Operating Systems must support GPU abstractions

Christopher J. Rossbach[1], Jon Currey[1], and Emmett Witchel[2]

[1]Microsoft Research *{crossbac, jcurrey}@microsoft.com*
[2]The University of Texas at Austin *{witchel}@cs.utexas.edu*

Abstract

This paper argues that lack of OS support for GPU abstractions fundamentally limits the usability of GPUs in many application domains. OSes offer abstractions for most common resources such as CPUs, input devices, and file systems. In contrast, OSes currently hide GPUs behind an awkward `ioctl` interface, shifting the burden for abstractions onto user libraries and run-times. Consequently, OSes cannot provide system-wide guarantees such as fairness and isolation for GPUs, and developers must sacrifice modularity and performance when composing systems that integrate GPUs along with other OS-managed resources. We propose new kernel abstractions to support GPUs and other accelerator devices as first class computing resources.

1 Introduction

Three of the top five supercomputers on the TOP500 list for November 2010 use GPUs: GPUs have surpassed CPUs as a source of high-density computing resources. The proliferation of fast GPU hardware has been accompanied by the emergence of software frameworks such as DirectX [3], CUDA [8], and OpenCL [6], enabling talented programmers to write high-performance code for GPU hardware. However, despite the success of GPUs in super-computing environments, GPU hardware and programming environments are not routinely integrated into many other types of systems because of programming difficulty, lack of modularity, and lackluster performance.

Current software and system support for GPUs allows their computational power to be used for high-performance rendering or for high-performance batch-oriented computations [4], but these tools address a limited set of application domains. The GPGPU ecosystem lacks rich OS abstractions that can enable a new class of compute-intensive interactive applications, such as gestural input, brain-computer interfaces, and interactive video recognition. In contrast to interactive games, which use GPUs as rendering engines, these applications use GPUs as compute engines in the processing of user input, which necessarily entails OS support. We believe these applications are not being built because of inadequate OS-level abstractions and interfaces.

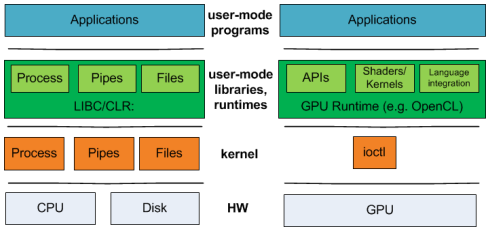

Figure 1: Technology stacks for CPU vs GPU programs. The 1-to-1 correspondence of OS-level and user-mode runtime abstractions for CPU programs is absent for GPU programs

The OS safely multiplexes a computer's hardware and therefore mediates user access to hardware, especially in interactive contexts. Even such simple interactive tasks as delivering mouse input to user application windows requires the coordination of the kernel with user-level libraries. In addition to safety, OSes also export abstractions that support programming APIs and run-times. In contrast to most common hardware, kernel-level abstractions for GPUs are limited. While OSes provide a driver interface to GPUs, that interface locks away the full potential of the graphics hardware behind an awkward `ioctl`-oriented interface designed for reading and writing blocks of data to millisecond-latency disks and networks. Commonly, communication between GPU run-times and GPU hardware is performed through shared memory mapped into both the GPU driver and the user-mode runtime, making GPU operations completely invisible to the OS. In Windows, and other closed-source OSes, using the GPU from a kernel mode driver is not even currently supported using publicly documented APIs.

Interaction with GPUs through `ioctl` system calls and/or shared memory typically follows protocols that are vendor-specific and proprietary. Consequently, developers wishing to leverage GPUs for general purpose tasks must use vendor-supplied programming interfaces and run-times (such as CUDA) that are specialized in

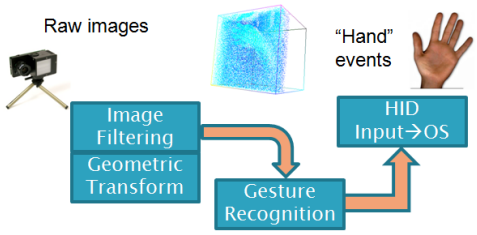

Figure 2: A gesture recognition system based on cameras

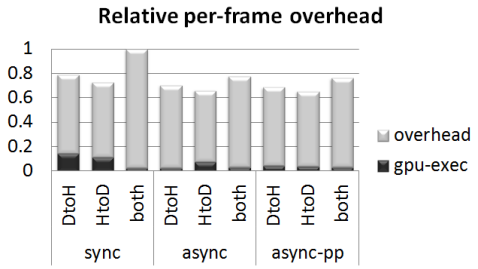

Figure 3: Relative GPU execution time and overhead (lower is better) for CUDA-based implementation of the **xform** program in our prototype system. **sync** uses synchronous communication of buffers between the CPU and GPU, **async** uses asynchronous communication, and **async-pp** uses both asynchrony and ping-pong buffers to further hide latency. Bars are divided into time spent executing on the GPU and system overhead. **DtoH** represents an implementation that communicates between the device and the host on every frame, **HtoD** the reverse, and **both** represent bi-directional communication for every frame. Reported execution time is relative to the synchronous, bi-directional case (sync-both).

ways that do not meet the programmer's needs and introduce layers of run-time support that can frustrate the programmer's performance goals.

Because the OS manages GPUs as peripherals rather than as shared compute resources, the OS cannot provide guarantees of fairness and performance isolation, making GPUs a non-starter for applications that rely on such guarantees.

This paper advocates new kernel abstractions for managing interactive, high-compute devices. GPUs represent a new kind of peripheral device, whose computation and data bandwidth exceed that of the CPU: the OS should not manage GPUs as simple I/O devices. The kernel must expose enough hardware detail of GPUs to allow programmers to take advantage of their enormous processing capabilities. But the kernel must hide programmer inconveniences like memory that is incoherent between the CPU and GPU, and must do so in a way that preserves performance. GPUs must be promoted to first-class computing resources, with traditional OS guarantees such as fairness and isolation. Finally, the OS must provide abstractions that promote modularity and simplify the development of interactive and throughput-oriented programs.

2 Motivation

To motivate our proposed reorganization of kernel abstractions we explore a case study: interactive gesture-recognition. A gestural interface turns a user's hand motions into OS input events such as mouse movements or clicks. Forcing the user to wear special gloves makes gesture recognition easier for the machine, but it is unnatural. The gestural interface we consider does not require the user to wear any special clothing. Such a system must be tolerant to visual noise on the hands, like poor lighting and rings, and must use cheap, commodity cameras to do the gesture sensing. A gestural interface workload is computationally demanding, has real-time latency constraints, and is rich with data-independent algorithms, making it a natural fit for GPU-acceleration.

Figure 2 shows a basic decomposition of the gesture recognition problem. The system consists of some number of cameras and software to analyze images captured from the cameras. Because such a system functions as a user input device, gesture events recognized by the system must be multiplexed across applications by the OS; to be usable, the system must deliver those events with high frequency and very low latency. The design decomposes the system into four components, implemented as separate programs:

- **catusb:** Captures image data from cameras connected on a USB bus.
- **xform:** Uses image-processing algorithms such as noise filtering and geometric transformations to transform images in the camera perspective to a point cloud in the coordinate system of the screen or user. Inherently data-parallel.
- **detect:** Detects gestures in a point cloud and features significant data parallelism.
- **hidinput:** Accepts gestures found by the detect program and sends them to the OS as human interface device (HID) input.

Given these four programs, a gestural interface system can be composed using POSIX pipes as follows:
catusb | xform | detect | hidinput &

This design is desirable because it is modular, (making its components easily reusable) and because it relies on familiar OS-level abstractions to communicate between components in the pipeline. Inherent data-parallelism in the **xform** and **detect** programs suggest they may be a good fit for GPU-acceleration. We have prototyped this system, and our measurements show they are not only a good fit for GPU-acceleration, they actually *require* it. The camera data rates combine with compute-hungry algorithms to saturate modern multi-core CPUs. The noise-filtering in the **xform** step alone, which relies on bilateral filtering [10] consumes 100% of a 4-core 2.66GHz Intel Core Quad, but still fails to deliver real-time frame rates. The system can only deliver real-time performance at reasonable CPU utilization rates if they system can offload data-parallel computation to a GPU.

2.1 Why OS abstractions are necessary

No direct OS support for GPU abstractions exists, so leveraging a GPU for this workload necessarily entails

a user-level GPU programming framework and run-time such as CUDA or OpenCL. Implementing **xform** and **detect** in these frameworks yields dramatic speedups for the components operating in isolation, but the composed system (`catusb | xform | detect | hidinput`) is crippled by excessive data movement across both the user-kernel boundary and through the hardware across the PCI-e bus.

For example, reading data from a camera requires copying image buffers out of kernel space to user space. Writing to the pipe connecting **catusb** to **xform** causes the same buffer to be written back into kernel space. To run **xform** on the GPU, the system must read buffers out of kernel space into user space, where a user-mode runtime such as CUDA must subsequently write the buffer back into kernel space and transfer it to the GPU and back. This pattern repeats as data moves from the **xform** to the **detect** program and so on. This simple example incurs 12 user/kernel boundary crossings. Excessive data copying also occurs across hardware components. Image buffers must migrate back and forth between main memory and GPU memory repeatedly, increasing latency, and wasting bandwidth and power.

The problem of data migration between GPU and CPU memory spaces is well-recognized by the developers of CUDA-like frameworks. CUDA, for example, supports mechanisms such as asynchronous buffer copy, CUDA streams (a generalization of the latter), pinning and memory mapping of memory buffers to help tolerate latency by overlapping computation and communication. However, to use such features, a programmer must understand OS-level issues like memory mapping. Using streams effectively requires a static knowledge of which transfers can be overlapped with which computations; such knowledge may not always be available statically.

New architectures may alter the relative difficulty of managing data across GPU and CPU memory domains, but software will retain an important role, and optimizing data movement will remain important for the foreseeable future. AMD's Fusion integrates the CPU and GPU onto a single die, however, it leaves CPU and GPU memory partitioned. Intel's Sandy Bridge, another CPU/GPU combination, indicates that the coming years will see various forms of integrated CPU/GPU hardware coming to market. New hybrid systems, e.g., NVIDIA Optimus, with both power-efficient on-die and high-performance discrete graphics cards, make data management explicit even with combined CPU/GPU chips. But even a completely integrated virtual memory system requires system support for minimizing data copies.

Overheads introduced by run-time systems can severely limit the effectiveness of latency-hiding mechanisms. Figure 3 shows relative GPU execution time and system overhead per image frame for a CUDA-based implementation of the **xform** program in our prototype. The figure compares implementations that use synchronous and asynchronous communication as well

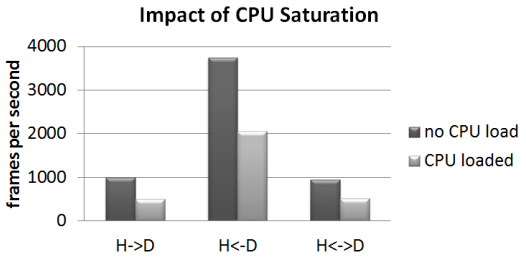

Figure 4: The effect CPU-bound work can have on GPU-bound tasks.Current OS abstractions limit the OSes ability to provide performance isolation when there is concurrent GPU and CPU work in the system. H→D is a CUDA workload that has communication from the host to the GPU device, while H←D has communication from the GPU to the host, and H↔D has bidirectional communication.

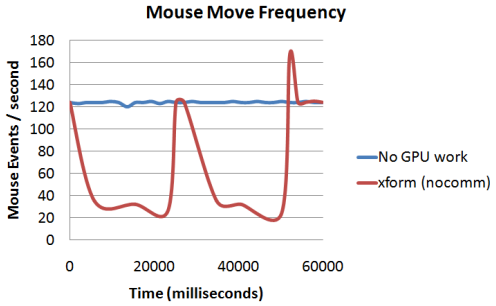

Figure 5: The effect of GPU-bound work on CPU-bound tasks. The graphs shows the frequency (in Hz) with which the OS is able to deliver mouse movement events over a period of 60 seconds during which a program makes heavy use of the GPU. Average CPU utilization over the period is under 25%.

ping pong buffers, which is another technique that overlaps communication with computation. The data illustrate that the system spends far more time marshaling data structures and migrating data than it does actually computing on the GPU. While latency-hiding does improve performance, the improvements are modest at best.

Finally, while user-level frameworks do provide mechanisms to minimize redundant hardware-level communication within a single process' address space, addressing such redundancy for cross-process or cross-device communication requires OS-level support and a programmer-visible interface. For example, currently, USB data captured from cameras must be copied into system RAM before it can be copied to the GPU: with OS support, it could be copied directly into GPU memory[1]

2.2 Why GPU workloads need OS scheduling:

Modern OSes cannot currently guarantee fairness and performance isolation for GPUs, largely because GPUs are not managed as shared computation resources (like CPUs), but as I/O devices whose interfaces are limited to a small, known set of operations (e.g. `init_module, read, write, ioctl`). This design becomes a severe limitation when the OS needs to use the GPU for its

[1]Indeed, NVIDIA GPU Direct [1] implements just such a feature, but requires specialized support in the driver of any I/O device involved.

own computation (e.g., as Windows 7 does with the Aero user-interface). Under the current regime, time-slicing and timeouts ensure that screen refresh rates are maintained, but the OS is largely at the mercy of the GPU driver when it comes to enforcing fairness and load balancing the system.

Figure 4 shows the inability of Windows 7 to load balance a system that has concurrent, but fundamentally unrelated work on the GPU and CPUs. The data in the figure were collected on a machine with 64-bit Windows 7, Intel Core 2 Quad 2.66GHz, 8GB RAM, and an nVidia GeForce GT230 GPU. The figure shows the impact of a CPU-bound process (using all 4 cores to increment counter variables) on the frame rate of a shader program (the **xform** program from our prototype implementation). The frame rate of the GPU program drops by 2x, despite the near complete absence of CPU work in the program: **xform** uses the CPU only to trigger the next computation on the device.

Figure 5 shows the impact of GPU-bound work on the frequency with which the system can collect and deliver mouse movements. In our experiments, significant GPU-work at high frame rates causes Windows 7 to be unresponsive for seconds at a time. To measure this phenomenon, we instrumented the OS to record the frequency of mouse events delivered through the HID class driver over a 60 second period. When no concurrent GPU work is executing, the system is able to deliver mouse events at a stable 120 Hz. However, when the GPU is heavily loaded, the rate at which the system can deliver mouse events plummets, often to below 30 Hz. The GPU-bound task is console-based (does not update the screen) and performs unrelated work in another process context. Moreover, CPU utilization is below 25%, showing that the OS has compute resources available to deliver events. The inability of the OS to manage the GPU as a first-class resource inhibits it's ability to load balance the entire system effectively.

GPUs need to be treated as a first-class computing resource and managed by the OS scheduler like a normal CPU. User programs should interact with GPUs using abstractions similar to threads and processes. Current OSes provide no abstractions that fit this model. In the following sections, we propose abstractions to address precisely this problem.

3 New OS abstractions

We propose the following new OS abstractions which can make GPUs applicable across a broader set of application domains. These abstractions allow efficient data movement and efficient and fair scheduling by allowing computation to be expressed as a directed graph.

PTask. A PTask is analogous to the traditional OS process abstraction, but a PTask runs substantially on a GPU. A PTask requires some orchestration from the OS to coordinate its execution, but does not require a user-mode host process. A PTask has a list of input and output

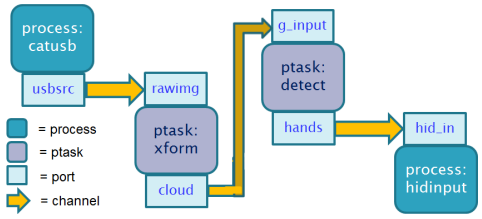

Figure 6: A dataflow graph for the gesture recognition system using the ptask, port, and channel OS abstractions.

PTask system call	Description
`sys_open_graph(name)`	Open or create a PTask graph
`sys_open_port(name, type, template)`	Open or create a port
`sys_open_ptask(name, kernel, graph, portlist)`	Open or create a new PTask bound to a graph and ports
`sys_open_channel(name, src, dest)`	Open or create channel bound to given ports
`sys_push(channel|port)`	Write to a channel or port
`sys_pull(channel|port)`	Read from a channel or port
`sys_run_graph(graph)`	Move graph to running state
`sys_terminate_graph(graph)`	Terminate graph execution
`sys_set_ptask_prio(name, prio)`	Set the priority for a node in a PTask graph

Table 1: Proposed new system calls to support PTasks.

resources (analogous to the POSIX `stdin, stdout, stderr` file descriptors) that can be bound to ports.

Port. A port is an object in the kernel namespace that can be bound to PTask input and output resources. A port is a data source or sink, that provides a way to expose data and parameters in GPU code that must be dynamically bound, and can be populated by, for example, buffers in GPU or CPU memory.

Channel. A channel is analogous to a POSIX pipe: it connects ports to other ports, or to other data sources and sinks in the system such as I/O buses, files, and so on. Channel has sub-types GraphInputChannel, GraphOutputChannel, and GraphInternalChannel.

Graph. A graph is collection of PTask nodes whose input and output ports are connected by channels. Multiple Graphs may be created and executed independently, with the PTask runtime being responsible for scheduling them fairly.

Supporting these new abstractions at the OS interface entails new system calls for creating and managing ptasks, ports, and channels, shown in Table 1. The additional system calls are analogous to the process API, inter-process communication API, and scheduler hint API in POSIX.

3.1 Scheduling the GPU for efficiency and fairness

The two chief benefits of coordinating OS scheduling with the GPU are efficiency and fairness. By efficiency we mean low latency between when a ptask is ready and when it is scheduled on the GPU, and scheduling enough ptask work on the GPU to fully utilize its computational bandwidth. By fairness we mean that the OS scheduler balances GPU utilization with user interface responsive-

ness. Also, ptasks competing for the GPU all get some reasonable share of its computational bandwidth. Not all ptasks have all of these scheduling requirements. A typical CUDA program does not care about low latency scheduling. Gestural interfaces require all of these mechanisms.

While a ptask can exist without a dedicated user-mode host process to manage it, parent-child relationships can exist between processes and ptasks. In such a case, a ptask is associated with the process that creates it and a process may have any number of concurrent ptasks. The scheduling needs of ptasks may be different from the scheduling needs of the process with which they are associated. For example, it may be desirable to gang schedule a ptask and threads: if threads are waiting for a low-latency computation to complete the CPU thread may wish to busy-wait for it to minimize overall latency for that application, or may wish to overlap some of its own computation with ptask computation. Conversely, a ptask that is executing a long latency computation on whose results some thread depends may be better served by scheduling the ptask when the process' CPU thread(s) are blocked.

Like processes and threads, ptasks have scheduling priority and policy, allowing the OS to provide the same priority guarantees for GPU computation as it provides for processes and threads. Natural scheduling semantics are clearly desirable and some user-mode GPGPU frameworks like CUDA provide APIs to allow applications to specify scheduling constraints. However, because the OS does not mediate interaction between these frameworks and GPU devices, system-wide guarantees are difficult to deliver without OS support. Scheduling ptasks at the OS-level is complicated by the fact that GPU hardware cannot currently be preempted or context-switched, ruling out traditional approaches to time-slicing hardware. As a result ptask guarantees are necessarily "best-effort". On average, ptasks will see throughput that reflects a share of GPU-compute proportional to its priority.

3.2 Gestural interface graph

The gestural interface system can be recast as a graph using these abstractions, yielding multiple advantages (see Figure 6). First, the graph eliminates unnecessary communication. A channel connects the USB source port ("usbsrc") to the image input port ("rawimg"). Data transfer across this channel goes directly from the USB device to GPU memory, rather than taking an unnecessary detour through system memory. A channel connecting the output port of xform ("cloud") to the gesture input ("g_input") port of detect can avoid copy altogether by reusing the output of one ptask as the input of the next. The design also minimizes involvement of host-based user-mode applications to coordinate common GPU activities. For example, the arrival of data at the raw image input of the xform program can trigger the computation for the new frame using interrupt handlers in the OS, rather than waiting for a host-based program to be scheduled to start the GPU-based processing of the new frame.

4 Related work

The Hydra framework [11] provides a dataflow programming model for offloading tasks to peripheral devices. Hydra components communicate through a common runtime and API that cannot be supported by current GPU hardware. Helios [7] and Barrelfish [2] address the problem of OS support in heterogeneous platforms, proposing abstractions that current GPUs cannot support because they lack architectural features to run OS code or use RPC.

Monsoon [9] targets environments in which dataflow execution is supported directly by the processor hardware, while we propose kernel-level abstractions. The Dryad [5] execution engine provides a graph-based fault-tolerant programming model for managing distributed parallel execution in data center environments. While we espouse a similar model for representing parallel execution as a graph, the target platforms are entirely different.

5 Conclusion

This paper advocates a fundamental reorganization of kernel abstractions for managing interactive, massively parallel devices. The kernel must expose only enough hardware detail as required to enable programmers to achieve good performance and low latency, while providing communication abstractions that encapsulate and specialize according to the topology of the machine. GPUs are a general-purpose, shared compute resource, and must be managed by the OS as such to provide fairness and isolation.

References

[1] NVIDIA GPUDirect. 2011.

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP '09*.

[3] D. Blythe. The Direct3D 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.

[4] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel Computing Experiences with CUDA. *Micro, IEEE*, 28(4):13–27, 2008.

[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys 2007*.

[6] Khronos Group. *The OpenCL Specification, Version 1.0*, 2009.

[7] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP 2009*.

[8] NVIDIA. *NVIDIA CUDA Programming Guide*, 2011.

[9] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. *SIGARCH Comput. Archit. News*, 1990.

[10] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *ICCV 1998*.

[11] Y. Weinsberg, D. Dolev, T. Anker, M. Ben-Yehuda, and P. Wyckoff. Tapping into the fountain of CPUs: on operating system support for programmable devices. In *ASPLOS 2008*.