GLoop: An Event-driven Runtime for Consolidating GPGPU Applications

Yusuke Suzuki
Keio University
yusuke.suzuki@sslab.ics.keio.ac.jp

Hiroshi Yamada
TUAT
hiroshiy@cc.tuat.ac.jp

Shinpei Kato
The University of Tokyo
shinpei@pf.is.s.u-tokyo.ac.jp

Kenji Kono
Keio University
kono@ics.keio.ac.jp

ABSTRACT

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 *GPU eaters*, 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 GPU 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.

CCS CONCEPTS

• **Software and its engineering** → *Operating systems*; *Cloud computing*;

KEYWORDS

GPGPU, Cloud Computing, Operating Systems

ACM Reference Format:
Yusuke Suzuki, Hiroshi Yamada, Shinpei Kato, and Kenji Kono. 2017. GLoop: An Event-driven Runtime for Consolidating GPGPU Applications. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 14 pages.
https://doi.org/10.1145/3127479.3132023

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, 8, 22, 36], scientific simulations [31, 47], file systems [48, 52], complex control systems [24, 41], autonomous vehicles [20, 32], and server apps including network systems [17, 21], web servers [2], key-value stores [19] and databases [18, 23, 29, 44].

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 [5], 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 [38].

Recent high-functioning GPGPU apps pose a new challenge for multi-tenant consolidation. Such GPGPU apps, referred to as *GPU 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, GPUfs- [48] and GPUnet-based [30] apps poll completions of I/O requests on the GPU. Scientific apps [31, 47] exclusively use GPUs to compute their simulations. Existing GPU resource managers, including GPU command-based schedulers [26, 33, 53], novel GPU kernel launchers [27, 42], and thread block schedulers [7, 56], 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 [37, 55] and persistent threads [14], 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 [38] have mechanisms to preempt long-running GPU kernels. However, as recent literature [56] reported, no publicly available information shows the availability of software-level preemption control. Because of the lack of software control,

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3132023

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 [46]. 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.

This paper makes four main contributions.

- We apply an event-driven programming model to GPGPU apps. We treat GPU kernel operations as *events* except for GPU computations, such as file I/O, network I/O, and GPU yields. A GPGPU app in our programming model is composed of *callbacks*, each of which is associated with an event. When a requested event has been completed, GLoop invokes the corresponding callback registered in advance (Sec. 3).
- We introduce GLoop, a pure software runtime that executes GPU kernels within their own context and orchestrates device- and host-side software mechanisms. An event request made by the GPU kernel and its completion are passed between the mechanisms. The device-side mechanism combines a GPU thread block controlling technique [7] and lightweight scheduling points for efficiently consolidating GPU eaters. The host-side mechanism performs proportional share scheduling combined with weighted fair queuing (Sec. 4 and 5).
- We implement a prototype of GLoop on an unmodified proprietary NVIDIA driver and CUDA SDK 7.5. We also port 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 software [1] (Sec. 6 and 7).
- 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 (Sec. 8).

[1] https://github.com/CPFL/gloop

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 [14, 30, 31, 47, 48]. Such GPU apps launch long- or infinite-running GPU kernels. For example, GPUfs [48] exposes file systems APIs to a GPU program to efficiently execute a GPGPU app involving file operations and facilitate its development. GPUnet [30] also provides a socket abstraction and APIs suitable for GPU processing. The persistent threads model [14] 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 [3]. GPGPU apps performing scientific simulations, sorts, and bioinformatics, typically launch a long-running GPU kernel [31, 47].

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 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 [14, 37, 55], 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 [38], no publicly available information shows the availability of software-level preemption control [56]. 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 [57]. 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/exits are time-consuming, making the latency of the scheduling points high even if a sequence of split GPU kernels does

Table 1: Comparison of GLoop with previous work.

	Consolidating GPU eaters	Resource Isolation	Proprietary GPGPU stack
Hardware preemption [38]		√	√
Kernel splitting		√	√
TimeGraph [26]		√	
GPUvm [53]		√	
Disengaged scheduling [33]		√	√
Gdev [27]		√	
PTask [42]		√	√
Elastic kernels [40]		√	√
EffiSha [7]		√	√
GPUpIO [57]		√	√
GPUShare [11]		√	√
MPS [37], Context funneling [55]			√
FLEP [56]			√
Volta MPS [39]			√
Persistent threads [14]	√		√
GLoop	√	√	√

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 [26] and GPUvm [53] 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 [33] 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. PTask [42], 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 [40] 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 [7] 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 [57] 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.

GPUShare [11] 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, GPUShare 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 [37] (MPS), which is also known as context funneling [55], 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 [56] is similar to EffiSha, but combines MPS with a TB scheduler to offer spatial multitasking. The persistent threads approach [14] 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 can destroy or easily hijack other GPU kernels. This is unacceptable in multi-tenant cloud platforms.

NVIDIA recently announced its NVIDIA Volta architecture with new mechanisms for MPS, called Volta MPS [39]. It enables multiple GPU kernels to run concurrently with their own GPU address spaces. However, a GPU kernel typically exhausts one type of GPU resource and prevents other GPU kernels from running concurrently [40]. Therefore, Volta MPS's fair-sharing scheduling does not work well in multi-tenant use cases, as described in the white paper [39].

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 and GPU resource managers. We carefully designed GLoop to overcome the limitations of existing GPU resource managers. 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 7.5.

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 [30]. Recent research has leveraged discrete GPUs to create high-performance, scalable, and more energy efficient cloud apps [30, 43, 51]. Although GLoop is portable onto integrated (on-chip) GPUs such as Intel GPUs and AMD Kaveri [4], 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 [10]. 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 GPUnet, 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 [46]. 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.

```
1  __device__ void doRead(DeviceLoop* loop,
2    uchar* buf, int fd, size_t offset, size_t size){
3    if (offset < size){
4      size_t sizeToRead = min(PAGE_SIZE, size-offset);
5      auto callback = [=](DeviceLoop* loop, int read){
6        // ...
7        doRead(loop, buf, fd, offset + PAGE_SIZE * gridDim.x, size);
8      };
9      fs::read(loop, fd, offset, sizeToRead, buf, callback);
10     return;
11   }
12   fs::close(loop, fd, [=](DeviceLoop* loop, int err){ });
13 }
```

Figure 1: File Read Program on GLoop.

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.

Fig. 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 GPUfs [48] and GPUnet [30]. 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 [30, 40]. 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 Sec. 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-living, there is a concern that numerous short-living TBs can occupy a shared GPU. Since GLoop treats TB completion as scheduling points, as described in Section 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 environment which isolates GPGPU apps and requires no modifications to the proprietary GPGPU stack. GLoop forces GLoop-based apps to isolate the execution of GPU kernels in order to establish their own GPU contexts, each of which has its GPU virtual address space [12, 28, 53]. In addition, GLoop mechanisms are on top of existing GPGPU runtime libraries, such as CUDA.

Fig. 2 shows an overview of GLoop. The GLoop runtime assigns a GLoop-based app a *host event loop* and *device event loops*, called a

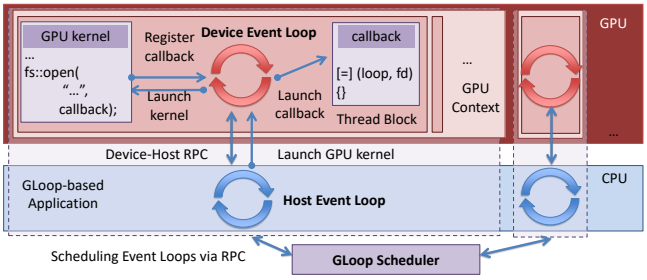


Figure 2: Overall architecture of GLoop.

host loop and device loops for simplicity. *GLoop scheduler*, running as a privileged daemon on a CPU, schedules GLoop-based apps on the basis of its scheduling policy.

Device Event Loop: The device loops, running on a shared GPU, drive the GPU kernel composed of callbacks. They receive an event request from the GPU kernel, pass it to the host loop via the remote procedure call (RPC) slots, and register the callback. This architecture does not involve the GPU kernel finishing/relaunching every event request; device loops can keep running until the suspend signal from the gloop scheduler arrives. When an event has completed, the device loops invoke a corresponding callback.

Note that not all events are pushed to RPC slots. For example, in postTask(), device loops do not access the RPC slots when finding and invoking a registered callback. All the operations are done on the device side, and thus, postTask() offers lightweight scheduling points.

Host Event Loop: The host loop, running on a host CPU, performs events such as I/Os issued by device loops and notifies them of event completion by pushing it to the RPC slots. GLoop-based apps launch GPU kernels through host loop. The host loop communicates with the gloop scheduler to acquire and release the scheduling token, which is the right to use the underlying GPU. When acquiring the token, the host loop launches a GPU kernel that generates device loops.

GLoop Scheduler: The gloop scheduler manages the scheduling token and schedules GLoop-based apps by suspending and resuming the device loops. GLoop sets up a signal slot between the gloop scheduler and device loops to deliver the suspend signal. The gloop scheduler accounts the GPU time of every GLoop-based app. In GLoop, GPU time is an interval during which the host loop holds the scheduling token.

4.1 RPCs between Host and Device Loop

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.

A host loop polls the RPC slots until the device loops issue an RPC operation. When the RPC operation is detected, the host loop calls a predefined function corresponding to it. After the function has finished, the host loop stores the result and event completion in the RPC slot. The device loops invoke the associated callback when detecting the completion notification.

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).

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, and (3) how do we schedule GLoop-based apps in a fair-share manner? To address these challenges, we integrate an efficient scheme 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 [6] 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 [40] and EffiSha [7]. 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 gridDim). 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 [35].

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 kernels, such as running, ready, and blocked, as in traditional process abstraction because it manages the event invocations of the hosted GPU kernels. We believe that this feature would enable us to integrate various CPU scheduling 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 multi-tenant 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 [13]. 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 Sec. 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 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 [25, 40] 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 2.5 µs, as shown in Sec. 8.1.

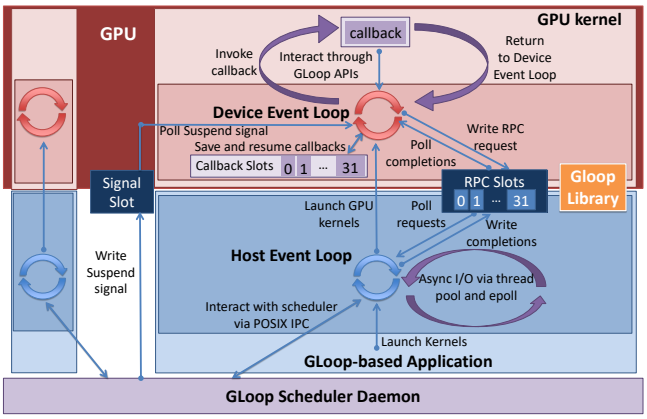


Figure 3: Overview of GLoop prototype.

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 [15, 16, 53, 54], 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.

6 IMPLEMENTATION

We implemented a prototype of GLoop on Linux kernel 4.4.0-34 with CUDA 7.5 for NVIDIA Kepler GPUs [34]. The current prototype is tailored to Linux container-based platforms, which means that GLoop-based apps in each container run on a shared GPU. Fig. 3 overviews our prototype, which consists of a GLoop library and a gloop scheduler daemon.

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 gloop 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.

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.

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.

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 detected. This is the similar to the GPU RPC implementation in previous work [48, 51].

7 CASE STUDIES

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

TPACF:. This app from Parboil2 [50] 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.

LavaMD:. LavaMD from Rodinia [6] 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 Sec. 5.1. GLoop schedules logical TBs on 30 physical TBs. Every time the logical TBs finish, control returns to GLoop.

MUMmerGPU:. MUMmerGPU [45] from Rodinia [6] 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.

Hybridsort: Hybridsort [49] 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.

Grep: Grep from the GPUfs project[2] 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.

Approximate Image Matching: This app (*img*) from GPUfs [48] scans 25,000 images (391 MB 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 GPUfs, 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.

Echo Server: To demonstrate that GLoop can support socket-like APIs for networking, the echo server from the GPUnet project [30] 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.

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 [30]. 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.

[2] https://github.com/gpufs/gpufs

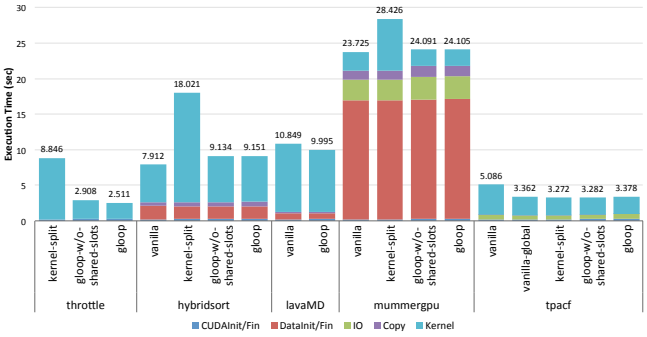


Figure 4: Execution times and their breakdown.

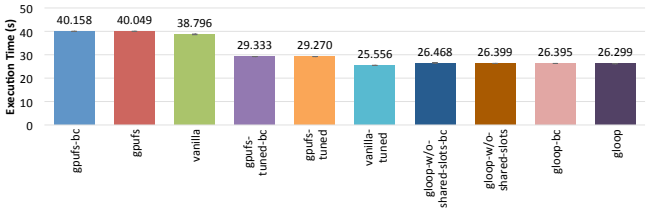


Figure 5: Execution times for grep variations.

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 HP Z840 machine with six Xeon E5-2620 v3 2.4-GHz cores, 8GB of memory and one 500-GB SATA hard disk. We used an NVIDIA Tesla K40c Kepler GPU with 12-GB GDDR5 memory and NVIDIA GPU driver 361.42. The hard disk performance reported by hdparm is 10192.17 and 188.90 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 Sec. 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 postfixed as *-w/o-shared-slots*.

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 3.5× faster than kernel-split, because GLoop offers lightweight scheduling points that do not involve kernel launches. GLoop version with the shared memory optimization is 16% faster than gloop-w/o-shared-slots, since throttle frequently writes a callback in slots on the shared memory.

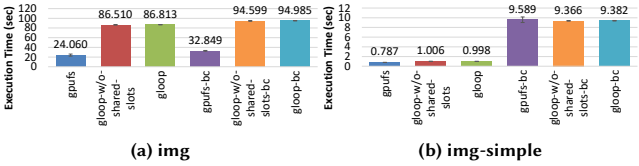
From the execution time and number of scheduling points, we estimated that the latency of a scheduling point, which does not include the scheduling algorithm or GPU kernel switch costs, is less than 2.5 µs. We can hide this latency since GPU executes different warps when accessing GPU memory. In fact, the subsequent experiments revealed that the scheduling point latency of GLoop is hidden or amortized in app benchmarks.

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 laveMD, 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 Sec. 7. 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.

Fig. 4 presents the results. GLoop's overhead is shown in the Kernel category, and it is small or negligible (-8% – 2%) in all cases except for hybridsort. The other categories are similar in all cases. GLoop outperforms vanilla in lavaMD since GLoop controls TBs on SMs and thus executes them more efficiently than those without TB control. The performance penalty of GLoop is 16% in hybridsort, which 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.

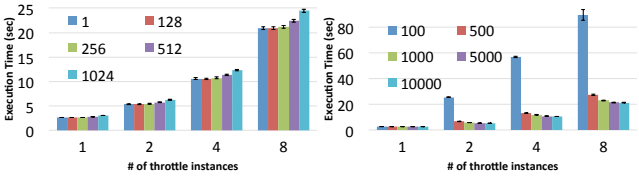
GLoop outperforms kernel-split versions in almost all cases. It is 2.0× and 1.2× faster than the kernel-split of hybridsort and mummergpu, respectively. The kernel-split versions cause numerous kernel launches for scheduling, whereas GLoop does not involve such additional kernel launches. In addition, 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.

I/O-intensive Apps: We measured the execution times of grep and img. We prepared GPUfs- and GLoop-based versions labeled *gpufs* and *gloop*, and prepared a workload for comparison that preallocates 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



(a) img (b) img-simple

Figure 6: Execution times for img variations.



(a) w/ varied # of threads (b) w/ varied timeslice lengths (µs)

Figure 7: Performance across multiple throttle instances.

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.

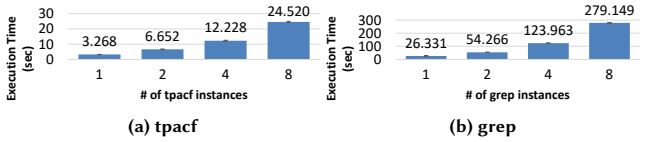
The results obtained for grep are in Fig. 5. The performance of gloop is comparable to the other grep implementations. Gloop outperforms the untuned versions and is 3% slower than the vanilla-tuned and 10% faster than gpufs-tuned versions. Clearing the buffer cache degrades performance by 0.1–0.3%. Since grep repeatedly reads the same set of files, no buffer cache misses occur except for the first read.

Fig. 6a shows the results for img. The execution time for gloop is 3.6× longer than that for gpufs. This is because gpufs can benefit from the GPUfs' 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. 6b indicate that the execution time of gloop is just 1.3× longer than that of gpufs. 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.

The cold buffer cache degrades performance by 9.4–37% because img issues I/O requests more frequently than grep.



(a) tpacf (b) grep

Figure 8: Performance across multiple app instances.

8.2 Performance at Scale

We concurrently ran multiple instances of GLoop-based apps to find the performance penalty of GLoop's consolidation. We launched one, two, four, and eight instances and measured each of their execution times. We used throttle as the microbenchmark, tpacf as a compute-intensive app, and grep as an I/O-intensive app.

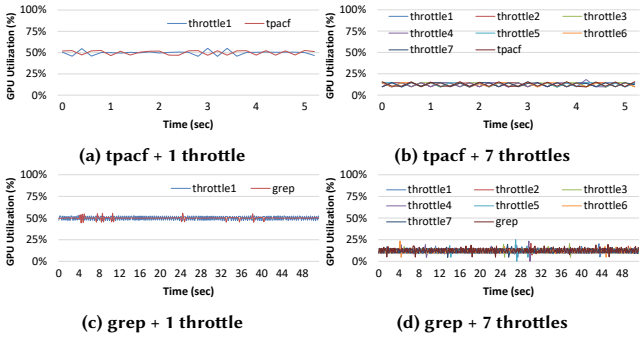
Context Switching Latency: To discern the context switching overhead, we ran throttle by varying the number of threads and timeslice lengths. Fig. 7a plots the execution times for different numbers of threads. In the single instance case, the increase in threads from 1 to 1024 lengthens the execution time by 16.2%. 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 (17.0% longer execution time from 1 to 1024). The slight overhead results from thread synchronization done in context switching.

Fig. 7b plots the results for varied timeslice lengths. Due to the optimization of GLoop to avoid polling on the PCIe bus, as described in Sec. 4.2, the actual timeslice consumed slightly differs from the specified value. GLoop performs 2631 context switches on average per app in the two instances with 10000 µs timeslices. When the timeslices are too short, context switches are dominant in the execution times. The execution time with a 100 µs timeslice is 4.2× longer than that with a 10000 µs timeslice for eight instances, where the execution time with the 100 µs timeslice is 31.6× 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 (8.0×).

Apps: Fig. 8 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 2.8%.

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.

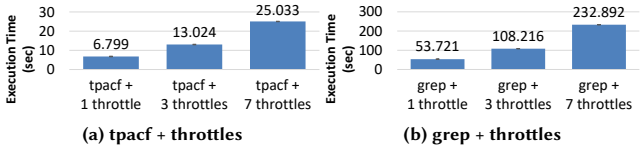
The execution times for eight grep instances exceed eight times the standalone's execution time (10.6×) as a result of disk I/O contention. The result in the next section validates this finding: the execution time for grep with seven throttles is 8.84× longer than that for the standalone since throttle does not issue any I/O requests. The remaining slowdown stems from the overhead imposed by scheduling the running instances.



(a) tpacf + 1 throttle (b) tpacf + 7 throttles

(c) grep + 1 throttle (d) grep + 7 throttles

Figure 9: GPU utilization for GPU apps (over 200 ms).



(a) tpacf + throttles (b) grep + throttles

Figure 10: Execution times for GPU apps with throttles.

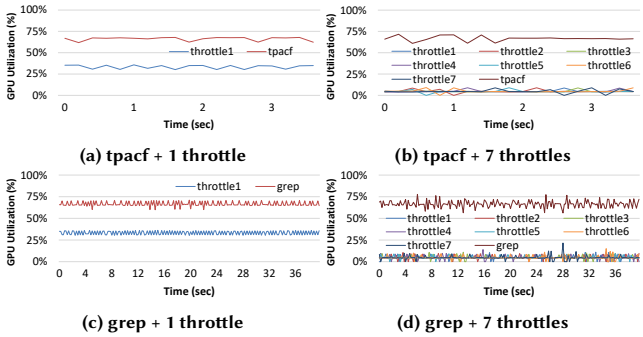
8.3 Performance Isolation

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. Fig. 9 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. We have omitted the results obtained from four instances due to space limitations but their tendencies are quite similar to those in the other cases. 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%.

Fig. 10 shows the execution times for each instance. The execution time for eight instances for tpacf is 3.68× 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 4.34× longer than in two instances. This results from the overhead for scheduling multiple GPU apps.

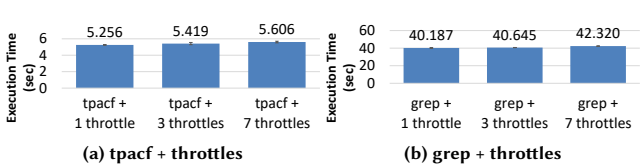
Next we changed the resources assigned to the GPU apps. We assigned 66% of the utilization to a target app (tpacf or grep) while co-running throttle instances shared the GPU with one another. Fig. 11 plots the results, which reveal that GLoop successfully assigns a target app the weighted GPU utilization and the other throttles share the remaining resources. The execution time shown in Fig. 12 slightly increases with the number of apps due to the accumulated overhead of the scheduler.



(a) tpacf + 1 throttle (b) tpacf + 7 throttles

(c) grep + 1 throttle (d) grep + 7 throttles

Figure 11: GPU utilization for GPU apps (over 200 ms). 66% of the GPU resources is assigned to the apps.



(a) tpacf + throttles (b) grep + throttles

Figure 12: Execution times for GPU apps with throttles. 66% of the GPU resources is assigned to the target app.

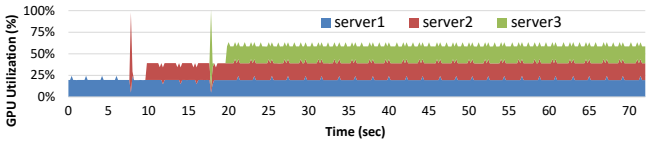


Figure 13: Stacked GPU utilization for consolidated GPU matmul servers (over 200 ms).

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.

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.

Fig. 13 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

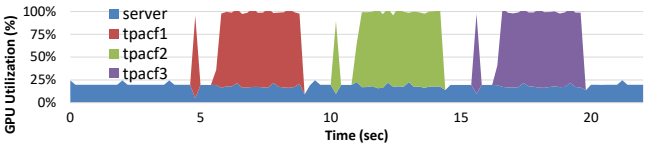


Figure 14: Stacked GPU utilization for GPU matmul server and tpacf app (over 200 ms).

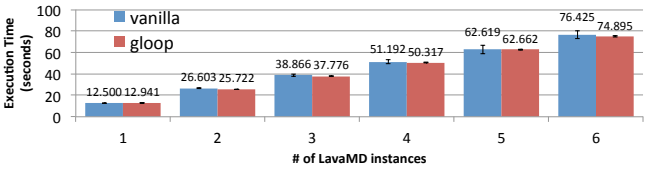


Figure 15: Execution times of LavaMD with GLoop and hardware preemption.

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 Sec. 8.1) takes 200 ms and thus GPU utilization is temporarily occupied.

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. 14. 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 information 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 ported GLoop to Pascal GPUs and ran LavaMD benchmark. In this experiment, we replaced a Kepler K40 GPU with a GTX 1080 Pascal GPU. To support Pascal GPUs, we upgraded our GPU driver and CUDA runtime to 8.0. In each trial, we launched multiple instances of vanilla and GLoop-based LavaMD (1 to 6) and measure the execution time.

Fig. 15 shows the average execution time and standard deviation. The standalone performance shows that GLoop causes a 3.5% performance penalty 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 to or outperforms the vanilla version (-0.1% to 3.3%). GLoop-based LavaMD outperforms the vanilla one in the two-instance case (3.3%) while the standard deviation of the vanilla version is large in the four- and six-instance cases. This result shows that software-level approach can potentially perform more efficient context switches compared with hardware preemption in some situations.

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 [9].

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.

ACKNOWLEDGMENTS

We acknowledge our shepherd Christopher J. Rossbach and the anonymous reviewers for their insightful comments. 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

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proc. of the 12th USENIX Conf. on Operating Systems Design and Implementation*. USENIX, 265–283.

[2] Sandeep R Agrawal and Alvin R Lebeck. 2014. Rhythm : Harnessing Data Parallel Hardware for Server Workloads. In *Proc. of the 19th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, 19–34.

[3] Timo Aila and Samuli Laine. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proc. of the Conf. on High Performance Graphics*. ACM, 145–149.

[4] AMD. [n. d.]. AMD Kaveri. http://www.amd.com/en-us/products/processors/desktop/a-series-apu. ([n. d.]).

[5] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation*. USENIX.

[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proc. of the 2009 Int'l Symp. on Workload Characterization*. IEEE, 44–54.

[7] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. 2017. EffiSha: A Software Framework for Enabling Effficient Preemptive Scheduling of GPU. In *Proc. of the 22nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. ACM, 3–16.

[8] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. 2016. GeePS: scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proc. of the 11th European Conference on Computer Systems*. ACM Press, 1–16.

[9] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. 2010. Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems. In *Proc. of the 19th Int'l Conf. on Parallel Architectures and Compilation Techniques*. ACM, 353–364.

[10] Node.js Foundation. 2016. Node.js. https://nodejs.org. (2016).

[11] Anshuman Goswami, Jeffrey Young, Karsten Schwan, Naila Farooqui, Ada Gavrilovska, Matthew Wolf, and Greg Eisenhauer. 2016. GPUShare: Fair-Sharing Middleware for GPU Clouds. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops*. IEEE, 1769–1776.

[12] Mathias Gottschlag, Marius Hillenbrand, Jens Kehne, Jan Stoess, and Frank Bellosa. 2013. LoGV: Low-overhead GPGPU Virtualization. In *Proc. of the 4th Int'l Workshop on Frontiers of Heterogeneous Computing*. IEEE, 1721–1726.

[13] A. G. Greenberg and N. Madras. 1992. How fair is fair queuing. *J. ACM* 39, 3 (1992), 568–598.

[14] K Gupta, J A Stuart, and J D Owens. 2012. A Study of Persistent Threads Style GPU Programming for GPGPU Workloads. In *Proc. of the Innovative Parallel Computing*. IEEE, 1–14.

[15] Vishakha Gupta, Adam Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. 2009. GViM: GPU-Accelerated Virtual Machines. In *Proc. of the 3rd Workshop on System-level Virtualization for High Performance Computing*. ACM, 17–24.

[16] Vishakha Gupta, Karsten Schwan, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. 2011. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In *Proc. of the 2011 USENIX Annual Technical Conf.* USENIX, 31–44.

[17] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: A GPU-Accelerated Software Router. In *Proc. of the ACM SIGCOMM 2010 Conf.* ACM, 195–206.

[18] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational Joins on Graphics Processors. In *Proc. of the 2008 ACM SIGMOD Int'l Conf. on Management of Data*. ACM, 511–524.

[19] Tayler H Hetherington, Mike O'Connor, and Tor M Aamodt. 2015. MemcachedGPU: Scaling-up Scale-out Key-value Stores. *Proc. of the 6th ACM Symp. on Cloud Computing*, 43–57.

[20] Manato Hirabayashi, Shinpei Kato, Masato Edahiro, Kazuya Takeda, Taiki Kawano, and Seiichi Mita. 2013. GPU Implementations of Object Detection using HOG Features and Deformable Models. In *Proc. of the 1st Int'l Conf. on Cyber-Physical Systems, Networks, and Applications*. IEEE, 106–111.

[21] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and KyoungSoo Park. 2011. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation*. USENIX, 1–14.

[22] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proc. of the 22nd ACM International Conference on Multimedia*. ACM, New York, New York, USA, 675–678.

[23] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU Join Processing Revisited. In *Proc. of the 8th Int'l Workshop on Data Management on New Hardware*. ACM, 55–62.

[24] Shinpei Kato, Jason Aumiller, and Scott Brandt. 2013. Zero-copy I/O processing for low-latency GPU computing. In *Proc. of the 4th Int'l Conf. on Cyber-Physical Systems*. ACM/IEEE, 170–178.

[25] Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Ragunathan Rajkumar. 2011. RGEM: A responsive GPGPU execution model for runtime engines. In *Proc. of the 32nd Real-Time Systems Symposium*. IEEE, 57–66.

[26] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. 2011. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *Proc. of the 2011 USENIX Annual Technical Conf.* USENIX, 17–30.

[27] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. 2012. Gdev: First-Class GPU Resource Management in the Operating System. In *Proc. of the 2012 USENIX Annual Technical Conf.* USENIX, 401–412.

[28] Jens Kehne, Jonathan Metter, and Frank Bellosa. 2015. GPUswap: Enabling Oversubscription of GPU Memory through Transparent Swapping. In *Proc. of the 11th ACM Int'l Conf. on Virtual Execution Environments*. ACM, 65–77.

[29] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proc. of the 2010 Int'l Conf. on Management of Data*. ACM, 339–350.

[30] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. 2014. GPUnet: Networking Abstractions for GPU Programs. In *Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation*. USENIX, 201–216.

[31] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. 2011. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers. In *Proc. of the 2011 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*. ACM, 11:1–11:12.

[32] Matthew McNaughton, Chris Urmson, John M. Dolan, and Jin-Woo Lee. 2011. Motion Planning for Autonomous Driving with a Conformal Spatiotemporal Lattice. In *Proc. of the 2011 Int'l Conf. on Robotics and Automation*. IEEE, 4889–4895.

[33] Konstantinos Menychtas, Kai Shen, and Michael L. Scott. 2014. Disengaged scheduling for fair, protected access to fast computational accelerators. In *Proc. of the 19th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, 301–316.

[34] NVIDIA. 2012. NVIDIA's next generation CUDA computer architecture: Kepler GK110. http://www.nvidia.com/. (2012).

[35] NVIDIA. 2014. CUDA Pro Tip: Occupancy API Simplifies Launch Configuration. https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-occupancy-api-simplifies-launch-configuration/. (2014).

[36] NVIDIA. 2015. GPU-Based Deep Learning Inference: A Performance and Power Analysis. http://developer.download.nvidia.com/embedded/jetson/TX1/docs/jetson_tx1_whitepaper.pdf. (2015).

[37] NVIDIA. 2015. Multi-Process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf. (2015).

[38] NVIDIA. 2016. NVIDIA Tesla P100 – The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU. http://www.nvidia.com/object/pascal-architecture-whitepaper.html. (2016).

[39] NVIDIA. 2017. NVIDIA TESLA V100 GPU ARCHITECTURE. June (2017).

[40] Sreepathi Pai, Matthew J Thazhuthaveetil, and R Govindarajan. 2013. Improving GPGPU concurrency with elastic kernels. In *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Vol. 41. ACM, 407.

[41] N Rath, J Bialek, P J Byrne, B DeBono, J P Levesque, B Li, M E Mauel, D A Maurer, G A Navratil, and D Shiraki. 2012. High-speed, multi-input, multi-output control using GPU processing in the HBT-EP tokamak. *Fusion Engineering and Design* (2012), 1895–1899.

[42] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: Operating System Abstractions To Manage GPUs as Compute Devices. In *Proc. of the 23rd Symp. on Operating Systems Principles*. ACM, 233–248.

[43] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: a Compiler and Runtime for Heterogeneous Systems. In *Proc. of the 24th Symp. on Operating Systems Principles*. ACM, 49–68.

[44] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. 2010. Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. In *Proc. of the 2010 Int'l Conf. on Management of Data*. ACM, 351–362.

[45] Michael C Schatz, Cole Trapnell, Arthur L Delcher, and Amitabh Varshney. 2007. High-throughput sequence alignment using Graphics Processing Units. *BMC bioinformatics* 8, 1 (2007), 474.

[46] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. 2016. EbbRT: A Framework for Building Per-Application Library Operating Systems. In *Proc. of the 12th USENIX Conf. on Operating Systems Design and*

Implementation. USENIX, 671–688.

[47] Takashi Shimokawabe, Takayuki Aoki, Tomohiro Takaki, Toshio Endo, Akinori Yamanaka, Naoya Maruyama, Akira Nukada, and Satoshi Matsuoka. 2011. Peta-scale Phase-Field Simulation for Dendritic Solidification on the TSUBAME 2.0 Supercomputer. In *Proc. of the 2011 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*. ACM, 3:1–3:11.

[48] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating a File System with GPUs. In *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, 485–498.

[49] Erik Sintorn and Ulf Assarsson. 2008. Fast parallel GPU-sorting using a hybrid algorithm. *J. Parallel and Distrib. Comput.* 68, 10 (2008), 1381–1388.

[50] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Technical Report IMPACT-12-01* (2012).

[51] Jeff A. Stuart and John D. Owens. 2011. Multi-GPU MapReduce on GPU Clusters. In *Proc. of the 2011 Int'l Parallel & Distributed Processing Symp.* IEEE, 1068–1079.

[52] Weibin Sun, Robert Ricci, and Matthew L. Curry. 2012. GPUstore. In *Proc. of the 5th Annual Int'l Systems and Storage Conf.* ACM, 1–12.

[53] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. 2014. GPUvm: Why Not Virtualizing GPUs at the Hypervisor?. In *Proc. of the 2014 USENIX Annual Technical Conf.* USENIX, 109–120.

[54] Kun Tian, Yaozu Dong, and David Cowperthwaite. 2014. A Full GPU Virtualization Solution with Mediated Pass-Through. In *Proc. of the 2014 USENIX Annual Technical Conf.* USENIX, 121–132.

[55] Lingyuan Wang, Miaoqing Huang, and Tarek El-Ghazawi. 2011. Exploiting Concurrent Kernel Execution on Graphic Processing Units. In *Proc. of the Int'l Conf. on High Performance Computing and Simulation*. IEEE, 24–32.

[56] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. 2017. FLEP: Enabling Flexible and Efficient Preemption on GPUs. In *Proc. of the 22nd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, 483–496.

[57] Lior Zeno, Avi Mendelson, and Mark Silberstein. 2016. GPUpIO: The Case for I/O-Driven Preemption on GPUs. In *Proc. of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. ACM, 63–71.