GPUfs: The Case for Operating System Services on GPUs

Mark Silberstein UT Austin and Technion
Bryan Ford Yale University
Emmett Witchel UT Austin

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 [14]. 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 *core system abstractions*, such as files and sockets, that CPU programmers have taken for granted for decades. Today's GPU is capable of amazing computational feats when fed with the right data and managed by application code on the host CPU, but it is 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 today face many of the same challenges CPU application developers did a half-century ago—particularly the constant reimplementation of system abstractions such as data movement and management operations.

We feel the time has come to provide GPU programs with the useful system services that CPU code already enjoys. This goal emerges from a broader trend to integrate GPUs more cleanly with operating systems (OS), as exemplified by recent work to support pipeline composition of GPU tasks [15] or improve the OS's management of GPU resources [4]. This article focuses on making core OS abstractions available to GPU code, and on the challenges, design, and lessons we learned from building *GPUfs* [18, 19], our prototype file system layer for GPU software.

Two key GPU characteristics make developing OS abstractions for GPUs challenging: data parallelism, and independent memory system. GPUs are optimized for Single 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 OS services, such as the POSIX file system API, were not built with such an execution environment in mind. In 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 DRAM storage. A massively parallel memory interface to this DRAM offers high bandwidth for local access by GPU code, but GPU access to the CPU's system memory is an order of magnitude slower, because it requires communication over bandwidth-constrained, higher latency PCI Express bus. In the increasingly common case of systems with multiple discrete GPUs—standard in Apple's new Mac Pro, for example—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 OS to optimize for access locality in data placement and reuse across CPU and GPU memories. GPUfs, for example, distributes its buffer cache across all CPU and GPU memories to enable idioms like process pipelines that read and write files from the same or different processors.

To highlight the benefits of bringing core OS abstractions such as files 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 might be arbitrarily large and determined at runtime. Our version is about seven times faster than a parallel 8-core CPU run on a full Linux kernel source stored in about 33,000 small files.

While currently our prototype benchmarks represent only a few simple application data points for a single OS abstraction, they suggest that OS services for GPU code are not only hypothetically desirable, but are feasible and efficient in practice.

© ACM, 2014. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Communications of the ACM, Volume 57 Issue 12, Pages 68-79, December 2014.
http://doi.acm.org/10.1145/2656206

1. GPU ARCHITECTURE OVERVIEW

We provide a simplified overview of the GPU software/hardware model, highlighting properties that are particularly relevant to GPUfs. An in-depth description can be found elsewhere (e.g. [7]). We use NVIDIA CUDA terminology because we implement GPUfs on NVIDIA GPUs, but most other GPUs that support the cross-platform OpenCL standard [6] share the same concepts.

Hardware model.

GPUs are parallel processors which expose programmers to hierarchically structured hardware parallelism. At the highest level GPUs are similar to CPU shared-memory multicores – they comprise several powerful cores called *multiprocessors* (*MPs*). GPUs, therefore, 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 GPUs execute parallel code on each MP. A single GPU program, termed a *kernel* (unrelated to an OS kernel) comprises tens of thousands of individual *threads*. Each GPU thread forms a basic sequential unit of execution. Unlike CPU threads which usually exclusively occupy 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 get stalled, for example while waiting on a slow memory access. This type of parallelism, sometimes called thread-level parallelism, or fine-grained multithreading, is essential to achieving high hardware utilization and performance in GPUs.

At the lowest 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 SIMD (Single Instruction Multiple Data) fashion enabling fine-grained 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 it 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 only few restrictions and minor language extensions [1]. 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 a thousand threads which may communicate, share state and synchronize efficiently, 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 at once on a single MP.

An application enqueues a kernel execution request into a GPU by specifying the desired number of threadblocks comprising the kernel and the number of threads per threadblock. The number of threadblocks in a kernel may range from tens to hundreds, and typically exceeds the number of MPs improving load balancing and portability across GPU systems. Once a threadblock has been dispatched to an MP, it occupies the resources of that MP until all of the threadblock's threads terminate. Most importantly, a threadblock cannot be *preempted* in favor of another threadblock waiting for execution in the global hardware queue. The hardware executes different threadblocks in an arbitrary, non-deterministic order. Therefore, threadblocks generally may not have data dependencies, because such dependencies could lead to deadlock.

System integration model.

From the software perspective, GPUs are programmed as peripheral devices: they are slave processors that must be managed by a CPU application which 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, which offer a rich set of functions covering various aspects of memory and execution state management.

From the 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 (DMA)[2]. The GPU's bandwidth to local memory is an order of magnitude higher – over $30\times$ 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.

In this paper we focus on discrete GPUs, but discuss our work in the context of hybrid GPUs in section §6.

[1]There are many productivity frameworks that do not require any low-level GPU programming. Some examples are available at `https://developer.nvidia.com/cuda-tools-ecosystem`.

[2]Traditionally, discrete GPUs had a separate address space that could not be directly referenced by CPU programs and required developers to manage CPU-GPU transfers explicitly. The recent NVIDIA CUDA 6 release introduces runtime support for automatic data movement.

2. GPU PROGRAMMING: BARRIERS AND OPPORTUNITIES

Despite their popularity in high-performance and scientific computing, the use of GPUs for accelerating general-purpose applications in commodity systems is quite limited. The list of 200 popular general-purpose GPU applications recently published by NVIDIA [14] has no mention of GPU-accelerated desktop services, such as real-time virus scanning, text search, or data encryption, although GPU algorithms for encryption and pattern matching are well-known and provide significant speedups [8, 2]. We believe that enabling GPUs to access host resources directly, via familiar system abstractions such as files, will hasten GPU integration in widely deployed software systems.

Current GPU programming model requires application developers to 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, for example, the CPU-side code handles system-level I/O issues, such as 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.

Operating systems have historically been 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.

2.1 Difficulties of programming GPUs

Consider an application that searches a set of files for text patterns. It is trivial to speed up this task using multi-core CPUs, for example by scanning different files in parallel on different cores. Algorithmically, this task is also a good candidate for acceleration on GPUs, given the speedups already demonstrated for pattern matching algorithms on GPUs [8].

Using GPUs present several *system-level* caveats, however.

Complex low-level data management code. Since GPU code cannot directly access files, CPU code must assist in reading the file data and managing data transfers to the GPU. Thus, a substantial part of an overall GPU *program* is actually *CPU-based* code needed to manage data for the GPU. This CPU-based code needs to understand low-level GPU details and performance characteristics to allocate GPU memory and manage data transfers efficiently.

No overlap between data transfer and computations. Unlike in 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. Further, the application cannot easily retrieve partial output from GPU memory until the GPU kernel terminates. Optimized GPU software alleviates these performance problems via 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. While 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 prevent running out of buffer space, a common practice is to allocate overly large buffers, making inefficient use of GPU memory.

No support for data reuse. A CPU application deallocates all of its GPU-side memory buffers that hold file contents when it terminates. For example, the pattern matching application might read (and not modify) many input files, but when it is 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 and in any file referenced by the HTML file. The list of files that must be searched is only known during execution because it depends 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 it. In GPU code, however, the file reading logic occurs on the CPU separately from the GPU-based processing code. The application's CPU and GPU code must therefore coordinate explicitly on which files to read next.

2.2 GPUfs makes GPU programming easier

GPUfs is intended to alleviate these problems. GPUfs is a software layer providing native support for accessing host files on discrete GPUs. It offers tasks running on GPUs a convenience well-established in the CPU context: to be largely oblivious to where data is located – whether on disk, in main memory, in a GPU's local memory, or across several GPUs. Further, 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 computation pipelines. A familiar file API abstracts away the low-level details of different GPU hardware architectures and their complex inter-device memory consistency models, improving code and performance portability. Finally, users of GPUfs can build self-sufficient GPU applications without developing the complex CPU support code required to feed data to GPU computations. We further illustrate these benefits in the code example in Section 5.

We think of GPUfs as GPU system-level code, but modern GPUs do not support a publicly documented privileged mode. Therefore, GPUfs cannot 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 believe that GPU vendors will eventually provide some combination of software and hardware support for executive-level software, e.g., to explicitly manage memory permissions across multiple GPU kernels.

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

3. DESIGN

We describe the GPUfs API and file system semantics, focusing on the similarities and differences from the standard APIs used in CPU programs, and the properties of GPUs that motivate these design choices. For lack of space we omit the detailed description of some APIs and refer the interested reader to our full paper [19].

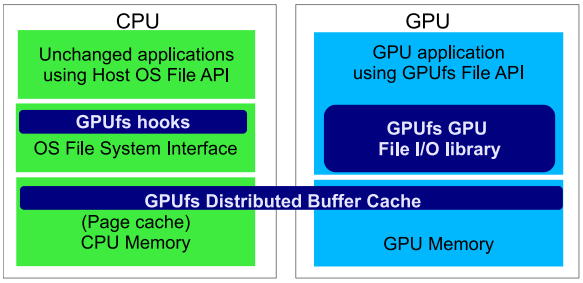

Figure 1: GPUfs architecture

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

There are three properties of discrete GPUs that make designing GPUfs challenging: massive hardware parallelism, a fast and separate physical memory, and non-preemptive hardware scheduling. We first summarize their implications on the design of GPUfs in Table 1, with the detailed analysis in the rest of this section.

3.1 Buffer cache for GPUs

Operating systems strive to minimize slow disk accesses by introducing a *buffer cache*, which stores file contents in memory when file data is first accessed. The OS serves subsequent accesses directly from the buffer cache, thereby improving performance transparently to applications. Moreover, buffer cache enables whole-system performance optimizations such as 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. For GPUfs performance, it is therefore critical to extend the buffer cache into GPUs by caching file contents in GPU memory. In multi-processor, 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.

One important design decision is the choice of a file system data *consistency model*, which determines 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, then cached data becomes stale, and must be refreshed by a consistency mechanism. Strong consistency models (e.g., sequential consistency) permit no or little disparity in the order different processes can observe updates. 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 NFS distributed file system provides no such guarantee if processes are running on different machines. In general, distributed file systems tend to provide weaker consistency than local file systems, because 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 that it is used by processors in the same physical machine. However, the disparity between

	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

Table 1: Implications of the GPU hardware characteristics on the GPUfs design.

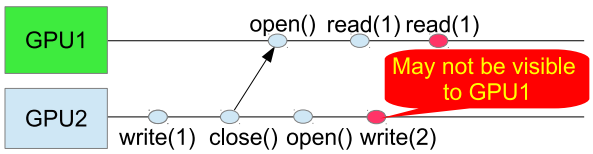

Figure 2: Close-to-open file system data consistency in GPUfs.

the bandwidth from the GPU to system memory and to local GPU memory, makes the system more similar to a distributed environment with a slow communication network rather than a tightly coupled local environment.

GPUfs therefore implements a weak file system data consistency model (close-to-open consistency), similar to the Andrew file system (AFS [3]) and NFS clients (since Linux 2.4.20). Once 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 that local file changes propagate to other processors when the file is closed on the modifying processor first, and subsequently opened on other processors. In the example in Figure 2, GPU2 writes two different values to a file. However GPU1 will see "1" and may not see "2", because 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, which guarantees that file updates are immediately visible to all the threads in that GPU. To achieve this guarantee, writes explicitly bypass the L1 cache via write-through L2 write instructions, and are followed by memory fences. We found the overhead of doing so sufficiently small to make such a strong consistency model affordable. If a file is mapped using `mmap`, however, 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 such as flushing modified (dirty) pages out of an application's performance path. GPUs unfortunately have a scheduling-related weakness that makes daemon threads inefficient. GPU threadblocks are non-preemptive, so a daemon would require its own permanently running threadblock. This dedicated threadblock could be either an independent, constantly running GPU kernel, or it could be part of each GPU application. Both approaches permanently consume a portion of GPU hardware resources, thereby reducing the performance of all GPU applications including those not using GPUfs.

Alternatively, offloading all of the GPU cache management functionality to a CPU daemon is impractical on existing hardware due to the lack of atomic operations over a PCIe bus [3]. This limitation precludes the use of efficient one-sided communication protocols. A CPU cannot reliably lock and copy a page from GPU memory, for example, without GPU code being involved in acknowledging that the page has been locked. Consequently, our design uses a less efficient message-passing protocol for synchronization.

Organizing GPUfs without daemon threads has important design consequences, such as the need to optimize the page replacement algorithm for speed. GPUfs performs page replacement as a part of regular file operations such as `write`, 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. It is unclear, however, how to implement standard replacement mechanisms, such as the clock algorithm [16], because they require periodic scanning of all pages in use. Performing the scan as a part of the file system operations is aperiodic and expensive. Instead, the GPUfs prototype 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 which it shares with the CPU so the CPU can asynchronously complete the transfer. This single producer, single consumer pattern does not require atomic operations.

3.2 GPUfs API

It is not clear 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 of 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 to be of little use to programmers, because they do not match GPU's data-parallel programming idioms and hardware execution model.

Our key observation is that GPU and CPU threads have very dif-

[3]The PCIe 3.0 standard includes atomics, but implementation is optional and we know of no hardware currently supporting it.

ferent properties, and thus are 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 C2070 TESLA GPU than on Xeon L5630 CPU. Hence, 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 (see more detail in § 1). The threads in the same warp are executed in lockstep. Thus, processing is efficient when all threads in a warp follow the same code paths, but highly inefficient if they follow divergent paths: *all* the threads must explore *all* possible divergent paths together, masking instructions applicable only to some threads at every execution step. Similarly, memory hardware is optimized for a warp-strided access pattern in which all the 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 the threads execute data-parallel code in a coordinated fashion. Hence, *data-parallel* file API calls might be more convenient to use than the traditional per-thread API used in CPU programs. Furthermore, per-thread file API calls would be highly inefficient: their implementations are control-flow heavy, they require synchronization on globally shared data structures, e.g. a buffer cache, and they often involve large memory copies between system and user buffers, as in `write` and `read`. Therefore, if GPUfs allowed API calls at thread granularity, the threads would quickly encounter divergent control and data paths within GPUfs, resulting in hardware serialization and inefficiency in the GPUfs layer.

Consequently, GPUfs requires 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 comprise 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—parallelizing file table search operations, for example. We also support a more fine-grained per-warp API granularity of function calls. However the code relies on lockstep execution of the warp's threads, because the hardware does not provide intra-warp synchronization primitives. Making this assumption, however, is generally discouraged because intra-warp scheduling may change in future GPU architectures. In fact, we found per-threadblock calls to be more efficient than per-warp calls, and sufficient for the GPUfs applications we 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 to initialize a file descriptor only once, and then reuse it by all other GPU threads to save the slow CPU file system accesses. Unfortunately, implementing such globally-shared objects on a GPU is a non-trivial task 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 found `gopen` to be more convenient and efficient to use than the low-level GPU `open` call.

Similarly, at the lowest level, GPUfs removes the file pointer from the global file descriptor data structure to prevent its update from becoming a serializing bottleneck. It implements a subset of POSIX file system functionality, for example by providing the `pread` and `pwrite` system calls, which take an explicit file offset parameter. At a higher level, however, GPUfs provides programmer convenience, such as per-threadblock or per-warp file pointers. Thus a programmer can choose to program to the low-level `pread` interface, or she can initialize a local file offset and make calls to the more familiar `read` interface. This division of labor is somewhat similar to the division on the 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 `gmmap`/`gmunmap`. As with traditional `mmap`, file mapping offers two 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, which current GPUs lack. Even in future GPUs that may offer such control, traditional `mmap` semantics might be difficult to implement efficiently in data parallel contexts. GPU hardware shares control plane logic, including memory management, across compute units running thousands of threads at once. Thus, any translation change has global impact, likely requiring synchronization too expensive for fine-grained use within individual threads.

GPUfs therefore offers a more relaxed alternative to `mmap`, permitting more efficient implementation without frequent translation updates. There is no guarantee that `gmmap` will map the entire file region the application requests—instead it may map only a prefix of the requested region, and return the size of the successfully mapped prefix. Further, `gmmap` is not guaranteed ever to succeed when the application requests a mapping at a particular address: i.e., `MMAP_FIXED` may not work. Finally, `gmmap` does not guarantee that the mapping will have *only* the requested permissions: mapping a read-only file may return a pointer to read/write memory, and GPUfs trusts the GPU kernel not to 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.

3.3 Failure semantics

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

In existing systems, unfortunately, a GPU program failure—such as an invalid memory access or assertion failure—may require restarting the GPU card. As a result, the entire GPU memory state of all

resident GPU kernels is lost, including GPUfs data structures shared across all of them. This lack of fault isolation between GPU kernels makes maintaining long-living OS data structures in GPU memory impractical. As GPUs continue to become more general-purpose, we expect GPU hardware and software to provide better fault isolation and gain more resilience to software failures.

3.4 Resource overheads

Operating systems are known to compete with user programs for hardware resources such as cachesand are often blamed for decreased performance in high-performance computing environments. GPUfs is system software co-resident with GPU programs, but it is less intrusive than a complete OS in that it has no active, continuously running components on the GPU. GPUfs by design imposes no overhead on GPU kernels that use no file system functionality. We deliberately avoided design alternatives involving "daemon" threads: i.e., persistent GPU threads dedicated to file system management, such as paging or CPU-GPU synchronization. While enabling more efficient implementation of the file system layer, such threads would violate this "pay-as-you-go" design principle.

GPUfs necessarily adds some overhead, however, 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, which will provide larger memory, larger register files, and larger instruction caches.

3.5 Discussion

With GPU hardware changing so rapidly, a key design challenge for GPUfs has been 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-grained hardware parallelism indicate that the non-uniform memory performance and structured fine-grained parallelism of today's GPUs will persist as GPU architectures evolve. Therefore, we believe that data parallel API semantics and locality optimized caching in GPUfs, being motivated by these hardware characteristics, will remain important design traits in future GPU systems software.

Some GPUfs design choices, however, are dictated by potentially transient constraints of the current hardware and software. This is the case, for example, for a synchronous buffer cache replacement algorithm, `gmmap` limitations and the lack of buffer cache memory protection. The replacement algorithm, for example, could be improved by offloading it to a CPU outside of the critical path provided the hardware support for atomics across PCI Express. While these aspects of our design may potentially become obsolete in the future, they may influence hardware designers to include the necessary architectural support for efficient OS abstractions in next-generation accelerators.

4. IMPLEMENTATION

This section briefly describes 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 a user-level thread in the host application, giving it access to the application's GPU state and host resources, and an OS kernel module for buffer cache consistency management.

GPU buffer cache pages. GPUfs manages its buffer cache at the granularity of an aligned power-of-two sized memory page, which can differ in size from the hardware-supported page. Performance considerations typically dictate page sizes larger than the standard 4KB OS-managed pages on the host CPU, e.g., 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 lockless reads and locked updates, similar to Linux's seqlocks [20]. Specifically, readers access the tree without locks or memory atomics, but they may need retry if the tree gets concurrently modified by a writer.

GPU-CPU RPC mechanism. GPUfs implements an RPC protocol to coordinate data transfers between the 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. One of the main challenges of implementing an efficient RPC protocol lies in 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 (e.g., via fences). Currently only NVIDIA GPUs provide this functionality [4]

File consistency management. The current prototype implements the locality-optimized file consistency model described in Section 3. 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, if and when the GPU caching this stale data later reopens 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 currently 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 OS-maintained buffer cache on the host CPU. Unfortunately, GPUs still lack the programmable memory protection that is necessary to protect a shared GPUfs buffer cache from errant host processes or GPU kernels. We anticipate that the necessary protection features will become available in newer GPUs.

The lack of interface to memory protection also means that GPUfs cannot protect its GPU buffer caches from corruption by the application it serves. GPUfs uses host OS to enforce file access protection, however. The host OS prevents a GPUfs application from opening host files the application does not have permission to access, and it 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.

5. EVALUATION

We implemented GPUfs for NVIDIA GPUs and evaluated its utility and performance with micro-benchmarks and realistic I/O intensive applications. GPUfs enabled simpler application design and implementation, and resulted in high application performance on par with fine-tuned, hand-written versions not using GPUfs. The complete results of the evaluation are presented in our prior publications [18, 19]. The source code of the benchmarks and the GPUfs library are available for download [17].

Here we show only an example of a string search GPU application as an illustration of a typical use of GPUfs. Given a dictionary and a set of text files, for each word in the dictionary, 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 an index-free search in databases [9]. One of the challenges in implementing these applica-

[4]OpenCL 2.0 will enable GPU-CPU RPC via shared virtual memory, however no implementation has been made available publicly yet.

```
/* 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);
}
```

Figure 3: A sketch of a GPU string search implementation using GPUfs. The names of the variables shared across all GPU threads are prefixed with "g", and of those shared only across a single threadblock (TB) are prefixed with "tb".

tions without GPUfs (a detailed analysis is in Section 2) is that the output size is unknown and potentially large, which usually requires pre-allocation of large amounts of memory to avoid buffer overflow. GPUfs eliminates this problem by writing the output directly to a file.

We choose a simple algorithm where each threadblock opens one file at a time, then each thread scans the file for a subset of the dictionary that it is allocated to match. This workload places high pressure on GPUfs because most of the files are small (a few kilobytes on average), leading to frequent calls to `gopen` and `gclose`.

Figure 3 shows a sketch of the self-contained GPU program used in this experiment. The program does not require CPU code development. For simplicity, this example handles only a single input file, and does not show the details of the string search algorithm. This 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 that if the file fits in the GPUfs buffer cache, only the first threadblock will effectively access the disk, while all others will fetch the data directly from the GPU buffer cache without CPU communication.

The code structure is very similar to a standard CPU implementation. Developers require no special expertise in low-level system programing, and may focus entirely on optimizing the parallel string search algorithm.

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

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

Table 2: GPU string search performance.

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 search for a dictionary of 58,000 modern English words[5], within the complete works of Shakespeare as a single 6MB text file[6], and within the Linux 3.3.1 kernel source containing about 33,000 files holding 524MB of data. The list of input files is itself specified in a file.

As a point of reference we compared two other implementations: a simple CPU program performing the same task on 8 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 only supports inputs and outputs that fit in the GPU's physical memory. In contrast, the GPUfs implementation has no such limitations.

We perform the evaluation on a SuperMicro server with two 4-core Intel Xeon L5630 CPUs and NVIDIA C2075 GPU. The results are presented in Table 2. Even for such a file-system intensive workload, a single GPU outperforms the 8-core CPU by 6.8×. The GPUfs version performs similarly to the vanilla GPU implementation on one large input file, with the GPUfs version being only 9% slower than the vanilla one on the Linux kernel input. Yet, GPUfs-based code is about half the length of the vanilla version.

6. DISCUSSION

This article advocates 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 key to broader adoption of GPUs now and in the future. In fact, our work marks yet another turn of the "wheel of reincarnation" [10], demonstrating inevitable evolution of GPUs toward providing more general-purpose functionality. We implement a file system for discrete GPUs to demonstrate the feasibility and value of this goal on real GPU hardware.

However, our work has limitations. We focus primarily on a file system layer, which might not reflect the challenges and tradeoffs for other operating system services on GPUs. However, we believe that the main principles underpinning the file system layer design are general, as they address the core characteristics of GPU hardware: massive parallelism, slow sequential execution, and NUMA memory organization. In fact, we found them equally applicable to the design of a networking layer for GPUs, which is ongoing work.

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 casts the GPU as a coprocessor with the CPU managing the interaction between the GPU and the rest of the system. As a result,

[5] `http://www.mieliestronk.com/wordlist.html`
[6] `http://www.gutenberg.org/ebooks/100`

the bulk of GPU research has been on devising new CPU-side abstractions to make this interaction easier to manage. For example, flow graph execution frameworks like PTask [15], facilitate the development of complex applications but treat GPU tasks as opaque monolithic execution units. In contrast, we propose to offer 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 has been a natural and convenient way to provide I/O functionality to GPU programs.

This work targets discrete GPUs and does not directly address emerging hybrid processor architectures like AMD Kaveri and mobile System-on-Chip (SoC) that combine a CPU and a GPU on the same die. The most recent processors add support for shared physical memory and virtual address space between a CPU and GPU. Communicating through shared memory makes CPU-GPU data transfers unnecessary, providing a much faster and easier way to exchange data between the processors. Consequently, such tight CPU-GPU coupling might seem to obviate the need for the separate system abstraction layer on GPUs advocated here, making sufficient the GPU-as-coprocessor model with its reliance on the CPU operating system.

We argue that shared memory hardware is only a low-level interface that optimizes CPU-GPU interaction, it does not eliminate the software complexities of implementing interactions between the GPU and the rest of the system. Just 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 [1], using this mechanism for implementing file or network I/O from thousands of GPU threads would overwhelm the CPU operating system, and warrants explicit handling of massive parallelism on both the GPU and the CPU, as done in GPUfs. Moreover, CPU-GPU coordination requires an 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 to the OpenCL standard. In contrast, familiar system abstractions and portable, platform-optimized APIs hide these and other low-level details and naturally facilitate the code development on hybrid architectures, as they do in CPUs and discrete GPUs.

Discrete GPUs remain relevant.

We believe discrete and integrated GPUs will continue to co-exist for years to come. They embody different tradeoffs between 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 stand-alone, add-on peripheral device, which gives the designers much greater hardware design flexibility than integrated systems. The aggressively throughput-optimized hardware designs of discrete GPUs heavily rely on a fully-dedicated, multi-billion transistor budget, tight integration with specialized high-throughput memory, and increased thermal design power (TDP). As a result, discrete GPUs offer the highest compute performance and compute performance per watt which make 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, GPUs now provide direct access to peripheral devices, such as storage and network adapters, eliminating the CPU from the hardware data path. Future high throughput processors [5] are expected to enable more efficient sequential processing, and some indications of this trend can already be observed. For example, the AMD Graphics Core Next 1.1 used in all modern AMD GPUs, contains a scalar processing unit. In addition, NVIDIA-IBM recently announced 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 is 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. The current Xeon-Phi system stack, however, does not allow efficient access to host files and network, and programmers are encouraged to follow a more traditional co-processor 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 processor memory subsystem will include high-bandwidth, size-limited 3D stacked memory. This stacked memory will have exaggerated NUMA properties, and the ideal system stack design on such memory remains to be seen. We expect that many aspects of GPU system abstractions described here, e.g., NUMA-aware file cache locality optimizations, will be relevant to the coming and future generations of these processors.

GPU productivity efforts.

Recent developments make it significantly easier to accelerate computations on GPUs without writing any GPU code. There are comprehensive STL-like libraries of GPU-accelerated algorithms [12], efficient domain-specific APIs [11], and offloading compilers [13] that parallelize and execute specially annotated loops on GPUs.

These and other GPU productivity projects use the GPU as a co-processor and passive consumer of data. Applications that need to orchestrate data movement are cumbersome to implement because GPU code cannot perform I/O calls directly. System-wide 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.

7. REFERENCES

[1] I. Bratt. *HSA Queuing, Hot Chips Tutorial 2013. Retrieved 05/2014.* `http://www.slideshare.net/hsafoundation/hsa-queuing-hot-chips-2013`.

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

[3] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computing Systems*, 6(1), February 1988.

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

[5] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the future of parallel computing. *Micro, IEEE*, 31(5):7–17, 2011.

[6] Khronos. *Khronos Group: The OpenCL Specification*, 2013.

[7] D. B. Kirk and W.-m. W. Hwo. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2010.

[8] D. Lehavi and S. Schein. *Fast Regexp parsing on GPUs, Retrieved 09/14*. `http://on-demand.gputechconf.com/gtc/2012/presentations/S0043-GTC2012-30x-Faster-GPU.pdf`.

[9] T. Mostak. An overview of mapd (massively parallel database). Technical report, 2013.

[10] T. H. Myer and I. E. Sutherland. On the Design of Display Processors. In *Communications of the ACM*, volume 11, June 1968.

[11] NVIDIA. *GPU-accelerated high performance libraries, Retrieved on 09/14*. `https://developer.nvidia.com/gpu-accelerated-libraries`.

[12] NVIDIA. *NVIDIA Thrust library, Retrieved 09/2014*. `https://developer.nvidia.com/thrust`.

[13] NVIDIA. *PGI accelerator compilers with OpenACC directives. Retrieved 09/2014*. `www.pgroup.com/resources/accel.htm`.

[14] NVIDIA. *Popular GPU-accelerated applications*. `http://www.nvidia.com/object/gpu-applications.html`.

[15] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 233–248, 2011.

[16] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating Systems Principles*, pages 378–379. Wiley, 2008.

[17] M. Silberstein. *GPUfs home page, Retrieved on 09/14*. `https://sites.google.com/site/silbersteinmark/Home/gpufs`.

[18] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: integrating file systems with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13. ACM, 2013.

[19] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: integrating file systems with GPUs. *ACM Transactions on Computer Systems*, 32(1):1:1–1:31, Feb. 2014.

[20] Wikipedia. Seqlock. Retrieved 03/2014, 2002. `http://en.wikipedia.org/wiki/Seqlock`.