Cooperative GPGPU Scheduling for Consolidating Server Workloads*

Yusuke SUZUKI†*, Nonmember, Hiroshi YAMADA††, Member, Shinpei KATO†††, Nonmember, and Kenji KONO†, Member

SUMMARY Graphics processing units (GPUs) have become an attractive platform for general-purpose computing (GPGPU) in various domains. Making GPUs a time-multiplexing resource is a key to consolidating GPGPU applications (apps) in multi-tenant cloud platforms. However, advanced GPGPU apps pose a new challenge for consolidation. Such highly functional GPGPU apps, referred to as GPGPU apps, can easily monopolize a shared GPU and starve collocated GPGPU apps. This paper presents GLoop, which is a software runtime that enables us to consolidate GPGPU apps including GPGPU eaters. GLoop offers an event-driven programming model, which allows GLoop-based apps to inherit the GPU eaters’ high functionality while proportionally scheduling them on a shared GPU in an isolated manner. We implemented a prototype of GLoop and ported eight GPU eaters on it. The experimental results demonstrate that our prototype successfully schedules the consolidated GPGPU apps on the basis of its scheduling policy and isolates resources among them.

key words: GPGPU, cloud computing, operating systems

1. Introduction

Graphics processing units (GPUs) have become an attractive platform for a broad range of application (app) domains. General-purpose computing on GPUs (GPGPU) is a widely used technique used to accelerate target apps by harnessing the massive data parallel computing capacities of GPUs. GPGPU is applicable to various apps such as deep learning [1]–[4], scientific simulations [5], [6], file systems [7], [8], complex control systems [9], [10], autonomous vehicles [11], [12], and server apps including network systems [13], [14], web servers [15], key-value stores [16] and databases [17]–[20].

Making GPUs a time-multiplexing resource is a key requirement for hosting GPGPU apps in multi-tenant cloud platforms whose resources are shared among multiple customers. Consolidating GPGPU apps on a GPU brings several benefits. For example, since the load of cloud services varies with diurnal patterns and spikes [21], GPGPU server consolidation can improve GPU utilization by assigning the idle-time of the GPU to not only other GPGPU servers but also compute-intensive GPGPU apps including those of deep learning. The motivation for consolidation is strengthened by the fact that GPUs are continuously scaling up. NVIDIA has reported that the number of streaming processors and size of memory in Tesla M40 GPUs are 1.6 times and 2.0 times larger than those of the previous generation [22].

Recent high-functioning GPGPU apps pose a new challenge for multi-tenant consolidation. Such GPGPU apps, referred to as GPGPU eaters, typically launch a long- or infinite-running GPU kernel and monopolize a shared GPU, easily starving other GPGPU apps collocated on it. For example, GPUs- [7] and GPUnet-based [23] apps poll completions of I/O requests on the GPU. Scientific apps [5], [6] exclusively use GPUs to compute their simulations. Existing GPU resource managers, including GPU command-based schedulers [24]–[26], novel GPU kernel launchers [27], [28], and thread block schedulers [29], [30], fail to schedule GPU eaters appropriately since GPU eaters do not provide scheduling points such as kernel launches or thread block completion; thus, a hosted GPU eater may monopolize the GPU. Other techniques, such as context funneling [31], [32] and persistent threads [33], effectively schedule GPU eaters but fail to isolate GPGPU apps; thus, a hosted GPGPU app may access and modify the memory of other GPGPU apps, which is not suitable for multi-tenant cloud platforms.

The current hardware preemption is not a perfect solution to consolidate GPU eaters in multi-tenant cloud platforms. The recent NVIDIA Pascal GPUs [22] have mechanisms to preempt long-running GPU kernels. However, as recent literature [30] reported, no publicly available information shows the availability of software-level preemption control. Because of the lack of software control, we cannot apply a proportional share policy to GPU kernels that is based on various indicators such as customer payment. Therefore, if a user starts many GPU contexts, this user can simply occupy the GPU’s computing resources. In addition, if a GPU eater polls I/O completion, GPU cycles are wasted because the hardware-level scheduler assigns timeslices to it without recognizing the polling.

GLoop, presented in this paper, is a runtime system to consolidate GPGPU apps including GPU eaters. GLoop controls GPU eaters’ scheduling and their isolated
execution, both of which are mandatory for multi-tenant cloud platforms. The key insight behind GLoop is the use of an event-driven programming model, which is widely used in cloud apps driven by I/O events such as network packet arrival [34]. The event-driven programming model allows GPU eaters to be consolidated without wasting GPU time, while GLoop schedules them on a shared GPU in an isolated manner. In addition to consolidating I/O-driven GPU eaters, GLoop allows compute-intensive GPU eaters written in the event-driven programming model to exploit the idle-time of an under-utilized GPU. The GLoop runtime also schedules GPGPU apps on the basis of a proportional share scheduling policy.

We prototyped GLoop on an unmodified proprietary NVIDIA driver and CUDA SDK. In addition, we ported eight GPU eaters on GLoop: TPACF, LavaMD, MUMmerGPU, Hybridsort, Grep, Approximate Image Matching, Echo Server, and Matrix Multiplication Server. The GLoop runtime and all the apps described in this paper are released as open-source software1. We perform an experimental evaluation of our prototype demonstrating that our GLoop-based apps are comparable in performance to the original versions and that GLoop successfully consolidates and schedules them on the basis of our scheduling policy. We also show that GLoop’s consolidation significantly contributes to improving GPU utilization in two consolidation scenarios: GPU server consolidation and GPU idle-time exploitation.

This paper substantially extends our previous work [35] as follows:

• We add description on the GLoop runtime design related to CUDA APIs to clearly show how our design decision meets one of the design goals: no modification of proprietary GPGPU stacks. We describe a point of building GLoop with the current CUDA.

• We port the prototype of GLoop, used in the previous work, onto the newer software stack to show the applicability of GLoop. We upgrade the CUDA SDK from 7.5 to 9.0 which has the upgraded GPU driver, runtime, and NVCC CUDA compiler. We can successfully port it to the target software stack with minor engineering efforts.

• To demonstrate the effectiveness of GLoop on modern machines, we evaluate our new prototype on a superior machine compared to the previous one. We conduct experiments with a SSD and two types of GPUs: the Pascal Tesla P100 GPU and the Kepler Tesla K40c GPU. From the experiments, we show a new performance trend of GLoop on the hardware.

Porting GLoop to another software/hardware stack reveals the following GLoop’s characteristics: portability and applicability, each of which is an important factor for evaluating the design of system software. In this research, we give some insights on these abilities from our porting experience and experiments. For the portability, we evaluate how efforts we need to pay to port the old GLoop prototype used in the previous research [35], to the newer CUDA SDK. Although the porting involves the adjustment of parameters for GPUs including the number of thread blocks launched by apps, significant changes are not required in both apps and GLoop runtime; our porting completes successfully with trivial engineering efforts. For the applicability, we conduct experiments using the ported GLoop prototype on the newer software/hardware stacks and analyze the difference between the previous and current results. Our experimental results reveal that GLoop shows its effectiveness even with the different performance characteristics. In the case of throttle which is a microbenchmark used for evaluating the performance characteristics of GLoop, GLoop on Pascal GPU shows better results compared to one on Kepler GPU. The result of grep_text shows that GLoop successfully benefits from the fast I/O of the SSD while our previous work evaluates GLoop with HDD.

The rest of this paper is organized as follows. Section 2 briefly introduces the background of our work and summarizes related work. Section 3 describes a programming model offered by GLoop. Section 4 and 5 shows the GLoop runtime design and its details. Section 6 and Sect. 7 reports a prototype of GLoop and our porting experience of eight real-world GPGPU apps. Section 8 shows experimental results with the prototype, and Sect. 9 finally concludes this paper.

2. Motivation

2.1 GPU Eaters

Numerous researchers have studied how GPGPU apps can be made to become more highly functional to fully utilize GPU capacities [5]–[7], [23], [33]. Such GPU apps launch long- or infinite-running GPU kernels. For example, GPUfs [7] exposes file systems APIs to a GPU program to efficiently execute a GPGPU app involving file operations and facilitate its development. GPUnet [23] also provides a socket abstraction and APIs suitable for GPU processing. The persistent threads model [33] launches a maximum-sized grid on a GPU. In this model, thread blocks (TBs) continuously fetch GPU tasks from work queues to execute them without costly kernel launches. The model is effective for irregular parallel apps such as ray traversal [36]. GPGPU apps performing scientific simulations, sorts, and bioinformatics, typically launch a long-running GPU kernel [5], [6].

Unfortunately, these app designs implicitly assume that only one GPGPU app at a time runs on a GPU. Consolidating these types of app, called GPU eaters, on a shared GPU poses an interesting challenge: How can we effectively share a GPU among GPU eaters in an isolated manner? GPUfs- and GPUnet-based apps poll I/O completion to avoid costly GPU kernel launches so that the other GPU kernels can do nothing until the running kernel finishes. We cannot execute

---

1https://github.com/CPFL/gloop
two or more persistent thread apps concurrently since the TBs in one app are long- or infinite-running over GPU tasks. The GPU kernels of scientific simulations typically monopolize a GPU for seconds, minutes, or even hours. Advanced GPU resource managers [31–33], described in detail in the next section, can exert scheduling control over GPU eaters at the expense of resource isolation between hosted GPU eaters; GPU eaters are executed in the same GPU context.

Current hardware support for GPU kernel preemption is not a perfect solution to GPU eater consolidation. Although hardware preemption is supported in NVIDIA Pascal GPUs [22], no publicly available information shows the availability of software-level preemption control [30]. We have no control over GPU eaters to schedule them flexibly. For example, cloud vendors cannot proportionally assign GPU resources to a customer’s application on the basis of their payments. Moreover, since GPU hardware is not aware of whether an active GPU eater is polling for I/O completion, the hardware-level scheduler blindly assigns timeslices to the polling GPU eater, leading to wasting GPU time [37]. With control over scheduling GPU eaters, the GPU resource managers would be able to intercept I/O requests of GPU eaters and dispatch other hosted GPGPU apps instead of polling-based blocks.

A naive software approach to scheduling GPU eaters in an isolated manner is to divide the GPU eater’s kernels into smaller GPU kernels by splitting the GPU computations and finishing all the running TBs. This approach, called kernel splitting, offers scheduling points to typical GPU schedulers that use GPU kernel launches as scheduling points. However, it degrades the performance of the GPU kernels and incurs non-trivial development costs. Since each GPU kernel has GPU hardware resources, including tremendous numbers of registers and shared memory, their allocations/releases in launches/exists are time-consuming, making the latency of the scheduling points high even if a sequence of split GPU kernels does not need to be descheduled. Moreover, it is difficult to divide a GPU kernel into chunks of an appropriate size to offer timely scheduling opportunities because we cannot exactly know the execution time for each part of the kernel in the development phase. In addition, efficient coordination of multiple kernels requires overlapping communications and computations, which involves significant development effort such as fine-tuned pipelining between CPU sends, CPU-GPU data transfers, and GPU kernel invocations.

2.2 Related Work

Existing GPU resource managers aim at sharing GPU resources among GPU apps, but these resource managers are of limited use when GPU eaters are executed concurrently on a GPU. TimeGraph [24] and GPVvm [26] offer a command-based scheduler that issues GPU commands received from processes or virtual machines (VMs) on the basis of their scheduling policies. Disengaged scheduler [25] schedules GPU commands with a sophisticated probabilistic model. Even with these command-based schedulers, a GPU eater can still monopolize a GPU by issuing a command for polling or launching a long-running kernel. To avoid this situation, we have to redesign such apps to issue numerous GPU commands instead of one polling command or split their GPU kernels. In addition, some GPU schedulers are difficult to run on proprietary software stacks due to the requirement of GPU driver modifications.

Gdev [27] multiplexes a GPU device at the operating system (OS) level. It has a GPU scheduler whose scheduling points are GPU kernel launches. If a GPU kernel has been running for a long time, the Gdev scheduler assigns long slices of time to other GPU app kernels to achieve fair GPU utilization. PTech [28], where a GPGPU app is designed as a data flow graph that consists of GPU kernel modules, schedules GPU kernels when they are launched. These kernel-based schedulers suffer from the same problem as the command-based ones.

The elastic kernel [38] transforms physical TBs into logical TBs and dispatches them to physical resources. It schedules GPU kernels by adjusting the number and size of logical TBs spawned in one launch. EffiSha [29] dispatches logical TBs on the basis of the scheduler’s decisions. These approaches use the ends of logical TBs as scheduling points. Therefore, even with them, a GPU eater with long-running TBs can still monopolize a shared GPU.

GPUpIO [37] achieves I/O-driven preemption in GPU apps by instrumenting code with save and restore procedures. Instead of waiting for I/O completions by polling, an inserted procedure saves the state of the executing TB and finishes it. When the I/O operation is completed, GPUpIO executes another GPU kernel that restores the saved state of the TB. While GPUpIO is effective for I/O polling-based GPU eaters, long-running kernels such as scientific simulations and persistent threads can still monopolize a shared GPU.

GPUShae [39] schedules GPU kernels by controlling the number of executed TBs. When the TBs are dispatched, each of them checks whether the execution time of the kernel has exceeded a specified period. If so, the TB does not start its actual code and finishes early. However, GPUShae fails to achieve fine-grained scheduling for polling-based GPU eaters or GPU kernels whose TB execution is too long because the TBs cannot perform periodic checks.

Multi-process service [31] (MPS), which is also known as context funneling [32], concurrently executes multiple GPU kernels on a GPU. MPS redirects all the streams of the running GPGPU apps to one GPU context in a service process. Thus, the redirected GPU kernels simultaneously run within one GPU context. FLEP [30] is similar to EffiSha, but combines MPS with a TB scheduler to offer spatial multitasking. The persistent threads approach [33] can schedule GPU kernels requested from GPGPU apps. GPU apps add their GPU tasks to a work queue, and active TBs execute GPU tasks in the work queue. Since all GPU tasks in these approaches run in the same GPU virtual address space, a GPU request from a buggy or malicious GPU app
Table 1 Comparison of GLoop with previous work.

<table>
<thead>
<tr>
<th></th>
<th>Consolidating GPU eaters</th>
<th>Resource Isolation</th>
<th>Proprietary GPGPU stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware preemption [22]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Kernel splitting</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TimeGraph [24]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GPUnode [26]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Disengaged scheduling [25]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Gdev [27]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Flink [28]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Elastic kernels [38]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>EffShake [29]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GPUmalloc [37]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GPUShare [39]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MPS [31]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Context funneling [32]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FLEP [30]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Volta MPS [40]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Persistent threads [33]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GLoop</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

2.3 GLoop

This paper presents GLoop, which allows us to host multiple GPU eaters on a single GPU in multi-tenant cloud platforms. Table 1 briefly compares GLoop with previous work. GLoop has three goals, as follows.

- **Consolidates GPU eaters efficiently**: GLoop concurrently executes GPGPU apps on a shared GPU. It dispatches them according to a scheduling policy and lowers scheduling point latency.
- **Provides GPU resource isolation**: GLoop isolates GPU kernel execution. A malicious or buggy GPGPU app cannot destroy GLoop or other GPGPU apps’ contexts.
- **Does not modify proprietary GPGPU stacks**: GLoop works on top of proprietary GPGPU device drivers, GPGPU libraries, and GPU hardware. The current prototype runs on an unmodified NVIDIA device driver and CUDA SDK 9.0.

Our design is based on the discrete (off-chip) GPU model that is widely used for its intensive computational abilities. Discrete GPUs are connected on the PCI express bus (PCIe) and are composed of a huge number of cores tightly coupled with a specialized high-bandwidth device memory. The discrete GPU design delivers greater computational performance and higher energy efficiency [23]. Recent research has leveraged discrete GPUs to create high-performance, scalable, and more energy efficient cloud apps [23], [41], [42]. Although GLoop is portable onto integrated (on-chip) GPUs such as Intel GPUs and AMD Kaveri [43], our optimization techniques would not work as well since integrated GPUs have different performance characteristics from those of discrete GPUs. In this case, alternative mechanisms are needed.

3. GLoop Programming Model

An important role of GLoop is to offer low-latency scheduling points to GPU kernels without sacrificing isolation. GLoop provides an *event-driven programming model* to GPGPU apps by borrowing the idea from Node.js [44]. We treat all kernel operations in this model as *events*, except for the GPU computation. The events include file I/O, network I/O, and GPU yields. GLoop-based apps register their own *callbacks* of events of interest, and GLoop dispatches a callback when the corresponding event has completed.

Our programming model has three important features. First, we can develop highly functioning GPGPU kernels. GLoop exposes APIs for event requests such as file I/O, network I/O, and GPU yields; thus, like GPUfs and GPUUnet, the development of GLoop-based apps does not involve laborious efforts such as pipelining or asynchronous data copies. Second, we can set scheduling points without splitting GPU kernels or finishing all the running TBs. GLoop uses not only GPU kernel launches but also event requests as scheduling points. In addition, the latency of scheduling points decreases since event requests do not involve kernel launches. Third, GLoop’s event request APIs are non-blocking. This style of programming allows GLoop to avoid polling-based block behavior and dispatch other hosted GPGPU apps. Namely, I/O operations and GPU computations can be overlapped.

The event-driven programming model is known to be effective in server apps because they are driven by external I/O requests such as network packet arrival [34]. We adopted this model for GPGPU servers in which the GPU apps are driven by events. In addition, this model offers a chance for compute-intensive GPU eaters to exploit the idle resources of an under-utilized GPU. Since the utilization of server GPGPU apps varies, GLoop-based compute-intensive GPU eaters can exploit the idle resources of the GPU. GLoop efficiently schedules compute-intensive GPU eaters with server GPU eaters.

Note that the recent hardware GPU preemption will be complementary to the GLoop technique once preemption technology becomes widespread. When GLoop fails to offer appropriate scheduling points, we can fall back on the GPU preemption to prevent GPU eaters from monopolizing GPUs. The current prototype of GLoop kills GPU apps that monopolize a shared GPU.
3.1 Event-Driven Programming

In our programming model, GPU kernels of GLoop-based apps are composed of callbacks, each of which is associated with events such as an I/O operation (e.g. file read and write) and GPU yield. When an event is completed, GLoop executes the corresponding callback and unregisters it. The GLoop-based app does not finish until all the registered callbacks have been consumed. A typical GLoop-based app starts and then registers a callback. When the callback is invoked after the corresponding event has completed, it registers a new callback in the running callback.

Figure 1 shows an example program where TBs read a file. The function, doRead, reads a specified file up to the specified bytes, size. We define a callback function, callback, in the C++ lambda style at line 5. This callback processes read data and then calls doRead() again (lines 5–8). The program executes fs::read(..., callback), which requests a file read from the host and registers the passed callback. GLoop executes the registered callback once the requested read has completed. Since doRead() is called in the callback, the running callback registers itself again via fs::read() (line 9). These steps are repeated until the read size becomes equal to the specified size.

In addition, GLoop allows us to register continuation callbacks for GPU yields. This means that we can insert scheduling points into the middle of a TB execution by posting the continuation as a callback. GLoop supports postTask(), whose argument is the next callback.

3.2 Coalesced APIs

GPU execution is based on the hierarchical parallelism of hardware. The GPU kernel is composed of TBs. A TB consists of grouped threads called warps, where the GPU executes threads in lock-step, and this poses the problem of inefficiency when the threads follow divergent paths.

GLoop adopts coalesced API calls, inspired by GPUs [7] and GPUTnet [23]. GLoop specifically forces all the threads in a TB to call the same APIs with the same arguments at the same point in the app code. The TB-level approach is reasonable because the GPU offers efficient sharing and synchronization primitives for TBs. This means that TBs have a coarse-grained parallelism: all the threads in each TB perform a single task [23], [38]. In addition, managing the GPU kernel at the thread or warp level involves management of much larger metadata per GPU kernel as GPU kernels typically consist of tremendous numbers of threads.

3.3 Programming Model Adoption

GLoop programming is a continuation-passing style where each callback represents the next control state. Although we need to modify the app code to consolidate GLoop apps, this is not a complicated task from our experience; we successfully implemented eight GPU eaters, as described in Sect. 6.

Most GPGPU apps can be made GLoop-based with little effort since there is no need to modify the core logic of the apps. If GPU kernels are short-running, what we should do is to launch the kernels through GLoop. Even if the TBs are short-running, there is a concern that numerous short-running TBs can occupy a shared GPU. Since GLoop treats TB completion as scheduling points, as described in Sect. 5.1, no scheduling point insertion is needed in such GPU kernels.

If GPU apps have long- or infinite-running TBs, the key to adopting the GLoop programming model is to identify where to insert scheduling points in the target GPU app code. Developers have to pay attention to two types of kernel: (1) I/O-intensive and (2) compute-intensive due to a long-running TBs. Regarding the first type, they do not need to insert scheduling points explicitly, since I/O requests are used as scheduling points. Regarding the second, they need to insert a continuation callback into a long-running code path. For example, they set a continuation callback per iteration in a long loop. Therefore, the adoption of our programming model does not involve drastic changes to the app logic.

Inserting callbacks at appropriate points of the code would not impose a huge burden on developers. As a rule of thumb, they should insert continuation callbacks at every location where execution is long. GLoop decreases scheduling point latency by not performing GPU kernel launches in every continuation callbacks. Instead, GLoop only switches kernels if necessary; GLoop dynamically decides to perform context switching at the current scheduling point based on the execution time. If the size of the input for processing becomes larger and thus the kernel execution time becomes longer, GLoop will perform context switching more times.

This means that GLoop imposes performance penalties that are small even if a tremendous number of continuation callbacks is set to the app. Automatic insertion of continuation callbacks to appropriate code points is a challenge since it is inherently difficult to obtain such information by statically analyzing the source code. Investigation of this issue is beyond the scope of the paper.

4. GLoop Runtime

GLoop runtime offers an event-driven execution
Device loops interact with the host loop via RPC slots that are on the host-device shared memory. A device loop creates a callback slot in the device memory when requesting an event to the host loop and initializes an RPC slot. The device loop saves the next callback (a lambda function) into the callback slot while writing RPC arguments to the RPC slot. It then issues the RPC by pushing an operation code to the RPC slot. The device loop starts polling the RPC slots for event completion and the signal slot to check whether the suspend signal has arrived.

For example, device loops request the host loop to read a file and associate a callback with the file read’s completion. The host loop reads the file, transfers the read data to the device memory through GPU direct memory access (DMA) engines and writes the notification of the read completion to the RPC slot. After the device loop, which is polling the RPC slot, detects the data read by the host loop, it writes the data back to the specified buffer and invokes an associated callback.

4.2 Suspend and Resume

Suspending and resuming operations involve all the GLoop components. The host loop requests a scheduling token from the gloop scheduler to start or resume the GPU kernel. The scheduler suspends the currently running device loops by pushing the suspend signal into the signal slot after the device loops have exhausted their timeslice.

The device loops check for the arrival of the suspend signal before invoking a new callback. When detecting a suspend signal, the device loops stop callback invocation and finish execution of the GPU kernel. After the corresponding host loop acknowledges that the device loops have been suspended, the host loop releases the scheduling token to the gloop scheduler and requests it again. The gloop scheduler selects the next host loop to run and passes it the scheduling token. The host loop resumes the GPU kernel by launching a GPU kernel that reconstructs device loops.

Checking the signal slot in device loops is a time-consuming task since the host-device shared memory is accessed through the PCIe bus. This latency in accessing DRAM makes scheduling checks quite slow. This cost is relatively high in postTask() that offers lightweight scheduling points. To reduce this cost, we periodically check the
signal slot. The device loops monitor GPU clocks to check for exhaustion of their timeslices (10 ms). The device loops in our prototype access the signal slot every quarter of a timeslice (e.g., 2.5 ms).

4.3 An Example of Runtime Execution

We demonstrate how the GLoop runtime runs on our example 1. First, a device loop launches the GPU kernel issuing doRead(). If offset is less than size, doRead() computes sizeToRead and invokes fs::read() API with the callback. At this point, the device loop stores the given callback into the GPU memory, starts RPC for reading a file, and finishes the current doRead() execution. The device loop polls the RPC slot since the device loop has the pending slot for the RPC. On the other hand, the host loop receives the RPC from the device loop, performs asynchronous I/O on the host, transfers the data to GPU, and notifies the device loop of the completion of reading the file. The device loop detects the completion of the RPC, loads the callback from the GPU memory, and invokes it with the transferred data. The callback then starts running with the data and does processing.

When the device loop receives a suspension request from gloop scheduler while polling the RPC slot, the device loop stops polling and finishes the GPU kernel. After the host loop receives the scheduling token from gloop scheduler, the corresponding device loop resumes its GPU kernel by starting the callback stored in the GPU memory.

5. Design Details

Efficiently consolidating GPU eaters raises three design challenges: (1) how do we control GPU kernels spawning numerous TBs, (2) how can we lower scheduling point latency as much as possible, (3) how do we schedule GLoop-based apps in a fair-share manner, (4) how do we build the GLoop runtime on the CUDA-based software stack? To address these challenges, we integrate efficient schemes with the GLoop runtime.

5.1 Thread Block Control

The number of TBs inside a GPU kernel is critical for scheduling. Suspending all the running TBs to de-schedule the GPU kernel is a time-consuming task if the GPU kernel consists of numerous TBs. To stop the GPU kernel, a host loop has to wait until the GPU hardware has dispatched all the TBs to the SMs. In addition, a GPU kernel generating numerous short-lived TBs is difficult to schedule since its code path is too short to insert continuation callbacks. For example, “MUMmerGPU” and “LavaMD” from Rodinia [47] respectively generate up to 65,535 and 125,000 short-lived TBs for a single kernel launch.

To address these problems, we introduce a thin TB scheduler, inspired by the idea of Elastic kernels [38] and EffiSha [29]. Our TB scheduler, which is a software mechanism running inside the device, puts all the TBs in its queue and only executes the same number of TBs as concurrently runnable TBs on the SMs. The running physical TBs fetch logical TBs from the queue and execute them. The changes to the app code in order to use this scheduler are trivial: using GLoop’s API to retrieve logical TB information instead of physical ones (e.g. blockIdx and blockDim). The TB scheduler allows us to complete the suspension of a GPU kernel by only stopping physical TBs. When fetching logical TBs, the physical TBs check for the arrival of a suspend signal. We note that the appropriate number of physical TBs relies on resource usage by the TB. Although we manually specify this number for each GPU kernel on our prototype, an appropriate number can be automatically calculated by using the CUDA occupancy calculator API [48].

5.2 Scheduling Point Optimization

To lower the latency of the scheduling points as much as possible, we leverage GPU shared memory regions whose access is faster but whose size is smaller than that of regular device memory regions. We place the control state of the GLoop runtime in a shared memory region.

In addition, GLoop manages two callback slots on the shared memory region. We observe that a GPU kernel typically waits for only one event, which means that it only uses two callback slots. One is for the currently running callback, and the other is for pushing the next callback. We therefore place two callback slots that are currently used in the shared memory region, which leads to quick invoking and saving of callbacks. Although the cost of context switches is logically increased slightly since we need to store the slots from shared memory into the regular memory region, this overhead is negligible.

This optimization can degrade the performance of a GPU kernel if it fully utilizes GPU shared memory. GLoop’s shared memory use sometimes results in fewer runnable TBs. GLoop can switch the optimization on and off. Developers can thus choose the appropriate GLoop mode for their GPU kernels.

GLoop supports the postTaskIfNecessary() API in order to further lower the latency of scheduling points. This API, used for long loops in the program, allows the GPU kernel to perform lightweight scheduling checks per iteration. This saves a callback in its argument and returns true only when the device loop checks the suspend signal. Otherwise, it returns false without saving the callback, and the GPU kernel then performs the next iteration. Like postTask(), the API checks the GPU clock to efficiently poll the suspend signal slot.

5.3 Scheduling Policy

We integrate a scheduling policy into the gloop scheduler. An advantage of GLoop over existing GPU resource managers is that it can assign running states to hosted GPU
In designing the GLoop 5.4 CUDA API Scheduling consolidation. TBs wait for the events completion. For example, a GPU assigning on the basis of the runtime states. The thus assigns the runtime states of the GPU kernels such as focusing policies into the gloop scheduler. In particular, the main focus of this paper is that GLoop can schedule hosted GPU apps in a fine-grained manner, which is essential for multitenant cloud platforms.

Our scheduler proportionally dispatches GPU kernels in a work-conserving manner to fully utilize GPU resources. The scheduler is based on weighted fair queuing [49]. It prepares each user’s queue and assigns more GPU time to high priority users by weighting their queues. When a GPU app requests the launch of a GPU kernel, the gloop scheduler pushes its GPU kernel launching request into the app’s queue and sets the queue to active if it is inactive. Each queue has a virtual time that elapses during execution of the GPU kernels fetched from the queue. The gloop scheduler selects the active queue whose virtual time is shortest and passes the scheduling token to the host loop. The execution of the GPU app is controlled by GLoop’s suspend and resume mechanism explained in Sect. 4.2. When all the GPU kernels in a queue complete, the queue becomes inactive. To achieve a work-conserving scheduling, the gloop scheduler adjusts the virtual time of the queue that just becomes active. Specifically, the gloop scheduler resets the virtual time to the shortest virtual time among the active queues.

GLoop’s runtime intermediates event invocations and thus assigns the runtime states of the GPU kernels such as I/O waiting, and the gloop scheduler manages the scheduling token assignment on the basis of the runtime states. The gloop scheduler can deschedule GPU apps when all of the TBs wait for the events completion. For example, a GPU app is descheduled when all its TBs are waiting for newly incoming packets. This feature is effective in the context of consolidation.

5.4 CUDA API Scheduling

In designing the GLoop runtime, we take into account the invocation of the CUDA APIs that involve exclusive access to the underlying GPU. While a GPU kernel spawned from another CUDA context is running, some CUDA API call blocks the apps. For example, we cannot initialize a new CUDA context in CUDA 7.5 on Kepler GPUs during the other GPU kernel execution. In this case, the target CUDA context is never initialized until the running GPU kernel finishes. The GLoop runtime needs to suspend the GPU kernel to execute such CUDA APIs.

To address this issue, we reuse the scheduling token used for GPU kernel scheduling. Specifically, the apps acquire the scheduling token from the gloop scheduler to invoke a CUDA API of them. The current design conservatively forces the apps to try to get the token in calling all CUDA APIs. When an app has acquired the scheduling token for the CUDA API invocation, the gloop scheduler suspends the running app, namely stops the running GPU kernel. For example, an app initializes its new CUDA context after acquiring the scheduling token. At this time, the gloop scheduler has already suspended the GPU kernel execution.

5.5 Discussion

Large GPU memory transfers between the host and device can also monopolize GPUs. Memory transfers can occupy GPU DMA engines for a long time and block subsequent memory transfer requests. Previous studies [38], [50] suggest splitting large memory transfers into small chunks and scheduling split requests. While GLoop does not focus on memory transfers, these techniques can be integrated into it. If GLoop-based apps leverage shared memory, their developers need to pay attention to GPU kernel context switching, which clears the shared memory content. There are two ways of addressing this issue at callback boundaries: saving and restoring the content of the shared memory or reconstructing the content. Our ported apps using shared memory can take either way.

Although GLoop provides low-latency scheduling points, the tremendously large number of scheduling points affects overall performance. The scheduling point frequency depends on scheduling point places in the app code. We note that developers can adjust the trade-off between scheduling point frequency and performance penalty by taking into account that the latency is less than 2.26 µs in Kepler and 1.23 µs in Pascal, as shown in Sect. (1).

The GLoop architecture is portable to various resource-shared environments. In a container-based system, one possible setup is that the gloop scheduler runs on the host OS as a service process, the host loops in GPU apps use CPU slices assigned to their own containers, and their device loops run on the shared GPU. The gloop scheduler schedules the running GLoop-based apps in the containers. In a VM system where a GPU is virtualized [26], [51]–[53], the gloop scheduler is inside the hypervisor or privileged VM, and each loop of the GPU apps runs on virtualized CPUs and the GPU of their VMs.

While GPUs are getting better hardware preemption support, the other types of simple accelerators do not support preemptions (low-end GPUs, FPGAs etc.). The GLoop design can be applied to such accelerators to offer scheduling mechanism in software, and this enables sharing of accelerators in multi-programmed environments without adding complexity to the accelerators themselves.

GLoop fails to control the GPU kernels if users do not modify their GPU eaters appropriately for scheduling. We categorize such illegal GPU kernels into two types: (1) GPU kernels that do not offer GLoop’s scheduling points, and (2) GPU kernels that do not take a scheduling token from the gloop scheduler. GLoop cannot switch the GPU kernel execution of the former GPU eaters to another one due to no issues of scheduling points. A way to prevent the GPU monopolization of the GPU eaters is to kill their processes. To do so, we can use the kill command. A possible future
extension of our prototype is automatically killing them by monitoring the GPU usage of the GPU kernels and sending the termination signal to the processes whose GPU usage is over the threshold.

Also, GLoop cannot control GPU eaters that ignore a scheduling token from the gloop scheduler. For example, such a GPU eater can launch its GPU kernel even if another GPU kernel is running, thus causing GPU resource contention. We can force these types of the GPU eaters to follow the GLoop scheduling by hooking GPU kernel launches. Since the GPU apps launch the GPU kernel by ioctl() to a GPU device file or MMIO to a specific region of GPU, a kernel module can hook the launches by monitoring the ioctl() issues or trapping MMIO [54]. The recent literature [25] demonstrates handling GPU kernel launches. This scheduling token request is automatically conducted by the GLoop library.

6. Implementation

We implemented a prototype of GLoop on Linux kernel 4.10.0-37 with CUDA 9.0 for NVIDIA Kepler and Pascal GPUs [22], [55]. The current prototype is tailored to Linux container-based platforms, which means that GLoop-based apps in each container run on a shared GPU. Figure 3 overviews our prototype, which consists of a GLoop library and a gloop scheduler daemon.

We note that GLoop can be prototyped with the existing tool chains such as CUDA. We do not need to prepare special compilers or preprocessing tools for the implementation. This property can improve the GLoop portability that allows us to easily install the runtime and follow the updates of the underlying CUDA runtime.

(1) GLoop Library:

The GLoop library and apps are implemented using CUDA, and they are compiled in the NVIDIA CUDA Compiler (NVCC). GLoop-based apps are linked to the library to spawn the host and device loops. Each host loop communicates with the scheduler via the POSIX inter-process communication (IPC). The GLoop-based app invokes GPU kernels through the host loop. To execute GPU kernels, the host loop requests the scheduling token from the gloop scheduler by using the POSIX IPC. When the host loop acquires the token, it starts a GPU kernel that constructs or resumes the device loops on the device side and executes user-written GPU code on the top of them. If the kernel is suspended by the gloop scheduler, the host loop releases the scheduling token to the gloop scheduler and requests the scheduling token again to resume the suspended GPU kernel. This scheduling token request is automatically conducted by the GLoop library.

(2) GLoop Scheduler Daemon:

The scheduler runs in the host as a daemon and manages the scheduling token. The host loop in each GLoop-based app tries to obtain the scheduling token from the gloop scheduler to execute its GPU kernels. The gloop scheduler sends a suspend signal to the active device loop every timeslice if two or more host loops request the scheduling token. When receiving the suspend signal, the device loops save their state, stop themselves, and finish the GPU kernel. The corresponding host loop releases the token to the gloop scheduler. The gloop scheduler then selects the next host loop to run and sends it the scheduling token.

(3) Callback:

We use C++11 lambda supported in the recent NVCC to represent callbacks. C++11 lambda saves data necessary to resume the GPU kernel, i.e., an instruction pointer and captured context data. The NVCC automatically captures variables referred by the lambda and saves them as a lambda object. The device loops use it as the callback.

The PTX ISA, a virtual instruction set for NVIDIA GPUs, version 2.1 introduces the indirect call [56] that allows CUDA programs to use lambda functions stored in GPU memory. The NVCC automatically converts CUDA lambdas to C structures holding a pointer to a function and captured variables. Since CUDA lambdas are C structures, we can store and load lambdas in GPU memory. The NVCC emits indirect calls for this function pointer in calling a CUDA lambda. While our prototype uses indirect calls, GLoop can be implemented conceptually even without indirect calls by employing a compiler-level approach to collecting the types of lambdas and emitting a large switch statement in a call site.

(4) RPC Slots:

Due to the lack of atomic operations over the PCIe bus, the device loops and host loop poll their RPC slots and behave in a producer-consumer manner; synchronization is not required since neither will simultaneously read or write to the same slot. The host and device loops issue RPCs in two phases to ensure memory consistency of the shared RPC slots. First, a host or device loop writes RPC arguments and flushes them by issuing a memory fence. The loop then writes and flushes the one word operation code. In
checking the slot, the host and device loop bypass CPU and GPU caches, respectively. This protocol guarantees that the arguments are visible when the RPC operation code is de- 

The slot holds a callback associated with the given RPC completion. Our prototype stores serialized CUDA lambdas to the memory region allocated for the slots. When the RPC completes, the device loop loads the corresponding CUDA lambda from the slot and invoke it.

(5) Host and Device Event Loops:

The Algorithm 1 is the device loops’s pseudo code in our prototype. Each device loop in the TB invokes this Drain function that repeatedly drains populated callbacks from the program. If the suspension request arrives or there are no pending RPC requests, the device loops finish their execution. Otherwise, device loop polls RPC slots to check RPC completion. On the other hand, the Algorithm 2 is the host loop’s pseudo code in our prototype. The host loop acquires a scheduling token from the gloop scheduler, launches a kernel, and releases a scheduling token when the kernel

(2) LavaMD:

LavaMD from Rodinia [47] launches a single kernel composed of long-running TBs. TPACF calculates the distances between all pairs of astronomical bodies. GLoop splits the kernel to insert scheduling points; we inserted postTask() into the loop calculating the distances of pairs. Note that the original TPACF intensively uses shared memory. Since GLoop switches GPU kernels, the content of the shared memory must be saved and restored every time the kernel is switched. The GLoop version uses regular device memory called global memory instead of shared memory. For comparison, we integrated the same change into the original one in addition to the original TPACF.

(3) MUMmerGPU:

MUMmerGPU [58] from Rodinia [47] consists of two long-running kernels (mummergpuKernel and printKernel). The kernels have different features; the former generates 9,766 long-running TBs, and the latter spawns 65,535 short-lived

7. Case Studies

GLoop is applicable to various GPGPU apps. We ported eight GPU eaters with different features.

(1) TPACF:

This app from Parboil2 [57] launches a single kernel composed of long-running TBs. TPACF calculates the distances between all pairs of astronomical bodies. GLoop splits the kernel to insert scheduling points; we inserted postTask() into the loop calculating the distances of pairs. Note that the original TPACF intensively uses shared memory. Since GLoop switches GPU kernels, the content of the shared memory must be saved and restored every time the kernel is switched. The GLoop version uses regular device memory called global memory instead of shared memory. For comparison, we integrated the same change into the original one in addition to the original TPACF.

(2) LavaMD:

LavaMD from Rodinia [47] launches a single kernel that generates many (125,000 in our setting) short-lived TBs. Since the TBs are short-lived, the kernel cannot be split. Instead, GLoop schedules the logical TBs as described in Sect. 5.1. GLoop schedules logical TBs on 30 physical TBs. Every time the logical TBs finish, control returns to GLoop.

(3) MUMmerGPU:

MUMmerGPU [58] from Rodinia [47] consists of two long-running kernels (mummergpuKernel and printKernel). The kernels have different features; the former generates 9,766 long-running TBs, and the latter spawns 65,535 short-lived
TBs. We inserted `postTaskIfNecessary()` into the long loop of `mummergpuKernel`. To obtain more scheduling points, logical TBs are scheduled on 30 physical TBs in `mummergpuKernel`. Logical TBs in `printKernel` are scheduled on 60 physical TBs.

(4) Hybridsort:

Hybridsort [59] from Rodinia launches two kernels: bucket-sort and merge-sort kernels. The bucket-sort kernel spawns many short-lived TBs, while the merge-sort kernel generates long-running TBs whose numbers vary from 8 to 81,000. Scheduling points can be inserted using the same techniques as those that are described above. A point to note here is that the bucket-sort kernel is carefully implemented to make full use of the shared memory per SM. This implementation can lead to severely degraded performance if `GLoop` runtime uses a small amount of shared memory. We therefore disabled the use of shared memory in the `GLoop` runtime.

(5) Grep:

Grep from the GPUs project\(^\text{1}\) was ported to `GLoop`. `GLoop` grep demonstrates that `GLoop` can support POSIX-like file system APIs since grep reads and writes files stored in the file system of the host operating system. `GLoop` grep invokes `postTask()` in each word-search iteration. Every time a string match succeeds, a callback with the file write request is queued to output the result to a file.

(6) Approximate Image Matching:

This app (`img`) from GPUs project\(^\text{1}\) scans three databases, each of which has 390MB (25,000 images in total) for learning, and finds images similar to the ones given as queries (2,000 images: 32 MB in total). The original img uses `gmmap()` and `gunmap()` APIs in GPUs, which enable file caches to be placed in the GPU memory. This feature results in significant performance benefits because the same files are accessed repeatedly in `img`. Our current prototype of `GLoop` lacks this feature, which results in poorer performance than that of the original. However, this feature is orthogonal to our design of `GLoop` and can be incorporated into `GLoop`.

(7) Echo Server:

To demonstrate that `GLoop` can support socket-like APIs for networking, the echo server from the GPUNet project [23] was ported. `GLoop` provides TCP/IP networking APIs such as `accept()`, `recv()`, and `send()` although the GPUNet assumes RDMA for communication `GLoop` prepares bounce buffers in the GPU memory to enable DMA between a host and device loops. Every time the echo server invokes networking APIs, which provide scheduling opportunities.

(8) Matrix Multiplication Server:

This app (matmul server) from the GPUNet is a network server that multiplies two 256×256 matrices of floats in a tiling manner. Matmul server mimics a typical GPGPU server behavior in which the GPU consumes the transferred input and sends back the result, such as a face verification server [23]. The invocation of networking APIs provides scheduling opportunities, as is done in the echo server. In addition, every time a tile is calculated, `postTaskIfNecessary()` is invoked to incorporate more scheduling points.

8. Experiments

We conducted experiments to find answers to four questions: (1) how much overhead does `GLoop` incur, (2) how well does the `GLoop` app perform when multiple GPU apps are consolidated, (3) can we achieve performance isolation on `GLoop`, and (4) is `GLoop` effective in consolidation scenarios?

We evaluated our prototype on an MAX-XW-E5HG machine with two Xeon E5-2620 v4 2.10-GHz CPUs (each has eight cores), 64-GB of memory and one 726-GB SSD. We used two NVIDIA Tesla GPUs: K40c Kepler GPU with 12-GB GDDR5 memory and P100 Pascal GPU with 16GB HBM2 memory. The NVIDIA GPU driver version is 384.81. The disk performance reported by `hdparm` is 8855.98 and 337.46 MB/s for cached and disk reads, respectively.

The workload was executed eleven times, i.e., once to warm up and ten times to obtain results. The measurements reported below are average values of the ten executions.

8.1 Standalone Overhead

To find how much overhead `GLoop` incurred, we ran the apps described in Sect. 7 in a standalone manner and measured their execution times. We grouped our apps into two categories: GPU- and I/O-intensive. We discuss `GLoop`’s overhead based on these two groups.

We ran three versions of GPU apps: an unmodified one (vanilla), a `GLoop`-based one (gloop), and a split version where the original GPU kernels were split into multiple short kernels (kernel-split). We also ran `GLoop` versions without the shared memory optimization, which were post-fixed as -w/o-shared-slots.

1. Scheduling Point Latency:

To examine the scheduling point latency, we first measured the execution time for the microbenchmark called `throttle`. `Throttle` is a program that invokes `postTask()` one million times with one TB that consists of one thread. The left end of Fig. 4 shows the execution time. `GLoop` version is 3.69× faster than kernel-split in Kepler, because `GLoop` offers lightweight scheduling points that do not involve kernel launches. `GLoop` version with the shared memory optimization is 8.2% faster than gloop-w/o-shared-slots, since `throttle` frequently writes a callback in slots on the shared memory.

On Pascal GPUs, `GLoop` is more effective than on

\(^\text{1}\)https://github.com/gpufs/gpufs
kernel-split. The middle of Fig. 4 shows that GLoop version is 4.36× faster than kernel-split in Pascal. This is because time for GPU kernel launches is not changed while the GPU clock of P100 is faster than K40c. This result means that the GPU kernel launch are relatively costly on the Pascal GPU.

From the execution time and number of scheduling points, we estimated that the latency of a scheduling point, which does not include the scheduling algorithm or GPU kernel switch costs, is less than 2.26 µs in Kepler and 1.23 µs in Pascal, calculated by \( \frac{\text{Execution Time of GPU Kernel}}{1000000} \). We can hide this latency since GPU executes different warps when accessing GPU memory. In fact, the subsequent experiments revealed that the scheduling point latency of GLoop is hidden or amortized in app benchmarks.

(2) Compute-intensive Apps:

We measured the execution times for hybridsort, lavaMD, mummergpu, and tpacf. We did not split the GPU kernel of lavaMD or the printKernel of MUMmergpu because their TBs are short-lived. For lavaMD, we did not prepare a -w/o-shared-slots version because it does not use postTask(). For tpacf, we also prepared a vanilla-global version that uses the global memory based on the original one, as described in Sect. 7. While vanilla hybridsort works with CUDA 7.5 [35], it crashes with CUDA 9.0. Thus, we omitted the result from the figure. We divided the total execution time into five categories: CUDAInit/FIn, DataInit/FIn, IO, Copy, and Kernel. They correspond to the times for CUDA context initialization and finalization, constructing and destroying data, reading and writing files, transferring data between the host and device, and GPU kernel execution.

Figure 4 presents the results. GLoop's overhead is shown in the CUDAInit/FIn and Kernel categories, and is small (−0.2% − 7.3%). The overhead in the CUDAInit/FIn category comes from allocating additional GPU memory and threads for GLoop. The time is small (539 ms − 725 ms) compared to the Kernel category (2258 ms − 6506 ms). Since this is one time overhead during GPU app execution, it is amortized by executing GPU kernels for long time; such GPU kernels are our target app. The overhead in the Kernel category is small or negligible (−10.5% − 4.3%) in all the cases except for hybridsort. The other categories are similar in all cases. The kernel performance penalty of GLoop is 23.8% in hybridsort, compared to CUDA 7.5 hybridsort data reported in our previous work[35]. This is caused by the balance of the two GPU kernel executions. The bucket-sort kernel is faster in the non-shared mode because of its shared memory utilization. The merge-sort kernel performance, on the other hand, is better in the shared mode because it does not use shared memory. All the kernels in one app are currently compiled in either the shared or the non-shared mode due to limitations in the toolchain. We can extend GLoop to mitigate the overhead by changing this mode per GPU kernel.

GLoop outperforms kernel-split versions in almost all cases. It is 1.8× and 1.2× faster than the kernel-split of hybridsort and mummergpu, respectively. The kernel-split versions cause numerous kernel launches for scheduling, whereas GLoop does not involve such additional kernel launches. In addition, GLoop reduces callback saves and restores by using postTaskIfNecessary(). The execution time of the kernel-split version in tpacf is comparable to that of GLoop because the execution time of each kernel is sufficiently long to amortize the kernel launch overhead.

The kernel execution time on Pascal is faster than that on Kepler. The main reason of this speed up is for the number of SMs: Pascal P100 has 3.73× more SMs (56) than Kepler K40c (15). On Pascal, the execution time of GLoop is comparable to that of vanilla and is much faster than that of kernel-split in several benchmarks (2.07× in hybridsort and 1.02× in mummergpu).

We can also see an interesting performance trend that time for GPU kernel launches is almost the same among two GPUs, which means that the overhead of GPU kernel launches is bigger on P100 than Kepler. If the GPU kernel launch incurs the same overhead while GPUs become faster, its relatively worse overhead motivates developers to consolidate numerous computation into one GPU kernel to minimize the GPU kernel launches. The optimization of merging tiny GPU kernels that is employed in a recent research system[60] can be used to reduce GPU kernel launches.

(3) I/O-intensive Apps:

We measured the execution times of grep and img. We prepared GPUs- and GLoop-based versions labeled gpufs and gloop, and prepared a workload for comparison that pre-
allocates a large amount of GPU device memory to transfer all the datasets before starting the GPU kernel, called vanilla. The execution time measured just after the host buffer cache is cleared is postfixed as -bc. We tuned gpufs and vanilla in grep to gain further CUDA occupancy with our GPU. These tuned versions are postfixed as -tuned. We modified the source code of the downloaded GPUfs to run it on our GPU.

The results obtained for grep are in Fig. 5. The performance of gloop is comparable to the other grep implementations on K40c and P100. Gloop in K40c outperforms the untuned versions and is 7.7% slower than the vanilla-tuned and 4.3% faster than gpufs-tuned versions.

Clearing the buffer cache does not degrade gloop performance (−0.1% in K40c and 0.2% in P100). Since grep repeatedly reads the same set of files, no buffer cache misses occur except for the first read. The execution time of gpufs-bc and gpufs-tuned-bc on P100 is slower than that of gloop. This comes from the difference of I/O handling between them. Since P100 GPU core performance is better than K40c, I/O throughput becomes a relatively major factor in the grep execution. We believe that the gloop’s multi-threaded asynchronous I/O feature maximizes I/O throughput in grep. The GPUfs implementation executes only one I/O thread in the host-side and thus fails to fully utilize the disk bandwidth.

Figure 6 shows the results for img. Since img-gpufs and img-gpufs-bc occasionally crash with CUDA 9.0 SDK, we measure the average execution time of first eleven successful execution. The execution time for gloop is 3.88× and 5.68× longer than that for gpufs in K40c and P100. This is because gpufs can benefit from the GPUs’ GPU buffer cache: GPUfs builds its buffer cache in the GPU device memory, and thus, the cache works effectively as the workload repeatedly reads the same files.

We used another data set called img-simple to validate this expectation. Img-simple uses only one image as the query data, runs one TB, and never provides matches against dataset images. This avoids reading the same data from the file system multiple times and reduces the effect of the GPU buffer cache as much as possible.

The results in Fig. 7 indicate that the execution time of gloop is just 1.45× and 1.27× longer than that of gpufs in K40c and P100 respectively. The remaining overhead is caused by the additional data copies in the gloop version. While gpufs exposes mmap-like APIs that only cause one data copy from the host to the GPU buffer cache, gloop provides write/read like APIs that perform copies twice, from the host memory to the GPU bounce buffer, and from the bounce buffer to the GPU user buffer. This overhead can be eliminated by adding a gpufs-like buffer cache mechanism to the gloop runtime. This implementation just requires engineering effort but a description of the implementation of a buffer cache is beyond the scope of this paper.

Different from the grep case, the cold buffer cache degrades performance by 0.8% in K40c and 0.1% in P100 because img issues I/O requests more frequently than grep.

8.2 Performance at Scale

We concurrently ran multiple instances of gloop-based apps to find the performance penalty of gloop’s consoli-
The execution time of these benchmarks on K40c GPU.

1. Context Switching Latency:

To discern the context switching overhead, we ran throttle by varying the number of threads, timeslice lengths, and the number of TBs. Figure 8 plots the execution times for different numbers of threads. In the single instance case, the increase in threads from 1 to 1024 lengthens the execution time by 13.6%. This is because the scheduling points require thread synchronization. The tendency in the case of eight instances is almost the same as that in the single instance (14.3% longer execution time from 1 to 1024). The slight overhead results from thread synchronization done in context switching.

Figure 8 plots the results for varied timeslice lengths. Due to the optimization of GLoop to avoid polling on the PCIe bus, as described in Sect. 4.2, the actual timeslice consumed slightly differs from the specified value. GLoop performs 2636 context switches on average per app in the two instances with 10000 µs timeslices in eleven runs. Thus, throttle in the two instances performs 239.6 context switches for each execution on average. When the timeslices are too short, context switches are dominant in the execution times. The execution time with a 100 µs timeslice is 5.8x longer than that with a 10000 µs timeslice for eight instances, where the execution time with the 100 µs timeslice is 42.5x longer than that of the one instance. The longer timeslice mitigates the context switching overhead. The increase in execution time for the 10000 µs timeslice is linear from one to eight instances (7.88x).

Figure 8 shows the execution time for throttle with different number of TBs. The execution time does not increase linearly up to 240 since K40c GPU can execute 240 TBs of throttle concurrently. However, in the eight instance case, the execution time with 240 TBs is 20.3% longer than that with 1 TB. This time increase comes from the limitation of GLoop’s suspend mechanism. The TB scheduler stops the running TBs asynchronously; it checks the suspend signal only every time slice and thus some GPU cores used by suspended TBs become idle until all the TBs are suspended. Suspending more running TBs causes more idle cores in the GPU kernel suspension, leading to waste computing resource of a GPU.

2. Apps:

Figure 9 presents the results obtained for tpacf and grep. The x-axis in the figures represents the number of launched apps, and the y-axis represents the execution time. GLoop schedules the apps in a fair-share manner, and the standard deviation for the results is at most 2.5%.

The figure indicates that the execution time for tpacf apps increases in proportion to the number of instances. The execution time for eight instances is slightly better than eight times the standalone’s execution time because of the short I/O time in tpacf. From Fig. 4, tpacf performs file I/O, which can be overlapped with execution of the other tpacf kernels.

The execution times for eight grep instances exceed eight times the standalone’s execution time (8.43x) as a result of disk I/O contention. The result in the next section validates this finding: the execution time for grep with seven throttles is 8.28x longer than that for the standalone since throttle does not issue any I/O requests. The remaining slowdown stems from the overhead imposed by scheduling the running instances.

8.3 Performance Isolation

We demonstrated that GLoop isolates performance among GPU apps. We controlled the GPU utilization of consolidated apps by using our proportional scheduler and measured the GPU utilization of each app. We launched one GPU app instance. We used tpacf and grep as GPGPU apps. We also ran one, three, and seven instances of throttle together.

First, we ran all the apps with the same utilization assignment. Figure 10 plots the GPU utilization. The x-axis represents the elapsed time, and the y-axis is the GPU utilization of the apps over 200 ms. The figure reveals that GLoop achieves performance isolation in all cases. The apps share one GPU and the computation resources are fairly divided. When running two, four, and eight instances, their GPU utilizations correspond to 50%, 25%, and 12.5%.

Figure 11 shows the execution times for each instance.
The execution time for eight instances for *tpacf* is $3.71 \times$ longer than that for the two instances. The increase in the execution time is not linear since the short I/O time in *tpacf* is constant in all cases. The execution time for *grep* in the eight instances is $4.02 \times$ longer than in two instances. This results from the overhead for scheduling multiple GPU apps.

Next we changed the resources assigned to the GPU apps. We assigned 66% of the utilization to a target app (*tpacf* or *grep*) while co-running throttle instances shared the GPU with one another. Figure 12 plots the results, which reveal that **GLoop** successfully assigns a target app the weighted GPU utilization and the other throttles share the remaining resources. The execution time shown in Fig. 13 slightly increases with the number of apps due to the accumulated overhead of the scheduler.

8.4 Consolidation Scenarios

To confirm the effectiveness of consolidating **GLoop**’s GPU apps, we devised two scenarios: **GPU Server Consolidation** and **GPU Idle-time Exploitation**. The **GPU Server Consolidation** is a situation where under-utilized GPU servers are consolidated into one GPU, while the **GPU Idle-time Exploitation** is where a compute-intensive app exploits the idle time of an under-utilized GPU.

(1) **GPU Server Consolidation:**

To demonstrate that **GLoop** successfully consolidates under-utilized GPU servers on a single GPU, we configured the apps as follows. We first ran an under-utilized GPU *matmul* server (*server1*) on a GPU whose utilization was roughly 20%. Then we gradually launched two additional under-utilized GPU servers (*server2* and *server3*) on the same GPU. When a single server was running, **GLoop** assigned it 100% of GPU utilization for the polling of device loops. To clearly demonstrate the effectiveness of **GLoop**’s consolidation, we launched a low priority throttle to drain the remaining utilization.

Figure 14 shows the stacked GPU utilization per 200 ms. The x-axis indicates the stacked GPU utilization of the three under-utilized GPU servers, and the y-axis plots the time series. While the new servers are being launched, **GLoop** successfully maintains the GPU utilization of the running servers at 20%. The figure also plots that the resource utilization spikes when *server2* and *server3* start (at the points of 7.5 and 17.5 s). This is because the CUDA initialization (CUDAInit/Fin in Sect. 8.1) takes 500 ms and thus GPU utilization is temporarily occupied. We note that it was 200 ms in 361.42 GPU driver. We guess that the internal initialization procedure is changed.

(2) **GPU Idle-time Exploitation:**

We also demonstrated that **GLoop** effectively assigns idle resources to the compute-intensive app while maintaining the performance of low utilized GPU servers. We ran one under-utilized GPU server (*server*) and then sequentially launched *tpacf* (*tpacf1*, *tpacf2*, and *tpacf3*).

The results are plotted in Fig. 15. When we launch a *tpacf* instance, the *tpacf* first occupies the GPU for its initialization. After that, while the server process runs under the assigned resources, *tpacf* utilizes the rest of the GPU resources. **GLoop** successfully assigns idle GPU utilization to *tpacf* instances while the resource utilization of the server is preserved.

8.5 Hardware Preemption

Finally, we describe an anecdotal situation showing that our software-level preemption can be more effective than hardware preemption. Pascal’s compute preemption offers instruction-level granularity preemption. The preemption mechanism saves/restores context information on the GPU kernels to/from GPU DRAM. Since the context informa-
tion includes thousands of registers’ values and large shared memory contents, the context switching could cause high latency. On the other hand, GLoop allows developers to insert scheduling points at appropriate places where the size of the context information becomes small. For example, we do not need to save register values and shared memory content as context information at a scheduling point when a thread block finishes. This implies our approach can achieve efficient context switches.

To validate the above assumption, we run LavaMD benchmark on two Pascal GPUs, GTX 1080 and Tesla P100 Pascal GPUs. In each trial, we launch multiple instances of vanilla and GLoop-based LavaMD (1 to 6) and measure their execution time.

Figure 16 shows the average execution time and standard deviation. The standalone performance shows that GLoop causes 2.1% in GTX 1080 and 4.7% in P100 performance penalties stemming from its runtime overhead. We believe this penalty would be further mitigated once we optimize GLoop for Pascal GPUs. On the other hand, the cases of two or more instances show that the GLoop version is comparable the vanilla version in GTX 1080 (−0.2 – 3.1%). The standard deviation of the vanilla version is large in the four, five, and six instance cases. This is because the vanilla LavaMD execution is so short that the LavaMD sometimes completes without any hardware preemption. This completion is observed more frequently in P100. GLoop can schedule apps stably with its software mechanism.

The overhead of GLoop is higher on P100 (4.7 – 46.2%) than that of GTX 1080 due to the difference of the LavaMD configurations, each of which is tuned to each GPU. The major difference is the number of the running TBs. In LavaMD, P100 executes more TBs at once than GTX 1080 since the number of SMs on P100 is 2.8× more than that on GTX 1080. This causes two negative impacts related to the overhead on P100. First, the standalone performance of LavaMD on P100 is worse than that on GTX 1080 and thus the performance penalty on P100 is relatively...
bigger. Second, LavaMD on P100 launches more TBs and causes context switching overhead shown in Sect. 8.2. We could mitigate this overhead by a NVIDIA Volta GPU’s feature [40] that statically partitions SMs to lower the number of the running TBs.

This result shows that software-level approach can potentially perform efficient context switches compared with hardware preemption in some situations with flexibly scheduling control.

9. Conclusion

This paper presented GLoop, a pure software runtime that allows us to consolidate GPGPU apps including GPU eaters on a GPU. GLoop offers an event-driven programming model so that we can develop highly functional GPGPU apps and schedule them on a shared GPU in a fine-grained manner. The GLoop runtime executes the GPU kernels that are isolated from one another, provides lightweight scheduling points, and schedules them according to a proportional share scheduling policy. In addition, it runs on a proprietary GPGPU software stack including the NVIDIA driver and CUDA library. We implemented a prototype of GLoop and ported eight GPU eaters on it. The experimental results demonstrate that our prototype efficiently consolidates GPGPU apps.

Recent adoption of binary-offered GPU kernels such as NVIDIA cuDNN poses an issue of GLoop applicability: their proprietary nature means that we cannot modify them. One of our future work will be to explore ways to transform GPU kernels including binary blobs into GLoop-based apps. One possible direction is to transform GPU apps into GLoop-based apps and insert scheduling points by using GPU binary analysis frameworks [61].

Completely automatic transformation into GLoop-based apps is also challenging. This is because we cannot estimate the execution time of a specific part of the app from the static information. Thus, profiling-based or semi-automatic insertion techniques are promising. One solution is inserting many scheduling points into the program mechanically, taking profiling information, and removing unnecessary scheduling points. Another is that programmers can specify which scheduling points are necessary by referring to the reported profiling information. This mechanism can further reduce the programmer effort.

Acknowledgements

This work was supported in part by the Japan Society for the Promotion of Science (JSPS KAKENHI 15J09761) and Japan Science and Technology Agency (JST CREST JPMJCR1683).

References


throughput sequence alignment using Graphics Processing Units,”
hybrid algorithm,” Journal of Parallel and Distributed Computing,
DNN Execution Framework on Web Browser,” Proc. 2017 ACM on
Dynamic Optimization Framework for Bulk-Synchronous Applica-
tions in Heterogeneous Systems,” Proc. 19th Int’l Conf. on Paral-
lel Architectures and Compilation Techniques, pp.353–364, ACM,
2010.

Yusuke Suzuki received his B.E. and M.E.
degrees from Keio University in 2013 and 2015.
He is currently a Ph.D. student in the Depart-
ment of Information and Computer Science at
Keio University. His research interests include
operating systems, virtualization and program-
ing languages on heterogeneous systems.

Hiroshi Yamada received his B.E. and
M.E. degrees from the University of Electro-
communications in 2004 and 2006. He received
his Ph.D. degree from Keio University in 2009.
He is currently an associate professor at Tokyo
University of Agriculture and Technology. His
research interests include operating systems,
virtualization, and cloud computing. He is a
member of IEEE/CS, ACM, and USENIX.

Shinpei Kato is an associate professor
in the Graduate School of Information Science
and Engineering at Tokyo University. He re-
ceived his B.S., M.S., and Ph.D. degrees from
has also worked at Nagoya University, Carnegie
Mellon University, and the University of
California, Santa Cruz from 2009 to 2016. His
research interests include operating systems,
real-time systems, and parallel and distributed
systems.

Kenji Kono received the BSc degree in
1993, MSc degree in 1995, and PhD degree in
2000, all in computer science from the Univer-
sity of Tokyo. He is a professor in the Depart-
ment of Information and Computer Science at
Keio University. His research interests include
operating systems, system software, and Inter-
net security. He is a member of the IEEE, ACM,
and USENIX.