CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2011; **23**:2477–2495
Published online 21 September 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1810

Gibraltar: A Reed-Solomon Coding Library for Storage Applications on Programmable Graphics Processors

Matthew L. Curry [1,*,†], Anthony Skjellum [2], H. Lee Ward [1] and Ron Brightwell [1]

[1] *Sandia National Laboratories, P.O. Box 5800, Albuquerque, NM 87185-1319, USA*
[2] *Computer and Information Sciences, The University of Alabama at Birmingham, 115A Campbell Hall, 1300 University Blvd, Birmingham, AL 35294-1170, USA*

SUMMARY

Reed–Solomon coding is a method for generating arbitrary amounts of erasure correction information from original data via matrix–vector multiplication in finite fields. Previous work has shown that modern CPUs are not well-matched to this type of computation, requiring applications that depend on Reed–Solomon coding at high speeds (such as high-performance storage arrays) to use hardware implementations. This work demonstrates that high performance is possible with current cost-effective graphics processing units across a wide range of operating conditions and describes how performance will likely evolve in similar architectures. It describes the characteristics of the graphics processing unit architecture that enable high-speed Reed–Solomon coding. A high-performance practical library, Gibraltar, has been prototyped that performs Reed–Solomon coding on graphics processors in a manner suitable for storage arrays, along with applications with similar data resiliency needs. This library enables variably resilient erasure correcting codes to be used in a broad range of applications. Its performance is compared with that of a widely available CPU implementation, and a rationale for its API is presented. Its practicality is demonstrated through a usage example. Copyright © 2011 John Wiley & Sons, Ltd.

Received 9 January 2010; Revised 7 January 2011; Accepted 29 May 2011

KEY WORDS: graphics processors; storage; Reed–Solomon coding; reliability; fault tolerance

1. INTRODUCTION

Redundant array of independent disks (RAID) is a method for using large numbers of disks to provide increased secondary storage access speed, fault tolerance, and availability in the presence of failures [1]. Increased access speed arises from using many disks in parallel to perform a single read or write operation, or multiple unrelated reads and writes, whereas fault tolerance and availability are introduced through the use of parity bytes, blocks, or drives. Generally, by introducing m appropriately distributed parity blocks (created with a maximum-distance separable code [2]) per k data blocks in a disk array stripe, a disk array can withstand the total failure of up to m disk drives. RAID levels 5 and 6 are two common RAID levels that are capable of $m = 1$ and $m = 2$, respectively. Although $m = 1$ can be implemented with $k - 1$ exclusive-or operations, $m \geqslant 2$ involves a more complex calculation [3].

*Correspondence to: Matthew L. Curry, Sandia National Laboratories, P.O. Box 5800, Albuquerque, NM 87185-1319, USA.
†E-mail: mlcurry@sandia.gov
Contract/grant sponsor: United States Department of Energy; contract/grant number: DE-FC02-06ER25767
Contract/grant sponsor: National Science Foundation; contract/grant number: MRI-0821497

One standard method of generating more than one parity block is Reed–Solomon coding [4]. Many RAID 6-specific codes exist [3,5,6], but Reed–Solomon coding is often used today in RAID 6 systems, including the Linux kernel's software RAID [7]. Although RAID 6 only requires two coded blocks, Reed–Solomon coding can be used to generate an arbitrary number of independent coded blocks. Unfortunately, most conventional commodity processors are not well-suited architecturally to Reed–Solomon coding with arbitrary m. This quality results in slow 'extended' software RAID with $m > 2$, necessitating alternative hardware solutions.

Reed–Solomon coding for parity computation is generally performed by hardware RAID controllers that are positioned in the data path between the computer and its storage resources. The controller can present the multiple attached storage resources as a single block device to the operating system, resulting in transparency and performance. However, a RAID controller is limited in the tasks it can accomplish. Relocating the Reed–Solomon coding to a more general purpose piece of hardware results in new flexibility. Any application that can benefit from Reed–Solomon coding would be able to offload this computation, increasing the performance of that portion of the application, either by increasing the speed of the task or overlapping it with a different computation running on the CPU(s).

Graphics processing units, also known as GPUs, are highly parallel devices designed to exploit the embarrassingly parallel nature of graphics rendering tasks [8, 9]. As conventional CPUs have transitioned through single core, multi-thread, and multi-core devices, the anticipation is high that CPUs will evolve into many-core devices to keep pace with the demands of Moore's Law while mitigating increasing power consumption [10]. Application developers have been considering the GPU as a currently mature and highly developed computational platform with dozens to hundreds of cores. GPUs have been successfully applied to applications that are either embarrassingly parallel or have embarrassingly parallel sub-steps [11, 12].

Until recently, however, GPU platforms were restricted in terms of usable data types. Only certain floating point operations and data types were well-supported, as GPUs only contained floating point units for use within shader programs [13]. Some other types could be emulated through limited range of floating point types; however, unless used judiciously, emulation often proved to be an inefficient use of the GPU's resources [14]. With the native types available, only applications that heavily utilized single-precision floating point data and calculations could be meaningfully accelerated via GPU computation.

To provide acceleration for more general-purpose GPU applications, NVIDIA released its CUDA API (NVIDIA Corporation, Sta. Clara, CA, USA) and architecture [15], and ATI/AMD (Advanced Micro Devices, Sunnyvale, CA, USA) released a stream computing software stack that potentially allows the use of many high-level languages to program ATI GPUs [16, 17]. Both ATI and NVIDIA technology allow bitwise operations to be performed on arbitrary binary data using their GPUs and software. This presents the opportunity for applications to perform more general data processing tasks that are well-suited to the GPU's overall architecture, but need different kinds of computation than floating point units can provide.

This work identifies Reed–Solomon coding as an application that both suits the general architecture of GPUs and requires the added primitive types and operations available through CUDA. In particular, results presented previously by the authors demonstrated that GPUs could show superior performance as part of a software RAID system that includes more than two parity disks by achieving an overall tenfold speedup over a CPU implementation for RAID 6 workloads and beyond [18, 19].

This paper describes generalized Reed–Solomon coding on programmable GPU hardware, specifically for use with applications requiring data resiliency in a manner similar to RAID. This paper begins by describing Reed–Solomon coding in more detail within the context of RAID and distributed data applications. It goes on to detail the mapping of Reed–Solomon coding to NVIDIA GPUs. A performance evaluation is provided, including comparison with a well-known CPU library that also implements the Reed–Solomon coding as used in this work [20]. Future potential trends for GPUs and other highly multicore devices and their effects on this application are described. A publicly available prototype library, Gibraltar, demonstrates the findings of this work. To demonstrate its practicality, a usage example and design rationale are provided.

DOI: 10.1002/cpe

2. MOTIVATION

Given the statistics provided by drive manufacturers, RAID 6 allows for an impressive mean time to data loss. For example, some disk drives are described as having a mean time to failure of 1.2 million h [21]. Chen *et al.* derived the formula for calculating reliability of an array with several RAID 6 groups [1]. This formula yields a mean time to data loss of over 100 billion years for an array of 16 disks if repairs require 24 h to complete, and the mean time to failure is 1.2 million h.

Unfortunately, this formulaic representation of reliability is not representative of real installations. Recent studies have raised several concerns about the reliability of disk manufacturers' statistics. Pinheiro *et al.* found that the annualized failure rates of their disk drives are much higher than are implied by data sheets [22]. Schroeder and Gibson found similar results through a survey of drives under multiple administrative domains [23]. These results imply that RAID 6 reliability cannot be directly calculated from vendor data, but is more likely to be much lower than usually estimated.

There exist rare—yet grave—disk reliability problems, such as batch-correlated failures. Batch-correlated failures result from a manufacturing defect in a group of drives. Pâris and Long show that if several failure-prone disks from a defective batch compose a RAID 5 array, chances are poor that one can rebuild a failed disk before *another* failure occurs [24]. They also show that adding another parity disk can drastically increase chances of recovery, but having only two parity disks can still be a risky proposition. For example, for a scenario where recovery requires 24 h, and one disk fails per week on average, a RAID 6 array has less than 70% probability of being able to recover from a disk failure before data loss occurs [24].

Another issue is double disk failures combined with read errors [25]. The areal density for disk platters is approximately doubling every year, but disk speeds are not increasing that quickly [26]. Therefore, the time required to rebuild an array is increasing, causing a related increase in the probability of losing all redundancy in an array. Although this situation alone does not cause data loss, there would be no protection in the case of a read error from any of the remaining disks. If such an error occurs on one drive, that block must be recovered from backups if available. This is a significant task for system administrators.

Given the bit error rate statistics from drive manufacturers, the probability that a large drive will encounter an unrecoverable read error during the course of reconstruction is too large to safely ignore. A typical disk could have a capacity of 1 TB, and an error rate of one read error per 10^{15} bits [21]. Consequently, the probability of a failure during a reconstruction pass through one disk is $8 \times 10^{12}/10^{15}$, or 0.008. However, extrapolating this to 10 drives with no redundant disks yields a survival rate of only 92.3% after a single pass:

$$P = e^{-1 \times numdisks \times \lambda \times T_R} \tag{1}$$

$$= e^{-10 \times \frac{1}{10^{15}} \times 8 \times 10^{12}} \tag{2}$$

$$= 0.92312. \tag{3}$$

The RAID 1 can be used to increase and provide more reliability in an array, but is an inefficient use of storage resources. For example, to mitigate the risk of unrecoverable read errors (UREs), data must be stored at least three times within an array, tripling storage requirements. A more efficient way to mitigate current reliability problems is to add more parity to a RAID array based on the likelihood of data loss for the size of the array, the operating environment, and the properties of the disks attached. This is not a commonly implemented solution because of the computational expense of generating more than two parity blocks per stripe. However, one can utilize the computational power of GPUs as part of a RAID system, allowing two or more parity blocks per stripe as needed while maintaining high performance.

DOI: 10.1002/cpe

3. REED–SOLOMON CODING FOR REDUNDANT ARRAY OF INDEPENDENT DISKS

The primary operation in Reed–Solomon coding is the multiplication of F, the lower m rows of an information dispersal matrix $A = \begin{bmatrix} I \\ F \end{bmatrix}$, with a vector of data elements d [27].

$$\begin{bmatrix} I \\ F \end{bmatrix} d = \begin{bmatrix} d \\ c \end{bmatrix} \tag{4}$$

This yields another vector of redundant elements (the coding vector, c). The redundancy of the operation comes from the over-qualification of the system: Any k elements of $e = \begin{bmatrix} d \\ c \end{bmatrix}$ may be used to recover d, even if some (or all) elements of d are not available [4]. A more in-depth discussion is provided in the next section.

Rather than relying on integer or floating point arithmetic, the operations are performed on members of a finite field [28]. Addition of two numbers is implemented with an exclusive-or operation, whereas multiplication by two is implemented with a linear feedback shift register [7]. Multiplying two arbitrary numbers involves decomposing the problem into addition of products involving powers of two, which potentially requires a large number of operations. One useful identity that holds true in finite fields of size 2^w (where w is the number of bits per symbol) is as follows:

$$x \times y = \exp(\log(x) + \log(y)) \tag{5}$$

where the addition operator denotes normal integer addition modulo $2^w - 1$, whereas exp() and log() are, respectively, exponentiation and logarithm operations in the finite field using a common base [27]. Because $w = 8$ for RAID systems, an implementation can contain pre-calculated tables for the exp and log operations, which are each 256 B in length. Multiplications can be implemented using these tables with three table lookup operations and addition modulo $2^w - 1$ instead of potentially many more logical operations. Decoding (recovering missing data elements from k remaining data and/or coding elements) is a similar operation, so all of these also hold true for decoding.

Unfortunately, the type of table lookup operations used in Reed–Solomon coding does not exploit the internal vector-based parallelism of CPUs. Although fast vector instructions have been included in modern CPUs, few CPU models include a parallel table lookup instruction. In the case of IBM's power architecture, which has the AltiVec instruction set that includes a parallel table lookup instruction, multiplication in finite fields is faster than implementations for x86 processors [29]. Unfortunately, parallel table lookup capability is not common, and CPU implementations of finite field arithmetic tend to suffer accordingly. A more in-depth treatment of implementation details of Reed–Solomon coding is available [27].

4. MAPPING REED–SOLOMON CODING TO GPUS

GPUs are architecturally quite different from CPUs. The emphasis of GPU architecture is to accomplish hundreds of millions of small, independent, and memory intensive computations per second to provide interactive graphics to a user. As such, GPUs have several interesting qualities that are directly applicable to the task of Reed–Solomon coding. In this discussion, a buffer is a particular slice of data of s bytes. A $k + m$ coding would require k data buffers and m coding buffers. These buffers are typically stored together, one after another, within a continuous memory region that is referred to as the buffer space. One may refer to the ith buffer in the buffer space, which specifically refers to the bytes between $i \times s \ldots (i + 1) \times s - 1$ within the buffer space. However, when not explicitly qualified, the ith buffer indicates its contents rather than its position. For the first k buffers, buffer 0 contains the first s bytes of data, buffer 1 contains the next s bytes of data, and so on. The following m buffers contain the coding data.

One of the more well-known features of a CUDA-based GPU is its vast number of multi-threaded processing cores organized into groups called multiprocessors. The GeForce GTX 8800 (NVIDIA Corp., Sta. Clara, CA, USA), for example, features 128 cores spread among 16 multiprocessors,

DOI: 10.1002/cpe

whereas the GeForce GTX 285 (NVIDIA Corp., Sta. Clara, CA, USA), contains 240 cores spread among 30 multiprocessors [15]. These cores are designed to be effective for many threads of execution as small as a few instructions. In the context of parity generation, each set of bytes in the data stream (i.e., byte b of all buffers) can be modeled accurately as an independent computation. In the approach presented here, each thread is responsible for 4 B per disk [‡]; this takes advantage of the relatively wide 384-bit memory bus of the GeForce GTX 285, which allows multiple threads to transfer all 32 bits each in parallel. It is unnecessary to load each thread with a significant portion of the workload to achieve good performance, even though this may result in many thousands of threads. Therefore, each thread can remain relatively small, and the GPU's scheduling of the threads can efficiently accomplish the work of creating parity for many megabytes of data at a time.

Each 200-series multiprocessor contains eight cores and a shared memory. This shared memory is banked, allowing up to 16, 32-bit accesses to occur in parallel, and is as fast as registers for each core. Bank conflicts can adversely affect performance because they can reduce the parallelism of memory accesses. Given the previously mentioned 30 multiprocessors of the GeForce GTX 285, each thread within a half-warp (a group of coscheduled threads that may have conflicting memory accesses, or 16 threads per 8-core multiprocessor in the 200-series GeForce GPUs) can access a separate bank of memory, allowing up to 240 table lookup operations in two clock cycles throughout the GPU. Unfortunately, it is difficult to manage the tables in each shared memory block to eliminate conflicts completely because the table accesses can be completely random. Mapping random accesses to memory banks invokes the same statistics as the birthday problem [30]. The birthday problem is the observation that, given a number of random selections from a finite group of values (e.g., the birthdays of people in a room), the probability of having multiple selections match increases quickly with the number of random selections made. A match for memory bank accesses equates to a conflict if the threads are not accessing the same address.

Simulations of this application's workload (i.e., random lookup operations within 256-B tables) have shown that there are on average 2.92 conflicting accesses to satisfy simultaneous random table lookup operations for a half-warp. On the GeForce GTX 285, this corresponds to an overall reduction in average performance to approximately 41.1 lookup operations per clock across the GPU[§]. Although this is significantly less than the peak of 120 accesses per clock, this is also significantly better than the one lookup per clock on a Core i7 CPU with four cores (based on a four-cycle latency for accessing data in the L1 cache [31]). This corresponds to a 20x performance superiority of a 1.5 Ghz GeForce GTX 285 when compared with a 3.0 Ghz Intel quad-core Core i7 processor (Intel Corp, Sta. Clara, CA, USA).

Another interesting hardware feature benefiting Reed–Solomon coding is the support for constant memory values. As an architecture that deals primarily in accessing read-only textures for mapping onto polygons, GPUs require fast constant accesses to provide high graphics performance[¶]. Unlike other types of memory within the CUDA architecture, constant memory is immutable. Therefore, to increase performance of constant memory accesses, each multiprocessor's constant accesses are cached. Once data is loaded into this cache, accesses to this data are, like shared memory, as fast as register accesses. With this in mind, the constant memory is a prime location for the information dispersal matrix. The matrices are small, and each element is accessed in lock-step across all cores within a multiprocessor.

To reduce register usage, the computations are arranged on the GPU as follows. The 4-B output values (i.e., the computed parity) are accumulated in registers. The input values (i.e., the user data) are read one buffer at a time, used in all necessary computations, and overwritten with the data bytes from the next buffer. The computations are arranged in this manner because $k > m$ in most situations, so storing the partially computed parity bytes is less expensive than storing all of the data bytes. This efficiency is enabled by the loop ordering, which is as follows.

[‡]The use of 4 B per disk per thread differs from the approach used in the first effort [11]. Efficiency was improved by reducing the size of operands and fetches from the main memory to the size of the memory banks of the GPU.

[§]This is an imprecise calculation, as the interactions with the thread scheduler are complex, but this is a reasonable estimate.

[¶]This fact is derived from the emphasis on texture fill rates in marketing materials for consumer GPUs. The texture fill rate is the theoretical limit of how fast a GPU can apply a texture to a polygon.

DOI: 10.1002/cpe

```
for (int i = 0; i < K; ++i) {
  in.f = bufs[rank+buf_size/sizeof(int)*i].f;
  for (int j = 0; j < M; ++j) {
    int F_val = sh_log[F_d[j*K+i]];
    for (int byte = 0; byte < sizeof(int); ++byte) {
        int log_val = F_val + sh_log[(in.b)[byte]];
        if (log_val >= 255) log_val -= 255;
        (out[j].b)[byte] ^= sh_exp[log_val];
    }
  }
}
for (int i = 0; i < M; ++i)
  bufs[rank+buf_size/sizeof(int)*(i+K)].f = out[i].f;
```

The `sh_log[256]` and `sh_exp[256]` variables are shared memory arrays initialized at the beginning of the kernel that hold the log and exp tables. These tables are used to multiply elements with the method summarized by Equation 5. The `in` variable, stored in a register, holds the data bytes currently being processed, whereas the `out[M]` array (also stored in registers) holds the parity as it is being computed. Their data type is a union of four `char` variables and an `int` variable, where `in.b[i]` is the ith byte, and `in.f` is the contents of the entire data structure. This allows independent addressing of the bytes within the kernel for calculations, but wider data types for accessing global GPU memory.

4.1. Reed–Solomon Decoding

One major contribution of this work is the improvement over previous GPU-based decoding performance [19]. The implementation of coding presented in this work is already highly optimized. From the perspective of RAID, decoding should be as fast so that performance of the server is not degraded upon failure of a disk. To ensure adherence to this performance requirement, the authors decided to make the implementations of coding and decoding on the GPU as similar as possible.

When designing a decoding routine, there is a design decision to be made about whether buffers must remain ordered, the user of the routine may be allowed to choose arbitrary orderings of buffers, or some combination of the two. If the buffers are ordered, the kernel has no static knowledge of which buffers require recovery, resulting in a need for indirect indexing through a table of failed buffers provided by the user. By imposing a partial ordering on the buffers, the library can statically determine how calculations should be performed without indirect indexing so as long as the generation matrix is adjusted to reflect buffer orderings. Therefore, in Gibraltar, a buffer ordering where all good buffers are listed at the beginning of the buffer space is required. Further, memory for decoded buffer contents is reserved after the original buffers within the buffer space. There are two further arguments that justify imposing this alternative partial order:

- Contents of the buffers for data are not typically resident in the memory of the host performing the decoding until requested. Instead, they are spread over devices (or another host) that can become unavailable because of failure or fault. Therefore, a transfer must be incurred between the host and device, and the host is free to place the contents of the transfer anywhere in its memory.
- If k is significantly greater than m, there are simple manners of changing the order of the buffers to satisfy the layout requirement quickly in CPU memory. If $1 \leqslant q \leqslant m$ buffers have failed, only q buffers must be moved within the buffer space after decoding.

The following example demonstrates the desired reordering requirements, along with a description of the quick reordering method. In an example $k = 4, m = 4$ coding system, codes are generated in a straightforward way. An overdetermined information dispersal matrix A is generated. To generate the parity vector c, one multiplies the lower m rows of A (usually denoted as F) by the data vector d. The memory layout for such an operation is simple, as each buffer is arranged linearly in memory. (See the API discussion for more information.) It is known at compile time where the input

DOI: 10.1002/cpe

bytes and output bytes are for a given step, so the generation routine can be highly efficient. Indices do not need to be read from memory, but are computed. This simplicity allows many optimizations, such as unrolling of loops within the kernel by the compiler.

However, there is some variability in how a recovery can be accomplished. Suppose that devices 0, 2, 3, and 5 fail. The state is now represented by the following equation, with unavailable elements denoted with a question mark.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ F_{0,0} & F_{0,1} & F_{0,2} & F_{0,3} \\ F_{1,0} & F_{1,1} & F_{1,2} & F_{1,3} \\ F_{2,0} & F_{2,1} & F_{2,2} & F_{2,3} \\ F_{3,0} & F_{3,1} & F_{3,2} & F_{3,3} \end{pmatrix} \begin{pmatrix} d_0? \\ d_1 \\ d_2? \\ d_3? \end{pmatrix} = \begin{pmatrix} d_0? \\ d_1 \\ d_2? \\ d_3? \\ c_0 \\ c_1? \\ c_2 \\ c_3 \end{pmatrix} \tag{6}$$

The goal of this operation is to solve for missing portions of d with the present portions of e, which includes elements of d and c. By construction, A has full rank, so rows may be eliminated without losing information until only k rows remain. Doing so requires removing the corresponding elements of e to maintain the correctness of the equation. The vector e and A are modified (creating e' and A', respectively) to preserve the equality, yet reflect in e' only the elements that have not been lost.

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ F_{0,0} & F_{0,1} & F_{0,2} & F_{0,3} \\ F_{2,0} & F_{2,1} & F_{2,2} & F_{2,3} \\ F_{3,0} & F_{3,1} & F_{3,2} & F_{3,3} \end{pmatrix} \begin{pmatrix} d_0? \\ d_1 \\ d_2? \\ d_3? \end{pmatrix} = \begin{pmatrix} d_1 \\ c_0 \\ c_2 \\ c_3 \end{pmatrix} \tag{7}$$

Pre-multiplying each side by A'^{-1} will yield the desired configuration that will yield the missing data elements with only the known data elements.

$$\begin{pmatrix} d_0? \\ d_1 \\ d_2? \\ d_3? \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ F_{0,0} & F_{0,1} & F_{0,2} & F_{0,3} \\ F_{2,0} & F_{2,1} & F_{2,2} & F_{2,3} \\ F_{3,0} & F_{3,1} & F_{3,2} & F_{3,3} \end{pmatrix}^{-1} \begin{pmatrix} d_1 \\ c_0 \\ c_2 \\ c_3 \end{pmatrix} \tag{8}$$

The properties of finite field arithmetic and the methods of generation for A will ensure that these equalities hold, A' is invertible, and all elements generated are within the bounds specified by the domain of the code (e.g., 8-bit values) [27].

One obvious optimization would be to not explicitly perform any of the row multiplications for elements that are already present in d. This requires a vector that indicates the buffers that need recomputing. In particular, to accomplish the decoding task on a GPU, this vector must be copied into every thread block, and referenced heavily via indirect indexing, reducing the rate of finite field multiplications that may be computed. To improve efficiency, the equations and buffers are reordered. The change is original, but simple: When creating the matrix for recovery from the initial information dispersal matrix, one rearranges rows of A and e so that data elements still present in e are in the lower portion of the matrix, then ignore the lower rows when performing

DOI: 10.1002/cpe

the multiplication. The problem now more closely resembles a generation problem, with no need for indirect indexing on the GPU. The previous example is now recast into the following equation:

$$B = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \tag{9}$$

$$B \begin{pmatrix} d_0? \\ d_1 \\ d_2? \\ d_3? \end{pmatrix} = BA'^{-1} \begin{pmatrix} d_1 \\ c_0 \\ c_2 \\ c_3 \end{pmatrix}. \tag{10}$$

In this formulation, the only extra parameter (aside from the coding scheme and the buffer space's address) that is required to work efficiently is the quantity of data buffers that need recovery. There will always be a need for k data or parity elements to recover the missing data elements, so buffers can be statically arranged such that k intact buffers are at the beginning of the memory region, and the remaining buffers requiring recovery can be output in the next portion of the buffer region. The order of the buffers themselves is unimportant; instead, it is the ability for the kernel to refer directly to memory in a streaming manner without the use of a table of indices, as that table has been propagated into the buffer order and information dispersal matrix. This propagation allows for direct, efficient access to the data.

5. PERFORMANCE RESULTS

An experimental program was constructed to measure the performance of Gibraltar. It encodes a set of data at varying values of $k + m$ with each buffer occupying one megabyte of memory. The program erases $\min(k, m)$ random data buffers, then recovers their original contents. The same operations are performed using Jerasure [20], a well-known library implementing many erasure correcting codes including Reed–Solomon codes. The results are reported from the perspective of user-visible throughput, as the basic operation is cast into the idea that this library implements a filter for ensuring reliability or recovery of data intended for other purposes. An example case would be that coding is being performed on 10, 1-MB buffers in a $k = 6, m = 4$ configuration. A user does not interact with parity, so throughput is calculated by dividing the size of the data (6 MB) by the time required to code it. Similarly, the throughput for recovery is calculated to be the size of the data (6 MB) divided by the time required to return that data.

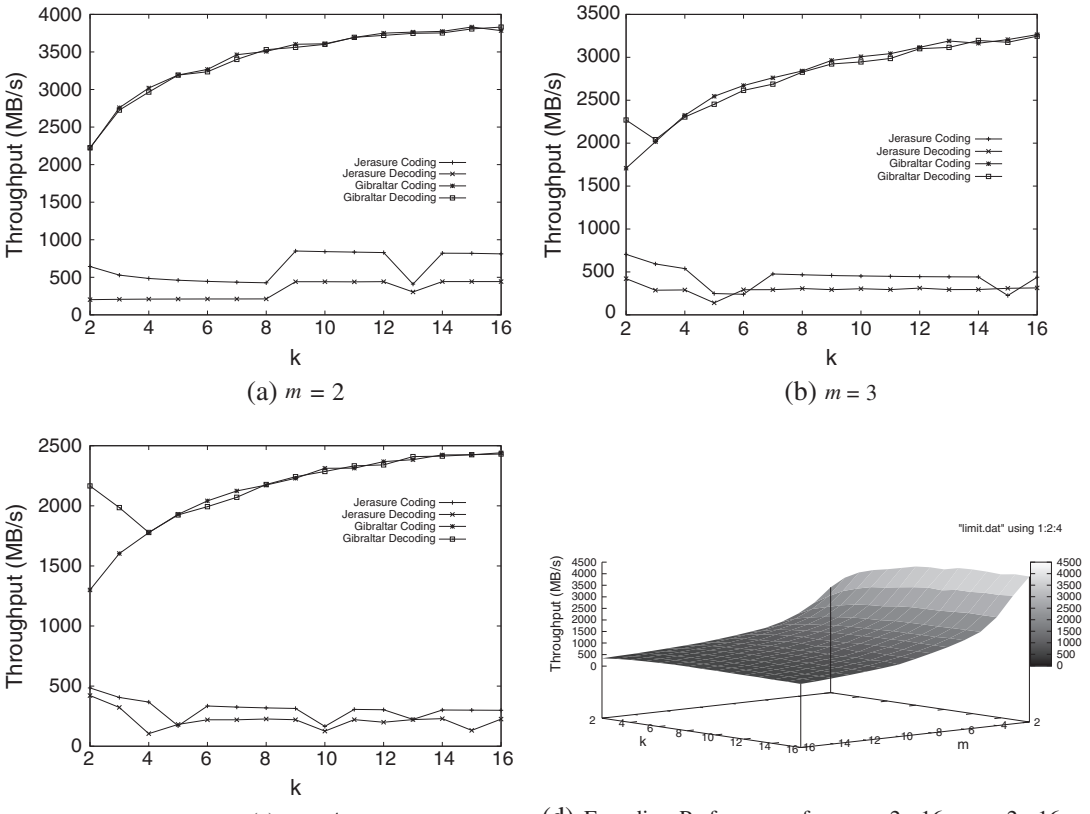
It is important to note that recovery operations in Gibraltar are only intended to recover data buffers, and that coding buffers should be recovered with a subsequent call to a generation routine. This is different from Jerasure, as the coding buffers are recovered automatically if missing, even when the coding buffers are not needed or devices to store the coding buffers are unavailable. To ensure fair benchmarking, only data buffers are erased. The CPU operations are performed using an unaltered version of the latest available Jerasure, version 1.2. The machine used in this test includes an Intel Extreme 965. The memory used is 6 GB of tri-channel DDR3 memory clocked at 1333 Mhz. The GPU used is an NVIDIA GeForce GTX 285.

Gibraltar was designed to allow support for a wide range of back-end devices, some of which allow certain significant optimizations pertaining to memory allocation. For example, the NVIDIA CUDA API allows for allocation of memory regions that are not swappable by the virtual memory subsystem and are mapped into the GPU's address space, allowing increased computation and PCI-Express traffic overlap. To allow the user programs to transparently access these features, Gibraltar API calls `gib_alloc` and `gib_free` have been defined as wrappers to these specialized memory management routines. These wrappers have been used for these performance evaluations. A user is allowed to use standard `malloc` and `free` calls, but can experience reduced performance.

DOI: 10.1002/cpe

Given that the focus of Gibraltar is integration into a software RAID system, it is illustrative to compare the bandwidth achieved by Gibraltar to that of a modern disk. Given recent benchmarks of solid state drives and more conventional disks [32], 100 MB/s is a representative average bandwidth. It is clear from Figure 1c, which shows Jerasure and Gibraltar performance for $k+4$ codings, that Gibraltar can provide enough performance to support more than two dozen disks with enough fault tolerance to withstand failure of any four disks in the array. This is quite different from nested RAID levels such as RAID 6+0, which can combine two or more RAID 6 sets into an outer RAID 0 volume. RAID 6+0 can allow up to four failures, but not allow any four arbitrary disks to fail. Such failures must be limited to two per inner RAID 6 volume. Figure 1a shows that Gibraltar can support RAID 6 arrays at full streaming bandwidth for at least three dozen data disks.

An interesting performance characteristic is that Gibraltar has nearly identical performance for coding and decoding, whereas Jerasure tends to experience reduced performance for recovery. This is Gibraltar's expected behavior, as the generation and recovery are made to be extremely similar. Jerasure does not, however, reconfigure the matrix in memory as needed, but instead indexes within it. Several factors can cause Jerasure to be slower for decoding, including having less prefetch-friendly access patterns and creating the need for additional memory reads for indirect array accesses. A further note: When $m > k$, the tests performed show an increase in performance because of the restrictions on which buffers may be erased during this experiment. Although the tests show the efficiency of Gibraltar performing coding tasks involving large data transfers, single-stripe writes are an important use-case for RAID controllers. Single-stripe writes can cause workloads that are restricted to 64 KB of data per buffer. Although such workloads can potentially show reduced performance by restricting the ability to overlap computation and PCI-Express bus transfers, it is possible to bundle several non-contiguous updates into a single Gibraltar call by packing the stripes in the buffer space. Therefore, many requests from distinct readers and/or writers can be processed



(a) $m = 2$ (b) $m = 3$

(c) $m = 4$ (d) Encoding Performance for $m = 2 : 16$, $n = 2 : 16$

Figure 1. Performance and comparison with Jerasure.

DOI: 10.1002/cpe

at the same throughput as single large contiguous operations. However, there are workloads that may not be able to take advantage of these packing techniques. Figure 2 shows the impact of varying buffer sizes on the throughput of $k+2$, or RAID 6, coding performance. Although using smaller buffers can significantly impact performance, performance decreases less than 50% for a factor of 32 reduction in workload size. Gibraltar does not provide support for partial stripe updates, but instead requires parity generation from updated data and unmodified data within a stripe. With the advent of multiple kernel support, stripe updates may have become feasible. Currently, it is undetermined which strategies are best for partial stripe updates, and how this will change with GPU architectures.

6. FUTURE TRENDS

To demonstrate PCI-Express and compute throughput for Reed–Solomon coding on the GeForce GTX 285, three modes of operation have been tested with 1-MB buffers:

- Operations performed on host memory. This is the typical mode of operation. This is denoted as 'total throughput' in Figures 3a, 3b, and 3c.
- Operations performed on GPU memory. Given the high bandwidth of GPU memory compared with PCI-Express bandwidth, this is an approximation of the maximum performance of the GPU kernel without the impacts of PCI-Express transfers. This is denoted as 'GPU throughput' in Figures 3a, 3b, and 3c.
- No operations performed on host memory, while performing transfers as if the kernel were performing computations. This is an approximation of the maximum performance of the PCI-Express bus without the impacts of GPU computation. This is denoted as 'PCI-Express throughput' in Figures 3a, 3b, and 3c.

Results show that, for a RAID 6 workload (Figure 3a), the system is well-balanced. For a RAID-TP workload (Figure 3b), the balance is also good. The 285 starts to show more degraded balance with $m = 4$ (Figure 3c) and beyond (Figure 4).

One can see the effects of the overlapping computation and communication via the non-parallelizable overheads incurred, as represented by the near, but not precise, approximation of the minimum performance of PCI-Express bandwidth and compute throughput. This is interesting because the tendency for performance of the total system is to trend the PCI-Express performance with significant overhead (approximately 1 GB/s of throughput) when bandwidth-bound. This overhead is much smaller when coding becomes computation-bound. This is due to the large amount of data that must be transported to and from the GPU before work can start and after work has ended, respectively. The volume of data is required by the large number of GPU cores, and the NVIDIA runtime's attempts to use the entire GPU's computational capacity.

Such performance characteristics at this stage imply that, to have appreciably large performance improvements for $m = 2$ and $m = 3$, both the PCI-Express performance and the computation power

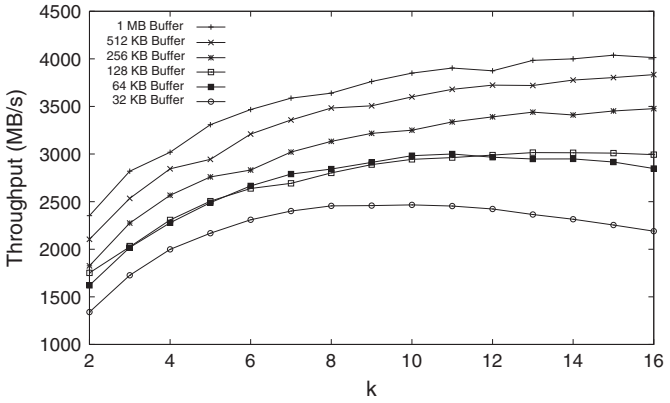
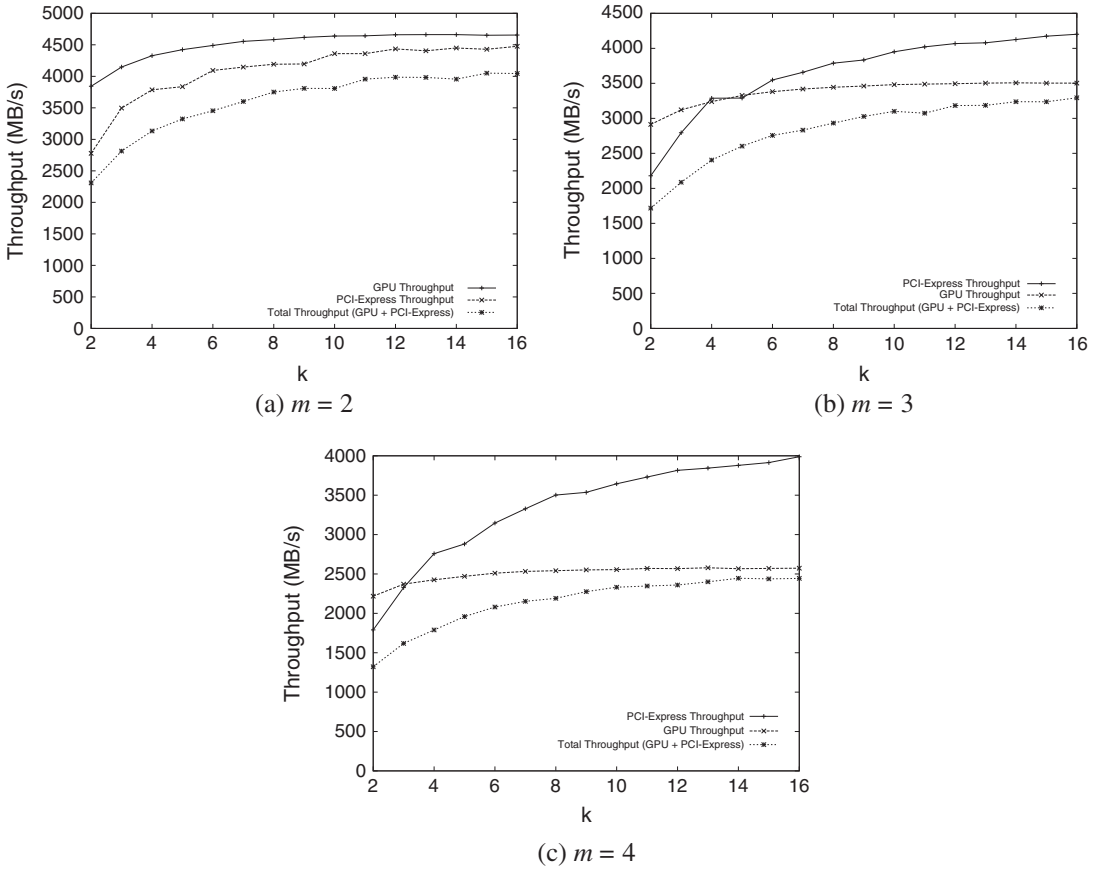


Figure 2. Effects on coding throughput of varying buffer size for $k + 2$ codings.

DOI: 10.1002/cpe



(a) $m = 2$ (b) $m = 3$

(c) $m = 4$

Figure 3. Performance of individual components.

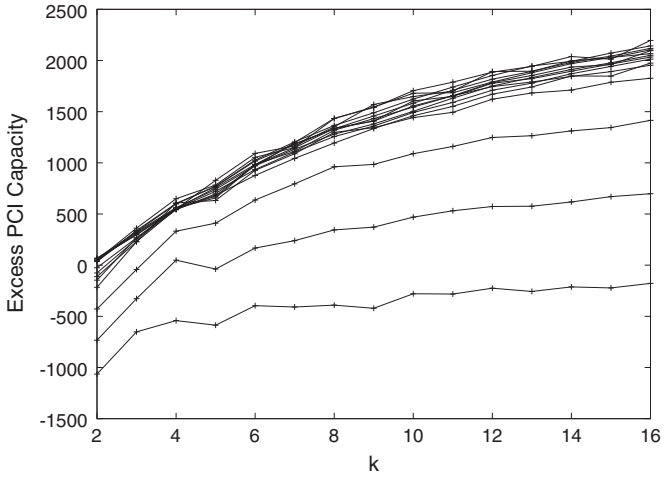


Figure 4. Excess PCI-Express performance over GPU performance for $m = 2 \ldots 16$; higher lines correspond to greater m.

within the GPU must increase in concert. However, if GPU power increases disproportionately, it becomes possible and reasonable to perform encodings and decodings with increasing values of m at higher rates.

DOI: 10.1002/cpe

Another view into this data is the mismatch between PCI-Express bandwidth and computational capacity of the GPU, as illustrated by Figure 4. Each line on the graph corresponds to a particular value for m. Currently, the only value of m that, for all values of k, has excess computational capacity compared with PCI-Express bandwidth is $m = 2$, which corresponds to RAID 6. $m = 3$ comes much closer for many useful values of k to being balanced, but extra compute capacity in future GPUs will be useful in bringing the other values of m into reach of fully using PCI-Express bandwidth. Furthermore, solutions that trade PCI-Express bandwidth for extra computation power remain effective. For example, it would be possible to split the $m = 4$ computation across two GPUs by copying all data buffers to both GPUs, then have each GPU compute two of the parity buffers each.

Unfortunately, the major GPU architectural feature that most limits the speed of this computation is the same feature that makes the GPU attractive: The multi-banked memories. Although these memories are useful for making these computations more efficient, there are changes that can improve performance of random table lookup operations in shared memory. Through simulation, the authors have quantified the effects of many potential design changes to the 200-series NVIDIA GPUs, which currently have on average 2.92 conflicting accesses per half-warp.

- Increase the number of banks per shared memory unit, but reduce the core-to-bank ratio. By increasing the banks to 32 per shared memory unit and keeping eight cores per streaming multiprocessor, the number of conflicting accesses is reduced to 2.25 per half-warp. This is distinct from the design changes of 400-series GPUs, where the number of cores increased with the number of banks [9], resulting in a streaming multiprocessor with 32 banks and 32 cores, with any thread in the warp potentially causing interfering accesses with other thread. This change causes an increase in conflicting accesses to 3.15 per warp.
- Decrease the number of cores per multiprocessor, and increase the number of multiprocessors. This course of action would reduce the effect of the birthday problem by reducing the number of competing cores. This would likely be an expensive option, as the number of shared memory units on the chip would grow quickly, but it would also be highly effective. Halving the number of cores to four, which run 16 threads together, would reduce conflicting accesses to 2.01 per half-warp, whereas two cores (with eight threads) would only have 1.32 conflicting accesses per half-warp. A single core for four threads and the same shared memory would average 1.06 conflicting accesses per half-warp.
- Increase the speed of the shared memory. The 200-series GPUs requires two clock cycles to satisfy a shared memory request, which allows conflicting accesses between threads that are not necessarily running simultaneously. Allowing the memory to satisfy single-byte requests in a single memory cycle would decrease the number of conflicting accesses per half-warp to 2.01. This would have the same effect as halving the number of cores per shared memory unit and doubling the number of shared memory units.

A potential feature that could improve the speed of random lookup operations in small tables is the ability to load a small array into a particular bank and index into that table. The programmer could then load multiple copies of the table simultaneously, querying each through different threads without conflict, or with controlled conflict. The authors experimented with this approach via computing indices, but the compute overhead was somewhat worse than the delay incurred by bank conflicts.

7. CONCLUSIONS AND FUTURE WORK

This paper describes a method of performing Reed–Solomon coding on graphics processors and general-purpose Reed–Solomon coding library, Gibraltar[‖], a prototype library that is suitable for use in applications requiring efficient data resiliency in a manner similar to RAID. Its immediate value stems from using CUDA-enabled GPUs to perform coding and decoding tasks, which has proven to be between fivefold and 10-fold faster than a well-known CPU code running on a processor that costs three times as much as the GPU used by Gibraltar.

[‖]Gibraltar is available at http://www.cis.uab.edu/hpcl/gibraltar, along with sample applications to test it.

DOI: 10.1002/cpe

There are at least two storage situations for which this library can be beneficial. First, in the realm of high-performance storage systems, software RAID can be implemented that can provide better reliability against disk failure than any RAID hardware available. This library can also be used for end-user applications without the availability of local high-speed storage resources. Even relatively modest GPUs are capable of performing Reed–Solomon coding, leading to a situation where a home computer can create a reliable online backup system by aggregating storage from many providers and administrative domains. Such backups can be performed without incurring high load on a system's CPU, whereas also using only a small amount of the computational capacity of a GPU. This frees much of the CPU computation resources on the system for other applications.

Several other non-storage applications can benefit from this high-speed Reed–Solomon coding. For example, new types of high-speed encryption and pseudorandom number generation become feasible [33]. Similarly, secret sharing can be performed with large secrets at high speeds [34].

This work demonstrates the applicability of certain multicore architectures to Reed–Solomon coding, as well as provides analysis into the effects of the design parameters for these architectures. Guidance has been provided as to the most beneficial architectural decisions related to Reed–Solomon coding, which apply equally well to other applications with the same data access patterns.

Gibraltar is intended to form the basis of a robust, high performance, and low cost alternative to traditional types of RAID hardware, and has been demonstrated to be effective in a prototype RAID system [35]. Performance indicates that reasonably sized RAID arrays can be supported with inexpensive GPUs with good performance in both normal operation and degraded mode. Further, the library's flexible nature allows for the parity of an array to scale with its size, unlike the standard set of RAID levels that only allow for a set maximum reliability.

APPENDIX A: GIBRALTAR APPLICATION PROGRAMMING INTERFACE

To address the need for improved RAID implementations, as well as provide this functionality to other software, the authors have created Gibraltar, a C library using NVIDIA's CUDA technology. Gibraltar provides data parity and recovery calculations through a generic, flexible API that hides the details of where the computations are being performed. Although Gibraltar does support GPU computation, it can allow for CPU failover, as well as the ability to use alternate computational methods with the same API. Furthermore, it is designed to provide the higher performance of a GPU whereas also being easy to use. This section provides an overview of the API, along with comments about design decisions.

gib_init

```
int gib_init(int k, int m, gib_context *gc);
```

This function initializes the Gibraltar runtime in order to perform $k + m$ codings. Gibraltar is capable of performing many different types of codings simultaneously, so this function may be called several times with varying values for k and m. The `gib_context` object is used in future calls in order to identify the type of coding to be performed. As will be verified by examining the further function definitions, the return values are error codes, whereas the products of the functions are returned by reference.

When this function is called, the GPU is initialized (if necessary), the routines to perform coding and decoding are compiled for the specific values of k and m (if necessary), and the programs are loaded into the GPU for later use.

gib_destroy

```
int gib_destroy(gib_context gc);
```

When the user no longer anticipates performing the coding represented by a context, its resources on the GPU can be released for other uses. Once destroyed, `gc` can no longer be used by the program unless it is re-initialized.

DOI: 10.1002/cpe

gib_alloc

```
int gib_alloc(void **buffers, int buf_size, int *ld, gib_context gc);
```

Gibraltar works with the assumption that all buffers are allocated end-to-end in main memory, with the start of each buffer separated by `ld`, which is an abbreviation of 'leading dimension', bytes, while the first `buf_size` bytes are used. When buffers are placed directly next to each other, `ld` = `buf_size`. In general, `ld` $\geqslant$ `buf_size`, and `ld` is used to satisfy some memory alignment constraint. Notice that `ld` is a return value instead of a parameter, as `ld` is determined by the library based on the target device. The amount of total memory allocated is `ld` $\times (k + m)$.

It is not necessary to allocate buffers with this function. However, there are often ways to increase performance depending on the underlying device (e.g., by using specialized memory allocation functions, such as CUDA's `cuMemAllocHost` [10]), or by altering the stride of memory accesses during the coding process (as governed by `ld`). For example, with certain methods of CPU coding, we noticed that performance doubled if `ld` was not divisible by two. However, it is also not necessary to use `ld` as given.

gib_free

```
int gib_free(void *buffers, gib_context gc);
```

Gibraltar's allocation function is able to use non-standard memory allocation functions, so the user may not know how to appropriately free the memory associated. Furthermore, the use of functions can change over time because of limited resources or the size of allocation. `gib_free` is used to free memory allocated with `gib_alloc`.

gib_generate

```
int gib_generate(void *buffers, int buf_size, gib_context gc);
```

This function performs the encoding of the data buffers into parity buffers according to the parameters set by the `gib_context`. The `buf_size` is the size of one coding buffer, or ld as returned by `gib_alloc`.

The situations, where data structures are coded for the sake of coding, are rare. Coding operations are usually part of another data moving task, such as writing to files, transfer over networks, etc. Furthermore, in order to improve performance, the library encourages that applications load data into buffers pre-allocated with `gib_alloc`. To take advantage of these conditions, Gibraltar imposes a particular style of layout on the data that it operates upon, which is in the form of a multi-dimensional array embedded within a single array.

Figure A.1 illustrates how the layout is interpreted. In short, the buffers are situated in memory such that byte i of data buffer j (i.e., $D_{j,i}$) is at the location `buf[`$j \times ld + i$`]`. Once the routine

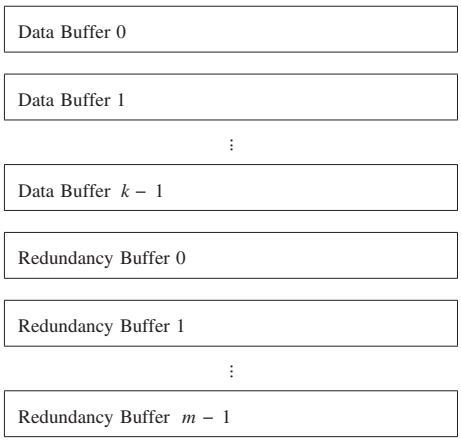


Figure A.1. Buffer layout for `gib_generate`.

DOI: 10.1002/cpe

has been run, the output is situated at the end of the allocated space, such that byte i of parity buffer j (i.e., $C_{j,i}$) is at location `buf[(k + j) × ld + i]`. The output can be considered to have been computed by a function RS such as that for byte i in all input buffers as follows:

$$\mathrm{RS}(\{D_{0,i}, D_{1,i}, \ldots, D_{k,i}\}) = \{C_{0,i}, C_{1,i}, \ldots, C_{m,i}\}$$

gib_recover

```
int gib_recover(void *buffers, int buf_size, int *buf_ids, int re-
cover_last, gib_context gc);
```

This function, given k intact buffers, will regenerate the contents up to $f \leqslant m$ of the remaining buffers, which have presumably lost their data. The `buf_ids` variable contains a list of integers identifying the order of the buffers found in the variable `buffers`.

Although `gib_generate` took some liberties in dictating the layout of the buffers in memory, this function is more strict in some ways, less strict in others. Figure A.2 indicates the layout required to use `gib_recover`.

In order to use the function, all of the k intact buffers should be positioned in the first $k \times ld$ bytes of `buffers`. The order is not imposed by Gibraltar, *i.e.* they do not have to appear in sorted order. However, `buf_ids` should be populated with the order of the buffers as given.

The last entries in `buf_ids` should be set to the identities of the buffers that the application wishes to have recovered. The variable `recover_last` indicates the value f, which is the number of buffers that should be recalculated. When the function returns, the last f buffers will contain the original data.

If it is not necessary to recover m buffers, it is acceptable to pass only $k + f$ entries. However, all buffers should be situated adjacent to each other with no gaps (except constant stride between buffers as provided for by the `ld` parameter.)

APPENDIX B: OPERATIONAL EXAMPLE AND DESCRIPTION

Although Gibraltar does make use of GPUs to accomplish the vast majority of the computations required for Reed–Solomon coding, it is not necessary to use a GPU to implement all operations. Many of the operations, which are performed, require insignificant amounts of time to execute comparatively. With this simplified, faster implementation, the GPU is only required to perform large numbers of small matrix–vector multiplications. All other matrix algebra, including the necessary factorizations and inversions, occur within the CPU.

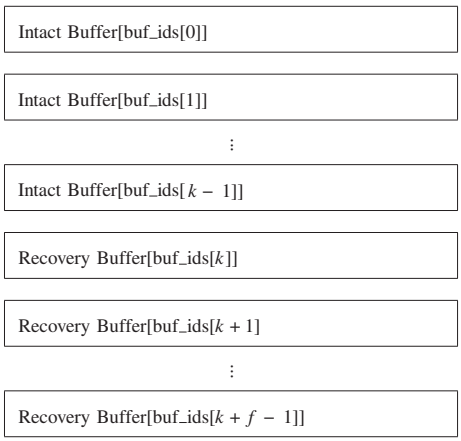


Figure A.2. Buffer layout for `gib_recover`.

DOI: 10.1002/cpe

An example program

In order to illustrate the way Gibraltar can be used, this section will present an example program. Its operation is simple: Perform a coding operation to yield parity, erase some buffers, and decode the remaining buffers to yield the original data.

Initializing Gibraltar. A context is used to manage Gibraltar's operation. Different kernels are required to perform different coding tasks, and one application can reasonably use multiple $k + m$ coding schemes simultaneously. To satisfy this requirement, a context is referenced in each call, which identifies which compiled kernels to execute when performing tasks.

Although Gibraltar does make use of compile-time knowledge in order to allow static optimization of the kernels, the user executable is not required to manage the configuration of Gibraltar. Initialization of a context will compile kernels on demand for the appropriate matrix–vector operations. Such kernels are cached for later initializations with the same parameters. As such, the first use of a particular encoding involves a delay.

```
int k = 16, m = 3;
gib_context gc;
gib_init(k, m, &gc);
```

After executing the above listing, the handle `gc` can now be used for coding operations.

In order to code and decode buffers, a buffer space must be allocated. To perform a $k + m$ encoding on buffers of size s, a region of size $(k + m) \times s$ must be allocated. Gibraltar includes an allocation function which can improve the performance of the GPU functions by introducing an optional stride between buffers and using specialized memory allocation routines. Such allocation routines can prepare buffers for high-speed transfers over the PCI-Express bus.

Because Gibraltar, via its context, has already obtained the value of $k + m$, it is only necessary to provide s. In this example, Gibraltar will be used to manage an array of integers.

```
int s = 1024 * sizeof(int);
int *buf;
// Allocate an array of (k+m)*s integers for coding
gib_alloc((void **)(&buf), s, &s, gc);
```

At this point in the program, the user can fill the buffer with the data, which requires coding.

Coding. Once again, this call requires no explicit mention of k or m. The coding function call is called `gib_generate`, as it is simplest to view the coding task as generating parity data from original data. Conceptually, this is a very simple call. The initialization routine had already calculated F, so the only task of this function is to manage the GPU.

```
gib_generate(buf, s, gc);
```

Losing data. Any interesting use of Gibraltar involves unavailability of data. This example will assume that an integer array, called `fail_config` of size $k + m$, contains a zero for a buffer that is available or a one for a buffer that is not available. At most m entries in `fail_config` may be set to one.

DOI: 10.1002/cpe

Decoding. Technically, the only requirements for buffer positioning for the decoding routine are:

1. The buffers indicated in the list of buffer assignments within the first k entries must be available and located in the same order at the beginning of the buffer space, and
2. the only buffers indicated for recovery should be data buffers.

However, in order to attain the highest performance when the application requires buffers to be in order, some previously mentioned layout requirements must be followed. If buffer $0 \leqslant i < k$, which is a data buffer, is available, it should be positioned in the i^{th} buffer position within the buffer space. The remaining spaces can be filled with available coding buffers. Upon completion of the routine, the user can move the contents of the recovered buffers, which are located after initial k available buffers within the buffer space, into proper positions.

Note: The function is called `gib_recover`, as this routine recovers lost data from available data.

```
int good_buffers[256]; /* k of these are needed. */
int bad_buffers[256]; /* Up to m of these can be used */
int ngood = 0;
int nbad = 0;
for (int i = 0; i < gc->k + gc->m; i++) {
  if (fail_config[i] == 0)
    good_buffers[ngood++] = i;
  else if (i < gc->k) {
    bad_buffers[nbad++] = i;
  }
}

/* Reshuffle to prevent extraneous memory copies later */
for (int i = 0; i < ngood; i++) {
  if (good_buffers[i] != i && good_buffers[i] < gc->k) {
    int j = i+1;
    while(good_buffers[j] < gc->k)
      j++;
    int tmp = good_buffers[j];
    memmove(good_buffers+i+1, good_buffers+i,
            sizeof(int)*(j-i));
    good_buffers[i] = tmp;
  }
}

/* Perform memory copies */
for (int i = 0; i < gc->k; i++)
  if (good_buffers[i] != i)
    memcpy(buf + s*i, buf + s*good_buffers[i], s);

int buf_ids[256];
memcpy(buf_ids, good_buffers, gc->k*sizeof(int));
memcpy(buf_ids+gc->k, bad_buffers, nbad*sizeof(int));
gib_recover(buf, s, buf_ids, nbad, gc);

/* Replace buffers */
for (int i = 0; i < gc->k; i++) {
  if (buf_ids[i] != i) {
    int j = i+1;
    while (buf_ids[j] != i) j++;
    memcpy(buf + s*i, buf + s*j, s);
    buf_ids[i] = i;
  }
}
```

At the termination of this portion of the program, the first k buffers contain their original contents.

DOI: 10.1002/cpe

REFERENCES

1. Chen PM, Lee EK, Gibson GA, Katz RH, Patterson DA. RAID: High performance, reliable secondary storage. *ACM Computing Surveys* 1994; **26**(2):145–185.
2. Reed IS, Chen X. *Error-control Coding for Data Networks*. Kluwer Academic Publishers: Dordrecht, Netherlands, 1999.
3. Blaum M, Brady J, Bruck J, Menon J. EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994; 245–254.
4. Reed IS, Solomon G. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics* 1960; **8**(2):300–304.
5. Corbett P, English B, Goel A, Grcanac T, Kleiman S, Leong J, Sankar S. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST 04)*, 2004; 1–14.
6. Plank JS. A new MDS erasure code for RAID-6. *Technical Report CS-07-602*, University of Tennessee, September 2007.
7. Peter Anvin H. The mathematics of RAID-6. Available from: http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf, [Accessed on January 6, 2010].
8. Foley JD, van Dam A, Feiner SK, Hughes JF. *Computer Graphics: Principles and Practice in C*, 2nd edn. Addison-Wesley Professional: Boston, Massachusetts, 1995.
9. NVIDIA Corporation. NVIDIA's next generation CUDA compute architecture: Fermi, 2009.
10. From a few cores to many: A tera-scale computing research overview, 2006. Available from: http://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf.
11. Carr NA, Hoberock J, Crane K, Hart JC. Fast GPU ray tracing of dynamic meshes using geometry images. In *GI '06: Proceedings of Graphics Interface 2006*. Canadian Information Processing Society: Toronto, Ont., Canada, 2006; 203–209.
12. Galoppo N, Govindaraju NK, Henson M, Manocha D. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society: Washington, DC, USA, 2005; 3.
13. Patidar S, Bhattacharjee S, Singh JM, Narayanan PJ. Exploiting the shader model 4.0 architecture. *Technical Report 145*, International Institute of Information Technology, Hyderabad, 2007.
14. Göddeke D, Strzodka R, Turek S. Accelerating double precision FEM simulations with GPUs. In *Proceedings of the 18th Symposium on Simulation Technique (ASIM 2005)*, Hülsemann F, Kowarschik M, Rüde U (eds). SCS Publishing House e.V: Erlangen, Germany, September 2005; 139–144.
15. NVIDIA Corporation. NVIDIA CUDA C programming guide, version 3.2, November 2010.
16. AMD. ATI CTM guide, 2006. Available from: http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf, [Accessed on July 27, 2009].
17. AMD. ATI Stream technology: Technical overview, 2008. Available from: http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pdf,[Accessed on July 27, 2009].
18. Curry ML, Skjellum A, Ward HL, Brightwell R. Accelerating Reed–Solomon coding in RAID systems with GPUs. In *IEEE International Symposium on Parallel and Distributed Processing, 2008*, April 2008; 1–6.
19. Curry ML, Ward HL, Skjellum A, Brightwell R. Arbitrary dimension Reed–Solomon coding and decoding for extended RAID on GPUs. *3rd Petascale Data Storage Workshop held in conjunction with SC08*, November 2008.
20. Plank JS, Simmerman S, Schuman CD. Jerasure: a library in C/C++ facilitating erasure coding for storage applications, - Version 1.2. *Technical Report CS-08-627*, University of Tennessee, August 2008.
21. Seagate Technology LLC. Barracuda ES.2 data sheet, 2008. Available from: http://www.seagate.com/docs/pdf/datasheet/disc/ds_barracuda_es_2.pdf.
22. Pinheiro E, Weber W-D, Barroso LA. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*. USENIX Association: Berkeley, CA, USA, 2007; 17–28.
23. Schroeder B, Gibson GA. Disk failures in the real world: what does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*. USENIX Association: Berkeley, CA, USA, 2007; 1–1.
24. Pâris J-F, Long DDE. Using device diversity to protect data against batch-correlated disk failures. In *StorageSS '06: Proceedings of the Second ACM Workshop on Storage Security and Survivability*. ACM Press: New York, NY, USA, 2006; 47–52.
25. Bairavasundaram LN, Goodson GR, Pasupathy S, Schindler J. An analysis of latent sector errors in disk drives. In *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. ACM: New York, NY, USA, 2007; 289–300.
26. Grochowski E, Halem RD. Technological impact of magnetic hard disk drives on storage systems. *IBM Systems Journal* 2003; **42**(2):338–346.
27. Plank JS. A tutorial on Reed–Solomon coding for fault-tolerance in RAID-like systems. *Software—Practice & Experience* September 1997; **27**(9):995–1012.
28. Lidl R, Niedrreiter H. *Introduction to Finite Fields and their Applications*. Cambridge University Press: New York, 1994.
29. Raghav B, Dubey PK, Kumar V, Rudra A. Efficient Galois field arithmetic on SIMD architectures. In *SPAA '03: Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM Press: New York, NY, USA, 2003; 256–257.

DOI: 10.1002/cpe

30. DasGupta A. The matching birthday and the strong birthday problem: a contemporary review. *Journal of Statistical Planning and Inference* 2005; **130**(1–2):377–389. Herman Chernoff: Eightieth Birthday Felicitation Volume.
31. Levinthal D. Performance analysis guide for Intel Core i7 processor and Intel Xeon 5500 processors, version 1.0. Available from: http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.
32. Narayanan D, Thereska E, Donnelly A, Elnikety S, Rowstron A. Migrating server storage to SSDs: analysis of tradeoffs. In *EuroSys '09: Proceedings of the fourth ACM european conference on Computer systems*. ACM: New York, NY, USA, 2009; 145–158.
33. Kiayias A, Yung M. Cryptography and decoding Reed–Solomon codes as a hard problem. In *Theory and Practice in Information-Theoretic Security, 2005. IEEE Information Theory Workshop on*, Oct. 2005; 48–48.
34. Shamir A. How to share a secret. *Communications of the ACM* 1979; **22**(11):612–613.
35. Curry ML, Ward HL, Skjellum A, Brightwell R. A lightweight, GPU-based software RAID system. In *International Conference on Parallel Processing*, 2010; 565–572.

DOI: 10.1002/cpe