GPUfs: Integrating a File System with GPUs

Mark Silberstein
University of Texas at Austin
marks@cs.utexas.edu

Bryan Ford
Yale University
bryan.ford@yale.edu

Idit Keidar
Technion – Israel Institute of Technology
idish@ee.technion.ac.il

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

As GPU hardware becomes increasingly general-purpose, it is quickly outgrowing the traditional, constrained GPU-as-coprocessor programming model. To make GPUs easier to program and improve their integration with operating systems, we propose making the host's file system directly accessible to GPU code. GPUfs provides a POSIX-like API for GPU programs, 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

Keywords Operating Systems Design, GPGPUs, File Systems, accelerators

1. Introduction

The file system is a successful, proven operating system abstraction, which benefits developer productivity by decoupling the application's logical view of storage from low-level details of the location and type of devices on which data physically resides. Modern file systems have buffer caches enabling the system to optimize data access locality across cooperating processes or modules, e.g., by keeping frequently-accessed data or intermediate results in memory to minimize the cost of accesses to slow storage devices.

Unlike CPU applications, programs running on graphical processing units (GPUs) currently have no direct access to files on the host OS file system. Although the power, functionality, and utility of today's GPUs now extend far beyond graphics processing, the coprocessor-style GPU programming model still requires developers to manage movement of data explicitly between its "home" in the CPU's main memory and the GPU's local memory. Managing data transfers between CPU and GPU increases the design complexity and code size of even simple GPU programs requiring file access. While programmers can explicitly optimize data movement, this performance is often not portable to new generations of hardware. Over time, application code to transfer and reuse recently computed data becomes entwined with program logic, making 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. GPUs are quickly evolving toward general high-performance processors useful for a wide variety of massively parallel, throughput-oriented tasks, and we believe GPU programming should reap the same benefits from the file system abstraction enjoyed by CPU programmers for decades.

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.

As a simple example, consider multiplying a matrix by a vector. GPUs excel at such computational tasks, but most GPU programs will assume that the matrix fits in GPU memory. If the input matrix becomes larger than GPU memory, the program must be invasively modified and its complexity increases sharply—for example, the input must be split into chunks, each chunk processed separately, and data transfers must be overlapped with computation for good performance. File systems traditionally excel at insulating the developer from such low-level data movement details. File systems also excel as a communication substrate for composing different programs. Currently, GPUs are more often used for stand-alone monolithic applications, because the complexity of integrating a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.
Copyright © 2013 ACM 978-1-4503-1870-9/13/03…$15.00

GPU program into a complex processing pipeline is too high. Data movement is a major source of this complexity.

With GPU hardware changing so rapidly, a key design challenge for GPUfs is to focus on properties likely to be essential to the performance of future as well as current GPUs. Some properties of current GPUs, such as particular memory consistency models, may continue changing rapidly and unpredictably. We believe, however, that two key characteristics—data parallelism and dependence on access locality—will persist as GPU architectures evolve, and GPUfs must address these concerns in order to succeed.

GPUs are designed to optimize for massive data parallelism, by sharing a limited set of "control plane" logic—for instruction fetch, memory management, etc.—among far more numerous "data plane" resources such as vector ALUs. As a result, GPUs are efficient when thousands of lightweight threads run similar or identical code, with little control-flow variation. The traditional file system API was not built with such an execution environment in mind. In GPUfs, therefore, both the API semantics and the file system implementation must be designed to support such massive parallelism, efficiently allowing thousands of GPU hardware threads to invoke `open`, `close`, `read`, or `write` calls simultaneously.

Second, memory locality is vital for performance, due to the variety in memory system speeds and interconnect topologies. Our work currently focuses on discrete GPUs, which today provide higher performance than GPUs integrated with CPUs on the same die. Discrete GPUs have high-bandwidth DRAM, but connect to the host via a peripheral interconnect bus, typically PCIe. These buses are high latency and low bandwidth relative to DRAM buses, effectively 6GB/s for PCIe 2.0. Future multi-GPU systems with both integrated and discrete GPUs will further increase variance in hardware speeds and topologies. The only way for GPUfs to perform well in such systems is with comprehensive OS management of the memory system. The OS must have a global policy to manage data placement and data reuse across CPU and GPU memories based on dynamic access patterns. GPUfs's buffer cache is distributed across all system memories to enable such policies.

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 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 from 5% to $4\times$ faster than the manual double-buffering typical in current GPU code, and is about $2\times$ smaller in code size. 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 data parallel efficiency.
3. A proof-of-concept implementation of GPUfs on NVIDIA FERMI GPUs [21], 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. We then explain and justify the design choices that we made while building GPUfs, followed by the implementation details of GPUfs on NVIDIA FERMI GPUs. We evaluate the GPUfs prototype implementation in Section 5, and conclude with related work.

2. GPU architecture overview

This section provides a brief, simplified overview of the GPU software/hardware model, highlighting the properties that are particularly relevant to GPUfs. We use NVIDIA CUDA terminology because we implement GPUfs on NVIDIA cards; for more details we refer the reader to the CUDA reference [20].

GPUs are multicore processors. Each core, called *multiprocessor* or *MP*, features a wide SIMD vector unit, which a hardware scheduler multiplexes between multiple execution contexts.

A GPU's basic sequential unit of execution is a *thread*. GPU hardware groups a number of threads (32 in NVIDIA GPUs) into *warps*, and executes all threads in a warp concurrently in lockstep on a single hardware vector unit. Each MP multiplexes many warps onto the same SIMD unit to maximize hardware utilization, executing one warp while another is blocked on a memory access for example. Several warps (up to 32) form a *threadblock*. Threads in a threadblock are always executed on a single MP. Multiple threadblocks form a single complete GPU program, often termed a *kernel* (unrelated to an OS kernel).

A CPU application using the GPU enqueues all threadblocks comprising a kernel into a single global hardware queue. The hardware scheduler dispatches threadblocks onto MPs while assigning unique IDs to each threadblock and each thread in the block. The scheduler strives to maximize the number of warps concurrently scheduled on an MP without exceeding MP hardware resources such as available registers. Given sufficient hardware resources the scheduler can schedule multiple independent kernels at once.

Challenges for GPUfs. There are two characteristics of the GPU execution model that are particularly important in designing system abstractions such as a file system API on a GPU. First, once invoked, warps run to completion *without preemption*. This implies, for example, that using spinlocks to synchronize between running threads in the same kernel may lead to a deadlock. Second, the hardware schedules threadblocks for execution in *a non-deterministic order*, driven solely by the goal of maximizing hardware utilization. This behavior creates challenges in implementing reference-count based parallel versions of `open` and `close` file calls, for example, as described later (§ 4).

3. Design

This section outlines the GPUfs API and file system semantics, focusing on the similarities and differences from the POSIX API, and the properties of GPUs that motivate these design choices. We believe that our design reflects several key properties of data parallel architectures that will apply to future as well as current GPUs and hybrid processors. This section focuses on the high-level aspects of the design and API that are visible to applications using GPUfs, deferring lower-level implementation considerations largely invisible to applications to § 4.

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

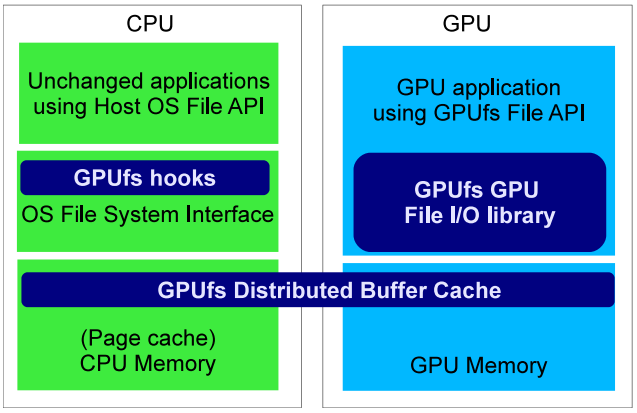

Figure 1. GPUfs architecture

3.1 GPUfs design principles

Two key principles underly the design of GPUfs and the ways in which it deviates from traditional POSIX semantics: making common API operations efficient when executed in data parallel fashion, and choosing consistency and data movement semantics that maximizes file data access locality and minimizes expensive global transfers between GPU and host memory.

API design for structured data parallelism. As we summarized in § 2, GPU hardware offers parallelism at multiple granularities, which combine to achieve high throughput. At finer granularities, the hardware achieves efficiency by sharing control logic across all threads comprising a warp. Processing is efficient when these threads follow the same code paths in lockstep, but highly inefficient if the threads follow divergent paths: *all* the threads in a warp must explore *all* possible divergent paths together, merely masking instructions applicable only to some threads.

Because of these hardware characteristics, a key semantic design decision for any GPU-accessible library or system API is the granularity at which API calls are to be invoked. Operations affecting globally shared file system state, such as `open` and `close` calls, involve control-flow heavy operations and require serialization. Even basic `read` and `write` API calls require updates to the file system's buffer cache data structures. If GPUfs allowed each application to invoke these operations at thread granularity—e.g., each thread opening different files or reading different blocks—these threads would quickly encounter divergent control paths within GPUfs, entailing hardware serialization and inefficiency. Moreover, hardware provides the highest memory throughput when the accesses of all threads in a warp are aligned and can be coalesced into one memory transaction.

For these reasons, GPUfs follows common GPU programming practices by requiring all threads in a warp to cooperate to perform the same operation, and requires applications to invoke the file system API at warp—rather than thread—granularity.

Thus, all application threads in a warp must invoke the same GPUfs call, with the same arguments, at the same point in application code. These lockstep calls together comprise one logical GPUfs operation. For example, GPUfs does not allow an application to open one file *per thread* in parallel, but only one file *per warp*. On the other hand, this warp-granularity API allows the GPUfs *implementation* to parallelize the handling of API calls across threads in the invoking warp—parallelizing file table search operations or data movement, for example. Our prototype currently makes use of this capability only in a few performance-critical cases, highlighting this principle by accelerating memory transfers between the user and system buffers in read/write calls.

As detailed below, considerations of data parallelism also motivated several other design decisions we made in GPUfs: to minimize per-open file state by eliminating seek pointers, decouple the synchronizing side-effects traditionally bound into API calls such as `close`, and constrain `mmap` semantics to avoid the need for complex memory management on critical data parallel paths.

Locality-optimized file consistency model. GPU memory systems have pronounced NUMA characteristics, offering far more bandwidth—over 30× in current systems—and lower latencies to access local GPU memory than to main CPU memory or to that of other discrete GPUs. Performance therefore critically depends on minimizing file data movement between the GPU and CPU, or between GPUs. To enable each GPU to access locally cached file data as independently as possible, GPUfs implements a weak consistency model similar to the *private workspace* model in Determinator [1], and to distributed file systems such as LOCUS [26] or AFS [10]. Once a file page is accessed and cached on a GPU, its threads can read and write to that page locally without further communication with the host—even if the host and/or other GPUs may concurrently read and/or modify that file. GPUfs guarantees that local file modifications propagate to main CPU memory only when the application explicitly *synchronizes* the file or individual pages with backing store, thereby persisting its content. These modifications become visible to other GPUs when they re-open the file.

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 is assigned to produce. Workspace consistency allows multiple writers without causing memory page thrashing between different GPUs, as a single-writer MESI protocol would exhibit. An important challenge, however, is that GPUfs must be able to handle false sharing of buffer cache pages among different GPUs. As a result, it has to determine which specific portions of a given page were modified on a given GPU when propagating those modifications to the host, to avoid accidentally reverting other portions of the same page that have been modified concurrently by other GPUs. In general, for files opened for *writing*, GPUfs must maintain *two* copies of each cached block per GPU: a working copy to which local writes are performed, and a *pristine* copy preserved when the page is first read. GPUfs "diffs" the working and the pristine copies at the next synchronization point to determine which bytes have been modified and should be written back.

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 both making and storing two copies of file blocks 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, and instead implicitly assumes the pristine copy of any file block is all zeros—even if the host or some other GPUs may in fact have already written to parts of that page in the underlying file. As a result, when the GPU propagates locally written pages back to the host, determining which bytes have been modified locally reduces to a trivial "diff against zeros." These semantics imply that one GPU's threads will typically never observe writes from other processors while the file is opened for writing, and that multiple GPUs' concurrent writes 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.

API	Explanation
gread/gwrite	Reads and writes always supply explicit file offsets, to avoid the file seek pointer becoming a sequential bottleneck.
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. The precise semantics are further discussed in the text.
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 the text.
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.

Table 1. GPUfs API, and discussion of relaxed file system semantics for GPUfs.

3.2 GPUfs API

Guided by the above principles, we now explore their implications on specific parts of the GPUfs API. GPUfs attempts to preserve the POSIX API's familiar semantics when practical, while diverging as needed for efficiency in GPU environments. Table 1 summarizes the API and its deviations from from POSIX semantics. We prepend a 'g' to GPUfs' API function names to emphasize that their semantics deviates from strict POSIX.

Open and file descriptors. Traditionally, if several POSIX threads concurrently open the same file, each thread obtains a fresh file descriptor in a process-global file table, each descriptor containing a separate seek pointer and other file-open state. For GPU kernels we expect it to be commonplace to open the same file in parallel across hundreds of concurrently running warps, for example when each warp is assigned a given chunk of a given file. Preserving POSIX semantics would in such cases require these gopen calls to coordinate the efficient simultaneous allocation and initialization of large "batches" of file descriptors at once, a complex and likely inefficient file descriptor management task. In GPUfs, therefore, "file descriptors" do not represent individual *file opens* but merely correspond directly to *files*, so that all GPU threads opening the same file obtain a single shared file descriptor. GPUfs forwards the first gopen call on a given file to the CPU to open the file on the underlying host file system. GPUfs then reference counts these open files, so a gopen on an already-open file just increments the file's open count without requiring CPU communication.

Besides the standard open flags, gopen introduces two new flags enabling useful performance optimizations.

- O_GWRONCE: Creates a new write-only file, in which the application will write each byte *at most once*. If data is overwritten, partial updates may occur. This flag eliminates fetching of file content from the CPU before writing, as described above in §3.1.
- O_NOSYNC: Creates a temporary file to be used only by the GPU opening it. GPUfs never writes the file's data to disk on close, and never writes it at all except to reclaim GPU buffer cache space.

Read and write. Because GPUfs dispenses with most per-open state, its file descriptors have no seek pointers. As a result, gread and gwrite correspond to POSIX's pread and pwrite—taking file offsets explicitly as arguments—instead of the traditional streaming read and write. This convention matches common practice in parallel workloads anyway [14], and application threads can maintain their own explicit seek pointers if required, as we demonstrate in our experiments (§5.2.2).

Close and synchronization. POSIX semantics require the contents of a file to be synchronized to stable storage (e.g., disk) after each close. In the common-case sequence of gopen, gwrite, gclose, executed by many GPU threads, POSIX close semantics cause many costly write-backs due to the nondeterministic GPU scheduler. Even if we reference count files, it is common for several threadblocks to open, write, then close the file, sending its reference count temporarily to zero before other threadblocks are scheduled that open the file again. In our experience this situation can be common, and synchronizing files each time the reference count drops to zero results in many unnecessary writes.

GPUfs therefore decouples the file "close" and "synchronize" operations. In particular, gclose does not propagate locally written data back to the CPU, or to other GPUs, until the application explicitly synchronizes file data, by calling gfsync to synchronize either an entire file or a specific offset range.

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 user-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 hundreds or thousands of threads at once. Thus, any translation change has global impact, likely requiring synchronization too expensive for fine-grained use within individual threads.

GPUfs therefore offers a more relaxed alternative to mmap, permitting more efficient implementation 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 application not to modify that memory. Improper updates to such "quasi-read-only" pages are never propagated back to the host CPU, so GPUfs ensures host file system integrity despite less stringent page-level access enforcement on data resident in local GPU 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.

3.3 Buffer cache

An essential component of a file system layer is a buffer cache. In CPUs, the buffer cache minimizes disk accesses, which can be a thousand times slower than memory. The GPU page cache extends this principle to GPU file accesses, caching file data in fast local GPU memory to minimize transfers across the relatively slow and bandwidth-constrained peripheral interconnect.

The role of the buffer cache extends beyond simple caching. As on a CPU, a GPU buffer cache enables further performance optimizations such as read-ahead, data transfer scheduling, and asynchronous writes. 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.

Due to limitations in the software interfaces available to today's GPU hardware, GPUfs currently implements a private GPU buffer cache for each host CPU process: buffer cache pages are not shared across host applications, as in the OS-maintained buffer cache on the host CPU. Programmable memory protection interfaces on future GPUs could eliminate this limitation, enabling a true cross-user, cross-application GPU buffer cache.[1] On the other hand, multiple kernels launched by the same process can share data via the buffer cache, and we use that feature in our experiments (§5.1.3).

Replacement policies. A GPU file system allows the OS to coordinate file system replacement policies across all hardware in the system. For example, if the GPU is idle, the OS could use GPU memory as a staging area for data before writing it to disk. As GPU computations evolve to become part of heterogeneous processing pipelines, OS management of file system data will enable significant performance optimizations.

Whether standard LRU replacement policies for CPUs will be appropriate for the GPU buffer cache is not yet clear. Typical file access patterns in GPU applications remain to be seen, but we already observe accesses to be fairly chaotic even in workloads with logically sequential accesses, due to the non-deterministic scheduling of threadblocks in the GPU execution model.

Failure semantics. GPUfs has failure semantics similar to the CPU page cache: on failure, file updates not yet committed to disk may be lost. From the application's perspective, successful completion of `gfsync` or `gmsync` ensures that data has been written to the host page cache. The API also allows forcing writes to stable storage, equivalent to `fsync` or `msync` on CPUs.

Unfortunately, GPU failures are more frequent and have severe implications. In existing systems, a GPU program failure—such as an invalid memory access or assertion failure [20]—may require restarting the GPU card, thus losing the GPU's entire memory state. As GPUs continue to become more general-purpose, we expect GPU hardware to gain more resilience to such software failures.

3.4 Resource contention with GPU programs

Operating systems are known to compete with user programs for hardware resources such as caches [23], and are often blamed for decreased performance in high performance computing environments. GPUfs is less intrusive than a complete OS because it has no active, continuously running components. It necessarily adds some overheads, however, in the form of memory consumption, increased program instruction footprint, and the use of GPU hardware registers. We expect the relative effect of these overheads on performance to decrease with future hardware generations, which will provide larger memory, larger register files, and larger instruction caches. And despite these additional costs, we find GPUfs to have good performance in useful application scenarios (§5).

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 the "pay-as-you-go" design principle to be discussed further in §4.2

[1] A cross-application buffer cache could potentially be implemented already using the new IPC feature in CUDA 5.0, but this interface still lacks the programmable memory protection that would be necessary to protect a shared GPUfs buffer cache from errant host processes or GPU kernels.

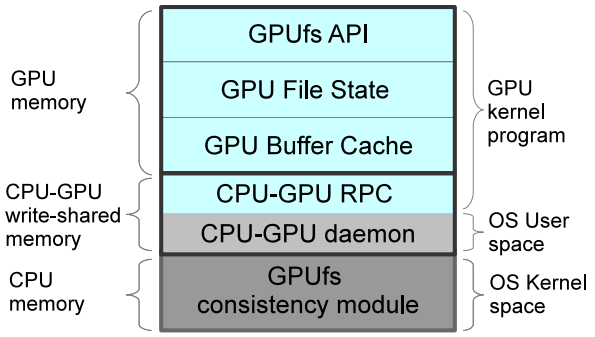

Figure 2. Main GPUfs software layers and their location in the software stack and physical memory.

4. Implementation

This section describes our GPUfs prototype for NVIDIA FERMI 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 the application's CUDA context.

Figure 2 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 4.3.

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 §3.

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

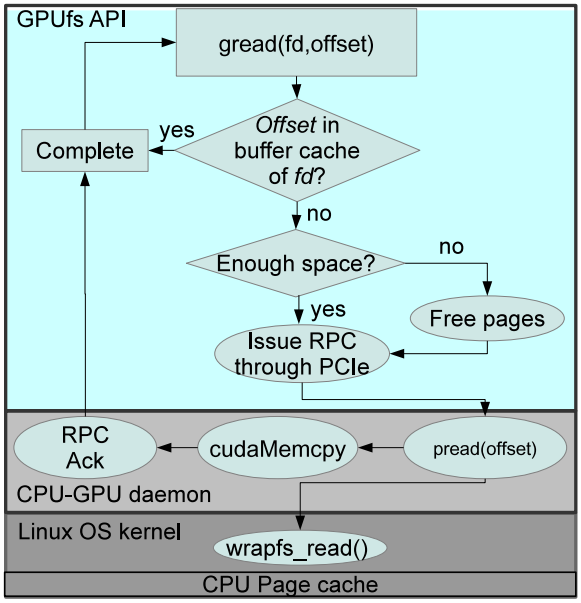

Figure 3. Functional diagram of a call to `gread`. Color scheme is the same as Figure 2.

4.1 File system operations

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 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 (§3.2). 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.

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 3 shows a functional summary of `gread`'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 `gwrite` 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.

4.2 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 §5. 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.

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.

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.

Buffer cache management. CPUs handle buffer cache management tasks in a daemon thread, keeping costly activities such as flushing dirty pages out of an application's performance path. GPUs unfortunately have a scheduling-related weakness that makes daemon threads inefficient, affecting the performance of all GPU applications including those not using GPUfs. GPU threadblocks are non-preemptive, so a daemon would require its own 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, whereas the latter breaks the correctness of GPU applications that rely on the availability of a specific number of threadblocks for execution. Alternatively, offloading GPU cache management to a CPU daemon is impractical on existing hardware due to the lack of atomic operations over a PCIe bus, as explained later in §4.3.

Organizing the file system to avoid asynchronous GPU-initiated activity has important design consequences, such as the need to optimize the paging algorithm for speed. GPUfs performs paging as a part of regular file operations such as `gwrite`, with the GPUfs code hijacking the calling thread to perform paging. To keep paging fast, GPUfs does not use replacement policies that perform a variable amount of work, such as the clock algorithm [5].

GPUfs implements a FIFO-like policy by tracking allocation of last-level radix tree nodes. Newly allocated nodes are placed at the head of a doubly-linked list. When a thread needs to evict pages back to the CPU, 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 clean, so they can be reclaimed without GPU-CPU communication.[2] 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 [9]. 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.

4.3 GPU-CPU Remote Procedure Call

GPUfs implements an RPC protocol to coordinate data transfers between the CPU and GPU. The GPU serves as a client that issues requests to a file server running on the host CPU. This GPU-as-client design contrasts with the traditional GPU-as-coprocessor programming model, reversing the roles of CPU and GPU.

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 OpenCL [7] standard, and consequently AMD's discrete GPUs, currently do not support these features. Hybrid AMD GPUs are adding support for the first feature, but are not yet available. 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. 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.

[2] For clarity we omit some technical details on handling dirty files on close.

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, stateless 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.

4.4 File consistency management

The current prototype implements the locality-optimized file consistency model described in Section 3, though currently only for the common cases of files opened in either read-only (`O_RDONLY`) or write-once mode (`O_GWRONCE`, see §3.2). 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 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 §4.2).

GPUfs uses WRAPFS [28], 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.

4.5 Implementation limitations

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. 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 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.

5. 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 run 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.

5.1 Microbenchmarks

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

5.1.1 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 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.

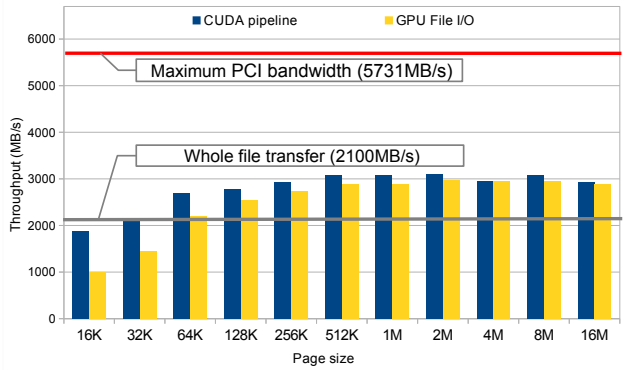

Figure 4. Sequential read performance as a function of the page size. The red line is the maximum achievable PCI bandwidth on this hardware configuration. Higher is better.

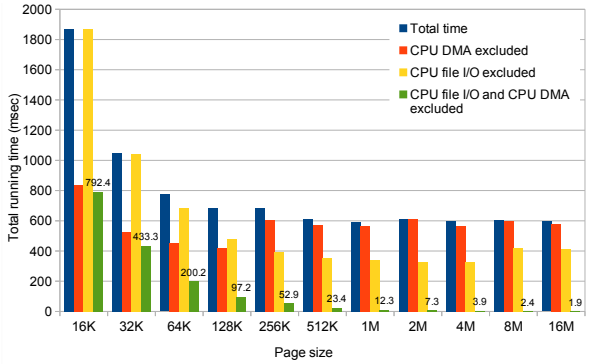

Figure 5. Contribution of different factors to the file I/O performance as a function of the page size. Lower is better.

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 4 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. The CUDA pipeline implementation appears to achieve the maximum possible file-to-GPU transfer performance on this machine.

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 5 breaks down the timing of the microbenchmark, by eliminating PCI data transfer time while leaving only the RPC

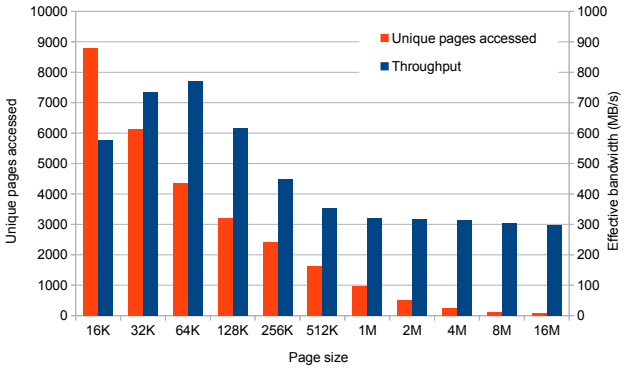

Figure 6. Random read/write performance as a function of page size. Higher is better.

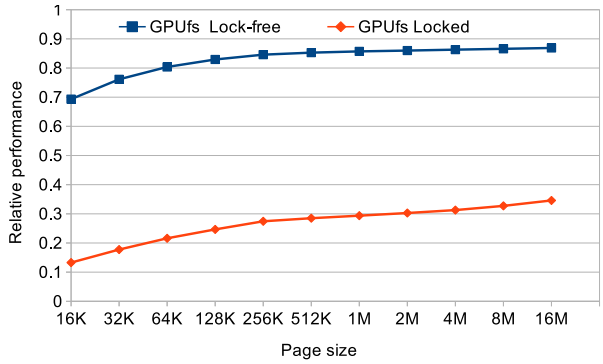

Figure 7. Buffer cache access performance with and without lock-free radix tree traversal, normalized by the raw memory access time.

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 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.

5.1.2 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 6 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 4), 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.

5.1.3 Buffer cache access performance

As the "GPUfs-lock-free" case in Figure 7 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 §4.2, 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 2. As a result, Figure 7 shows that the lock-free protocol performs nearly $3\times$ better than the locked protocol across various page sizes.

5.1.4 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 8), 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 `gfsync` 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

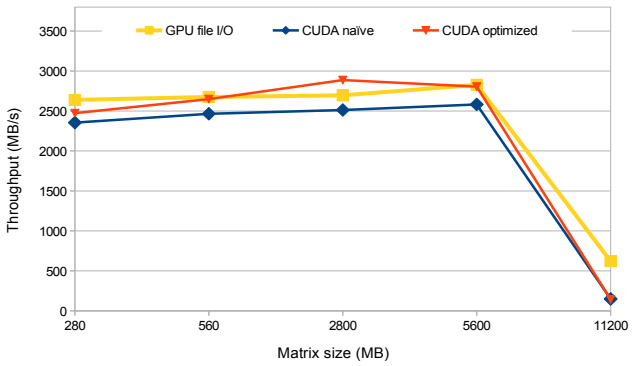

Figure 8. Matrix-vector product for large matrices

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.

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 8 shows that the GPUfs based implementation outperforms the double-buffering implementation, achieving maximum PCI bandwidth equivalent to reading sequential files (see Figure 4). 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 CPU's 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.

5.2 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.

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

Table 2. 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.

5.2.1 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, 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.[3] 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 2. Observe that as the amount of available memory decreases, the ratio between lock-free and locked accesses drops due to the paging algorithm's

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

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

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

Table 4. GPU exact string match "grep -w" performance.

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 3, 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.

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.

5.2.2 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[4], within the complete works of Shakespeare as a single 6MB text file[5], 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 implemented 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 4. 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.

6. 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 [20], OpenCL [7], and others [3, 4, 8, 17, 25] provide

[4] http://www.mieliestronk.com/wordlist.html

[5] http://www.gutenberg.org/ebooks/100

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 [12] 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 [11] 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 [15] 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 [24] 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 [27] and PTask [22] 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 [13] 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 [6] suggested ADSM, an asymmetric, CPU-centric shared memory [6]. 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.

Heterogeneous and multi-core OS design. A number of researchers considered the general problem of building OSes for heterogeneous architectures. The Helios operating system [19] 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 [2] 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 [16]. Our algorithm was inspired by seqlocks [9] and read-copy update (RCU) [18]. We are unaware of any prior radix tree designs with lock-free traversal available for GPUs.

7. Conclusions

This paper describes the design and implementation of GPUfs, a file system API and implementation allowing data parallel GPU software to access host files directly. GPUfs extends the constrained GPU-as-coprocessor programming model, turning GPUs into first-class computing devices with full file I/O support. GPUfs exploits fine-grained parallelism and memory locality to offer a familiar and efficient file system for GPU programs, and simplifies GPU programming by hiding the complexities of low-level data movement between GPUs and CPUs and among GPUs. Our prototype shows that file system access for GPU kernels enables good performance and ease of development for applications that have not typically been considered suitable for GPU processing. GPUfs achieves good performance on today's architecture, and is likely to benefit from future hardware architecture advances.

Acknowledgments

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.

References

[1] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, October 2010.

[2] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles*, pages 29–44, New York, NY, USA, 2009.

[3] Amr Bayoumi, Michael Chu, Yasser Hanafy, Patricia Harrell, and Gamal Refai-Ahmed. Scientific and Engineering Computing Using ATI Stream Technology. *Computing in Science and Engineering*, 11(6):92–97, 2009.

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

[5] Wolfgang Effelsberg and Theo Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4):560–595, December 1984.

[6] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. 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*, pages 347–358, New York, NY, USA, 2010.

[7] Khronos Group. *OpenCL - the open standard for parallel programming of heterogeneous systems*. `http://www.khronos.org/opencl`.

[8] Tianyi David Han and Tarek S. Abdelrahman. *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)*, March 2009.

[9] Stephen Hemminger. fast reader/writer lock for gettimeofday 2.5.30, 2002. `http://lwn.net/Articles/7388/`.

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

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

[12] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49:589–604, July 2005.

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

[14] Walt Ligon and Rob Ross. Parallel i/o and the parallel virtual file system. In William Gropp, Ewing Lusk, and Thomas Sterling, editors, *Beowulf Cluster Computing with Linux*, pages 493–535. MIT Press, 2003.

[15] Yuki Matsuo, Taku Shimosawa, and Yutaka Ishikawa. A file I/O system for many-core based clusters. In *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers*, pages 3:1–3:8, New York, NY, USA, 2012.

[16] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

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

[18] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read-copy update. In *Ottawa Linux Symposium*, pages 338–367, June 2002.

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

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

[21] NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2011. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

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

[23] Livio Soares and Michael Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–8, Berkeley, CA, USA, 2010.

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

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

[26] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings of the ninth ACM symposium on Operating systems principles*, pages 49–70, New York, NY, USA, 1983.

[27] Yaron Weinsberg, Danny Dolev, Tal Anker, Muli Ben-Yehuda, and Pete Wyckoff. 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)*, March 2008.

[28] E. Zadok and I. Bădulescu. A stackable file system interface for Linux. In *LinuxExpo Conference Proceedings*, pages 141–151, Raleigh, NC, May 1999.