PTask: Operating System Abstractions To Manage GPUs as Compute Devices

Christopher J. Rossbach
Microsoft Research
crossbac@microsoft.com

Jon Currey
Microsoft Research
jcurrey@microsoft.com

Mark Silberstein
Technion
marks@cs.technion.ac.il

Baishakhi Ray
University of Texas at Austin
bray@cs.utexas.edu

Emmett Witchel
University of Texas at Austin
witchel@cs.utexas.edu

ABSTRACT

We propose a new set of OS abstractions to support GPUs and other accelerator devices as first class computing resources. These new abstractions, collectively called the **PTask API**, support a dataflow programming model. Because a PTask graph consists of OS-managed objects, the kernel has sufficient visibility and control to provide system-wide guarantees like fairness and performance isolation, and can streamline data movement in ways that are impossible under current GPU programming models.

Our experience developing the PTask API, along with a gestural interface on Windows 7 and a FUSE-based encrypted file system on Linux show that the PTask API can provide important system-wide guarantees where there were previously none, and can enable significant performance improvements, for example gaining a 5× improvement in maximum throughput for the gestural interface.

Categories and Subject Descriptors

D.4.8 [**Operating systems**]: [Performance]; D.4.7 [**Operating systems**]: [Organization and Design]; I.3.1 [**Hardware Architecture**]: [Graphics processors]; D.1.3 [**Programming Techniques**]: [Concurrent Programming]

General Terms

OS Design, GPUs, Performance

Keywords

Dataflow, GPUs, operating systems, GPGPU, gestural interface, accelerators

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP '11, October 23-26, 2011, Cascais, Portugal.
Copyright © 2011 ACM 978-1-4503-0977-6/11/10 ... $10.00.

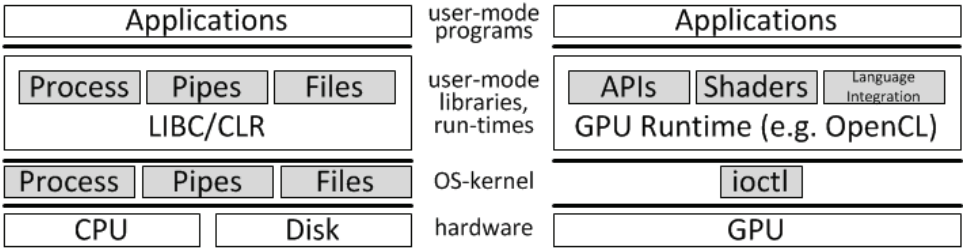

Figure 1: Technology stacks for CPU vs GPU programs. The 1-to-1 correspondence of OS-level and user-mode runtime abstractions for CPU programs is absent for GPU programs

1. INTRODUCTION

Three of the top five supercomputers on the TOP500 list for June 2011 (the most recent ranking) use graphics processing units (GPUs) [6]. 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 general purpose GPU (GPGPU) frameworks such as DirectX, CUDA [59], and OpenCL [47], enabling talented programmers to write high-performance code for GPU hardware. However, despite the success of GPUs in supercomputing environments, GPU hardware and programming environments are not routinely integrated into many other types of systems because of programming difficulty, lack of modularity, and unpredictable performance artifacts.

Current software and system support for GPUs allows their computational power to be used for high-performance rendering or for a wide array of high-performance batch-oriented computations [26], but GPU use is limited to certain application domains. The GPGPU ecosystem lacks rich operating system (OS) abstractions that would enable new classes of compute-intensive interactive applications, such as gestural input, brain-computer interfaces, and interactive video recognition, or applications in which the OS uses the GPU for its own computation such as encrypted file systems. In contrast to interactive games, which use GPUs as rendering engines, these applications use GPUs as compute engines in contexts that require OS support. We believe these applications are not being built because of inadequate OS-level abstractions and interfaces. The time has come for OSes to stop managing graphics processing devices (GPUs) as I/O devices and start managing them as a computational devices, like CPUs.

Figure 1 compares OS-level support for traditional hardware to OS-level support for GPUs. In contrast to most common system resources such as CPUs and storage devices, kernel-level abstractions for GPUs are severely 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. Moreover, lack of a general kernel-facing interface severely limits what the OS can do to provide high-level abstractions for GPUs: in Windows, and other closed-source OSes, using the GPU from a kernel mode driver is not currently supported using any publicly documented APIs. Additionally, because the OS manages GPUs as peripherals rather than as shared compute resources, the OS leaves resource management for GPUs to vendor-supplied drivers and user-mode run-times. With no role in GPU resource-management, the OS cannot provide guarantees of fairness and performance isolation. For applications that rely on such guarantees, GPUs are consequently an impractical choice.

This paper proposes a set of kernel-level 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 kernel must expose enough hardware detail of these peripherals to allow programmers to take advantage of their enormous processing capabilities. But the kernel must hide programmer inconveniences like memory that is non-coherent 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, and the OS must provide abstractions that allow programmers to write code that is both modular and performant.

Our new abstractions, collectively called the **PTask API**, provide a dataflow programming model in which the programmer writes code to manage a graph-structured computation. The vertices in the graph are called **ptasks** (short for **p**arallel **task**) which are units of work such as a shader program that runs on a GPU, or a code fragment that runs on the CPU or another accelerator device. PTask vertices in the graph have input and output **ports** exposing data sources and sinks in the code, and are connected by **channels**, which represent a data flow edge in the graph. The graph expresses both data movement and potential concurrency directly, which can greatly simplify programming. The programmer must express only *where* data must move, but not *how* or *when*, allowing the system to parallelize execution and optimize data movement without any additional code from the programmer. For example, two sibling ptasks in a graph can run concurrently in a system with multiple GPUs without additional GPU management code, and double buffering is eliminated when multiple ptasks that run on a single accelerator are dependent and sequentially ordered. Under current GPU programming models, such optimizations require direct programmer intervention, but with the PTask API, the same code adapts to run optimally on different hardware substrates.

A PTask graph consists of OS-managed objects, so the kernel has sufficient visibility and control to provide system-wide guarantees like fairness and performance isolation. The PTask runtime tracks GPU usage and provides a state machine for ptasks that allows the kernel to schedule them in a way similar to processes. Under current GPU frameworks, GPU scheduling is completely hidden from the kernel by vendor-provided driver code, and often implements simplistic policies such as round-robin. These simple policies can thwart kernel scheduling priorities, undermining fairness and inverting priorities, often in a dramatic way.

Kernel-level ptasks enable data movement optimizations that are impossible with current GPU programming frameworks. For example, consider an application that uses the GPU to accelerate real-time image processing for data coming from a peripheral like a camera. Current GPU frameworks induce excessive data copy by causing data to migrate back and forth across the user-kernel boundary, and by double-buffering in driver code. A PTask graph, conversely, provides the OS with precise information about data's origin(s) and destination(s). The OS uses this information to eliminate unnecessary data copies. In the case of real-time processing of image data from a camera, the PTask graph enables the elimination of two layers of buffering. Because data flows directly from the camera driver to the GPU driver, an intermediate buffer is unnecessary, and a copy to user space is obviated.

We have implemented the full PTask API for Windows 7 and PTask scheduling in Linux. Our experience using PTask to accelerate a gestural interface in Windows and a FUSE-based encrypted file system in Linux shows that kernel-level support for GPU abstractions provides system-wide guarantees, enables significant performance gains, and can make GPU acceleration practical in application domains where previously it was not.

This paper makes the following contributions.

- Provides quantitative evidence that modern OS abstractions are insufficient to support a class of "interactive" applications that use GPUs, showing that simple GPU programs can reduce the response times for a desktop that uses the GPU by nearly an order of magnitude.
- Provides a design for OS abstractions to support a wide range of GPU computations with traditional OS guarantees like fairness and isolation.
- Provides a prototype of the PTask API and a GPU-accelerated gestural interface, along with evidence that PTasks enable "interactive" applications that were previously impractical, while providing fairness and isolation guarantees that were previously absent from the GPGPU ecosystem. The data flow programming model supported by the PTask API delivers throughput improvements up to $4\times$ across a range of microbenchmarks and a $5\times$ improvement for our prototype gestural interface.

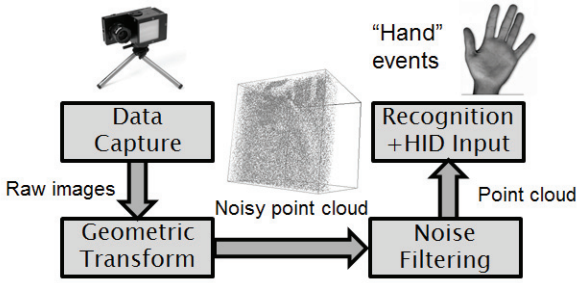

Figure 2: A gesture recognition system based on photogrammetric cameras

- Demonstrates a prototype of GPU-aware scheduling in the Linux kernel that forces GPU-using applications to respect kernel scheduling priorities.

2. MOTIVATION

This paper focuses on GPU support for interactive applications like gesture-based interfaces, neural interfaces (also called brain-computer interfaces or BCIs) [48], encrypting file systems and real-time audio/visual interfaces such as speech recognition. These tasks are computationally demanding, have real-time performance and latency constraints, and feature many data-independent phases of computation. GPUs are an ideal compute substrate for these tasks to achieve their latency deadlines, but lack of kernel support forces designers of these applications to make difficult and often untenable tradeoffs to use the GPU.

To motivate our new kernel abstractions we explore the problem of interactive gesture recognition as a case study. A gestural interface turns a user's hand motions into OS input events such as mouse movements or clicks [36]. 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-parallel algorithms, making it a natural fit for GPU-acceleration. Gesture recognition is similar to the computational task performed by Microsoft's Kinect, though that system has fewer cameras, lower data rates and grosser features. Kinect only runs a single application at a time (the current game), which can use all available GPU resources. An operating system must multiplex competing applications.

Figure 2 shows a basic decomposition of a gesture recognition system. The system consists of some number of cameras (in this example, photogrammetric sensors [28]), 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 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. Short for "`cat /dev/usb`".
- **xform:** Perform geometric transformations to transform images from multiple camera perspectives to a single point cloud in the coordinate system of the screen or user. Inherently data-parallel.
- **filter:** Performs noise filtering on image data produced by the xform step. Inherently data-parallel.
- **hidinput:** Detects gestures in a point cloud and sends them to the OS as human interface device (HID) input. Not data parallel.

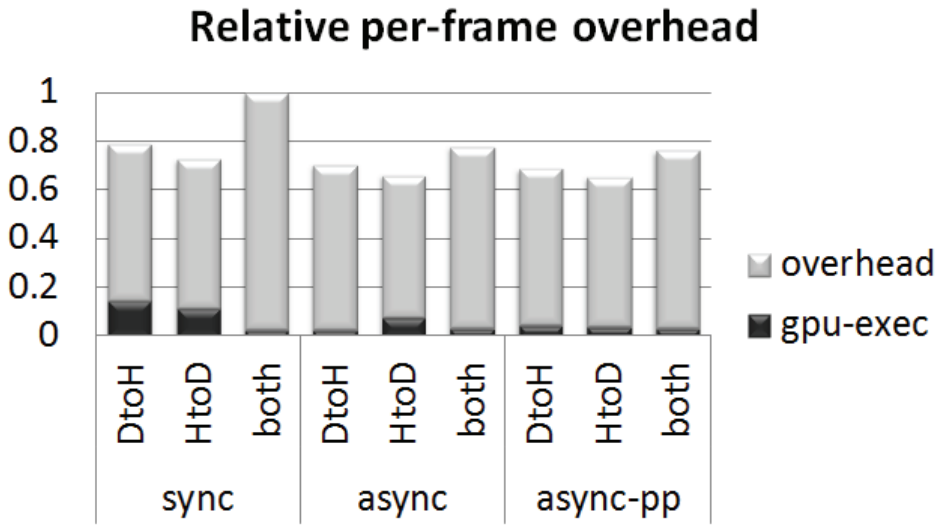

Figure 3: Relative GPU execution time and overhead (lower is better) for CUDA-based implementation of the xform program in our prototype system. sync uses synchronous communication of buffers between the CPU and GPU, async uses asynchronous communication, and async-pp uses both asynchrony and ping-pong 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).

Given these four programs, a gestural interface system can be composed using POSIX pipes as follows:

```
catusb | xform | filter | 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 **filter** programs strongly argue for GPU acceleration. We have prototyped these computations and our measurements show they are not only a good fit for GPU-acceleration, they actually *require* it. If the system uses multiple cameras with high data rates and large image sizes, these algorithms can easily saturate a modern chip multi-processor (CMP). For example, our **filter** prototype relies on bilateral filtering [67]. A well-optimized implementation using fork/join parallelism is unable to maintain real-time frame rates on a 4-core CMP despite consuming nearly 100% of the available CPU. In contrast, a GPU-based implementation easily realizes frame rates above the real-time rate, and has minimal affect on CPU utilization because nearly all of the work is done on the GPU.

2.1 The problem of data movement

No direct OS support for GPU abstractions exists, so computing on a GPU for a gestural interface necessarily entails a user-level GPU programming framework and run-time such as DirectX, CUDA, or OpenCL. Implementing **xform** and **filter** in these frameworks yields dramatic speedups for the components operating in isolation, but the system composed with pipes suffers from excessive data movement across both the user-kernel boundary and through the hardware across the PCI express (PCIe) 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 **filter** 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 while wasting bandwidth and power.

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 as ping-pong buffers, 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 techniques to hide the latency of communication improve performance, the improvements are modest at best.

User-level frameworks do provide mechanisms to minimize redundant hardware-level communication within a single process' address space. However, addressing such redundancy for cross-process or cross-device communication requires OS-level support and a programmer-visible interface. For example, 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 No easy fix for data movement

The problem of data migration between GPU and CPU memory spaces is well-recognized by the developers of GPGPU frameworks. CUDA, for example, supports mechanisms such as asynchronous buffer copy, CUDA streams (a generalization of the latter), and pinning of memory buffers to tolerate data movement latency by overlapping computation and communication. However, to use such features, a programmer must understand OS-level issues like memory mapping. For example, CUDA provides APIs to pin allocated memory buffers, allowing the programmer to avoid a layer of buffering above DMA transfer. The programmer is cautioned to use this feature sparingly as it reduces the amount of memory available to the system for paging [59].

Using streams effectively requires a static knowledge of which transfers can be overlapped with which computations; such knowledge may not always be available statically. Moreover, streams can only be effective if there is available communication to perform that is independent of the current computation. For example, copying data for stream a_1 to or from the device for execution by kernel A can be overlapped with the execution of kernel B; attempts to overlap with execution of A will cause serialization. Consequently, modules that offload logically separate computation to the GPU must be aware of each other's computation and communication patterns to maximize the effectiveness of asynchrony.

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, and enables coherent yet slow access to the shared memory by both processors. However high performance is only achievable via non-coherent accesses or by using private GPU memory, leaving data placement decisions to software. Intel's Sandy Bridge, another CPU/GPU combination, is further indication that the coming years will see various forms of integrated CPU/GPU hardware coming to market. New hybrid systems, such as NVIDIA Optimus, have a power-efficient on-die GPU and a high-performance discrete GPU. Despite the presence of a combined CPU/GPU chip, such sys-

[1] Indeed, NVIDIA GPU Direct [4] implements just such a feature, but requires specialized support in the driver of any I/O device involved.

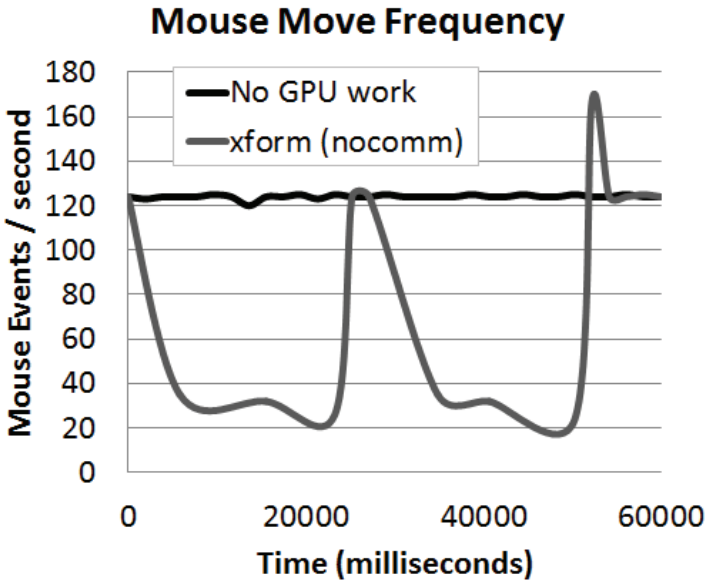

Figure 4: The effect of GPU-bound work on CPU-bound tasks The graph shows the frequency (in Hz) with which the OS is able to deliver mouse movement events over a period of 60 seconds during which a program makes heavy use of the GPU. Average CPU utilization over the period is under 25%.

tems still require explicit data management. While there is evidence that GPUs with coherent access to shared memory may eventually become common, even a completely integrated virtual memory system requires system support for minimizing data copies.

2.3 The scheduling problem

Modern OSes cannot currently guarantee fairness and performance for systems that use GPUs for computation. The OS does not treat GPUs as a shared computational resource, like a CPU, but rather as an I/O device. This design becomes a severe limitation when the OS needs to use the GPU for its own computation (e.g., as Windows 7 does with the Aero user interface). Under the current regime, watchdog timers ensure that screen refresh rates are maintained, but OS scheduling priorities are easily undermined by the GPU driver.

GPU work causes system pauses. Figure 4 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 instrument 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 mouse event rate plummets, often to below 20 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. A combination of factors are at work in this situation. GPUs are not preemptible, with the side-effect that in-progress I/O requests cannot be canceled once begun. Because Windows relies on cancelation to prioritize its own work, its priority mechanism fails. The problem is compounded because the developers of the GPU runtime use request batching to improve throughput for GPU programs. Ultimately, Windows is unable to interrupt a large number of GPU invocations submitted

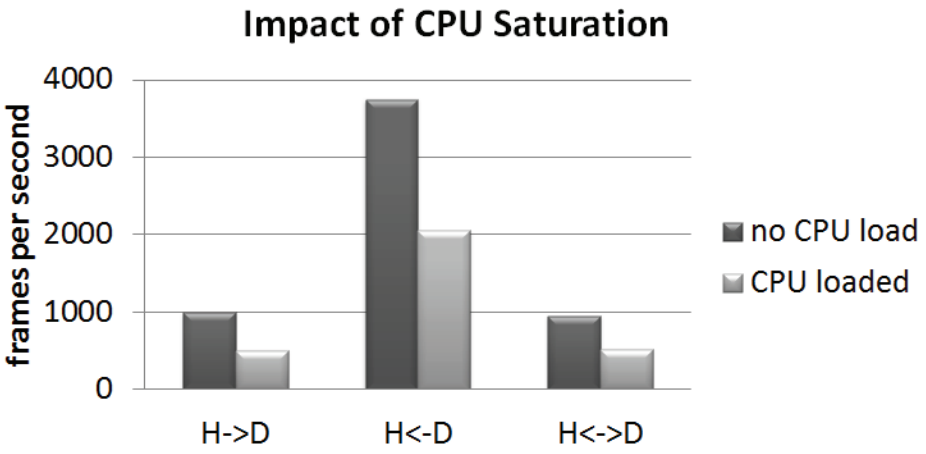

Figure 5: The effect of CPU-bound work on GPU-bound tasks. H→D is a CUDA workload that has communication from the host to the GPU device, while H←D has communication from the GPU to the host, and H↔D has bidirectional communication.

in batch, and the system appears unresponsive. The inability of the OS to manage the GPU as a first-class resource inhibits its ability to load balance the entire system effectively.

CPU work interferes with GPU throughput. Figure 5 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 absence of CPU work in the program: **xform** uses the CPU only to trigger the next computation on the GPU device.

These results suggest that GPUs need to be treated as a first-class computing resource and managed by the OS scheduler like a normal CPU. Such abstractions will allow the OS to provide system-wide properties like fairness and performance isolation. 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. DESIGN

We propose a set of new OS abstractions to support GPU programming called the PTask (**P**arallel **Task**) API. The PTask API consists of interfaces and runtime library support to simplify the offloading of compute-intensive tasks to accelerators such as GPUs. PTask supports a dataflow programming model in which individual tasks are assembled by the programmer into a directed acyclic graph: vertices, called **ptasks**, are executable code such as shader programs on the GPU, code fragments on other accelerators (e.g. a SmartNIC), or callbacks on the CPU. Edges in the graph represent data flow, connecting the inputs and outputs of each vertex. PTask is best suited for applications that have significant computational demands, feature both task- and data-level parallelism, and require both high throughput and low latency.

PTask was developed with three design goals. **(1)** Bring GPUs under the purview of a single (perhaps federated) resource manager, allowing that entity to provide meaningful guarantees for fairness and

isolation. **(2)** Provide a programming model that simplifies the development of code for accelerators by abstracting away code that manages devices, performs I/O, and deals with disjoint memory spaces. In a typical DirectX or CUDA program, only a fraction of the code implements algorithms that run on the GPU, while the bulk of the code manages the hardware and orchestrates data movement between CPU and GPU memories. In contrast, PTask encapsulates device-specific code, freeing the programmer to focus on application-level concerns such as algorithms and data flow. **(3)** Provide a programming environment that allows code to be both modular and fast. Because current GPU programming environments promote a tight coupling between device-memory management code and GPU-kernel code, writing reusable code to leverage a GPU means writing both algorithm code to run on the GPU and code to run on the host that transfers the results of a GPU-kernel computation when they are needed. This approach often translates to sub-optimal data movement, higher latency, and undesirable performance artifacts.

3.1 Integrating PTask scheduling with the OS

The two chief benefits of coordinating OS scheduling with the GPU are efficiency and fairness (design goals (1) and (3)). By efficiency we mean both low latency between when a ptask is ready and when it is scheduled on the GPU, and scheduling enough work on the GPU to fully utilize it. By fairness we mean that the OS scheduler provides OS priority-weighted access to processes contending for the GPU, and balances GPU utilization with other system tasks like user interface responsiveness.

Separate processes can communicate through, or share a graph. For example, processes A and B may produce data that is input to the graph, and another process C can consume the results. The scheduler must balance thread-specific scheduling needs with PTask-specific scheduling needs. For example, gang scheduling the producer and consumer threads for a given PTask graph will maximize system throughput.

3.2 Efficiency vs. modularity

Consider the pseudo-code in Figure 6, which reuses a matrix multiplication subroutine called `gemm` to implement $((A \times B) \times C)$. GPUs typically have private memory spaces that are not coherent with main memory and not addressable by the CPU. To offload computation to the GPU, the `gemm` implementation must copy input matrices A and B to GPU memory. It then invokes a GPU-kernel called `gemm_kernel` to perform the multiplication, and copies the result back to main memory. If the programmer reuses the code for `gemm` to compose the product $((A \times B) \times C)$ as `gemm(gemm(A,B),C)` (`modularSlowAxBxC` in Figure 6), the intermediate result $(A \times B)$ is copied back from the GPU at the end of the first invocation of `gemm` only to be copied from main memory to GPU memory again for the second invocation. The performance costs for data movement are significant. The problem can be trivially solved by writing code specialized to the problem, such as the `nonmodularFastAxBxC` in the figure. However, the code is no longer as easily reused.

Within a single address space, such code modularity issues can often be addressed with a layer of indirection and encapsulation for GPU-side resources. However, the problem of optimizing data movement inevitably becomes an OS-level issue as other devices and resources interact with the GPU or GPUs. With OS-level support, computations that involves GPUs and OS-managed resources such as cameras, network cards, and file systems can avoid problems like double-buffering.

By decoupling data flow from algorithm, PTask eliminates difficult tradeoffs between modularity and performance (design goal (3)): the run-time automatically avoids unnecessary data movement (design goal (2)). With PTask, matrix multiplication is expressed as a graph with A and B as inputs to one `gemm` node; the output of that node becomes an input to another `gemm` node that also takes C as input. The programmer expresses only the structure of the computation and the system is responsible for materializing a consistent view of the data in a memory domain *only when it is actually needed*. For example, the system knows that the intermediate result $(A \times B)$ is both produced and consumed by the GPU, so it can avoid moving that intermediate result to and from CPU memory at run-time. The code for `gemm` remains modular, and there is no performance penalty for composition.

```
matrix gemm(A, B) {
  matrix res = new matrix();
  copyToDevice(A);
  copyToDevice(B);
  invokeGPU(gemm_kernel, A, B, res);
  copyFromDevice(res);
  return res;
}
matrix modularSlowAxBxC(A, B, C) {
  matrix AxB = gemm(A, B);
  matrix AxBxC = gemm(AxB, C);
  return AxBxC;
}
matrix nonmodularFastAxBxC(A, B, C) {
  matrix intermed = new matrix();
  matrix res = new matrix();
  copyToDevice(A);
  copyToDevice(B);
  copyToDevice(C);
  invokeGPU(gemm_kernel, A, B, intermed);
  invokeGPU(gemm_kernel, intermed, C, res);
  copyFromDevice(res);
  return res;
}
```

Figure 6: Pseudo-code to offload matrix computation $(A \times B) \times C$ to a GPU. This modular approach uses the `gemm` subroutine to compute both $A \times B$ and $(A \times B) \times C$, forcing an unnecessary round-trip from GPU to main memory for the intermediate result.

Dataflow is also an attractive programming model for GPUs and heterogeneous platforms because a DAG explicitly expresses concurrency without telling the system how to leverage that concurrency. On a machine with multiple GPUs, a PTask graph can take advantage of them with *no* modifications (see Section 6.4); current GPU frameworks require significant additional work from the programmer to use multiple GPUs.

3.3 Limitations

The PTask programming model is based on dataflow, and it allows only directed acyclic graphs, so some limitations are inherent. PTask does not support dynamic graphs, and consequently, problems that are not well-expressed as static graphs, such as those featuring recursion, are not easily expressed with PTask. Data-dependent loop iteration can be handled by unrolling loops into linear sections of the graph along with conditional execution of GPU code (i.e. if the graph represents an conservative unrolling of a loop, some vertices will not always execute). PTask does not fundamentally limit the programmer's ability to express algorithms that do not map easily to a data flow model, because the programmer can always work with PTask graphs containing only a single ptask. For example, an algorithm that requires data-dependent loop bounds can be expressed as a single-ptask graph, where iteration conditions are computed on the host, much as such programs are currently expressed.

4. PTASK API

The PTask API is built on the OS-level abstractions listed below. Supporting these new abstractions at the OS interface entails new system calls shown in Table 1. The additional system calls are analogous to the process API, inter-process communication API, and scheduler hint API in POSIX.

PTask. A ptask is analogous to the traditional OS process abstraction, but a ptask runs substantially on a GPU or other accelerator. A ptask requires some orchestration from the OS to coordinate its execution, but does not always require a user-mode host process. Each ptask has a list of input and output resources (analogous to the POSIX `stdin, stdout, stderr` file descriptors) that can be bound to ports.

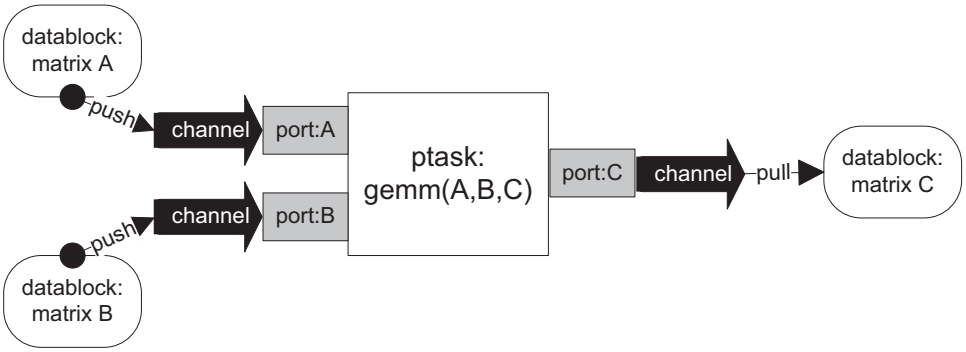

Figure 7: A dataflow graph for matrix multiplication.

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 for dynamically bound data and parameters in GPU code. Ports have sub-types **InputPort**, **OutputPort**, and **StickyPort**. The former two represent inputs and outputs respectively, while the latter represents an input which can retain its value across multiple invocations of a ptask.

Channel. A channel is analogous to a POSIX pipe: it connects a port to another port, or to other data sources and sinks in the system such as I/O buses, files, and so on. A channel can connect only a single source and destination port; InputPorts and StickyPorts can connect to only a single channel, while an OutputPort can connect to many channels.

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

Datablock and Template. A datablock represents a unit of data flow along an edge in a graph. A template provides meta-data describing datablocks, and helps map the raw data contained in datablocks to hardware threads on the GPU.

With the PTask API, ptasks encapsulate GPU code, and variables in GPU code are exposed with ports. The programmer composes computations by connecting ports to channels, indicating that data flowing through those channels should be bound dynamically to those variables. A ptask will execute when all its InputPorts have available data, and will produce data at its OutputPorts as a result. Figure 7 shows a PTask graph for the `gemm` multiplication computation discussed in Section 3.2. The variables for matrices A, B, and C in the `gemm` GPU code are exposed using ports:"port:A", "port:B", and "port:C". At run-time, datablocks containing input and output matrices are *pushed* by application code into input channels, causing the ptask to execute, and allowing a datablock containing the result matrix to be *pulled* from the output channel. A single template, which is not shown in the figure, provides meta-data describing the memory layout of matrices, allowing the run-time to orchestrate the computation.

4.1 Dataflow through the Graph

Data flows through a PTask graph as discrete datablocks, moving through channels, and arriving at ports which represent ptask inputs or outputs. When all a ptask's input ports have received data through a channel, the ptask can execute, producing output at its output ports.

Ports can be either "Occupied" or "Unoccupied", depending on whether the port holds a datablock or not. The three sub-types of Port behave differently. An InputPort reads datablocks from an associated

PTask system call	Description
sys_open_graph	Create/open a graph
sys_open_port	Create/open a port
sys_open_ptask	Create/open a ptask
sys_open_channel	Create and bind a channel
sys_open_template	Create/open a template
sys_push	Write to a channel/port
sys_pull	Read from a channel/port
sys_run_graph	Run a graph
sys_terminate_graph	Terminate graph
sys_set_ptask_prio	Set ptask priority
sys_set_geometry	Set iteration space

Table 1: PTask API system calls.

up-stream channel. If Unoccupied, it will read the next datablock from the channel as soon as the channel is not empty. An OutputPort acts as a conduit: an Occupied port pushes its datablock to all its down-stream channels. Unlike the other port types, OutputPorts can be bound to multiple channels. The StickyPort type is a specialization of InputPort that retains its datablock and remains in the Occupied state until a new datablock is written into its channel. The PTask run-time maintains a thread-pool from which it assigns threads to ptasks as necessary: port data movement (as well as GPU dispatch) is performed by threads from this pool. Ports have a *ptflags* member which indicates whether the port is bound to GPU-side resources or is consumed by the run-time. The *ptflags* also indicate whether datablocks flowing through that port are treated as in/out parameters by GPU code.

Channels have parameterizable (non-zero) *capacity*, which is the number of datablocks that the channel may queue between consuming them from its source and passing them to its destination in FIFO order. An application pushes data (using `sys_push`) into channels. If the channel is not ready (because it already has datablocks queued up to its capacity), the `sys_push` call blocks until the channel has available capacity. Likewise, a `sys_pull` call on a channel will block until a datablock arrives at that channel.

4.1.1 Datablocks and Templates

Data flows through a graph as discrete datablocks, even if the external input to and/or output from the graph is a continuous stream of data values. Datablocks refer to and are described by template objects (see below) which are meta-data describing the dimensions and layout of data in the block. The datablock abstraction provides a coherent view on data that may migrate between memory spaces. Datablocks encapsulate buffers in multiple memory spaces using a *buffer-map* property whose entries map memory spaces to device-specific buffer objects. The *buffer-map* tracks which buffer(s) represent the most up-to-date view(s) of the underlying data, enabling a datablock to materialize views in different memory spaces on demand. For example, a datablock may be created based on a buffer in CPU memory. When a ptask is about to execute using that datablock, the runtime will notice that no corresponding buffer exists in the GPU memory space where the ptask has been scheduled, and will create that view accordingly. The converse occurs for data written by the GPU—buffers in the CPU memory domain will be populated lazily based on the GPU version only when a request for that data occurs. Datablocks contain a *record-count* member, used to help manage downstream memory allocation for computations that work with record streams or variable-stride data (see below). Datablocks can be pushed concurrently into multiple channels, can be shared across processes, and are garbage-collected based on reference counts.

Iteration Space. GPU hardware executes in a SIMT (**S**ingle **I**nstruction **M**ultiple **T**hread) fashion, allocating a hardware thread for each point in an iteration space.[2] Hence, the data items on which a particular GPU thread operates must be deduced in GPU code from a unique identifier assigned to each thread. For example, in vector addition, each hardware thread sums the elements at a single index; the iteration space is set of all vector indices, and each thread multiplies its identifier by the element stride to find the offset of the elements it will add. To execute code on the GPU, the PTask run-time must know the iteration space to correctly configure the number of GPU threads. For cases where the mapping between GPU threads and the iteration space is not straightforward (e.g. because threads compute on multiple points in the iteration space, or because input elements do not have fixed stride), the `sys_set_geometry` call allows the programmer to specify GPU thread parameters explicitly. In the common case, the run-time can infer the iteration space and GPU thread configuration from templates.

Templates. Templates provide meta-data that describes the raw data in a datablock's buffers. A template contains a *dimensions* member (stride and three-dimensional array bounds), and a *dbtflags* member that indicates whether the data should be treated as an array of fixed-stride elements, a stream of variable-sized elements, or an opaque byte array. The *dbtflags* also indicate the type(s) of resource(s) the data will be bound to at execution time: examples include buffers in GPU global memory, buffers in GPU constant memory, or formal parameters. Templates serve several purposes. First, they allow the run-time to infer the iteration space when it is not specified by the programmer. In the common case, the iteration space is completely described by the product of the stride and array bounds, and the GPU should launch a hardware thread for every point in the iteration space. Second, templates bound to ports enable the run-time to allocate datablocks that will be consumed by internal and output channels in the graph. Finally, templates enable the run-time to give reasonable feedback to the programmer when API calls are mis-used. For example, constructing a channel requires a template; channel creation fails if either the source or destination port has a template that specifies an incompatible geometry. Similarly, attempts to push datablocks into channels also fail on any template mismatch.

4.1.2 Handling irregular data

Computations on data that lack a fixed stride require templates that describe a variable-geometry, meaning that data-layout is only available dynamically. In such cases, a template fundamentally can not carry sufficient information for the run-time to deduce the iteration space. Use of a variable-geometry channel requires an additional meta-data channel containing per-element geometry information. A meta-data channel carries, at some fixed stride, information that can be used by hardware threads as a map for data in the other channel. For example, if one input channel carries datablocks with records of variable length, its meta-data channel carries datablocks of integers indicating the offset and length of each record in datablocks received on the first channel.

PTask must use templates to allocate datablocks for downstream channels, because the programmer does not write code to allocate datablocks (except for those pushed into input channels), and because the runtime materializes device-side and host-side views of buffers on demand, typically after the datablock itself is allocated. For fixed-stride computations, allocating output datablocks is straightforward: the buffer size is the product of the stride and dimensions of the template. For computations that may produce a variable-length output (such as a select or join), the runtime needs additional information. To address this need, a template on an OutputPort with its variable-size record stream *ptflags* set contains a pointer to an InputPort which is designated to provide output size information. That InputPort's *ptflags* must indicate that it is bound to a run-time input, indicating it is used by the runtime and not by GPU-side code. The run-time generates an output size for each upstream invocation.

[2]The iteration space is the set of all possible assignments of control variables in a loop nest. Conceptually, GPUs execute subsets of an iteration space in parallel.

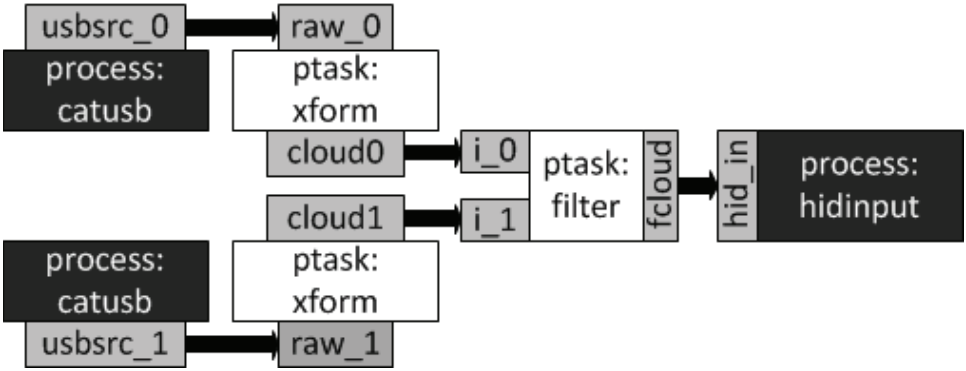

Figure 8: A dataflow graph for the gesture recognition system using the ptask, port, and channel abstractions.

Computations with dynamically determined output sizes require this structure because memory allocated on the GPU must be allocated by a call from the host: dynamic memory allocation in GPU code is generally not possible.[3] Consequently, the canonical approach is to first run a computation on the GPU that determines output size and computes a map of offsets where each hardware thread writes its output. The map of offsets is used to allocate output buffers and is consumed as an additional input to the original computation [30, 34, 35]. In short, we argue that all variable-length GPU computations follow this type of structure, and while the pattern may be burdensome to the programmer, that burden is fundamental to GPU programming, and is not imposed by the PTask programming model. The PTask API enables any variable-geometry structures that are possible with other GPU programming frameworks.

4.2 PTask invocation

A ptask can be in one of four states: **Waiting** (for inputs), **Queued** (inputs available, waiting for a GPU), **Executing** (running on the GPU), or **Completed** (finished execution, waiting to have its outputs consumed). When all of a ptask's input ports are Occupied, the runtime puts the ptask on a run queue and transitions it from Waiting to Queued. A ptask is *invoked* when it is at the head of the run queue and a GPU is available that is capable[4] of running it: invocation is the transition from the Queued to Executing state. When a ptask is invoked, the runtime reads the Datablocks occupying that ptask's InputPorts. For any non-sticky InputPort, this will remove the datablock from the port and will cause the port to pull from its upstream channel; the port goes to the Unoccupied state if the upstream channel is empty. In contrast, a StickyPort remains in the Occupied state when the runtime reads its datablock. When the runtime identifies that it has compute resources available, it chooses a ptask in the Queued state. Scheduling algorithms are considered in more detail in Section 5.1. Upon completion of an invocation, the runtime sets a ptask's state to Completed, and moves its output datablocks to its OutputPorts, if and only if all the output ports are in the Unoccupied state. The ptask remains in the Completed state until all its output ports are occupied, after which it is returned to the Waiting state. The Completed state is necessary because channels have finite capacity. As a result it is possible for execution on the GPU to complete before a downstream channel drains.

4.3 Gestural Interface PTask Graph

The gestural interface system can be expressed as a PTask graph (see Figure 8), yielding multiple advantages. First, the graph eliminates unnecessary communication. A channel connects USB

[3] Although some support for device-side memory allocation has arrived with CUDA 4.0, GPUs and their programming frameworks in general do not support it.

[4] A system may have multiple GPUs with different features, and a ptask can only run on GPUs that supports all features it requires.

source ports (`usbsrc_0`, `usbsrc_1`) to image input ports (`rawimg_0`, `rawimg_1`). Data transfer across this channel eliminates double buffering by sharing a buffer between the USB device driver, PTask run-time, and GPU driver, or with hardware support, going 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 inputs (`i_*`) port of the filter ptask can avoid data copying altogether by reusing the output of one ptask as the input of the next. Because the two **xform** ptasks and the **filter** ptasks run on the GPU, the system can detect that the source and destination memory domains are the same and elide any data movement as a result.

This PTask-based 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. The only application-level code required to cause data to move through the system is the `sys_pull` call on the output channel of the **hidinput** process.

Under this design, the graph expresses concurrency that the run-time exploits without requiring the programmer to write code with explicit threads. Data captured from different camera perspectives can be processed in parallel. When multiple GPUs are present, or a single GPU with support for concurrent kernels [5], the two **xform** PTasks can execute concurrently. Regardless of what GPU-level support for concurrency is present in the system, the PTask design leverages the pipeline parallelism expressed in the graph, for example, by performing data movement along channels in parallel with ptask execution on both the host and the CPU. No code modifications are required by the programmer for the system to take advantage of any of these opportunities for concurrency.

5. IMPLEMENTATION

We have implemented the PTask design described in Section 3 on Windows 7, and integrated it both into a stand-alone user-mode library, and into the device driver for the photogrammetric cameras used in the gestural interface.

The stand-alone library allows us to evaluate the benefits of the model in isolation from the OS. The user-mode framework supports ptasks coded in HLSL (DirectX), CUDA, and OpenCL, implementing dataflow graph support on top of DirectX 11, the CUDA 4.0 driver-API, and the OpenCL implementation provided with NVIDIA's GPU Computing Toolkit 4.0.

The driver-integrated version emulates kernel-level support for ptasks. When ptasks run in the driver, we assign a range of `ioctl` codes in 1:1 correspondence with the system call interface shown in table 1, allowing applications other than the gestural interface to use the PTask API by opening a handle to the camera driver, and calling `ioctl` (`DeviceIoControl` in Windows). The driver-level implementation supports only ptasks coded in HLSL, and is built on top of DXGI, which is the system call interface Windows 7 provides to manage the graphics pipeline.

GPU drivers are vendor-specific and proprietary, so no kernel-facing interface exists to control GPUs. While this remains the case, kernel-level management of GPUs must involve some user-mode component. The Windows Driver Foundation [7] enables a layered design for drivers, allowing us to implement the camera driver as a combination kernel-mode (KMDF) and user-mode (UMDF) driver, where responsibilities of the composite driver are split between kernel- and user-mode components. The component that manages the cameras runs in kernel mode, while the component that implements PTask runs in user-mode. The two components avoid data copy across the user/kernel boundary by mapping the memory used to buffer raw data from the cameras into the address space of the user-mode driver. When the kernel-mode component has captured a new set of frames from a camera, it signals the user-mode component, which can begin working directly on the captured data without requiring buffer copy.

The PTask library comprises roughly 8000 lines of C and C++ code, and the camera driver is implemented in about 3000 lines of C. Code to assemble and manage the ptask graph for the gestural interface introduces approximately an additional 400 LOC.

5.1 PTask Scheduling

PTask scheduling faces several challenges. First, GPU hardware cannot currently be preempted or context-switched, ruling out traditional approaches to time-slicing hardware. Second, true integration with the process scheduler is not currently possible due to lack of an OS-facing interface to control the GPU in Windows. Third, when multiple GPUs are present in a system, data locality becomes the primary determinant of performance. Parallel execution on multiple GPUs may not always be profitable, because the increased latency due to data migration can be greater than the latency reduction gained through concurrency.

Our prototype implements four scheduling modes, **first-available**, **fifo**, **priority**, and **data-aware**. In **first-available** mode, every ptask is assigned a manager thread, and those threads compete for available accelerators. Ready ptasks are not queued in this mode, so when ready ptasks outnumber available accelerators, access is arbitrated by locks on the accelerator data structures. In the common case, with only a single accelerator, this approach is somewhat reasonable because dataflow signaling will wake up threads that need the lock anyway. The **fifo** policy enhances the **first-available** policy with queuing.

In **priority mode**, ptasks are enhanced with a static priority, and proxy priority. The proxy priority is the OS priority of the thread managing its invocation and data flows. Proxy priority allows the system to avoid *priority laundering*, where the priority of the requesting process is ineffective because requests run with the priority of threads in the PTask run-time rather than with the priority of the requester. Proxy priority avoids this by enabling a ptask's manager thread to assume the priority of a requesting process. A ptask's static and proxy priority can both be set with the `sys_set_ptask_prio` system call.

The scheduler manages a ready queue of ptasks, and a list of available accelerators. When any ptask transitions into Queued, Executing, or Completed state, a scheduler thread wakes up and computes an effective priority value for each ptask in the queue. The effective priority is the weighted sum of the ptask's static priority and boost values derived from its current wait time, its average wait time (computed with an exponential moving average), average run time, and its proxy priority. Weights are chosen such that, in general, a ptask's effective priority will increase if a) it has longer than average wait time, b) it has lower than average GPU run time, or c) its proxy priority is high. Boosting priority in response to long waits avoids starvation, boosting in response to short run times increases throughput by preferring low-latency PTasks, and boosting for high proxy priority helps the PTask scheduler respect the priority of the OS process scheduler. Pseudo-code for computing effective priority is shown in Figure 9. When the effective priority update is complete, the scheduler sorts the ready queue in descending order of effective priority.

To assign an accelerator to a ptask, the scheduler first considers the head of the queue, and chooses from the list of available accelerators based on *fitness* and *strength*. An accelerator's fitness is a function of whether the accelerator supports the execution environment and feature set required by the ptask: unfit accelerators are simply eliminated from the pool of candidates. The strength of the accelerator is the product of the number of cores, the core clock speed, and the memory clock speed: in our experience, this is an imperfect but effective heuristic for ranking accelerators such that low-latency execution is preferred. The scheduler always chooses the strongest accelerator when a choice is available. If the scheduler is unable to assign an accelerator to the ptask at the head of the queue, it iterates over the rest of the queue until an assignment can be made. If no assignment can be made, the scheduler blocks. On a successful assignment, the scheduler removes the ptask from the queue, assigns the accelerator to it, moves the ptask to Executing state, and signals the ptask's manager thread that it can execute. The scheduler thread repeats this process until it runs out of

```
void update_eff_prio(ptasks) {
  avg_gpu = avg_gpu_time(ptasks);
  avg_cwait = avg_current_wait(ptasks);
  avg_dwait = avg_decayed_wait(ptasks);
  avg_pprio = avg_proxy_prio(ptasks);
  foreach(p in ptasks) {
    boost = W_0*(p->last_cwait - avg_cwait);
    boost += W_1*(p->avg_wait - avg_dwait);
    boost += W_2*(p->avg_gpu - avg_gpu);
    boost += W_3*(p->proxy_prio - avg_pprio);
    p->eff_prio = p->prio + boost;
}}
gpu match_gpu(ptask) {
  gpu_list = available_gpus();
  remove_unfit(gpu_list, ptask);
  sort(gpu_list); // by descending strength
  return remove_head(gpu_list);
}
void schedule() {
  update_eff_prio(ptasks);
  sort(ptasks); // by descending eff prio
  while(gpus_available() && size(ptasks)>0) {
    foreach(p in ptasks) {
      best_gpu = match_gpu(p);
      if(best_gpu != null) {
        remove(ptasks, p);
        p->dispatch_gpu = best_gpu;
        signal_dispatch(p);
        return;
}}}}
```

Figure 9: Pseudo-code for algorithms used by PTask's priority scheduling algorithm.

available accelerators or ptasks on the run queue, and then blocks waiting for the next scheduler-relevant event. Pseudo-code for scheduling and matching accelerators to ptasks is shown in Figure 9 as the `schedule` and `match_gpu` functions respectively.

The **data-aware** mode uses the same effective priority system that the **priority** policy uses, but alters the accelerator selection algorithm to consider the memory spaces where a ptask's inputs are currently up-to-date. If a system supports multiple GPUs, the inputs required by a ptask may have been most recently written in the memory space of another GPU. The dataaware policy finds the accelerator where the majority of a ptask's inputs are up-to-date, designating it the ptask's preferred accelerator. If the preferred accelerator is available, the scheduler assigns it. Otherwise, the scheduler examines the ptask's effective priority to decide whether to schedule it on an accelerator that requires data migration. PTasks with high effective priority relative to a parameterizable threshold (empirically determined) will be assigned to the strongest fit available accelerator. If a ptask has low effective priority the scheduler will leave it on on the queue in hopes that its preferred accelerator will become available again soon. The policy is not work-conserving: it is possible that Queued ptasks do not execute even when accelerators are available. However, waiting for a preferred accelerator often incurs lower latency than migrating data between memory spaces. The system guarantees that a ptask will eventually execute, because long wait time will cause effective priority boosts that ultimately ensure it runs on its preferred accelerator or runs elsewhere.

5.2 Prototype limitations

It is the express vision of this work that all GPU computations use ptasks. This vision can only be enforced if all access to the GPU is mediated by a kernel-mode implementation of PTask. Because a substantial part of our prototype must run in user-mode, it remains possible for processes to access

the GPU directly, potentially subverting the fairness and isolation mechanisms of our prototype. This limitation is a property of our prototype and is not fundamental to the proposed system.

The scheduling implementations in our prototype do not consider GPU memory demands that exceed the physical memory on GPUs. GPU drivers use swapping to virtualize GPU physical memory, but allow the programmer to create buffers in the GPU memory that cannot be swapped. Because our prototype mostly allocates unswappable resources, some additional memory management would be required to ensure that a PTask's inputs can always be materialized on the GPU under high memory load. Our prototype does not address this problem.

5.3 GPU scheduling on Linux

We do not port the PTask API to Linux, rather we exploit our control over the kernel and add GPU accounting state to Linux's `task_struct`, the data structure that holds information about a kernel thread. We add blocking system calls that inform the kernel about GPU activity. GPU kernel preemption is not supported by the hardware, which limits scheduling to non-preemptive policies enforced at GPU kernel boundaries. The GPU driver actually places computations on the GPU, but source code for commercial drivers for CUDA and OpenCL are not available, so we manually insert the Linux system calls into applications that use the GPU. By giving the kernel visibility into GPU use, it can enforce global properties, like fairness relative to scheduling priority.

The kernel uses a non-work-conserving scheduling algorithm similar to the token bucket algorithm. The choice of the algorithm is dictated by GPU execution being non-preemptive. Each process p using a GPU maintains its GPU budget B_p which reflects the current eligibility of a process to use the GPU. B_p is reduced by the execution time (t_p) used by p each time it executes on the GPU. B_p is incremented once per period T by quanta q to regain GPU access. If a process has a negative GPU budget, then the kernel will block any GPU invocation until the budget becomes positive. The actual GPU share of a process is governed by the maximum budget $Bmax_p$, which is set proportionally to the Linux priority n_p of the process p. The maximum budget $Bmax_p$ and the replenish period T are updated dynamically by the scheduler by taking into account dynamic GPU execution time and changes in the number of processes using a GPU. Namely, $Bmax_p = \frac{n_p}{\sum_{i \in P} n_i} \sum_{i \in P} t_i$ and $q = T = \alpha * \min_{i \in P} t_i$, where P denotes the set of kernel threads using a GPU and $\alpha \leq 1$ is a safety coefficient to avoid idling due to clock quantization. These values are updated upon every GPU completion, invocation or completion of a process using a GPU, and every 10 µsec, which is about an order of magnitude lower than the expected GPU kernel time.

The scheduler maintains the running average of the GPU usage time by each process. It assumes periodic GPU execution, which typically holds for GPU-bound workloads. TimeGraph [45] uses a similar scheduler but with statically assigned parameters.

6. EVALUATION

In this section, we evaluate our PTask prototype using a gestural UI on Windows 7, an encrypted file system on Linux, and microbenchmarks. Our experience with the gestural interface shows that kernel-level management of objects in a PTask graph is necessary to provide optimal data movement in the general case. Our work with the encrypted file system in Linux shows that OS visibility into GPU usage is necessary for the OS to provide fairness and isolation.

6.1 Gestural interface

The primary benefits of leveraging a GPU for gestural interface come from offloading work to the GPU. In this section, we show that GPU-offload is required to make a gestural interface possible, and we quantify the costs incurred by the offload and supporting framework. To this end, we compare the performance of five gestural interface implementations: *host-based, handcode, pipes, modular*, and *ptask*.

The *host-based* implementation uses a Quad-Core CPU to perform all steps of the computation in the

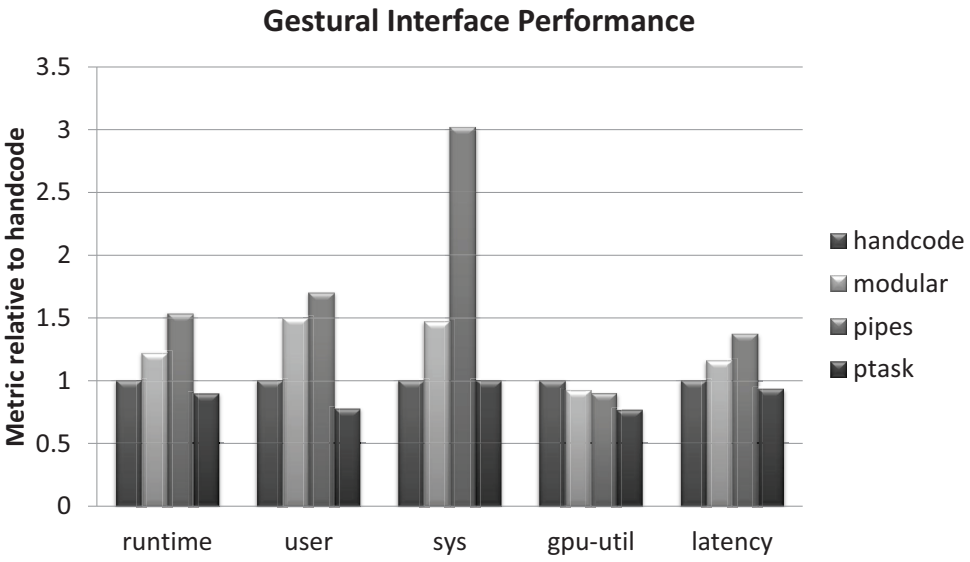

Figure 10: Relative performance metrics for different gestural interface implementations. The runtime column shows the relative wall-clock time taken for each implementation to process 1000 frames. User, system, and gpu-util are the relative user time, system time, and gpu utilization. The latency column is the relative end-to-end latency for a single frame. In all columns, lower is better.

user-mode component of the camera driver. This implementation provides a performance baseline, and does not use the GPU. The *handcode* version is implemented in the user-mode component of the camera driver using DXGI system calls to offload work to the GPU. This version is hand-coded to avoid all needless data copy where subsequent GPU-kernels have a producer-consumer relationship. The *pipes* implementation uses four separate processes connected by pipes, as described in Section 2. To avoid double-buffering, raw camera data buffers used by the kernel-mode component of the camera driver are mapped into the address space of the `catusb` program, and the driver signals when new data is available. The `xform` and `filter` programs read data from pipes, invoke GPU-kernels and write results to pipes. We consider this implementation because it is modular and composable. The *modular* version is uses the same module design used to build the *pipes* version, but includes all components in a single process. This eliminates IPC overheads incurred by using pipes to communicate between modules, while still incurring overheads for migrating data to and from the GPU at module boundaries. The *PTask* implementation is described in detail in Section 5. It uses the PTask runtime in the camera driver, and uses ptasks to perform the **xform** and **filter** steps of on the GPU. The **hidinput** step is performed on the CPU in the camera driver.

We compare these four implementations in a "real-time" mode and an "unconstrained" mode. The former is the deployment target: the two cameras in the system drive all data flow. In "unconstrained", the system is driven by in-memory recordings of 1,000 frames of raw camera data, allowing us to measure maximum throughput for implementations that would otherwise be I/O bound. In "real-time" mode we are primarily concerned with utilization and end-to-end latency, while for "unconstrained", throughput is the primary metric.

The PTask infrastructure incurs overheads: additional memory to represent the graph (ports, channels, ptasks, the scheduler), and additional threads to drive the system. We measure the total number of threads used by the system as well as the change in memory footprint over the simplest hand-coded implementation. Table 2 and Figure 10 characterize the performance of the gestural interface system

bnc			fps	tput (MB/s)	lat (ms)	user	sys	gpu	gmem	thrds	ws-delta
Core2-Quad		real-time	20.2	35.8	36.6	78.1	3.9	–	–	1	–
GTX580	handcode	real-time	30	53.8	10.5	4.0	5.3	21.0	138	1	–
		unconstrained	138	248.2	–	2.4	6.4	41.8	138	1	–
	modular	real-time	30	53.8	12.2	6.0	8.1	19.4	72	1	0.8 (1%)
		unconstrained	113	202.3	–	5.7	8.6	55.7	72	1	0.9 (1%)
	pipes	real-time	30	53.8	14.4	6.8	16.6	18.9	72	3	45.3 (58%)
		unconstrained	90	161.9	–	12.4	24.6	55.4	76	3	46.3 (59%)
	ptask	real-time	30	53.8	9.8	3.1	5.5	16.1	71	7	0.7 (1%)
		unconstrained	154	275.3	–	4.9	8.8	65.7	79	7	1.4 (2%)

Table 2: Gestural interface performance on a Core-2 Quad, and an NVIDIA GTX 580 GPU. PTasks achieve higher maximum throughput than a hand-coded implementation and can support real-time data rates with low CPU utilization. The fps column is camera frames-per-second, tput is throughput in MB/s, lat is end-to-end latency (time from capture to delivered user-input). CPU utilization is broken down into user and kernel (sys) percentages. The gpu and gmem columns are GPU utilization and GPU memory usage. The thrds column is number of threads, and ws-delta is the increase (in MB) of the main memory footprint over the handcode version. The host-based implementation cannot deliver real-time frame rates, so the real-time and unconstrained implementations have identical performance.

running on the Core2-Quad, and on an NVIDIA GTX580 GPU. The data include frames-per-second, throughput, end-to-end latency, CPU utilization (broken down into user/kernel), gpu utilization, the number of threads, and memory footprint delta.

The PTask implementation outperforms all others. The CPU-based implementation achieves a frame rate of only 20 frames per second, a mere 67% of the real-time frame rate of the cameras, using 82% of the CPU to do so. The bilateral filtering in the **filter** step is the primary bottleneck. The *pipes* version achieves real-time frame rates, using more than 23% of the CPU to do it, consuming about 45% more memory than the hand-coded and *ptask*. Both the *handcode* and *ptask* implementations deliver very low CPU utilization at real-time frame rates (9.3% and 8.6% respectively) but *ptask* achieves the highest throughput, outstripping *handcode* by 11.6%. Outperforming *handcode* is significant because the *handcode* system enjoys all the data movement optimizations that the *ptask* version enjoys: *ptask* has higher throughput because it overlaps communication and computation in ways the *handcode* system does not. The PTask implementation is not only significantly lower in code complexity, but retains a level of modularity that the handcode version sacrifices.

6.2 GPU sharing on Linux

We ran EncFS [29], a FUSE-based encrypted file system for Linux, modified to use a GPU for AES encryption and decryption. We used an NVIDIA GTX470 GPU and Intel Core i5 3.20GHz CPU, on a machine with 12GB of RAM and two SATA SSD 80GB drives connected in striped RAID configuration using the standard Linux software raid driver (`md`). We implemented AES with the XTS chaining mode, which is suitable for parallel hardware without compromising the cipher strength, and is now the standard mode used for encryption of storage devices [3]. We configured FUSE to pass data blocks of up to 1MB from the kernel, and we modified EncFS to pass the entire buffer to the GPU for encryption or decryption. The larger block size amortizes the cost of moving the data to and from the GPU and enables higher degree of parallelism.

A sequential read and write of a 200MB file are 17% and 28% faster for the version of EncFS that uses the GPU than the version that uses the SSL software library implementation (Table 3) , with results averaged over five executions.

On Linux, using the GPU can completely defeat the kernel's scheduling priority. With several processes periodically contending for the GPU, the one invoking longer GPU kernels will effectively

	CPU	1 GPU task		2 GPU tasks	
		Linux	PTSched	Linux	PTSched
read	247 MB/s	−10.0×	1.16×	−30.9×	1.16×
write	82 MB/s	−8.2×	1.21×	−10.0×	1.20×

Table 3: Bandwidth measurements for sequential read or write of a 200MB file on an encrypted file system, relative to the CPU performing encryption (CPU column). Negative numbers indicate reduced bandwidth. There are one or two concurrently executing background GPU-heavy tasks, running with the default Linux scheduler (Linux) or our scheduler (PTSched).

	MB/s	
r/w (nice)	Linux	PTSched
read (0)	170	132
read (-20)	171	224
write (0)	58	51
write (-20)	58	75

Table 4: Bandwidth measurements for sequential read or write of a 200MB file on an encrypted file system by two EncFS clients that have different nice values (0 or -20).

monopolize the GPU regardless of its OS priority. Our GPU scheduling mechanism in the kernel (§5.3), here called PTSched, eliminates this problem.

Table 3 shows the results of running EncFS concurrently with one or two competing background GPU-bound tasks (a loop of 19ms CUDA program invocations). To evaluate the Linux scheduler in the most favorable light, the GPU-bound tasks are set to the minimum scheduling priority (nice +19) while the EncFS process and its clients have the highest priority (-20). The tasks invoke GPU kernels in a tight loop, which challenges the scheduler. The results show that if Linux is not informed about GPU use, then GPU use can invert scheduling priority, leading to a drastic degradation of file system performance (e.g., a 30× slowdown). Once the kernel is informed of GPU use (the PTSched column), a relatively simple scheduling algorithm restores consistent, system-wide kernel scheduling priority.

Another experiment shows the effort needed to maintain global scheduling priorities (Table 4). We modified EncFS to propagate the client's OS scheduling priority to the EncFS worker thread that services that client's request. Each experiment has two EncFS threads of different nice values doing a read or write concurrently on two CPUs, but contending for the GPU. Without our GPU OS scheduling (Linux), the client throughput is not affected by the client's nice level. With PTSched, the client with higher priority (nice -20) achieves higher throughput at the expense of reduced throughput for the lower priority client. Note the aggregate throughput of this experiment is about 25% higher than the above experiment (e.g., 356 MB/s read bandwidth vs. 286 MB/s) because it uses two CPUs and EncFS has significant CPU work.

6.3 Microbenchmarks: benefits of dataflow

To demonstrate that PTask can improve performance and eliminate unnecessary data migration, we show that composition of basic GPU-accelerated programs into a dataflow graph can outperform not only designs that rely on IPC mechanisms such as pipes, but can also outperform programs that are hand-coded to minimize data movement.

The micro-benchmarks we measure are shown in Table 5. The benchmarks include bitonic `sort`, matrix multiplication (`gemm`), matrix additions (`madd`), and matrix copy (`mcopy`) kernels. To explore the impact on performance of different graph shapes, and relative costs of data copy and GPU-execution latency, we consider a range of data sizes (matrices ranging from 64 × 64 through 1024

bnc	description	P_{min}	P_{max}	C_{min}	C_{max}
gemm	matrix multiply	4	60	14	190
sort	bitonic sort	13	41	29	84
madd	matrix addition	4	60	14	190
mcopy	matrix copy	2	30	7	95
fdtd	3d stencil	30	60	85	170
pca	principal components	229	1197	796	4140
grpby	group by	7	7	13	13

Table 5: Micro-benchmarks evaluated across a range of PTask graph sizes and shapes. P_{min} and P_{max} are the number of ptasks, while C_{min} and C_{max} are the number of channels in the smallest and largest graphs.

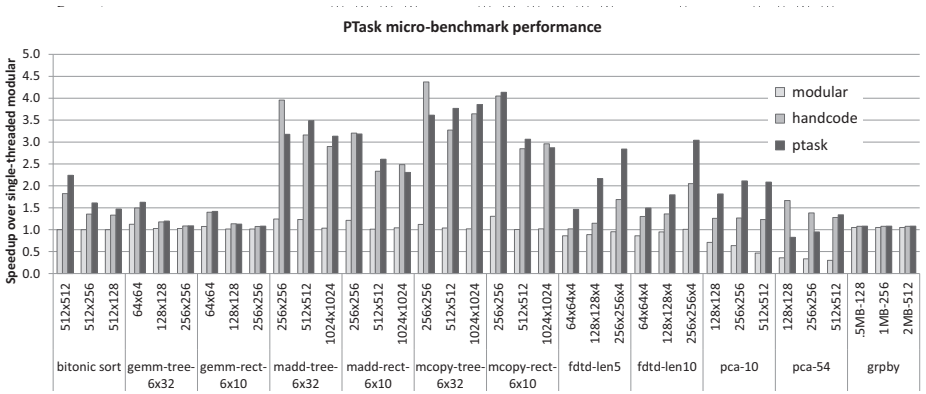

Figure 11: Speedup of various GPU-accelerated implementations of our benchmarks over single-thread, modular GPU-based implementations, for input matrix and image sizes ranging from 64x64 through 1024x1024. The PTask implementation has geometric mean speedup of 10% over handcoded, and $1.93\times$ over modular implementation.

$\times$ 1024), and graphs structured both as binary trees (for example, $((A \times B) \times (C \times D))$) and graphs structured as "rectangles", where each column is an independent computation (for example, $((A \times B) \times C)$ running concurrently with with $((D \times E) \times F)$ is a rectangular graph 2 columns wide and 2 rows deep). This allows us to explore a range of graph sizes and shapes for `gemm`, `madd`, and `mcopy`.

To illustrate the generality of PTask, we also consider four higher-level benchmarks including bitonic `sort`, `fdtd`, `pca`, and `grpby`. Bitonic sort is an example of a workload where the shape and size of the graph is fundamentally tied to the input dimensions. Bitonic sort is performed on a GPU as a series of pair-wise sort steps, alternated with transposition steps that effectively change the stride of the pairwise comparisons in the subsequent sort step. We express this as a linear graph connected by channels of 41 ptasks that implement either the sort or the transposition. The `pca` workload implements the iterative principal components analysis described by Andrecut [9]. We compute the first three components for each matrix with 10 and 54 iterations per component, which exceeds 95% and 99% convergence respectively for all cases. The graph for 10 iterations uses 229 ptasks and 796 channels, while the 54 iteration graph uses 1197 ptasks and 4140 channels. The `grpby` workload uses a graph of 7 ptasks and 9 channels to implement a "group by" operation over a range of integers (0.5, 1, and 2 million) and a known number of unique keys (128, 256, and 512). The `fdtd` workload is an electromagnetic analysis tool implementing a Finite Difference Time Domain

model of a hexahedral cavity with conducting walls [44], running 10 iterations over a stream of inputs of lengths 5 and 10.

For each benchmark, we compare the performance of four different implementations: single-threaded modular, modular, handcoded, and PTask. The *single-threaded modular* implementation performs the equivalent computation as a sequence of calls to a subroutine that encapsulates the calling of the underlying operation on the GPU. For example, for matrix addition, a single-thread makes a sequence of calls to a subroutine called `matrix_add`, which copies two matrices to the GPU, performs the addition, and copies the result back to host memory. The configuration is called modular because the subroutine can be freely composed, but as we saw in Figure 6, this modularity comes at the cost of unnecessary data copies to and from the GPU. The *modular* implementation performs the equivalent computation using the same strategy for encapsulating operations on the GPU that is used in the *single-threaded modular* implementation, but makes liberal use of multi-threading to attempt to hide the latency of data movement, overlapping data copy to and from the GPU with execution on the GPU. This approach entails a significant level of code complexity for thread management and producer/consumer synchronization, but it does give us a lower bound on the performance of a composed, IPC based implementation (for example `matrix_add A B | matrix_add C`). The *handcode* implementation represents a single-threaded implementation handcoded to minimize data copy, at the expense of sacrificing composability. We consider this case to represent the most likely solution an expert CUDA programmer would choose. The *PTask* implementation executes the workload as a data flow graph as described above. In the case of `pca`, our *handcode* implementation is based on the CUBLAS [58] implementation described in [8].

Performance measurements were taken on a Windows 7 x64 desktop machine with a 4-core Intel Xeon running at 2.67 GHz, 6GB of RAM, and an NVIDIA GeForce GTX 580 GPU which features 512 processing cores and 1.5GB of memory. Data are averaged over three runs. Figure 11 shows speedup over the single-threaded modular implementation for our modular, handcode, and PTask implementations. The single-threaded modular implementation induces considerable unnecessary data copy, and cannot hide transfer latency, so it is always the slowest. In general, the PTask implementations out-perform the handcode versions because a PTask graph can take advantage of inherent pipeline parallelism to lower end-to-end latency. In some instances the handcode version outperforms PTask (e.g. tree, matrix addition for 6x32 sized graphs). In these cases, PTask overheads such as additional threading, GPU scheduling, and synchronization outstrip performance gains from overlapping GPU-execution and data transfer. The modular implementation is strictly less performant than PTask and handcode, due to unnecessary data migration. The `grpby` workload relies on device-side memory allocation that has to be cleaned up between GPU invocations. Consequently, communication and invocation are synchronous both for PTask and handcode, and PTask's additional overheads make it slightly slower.

Overall, *PTask* has geometric mean speedup of 10% over handcode, and $1.93\times$ over modular implementation, showing that performing these computations in a dataflow manner is worthwhile both in terms of performance and in terms of preserving modularity.

6.4 PTask Scheduling

To evaluate the efficacy of PTask scheduling policies, we add a second GTX 580 to the system described in Section 6.3. Figure 12 shows speedup on 2 GPUs over execution on a single GPU, for three of the scheduling policies defined in Section 5.1: **first-available**, **priority**, and **data-aware** (**fifo** is omitted because it is similar to **first-available**). The data shown represent rectangular graphs of matrix multiplication and addition kernels. All graphs have a breadth of 8, and depth from 1 to 6. Producer-consumer relationships exist only along the vertical axis. The data show that data-aware scheduling is effective at providing scalability. As the PTask graph gets deeper, priority scheduling alone actually hurts performance due to data-oblivious ptask assignment. Migrating a datablock across GPUs must be done through main memory, incurring high latency. Because the priority algorithm does not consider locality, it induces frequent data-block transfers, making the system slower

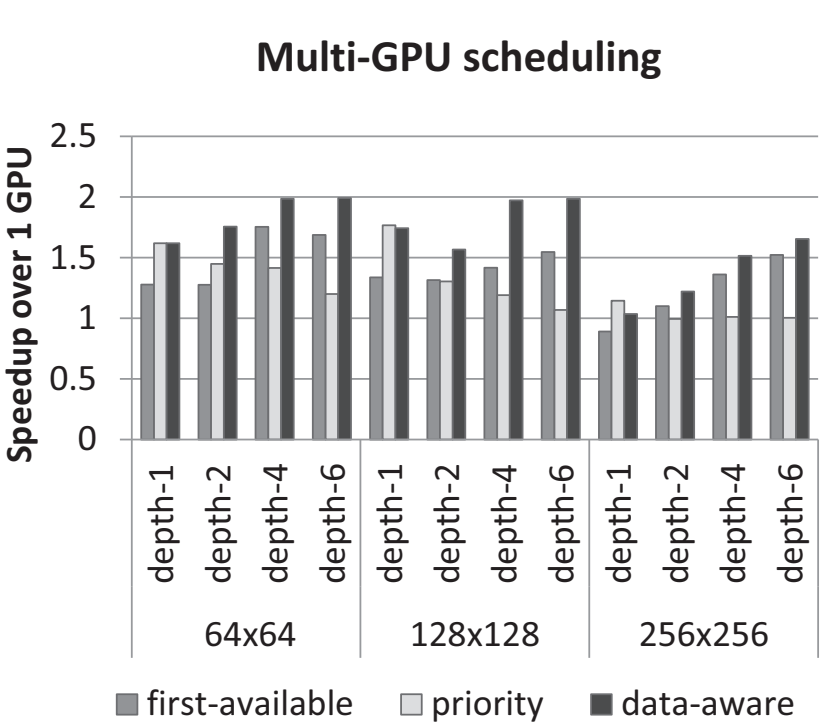

Figure 12: Speedup of PTask using 2 GTX580 GPUs over PTask using 1 GTX580 GPU, for a range of graph sizes using matrix multiplication ptasks. Data for the fifo scheduling policy is omitted because it is similar to that for first-available.

than the single GPU case for deep graphs. The rate of GPU-GPU migration goes from from an average over all workloads of 3.71% for **first-available** to 14.6% for **priority**. By contrast, the data-aware policy avoids needless data movement, bringing the average over all workloads down to 0.6%. Consequently, scalability generally improves as graph depth improves, because the scheduler can almost always make an assignment that preserves locality.

Figure 13 compares the throughput of our scheduling policies for 4 concurrently running PTask graphs (each of 36 ptasks) on a single GPU. Data for the data-aware policy are omitted because the data-aware and priority policies are identical with just one GPU. The first-available and fifo policies ignore priority completely, but the priority policy is able to deliver throughput that increases near-linearly with the priority of the graph.

7. RELATED WORK

General-purpose GPU computing. The research community has focused considerable effort on the problem of providing a general-purpose programming interface to the specialized hardware supported by GPUs (GPGPU). GPGPU computing frameworks such as CUDA [59], OpenCL [47], and others [16, 20, 33, 54, 68] provide an expressive platform, but are implemented using heavy-weight user-mode libraries and run-times combined with proprietary GPU device drivers. These systems integrate poorly with the operating system. The programming interfaces provided by these frameworks can be implemented on top of the PTask API.

Offloading. A large body of research has been dedicated to offloading techniques such as channel processors [1], smart disks [22, 46, 63], and TCP offload [24]. PTasks make it easier to create applications that offload work to a programmable device by exposing kernel abstractions. The Hydra

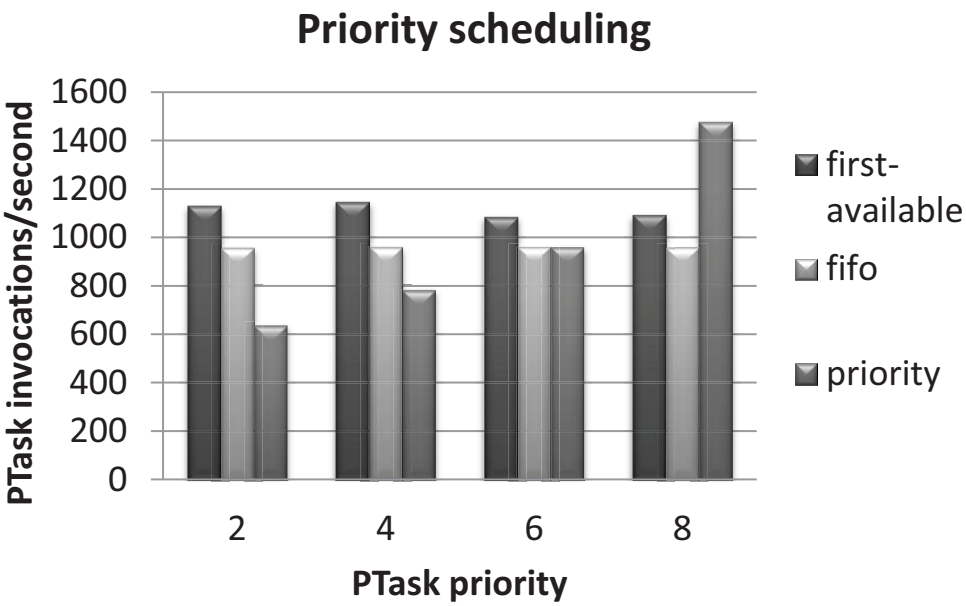

Figure 13: Number of ptask invocations per second on 1 GTX580 GPU for four competing PTask graphs with different ptask priorities (2,4,6,8) using 6x6 rectangular graphs of matrix multiplication ptasks. The trends are the same independent of graph size, shape, or ptask composition. Data for the data-aware policy is not shown because the the priority and data-aware policies are identical when only 1 GPU is present.

framework [69] also provides a graph-based dataflow programming model for offloading tasks to peripheral devices. Unlike our work, the Hydra model requires components to communicate using RPC through a common runtime and an API that cannot be supported by current GPU hardware. Hydra cannot provide the fairness and isolation guarantees of PTasks.

OS support and architectural support. The Synthesis kernel [53] orchestrates threads of execution to form a dataflow graph, using *switches* and *pumps* to connect producers and consumers, and allowing interrupts to start threads at nodes in the graph. Synthesis uses dynamic construction of code to orchestrate data movement; in contrast, our abstractions are implemented on top of events, deferred procedure calls, and work queues. PTasks are exposed through the system call interface. The Helios [57] OS supports *satellite kernels*, which communicate via RPC. The Barrelfish OS [15] treats the hardware as a network of independent, heterogeneous cores communicating using RPC. Unlike our model, both Helios and Barrelfish propose abstractions that cannot be supported by current GPUs, which lack the architectural features to run OS code or communicate via RPC.

The Scout operating system [55] supports the construction of optimized code paths, with the goals of enabling modularity while eliminating wasted data copy and improving scheduling by providing the scheduler with richer information about interdependences between modules along a path. PTask's ptasks and ports share some similar to Scout's routers and services respectively. Scout's paths are different from PTask's graphs in that a path is a single path of data through a series of modules, while a graph represents potentially many paths contributing to a single computation. Unlike Scout, PTask graphs can handle fan-out.

The SPIN operating system [19] allows applications to change the OS interface and specialize OS behavior to improve performance. With similar motivation to PTask's channel specialization, SPIN

enables data movement optimizations such as direct data streaming between disk and network to eliminate redundant buffering (such as `sendfile`). In SPIN, applications must implement these specializations explicitly, while a major goal of PTask is letting the system take advantage of optimization opportunities where they arise, without programmer intervention.

Gelado et al. propose ADSM [27]—an asymmetric distributed shared memory abstraction, in which in-memory data structures required by GPUs are automatically identified and marshalled to the GPU local memory and back, eliminating unnecessary data transfers. Unlike PTask, ADSM addresses only the data management issues without dealing with GPU scheduling.

Scheduling for heterogeneous processors. The TimeGraph [45] GPU scheduler provides isolation and prioritization capabilities for GPU resource management, targeting real-time environments. TimeGraph does not propose OS abstractions to manage GPUs: priorities are set statically in a file in `/etc/`, making TimeGraph unable to integrate with process priority. The design space of scheduling algorithms and policies for heterogeneous systems [13, 14, 17, 43] has received considerable research attention, along with approaches to dynamic selection between CPU and GPU execution [11, 12, 23, 64]. In contrast, ptasks are bound to either a class of accelerators or a CPU, and can not re-target dynamically.

Dataflow and streaming. Hardware systems such as Intel IXP [40] Imagine [2], and SCORE [21] enable programmers or compilers to express parallel computation as a graph of routines whose edges represent data movement. The classic data flow abstractions proposed for Monsoon and Id [61] target environments in which dataflow execution is supported directly by the processor hardware. A StreamIt [66] application is a graph of nodes which send and receive items to each other over channels. DirectShow [51] supports graph-based parallelism through "filter graphs" in which filters are connected to each other with input and output pins. StreamIt and DirectShow are programming models with dedicated compiler and user-mode runtime support. PTask is a streaming programming model that focuses on integration with OS abstractions.

Dryad [41] is a graph-based fault-tolerant programming model for distributed parallel execution in data center. PTask targets an entirely different execution environment. The Click modular router [49] provides a graph-based programming model. Click is always single-threaded, while PTask makes extensive use of threading to manage execution on and communication with the GPU. CODE2 [56], and P-RIO [52] use similar techniques to PTask to express computation as a graph with the goal of explicitly separating communication from computation at the programmer interface: neither system addresses problems of heterogeneity or OS support.

Sponge [38] is a compilation framework for GPUs using synchronous dataflow (SDF) streaming languages, addressing problems of portability for GPU-side code across different generations of GPUs and CPUs, as well abstracting hardware details such as memory hierarchy and threading models. Like other SDF languages [50] such as LUSTRE [31] and ESTEREL [18], Sponge provides a static schedule, while PTask does not. More importantly, Sponge is primarily concerned with optimizing compiler-generated GPU-side code, while PTask addresses systems-level issues. A PTask-based system could benefit from Sponge support, and vice-versa.

Liquid Metal [39] and Lime [10] provide programming environments for heterogeneous targets such as systems comprising CPUs and FGPAs. Lime's filters, and I/O containers allow a computation to be expressed (by the compiler, in intermediate form) as a pipeline, while PTask's graph-structured computation is expressed explicitly by the programmer. Lime's buffer objects provide a similar encapsulation of data movement across memory domains to that provided by PTask's channels and datablocks. Flextream [37] is compilation framework for the SDF model that dynamically adapts applications to target architectures in the face of changing availability of FPGA, GPU, or CPU resources. Like PTask, Flextream applications are represented as a graph. As language-level tools, Liquid Metal, Lime and Flextream provide no OS-level support and therefore cannot address isolation/fairness guarantees, and cannot address data sharing across processes or in contexts where

accelerators do not have the required language-level support. PTask is not coupled with a particular language or user-mode runtime.

I/O and data movement. PacketShader [32] is a software router that accelerates packet processing on GPUs and SSLShader [42] accelerates a secure sockets layer server by offloading AES and RSA computations to GPUs. Both SSLShader and PacketShader rely heavily on batching (along with overlap of computation/communication with the GPU) to address the overheads of I/O to the GPU and concomitant kernel/user switches. The PTask system could help by reducing kernel/user switches and eliminating double buffering between the GPU and NIC. IO-Lite [60] supports unified buffering and caching to minimize data movement. Unlike IO-Lite's *buffer aggregate* abstraction, PTask's datablocks are mutable, but PTask's channel implementations share IO-Lite's technique of eliminating double-buffering with memory-mapping (also similar to *fbufs* [25], Container Shipping [62], and zero-copy mechanisms proposed by Thadani et. al [65]). While IO-Lite addresses data-movement across protection domains, it does not address the problem of data movement across disjoint/incoherent memory spaces, such as those private to a GPU or other accelerator.

8. CONCLUSION

This paper proposes a new set of OS abstractions for accelerators such as GPUs called the PTask API. PTasks expose only enough hardware detail as is required to enable programmers to achieve good performance and low latency, while providing abstractions that preserve modularity and composability. The PTask API promotes GPUs to a general-purpose, shared compute resource, managed by the OS, which can provide fairness and isolation.

9. ACKNOWLEDGEMENTS

We thank Ashwin Prasad for implementation of the **fdtd** and **grpby** microbenchmarks. This research is supported by NSF Career award CNS-0644205, NSF award CNS-1017785, and a 2010 NVIDIA research grant. We thank our shepherd, Steve Hand for his valuable and detailed feedback.

10. REFERENCES

[1] *IBM 709 electronic data-processing system: advance description*. I.B.M., White Plains, NY, 1957.

[2] *The Imagine Stream Processor*, 2002.

[3] Recommendation for block cipher modesl of operation: the xts-aes mode for confidentiality on block-oriented storage devices. *National Institute of Standards and Technology, Special Publication 800-e8E*, 2009.

[4] NVIDIA GPUDirect. 2011.

[5] NVIDIA's Next Generation CUDATM Compute Architecture: Fermi. 2011.

[6] Top 500 supercomputer sites. 2011.

[7] Windows Driver Foundation (WDF). 2011.

[8] M. Andrecut. Parallel GPU Implementation of Iterative PCA Algorithms. *ArXiv e-prints*, Nov. 2008.

[9] M. Andrecut. Parallel GPU Implementation of Iterative PCA Algorithms. *Journal of Computational Biology*, 16(11), Nov. 2009.

[10] J. S. Auerbach, D. F. Bacon, P. Cheng, and R. M. Rabbah. Lime: a java-compatible and synthesizable language for heterogeneous architectures. In *OOPSLA*. ACM, 2010.

[11] C. Augonnet and R. Namyst. StarPU: A Unified Runtime System for Heterogeneous Multi-core Architectures.

[12] C. Augonnet, S. Thibault, R. Namyst, and M. Nijhuis. Exploiting the Cell/BE Architecture with the StarPU Unified Runtime System. In *SAMOS '09*, pages 329–339, 2009.

[13] R. M. Badia, J. Labarta, R. Sirvent, J. M. Pérez, J. M. Cela, and R. Grima. Programming Grid Applications with GRID Superscalar. *Journal of Grid Computing*, 1:2003, 2003.

[14] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor platforms. 2004.

[15] 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*, 2009.

[16] A. Bayoumi, M. Chu, Y. Hanafy, P. Harrell, and G. Refai-Ahmed. Scientific and Engineering Computing Using ATI Stream Technology. *Computing in Science and Engineering*, 11(6):92–97, 2009.

[17] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: a programming model for the cell BE architecture. In *SC 2006*.

[18] G. Berry and G. Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19:87–152, November 1992.

[19] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. *SIGOPS Oper. Syst. Rev.*, 29:267–283, December 1995.

[20] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM TRANSACTIONS ON GRAPHICS*, 2004.

[21] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream computations organized for reconfigurable execution (score). FPL '00, 2000.

[22] S. C. Chiu, W.-k. Liao, A. N. Choudhary, and M. T. Kandemir. Processor-embedded distributed smart disks for I/O-intensive workloads: architectures, performance models and evaluation. *J. Parallel Distrib. Comput.*, 65(4):532–551, 2005.

[23] C. H. Crawford, P. Henning, M. Kistler, and C. Wright. Accelerating computing with the cell broadband engine processor. In *CF 2008*, 2008.

[24] A. Currid. TCP offload to the rescue. *Queue*, 2(3):58–65, 2004.

[25] P. Druschel and L. L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. *SIGOPS Oper. Syst. Rev.*, 27:189–202, December 1993.

[26] 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.

[27] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. ASPLOS '10, 2010.

[28] S. B. Gokturk, H. Yalcin, and C. Bamji. A time-of-flight depth sensor - system description, issues and solutions. In *CVPRW*, 2004.

[29] V. Gough. *EncFs*. `http://www.arg0.net/encfs`.

[30] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[31] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[32] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a GPU-accelerated software router. *SIGCOMM Comput. Commun. Rev.*, 40:195–206, August 2010.

[33] T. D. Han and T. S. Abdelrahman. hiCUDA: a high-level directive-based language for GPU programming. In *GPGPU 2009*.

[34] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM.

[35] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. SIGMOD '08, 2008.

[36] M. Hirsch, D. Lanman, H. Holtzman, and R. Raskar. BiDi screen: a thin, depth-sensing LCD for 3D interaction using light fields. *ACM Trans. Graph.*, 28(5):1–9, 2009.

[37] A. Hormati, Y. Choi, M. Kudlur, R. M. Rabbah, T. Mudge, and S. A. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *PACT*, pages 214–223. IEEE Computer Society, 2009.

[38] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: portable stream programming on graphics engines. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2011.

[39] S. S. Huang, A. Hormati, D. F. Bacon, and R. M. Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In *ECOOP*, pages 76–103, 2008.

[40] Intel Corporation. *Intel IXP 2855 Network Processor*.

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

[42] K. Jang, S. Han, S. Han, S. Moon, and K. Park. Sslshader: cheap ssl acceleration with commodity processors. In *Proceedings of the 8th USENIX conference on Networked systems*

design and implementation, NSDI'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[43] V. J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *HiPEAC 2009*.

[44] P. G. Joisha and P. Banerjee. Static array storage optimization in matlab. In *In ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, 2003.

[45] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Timegraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, Berkeley, CA, USA, 2011. USENIX Association.

[46] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Rec.*, 27(3):42–52, 1998.

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

[48] S.-P. P. Kim, J. D. Simeral, L. R. Hochberg, J. P. Donoghue, and M. J. Black. Neural control of computer cursor velocity by decoding motor cortical spiking activity in humans with tetraplegia. *Journal of neural engineering*, 5(4):455–476, December 2008.

[49] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18, August 2000.

[50] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36:24–35, January 1987.

[51] M. Linetsky. *Programming Microsoft Directshow*. Wordware Publishing Inc., Plano, TX, USA, 2001.

[52] O. Loques, J. Leite, and E. V. Carrera E. P-rio: A modular parallel-programming environment. *IEEE Concurrency*, 6:47–57, January 1998.

[53] H. Massalin and C. Pu. Threads and input/output in the synthesis kernal. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 191–201, New York, NY, USA, 1989. ACM.

[54] M. D. McCool and B. D'Amora. Programming using RapidMind on the Cell BE. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 222, New York, NY, USA, 2006. ACM.

[55] D. Mosberger and L. L. Peterson. Making paths explicit in the scout operating system. pages 153–167, 1996.

[56] P. Newton and J. C. Browne. The code 2.0 graphical parallel programming language. In *Proceedings of the 6th international conference on Supercomputing*, ICS '92, pages 167–177, New York, NY, USA, 1992. ACM.

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

[58] NVIDIA. *CUDA Toolkit 4.0 CUBLAS Library*, 2011.

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

[60] V. S. Pai, P. Druschel, and W. Zwaenepoel. Io-lite: A unified i/o buffering and caching system. 1997.

[61] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. In *Proceedings of the 17th annual international symposium on Computer Architecture (ISCA)*, 1990.

[62] J. Pasquale, E. Anderson, S. Diego, I. K. Muller, T. Global, and I. Solutions. Container shipping: Operating system support for i/o-intensive applications. *IEEE Computer*, 27:84–93, 1994.

[63] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, 2001.

[64] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP 2008*.

[65] M. N. Thadani and Y. A. Khalidi. An efficient zero-copy i/o framework for unix. Technical report, 1995.

[66] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC 2002*.

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

[68] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W.-M. W. Hwu. CUDA-Lite: Reducing GPU Programming Complexity. In *LCPC 2008*.

[69] 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*.