



Adaptive Simultaneous Multi-tenancy for GPUs

Ramin Bashizade^(✉), Yuxuan Li, and Alvin R. Lebeck

Department of Computer Science, Duke University, Durham, NC, USA
{ramin,alvy}@cs.duke.edu, yuxuanlala@gmail.com

Abstract. Graphics Processing Units (GPUs) are energy-efficient massively parallel accelerators that are increasingly deployed in multi-tenant environments such as data-centers for general-purpose computing as well as graphics applications. Using GPUs in multi-tenant setups requires an efficient and low-overhead method for sharing the device among multiple users that improves system throughput while adapting to the changes in workload. This requires mechanisms to control the resources allocated to each kernel, and an efficient policy to make decisions about this allocation.

In this paper, we propose adaptive simultaneous multi-tenancy to address these issues. Adaptive simultaneous multi-tenancy allows for sharing the GPU among multiple kernels, as opposed to single kernel multi-tenancy that only runs one kernel on the GPU at any given time and static simultaneous multi-tenancy that does not adapt to events in the system. Our proposed system dynamically adjusts the kernels' parameters at run-time when a new kernel arrives or a running kernel ends. Evaluations using our prototype implementation show that, compared to sequentially executing the kernels, system throughput is improved by an average of 9.8% (and up to 22.4%) for combinations of kernels that include at least one low-utilization kernel.

1 Introduction

Graphics Processing Units (GPUs) are massively parallel accelerators that were originally intended to execute graphics applications, but their high throughput and energy-efficiency motivates their use by broader application domains. Numerous cloud service providers offer GPUs as part of their solutions [2, 9, 15]. In such environments a large number of kernels with different memory access and compute behaviors request running on GPUs. Running only one kernel on the GPU in these environments underutilizes resources, since a single kernel cannot utilize all the resources on the device most of the time [21]. Therefore, always dedicating the entire GPU to only one customer (or single kernel) is not cost-efficient either for the service provider or for the customer. One example to address this issue is Amazon Web Services' plan to provide fractional GPUs (Elastic GPUs [2]) for applications that have high compute, storage or memory needs that still could benefit from additional GPU resources. Another example

is the NVIDIA Volta architecture’s support for statically dividing the GPU into multiple smaller virtual GPUs [20].

Sequential execution of kernels on GPUs in multi-tenant environments, such as data-centers, leads to long wait times and reduces system throughput. Overcoming this limitation requires a method for sharing the device among multiple users that is efficient and adaptive to the events in the system (i.e., arrival and departure of kernels.) NVIDIA GPUs support simultaneous execution of multiple kernels and memory operations in a single application via Hyper-Q technology [17]. In addition, the CUDA Multi-Process Service (MPS) [18] facilitates concurrent execution of kernels and memory operations from different applications. However, the first-come-first-served (FCFS) and left-over resource allocation policies make concurrent execution on existing GPUs inefficient. The reason is that the FCFS policy creates a head-of-line blocking situation where the running kernel blocks other kernels until it has all its thread blocks mapped to Streaming Multi-processors (SMs). Additionally, simply allocating the left-over resources of the running kernel to the waiting kernels might not be the optimal solution, since such a policy ignores the different requirements of the kernels and only depends on the order in which the kernels arrive at the GPU. The Volta architecture tries to overcome this head-of-line blocking by adding the capability to statically divide the GPU into smaller virtual GPUs, but the above problems apply to each virtual GPU.

An effective and low-overhead scheme for sharing the GPU among multiple kernels should address both the resource underutilization and the adaptiveness issues. This requires overcoming the head-of-line blocking problem in thread block scheduling on the GPU to address the adaptiveness problem, and having a simple yet effective policy for resource allocation to tackle the underutilization issue. Previous work attempted to support multi-tenancy on the GPU either by a software-based approach [6, 30] or by adding the necessary hardware support [22, 27]. These works solely support preemption to make the system responsive, i.e., to force low-priority kernels to yield control of the GPU to high-priority kernels, and hence, do not alleviate the resource underutilization problem. A different class of work addresses multi-tasking on the GPU by modifying the hardware [1, 12, 14, 23, 28, 31] or artificially fusing the kernels from different applications together [10, 14, 21, 29]. In the hardware-based work, the resource allocation policy is fixed and cannot be changed. Furthermore, most of the necessary hardware support is not present in existing GPUs. Software-based approaches that rely on merging applications together are impractical in real world scenarios since it requires merging every possible combination of kernels beforehand. Our work does not suffer from these shortcomings since we use a low-overhead software approach to solve the GPU multi-tenancy problem at run-time.

These challenges inspired us to design a system that realizes multi-tenancy for commodity GPUs. In this paper, we propose adaptive simultaneous multi-tenancy for GPUs. Our system dynamically adjusts the resources allocated to kernels based on the requirements of all kernels requesting execution on the GPU

at run-time. We achieve this by adopting a cooperative approach between applications and a host-side service, supported via minimal application modifications and a lightweight API. Our approach focuses on a single server, as the problem of assigning work to specific servers in data-centers is addressed elsewhere [24]. Therefore, we assume that the work assigned to this machine is optimized by the higher level scheduler.

In our proposed system, we manage the resources allocated to each kernel, and control the mapping of kernels' thread blocks to SMs. Naively applying resource allocation policies can lead to unintended mappings of thread blocks to SMs and result in further underutilization of resources. To avoid this, we build on the concept of persistent threads [11] with a few modifications to implement our desired mapping policy on the GPU. Our work differs from previous work that uses persistent threads to support preemption [6, 30] in that we show how to control the assignment of thread blocks to SMs and use it to have control over resource allocation to kernels. Moreover, support for preemption comes almost for free when we adopt this approach.

To realize adaptive simultaneous multi-tenancy, we implement a host-side service with which applications communicate to obtain launch parameters for their kernels. The service monitors the kernels running on the GPU and makes decisions for launch parameters based on the adopted allocation policy. In this work, we use offline profiling of kernels and implement a greedy policy using this data with the goal of minimizing the maximum execution time among all running kernels, or in other words, maximizing system throughput (STP). We show that using our design, STP is improved by an average of 9.8% (and up to 22.4%) for combinations of kernels that include at least one low-utilization kernel, with respect to the sequential execution of the kernels. Compared to a system in which persistent threads transformation is applied to the kernels, the average STP improvement for these kernel combinations is 4.3%. We do not compare our system with other software-based multi-tenant systems [6, 30], since the target of those systems is to improve the turnaround time of high-priority kernels whereas our goal in this work is to improve the throughput of the whole system. Improving STP, assuming Service Level Agreements (SLAs) are not violated, translates into less energy consumption of the data-center by allowing for reduction in the number of servers for the same amount of work, or in higher scalability by doing more work with the same number of servers, both of which are crucial factors in determining the Total Cost of Ownership (TCO) [4].

We make the following contributions in this paper:

- We identify the need for sharing the GPU among multiple kernels by characterizing the behavior of a set of benchmark kernels. Our observations show that running only one kernel at a time leads to underutilization of different types of resources on the GPU. On the other hand, simply running two kernels together without considering resource utilization does not realize the potential STP gains.

- We use the concept of persistent threads to control the resources allocated to each kernel at run-time. This allows us to solve the head-of-line blocking at the GPU block scheduler¹.
- We design and implement an adaptive simultaneous multi-tenant prototype system that runs on current GPUs. Adaptive simultaneous multi-tenancy is a generalization of single-kernel multi-tenancy [6,30], and static simultaneous multi-tenancy supported by the NVIDIA Volta architecture, in which the GPU is shared by multiple kernels at the same time based on the requirements of all kernels. Our system is composed of a host-side service that makes decisions regarding the allocation of resources to kernels and preemption/relaunch of running kernels, and an application-side API that encapsulates the communications with the service in a few function calls.
- We evaluate the proposed system for a set of benchmark kernels using a full prototype on real GPUs and show the effectiveness of our approach in terms of improving STP.

The rest of this paper is organized as follows. A brief background about GPU execution model as well as motivation for our work are presented in Sect. 2. Section 3 elaborates our proposed system. Evaluation methodology and experimental results are discussed in Sect. 4. Section 5 covers the related work. Finally, Sect. 6 concludes the paper.

2 Background and Motivation

2.1 GPU Execution Model

GPUs are massively parallel accelerators that are composed of thousands of simple cores. A large number of cores together with cache, shared memory², register files, and some other components form streaming multi-processors (SMs). All SMs share a last level cache. Figure 1a and b show the architecture of and SM and the GPU.

Figure 1c illustrates the structure of a kernel. GPU kernels comprise a large number of threads that execute the same instructions on different data, hence the name Single-Instruction-Multiple-Thread (SIMT). These threads are grouped together to form thread blocks, and a set of thread blocks is called a grid. All thread blocks in a grid have the same dimensions, and all threads of the same thread block can only run on a single SM. A thread block is a logical notion that helps the programmers reason about their code. However, a limited number of thread blocks can fit on the device at the same time. We separate these concepts and refer to the physical thread blocks actually running on the GPU as concurrent thread arrays (CTAs). Note that in some other works thread blocks and CTAs are used interchangeably.

¹ The recently announced NVIDIA Volta architecture solves the head-of-line blocking at the GPU block scheduler by dividing the GPU into smaller virtual GPUs, but it lacks the flexibility provided by persistent threads.

² Scratchpad memory in NVIDIA terminology is called shared memory.

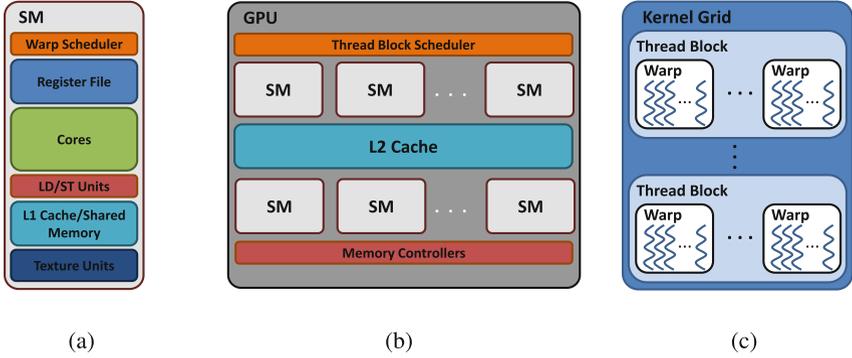


Fig. 1. (a) SM components, (b) GPU components, and (c) kernel structure.

When there are enough resources on an SM to host a waiting thread block, the block scheduler dispatches that thread block to the available SM for execution. If there are more than one SM with enough resources, the mapping happens in a round-robin fashion. In each SM then, warp scheduler dispatches ready warps to execute instructions. A warp is the smallest group of threads that execute in lockstep to reduce the overhead of instruction fetch and control flow. The programmer has no control over the size of a warp. Once a thread block is mapped to an SM, it continues execution until it finishes. In other words, there is no mechanism for preemption or yielding resources (the Pascal [19] and Volta [20] architectures perform context switching, but the programmer does not have control over the operation).

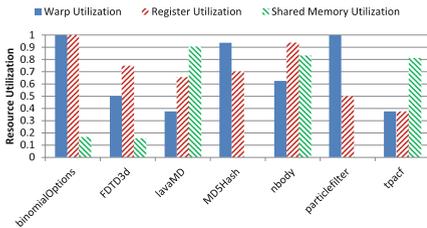


Fig. 2. Spatial utilization of different resources in SMs for the benchmark kernels.

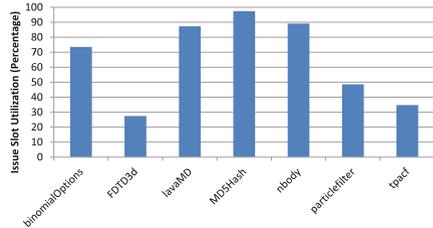


Fig. 3. Issue slot utilization for the benchmark kernels.

2.2 Resource Requirements

Figure 2 shows the amount of SM resources occupied by the benchmark kernels. The data are obtained from the system described in Sect. 4.1 using NVIDIA profiler and the details of the benchmarks are discussed in Sect. 4.2. This figure

does not show how often each of these resources are used, but demonstrates how much of each type is occupied by kernels when run in isolation. In order to distinguish utilization of resources over time, we refer to this metric as spatial resource utilization. There is a limiting resource for every kernel, i.e., the kernels exhaust one or two types of resources while there are more of the other types left unused. This creates opportunities to simultaneously accommodate more kernels with complementary requirements to maximize the throughput of the system. For instance, MD5Hash kernel needs more than 70% of the registers on the device, but uses no shared memory. It can be combined with lavaMD kernel which needs more than 90% of the shared memory to improve the spatial resource utilization of the GPU. *Taking advantage of this opportunity requires a method that shares each SM among multiple kernels, because sharing the GPU among multiple kernels while each SM is dedicated to a single kernel does not alleviate the SM resource underutilization.*

2.3 Issue Slot Utilization

A different metric for utilization is Issue Slot Utilization (ISU). ISU refers to the percentage of issue slots that issued at least one instruction. It is an indication of how busy the kernel keeps the device. Figure 3 shows ISU for the benchmark kernels. The contrast between ISU and spatial resource utilization is visible in Figs. 2 and 3. Figure 2 suggested that MD5Hash and lavaMD are good candidates to be combined together for throughput improvement, whereas Fig. 3 shows that both kernels keep the device busy more than 70% of the time. Thus, although the resource requirements of the two kernels are complementary, there are not many stall cycles during the execution of each of them that the other kernel can take advantage of. Based on the ISU values, lavaMD and tpacf are better candidates to run together, because despite their similar resource requirements, they have complementary ISUs. On the other hand, without complementary resource requirement, it is impossible to fit both kernels on the GPU. *Therefore, an efficient solution is needed to tune the resources allocated to each kernel such that the requirements for both metrics are met.*

2.4 Non-overlapping Execution

CUDA MPS [18] combines multiple CUDA contexts into one to allow for simultaneous execution of multiple kernels from different applications on the GPU. However, our observations show that the block scheduling algorithm on the GPU does not properly take advantage of this capability. This issue is covered in prior work too [6]. When multiple applications want to launch kernels on the GPU, their thread blocks are queued in the order they have arrived at the device. As the resources become available by completion of older thread blocks, newer ones are assigned to SMs. This FCFS policy leads to a head-of-line blocking situation where more resource-consuming kernels that arrived earlier block the execution of less resource-consuming kernels, even though there might be enough resources to accommodate thread blocks of the smaller kernels. To address this issue, we

use persistent threads [11] to restrict the number of CTAs of each kernel on the device, thus constraining the resources it uses.

To summarize, the challenges to sharing a GPU among multiple kernels are (i) managing the resources allocated to each kernel such that the GPU can accommodate all the kernels at the same time, (ii) allocating resources to kernels in order to create complementary utilizations, (iii) addressing the head-of-line blocking at the GPU block scheduler caused by the FCFS policy on the GPU, and (iv) doing all of these at run-time in an adaptive fashion.

3 Adaptive Simultaneous Multi-tenancy

Our proposed solution to the challenges mentioned in Sect. 2 is adaptive simultaneous multi-tenancy. This concept is a generalization of single-kernel multi-tenancy proposed in previous work [6, 30], and static simultaneous multi-tenancy supported by the NVIDIA Volta architecture [20]. The idea is to adaptively tune the resources allocated to each kernel to accommodate more kernels on the GPU while supporting kernel preemption to enhance the throughput of the multi-tenant system. To this end, we propose kernel code transformations and an application API to add flexibility to kernels, and employ a host-side service that monitors the kernels running on the GPU to make decisions regarding resource allocation which are then communicated to applications. In the rest of this section, we explain the details of our design.

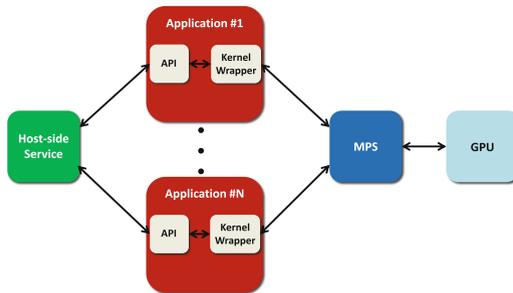


Fig. 4. Overview of the proposed adaptive simultaneous multi-tenant system.

3.1 Overview

Our proposed system is composed of a host-side service that manages resources allocated to each kernel and determines when kernel adaptation needs to occur, and an API for programmers to utilize the service. Figure 4 shows the overview of the adaptive simultaneous multi-tenant system. On arrival of a new kernel or departure of a running one, the service takes the following actions: (i) it asks the applications for the number of thread blocks their kernels have executed, used

in estimation of the remaining execution time. The remaining execution times are used in combination with profiling data in the allocation policy to maximize STP (addressing the challenge in Sect. 2.3); (ii) it computes new parameters for the kernels that are going to run on the GPU. The parameters, which in this work is the number of CTAs but could be extended to different types of resources such as the number registers or the amount of shared memory, must not cause the resources required by the kernels to exceed the available resources on the GPU (addressing challenges in Sects. 2.2 and 2.4); (iii) it communicates the new parameters to the applications.

On the application side, the kernel launch is wrapped inside a function that communicates with the host-side service. We provide an API for the common actions that need to be taken when an application wants to launch a kernel. Ideally these function calls would be included in the CUDA libraries so that the programmer does not have to add anything to their code, but for now they need to be included in the application code manually or via a source-to-source transformation.

In short, the applications that want to run kernels on the GPU have to contact the host-side service using the provided API. The service controls when and which applications should preempt/relaunch kernels and the kernel launch parameters. The kernel launch parameters are passed via the API to the kernel wrapper function that launches the kernel. MPS intercepts kernel calls and merges them in a single context to run on the GPU concurrently. The fact that all concurrent kernels share a virtual address space creates security concerns in a multi-tenant environment. This issue is addressed in the Volta architecture by supporting separation of virtual address spaces for kernels that run on different SMs. Since we share SMs among multiple kernels, this capability does not eliminate the security limitation of our work. However, adding a software address translation layer [25] can isolate the address spaces of different applications with minimal overhead when required.

3.2 Host-Side Service

Applications communicate to the service via the API (Sect. 3.3) at two occasions: (i) launching a kernel, and (ii) starting the execution of the last thread block. The reason that we notify the service at this point, and not once the kernel is finished, is that after the last thread block begins execution no changes can be made to the number of CTAs of the kernel. Therefore, by sending the notification before the kernel finishes, we can overlap the communications with and the parameter computation at the host-side service with the execution of the last round of thread blocks of the kernel, effectively hiding the latency of these operations without affecting the number of kernel’s CTAs.

The service receives messages on a shared message queue in a loop. Whenever it sees a message in this queue, it keeps reading until the queue is empty to aggregate the effects of back-to-back messages from different applications on the system in a single step. After reading all the messages in the queue, the service

opens two dedicated message queues for communications with each client application that has a kernel to run. These queues are used for sending preemption commands and launch parameters, and receiving the progress of the kernel.

When the service is notified by an application of a new event, i.e., a new kernel is arriving or an existing kernel began the execution of its last thread block, it queries other applications for kernel progress via dedicated message queues. Previous work [30] used elapsed time for this purpose, and thus, there was no need to query the application. Nevertheless, this metric is not suitable for our purpose. Elapsed time can be used to measure the progress when only one kernel runs on the GPU at a time, whereas in our proposed system multiple kernels share the device simultaneously and therefore, do not make progress with the same rate as they do when they run in isolation. To overcome this issue, we use the number of executed thread blocks as an indicator of kernel progress.

The service then waits for the response from all applications, as those data are necessary for making allocation decisions. We use asynchronous memory copy operations to overlap these queries with kernel execution. Having the number of executed thread blocks and kernels profiling data, we then estimate the remaining execution time of the kernels (Sect. 3.6). Once this operation is done, the parameters for each kernel are sent to the corresponding application via the dedicated message queues and the application makes the appropriate adjustments.

3.3 Application Side

On the application side, we initialize the shared and dedicated message queues, obtain launch parameters for the kernel, wait for notifications from the service for preemption and new launches, and release the resources on completion of the kernel. Table 1 summarizes the application API to support these actions.

init(): On a kernel launch, the application host code initializes the necessary variables. These include shared and dedicated message queues, necessary memory allocations for communications between the host and the GPU, and streams for asynchronous memory operations and kernel launches. The dedicated message queues are created based on the process ID of the application to ensure uniqueness. There are also pointers to kernel input arguments (`kernel_args`) that are used when launching a new instance of the kernel.

obtainParameters(): Once initialization is complete, the host code obtains parameters for the kernel it wants to launch from the host-side service. To this end, it sends a message composed of the kernel name (`kernel_name`, to retrieve its corresponding profiling data at the host-side service), the names of the dedicated message queues (created using process ID, to open connections to the queues at the service), total number of thread blocks the kernel wants to run (`total_blocks`, to be used for remaining execution time estimation), dimensions of a thread block (`block_dim`, to be used for resource usage calculation), and indication that this message is a request for a new kernel (as opposed to notification for the beginning of the execution of the last thread block of an existing kernel). After the message is sent, the host code waits to receive a response from the service.

Once the response arrives, the kernel is launched and two threads are created: one for listening to the host-side service for new launch parameters, and the other for monitoring the progress of the kernel. The first thread uses the stream for memory operations to asynchronously read the number of executed thread blocks from the device and to write to the memory location holding the preemption variable (`max_blocks` in Fig. 5).

Once a new message with launch parameters comes from the service, there are three possible scenarios: (i) the new number of CTAs is less than what the kernel is currently running with, (ii) new and old numbers of CTAs are equal (i.e., no actions required), or (iii) the new number is greater than the old number of CTAs. In the first case, the thread preempts the proper number of CTAs to match the new parameter by writing to the preemption variable (described in the following section). In the last case, the thread launches a new instance of the kernel on a new stream to run in parallel with the current instance. The second thread is responsible for sending notification to the service once the last thread block of the kernel has started execution.

release(): Finally, when the kernel finishes, the host code deallocates all the resources used for these communications.

Table 1. Application API.

Function	Description
<code>init(kernel_args)</code>	Initializes the necessary variables for communications with the service and launching new instances of the kernel
<code>obtainParameters(kernel_name, total_blocks, block_dim)</code>	Contacts the service with kernel’s information and obtains the number of CTAs to launch the kernel. Also upon receiving the response, creates threads for listening to the service and monitoring kernel progress
<code>release()</code>	Releases the allocated resources

3.4 Kernel Code Transformation

To have control over the number of CTAs and consequently, the resources allocated to the kernel, we use persistent threads [11]. The concept of persistent threads refers to limiting the number of threads to a value that the GPU can run simultaneously. In addition to control over resources, using persistent threads provides support for preemption at thread-level granularity. Preempting kernels only at thread completion mitigates the need for handling any remaining work due to the preemption.

Using a persistent thread transformation, we override the `blockIdx` variable in CUDA which refers to the logical thread block index. Figure 5 shows the

required transformation to the kernel code to implement persistent threads. It also includes the support for preemption and control over the assignment of CTAs to SMs.

```

1  __global__ void TransformedKernel(/*Original Arguments*/,
2                                  int grid_size,
3                                  int *block_index, int *max_blocks,
4                                  volatile int *concurrent_blocks) {
5      int smid = get_smid();
6      __shared__ int logicalBlockIdx;
7      __shared__ int physicalBlockIdx;
8      if(threadIdx.x == 0) {
9          physicalBlockIdx = atomicAdd(&(block_index[smid + 1]), 1);
10     }
11     __syncthreads();
12     while(physicalBlockIdx < *max_blocks) {
13         if(threadIdx.x == 0) {
14             logicalBlockIdx = atomicAdd(&(block_index[0]), 1);
15             *concurrent_blocks = logicalBlockIdx;
16         }
17         __syncthreads();
18         if(logicalBlockIdx >= grid_size) {
19             break;
20         }
21         /*
22         ...
23         Kernel Code
24         ...
25         */
26     }
27     if(threadIdx.x == 0) {
28         atomicSub(&(block_index[smid + 1]), 1);
29     }
30 }

```

Fig. 5. The kernel transformation required for supporting persistent threads and preemption.

3.5 Profiling and Pruning the Parameter Space

We use offline profiling of kernels in isolation to help estimate the remaining execution time of kernels in a multi-tenant environment, which is used in our allocation policy (Sect. 3.6); we want to minimize the maximum remaining execution time of the kernels to maximize STP. Once the transformation in Sect. 3.4 is applied to the kernel, we can use the number of CTAs as the control knob for the amount of resources allocated to it. After these data have been obtained (we will discuss the results of our profiling in Sect. 4.3), we sort the configuration points based on the number of CTAs and then prune the space such that the execution times of the remaining set of configurations monotonically decrease. Once the pruning is done, we store the remaining set of configurations in an array in the host-side service to be later retrieved by the allocation algorithm. Equation (1) shows how we use the profiling data for estimation of the remaining execution time of the kernel in a multi-tenant environment:

$$T_m^c = T_i^c \times \frac{TB_t - TB_e}{TB_t} \quad (1)$$

In (1), T_m^c is the remaining execution time of the kernel in multi-tenant environment when it is running with c CTAs, T_i^c is its execution time in isolation

when it has c CTAs, TB_t is the total number of its thread blocks, and TB_e is the number of thread blocks it has executed so far.

3.6 Sharing Policy

Our policy is a greedy method, in which the service starts at the point where all kernels have minimum resources, i.e., one CTA per kernel (line 2 in Algorithm 1.) The algorithm then descendingly sorts the kernels based on their estimated remaining execution times and initializes a variable, *marked*, to indicate the kernels whose resource allocation is determined (lines 3–4). It then iteratively advances to the next configuration point for the first unmarked kernel (lines 6–7) until all the kernels are marked. If this kernel currently has all the resources it can use, it is marked (line 15) and the loop proceeds to the next iteration. Note that due to sorting the kernels, this operation minimizes the maximum remaining execution time among all kernels. If the new configuration fits the device, i.e., the resources required for it do not exceed those available on the GPU, the kernels are re-sorted and the loop proceeds to the next iteration (lines 8–9). Otherwise, the operation is rolled back and the kernel is marked (lines 11–12). It continues until no kernel can have more resources allocated to it. Note that these steps take place only in the host-side service and the final result is communicated to the applications, i.e., the incremental resource allocation is only in computation, we do not incrementally add to the CTAs a kernel runs with. Algorithm 1 shows the pseudo-code for this policy. If no feasible configuration exists, we simply launch all kernels with one CTA. In this case, kernels will be queued at the block scheduler on the GPU and will start execution once resources become available. The complexity of this algorithm is linear with respect to the

Algorithm 1. Greedy resource allocation algorithm

```

1: procedure ALLOCATERESOURCES(KernelsList)
2:   Allocate the minimum resources to each kernel
3:   Descendingly sort kernels based on their estimated remaining execution times
4:   marked  $\leftarrow$  0
5:   while marked  $\leq$  KernelsList.size() do
6:     if KernelsList[marked].nextConfig() then
7:       KernelsList[marked].advanceToNextConfig()
8:       if New configurations fit the device then
9:         Re-sort the KernelsList from marked onwards
10:      else
11:        KernelsList[marked].rollBackToPrevConfig()
12:        marked++
13:      end if
14:    else
15:      marked++
16:    end if
17:  end while
18: end procedure

```

number of configurations of all kernels, i.e., $O(\sum_{k \in K} C_k)$ where C_k is the number of configurations for kernel k and K is the set of all kernels that want to run on the GPU.

It must be noted that any other policy can be easily plugged into our proposed system without affecting any of the parts related to the mechanisms necessary for supporting simultaneous multi-tenancy. The only requirement is that the policy needs to take a list of the kernels and their profiling data as the input and determine configurations for each kernel.

3.7 Example Scenario

An example scenario of two applications running kernels on the GPU concurrently is presented in Fig. 6. At step ①, Application #1 contacts the host-side service requesting to run kernel A on the GPU. The service runs the resource allocation algorithm and responds to Application #1 to run kernel A with five CTAs per SM, and consequently Application #1 launches A with the specified number of CTAs.

Then at step ②, Application #2 sends a message to the host-side service and requests to run kernel B on the device. The service queries Application #1 about the progress of kernel A at step ③, and Application #1 responds with the number of thread blocks that A has executed. Based on that information, the service runs the allocation algorithm and sends new launch parameters, i.e., three CTAs for each of A and B, to Applications #1 and #2 at step ④. At this step, Application #1 compares the new number of CTAs with what A is running with currently. Since the new number is smaller than the old one, preemption has to happen. Therefore, Application #1 writes the new value to the `max_blocks` location mentioned in Sect. 3.4. At the next iteration of the internal loop of A, the last two CTAs at each SM will preempt and make room for CTAs of kernel B. In parallel with this operation, Application #2 launches B with three CTAs per SM. Thread blocks of B will wait at the block scheduler on the GPU until resources become available to start execution.

Once kernel B grabs its last thread block at step ⑤, Application #2 notifies the host-side service. This triggers running the allocation algorithm again. The service sends new number of CTAs to Application #1, and since the new number is greater than the old one, new CTAs have to be launched. At step ⑥, Application #1 does so in a separate stream (CTAs inside the dotted rectangle), in order for them to run in parallel with the previous instance of the kernel. Kernel A grabs its last thread block at step ⑦ and Application #1 notifies the service of this event. Finally, Application #1 is finished and the GPU is empty.

3.8 Limitations

A limitation of our work is that the host-side service logically shares a single SM among all kernels, and then extrapolates that configuration to all other

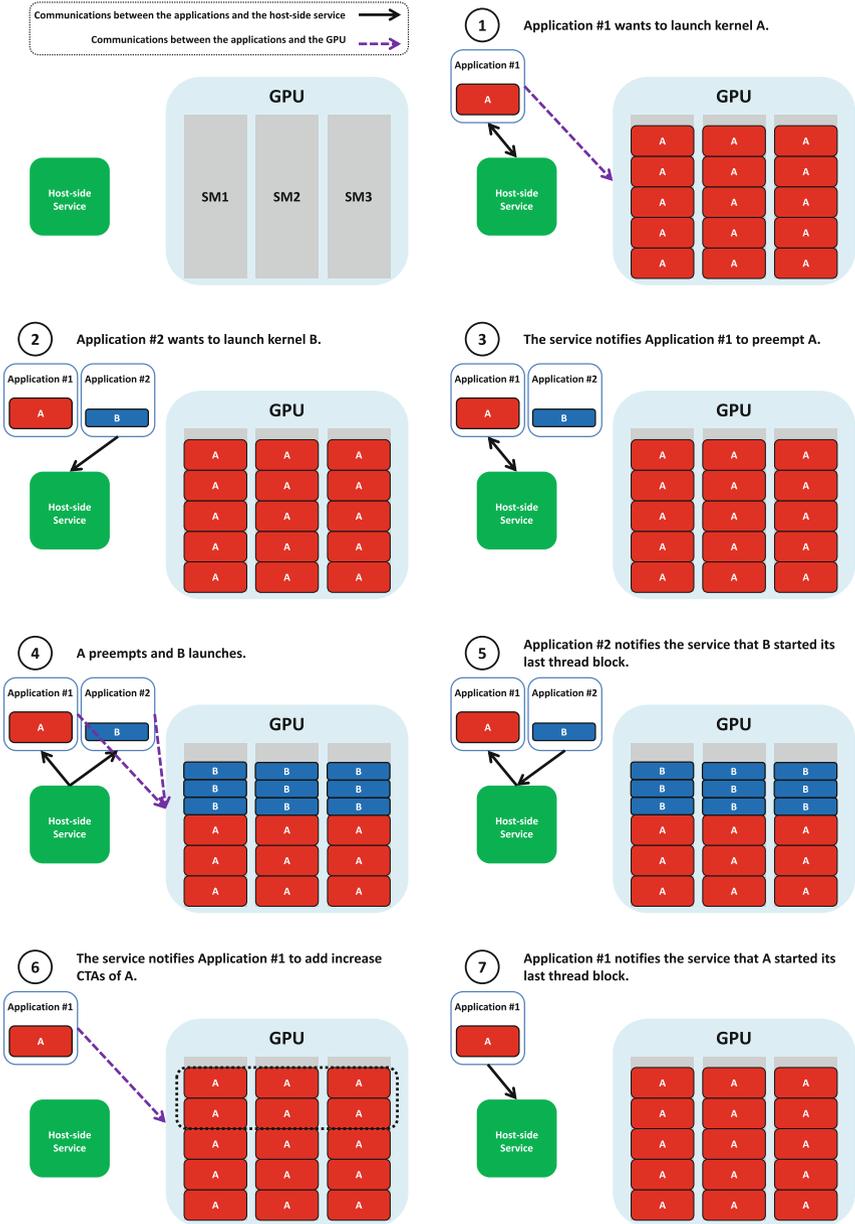


Fig. 6. An example scenario of two applications running kernels on the GPU simultaneously.

SMs, although there might be kernels that benefit from having the entire SM to themselves due to their intensive usage of cache. We plan to address this issue in our future work. One way to alleviate this shortcoming is to launch placeholder kernels from the host-side service to occupy a subset of SMs and force the GPU block scheduler to assign thread blocks to the free SMs. The overhead associated with this approach is expected to be minimal, since the placeholder kernels are only required for the short period of time between arriving a kernel launch request at the host-side service and the launch of the kernel at the GPU.

There are other limitations imposed by the choice of platform. One is due to the use of MPS in our system. NVIDIA GPUs do not support dynamic parallelism [13] while running MPS. Therefore, our system does not support running the kernels that use this feature. The other, perhaps more important limitation, is that NVIDIA GPUs prior to the Volta [20] do not support running kernels with separate virtual address spaces, which creates security concerns. As discussed earlier in the paper, adding a software address translation layer [25] can solve this problem by isolating the address spaces of different applications.

4 Evaluation

In this section, we first describe the platform for our experiments. Then we discuss the characteristics of the benchmark kernels we used. After that, the effect of the transformation in Sect. 3.4 on the performance of the benchmark kernels is evaluated. Finally, we present results for multi-kernel evaluations.

4.1 Platform

The machine we used for the experiments has an Intel Xeon E5-2640 CPU, and the experiments are conducted on an NVIDIA Tesla K40c GPU. The OS is Ubuntu 16.04, and NVIDIA driver version 375.26 and CUDA 8.0 were used to compile and run benchmarks. Table 2 shows the specifications of the GPU card accounted for while making decisions about feasibility of kernel configurations in the host-side service.

Table 2. NVIDIA Tesla K40c specifications.

Resource	Value
Threads per SM	2048
Registers per SM	65536
Shared Memory per SM	48 KB
Warps per SM	64
Thread Blocks per SM	16

4.2 Benchmark Kernels

Our goal was to have a mixture of kernels from various areas with different behaviors and requirements. To this end, we picked seven benchmark kernels, binomialOptions, FDTD3d, lavaMD, MD5hash, nbody, particlefilter, and tpacf, from CUDA SDK samples [16], Rodinia [5], SHOC [7], and Parboil [26] benchmark suites, for our evaluations. Figure 7 shows the behavior of these benchmarks that are representative of a variety of kernels. The values on the axes are from a scale of 0–10 and are obtained from NVIDIA profiler. The figure shows that some kernels are compute-intensive (MD5Hash, lavaMD), some demonstrate intensive use of memory and cache (FDTD3d), and some have a mixture of requirements (binomialOptions, nbody, particlefilter, tpacf).

Table 3 summarizes the characteristics of these benchmark kernels, after undergoing the transformations to support adaptive simultaneous multi-tenancy explained in Sect. 3.4. The abbreviations in front of kernels’ names are used in multi-kernel evaluation figures.

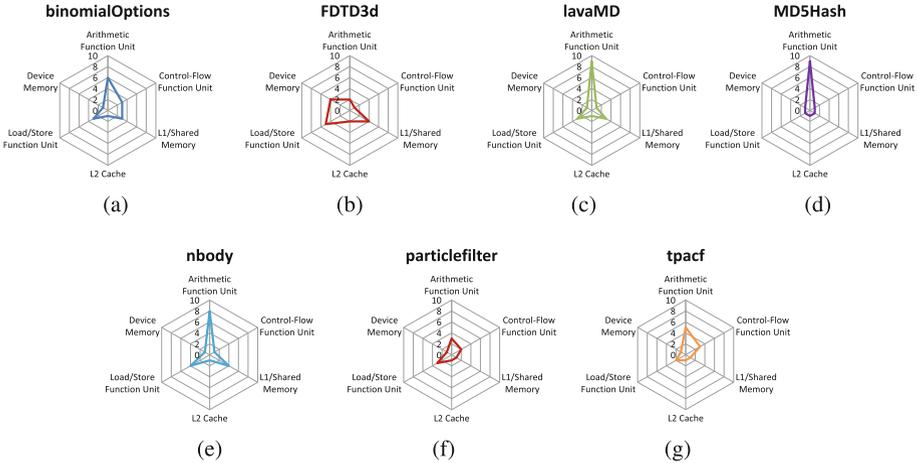


Fig. 7. Utilization of various resource types in (a) binomialOptions, (b) FDTD3d, (c) lavaMD, (d) MD5Hash, (e) nbody, (f) particlefilter, and (g) tpacf kernels.

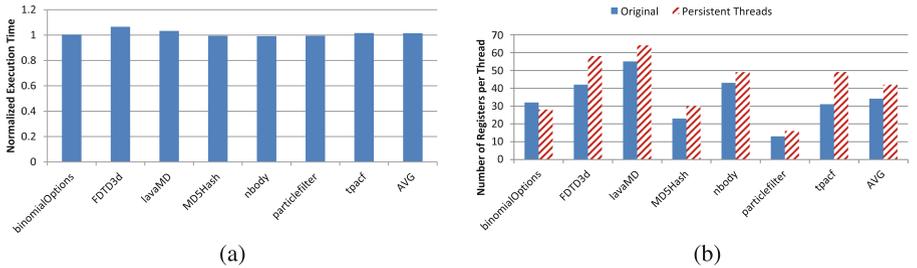
4.3 Single Kernel Performance

Figure 8a shows the normalized execution time of the transformed kernels with respect to the original code. As the figure illustrates, applying the transformation to the kernels has a negligible impact of 1.7% on the average performance of all kernels. However, it increases the register and shared memory usage of the kernels due to introducing additional variables. The register usage is increased by 23%, as shown in Fig. 8b. Increasing the number of registers per thread might result in

Table 3. Benchmark kernels characteristics.

Kernel	Thread blocks	Threads / Block	Registers / Thread	Shared memory/Block (B)	Execution time (ms)	ISU
binomialOptions (BO) [16]	1024	128	28	524	5.476	73.6
FDTD3d (FD) [16]	288	512	58	3848	8.821	27.5
lavaMD (LM) [5]	512	128	64	7208	8.958	87.3
MD5Hash (MD) [7]	25432	384	30	8	71.475	97.4
nbody (NB) [16]	128	256	49	8208	39.155	89.1
particlefilter (PF) [5]	512	128	16	8	43.105	48.6
tpacf (TP) [26]	201	256	49	13320	11.23	34.8

fewer CTAs fitting on the GPU. Nevertheless, as shown in Fig. 9, increasing the number of CTAs has marginal gain and after some point, the performance does not improve dramatically by increasing the number of CTAs. Anyway, a possible solution for reducing the register overhead of the proposed code transformations is to restrict the compiler to compile kernels with fewer registers. This incurs some performance overhead to kernels, but based on our observations accepting a 2% performance overhead results in the elimination of register usage overhead. The reason that we did not take this into consideration is that the same could be applied to the original kernels for register reduction. Thus, we picked the best-performing register configuration for both original and transformed kernels. We did not include the shared memory usage figures, because all kernels need a fixed eight bytes additional shared memory to store the logical block index and CTA IDs in SMs required for the persistent threads transformation.

**Fig. 8.** (a) Performance and (b) register usage of benchmark kernels under persistent threads transformation.

As stated in Sect. 3.4, applying persistent threads transformation to kernels allows to control the allocated resources by running them with the desired number of CTAs. Figure 9 shows the performance of transformed kernels for varying numbers of CTAs per SM. We use these data as input for our greedy allocation algorithm. Most of the time, there is a direct trade-off between the numbers

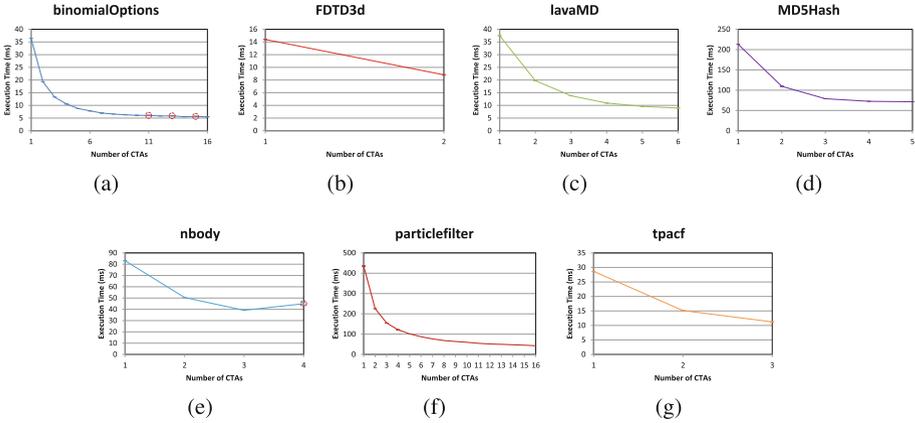


Fig. 9. Performance of (a) binomialOptions, (b) FDTD3d, (c) lavaMD, (d) MD5Hash, (e) nbody, (f) particlefilter, and (g) tpcf kernels with different numbers of CTAs.

of CTAs of a kernel and its performance. However, there are some exceptions for binomialOptions and nbody that are distinguished with red dotted circles. We omit these points from the decision-making process in the allocation algorithm, since they do not offer any beneficial trade-off. In other words, there exists another point in the space that uses less resources but delivers higher performance.

4.4 Multi-kernel Performance

In this section, we report two metrics that are common for measuring the performance of multi-program workloads [8]: (i) system throughput (STP), and (ii) average normalized turnaround time (ANTT) for kernels. We use the time it takes for all kernels to finish, i.e., the completion time of the last kernel that finishes minus the start time of the first kernel that begins, as an indication for STP. ANTT is the ratio of the time it takes for a kernel to finish in a multi-tenant environment and the time it takes for the same kernel to finish in isolation. Unfortunately, we cannot report ISU for multi-kernel experiments since NVIDIA profiler does not report it when MPS is running.

We also report results for two systems: (i) a system in which there is no host-side service, but persistent thread transformation is applied to kernels (PT), and (ii) our proposed adaptive simultaneous multi-tenant system (SiM).

We repeated our experiments five times for every ordering of the kernels (e.g., for combination of BO+FD, five times when BO arrived at the service first, five times when FD was the first, and for SiM only, five times when both arrived at almost the same time such that their effects were aggregated), and report the average of the results.

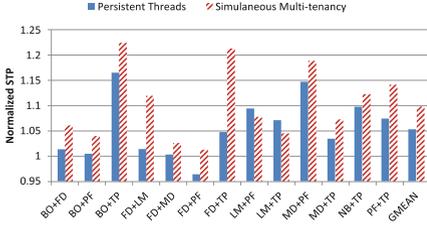


Fig. 10. Normalized STP under different combinations of kernels including at least one kernel with low ISU.

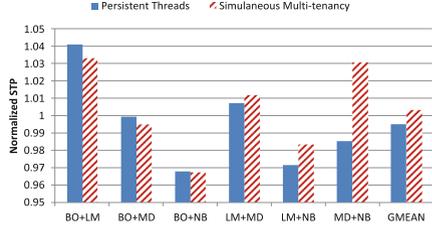


Fig. 11. Normalized STP under different combinations of kernels with high ISUs.

Kernels with Low ISU. We refer to kernels with an ISU of less than 50% as low ISU kernels. During the execution of such kernels, execution units are idles for more than half of the cycles due to various reasons, including synchronization, data request, execution dependency, busy pipeline, etc. It is expected that co-scheduling these kernels with another kernel results in higher STP, since the idle cycles can be taken advantage of. Figure 10 demonstrates the normalized STP for kernel combinations that include at least one kernel with low ISU. On average, PT improves STP by 5.3% with respect to the sequential execution of kernels. This improvement is due to the alleviation of the head-of-line blocking at the GPU block scheduler explained in Sect. 2.4. Nevertheless, addressing this issue alone is not sufficient to realize the potential STP improvement created by the underutilization of resources. To this end, by tuning the resources allocated to each kernel, SiM increases STP by 9.8%.

Not all kernel pairs experience similar improvements in STP. The higher the ISU of one of the kernels is, the less opportunities there are for STP improvement. This is evident in FD+MD and MD+TP pairs, because MD kernel has an ISU of 97.4%. This basically means that MD alone can utilize the device very well. There are other factors that impact the achieved STP improvement as well, such as the resource requirement of the kernels, the execution time of a single thread block of the kernels, and non-optimal allocation of resources by our greedy algorithm. The first one shows itself in high STPs achieved when running BO+TP and FD+TP pairs, since these kernels utilize different units on the device. The other two, however, explain the low STP improvement for LM+TP. In this kernel pair, whenever LM arrives first, there is no room for TP to run any thread blocks until the first round of thread blocks of LM are completed and preemption can take place. This takes long enough to offset a large enough fraction of the improvement achieved by the co-run of the rest of the thread blocks of the two kernels to cause lower improvements than simply running the two kernels with persistent thread transformation. In addition, the launch parameters determined by the greedy algorithm for the two kernels do not result in the best possible output. This highlights the need for a more accurate and sophisticated allocation policy, which is part of our future work.

The gains in STP come at the expense of 49.2% increase in ANTT. We must note that improving STP and keeping ANTT low are at odds with each

other, and the goal of our allocation policy is to maximize STP. If ANTT is an important factor in the system, other allocation algorithms can replace our greedy algorithm without affecting other parts of the system. Besides, we did not define priorities for kernels in our work. The works that target improving ANTT, do so for high priority kernels since it is impossible to improve this metric for all kernels in the system.

Kernels with High ISU. The STP improvement is not significant when all the kernels running on the GPU can utilize it well enough when executed in isolation. Figure 11 shows the normalized STP for pairs of kernels that both have high ISUs (i.e., greater than 50%). In these cases, the overheads of preemption and multiple launches, as well as cache thrashing, result in a negligible STP improvement of 0.3% in SiM. PT even imposes a 0.5% overhead on STP. The other downside of running high-ISU kernels together is a large increase in ANTT (78.7%).

These observations mean that adaptive simultaneous multi-tenancy is more effective when individual kernels are not highly optimized to have high ISUs. The positive side is that this also means that without putting extra effort for optimizing kernels, a higher STP can be achieved by merely running multiple kernels together.

5 Related Work

5.1 Persistent Threads

Gupta et al. studied the different use cases of persistent threads for a single kernel [11]. Recently, in independent works, Chen et al. [6] and Wu et al. [30] proposed taking advantage of persistent threads for supporting preemption. EffiSha [6] only supports execution of one kernel at any given time on the GPU. FLEP [30], on the other hand, has limited support for executing two kernels on the GPU at the same time, only in the case that one of the kernels is small enough to entirely fit on the GPU. Supporting preemption in these works helps favoring the high priority kernels over the low priority ones, but it does not solve the underutilization of resources on the device.

5.2 Software-Based Multi-tasking on GPUs

Pai et al. proposed Elastic Kernels [21] and showed that sharing the GPU among multiple kernels improves utilization. They artificially fuse multiple kernels together to form a super-kernel in a single GPU context. There are other works that adopt a similar approach [10, 14, 29]. This allows for concurrent execution of kernels, but such a scenario is impractical in the real world, since merging the kernels from different clients into a single kernel at run-time is impossible. We avoid this limitation by using a host-side service and taking advantage of MPS. Furthermore, the focus of this paper and [21] is on different parts of the solution. We propose a system to solve the problem of multi-tenancy while [21] proposes various policies that can be employed in our host-side service.

Preemptive Kernel Model [3] proposed by Basaran and Kang slices the kernel into smaller grids which in turn allows for sharing the GPU among multiple kernels. Several other works also rely on kernel slicing [12,32]. This approach incurs the overhead of multiple launches that cannot be avoided even if we do not need to preempt the kernel at all. By taking advantage of persistent threads model, our approach eliminates the unnecessary launch overhead introduced by kernel slicing.

5.3 Hardware-Based Multi-tasking on GPUs

The NVIDIA Volta architecture supports static simultaneous multi-tenancy [20]. In other words, it is possible to divide the GPU into multiple smaller virtual GPUs. There are also works that introduce hardware extensions to support preemption or multi-programming. Tanasic et al. [27] proposed context switching and draining by supporting preemption in hardware. Park et al. [22] extended this work by identifying idempotent kernels to faster preempt the running kernel by flushing the SMs.

Adriaens et al proposed spatial multi-tasking [1]. In this approach, each SM is entirely allocated to one kernel. Wang et al. propose partial context switching (PCS) [28], which is similar to our approach in that it only preempts a portion of SMs. Nevertheless, they use different allocation policies and hardware support is necessary for its implementation. Xu et al. [31] propose a software-hardware mechanism that similarly shares an SM among multiple kernels. Park et al. propose GPU Maestro [23] that based on performance predictions switches between spatial multi-tasking and PCS at run-time.

These works address the issues regarding supporting multi-tenancy on GPUs, but because of the required hardware modifications are not applicable to the existing devices. Our goal in this paper is to design a system that can take advantage of commodity GPUs.

6 Conclusion

In this paper, we identify the challenges of using GPUs in a multi-tenant environment. We propose adaptive simultaneous multi-tenancy for GPUs to overcome these challenges. Our approach comprises a host-side service that makes decisions about the kernel launch parameters and when the kernels should preempt. We also provide an API to facilitate using the system for programmers and allowing kernels to dynamically adapt resource usage at runtime and require minimal kernel modifications. Evaluation of our prototype system on NVIDIA K40c GPUs show that, on average, system throughput is improved by 9.8% for combinations of kernels that include at least one low-utilization kernel. This improvement is achieved at the cost of 49.2% increase in average normalized turn around time. Combinations of high-utilization kernels do not benefit from our system. Our observations indicate that using adaptive simultaneous multi-tenancy allows programmers to avoid highly optimizing their kernels to have high ISUs by providing higher STP for concurrent execution of low-utilization kernels.

As future work, we plan to investigate the effectiveness of more sophisticated mapping mechanisms and allocation policies to support asymmetric mapping of thread blocks to SMs. This way, cache-intensive kernels can benefit from having the entire L1 cache to themselves. We also plan to exploit various scheduling algorithms for dispatching thread blocks to SMs, to improve locality and reduce destructive interference in L2 cache.

Acknowledgments. This work is supported in part by the National Science Foundation (CCF-1335443) and equipment donations from NVIDIA.

References

1. Adriaens, J.T., Compton, K., Kim, N.S., Schulte, M.J.: The case for GPGPU spatial multitasking. In: Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture. HPCA 2012, pp. 1–12. IEEE Computer Society, Washington, DC (2012). <http://dx.doi.org/10.1109/HPCA.2012.6168946>
2. Amazon Web Services: Elastic GPUS (2017). <https://aws.amazon.com/ec2/Elastic-GPUs/>
3. Basaran, C., Kang, K.D.: Supporting preemptive task executions and memory copies in GPGPUS. In: Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems. ECRTS 2012, pp. 287–296. IEEE Computer Society, Washington, DC (2012). <http://dx.doi.org/10.1109/ECRTS.2012.15>
4. Chase, J.S., Anderson, D.C., Thakar, P.N., Vahdat, A.M., Doyle, R.P.: Managing energy and server resources in hosting centers. In: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles. SOSP 2001, pp. 103–116. ACM, New York (2001). <http://doi.acm.org/10.1145/502034.502045>
5. Che, S., Sheaffer, J.W., Boyer, M., Szafaryn, L.G., Wang, L., Skadron, K.: A characterization of the Rodinia benchmark suite with comparison to contemporary cmp workloads. In: Proceedings of the IEEE International Symposium on Workload Characterization (IISWC 2010), pp. 1–11. IISWC 2010. IEEE Computer Society, Washington, DC (2010). <http://dx.doi.org/10.1109/IISWC.2010.5650274>
6. Chen, G., Zhao, Y., Shen, X., Zhou, H.: Effisha: a software framework for enabling efficient preemptive scheduling of GPU. In: Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 3–16. PPOPP 2017, ACM, New York (2017). <http://doi.acm.org/10.1145/3018743.3018748>
7. Danalis, A., et al.: The scalable heterogeneous computing (shoc) benchmark suite. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, pp. 63–74. GPGPU-3, ACM, New York (2010). <http://doi.acm.org/10.1145/1735688.1735702>
8. Eyerman, S., Eeckhout, L.: System-level performance metrics for multiprogram workloads. *IEEE Micro* **28**(3), 42–53 (2008)
9. Google: Google cloud platforms (2017). <https://cloud.google.com/gpu/>
10. Gregg, C., Dorn, J., Hazelwood, K., Skadron, K.: Fine-grained Resource Sharing for Concurrent GPGPU Kernels. In: Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism. HotPar 2012, p. 10. USENIX Association, Berkeley, (2012). <http://dl.acm.org/citation.cfm?id=2342788.2342798>

11. Gupta, K., Stuart, J.A., Owens, J.D.: A study of persistent threads style GPU programming for GPGPU workloads. In: 2012 Innovative Parallel Computing (InPar), pp. 1–14, May 2012
12. Jiao, Q., Lu, M., Huynh, H.P., Mitra, T.: Improving GPGPU energy-efficiency through concurrent kernel execution and DVFs. In: Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization. CGO 2015, pp. 1–11. IEEE Computer Society, Washington, DC (2015). <http://dl.acm.org/citation.cfm?id=2738600.2738602>
13. Jones, S.: Introduction to dynamic parallelism. In: Nvidia GPU Technology Conference. NVIDIA (2012). <http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0338-GTC2012-CUDA-Programming-Model.pdf>
14. Liang, Y., Huynh, H.P., Rupnow, K., Goh, R.S.M., Chen, D.: Efficient gpu spatial-temporal multitasking. IEEE Trans. Parall. Distrib. Syst. **26**(3), 748–760 (2015)
15. Microsoft: Microsoft azure (2016). <https://azure.microsoft.com/en-us/blog/azure-n-series-general-availability-on-december-1/>
16. Nvidia: CUDA programming guide (2008). <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>
17. Nvidia: Next generation CUDA computer architecture Kepler GK110 (2012)
18. NVIDIA: Multi-process service (2015). https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf
19. NVIDIA: Pascal architecture whitepaper, June 2015. <http://www.nvidia.com/object/pascal-architecture-whitepaper.html>
20. NVIDIA: Volta architecture whitepaper, June 2015. <http://www.nvidia.com/object/volta-architecture-whitepaper.html>
21. Pai, S., Thazhuthaveetil, M.J., Govindarajan, R.: Improving GPGPU concurrency with elastic kernels. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 407–418. ASPLOS 2013, ACM, New York (2013). <http://doi.acm.org/10.1145/2451116.2451160>
22. Park, J.J.K., Park, Y., Mahlke, S.: Chimera: collaborative preemption for multitasking on a shared GPU. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS 2015, pp. 593–606. ACM, New York (2015). <http://doi.acm.org/10.1145/2694344.2694346>
23. Park, J.J.K., Park, Y., Mahlke, S.: Dynamic resource management for efficient utilization of multitasking GPUs. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS 2017, pp. 527–540. ACM, New York (2017). <http://doi.acm.org/10.1145/3037697.3037707>
24. Randles, M., Lamb, D., Taleb-Bendiab, A.: A comparative study into distributed load balancing algorithms for cloud computing. In: 2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops, pp. 551–556, April 2010
25. Shahar, S., Bergman, S., Silberstein, M.: Activepointers: a case for software address translation on GPUs. In: Proceedings of the 43rd International Symposium on Computer Architecture. ISCA 2016, pp. 596–608. IEEE Press, Piscataway (2016). <https://doi.org/10.1109/ISCA.2016.58>
26. Stratton, J.A., et al.: Parboil: a revised benchmark suite for scientific and commercial throughput computing. Technical report (2012). <https://scholar.google.com/scholar?oi=bibs&hl=en&cluster=14097255143770688510>

27. Tanasic, I., Gelado, I., Cabezas, J., Ramirez, A., Navarro, N., Valero, M.: Enabling preemptive multiprogramming on GPUs. In: Proceeding of the 41st Annual International Symposium on Computer Architecture, pp. 193–204. ISCA 2014, IEEE Press, Piscataway (2014). <http://dl.acm.org/citation.cfm?id=2665671.2665702>
28. Wang, Z., Yang, J., Melhem, R., Childers, B., Zhang, Y., Guo, M.: Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 358–369, March 2016
29. Wu, B., Chen, G., Li, D., Shen, X., Vetter, J.: Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In: Proceedings of the 29th ACM on International Conference on Supercomputing. ICS 2015, pp. 119–130. ACM, New York (2015). <http://doi.acm.org/10.1145/2751205.2751213>
30. Wu, B., Liu, X., Zhou, X., Jiang, C.: 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, pp. 483–496. ASPLOS 2017, ACM, New York (2017). <http://doi.acm.org/10.1145/3037697.3037742>
31. Xu, Q., Jeon, H., Kim, K., Ro, W.W., Annavaram, M.: 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), pp. 230–242, June 2016
32. Zhong, J., He, B.: Kernelet: high-throughput gpu kernel executions with dynamic slicing and scheduling. *IEEE Trans. Parallel Distrib. Syst.* **25**(6), 1522–1532 (2014). <https://doi.org/10.1109/TPDS.2013.257>