GPUfs: Integrating a File System with GPUs

MARK SILBERSTEIN, University of Texas at Austin
BRYAN FORD, Yale University
IDIT KEIDAR, Technion
EMMETT WITCHEL, University of Texas at Austin

As GPU hardware becomes increasingly general-purpose, it is quickly outgrowing the traditional, constrained GPU-as-coprocessor programming model. This article advocates for extending standard operating system services and abstractions to GPUs in order to facilitate program development and enable harmonious integration of GPUs in computing systems. As an example, we describe the design and implementation of GPUfs, a software layer which provides operating system support for accessing host files directly from GPU programs. GPUfs provides a POSIX-like API, exploits GPU parallelism for efficiency, and optimizes GPU file access by extending the host CPU's buffer cache into GPU memory. Our experiments, based on a set of real benchmarks adapted to use our file system, demonstrate the feasibility and benefits of the GPUfs approach. For example, a self-contained GPU program that searches for a set of strings throughout the Linux kernel source tree runs over seven times faster than on an eight-core CPU.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design; I.3.1 [**Hardware Architecture**]: Graphics processors

General Terms: Accelerators, Operating Systems, Performance

Additional Key Words and Phrases: Operating Systems Design, GPGPUs, File Systems, accelerators

ACM Reference Format:
Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel, 2013. GPUfs: Integrating a File System with GPUs. *ACM Trans. Comput. Syst.* 1, 1, Article 1 (January 1), 30 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

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 [NVI]. 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*, like files and sockets, that CPU programmers have taken for granted for decades. Today's GPU is a bit of an *idiot savant*: it is capable of amazing computational feats when spoon-fed with the right data and micro-managed by application code on the host CPU, but it is incapable of initiating even the simplest system interactions for itself, such as reading an input file from a disk. The traditional coprocessor-style GPU programming model

This research was supported in part by NSF grants CNS-1017785 and CNS-1017206, by the Andrew and Erna Fince Viterbi Fellowship, and by a 2010 NVIDIA research award.
Author's addresses: M. Silberstein and I. Keidar, Electrical Engineering Department, Technion; Bryan Ford, Computer Science Department, Yale University; Emmett Witchel, Computer Science Department, University of Texas at Austin.

requires developers to explicitly manage GPU I/O on the host CPU, which increases the design complexity and code size of even simple GPU programs that require file access. While programmers can explicitly optimize CPU-GPU interactions, these optimizations are not portable to new generations of hardware, affect software modularity and make it hard to maintain functionality and performance.

Drawing an analogy to pre-virtual memory days, applications often managed their own address spaces efficiently using manual overlays, but this complex and fragile overlay programming ultimately proved not worth the effort. Similarly, 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. As GPUs are quickly evolving toward general high-performance processors useful for a wide variety of massively parallel, throughput-oriented tasks, we believe GPU programming should reap the same benefits from the file system abstraction enjoyed by CPU programmers.

We propose **GPUfs**, an infrastructure that exposes the file system API to GPU programs, bringing the convenience and power of file systems to GPU developers. GPUfs offers compute-intensive applications 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 replicated across several GPUs or other coprocessors. Further, GPUfs lets the OS optimize data access locality across independently-developed GPU compute modules, using application-transparent caching and data replication, much like a traditional OS's buffer cache optimizes access locality across multi-process computation pipelines. A unified file API interface abstracts away the low-level details of different GPU hardware architectures and their complex inter-device memory consistency models, improving code and performance portability. GPUfs expands the appeal of GPU programming by offering familiar, well-established data manipulation interfaces instead of proprietary GPU APIs. Finally, GPUfs allows GPU code to be self-sufficient, by simplifying or eliminating the complex CPU support code traditionally required to feed data to GPU computations.

Two key GPU characteristics make developing OS abstractions for GPUs challenging– massive data parallelism, and independent memory systems. GPUs are optimized for data parallel processing, where the same program operates on 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 the 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 system memory is an order of magnitude slower, because it requires communication over the 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 to optimize for access locality in data placement and reuse across CPU and GPU memories. GPUfs distributes its buffer cache across all CPU and GPU memories to efficiently enable idioms like process pipelines that read and write files from the same or different processors.

We evaluate a prototype implementation of GPUfs on an x86 PC with four NVIDIA GPUs, using several microbenchmarks and two realistic I/O intensive applications. All the presented GPUfs workloads are implemented entirely in the GPU kernel without CPU-side

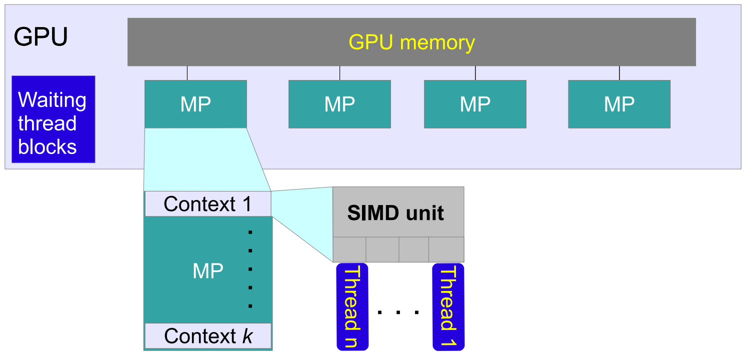

Fig. 1. Hierarchical hardware parallelism in a GPU.

application code. In sequential file access benchmarks, a trivial 16 line GPU kernel using GPUfs outperforms a simple GPU implementation with manual data transfer by up to 40%, and comes within 5% of a hand-optimized double-buffering implementation. A matrix multiply benchmark illustrates how GPUfs easily enables access to datasets larger than the GPU's physical memory, performs faster than the manual double-buffering typical in current GPU code, and is about 2× smaller in code size. Another matrix product benchmark demonstrates the performance benefits of the GPU buffer cache, improving performance by over 2× on average when the input data is reused. Two parallel data analysis applications, prioritized image matching and string search, highlight the ability of GPUfs to support irregular workloads in which parallel threads open and access dynamically-selected files of varying size and composition.

This paper makes the following main contributions.

(1) The first POSIX-like file system API we are aware of for GPU programs, with semantics modified to be appropriate for the data-parallel GPU programming environment.
(2) A design and implementation of a generic software-only buffer cache mechanism for GPUs, employing a lock-free traversal algorithm for parallel efficiency.
(3) A proof-of-concept implementation of GPUfs on NVIDIA Kepler GPUs, supporting multi-GPU systems.
(4) A quantitative evaluation of a GPU file system that identifies sensitive performance parameters such as page size, and evaluates efficiency relative to hand-coded solutions.

The next section provides an overview of the GPU architecture. Next we highlight the main GPU programming challenges caused by the lack of file access abstractions on GPUs. We then explain and justify the design choices that we made while building GPUfs, followed by the implementation details of GPUfs on NVIDIA FERMI and KEPLER GPUs. We evaluate the GPUfs prototype implementation in Section 6, summarize related work, and finally discuss applicability of GPUfs ideas in emerging software and hardware systems.

2. GPU ARCHITECTURE OVERVIEW

We provide a simplified overview of the GPU software/hardware model, highlighting properties that are particularly relevant to GPUfs. We use NVIDIA CUDA terminology because we implement GPUfs on NVIDIA GPUs, but most other GPUs that support the cross-platform OpenCL standard [OpenCL] share the same concepts.

Hardware model. GPUs are parallel processors which expose programmers to hierarchically structured hardware parallelism, as depicted in Figure 1. At the highest level, GPUs are similar to CPU shared-memory multicores. GPUs support coarse-grain task-level parallelism via concurrent execution of different tasks on different powerful cores, called *multiprocessors*

(*MP*). Each MP features several wide, single instruction multiple data (SIMD) vector units comprised of individual lightweight processing elements called *CUDA cores* (for example, 32 CUDA cores per SIMD unit). Similar to CPU vector hardware, MPs expose fine-grain data-level parallelism allowing the same instruction to concurrently process multiple data items.

The main architectural difference between CPUs and GPUs lies in the way GPUs execute parallel code. A single GPU program, termed a *kernel* (unrelated to an operating system kernel) comprises tens of thousands of individual *threads*. Each GPU thread forms a basic sequential unit of execution. Unlike CPU threads which usually run exclusively occupy one CPU core each, hundreds of GPU threads are concurrently scheduled to run on each MP. At any point a single CUDA core is executing only a single GPU thread (marked in blue in the figure), but the hardware scheduler multiplexes many of them onto a single core to maximize hardware utilization. When a running thread gets stalled, while waiting on a slow memory access for example, the hardware restarts another thread that is ready for execution. The switching between the threads is highly efficient because the hardware maintains the execution state, or *context*, of all threads, and not only those actively executing. Thus the scheduler restarts the thread by simply switching to its context. This type of parallelism, sometimes called thread-level parallelism, or simultaneous multithreading (SMT) [Tullsen et al. 1996], is essential to achieving high hardware utilization and performance in GPUs.

In order to amortize instruction fetch and execution overheads, a hardware scheduler manages threads in small groups called *warps* (32 threads in NVIDIA GPUs), executing all threads in a warp in lockstep on the same SIMD unit.

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. GPU programs can be implemented in plain C++/C or Fortran with only few restrictions and minor language extensions. 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 two thousand threads which may communicate, share state and synchronize efficiently, enabling coordinated data processing within a threadblock. A threadblock is a course-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. To draw analogy with CPUs, if we think of a GPU kernel as a single CPU process, then a threadblock is analogous to a CPU thread.

An application enqueues all threadblocks comprising a kernel into a global hardware queue on a GPU. The number of threadblocks in each kernel ranges from tens to hundreds, and typically exceeds the number of MPs, leaving some threadblocks waiting in the hardware queue until resources become available. Oversubscribing the MPs facilitates load balancing and portability across GPU systems with different number of MPs. 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. Discrete GPUs are peripheral devices connected to the host system via an internal PCI Express (PCIe) bus. They feature their own physical memory on the device itself. The GPU's bandwidth to local memory is an order of magnitude higher – over 30× in current systems –than the PCIe bandwidth to the memory on the host. Discrete GPU memory has a separate address space that cannot be directly referenced by

CPU programs. Moving the data in and out of GPU memory efficiently requires direct memory access (DMA).

Currently, 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. For example, there are about 50 different memory management functions in the CUDA API [NVIDIA 2013]. As a result, managing GPU computations in GPU-accelerated programs entails significant design and implementation complexity.

3. GPU PROGRAMMING CHALLENGES

Despite their popularity in high-performance scientific computing, GPUs remain underutilized in commodity systems. The list of 200 popular general-purpose GPU applications recently published by NVIDIA [NVI] 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 [HPL ; Han et al. 2010]. 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.

GPUs currently require 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 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 is intended to do the same for GPU programs.

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 GPU pattern matching algorithms [HPL].

Using GPUs presents several *system-level* challenges, 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 "spoon-feed" 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: they split inputs and outputs into smaller chunks, and asynchronously invoke 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.

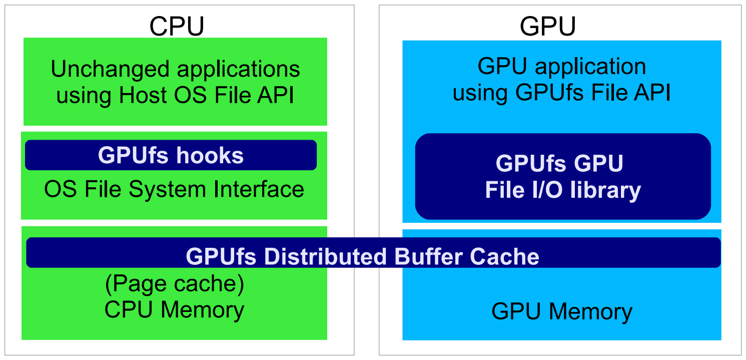
CPU
Unchanged applications using Host OS File API
GPUfs hooks
OS File System Interface
GPUfs Distributed Buffer Cache
(Page cache) CPU Memory
GPU
GPU application using GPUfs File API
GPUfs GPU File I/O library
GPU Memory

Fig. 2. GPUfs architecture

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.

GPUfs aims to alleviate these challenges. It exposes a single file system shared across all processors in the system and accessible via standard familiar API, thereby simplifying GPU development and facilitating integration of GPU programs into complex software systems.

4. 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.

Figure 2 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 essential properties of discrete GPUs that make designing GPUfs challenging: massive hardware parallelism, fast, separate physical memory, and non-preemptive hardware scheduling. We first summarize their implications on the design of GPUfs in Table I, with the detailed analysis in the rest of this section.

Table I. Implications of the GPU hardware characteristics on the 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
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 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 lock-step 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 an OS level, but library supports per-warp or per-threadblock local file descriptors

4.1. 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.

Data consistency model. One important design decision is the choice of a cache *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, the 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 processors 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 [NFS] 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 the bandwidth from the GPU to system memory and to local GPU memory, makes the system more similar to a distributed environment with slow communication network rather than a tightly coupled local environment.

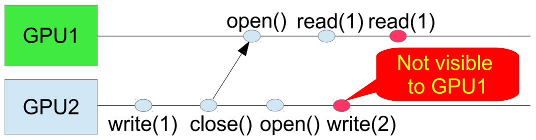

Fig. 3. Close-to-open data consistency in GPUfs.

GPUfs therefore implements a weak consistency model (close-to-open consistency), similar to the Andrew file system (AFS [Howard et al. 1988]) and modern versions of NFS [NFS]. 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 3, GPU2 writes two different values to a file. However GPU1 will see only "1" and may not see "2", because close-to-open consistency permits to postpone the 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. If a file is mapped using `mmap`, however, GPUfs naturally inherits the consistency model implemented by the hardware.

Concurrent non-overlapping writes to the same file. In the potentially common situation in which a parallel task is executing on several GPUs and CPUs in one system, the same file may be write-shared among all executing processors. Concurrent tasks typically write into different parts of the file: i.e., to the particular range each task is assigned to produce. While traditional implementations of close-to-open semantics like AFS leave the results of concurrent writes undefined, GPUfs is designed to properly handle concurrent writes to disjoint file regions.

Supporting concurrent disjoint writes requires solving the well-known problem of false sharing, in this case of buffer cache pages among different GPUs. False sharing arises because GPUfs uses large pages in its buffer cache. The pages are larger than CPU standard pages to allow efficient data transfers over PCIe bus and amortize management overhead. False sharing occurs when different processors write to different file offsets that are close enough to be mapped to the same page. Two writers who write to the same page end up caching two partially modified versions of that page. Writing back both pages in full results in a lost write—only the last writer's data is present and the first writer's data is lost. A proper implementation must merge the writes. GPUfs applies a classic diff-and-merge technique [Amza et al. 1996] to determine which specific portions of a page were modified on a given GPU when propagating those modifications to the host.

An important common case is *write-once* file access, where GPU application threads produce a new output file without ever reading it or overwriting already-written data. To avoid the costs of data transfers in this case, GPUfs attaches special semantics to files an application opens in a new (`O_GWRONCE`) open mode. GPUfs *never* reads pages of such files from the host into the GPU cache. Instead, when the GPU propagates locally written pages back to the host, the merge operation can be optimized to a byte-by-byte OR of the current and the new versions of the page. These semantics imply that concurrent writes by multiple GPUs are guaranteed to merge correctly only if threads write *only once* to *disjoint* file areas. We believe these constraints are consistent with common practices in file-producing data parallel applications, and thus place reasonable semantic demands on applications in order to enable important data movement optimizations.

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. The former approach reduces performance by permanently consuming a portion of GPU hardware resources, thereby reducing the performance of all GPU applications including those not using GPUfs; whereas the latter violates the correctness of GPU applications that rely on the availability of a specific number of threadblocks for execution (e.g., by causing deadlock).

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.[1] This limitation precludes the use of efficient one-side 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 [Effelsberg and Haerder 1984], 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 writeback of dirty pages in a ring buffer which it shares with the CPU so the CPU can asynchronously complete the transfer. The GPU produces dirty pages, enqueues them into the ring buffer, and the CPU dequeues them and copies them. This single producer, single consumer pattern does not require atomic operations.

4.2. GPUfs API

It is not obvious 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 different 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.

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

Threads in a warp execute in lockstep. Even though according to the programming model GPU threads are independent, the hardware executes threads in SIMD groups, or warps (see § 2 for detailed discussion). 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, and per-thread APIs would not fit in this design pattern. 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 warp—rather than thread—granularity. All application threads in a warp 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 warp granularity of the API allows the GPUfs *implementation* to parallelize the handling of API calls across threads in the invoking warp—parallelizing file table search operations, for example. Our current implementation supports even more course-grained per-threadblock granularity, which, in fact, we found 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 that must be closed with `close`. The benefit of this design is that a file descriptor can be initialized once, and then reused by all other GPU threads to save the overhead of 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 returns the *same* file descriptor when given the same file name argument. GPUfs reference counts these files, 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 much more convenient to use than the low-level GPU `open` call. It is also generally more efficient than an `open` call because it does not need to contact the CPU.

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` as 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, we expect performance considerations to render traditional `mmap` semantics impractical 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 `gmmap`, permitting more efficient implementation in a data parallel context by avoiding 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 increase efficiency 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.

4.3. 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` (GPUfs analogues of `fsync` and `msync`) ensures that data has been written to the host buffer cache. Note that successful completion of `close` 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.

Unfortunately, GPU failures are more frequent than CPU failures and have severe implications. In existing systems, a GPU program failure—such as an invalid memory access or assertion failure—may require restarting the GPU card, thus losing all GPU memory state. As GPUs continue to become more general-purpose, we expect GPU hardware to gain more resilience to software failures.

4.4. Resource overheads

Operating systems are known to compete with user programs for hardware resources such as caches [Soares and Stumm 2010], and are often blamed for decreased performance in high-performance computing environments. GPUfs is a 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 overheads, 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

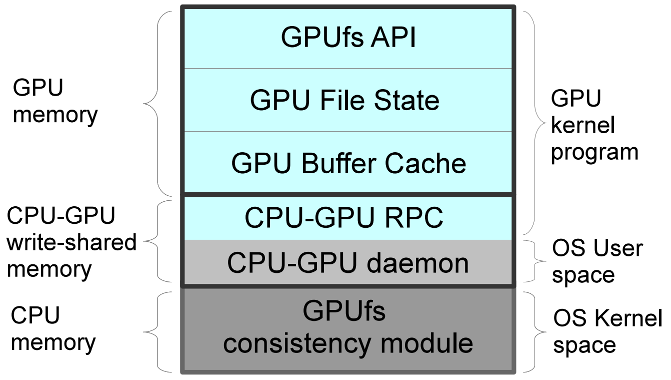

Fig. 4. Main GPUfs software layers and their location in the software stack and physical memory.

caches. And despite the cost, we find GPUfs to have good performance in useful application scenarios (§6).

5. IMPLEMENTATION

This section describes our GPUfs prototype for NVIDIA FERMI and KEPLER GPUs. We first outline the prototype's structure and how it implements the above API, then explore implementation details and challenges. We cover buffer cache management, GPU-CPU communication, file consistency management, and limitations of the current prototype. Some of these implementation choices are likely to be affected by future GPU evolution, but we feel that most considerations discussed here will remain relevant. For simplicity, our current implementation supports parallel invocation of the GPUfs API only at threadblock and not warp granularity. GPUfs calls represent an implicit synchronization barrier, and must be called at the same point with the same parameters from all threads in a threadblock.

Most of GPUfs is a GPU-side library linked with application code. The CPU-side portion runs as a user-level thread in the host application, giving it access to the application's CUDA context.

Figure 4 shows the three main software layers comprising GPUfs, their location in the overall software stack shown on the right and indicated by different colors, and the type of memory the relevant data structures are located in shown on the left.

The top layer is the core of GPUfs, which runs in the context of the application's GPU kernels and maintains its data structures in GPU memory. This layer implements the GPUfs API, tracks open file state, and implements the buffer cache and paging.

The communication layer manages GPU-CPU communications, and naturally spans the CPU and GPU components. Data structures shared between the GPU and CPU are stored in write-shared CPU memory accessible to both devices. This layer implements a GPU-CPU Remote Procedure Call (RPC) infrastructure, to be detailed in Section 5.3.

Finally, the GPUfs consistency layer is an OS kernel module running on the host CPU, which manages consistency between the host OS's CPU buffer cache and the GPU buffer caches, according to the file consistency model described above in §4.

The GPUfs file system is inspired by the Linux file system and buffer cache. We now examine its function in more detail.

5.1. File system operations

Table II summarizes the GPUfs API.

Open and close. GPUfs keeps track of open and recently closed files in several tables. Each open file has an entry in the *open file table*. This table holds a pointer to a radix tree

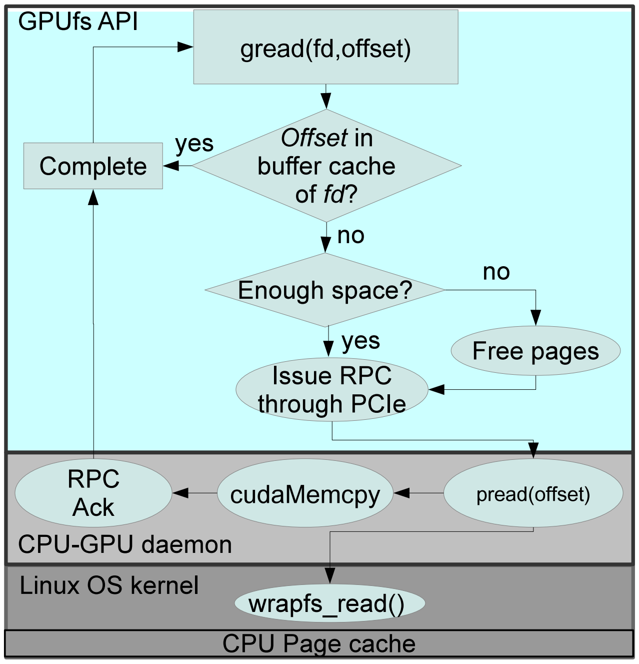

Fig. 5. Functional diagram of a call to `pread`. Color scheme is the same as Figure 4.

Table II. GPUfs API.

API	**Explanation**
`gread/gwrite`	Reads and writes always supply explicit file offsets, as in `pread` and `pwrite`.
`gopen/gclose`	Open and close files in the namespace of a single threadblock. Multiple concurrent open requests to open or close the same file are coalesced into one open/close.
`gfsync`	Synchronously write back to the host all dirty file pages that are currently neither memory-mapped nor being accessed concurrently via `gread` or `gwrite` calls.
`gmmap/gmunmap`	A relaxed form of `mmap` that avoids double copies in `gread/gwrite`. Imposes API constraints discussed in §4.2.
`gmsync`	Write back a specific dirty page to the host. The application must coordinate calls to `gmsync` with updates by other threadblocks.
`gunlink`	Remove a file. Files unlinked on the GPU have their local buffer space reclaimed immediately.
`gfstat`	Retrieve file metadata. File size reflects file size at the time of the first `gopen` call that opened this file on the host.
`gftruncate`	Truncate a file to a given size, and reclaim any relevant pages from the buffer cache.

indexing the file's pages. For each file, the table stores several file parameters, including the pathname and the CPU file descriptor used for data requests handled by the CPU. Finally, each entry stores a reference count of the number of threadblocks holding the file open.

When a file is closed its pages are retained in GPU memory until they are reclaimed for caching other data. The *closed file table* maintains pointers to the caches of closed files, and is a hash table indexed by file inode number in the CPU file system. Because of GPU hardware thread scheduling, files can appear to be closed while still in use by threadblocks that have yet to be scheduled. To optimize for this case and to support data reuse in and across kernels, `gopen` checks the closed file table first, and moves the file cache back to the open file table.

If a file was changed, `close` enqueues all the file's dirty pages to a *write queue*. The CPU then asynchronously fetches the pages from the queue and flushes them to the disk. The write queue is implemented as a lock-free ring buffer with the CPU as a consumer and the GPU as a producer. Single-producer/consumer ring buffer does not require atomic operations and thus works correctly over PCIe-2 bus that does not support inter-device atomics.

Reads and writes. Reads and writes work as expected, first checking the cache for the relevant block, and forwarding requests to the CPU and allocating cache space as necessary. Figure 5 shows a functional summary of `pread`'s operation. Reads and writes exploit the GPU's fine-grain parallelism by using many threads to copy data or initialize pages to zero collaboratively. Reference counts protect pages during memory transfers.

When `write` completes, each thread issues a memory fence to ensure that updates reach GPU memory, in case the GPU buffer cache needs to write the page back to the CPU. Otherwise, due to the GPU's weak memory consistency model, the data paged back via a DMA from the GPU memory might be left inconsistent because the writes might remain buffered in the GPU's L1 cache.

File management operations. File management operations such as `gunlink` and `gftruncate` each generate an RPC to the CPU to request the respective operation on the host. They also reclaim the file page cache on the GPU if necessary.

5.2. GPU buffer cache

Pages, page frames and page table. GPUfs manages file content at the granularity of a buffer cache *page*. This page size is configurable, though performance considerations typically dictate page sizes larger than OS-managed pages on the host CPU—e.g., 256KB, since GPU code often parallelizes the processing of a page across many threads in a threadblock (on the order of 256 threads). The ideal page size depends on empirical considerations explored further in §6. For efficiency, GPUfs pre-allocates pages in a large contiguous memory array, which we call the *raw data array*.

As in Linux, each page has an associated *pframe* structure holding metadata for that page, e.g., the size of the actual data in the page, dirty status, and others. Unlike Linux, pframes contain some file-related information, such as a unique file identifier used for lock-free traversal, and the page's offset in the file, because in GPUfs all pages are backed by a host OS file. We chose to eliminate additional layers of indirection where possible to speed up the access to the data.

Pframes are allocated in an array separate from the pages themselves, but the ith pframe in this array holds metadata for the ith page in the raw data array, making it easy to translate in both directions, as needed in operations such as `gmunmap` and `gmsync`.

Per-file buffer cache. The buffer cache keeps replicas of previously accessed file content for later reuse. For simplicity the GPUfs buffer cache is per-file, not per-block device as in Linux, but future GPU support for direct access to storage devices may motivate reconsideration of this decision.

We depict main buffer cache data structures in Figure 6. A dynamic radix tree indexes each file's buffer cache, enabling efficient page lookups given a file offset. Last-level nodes in the tree hold an array of *fpage* structures, each with a reference to a corresponding pframe. The fpages manage concurrent access to the respective pframes: each holds a read/write reference count and a spinlock, together preventing concurrent access by mutually exclusive operations such as initialization, read/write access, and paging out. The fpages are allocated not by reference, but by value within radix tree nodes. We use in-place data structures to avoid pointer traversal and minimize memory allocations, even though all dynamic memory is managed by GPUfs via special allocators.

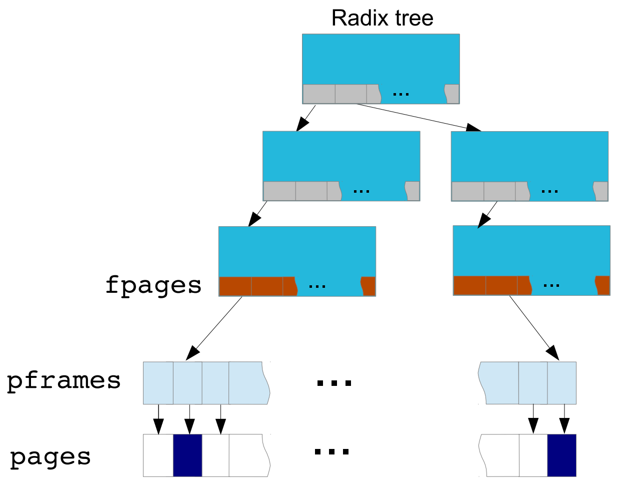

Fig. 6. Buffer cache data structures. Pages occupied with data are marked blue.

Buffer cache management. GPUfs implements a FIFO policy by tracking allocation of last-level radix tree nodes. Newly allocated nodes are placed at the head of a doubly-linked list. The node allocation time is a reasonable proxy for the allocation time of the individual pages referred by that node, because these pages are often initialized and accessed together by parallel threads. When a thread needs to evict pages from the buffer cache, it performs a lock-free traversal of this list to reclaim a desired number of pages from a particular file.

To choose the file whose pages will be reclaimed, GPUfs uses a policy similar to Linux's. GPUfs first looks at closed files, which are not in use so their content can be evicted with lower performance penalty for the running application. Furthermore, their pages are more likely to be clean, so they can be reclaimed without GPU-CPU communication. GPUfs then looks for pages from read-only open files, and as a last resort chooses pages from writable open files.

Lock-free buffer cache access. The buffer cache radix tree is a major contention point among threads accessing the same file. These accesses must be synchronized to avoid data races, such as concurrent attempts to initialize pages belonging to the same intermediate node, or node deletion due to page reclamation, which may be performed concurrently with page lookup.

GPUfs uses lock-free reads and locked updates, similar to Linux's seqlocks [Hemminger 2002]. Updates maintain the radix-tree invariants used by readers, and all fields are initialized before a new node becomes visible to readers. Reads can fail, in which case they retry. GPUfs retries once without locking, then locks on its third attempt. To check that the page found is correct, GPUfs assigns a unique identifier to each radix tree during initialization, then propagates this identifier to every page referenced by the tree. This identifier, combined with the page offset, uniquely identifies the page.

The paging algorithm also uses lock-free reads on a doubly linked list used as a FIFO queue.

5.3. GPU-CPU Remote Procedure Call

GPUfs implements an RPC protocol to coordinate data transfers between a CPU and a 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.

The challenge of implementing an efficient RPC protocol lies in the CPU/GPU memory consistency model. GPU consistency models are tailored to the bulk-synchronous GPU programming model, where GPU-CPU communications traditionally occur only at kernel invocation boundaries and not while the kernel is running. Except at these points, CPU/GPU consistency is not guaranteed. Our RPC system is thus not currently portable to all GPUs, but relies on hardware providing the following consistency features.

(1) **GPU-CPU memory fences.** GPU file read and write requests must be delivered to the CPU while the GPU kernel is running. This is only possible if consistent updates of the CPU-GPU write-shared memory can be enforced in both directions.
(2) **GPU cache bypass.** To allow consistent reads of GPU memory from the running GPU kernel, after this memory has been updated by CPU-initiated DMA transfers, GPU reads must either invalidate or bypass the GPU's L1 and L2 caches.

The latest stable version 1.2 of the OpenCL standard [OpenCL], and consequently AMD's discrete GPUs, currently do not support these features [2]. Only NVIDIA GPUs currently satisfy all of our requirements.

Challenges due to hardware constraints.. RPC implementation is complicated by the lack of atomic operations over the PCIe bus. The new PCIe-III standard includes atomics, but implementation is optional and we know of no hardware currently supporting it.

This limitation precludes the use of efficient one-side communication protocols, as discussed in §4.1. 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, the current implementation must resort to a less efficient message-passing protocol for synchronization.

Today's GPUs also lack a signal-like mechanism accessible to applications, to notify a host CPU process of events originating on the GPU. The current API offers the CPU only coarse-grained notifications when entire GPU kernels or memory transfers complete, and do not allow code *within* a GPU kernel to send notifications. The CPU must therefore poll the GPU-CPU shared memory region continuously to detect RPC requests from the GPU.

RPC protocol implementation.. GPU-CPU communications in GPUfs follow a synchronous client-server protocol, where the GPU sends requests to the CPU and waits for the CPU to acknowledge the request's completion. The RPC request channel is a FIFO queue in write-shared memory, which the CPU polls for requests. Each GPU in the system has a separate RPC request queue, managed exclusively by the GPU that owns that queue.

The GPU uses its request queue only to send commands: when the GPU issues a bulk transfer request such as a bulk data read or write, the CPU initiates a DMA-based bulk data transfer directly to or from the respective GPU buffer cache pages, using source or destination pointers supplied by GPU code. The CPU then notifies the GPU when the transfer completes.

The RPC queue usually contains multiple concurrent requests that, in principle, CPU code could handle in parallel. Our implementation uses a single-threaded, event-based design on the host to restrict the GPU-related CPU load to one CPU, simplify synchronization, and to avoid overwhelming the disk subsystem with concurrent requests. Our implementation thus currently orders file accesses, but data transfers to and from the GPU use multiple asynchronous CPU-GPU channels to utilize full-duplex DMA and overlap GPU-CPU transfers with disk accesses and GPU execution.

[2]OpenCL 2.0 formalizes inter-processor memory consistency model and enables more forms of synchronization, but which architectures will support that and their performance costs is yet to be known.

5.4. File consistency management

The current prototype implements the locality-optimized file consistency model described in Section 4, though currently only for the common cases of files opened in either read-only (`O_RDONLY`) or write-once mode (`O_GWRONCE`, see §4.2). The GPUfs prototype does not yet implement the diff-and-merge protocol required to support general write-sharing, and thus currently supports only one writer processor per file at a time.

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. We call this strategy lazy because closing a file on one GPU or CPU does not actively push an invalidation to other GPUs caching 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. Supporting such invalidations without PCI atomics would require GPUs to run daemon threads waiting for such an invalidation signal, an overhead we wish to avoid (see §5.2).

GPUfs uses WRAPFS [Zadok and Bădulescu 1999], a stackable passthru file system, on the CPU to implement file consistency. WRAPFS is a Linux module that introduces a thin software layer on top of any file system, enabling interposition on calls to the underlying file system. We modified WRAPFS to implement our consistency protocol, enabling seamless integration of GPUfs with unmodified CPU programs. The CPU-side GPUfs daemon communicates with this modified WRAPFS module via a special character device. This device is used solely to update and query file state to implement file consistency, and provides no access to actual file content, thereby leaving the host OS's file access policies uncompromised. We do not currently protect against denial-of-service by misbehaved applications via buffer cache invalidation, however.

In principle, the consistency protocol could be implemented in user space without kernel module, and use the last file modification time to determine the staleness state of the data cached on a GPU. In order to be correct, however, such an implementation must disallow asynchronous writebacks of dirty blocks from the GPU to prevent violation of the close-to-open data consistency.

5.5. Implementation limitations

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.

GPUs contain hardware translation and protection mechanisms that prevent GPU kernels launched by one CPU process from accessing the GPU memory of kernels launched by other processes. Today's GPUs do not offer software interfaces to control this memory protection hardware, however [3]. A GPUfs instance can therefore serve only a single CPU process, and GPUfs cannot share state across GPU invocations by different host processes. For the same reason, GPUfs cannot protect the contents of its GPU buffer caches from corruption by the application it serves. Such features may become feasible once GPU vendors offer appropriate interfaces.

GPUfs does preserve file access protection at the host OS level, however. The host OS prevents a GPUfs application from opening host files the application doesn't have permission

[3]NVIDIA CUDA recently added support for sharing GPU memory across CPU processes, but GPUfs does not use it yet.

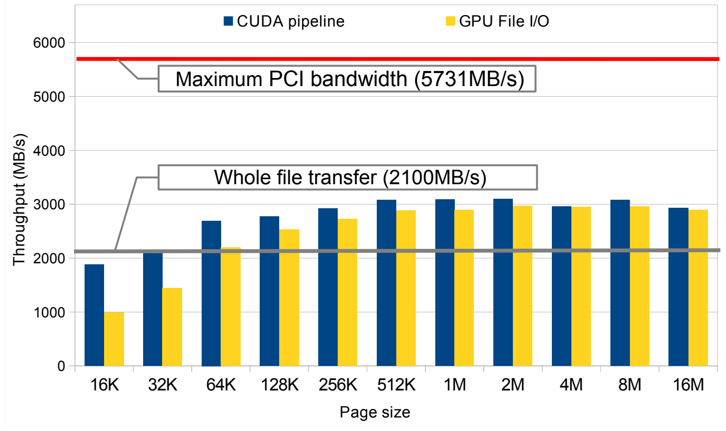

Fig. 7. Sequential read performance as a function of the page size. The red line is the maximum achievable PCI bandwidth on this hardware configuration. Higher is better.

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.

6. EVALUATION

We evaluate GPUfs on a SuperMicro server system featuring two 4-core Intel Xeon L5630 CPUs at 2.13GHz with 12MB L3 cache per CPU, and four NVIDIA TESLA C2075 GPUs, each with 6 GB of GDDR5 memory. We also used TESLA K20c GPU on the same machine. We use Ubuntu Linux kernel 3.0.0-27, with CUDA SDK 5.0, GPU driver 304.54. GPUfs is mounted atop a regular disk partition; the disk is a 500GB WDC WD5003, 7200RPM. The performance as reported by '`hdparm -t -T`' is 6,600MB/s and 132MB/s for cached and disk reads respectively.

We evaluate the system's performance and utility with several microbenchmarks, and also present two more realistic applications. For every data point we report the arithmetic mean of 5 executions after one warm up, unless stated otherwise. In all experiments we found the standard deviation of the results to be less than 1%.

One important property shared by all the test workloads is that their GPUfs implementation required almost no CPU code: they were entirely implemented in the GPU kernel. For all the workloads, the CPU code is identical, save the name of the GPU kernel to invoke. This is a remarkable contrast with standard GPU development, which always requires substantial CPU programming effort. From our experience we found it significantly easier to develop self-contained GPU programs, and believe that self-contained GPU programming will enable broader adoption of GPUs.

6.1. Microbenchmarks

The microbenchmarks below examine basic system performance and its sensitivity to several important configuration parameters.

6.1.1. Sequential file read. We first evaluate the effect of page size on sequential read performance. The benchmark transfers a single 1.8 GB file, in three ways: (a) reading data from the GPU kernel via GPUfs, (b) using the CUDA memory transfer API in chunks the same size as a GPUfs page (CUDA pipeline), and (c) reading the whole file in one chunk and transferring it to the GPU in one CUDA API call.

The GPU file reading kernel runs with 28 threadblocks (twice the number of active multiprocessors in the GPU), where each threadblock maps pages from a contiguous range

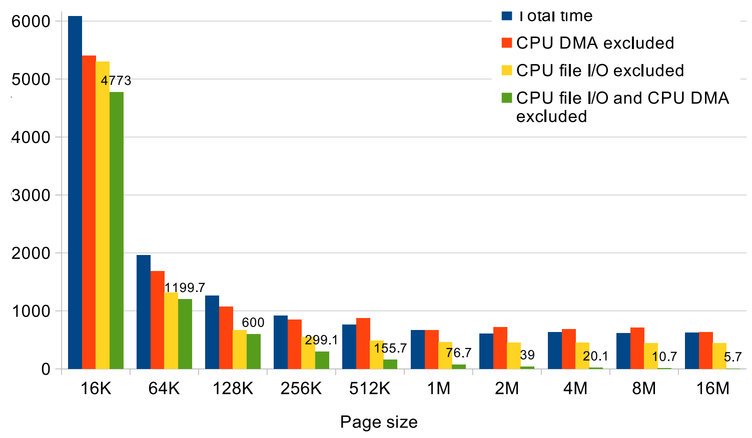

Fig. 8. Contribution of different factors to the file I/O performance as a function of the page size. Lower is better.

in the file. Each threadblock maps one page at a time, until the total 64MB of data is mapped. The number of map requests depends on the page size. The data itself is not accessed, but the pages are fetched from the CPU page cache into the GPU buffer cache. The threadblock then closes the file and exits. GPU file access is not strictly sequential because the order of reads by different threadblock is non-deterministic. We do not anticipate any measurable effect from these non-sequential reads, however, because the file data is cached in CPU memory and fits in the GPU page cache.

The CPU code uses `pread` to read each chunk of the file into pinned CPU memory allocated with `cudaHostMalloc`, then issues an asynchronous `cudaMemcpy` to enqueue a DMA transfer request for that chunk, then proceeds to the next chunk (except in the whole file transfer case in which there is only one big chunk). Dividing the file into chunks overlaps file access latency with DMA data transfers to the GPU. An alternative implementation, which copies file content directly from the CPU page cache exposed via `mmap`, performs worse because it prevents CUDA from optimizing DMA transactions and forces `cudaMemcpy` to be synchronous.

The graph in Figure 7 shows read bandwidth for different page sizes. As expected, small GPUfs pages (less than 64KB) result in low performance, and increasing page size increases performance, with diminishing returns after 512 KB. Reading entire files, a common practice among GPU programmers expecting larger transfers to amortize data transfer overheads most effectively, is in fact less efficient than breaking reads into chunks, as chunks allow overlap of `pread` from the CPU page cache with PCI data transfer. Similar observations were made in [Kato et al. 2012]. The CUDA pipeline implementation appears to achieve the maximum possible file-to-GPU transfer performance on this machine (because the CPU memory bandwidth and the PCIe bandwidth are about the same).

GPUfs outperforms simple CUDA whole file reads at 64 KB pages and higher, and achieves on average within 5% of the bandwidth of the hand-pipelined version, a cost we consider to be a reasonable tradeoff for the convenience GPUfs offers.

Figure 8 breaks down the timing of the microbenchmark, by eliminating PCI data transfer time while leaving only the RPC traffic, eliminating CPU file reads, and eliminating both. The graph shows latency, where lower is better.

Execution time with small pages is dominated by the DMA transfers, which copy too little data per transaction, and by GPUfs API costs. I/O operations become fully overlapped with GPUfs buffer cache code execution for pages larger than 64KB. We see that total page cache

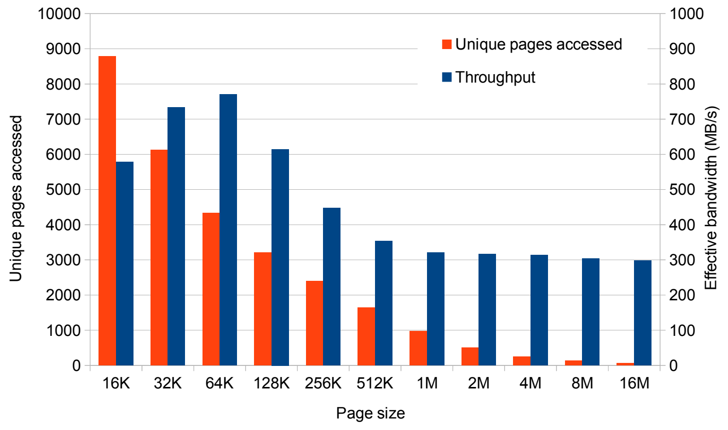

Fig. 9. Random read/write performance as a function of page size. Higher is better.

access overhead (the rightmost labeled column) diminishes proportionally to page size. This is because the total amount of memory mapped by each threadblock is fixed while the page size changes, so the number of map requests performed by each threadblock is reduced as the page size grows. For pages larger than 128K the CPU page cache becomes the main bottleneck.

6.1.2. Random file read. This experiment shows the performance of random file access for different page sizes. This kernel is invoked with 112 threadblocks, where each threadblock reads 32 32KB data blocks from random offsets in a 1GB file, for a total of 112 MB read. The kernel uses `gread` to read the data into a 32KB array allocated in the GPU on-die scratchpad memory. Unlike `gmmap`, `gread` is not constrained in size to a single cache page, hence it is more appropriate for accessing file data at random offsets. Occasionally, different threadblocks may access the same page and fetch it from the GPU buffer cache. Time measurements are an average over 8 runs.

Figure 9 shows that as with sequential reads, small pages lead to bad performance, but now large pages also lead to bad performance. Small pages fail to amortize transfer costs, while large pages transfer too much data that is not actually read by the application. 64KB achieves the best performance in this test.

We calculate effective throughput in this experiment assuming an ideal case of exactly 112MB of data transferred. To support random accesses from GPU code without GPUfs, a GPU program would typically transfer the whole 1GB and perform the random accesses in GPU memory. Assuming the maximum observed throughput of 3100MB/s (see Figure 7), using only one tenth of the total 1GB of transferred data results in an effective random-access throughput of only 310 MB/s, comparable to GPUfs's worst performance using very large pages. Further, without GPUfs, random access to files whose size exceeds the GPU's physical memory is complex and inefficient in hand-coded GPU programs, often requiring frequent, brief kernel invocations between each random access. GPUfs eliminates from the application the design and implementation complexity required to handle such cases efficiently.

In the above experiments, a 128KB page size achieves a reasonable balance between sequential and random access performance. The optimal page size in general depends on application access pattern, however. In the current implementation, in which GPUfs is deployed on a per-application basis, page size may easily be tailored to the particular application's access patterns if necessary.

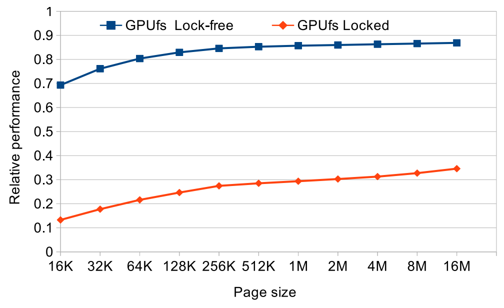

Fig. 10. Buffer cache access performance with and without lock-free radix tree traversal, normalized by the raw memory access time.

6.1.3. Buffer cache access performance. As the "GPUfs-lock-free" case in Figure 10 shows, GPUfs achieves 85–88% of raw memory access performance when accessing files in the GPU buffer cache, for 128KB pages or larger. In this experiment we invoke 112 threadblocks, each reading 64MB of data into the GPU's on-die scratchpad memory in chunks of 16KB. The baseline implementation reads data directly from the GPU's main memory, without using the GPUfs API. The GPUfs implementation reads data from the cached file via `gread`, passing to `gread` a direct pointer to the destination buffer in scratchpad memory. The file is fully prefetched into the GPU page cache by another previously invoked kernel, excluding PCI transfer time from the measurements. We randomized the memory accesses so that every 16KB chunk is read from a different file location, to cause non-trivial contention on the buffer cache data structure.

This workload mimics the behavior of linear algebra kernels, for example, which perform tiled operations on large matrices, prefetching data to be processed into scratchpad memory.

We ran this experiment with a locked traversal of the buffer cache's radix trees, for comparison against our default lock-free implementation. As described in §5.2, we normally use the lock-free traversal to access each page, resorting to locking only in cases of high contention. When file data is fully resident in the buffer cache, GPUfs locks the tree rarely, as confirmed later in Table III. As a result, Figure 10 shows that the lock-free protocol performs nearly 3× better than the locked protocol across various page sizes.

6.1.4. Matrix-vector product. We run a simple single-precision matrix-vector product kernel to highlight two key benefits of the file system API: automatic data transfer pipelining and code simplification.

This test reads an input matrix and vector from files, and writes the result to an output file. We compare three implementations: one using GPUfs, one that explicitly implements double buffering to overlap the PCI data transfer and the kernel execution (CUDA naïve in Figure 11), and an optimized version of the latter (CUDA optimized). The GPUfs implementation does not call the CUDA host-side API, employing `gmmap` to read the data in the kernel, `gftruncate` to truncate the output file at the start, `gwrite` to write the output, and finally `gclose` to synchronize the data to disk. The GPUfs buffer cache is sized to 2 GB, with 2MB pages. The "naïve" version implements a simple pipeline, splitting the file into four chunks and processing each chunk independently, overlapping the file read, data transfer and kernel execution between them. Note that the chunk size depends on the size of the input, which is convenient because every GPU kernel invocation may use the same number of threads. The optimized version is similar, but the chunk size is fixed at 70 MB and there are 16 independently processed chunks. Similarly to the CUDA naïve version, each chunk is processed separately, and the file read, data transfer and kernel execution are overlapped between the chunks. Both implementations run the same code for computing the inner-product.

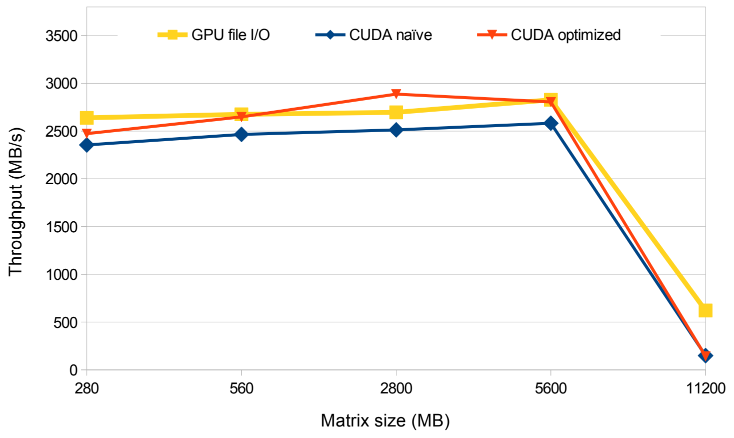

Fig. 11. Matrix-vector product for large matrices

We fix the input vector length to 128K elements, and vary the matrix size from a few megabytes up to 11GB. The largest input does not fit in the GPU's memory, and barely fits into the CPU's RAM. The GPUfs version requires no special treatment for this case, however. While this workload is entirely limited by the PCIe bus bandwidth, and for the largest inputs by the disk bandwidth, it is representative of many kernels that need to read data from disk as part of a large processing pipeline.

Figure 11 shows that the GPUfs based implementation outperforms the double-buffering implementation, achieving maximum PCI bandwidth equivalent to reading sequential files (see Figure 7). The main reason for the performance benefit is that the non-GPUfs code reads the input in large chunks (1GB each), which sometimes causes slowdowns due spurious paging of the CPU buffer cache, stalling the CPU-GPU transfer pipeline. GPUfs performs many shorter reads, due to the 2MB page size in this experiment, and the performance irregularities are smoothed by the fine-grained pipelining performed under the hood by the GPUfs' RPC daemon.

When file size exceeds available CPU buffer cache (the last data point in the graph), performance falls as the workload becomes disk bound. In this performance regime, GPUfs outperforms both CUDA versions by a factor of 4. The pinned memory allocated for large transfer buffers for the CUDA implementations competes with the CPU buffer cache, slowing it down significantly.

On the other hand, we observe no slowdown for inputs exceeding the size of the GPU buffer cache (larger than 2GB). The FIFO-like replacement policy employed by GPUfs appears to offer adequate efficiency for such streaming workloads.

6.1.5. Matrix product. Our matrix product program reads two matrices from two different files, multiplies them, and writes the result to a third file. We compare three implementations: CUDA naïve – the original CUDA SDK implementation with input,output and execution performed sequentially, CUDA pipelined – the SDK implementation but with data transfers to and from GPU memory overlapped between themselves and with the kernel execution, and GPUfs, which accesses files from the GPU code.

We evaluate two scenarios: compute-intensive and cross-application data reuse. The compute intensive case evaluates GPUfs's ability to overlap communication and computation. Thus, the inputs are sized so GPU computation time is at least as long as data transfer time. We generate inputs and outputs ranging from 1MB to 1GB by systematically doubling each dimension of the input matrices starting from 3,328x4096 by 4,096x256 to 106,492x4,096 by 4,096x2,048. We also evaluate the benefits of a cross-application GPU buffer cache when

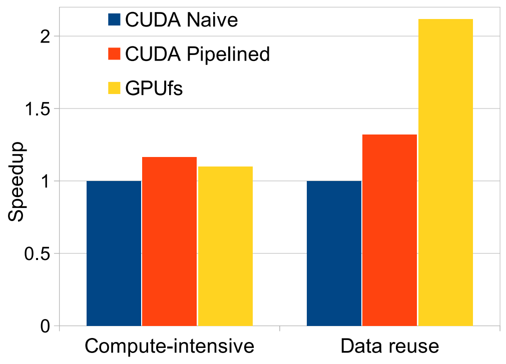

Fig. 12. Relative speedup in matrix product benchmarks

an application is invoked multiple times with different parameters but on the same data. Unfortunately, our prototype cannot maintain its buffer cache in GPU memory across different CPU applications. Consequently we emulate data reuse by comparing execution time for a program that reads its inputs from CPU buffer cache memory, and one that reads its inputs from GPUfs's buffer cache. Inputs range from 30MB to 1.5GB and the outputs from 1MB to 100MB, generated similarly to the compute-intensive benchmark.

We run these experiments on TESLA K20C NVIDIA GPU. Figure 12 shows the relative performance of GPUfs and the naïve and pipelined CUDA versions normalized by the performance of the CUDA naïve implementation. In the compute intensive scenario GPUfs reaches throughput of 180GFLOP/s, and is on average more efficient that the naïve CUDA implementation, but falls short of the hand-tuned implementation. However, GPUfs is more efficient than the pipelined version if file contents is reused — GPU-side cross-application data caching is capable to transparently improve application performance.

6.2. Application benchmarks

We now consider two more realistic I/O intensive workloads: image search, and a "grep"-like search of text files. Both applications have highly data-dependent, unbounded working sets that dynamically change during computations. Such dynamic data dependencies are challenging to handle in GPU programs without GPUfs.

6.2.1. Finding approximately matching images. The first application's input is a set of query images and several image databases containing many small images. The goal is to find which databases contain images matching the query images, where a match is defined by a threshold on a similarity metric, in our case Euclidian distance. While each image may be present in several databases, the databases must be scanned in a predefined order and only the first match output for a given query image. This process is representative of large-scale image registration tasks, e.g., when processing aerial photographs while attempting to find a matching image in a specific region first.

We can easily parallelize this problem by dynamically or statically splitting the input images between the threadblocks. The databases or/and the input set may not fit in GPU memory, however. Thus, the decision of which database to load and when must be done at runtime depending on the outcome of prior matching attempts. For example, if all the matching images are located at the beginning of the first database, the amount of data to be transferred is much lower than simply transferring all of the databases at once.

Without GPU access to the file system, the CPU must transfer the databases to the GPU first. To avoid redundant PCI transfers, the CPU is likely to split the databases into chunks,

Table III. Impact of the buffer cache size on the running time and locking behavior for the image search workload. Locked access count also includes unlocked retries.

Buffer cache size	Time (s)	Pages reclaimed	Lock-free accesses	Locked accesses
2G	53	0	1,088,838	21,516
1G	69	11,509	547,819	574,463
0.5G	99	38,317	176,758	1,351,903

small enough so that the amount of redundant data transferred would be negligible, but large enough to amortize the overheads of GPU invocation on each chunk. This heuristic is not only suboptimal and introduces additional overheads, but significantly complicates the code. Furthermore, before starting the kernel to process the next chunk, all previously matched images must be removed from the input set, requiring additional program logic to compact the input array.

GPUfs streamlines this task, making the implementation almost trivial and closely following the design for CPU code. Both the OpenMP parallel CPU and GPUfs-based versions of the program are about 130 ± 10 LOC, counting semicolons.[4] The associated CPU code for the GPU version is only a single line—the GPU kernel invocation.

In our synthetic workload, the images in the input and the databases are randomly generated. Each image is represented as a 4K-element vector. The input contains 2,016 images, amounting to 31.5MB of raw data. We use 3 database files, of sizes 383, 357 and 400 MB, containing about 25,000 images each. The images from the input are injected at random locations in the databases. We invoke the kernel with 28 threadblocks, 512 threads per threadblock.

We measure raw performance using a query set containing only images with no matches in the databases, forcing all databases to be read completely. We flush the OS page cache before each experiment. We set the GPU buffer cache size to 2GB, enough to keep all databases in GPU memory. The GPU throughput achieved is 18GFLOP/s, twice as fast as an 8-core CPU run using OpenMP.

Changing the buffer cache size. We examine the effect of the buffer cache size on program performance in Table III. Observe that as the amount of available memory decreases, the ratio between lock-free and locked accesses drops due to the paging algorithm's attempts to free pages being used. Each threadblock runs independently of the others, and may follow different execution paths, for example accessing the databases relevant to the set of input images it is processing. File access patterns among different threadblocks quickly desynchronizes, a well known effect in large-scale parallel environments, requiring careful implementation and possibly redundant work to avoid.

Finally, we evaluate our implementation's scalability by splitting the query list equally among up to 4 GPUs. We do not evaluate the diff-and-merge algorithm for write-sharing, but the system interaction with the WRAPFS-based consistency daemon is included (as is the case for all experiments presented in this section).

This set of experiments is performed with preliminary warmup in order to prefetch the data into the CPU buffer cache and highlight the scaling capabilities of the system. As confirmed in the experiments in Table IV, GPUfs shows near linear scaling with increasing GPU count because of the lightweight consistency protocol. The first run ("No match") shows the performance of the more regular workload, for which GPUfs shows ideal scaling. The second run is irregular because the number of exact matches per processor is different, and static input partitioning does not scale as well in either the GPUfs or CPU versions. All 4 GPUs together outperform a single CPU execution by about a factor of 9.

[4]We tried David Wheeler's SLOCCount but it fails to parse CUDA code.

Table IV. Approximate image matching performance. Speedup for multi-GPU runs relative to a single GPU are given in parentheses.

Input	CPUx8	#GPUs			
		1	2	3	4
No match	119s	53s	27s (2.0×)	18s (2.9×)	13s (4.1×)
Exact match	100s	40s	21s (1.9×)	14s (2.9×)	11s (3.6×)

Table V. GPU exact string match "grep -w" performance.

Input	CPUx8	GPU-GPUfs	GPU-vanilla
Linux source	6.07h	53m (6.8×)	50m (7.2×)
Shakespeare	292s	40s (7.3×)	40s (7.3×)
LOC (semicolon)	80	140 (+52)	178

The benefits of dynamic database loading becomes apparent as we relax the matching threshold, allowing searches to terminate earlier, and occasionally eliminating the need to accesses lower-priority databases altogether. Runtime decreases as expected; in the degenerate case where images always match the first entry in the first database, runtime falls by 400×—from 53 seconds to a minimum of 130ms—leaving only the costs of initialization, invocation, and matching the query list with the first database page.

6.2.2. Exact string matching in text files. The last experiment is an implementation of a constrained version of grep on a GPU. 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 it appears.

This application is conceptually similar to image matching, but with two key differences. The parallelization strategy is different because words are typically short (up to 32 symbols), so each GPU thread is assigned one word, instead of one image per threadblock in the previous case. Second, the output buffer becomes unbounded, so we need to write the output frequently to flush the per-threadblock internal buffer.

This experiment counts the frequencies of modern English words in two datasets: the works of William Shakespeare, and the Linux kernel source code. We search for a specific 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 for 524MB in total. To simplify the parsing of the dictionary file by a GPU, we reformat the dictionary to align every word on a 32 byte boundary; none of the words in the dictionary exceed that length. The list of input files is itself specified in a file.

Each threadblock opens one file at a time, then each thread searches for a subset of the dictionary that it is allocated to match. Matched words are printed out together with the file name and match count into an internal per-threadblock output buffer, which is then periodically flushed into a global output file. Various text parsing and formatted output tasks required us to implement limited GPU versions of the `sprintf, strtok, strlen, strcat` functions not normally available to GPU code.

This workload puts extremely high pressure on GPUfs because most of the files are fairly small (few kilobytes on average), leading to frequent calls to `gopen` and `gclose`. Since the progress of each threadblock depends on the actual number of matching words in its input subset, the number of concurrently open files eventually reaches the number of concurrently running threadblocks.

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 imple-

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

mented without GPUfs. Both implementations prefetch the contents of the input files into a large memory buffer first, then do not read from the file system during the matching phase.

The vanilla GPU version pre-allocates a large output buffer in the GPU memory (5GB—all remaining GPU memory), but if it overflows, the GPU kernel crashes. In general, our vanilla GPU version is more limited than the one using GPUfs because it conservatively assumes that the inputs and outputs fit in the GPU's physical memory. Large file support would substantially complicate the implementation, whereas the GPUfs-based version automatically supports arbitrarily large input files.

We present the results (no warmup) in Table V. Even for such a file-system intensive workload, a single GPU outperforms the 8-core CPU by 6.8×. The GPUfs version is only 9% slower than the vanilla GPU implementation on the Linux kernel input, but the two versions perform similarly on one large input file. The GPUfs-based code is shorter than the vanilla version if we exclude string parsing and formatted output functions (52 lines of code), which are not used in the vanilla version because they are executed on a CPU as a part of a post-processing phase.

We emphasize, however, that no serious effort has been made to optimize either the GPU or CPU version. The main point of this exercise is to highlight the utility of the file system API on GPUs, which opens up new ways to explore the computing power of these massively parallel processors.

7. RELATED WORK

To our knowledge GPUfs is the first extension of the file system abstraction to modern GPU architectures. This work touches on many areas from classic OS design and efficient lock-free synchronization to GPU architectures and programming techniques.

General-purpose GPU computing. The research community has focused considerable effort on the problem of providing a general-purpose programming interface to the specialized hardware supported by GPUs (GPGPU). GPGPU computing frameworks such as CUDA [NVIDIA], OpenCL [OpenCL], and others [Bayoumi et al. 2009; McCool and D'Amora 2006; Buck et al. 2004; Han and Abdelrahman 2009; Ueng et al. 2008] provide an expressive platform, but none provide any way for GPUs to use host OS services in general, or file system access in particular.

I/O for GPUs. GPUDirect from NVIDIA allows GPUs to access certain storage and network devices without the mediation of the host OS. This technology is exposed via proprietary, low-level hardware-specific interfaces, and does not provide higher-level abstractions, such as a file system API.

Other hardware architectures. The Cell processor [Kahle et al. 2005] pioneered the integration of parallel accelerators into the OS, allowing system calls and file accesses from its Synergistic Processor Elements (SPEs). The SPEs share the same die as the main processor, offering a high bandwidth channel with memory performance more like multicore SMPs than today's discrete GPUs. Also, we are unaware of any published work analyzing file system design tradeoffs or I/O intensive data parallel applications, the focus of this paper.

Intel's Xeon-Phi [Intel Corporation 2012] is a PCIe-attached accelerator sharing the NUMA characteristics of discrete GPUs, but built of more traditional CPU cores that can run a full OS such as Linux. To our knowledge Xeon-Phi does not expose the host's file system to software on the accelerator. We expect many aspects of GPUfs to be relevant to Xeon-Phi systems, particularly the NUMA-driven need to maximize file cache locality. Matuso et al [Matsuo et al. 2012] presented a file system layer for Xeon-phi, providing access to the host file system from the card. This design does not explore file I/O in fine-grain data-parallel workloads, however, one of the main foci of our work.

Host OS support for GPU programming. Stuart [Stuart et al. 2010] prototyped CPU-GPU communication via RPC, enabling GPU software to make host system calls. GPUfs includes such a mechanism, but focuses on coping with data parallelism and locality at design level via its GPU buffer cache, to avoid redundant data transfers and GPU-CPU interaction.

Hydra [Weinsberg et al. 2008] and PTask [Rossbach et al. 2011b] explore dataflow frameworks for GPU programming, offering host CPU software an API with which to compose GPU modules. GPUfs in contrast focuses on the complementary goal of enhancing the API available to GPU code.

Kato [Kato et al. 2012] introduces a host OS driver for GPUs that facilitates the OS-managed sharing of GPU resources, allowing different CPU processes to share GPU memory for example. We hope to leverage this complementary functionality to enable future cross-application file system support in GPUfs.

Simplifying data management in GPUs. The complexity of data management in discrete GPUs is well recognized. Gelado [Gelado et al. 2010] suggested ADSM, an asymmetric, CPU-centric shared memory [Gelado et al. 2010]. ADSM emulates a unified address space between CPUs and GPUs, alleviating management problems. Unlike GPUfs, ADSM does not support communications with a running kernel, and also introduces new accelerator-specific abstractions, which GPUfs avoids. Recently, Ji [Ji et al. 2013] introduced RSVM, a GPU region-based virtual memory mechanism for GPU memory management. Similarly to GPUfs, RSVM enables GPU kernels to access large datasets potentially exceeding GPU physical memory size by automatically swapping data in and out of GPU memory. RSVM, however, serves different purpose and does not provide access to CPU files from GPUs. It also introduces new APIs for manipulating memory regions in CPU and GPU code, whereas GPUfs strives to provide paging support largely transparently to programmers. Yet, both approaches have their benefits and their convergence is a subject of future research.

Heterogeneous and multi-core OS design. A number of researchers considered the general problem of building OSes for heterogeneous architectures. The Helios operating system [Nightingale et al. 2009] targets heterogeneous systems with multiple programmable devices. However, Helios requires the processors to expose interfaces to three basic hardware primitives: a timer, an interrupt controller, and the ability to catch exceptions. These services are currently not available on most GPUs, making Helios inapplicable to such architectures. Furthermore, Helios does not account for the specifics of massively parallel SIMD architectures, as GPUfs does.

The Barrelfish OS [Baumann et al. 2009] treats the hardware as a network of independent, heterogeneous cores communicating via RPC. Again, it is not clear if a GPU could run Barrelfish directly. Philosophically, Barrelfish argues for a ground-up OS redesign based on message passing. GPUfs takes a more pragmatic view of applications interacting through the file system, keeping the host OS largely intact.

Lock-free algorithms. Lock-free algorithms are a well known technique in parallel programming [Maurice Herlihy and Nir Shavit 2008]. Our algorithm was inspired by seqlocks [Hemminger 2002] and read-copy update (RCU) [McKenney et al. 2002]. We are unaware of any prior radix tree designs with lock-free traversal available for GPUs.

8. DISCUSSION

This article advocates for providing standard operating system services and abstractions on GPUs to facilitate their harmonious integration with the rest of the system, which we believe is a key to their broader adoption now and in the future. We implement a file system for discrete GPUs to demonstrate the feasibility and value of this goal on real GPU hardware. We focused initially on a file system layer because files are among the most popular data management abstractions. We primarily targeted discrete GPUs as they are among the most commonly deployed and most powerful high-throughput processors, and their adoption is on the rise.

We now discuss how our work relates to other efforts to improve GPU programmability, and analyze the applicability of our experience with GPUfs on discrete GPUs to emerging high-throughput processor architectures.

8.1. GPU productivity efforts

Recent developments make it significantly easier to accelerate computations on GPUs without even writing any GPU code. There are comprehensive STL-like libraries of GPU-accelerated algorithms [THR], efficient domain-specific APIs [CUD], and offloading compilers [Off] that parallelize and execute specially annotated loops on GPUs. Complex heterogeneous application pipelines may be developed using data flow programming frameworks, such as PTask [Rossbach et al. 2011b; Rossbach et al. 2011a] and StartPU [Augonnet et al. 2011].

These and other projects focus on GPU development using the traditional co-processor programming model, however, where a GPU task is a passive computational function with well-defined inputs and outputs. As a result, applications not fitting this pattern are cumbersome to implement, because GPU code cannot perform I/O calls directly. Thus, all running GPU threads need to stop in order to execute the I/O call on a CPU, or one must develop application-specific CPU code to handle a GPU's I/O requests asynchronously.

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.

8.2. Hardware trends

Discrete GPUs are not only getting faster with each generation, but are also becoming more programmable and flexible. Newer architectures enable GPUs to execute functions that previously required CPU-side code. For example, NVIDIA Kepler GPUs support *nested parallelism* in hardware, allowing invocation of new GPU kernels from GPU code without stopping the running kernel first. 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 [Keckler et al. 2011] are expected to enable more efficient sequential processing. These trends reemphasize 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 to 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. We believe that support for accessing host resources by Xeon-Phi programs will soon be available, but its exact design and performance characteristics remain to be seen. We expect many aspects of GPUfs to be relevant to Xeon-Phi systems, particularly the NUMA-driven need to maximize file cache locality.

Emerging hybrid processors combine both a GPU and a CPU on the same die. The GPU and CPU share physical memory, but have separate address spaces. Newer architectures, e.g., AMD's just announced Kaveri processor, are expected to add support for sharing a single virtual memory address space [hUM]. Communicating through shared memory makes CPU-GPU data transfers unnecessary, providing a much faster and easier way to exchange data between the processors. However, the programming models for hybrid and discrete GPUs are largely the same, which means that GPU programs executing on hybrid GPUs cannot directly access host resources. Thus, we believe that portable and familiar operating system interfaces would be useful for both discrete and hybrid GPUs. Of course, the availability of shared memory between the CPU and the GPU opens up many optimization opportunities for how the GPUs and CPUs interact. Regardless of where the engineering details end up, we expect GPUfs's design lessons to remain relevant for massively parallel API calls.

REFERENCES

AMD and HSA: A New Era of Vivid Digital Experiences. http://www.amd.com/us/products/technologies/hsa/Pages/hsa.aspx.

GPU-accelerated high performance libraries. https://developer.nvidia.com/gpu-accelerated-libraries.

GPU Regexp. http://www.hpl.hp.com/israel/research/gpu_regex.html.

Network File System (NFS) version 4 protocol. http://www.ietf.org/rfc/rfc3530.txt.

NVIDIA Thrust library. https://developer.nvidia.com/thrust.

PGI accelerator compilers with OpenACC directives. www.pgroup.com/resources/accel.htm.

Popular GPU-accelerated applications. http://www.nvidia.com/object/gpu-applications.html.

Cristiana Amza, Alan L Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. 1996. Treadmarks: Shared memory computing on networks of workstations. *Computer* 29, 2 (1996), 18–28.

Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* 23 (Feb. 2011), 187–198. Issue 2. DOI:http://dx.doi.org/10.1002/cpe.1631

Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles.* New York, NY, USA, 29–44. DOI:http://dx.doi.org/10.1145/1629575.1629579

Amr Bayoumi, Michael Chu, Yasser Hanafy, Patricia Harrell, and Gamal Refai-Ahmed. 2009. Scientific and Engineering Computing Using ATI Stream Technology. *Computing in Science and Engineering* 11, 6 (2009), 92–97. DOI:http://dx.doi.org/10.1109/MCSE.2009.204

Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. 2004. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics* 23, 3 (Aug. 2004).

Wolfgang Effelsberg and Theo Haerder. 1984. Principles of database buffer management. *ACM Transactions on Database Systems* 9, 4 (Dec. 1984), 560–595. DOI:http://dx.doi.org/10.1145/1994.2022

Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. 2010. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems.* New York, NY, USA, 347–358.

Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: a GPU-accelerated software router. *SIGCOMM Comput. Commun. Rev.* 40 (August 2010), 195–206. Issue 4. DOI:http://dx.doi.org/10.1145/1851275.1851207

Tianyi David Han and Tarek S. Abdelrahman. 2009. *hi*CUDA: a high-level directive-based language for GPU programming. In *Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2).*

Stephen Hemminger. 2002. Fast reader/writer lock for gettimeofday 2.5.30. (2002). http://lwn.net/Articles/7388/.

John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. 1988. Scale and performance in a distributed file system. *ACM Transactions on Computing Systems* 6, 1 (February 1988).

Intel Corporation 2012. *Intel Xeon-Phi Coprocessor: System Software Developers Guide.* http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-system-software-developers-guide.html.

Feng Ji, Heshan Lin, and Xiaosong Ma. 2013. RSVM: a region-based software virtual memory for GPU. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*. IEEE, 269–278.

J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. 2005. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development* 49 (July 2005), 589–604. Issue 4/5.

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

Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the future of parallel computing. *Micro, IEEE* 31, 5 (2011), 7–17.

Yuki Matsuo, Taku Shimosawa, and Yutaka Ishikawa. 2012. A file I/O system for many-core based clusters. In *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers*. New York, NY, USA, Article 3, 8 pages. DOI:http://dx.doi.org/10.1145/2318916.2318920

Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann.

Michael D. McCool and Bruce D'Amora. 2006. Programming using RapidMind on the Cell BE. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, New York, NY, USA, 222. DOI:http://dx.doi.org/10.1145/1188455.1188686

Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. 2002. Read-Copy Update. In *Ottawa Linux Symposium*. 338–367.

Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. 2009. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP '09: Proceedings of the 22nd ACM symposium on Operating systems principles*.

NVIDIA. *NVIDIA CUDA 4.2 Developer Guide*. http://developer.nvidia.com/category/zone/cuda-zone.

NVIDIA 2013. *NVIDIA CUDA Programming Guide*. NVIDIA. http://docs.nvidia.com

Khronos Group: OpenCL. *The open standard for parallel programming of heterogeneous systems*. http://www.khronos.org/opencl.

Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011b. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 233–248.

Christopher J. Rossbach, Jon Currey, and Emmett Witchel. 2011a. Operating Systems must support GPU abstractions. In *Hot Topics in Operating Systems (HotOS '11)*.

Livio Soares and Michael Stumm. 2010. FlexSC: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. Berkeley, CA, USA, 1–8. http://dl.acm.org/citation.cfm?id=1924943.1924946

Jeff A. Stuart, Michael Cox, and John D. Owens. 2010. GPU-to-CPU Callbacks. In *Third Workshop on UnConventional High Performance Computing (UCHPC 2010)*.

Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. 1996. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd annual international symposium on Computer architecture (ISCA '96)*. ACM, New York, NY, USA, 191–202. DOI:http://dx.doi.org/10.1145/232973.232993

Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen-Mei W. Hwu. 2008. CUDA-Lite: Reducing GPU Programming Complexity. In *LCPC 2008, 21th Annual Workshop on Languages and Compilers for Parallel Computing*.

Yaron Weinsberg, Danny Dolev, Tal Anker, Muli Ben-Yehuda, and Pete Wyckoff. 2008. Tapping into the fountain of CPUs: on operating system support for programmable devices. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*.

E. Zadok and I. Bădulescu. 1999. A Stackable File System Interface for Linux. In *LinuxExpo Conference Proceedings*. Raleigh, NC, 141–151.