Towards Multi-tenant GPGPU: Event-driven Programming Model for System-wide Scheduling on Shared GPUs

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

Hiroshi Yamada
Tokyo University of Agriculture and Technology
hiroshiy@cc.tuat.ac.jp

Shinpei Kato
Nagoya University
shinpei@is.nagoya-u.ac.jp

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

ABSTRACT

Graphics processing units (GPUs) are attractive to the general-purpose computing (GPGPU) beyond the graphics purpose. Sharing GPUs among such GPGPU applications is a key requirement especially for cloud platforms whose resources are utilized by various cloud users. However, consolidating recent GPU applications, referred to as *GPU eaters*, on a GPU poses a new challenge. Such advanced applications are designed based on the assumption that *only one* GPU application runs on a GPU at a time. In this paper, we present GLoop, a GPGPU framework that allows multiple GPU eaters to share a GPU. GLoop offers an event-driven programming model that generates scheduling points in GPU kernels. It achieves resource isolation among GPU applications and schedules applications by suspending/resuming GPU kernels only if necessary to reduce the number of kernel launches. The preliminary experiments demonstrate that GLoop model is applicable to a GPU eater. The results also show that GLoop enables two GPGPU applications to run concurrently on a shared GPU by interleaving GPU kernel execution.

1. INTRODUCTION

Graphics processing units (GPUs) are attractive processing devices for not only graphics purposes but also general-purpose computing. General-purpose computing on GPUs (GPGPU) is appealing in various application domains, such as scientific simulations [15, 25], network systems [4, 7], file systems [26, 27], web servers [1], database management systems [5, 8, 12, 24], complex control systems [9, 22], and autonomous vehicles [6, 16].

Sharing GPUs among such GPGPU applications is a key requirement for cloud platforms whose resources are utilized by various cloud users. Consolidating GPGPU applications on a GPU brings several benefits such as more efficient GPU utilization with fewer GPUs. Recent research has this goal, GPU resource managers including GPU command-based schedulers [10,17,28], novel GPU kernel launchers [11, 23], context funneling [19, 31], and persistent threads approaches [3] have been studied.

However, recent GPGPU software infrastructure poses a new challenge for consolidation, where *only one* GPU application is assumed to run on a GPU at a time. Scientific applications composed of long-running GPU kernels [15, 25] starve applications co-located on a shared GPU. To make matters worse, applications polling in GPU kernels are becoming common in recent research software. For example, the GPUfs- [26] and GPUnet-based [13] applications poll completions of I/O requests on the GPU. In the persistent threads approach [3], worker GPU thread blocks continuously poll their work queues on GPU. Since these applications, referred to as *GPU eaters*, do not provide scheduling points, one application can easily monopolize a shared GPU. Although dividing a GPU kernel of the applications into numerous small GPU requests provides more frequent scheduling opportunities, applications' performances are significantly degraded due to having more kernel launches whose cost is high.

This paper presents GLoop, a GPGPU framework that enables us to host advanced GPU applications on a shared GPU. GLoop provides an event-driven programming model for GPU kernels. This model allows GPU applications to inherit the high functionality of the GPU eaters while generating scheduling points. GLoop achieves GPU resource isolation among GPU applications and schedules each GPU kernel by suspending/resuming the GPU kernels only if it is necessary to minimize kernel launches.

We implemented a prototype of GLoop on the non-modified proprietary NVIDIA driver and CUDA 7.5 toolkit. The preliminary experiments show that GLoop is applicable to a GPU eater, *grep*, and its performance is comparable to the original grep with GPUfs. The experimental results also reveal that GLoop successfully interleaves two GPU applications on a shared GPU.

2. MOTIVATION

2.1 GPU Eaters

Numerous researchers have studied how GPGPU applications can be more effective [3, 13, 15, 25, 26]. For example, GPUfs [26] exposes file systems APIs to a GPU program to efficiently execute a GPU application involving file operations and facilitate its development. GPUnet [13] also provides a socket abstraction and APIs suitable for GPU processing. In the persistent threads model [3], a maximum-sized grid is launched on a GPU, and its thread blocks continuously fetch GPU tasks from work queues to execute them without costly kernel launches. GPGPU applications performing a scientific simulation on a GPU typically run for a long time [15, 25].

Unfortunately, these application developers implicitly as-

Table 1: Comparison of GLoop to other previous work.

	Consolidating GPU eaters.	Resource Isolation	Proprietary GPGPU stack
GLoop	√	√	√
PTask [23]		√	√
TimeGraph [10], Gdev [11], GPUvm [28]		√	
Disengaged scheduling [17], Elastic kernels [20]		√	√
NVIDIA MPS [19], Context funneling [31]	√		√
Persistent threads [3]	√		√
GPUpIO [32]		√	√

sume that *only one* GPU application runs on a GPU at a time. Consolidating these types of applications, called GPU eaters, on a shared GPU is an interesting challenge. Since a GPU is a non-preemptive device, GPU contexts cannot be switched like processes running on a CPU. GPUfs and GPUnet make their applications poll the I/O event to avoid costly GPU kernel launches so that the other GPU applications cannot do anything until the running application completes. We cannot execute two or more persistent threads applications concurrently since the thread blocks of one application infinitely run over GPU tasks. Many of the scientific GPGPU applications launch their own long-running kernel that monopolizes a GPU typically for seconds, minutes, or hours.

2.2 Previous Work

Current GPU resource managers aim to share GPU resources among GPU applications, but these resource managers are inherently of limited use when executing the GPU eaters concurrently on a GPU. GPUvm [28] and TimeGraph [10] offer a command-based scheduler that issues GPU commands received from virtual machines (VMs) or processes based on their scheduling policy. Disengaged scheduler [17] schedules GPU commands with a sophisticated probabilistic model. Even with these command-based schedulers, a GPU eater issuing commands for polling exclusively uses a shared GPU. To avoid this situation, we have to redesign such applications to issue numerous GPU commands instead of one polling command at the expense of a performance penalty caused by their GPU kernel launches.

Gdev [11] multiplexes a GPU device at the operating system (OS) level. It also has a GPU scheduler whose scheduling point is GPU kernel launches. If a GPU kernel is running for a long time, the Gdev scheduler assigns long slices of time to other GPU applications' kernels to achieve fair GPU utilization. PTask [23], where a GPGPU application is designed as a data flow graph consisting of GPU kernel modules, schedules GPU kernels at GPU kernel launches. The elastic kernel [20] transforms physical thread blocks to logical thread blocks and dispatches them to physical resources. It offers time-slicing by adjusting the amount of logical thread blocks executed in one launch. These schedulers suffer from the same problem as the command-based schedulers; sharing a GPU with the schedulers requires that running GPU applications issue small GPU requests for scheduling point generation.

GPUpIO [32] achieves I/O-driven preemption among GPU applications by instrumenting the code with the save and restore procedures. Instead of waiting for I/O completions in a polling manner, an inserted procedure saves the state of the executing thread block, finishes it, and executes another GPU kernel. When the I/O operation is completed, GPUpIO restores the saved state of the thread block. While GPUpIO is effective for polling-based GPU eaters, long-running kernels such as scientific calculation and persistent threads can still monopolize a shared GPU.

Multi-process service [19] (MPS), known as context funneling [31], allows us to concurrently execute multiple GPU kernels on a GPU. The MPS redirects all the streams of the running GPU applications to one GPU context in a service process. Thus, the redirected GPU kernels run in one GPU context simultaneously. The persistent threads approach [3] can schedule GPU kernels requested from GPGPU applications. GPU applications add their GPU tasks to the work queue and running thread blocks execute GPU tasks in the work queue. Since all GPU tasks in these approaches run in the same virtual address space, a GPU request from a buggy or malicious GPU application can destroy or easily hijack the other GPU kernel.

Note that device-level mechanisms for GPU preemption are not a perfect solution to realize GPGPU application consolidation. Although such preemption mechanisms are powerful and supported by real-life GPUs [14], recent research literatures [21,29] report that the preemption involves high latency due to processing a large amount of context data such as GPU registers. Thus, frequent GPU context switches significantly hurt applications' performances.

3. GLoop

This paper presents GLoop, which allows us to host multiple GPU eaters on a single GPU. Table 1 briefly summarizes the comparison between GLoop and GPU resource managers described in Sec. 2.2. To overcome the weakness of the existing GPU resource managers, we carefully designed GLoop to achieve the following goals.

- **Efficiently consolidates GPU eaters:** GLoop concurrently executes GPGPU applications on a shared GPU. It schedules them for a short period but minimizes GPU kernel launches like those of the persistent threads model.
- **Provides GPU resource isolation:** GLoop isolates GPU requests from GPGPU applications, which means that a malicious or buggy GPGPU application does not destroy other GPGPU applications' contexts and GLoop.
- **Not modify proprietary GPGPU software stack:** GLoop works atop proprietary GPGPU device drivers and GPGPU libraries. The current prototype runs on the non-modified NVIDIA device driver and CUDA SDK 7.5.

The key insight behind GLoop is to provide an *event-driven* programming model for GPGPU applications' developers, borrowing an idea from Node.js [2]. This model allows us to concurrently execute multiple GPU applications by scheduling their GPU requests at each callback execution. In addition, GLoop provides resource isolation among GPU applications by executing their callbacks in their own GPU contexts.

```
1  __device__ void performRead(
2      DeviceLoop* loop, uchar* scratch,
3      int fd, size_t cursor, size_t size) {
4      if (cursor < size) {
5          size_t sizeToRead = min(PAGE_SIZE, size - cursor);
6          auto callback = [=](DeviceLoop* loop, int read) {
7              // ...
8              performRead(loop, scratch, fd,
9              cursor + PAGE_SIZE * gridDim.x, size);
10         };
11         fs::read(loop, fd, cursor, sizeToRead, scratch, callback);
12         return;
13     }
14     fs::close(loop, fd, [=](DeviceLoop* loop, int err) { });
15 }
```

Figure 1: Example of GLoop-aware program, which reads a file.

3.1 Programming Model

Event-driven Programming Model: GLoop-aware GPU applications are composed of *callbacks*, each of which is associated with an event on the host such as an I/O operation (file read, write etc.). When an event completes, GLoop executes the corresponding callback and unregisters it. GLoop polls host event completions until all the remaining callbacks are invoked. A typical program in GLoop registers a callback before finishing its execution. The registered callback will be invoked after the associated event happens. The callback registers a new callback. The program does not finish until all the registered callbacks are executed. In other words, GLoop programming is a continuation-passing style (CPS) where each callback represents the next control state. We believe that GLoop is applicable to a broad range of GPU applications since direct style programs can be automatically transformed to CPS-based ones.

An example program that reads a file is shown in Fig.1. The function, performRead, reads a specified file up to size bytes. We define a callback function, callback, in the C++ lambda style at line 6. This callback processes read data and then performRead() executes again (line 6–10). The program executes fs::read(..., callback), which requests a file read to the host and registers the passed callback. GLoop executes the registered callback when the requested read completes. Since performRead() is called in the callback, the running callback registers itself again via the fs::read() (line 11). These steps are repeated until the file read size is equal to the size.

Coalesced API calls: GLoop adopts the *coalesced API calls*, inspired by GPUfs [26] and GPUnet [13]. GPU programs use hierarchical parallelism where thread blocks have a coarse-grained parallelism, and all the threads in each thread block perform a single task. Because threads in a thread block are executed in lock-step, divergent control paths in a thread block cause severe performance degradation. This performance degradation happens when threads execute different functions for host events in a callback. To alleviate the performance degradation, GLoop forces threads in a thread block to call the same APIs with the same arguments at the same point in the application code.

3.2 Architecture

An overview of the GLoop architecture and GLoop-aware GPU applications are illustrated in Fig. 2. The main components of GLoop are *gloop scheduler*, *host event loop*, and *device event loop*. The gloop scheduler runs at the privileged layer. The host event loops and device event loops are offered by GLoop to GPU applications and run on a CPU and GPU, respectively. GLoop is portable to various resource

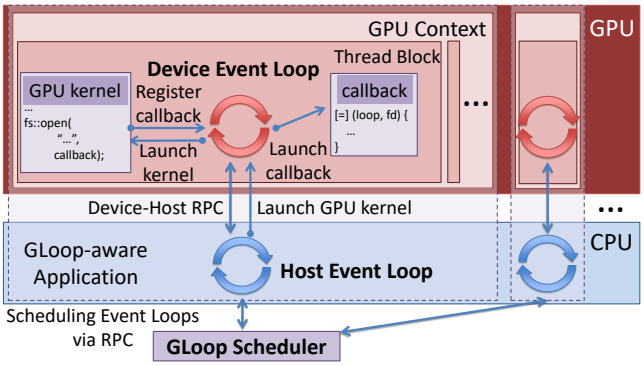


Figure 2: Overall architecture of GLoop.

shared environments. In a resource container-based system, one possible setup is that the gloop scheduler runs on the host OS as a service process, the host event loop in GPU applications uses CPU slices assigned to their own containers, and their device event loop runs on the shared GPU. In a VM system where a GPU is virtualized at the hypervisor [28, 30], the gloop scheduler is inside the hypervisor, and each loop of the GPU applications runs on virtualized CPUs and the GPU of their VMs.

The gloop scheduler monitors GPU utilization of each GPU application and makes a scheduling decision based on its own policy. The gloop scheduler instructs host event loops to suspend/resume GPU kernels so that an appropriate GPU kernel can be executed. A GLoop-aware application establishes the GPU context. Since each GPU kernel runs in its own GPU virtual address space, resource isolation among applications on a GPU is naturally achieved. To suspend/resume a GPU kernel, a host event loop communicates with the corresponding device event loops, each of which is allocated per thread block. The device event loop requests host resources to the host event loop, polls an event completion, and executes the corresponding callback.

3.2.1 *Gloop Scheduler and Host Event Loop*

A GLoop-aware application establishes a host event loop and connects the loop to the gloop scheduler. The application asks the host event loop to launch its GPU kernel. The host event loop requests the token from the gloop scheduler and starts the GPU kernel after receiving the token and its slice. Also, the host event loop invokes a registered host-side function corresponding to the event requested by the device event loop. The host event loop notifies the device event loop of the event completions. For example, when a GPU kernel issues a file read request, the host event loop reads the file, transfers the read data to device memory through GPU DMA engines and notifies the device event loop of the read completion through a host-device RPC.

The gloop scheduler sends suspend and resume requests to the running host event loops based on its scheduling policy. When a host event loop requests execution of a GPU kernel, the gloop scheduler sends suspend messages to the currently running host event loop or waits until the running kernel exhausts its slice. The gloop scheduler uses the points of dispatching the callback function as a scheduling point. The device event loop yields the GPU at the point if an application has exhausted its slice or its host event loop has received a suspend message. To switch the running GPU kernel to another one, GLoop saves the callback's states and

finishes the running GPU kernel. Then, the next host event loop launches its GPU kernel. The host event loop requests the gloop scheduler to relaunch the GPU kernel if pending callbacks remain.

In the current design, GLoop kills GPU applications monopolizing a shared GPU or not requesting host events. The gloop scheduler cannot suspend and resume such GPU applications since the GPU is a non-preemptive device. If the underlying GPU offers GPU preemption at the device level, we can preempt the GPU from the GPU applications instead of killing them.

3.2.2 Device Event Loop

When a GPU kernel requests a host operation such as file reads, the device event loop passes I/O requests to the host event loop. The host event loop performs the requested operation while the device event loop polls event completions. The host event loop sends the event results to the device event loop via a host-device RPC and the device event loop invokes the corresponding callback function. For example, in the case of reading a file, the device event loop requests the host event loop to read a file and associates a callback with its completion. While the host event loop performs the requested file read, the device event loop polls the completion. When the device event loop finds out that the file read is completed, it writes the transferred data back to a specified buffer and invokes the associated callback.

GLoop offers two modes of device event loop to balance a trade-off between the latency of I/O operation and GPU resource consumption; the blocking-based mode and polling-based mode. In the blocking-based mode, device event loop stops the GPU kernel and relaunches it after device event loop is notified of I/O completion. This mode leads to efficient utilization of GPU resources, but application performance can become worse since a GPU kernel launch is a time consuming operation [13]. In the polling-based mode, device event loop polls I/O completion on the GPU instead of stopping/resuming the GPU kernel. In this mode, we can minimize the latency of fetching I/O results since the GPU kernel is not relaunched. On the other hand, polling the I/O results consumes GPU resources. The polling-based mode is effective in a situation where only one GPU application runs on a system. When one of the running GPU applications is latency-sensitive, its device event loops and the other loops are set to the polling-based and blocking-based mode, respectively.

4. IMPLEMENTATION

We implemented a prototype of GLoop on Linux 3.16.0-41 with CUDA 7.5 for NVIDIA Kepler GPUs [18]. The current prototype is tailored to Linux container-based virtualized environments, which means that GLoop-aware applications in each container run on a shared GPU. We believe that the concept of GLoop is applicable to other GPU-shared environments such as GPU virtualization [28].

Our prototype consists of the gloop scheduler daemon and the GLoop library. The gloop scheduler runs on the host. The GLoop library is linked with GPGPU applications to provide the host event loop and device event loop. Each host event loop is connected with the gloop scheduler by the POSIX inter-process communication functionality and communicates with its device event loops through the host-device remote procedure call (RPC).

4.1 GLoop Scheduler

The gloop scheduler manages the token of kernel execution, and the host event loop is required to acquire the token before executing any GPU kernels. When there is a host event loop that attempts to acquire the token and it is blocked longer than the threshold, the gloop scheduler sends a suspend request to the active device event loop through RPC. Once this request is received by the device event loop, the device event loop saves its state, stops itself, and finishes the GPU kernel. Then, the host event loop associated with the stopped device event loops releases the token. The current prototype selects 10 milliseconds for the threshold.

4.2 GLoop Library

The host-device RPC uses the producer-consumer model on the top of the host-device shared memory. GLoop first allocates host memory and maps it to the GPU virtual memory space to make it accessible from both the host and the device. When the device event loop requests an RPC, the device event loop writes RPC arguments and an identifier (number) to this RPC memory. The host event loop polls this memory, and once the host event loop finds the contents of this memory are changed, it performs the requested RPC based on the written arguments and the identifier. When the host event loop finishes the requested RPC, it writes back the completion to the RPC memory.

The device event loop saves registered callback states in the callback slots every time GLoop APIs are called. Each device event loop has a fixed-sized (64 slots in our prototype) callback slots in the device memory. Each callback slot is associated with the RPC memory. The device event loop saves a callback in one of the unused callback slots and marks the slot used. Then, the device event loop requests an RPC using the RPC memory associated with the slot. At the boundary of callback executions, the device event loop peeks the RPC memory to find the completion. If the completion is found, the device event loop clears the RPC completion and invokes the associated callback. After the callback finishes, the device event loop destroys the callback saved in the slot and marks the invoked slot as unused.

When the device event loop is suspended, the device event loop saves small pieces of the control information (e.g. the bit flags that represent which callback slot is used) in the device memory. The callback slots, the largest state of the device event loop, are saved in the device memory so that the slots are live after the kernel finishes. After the device event loop is suspended, the GPU kernel exits. When the next GPU kernel is launched from another GPU context, the GPU hardware performs the context switch automatically.

5. PRELIMINARY EXPERIMENTS

We conduct experiments to answer the following questions: 1) is GLoop's model effective for a GPU eater, 2) is GLoop application's performance reasonable, and 3) can GLoop achieve GPU sharing between GPGPU applications?

We evaluate our prototype on a DELL PowerEdge T320 machine with eight Xeon E5-2470 2.3-GHz processors, 16-GB memory and one 2-TB SATA hard disk. We use an NVIDIA GTX 770 Kepler GPU with 2-GB GDDR5 memory. We run Ubuntu 14.04 with Linux kernel 3.16.0-41, CUDA SDK 7.5, and NVIDIA GPU driver 352.55. The hard disk performance reported by hdparm is 9615.03 MB/s for timing cached reads and 150.98 MB/s for timing buffered disk

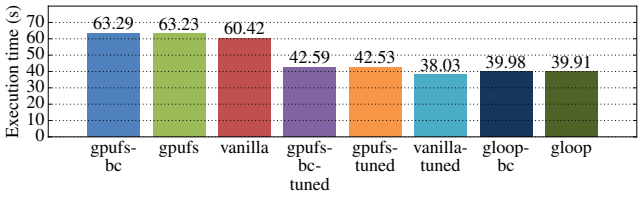


Figure 3: Execution times of grep_text applications. Standard deviation is within 0.1% of mean.

reads.

The workload is executed eleven times, once for warming up and ten times for getting results. The average of the ten times is used in the measurements. All the experiments invoke the CUDA grid, which consists of 28 thread blocks, each of which has 128 threads.

5.1 Case Study: Grep

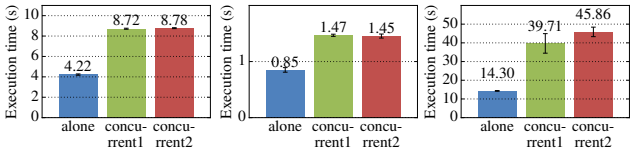
To show that the GLoop programming model is applicable to the existing highly functional GPGPU applications, we port an application *grep_text*, a GPUfs-based application. The source code of GPUfs and its applications is downloaded from the project site[1]. Grep_text is an application that counts the frequencies of English words. It searches for 58,000 words in the complete works of Shakespeare, which is a single 6 MB file.

We successfully ported GPUfs-based grep_text to GLoop. Similarly to the original one, thread blocks independently search for a subset of the 58,000 words. Thread blocks repeatedly read a chunk of text. For each word, thread blocks perform string matching on the chunk to count the word frequencies. If a given word appears more than once, a thread block logs this to an output file. While the original code uses a loop to iterate words, the ported code uses an event loop callback to iterate words. The callback performs string matching of one word on the text chunk and enqueues the callback with the next word to the event loop. This inserts enough scheduling points to the ported program. When string matching succeeds, the program enqueues the callback with the file write operation to log the result. Otherwise, it just enqueues the event loop callback.

To demonstrate the GLoop-aware application's performance, we compare several versions of grep_text's execution time. We prepare GPUfs- and GLoop-based grep_text labeled *gpufs* and *gloop*. For comparison, we also prepare a workload that preallocates large GPU device memory to transfer all the dataset before starting the GPU kernel, named *vanilla*. We prepare gpufs and vanilla optimized for our GPU, each of which are postfixed with *-tuned*. We modify the source code of the downloaded GPUfs to run on our GPU. Specifically, we changed the GPUfs memory pool size and the CUDA malloc heap size limit from 2 GB to 1 GB and from 1 GB to 256 MB, respectively. We also measured the execution time of our grep_text family with cold buffer caches. These are labeled *gpufs-bc*, *gpufs-bc-tuned*, and *gloop-bc*.

The results are shown in Fig. 3. From the graph, we can see that performance of the gloop is comparable to the other grep_text implementations. The gloop outperforms the untuned versions and it is 5% slower than vanilla-tuned and 6% faster than gpufs-tuned. The difference in the execution times under the cold buffer cache is negligible. Gpufs-bc, gpufs-bc-tuned, and gloop-bc are 0.1–0.2% slower than gpufs, gpufs-tuned, and gloop, respectively. Since grep_text repeatedly reads the same set of files, buffer cache miss does not occur except for the first read.

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



(a) throttle (b) copy1GB (c) copy1GB-BC

Figure 4: Execution time of each application. *alone* occupies GPU, while *concurrent1* and *concurrent2* run concurrently on one shared GPU.

5.2 Consolidating Two GPU Applications

To show GLoop interleaves kernel executions, we run two GPGPU applications on a GPU. We prepare three workloads: *throttle*, *copy1GB*, and *copy1GB-BC*. Throttle performs event loop callbacks 100,000 times while copy1GB and copy1GB-BC copy a 1 GB file with and without the warm buffer cache. We first run each workload in the standalone manner (*alone*) and then run two instances of the same workload concurrently (*concurrent1* and *concurrent2*).

The results are shown in Fig. 4. The x-axis is the workload name and the y-axis is the execution time. The execution times of concurrent1 and concurrent2 are almost the same in throttle and copy1GB (8.72s and 8.78s for throttle, 1.47s and 1.45 for copy1GB). In the copy1GB-BC case, the two execution times are different due to I/O time variance. The standard deviations of the copy1GB-BC case are up to 13% of the mean. Since GLoop switches their kernels in a fine-grained manner, the GPGPU applications are not executed serially.

We can also see that the execution times of concurrent1 and concurrent2 in throttle are more than double the execution time of the alone case, 2.07 times and 2.08 times, respectively. This is the overhead caused by suspending and resuming the device event loop to interleave kernel executions. On the other hand, the concurrent execution times in copy1GB are smaller than the doubled alone time, 1.73 times and 1.71 times, respectively. This is because splited thread blocks can utilize GPU computing cores more. In the copy1GB-BC case, the execution times of concurrent1 and 2 become slower than the doubled alone time, 2.78 times and 3.21 times. This performance degradation comes from the disk contention caused by I/O requests of the two kernels.

6. CONCLUSION

This paper presents GLoop, which offers an event-driven programming model for the GPGPU kernel. GLoop suspends and resumes the GPU kernels at the boundary of the callback executions and achieves GPU resource isolation among multiple GPU kernels by establishing a GPU context per kernel.

Designing scheduling policies is one of the most important tasks. To effectively schedule more than two GPGPU applications, scheduling policies such as budget-based policies are needed. To do so, scheduling the utilization of the GPU DMA engines needs to be considered. Applying our programming model to other GPU eaters is also an interesting challenge. For example, we explore a way to port GPUnet-based applications or applications based on the persistent threads model to our model.

7. REFERENCES

[1] Agrawal, S. R., and Lebeck, A. R. 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* (2014), ACM, pp. 19–34.

[2] Foundation, N. Node.js. https://nodejs.org, 2016.

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

[4] Han, S., Jang, K., Park, K., and Moon, S. PacketShader: A GPU-Accelerated Software Router. In *Proc. of the ACM SIGCOMM 2010 Conf.* (2010), ACM, pp. 195–206.

[5] He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., and Sander, P. Relational Joins on Graphics Processors. In *Proc. of the 2008 ACM SIGMOD Int'l Conf. on Management of Data* (2008), ACM, pp. 511–524.

[6] Hirabayashi, M., Kato, S., Edahiro, M., Takeda, K., Kawano, T., and Mita, S. 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* (2013), IEEE, pp. 106–111.

[7] Jang, K., Han, S., Han, S., Moon, S., and Park, K. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation* (2011), USENIX, pp. 1–14.

[8] Kaldewey, T., Lohman, G., Mueller, R., and Volk, P. GPU Join Processing Revisited. In *Proc. of the 8th Int'l Workshop on Data Management on New Hardware* (2012), ACM, pp. 55–62.

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

[10] Kato, S., Lakshmanan, K., Rajkumar, R., and Ishikawa, Y. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *Proc. of the 2011 USENIX Annual Technical Conf.* (2011), USENIX, pp. 17–30.

[11] Kato, S., McThrow, M., Maltzahn, C., and Brandt, S. Gdev: First-Class GPU Resource Management in the Operating System. In *Proc. of the 2012 USENIX Annual Technical Conf.* (2012), USENIX, pp. 401–412.

[12] Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A. D., Kaldewey, T., Lee, V. W., Brandt, S. A., and Dubey, P. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proc. of the 2010 Int'l Conf. on Management of Data* (2010), ACM, pp. 339–350.

[13] Kim, S., Huh, S., Zhang, X., Hu, Y., Wated, A., Witchel, E., and Silberstein, M. GPUnet: Networking Abstractions for GPU Programs. In *Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation* (2014), USENIX, pp. 201–216.

[14] Kyriazis, G. Heterogeneous System Architecture: A Technical Review. In *Technical report* (2012), AMD.

[15] Maruyama, N., Nomura, T., Sato, K., and Matsuoka, S. 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* (2011), ACM, pp. 11:1 –11:12.

[16] McNaughton, M., Urmson, C., Dolan, J. M., and Lee, J.-W. Motion Planning for Autonomous Driving with a Conformal Spatiotemporal Lattice. In *Proc. of the 2011 Int'l Conf. on Robotics and Automation* (2011), IEEE, pp. 4889–4895.

[17] Menychtas, K., Shen, K., and Scott, M. L. 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* (2014), ACM, pp. 301–316.

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

[19] NVIDIA. Multi-Process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2015.

[20] Pai, S., Thazhuthaveetil, M. J., and Govindarajan, R. Improving GPGPU concurrency with elastic kernels. In *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (2013), vol. 41, ACM, p. 407.

[21] Park, J. J. K., Park, Y., and Mahlke, S. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proc. of the 20th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* (2015), ACM, pp. 593–606.

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

[23] Rossbach, C. J., Currey, J., Silberstein, M., Ray, B., and Witchel, E. PTask: Operating System Abstractions To Manage GPUs as Compute Devices. In *Proc. of the 23rd Symp. on Operating Systems Principles* (2011), ACM, pp. 233–248.

[24] Satish, N., Kim, C., Chhugani, J., Nguyen, A. D., Lee, V. W., Kim, D., and Dubey, P. 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* (2010), ACM, pp. 351–362.

[25] Shimokawabe, T., Aoki, T., Takaki, T., Endo, T., Yamanaka, A., Maruyama, N., Nukada, A., and Matsuoka, S. 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* (2011), ACM, pp. 3:1–3:11.

[26] Silberstein, M., Ford, B., Keidar, I., and Witchel, E. GPUfs: Integrating a File System with GPUs. In *Proc. of the 18th Int'l Conf. on*

Architectural Support for Programming Languages and Operating Systems (2013), ACM, pp. 485–498.

[27] Sun, W., Ricci, R., and Curry, M. L. GPUstore. In *Proc. of the 5th Annual Int'l Systems and Storage Conf.* (2012), ACM, pp. 1–12.

[28] Suzuki, Y., Kato, S., Yamada, H., and Kono, K. GPUvm: Why Not Virtualizing GPUs at the Hypervisor? In *Proc. of the 2014 USENIX Annual Technical Conf.* (2014), USENIX, pp. 109–120.

[29] Tanasic, I., Gelado, I., Cabezas, J., Ramirez, A., Navarro, N., and Valero, M. Enabling preemptive multiprogramming on GPUs. In *Proc. of the 41st Int'l Symp. on Computer Architecture* (2014), ACM/IEEE, pp. 193–204.

[30] Tian, K., Dong, Y., and Cowperthwaite, D. A Full GPU Virtualization Solution with Mediated Pass-Through. In *Proc. of the 2014 USENIX Annual Technical Conf.* (2014), USENIX, pp. 121–132.

[31] Wang, L., Huang, M., and El-Ghazawi, T. Exploiting Concurrent Kernel Execution on Graphic Processing Units. In *Proc. of the Int'l Conf. on High Performance Computing and Simulation* (2011), IEEE, pp. 24–32.

[32] Zeno, L., Mendelson, A., and Silberstein, M. 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* (2016), ACM, pp. 63–71.