SMGuard: A Flexible and fine-grained resource management framework for GPUs

Chao Yu, Yuebin Bai, *Member, IEEE,* Hailong Yang, *Member, IEEE,* Kun Cheng, *Student Member, IEEE,* Yuhao Gu, Zhongzhi Luan, *Member, IEEE,* and Depei Qian, *Member, IEEE*

Abstract—GPUs have been becoming an indispensable computing platform in data centers, and co-locating multiple applications on the same GPU is widely used to improve resource utilization. However, performance interference due to uncontrolled resource contention severely degrades the performance of co-locating applications and fails to deliver satisfactory user experience. In this paper, we present SMGuard, a software approach to flexibly manage the GPU resources usage of multiple applications under co-location. We also propose a capacity based GPU resource model CapSM, which provisions the GPU resources in a fine-grained granularity among co-locating applications. When co-locating latency-sensitive applications with batch applications, SMGuard can prevent batch applications from occupying resources without constraint using quota based mechanism, and guarantee the resources usage of latency-sensitive applications with reservation based mechanism. In addition, SMGuard supports dynamic resource adjustment through evicting the running thread blocks of batch applications to release the occupied resources and remapping the uncompleted thread blocks to the remaining resources, which avoids the relaunch of the preempted kernel. The SMGuard is a pure software solution that does not rely on special GPU architecture or programming model, which is easy to adopt on commodity GPUs in datacenters. Our evaluation shows that SMGuard improves the average performance of latency-sensitive applications by 9.8$\times$ when co-located with batch applications. In the meanwhile, the GPU utilization can be improved by 35% on average.

Index Terms—GPU, parallel computing, resource management, application co-location

1 Introduction

Since the beginning of this century, GPUs have been becoming an important and powerful processing device for general-purpose computing, especially for massively parallel application. To accelerate key businesses, GPUs are widely adopted in large-scale datacenters. To improve resource utilization, GPU is usually shared by co-locating multiple types of application, such as latency-sensitive applications (LS applications) and batch applications. However, to satisfy the stringent Quality-of-Service (QoS) of LS applications, the GPUs are often over-provisioned and under-utilized for most of the time [1]. There exists a natural contradiction among GPU sharing and guaranteeing the QoS target of LS applications.

The root cause of the performance degradation is the contention of co-locating applications on shared resources, which can seriously violate the QoS of LS applications [1], [2], [3], [4]. To mitigate the performance interference under co-location, an effective and flexible resource management method needs to be proposed to guarantee the QoS of LS application while improving GPU resource utilization. In the past decade, a large body of research [1], [3], [4], [5], [6], [7], [8], [9], [10] has been devoted to solve the co-location problem. In the industry, Multi-Process Server (MPS) [6], proposed by Nvidia, is the current de facto industrial standard to allow GPU kernels from different applications to be processed concurrently on the same GPU, and MPS has been widely adopted by Nvidia Kelper, Maxwell, Pascal and current cutting-edge Volta architecture [11]. However, MPS lacks efficient control over the GPU resources. If an application is capable of using up all resources, then there will be no resource available to the subsequent application and it will wait until the former application releases resources [5], [12]. The waiting time varies depending on the kernel length and the number of queued thread blocks, especially, when the subsequent application is a LS application and the kernel length of the former application is long or the number of queued thread blocks is large, then the QoS of the LS application can be seriously violated.

To avoid the arbitrary occupation of GPU resource in co-location, resource used by batch applications should be limited in order to reserve enough resources to LS applications. Thus, when a task from LS application comes, it can be served immediately. However, existing GPU architectures do not support resources partition. Previous works [13], [14], [15] proposed hardware extensions to enable GPU partition in a simulated manner, which cannot be applied in the real GPUs immediately. Existing software solutions rely on the state-of-the-art Filling & Retreating mechanism [16], [17], [18]. When a kernel needs to use the GPU, no matter how small the kernel is, it should first launch enough thread blocks to fill the whole GPU and then the streaming multiprocessors (SMs) can be retreated through retrieving SM IDs online. In the Filling stage, resource used by a task is not restricted. In addition, Filling & Retreating mechanism heavily relies on retrieving SM IDs online. Although SM ID can be easily obtained in CUDA on Nvidia GPUs, it is impossible for OpenCL or any other GPU programming model to get that because of the lack of corresponding

- *C. Yu, Y. Bai, H. Yang, K. Cheng, Y. Gu, Z. Luan and D. Qian are with Sino-German Joint Software Institute, the School of Computer Science, Beihang University, Beijing, China, 100191.*
  E-mail: {yuchao, byb, hailong.yang, chengkun, guyuhao, zhongzhi.luan, depeiq}@buaa.edu.cn

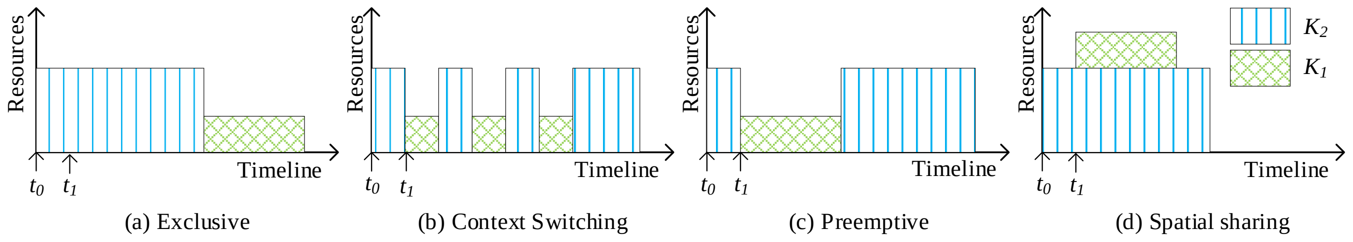


Fig. 1. GPU sharing modes. Assumed that kernel K1 is launched at t0 and kernel K2 is launched at t1, and they share a GPU. Fig.(a) is exclusive using mode, and K2 needs to wait for the completion of K1. Fig.(b) is context switching mode, and the round robin algorithm is used by the GPU scheduler to switch the co-locating kernels. Fig.(c) is preemptive mode, when K2 is launched, it preempts the execution of K1. Fig.(d) is spatial sharing mode, K1 and K2 are executed simultaneously as long as there are enough resources.

interface.

In this paper, we propose a flexible and fine-grained resource management framework *SMGuard*, a runtime system different from Filling & Retreating mechanism, to address above problems. SMGuard limits the resource used by a task flexibly using a capacity based GPU resource model *CapSM*, which does not rely on special hardware to retrieve SM IDs online. To guarantee the performance of LS applications, SMGuard reserves sufficient GPU resource for LS applications by limiting the available GPU resource to co-located batch applications, which eliminates the delay of LS applications (especially during the load burst) significantly. Different from the multi-stage of Filling & Retreating mechanism, with SMGuard, the resource used by an application is restricted directly before it starts running on GPU. In addition, when the reserved resource need to be increased, SMGuard supports runtime resource adjustment using similar resource eviction mechanism as EffiSha [17] and FLEP [18]. However, different from EffiSha and FLEP, the relaunch of the preempted tasks is eliminated in SMGuard by remapping the evicted thread blocks to remaining resource and continuing the execution of these thread blocks.

Specifically, this paper makes the following contributions:

1) **A flexible GPU resource model based on capacity slice *CapSM*** - We abstract the physical SM into small capacity slices, where capacity slices from different SMs that equal to a physical SM in total can form a CapSM. SMGuard benefits from CapSM that no special hardware or programming model is required to retrieve SM IDs online and CapSM can be easily applied to other programming models.

2) **Fine-grained resource reservation mechanism using quota** - Based on CapSM, we design resource quota and reservation mechanism that effectively limit the resource used by batch applications and reserve enough resource for LS applications.

3) **Runtime system for dynamic resource adjustment** - We propose a runtime mechanism that dynamically reduces resource occupied by a task and remaps uncompleted thread blocks onto remaining resource without the need of relaunching the preempted task.

We implement SMGuard framework using all above techniques and evaluate it with co-locations from various GPU applications. The experiment results show that SMGuard can improve the average performance of LS tasks by 9.8× when co-located with batch tasks. And the GPU utilization can also be improved by 35% on average. With the benefits of online task remapping, the average normalized latency of preempted tasks is reduced by 60% on average.

The rest of this paper is organized as follows: Section 2 presents the background and motivation of this paper. Section 3 presents the overview of SMGuard. Section 4 proposes the fine-grained SM model. The detailed resource management mechanism is described in Section 5. Section 6 provides performance and efficiency evaluation of SMGuard. Section 7 presents the analysis of the related work, while Section 8 is devoted to the conclusions.

2 Background and Motivation

In this section, we first briefly introduce the background of GPU computing model, and then provide the motivation of this paper. Although we target on CUDA in this paper, other GPU computing models are also applied to the contents of this paper.

2.1 GPU Parallel Computing Model

A modern GPU mainly consists of cores (known as CUDA core in NVIDIA), global memory, shared memory, register files, load/store units, warp schedulers and some other components. Among these components, there usually be thousands of cores, and every hundreds of cores are organized into a SM. SM is designed to support hundreds of threads to execute parallelly, so there could be thousands of threads in parallel execution in a GPU with multiple SMs, which makes the GPU especially suitable for massively parallel computing.

In GPU, code executed on device is called kernel. When a kernel is launched, thousands of threads are created by GPU. GPU uses a Single Instruction Multiple Thread (SIMT) architecture to manage and execute these threads, and every 32 of these threads are grouped into a unit, namely warp, which is the basic unit of warp scheduler; further, multiple warps are grouped into thread blocks; last, all the thread blocks are grouped into one grid, and the size of a grid is determined by launch configuration. After launched, each thread block will be dispatched into a SM by GPU hardware scheduler if there remained enough resources (eg. shared memory and register files) in that SM and all dispatched thread blocks become active thread blocks; however, if none of all the SMs have enough resources, all thread blocks not dispatched will be blocked until sufficient resources are released, which means that all subsequent kernel tasks will

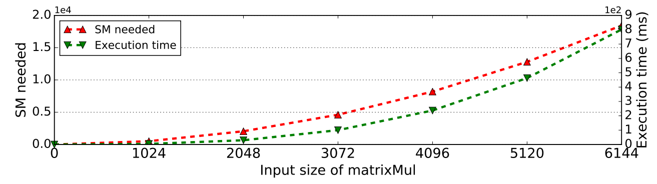


Fig. 2. The kernel execution time with the required number of SMs under different input sizes for matrix multiplication.

be blocked if former launched kernel can fully utilize all available resources.

2.2 GPU Sharing Mode

GPU sharing mode determines how and when a task will be performed. Fig. 1 shows four modes for GPU sharing. The first is *exclusive mode*; multiple tasks will use the GPU in a FIFO order, when one task is using the GPU, other tasks will be blocked until the running task releases the GPU. Under *context switching mode*, tasks from multiple applications can use the GPU concurrently through context switching and each task is assigned a serially scheduled time-slice on the whole GPU, thus the block time of tasks can be reduced greatly. However, in every switching, a large amount of context data should be stored/restored and the overhead of this process is not negligible, which leading to the underutilization of the GPU.

Some software solutions [19], [20], [21] are also proposed to support *preemptive mode*, where, higher priority tasks can preempt the running of lower priority tasks on GPU. Through the priority based preemption, the response time of high priority tasks can be guaranteed while the context switching times can also be reduced.

Through the above analysis, we can find that only one task can use the GPU at every moment even though a task cannot fully utilize the whole GPU. To make full use of the GPU resources, the *spatial sharing mode* was proposed. The MPS, for instance, can make multiple tasks use the GPU simultaneously if there are sufficient resources. Spatial sharing mode can increase GPU utilization and improve system performance compared to other three GPU sharing modes. The work of this paper is based on MPS.

2.3 Demands of GPU Resource Reservation

When a GPU kernel task is launched, many thread blocks are created. And the GPU hardware scheduler dispatches all thread blocks to SMs. If the number of thread blocks from the kernel task is less than the number of SMs on the GPU, each SM will execute at most one thread block [21]. On the other hand, each SM may execute multiple thread blocks. If the GPU is not fully utilized by the currently running task, the hardware scheduler will dispatch thread blocks from other tasks to SMs. Otherwise, all other tasks are blocked until all thread blocks of currently running task are dispatched, which leads to long waiting time for other tasks, especially in the multi-tenant data centers. The long waiting time causes severe performance degradation to LS tasks when co-located with batch tasks. In addition, the number of thread blocks launched by a kernel task varies

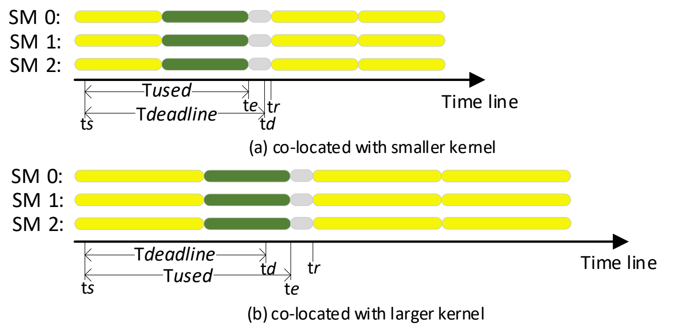


Fig. 3. Preemption based Scheduling. We assume the GPU has 3 SMs and only one active thread block can be hosted in each SM. The batch kernel(K1) represented by yellow bars has 9 thread blocks, thus, each SMs will assign 3 thread blocks. The LS kernel (K2) represented by green bars has 3 thread block. To run K2, K1 needs to be preempted. *ts*, *te* and *td* respectively represent the launching time, the end time and the deadline of K2; *tr* represents the relaunch point of K1; gray bars represent the free and launching time. Fig. (a) shows the scenario that K2 is co-located with a small kernel K1 and the QoS of K2 can be satisfied. Fig. (b) demonstrates that K2 is co-located with a large kernel K1 and the QoS is violated because of the long waiting time for the completion of batch task.

by input data size. Therefore, it is challenging to provide an effective resource management mechanism on GPU that can explicitly restrict the resource usage of batch task and reserve enough resource for LS task to satisfy the QoS target.

Fig. 2 shows the required number of SMs of matrix multiplication (*matrixMul*) with different input data sizes. In Fig. 2, the x-axis indicates the different input data sizes, and the y-axis shows the maximum number of SMs that *matrixMul* may occupy if having unlimited number of SMs on GPU. The maximum number of SMs is the ratio between the number of launched thread blocks and the maximum number of active thread blocks an SM can host. The number of launched thread blocks can be retrieved using nvprof [22] and the maximum number of active thread blocks can be obtained by the *cudaOccupancyMaxActiveBlocksPerMultiprocessor* API provided by CUDA runtime. As shown in Fig. 2, *matrixMul* is able to utilize thousands of SMs if there are unlimited SM resource. However, in realistic GPU devices, the number of SMs is usually quite limited. When running on Nvidia GTX 970 with 13 SMs, the execution time of *matrixMul* with different input data sizes is shown in Fig. 2 in green line. The curve of the execution time is similar to the curve of required number of SMs. Because the maximum number of active thread blocks of a kernel task is determined on a certain GPU, the other thread blocks of the task should be queued until the currently active thread blocks are completed. The larger the number of launched thread blocks the longer the total execution time.

The state-of-the-art software based preemption mechanism proposed in FLEP and EffiSha reduces the delay of long running kernel task by preempting the low-priority task if a high-priority task is ready to run. The granularity of preemption in the software based mechanism is block-task. A block-task is the set of work done by a thread block. However, the overhead of the block-task level preemption heavily depends on the length of block-task, which is illustrated in Fig. 3. In the software based mechanism, the preemption point is at the end of the block-task loop,

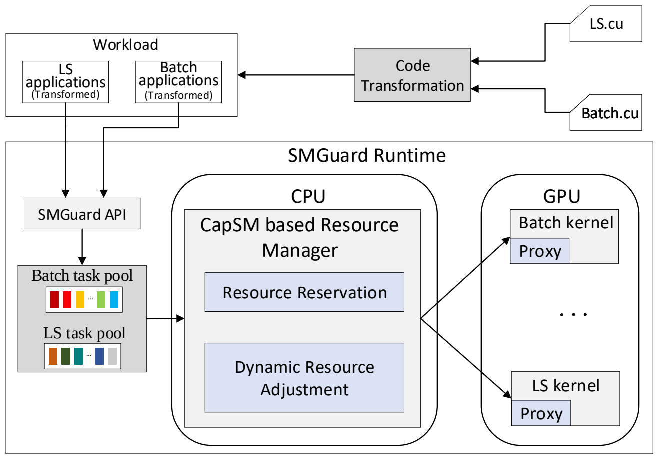


Fig. 4. Overview of SMGuard.

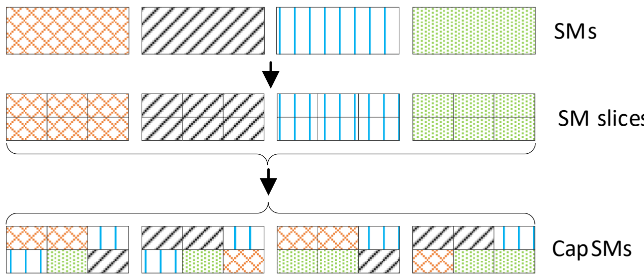


Fig. 5. Capacity based SM abstraction and CapSM model.

and only the block-task in each iteration is completed, the preemption point can be granted. Therefore, the preemption overhead can be as long as the length of the block-task in the worst case. In addition, the preempted block-tasks can only be executed by relaunching the whole kernel task. Thus, the pure preemption mechanism is not suitable when LS tasks are co-located with batch tasks that could block LS tasks for a long time even if the preemption is enabled. To strictly guarantee the QoS of LS tasks when co-located with batch tasks, in addition to the state-of-the-art preemption mechanism, it is also necessary to provide the capability of resource reservation that satisfies the resource requirement of LS tasks. In the meanwhile, to improve the GPU resource utilization while guaranteeing the QoS, the resource allocation should be adjusted dynamically during runtime between LS and batch applications.

3 Overview of SMGuard

The goal of SMGuard is to provide a flexible and fine-grained resource management that can thoroughly restrict the available resources to the co-located batch kernels and reserve sufficient resources for LS kernels to ensure that QoS is met.

Fig. 4 shows the overview of SMGuard. Due to the inaccessibility to the underlying GPU driver and CUDA runtime, it is difficult to add new features to the existing GPU, runtime or driver. Similar to existing work [17], [18], [20], we use source to source transformation compiler to automatically add supports of SMGuard to GPU applications. The kernel invocation API on the CPU side is transformed to intercept the invocation, and kernel launching configuration is also transformed to launch special number of thread blocks. Optionally, SMGuard also provides a set of APIs to programmers to control the kernel tasks to be issued. The APIs allow the programmers to push kernel tasks to one of the two task pools according their task type.

When a kernel task arrives, the *Resource Reservation* module in the *CapSM based Resource Manager* determines the GPU resource quota of that task according to its resource request and the current GPU usage. To achieve the resource quota and reservation mechanism flexibly, we propose a capacity based SM model *CapSM*. In addition, to reduce the resource quota of co-located batch tasks during runtime, the *Dynamic Resource Adjustment* module preempts batch tasks to release resources, which in turn increases available resources to LS task accordingly. For those block-tasks that are preempted by *Dynamic Resource Adjustment* module, the block-task remapping *Proxy* inserted into the kernel during source transformation remaps them to remaining resources, eliminating the unnecessary relaunch of preempted kernel task.

4 A Fine-grained SM Resource Model

In this section, we present the abstract SM model CapSM, which is the basis of SMGuard. Then we illustrate how to use CapSM to restrict the resource usage of kernel task. To demonstrate, we describe how to implement CapSM on CUDA kernel.

4.1 CapSM: a Capacity Based SM Model

According to the analyses of previous sections, it is necessary to manage the GPU resource usage in co-location scenarios of large scale cloud datacenters and strictly restrict the resources can be used for co-located batch tasks to minimize the shared resource competition, which can eliminate performance interference and guarantee the QoS target of the LS tasks. However, due to the restrictions of business secrets, we cannot have a detailed understanding of the task scheduling and execution in GPU, thus it is very hard to limit the GPU resources used by GPU kernels accurately and directly. The goal of CapSM is to explore a way to provide a flexible and fine-grained resource quota and reservation mechanism.

In SMGuard, we introduce the capacity concept to SM. The capacity of a SM in SMGuard is the abstraction of the SM resource. We abstract the physical SM into small capacity slices, and capacity slices from different SMs or the same SM that equal to a physical SM in total capacity make up a CapSM. Above process can be illustrated in Fig. 5. In other words, if the maximum number of active thread blocks of a kernel task that a SM can host is M, we consider the total resource capacity occupied by these M thread blocks is equivalent to the capacity of a SM, and each thread block can be seen to use $1/M$ capacity of a SM or utilize $1/M$ SM, and the capacity slice size can be set to $1/M$. Further, we can say every M thread blocks whether dispatched to the same SM or different SMs will utilize one SM in capacity. For simplicity, we call capacity slices, whose total capacity is equal to a physical SM, from

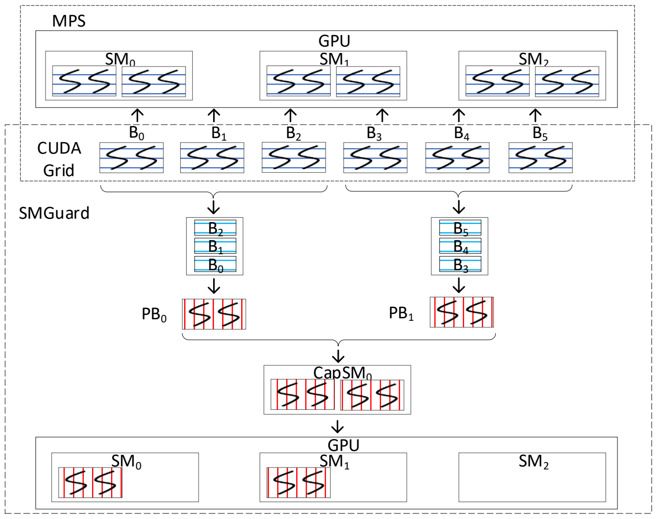
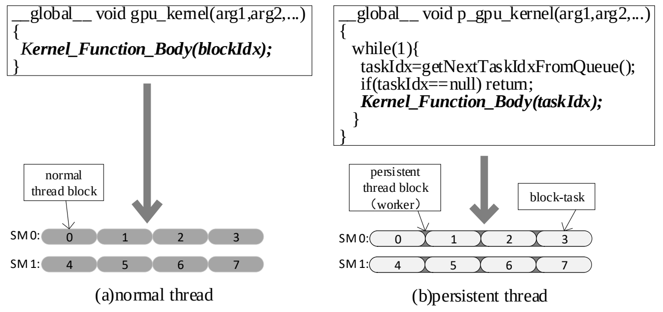


Fig. 6. CapSM based resource quota.



```
__global__ void gpu_kernel(arg1,arg2,...)
{
  Kernel_Function_Body(blockIdx);
}
```

```
__global__ void p_gpu_kernel(arg1,arg2,...)
{
  while(1){
    taskIdx=getNextTaskIdxFromQueue();
    if(taskIdx==null) return;
    Kernel_Function_Body(taskIdx);
  }
}
```

Fig. 7. The relationship between normal thread and persistent thread. Suppose a kernel task K has 8 thread blocks and each SM can only host one active thread block. There are total 2 SMs in a GPU. Fig. (a): in default model, each SM is dispatched 4 thread blocks and each of the 4 thread blocks is executed in turn; Fig. (b):in persistent model, each SM is dispatched 1 persistent thread block and each persistent thread block executes 4 assigned block-tasks in turn.

the same SM or different SMs a CapSM, and the amount of CapSM is equal to the total number of SMs in a GPU. Because the maximum number of active thread blocks of a kernel task is determined in a SM, the number of total active thread blocks at every moment indicates the number of used CapSMs, which shows the used SMs in the view of capacity. In SMGuard, the restriction on the use of SM is transformed to the restriction on the use of capacity based CapSM. In this paper, we say SM resource reservation also means the reservation of SM capacity.

Given a resource constraint N, if the number of launched thread blocks does not exceed the maximum number of active thread blocks of N SMs, the constraint can be satisfied automatically. However, in real system, the number of launched thread blocks of a kernel usually exceeds the maximum number of active thread blocks of that kernel hosted by the whole GPU, then the kernel will use more than the constrained resources. Thus, there should be a mechanism to ensure the resources used by any number of thread blocks can be limited to a certain number of CapSMs. We utilize the similar persistent threads [23] technique to transform any number of original thread blocks to a certain number of persistent thread blocks (also called workers), thus the resource constraint can be easily satisfied by launching a certain number of workers rather than any number of original block-tasks. To enable preemption on GPU, previous work need to launch the maximum number of workers the GPU can simultaneously host even if the number of block-tasks is far less than the maximum number. Different from previous work, SMGuard is more flexible and it can launch the maximum number of workers any number of SMs, not the whole GPU, can simultaneously host by using CapSM model. The main idea of how SMGuard can do that is presented in section 4.2.

4.2 Restricting Resource Usage With CapSM

Through the previous analyses, we can see that the more launched thread blocks of a kernel, the more resources it will use. SMGuard can achieve resource quota by launching limited thread blocks using CapSM model. A use case can be shown in Fig. 6. In Fig. 6, we assume that the GPU has three SMs and the CUDA grid of the kernel has 6 thread blocks, and we also suppose that the maximum number of active thread blocks of the kernel hosted in a SM is 2. If we use the default MPS mechanism, all the 6 thread blocks will be dispatched to the GPU, and the kernel task will fully utilize the GPU, which will block subsequent kernels. The above problems can be solved using SMGuard. Firstly, the original kernel function should be transformed as illustrated in Fig. 7 through the source to source compilation. Supposing the resource constraint is 1 and the maximum number of active thread blocks of the transformed kernel hosted in a CapSM or SM is also 2. Then the grid size of the transformed kernel is 2, which means only 2 workers will be launched and the 2 workers will fill one CapSM. The original 6 block-tasks will be assigned to these 2 workers, so each worker will execute 3 block-tasks. After workers launched, they will execute assigned block-tasks in a loop. Until the kernel task is completed, only 2 workers have processed all the 6 block-tasks and only 1 CapSM is used. Although these 2 workers may be dispatched to 2 different SMs, the most important thing is that used resources of the task can be restricted.

4.3 Adopting CapSM to GPU Kernel

SMGuard uses persistent threads technique to limit the number of launched thread blocks and restrict resources used by a kernel task. Fig. 7 illustrates the relationship between normal thread and persistent thread. To transform the origman CUDA kernel to support persistent thread, we apply static code modification to insert customized APIs automatically using source to source compilation. In order to use SMGuard to manage GPU resource, we also implement coordination mechanism between CPU and GPU.

The CPU is primarily responsible for the interference of normal kernel invocation. In SMGuard, every kernel invocation is redirected to SMGuard runtime, then the SMGuard runtime determines when to submit, what is the resource quota or reservation. And the CPU determines the number of launched workers based on resource quota. In addition, the CPU also notifies GPU kernel to reduce currently utilized GPU resources (CapSMs) when online resource adjustment is performed.

TABLE 1
Parameters used in SMGuard

Symbol	Meaning
$BlockTaskNumber_i$	The block-task number, also the grid size of original kernel task
$PBWorkerNumber_i$	The grid size of transformed kernel
$MaxActivePBlock_i$	The maximum number of active thread blocks of transformed kernel hosted be a SM
$CapSMQuota_i$	Resource quota of the task
$CapSMReser_i$	Resource reservation of the task
$BlockTasksPerPBlock_i$	The number of block-tasks assigned to a worker

The GPU is mainly to launch specified number of workers, which is transformed to persistent mode from the original kernel. All block-tasks are executed within an inserted loop as illustrated in Fig. 7. In SMGuard, each worker belongs to a CapSM, and block-tasks are assigned based on the ID of CapSM. Therefore code segments need to be inserted to kernels to assist workers to know which CapSM it belongs to and which block-tasks it needs to process. In this paper, we call all inserted segments in GPU side as *Proxy*, which serves as the proxy of CPU side *CapSM based Resource Manager*. The *Proxy* component mainly holds 3 roles. First, *Proxy* performs the initialization operations and assigns block-tasks for each worker (see Section 5.1). Second, the GPU side *Proxy* needs to coordinate with CPU to dynamically adjust used resources. There is a detection point at the end of each block-task to check if the CPU sends a resource adjustment signal. If current CapSM is included in the evicted range, then all workers in current CapSM will exit. Furthermore, *Proxy* is also responsible for remapping uncompleted block-tasks from evicted workers to remaining workers, which avoids the relaunch of evicted kernel task. More details about the *Proxy* are explained in Section 5.2.

5 CapSM based Resource Management

The SMGuard resource management mainly consists of offline resource reservation and dynamic resource adjustment. In co-location, the resources assigned to batch tasks should be limited to reserve adequate resources for LS tasks; meanwhile, if the reserved resources are not enough to meet the performance requirement, resources utilized by batch tasks should be evicted online.

5.1 Resource Reservation on GPU

In SMGuard, we use the CapSM model as resource management unit. Resource quota can prevent co-located batch tasks from running out all the available resources and retain enough resources for LS tasks. Thus, when a LS task launched into the GPU, it can immediately obtain needed resources and begin to run. To better describe the mechanism, we define some parameters in Table 1. If a kernel task i needs to use the GPU, it should be submitted to SMGuard runtime using SMGuard API. SMGuard runtime provides two ways to submit a task:

1) **a quota based method**, which ensures the amount of resources used by a task does not exceed its quota; batch tasks can be submitted using this method to restrict available resources;

2) **a reservation based method**, which guarantees the resources needed by a task; LS tasks can be submitted using this method to meet their minimum resource requirements.

Algorithm 1: Resource Decision

```
Input: $Method_{type}$ , $R_{request}$ , $BlockTaskNumber$ ,
       $MaxActivePBlock$ , $R_{remain}$
Output: $PBWorkerNumber$, $BlockTasksPerPBlock$
1  if $R_{request} <= R_{remain}$ then
       /* With sufficient resources, resource request can be met. */
2      $PBWorkerNumber \leftarrow R_{request} * MaxActivePBlock$;
3  else
4      if Method_type == $RESERV$ then
           /* If the remaining resources do not meet the demand
              of a task submitted using reservation based
              method, the online resource adjustment operation
              should be done to prepare enough resources. */
5          $R_{gap} \leftarrow R_{request} - R_{remain}$;
6          do $onlineResourceAdjust(R_{gap})$;
7          $PBWorkerNumber \leftarrow R_{request} * MaxActivePBlock$;
8      else
           /* The resource request of a task submitted using
              quota based method must not exceed the
              remaining resources. */
9          $R_{quota} \leftarrow R_{remain}$;
10         $PBWorkerNumber \leftarrow R_{quota} * MaxActivePBlock$;
11     end
12 end
13 $BlockTasksPerPBlock \leftarrow \frac{BlockTaskNumber}{PBWorkerNumber}$;
```

When the quota based method is used, resource quota $CapSMQuota_i$ should be provided; when the reservation based method is used, resource reservation $CapSMReser_i$ should be provided.

When a task is submitted to its task pool, the resource reservation module in SMGuard will do operations illustrated in Algorithm 1 to determine the actual resources can be used by that task. As shown in Algorithm 1, $Method_{type}$ is the method a task is submitted, and $R_{request}$ is $CapSMQuota_i$ for a task submitted using method 1) and $CapSMReser_i$ for method 2). R_{remain} is the amount of current remaining resources. If R_{remain} is no less than $R_{request}$, $R_{request}$ can be met no matter the task is submitted using method 1) or method 2). Otherwise, if the task is submitted using method 1), its $R_{request}$ should not exceed R_{remain} and it can use all the remaining resources; if the task is submitted using method 2), R_{gap} will be calculated to get the resource difference, then the *onlineResourceAdjust* module will be called to evict resources specified by R_{gap} from tasks currently running on GPU in order to satisfy $R_{request}$.

After the above operations are completed, $PBWorkerNumber$ and $BlockTasksPerPBlock$ can be got. Then the task can be issued to GPU, which is configured

Algorithm 2: GPU Kernel Proxy

```
Input: $kernel\_arg\_list$ , $BlockTasksPerPBlock$ ,
       $MaxActivePBlock$ , $CapSMQuota$
/* some initialization                                  */
1  $WorkerIdxInCapSM \leftarrow getWorkerIndxInCapSM()$;
2  $PBlockId \leftarrow blockDim.x * blockIdx.y + blockIdx.x$;
3  $CapSMId \leftarrow \lceil \frac{PBlockId}{MaxActivePBlock} \rceil$;
4  do
5      $StartTaskId \leftarrow$
         $PBlockId * BlockTasksPerPBlock$;
6      $EndTaskId \leftarrow$
         $StartTaskId + BlockTasksPerPBlock$;
       /* if it is processing block-tasks from remapped CapSM,
          then get the first uncompleted block-task in
          corresponding worker                          */
7      $StartTaskId \leftarrow getFirstUncompleted(PBlockId)$;
       /* process each block-task in current block-task
          queue:from $StartTaskId$ to $BlockTasksPerPBlock$   */
8      for $CurTaskId \leftarrow [\ StartTaskId, EndTaskId\ )$ do
9          if $CapSMId < evictCapSMFlag$ then
10             same status and return;
11         end
12         do original kernel body ;   /* replace the original
             thread block $ID$ with $CapSMId$ */
13     end
       /* online task remapping                         */
14     if task remap is not checked then
15         if $evictCapSMFlag \in (0, CapSMId\ ]$ then
16             $NumberPerCapSM \leftarrow$
                 $\lceil \frac{evictCapSMFlag}{CapSMQuota - evictCapSMFlag} \rceil$;
17             $StartCapSMId \leftarrow$
                 $(CapSMId - evictCapSMFlag) *$
                 $NumberPerCapSM$;
18             $EndCapSMId \leftarrow$
                 $StartCapSMId + NumberPerCapSM$;
19             $CurCapSMId \leftarrow StartCapSMId$;
20         else
21             return;
22         end
23     else
24         $CurCapSMId \leftarrow CurCapSMId + 1$;
25     end
26     $PBlockId \leftarrow CurCapSMId * MaxActivePBlock +$
         $WorkerIdxInCapSM$;
27 while $CurCapSMId \in [\ StartCapSMId, EndCapSMId\ )$;
```

to create $PBWorkerNumber$ thread blocks (workers) regardless of its input data size or $BlockTaskNumber$ and each worker will assign $BlockTasksPerPBlock$ block-tasks.

After $PBWorkerNumber$ workers are created and the task starts running on GPU, each worker will first do some initializations to calculate its global block ID (also called global worker ID), the ID of CapSM it belongs to and the range of block-tasks it will process.

The global block ID $PBlockId$ can be got easily using the private macros of each thread provided by CUDA:

$$PBlockId = blockDim.x * blockIdx.y + blockIdx.x \quad (1)$$

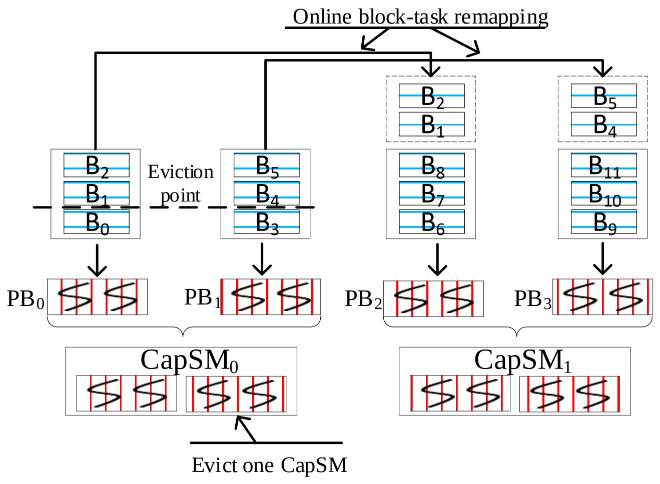


Fig. 8. Overview of online task remapping proxy on the GPU side.

Then, $PBlockId$ divided by $MaxActivePBlock$ is CapSM ID:

$$CapSMId = \left\lceil \frac{PBlockId}{MaxActivePBlock} \right\rceil \quad (2)$$

In SMGuard, every $MaxActivePBlock$ consecutive workers are logically mapped into the same CapSM, although they may be dispatched across several different SMs. Next, the block-task range is calculated for the worker to execute. The start block-task ID $StartTaskId$ is:

$$StartTaskId = PBlockId * BlockTasksPerPBlock \quad (3)$$

and the end block-task ID $EndTaskId$ is:

$$EndTaskId = StartTaskId + BlockTasksPerPBlock \quad (4)$$

The above initializations can be illustrated in line 1-6 of Algorithm 2. After initializations, each worker will execute every block-task in the block-task range from $StartTaskId$ to $EndTaskId$ in a loop, which can be illustrated in line 8-13 of Algorithm 2.

5.2 Dynamic Resource Adjustment

From previous section, we can see that if the resource request $R_{request}$ of a task that submitted using the reservation based method cannot be satisfied by the current remaining GPU resources R_{remain} , the *onlineResourceAdjust* module will be called to evict certain resources from tasks currently running on GPU in order to satisfy $R_{request}$. This section describes the mechanism of dynamic resource adjustment.

To send resource eviction signal to running tasks on GPU, we create a volatile variable $evictCapSMFlag$, which means the number of CapSMs should be evicted, for each task in the Unified Memory in CUDA, which can make $evictCapSMFlag$ visible on both CPU and GPU.

If the resources of a task should be evicted, SMGuard will assign R_{gap} or some other values to $evictCapSMFlag$ of that task. To receive the eviction signal in a running task, each worker will check the variable $evictCapSMFlag$ at the end of a block-task. As shown in line 9-11 of Algorithm 2, if $evictCapSMFlag$ is set, all workers that CapSM IDs are less than $evictCapSMFlag$ should exit. In this way, all workers belong to certain CapSMs will exit and the corresponding

```
//CPU code:
  ...
int main(int argc, char *argv[]){
  ...
   vectorAdd<<<blocksPerGrid,          \\
                  threadsPerBlock>>>
     (d_A, d_B, d_C, numElements);
  ...
}
// GPU kernel:
__global__ void
vectorAdd(const float *A,
              const float *B,
              float *C,
              int numElements)
{
   int i = blockDim.x * blockIdx.x +   \\
                          threadIdx.x;
   if (i < numElements)
   {
      C[i] = A[i] + B[i];
   }
}
```

(a) Original Code

```
//CPU code:
  ...
#include "SMGuard.h"
int main(int argc, char *argv[]){
  ...
   /* initialization and resource decision */
hook=SMGuard_Kernel_Init((void*)vectorAdd,  \\
           blocksPerGrid, threadsPerBlock);
   /* wait kernel to be launched */
   _SMGuard_Pre_Kernel_Launch  //macro
   vectorAdd<<<_SMGuard_gridSize(hook),        \\
                         threadsPerBlock>>>     \\
                 (d_A, d_B, d_C, numElements,   \\
                 _SMGuard_runParams(hook));
   _SMGuard_Post_Kernel_Launch //macro
   /* kernel completed */
  ...
}
// GPU kernel:
__global__ void
vectorAdd(const float *A, const float *B, float *C,
      int numElements,_SMGuard_declareParams)
{
   /* some worker initializations */
   _SMGuard_Kernel_Begin //macro
   /* replace blockIdx in kernel body with taskIdx */
   int i = blockDim.x * taskIdx.x + threadIdx.x;
   if (i < numElements)
   { C[i] = A[i] + B[i]; }
    /* wait  all block_tasks to be completed and */
    /* block_task  remapping*/
   _SMGuard_Kernel_End //macro
}
```

(b)Transformed Code

Fig. 9. Transformed GPU program example (*vectorAdd*) using SMGuard API (inserted codes are shown in bold font).

resources are evicted. Besides, the information about which block-tasks are not processed should also be recorded on exiting. Considering that block-tasks are processed in turn in each worker, it is enough to just record the ID of next block-task should be processed in each exiting worker.

For these uncompleted block-tasks, a relaunch of the kernel task, which is adopted in previous works, can complete all these uncompleted block-tasks. However, different from previous works, SMGuard provides an online block-task remapping mechanism illustrated in Fig. 8 to avoid unnecessary kernel relaunch. Block-tasks, which are not completed because of eviction, will be remapped to remaining workers. After each worker completes originally assigned block-tasks, they will first check if resource eviction happened. If resource eviction has happened, workers have to know which block-tasks of the evicted workers should be remapped to them. To get that information, workers first get the number $NumberPerCapSM$ of evicted CapSMs will remapped to each not evicted CapSM:

$$\begin{aligned} &NumberPerCapSM \\ &= \left\lceil \frac{evictCapSMFlag}{CapSMQuota - evictCapSMFlag} \right\rceil \end{aligned} \tag{5}$$

Then, the remapped CapSM range of each worker is also calculated. The start CapSM ID $StartCapSMId$ is:

$$\begin{aligned} StartCapSMId = (&CapSMId - evictCapSMFlag) \\ &* NumberPerCapSM \end{aligned} \tag{6}$$

and the end CapSM ID $EndCapSMId$ is:

$$EndCapSMId = StartCapSMId + NumberPerCapSM \tag{7}$$

Gotten the above information, each worker in CapSMs that are not evicted will process those uncompleted block-tasks of the corresponding worker in each CapSM in the remapped CapSM range from $StartCapSMId$ to $EndCapSMId$ in a loop. Before a worker begins to process

TABLE 2
Hardware and software specifications

	Specifications
Hardware	CPU:Intel Xeon E5-2620 v4 @ 2.10GHZ GPU:Nvidia GTX 970
Software	OS:Centos 7.3 x86_64 with kernel 3.10.0-514 GPU driver:375.26 CUDA version:8.0

the uncompleted block-tasks of an evicted worker, its global ID $PBlockId$ will be changed to the following value:

$$\begin{aligned} PBlockId = &CurCapSMId * MaxActivePBlock \\ &+ WorkerIdxInCapSM \end{aligned} \tag{8}$$

This allows the worker to process the uncompleted block-tasks of an evicted worker as if it were the evicted worker itself. The above details of online block-task remapping mechanism are illustrated in line 14-27 of Algorithm 2.

5.3 SMGuard API and Code Transformation

For the convenience of code transformation, we define a set of SMGuard API for both GPU kernel and CPU code. Fig. 9 shows the example of using the API. All APIs are declared in the *SMGuard.h* header file, thus the header file should first be included to use the SMGuard API. Before submitting kernel task to GPU for execution, the *SMGuard_Kernel_Init* API provided by SMGuard runtime will be invocated. *SMGuard_Kernel_Init* first does some initialization and allocates structure for the corresponding kernel, then SMGuard gets the actual resource allocated to that kernel task according to Algorithm 1. In this paper, the resource decision method and resource requirement for each task are obtained from a configuration file. However, it should be noted that SMGuard runtime can be easily extended with a similar performance model proposed in previous work [5] to predict resource requirements dynamically. After resource decision, the grid size of kernel task is changed to the value represented by macro *_SMGuard_gridSize*. Moreover, some extra parameters are needed for GPU-side proxy to perform functions illustrated in Algorithm 2. These parameters are declared as macro *_SMGuard_declareParams*, and the actual parameters passed to the launched kernel are represented by macro *_SMGuard_runParams*. Each kernel invocation is wrapped up by two macros: *_SMGuard_Pre_Kernel_Launch*, which waits the kernel to be launched by SMGuard runtime, and *_SMGuard_Post_Kernel_Launch*, which guarantees all block-tasks are completed. Besides, the kernel body is also wrapped up by two macros: *_SMGuard_Kernel_Begin* and *_SMGuard_Kernel_End*, and the *blockIdx* in kernel body is changed to *taskIdx*. The two macros help to convert the entire kernel body into the body of a loop, and detailed information about the macros can be found in previous sections.

To achieve automatic code transformation, we design a source-to-source compiler to transform original GPU program into SMGuard enabled version. The compiler is implemented based on clang LibTooling and LibASTMatchers [24], which can be used to manipulate the abstract syntax

TABLE 3
Benchmark specification

Benchmark	Source	Kernel	Gride size	Duration	TB size	TBs /SM	Transformed PBs/SM
nn	Rodinia [25]	euclid	32768x1x1	0.65ms	256x1x1	8	8
particlefilter (pl)	Rodinia [25]	kernel	79x1x1	2.81ms	128x1x1	16	16
			1563x1x1	312.51ms			
pathfinder (pf)	Rodinia [25]	dynproc_kernel	23149x1x1	3.7ms	256x1x1	8	8
myocyte (mc)	Rodinia [25]	solver2	4x1x1	531.73ms	32x1x1	16	12
			96x1x1	2123.46ms			
lavaMD (md)	Rodinia [25]	kerne_gpu_cuda	64x1x1	6.74ms	128x1x1	10	9
			1000x1x1	143.61ms			
matrixMul (mm)	CUDA SDK [26]	matrixMulCUDA	32x32x1	4.12ms	32x32x1	2	2
			128x128x1	236.91ms			
vectorAdd (va)	CUDA SDK [26]	vectorAdd	156250x1x1	3.15ms	256x1x1	8	8
blackscholes (bs)	CUDA SDK [26]	BlackScholesGPU	39063x1x1	1.34ms	128x1x1	16	16
			390625x1x1	13.33ms			
bfs	SHOC [27]	BFS_kernel_warp	3907x1x1	2.01ms	1024x1x1	2	2
md5hash (md5)	SHOC [27]	FindKeyWithDigest _Kernel	2605x1x1	1.67ms	384x1x1	5	5
			25432x1x1	39.13ms			

tree of CUDA program (both CPU code and GPU code) directly. The source-to-source compiler first extracts the syntax tree of CUDA program then inserts above header file and APIs at corresponding positions.

6 Evaluation

In this section, we evaluate the efficiency of SMGuard for eliminating the performance interference of application co-location as well as improving resource utilization on GPU.

6.1 Experimental Setup

The hardware and software specifications in our experiments are showed in Table 2. CUDA MPS is used to enable concurrent task execution on GPU. As shown in Table 3, Benchmarks are chosen from three benchmark suites, including Rodinia [25], CUDA SDK [26] and SHOC [27] benchmark suite. We show the grid size, the thread block size, the average kernel duration when runs alone on the GPU, the maximal number of concurrent thread blocks of original kernel per SM can host and the maximal number of concurrent thread blocks of transformed kernel per SM can host. We select a subset of benchmarks from each benchmark suite so that they show diverse characteristics. From these benchmarks, we select *pl*, *mc*, *md*, *mm*, *bs* and *md5* as batch applications, and *nn*, *pf*, *va* and *bfs* as LS applications. All LS applications use the reservation based method to get enough resources to ensure their performance, and all batch applications use the quota based method to limit their resource usages.

Based on these benchmarks, we co-locate each LS application with all batch applications, and a LS application and a batch application form a co-location pair *A*+*B*. In each co-location, LS application issues its requests for GPU 1ms after the batch application. To ensure the statistical significance of the experimental results, every experiment is conducted 20 times and the average result is recorded. Besides, we also evaluate SMGuard on a real application (LULESH [28]), and the results show that the performance of LULESH can be improved by about 7.6× on average when co-located with other applications, and the average introduced overhead of the turnaround time is only about 5.7%. The results demonstrate that SMGuard is effective to preserve GPU resources for real applications such as LULESH.

We use the metrics suggested by [29] to characterize co-location performance. Average normalized turnaround time (ANTT) is used to quantify the slowdown due to co-location, which is defined in Equation 9, where N denotes the number co-located kernels, T_i^c is the kernel duration when a kernel is executed under co-location, and T_i^s is the kernel duration when a kernel is executed alone. System overall throughput (STP) defined in Equation 10 measures the progress of the system under co-location, where parameters are the same as in ANTT.

$$ANTT = \frac{1}{N}\sum_{i=1}^{N} NTT = \frac{1}{N}\sum_{i=1}^{N}\frac{T_i^c}{T_i^s} \tag{9}$$

$$STP = \sum_{i=1}^{N}\frac{T_i^s}{T_i^c} \tag{10}$$

6.2 Results on Resource Reservation

6.2.1 Efficiency of CapSM

SMGuard uses CapSM as the resource management unit to provide a flexible way to guarantee the reserved resources for LS tasks. We use Filling & Retreating mechanism (SM) as the baseline to demonstrate the efficiency of CapSM. Fig. 10 shows the normalized performance degradation of CapSM over SM. The x-axis indicates the benchmark pair *A*+*B*, and we restrict the resources used by batch application *B* and reserve the rest resources to LS application *A*. The y-axis shows the kernel execution time of *A* using CapSM based resource reservation mechanism normalized to the kernel execution time of *A* using Filling & Retreating based mechanism. To accurately collect performance data of *A* using the reserved resources when co-located with *B*, we run *B* with large input and *A* with smaller input to make

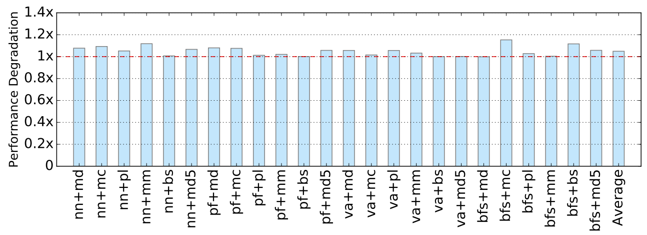


Fig. 10. Normalized performance degradation of CapSM based resource reservation over Filling & Retreating based resource reservation.

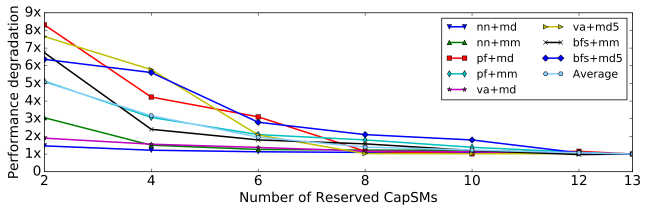


Fig. 11. Performance degradation of bursty LS kernels over standalone performance with different number of reserved CapSM.

B occupy allocated resources as long as possible, which ensures when *A* request the GPU resource, *B* is still running on the GPU with allocated resources. In addition, to set enough time for *B* to evict designated SMs when using Filling & Retreating mechanism, we let *B* run first and then launch *A* after 1ms interval. The experiments are repeated multiple times with different number of reserved SMs and CapSMs, and the average results are shown in Fig. 10.

Results shown in Fig. 10 reflect the efficiency of CapSM to reserve GPU resources. We observe that the normalized performance degradation of CapSM compared to SM is only 1.049× on average, which means that CapSM can be considered equivalent to SM if the slight performance loss is tolerated. There are a few cases, such as *nn+mc*, *nn+mm*, *bfs+mc* and *bfs+bs*, where the normalized performance degradations are close to 1.2×. However, unlike Filling & Retreating mechanism, which enables each SM exposed to only one kernel, CapSM allows thread blocks from different kernels being dispatched to the same SM, which may lead to shared resource contention. Although the performance degradation is close to 1.2× in the worst case, the overall turnaround time of the application does not increase significantly. The reason is explained in section 6.4. Considering CapSM provides a more flexible and efficient way of managing GPU resources, it is beneficial to use CapSM as the resource management unit.

6.2.2 Response to Bursty Task

Reservation based resource management mechanism prevents batch tasks from exhausting all resources and retain enough resources for LS tasks. Thus, when LS tasks, specially when the tasks are bursty, are launched onto the GPU, they can immediately obtain required resources and begin to run. To evaluate the effect of different amount of reserved CapSMs on the performance of LS tasks, we run several pairs of co-location (*A*+*B*). Each application *B* in the co-location pair keeps invoking its kernel in a loop, and each application *A* will invoke its kernel in a random interval to simulate bursty tasks. When using the resource reservation, each kernel from application *B* is assigned with different number of CapSMs and task *A* use the rest of the resources. We use the performance of *A* in its standalone run as the baseline. As shown in Fig. 11, as the number of reserved CapSMs increases, the performance of *A* is approaching the baseline. When the number of reserved CapSMs is less than 8, the performance degradation changes rapidly with the increase of reserved CapSMs. When the reserved number of CapSMs reaches to 8, the performance degradation of *A* in most co-location pairs is less than 2× of the baseline. As the number of reserved CapSMs continues to increase, the performance variation becomes small. Thus, using SMGuard, the resources available to a kernel can be effectively reduced while the reduction in performance is maintained at a relatively low level. In the following experiments, the default number of reserved CapSMs is 8 if not specified explicitly.

6.2.3 Turnaround time and Throughput

Fig. 12 presents the performance improvements for LS applications. Compared to MPS based co-locations, our approach achieves the performance speedup of LS applications by 9.8× on average and up to 55.2× (*nn* co-located with *md5*). The minimum speedup is 1.9× in the case of *va* co-located with *bs*. The reason for the varying speedup is that different applications have different kernel execution time and requirements for computing resources. Therefore, the extent of performance interference varies across co-located pairs. For instance, with *nn*, its kernel execution time is only 0.65ms when running alone and hence its turnaround time is quite sensitive to waiting time under MPS based co-locations. With SMGuard, it reserves a certain amount of resources to reduce the waiting time for *nn*. In addition, although the kernel execution time of *mc* is up to 2123ms, it does not expose too much interference on its co-locators. This is because *mc* only needs 6 SMs to host all its launched thread blocks, which generates less interference to its co-locators even in MPS based scheduling. Therefore, the speedups of its co-locators using SMGuard over MPS are relatively low.

Fig. 13 shows the ANTT of each co-location pair in SMGuard and MPS respectively. The default MPS based co-locations causes the ANTT increase up to 30.3 in the worst case and the average ANTT of all the co-location pairs is 6.5. Besides, the ANTTs of different co-location pairs in MPS differ greatly between 1 and 30. On the contrary, the ANTTs in SMGuard fit in a smaller range and the average ANTT of all co-location pairs is only 1.56. Note that the average ANTT improvement with SMGuard is 4× over MPS, which is less than the performance improvement of LS kernels only. This is because the ANTT is the average NTT of the two kernels in a co-location pair. Although the performance of LS kernel is improved significantly, it is offseted by the performance degradation of batch kernel due to limited resource quota.

Fig. 14 shows that SMGuard improves the performance of LS applications while increasing the overall system throughput (STP). The average STP of SMGuard is 1.57 while 1.22 in MPS. Compared to MPS, the STPs of most co-

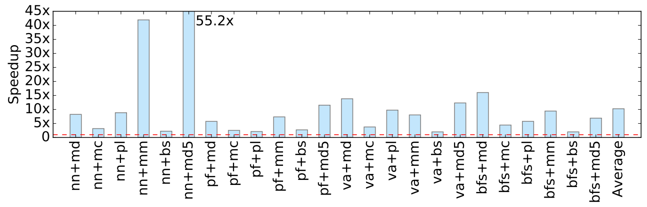


Fig. 12. Performance improvement for LS kernels over MPS.

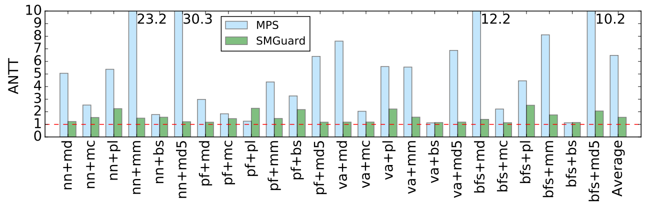


Fig. 13. Average normalized turnaround time (ANTT) comparison.

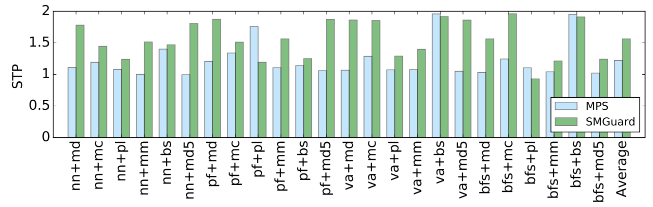


Fig. 14. The comparison of overall system throughput (STP) between SMGuard and MPS.

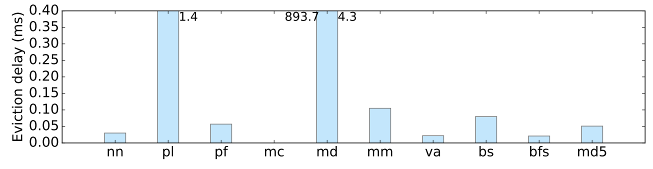


Fig. 15. The average eviction delay for different kernels.

location pairs are increased with SMGuard, the SPTs reduce in very few cases such as *pf* co-located with *pl*, *bfs* co-located with *pl* and *va* co-located with *bs*. In these three cases, the performance improvement of LS kernels is less than the performance degradation of batch co-locators, thus the STPs in these cases decrease. Note that, in systems that LS applications co-located with batch applications, guaranteeing the performance of LS applications is prioritied over improving overall system throughput.

6.3 Results on Dynamic Resource Adjustment

6.3.1 Runtime Task Eviction

Through runtime task eviction, SMGuard can release the occupied resources in the middle of the batch kernel execution on GPU. To measure the duration of this process, we invocate each kernel repeatly in a loop, and send an eviction signal to the kernel randomly after each invocation. Both the signal time and the completion time are recorded to measure the duration. Fig. 15 shows the average eviction duration of each kernel. Eviction durations of *nn*, *va* and *bfs* are less than 0.5ms, whereas the eviction duration of *mc* is over 800ms. The time difference among the above cases is due to the eviction mechanism. Since the eviction flag is checked at the end of a block-task, the eviction is delayed as long as the length of a block-task. Workload *mc* contains a large block-task, which is more than 1000ms. Therefore it is reasonable that the average eviction duration of *mc* is very long. That result also proves that just preemption mechanism is not enough and resource quota is also necessary in co-location to ensure LS tasks obtain enough resources as quickly as possible. For large batch kernels, enough resources can be reserved for LS kernels to avoid the long eviction operation. And for small batch kernels, since they can release resources quickly when LS kernels arrive, more resources are assigned to them to improve GPU utilization.

6.3.2 Online Block-Task Remapping

To evaluate the performance of online block-task remapping, for each kernel, we first fill all SMs or CapSMs with each kernel, then evict certain number of SMs or CapSMs. Fig. 16 shows the normalized kernel latency using SMGuard and Filling & Retreating mechanism. In Filling & Retreating mechanism, if a kernel is evicted, it needs to be relaunched more than once to complete uncompleted block-tasks. We observe the normalized latency of Filling & Retreating mechanism is 1.8× of the baseline on average and up to 2.9× in the worst case. And the normalized latency of SMGuard is 1.2× of the baseline on average and up to 1.7× in the worst case, which is better than the Filling & Retreating mechanism. The reason for the significant difference is that there is no online block-task remapping in Filling & Retreating mechanism and the evicted kernel needs to be relaunched in order to complete the evicted block-tasks. On the contrary, in SMGuard, block-tasks in evicted CapSMs are dynamically remapped to remaining CapSMs and all block-tasks are completed in one kernel invocation, which avoids unnecessary relaunch of preempted kernel.

It should be noted that there is no delay between the two invocations of the preempted kernel in the above experiment and the kernel can use all GPU resources in the second invocation. However, in realistic, when a kernel is preempted, the evicted SMs are usually occupied by other kernel for a period of time, and thus the block-tasks dispatched to the evicted SMs are not executed until the evicted SMs are available again. This is because block-tasks are dispatched according to SM IDs in Filling & Retreating mechanism and each SM is only responsible for the block-tasks assigned to it. In other word, one SM can not process block-tasks evicted from other SMs even if it is idle.

6.4 Overhead Analysis

To evaluate the overhead introduced by SMGuard, we run original and transformed versions of each benchmark respectively. Then we calculate the execution time difference of the transformed and original versions, and use the ratio between execution time difference and original execution time as the overhead. To make the evaluation statistically significant, each benchmark is executed 20 times and the average results is reported. Both kernel execution time

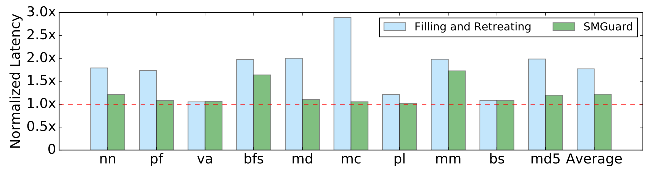


Fig. 16. Normalized latency of different kernels when using SMGuard with online block-task remapping and Filling & Retreating mechanism respectively.

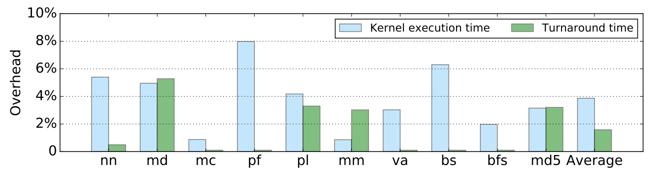


Fig. 17. The overhead of SMGuard for kernel execution time and turnaround time of different benchmarks over baseline.

and overall turnaround time of each benchmark are presented. Fig. 17 demonstrates that the introduced overhead of SMGuard for both kernel execution time and overall turnaround time of most benchmarks are less than 5%. Specifically, the average introduced overhead of SMGuard for kernel execution time and turnaround time are only 3.87% and 1.58% respectively. This indicates that SMGuard itself does not introduce too much overhead to the original kernels. The introduced overhead for turnaround time is not always consistent with execution time, this is because if the kernel execution time takes only a small proportion of the overall turnaround time, such as *nn*, *pf* and *bs*, the introduced overhead for kernel execution time does not have a great impact on the overall turnaround time.

6.5 Limitations

In our current implemention of SMGuard, we use static code compilation to transform original kernel into SMGuard enabled version. In reality, the source code may not be available. It is meaningful to enable executable binary to support SMGuard. To achieve this goal, it is necessary to take measures for CPU and GPU respectively during the code loading process. In CPU, the kernel invocation API and its parameters should be intercepted, which subsequently are sent to SMGuard runtime for resource management decision. In Linux, we can combine LD_PRELOAD [30] or libmonitor [31] with symtabAPI [32] to dynamically intercept kernel invocation API and extract its parameters, and SMGuard can make decisions based on these information. Unlink CPU, the GPU kernel is a pure function and doesn't invocate any CUDA API. We can use the just in time kernel code rewriting mechanism proposed in GPES [20] to explore possibilities to dynamically insert all needed code segments to .cubin files when loaded. We would like to explore the method of dynamic API intercepting and kernel rewriting mechanism in our future work.

7 Related Work

There has been a large amount of previous works [3], [4], [7], [8], [9], [10] focus on resource contention when co-running batch workloads with LS workloads. Ubik [7] and Vantage [8] use fine-grained cache partition mechanism to boost the allocation for LS workloads. Elfen [9] reserves the two SMT lanes in a core to a batch thread and a LS thread respectively and only run the batch thread when no LS thread is executing. Dirigent [10] first accurately predicts the completion time of a running task then controls resource during the tasks execution to meet deadlines and maximize batch throughput. Bubble-Up [3] and Bubble-Flux [4] identify safe co-locations that bound performance degradation while improving chip multiprocessor utilization. However, all these techniques are applied to CPU not GPU because of the architecture difference between CPU and GPU. Baymax [5] provides a task duration modeling methodology to predict the duration of GPU tasks and reorders the tasks execution according to prediction results. Prophet [12] builds detailed analysis models to predict performance interference precisely for application co-location on GPU.

GPU scheduling is another research direction related to SMGuard. Kato et al. designed and implemented TimeGraph [33], RGEM [34] and Gdev [35] for real time system. TimeGraph is a device driver solution to provide a kernel space real-time GPU scheduler designed for computer graphics. RGEM supports the same concept as TimeGraph at the user-space runtime level. Gdev integrates the real-time scheduler into the open source CUDA framework they developed. GPUSync [36] is another framework for managing GPUs in multi-GPU multicore real-time systems. Suzuki et al. [37] realize real-time GPU resource manage with loadable kernel models. Xu et al. [38] improves the schedulability of real-time tasks with mixed timing constraints. These techniques focus on increasing throughput for high-priority tasks, and schedule GPU tasks in priority order. Thus, when high-priority task is running on GPU, low-priority tasks need to be blocked even though the GPU is not fully utilized by high-priority task, which overlooks the overall system utilization of GPU.

To intercept the execution of a long running kernel, kernel slicing has been adopted by prior works [20], [21], [27] to support preemptive scheduling. This technique slices long-running kernel into multiple short-running ones through invocating the same kernel multiple times but each invocation of the kernel only executes a range of all the thread blocks. After each short-running sub-kernel completed, the subsequent sub-kernel invocation can be preempted by other kernels. Kernel slicing will invocate the same kernel multiple times even if there is no other kernel need to use the GPU, thus, unnecessary overhead will be introduced.

Persistent threads [23] is another very important technique that has been adopted by SM-Centric [16], EffiSha [17], FLEP [18] and Versapipe [39]. SM-Centric [16] first proposes the state-of-the-art Filling & Retreating mechanism using persistent threads to enable tasks can be scheduled to the suitable SMs. EffiSha [17] and FLEP [18] put forward spatial preemption, which can schedule tasks on the block-task level, on the basis of Filling & Retreating mechanism. Versapipe [39] also relies on SM-Centric to bind kernels onto target SMs and map blocks on corresponding SMs for pipelined computing. SMGuard adopts a similar technique based on persistent threads, but the difference is that SMGuard doesn't have to fill the whole GPU when a kernel is launched. Besides, SMGuard supports a hardware independent preemption mechanism while the spatial preemption in [17], [18] only works in Nvidia GPU with CUDA programming model.

At the hardware level, many works [13], [14], [15], [40], [41] exploit hardware mechanisms for multitask situations. Aguilera et al. [40] propose a runtime technique in GPU to dynamically partition GPU resources between concurrently running applications to satisfy QoS requirements. Tanasic et al. [14] propose a SM-draining technique that improves performance of high priority processes by enabling preemptive scheduling on GPUs. However, the SM-draining technique can cause long preemption latency. To address the long preemption latency, Park et al. [15] propose SM-flushing, which can immediately stop the execution of a kernel and flush all intermediate results. Lin et al. [41] reduce the overhead of context switching by compressing the TB-level state. Adriaens et al. [13] demonstrate the potential benefits of spatial multitasking using simulation. All these works are done in simulator [42] and need hardware modifications or extensions for realistic use, which are impractical to be applied in commodity GPUs immediately.

8 Conclusions

In this paper, we propose a flexible and fine-grained resource management framework *SMGuard*, for application co-location on GPU. SMGuard allows multiple applications to share the GPU resources under a controllable performance degradation. SMGuard uses the capacity based SM model *CapSM* as the resource management unit. Through CapSM, SMGuard implements a flexible resource quota mechanism to prevent batch tasks from exhausting all available resources and reserve enough resources for LS tasks. Thus, when a LS task is launched onto the GPU, it can immediately obtain needed resources and begin to run. The resource reservation mechanism improves the response latency of the LS task. SMGuard also supports dynamic resource adjustment to manage the resource quota of the batch task during runtime and remap preempted block-tasks into remaining resources without relaunching the preempted task. The experiments show that SMGuard improves the performance of the LS tasks by 9.8× on average. In addition, the overall system throughput is also improved by 35% on average. With the benefits of online block-task remapping, the normalized latency of the preempted tasks is reduced by 60% on average. SMGuard does not rely on special hardware or programming model, and can be easily adopted on commodity GPUs.

Acknowledgments

We thank our anonymous reviewers for their valuable feedback and suggestion. This work is supported by the National Key Research and Development Program of China under Grant No. 2016YFB1000503 and the National Science Foundation of China under Grant No. 61572062 and No. 61502019. Hailong Yang is the corresponding author.

References

[1] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and QoS-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.
[2] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 4.
[3] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011, pp. 248–259.
[4] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 607–618.
[5] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 681–696, 2016.
[6] Nvidia, "CUDA MPS," https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.
[7] H. Kasture and D. Sanchez, "Ubik: efficient cache sharing with strict qos for latency-critical workloads," in *ACM SIGPLAN Notices*, vol. 49, no. 4. ACM, 2014, pp. 729–742.
[8] D. Sanchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3, pp. 57–68, 2011.
[9] X. Yang, S. M. Blackburn, and K. S. McKinley, "Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading." in *USENIX Annual Technical Conference*, 2016, pp. 309–322.
[10] H. Zhu and M. Erez, "Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 33–47, 2016.
[11] Nvidia, "Nvidia Tesla V100 GPU Architecture," http://composter.com.ua/documents/Volta-Architecture-Whitepaper.pdf.
[12] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 17–32.
[13] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for GPGPU spatial multitasking," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. IEEE, 2012, pp. 1–12.
[14] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on GPUs," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 2014, pp. 193–204.
[15] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared GPU," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 593–606, 2015.
[16] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, "Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 119–130.
[17] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "EffiSha: A software framework for enabling effficient preemptive scheduling of GPU," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2017, pp. 3–16.
[18] B. Wu, X. Liu, X. Zhou, and C. Jiang, "FLEP: Enabling flexible and efficient preemption on GPUs," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 483–496.
[19] C. Basaran and K.-D. Kang, "Supporting preemptive task executions and memory copies in gpgpus," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*. IEEE, 2012, pp. 287–296.
[20] H. Zhou, G. Tong, and C. Liu, "Gpes: a preemptive execution system for gpgpu computing," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*. IEEE, 2015, pp. 87–97.
[21] J. Zhong and B. He, "Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling," *IEEE Transactions*

on Parallel and Distributed Systems, vol. 25, no. 6, pp. 1522–1532, 2014.

[22] Nvidia, "Profiler Users Guide," http://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[23] K. Gupta, J. A. Stuart, and J. D. Owens, "A study of persistent threads style GPU programming for GPGPU workloads," in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–14.

[24] clang, "clang: a C language family frontend for LLVM," https://clang.llvm.org/.

[25] 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 *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 44–54.

[26] Nvidia, "Nvidia CUDA SDK," http://developer.nvidia.com/gpu-computing-sdk.

[27] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 63–74.

[28] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong, "LULESH Programming Model and Performance Ports Overview," Tech. Rep. LLNL-TR-608824, December 2012.

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

[30] K. Pulo, "Fun with LD_PRELOAD," in *linux. conf. au*, 2009.

[31] M. W. Krentel, "Libmonitor: A tool for first-party monitoring," *Parallel Computing*, vol. 39, no. 3, pp. 114–119, 2013.

[32] B. Buck and J. K. Hollingsworth, "An API for runtime code patching," *The International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, 2000.

[33] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *Proc. USENIX ATC*, 2011, pp. 17–30.

[34] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "RGEM: A responsive GPGPU execution model for runtime engines," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*. IEEE, 2011, pp. 57–66.

[35] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt, "Gdev: First-Class GPU Resource Management in the Operating System." in *USENIX Annual Technical Conference*. Boston, MA;, 2012, pp. 401–412.

[36] G. A. Elliott, B. C. Ward, and J. H. Anderson, "GPUSync: A framework for real-time GPU management," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*. IEEE, 2013, pp. 33–44.

[37] Y. Suzuki, Y. Fujii, T. Azumi, N. Nishio, and S. Kato, "Real-Time GPU Resource Management with Loadable Kernel Modules," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1715–1727, 2017.

[38] Y. Xu, R. Wang, T. Li, M. Song, L. Gao, Z. Luan, and D. Qian, "Scheduling tasks with mixed timing constraints in GPU-powered real-time systems," in *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 2016, p. 30.

[39] Z. Zheng, C. Oh, J. Zhai, X. Shen, Y. Yi, and W. Chen, "Versapipe: a versatile programming framework for pipelined computing on GPU," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 587–599.

[40] P. Aguilera, K. Morrow, and N. S. Kim, "QoS-aware dynamic resource allocation for spatial-multitasking GPUs," in *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*. IEEE, 2014, pp. 726–731.

[41] Z. Lin, L. Nyland, and H. Zhou, "Enabling efficient preemption for SIMT architectures with lightweight context switching," in *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 2016, pp. 898–908.

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

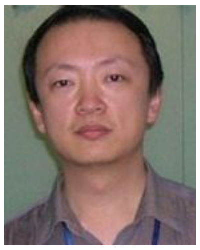
Chao Yu received the BE degree in the School of Information Engineering from Zhengzhou University, China in 2012, and the ME degree in the School of Computer Science and Engineering from University of Electronic Science and Technology of China in 2015. He is currently working toward the Ph.D. degree in the School of Computer Science and Engineering from Beihang University. His research interests including computer architecture, high-performance computing, GPUs.

Yuebin Bai received the Ph.D. degree in computer science from Xian Jiaotong University, Xian, China, in 2001. In 2003, he joined the faculty of Beihang University, where he is currently a full professor in School of Computer Science and Engineering. His current research interests include computing system virtualization, real time and distributed systems, wireless networks. He is a senior member of the China Computer Federation (CCF), a member of the ACM and IEEE, and also a member of the IEICE.

Hailong Yang received the Ph.D degree in School of Computer Science and Engineering, Beihang University, China in 2014. He is an assistant professor in School of Computer Science and Engineering, Beihang University since 2014. His research interests include parallel and distributed computing, HPC, performance optimization and energy efficiency. He is also a member of IEEE and China Computer Federation (CCF).

Kun Cheng received the BS degree in the School of Computer Science and Engineering from Beihang University, China in 2012. He is currently working toward the Ph.D. degree in the School of Computer Science and Engineering from Beihang University. His research interests include system virtualization, cloud computing and real-time systems. He is a student member of both IEEE and the China Computer Federation (CCF).

Yuhao Gu received the BE degree in the School of Computer Science and Engineering, Beihang University, China in 2015. Since 2016, he is currently working toward the Ph.D. degree in the School of Computer Science and Engineering from Beihang University. His research interests focus on the resource management in the area of operating system and cloud computing.



Zhongzhi Luan received the Ph.D. in the School of Computer Science of Xian Jiaotong University. He is an Associate Professor of Computer Science and Engineering, and Assistant Director of the Sino-German Joint Software Institute (JSI) Laboratory at Beihang University, China. Since 2003, His research interests including distributed computing, parallel computing, grid computing, HPC and the new generation of network technology.

Depei Qian received his master degree from University of North Texas in 1984. He is a professor at the School of Computer Science and Engineering, Beihang University, China. He is currently serving as the chairman of China National High Technology Pro-gram (863 Program) on highly efficient computer design and grid service facility. His research interests include innovative technologies in computer network, high performance computer architecture and grid computing. He is also a member of IEEE and China Computer Federation (CCF).