

DOI:10.1145/2656206

**The GPUfs file system layer for GPU software makes core operating system abstractions available to GPU code.**

**BY MARK SILBERSTEIN, BRYAN FORD, AND EMMETT WITCHEL**

# GPUfs: The Case for Operating System Services on GPUs

DUE TO THEIR impressive price/performance and performance/watt curves, GPUs have become the processor of choice for many types of intensively parallel computations, from data mining to molecular dynamics simulations.<sup>14</sup> As GPUs have matured and acquired increasingly general-purpose processing capabilities, a richer and more powerful set of languages, tools, and computational algorithms have evolved to make use of GPU hardware. Unfortunately,

GPU programming models are still almost entirely lacking in core system abstractions (such as files and network sockets) CPU programmers have taken for granted for decades. Today's GPU is capable of amazing computational feats when fed the right data and managed by application code on the host CPU but incapable of initiating basic system interactions for itself (such as reading an input file from a disk). Because core system abstractions are unavailable to GPU code, GPU programmers face many of the same challenges CPU application developers did a half-century ago, particularly the constant reimplementations of system abstractions (such as data movement and management operations).

The time has come to provide GPU programs the useful system services CPU code already enjoys. This goal emerges from a broader trend toward integrating GPUs more cleanly with operating systems, as exemplified by work to support pipeline composition of GPU tasks<sup>16</sup> and improve the operating system's management of GPU resources.<sup>4</sup> Here, we focus on making core operating system abstractions available to GPU code and explain the lessons we learned building GPUfs,<sup>19,20</sup> our prototype file system layer for GPU software.

Two key GPU characteristics make developing operating system abstractions for GPUs a challenge: data parallelism and an independent memory system. GPUs are optimized for single

## » key insights

- **GPUs have matured, acquiring increasingly general-purpose parallel processing capabilities, though still lack core operating system abstractions like files and network sockets.**
- **The traditional CPU-centric programming model exposes low-level system details so is no longer adequate for developing modern accelerated applications.**
- **By allowing developers to access files directly from GPU programs, GPUfs demonstrates the productivity and performance benefits of allowing GPUs to guide the flow of data in a system.**



program multiple data (SPMD) parallelism, where the same program is used to concurrently process many different parts of the input data. GPU programs typically use tens of thousands of lightweight threads running similar or identical code with little control-flow variation. Conventional operating system services (such as the POSIX file system API) were not built with such an execution environment in mind. In developing GPUfs, we had to adapt both the API semantics and its implementation to support such massive parallelism, allowing thousands of threads to efficiently invoke open, close, read,

or write calls simultaneously.

To feed their voracious appetites for data, high-end GPUs usually have their own dedicated DRAM storage. A massively parallel memory interface to GPU memory offers high bandwidth for local access by GPU code, but GPU access to the CPU's system memory is an order of magnitude slower, as it requires communication over a bandwidth-constrained, higher-latency PCI Express bus. In the increasingly common case of systems with multiple discrete GPUs (standard in Apple's Mac Pro) each GPU has its own local memory, and accessing a GPU's own memory

can be an order of magnitude more efficient than accessing a sibling GPU's memory. GPUs thus exhibit a particularly extreme non-uniform memory access (NUMA) property, making it performance-critical for the operating system to optimize for access locality in data placement and reuse across CPU and GPU memories; for example, GPUfs distributes its buffer cache across all CPU and GPU memories to enable idioms like process pipelines that produce and consume files across multiple processors.

To highlight the benefits of bringing file system abstractions to GPUs,

we show the use of GPUfs in a self-contained GPU application for string search on Nvidia GPUs. This application illustrates how GPUfs can efficiently support irregular workloads in which parallel threads open and access dynamically selected files of varying size and composition, and the output size can be arbitrarily large and determined at runtime. Our version is approximately seven times faster than a parallel eight-core CPU run on a full Linux kernel source stored in approximately 33,000 small files.


While our prototype benchmarks represent only a few simple application data points for a single operating system abstraction, they suggest operating system services for GPU code are feasible and efficient in practice.

### GPU Architecture


We provide a simplified overview of the GPU software/hardware model, highlighting properties particularly relevant to GPUfs; an in-depth description can be found elsewhere, as in Kirk and Hwu.<sup>8</sup> We use Nvidia CUDA terminology because we implement GPUfs on Nvidia GPUs, although most other GPUs supporting the cross-platform OpenCL standard<sup>6</sup> share the same concepts.

*Hardware model.* GPUs are parallel processors that expose programmers to hierarchically structured hardware parallelism. At the highest conceptual level, GPUs are similar to CPU shared-memory multicores, comprising several powerful cores called multiprocessors, or MPs. GPUs support coarse-grain task-level parallelism via concurrent execution of different tasks on different MPs.

The main architectural difference between CPUs and GPUs lies in the way a GPU executes parallel code on each MP. A single GPU program, or “kernel” (unrelated to an operating system kernel), consists of tens of thousands of individual threads. Each GPU thread forms a basic sequential unit of execution. Unlike a CPU thread, which usually occupies only one CPU core, hundreds of GPU threads are concurrently scheduled to run on each MP. The hardware scheduler interleaves instructions from alternate threads to maximize hardware utilization when threads are stalled (such as when wait-



**The time has come  
to provide  
GPU programs  
the useful system  
services CPU code  
already enjoys.**



ing on a slow memory access). Such parallelism, sometimes called “thread-level parallelism,” or “fine-grain multithreading,” is essential to achieving high hardware utilization and performance in GPUs.

At the lowest conceptual level, the hardware scheduler manages threads in small groups called “warps” (32 threads in Nvidia GPUs). All threads in a warp are invoked in a single-instruction-multiple-data, or SIMD, fashion, enabling fine-grain data-parallel execution similar to the execution of vector instructions on a CPU.

*Software model.* A GPU program looks like an ordinary sequential program but is executed by all GPU threads. The hardware supplies each thread with a unique identifier allowing different threads to select different data and control paths. Developers are no longer limited to shader programming languages like GLSL and may write GPU programs in plain C++/C or Fortran with a few restrictions and minor language extensions.<sup>a</sup> The programming model closely matches the hierarchy of parallelism in the hardware. Threads in a GPU kernel are subdivided into threadblocks—static groups of up to 1,024 threads—that may communicate, share state, and synchronize efficiently, thereby enabling coordinated data processing within a threadblock. A threadblock is a coarse-grain unit of execution that matches the task-level parallelism support in the hardware; all threads in a single threadblock are scheduled and executed on a single MP.

An application “enqueues” a kernel execution request into a GPU by specifying the desired number of threadblocks in the kernel and the number of threads per threadblock. The number of threadblocks in a kernel can range from tens to hundreds, typically exceeding the number of MPs, thus improving load balancing and portability across GPU systems. When a threadblock is dispatched to an MP, it occupies the resources of that MP until all its threads terminate. Most important, a threadblock cannot be preempted

<sup>a</sup> Many productivity frameworks do not require low-level GPU programming; see examples at <https://developer.nvidia.com/cuda-tools-ecosystem>

in favor of another threadblock waiting for execution in the global hardware queue. The hardware executes different threadblocks in an arbitrary, nondeterministic order. Threadblocks generally may not have data dependencies, as they could lead to deadlock.

*System integration model.* From a software perspective, GPUs are programmed as peripheral devices; they are slave processors that must be managed by a CPU application that uses the GPU to offload specific computations. The CPU application prepares the input data for GPU processing, invokes the kernel on the GPU, and then obtains the results from GPU memory after the kernel terminates. All these operations use GPU-specific APIs offering a rich set of functions covering various aspects of memory and execution-state management.

From a hardware perspective, there are two classes of GPUs: discrete GPUs and hybrid GPUs. Discrete GPUs are connected to the host system via a PCI Express (PCIe) bus and feature their own physical memory on the device itself. Moving the data in and out of GPU memory efficiently requires direct memory access, or DMA.<sup>b</sup> The GPU's bandwidth to local memory is an order of magnitude higher, more than 30x in current systems, than the PCIe bandwidth to the memory on the host. Hybrid GPUs are integrated on the same die with the host CPU and share the same physical memory with CPUs. Here, we focus on discrete GPUs but also discuss our work in the context of hybrid GPUs.

## GPU Programming

Despite their popularity in high-performance and scientific computing, using GPUs to accelerate general-purpose applications in commodity systems is quite limited. The list of 200 popular general-purpose GPU applications published by Nvidia<sup>14</sup> makes no mention of GPU-accelerated real-time virus scanning, text search, or other desktop services, though GPU algorithms for

pattern matching are well known and provide significant speedups.<sup>2,9</sup> Enabling GPUs to access host resources directly via familiar system abstractions will hasten GPU integration in widely deployed software systems.

The current GPU programming model requires application developers build complicated CPU-side code to manage access to the host's network and storage. If an input to a GPU task is stored in a file, the CPU-side code handles all system-level I/O functions, including how much of the file to read into system memory, how to overlap data access with the GPU execution, and how to optimize the size of memory transfer buffers. This code dramatically complicates the design and implementation of GPU-accelerated programs, turning application development into a low-level systems programming task.

Historically, operating systems are instrumental in eliminating or hiding this complexity from ordinary CPU-based application development. GPUfs aims to address these same data-management challenges for GPU-based application code.

**GPU programming difficulties.** Consider an application that searches a set of files for text patterns. It is trivial to speed up this task through multi-core CPUs by, say, scanning different files in parallel on different cores. Algorithmically, the task is also a good candidate for acceleration on GPUs, given the speedups already demonstrated for pattern-matching algorithms on GPUs.<sup>9</sup>

However, using GPUs involves several system-level caveats:

*Complex low-level data-management code.* Since GPU code cannot access files directly, CPU code must assist in reading the file data and managing data transfers to the GPU. A substantial part of an overall GPU program is thus CPU-based code needed to manage data for the GPU. This CPU-based code must deal with low-level details of GPU memory allocation and be optimized for the performance characteristics of a specific GPU model to perform data transfers efficiently.

*No overlap between data transfer and computations.* Unlike CPUs, where operating systems use threads and device interrupts to overlap data

processing and I/O, GPU code traditionally requires all input to be transferred in full to local GPU memory before processing starts. Moreover, the application cannot easily retrieve partial output from GPU memory until the GPU kernel terminates. Optimized GPU software alleviates these performance problems through pipelining; it splits inputs and outputs into smaller chunks and asynchronously invokes the kernel on one chunk while simultaneously transferring the next input chunk to the GPU and the prior output chunk from the GPU. Though effective, pipelining often complicates the algorithm and its implementation significantly.

*Bounded input/output size.* If a file's contents are too large to fit into an input buffer in GPU memory, the application must split the input and process it in smaller chunks, tying the algorithm to low-level hardware details. The size of any output buffer for a GPU program's results must be specified when the program starts, not when it generates its output, further complicating algorithms that produce unpredictable amounts of output. To ensure adequate buffer space, a common practice among developers is to allocate overly large buffers, making inefficient use of GPU memory.

*No support for data reuse.* A CPU application "deallocates" all its GPU-side memory buffers holding file contents when it terminates; for example, a pattern-matching application might read many input files, but, when invoked again, the files are read again from CPU memory or disk. In contrast, CPU applications rely on the operating system's buffer cache to transparently protect them from expensive redundant reads.

*No support for data-dependent accesses.* A program's inputs can depend on its execution history; for example, a program might search for a string in an HTML file, as well as in any file referenced by the HTML file. The names of files that must be searched are known only during execution because they depend on the link structure within the HTML files themselves. A CPU implementation might read the next input file the moment it encounters a reference to that file. However, in GPU code, the file-reading logic occurs on

<sup>b</sup> Discrete GPUs traditionally had a separate address space that could not be referenced directly by CPU programs and required developers to manage nCPU-GPU transfers explicitly; Nvidia's CUDA 6 release (February 2014) introduced runtime support for automatic data movement.



the CPU separate from the GPU-based processing code. The application's CPU and GPU code must therefore coordinate explicitly to determine which files to read next.

**GPUfs makes GPU programming easier.** GPUfs was developed to alleviate these problems. GPUfs is a software layer providing native support for accessing host files on discrete GPUs. GPUfs allows tasks running on GPUs to be largely oblivious as to where data is located, whether on disk, in main memory, in a GPU's local memory, or across several GPUs—a convenience offered for CPU programs by standard file system services. Moreover, GPUfs lets the operating system optimize data access locality across independently developed GPU compute modules using application-transparent caching, much like a traditional operating system's buffer cache optimizes access locality across multi-process compu-

tation pipelines. A familiar file API abstracts away the low-level details of different GPU hardware architectures and their complex inter-device memory consistency models, thus improving code and performance portability. Finally, programmers using GPUfs can build self-sufficient GPU applications without having to develop the complex CPU support code required to feed data to GPU computations.

We view GPUfs as GPU system-level code, but modern GPUs do not support a publicly documented privileged mode. GPUfs cannot therefore run in privileged mode on the GPU, and our GPUfs prototype is a library linked with the application. However, the library is structured in two layers, with the top layer intended to remain a library; the bottom layer would execute in privileged mode when GPUs add such a capability. We expect GPU vendors will eventually provide some combination

of software and hardware support for executive-level software, to, for example, explicitly manage memory permissions across multiple GPU kernels.

The host operating system and the GPU operating-system-level code work together to provide a single file system shared across all processors in the system, though with some semantics that are more like a distributed file system. In contrast, current GPU-like accelerators (such as Intel's Xeon-Phi) run an independent operating system that supports only block-level exclusive access to storage or distributed-systems protocols to communicate with the host operating system. The GPUfs approach results in a system that is easier to use, enables whole-system optimizations like scheduling of data transfers, and provides finer control of shared resources (such as system memory).

Design

Here, we describe the GPUfs API and file-system semantics, focusing on the similarities to and differences from the standard APIs used in CPU programs and the properties of GPUs that motivate these design choices; we omit details of some APIs, though they are included in Silberstein et al.<sup>20</sup>

Figure 1 outlines the architecture of GPUfs. CPU programs are unchanged, but GPU programs can access the host's file system through a GPUfs library linked to the application's GPU code. The GPUfs library works with the host operating system on the CPU to coordinate the file system's namespace and data.

Three properties of discrete GPUs

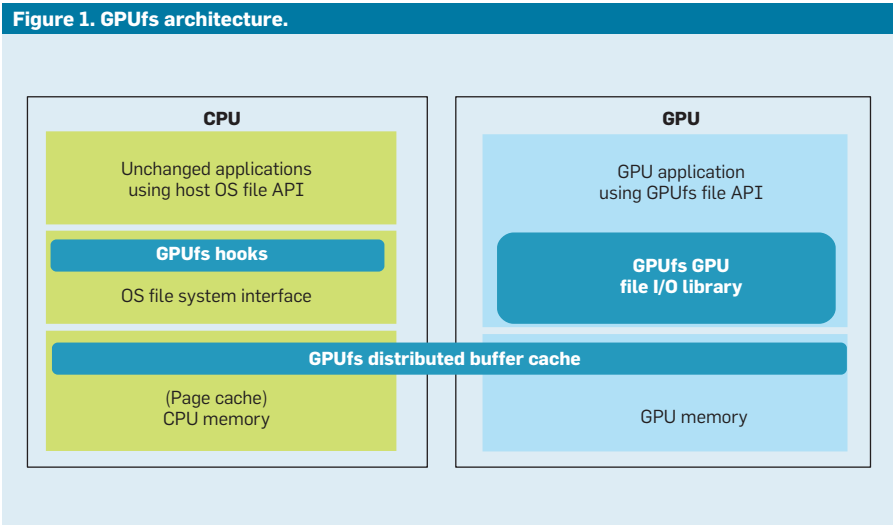


Table 1. Implications of GPU hardware characteristics on GPUfs design.			
	Behavior on CPU	GPU hardware characteristics	GPUfs design implications
Buffer cache	Caches file contents in CPU memory to hide disk access latency	Separate physical memory	Caches file contents in GPU memory to hide accesses to disks and CPU memory
File system data consistency	Strong consistency; file writes are immediately visible to all processes	Slow CPU-GPU communications	Close-to-open consistency; file writes are immediately visible to all GPU local threads but require explicit close and open to be visible on another processor
Cache replacement algorithm	Approximate LRU invoked asynchronously and periodically in a background thread	Non-preemptive hardware scheduling	Synchronous and fast but inaccurate
API call granularity	File APIs are called independently in every thread	Data-parallel lockstep execution of threads in a warp	File APIs are invoked collaboratively by all threads in the same warp
File descriptors	Each descriptor is associated with a file pointer	Massive data parallelism	No file pointers at OS level, but library supports per-warp or per-threadblock local file descriptors

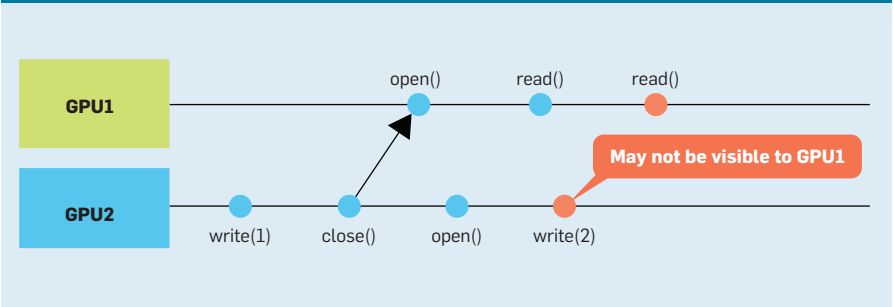
make designing GPUfs a challenge: massive hardware parallelism; a fast and separate physical memory; and non-preemptive hardware scheduling. Table 1 lists their implications for the design of GPUfs, with analysis following in this section.

**Buffer cache for GPUs.** Operating systems strive to minimize slow disk accesses by introducing a buffer cache that stores file contents in memory when file data is first accessed. The operating system serves subsequent accesses directly from the buffer cache, thereby improving performance transparently to applications. Moreover, a buffer cache enables whole-system performance optimizations, most notably read-ahead, data-transfer scheduling, asynchronous writes, and data reuse across process boundaries.

Imagine a GPU program accessing a file. Even if the file data is resident in the CPU buffer cache, it must be transferred from CPU memory to the local GPU memory for every program access. However, GPUs provide far more bandwidth and lower latencies to access local GPU memory than to access the main CPU memory. To achieve high performance in GPU programs it is thus critical to extend GPUfs buffer cache into GPUs by caching file contents in GPU memory. In multi-GPU systems the buffer cache spans multiple GPUs and serves as an abstraction hiding the low-level details of the shared I/O subsystem.

**File system data consistency.** An important design question we faced in developing GPUfs is the choice of a file system data-consistency model (how and when file updates performed by one processor are observed by other processors in a system); for example, if a file is cached by one GPU and then changed by another GPU or a CPU, cached data then becomes stale and must be refreshed by a consistency mechanism. Strong consistency models (such as sequential consistency) permit no or little disparity in the order different processes observe update; for example, in Linux, file writes executed by one process become immediately visible to all other processes running on the same machine. On the other hand, the popular Network File System (NFS) provides no such guarantee if processes are running on differ-

**Figure 2. Close-to-open file system data consistency in GPUfs.**



ent machines. In general, distributed file systems like NFS tend to provide weaker consistency than local file systems, as weaker consistency permits less-frequent data synchronization among caches and is thus more efficient in systems with higher communication costs.

GPUfs is a local file system in the sense it is used by processors in the same physical machine. However, the disparity between the bandwidth from GPU to system memory and to local GPU memory makes the system more like a distributed environment with a slow communication network than a tightly coupled local environment.

GPUfs implements a weak file-system data-consistency model (close-to-open consistency) similar to the Andrew File System<sup>3</sup> and NFS clients (since Linux 2.4.20). When a file's content is cached on a GPU, its threads can read and write the file locally without further communication with other processors, even if the host and/or other GPUs concurrently read and/or modify that file. GPUfs guarantees local file changes propagate to other processors when the file is closed on the modifying processor first and subsequently opened on other processors. In Figure 2, GPU2 writes two different values to a file, “1” and “2.” However, GPU1 will see “1” and may not see “2,” as close-to-open consistency permits postponing updates to other processors operating on the same file instead of propagating them as they happen.

For the GPU threads running on the same GPU, GPUfs provides strong consistency, guaranteeing file updates are visible immediately to all threads in that GPU. To achieve this guarantee, memory writes explicitly bypass the L1 cache via write-through L2 write instructions and are followed by a memory fence. We found the overhead

of doing so sufficiently small to make such a strong consistency model affordable. However, if a file is mapped using `mmap`, GPUfs naturally inherits the memory-consistency model implemented by the hardware.

**Buffer cache management.** CPUs handle buffer-cache-management tasks in daemon threads, keeping costly activities like flushing modified “dirty” pages out of an application’s performance path. Unfortunately, GPUs have a scheduling-related weakness that makes daemon threads inefficient. GPU threadblocks are non-preemptive, so a daemon thread would require its own permanently running threadblock. This dedicated threadblock could be either an independent, constantly running GPU kernel or part of each GPU application. Each approach permanently consumes a portion of GPU hardware resources, thereby degrading the performance of all GPU applications, including those not using GPUfs.

Alternatively, offloading all GPU cache-management functionality to a CPU daemon is impractical on existing hardware due to the lack of atomic operations over a PCIe bus.<sup>c</sup> This limitation precludes use of efficient one-sided communication protocols. A CPU cannot reliably lock and copy a page from GPU memory without GPU code being involved in acknowledging the page has been locked. Consequently, GPUfs uses a less efficient message-passing protocol for synchronization.

Organizing GPUfs without daemon threads has important design consequences, including the need to optimize the page replacement algorithm for speed. GPUfs performs

<sup>c</sup> The PCIe 3.0 standard includes atomics, but implementation is optional and we know of no hardware supporting it.

page replacement as a part of regular read/write file operations, with the GPUfs code hijacking the calling thread to perform the operation. The call is often on the critical path, so reducing the latency of the replacement algorithm is important for maintaining system performance. However, it is unclear how to implement standard replacement mechanisms (such as the clock algorithm<sup>17</sup>), as they require periodic scanning of all pages in use. Performing the scan as part of the file system operations is aperiodic and expensive. The GPUfs prototype instead implements a simple heuristic that evicts a page with the oldest allocation time. While it works well for streaming workloads, the best replacement policy across diverse workloads is an area for future work.

Although GPUfs must invoke the replacement algorithm synchronously, writing modified pages from the GPU memory back to the CPU can be done asynchronously. GPUfs enqueues dirty pages in a ring buffer it shares with the CPU, so the CPU can asynchronously complete the transfer. This single-producer, single-consumer pattern does not require atomic operations.

**GPUfs API.** Not apparent is whether the traditional single-thread CPU API semantics is necessary or even appropriate for massively parallel GPU programs. Consider a program with multiple threads accessing the same file. On a CPU, each thread that opens a file obtains its own file descriptor and accesses the file independently from other threads. The same semantics on a GPU would result in tens of thousands of file descriptors, one for each GPU thread. But such semantics are likely of little use to programmers, as they do not match a GPU's data-parallel programming idioms and hardware-execution model.

The key observation is that GPU and CPU threads have very different properties and are thus used in different ways in programs.

*A single GPU thread is slow.* GPUs are fast when running many threads but drastically slower when running only one; for example, multiplying a vector by a scalar in a single thread is about two orders of magnitude slower on a C2070 TESLA GPU than on a Xeon L5630 CPU. GPUs invoke thousands of

threads to achieve high throughput.

*Threads in a warp execute in lockstep.* Even though GPU threads are independent according to the programming model, the hardware executes threads in SIMD groups, or warps. The threads in the same warp are executed in lockstep. Processing is efficient when all threads in a warp follow the same code paths but highly inefficient if they follow divergent paths; all threads must explore all possible divergent paths together, masking instructions applicable to only some threads at every execution step. Likewise, memory hardware is optimized for a warp-strided access pattern in which all warp threads jointly access a single aligned memory block; the hardware coalesces multiple accesses into a single large memory transaction to maximize memory throughput.

As a result, GPU programs are typically designed to execute a task collaboratively in a group of threads (such as a warp or a threadblock) rather than in each thread separately. In such a group, all threads execute data-parallel code in a coordinated fashion. Data-parallel file API calls might thus be more convenient to use than the traditional per-thread API in CPU programs. Moreover, per-thread file API calls would be highly inefficient; their implementations are control-flow heavy and require synchronization on globally shared data structures (such as a buffer cache) and often involve large memory copies between system and user buffers, as in `write` and `read`. If GPUfs allowed API calls at thread granularity, the threads would thus quickly encounter divergent control and data paths within GPUfs, resulting in hardware serialization and inefficiency in the GPUfs layer.

GPUfs consequently require applications to invoke the file system API at threadblock—rather than thread—granularity. All application threads in a threadblock must invoke the same GPUfs call, with the same arguments, at the same point in application code. These collaborative calls together form one logical GPUfs operation. The threadblock granularity of the API allows the GPUfs implementation to parallelize the handling of API calls across threads in the invoking threadblock like performing table search opera-

tions in parallel. GPUfs also supports a more fine-grain per-warp API granularity of function calls. In it the code relies on lockstep execution of the warp's threads because the hardware does not provide intra-warp synchronization primitives. However, making this assumption is generally discouraged because intra-warp scheduling could change in future GPU architectures. We find per-threadblock calls to be more efficient than per-warp calls and sufficient for the GPUfs applications we have implemented.

*Layered API design.* File descriptors in GPUfs are global to a GPU kernel, just as they are global to a CPU process. Each GPU open returns a distinct file descriptor available to all GPU threads that must be closed with `close`. This design allows a file descriptor to be initialized only once, then reused by all other GPU threads, thereby saving the slow CPU file system accesses. Unfortunately, implementing such globally shared objects on a GPU is nontrivial due to the lack of GPU-wide barriers and subtleties of the GPU memory model.

GPUfs balances programmer convenience with implementation efficiency by layering its API. The open call on the GPU is wrapped into a library function `gopen` that may return the same file descriptor when given the same file name argument. GPUfs reference counts these calls, so a `gopen` on an already open file just increments the file's open count without CPU communication. In our experiments with GPUfs, we have found `gopen` to be more convenient and efficient than the low-level GPU open call.

Likewise, GPUfs removes the file pointer from the global file-descriptor data structure to prevent its update from becoming a serialization bottleneck. GPUfs implements a subset of POSIX file system functionality by providing, for example, the `pread` and `pwrite` system calls that take an explicit file offset parameter.

However, the GPUfs library gives programmers a high-level abstraction of per-threadblock or per-warp file pointers. A programmer can thus choose to program to the low-level `pread` interface or initialize a local file offset and make calls to the more familiar `read` interface. This division of

labor is similar to the division on a CPU between system calls like `read` and C library functions like `fread`.


**File mapping.** GPUfs allows GPU threads to map portions of files directly into local GPU memory via `gmmmap`/`gmmap`. As with traditional `mmap`, file mapping offers two compelling benefits: the convenience to applications of not having to allocate a buffer and separately read data into it and opportunities for the system to improve performance by avoiding unnecessary data copying.

Full-featured memory-mapping functionality requires software-programmable hardware virtual memory unavailable in current GPUs. Even in future GPUs that might offer such control, traditional `mmap` semantics might be difficult to implement efficiently in a data-parallel context. GPU hardware shares control-plane logic, including memory management, across compute units running thousands of threads simultaneously. Any translation change thus has global impact likely requiring synchronization too expensive for fine-grain use within individual threads.


GPUfs offers a more relaxed alternative to `mmap`, permitting more efficient implementation without frequent translation updates. GPUfs provides no guarantee `gmmmap` will map the entire file region the application requests; it could instead map only a prefix of the requested region and return the size of the successfully mapped prefix. Moreover, `gmmmap` is not guaranteed to succeed when the application requests a mapping at a particular address, so `MMAP_FIXED` might not work. Finally, `gmmmap` does not guarantee the mapping will have only the requested permissions; mapping a read-only file could return a pointer to read/write memory, with GPUfs trusting the GPU kernel to not modify that memory.

These looser semantics ultimately enable efficient implementation on the existing hardware by allowing GPUfs to give the application pointers directly into GPU-local buffer cache pages, residing in the same address space (and protection domain) as the application's GPU code.

**Failure semantics.** GPUfs has failure semantics similar to the CPU page cache; on GPU failure, file updates not



**GPUfs is a system software co-resident with GPU programs but is less intrusive than a complete operating system in that it has no active continuously running components on the GPU.**



yet committed to disk could be lost. From the application's perspective, successful completion of `gfsync` or `gmsync` ensures data has been written to the host buffer cache and, if requested, to stable storage. While a successful completion of `gclos` in all GPU threads guarantees another processor reopening the file will observe consistent updates, it does not guarantee the data has been written to disk or even to the CPU page cache, as the transfers might be performed asynchronously.

Unfortunately, in existing systems, a GPU program failure (due to, for example, an invalid memory access or assertion failure) could require restarting the GPU card. As a result, the entire GPU memory state of all resident GPU kernels would be lost, including GPUfs data structures shared across all of them. This lack of fault isolation between GPU kernels makes maintaining long-living operating system data structures in GPU memory impractical. As GPUs continue to be more general purpose, we expect GPU hardware and software to provide better fault isolation and gain more resilience to software failures.

**Resource overheads.** Operating systems are known to compete with user programs for hardware resources like caches and are often blamed for diminished performance in high-performance computing environments. GPUfs is a system software co-resident with GPU programs but is less intrusive than a complete operating system in that it has no active continuously running components on the GPU. GPUfs follows a “pay-as-you-go” design; that is, it imposes no overhead on GPU kernels that use no file system functionality. We deliberately avoided design alternatives involving “daemon threads,” or persistent GPU threads dedicated to file system management tasks like paging and CPU-GPU synchronization. While enabling more efficient implementation of the file system layer, such threads would violate this principle.

However, GPUfs unavoidably adds some overhead in the form of memory consumption, increased program instruction footprint, and use of GPU hardware registers. We expect the relative effect of these overheads on performance to decrease with future hardware generations providing larger



memory, register files, and instruction caches.

**Discussion.** With GPU hardware changing so rapidly, a key design challenge for GPUfs is to focus on properties likely to be essential to the performance of future, as well as current, GPUs. Current technological trends (such as a constantly increasing on-card memory capacity and bandwidth, as well as increasing fine-grain hardware parallelism) suggest the non-uniform memory performance and structured fine-grain parallelism of today's GPUs will persist as GPU architectures evolve. We expect data parallel API semantics and locality optimized caching in GPUfs, being motivated by these hardware characteristics, will thus remain important design features in future GPU systems software.

However, some of our GPUfs design choices were dictated by po-

tentially transient constraints of the current hardware and software, as in the case of a synchronous buffer cache replacement algorithm, gmmmap limitations, and lack of buffer cache memory protection; for example, the replacement algorithm could be improved by offloading it to a CPU outside the critical path, provided the hardware support for atomics across PCI Express. While these aspects of the GPUfs design could potentially become obsolete, they could still influence hardware designers to include the necessary architectural support for efficient operating system abstractions in next-generation accelerators.

### Implementation

Here, we describe our GPUfs prototype for Nvidia Fermi and Kepler GPUs.

Most of GPUfs is a GPU-side library linked with application code. The CPU-

side portion consists of an operating system kernel module for buffer cache consistency management and a user-level thread in the host application, giving GPUfs access to the application's GPU state and host resources.

*GPU buffer cache pages.* GPUfs manages its buffer cache at the granularity of an aligned power-of-two-size memory page that can differ in size from the hardware-supported page. Performance considerations typically dictate page size larger than the standard 4KB operating system-managed pages on the host CPU (such as 256KB).

*Lock-free buffer cache.* The buffer cache uses a radix tree that can be a contention point among threads accessing the same file. GPUfs uses lock-less reads and locked updates, similar to Linux's seqlocks.<sup>21</sup> Specifically, readers access the tree without locks or memory atomics but may need “retry” if the tree is concurrently modified by a writer.

*GPU-CPU RPC mechanism.* GPUfs implements an RPC protocol to coordinate data transfers between CPU and GPU. The GPU serves as a client that issues requests to a file server running on the host CPU. This GPU-as-client design contrasts with the traditional GPU-as-coprocessor programming model, reversing the roles of CPU and GPU. Implementing RPC protocol efficiently is complicated by the CPU/GPU memory consistency model. For arbitrary file contents to be accessible from a running GPU kernel, a GPU and a CPU must be able to access shared memory and provide the means to enforce a global write order (such as via memory fences). Only Nvidia GPUs officially provide this functionality (as of December 2014).<sup>d</sup>

*File consistency management.* The current prototype implements the locality-optimized file-consistency model described earlier. If a GPU is caching the contents of a closed file, this cache must be invalidated if the file is opened for write or unlinked by another GPU or CPU. GPUfs propagates such invalidations lazily; that is, only if and when the GPU caching this stale data later re-

**Figure 3. A simplified GPU string search implementation using GPUfs; the names of the variables shared across all GPU threads are prefixed with “g” and those shared across only a single threadblock (TB) with “tb.”**

```
/* list of words, input and output file names */
gpu_string_search(char* gWords,
                  char* gInputfile,
                  char* gOutfile)
{
    tbFin=gopen(gInputfile,O_RDONLY);
    tbFout=gopen(gOutfile,O_WRONLY);
    tbWords=getTBwords(gWords,TBID);

    /* all threads in a TB read an input chunk *
     * into a TB-shared buffer */
    while ( gread(tbFin, tbInput,
                  tbInputSize, tbReadOffset)>0)
    {
        /* TB-parallel search in a file chunk, then *
         * store each thread's output in tbOutput */
        tb_string_search(tbInput,tbWords, tbOutput);
        /* wait for all TB threads to finish */
        wait_for_all_tb_threads();
        /* run only in one thread of each TB */
        EXECUTE_IN_ONE_THREAD
        {
            /* reserve space in the output file */
            tbWriteOffset=
                atomicAdd(gOutOffset,tbOutputSize);
            /* proceed to the next input chunk */
            tbReadOffset+=tbInputSize;
        }
        /* write output buffer to file by all TB
         * threads together */
        gwrite(tbFout, tbOutput,
              tbWriteSize, tbWriteOffset);
    }
    /* close input/output files in TB*/
    gclose(fin);
    gclose(fout);
}
```

<sup>d</sup> OpenCL 2.0 will enable GPU-CPU RPC via shared virtual memory, though no implementation is publicly available as of December 2014.

opens the file. The GPUfs API currently offers no direct way to push changes made on one GPU to another GPU, except when the latter reopens the file.

**Implementation limitations.** GPUfs implements a private GPU buffer cache for each host CPU process; the buffer cache is not shared across host applications, as it is in the operating system-maintained buffer cache on the host CPU. Unfortunately, GPUs still lack the programmable memory protection necessary to protect a shared GPUfs buffer cache from errant host processes or GPU kernels. We anticipate the necessary protection features will become available in newer GPUs.

Lack of user-controlled memory protection also means GPUfs cannot protect its GPU buffer caches from corruption by the application it serves. However, GPUfs uses the host operating system to enforce file-access protection. The host operating system prevents a GPUfs application from opening host files the application does not have permission to access and denies writes of dirty blocks back to the host file system if the GPUfs application has opened the file read-only.

Our prototype does not yet support directory operations.

## Evaluation

We implemented GPUfs for Nvidia Fermi and Kepler GPUs and evaluated its utility and performance with micro-benchmarks and realistic I/O-intensive applications, finding GPUfs enabled simpler application design and implementation and resulted in high application performance on par with fine-tuned, handwritten versions not using GPUfs. The complete results of the evaluation are in our prior publications,<sup>19,20</sup> and the source code of the benchmarks and the GPUfs library is available for download.<sup>18</sup>

Here, we outline an example of a string-search GPU application to illustrate a typical use of GPUfs. Given a dictionary and a set of text files, the program determines how many times and in which files the word appears. This workload is an example of a broad class of applications performing a full input scan, like n-gram generation or index-free search in databases.<sup>10</sup> One challenge in imple-

menting such applications without GPUfs is unknown, potentially large output size, usually requiring pre-allocation of large amounts of memory to avoid buffer overflow. GPUfs eliminates this problem by writing the output directly to a file.

In our string-search implementation we chose a simple algorithm where each threadblock opens one file at a time; each thread then scans the file for a subset of the dictionary it is allocated to match. This workload puts pressure on GPUfs because most files are small (a few kilobytes on average), leading to frequent calls to `gopen` and `gclose`.

Figure 3 is a high-level sketch of the self-contained GPU program used in this experiment. The program does not require CPU code development. For simplicity, it handles only a single input file and does not include details of the string-matching algorithm. The code is executed in every GPU thread, but the threads of a given threadblock obtain the unique set of search words by calling the `getTBWords` function. Threads in the same threadblock collaboratively fetch a chunk of the input file, scan through it, write the output, and continue until the file is fully scanned. Note if the file fits in the GPUfs buffer cache, only the first threadblock will effectively access the disk, while all others will fetch data directly from the GPU buffer cache without CPU communication.

The code structure is similar to a standard CPU implementation. Developers require no special expertise in low-level system programming; they may thus focus entirely on optimizing the parallel string search algorithm.

This implementation demonstrates one important property shared by all applications in our GPUfs benchmarking suite: the GPUfs implementation requires almost no CPU code development, as functionality resides entirely in the GPU kernel. For all workloads

in the benchmarking suite, the CPU code is identical, save the name of the GPU kernel to invoke. This is a remarkable contrast to standard GPU development, which always requires substantial CPU programming effort. We found that eliminating CPU code made development easier.

We ran two experiments with this code counting the frequency of modern English words in two datasets: the works of William Shakespeare and the Linux kernel source code. We invoked the string search benchmark with a dictionary of 58,000 modern English words<sup>e</sup> within the complete works of Shakespeare as a single 6MB text file<sup>f</sup> and within the Linux 3.3.1 kernel source containing approximately 33,000 files holding 524MB of data. The list of input files was itself specified in a file.

As a point of reference we compare two other implementations: a simple CPU program performing the same task on eight cores using OpenMP and a “vanilla” GPU version implemented without GPUfs. Both implementations prefetch the contents of the input files into a large memory buffer first and do not read from the file system during the matching phase. The vanilla GPU version supports only inputs and outputs that fit in the GPU’s physical memory. In contrast, the GPUfs implementation has no such limitations.

We performed the evaluation on a SuperMicro server with two four-core Intel Xeon L5630 CPUs and Nvidia C2075 GPU (see Table 2). Even for such a file-system-intensive workload, a single GPU outperforms the eight-core CPU by 6.8x. The GPUfs version performance was similar to the vanilla GPU implementation on one large input file, with the GPUfs version only 9% slower than the vanilla one on the Linux kernel input. However, GPUfs-

<sup>e</sup> <http://www.mieliestronk.com/wordlist.html>

<sup>f</sup> <http://www.gutenberg.org/ebooks/100>

**Table 2. GPU string-search performance.**

Input	CPUx8	GPU-GPUfs	GPU-vanilla
Linux source	6.1h	53m (6.8×)	50m (7.2×)
Shakespeare	292s	40s (7.3×)	40s (7.3×)

based code was approximately half the length of the vanilla version.


### Discussion

We have advocated for providing standard operating system services and abstractions on GPUs to facilitate their harmonious integration with the rest of the computer system. Such integration is the key to broader adoption of GPUs now and in the future. Our work designing GPUfs marks yet another turn of the “wheel of reincarnation,”<sup>11</sup> demonstrating inevitable evolution of GPUs toward providing more general-purpose functionality. We implemented a file system for discrete GPUs to demonstrate the feasibility and value of this goal on real GPU hardware.


This work also has limitations. We have focused primarily on a file-system layer that might not reflect the challenges and trade-offs in other operating system services on GPUs. The main design principles underpinning the GPUfs file system layer design are general, as they address the core characteristics of GPU hardware: massive parallelism, slow sequential execution, and NUMA memory organization. Indeed, we found them equally applicable to the design of a networking layer for GPUs, as described by Kim et al.<sup>7</sup>

GPUfs makes standard system abstractions and interfaces commonly used in CPU systems available on GPUs, but such abstractions might not be the most suitable for GPU programs. Prior work cast the GPU as a coprocessor, with the CPU managing the interaction between the GPU and the rest of the system. As a result, the bulk of GPU research has focused on devising new CPU-side abstractions to make this interaction easier to manage; for example, flow graph execution frameworks like PTask<sup>16</sup> facilitate development of complex applications but treat GPU tasks as opaque monolithic execution units. In contrast, we propose to give a GPU program the flexibility of managing its own input and output data. In our experience, using the familiar CPU file system abstractions and API adapted to suit GPU programming idioms is a natural and convenient way to provide I/O functionality to GPU programs.

This work targets discrete GPUs and does not directly address emerg-



**Our work designing GPUfs marks yet another turn of the “wheel of reincarnation,” demonstrating inevitable evolution of GPUs toward providing more general-purpose functionality.**



ing hybrid processor architectures like AMD Kaveri and mobile system-on-a-chip that combine CPU and GPU on the same die. The most recent processors add support for shared physical memory and virtual address space between CPU and GPU. Communicating through shared memory makes CPU-GPU data transfers unnecessary, providing a much faster and easier way to exchange data between processors. Consequently, tight CPU-GPU coupling might seem to obviate the need for the separate system abstraction layer on GPUs we have advocated here, making sufficient the GPU-as-coprocessor model, with its reliance on the CPU operating system.

Shared-memory hardware is only a low-level interface that optimizes CPU-GPU interaction but does not eliminate the software complexities of implementing interactions between the GPU and the rest of the system. As with discrete GPUs, the inability to directly access host resources from GPU code complicates natural program design, necessitating CPU management code to prepare data for GPU processing. While calling CPU functions from GPU code becomes possible with integrated GPUs,<sup>1</sup> using this mechanism to implement file or network I/O from thousands of GPU threads would overwhelm the CPU operating system and thus warrants explicit handling of massive parallelism on both the GPU and the CPU, as in GPUfs. Moreover, CPU-GPU coordination requires intimate knowledge of the memory-consistency semantics of CPU-GPU shared memory, which is not only complicated but also constantly evolving, as evident from successive revisions of the OpenCL standard. In contrast, familiar system abstractions and portable, platform-optimized APIs hide these and other low-level details and naturally facilitate code development on hybrid architectures, as they do in CPUs and discrete GPUs.

*Discrete GPUs remain relevant.* We expect discrete and integrated GPUs will continue to coexist for years to come. They embody different trade-offs among power consumption, production costs, and system performance, and thus serve different application domains. Discrete GPUs




have consistently shown performance and power-efficiency growth over the past few hardware generations. This growth is facilitated by discrete GPUs residing on a standalone, add-on peripheral device, giving designers much greater hardware design flexibility than integrated systems. The aggressively throughput-optimized hardware designs of discrete GPUs rely heavily on a fully dedicated, multibillion-transistor budget, tight integration with specialized high-throughput memory, and increased thermal design power. As a result, discrete GPUs offer the highest compute performance and compute performance per watt, making them the computational accelerator of choice in data centers and supercomputers. In contrast, hybrid GPUs are allocated only a small fraction of the silicon and power resources available to discrete processors and thus offer an order-of-magnitude-lower computing capacity and memory bandwidth.

Discrete architectures have been so successful that manufacturers continue to migrate functions to the GPU that previously required CPU-side code; for example, Nvidia GPUs support nested parallelism in hardware, allowing invocation of new GPU kernels from GPU code without first stopping the running kernel. Similarly, modern GPUs provide direct access to peripheral devices (such as storage and network adapters), eliminating the CPU from the hardware data path. Future high-throughput processors<sup>5</sup> are expected to enable more efficient sequential processing.

Indications of this trend are already apparent; for example, the AMD Graphics Core Next 1.1 used in all modern AMD GPUs contains a scalar processing unit. In addition, Nvidia and IBM announced (November 2013) a partnership that aims to integrate Nvidia GPUs and IBM Power CPUs targeting data-center environments. These trends reinforce the need for high-level services on GPUs themselves. Besides making GPUs easier to program, these services will naturally exploit emerging hardware capabilities and avoid performance and power penalties of switching between the CPU and the GPU to perform I/O calls.

Intel's Xeon-Phi represents an extreme example of GPUs gaining more CPU-like capabilities. Xeon-Phi shares many conceptual similarities with discrete GPUs (such as slow sequential performance and fast local memory). However, it uses more traditional CPU cores and runs a full Linux operating system, providing a familiar execution environment for the programs it executes. Xeon-Phi's software architecture supports standard operating system services. However, the current Xeon-Phi system does not allow efficient access to host files and network, and programmers are encouraged to follow a more traditional coprocessor programming model, as in GPUs. The recently announced next processor generation, Knight's Landing, is expected to serve as the main system CPU, eliminating the host-accelerator separation. The new processor memory subsystem will include high-bandwidth, size-limited 3D stacked memory. We expect this stacked memory will have exaggerated NUMA properties, though the ideal system stack design on such memory remains to be seen. Meanwhile, many aspects of GPU system abstractions described here (such as NUMA-aware file cache locality optimizations) will be relevant to the coming and future generations of these processors.

**GPU productivity efforts.** Recent developments in GPU software make it much easier for programmers to accelerate computations on GPUs without writing any GPU code. Commercially available comprehensive STL-like libraries of GPU-accelerated algorithms,<sup>13</sup> efficient domain-specific APIs,<sup>11</sup> and offloading compilers<sup>12</sup> parallelize and execute specially annotated loops on GPUs.

These and other GPU productivity projects use the GPU as a coprocessor and passive consumer of data. Applications that must orchestrate data movement are cumbersome for programmers to implement because GPU code cannot perform I/O calls directly. Systemwide support for operating system services, as demonstrated by GPUfs, alleviates this basic constraint of the programming model and could benefit many GPU applications, including those developed with the help of other GPU productivity tools. 

## References

1. Bratt, T. *HSA Queuing*. Hot Chips 2013 tutorial, 2013; <http://www.slideshare.net/hsafoundation/hsaqueuing-hot-chips-2013>
2. Han, S., Jang, K., Park, K., and Moon, S. PacketShader: A GPU-accelerated software router. *SIIGCOMM Computer Communication Review* 40, 4 (Aug. 2010), 195–206.
3. Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., and West, M.J. Scale and performance in a distributed file system. *ACM Transactions on Computing Systems* 6, 1 (Feb. 1988), 51–81.
4. Kato, S., McThrow, M., Maltzahn, C., and Brandt, S. Gdev: First-class GPU resource management in the operating system. In *Proceedings of the USENIX Annual Technical Conference* (Boston, June 13–15). USENIX Association, Berkeley, CA, 2012.
5. Keckler, S.W., Dally, W.J., Khailany, B., Garland, M., and Glasco, D. GPUs and the future of parallel computing. *IEEE Micro* 31, 5 (Sept.–Oct. 2011), 7–17.
6. Khronos Group. *The OpenCL Specification*, 2013; <https://www.khronos.org/opencl/>
7. Kim, S., Huh, S., Hu, Y., Zhang, X., Wated, A., Witchel, E., and Silberstein, M. GPUnet: Networking abstractions for GPU programs. In *Proceedings of the International Conference on Operating Systems Design and Implementation* (Broomfield, CO, Oct. 6–8). USENIX Association, Berkeley, CA, 2014.
8. Kirk, D.B. and Hwu, W.-m. *Programming Massively Parallel Processors: A Hands-On Approach*. Morgan Kaufmann, San Francisco, 2010.
9. Lehari, D. and Schein, S. Fast RegEx parsing on GPUs. Presentation at NVIDIA Global Technical Conference (San Jose, CA, 2012); <http://on-demand.gputechconf.com/gtc/2012/presentations/S0043-GTC2012-30x-Faster-GPU.pdf>
10. Mostak, T. *An Overview of MapD (Massively Parallel Database)*. Technical Report. Map-D, 2013; <http://www.map-d.com/docs/mapd-whitepaper.pdf>
11. Myer, T.H. and Sutherland, I.E. On the design of display processors. *Commun. ACM* 11, 6 (June 1968), 410–414.
12. Nvidia. *GPU-Accelerated High-Performance Libraries*; <https://developer.nvidia.com/gpuaccelerated-libraries>
13. Nvidia. *Nvidia Thrust library*; <https://developer.nvidia.com/thrust>
14. Nvidia. *Popular GPU-Accelerated Applications*; <http://www.nvidia.com/object/gpu-applications.html>
15. The Portland Group. *PGI Accelerator Compilers with OpenACC Directives*; <http://www.pgroup.com/resources/accel.htm>
16. Rossbach, C.J., Currey, J., Silberstein, M., Ray, B., and Witchel, E. PTask: Operating system abstractions to manage GPUs as compute devices. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal, Oct. 23–26). ACM Press, New York, 2011, 233–248.
17. Silberschatz, A., Galvin, P.B., and Gagne, G. *Operating Systems Principles*. John Wiley & Sons, Inc., New York, 2008.
18. Silberstein, M. GPUfs home page; <https://sites.google.com/site/silbersteinmark/Home/gpufs>
19. Silberstein, M., Ford, B., Keidar, I., and Witchel, E. GPUfs: Integrating file systems with GPUs. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Mar. 16–20). ACM Press, New York, 2013, 485–498.
20. Silberstein, M., Ford, B., Keidar, I., and Witchel, E. GPUfs: Integrating file systems with GPUs. *ACM Transactions on Computer Systems* 32, 1 (Feb. 2014), 1:1–1:31.
21. Wikipedia. Seqlock; <http://en.wikipedia.org/wiki/Seqlock>

**Mark Silberstein** (mark@ee.technion.ac.il) is an assistant professor in the Department of Electrical Engineering at The Technion – Israel Institute of Technology, Haifa, Israel.

**Bryan Ford** (bryan.ford@yale.edu) is an associate professor in the Department of Computer Science at Yale University, New Haven, CT.

**Emmett Witchel** (witchel@cs.utexas.edu) is an associate professor in the Department of Computer Science at the University of Texas at Austin.