Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack*

Konstantinos Menychtas Kai Shen Michael L. Scott
Department of Computer Science, University of Rochester

Abstract

General-purpose GPUs now account for substantial computing power on many platforms, but the management of GPU resources—cycles, memory, bandwidth—is frequently hidden in black-box libraries, drivers, and devices, outside the control of mainstream OS kernels. We believe that this situation is untenable, and that vendors will eventually expose sufficient information about cross-black-box interactions to enable whole-system resource management. In the meantime, we want to enable research into what that management should look like.

We systematize, in this paper, a methodology to uncover the interactions within black-box GPU stacks. The product of this methodology is a state machine that captures interactions as transitions among semantically meaningful states. The uncovered semantics can be of significant help in understanding and tuning application performance. More importantly, they allow the OS kernel to intercept—and act upon—the initiation and completion of arbitrary GPU requests, affording it full control over scheduling and other resource management. While insufficiently robust for production use, our tools open whole new fields of exploration to researchers outside the GPU vendor labs.

1 Introduction

With hardware advances and the spread of programming systems like CUDA and OpenCL, GPUs have become a precious system resource, with a major impact on the power and performance of modern systems. In today's typical GPU architecture (Figure 1), the GPU device, driver, and user-level library are all vendor-provided black boxes. All that is open and documented is the high-level programming model, the library interface to programs, and some architectural characteristics useful for high-level programming and performance tuning.

For the sake of minimal overhead on very low latency GPU requests, the user-level library frequently communicates directly with the device (in both directions) through memory-mapped buffers and registers, bypassing the OS kernel entirely. A buggy or malicious application can easily obtain an unfair share of GPU resources (cycles, memory, and bandwidth). With no control over such basic functions as GPU scheduling, the kernel has no way to coordinate GPU activity with other aspects of resource management in pursuit of system-wide objectives. Application programmers, likewise, are seldom able to reason about—much less correct—performance anomalies due to contention or other interactions among the GPU requests of concurrently running applications.[1]

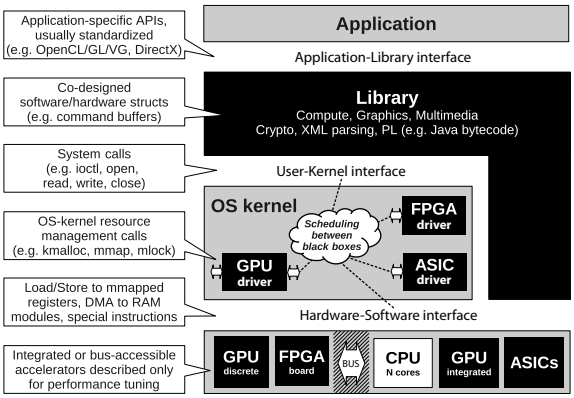

Figure 1: The GPU software / hardware architecture, with notes on interfaces and components. Gray areas indicate open system / application components while black areas indicate black-box components without published specifications or behaviors.

When GPUs were of limited programmability, and every request completed in a small, bounded amount of time, kernel control and performance transparency were much less important. As GPUs and other accelerators become more and more central to general-purpose computing, affecting thus whole-system resource management objectives, protected OS-level resource management will inevitably become a pressing need. To satisfy this need, the kernel must be able to identify and delay GPU request submissions, and tell when requests complete. A clean interface to expose this information need not compromise either proprietary technology or backward compatibility, and will hopefully be provided by vendors in the near future.

*This work was supported in part by the National Science Foundation under grants CCF-0937571, CCR-0963759, CCF-1116055, CNS-1116109, CNS-1217372, and CNS-1239423, as well as a Google Research Award.

[1] We use the term *request* to refer to a set of operations that run without interruption on the GPU — typically a GPU-accelerated compute or shader function, or a DMA request.

In anticipation of a future in which production systems manage GPUs and other accelerators as first-class computational resources, we wish to enable research with existing black-box stacks. Toward that end, we present a systematic methodology to uncover the same (limited) information about black-box interactions that vendors will eventually need to reveal. This includes the `ioctl` and `mmap` calls used to allocate and map buffers, the layout of command queues in memory shared between the library and the device, and, within that memory, the locations of device registers used to trigger GPU activity and of flags that announce request completion. While our inference is unlikely to be robust enough for production use, it provides a powerful tool for OS researchers exploring the field of GPU management.

Our work has clear parallels in white and gray-box projects, where vendors have supported certain free and open-source software (FOSS) (e.g., Intel contributes to the FOSS graphics stack used in GNU/Linux), or the FOSS community has exhaustively uncovered and rebuilt the entire stack. Projects such as Nouveau for Nvidia devices have proven successful and stable enough to become part of the mainline Linux kernel, and their developed expertise has proven invaluable to our initial efforts at reverse engineering. Our goal, however, is different: rather than develop a completely open stack for production use—which could require running a generation or two behind—we aim to model the black-box stack as a state machine that captures only as much as we need to know to manage interactions with the rest of the system. This machine can then be used with either FOSS or proprietary libraries and drivers. Compatibility with the latter allows us, in a research setting, to track the cutting edge of advancing technologies.

Previous work on GPU scheduling, including GERM [5], TimeGraph [11], and Gdev [12], has worked primarily with FOSS libraries and drivers. Our goal is to enable comparable OS-kernel management of black-box GPU stacks. PTask [13] proposes a dataflow programming model that re-envisions the entire stack, eliminating direct communication between the library and device, and building new functionality above the binary driver. Pegasus [9] and Elliott and Anderson [6] introduce new (non-standard) system interfaces that depend on application adoption. By contrast, we keep the standard system architecture, but uncover the information necessary for OS control.

2 Learning Black-Box Interactions

Our inference has three steps: (a) collect detailed traces of events as they occur across all interfaces; (b) automatically infer a state machine that describes these traces, and that focuses our attention on key structure (loops in particular); (c) manually post-process the machine to merge states where appropriate, and to identify transitions of semantic importance. We describe the first two steps below; the third is discussed in Section 3.

Trace Collection We collect traces at all relevant black-box interfaces (Figure 1). The application / library interface is defined by library calls with standardized APIs (e.g., OpenCL). The library / driver interface comprises a set of system calls, including `open`, `read`, `write`, `ioctl`, and `mmap`. The driver / kernel interface is also visible from the hosting operating system, with calls to allocate, track, release, memory-map and lock in kernel (pin) virtual memory areas. For the driver / hardware interface, we must intercept reads and writes of memory-mapped bus addresses, as well as GPU-raised interrupts.

We collect user- and kernel-level events, together with their arguments, and merge them, synchronously and in (near) order, into a unified kernel trace. Using DLL redirection, we insert a system call to enable kernel logging of each library API call. To capture memory accesses, we invalidate the pages of all virtual memory areas mapped during a tracked application's lifetime so that any access to an address in their range will trigger a page fault. Custom page fault handlers then log and reissue accesses. We employ the Linux Trace Toolkit (LTTng) [4] to record, buffer, and output the collected event traces.

For the purposes of OS-level resource management, it is sufficient to capture black-box interactions that stem from a GPU library call. Events inside the black boxes (e.g., loads and stores of GPU memory by GPU cores, or driver-initiated DMA during device initialization) can safely be ignored.

Automatic State Machine Inference If we consider the events that constitute each trace as letters from a fixed vocabulary, then each trace can be considered as a word of an unknown language. Thus, uncovering the GPU state machine is equivalent to inferring the language that produced these words—samples of event sequences from realistic GPU executions. We assume that the language is regular, and that the desired machine is a finite automaton (DFA). This assumption works well in our context for at least three reasons:

1. Automatic DFA inference requires no prior knowledge of the semantics of the traced events. For instance, it distinguishes `ioctl` calls with different identifiers as unique symbols but it does not need to know the semantic meaning of each identifier.

2. The GPU black-box interaction patterns that we are trying to uncover are part of an artificial "machine," created by GPU vendors using standard programming tools. We expect the emerging interactions to be well described by a finite-state machine.

3. The state machine provides a precise, succinct abstraction of the black-box interactions that can be used to

Event type	Meaning
`ioctl:0x??`	ioctl request : unique hex id
`map:[pin|reg|fb|sys]`	mmap : address space
`R:[pin|reg|fb|sys]`	read : address space
`W:[pin|reg|fb|sys]`	write : address space
`pin`	locked (pinned) pages
`reg`	GPU register area
`fb`	GPU frame buffer
`sys`	kernel (system) pages

Table 1: Event types and (for `map`, `R`, and `W`) associated address spaces constitute the alphabet of the regular language / GPU state machine we are trying to infer.

drive OS-level resource management. It provides a natural framework in which vendors might describe inter-black-box interactions without necessarily disclosing internal black-box semantics.

In the GPU state machine we aim to uncover, each transition (edge between two adjacent states) is labeled with a single event (or event type) drawn from an event set (or alphabet of the corresponding language). In practice, we have discovered that bigger alphabets for state transition events lead to larger and harder to comprehend state machines. We therefore pre-filter our traces to remove any detail that we expect, a priori, is unlikely to be needed. Specifically, we (a) elide the user-level API calls, many of which are handled entirely within the library; (b) replace memory addresses with the areas to which they are known to belong (e.g., registers, GPU frame buffer, system memory); and (c) elide `ioctl` parameters other than a unique hex id. This leaves us with the four basic event types shown in Table 1.

Given a set of pre-filtered traces, each of which represents the execution of the target GPU system on a given program and input, a trivial ("canonical") machine would have a single start state and a separate long chain of states for each trace, with a transition for every event. This machine, of course, is no easier to understand than the traces themselves. We wish to merge semantically equivalent states so that the resulting machine is amenable to human understanding and abstraction of interaction patterns. Note that our goal is not to identify the smallest machine that accepts the input event samples—the single-state machine that accepts everything fits this goal but it does not illustrate anything useful. So we must also be careful to avoid over-merging.

State machine reduction is a classic problem that dates from the 1950s [8], and has been studied in various forms over the years [2, 3, 7]. The problem is also related to state reduction for Hidden Markov Models [14]. Several past approaches provide heuristics that can be used to merge the states of the canonical machine. State merging in Hidden Markov Models, however, is more applicable to the modeling of (imprecise, probabilistic) natural phenomena than to capturing the DFA of a (precise) artificial system. Some reduction techniques target restricted domains (like the reversible languages of Angluin [2]); the applicability of these is hard to assess.

After reviewing the alternatives, we found Biermann and Feldman's k-tail state merging method [3] to be intuitive and easy to realize. Under this method, we merge states from which the sets of acceptable length-k-or-shorter event strings are the same. That is, when looking "downstream" from two particular states, if the sets of possible event strings (up to length k) in their upcoming transitions are exactly the same, then these two states are considered semantically equivalent and are merged. We make the following adaptations in our work:

• In theory, Biermann and Feldman's original approach can capture the precise machine if one exists. However, this guarantee [3, Theorem 5] requires an infeasible amount of trace data—exponential in the minimum string length that can distinguish two arbitrary states. We apply k-tail state merging on the canonical machine produced from a limited number of event samples, which can be easily collected. Our goal is not to fully automate precise inference, but rather to guide human understanding—to identify the transitions of importance for scheduling and other resource management.

• Our limited-sample k-tail state merging typically leaves us with a non-deterministic machine (multiple transitions on a given event from a given state). We perform additional state merging to produce a deterministic machine. Specifically, we repeatedly merge states that are reached from a common preceding state with the same transition event.

• As shown in Figure 2, larger ks yield larger machines. Intuitively, this happens because of more cautious state merging: the farther we look ahead, the harder it is to find states with exactly the same sets of upcoming event strings. If k is too large, the resulting machine will have too many states to guide human discovery of interaction patterns. If k is too small, we will merge too many states, leading to a loss of useful semantic information. To balance these factors, we choose a k that is in both a size plateau (the number of nodes in the graph does not change with small changes in k) and a node-complexity plateau (the number of nodes with in-degree or out-degree larger than 1 does not change). We avoid extreme values of k, favoring machines that are neither trivial nor unnecessarily complex. We also exploit the assumption that repetitive API-level commands should manifest as small cycles; we pick the smallest nontrivial machine in which such cycles appear.

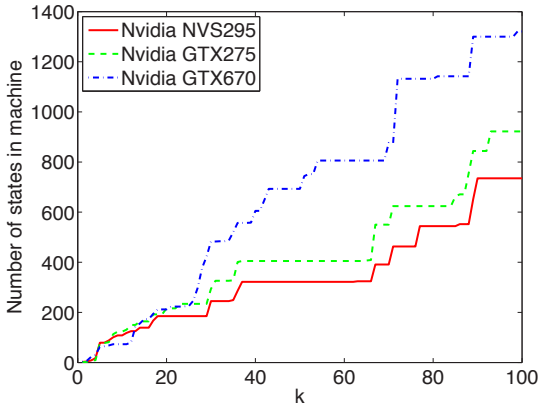

Figure 2: Sizes of state machines inferred by k-tail state merging for different values of k on three GPU devices.

3 State Machine Case Study

We have applied the techniques described in Section 2 on a system with an Intel Xeon E5520 "Nehalem" CPU and three Nvidia GPUs: (a) the NVS295, with a G98 chip core (CUDA 1.1 capability), (b) the GTX275, with a G200b chip core (CUDA 1.3 capability, "Telsa" micro-architecture), and (c) the GTX670, with a GK104 chip core (CUDA 3.0 capability, "Kepler" micro-architecture). We used a stock 3.4.7 Linux kernel and the Nvidia 310.14 driver and related libraries (CUDA 5.0, OpenCL 1.1 and OpenGL 4.2).

As input to our k-tail state merging, we used a collection of 5 traces of 4 to 9 thousand events, captured from running parameterized microbenchmarks. The microbenchmarks allow us to vary the numbers of contexts (GPU-accessible address spaces, analogous to CPU processes) and command queues (in-memory buffers through which requests are passed to the GPU); the numbers and types of requests; and the complexity of those requests (i.e., how long they occupy the GPU). The inferred state machines varied not only with k (Figure 2), but also with the model of GPU, even with the same software stack. Guided by the process described in Section 2, we ended up choosing k=35 for the NVS295, k=37 for the GTX275 and k=20 for the GTX670. As the earliest technology, the NVS295 is easiest to describe; we present it below.

Understanding Inferred DFAs Figure 3 presents the automatically inferred DFA for the NVS295 GPU for k=35; for k=34 short loops (e.g. nodes 159, 177, 178) disappear in favor of longer straight paths, while for smaller/larger ks, the graph becomes either too trivial or too complex to carry useful information. The semantics of each state are not yet clear; standard reverse-engineering techniques can help us attach meaning to both the states and the transitions by utilizing previously elided trace details. By guiding our attention to a hand-

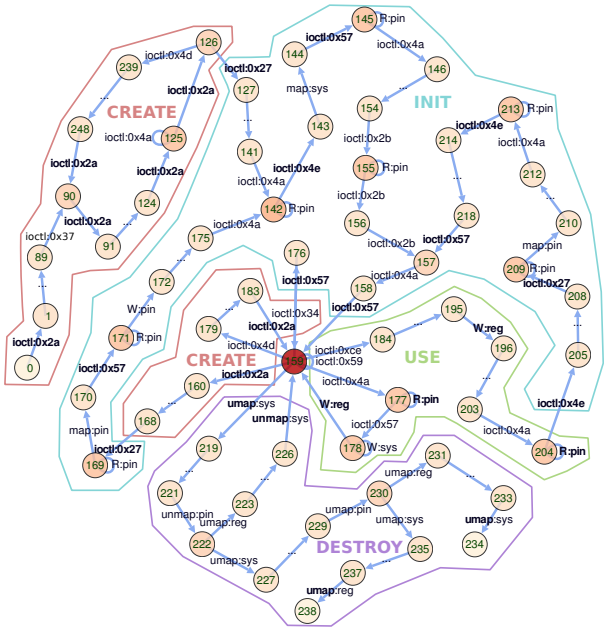

Figure 3: Inferred state machine of Nvidia NVS295 for k=35. Node "temperature" indicates frequency of occurrence. For clarity, some straight-line paths have been collapsed to a single edge (event id "...").

ful of states, and thus events in the trace, the inferred state-machine simplifies significantly the manual effort involved in this step. For example, we discovered that interrupts appear, in the general case, to occur at unpredictable times with respect to other events, and can be discarded.[2]

Identifying the start (0) and end nodes (234, 238) of the DFA in Figure 3 is trivial. The first `ioctl` event after node 0, with id `0x2a`, should be associated with the context setup process (CREATE), but is not unique to this phase. We employ previously suppressed `ioctl` arguments to uncover unique identifiers that act as invariants appearing only at or after setup. Unmapping events appearing in the graph's exit path (DESTROY) can be associated with previously established mapping calls through their arguments (`map(0xDEAD)` $\neq$ `unmap(0xBEEF)`). Realizing that important mappings have been setup or destroyed, allows us to expect (or stop waiting for) new GPU access requests.

The "epicenter" of the graph of Figure 3 is clearly node 159, so we focus on cycles around it. Since our traces can create and setup multiple contexts, command queues, etc, `ioctl` events with id `0x2a` and un-

[2]Options such as the `cudaDeviceBlockingSync` flag of the CUDA API can direct the GPU to raise an interrupt at request completion. This option may allow completion to be detected slightly earlier than it is with polling, but with either mechanism the state machine realizes completion as a (post-polling or post-interrupt) read, by the user library, of a reference counter updated by the GPU.

map events appear naturally around node 159 in cycles. Longer paths, composed primarily of `ioctl` and mapping calls (e.g., including nodes 158, 169 and 176) initialize buffers, establishing appropriate memory mappings for structures like the command queue. We identify and understand their form by storing and tracking carefully encoded bit patterns as they appear in buffers trough the trace. We also compare elided `ioctl` arguments with addresses appearing as map / unmap arguments so as to correlate events; we can thus know and expect particular buffers to be mapped and ready for use (INIT). For example, we understand that `ioctl` id `0x57` seems to associate the bus address and the system address of a memory-mapped area, which is a necessary command buffer initialization step.

Next, we focus on read / write patterns in the graph (e.g. cycle including nodes 177, 178). DMA and compute requests have a clear cause-effect relationship with the `W:reg` event on edge 178→159: a write to a mapped register must initiate a GPU request (USE). Similarly, the spin-like `R:pin` loop (e.g., at node 177) follows many GPU requests, and its frequency appears affected by the complexity of the requests; spinning on some pinned memory address must be a mechanism to check for completion of previously submitted requests. Last, we observe repetitive `W:sys` events (node 178) on the (request) path to `W:reg`, implying a causal relationship between the two. By manually observing the patterns exhibited by regularly changing values, we discover that GPU commands are saved in system memory buffers and indexed via `W:reg`.

The DFA inferred for the GTX275 (Tesla) GPU exhibits patterns very similar to the NVS295; the previous description applies almost verbatim. However, the newer GTX670 (Kepler) has brought changes to the software / hardware interface: it is `W:fb` events that capture the making of new DMA or compute / rendering requests. This means that the memory areas that seem to be causally related to GPU request submission are now accessible through the same region as the the frame buffer. Subtle differences in the use of `W:reg` can be noticed in the indexing pattern demonstrated by the `W:fb` arguments. In all other aspects, the Kepler GPU state machine remains the same as in previous generations, at least at the level of observable cross-black-box interactions.

The GPU Driver State Machine Figure 4 presents a distilled and simplified state machine (left) that captures the behavior common to our three example GPUs, and the OpenCL API calls that push the DFA into various states (right). For clarity of presentation, we have omitted certain global state and transition identifiers. Correspondences for other libraries (OpenGL, CUDA) are

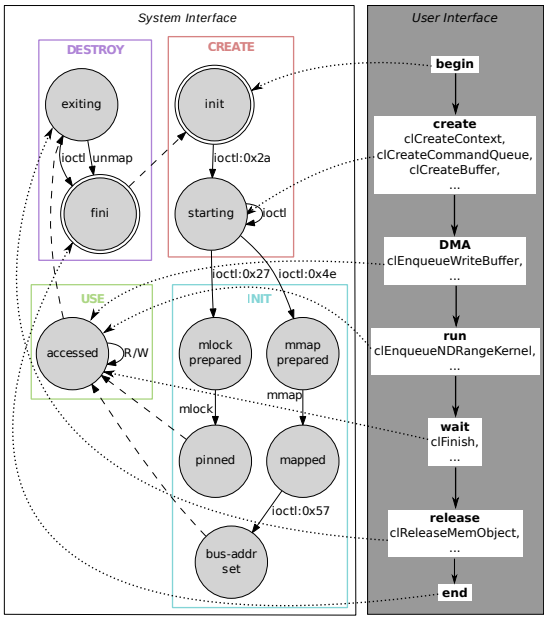

Figure 4: Semantically rich user-level API events can be mapped to state transitions at the system level.

similar. We have confirmed the validity of this state machine using realistic OpenCL / GL and CUDA applications. There exist only quantitative differences in the traces collected from 3D and compute libraries, such as the number and size of buffers mapped or variability in some elided `ioctl` parameters. Such differences do not alter the higher-level GPU driver model produced.

A GPU accelerated application typically begins with a sequence of `ioctl`, memory mapping and locking calls to build a new context (CREATE) and associate with it a set of memory areas (INIT). A small set of those, typically a command buffer and a circular queue of pointers to commands (the ring buffer), comprise the GPU command queue—a standard producer consumer communication mechanism [11]. Once initialization is complete, stores to memory-mapped GPU registers (USE) are used to point the GPU to new DMA and compute / rendering requests. Spins on associated system addresses (USE) are used to notice request completion. Cross references among these areas and elided tracing information make it possible to identify them uniquely. Unmapping operations (DESTROY) mark the ends of lifetimes of the command queue's buffers, and eventually context.

Given an abstract GPU state machine, one can build kernel-level mechanisms to intercept and intercede on edges / events (e.g. as appearing in Figure 4) that indicate preparation and utilization of the GPU ring buffer. Intercession in turn enables the construction of GPU resource managers that control the software / hardware interface, independently of the driver, yet in the protected setting of the OS kernel.

4 Conclusions and Future Work

We have outlined, in this paper, a systematic methodology to generate, analyze, and understand traces of cross-black-box interactions relevant to resource management for systems such as those of the GPU software / hardware stack. We used classic state machine inferencing to distill the numerous interactions to just a handful, and with the help of common reverse engineering techniques, revealed and assigned semantics to events and states that characterize an abstract black-box GPU state machine. In the process, we uncovered details about how OpenCL API requests (e.g., compute kernels) are transformed into commands executed on the GPU.

The suggested methodology is not fully automated, but significantly simplifies the clean-room reverse engineering task by focusing attention on important events. While we do not claim completeness in the inferred state machine description, we have defined and tested almost all combinations of run-time–affecting parameters that the library APIs allow to be set (e.g., multiple contexts, command queues, etc), and we have used a variety of graphics and compute applications as input to the inference process. No qualitative differences arise among different APIs: the inferred results remain the same. These experiments give us substantial confidence that the abstract, distilled machine captures all aspects of black-box interaction needed to drive research in OS-level GPU resource management. Validation through comparison to FOSS stacks is among our future research plans.

While our case study considered GPUs from only a single vendor (Nvidia), our methodology should apply equally well to discrete GPUs from other vendors (e.g., AMD and Intel) and to chip architectures with integrated CPU and GPU. As long as the GPU remains a co-processor, fenced behind a driver, library, and run-time stack, we expect that the command producer / consumer model of CPU / GPU interactions will require a similar, high-performance, memory-mapped ring-buffer mechanism. Available information in the form of previously released developer manuals from vendors like AMD [1] and Intel [10] supports this expectation.

The developed state machine provides application programmers with insight into how their requests are handled by the underlying system, giving hints about possible performance anomalies (e.g., request interleaving) that were previously hard to detect. More important, the machine identifies and defines an interface that can allow more direct involvement of the operating system in the management of GPUs. To effect this involvement, one would have to intercept and intercede on request-making and request-completion events, allowing an appropriate kernel module to make a scheduling decision that reflects its own priorities and policy. We consider the opportunity to build such OS-kernel level schedulers today, for cutting edge GPU software / hardware stacks, to be an exciting opportunity for the research community.

Acknowledgment

We are grateful to Daniel Gildea for helpful conversations on language inference. We also thank the anonymous reviewers and our shepherd Rama Ramasubramanian for comments that helped improve this paper.

References

[1] AMD. Radeon R5xx Acceleration: version 1.2, 2008.

[2] D. Angluin. Inference of reversible languages. *Journal of the ACM*, 29(3):741–765, July 1982.

[3] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. on Computers*, 21(6):592–597, June 1972.

[4] M. Desnoyers and M. Dagenais. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. In *Ottawa Linux Symposium*, pages 209–224, Ottawa, Canada, July 2006.

[5] A. Dwarakinath. A fair-share scheduler for the graphics processing unit. Master's thesis, Stony Brook University, Aug. 2008.

[6] G. A. Elliott and J. H. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48(1):34–74, Jan. 2012.

[7] K.-S. Fu and T. L. Booth. Grammatical inference: Introduction and survey. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 8(3):343–375, May 1986.

[8] S. Ginsburg. A technique for the reduction of a given machine to a minimal-state machine. *IRE Trans. on Electronic Computers*, EC-8(3):346–355, Sept. 1959.

[9] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *USENIX Annual Technical Conf.*, Portland, OR, June 2011.

[10] Intel. OpenSource HD Graphics Programmers Reference Manual: volume 1, part 2, 2012.

[11] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX Annual Technical Conf.*, Portland, OR, June 2011.

[12] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-class GPU resource management in the operating system. In *USENIX Annual Technical Conf.*, Boston, MA, June 2012.

[13] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating system abstractions to manage GPUs as compute devices. In *23th ACM Symp. on Operating Systems Principles*, pages 233–248, Cascais, Portugal, Oct. 2011.

[14] A. Stolcke and S. M. Omohundro. Hidden Markov model induction by Bayesian model merging. In *Advances in Neural Information Processing Systems 5*, pages 11–18, San Mateo, CA, 1993.