GPUstore

Harnessing GPU Computing for Storage Systems in the OS Kernel

Weibin Sun *
University of Utah
wbsun@cs.utah.edu

Robert Ricci
University of Utah
ricci@cs.utah.edu

Matthew L. Curry
Sandia National Laboratories †
mlcurry@sandia.gov

Abstract

Many storage systems include computationally expensive components. Examples include encryption for confidentiality, checksums for integrity, and error correcting codes for reliability. As storage systems become larger, faster, and serve more clients, the demands placed on their computational components increase and they can become performance bottlenecks. Many of these computational tasks are inherently parallel: they can be run independently for different blocks, files, or I/O requests. This makes them a good fit for GPUs, a class of processor designed specifically for high degrees of parallelism: consumer-grade GPUs have hundreds of cores and are capable of running hundreds of thousands of concurrent threads. However, because the software frameworks built for GPUs have been designed primarily for the long-running, data-intensive workloads seen in graphics or high-performance computing, they are not well-suited to the needs of storage systems.

In this paper, we present `GPUstore`, a framework for integrating GPU computing into storage systems. `GPUstore` is designed to match the programming models already used these systems. We have prototyped `GPUstore` in the Linux kernel and demonstrate its use in three storage subsystems: file-level encryption, block-level encryption, and RAID 6 data recovery. Comparing our GPU-accelerated drivers with the mature CPU-based implementations in the Linux kernel, we show performance improvements of up to an order of magnitude.

* Weibin Sun's work is supported by an award from the NVIDIA Gradate Fellowship program.

† Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

SYSTOR '12 June 4–6, 2012, Haifa, Israel.

1. Introduction

Many computational tasks in storage systems are inherently parallel. For example, in an encrypted filesystem, decryption of separate blocks can occur concurrently. Similarly, RAID systems can compute parity independently for separate stripes or blocks. Hash computation for integrity protection, content-addressable lookup, or duplicate detection is likewise parallel across blocks. When busy storage servers receive concurrent streams of I/O requests, read-heavy workloads permit these streams to be processed independently. Consequently, as storage systems grow larger, get faster, and serve more clients, parallel processing can be used to meet some of their computational demands.

Processors designed for high degrees of parallelism are widely available in the form of GPUs. Through frameworks like CUDA [24] and OpenCL [22], GPUs can run general-purpose programs. While not well-suited to *all* types of programs, they excel on parallel code: consumer GPUs presently contain up to 2048 cores and are fairly inexpensive; the GPU we use for evaluation (an NVIDIA GTX 480) has 480 cores and a current market value of about $250, or about 50 cents per core. Core counts in GPUs are also increasing rapidly: in the last four years, mass market GPUs have grown from 32 cores to 2048. In comparison, over the same time period, x86 CPUs have "only" increased their core count from two to eight. Thus, when faced with highly parallel computation, there is clear incentive to investigate the use of GPUs.

We find that some computation in storage systems is, indeed, well suited to GPU computing. For example, as we will see in Section 4, a CPU is sufficient for encrypting a filesystem stored on a traditional hard disk drive: an AES cipher on the CPU can sustain approximately 150 MB/s of throughput. Moving the encryption to a GPU, we are easily able to keep up with a fast SATA-attached SSD with a sustained read throughput of over 250 MB/s. Further experiments demonstrate that the drive is the bottleneck, and our GPU is capable of sustaining nearly 1.4 GB/s of encrypted read throughput. This suggests that a single GPU has sufficient processing power to keep up with the rated transfer speeds of six SSDs or ten hard drives. It also puts

the GPU's throughput in a class with some of the fastest PCI-attached flash storage devices currently in production [11]. As the throughput of storage systems rises, GPUs present a promising way to place computation into those systems while taking full advantage of the speed of the underlying storage devices.

There are a number of obstacles, however, to achieving this level of performance while integrating seamlessly with existing storage architectures. The first contribution of this paper is identification of the challenges involved in building a general-purpose framework for GPU acceleration of storage systems. The second contribution is the design and implementation of `GPUstore`, which gives storage subsystems in the operating system kernel a straightforward and high-performance interface for running code on the GPU.

The first obstacle is that most frameworks available for "GPGPU" (general purpose GPU) computing are fundamentally built around different workloads than those encountered in storage: they are designed for long-running computation on large datasets, as seen in graphics and high performance computing. In contrast, storage systems are built, at their lowest levels, on sectors, pages, stripes, and other relatively small structures, and rely on a great deal of asynchrony to get good performance. Requests may arrive at any time, alone or in bursts, and the computation required to service each individual request is relatively modest. While there exist GPU computing frameworks that target alternative types of workloads [6, 32], none match up directly with the needs of storage. For example, PTask [32] uses a dataflow model analogous to UNIX pipes, while kernel storage systems pass data and control through kernel-managed abstractions such as the virtual filesystem layer and page cache. Modifying a storage system to use a pipe-like model would necessitate significant fundamental changes to its architecture.

The second obstacle is that GPUs have resources that must be managed, including on-board RAM, DMA engines, and execution streams. Managing these resources on behalf of storage systems is more than a matter of convenience for the implementers of such systems: it has a direct impact on efficiency and performance as well. By designing a framework specifically for the types of computation used in storage systems, we can optimize resource management for these workloads. For example, GPUs have their own RAM that is separate from main memory, and there are overheads associated with moving data between the two; by taking advantage of the independent nature of computations on blocks, pages, and files, we can define operations that change the granularity of requests to minimize memory copy overheads.

This paper is divided into four parts. First, we detail the challenges of designing a GPU computing framework for storage systems and present `GPUstore`. Second, we demonstrate `GPUstore` by using it to implement GPU acceleration for standard Linux subsystems: the `dm-crypt` encrypted block device, the eCryptfs encrypted filesystem, and the `md` software RAID layer. Third, we evaluate these implementations, showing that the GPU-accelerated subsystems outperform their CPU counterparts by a wide margin—in some cases, as much as an order of magnitude. We finish by discussing related work on GPU computing frameworks and instances of GPU use in storage systems.

2. `GPUstore` Framework

The goal of the `GPUstore` framework is to allow the GPU to be used as a "co-processor" for storage subsystems, fitting naturally into the abstractions and patterns common to this domain. Storage subsystems should be able to call functions that run on the GPU as easily as they call functions from any other subsystem in the kernel.

`GPUstore` is built on top of a popular framework for GPU computing, CUDA [24]. CUDA provides a custom compiler for C (`nvcc`) and runtime libraries for launching `nvcc`-compiled code on the GPU. CUDA treats the GPU as a co-processor; the CPU (1) uploads individual programs called "kernels" (hereafter referred to as "GPU-kernels" to distinguish them from the operating system kernel); (2) provides the GPU-kernel with input in the form of memory buffers copied from main memory using DMA over a PCI-E bus; (3) waits for the GPU-kernel to complete, copying back a buffer of results. Embedded in these steps is the first design challenge faced by `GPUstore`: managing the overheads inherent in copying memory across the PCI-E bus. The CUDA runtime executes in userspace, communicating with the GPU through a kernel driver. The CUDA runtime and NVIDIA GPU driver are closed source; we are not able to modify them, nor are we able to interact with the GPU without going through CUDA. (An alternative to CUDA, OpenCL, is more open; however, it less mature, and has not yet implemented several features that we require.) This arrangement introduces challenges for the implementation of `GPUstore`, but does not fundamentally affect its design.

2.1 `GPUstore` Design Challenges

To get a clear picture of the requirements for `GPUstore`, and thus the challenges it must address, we begin by identifying potential "client" subsystems. Previous studies [9, 14, 16, 25] have shown that GPUs excel at encryption, hashing, checksums, parity calculation, table lookups, scanning, sorting, and searching. Many of these tasks are used in storage systems to provide:

- **Security and Confidentiality**: eCryptfs, Cryptfs [39], CFS [4], `dm-crypt`, and more
- **Reliability and Integrity**: ZFS [37], software RAID (such as Linux's `md`), I^{3}FS [28], etc.
- **Performance**: Venti [30], content-addressable storage such as HydraFS [35] and CAPFS [23], etc.
- **New Functionality**: in-kernel databases [20], steganographic filesystems [27], and more

These storage subsystems fall into two main categories: filesystems, such as eCryptfs and I^{3}FS, and virtual block device drivers, such as `dm-crypt` and `md` . After analyzing the properties of these two classes of components, we identified five main factors that guide `GPUstore`'s design: asynchrony, redundant buffering, GPU memory copy and access overhead, latency for large operations, and GPU resource management.

2.1.1 Asynchrony

Both filesystems and virtual block device drivers exploit asynchrony in order to achieve high performance.

Filesystems work with the Virtual Filesystem (VFS) layer, and often rely on the kernel page-cache for reading and writing. By its nature, the page-cache makes all I/O operations asynchronous: read and write requests to the page cache are not synchronous unless an explicit `sync` operation is called or a *sync* flag is set when opening a file.

Virtual block device drivers work with the kernel's block-I/O layer, which is an asynchronous request processing system. Once submitted, block-I/O requests are maintained in queues. Device drivers, such as SCSI or SATA drivers, are responsible for processing the queue and invoking callbacks when the I/O is complete.

Some filesystems and block devices, such as NFS, CIFS, and iSCSI, depend on the network stack to provide their functionality. Because of the potential for high, unpredictable, latencies on a network, these subsystems are asynchronous by necessity.

As a result, most filesystems and block devices work asynchronously to avoid blocking and waiting. This requires `GPUstore` to support asynchronous clients, and it gives us the opportunity to take advantage of asynchrony for performance optimizations.

2.1.2 Redundant Buffering

GPU drivers allocate and maintain pages in main memory for the purpose of copying data back and forth to the GPU. At the same time, the memory used for block devices and filesystems is managed (or sometimes allocated) by the kernel block I/O layer or page cache. This leads to redundant buffers: in order to move data from a client subsystem to the GPU, it must be copied from a source buffer to the GPU driver's memory, and the results must similarly be copied back. These redundant memory copies and allocations cost both time and space, and `GPUstore` should manage memory in a way that minimizes or eliminates this overhead. The CUDA framework has a feature known as "GPU-Direct technology" [24] to avoid redundant memory buffers between the GPU driver and certain device drivers, but this is only available to a select set of device drivers and not to general kernel components.

2.1.3 GPU Memory Copy and Access Overhead

Good performance in a GPU program usually requires operating on a large dataset [24] for four reasons. First, running code on the GPU requires the launch of a GPU-kernel, the overhead of which is hundreds of times larger than a simple CPU function call. This overhead is constant: with larger data sizes, the relative penalty of the launch decreases. Second, inputs and outputs to GPU-kernels must be copied between main memory and the GPU over the PCI-E bus. While this is a high-bandwidth bus, it nonetheless imposes overhead. Third, each GPU core is, individually, much slower than a CPU core. In order to see a speedup over CPU performance, a large number of cores must be used, meaning that a large amount of data must be available for parallel processing. Finally, GPUs hide the latency of accessing their own memory by using fast thread switching; maximum performance is reached with many threads (up to several dozen) per core.

However, computing on large buffers is not the common case in the Linux kernel's storage subsystems. For example, in the `dm-crypt` block-level encryption driver, data is encrypted/decrypted one memory page (4KB) at a time. We cannot rely on kernel's I/O scheduler to overcome this limitation, since `dm-crypt` intentionally splits requests after they go through the I/O scheduler.

Although it is possible to modify some existing subsystems to support larger requests or to merge small requests, this is often undesirable; many subsystems are fundamentally designed around processing small amounts of data, and changing this may require significant re-design. `GPUstore` must bridge the gap between clients that make small requests and the large-data needs of GPU computing.

2.1.4 Latency For Large Operations

While operating on large data blocks reduces GPU overhead, it also has a downside: computation on the GPU does not begin until all data has been copied from main memory. This can hurt both latency and throughput: if executing a single GPU-kernel at a time, memory copies and computation are serialized rather than pipelined. To deal with this problem, GPU computing libraries such as CUDA provide asynchronous memory copies and allow GPU-kernel execution to overlap with memory copies. Splitting large computations into smaller ones and pipelining execution of each fragment with the memory copy for the next both reduces latency and improves efficiency. Selecting an appropriate size for GPU tasks is a delicate balancing act that depends on the particulars of the GPU and GPU-kernel in use. `GPUstore` should manage this complexity itself, rather than exposing it to every storage subsystem.

2.1.5 GPU Resource Management

GPUs have more resources than just computing cores: they also have on-board memory and many have multiple DMA engines. There may be many subsystems using a GPU at once, and a single computer may contain multiple GPUs. Though current GPUs do not support software thread-level scheduling on GPU cores, the CUDA stream [24] model includes higher-level abstraction for computing cores and DMA copy engines. To provide a general framework for

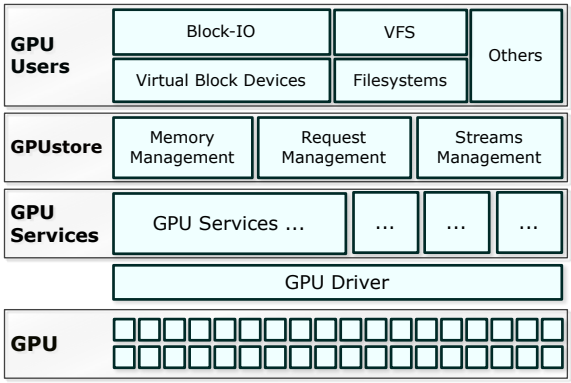

Figure 1. The architecture of `GPUstore`.

GPU computing inside the kernel, these resources must be shared by all client subsystems, and managed by `GPUstore`.

2.2 `GPUstore` Design

Figure 1 shows the architecture of the `GPUstore` framework and the context it resides in. In `GPUstore`, GPU computing tasks are abstracted into "GPU services." GPU services are written as C programs, compiled with the `nvcc` compiler, and are launched on the GPU using CUDA. Since CUDA must run in userspace, we have a "helper" process that runs outside the kernel and manages interaction with the GPU; this helper is invisible to `GPUstore`'s clients, which interact with it solely through its kernel API. Through careful use of shared memory and synchronization primitives, we have minimized the overhead of passing requests between the kernel and userspace portions of `GPUstore`.

Requests to services are submitted asynchronously to `GPUstore` by client subsystems and managed in a queue. GPU services may implement service-specific policies for handling of their requests in the queue. A request indicates which GPU service is to be invoked and includes two sets of parameters: a set of buffers in main memory to be used as the primary arguments to the service, and a set of service-specific parameters. Service requests are dispatched to the appropriate GPU-kernels by `GPUstore`, where they are processed asynchronously. When results are available from the service, a callback is made to the client. Clients of `GPUstore` may optionally choose to treat the calls as synchronous, by blocking themselves and waiting for the service to return. Before dispatching requests, `GPUstore` is responsible for assigning them GPU computing resources, which are implemented as CUDA streams.

The processing of each request to a GPU service includes three major steps: (1) copy buffers from main memory to the GPU; (2) invoke the GPU-kernel corresponding to the requested service; (3) copy results back after completion. These steps are performed sequentially for each request, but steps from different requests (either for the same service or different services) can proceed in parallel, pipelining the process.

2.2.1 Request Management

To ensure that requests are neither too small, as described in Section 2.1.3, nor too large, as described in Section 2.1.4, `GPUstore` defines *merge* and *split* operations on requests. These operations have precedent in storage systems, with similar operations being provided by the Linux I/O scheduler.

To invoke a *merge*, `GPUstore` defers processing small requests. Instead, it queues up many requests for the same service and processes them together. This way, clients that always produce small requests (such as those that process a single page at a time) can accumulate large enough requests to benefit from GPU computing acceleration.

To perform a *split*, `GPUstore` divides a large request into smaller ones and pipelines computation and memory copies. *split* helps to reduce latency and hide the GPU memory copy overhead in the case of requests that are very large. *split* is made possible by the fact that storage computations are typically performed independently on objects such as blocks, pages, and files; because the service knows the boundaries of these objects, it can safely partition them into subsets before sending them to the GPU. Consistency or atomicity guarantees in the storage system are unchanged, as the *split* is invisible to the caller; `GPUstore` waits for all parts of the original request to complete before invoking its callback.

Both *merge* and *split* need service-specific knowledge to determine appropriate data sizes and number of requests: thresholds are dependent on factors such as the relative costs of memory copies and computation for a particular service. `GPUstore` allows services to implement their own request management and scheduling policies, such as the number of requests to be batched for *merge*s and the maximum size for requests before they are *split*. These policies can be specified statically in the service implementation or defined dynamically; for example, by running small benchmarks at boot time to customize them for particular GPUs.

Of course, *merge* may increase latency when requests are small and infrequent, because it waits for several requests to arrive before processing. Each service in `GPUstore` can set its own timeout threshold for *merge* operations, bounding the latency that this operation can incur. Still, determining an appropriate value for the timeout is tricky. It is preferable for the client subsystems to support large requests natively, giving them more control over latency. *merge* is useful primarily for cases in which it is too difficult to re-architect a subsystem to operate on larger data sizes. It should be noted that *split* does not suffer from the same problem; it introduces no additional latency, so it can be safely applied by any GPU service.

As we will see in Section 4, the CPU versions of services sometimes outperform their GPU counterparts for small block sizes. `GPUstore` enables services to run in *hybrid mode*—each service may set a size threshold for requests; below this threshold, the service is executed on the CPU.

2.2.2 Memory Management in GPUstore

There are two kinds of memory that GPUstore must manage: the main memory that is used by the GPU driver for DMA and on-board GPU device memory. In GPUstore, we use CUDA's mapped page-locked memory [24] to bind each buffer in main memory to a corresponding buffer of the same size on the GPU device. This one-to-one mapping simplifies device memory management and gets the highest performance for memory copies [24]. The downside, however, is that it makes suboptimal use of host memory: buffers must remain allocated even when not in active use for copies to or from the GPU.

In order to remove the redundant buffer overhead described in Section 2.1.2, GPUstore supports two different approaches to client subsystems' memory allocation. Each is useful in different types of code. When the client subsystem manages its own buffers in memory, *direct allocation* is appropriate. When the client operates on buffers that are allocated and managed by another kernel subsystem (such as the page cache), *remapping* eliminates redundant buffering for pre-existing allocations.

Direct Allocation from the GPU Driver: A client can ask GPUstore for pages that have been allocated by the GPU driver: this is done by re-mapping memory allocated by the driver into the client's own virtual memory space. Most of time, this virtual space is simply the kernel space, because GPUstore targets kernel components. However, there are some cases that map to a userspace process, such as filesystems that support `mmap()`ed I/O. To directly allocate GPU driver memory, the client calls a `vmalloc()` or `kmalloc()` function provided by GPUstore, rather than the standard versions of these functions provided by the Linux kernel. The client may treat the allocated memory exactly as if it came from the normal allocation functions. When using memory allocated this way, the client manipulates its data (such as pages used to store data from files or block devices) directly in pages that can be DMAed to the GPU. As a result, no extra buffers are needed in main memory, and most copies between buffers in main memory are eliminated.

Remapping GPU User's Memory Pages: The other approach operates in the opposite direction: instead of allocating pages from the GPU driver, GPUstore takes existing pages from the client and maps them into the GPU driver's space. Page frames are locked (and DMA to and from them becomes possible) at the time of remapping. This approach also eliminates redundant buffers, and allows the client system to use memory that was allocated for it by other kernel subsystems, such as the page cache. The downside, however, is that this results in frequent re-mapping of pages, and has a tendency towards use of many small fragmented pages.

2.2.3 GPU Stream Management

GPUs can execute multiple GPU-kernels concurrently. This introduces scheduling of execution. A GPU can also have multiple DMA engines, enabling it to sustain multiple concurrent copies between main memory and GPU device memory. This introduces scheduling of copies. Since a computer system can have multiple GPUs, scheduling of execution and copies between tasks and across GPUs can be quite complicated. In the CUDA framework, the copy and execution resources are abstracted into "streams" [24]. Streams are the resources that GPUstore manages for scheduling service execution. Although it is not currently possible to do software-controlled preemption on GPUs, a stream still represents a non-preemptible computing or copy resource. GPUstore maintains GPU streams and assigns them to service requests for service execution and memory copies. Our current GPUstore implementation uses a simple first-come-first-served policy to schedule streams. However, it wold be straightforward, as future work, to adopt scheduling algorithms from the related work discussed in Section 5.

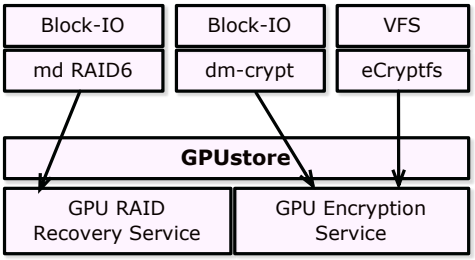

Figure 2. Storage services implemented with GPUstore.

Subsystem	Total LOC	Modified LOC	Percent
dm-crypt	1,800	50	3%
eCrytpfs	11,000	200	2%
md	6,000	20	0.3%

Table 1. Approximate lines of code required to call GPU services using GPUstore.

3. Implementing Accelerated Storage

We have used GPUstore to accelerate three existing Linux kernel storage components. We enhanced encrypted storage with dm-crypt and eCryptfs, and the software RAID driver md. We chose these three subsystems because they interact with the kernel in different ways: md and dm-crypt implement the block I/O interface, and eCryptfs works with the virtual filesystem (VFS) layer. The architecture of these implementations is shown in Figure 2.

The design of GPUstore ensures that client subsystems need only minor modifications to call GPU services. Table 1 gives the approximate number of lines of code that we had to modify for our example subsystems. The lines of code reported in this table are those in the subsystems that are modified to call GPUstore, and do not include the lines of code used to implement the GPU services. Linux storage subsystems typically call out to other re-

usable kernel components to perform common operations such as encryption: essentially, we replace these with calls to `GPUstore` and make minor changes to memory management.

3.1 Encrypted Storage

Encrypted storage is a good match for the GPU: while encryption and decryption are computationally expensive, most ciphers used by storage systems are parallelizable. This parallelization can happen at different levels, such as files, requests or disk sectors.

We added support for `GPUstore`-accelerated ciphers to the Linux cryptography subsystem, where they can be selected by any client of the Linux cryptography API; `dm-crypt` and eCryptfs are already make use of this API. To achieve high levels of performance, modest changes are required in the storage systems. These changes address the buffer allocation and overhead issues discussed in Section 2.2.

Although there exist dedicated encryption accelerators and CPUs with special encryption features (such as AES-NI), GPUs are nonetheless quite competitive. As we will see in Section 4, our AES cipher is able to sustain a rate of over 4 GB/s; results reported for a high-end CPU with AES-NI [19] are less than half of this speed. GPU implementations of ciphers are also much more flexible: since they are "just code," they allow for any cipher to be implemented, rather than "baking in" support for a particular cipher into hardware as AES-NI does.

3.1.1 `dm-crypt`

`dm-crypt` is a crypto target in Linux's device mapper framework. It works as a virtual block device, running on top of other block devices such as disks. `dm-crypt` provides an encrypted view of the underlying devices by performing block-level encryption. It works asynchronously, as do most other block drivers.

`dm-crypt` interacts with the block I/O layer of the Linux kernel, taking I/O requests and splitting them into 4KB pages, which are encrypted or decrypted as required. As we will see in Section 4.1, for 4KB blocks, the size used by the unmodified `dm-crypt`, GPU overheads dominate and CPU encryption is faster. In our modified version of `dm-crypt`, we leave the task of combining these 4KB pages into larger blocks to the `GPUstore` framework: it relies on the *merge* operation described in Section 2.2.1 to coalesce several small requests into one large one. We use `GPUstore`'s memory allocator to request memory pages from the GPU device driver so that the data manipulated by `dm-crypt` is ready for direct DMA copy to the GPU. This means that our changes to `dm-crypt` are minor: our implementation changes only approximately 50 lines of code, less than 3% of the 1,800 total lines in `dm-crypt`.

3.1.2 eCryptfs

eCryptfs, which is derived from Cryptfs, is a "stacking" filesystem. To use it, one creates another standard Linux filesystem, such as EXT3, and mounts eCryptfs "on top" of it. eCryptfs uses the underlying filesystem as a backing store and handles encryption and decryption of the files stored to it. It is part of the mainline Linux kernel, and is used for user home directory data security in many Linux distributions such as Ubuntu and Fedora.

To get good performance from eCryptfs, we made two changes. The first was to change the cipher mode from CBC to CTR: CBC encryption cannot be parallelized because ciphertext from each block is used as input to the following block, creating dependencies. CTR, in contrast, uses a counter to provide non-uniform input, creating no such dependencies and allowing us to evaluate both read and write (decryption and encryption) workloads. In practice, implementations could still choose CBC mode and forgo GPU acceleration for encryption while still taking advantage of it for decryption. This would be a good tradeoff for read-heavy workloads.

The second change was to enable eCryptfs to perform cryptography on larger blocks of data. Like `dm-crypt`, eCryptfs uses page-sized units for cryptographic operations, which means that even when a large I/O buffer is used by higher level applications, eCryptfs still splits it into small pages and processes them one by one. This is not a problem when there are many concurrent readers and writers, as `GPUstore`'s *merge* operation can coalesce requests from different I/O streams into large GPU requests. However, with a single reader or writer, large individual I/O requests must be supported in order to attain high performance. We therefore modified eCryptfs to support the `readpages` interface, which allows the kernel's page cache layer to pass multi-page reads directly to the filesystem. Large write support was provided by a replacement `write` operation at the VFS layer instead of `writepages` due to intricacies of the Linux kernel page cache system. Implementing these multi-page read and write interfaces enabled the kernel's existing mechanisms, such as readahead, I/O scheduling, and caching, to also take advantage of GPU performance enhancements.

Despite the fact that we made more changes to eCryptfs than `dm-crypt`, our modifications to eCryptfs are still relatively small: we modified approximately 200 lines of code, or less than 2%, of eCryptfs. Those changes also include remapping eCryptfs pages into the GPU driver's address space for GPU service requests; these pages are managed by the kernel's page cache, and it is therefore infeasible to allocate them from the GPU driver.

3.2 RAID 6 Recovery

RAID services may be provided in hardware or in software; the Linux kernel includes a software RAID implementation called `md`. RAID 6, which can continue operating with two failed disks, is widely deployed as a space-efficient, fault-tolerant RAID configuration [7]. RAID 6 employs erasure codes to allow a set number of disks in a system to fail with no data loss. While many proprietary RAID 6 codes exist [3, 8],

the Linux kernel includes a RAID 6 implementation based on Reed-Solomon coding [31].

RAID 6 coding, which occurs during write workloads, has efficient SSE implementations for x86 and x86-64 processors. Unfortunately, the available vector instructions are not sufficient for RAID 6 decoding, or data recovery, which must occur during reads when a disk has failed [1]. The primary issue is that Reed-Solomon coding with an eight-bit word size relies on a lookup table with 256 entries, which is not vectorizable with current CPU instructions, resulting in serialized access to these tables.

We have implemented GPU-accelerated data recovery for RAID 6 and integrated it into the `md` device driver. Our implementation of "degraded mode" makes use of the fact that GPUs have hundreds of cores with which to issue table lookups, along with high-speed scratch memories that service several accesses simultaneously [24]. Previous work has reported that this organization allows a GeForce GTX 285 GPU to access 41.1 table entries per clock cycle, on average [10]. Our recovery code handles the particularly challenging case where two data strips within a stripe holding data have failed: in this case, all remaining data strips, plus both parity disks, must be used to recover the missing data. Since RAID 6 distributes parity strips across the entire disk array, failure of any two disks will trigger this recovery case.

While RAID stripes can be large enough in size to overcome GPU overheads, the `md` driver, like `dm-crypt`, is fundamentally organized around processing individual 4KB memory pages. `md` is a very complex system that includes working threads, asynchronous parity operations, and multiple internal request queues. We decided not to modify `md` to operate on larger requests, as that would require fundamental changes in its architecture. Instead, we rely on `GPUstore`'s *merge* to deal with small requests, and use the allocator provided by `GPUstore` to allocate memory pages from the GPU driver's space. As a result, our GPU integration for `md` requires only 20 lines of changes, most of which are simply saving buffer pointers for safe sharing of asynchronous calls.

4. Evaluation

We benchmarked the `GPUstore` framework itself as well as the three storage subsystems that we adapted to use it. We used two machine configurations for our evaluation. The first, system *S1*, has an Intel Core i7 930 Quad-Core CPU, 6 GB memory, and uses a 32 GB Intel X25-E SSD. The SSD is SATA-attached, and has rated read and write speeds of 250 MB/s and 170 MB/s respectively. *S1* is used for filesystem and block device tests. The second, *S2*, has an Intel Core i7 975 Quad-Core CPU, 6 GB DDR3 memory, and a 2 port Fibre Channel 4 Gb adapter. It is connected to two switched JBODs each containing sixteen 750 GB 7200 RPM SATA disks, for a total of 32 disks. *S2* is used for the RAID benchmarks. Both *S1*and *S2* have an NVIDIA GTX 480 GPU. This GPU has 480 cores running at 1.4GHz and 1.5 GB of GDDR5 RAM. Both systems run the 2.6.39.4 Linux kernel and use CUDA version 4.0.

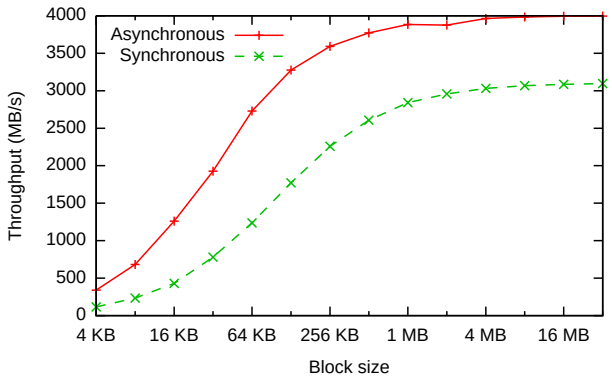

Figure 3. Throughput for a GPU-kernel that copies data, but performs no computation

.

All benchmarks were run without use of *hybrid mode* in `GPUstore`; that is, GPU services were not allowed to fall back to the CPU for small requests. This has the effect of clearly illustrating the points where the GPU implementation, by itself, underperforms the CPU, as well as the points where their performance crosses. With *hybrid mode* enabled, `GPUstore` would use the CPU for small requests, and the CPU performance can thus be considered an approximate lower bound for `GPUstore`'s hybrid mode performance.

In many cases, our GPU-accelerated systems are capable of out-performing the physical storage devices in our systems; in those cases, we also evaluate them on RAM-backed storage in order to understand their limits. These RAM-based results suggest that some `GPUstore` accelerated subsystems will be capable of keeping up with multiple fast storage devices in the same system, or PCI-attached flash storage, which is much faster than the drives available for our benchmarks.

4.1 `GPUstore` Framework Performance

Our first microbenchmark examines the effect of block sizes on `GPUstore`'s performance and compares synchronous operation with asynchronous. On *S1*, we called a GPU service which performs no computation: it merely copies data back and forth between main memory and GPU memory. Note that total data transfer is double the block size, since the data block is first copied to GPU memory and then back to main memory. In Figure 3, we can see that at small block sizes, the overheads discussed in Section 2.1.3 dominate, limiting throughput. Performance steadily increases along with block size, and reaches approximately 4 GB/s on our system. This benchmark reveals three things. First, it demonstrates the value to be gained from our *merge* operation, which increases block sizes. Second, it shows a performance boost of 30% when using asynchronous, rather than synchronous, requests to the GPU. Finally, it serves as an upper bound for performance of GPU services, since our test service performs no computation.

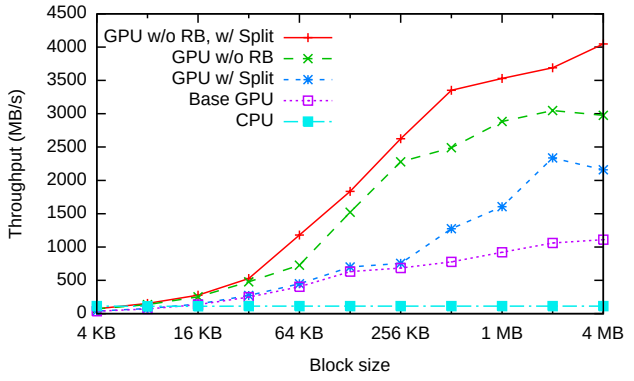

Figure 4. GPU AES cipher throughput with different optimizations compared with Linux kernel's CPU implementation. The experiments marked "w/o RB" use the techniques described in Section 2.2.2 to avoid redundant buffering.

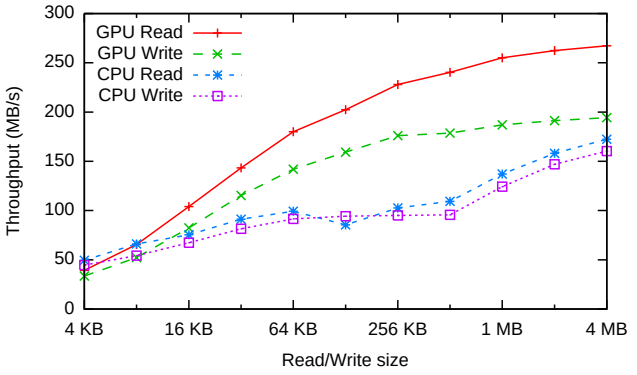

Figure 5. `dm-crypt` throughput on an SSD-backed device.

Our second microbenchmark shows the effects of our optimization to remove redundant buffering and the *split* operation. This benchmark, also run on *S1*, uses the AES cipher service on the GPU, and the results can be seen in Figure 4. The baseline GPU result shows a speedup over the CPU cipher, demonstrating the feasibility of GPU acceleration for such computation. Our *split* operation doubles performance at large block sizes, and eliminating redundant buffering triples performance at sizes of 256 KB or larger. Together, these two optimizations give a speedup of approximately four times, and with them, the GPU-accelerated AES cipher achieves a speedup of 36 times over the CPU AES implementation in the Linux kernel. The performance levels approach those seen in Figure 3, implying that the memory copy, rather than the AES cipher computation, is the bottleneck.

4.2 `dm-crypt` Sequential I/O

Next, we use the `dd` tool to measure raw sequential I/O speed in `dm-crypt`. The results shown in Figure 5 indicate that with read and write sizes of about 1MB or larger, the GPU-accelerated `dm-crypt` easily reaches our SSD's maximum throughput (250MB/s read and 170MB/s write). The CPU

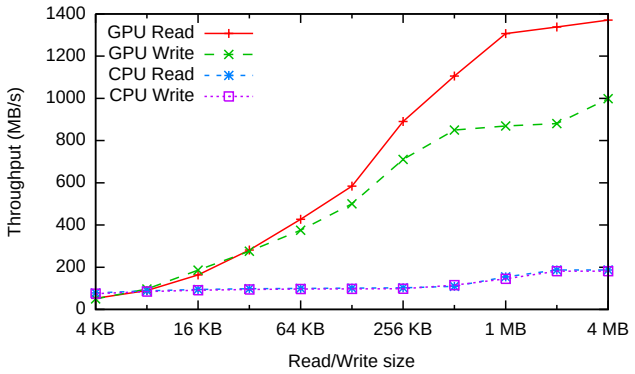

Figure 6. `dm-crypt` throughput on a RAM-backed device.

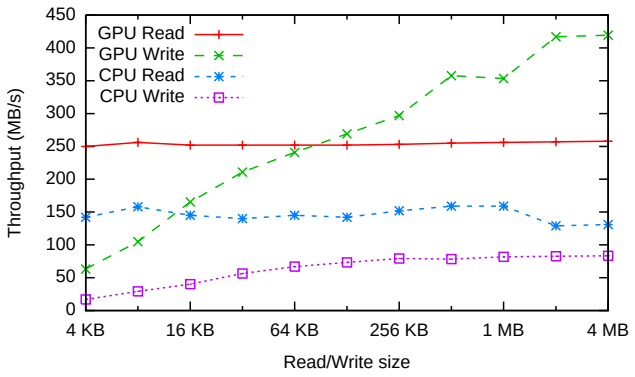

Figure 7. eCryptfs throughput on an SSD-backed filesystem.

version is 60% slower; while it would be fast enough to keep up with a mechanical hard disk, it is unable reach the full potential of the SSD. Substituting a RAM disk for the SSD (Figure 6), we see that the GPU-accelerated `dm-crypt` was limited by the speed of the drive: it is able to achieve a maximum read throughput of 1.4 GB/s, more than six times as fast as the CPU implementation. This is almost exactly the rated read speed for the ioDrive Duo, currently the third fastest SSD in production [11]. As the throughput of storage systems rises, GPUs present a promising way to place computation into those systems while taking full advantage of the speed of the underlying storage devices.

4.3 eCryptfs Sequential and Concurrent Access

Figure 7 and Figure 8 compare the sequential performance for the CPU and GPU implementation of eCryptfs. We used the `iozone` tool to do sequential reads and writes using varying block sizes and measured the resulting throughput. Because eCryptfs does not support direct I/O, effects from kernel features such as the page cache and readahead affect our results. To minimize (but not completely eliminate) these effects, we cleared the page cache before running read-only benchmarks, and all writes were done synchronously.

Figure 7 shows that on the SSD, the GPU achieves 250 MBps when reading, compared with about 150 MBps

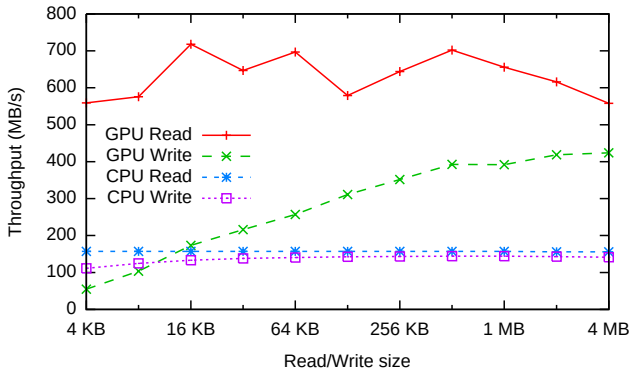

Figure 8. eCryptfs throughput on a RAM-backed filesystem.

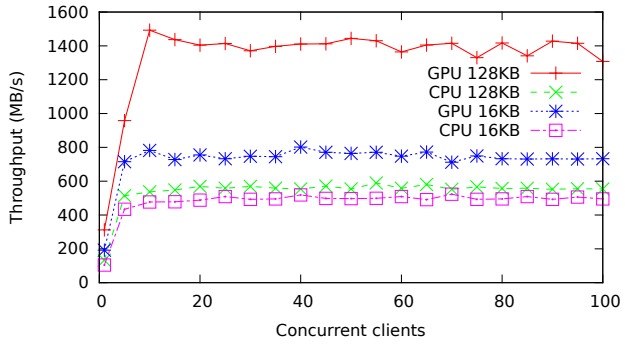

Figure 9. eCryptfs concurrent write throughput on a RAM disk for two different block sizes.

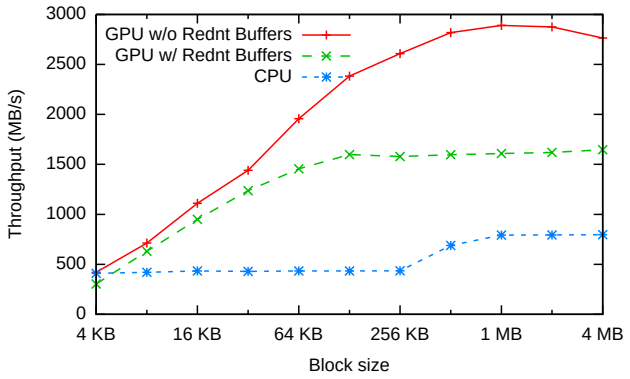

Figure 10. Throughput for the RAID 6 recovery algorithm with and without optimizations to avoid redundant buffers.

for the CPU, a 70% speed increase. Unlike our earlier benchmarks, read speeds remain nearly constant across all block sizes. This is explained by the Linux page-cache's readahead behavior: when small reads were performed by `iozone`, the page-cache chose to issue larger reads to the filesystem in anticipation of future reads. The default readahead size of 128 KB is large enough to reach the SSD's full read speed of 250MB/s. This illustrates an important point: by designing `GPUstore` to fit naturally into existing storage subsystems, we enable it to work smoothly with the rest of the kernel. Thus, by simply implementing the multi-page `readpages` interface for eCryptfs, we enabled existing I/O optimizations in the Linux kernel to kick in, maximizing performance even though they are unaware of `GPUstore`.

Another surprising result in Figure 7 is that the GPU write speed exceeds the write speed of the SSD, and even its read speed, when block size increases beyond 128 KB. This happens because eCryptfs is, by design, "stacked" on top of another filesystem. Even though we take care to `sync` writes to eCryptfs, the underlying filesystem still operates asynchronously and caches the writes, returning before the actual disk operation has completed. This demonstrates another important property of `GPUstore`: it does not change the behavior of the storage stack with respect to caching, so client subsystems still get the full effect of these caches without any special effort.

We tested the throughput limits of our GPU eCryptfs implementation by repeating the previous experiment on a RAM disk, as shown in Figure 8. Our GPU-accelerated eCryptfs achieves more than 700 MBps when reading and 420 Mbps when writing. Compared to the CPU, which does not perform much better than it did on the SSD, this is a speed increase of nearly five times for reads and close to three times for writes. It is worth noting that Linux's readahead mechanism not only "rounds up" read requests to 128 KB, it "rounds down" larger ones as well, preventing eCryptfs from reaching even higher levels of performance.

Finally, we used `filebench` to evaluate eCryptfs under concurrent workloads. We varied the number of concurrent writers from one to one hundred, and used the RAM-backed filesystem. Each client writes sequentially to a separate file. The effects of `GPUstore`'s *merge* operation are clearly visible in Figure 9: with a single client, performance is low, because we use relatively small block sizes (128 KB and 16 KB) for this test. But with ten clients, `GPUstore` is able to *merge* enough requests to get performance on par with `dm-crypt` at a 1 MB blocksize. This demonstrates that `GPUstore` is useful not only for storage systems with heavy single-threaded workloads, but also for workloads with many simultaneous clients. While block size still has a significant effect on performance, `GPUstore` is able to amortize overheads across concurrent access streams to achieve high performance even for relatively small I/O sizes.

4.4 `md` RAID 6 Data Recovery

As with encryption, the performance of our GPU-based RAID 6 recovery algorithm increases with larger block sizes, eventually reaching six times the CPU's performance, as seen in Figure 10.

We measured the sequential bandwidth of a degraded RAID 6 array consisting of 32 disks in our *S2* experiment environment. The results are shown in Figure 11. We find that GPU accelerated RAID 6 data recovery does not achieve

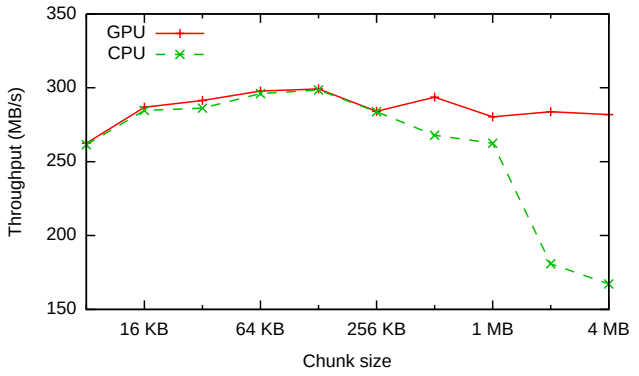

Figure 11. RAID 6 read bandwidth in degraded mode

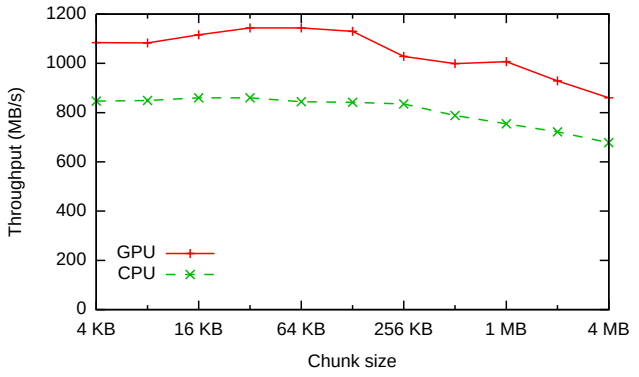

Figure 12. RAID 6 read bandwidth in degraded mode on RAM-backed devices

significant speedup unless the array is configured with a large chunk size, or strip size. Interestingly, the speedup is caused by decreasing CPU performance. We believe the decrease is caused by the design of `md`'s I/O request memory pool, which does not efficiently handle large numbers of requests on large stripes. Because `GPUstore` *merge*s these requests into larger ones, it avoids suffering from the same problem.

We also measured the degraded mode performance of a RAID 6 array in the *S1* system using 6 RAM disks. The results are shown in Figure 12. We find that our previous recovery experiment was limited by the speed of the hard disks, and both CPU and GPU implementations would be capable of faster performance given faster disks. With RAM disks, the CPU based recovery reaches the maximum throughput we saw in Figure 10, while the GPU version is still far from its own maximum in the figure.

However, with chunk sizes below 16KB, the throughputs on RAM disk arrays are actually much higher than we saw for the raw algorithm in Figure 10. This result demonstrates the effectiveness of the request *merge* operation in `GPUstore`. *merge* was in use for recovery benchmark, but not the raw algorithm test, and the former therefore saw larger effective block sizes.

5. Related Work

There has been much work [26, 33] on accelerating storage systems by enhancing I/O patterns, improving data layout, effective caching, performing readahead, etc. With more and more modern storage systems providing value-added features such as encryption, RAID, and content-addressability, the computational requirements of storage systems increases, and can become a bottleneck. `GPUstore` focuses on the computational needs of storage systems, enabling acceleration on highly-parallel GPUs by providing a framework for integrating them into the storage stack.

GPU Computing Frameworks: There are a number of existing GPGPU computing frameworks. These include CUDA [24], OpenCL [22], hiCUDA [15], CUDALight [34], and Brook [6]. Users of these frameworks typically load large datasets onto the GPU and run very expensive computations synchronously, such as graphics processing or physics simulations. As a result, these traditional GPGPU frameworks place less emphasis on the ability to launch many small tasks on short timescales, and do not include optimizations like our *split* and *merge* operations for the small-task workloads found in storage systems.

In-kernel GPU Computing: Gdev [21] and Barracuda [5] bring GPU computing into the OS kernel. Barracuda uses a microdriver-based design and a userspace helper, but provides only a bare minimum synchronous API with no request scheduling, merging, splitting, or memory management. Gdev provides a "native" in-kernel CUDA library as well as a lower-level API. Gdev and `GPUstore` are complementary: Gdev concentrates on providing basic access to the GPU from the kernel while `GPUstore` focuses on higher-level access tailored specifically for storage systems. We plan to replace `GPUstore`'s userspace helper with Gdev, which will allow `GPUstore` to run entirely inside the kernel.

GPUs in Storage Systems: Curry et al. introduced Gibraltar, a software RAID infrastructure that uses a GPU to perform Reed Solomon coding [9]. It allows for parity-based RAID levels that exceed the specifications of RAID 6, increasing the resiliency of the array. Lack of a kernel framework for GPU computing forced Gibraltar to be implemented in userspace. As a result, the builders of Gibraltar had to write over two thousand lines of code to re-implement storage stack features that already exist in the kernel, including caching, victimization, asynchronous flushing, and request combining. Being placed outside of the kernel also creates challenges for robustness, ease of use, and administration. `GPUstore` is a general-purpose framework for GPU storage acceleration and enables existing storage infrastructure to be re-used. With it, software RAID can be GPU-accelerated by replacing a few memory management and function calls. Future versions of Gibraltar may use `GPUstore` to move back inside the kernel.

Gharaibeh et al. [13] implemented GPU-accelerated hashing algorithms for content-addressed storage systems. Bhatotia et al. [2] presented Shredder, a framework specially

designed for content-based chunking to support incremental storage and computation. These represent specific instances of GPU computing applied to storage, and are good examples of the benefits that can be gained from GPUs in storage systems. GPUstore is a general framework, designed to aid the implementation of applications like these and others.

GPU Resource Management: The asymmetric distributed shared memory designed by Gelado et al. [12] eases memory management for GPU computing; GPUstore manages computing resources such as cores, DMA engines, and streams in addition to memory. Sponge [18] provides a compilation framework that concentrates on optimizing data-flow management while GPUstore is a runtime framework. Hydra, created by Weinsberg et al. [38], unifies heterogeneous computing devices, and could, in theory, be used to run GPU code from CPU-based storage systems; however, it depends on functionality that current GPUs do not support.

PTask [32] optimizes data-flow in GPU tasks by providing a channel abstraction analogous to UNIX "pipes" between GPU-kernels. While PTask is ideal for programs that use pipe-like communication, it is less suited for components in the kernel storage stack; it organizes programs in explicit dataflow graphs and manages all main memory that will be transferred to the GPU. Neither of these properties mesh well with the kernel, where different parts of the storage stack interact through kernel-managed structures and interfaces such as the page cache and I/O schedulers. GPUstore also includes optimizations that are useful to storage systems, such as *split* and *merge*, that are not present in PTask. PTask includes a scheduler that integrates with the operating system's scheduler to preserve priorities. It also includes optimizations that allow that two back-to-back GPU kernels to re-use computation results without an extra copy to host memory and back. Learning from PTask, we hope to add similar features to GPUstore in the future.

System-level Tasks: While we concentrate on use of GPUstore in storage systems, it has wider potential: we have experimented with using GPUstore in other kernel subsystems. In this respect, it is related to a number of attempts to accelerate system-level tasks with the GPU. These include: PacketShader [14] and SSLShader [19] for IP routing and SSL session acceleration; Gnort [36] for accelerating network intrusion detection; Linux kernel cryptography acceleration [17]; and program analysis with EigenCFA [29]. Each of these efforts focuses on acceleration of a single task; GPUstore is complementary to them, because it provides a general framework which can be used by many kernel subsystems.

6. Conclusion

We have presented GPUstore, a general-purpose framework for using GPU computing power in storage systems within the Linux kernel. By designing GPUstore around common storage paradigms, we have made it simple to use from existing code, and have enabled a number of optimizations that are transparent to the calling system. We modified several standard Linux subsystems to use GPUstore, and were able to achieve substantial improvements in performance by moving parts of the systems' computation on to the GPU. Our benchmark results also demonstrate the effectiveness of the optimizations adopted by GPUstore for matching the storage subsystems requirements. We have released GPUstore as part of kgpu, our general framework for kernel-level GPU computing. The code is licensed under the GPL and can be downloaded from `http://code.google.com/p/kgpu/`.

Acknowledgments

The authors would like to thank Anton Burtsev for his comments on early drafts and the SYSTOR reviewers for their useful feedback and suggestions.

References

[1] R. Bhaskar, P. K. Dubey, V. Kumar, and A. Rudra. Efficient Galois field arithmetic on SIMD architectures. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, 2003.

[2] P. Bhatotia, R. Rodrigues, and A. Verma. Shredder: GPU-accelerated incremental storage and computation. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[3] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, 1994.

[4] M. Blaze. A cryptographic file system for UNIX. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, 1993.

[5] A. Brinkmann and D. Eschweiler. A microdriver architecture for error correcting codes inside the Linux kernel. In *Proccedings of the SC09 Conference*, 2009.

[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *Proceedings of the ACM SIGGRAPH Annual Conference*, 2004.

[7] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[8] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST)*, 2004.

[9] M. L. Curry, H. L. Ward, A. Skjellum, and R. Brightwell. A lightweight, GPU-based software RAID system. In *International Confernece on Parallel Processing (ICPP)*.

[10] M. L. Curry, A. Skjellum, H. L. Ward, and R. Brightwell. Gibraltar: A Reed-Solomon coding library for storage applications on programmable graphics processors. *Concurrency and Computation: Practice and Experience*, 2010.

[11] FastestSSD.com. SSD ranking: The fastest solid state drives, Apr. 2012. `http://www.fastestssd.com/featured/ssd-rankings-the-fastest-solid-state-drives/#pcie`; accessed April 27, 2012.

[12] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and 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*, 2010.

[13] A. Gharaibeh, S. Al-Kiswany, S. Gopalakrishnan, and M. Ripeanu. A GPU accelerated storage system. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2010.

[14] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *Proceedings of the ACM SIGCOMM Conference*, 2010.

[15] T. D. Han and T. S. Abdelrahman. hiCUDA: a high-level directive-based language for GPU programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, 2009.

[16] O. Harrison and J. Waldron. Practical symmetric key cryptography on modern graphics hardware. In *Proceedings of the 17th USENIX Security Symposium*, 2008.

[17] O. Harrison and J. Waldron. GPU accelerated cryptography as an OS service. In *Transactions on Computational Science XI*. Springer-Verlag, 2010.

[18] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: portable stream programming on graphics engines. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[19] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: cheap SSL acceleration with commodity processors. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2011.

[20] A. Kashyap and A. Kashyap. File system extensibility and reliability using an in-kernel database. Technical report, Stony Brook University, 2004.

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

[22] Khronos Group. OpenCL Specification 1.1. `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`.

[23] P. Nath, B. Urgaonkar, and A. Sivasubramaniam. Evaluating the usefulness of content addressable storage for high-performance data intensive applications. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing (HPDC)*, 2008.

[24] NVIDIA Inc. CUDA C Programming Guide 4.0.

[25] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[26] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18:37–66, February 2000.

[27] H. Pang, K.-L. Tan, and X. Zhou. Stegfs: A steganographic file system. *International Conference on Data Engineering*, 2003.

[28] S. Patil, G. Sivathanu, and E. Zadok. I^3fs: An in-kernel integrity checker and intrusion detection file system. In *Proceedings of the 18th Annual Large Installation System Administration Conference (LISA)*, 2004.

[29] T. Prabhu, S. Ramalingam, M. Might, and M. Hall. EigenCFA: accelerating flow analysis with GPUs. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL)*, 2011.

[30] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[31] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[32] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating system abstractions to manage GPUs as compute devices. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[33] J. Schindler, S. Shete, and K. A. Smith. Improving throughput for small disk requests with proximal I/O. In *Proceedings of the 9th USENIX conference on File and Stroage Technologies (FAST)*, 2011.

[34] S. Ueng, M. Lathara, S. S. Baghsorkhi, and W. Hwu. CUDA-Lite: Reducing GPU programming complexity. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 2008.

[35] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. HydraFS: a high-throughput file system for the HYDRAstor content-addressable storage system. In *Proceedings of the 8th USENIX conference on File and Storage Technologies (FAST)*, 2010.

[36] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[37] S. Watanabe. *Solaris 10 ZFS Essentials*. Prentice Hall, 2009.

[38] Y. Weinsberg, D. Dolev, T. Anker, M. Ben-Yehuda, and P. Wyckoff. Tapping into the fountain of CPUs: on operating system support for programmable devices. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[39] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science, Columbia University, 1998.