the latency hiding characteristic of GPU architectures. Wavefronts from different corunning kernels are independent. Taking wavefronts from the same kernel as a whole, we assume that a kernel k occupying a dynamic resource R is an independent event and denote $p(k, R)$ to be the probability of this event.

Table 3. Conflicts on Memory System with Two Concurrent Kernels

$k_1 \backslash k_2$	Occupying	Stalled	Idle
Occupying	Bandwidth sharing	Conflicts	No conflicts
Stalled	Conflicts	Conflicts	No conflicts
Idle	No conflicts	No conflicts	No conflicts

Then, the probability that a conflict happens on the dynamic resource R between these two kernels is calculated by Equation (1), denoted as $p(R)$. How to get the slowdown caused by conflicts of each dynamic resource and how these slowdowns affect the parallel execution time are introduced in the next subsection.

$$p(R) = p(k_1, R) * p(k_2, R), \quad R \in \{S, V, M\}. \quad (1)$$

3.2 Slowdown Caused by Conflicts

When conflicts happen, the GPU will delay the execution of the blocked instruction. Since the execution time of the blocked instruction changes little, we can simply add the busy time of each functional unit as the busy time of the functional unit for corunning kernels (see Equation (2)).

$$T(R) = T(k_1, R) + T(k_2, R), \quad R \in \{S, V, M\}. \quad (2)$$

Here, $T(k, R)$ is the time that kernel k uses the dynamic resource R and $T(R)$ is the time that the dynamic resource R is busy.

For computing units SALU and VALU, when conflicts happen, a newly issued instruction has to wait for the previous one to finish execution. Then, the slowdown $T_{slowdown}$ of computing units during the overlap period $T_{overlap}$ is calculated by Equation (3).

$$T_{slowdown}(R) = p(R) \cdot T_{overlap}, \quad R \in \{S, V\}. \quad (3)$$

The states of memory units can be idle, occupying (active but not stalled), and stalled. Table 3 illustrates what happens on the memory system when two kernels execute simultaneously. We first consider when a kernel is in the stalled state, and $p(k, M_{std})$ is the probability that a kernel is stalled. In this case, the interconnection of a GPU memory system is busy in fetching or storing data, and the memory pipeline may be stalled by cache-miss-related resource saturation [6, 23]. This will impose the memory operations of all other delayed corunning kernels. The probability of such conflicts $p(M_{std})$ is calculated by Equation (4).

$$p(M_{std}) = p(k_1, M_{std}) \cdot p(k_2, M) + p(k_2, M_{std}) \cdot p(k_1, M) - p(k_1, M_{std}) \cdot p(k_2, M_{std}). \quad (4)$$

If the states of memory units for both kernels are occupying, two memory operations may share the memory bandwidth, and no conflict happens. The experiments of Lin et al. [15] show that as the fetch and write size increases, the utilization of bandwidth saturates around 60% of the peak bandwidth. We estimate the bandwidth efficiency by following the experiments in Mei et al. [18], and the results are shown in Figure 3. We denote the practical peak bandwidth as B_{peak} . Therefore, if the sum of the bandwidth of these two kernels is higher than B_{peak} , the bandwidth used by corunning kernels does not exceed B_{peak} , and the slowdown is calculated by Equation (5).

$$T_{slowdown}(B) = \frac{(B(k_1) + B(k_2) - B_{peak}) \cdot T_{overlap}}{B_{peak}}, \quad (5)$$

where $B(k)$ is the bandwidth of kernel k , and B_{peak} is the practical peak bandwidth of the GPU. In conclusion, the slowdown of memory units during the overlap period $T_{overlap}$ is calculated

Fig. 3. The maximum utilization of bandwidth on AMD GPUs.

by Equation (6). When the total bandwidth does not exceed B_{peak} , the slowdown caused on the memory system is $p(M_{std}) \cdot T_{overlap}$; otherwise, it is the maximum slowdown between $p(M_{std}) \cdot T_{overlap}$ and $T_{slowdown}(B)$.

$$T_{slowdown}(M) = \begin{cases} p(M_{std}) \cdot T_{overlap} & \text{if } B(k_1) + B(k_2) \leq B_{peak}, \\ \max(T_{slowdown}(B), p(M_{std}) \cdot T_{overlap}) & \text{others.} \end{cases} \quad (6)$$

How these slowdowns affect the execution time of concurrent kernels is complicated. First, a stall can delay the execution of other instructions on other dynamic resources because of the data dependency between them. A memory operation may be dependent on a scalar instruction to calculate the basic address. Second, a stall can be hidden because of the fast context switching of a GPU. For example, for a memory-intensive kernel, stalls on computing resources can be totally hidden by memory operations. Therefore, by assuming that all conflicts cannot be hidden and cause slowdowns of the execution time, we get the upper boundary of execution time during the overlap period $T_{overlap}$, which is the sum of slowdowns of SALU, VALU, and the memory system (Equation (7)).

$$T_{co} \leq T_{comax} = \sum_{R \in \{S, V, M\}} T_{slowdown}(R) + T_{overlap}. \quad (7)$$

It is obvious that the execution time cannot be smaller than the busy time of each functional unit. Additionally, a stall on a resource R causes delay of the finished execution time of R for each corunning kernel. Then, we can get the kernel k_{max} whose delay leads to the maximum delay of the finished execution time on resource R_{max} by Equation (8). Such a delay can hardly be hidden.

$$(k_{max}, R_{max}) = \argmax\{T(k_i, R) + T_{slowdown}(R)\}, \quad R \in \{S, V, M\}, i \in \{1, 2\}. \quad (8)$$

Therefore, we have the lower boundary by Equation (9).

$$T_{co} \geq T_{comin} = \max\{T(S), T(V), T(M), T_{overlap} + T_{slowdown}(R_{max})\}, \quad (9)$$

where $T(S)$, $T(V)$, and $T(M)$ is the busy time by Equation (2). Here, to simplify the calculation, we use the mean value of these two boundaries as the execution time of the overlap period, denoted as T_{co} in Equation (10).

$$T_{co} = \frac{1}{2} \cdot (T_{comin} + T_{comax}). \quad (10)$$

Table 4. Symbols of Our Performance Model for AMD GPUs

Symbol	Relationship of PCs	Definition
$T(k)$	$GPUPTime$	The execution time of kernel k
$p(k, S)$	$SALUBusy(\%)/100$	The percentage SALU is occupied by kernel k
$p(k, V)$	$VALUBusy(\%)/100$	The percentage VALU is occupied by kernel k
$p(k, M)$	$MemoryUnitBusy(\%)/100$	The percentage the memory unit is occupied by kernel k , including the stall time
$p(k, M_{std})$	$MemUnitStalled(\%)/100$	The percentage the memory unit is stalled by kernel k
$D(k, F)$	$FetchSize$	The total kilobytes fetched from DRAM by kernel k
$D(k, W)$	$WriteSize$	The total kilobytes written to DRAM by kernel k
$T(k, R)$	$T(k, R) = p(k, R) \cdot T(k)$	The time dynamic resource R is occupied by kernel k
Symbol	Definition	
$T_{overlap}$	The overlap period by assuming no conflicts happen among concurrent kernels	
$T_{slowdown}(R)$	The slowdown caused by conflicts on a dynamic resource R	
T_{co}	The estimated execution time of $T_{overlap}$ by our model	
$T_{alone}(k)$	The execution time when kernel k runs alone on the GPU	
sl	The slicing factor that shows the number of kernels by <i>kernel slicing</i>	
st	The stretching factor that shows how many workgroups are stretched together by <i>kernel stretching</i>	

Note: PC means “performance counter.” For a transformed kernel (k, s) , k in all the above notions are expressed as (k, s) .

Finally, we get the execution time of these two corunning kernels T by Equation (11).

$$T = T_{alone}(k_1) + T_{alone}(k_2) - 2 \cdot T_{overlap} + T_{co}, \quad (11)$$

where $T_{alone}(k_i)$ is the execution time when running k_i alone.

3.3 Characterizing Runtime Features of a Kernel

It is hard to characterize the runtime behaviors of kernels. We define the probability that a kernel k occupies a functional unit (a dynamic resource) as the average percentage of kernel execution time that the functional unit is active. Therefore, from the idea of our model, the execution time and the probabilities that dynamic resources are occupied (e.g., *VALUBusy* for AMD GPUs and **_utilization* for Nvidia GPUs) are used in our model (listed in the first four rows in Table 4). Table 4 shows all parameters we selected from tens of performance counters extracted by a profiling tool, CodeXL [2] for AMD GPUs, and how they are used in our model. For an AMD GPU, there are three primary dynamic resources: VALU, SALU, and memory unit. *VALUBusy*, *SALUBusy*, and *MemoryUnitBusy* are the probabilities that a kernel k occupies these three dynamic resources, respectively. *MemUnitStalled* is the probability that the memory unit is stalled for a kernel k . To calculate the bandwidth, we also get the data size fetched from and stored to the DRAM of GPU (*FetchSize* and *WriteSize*). Other performance counters are not selected, primarily because, first, some of them (e.g., the efficiency of the SIMD unit) change slightly when kernels are executed simultaneously, and, second, some (e.g., cache hit rate) are correlated with others, which is too complicated to model.

3.4 Selecting the Best SMK Scheme

For an SMK scheme $\mathcal{SS} = \{(k_i, s_i)\}$, the execution time of a kernel k_i , T_{k_i} , is approximated by Equation (12).

$$T_{k_i} = T(k_i, s_i) \cdot \frac{N_{wf}(k_i)}{N_{simd} \cdot s_i}, \quad (12)$$

where $N_{wf}(k_i)$ is the number of wavefronts of k_i , and $N_{simd} \cdot s_i$ is the number of wavefronts of (k_i, s_i) . If a kernel k_i runs on a GPU alone, its SMK scheme is $\mathcal{SS} = \{(k_i, s_{up_i})\}$, where s_{up_i} is the maximum number of wavefronts that can be active on a SIMD unit for kernel k_i .

We use *sympiotic factor*(SF) to express the extent of performance gain of an SMK scheme. For an SMK scheme $\mathcal{SS} = \{(k_1, s_1), (k_2, s_2)\}$, the overlap period $T_{overlap}$ is calculated as follows according to Equation (12).

$$T_{overlap} = \min \left\{ T(k_i, s_i) \cdot \frac{N_{wf}(k_i)}{N_{simd} \cdot s_i} \right\}, i = 1, 2. \quad (13)$$

Then, by applying our performance model, we can get the execution time T_{co} for the overlap period by Equation (10).

Assume $T_{k_1} \geq T_{k_2}$. After the concurrent execution, the number of wavefronts remaining for k_1 is denoted as $N_{wf_remained}(k_1)$, as shown in Equation (14).

$$N_{wf_remained}(k_1) = N_{wf}(k_1) - s_1 \cdot N_{simd} \cdot T_{overlap} / T(k_1, s_1). \quad (14)$$

Then, k_1 can use the entire GPU with $\mathcal{SS} = \{(k_1, s_{up_1})\}$. Therefore, according to Equation (12), the execution time of remained wavefronts $T_{k_1_remained}$ is approximated as follows:

$$T_{k_1_remained} = T(k_1, s_{up_1}) \cdot \frac{N_{wf_remained}(k_1)}{N_{simd} \cdot s_{up_1}}. \quad (15)$$

Our *sympiotic factor*(SF) is defined by the speedup of our predicting execution time over the sum of execution time alone.

$$SF = \frac{T_{alone}(k_1) + T_{alone}(k_2)}{T_{co} + T_{k_1_remained}} \quad (16)$$

$$T_{alone}(k_i) = T(k_i, s_{up_i}) \cdot \frac{N_{wf}(k_i)}{N_{simd} \cdot s_{up_i}}. \quad (17)$$

We can find the best SMK scheme by estimating the highest SF among all possible SMK schemes.

4 IMPLEMENTATION OF SMK SCHEMES ON REAL GPUS

The SMK scheme can be implemented on GPUs by reducing the number of workgroups of a kernel according to the SMK scheme. As we know, if the total usage of static resources by multiple kernels is under hardware constraints, these kernels can run within a CU/SM concurrently. Luckily, we can reduce the number of workgroups of a kernel, because the execution order of workgroups does not affect the correctness of GPGPU programs and can preserve OpenCL or CUDA semantics: (1) threads in different workgroups cannot be synchronized and the programming model of GPGPU computing, like OpenCL and CUDA, only provides for synchronization of threads in the same work-group, and (2) the execution order of workgroups is random. In this article, we adopt two techniques, *kernel slicing* and *kernel stretching*, inspired by [21, 33] to reduce the number of workgroups of a kernel. Although these two techniques have been proposed for years, according to our survey, neither of them has been used so far to implement an SMK scheme. In this section,

Fig. 4. An example of kernel slicing and kernel stretching with slicing factor $sl = 2$ and stretching factor $st = 4$.

we first investigate the advantages and disadvantages of adopting these two techniques to implement SMK schemes and then introduce how to extend our performance model to facilitate both techniques.

4.1 Implementation of Kernel Slicing and Kernel Stretching

First proposed by Zhong and He [33], *kernel slicing* cuts a kernel into small kernels at the granularity of workgroup. We define slicing factor sl as the number of small kernels after kernel slicing. For small kernels, we need to specify an offset of workgroup ID to ensure the correctness of the kernel and to preserve OpenCL or CUDA semantics. Different from the ptx level they have done, we implement kernel slicing at the source-code level.

Pai et al. propose *kernel stretching* and call the changed kernel an *elastic kernel* [21]. In this article, we use the term *kernel stretching* to express our implementation of this technique. We implement *kernel stretching* by adding a loop to the body of a kernel code. Thus, we can merge workgroups from the same kernel into one big workgroup. For each iteration, we assign a new workgroup index to the workgroup. By stretching st workgroups into a workgroup of kernel k , we get a new kernel with a reduced number of workgroups, and we denote st as stretching factor.

We reshape kernels using both of these techniques. Figure 4 shows an example that transforms a kernel with 24 workgroups into two kernels with 3 workgroups with kernel slicing factor $sl = 2$ and kernel stretching factor $st = 4$. We do not change the structure of a workgroup, to preserve program semantics. First, we flatten all workgroups to x dimension so that the workgroup number in y-dim and z-dim is 1. Then, we slice the kernel into small kernels by setting the *global_work_offset* in *clEnqueueNDRangeKernel()* OpenCL function (this function submits a kernel to execute on the GPU). Finally, we add a loop in the kernel body to group workloads of several threads to a thread (line 3 in Listings 1, 2, and 3). To ensure the program correctness, we add a synchronization at the end of each iteration (line 1 in Listing 4), so that the new workload will only execute when the original workgroup finishes its execution.

In the kernel code, we need to calculate the original work-item index. Table 5 lists all built-in work-item functions that need to be replaced and Table 6 shows all parameters added to the kernel function. Variable *_id0* records the global id in x-dim for the flattened kernel (line 2 in Listings 1, 2, and 3 and line 2 in Listing 4). The global id for each dimension is calculated at the beginning of each iteration (lines 4–7 in Listing 1, lines 4 and 5 in Listing 2, and line 4 in Listing 3).

Table 5. Replacing the Built-in Work-Item Functions

Built-in Work-Item Function	Replaced Notion
<code>get_global_id(#)</code>	<code>_id#</code>
<code>get_group_id(#)</code>	<code>_id#/get_local_size(#)</code>
<code>get_num_groups(0)</code>	<code>_gs0/get_local_size(0)</code>
<code>get_num_groups(1)</code>	<code>_num1</code>
<code>get_num_groups(2)</code>	<code>_num2</code>
<code>get_global_size(0)</code>	<code>_gs0</code>
<code>get_global_size(1)</code>	<code>_num1*get_local_size(1)</code>
<code>get_global_size(2)</code>	<code>_num2*get_local_size(2)</code>

“#” represents the dimension of a kernel. It can be 0, 1, or 2.

Table 6. Parameters Added to a Kernel Function

Parameters	Description	Kernel Dims
<code>_gs0</code>	the global_work_size in x dimension before transformation	0, 1, 2
<code>_num1</code>	the workgroup number in y dimension before transformation	0, 1
<code>_num2</code>	the workgroup number in z dimension before transformation	0
<code>_strNum</code>	the iteration number for each thread before transformation	0, 1, 2

Listings 1, 2, and 3 are added to the front of a kernel body for 3D, 2D, and 1D kernels, respectively. Listing 4 is added to the end of a kernel body.

Because the number of workgroups of each dimension is changed after *kernel slicing* and *kernel stretching*, we add three parameters, `_gs0`, `_num1`, and `_num2`, to get the original numbers of workgroups of all dimensions. `_strNum` is passed to record the stretching factor *sl* of a kernel (line 3 in Listings 1, 2, and 3).

```

1  int _idb, _id0, _id1, _id2;
2  _idb = get_global_id(0);
3  for(int _index = 0; _index < _strNum; _index++){
4      _id2 = _idb / (_gs0 * _num1);
5      _id1 = (_idb - _id2 * _gs0 * _num1) / _gs0 * get_local_size(1) + get_local_id(1);
6      _id2 = _id2 * get_local_size(2) + get_local_id(2);
7      _id0 = _idb % _gs0;

```

Listing 1. Code added in front of kernel body for kernel with three dimensions.

```

1  int _idb, _id0, _id1;
2  _idb = get_global_id(0);
3  for(int index = 0; index < _strNum; index++){
4      _id1 = _idb / _gs0 * get_local_size(1) + get_local_id(1);
5      _id0 = _idb % _gs0;

```

Listing 2. Code added in front of kernel body for kernel with two dimensions.

4.2 Performance Analysis of Kernel Slicing and Kernel Stretching

The utilization of registers changes slightly after applying *kernel slicing* and *kernel stretching*, as shown in Figure 5. The usage of SGPRs grows the most drastically, because it needs more SGPRs to deal with the added loop, such as the basic address calculation. The main disadvantage of *kernel*

Fig. 5. The change of static resources by enabling kernel slicing and kernel stretching on AMD GPUs. Positive number represents that the requirement of the resource is increased, while negative number represents that the requirement is decreased.

```

1  int _idb, _id0;
2  _idb = get_global_id(0);
3  for(int index = 0; index < _strNum; index++){
4      _id0 = _idb % _gs0;

```

Listing 3. Code added in front of kernel body for kernel with one dimension.

```

1  barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
2  _idb += get_global_size(0);
3  }

```

Listing 4. Code added at the end of kernel body.

slicing is the overhead of kernel launching for small slices. The time gaps T_{gap} between two successive slices can accumulate to a considerable level as the number of slices increases. Since we need to reduce the number of workgroups to be small enough to implement the SMK scheme, the sum of T_{gap} cannot be ignored. Another effect of *kernel slicing* is that it may change the memory access patterns of a kernel. It is hard and complicated to precisely evaluate and model this. Although *kernel stretching* does not involve kernel launching overhead T_{gap} , it still involves other types of overhead. As the execution time of a wavefront grows too large, the imbalance of the finish time between CUs gets worse. As a result, the response time of a kernel is stretched.

Given an optimized SMK scheme $\mathcal{SS} = \{(k_i, s_i) | i = 1, 2, \dots, n\}$, we need to decide the proper slicing factor sl_i and stretching factor st_i for kernel k_i . In order to maximize the benefits of concurrent execution, we try to maximize the $T_{overlap}$ and launch kernel with a decreasing order, which is sorted by the kernel execution time of Equation (12). Then, we get the overlap period $T_{overlap}$ as follows:

$$T_{overlap} = \min\{T_{k_i} + (i - 1) \cdot T_{head}\}, \quad T_{k_i} > T_{k_{i+1}}. \quad (18)$$

The stretching factor st_i for kernel k_i is calculated by Equation (19). If there are remaining wavefronts, where $st_i \cdot s_i \cdot N_{simd} < N_{wf}(k_i)$, the slicing factor sl_i is set to 2; otherwise, $sl_i = 1$.

$$st_i = \max\{\lfloor (T_{overlap} - (i - 1) \cdot T_{head}) / T(k_i, s_i) \rfloor, 1\}. \quad (19)$$

5 EXPERIMENTAL EVALUATION

5.1 Experiment Setup and Benchmark Selection

We carry out our experiments on Windows 10 with AMD R9 290x, RX 480, and Vega 64 (see Table 7). We studied a total of 83 GPU kernels from the PolyBench [7] and Rodinia [5] benchmark

Table 7. Hardware Characteristics of AMD GPUs

Specification Items	R9 290x	RX 480	Vega 64
No. of compute units	44	36	64
Local memory (KB)	64	64	64
SIMDs per CU	4xSIMD-16	4xSIMD-16	4xSIMD-32
Vector GPR/CU (KB)	4x64	4x64	4x64
Scalar GPR/CU (KB)	4	8	4
Max workgroup size	256	256	256
Theoretical bandwidth (GB/s)	320	256	483.8
Practical bandwidth (GB/s)	280.69	181.49	371.38

suites. We rank all these kernels with the utilization of each dynamic resource on AMD RX 480 and select the top four kernels, as shown in Figure 2. Finally, we get 18 benchmarks, and Table 1 provides the information of our benchmarks. We use the same input data to generate profiling data and carry out our experiments.

Metrics. We use three metrics in this article: speedup, harmonic speedup, and weighted speedup. The speedup, *speedup*, shows the performance improvement (Equation (20)). Harmonic speedup *HS* [10, 14] is a balanced metric for both system throughput and fairness (Equation (21)), and weighted speedup *WS* [6, 14] is a metric for system throughput (Equation (22)). For these three metrics, the higher one is the better one.

$$speedup = \frac{\sum_i T_{alone}(k_i)}{T_{cur}}, \quad (20)$$

$$HS = \frac{N}{\sum_i \frac{T_{share}(k_i)}{T_{alone}(k_i)}}, \quad (21)$$

$$WS = \sum_i \frac{T_{alone}(k_i)}{T_{share}(k_i)}, \quad (22)$$

where $T_{alone}(k_i)$ is the execution time when k_i runs alone, T_{cur} is the total execution time that all kernels corun together, and $T_{share}(k_i)$ is the execution time of k_i when corunning with other kernels.

5.2 Performance Model Validation

For OpenCL kernels, it is impossible to validate our performance model by collecting hardware performance counters for concurrently executed kernels on GPUs, because there is no profiling tool that supports this. OpenCL kernels run serialized when enabling hardware performance counterprofiling on current GPUs. Therefore, we validate our performance model using two methods: (1) collecting hardware performance counters of two sliced kernels and their merged kernel (from the same kernel) and (2) estimating the execution times of two reshaped kernels (from two different kernels).

5.2.1 Model Validation on Hardware Performance Counters. We carry out model validation experiments on AMD RX 480x. According to their most utilized functional unit (Table 8), the runtime characteristics of the benchmark are diverse. Three benchmarks are purely memory intensive, including SYR, KMS, and PFK. Benchmarks HOT and PAT are purely SALU intensive, while LMD and SRS are purely VALU intensive. The MEMS row represents the ratio of the stall time

Table 8. Runtime Characteristics of Benchmarks on AMD RX 480

kernel	FD1	FD3	SYR	CDI	CDT	GAU	HOT	HYB	KMS
SALU							✓		
VALU									
MEM	✓	✓	✓		✓			✓	✓
MEMS				✓	✓	✓		✓	
Bandwidth	✓	✓			✓				
kernel	KMC	LMD	LEG	LED	NNE	PFK	PFF	PAT	SRS
SALU				✓			✓	✓	
VALU		✓	✓						✓
MEM			✓	✓		✓	✓		
MEMS									
Bandwidth	✓				✓				

Table 9. Size of Fixed Sliced Kernel

$\lfloor s_{up}/2 \rfloor$	kernel
2	HYB, LMD, NNE
4	LEG
5	FD1, FD3, SYR, CDI, CDT, GAU, HOT, KMS, KMC, LED, PFK, PFF, PAT, SRS

to the busy time on the memory unit. Benchmarks CDI and GAU are purely MEMS intensive. Benchmarks KMC and NNE are purely bandwidth intensive. Others show more than one runtime characteristic.

We design experiments on two sliced kernels with a homogeneous workload (sliced from the same kernel) mainly for two reasons. First, we can only get hardware performance counters of OpenCL kernels for single-kernel execution. Second, by running the merged kernel of two sliced kernels, we can ensure that the two sliced kernels are executed on the GPU concurrently and exclude the kernel launching overhead. To make these two sliced kernels actually corun on the CU level, the workgroups of one of them should be smaller than the maximum active workgroups (column WG_{active} in Table 1). Let one sliced kernel be $\lfloor 0.5WG_{active} \rfloor$, denoted by $(k, \lfloor 0.5s_{up} \rfloor)$. Table 9 shows its size for each benchmark. Then, we vary the size of the other sliced kernel (k, s_i) where $s_i = 1, 2, \dots, (30 - 0.5s_{up})$. The merged kernel of these two kernels is $(k, \lfloor 0.5s_{up} + s_i \rfloor)$. According to our performance model, we can get the runtime characteristics of the merged kernel. The accuracy of our model is measured by the error between the estimated data and the actual data collected by CodeXL [2] for the merged kernel (Equation (23)).

$$Error = \frac{|Estimated - Actual|}{Actual}. \quad (23)$$

Evaluation results of the busy time of computing units (SALUs and VALUs) and memory units are shown in Figure 6, which is calculated by Equation (2). We estimated the time by Equation (2). The left x-axis of Figure 6 shows the s_i of the other slice. The y-axis shows the error of the estimated busy time for each functional unit. The right of Figure 6 shows the box plot of each benchmark. We can see a decrease in the error as the increase of s_i , especially when s_i is larger than s_{up} . That is mainly because the variation of the execution time of these two kernels is getting larger so that the concurrent kernel execution has fewer effects on the overall performance. Another reason is

Fig. 6. The error of estimated busy time for each functional unit.

that when s is small, it does not have enough wavefronts to support the latency hiding feature of the GPU architecture. Therefore, the mean value we used to calculate the execution time of the overlap period is not accurate. The average error of s in range (2, 3) degrades slightly the SALU busy time and VALU busy time, about 3%, while the error averages 16% for the memory unit busy time. The average error of all tests is 6.6%, 6.3%, and 8.9% for SALU busy time, VALU busy time, and memory unit busy time, respectively. The 95% confidence intervals for most benchmarks are below 20%, except CDI. From the figure, CDI performs the worst for all types of dynamic resources. It is because CDI is an array-initializing kernel whose data access operations are mostly memory store operations, while we use data-fetching operations to estimate the bandwidth of GPU memory.

The results of the estimated execution time are shown in Figure 7. The average error is 11.4%. The 95% confidence intervals for most benchmarks are below 20%, except CDI, HYB, KMC, and LED. The reason for CDI and HYB has been described above. The average error of s in range (2, 3) degrades to 16%. We also calculate our estimated probability of each functional unit for merged kernels by definitions in Table 4. Figure 8 shows the accuracy. From Figure 8, we can see that the error for CDI and HYB has decreased, while it has increased for KMC and LED, especially KMC. That means for these two benchmarks, the error of the estimated busy time of each functional units is not the major factor producing the larger error of estimated execution time. The reason is

Fig. 7. The accuracy of estimated execution time.

Fig. 8. The error of probability of functional units.

Fig. 9. Get all tests for each pair of kernels (k_1, k_2) . Fig. 10. Four cases of concurrent execution timeline of (k_1, s_1) and (k_2, s_2) . T_{start} represents the kernel start time.

that we use the mean value of the boundary as the execution time of the corunning period. The latency hiding method for these two benchmarks is not accurate enough.

5.2.2 Model Validation on Execution Time Estimation for Concurrent Kernels. We can get the timeline of each kernel for corunning OpenCL kernels. Therefore, the accuracy of our performance model on a heterogeneous workload is validated using the error of the estimated execution time and actual execution time. Figure 9 shows the procedure to get the SMK schemes we test. For each kernel pair k_1 and k_2 , we first generate all possible SMK schemes by the requirements of static resources (threads, registers, and local memory). Then, we prune the schemes so that the static resources required by a new workgroup are smaller than the remaining static resources on the GPU. This is to narrow the searching space and gain more performance benefits, which can be obtained by having more active wavefronts within a CU, because of the context-switching feature of a GPU architecture. For all 153 pairs, the total number of SMK schemes is 5,447, 5,601, and 5,477 on AMD R9 290x, RX 480, and Vega respectively. After pruning, the number of SMK schemes is 1,052, 1,009, and 996 on AMD R9 290x, RX 480, and Vega, respectively. For each pair, the mean value of SMK schemes after pruning is 6.9, 6.6, and 6.5 on AMD R9 290x, RX 480, and Vega, respectively. To run an SMK scheme $SS = \{(k_1, s_1), (k_2, s_2)\}$ on a GPU, we need to transform the size of kernel k_1 and kernel k_2 to s_1 and s_2 , respectively. We collect the execution time of (k_1, s_1) and (k_2, s_2) when they are executed alone on the GPU, denoted as $T_{alone}(k_1, s_1)$ and $T_{alone}(k_2, s_2)$.

When we concurrently execute multiple kernels on a GPU, the start time of kernels is different. Figure 10 shows all the cases of the execution timeline when (k_1, s_1) and (k_2, s_2) corun on a GPU. We denote the start time of a kernel as T_{start} and the end time of a kernel as T_{end} . To eliminate the differences, we subtract the T_{head} (in Figure 10) from the execution time of the kernel, which starts first as Equation (24) and Equation (25). The overlap period is calculated by Equation (26).

$$T'_{alone}(k_1, s_1) = T_{alone}(k_1, s_1) - T_{head} \quad \text{for } \textcircled{1}\textcircled{2}, \quad (24)$$

$$T'_{alone}(k_2, s_2) = T_{alone}(k_2, s_2) - T_{head} \quad \text{for } \textcircled{3}\textcircled{4}, \quad (25)$$

$$T_{overlap} = \min\{T'_{alone}(k_1, s_1), T'_{alone}(k_2, s_2)\}. \quad (26)$$

By applying our performance model, we then get our estimated execution time T_{est} by Equation (11). We also adjust the actual execution time T_{act} for corunning these two kernels by Equation (27).

$$T_{act} = \max\{T_{end}(k_1), T_{end}(k_2)\} - \min\{T_{start}(k_1), T_{start}(k_2)\}. \quad (27)$$

Fig. 11. The execution time error on AMD GPUs.

Fig. 12. The range of performance for schemes of each kernel pair in terms of three metrics. S: speedup; HS: harmonic speedup; WS: weighted speedup; Average: average range; Max: maximum range.

Figure 11 shows the histogram and probability of the error of the execution time estimated on three AMD GPUs. Some tests that cannot actually overlap together are removed, and the number is 51, 124, and 233 on AMD R9 290x, RX 480, and Vega, respectively. In about 90% of tests, the error is smaller than 20%.

We also collect the speedup by Equation (20), the harmonic speedup by Equation (21), and the weighted speedup by Equation (22). Then, we calculate the range of all schemes of each kernel pair by subtracting the lowest value from the highest value. Figure 12 shows the average and maximum of the range in terms of these metrics on three GPUs. The maximum value of the speedup range is 74%, 193%, and 92% on AMD R9 290x, RX 480, and Vega, respectively. The average of the speedup range is about 14% on the three GPUs. The average range in terms of harmonic speedup and weighted speedup is about 30% on all three GPUs.

5.3 SMK Scheduling Validation

5.3.1 SMK Scheduling Evaluation for Kernel Pairs. We evaluate our *SF* for all 153 pairs of benchmarks in Table 1. We compare our selected scheme with the original scheme of the hardware scheduler and the scheme proposed by the Warped-Slicer [31]. Since the kernel launching order can affect the execution time, we choose the execution time of optimized order for the hardware scheduler, and the speedup is shown as ORI points in Figure 13. In Figure 13, the WAS points show the speedup of the execution time with an SMK scheme by the Warped-Slicer (WAS) [31]; the MDL points show the speedup of the execution time with an optimized SMK scheme by our performance model. Here, if the SMK scheme proposed by MDL or WAS is the same as ORI, we use the default hardware scheduler, because ORI does not involve the overhead of kernel slicing and kernel stretching. Compared to the optimized hardware-based concurrent kernel execution (ORI), our model-based solution shows 11%, 18%, and 12% speedup on AMD R9 290X, RX 480, and Vega 64, respectively, on average. The results show that our method outperforms the

Fig. 13. The speedup compared to the sum of single-kernel execution times on three AMD GPUs. ORI: the execution time of concurrent kernels by the hardware scheduler; WAS: the execution time of concurrent kernels with an SMK scheme by the Warped-Slicer [31]; MDL: the execution time of concurrent kernels with an optimized SMK scheme by our performance model.

Warped-Slicer [31] (WAS) by 29%, 18%, and 51% in performance on AMD R9 290X, RX 480, and Vega 64, respectively, on average. By applying the optimized SMK scheme, our software solution can approximately overcome the overhead incurred by kernel slicing and kernel stretching. We compare *SF* with the scheme proposed by our method and the WAS. There are 15, 14, and 3 pairs

where our method loses to WAS on AMD R9 290X, RX 480, and Vega 64, respectively. The main reason our method outperforms WAS is that WAS fails to investigate the conflicts between concurrent kernels, especially the contentions of the GPU memory system. The performance vs. increasing CTA occupancy curves, which WAS uses to select an optimized SMK scheme, change dramatically when kernels corun together [22].

5.3.2 System Throughput and Fairness. Since our model can predict the execution time for each corunning kernel, we can change SF according to other metrics, such as harmonic speedup and weighted speedup. We denote SF_{HS} as the metric to select the SMK scheme of the highest estimated harmonic speedup, and SF_{WS} as the metric to select the SMK scheme with the highest estimated weighted speedup. Then, Equation (16) can be replaced by SF_{HS} and SF_{WS} according to Equation (21) and Equation (22), respectively.

$$SF_{HS} = \frac{2}{\frac{(T_{co} + T_{k1_remained})}{T_{alone}(k_1)} + \frac{T_{co}}{T_{alone}(k_2)}}, \quad (28)$$

$$SF_{WS} = \frac{T_{alone}(k_1)}{(T_{co} + T_{k1_remained})} + \frac{T_{alone}(k_2)}{T_{co}}. \quad (29)$$

Figure 14 shows the WS and HS compared to the hardware scheduler and the Warped-Slicer on AMD RX 480. On average, our method with SF_{WS} improves weighted speedup by 50% and 21.2% compared to the hardware scheduler (ORI) and the Warped-Slicer (WAS), respectively. Our method with SF_{WS} outperforms the hardware scheduler and the Warped-Slicer by 120% and 44%, respectively, in terms of harmonic speedup.

5.3.3 Evaluation on Corunning Three Kernels. Figure 15 shows our evaluation on corunning three kernels. The *sybiotic factor* (SF) is extended by Equation (16). Because the kernel combinations are too large, we select at least one VALU intensive benchmark (see Table 8) for each combination, 408 tests in total. Our method improves performance by 12% and 28% compared to the hardware scheduler (ORI) and the Warped-Slicer (WAS), respectively. The result also shows improvements in harmonic speedup and weighted speedup. Our method outperforms ORI and WAS by 26% and 79% in terms of harmonic speedup and 17% and 194% in terms of weighted speedup, respectively.

6 RELATED WORK

Performance Models for Single-Kernel Execution on a GPU. The MWP-CWP (memory warp parallelism and computation warp parallelism) model [9] is an analytical performance model to estimate the execution time of a GPGPU application. Later, GPUPerf [25], an enhanced version of the MWP-CWP model, is proposed to identify potential bottlenecks of GPGPU applications. Bagsorkhi et al. [4] have developed a compiler-based analytical model to predict the performance of GPU kernels. They predict execution time by building a workflow graph to abstractly interpret GPU kernels and use symbolic evaluation to analyze memory access. Kerr et al. [11] use a statistical data analysis methodology on over 20 metrics of CUDA applications to extract the performance metrics. Zhang and Owens [32] developed a throughput model to help programmers identify low-level bottlenecks, e.g., the instruction pipeline, for Nvidia GPUs. However, these models cannot be directly applied to estimate the performance of concurrent kernel execution. Most of them [4, 9, 11, 25] are a heavy burden for users, because they take a lot of low-level parameters from source code analysis (e.g., the total number of coalesced/uncoalesced memory access operations [9, 25]) or complex

Fig. 14. The weight speedup and harmonic speedup on RX 480. ORI_WAS and ORI_HS: the weighted speedup and the harmonic speedup from the hardware scheduler; MDL_WS and MDL_HS: the weighted speedup and the harmonic speedup from our method; WAS_WS and WAS_HS: the weighted speedup and the harmonic speedup from the Warped-Slicer [31].

evaluation methods (e.g., the construction of a workflow graph [4] and the principal component analysis [11] on over 20 parameters).

Performance Models for Concurrently Executed Kernels. Li et al. [12] propose an analysis model to schedule a combination of kernels that can fit into a GPU within an execution round and have complementary memory bandwidth usage together. However, their model only suits small kernels. Zhong and He [33] propose a performance model based on the Markov model for concurrent kernel

Fig. 15. The performance evaluation for corunning three kernels on AMD RX 480. ORI: the hardware scheduler; WAS: the Warped-Slicer [31]; MDL: our performance model.

execution. However, to reach the steady state, the time complexity of their model is intolerably high for a real-time scheduler. Furthermore, these two models [12, 33] cannot be applied to SPS and SMK. Hu et al. [10] developed an accurate slowdown estimation model for SPS, while we aim to build a model for SMK, because in general SMK is better than SPS for efficient GPU sharing. Furthermore, their model [10] needs a simulator to collect parameters, which cannot be applied to real GPU cards.

GPU Intra-SM Sharing on Hardware Approaches. The Simultaneous Multikernel [29] proposes a “Dominant Resource Fairness” metric to fairly partition the static resources for SMK. SMK also proposes a fairly dynamic resource allocation method via warp scheduling, which periodically assigns each kernel a time quota in the warp scheduler. The quota is based on profiling of the standalone execution of each individual kernel. The Warped-Slicer [31] utilizes performance vs. increasing CTA occupancy curves to select an optimized SMK scheme, which failed to address the interference between kernels. GPU Maestro [22] is a dynamic resource management framework that uses history performance counters to repartition resources among kernels by preempting techniques. Recently, contentions from the GPU memory system drew extensive attention. Work [6] shows performance improvements by reducing memory pipeline stalls. They balance the memory accesses of concurrent kernels and limit the number of in-flight memory instructions issued by each kernel. Work [28] proposes an efficient bandwidth management for concurrent kernels, which is based on the patterns of an application’s effective bandwidth changes with different thread-level parallelism. Lin et al. [14] address the interferences of DRAM bandwidth and L2-to-L1 Network-on-Chip (NoC) bandwidth for corunning kernels and propose a coordinated approach to find an optimized SMK scheme and bandwidth partition for simultaneously running kernels.

7 CONCLUSION

In this article, we focus on building a performance model for SMK and evaluate the interferences of kernels within a CU. Our performance model is based on the probability of independent events and uses the percentage of time that a kernel occupies the dynamic resource as the probability that the kernel requests the dynamic resource at runtime. Additionally, we define a metric, *symbiotic factor*, to evaluate the performance gain of an SMK scheme and which can guide a scheduler to choose the best SMK scheme. Finally, we present a software solution to apply an SMK scheme to real GPUs by using kernel slicing and kernel stretching. We estimate our model and our software implementation of SMK on 18 benchmarks from PolyBench [7] and Rodinia [5]. The result shows that the average error of our model is about 9% for functional unit busy time and 11% for execution time. Compared to the optimized hardware-based concurrent kernel execution (ORI) whose kernel launching order brings fast execution time, the results of corunning kernel pairs show 11%, 18%, and 12% speedup on AMD R9 290X, RX 480, and Vega 64, respectively, on average.

Compared to the Warped-Slicer [31] (WAS), the results show 29%, 18%, and 51% speedup on AMD R9 290X, RX 480 and Vega 64, respectively, on average. Experiments with our extended metric SF_{HS} show improvements of 50% and 21.2% in terms of harmonic speedup compared to ORI and WAS, respectively. Results by using the SF_{WS} metric outperform ORI and WAS by 120% and 44%, respectively, in terms of weighted speedup. We also show results on corunning three kernels with SF . Our method improves performance by 12% and 28% compared to ORI and WAS, respectively. In terms of harmonic speedup and weighted speedup, our method also has improvements of over 15% compared to ORI and WAS.

REFERENCES

- [1] P. Aguilera, K. Morrow, and N. S. Kim. 2014. Fair share: Allocation of GPU resources for both performance and fairness. In *2014 IEEE 32nd International Conference on Computer Design (ICCD'14)*. 440–447. DOI : <https://doi.org/10.1109/ICCD.2014.6974717>
- [2] AMD. [n.d.]. CodeXL. <http://gpuopen.com/compute-product/codexl/>.
- [3] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. 2008. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J. Comput. Phys.* 227, 10 (2008), 5342–5359. DOI : <https://doi.org/10.1016/j.jcp.2008.01.047>
- [4] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. 2010. An adaptive performance modeling tool for GPU architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 105–114. DOI : <https://doi.org/10.1145/1693453.1693470>
- [5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC'09)*. 44–54. DOI : <https://doi.org/10.1109/IISWC.2009.5306797>
- [6] H. Dai, Z. Lin, C. Li, C. Zhao, F. Wang, N. Zheng, and H. Zhou. 2018. Accelerate GPU concurrent kernel execution by mitigating memory pipeline stalls. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. 208–220. DOI : <https://doi.org/10.1109/HPCA.2018.00027>
- [7] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar'12)*. 1–10. DOI : <https://doi.org/10.1109/InPar.2012.6339595>
- [8] Khronos OpenCL Working Group et al. 2008. The Opencl specification. *Version 1*, 29 (2008), 8.
- [9] Sunpyo Hong and Hyesoon Kim. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News* 37, 3 (June 2009), 152–163. DOI : <https://doi.org/10.1145/1555815.1555775>
- [10] Q. Hu, J. Shu, J. Fan, and Y. Lu. 2016. Run-time performance estimation and fairness-oriented scheduling policy for concurrent GPGPU applications. In *2016 45th International Conference on Parallel Processing (ICPP'16)*. 57–66. DOI : <https://doi.org/10.1109/ICPP.2016.14>
- [11] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. 2010. Modeling GPU-CPU workloads and systems. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 31–42. DOI : <https://doi.org/10.1145/1735688.1735696>
- [12] Teng Li, Vikram K. Narayana, and Tarek El-Ghazawi. 2014. Symbiotic scheduling of concurrent GPU kernels for performance and energy optimizations. In *Proceedings of the 11th ACM Conference on Computing Frontiers*. ACM, Article 36, 10 pages. DOI : <https://doi.org/10.1145/2597917.2597925>
- [13] Y. Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, and D. Chen. 2015. Efficient GPU spatial-temporal multitasking. *IEEE Trans. Parallel Distrib. Syst.* 26, 3 (March 2015), 748–760. DOI : <https://doi.org/10.1109/TPDS.2014.2313342>
- [14] Zhen Lin, Hongwen Dai, Michael Mantor, and Huiyang Zhou. 2019. Coordinated CTA combination and bandwidth partitioning for GPU concurrent kernel execution. *ACM Trans. Archit. Code Optim.* 16, 3, Article 23 (June 2019), 27 pages. DOI : <https://doi.org/10.1145/3326124>
- [15] Zhen Lin, Michael Mantor, and Huiyang Zhou. 2018. GPU performance vs. thread-level parallelism: Scalability analysis and a novel way to improve TLP. *ACM Trans. Archit. Code Optim.* 15, 1, Article 15 (March 2018), 21 pages. DOI : <https://doi.org/10.1145/3177964>
- [16] Mike Mantor. 2012. AMD RadeonTM HD 7970 with graphics core next (GCN) architecture. In *2012 IEEE Hot Chips 24 Symposium (HCS'12)*. IEEE, 1–35.
- [17] Christos Margiolas and Michael F. P. O'Boyle. 2016. Portable and transparent software managed scheduling on accelerators for fair resource sharing. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, 82–93. DOI : <https://doi.org/10.1145/2854038.2854040>

- [18] X. Mei and X. Chu. 2017. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Trans. Parallel Distrib. Syst.* 28, 1 (Jan. 2017), 72–86. DOI : <https://doi.org/10.1109/TPDS.2016.2549523>
- [19] Nvidia. [n.d.]. CUDA C Programming Guide. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
- [20] Nvidia. [n.d.]. Tuning CUDA Applications for Kepler. <https://docs.nvidia.com/cuda/kepler-tuning-guide/index.html>.
- [21] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. 2013. Improving GPGPU concurrency with elastic kernels. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 407–418. DOI : <https://doi.org/10.1145/2451116.2451160>
- [22] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2017. Dynamic resource management for efficient utilization of multitasking GPUs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 527–540.
- [23] A. Sethia, D. A. Jamshidi, and S. Mahlke. 2015. Mascar: Speeding up GPU warps by reducing memory pitstops. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. 174–185. DOI : <https://doi.org/10.1109/HPCA.2015.7056031>
- [24] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating a file system with GPUs. *SIGARCH Comput. Arch. News* 41, 1 (March 2013), 485–498. DOI : <https://doi.org/10.1145/2490301.2451169>
- [25] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. 2012. A performance analysis framework for identifying potential benefits in GPGPU applications. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 11–22. DOI : <https://doi.org/10.1145/2145816.2145819>
- [26] Jeff A. Stuart and John D. Owens. 2011. Multi-GPU MapReduce on GPU clusters. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE Computer Society, 1068–1079. DOI : <https://doi.org/10.1109/IPDPS.2011.102>
- [27] Weibin Sun and Robert Ricci. 2013. Fast and flexible: Parallel packet processing with GPUs and click. In *Proceedings of the 9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. IEEE Press, 25–36.
- [28] H. Wang, F. Luo, M. Ibrahim, O. Kayiran, and A. Jog. 2018. Efficient and fair multi-programming in GPUs via effective bandwidth management. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. 247–258. DOI : <https://doi.org/10.1109/HPCA.2018.00030>
- [29] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. 2016. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. 358–369. DOI : <https://doi.org/10.1109/HPCA.2016.7446078>
- [30] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. 2012. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 107–118. DOI : <https://doi.org/10.1109/MICRO.2012.19>
- [31] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram. 2016. Warped-Slicer: Efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. 230–242. DOI : <https://doi.org/10.1109/ISCA.2016.29>
- [32] Yao Zhang and John D. Owens. 2011. A quantitative performance analysis model for GPU architectures. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE Computer Society, 382–393. <http://dl.acm.org/citation.cfm?id=2014698.2014875>
- [33] J. Zhong and B. He. 2014. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (June 2014), 1522–1532. DOI : <https://doi.org/10.1109/TPDS.2013.257>

Received July 2019; revised December 2019; accepted December 2019