LoGA: Low-overhead GPU accounting using events

Jens Kehne Stanislav Spassov Marius Hillenbrand
Marc Rittinghaus Frank Bellosa

Karlsruhe Institute of Technology (KIT)
Operating Systems Group
os@itec.kit.edu

ABSTRACT

Over the last few years, GPUs have become common in computing. However, current GPUs are not designed for a shared environment like a cloud, creating a number of challenges whenever a GPU must be multiplexed between multiple users. In particular, the round-robin scheduling used by today's GPUs does not distribute the available GPU computation time fairly among applications. Most of the previous work addressing this problem resorted to scheduling all GPU computation in software, which induces high overhead. While there is a GPU scheduler called NEON which reduces the scheduling overhead compared to previous work, NEON's accounting mechanism frequently disables GPU access for all but one application, resulting in considerable overhead if that application does not saturate the GPU by itself.

In this paper, we present LoGA, a novel accounting mechanism for GPU computation time. LoGA monitors the GPU's state to detect GPU-internal context switches, and infers the amount of GPU computation time consumed by each process from the time between these context switches. This method allows LoGA to measure GPU computation time consumed by applications while keeping all applications running concurrently. As a result, LoGA achieves a lower accounting overhead than previous work, especially for applications that do not saturate the GPU by themselves. We have developed a prototype which combines LoGA with the pre-existing NEON scheduler. Experiments with that prototype have shown that LoGA induces no accounting overhead while still delivering accurate measurements of applications' consumed GPU computation time.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*Multiprocessing/multiprogramming/multitasking, Scheduling*

Keywords

Accounting, event-based, scheduling, GPU

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SYSTOR 2017, May 22-24, 2017, Haifa, Israel

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-5035-8/17/05…$15.00

DOI: https://dx.doi.org/10.1145/3078468.3078473

1. INTRODUCTION

Over the last few years, GPUs have become increasingly common in computing. Especially in the field of High Performance Computing (HPC), GPUs deliver unparalleled levels of performance for certain classes of applications. More recently, the increasing demand for GPUs has prompted cloud providers to integrate GPUs into their cloud offerings as well. Integrating a GPU into the cloud is particularly interesting for users which need only sporadic access to a GPU – and are thus unlikely to saturate an entire GPU by themselves – since consolidating multiple such users on a single GPU allows the cloud provider to increase the GPU's utilization and thus offer GPU access at a lower price. However, today's GPUs are not designed for a shared environment like a cloud. Consequently, a number of challenges arise whenever a GPU must be multiplexed between multiple users.

Traditionally, GPUs were used exclusively by a single application to display graphics on a screen. While GPUs have evolved considerably over the years, current GPUs have inherited the design principles of their ancestors. These characteristics make these GPUs difficult to use in a shared environment like a cloud platform. For example, current GPUs schedule commands – such as kernel launches or DMA transfer requests – from different applications in a round-robin fashion, without preemption and irrespective of the runtime of these commands. This simple scheduling causes applications which submit longer commands to receive more GPU computation time. Therefore, the operating system – or the hypervisor in a virtualized environment – must take additional measures to ensure fairness between applications.

Previous work has produced two models for establishing fairness between multiple applications sharing a GPU: *Host-based software scheduling* and *disengaged scheduling*. Most previous projects [15, 7, 8, 5, 2, 17, 19, 20] utilize host-based software scheduling: A scheduler in the host system selects a single GPU command for execution, and waits for that command to complete before selecting the next command. While this approach can achieve fairness, host-based software scheduling necessarily interferes with or completely disables the GPU's own, highly efficient dispatching and context switching and thereby induces considerable runtime overhead in applications. In contrast, NEON [11] was the first project to employ disengaged scheduling. NEON grants applications direct access to the GPU whenever possible and only interferes with applications' execution when actual scheduling is necessary to ensure fairness. While this approach achieves a much lower overhead than previous designs, NEON's resource accounting mechanism frequently disables

GPU access for all but one application to observe the applications' behavior in isolation. If the applications do not fully utilize the GPU by themselves, this approach to accounting leads to poor utilization, which can manifest as an application overhead of more than 20 % for some applications. Note that we expect individual applications not saturating the GPU to be the common case in cloud environments since applications saturating the GPU by themselves are likely not interested in sharing a GPU with other applications.

In this paper, we present LoGA, a novel accounting mechanism for GPUs which operates in the disengaged scheduling model. LoGA replaces NEON's accounting mechanism while keeping NEON's original scheduler. LoGA monitors the GPU's internal state to detect GPU-internal scheduling events and infers each application's GPU usage from these events. This information can then be used, for example, for billing or scheduling. In contrast to previous work, LoGA measures the applications' GPU usage while all applications are running concurrently, resulting in a higher GPU utilization and thus less overhead. Our experiments with a prototype implementation show that LoGA causes no accounting overhead. At the same time, LoGA achieves a level of fairness comparable to that of NEON overall, and even yields better fairness than NEON for some applications.

The rest of this paper is organized as follows: In Section 2, we present background on GPU hardware and describe NEON's scheduling approach, which forms the primary use case of our work. We give a detailed description of LoGA's design in Section 3. In Section 4, we describe our prototype implementation, before presenting the results of our evaluation of our prototype in Section 5. Finally, we discuss related work in Section 6, before concluding the paper in Section 7.

2. BACKGROUND

Current GPUs typically function as asynchronous computational accelerators. The CPU offloads work – such as CUDA kernels or rendering operations – to the GPU and is then free to perform other tasks while the GPU processes the offloaded work asynchronously. Modern GPUs allow applications to submit work to the GPU without operating system intervention by writing commands into memory-mapped *command submission channels*. We describe this process in more detail in Section 2.1. The GPU then executes the submitted commands and, if requested by the application, signals completion of each command to the submitting application. We present details on GPU command execution in Section 2.2.

Current GPUs typically do not support preemption: Once a command has been submitted, it must run to completion, which greatly complicates scheduling. As a consequence, NEON [11] employs *disengaged scheduling*, a variant of fair queuing [14] which achieves a fair distribution of GPU computation time on average without requiring support for preemption. We describe NEON's approach in Section 2.3.

2.1 Command Submission Interface

Applications typically use command submission channels to submit work to current GPUs. These channels are ring buffers in memory, which contain a queue of high-level commands like kernel launches or DMA transfer requests. Command submission channels can be mapped directly into an application's address space, allowing the application to submit commands without involving the operating system. The command submission process is depicted in Figure 1: The

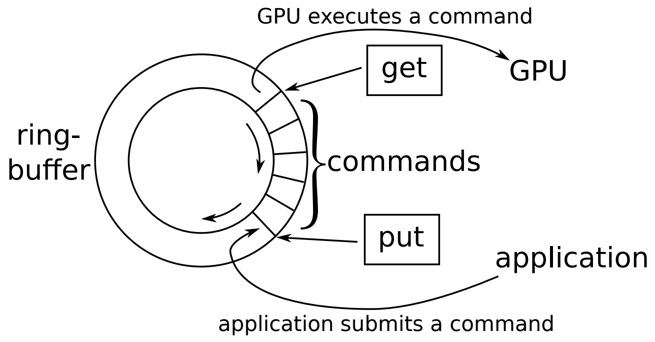


Figure 1: GPU command submission channel. The ring buffer contains a queue of commands awaiting execution. The *get*- and *put*-pointers – which reside in memory-mapped device registers – point to the current head and tail of the queue. (© 2013 IEEE. Reprinted, with permission, from [4].)

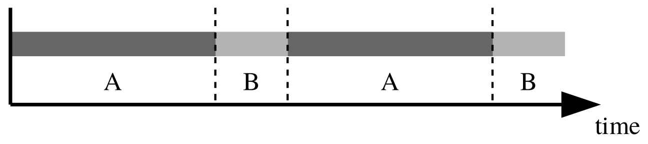


Figure 2: Example of a GPU's internal round-robin scheduling. Process A submits longer-running commands than process B and thus receives more computation time.

application writes commands into the ring buffer and advances the *put*-pointer, which resides in a memory-mapped device register, to inform the GPU that a command has been submitted. The GPU in turn fetches commands from the ring buffer, advancing the *get*-pointer after fetching a command. Current GPUs support a large number of such command submission channels, allowing the driver to assign a dedicated channel to each application.

2.2 GPU Command Execution

The actual command execution is carried out by several hardware *engines*. The first of these engines is a *dispatcher*, which fetches commands from the command submission channels, decodes these commands and directs them to the other engines for execution. If no appropriate engine is available for a command, the dispatcher delays fetching of further commands until an engine is ready to accept the stalled command. Most of the actual command execution is then performed by two other engines: A *compute engine* executes all actual computation – such as CUDA kernels or rendering operations – as well as synchronous DMA operations, while a *DMA engine* handles all asynchronous DMA operations. There is also a number of other engines dealing with more specialized operations such as video decoding; however, we omit descriptions of these engines as they are not relevant in the context of this paper.

The dispatcher performs the GPU's internal command scheduling and context switching by fetching commands from all active channels in a round-robin fashion. Both scheduling and context switching are implemented inside the GPU and therefore cause virtually no runtime overhead. However, since GPU commands are not preemptible, such simple round-robin scheding inherently favors applications submitting long-running commands. For example, consider

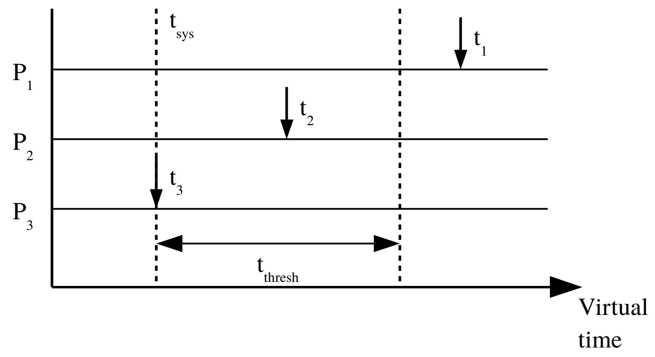


Figure 3: Fair queuing applied to GPU kernel execution. NEON maintains a virtual time for each application as well as a system-wide virtual time t_{sys}. Applications are allowed to run while their virtual time is less than t_{thresh} ahead of t_{sys}. In this example, t_1 would be suspended, while t_2 and t_3 would be allowed to run.

two applications submitting commands of different length to a GPU as depicted in Figure 2. Since the GPU alternates between commands from both applications irrespective of each command's runtime, application A receives a larger amount of computation time than application B.

Once the dispatcher has fetched a command from a channel, that command must typically run to completion. Any event preventing a command from executing to completion constitutes a fatal error which typically causes the command to fail altogether. While the latest generation of GPUs can tolerate certain interruptions in kernel execution – such as page faults – there is currently little information available about the extent of these features or on how to exploit them to improve GPU scheduling.

2.3 Disengaged Scheduling

One way of scheduling GPU command execution is to employ a variant of fair queuing [14], which is depicted in Figure 3. To implement fair queuing for GPUs, the scheduler maintains a counter of consumed GPU time for each application (t_1 to t_3). Whenever an application executes a command on the GPU, the scheduler adds the runtime of that command to the counter of the corresponding application. In addition, the scheduler maintains a global system time (t_{sys}), which is always equal to the timer value of the application that has consumed the least amount of GPU time so far. The scheduler then allows all applications direct access to the GPU as long as their timer values are smaller than the sum of t_{sys} and a configurable value (t_{thresh}). Conversely, the scheduler suspends GPU access for all applications getting too far ahead of t_{sys}. While this scheme does not eliminate unfairness, it effectively limits the amount of unfairness permitted in the system.

A major obstacle in implementing fair queuing for GPUs is the GPUs' asynchronous nature: The scheduler must detect the start and finish times of each command to accurately measure the GPU computation time consumed by each application, which is difficult since GPU commands start and finish asynchronously. To detect command start times, NEON traps each GPU command submission by mapping the application's command submission channels read-only, prompting a page fault on each command submission. When the application subsequently attempts to submit a command, the page fault handler records the current time as the new command's start time and forwards the command to the channel. To detect finish times, NEON programs the GPU to issue an interrupt on each command completion, and records the command's finish time in the interrupt handler. This method causes a high number of page faults and interrupts, leading to a significant application overhead. In addition, trapping command submissions using page faults can lead to inaccurate accounting data since commands do not necessarily begin executing immediately after submission.

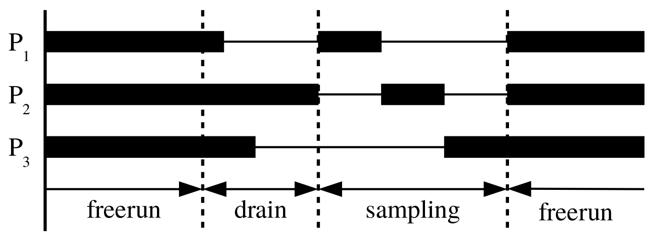


Figure 4: GPU time accounting in NEON. At the end of the freerun phase, NEON suspends all applications and waits for their remaining commands to finish. Then, each application runs alone for a short interval while the scheduler profiles the application's commands.

To overcome this problem, NEON [11], which first implemented fair queuing for GPUs, does not employ precise accounting, but instead approximates each application's GPU time consumption. As depicted in Figure 4, NEON alternates between a *sampling phase* in which the scheduler measures the applications' GPU usage and computes scheduling decisions, and a *freerun phase* in which all applications have direct access to the GPU and no accounting or scheduling takes place. During the sampling phase, the scheduler unmaps all command submission channels from all applications, and allows these channels to run empty *(draining phase)*. Then, the scheduler allows each application to execute commands on the GPU for a configurable amount of time – typically a few milliseconds – while all other applications remain suspended. While each application is running, the scheduler traps all command submissions and polls for command completions at a default rate of 1 kHz to determine the runtime of each command. At the end of each sampling phase, the scheduler computes the average command runtime for each application, and uses these runtimes to estimate each application's GPU time consumption during the previous freerun phase. This estimate is then used to update each application's counter for the fair queuing algorithm. During the following freerun phase, all applications not found to over-use the GPU have direct access to their command submission channels, while channels of over-using applications remain unmapped.

NEON's estimation-based approach significantly reduces scheduling overhead compared to previous work, while achieving good fairness for GPU applications. However, NEON still suffers from an important drawback: NEON's accounting mechanism only executes one application at a time during sampling phases. If that one application does not fully saturate the GPU, this approach results in the GPU frequently falling completely idle during the sampling phase, resulting in considerable application overhead. Note that we expect a single application not saturating the GPU to be a fairly common scenario in a cloud environment, since applications which can saturate a GPU by themselves are likely not interested in sharing a GPU with other applications.

3. DESIGN

To minimize the accounting overhead, LoGA aims to keep the GPU utilized at all times. To that end, we restrict ourselves to passive monitoring of the GPU's state as long as actual scheduling is not strictly necessary.

In this section, we first present the goals of our design in Section 3.1. Next, we give an overview of our design in Section 3.2. Finally, we describe our accounting mechanism and our scheduler in more detail in Section 3.3 and 3.4, respectively.

3.1 Design Goals

While NEON does well at fairly scheduling GPU computation, NEON's accounting mechanism can cause significant application overhead by frequently disabling GPU access for most running applications even if no scheduling is required. Therefore, LoGA's main objective is to reduce this accounting overhead without sacrificing scheduling quality. More specifically, we formulate two main goals for LoGA:

Performance: Applications should not have to pay for improved scheduling with reduced performance. As a consequence, LoGA should keep applications running whenever possible to maintain a high GPU utilization at all times.

Fairness: When using LoGA, the scheduler should achieve at least the same level of fairness as when using NEON's original accounting mechanism.

3.2 Design Overview

LoGA measures the applications' GPU time consumption by monitoring status registers reflecting the internal state of the GPU and deducing the amount of GPU computation time consumed by each application from changes in that state. When LoGA detects an application over-using the GPU, the scheduler temporarily suspends GPU access for that application to ensure that other applications receive a fair amount of GPU time as well.

In contrast to NEON, LoGA disables an application's GPU access – and thus causes overhead – only *after* an application has been found to over-use the GPU. All applications not over-using the GPU keep running concurrently as shown in Section 5.3, which keeps GPU utilization high. In addition, LoGA is able to accurately measure each application's GPU time consumption even while all applications are active concurrently on the GPU. The scheduler can thus make scheduling decisions as accurate as – and in some cases, even more accurate than – NEON, as shown in Section 5.5.

LoGA's architecture consists of two main components as depicted in Figure 5: An *accounting thread* which monitors each application's resource usage, and a *scheduler* which makes scheduling decisions based on the information collected by the accounting thread. The accounting thread periodically polls status registers of the GPU's compute engine in order to detect GPU-internal context switches, and sums up the time between these context switches to compute the total amount of GPU time consumed by each application. The scheduler periodically requests each application's consumed GPU time from the accounting thread and uses this information as input for the disengaged fair queuing algorithm introduced by NEON. Whenever the scheduling algorithm decides that an

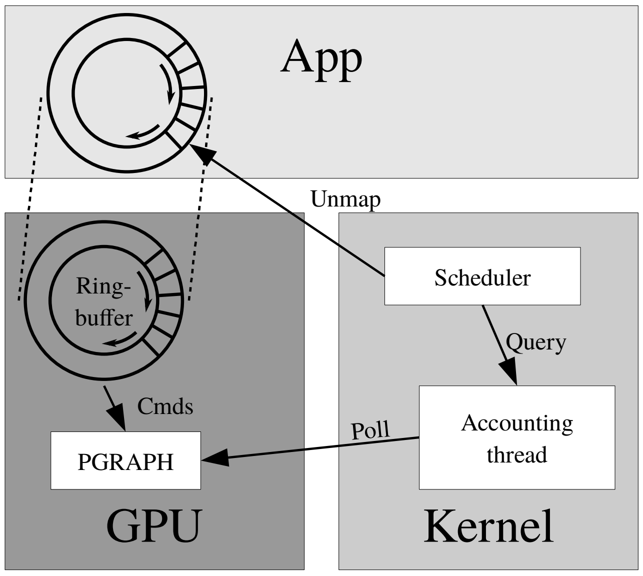


Figure 5: Basic architecture of LoGA

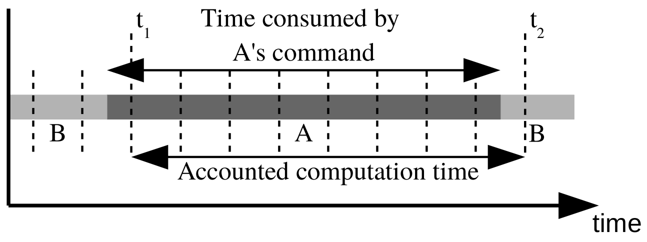


Figure 6: Computation time measurement using context switches. The status registers are read at each of the dashed vertical lines. Context switches are detected at t_1 and t_2, and the runtime of A's command is computed as $t_2 - t_1$.

application is over-using the GPU, the scheduler temporarily suspends GPU access for that application.

3.3 Computation Time Accounting

To measure the amount of GPU time used by each application without negatively affecting GPU utilization, the accounting thread polls a GPU status register which identifies the command submission channel the currently executing command originated from. Whenever the content of that register changes – which indicates a context switch – the accounting thread records the time since the last context switch as GPU computation time used by the application that ran before the context switch. An example of our approach is shown in Figure 6: The accounting thread detects a context switch from B to A at time t_1, followed by a context switch from A to B at time t_2. Consequently, the time between the two context switches must have been consumed by A's command. Note that context switches are not visible to the accounting thread until the next time the status register is read. This delay has two implications: First, t_1 and t_2 may be delayed from the actual context switches by the time it takes to read a device register. In practice, however, these delays do not cause significant inaccuracy in the measured command runtimes since i) the delays are limited to at most one iteration of the polling loop, which is about 1 μs on our hardware, and ii) the delays appear at both the beginning and end of each command and thus cancel out on average. Second, LoGA may not detect commands with an execution time shorter than one polling loop iteration. While this situation does occur for some short setup commands, the runtime of a GPU kernel is typically much longer than one loop iteration. In practice, we therefore do not expect this problem to cause significant inaccuracy.

Polling the GPU's status registers allows LoGA to account GPU computation time without disturbing the GPU execution of any application: Reading a status register does not cause any interruptions in GPU computation. However, high-frequency polling inevitably causes high CPU overhead. In order to reduce that overhead, the accounting thread does not poll the GPU constantly, but in periodic intervals: A short polling phase is followed by a longer phase without polling. Since application behavior is not necessarily constant over time, the accounting thread recomputes each application's consumed GPU time after each polling phase. However, this periodic polling results in a trade-off between CPU overhead and accounting accuracy: The longer the polling phase is compared to the non-polling phase, the greater both accuracy and CPU overhead become. However, it is important to note that this trade-off depends only on the ratio – but not on the absolute lengths – of the two intervals. Specifically, if both intervals are scaled by the same amount, neither accuracy nor CPU overhead will change significantly.

While the absolute interval lengths do not affect the trade-off between accuracy and overhead, the polling phase must be long enough to observe a large enough number of context switches. With a fixed interval length, however, starting a large number of applications leads to fewer context switches per application during each interval. Consequently, our accounting thread increases the length of both intervals linearly with the number of GPU applications in the system. In addition, if a single command runs longer than the length of the polling phase, LoGA observes no context switches at all during that polling phase. In that case, LoGA accounts the entire length of the polling phase as GPU time consumed by the application running at the end of the polling phase.

An additional disadvantage of fixed interval lengths is that the accounting thread might sample periodic application behavior. To mitigate this issue, the accounting thread can randomize the lengths of both intervals, while keeping the average interval lengths equal to the current, fixed lengths.

3.4 Compute Kernel Scheduling

Our scheduler is based on the disengaged fair queuing algorithm first introduced in NEON [11]. The scheduler maintains a *virtual time* for each application that is using the GPU. At the end of each polling phase, the scheduler queries each application's GPU time consumption and uses this value to advance the application's virtual time. To that end, our scheduler performs two steps: First, it computes the total amount of GPU time consumed by all applications (t_{total}) during the polling phase. Then, it advances each application's virtual time by the proportion of that application's consumed GPU time (t_{app}) during the polling phase to the GPU time consumed by all applications combined (t_{total}), scaled by the sum of the lengths of both polling (t_{poll}) and non-polling phase ($t_{non-poll}$):

$$\Delta vt_{app} = \frac{t_{app}}{t_{total}} \times (t_{poll} + t_{non-poll})$$

For example, if an application has consumed half of the total amount of GPU time, the scheduler advances that application's virtual time by half of the sum of the lengths of the polling and the non-polling phase.

After updating all applications, the scheduler updates the *system time* (t_{sys}) to the smallest virtual time in the system. Then, the scheduler compares each application's virtual time to t_{sys}. If an application is found to be ahead of t_{sys} by more than ($t_{poll} + t_{non-poll}$), the scheduler suspends GPU access for that application for the next non-polling phase.

To prevent applications from "saving up" GPU time, the scheduler ignores applications that were not active in the preceding polling phase when updating t_{sys}. In addition, if an idle application's virtual time falls behind t_{sys}, the scheduler advances that application's virtual time to t_{sys}.

4. IMPLEMENTATION

To show the viability of our approach, we integrated a prototype implementation of LoGA into the NEON kernel module, which is placed between the application and Nvidia's proprietary GPU driver. The module intercepts calls from the application to the GPU driver, tracks which command submission channel belongs to which application, and forwards all calls to the GPU driver. The module's operation is thus completely transparent to the applications. Since NEON comes with support for multiple, pluggable scheduling policies, we implemented our prototype as an additional policy, called the *event policy*.

Integrating LoGA into the NEON module allows us to re-use NEON's infrastructure without modification. Our event policy is based on NEON's original sampling policy, but replaces the sampling policy's accounting mechanism with LoGA. Our implementation of disengaged fair queuing re-uses most code from the sampling policy's original implementation, with only small modifications to integrate the algorithm with LoGA.

The rest of this section is organized as follows: In Section 4.1, we present the implementation of our accounting mechanism, before describing the implementation of our scheduler in Section 4.2. Finally, we describe our mechanism for suspending GPU applications in Section 4.3.

4.1 Computation Time Accounting

To account consumed GPU computation time, LoGA must detect GPU-internal context switches with high accuracy. To that end, LoGA launches a kernel thread for each GPU in the system. During polling phases, this thread polls a GPU status register indicating which command submission channel is currently executing commands on the GPU. Any change in that status register denotes a context switch, prompting LoGA to account the time since the last context switch to the last application to run. We found appropriate status registers to be present in all recent Nvidia GPUs.

Current GPUs typically contain more than one status register identifying the "current" command submission channel. Each of the GPU's engines typically contains a status register reflecting the current command submission channel from that engine's point of view. Since our main focus is on GPU computation, LoGA uses the compute engine's status registers, which we found to correlate best with the kernel runtimes seen by applications.

An issue during implementation was that the status registers only reflect the identity, but not the status of the current channel. As a consequence, LoGA cannot determine whether the current channel is busy or idle from the status register alone. To detect idle channels, LoGA polls the current channel's entry in the GPU's *channels table* – an in-memory, GPU-managed data structure containing information about all allocated channels – in addition to the status register. The channels table contains a status bit for each channel, which indicates whether this channel is currently active – that is, whether this channel is either executing a command, or contains commands awaiting execution. If no commands are pending for the current channel seen by the compute engine, LoGA considers the GPU to be idle and does not account any computation time.

4.2 Compute Kernel Scheduling

Our scheduler runs in the context of NEON's main thread. This thread is periodically woken by a timer. In addition, we wake the main thread whenever an application creates or destroys a channel since these events may necessitate a scheduling decision. On each wakeup event, the thread performs two main tasks: i) coordinate the accounting threads, and ii) compute scheduling decisions.

Each wakeup event induces a transition between the polling and non-polling phases, which the main thread must communicate to the accounting threads. At the end of each polling phase, the main thread signals the accounting threads to finish polling, and then waits for all accounting threads to acknowledge that signal to ensure that the accounting information is up to date before computing a scheduling decision. At the end of each non-polling phase, the main thread signals the accounting threads to resume polling, but does not wait for that signal to be acknowledged. While omitting the acknowledgment causes the length of the polling phase to vary slightly, this variance is tolerable since LoGA uses the actual length of the polling phase in all computations.

4.3 Suspending Applications

Our scheduler currently suspends GPU access for applications by unmapping all command submission channels from the application's address space. To unmap a channel, our scheduler disables the valid bit of the channel's page table entry and flushes the corresponding TLB entries. If the application attempts to access an unmapped channel, the page fault handler blocks the accessing thread on a per-application semaphore until the scheduler restores GPU access for the application. To restore GPU access for the application, the scheduler switches the valid bit in the appropriate page table entry back on and unblocks all threads waiting on the application's semaphore.

Current GPUs cannot stop the execution of commands after these commands have been submitted to a channel. Therefore, our scheduler must allow any commands already present in the channel to execute to completion after the channel has been unmapped. However, any computation time consumed by these commands will be accounted to the application owning the channel, which potentially prolongs the time the application remains suspended later on.

5. EVALUATION

To verify that LoGA reduces the overhead of GPU accounting, we conducted experiments with our prototype implementation. In this section, we present the results of these experiments. First, we describe our experimental setup in Section 5.1 and our application for generating defined GPU loads in Section 5.2. In Section 5.3, we show that LoGA reduces the overhead of GPU computation time accounting compared to previous work. Next, we address the accounting accuracy of LoGA: We show that our approach is able to measure consumed GPU time with high accuracy in Section 5.4, before showing that LoGA achieves an end-to-end scheduling quality comparable to that of previous work in Section 5.5. Finally, we address the CPU overhead caused by LoGA's accounting mechanism in Section 5.6.

5.1 Experimental Setup

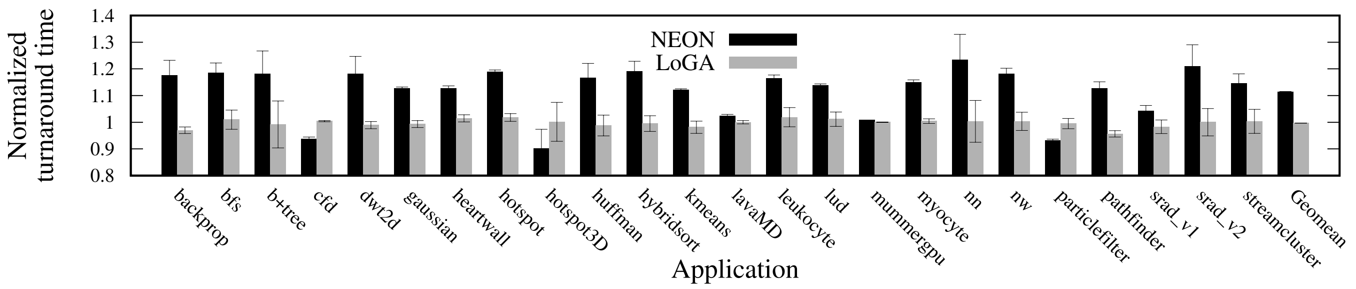
In our experiments, we used an Nvidia GeForce GTX Titan Black, which is based on the Kepler microarchitecture and contains 2880 CUDA cores and 6 GiB of GDDR5 memory. Besides the GPU, our test system contains two Xeon E5-2620 CPUs for a total of 24 cores and 32 GB RAM. Our test system runs Linux 3.4.7, version 331.62 of Nvidia's proprietary GPU driver and CUDA 6.0. For our experiments, we used version 3.1 of the Rodinia benchmark suite [3].

Unfortunately, most of Rodinia's applications execute only one iteration of their main algorithm. However, since both NEON and LoGA both aim to correct imbalance after an application has been found to over-use the GPU, executing only one iteration does not give the scheduler a chance to correct any imbalance it may have detected during the first iteration. Therefore, we modified those Rodinia applications that do not execute multiple iterations to execute both their GPU kernels and their I/O multiple times. While we did not use one fixed iteration count for all applications, we generally aimed for a total run time of a few seconds, which is sufficient for the scheduler to correct any detected imbalance.

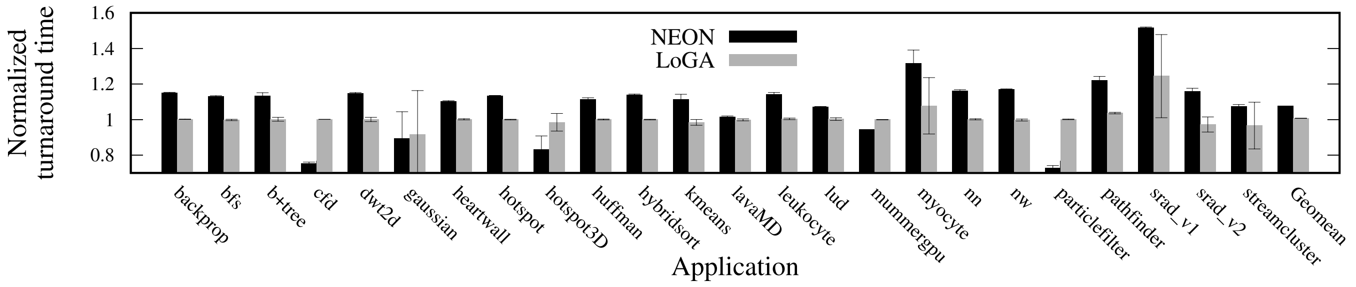
The numbers reported are the average of 10 executions, with the error bars indicating the standard deviation. We repeated each experiment with both our event policy and NEON's original sampling policy enabled, as well as the application running directly on top of the GPU driver without accounting or scheduling. We configured both schedulers to use a polling phase of 1 ms and an non-poll phase of 5 ms per allocated command submission channel. While a polling phase of 1 ms may seem rather short, we found CUDA 6.0 to allocate 12 command submission channels per application, even though only one is actually used. Since the length of the polling phase increases with the number of channels, this behavior results in the polling phase being long enough to accurately measure the application's behavior.

5.2 Throttle

For our experiments, we created a *Throttle* benchmark similar to that used in the evaluation of NEON. Throttle allows us to create different load patterns on the GPU and study the effects these load patterns have on other applications. Throttle launches small CUDA kernels which execute a tight loop for a configurable amount of time. While one of Throttle's kernels is executing, GPU execution is blocked for other applications since GPU kernels are non-preemptive and the GPU used in our experiments cannot execute kernels from different applications concurrently. After running each kernel, throttle can optionally sleep for a configurable amount of time before submitting the next kernel. Sleeping between kernel launches allows throttle to simulate applications which do not saturate the GPU by themselves.



(a) Each throttle instance launches kernels of 10 μs, followed by a sleeping period of 1000 μs.



(b) Each throttle instance launches kernels of 100 μs, followed by a sleeping period of 1000 μs.

Figure 7: Turnaround times of a single instance of various OpenCL applications running concurrently with ten instances of throttle for both LoGA's event policy and NEON's sampling policy, normalized against the turnaround time without any accounting. Scheduling is disabled in all cases. The rightmost bars show the geometric mean of all applications.

5.3 Accounting Overhead

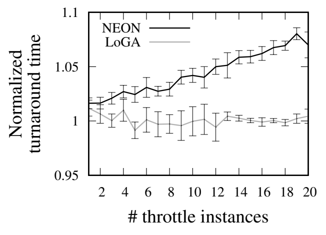
In our first experiment, we show that LoGA's accounting mechanism induces no measurable overhead. To that end, we started a single instance of each of our benchmark applications concurrently with ten instances of throttle, and recorded the turnaround time per iteration. Since NEON's accounting mechanism is especially problematic for applications that do not saturate the GPU by themselves, we configured each throttle instance to launch kernels of 10 μs, followed by a sleeping period of 1000 μs. Mathematically, the throttle instances thus create a GPU load of 9.9 %. Since we found scheduling decisions to mask the accounting overhead, we disabled all scheduling in this experiment, leaving only the accounting mechanism of each policy active.

Figure 7a shows each application's turnaround time under both LoGA's event policy and NEON's sampling policy, normalized against the turnaround time without any accounting. The results indicate that the event policy indeed achieves a higher GPU utilization – and thus causes less accounting overhead – than the sampling policy. In fact, the turnaround time increases by 11.4 % on average for the sampling policy, with the worst case (nn) being as high as 23.4 %. In contrast, the average turnaround time mathematically decreased under LoGA, although not by a statistically significant amount. These results are consistent with our expectations: Since the throttle instances do not saturate the GPU by themselves, disabling all but one application during sampling phases forces the GPU to idle most of the time, resulting in a high overhead for the sampling policy. In contrast, the event policy does not suspend GPU access for applications during sampling phases and thus achieves a higher GPU utilization – and therefore induces less overhead – during these phases.

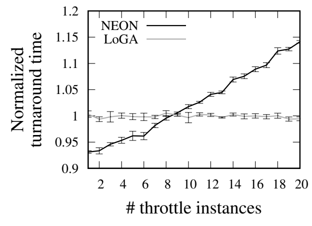
We repeated the previous experiment with an increased throttle kernel length of 100 μs, again followed by a 1000 μs sleeping period. Throttle thus caused a total GPU load of 90.9 % in this experiment. The results, which are shown in Figure 7b, show that our event policy still causes less overhead than the sampling policy: The average increase in turnaround time caused by the sampling policy was 7.6 %, while the event policy showed an average increase of only 0.7 %. Finally, we also measured the overhead for applications saturating the GPU, but found no significant difference between NEON and LoGA in this case. We therefore omit these results for brevity.

In our next experiment, we show that LoGA's overhead – or lack thereof – is independent of the number of concurrent applications. To that end, we launched each of our benchmark applications concurrently with a varying number of throttle instances. We configured each throttle instance to launch kernels of 10 μs, followed by a 1000 μs sleeping period. As in the previous experiment, we disabled all scheduling to avoid masking the accounting overhead. To make the accounting overhead visible, we normalized the benchmark turnaround times under NEON and LoGA against the turnaround time without any accounting or scheduling.

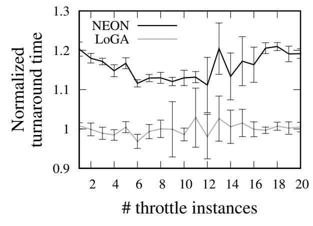
Figure 8 shows the normalized turnaround times for four of our benchmark applications running concurrently with one to 20 instances of throttle. We found most of our benchmark applications to behave like heartwall, which is shown in Figure 8a. For these applications, the turnaround time under NEON increases with the number of concurrent throttle pro-



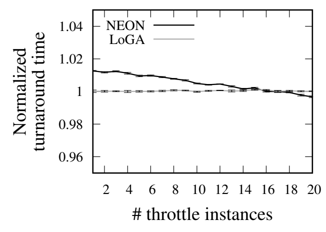
(a) heartwall



(b) lavaMD



(c) dwt2d



(d) mummergpu

Figure 8: Overhead of NEON and LoGA for four applications running concurrently with various numbers of throttle instances

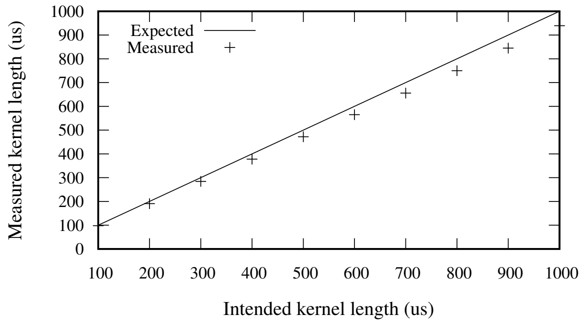


Figure 9: Intended vs. measured kernel length of throttle. Note that throttle waits for each kernel to complete before submitting the next one, which slightly increases the kernel length.

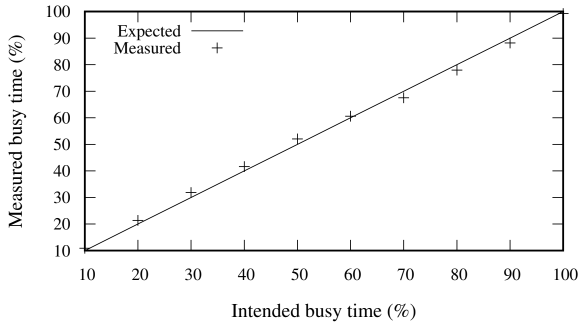


Figure 10: Intended vs. measured GPU load. The numbers indicate the fraction of time that the GPU is busy.

cesses, indicating an increase in NEON's accounting overhead for larger numbers of concurrent applications. The reason for this result is that the length of the sampling phases increases with the number of concurrent applications. Since NEON disables all but one application during sampling phases, the GPU is forced to idle most of the time during sampling phases. Therefore, increasing the length of NEON's sampling phase directly translates to more overhead. In contrast, LoGA keeps all applications running concurrently during sampling phases and therefore shows no significant overhead, irrespective of the number of concurrent applications. An interesting variant of NEON's behavior is shown in Figure 8b: For lavaMD, the turnaround time under NEON is initially even lower than without any accounting mechanism enabled, indicating that letting lavaMD run alone from time to time is advantageous for this application. However, as the number of throttle processes increases, the resulting increase in idle time during sampling phases results in the turnaround time under NEON becoming larger than that under LoGA.

While most applications showed results consistent with our expectations, some applications exhibited different behavior. For dwt2d, which is shown in Figure 8c, the turnaround time initially decreases for small numbers of throttle instances, and rises again for larger numbers of throttle instances. Bfs, gaussian and huffman exhibited similar behavior. However, NEON's turnaround times for all of these applications were consistently higher than LoGA's. Finally, mummergpu's turnaround times, shown in Figure 8d, decreased with the number of concurrent throttle instances when running under NEON, even becoming smaller than those under LoGA for more than 16 instances of throttle. Myocyte and nn showed a similar decrease in turnaround time under NEON, though for both of these applications, NEON's turnaround times were consistently higher than LoGA's.

5.4 Accounting Quality

Our next experiments address the quality of the GPU usage data collected by LoGA. To determine if our method of polling status registers is capable of measuring consumed GPU computation time with high accuracy, we used LoGA to measure the kernel runtimes of our throttle application. Using throttle, we can exactly control the length of the submitted kernels, which allows us to easily verify whether the results obtained by LoGA are correct. Since our initial experiments are only concerned with the general feasibility of our method, we configured LoGA to poll 100 % of the time to eliminate any estimation inaccuracy.

In our first experiment, we show that LoGA can measure an application's kernel length with sufficient accuracy. To that end, we executed a single throttle instance generating a GPU load of 100 % without any other applications accessing the GPU, while increasing throttle's kernel length from 100 to 1000 μs in 100 μs increments. We ran each kernel length ten times, sampling for 10 seconds in each run. The results, shown in Figure 9, indicate that LoGA underestimates throttle's kernel length by up to 6 %. This error is caused by the fact that LoGA must read GPU state from both the GPU's dispatcher and the compute engine. However, these two engines sometimes reflect context switches at different times, which causes some inaccuracy. The relative error appears to be independent of the kernel length and thus should affect

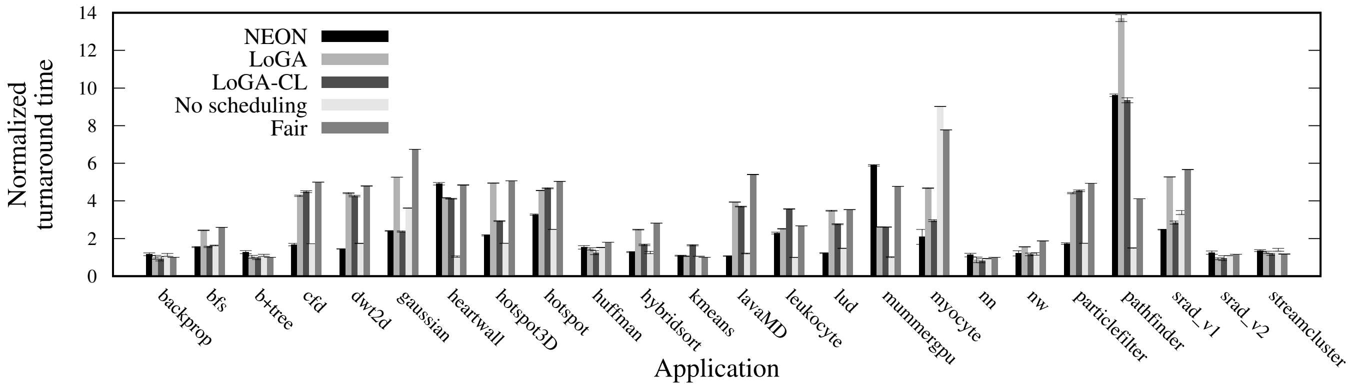


Figure 11: Normalized runtimes of various applications running under NEON, LoGA and without any scheduling. LoGA-CL approximates NEON by using the average time between context switches as a metric for the GPU load caused by each application, while LoGA uses the total time consumed per sampling phase. “Fair” indicates the application’s runtime if the distribution of GPU time was completely fair. Note that faster is not better in this experiment, as a faster turnaround time for the benchmark application indicates that throttle is slowed down more than it should be.

all applications equally. We therefore do not expect this type of error to have a significant impact on scheduling quality.

In a second experiment, we show that LoGA is able to measure the average load of the GPU. To that end, we inserted idle time between kernel launches to create a defined load on the GPU. Specifically, we configured throttle to run in cycles of 1000 μs, with each cycle containing exactly one GPU kernel. We then increased the length of that kernel from 100 to 1000 μs in 100 μs increments, while keeping the GPU idle during the rest of the cycle. As in the previous experiment, we executed each step 10 times, each time sampling for 10 seconds. The results, shown in Figure 10, show that the GPU load measured by LoGA is within 2.5 % of the intended load in all cases. We therefore conclude that our basic approach to measuring GPU time is feasible.

5.5 Scheduling Quality

In our final experiment, we show that a LoGA-based scheduler can achieve the same level of fairness as a NEON-based one. To that end, we ran four instances of throttle, each configured to execute kernels of 200 μs followed by 800 μs of idle time, concurrently with each of our benchmark applications. With this setup, all applications combined come close to fully utilizing the GPU, while each throttle instance by itself generates only moderate load. We then measured the turnaround time for each application, and normalized that turnaround time against the turnaround time of the same application running alone on the GPU under the same accounting mechanism to factor out the accounting overhead. As in the previous experiments, the numbers reported are the average of 10 runs. The results of this experiment are shown in Figure 11.

As a first step in this experiment, we quantified what the turnaround time of each application should be if GPU time was fairly distributed. To that end, we used LoGA – polling 100 % of the time as in Section 5.4 – to measure the total amount of GPU time received by the application itself and each throttle instance over the entire runtime of the application when no scheduling is taking place. From these GPU times, we calculated how much the scheduler should slow down either the application or throttle to make throttle receive the same amount of GPU computation time as the application over the course of the application’s runtime. For simplicity, our calculation assumes that each application by itself saturates the GPU 100 % of the time. As a result, our calculation tends to overestimate the change needed for a fair distribution of GPU computation time since those parts of the application that do not run GPU computation – such as I/O or CPU computation – are not slowed down if the application’s GPU access is suspended. Finally, we multiplied the calculated slowdown with the application’s measured turnaround time without scheduling to obtain an “optimal” turnaround time for each application. We show the optimal turnaround time as “Optimal” in Figure 11.

Finally, we ran each application under both LoGA and NEON, as well as without scheduling. We enabled estimation-based accounting, with a poll phase of 1 ms and a non-poll phase of 5 ms per allocated command submission channel, for both accounting mechanisms. The results – labeled “LoGA”, “NEON” and “No scheduling”, respectively – show that our LoGA-based scheduler brings the turnaround time of all applications closer to the calculated optimum, but in most cases keeps the application’s runtime below the calculated value. This behavior is expected since our calculated optimum likely overestimates the change necessary for a fair distribution of GPU computation time. NEON’s results are similar to LoGA’s in most cases; however, there are many cases where NEON either did not detect as much imbalance as LoGA (bfs, cfd, hotpot, hotspot3D, hybridsort, particlefilter) or even slowed down the wrong application (dwt2d, gaussian, lavaMD, lud, srad_v1). Overall, we conclude that the accounting data produced by LoGA is at least as good as – and likely better than – that produced by NEON’s original accounting mechanism.

One major difference between NEON and LoGA is that both use different metrics to estimate the applications’ GPU time consumption: NEON uses the application’s average command length, while LoGA uses the application’s total amount of consumed GPU time per poll phase. To investigate whether the choice of metric is in fact the cause of NEON's bad results, we created a version of LoGA that uses the average time between context switches as an approximation of the average command length. The results for this version of NEON – labeled "LoGA-CL" in Figure 11 – show that using the average time between context switches indeed yields results similar to those of NEON in many – albeit not all – cases. However, even using the average time between context switches, LoGA is less prone to slowing down the wrong application than NEON. In addition, there are applications where the results for LoGA-CL are almost identical to those of LoGA. Overall, the total amount of consumed GPU time per poll phase appears to be a better metric for the application's GPU usage than the average command length, although the choice of metric does not explain all the differences between NEON and LoGA. Another difference between NEON and LoGA is that NEON only observes applications running in isolation, while LoGA observes all applications running concurrently. For CPUs, it has been shown that applications can exhibit dramatic changes in behavior when running concurrently with other applications [13]. While the same likely happens on the GPU, the resulting changes in kernel runtime are invisible to NEON since NEON only sees the applications running alone.

While LoGA improves the accuracy of GPU computation time measurements, it is important to remember that even if the accounting data is perfect, a scheduler may still make wrong decisions based on that data. In our current implementation, the scheduling algorithm is essentially the same as in NEON. Since our focus is on accounting rather than scheduling, we see the scheduling algorithm as outside the scope of this paper. However, we plan to further address the GPU scheduling problem in the future.

5.6 CPU Overhead

While polling delivers accurate measurements of GPU time consumption, it also causes non-trivial CPU overhead. With the non-poll phase being five times as long as the polling phase, LoGA should keep a single core busy 1/6 – or 16.7 % – of the time. In practice, we measured an overhead of 17 %; we attribute the remaining 0.3 % to computing of scheduling decisions. In contrast, NEON uses low-frequency polling, which is implemented using high-resolution timers rather than spinning. As a result, NEON's CPU load is virtually non-existent – in fact, we were unable to separate it from the regular background noise of the operating system.

Since LoGA's CPU overhead was higher than that of NEON, we expect that it will be necessary to set aside a dedicated core for LoGA. However, doing so becomes increasingly feasible as we move into the manycore era. In addition, we expect one such dedicated core to be sufficient since a single CPU core can handle multiple GPUs by visiting these GPUs in a round-robin fashion, with exactly one GPU being inside the poll phase at any given time. Using the default parameters – i.e., with the non-poll phase being five times as long as the poll phase – this scheme allows one core to handle six GPUs without any loss in accounting quality.

6. RELATED WORK

Several research projects have addressed the problem of fairly sharing GPU computation time. PTASK [15], GERM [2], TimeGraph [7], Gdev [8], and Pegasus [5] schedule GPU commands in the kernel, but define new APIs that the application must support. VGRIS [20] instead intercepts user commands in a modified OpenGL library, thus requiring no modifications to the application. GPUvm [17] and gVirt [19] enable full virtualization of GPUs while maintaining fairness and are thus completely transparent to applications. However, all of these projects implement GPU scheduling and context switching in software, causing frequent interruption in GPU program execution and thus inducing high application overhead.

NEON [11] comes closest to our own philosophy and forms the basis of our work. While previous work scheduled work on the GPU much like on another kind of CPU, NEON treats GPUs as the independent accelerators they are. Consequently, NEON's disengaged fair queuing strategy allows the GPU a high degree of independence, which is consistent with our own assumption that interference with the inner workings of the GPU should be avoided to minimize application overhead. However, NEON's accounting mechanism still interrupts GPU applications frequently, inducing considerable runtime overhead in applications. Our work instead integrates NEON's disengaged fair queuing algorithm with an interruption-free accounting mechanism which causes no application overhead unless actual scheduling is required.

The idea of indirectly monitoring events and inferring accounting information from these events is not a new one. In the past, the same principle has been applied to various problem areas, such as performance profiling [1], energy accounting [12] or scheduling [18]. To our knowledge, however, we are the first to apply this principle to computation time accounting for GPUs. Likewise, the concept of extrapolating total resource usage from samples has been used, for example, to measure the resource consumption of virtual machines [6].

There have also been several research projects which address GPU applications sharing resources other than computation time, such as networking [10], GPU memory [9] and secondary storage [16]. These projects are orthogonal to our work and can easily co-exist with LoGA.

7. CONCLUSION

In this paper, we have presented LoGA, a novel accounting mechanism for GPUs. LoGA monitors the GPU's state to detect GPU-internal context switches, and infers the amount of GPU computation time consumed by each process from the time between these context switches. This approach allows LoGA to account GPU computation time without interrupting GPU access for the applications, resulting in a higher GPU utilization if individual applications do not saturate the GPU by themselves. In addition, LoGA is able to observe the interactions between concurrent applications, leading to more accurate measurements of resource consumption for some applications. Experiments with our prototype implementation have shown that LoGA induces less runtime overhead and at the same time allows the scheduler to achieve a degree of fairness comparable to that of previous work.

In the future, we plan to improve LoGA's support for virtualization. Currently, LoGA cannot distinguish between virtual machines and regular applications.

Our prototype implementation of LoGA is open source and can be downloaded from https://github.com/jkehne/loga.

8. REFERENCES

[1] Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S.-T. A., Sites, R. L., Vandevoorde, M. T., Waldspurger, C. A., and Weihl, W. E. Continuous Profiling: Where Have All the Cycles Gone? *ACM Transactions on Computer Systems 15*, 4 (Nov. 1997), 357–390.

[2] Bautin, M., Dwarakinath, A., and Chiueh, T.-c. Graphic engine resource management. In *Proceedings of the 15th Annual Multimedia Computing and Networking Conference* (San Jose, CA, USA, Jan. 2008), vol. 6818 of *MMCN '08*, SPIE, p. 12.

[3] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.-H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization* (Austin, TX, USA, Oct. 2009), IISWC '09, IEEE, pp. 44–54.

[4] Gottschlag, M., Hillenbrand, M., Kehne, J., Stoess, J., and Bellosa, F. LoGV: Low-Overhead GPGPU Virtualization. In *Proceedings of the 4th International Workshop on Frontiers of Heterogeneous Computing* (Zhanjiajie, China, Nov. 2013), FHC '13, IEEE, pp. 1721–1726.

[5] Gupta, V., Schwan, K., Tolia, N., Talwar, V., and Ranganathan, P. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *Proceedings of the 2011 USENIX Annual Technical Conference* (Portland, OR, USA, June 2011), USENIX ATC '11, USENIX Association, pp. 31–44.

[6] Jin, S., Seol, J., Huh, J., and Maeng, S. Hardware-Assisted Secure Resource Accounting Under a Vulnerable Hypervisor. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Istanbul, Turkey, Mar. 2015), VEE '15, ACM, pp. 201–213.

[7] Kato, S., Lakshmanan, K., Rajkumar, R. R., and Ishikawa, Y. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX Annual Technical Conference* (Portland, OR, USA, June 2011), USENIX ATC '11, USENIX Association, pp. 17–30.

[8] Kato, S., McThrow, M., Maltzahn, C., and Brandt, S. A. Gdev: First-Class GPU Resource Management in the Operating System. In *Proceedings of the 2012 USENIX Annual Technical Conference* (Boston, MA, June 2012), USENIX ATC '12, USENIX Association, pp. 401–412.

[9] Kehne, J., Metter, J., and Bellosa, F. GPUswap: Enabling Oversubscription of GPU Memory Through Transparent Swapping. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Istanbul, Turkey, Mar. 2015), VEE '15, ACM, pp. 65–77.

[10] Kim, S., Huh, S., Hu, Y., Zhang, X., Wated, A., Witchel, E., and Silberstein, M. GPUnet: Networking abstractions for GPU programs. In *Proceedings of the 11th International Conference on Operating Systems Design and Implementation* (Broomfield, CO, Oct. 2014), OSDI '14, USENIX Association, pp. 6–8.

[11] Menychtas, K., Shen, K., and Scott, M. L. Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, UT, USA, June 2014), ASPLOS '14, ACM, pp. 301–316.

[12] Merkel, A., and Bellosa, F. Balancing Power Consumption in Multiprocessor Systems. In *Proceedings of the 1st ACM SIGOPS EuroSys Conference* (Leuven, Belgium, Apr. 2006), EuroSys '06, ACM, pp. 403–414.

[13] Merkel, A., Stoess, J., and Bellosa, F. Resource-conscious Scheduling for Energy Efficiency on Multicore Processors. In *Proceedings of the 5th ACM SIGOPS EuroSys Conference* (Paris, France, Apr. 2010), EuroSys '10, ACM, pp. 153–166.

[14] Nagle, J. On Packet Switches with Infinite Storage. *IEEE Transactions on Communications 35*, 4 (Apr. 1987), 435–438.

[15] Rossbach, C. J., Currey, J., Silberstein, M., Ray, B., and Witchel, E. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal, Sept. 2011), SOSP '11, ACM, pp. 233–248.

[16] Silberstein, M., Ford, B., Keidar, I., and Witchel, E. GPUfs: Integrating a File System with GPUs. *ACM Transactions on Computer Systems 32*, 1 (Feb. 2014), 1:1–1:31.

[17] Suzuki, Y., Kato, S., Yamada, H., and Kono, K. GPUvm: Why not virtualizing GPUs at the hypervisor? In *Proceedings of the 2014 USENIX Annual Technical Conference* (Philadelphia, PA, June 2014), USENIX ATC '14, USENIX Association, pp. 109–120.

[18] Tam, D., Azimi, R., and Stumm, M. Thread Clustering: Sharing-aware Scheduling on SMP-CMP-SMT Multiprocessors. In *Proceedings of the 2nd ACM SIGOPS EuroSys Conference* (Lisbon, Portugal, Mar. 2007), EuroSys '07, ACM, pp. 47–58.

[19] Tian, K., Dong, Y., and Cowperthwaite, D. A full GPU virtualization solution with mediated pass-through. In *Proceedings of the 2014 USENIX Annual Technical Conference* (Philadelphia, PA, June 2014), USENIX ATC '14, USENIX Association, pp. 121–132.

[20] Yu, M., Zhang, C., Qi, Z., Yao, J., Wang, Y., and Guan, H. VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, June 2013), HPDC '13, ACM, pp. 203–214.