On the Use of GPUs in Realizing Cost-Effective Distributed RAID

Aleksandr Khasymski[†], M. Mustafa Rafique[†‡], Ali R. Butt[†],
Sudharshan S. Vazhkudai[⋆], Dimitrios S. Nikolopoulos[λ]
[†]Virginia Tech, [‡]Qatar Foundation, [⋆]Oak Ridge National Laboratory, [λ]Queen's University of Belfast
{khasymskia, mustafa, butta}@cs.vt.edu, vazhkudaiss@ornl.gov, d.nikolopoulos@qub.ac.uk

***Abstract*—The exponential growth in user and application data entails new means for providing fault tolerance and protection against data loss. High Performance Computing (HPC) storage systems, which are at the forefront of handling the data deluge, typically employ hardware RAID at the backend. However, such solutions are costly, do not ensure end-to-end data integrity, and can become a bottleneck during data reconstruction. In this paper, we design an innovative solution to achieve a flexible, fault-tolerant, and high-performance RAID-6 solution for a parallel file system (PFS). Our system utilizes low-cost, strategically placed GPUs — both on the client and server sides — to accelerate parity computation. In contrast to hardware-based approaches, we provide full control over the size, length and location of a RAID array on a per file basis, end-to-end data integrity checking, and parallelization of RAID array reconstruction. We have deployed our system in conjunction with the widely-used Lustre PFS, and show that our approach is feasible and imposes acceptable overhead.**

I. Introduction

The deluge of data from scientific instruments (SNS [1], LHC [2]), experiments (DZero [3]) and observations (SDSS [4]) will soon surpass the ability of storage systems to store and retrieve data in a reliable and cost-effective manner. While the capacity, performance and the mean time to failure (MTTF) of a single disk has been improving, large-scale storage systems and parallel file systems (PFS) can comprise tens of thousands of drives, thus bringing down the overall mean time to data loss (MTTDL) of the entire system to unacceptably low levels. For example, the Lustre-based Spider PFS of the Jaguar supercomputer (No. 3 machine on the Top500 [5] list) comprises 10,000+ disks [6]. An exaflop machine in 2018 is projected [7] to host hundreds of thousands of drives to support the desired I/O throughput. The "law of large numbers" in this case only reiterates that failure will be a norm and not an exception. The reliability and robustness of the I/O system is crucial to large-scale applications that generate and analyze terabytes of data. Trends from commercial and HPC centers suggest, that on average, 3% to 7% of disks fail per year [8]. Thus, storage systems are a significant contributor to system failure.

To increase the fault tolerance of storage systems, disks in each storage server are usually combined in a RAID (Redundant Array of Independent Disks) array that provides some level of redundancy. For example, RAID-6 contains $k + m$ disks, with k data disks and $m = 2$ parity disks. The array can recover from simultaneous failure of up to m disks. Data reconstruction time (in the event of a failure) is proportional to the drive size, load and the number of drives in the RAID group, and is in the order of a few hours even for standard disk sizes. For example, the reconstruction time for a 2 GB disk is approximately 30 minutes [9]. During reconstruction, applications achieve degraded I/O rates, at best. This is only bound to get worse with large-scale storage systems.

Historically, RAID has been implemented in hardware because of its high throughput compared to a software based solution. Hardware RAID controllers, unlike the rest of the storage system, rely on proprietary hardware. These non-commodity parts are usually expensive, receive infrequent software upgrades, and can become a bottleneck, especially during degraded array reconstruction. The high cost of sophisticated embedded RAID controllers — a typical 16 TB RAID setup could easily cost in excess of $15000 [10] — also implies that such solutions are beyond the reach of mid-sized institutions, small-scale clusters, and storage systems with limited provisioning budgets. Even with supercomputing centers, where the cost of provisioning and operating a scalable, reliable storage system can run on the order of millions of dollars, there is a need to reconcile the storage cost against the FLOPS purchased, as machines are often ranked in terms of peak FLOPS. Thus, providing a desired level of reliability and redundancy for the storage system under a given budget constraint is always a challenge, be it in supercomputing centers, mid-sized or small-scale systems.

In recent years, GPUs from NVIDIA and AMD have shifted from closed peripherals used to render graphics images to inexpensive commodity parallel accelerators. They provide general-purpose APIs that can be used to accelerate many types of computations. While currently GPUs are mainly used in scientific workloads, recent studies have explored applying them to I/O workloads [11]–[15]. These efforts have shown that GPUs can be used effectively for parity computation using Reed-Solomon coding [16] as well as other I/O workloads, such as hashing [17].

Further, large-scale machines are beginning to be provisioned with GPUs. For example, the state-of-the-art Keeneland supercomputer [18], is a combination of Intel Nehalem and NVIDIA Tesla GPUs. Similarly, a planned 20 petaflop machine for 2012, Titan [20], will use a hybrid architecture, with each node featuring two 16-core AMD Opteron processors and two Tesla X2090 GPUs. Additionally, GPUs provide a cost-efficient solution compared to general purpose CPUs (GPPs), especially when GPUs are coupled with a few GPPs [21].

Thus, GPUs are quickly being adopted in a myriad of fields, ranging from scientific workload processing [22] to education in the developing world [23]. These architectures present opportunities to explore the utility of GPUs towards improving storage system reliability.

In this paper we propose a novel way to utilize low-cost GPUs in conjunction with a PFS to provide fault tolerance and end-to-end data integrity. We capitalize the resources provided by the PFS, such as striping individual files over multiple disks, with the computational power of a GPU to provide flexible and fast parity computation for encoding and rebuilding of degraded RAID arrays. We attain end-to-end data integrity by performing encoding and decoding at the compute node, where data is produced and consumed. We implement our client-driven, per-file RAID in the widely used Lustre PFS [24], which will facilitate wider adoption of our system.

Specifically, this paper makes the following contributions:

- We parallelize and accelerate two state-of-the-art minimum density RAID-6 coding schemes – Blaum-Roth [25] and Liberation codes [26] – on GPUs using CUDA.
- We analyze the coding process and extract fine and coarse-grain parallelism. We leverage both types of parallelism to maximize the acceleration at GPUs.
- We present a novel system design with client-driven, per-file parity computation accelerated by a GPU. Unlike traditional hardware-based approaches it is flexible, as it can transparently switch between a per-file RAID-1 for small files to a desired sized RAID-6 as the file grows, and provides end-to-end data integrity guarantees.
- We show through a prototype implementation that client-side parity generation using a GPU imposes an acceptable overhead on the overall client performance and achieves a coding throughput of over 3.0 GB/s on each client, thus saturating the network.

We evaluate our system using a medium-scale cluster based on nodes with off-the-shelf, GPUs and show that our approach: provides a customizable interface for an application to tailor the RAID array parameters and provides default values to support legacy applications. The results demonstrate that leveraging GPUs for I/O support functions, i.e., RAID parity computation, is a feasible approach and can provide an efficient alternative to specialized-hardware-based solutions.

II. BACKGROUND

A. Rationale

A PFS typically provides fault tolerance at the storage backend. For example, the data drives on each storage server are arranged in a RAID-5 (or higher) configuration. An alternative of computing parity at the backend is client-driven, per-file RAID [27]. In this section, we highlight the potential benefits and drawbacks of this approach, and make the case that such a framework is a good candidate for GPU acceleration.

1) Backend vs. Client-driven Parity Generation: GPUs are becoming ubiquitous, with good performance and flexibility features. Modern HPC clusters and supercomputers are being equipped with GPUs. In such settings, a client-driven parity generation can utilize the GPU resources already available on the client machines. Such a trend will also be supported by emerging technologies such as the Intel Sandy Bridge chip, which supports an integrated GPU and CPU [28]. Furthermore, client-driven parity generation allows for unprecedented flexibility. For example, the parity computation power of the system is not constrained by the hardware at the backend, and can be changed dynamically with the number of clients.

Hardware RAID controllers typically require all disks in an array to be co-located on the same blade. This can result in data loss because all the drives in the array can fail simultaneously, due to power failure, over-heating, etc. A client driven per-file RAID system does not impose any spatial limitation on the locality of drives, allowing data to be spread across the system and not just one location. Furthermore, the ability of each client to generate parity opens the door for end-to-end data integrity checking. Typically, data has to pass through several network interconnects, and memory and storage hierarchies, all of which can introduce errors, albeit with very small probability. If absolute data integrity is required, the client can choose to obtain parity as part of a read operation and check consistency of the data on the fly.

2) Block-based vs. Per-file RAID: When compared to block-based RAID, a per-file RAID scheme will allow each file or directory tree to have a desired fault tolerance level. For example, small files can use a simple RAID-1, while large ones can use the state-of-the-art RAID-6 code. In a block-based RAID, it is difficult or impossible to directly map any lost sectors back to higher-level file system data structures. In fact, it has been recently argued [29] that such factors will continue to diminish the utility of simple block-based RAID. In a conventional hardware RAID, a single RAID controller is responsible for parity coding. For a large array, that can mean hours until the array is rebuilt — during which time, an unrecoverable read error (URE) can occur, potentially causing the entire array to fail. Using a software RAID, rebuilding of the array can be done in parallel. A number of machines at the backend can be equipped with GPUs and rebuilding of separate files can be farmed out to different machines.

3) Hardware vs. Accelerated Software RAID: Direct performance comparison between a hardware RAID controller and a GPU is hard to quantify, given that the two carry very different hardware. In the context of our approach, however, such a performance comparison is not necessary. For one, unlike a hardware RAID, the GPU in our system resides on the client and as such its available throughput to the storage drives is limited to the network throughput of the client. We show that a GPU can sustain encoding throughput that exceeds available network bandwidth even if the client is connected over 10 Gbps interconnect. A CPU alone, however, is not enough to meet such performance requirements. Previous research has shown that unlike GPUs, conventional x86-based processors are slow in performing a large number of finite field multiplications – the majority of operations required for parity generation [16].

One area that a GPU has a clear advantage over a hardware RAID solution, however, is programmability. The best fault

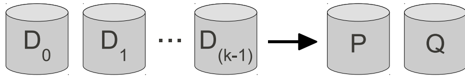

Fig. 1: Logical overview of a RAID-6 system.

tolerance that a hardware RAID controller typically supports is a Reed-Solomon [30] implementation of RAID-6. In contrast, any number of coding techniques can be used in a software solution, including triple parity RAID or any implementation of RAID-6. The programmability of the GPUs, thus, provides a unique opportunity to exploit the advances in parity encoding, such as minimum density coding schemes like Blaum-Roth [25] and Liberation codes [26].

4) Discussion: Unprecedented flexibility and increased fault tolerance do not come for free, of course, because more data has to move over the client-server interconnects. Moreover, parity generation can be computationally expensive and thus a burden to the clients. We address the later by offloading the computationally expensive codes to a GPU and show that in doing so we introduce acceptable overheads to the client systems. While some increase in data traffic is unavoidable in a client-driven approach, modern PFSs, like Lustre, maintain large caches on the client side, which absorb a large portion of parity modification caused by frequent small writes. Thus, a large portion of parity updates never hit the interconnect. Even in the context of frequent large writes that exceed client caches, our system provides enough flexibility to address the increase in traffic. Applications can set their own operating point with respect to data reliability and I/O performance. For example, by switching a file to a RAID-5 from a RAID-6, an application achieves a 2× decrease in network traffic due to parity. Another approach is increasing the per-file RAID array size. For example, moving from a (8,2) to a (16,2) RAID-6 array, drops parity from 20% to 11% of overall data. Normally a hardware RAID array is not larger than 16 drives, because increasing its size would result in unacceptably long time to rebuild it. As per-file arrays can be rebuilt in parallel in a client-driven approach, an application can set its desired array size based on network traffic and GPUs available to rebuild the array.

B. Enabling Technologies

In the following, we describe the enabling technologies that are used in realizing our GPU accelerated software-based RAID-6 distributed PFS.

1) Erasure Codes: In recent years, RAID-6 systems have become increasingly important as they can tolerate a complete failure of one drive occurring in combination with a latent failure of a block on a second drive. Such a failure scenario would result in a permanent data loss on a RAID-5 system. Unlike RAID-1 through RAID-5, which provide exact data encoding techniques, RAID-6 is only a specification and as a consequence there are a number of available coding techniques. The recently introduced Liberation codes promise to become a standard for RAID-6.

A RAID-6 system (Figure 1) is composed of $k + 2$ data nodes and can tolerate the failure of any two devices. Devices D_0 through D_{k-1} can each store B bytes, whereas the remaining 2B bytes are in the P and Q coding devices. The P device is calculated to be the parity of all data devices, while the implementation of the Q device is left to the designer, with the sole constraint that it cannot hold more than B bytes and the resulting system must be able to recover from the failure of any two devices.

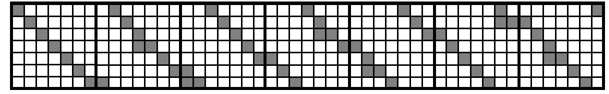

Fig. 2: Bottom row of BDM used to compute parity for the Q device for a system with 7 devices and word size of 7. Gray boxes represent a 1, white a 0.

Liberation coding (Figure 2) is similar to Cauchy Reed-Solomon coding [31]. The system splits each data device into w words and uses a $w(k + m) \times wk$ matrix to perform the encoding, where k and m represent the number of data and encoding devices respectively. For all RAID-6 techniques, the value for m is two. All operations are performed in Galois Field (2), where addition and multiplication are bitwise XOR and AND operations, respectively. The matrix is called a Binary Distribution Matrix (BDM) and each element is either one or zero. BDM is multiplied by the vector representing device bits, to produce a vector representing the data and encoding devices. The BDM is quite restricted as the top $k(w \times w)$ portion of the matrix is the identity, $D_{0,1}$ through $D_{0,k-1}$ are also identity matrices that produce the P device, and the bottom row can be customized as per rules laid out in [31].

The encoding matrices for the Liberation codes are shown to be optimal or close to optimal. The decoding matrix is produced by inverting the portion of the encoding matrix that corresponds to the data devices that are still active. However, the resulting matrix typically has far more $1s$ than optimal and in some cases it is more efficient to calculate a word in one of the failed devices from a previously computed product, rather than from the original BDM matrix by data vector product. To take advantage of this, a schedule is created from the BDM that does the least number of XORs. The optimized schedule produces a significant speedup for decoding. A schedule can also be used in the encoding process as it is a more compact representation of the operations than the BDM itself [26].

2) The Lustre Parallel File System: Lustre [24] is a storage architecture for Linux-based clusters and provides a POSIX-compliant UNIX file system interface. It is best known for powering seven of the ten largest HPC machines worldwide, with thousands of client systems, petabytes of storage and hundreds of gigabytes per second I/O throughput. Many HPC sites use Lustre as a site-wide global file system, serving dozens of clusters on an unprecedented scale, e.g., the Spider file system [32]. A Lustre file system comprises of the following key components: *Client*, *MDS* (MetaData Server) and *OSS* (Object Storage Server). Each OSS can be configured to host several *OST*s (Object Storage Target) that manage the storage devices.

The Lustre client that runs on the compute nodes of the

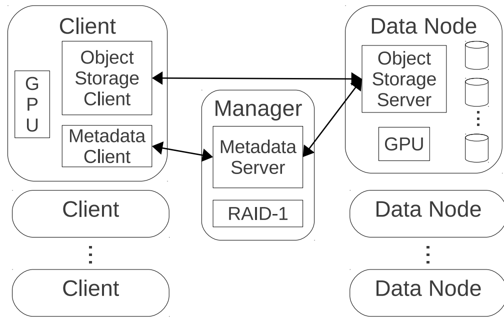

Fig. 3: High-level architecture of the GPU-enabled RAID system.

cluster communicates with the MDS to obtain privileges and layout for a given file. Once file metadata has been received, the client is able to directly communicate with the OSTs that house the objects associated with the file. An important feature of the Lustre file system that we exploit in our design is its ability to store files in multiple same-sized objects striped over multiple OSTs. Moreover Lustre provides extensive management and recovery features that are useful in identifying the files affected in the event of an OST failure. Thus, Lustre provides some key building blocks to turn each file into its own RAID array.

Lustre also supports hot-swappable hard-drives on each OSS. In the case of a disk failure, a new disk can easily replace the failed disk. Upon a mount, the Lustre manager node detects and recreates the objects that were present in the failed disk. During the per-file RAID array rebuild process, our system restores the data in the lost objects, while reusing the objects that have not failed.

3) KGPU: Recent research has explored the potential of GPUs to accelerate computationally intense OS operations [33], [34]. The current state-of-the-art NVIDIA's and AMD's proprietary drivers do not support accessing the GPU from kernel space, therefore all these efforts rely on a userspace daemon to execute the GPU requests.

We use KGPU [33] in our implementation because of its service oriented approach and associated low latency. In contrast to the standard approach, where both data and kernel code are copied to the GPU before each execution, KGPU substantially decreases the latency of a GPU kernel launch by keeping the kernel alive even after it has completed its execution. KGPU incurs full latency only when a GPU kernel that provides a different service needs to be loaded. In our implementation, all RAID-6 array sizes are processed by a single kernel, therefore KGPU never incurs the extra latency of loading and unloading a kernel.

III. Design

In this section, we present the design of our GPU-enabled RAID system and its realization within the Lustre PFS [24]. We also describe the use of KGPU [33], a GPU management framework, in our system.

A. System Overview

A high-level overview of the hardware and software components used in our system is shown in Figure 3. The *Data Nodes* serve as the main storage components; the *Client* provides the user-side interface to the system; and the *Manager* directs and facilitates the interactions between all components. All system components are tightly integrated with the Lustre PFS. The clients typically run on the GPU-enabled compute nodes of the cluster. All or a subset of the Data nodes are equipped with a GPU to perform parity computation during a RAID array rebuild. This hardware addition is feasible on many deployments, since modern motherboards typically have a built-in PCI-Express (PCIe) slot. For the setups where installing a GPU on Data nodes is not an option, the array rebuild process can be offloaded to idle client machines. Each Data node runs an *Object Storage Server* (OSS), which provides file I/O services and network request handling for all the *Object Storage Targets* (OSTs). The OSTs manage the disk drives that store chunks of files called objects. A file in the Lustre PFS can be striped over any number of equally sized objects.

In our design, the Manager is equipped with a hardware or software RAID-1. The Lustre guidelines suggest using RAID-1 or RAID-1+0 for the disks on the Manager, which efficiently performs frequent updates on small metadata files. The Manager runs a MDS that only stores metadata (such as file names and layout, directories, and permissions), which generally accounts for only 1% of the total storage capacity of the system [35]. This ensures that only a small number of disks are required to store the entire metadata in a typical deployment. Hence, equipping the Manager with a low-end RAID-1 controller with a small number of ports (or utilizing a software RAID) fits with our overall goal of achieving fault tolerance with minimal cost.

Each client node in our design is equipped with a programmable GPU that is used to accelerate the file encoding and decoding process. Each client node runs a *Metadata Client*, which communicates with the MDS at the Manager to serve all directory and file operations, such as opening and closing, on behalf of the client. Each client also runs an *Object Storage Client*, which interacts with the OSS at the Data node to read and write to the file objects in parallel. This enables the client to bypass the Manager for all subsequent read and write operations after opening a file and receiving its layout on the Data nodes.

We use the fault tolerant Manager to "bootstrap" the per-file RAID-6 arrays created by our system. If an OST device fails, the Manager identifies all the surviving objects of a given file, which are then used to reconstruct the lost objects.

B. RAID-enabled PFS Design

One of our key design objectives is to make our system compatible with Lustre so that it can be easily integrated with extant Lustre deployments. To this end, our first design choice is to keep the Lustre backend software infrastructure (Manager, OSS, etc.) intact and limit our software-level modifications to the client nodes.

One design obstacle for integrating parity acceleration on the client-side is that the NVIDIA CUDA toolkit is designed to run in user-space, while the Lustre client is implemented as

a kernel module. One option is to augment `liblustre` [36], a user-space implementation of the Lustre client, to handle parity generation and storage. This approach decreases the number of context switches that are otherwise required to send data between the client module and the GPU. However, `liblustre` is not widely used in practice as it does not support many performance enhancing features of the kernel implementation, including client-side caching and the support for multi-threaded applications. Hence, we integrate all parity generation inside the Lustre client module and use KGPU to access the GPU directly from kernel space. We implement parity encoding and decoding as a service provided by the user-space component of KGPU.

Another challenge is to find the appropriate location to transparently store the extra parity information. One option is to create a separate "shadow" parity file for each file. This is promising, especially in Lustre, where a file can be striped over any collection of OSTs and by ensuring that the shadow file is stored on different OSTs from the OSTs containing the actual file contents, we can provide a complete RAID-6 array. Moreover, the data file and its attributes remains intact and can be accessed without modification. However, this approach doubles the number of files in the storage system and may introduce a bottleneck at the manager node. Additionally, updating the parity would require write locks on two different files simultaneously and would complicate the locking procedure. An alternative approach that incurs minimal bookkeeping overhead is to interleave data and parity in the same file. However, utilizing this approach requires an effective mechanism to hide the parity from the user. To this end, we modify all file and inode operations that can expose the parity information, such as `write`, `read`, `seek`, `get` and `set` attribute. For operations such as `seek`, and `get/set` attribute, we perform a translation between the size of the actual file including the parity and size of the data. The bulk of the parity generation modifications are contained in the `write` call.

An important feature of our system that significantly decreases overheads when writing to small files is that as long as a file is smaller than a single object it is configured as a RAID-1. We achieve this by mirroring each write into the first parity object, while keeping the second one empty. If a write anywhere outside the first data object is submitted to the system it automatically locks all stripes and converts the file to a RAID-6 array. Note that while in the RAID-1 state no extra space is wasted as the empty blocks inside the second parity object are never written to disk. To maintain consistency we lock the parity object instead of the data, which ensures that a concurrent write to any portion of the stripe, would conflict with the current write and thus will be properly serialized. In this RAID-1 state the GPU is completely bypassed eliminating the expensive read-modify-write step.

C. Control Flow

We now describe the interactions between the different components of our system and how they come together to realize the flexible RAID-6 solution.

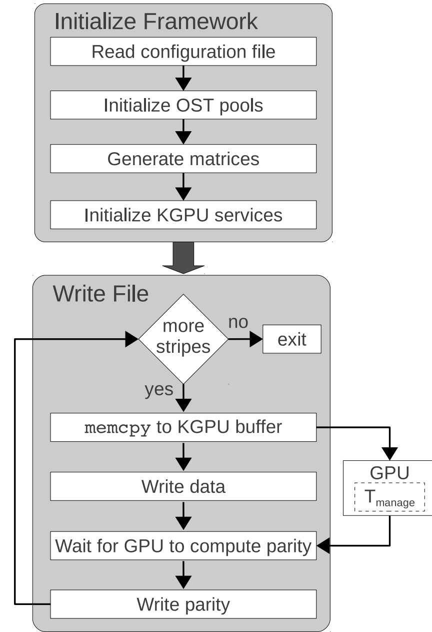

Fig. 4: Control flow in our GPU-enabled RAID system.

Figure 4 illustrates the control flow between different components of the system. The system is initialized by reading a configuration file, which specifies different architectural and RAID array specific parameters, such as available GPU memory, maximum supported file object size, and maximum number of disks a file can be striped over. These parameters are used to initialize global defaults, such as the coding bit matrix used in the default parity algorithm. Some of these parameters are passed on to the KGPU framework, which spawns a GPU management daemon, T_{manage}, that we later utilize to compute the parity. T_{manage} initializes its *request* containers and allocates their associated buffers. The daemon then waits for the jobs to be submitted to the request queue.

The Manager initializes the appropriate storage pools before the Lustre file system can be mounted. In Lustre, any OST can be assigned to a number of storage pools, which we use to define default RAID arrays in our system. Storage pools can be modified at runtime to support addition or removal of storage devices. Once Lustre is mounted on the client, the root directory is assigned to the default storage pool. Files and directories created under the root are recursively assigned the default pool. On creation, each file receives a randomized order in which to write to the OSTs in its pool, which ensures that parity is spreads around the OSTs. In addition to the given defaults, applications have full control to assign files and directory trees to any other pool using standard Lustre system calls.

The bulk of the operations are performed during a write. Lustre caches data on the client side and as a consequence most data writes are processed asynchronously. Synchronous I/O is triggered when the Lustre cache fills up. Lustre breaks down the write in a loop based on the object size. In each iteration, the client asks for a lock on the object and proceeds

to update the object, releases the lock, and moves on to the next object. In order to ensure consistency of the parity during simultaneous writes to the same file stripe, we acquire a lock that spans all of objects in a stripe. Thus, we increase the granularity of Lustre's locking from an object to a stripe of objects. Note that we still allow multiple clients to be simultaneously reading and writing to the same file, as long as it is to a different stripe.

After acquiring the lock on a stripe, we copy the relevant portion of the write buffer to CUDA page-locked buffer previously initialized by KGPU and send a request to the KGPU module. The copy is required to maximize the PCIe bandwidth utilization ensured by the CUDA page-locked buffer. The request is then forwarded to our parity generation service implemented in the KGPU user-space daemon that interacts with the GPU to compute the parity for the buffer and return it to the caller. KGPU's call is asynchronous with the data write to the Lustre cache and for a full stripe write completes before it, thus hiding all the latencies associated with moving data to and from the GPU and computing parity. The only overhead exposed is due to the `memcpy` call and parity write. We quantify these latencies in our evaluation.

A read operation also acquires the lock in a loop. In the common case when only data is read, the read loop skips over the parity objects in each stripe of the file. However, a user can also read parity along with the data to ensure end-to-end integrity. In that case, locking is again done at the granularity of the stripe and data and parity is sent to the GPU for validation. If data corruption is detected the read call is restarted. However, if the call fails again an error is returned, as it indicates a permanent error in one of the system components.

D. Degraded Array Reconstruction

Unlike a conventional hardware RAID controller, our system is capable of utilizing multiple GPUs to reconstruct a degraded array. If a disk fails, it can be replaced manually or via a hot spare. The disk is formatted if necessary and assigned the same internal Lustre ID as the failed disk. When the new disk is mounted, the Manager recreates all the missing objects and relinks them to the file objects on the surviving disks. Next, the system requests a list of files that have been affected, and based on the location of the failed disks and the availability of GPUs, mounts a Lustre client on the machines to reconstruct the lost objects. The list of affected files is then split accordingly and forwarded to the reconstructing clients to rebuild the affected files in parallel.

IV. Implementation

We have implemented our system as described in Section III using 1272 lines of C/C++ and CUDA code. The implementation runs on Linux (kernel version 2.6.32) and is portable to CUDA-enabled GPUs. We based our parity generation implementation on the definition of Liberation Codes [26], which is provided in a freely available library, called *Jerasure* [37]. *Jerasure* provides a single threaded implementation for both Liberation and Blaum-Roth functionality.

Our analysis of Liberation and Blaum-Roth codes' single threaded implementation revealed that more than 95% of the time is spent in the function that performs the XOR operations. We also noted that the same function is used for both encoding and decoding, with the only difference being the schedule. Furthermore, the work done in this function has the potential for both coarse and fine-grain parallelism, making it a good candidate for offloading to the GPU. Therefore in our implementation, we offload only XOR operations on the data to the GPU to maximize SIMD parallelism. Note that most of the other operations in the coding process, such as creating the BDM and converting it to a schedule are computed once and sent to the KGPU service at initialization. As these operations are at most quadratic in the number of drives, which are usually in the tens in a typical RAID array, the overhead for these tasks is negligible.

A. Basic GPU Implementation

As described earlier, a schedule is derived from the original BDM matrix while performing the XOR operations on the given data or while copying it between different devices. The schedule is a two dimensional array of integers of size $5 \times N$, where N is the number of operations that need to be performed for encoding. The operations defined are XOR or `memcpy`. The five integers in each tuple identify which words will be operated upon. The first two integers identify the id of the device and the word that will serve as source, while the next two identify the destination. The last integer is either 1 for XOR or 0 for `memcpy`. For example, the operation $< 00700 >$ can be interpreted as the first word of device 0 is to be copied over the first word of device 7. In the case of encoding, the schedule is used to compactly represent the BDM. In the offloaded function, the schedule is also flattened to a single dimensional array for easier copying of the data from the host to the GPU memory. Furthermore, since this data is relatively small and does not change during GPU kernel execution, it is copied directly to the GPU's constant memory to enable faster access by the GPU threads. The kernel iterates over all the operations in the given schedule and each thread performs an XOR or `memcpy` operation on the corresponding words (in 4 byte chunks) in parallel. Hence, the amount of parallelism exposed depends directly on the word size, which is determined by the size of each data object.

B. Optimizations

The main drawback of the basic GPU port is that it reveals only the fine-grain parallelism that is present within a scheduled operation. We analyzed data dependencies and found that entire operations can be done in parallel as well. Specifically, the schedule produces the $2w$ words of the coding devices of a RAID-6 array, where w is 8 and 16 for the Liberation and Blaum-Roth coding, respectively. All the operations associated with computing a single word in a coding device can be performed in parallel with the ones that encode the rest of

```
__constant__ int d_schedule[8192];
__constant__ int d_num_reads[1024];
__constant__ long d_data[64];

__global__ void xor_gpu(int packetsize,
                        int blocksize, int k, int w) {
  int dest, source, i, y = (k + 2)*2*blockIdx.y;
  const unsigned int g_tid = blockIdx.x*blockDim.x +
                             threadIdx.x;
  int *dptr = (int *)((char *)d_data[d_schedule[y]] +
                      d_schedule[y+1]*packetsize);
  int *sptr = (int *)((char *)d_data[d_schedule[y+2]] +
                      d_schedule[y+3]*packetsize);
  dest = sptr[g_tid];
#pragma unroll
  for(i = 4; i < d_num_reads[blockIdx.y] * 2; i += 2) {
    sptr = (int *)((char *)d_data[d_schedule[y+i]] +
                   d_schedule[y+i+1]*packetsize);
    source = sptr[g_tid];
    dest = dest ^ source;
  }
  dptr[g_tid] = dest;
}
```

Fig. 5: GPU parity computation kernel.

the words. To exploit this, we modify the schedule and create our optimized port shown in Figure 5.

We create a two dimensional grid of thread blocks and assign each of the $2w$ rows of blocks to perform the operations associated with one of the encoding words. We use an additional structure, `num_reads`, to store the number of operations needed to compute each of the $2w$ coding words. This enables the kernel to execute fewer iterations compared to the basic port, thus simultaneously reducing the work of each thread and exposing more parallelism.

V. Evaluation

In this section, we present the evaluation of our GPU-enabled RAID system. We first describe our testbed, and then present the I/O measurements of our system. The goal is to show the impact of different design parameters and features, such as RAID stripe size and end-to-end integrity checking, on the overall system performance. Next, we evaluate the performance of RAID array reconstruction. Finally, we quantify performance under a real workload.

A. Experimental Setup

We have set up a Lustre cluster, consisting of one Manager node and three OSSs, each with six OSTs. The Lustre server machines are identical with four Opteron quad-cores each, and 64 GB of main memory. Additionally, each OSS has a GeForce 9500 GT GPU with 1 GB of graphics memory connected to an 8× PCIe slot. Our client machine has two Intel Xeon quad-cores, 48 GB of RAM and a Tesla C2070 GPU with 6 GB of GDDR memory. All the machines are connected using a dedicated Gigabit switch. We use Lustre patched Linux 2.6.32 kernel, Lustre 1.8.5, and CUDA SDK 4.0.

B. I/O Throughput Measurement

1) Raw Throughput: We first measure the raw write throughput that our client machine can achieve. Figure 6 compares the throughput of writing a file striped over 16 OSTs with a stripe/block size of 1 MB, denoted as *data write*. The file size ranges between 16 MB and 2 GB. A Lustre client

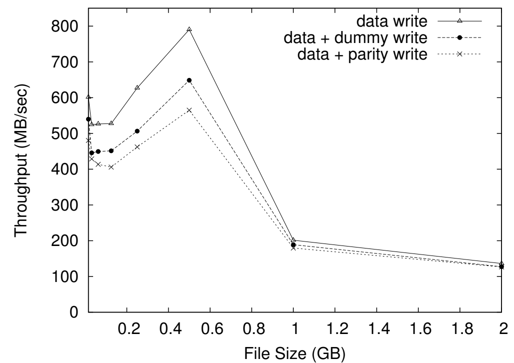

Fig. 6: Write throughput for a file striped over 16 OSTs + 2 parity OSTs.

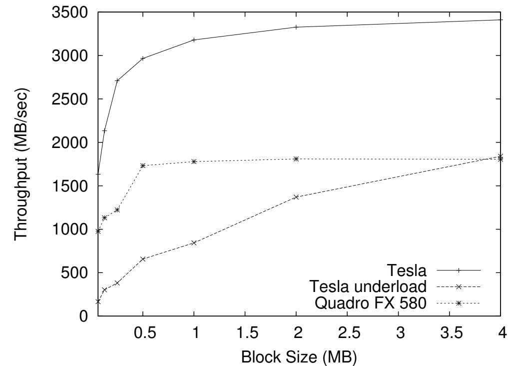

Fig. 7: GPU encoding throughput.

maintains a 32 MB local cache per OST, which is flushed periodically. If a write submitted by a client fits in the cache, the Lustre module returns from the write immediately after the write buffer is written to the cache. If there is no space left in the cache, it is flushed to the corresponding OST and the write returns after the write buffer has been written to the Lustre back-end. Since we are writing to 16 OSTs, the combined available cache is 512 MB, and consequently writes smaller than 512 MB exhibit throughput higher than the theoretical throughput of Gigabit Ethernet. The throughput of files larger than the cache quickly levels out to an effective available bandwidth of around 125 MB/s.

In the Figure, *data + parity write* curve shows the throughput when our RAID encoding system is turned on. In this case, the same data as in the base case is striped over 16 OSTs and concurrently the parity is generated and written to the remaining 2 OSTs. As a point of reference we also include a *data + dummy write* curve, which generates the same traffic as the RAID encoding, without computing the parity.

Writes that fit into the Lustre cache exhibit a very high throughput and as a result the overhead of `memcpy`-ing data into KGPU buffers results in around 10% overhead (difference between *data + dummy write* and parity *data + parity write*). The rest is attributed to copying the extra parity ($1/8^{th}$ of data in this case) to the Lustre cache. It is important to note that in this experiment all the overhead associated with parity generation remains completely hidden from the application.

2) Encoding Throughput: Next, we evaluate the parity encoding throughput delivered by the GPU. We measured throughput delivered by our high and low-end GPUs, which

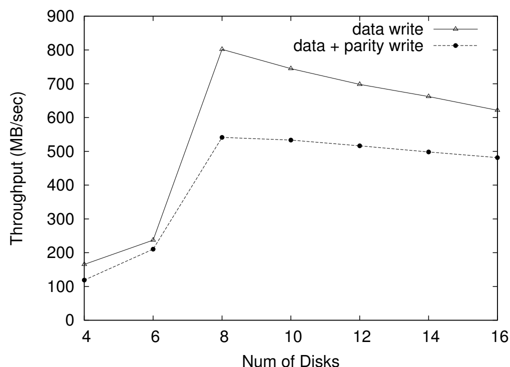

Fig. 8: Effect of number of disks on throughput (file size = 256 MB).

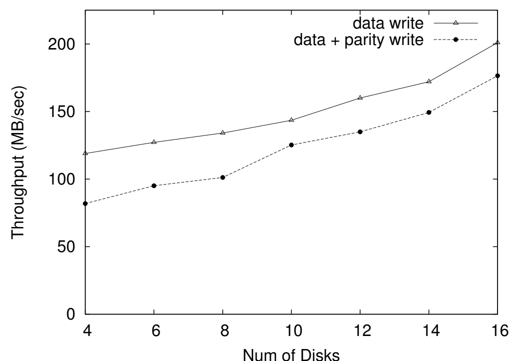

Fig. 9: Effect of number of disks on throughput (file size = 1024 MB).

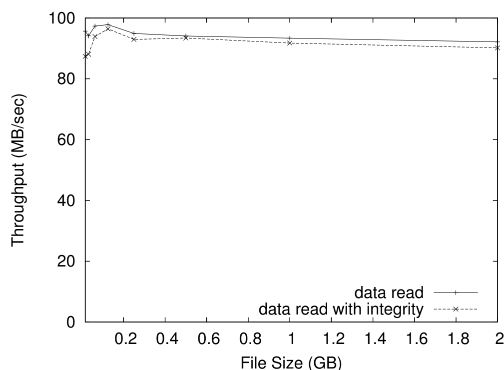

Fig. 10: Read throughput with end-to-end data integrity.

includes moving the data to and from the GPU's memory as well as actual parity computation on the GPU (Figure 7). A low-end GPU can deliver encoding throughput around 1 GB/s for 512 KB files, which quickly increases to 1.7 GB/s for files large than 8 MB. The Tesla GPU delivers encoding rates of 1.6 GB/s for 512 KB files and in excess of 3 GB/s for files larger than 8 MB. Therefore, our system using a low-end GPU can generate parity faster than the speed at which Lustre commits the data to its caches. As parity is encoded asynchronously with the commit to cache, the overhead of generating it remains hidden.

We also include the encoding rates of the Tesla GPU, while it is under heavy load from an N-body simulation. We used the N-body simulation from the CUDA SDK and ran multiple iterations in the benchmark mode using the default number of objects based on the specifications of the Tesla GPU. Even under heavy load, the Tesla GPU delivers sufficient throughput for all but the smallest files, for which the encoding overheads are exposed to the system as increased latencies due to the heavy load. However, the performance of the simulation is unaffected by the parity generation kernels and remains constant at 484.650 single-precision GFLOP/s. This is because Tesla GPU can perform efficient context switches at the kernel boundary and asynchronous data transfer for different contexts can run simultaneously. Therefore the parity data can be transfered to the GPU, while the simulation kernel is running and vice versa. Moreover, the parity generation kernels complete 2-3$\times$ faster than the simulation kernels.

It is important to note that not all background loads on the GPU have the same effect on the parity generation. An iterative workload with kernel execution times in the order of milliseconds, such as the N-body simulation that can be rendered in real time, would not block the parity kernels and cause an unacceptable slowdowns to our system. However, for GPU kernels with execution times in seconds, alternative techniques such as "context funneling"[1] can be used to minimize the overhead. The down side is that the GPU workloads need to be modified, e.g., as a KGPU service. This enables even a long running kernel to run parity generation concurrently.

3) Impact of Number of Disks on Throughput: Next, we study the effect of number of OSTs that a file is striped over on the write throughput of our system. Figure 8 shows the baseline write throughput and the throughput of our system for writing a 256 MB file. When the file is striped over 6 OSTs or less, it cannot fit in the client caches and as a result, raw network bandwidth is exposed to the application. If the 256 MB file is striped over more than 6 drives, parity is cached and flushed after the write returns. As the write fits in the caches, throughput levels out. As the file size remains constant splitting and committing it to more caches becomes less efficient, which causes the slight dip in the throughput.

Striping does not have such an effect when writing a 1024 MB file as the file and its parity cannot be cached (Figure 9). In the *data write* case, writing to more drives achieves better throughput because of efficient bandwidth utilization when the file is striped over all available OSTs. For the *data + parity write* case, decreasing the number of drives has the effect of decreasing the length of the RAID 6 array, e.g., striping data over four disks produces a (4,2) RAID 6 array where four objects in a stripe are data and the rest are parity, having a parity overhead of 50%. Increasing the array length decreases the relative size of the parity, e.g., in a (16,2) RAID 6 array parity is 12.5%. Thus, for the *data + parity write* case, there is a linear increase in throughput available for data with the increase in the array length.

4) End-to-End Data Integrity: One of the important features of our system is that it can provide end-to-end data integrity checks for the I/O operations. Figure 10 shows the achieved read throughput when the end-to-end integrity check is enabled. To ensure that data is not corrupted on the disk or

[1] Context funneling uses advanced features of the Fermi architecture to execute concurrent kernels, which must be launched from the same context [38].

Data Size (GB)	1 Node	2 Nodes		3 Nodes	
	Time (s)	Time (s)	Speedup	Time (s)	Speedup
5	46	30	1.53	25	1.84
25	237	151	1.57	125	1.90
50	493	345	1.43	258	1.91

TABLE I: RAID reconstruction time and normalized speedup with respect to 1 Node.

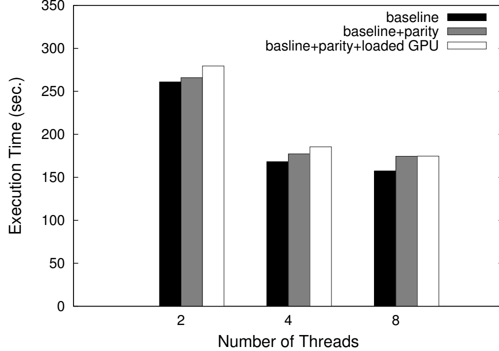

Fig. 11: Performance of NAS DC benchmark.

on the network interconnects, one of the parity objects in each stripe is read along with the data. Both data and parity needs to be sent to the GPU for verification to successfully complete the `read` call. For files with single stripes, the synchronous call causes an overhead of 9%. However, when reading files with more than one stripe, the parity check for each stripe is performed in parallel with the read of the next stripe, resulting in a negligible overhead of 2%.

C. RAID Reconstruction Cost

It is critical to minimize the degraded RAID reconstruction time for maintaining the integrity of data, as the system is exposed to unrecoverable read errors during the reconstruction process. Table I shows the reconstruction time for rebuilding 20, 100, and 200 degraded files with a combined size of 5 GB, 25 GB, and 50 GB, respectively. Files are striped on all 18 OSTs (16 for data and 2 for parity). As the RAID arrays are defined per file, their rebuilding can be distributed between the available machines, resulting in a speedup of close to 2× when reconstructing for the 200 files case. During this test, we use all the machines in our setup, which results in the utilization of all the available network bandwidth. It is important to note that each of our low-end GPU achieves an effective reconstruction rate of 1.5 GB/s, therefore even higher speedup is possible, if network and disk throughput permit it.

These results show that our GPU-enabled RAID solution is feasible, and provides a configurable and flexible solution.

D. Impact on Applications

Next, we examine the performance of our system under load by a real-world application, the Data Cube (DC) NAS OpenMP [39] benchmark. DC performs a data-intensive operation known in data mining as the Data Cube Operator (DCO), which computes views of a dataset represented as a set of `n` tuples and involves O(log n) memory accesses per tuple.

Figure 11 shows the performance of DC executing on our client machine with varying number of threads. It is configured to write out the views to disk as they are computed, thus stressing both memory and the storage subsystem. At two threads the benchmark is actually CPU-bound, thus generating and writing out the extra parity for each view introduces a small additional slowdown of 2%. Beyond four threads the benchmark becomes I/O-bound and as a result, the overheads due to parity produce a 5% and 10% slowdown for four and eight threads, respectively. We also measured performance of our parity generation system under a heavy background GPU load produced by an N-body simulation application running on the GPU. When the DC benchmark is running with eight threads, the background job does not affect performance at all, because our system is able to schedule the workload for the eight threads more effectively. With fewer threads requesting parity, the system cannot obtain enough time on the GPU and as a result exposes parity generation overheads to a portion of the write operations, resulting in a slowdown of around 5%.

VI. Related work

There are a number of parallel file systems that were built from the ground up to withstand failure, such as ZFS, Ceph, and Panasas [27], [40], [41]. ZFS maintains data integrity by using checksums for on-disk blocks, while Ceph relies on replication at the granularity of an entire object storage device, and thus both file systems cannot detect errors introduce during network transmission to the client. Panasas [27] offers a commercial solution with features similar to the system presented in this paper. However, it relies on the CPU to generate a per-file RAID-5 array. It does not support RAID-6, since generating the required parity without a hardware accelerator would result in high overheads.

Utilizing GPUs as commodity accelerators for general purpose applications has been on the rise [42]. stdchk [43] uses hashing to detect content similarity between two successive checkpoint images. Several efforts [44]–[46] have attempted to improve the performance of such hash computations by offloading them to the GPU. Similarly, GPUs have also been used to accelerate parity computation [47] and data encryption [48], [49] for storage systems.

The work most similar to ours is Gibraltar GPU based RAID [50], which focuses on accelerating Reed-Solomon [30] based parity codes to create a block based RAID in user-space. Gibraltar has several limitation, including the need to use the O_DIRECT flag in order to bypass the Linux buffer cache, which hurts performance. Additionally, Gibraltar cannot perform end-to-end integrity checks, parallel array rebuild of degraded arrays, or provide the same level of flexibility delivered by our per-file arrays tightly integrated with a PFS.

VII. Conclusion

Fault tolerance on large-scale storage servers is largely based on proprietary, expensive, hardware-based solutions with limited flexibility and scalability. In this paper, we have presented a cost-effective alternative that uses commodity

GPUs to implement RAID-6 in software, in conjunction with the Lustre PFS. Our solution leverages low-cost GPUs on the client and server nodes to accelerate minimum-density RAID-6 coding schemes. We have shown, through a prototype implementation, that our software-controlled parity computation scheme imposes acceptable overhead on application performance, and constitutes, overall, a feasible, low-cost, and efficient alternative to specialized hardware-based solutions.

As future work, we will extend our RAID solution to other accelerators in a heterogeneous setting, to expedite the encoding schemes. We will also explore the use of CPUs from emerging many-core nodes, those that cannot be fully utilized by a typical application, to compute the RAID encoding.

ACKNOWLEDGMENT

This paper is based upon work supported in part by the National Science Foundation under Grants CCF-0746832, CNS-1016793, and CNS-1016408.

REFERENCES

[1] "Spallation Neutron Source," 2008. http://www.sns.gov/.
[2] Conseil Européen pour la Recherche Nucléaire (CERN), "LHC– the large hadron collider," July 2007. http://lhc.web.cern.ch/lhc/.
[3] B. Abbott, A. Baranovski, M. Diesburg, G. Garzoglio, T. Kurca, and P. Mhashilkar, "Dzero data-intensive computing on the open science grid," *Journal of Physics: Conference Series*, 119, 2008.
[4] "Sloan digital sky survey," 2005. http://www.sdss.org.
[5] "Top500 supercomputer sites," http://www.top500.org/.
[6] S. Oral, F. Wang, D. Dillow, G. M. Shipman, R. Miller, and O. Drokin, "Efficient object storage journaling in a distributed parallel file system," in *Proc. USENIX FAST*, 2010.
[7] Brooke Crothers, "DARPA 'exascale' supercomputer in the works," Aug. 2010. http://news.cnet.com/8301-13924_3-20013088-64.html.
[8] Z. Zhang, C. Wang, S. S. Vazhkudai, X. Ma, G. Pike, J. Cobb, and F. Mueller, "Optimizing center performance through coordinated data staging, scheduling and recovery," in *Proc. IEEE SC*, 2007.
[9] M. D. R. Alex Osuna, and Siebo Friesenborg, "Considerations for raid-6 availability and format/rebuild performance on the ds5000," 2009. Document Number: REDP-4484-00.
[10] B & H Foto & Electronics Corp., "Active Storage 16TB ActiveRAID Hard Drive Array," 2011. http://www.bhphotovideo.com/c/product/697437-REG/Active_Storage_AC16SFC02_16TB_ActiveRAID_Hard_Drive.html.
[11] J. Michalakes and M. Vachharajani, "GPU acceleration of numerical weather prediction," in *Proc. IEEE IPDPS*, 2008.
[12] C. Trapnell and M. C. Schatz, "Optimizing data intensive GPGPU computations for DNA sequence alignment," *Parallel Comput.*, 35:429–440, Aug. 2009.
[13] M. Fatica, "Accelerating linpack with cuda on heterogenous clusters," in *Proc. ACM GPGPU*, 2009.
[14] T. D. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon, "Biomedical image analysis on a cooperative cluster of GPUs and multicores," in *Proc. ACM ICS*, 2008.
[15] M. M. Rafique, A. R. Butt, and D. S. Nikolopoulos, "A capabilities-aware framework for using computational accelerators in data-intensive computing," *J. Parallel Distrib. Comput.*, 71:185–197, Feb. 2011.
[16] M. Curry, A. Skjellum, H. Ward, and R. Brightwell, "Arbitrary dimension reed-solomon coding and decoding for extended raid on GPUs," in *Proc. PDSW*, 2008.
[17] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Real-time parallel hashing on the GPU,". *ACM Trans. Graph.*, 28:154:1–154:9, Dec. 2009.
[18] G. I. of Technology, "Keenland," 2010. http://keeneland.gatech.edu/.
[19] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "Gvim: GPU-accelerated virtual machines," in *Proc. ACM HPCVirt*, 2009.
[20] Damon Poeter, "Cray's Titan Supercomputer for ORNL Could Be World's Fastest," 2011. http://www.pcmag.com/article2/0,2817,2394515,00.asp.
[21] M. M. Rafique, A. R. Butt, and D. S. Nikolopoulos, "Designing accelerator-based distributed systems for high performance," in *Proc. IEEE/ACM CCGRID*, 2010.
[22] M. A. Clark, "Qcd on GPUs: cost effective supercomputing," 2009.
[23] NVIDIA Corporation, "Science & Education," 2011. http://www.nvidia.com/object/nvidia_userful_success.html.
[24] Sun Microsystems, Inc., "Lustre file system - High-performance storage architecture and scalable cluster file system," 2007.
[25] M. Blaum and R. Roth, "New array codes for multiple phased burst correction," *IEEE Trans. on Information Theory*, 39(1):66–77, Jan. 1993.
[26] J. S. Plank, "The raid-6 liberation codes," in *Proc. USENIX FAST*, 2008.
[27] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system," in *Proc. USENIX FAST*, 2008.
[28] Intel Corporation, "Intel Microarchitecture Codename Sandy Bridge," 2011. http://www.intel.com/technology/architecture-silicon/2ndgen/index.htm.
[29] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum, "Block-level raid is dead," in *Proc. USENIX HotStorage*, 2010.
[30] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.
[31] J. S. Plank and L. Xu, "Optimizing cauchy reed-solomon codes for fault-tolerant network storage applications," in *Proc. IEEE NCA*, 2006.
[32] G. M. Shipman, D. A. Dillow, S. Oral, and F. Wang, *The Spider center wide file system: From concept to reality*. 2009.
[33] KGPU, "KGPU: enabling GPU computing in Linux kernel," 2011, http://code.google.com/p/kgpu.
[34] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating System Abstractions To Manage GPUs as Compute Devices," in *Proc. ACM SOSP*, 2011.
[35] Oracle Corporation, "Lustre Documentation," 2011, http://wiki.lustre.org/index.php/Lustre_Documentation.
[36] Sun Microsystems, Inc., "LibLustre How-To Guide," 2010, http://wiki.lustre.org/index.php/LibLustre_How-To_Guide.
[37] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2," Technical Report CS-08-627, University of Tennessee, Aug. 2008.
[38] L. Wang, M. Huang, and T. El-Ghazawi, "Towards efficient gpu sharing on multicore processors," in *Proc. ACM PMBS*, 2011.
[39] M. A. Frumkin and L. V. Shabanov, "Benchmarking Memory Performance with the Data Cube Operator," in *Proc. ISCA*, 2004.
[40] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "End-to-end data integrity for file systems: a zfs case study," in *Proc. USENIX FAST*, 2010.
[41] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," in *Proc. USENIX OSDI*, 2006.
[42] 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," in *Eurographics, State of the Art Reports*, pages 21–51, Aug. 2005.
[43] S. Al-Kiswany, M. Ripeanu, S. S. Vazhkudai, and A. Gharaibeh, "stdchk: A checkpoint storage system for desktop grid computing," in *Proc. IEEE ICDCS*, 2008.
[44] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu, "StoreGPU: exploiting graphics processing units to accelerate distributed storage systems," in *Proc. ACM HPDC*, 2008.
[45] A. Gharaibeh, S. Al-Kiswany, S. Gopalakrishnan, and M. Ripeanu, "A GPU accelerated storage system," in *Proc. ACM HPDC*, 2010.
[46] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, and M. Ripeanu, "On GPU's viability as a middleware accelerator," *Cluster Computing*, 12:123–140, 2009.
[47] G. Falcão, L. Sousa, and V. Silva, "Massive parallel ldpc decoding on GPU," in *Proc. ACM PPoPP*, 2008.
[48] O. Harrison and J. Waldron, "Practical symmetric key cryptography on modern graphics hardware," in *Proc. USENIX Security Symposium*, 2008.
[49] A. Moss, D. Page, and N. P. Smart, "Toward acceleration of RSA using 3D graphics hardware," in *Proc. IMA Cryptography and Coding*, 2007.
[50] M. L. Curry, H. L. Ward, A. Skjellum, and R. Brightwell, "A lightweight, GPU-based software raid system," in *Proc. ICPP*, 2010.