Arbitrary Dimension Reed-Solomon Coding and Decoding for Extended RAID on GPUs

Matthew L. Curry and Anthony Skjellum
Department of Computer and Information Sciences
University of Alabama at Birmingham
1300 University Blvd, Rm 115A
Birmingham, AL 35294-1170
Email: {curryml, tony}@cis.uab.edu

H. Lee Ward and Ron Brightwell
Computer Science Research Institute
Sandia National Laboratories
PO Box 5800
Albuquerque, NM 87185-1319
Email: {lee, rbbrigh}@sandia.gov

***Abstract*—Reed-Solomon coding is a method of generating arbitrary amounts of checksum information from original data via matrix-vector multiplication in finite fields. Previous work has shown that CPUs are not well-matched to this type of computation, but recent graphical processing units (GPUs) have been shown through a case study to perform this encoding quickly for the $3 + 3$ (three data + three parity) case. In order to be utilized in a true RAID-like system, it is important to understand how well this computation can scale in the number of data disks supported. This paper details the performance of a general Reed-Solomon encoding and decoding library that is suitable for use in RAID-like systems. Both generation and recovery are performance-tested and discussed.**

I. Introduction

Our previous work [1] has given a thorough treatment of the reliability of disk drives composed into arrays. Some metrics of the individual drives, such as the mean time to failure (MTTF), are more optimistic than real-world measured results [2]. Furthermore, preventative reporting mechanisms like SMART (Self-Monitoring, Analysis, and Reporting Technology) are not a reliable means of proactively preventing data loss by identifying likely drive failures [3]. Even bit error rates are becoming more important as individual drives become larger. By showing failure rates for several configurations and situations, we concluded that more reliable storage configurations are required to avoid data corruption. While nested RAID configurations (*e.g.*, RAID 1+0 and RAID 5+0) can somewhat improve reliability for large arrays, this is an inefficient use of hardware resources and doesn't increase reliability drastically. Furthermore, it is an inefficient use of hardware resources, requiring more disks and (in some cases) more RAID controllers. This increases the cost per unit of storage.

We proposed extending the RAID philosophy by implementing storage groups with arbitrary numbers of parity disks, then described the challenges of implementing such a system. Generation of data on the parity disks requires some type of error correcting code. An example of a common space-efficient coding scheme for generating arbitrary amounts of parity is Reed-Solomon coding [4]. In an example system of $n + m$ disks, a code can be created such that any n disks can be used to regenerate the content of any other m disks in the set.

While space-efficient, Reed-Solomon coding is difficult to implement well on CPUs. For an $n+m$ system, generating m bytes of parity for n bytes of data requires nm multiplication operations in a finite field [5], where n is customarily larger than m. Unfortunately, a typical multiplication of two elements can involve several dozen elementary operations. A lookup table is a common optimization [5], but most x86/x86-64 CPUs do not implement an operator to perform parallel table lookups [6]. SSE5 may allow such vectorization in the future, but this instruction set has yet to be implemented. These conditions cause x86/x86-64 CPUs to be slow in performing large numbers of finite field multiplications, the majority of operations required for Reed-Solomon coding.

In order to deal with the high cost of computing the parity for $n + m$ systems where $m > 2$, we proposed using an outboard compute device which is better suited to Reed-Solomon coding. In particular, we detailed the advantages of using a GPU, while addressing integration into a RAID system. We ran benchmarks on a $3 + 3$ implementation of the Reed-Solomon coding component, showing up to a tenfold improvement over a CPU running a well-known Reed-Solomon coding library.

In this paper, we show the implementation and performance of a generalized GPU coding library. It is capable of generating arbitrary amounts of parity and recovering up to m erasures.

II. Motivation: GPU RAID

The intended application of this library will be a RAID system which utilizes a GPU to perform parity generation. This is a challenging application because of the nature of GPU computation, which is vastly different from that of traditional RAID controllers and CPUs used for software RAID. Being that RAID controllers are in-line computation devices, high bandwidth and low latency are easily achievable while performing the parity computations. Similarly, CPUs operate directly on main memory, allowing partial results to be written to disk while the CPU is still working on other parts of the same task. GPUs have a different method of computing: In order to be efficient, large contiguous buffers must be transferred to the GPU, operated upon, then transferred back into main memory for writing to disk. This introduces an unavoidable amount of latency.

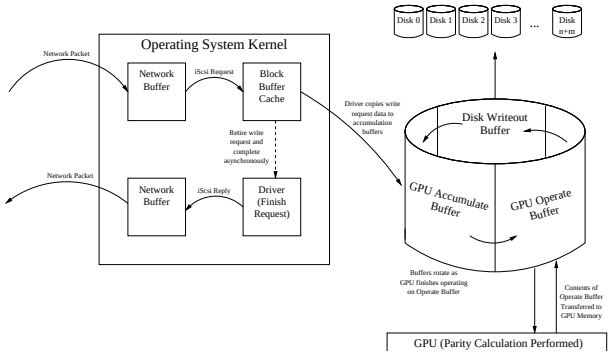

Fig. 1. A GPU RAID Architecture for Write Access

In order to combat such latency, a system architecture (shown in Figure 1) has been proposed which allows the use of the full bandwidth available from either the GPU or the disk system, whichever is slower, with little apparent latency for writes from the user perspective. The main operation of the host-side portion of the system is the gathering of requests into buffers. As shown in Figure 1, the driver which accepts the write requests copies the relevant data into the GPU Accumulate Buffer, which gathers requests while the GPU is busy doing other work. Once the GPU becomes available, the buffers are rotated so new requests go into an empty buffer and the GPU Accumulate Buffer is made to be the GPU Operate Buffer. The GPU performs DMA transfers from this buffer, operates on its contents, and returns the parity portions to the GPU Operate Buffer. When this is completed (and outstanding disk write requests are completed), the buffers are rotated again. The GPU Operate Buffer moves to the Disk Writeout Buffer position, and its contents are flushed to disk.

Figure 1 is a simplified diagram which assumes full stripe updates rather than partial stripe updates, with the implementation of the experiments mirroring this assumption. Further revisions of this work will include update calculations for partial stripe updates. Also, an often-discussed downside to this architecture is that there is a need for three times the buffer space compared to other more traditional RAID implementations.

III. Reed-Solomon and GPU Architecture

Implementation of Reed-Solomon coding on CPUs leads to one of two situations which are extremely harmful to performance. They both related to Reed-Solomon coding's most basic operation, multiplication of elements within a finite field. The implementer must either use dozens of elementary operations per finite field multiplication [5], or (if the implementer is targeting an x86-based CPU) the implementer must forgo use of vector instruction sets [7] by using lookup tables. While the lookup table option is faster, it still only offers speeds of tens of megabytes per second [1].

GPUs, on the other hand, are well-suited to performing parallel table lookups. Graphics workloads often involve sampling textures in parallel, so it is in the manufacturer's best interest to make this operation very fast in modern GPUs. In NVIDIA's

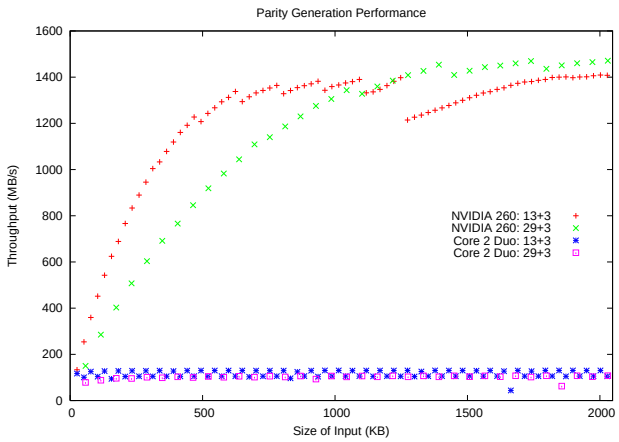

Fig. 2. Throughput of Parity Generation with 2MB Maximum Buffer

CUDA architecture as implemented on the NVIDIA 280, for example, while there is a large global memory, there are thirty small local memories with sixteen banks each [8]. Each memory is accessed by eight processing cores simultaneously, allowing very high numbers of parallel accesses to occur every clock cycle.

IV. Results

For the experimental validation of the GPU's effectiveness at performing RAID computations, a generalized version of the GPU stage of the pipeline was implemented. This implementation fixes at compile time the number of parity and data blocks per stripe. The test hardware includes an Intel Core 2 Duo E8500, 2 GB of DDR3 RAM, and an NVIDIA GeForce 260.

The first experiment was to measure the bandwidth of parity generation. The timing consisted of transferring some amount of data (denoted on the horizontal axis) to the GPU, allowing the GPU to perform the parity generation, then transferring the parity back to the host memory. The throughput is calculated from the perspective of the user application; that is, it is calculated relative to the user data transferred through the system instead of the data and parity together. In order to show the relatively small host footprint of this system, the buffer for data was restricted to two megabytes.

As shown in Figure 2, even with the restricted memory requirements, both systems tested performed well, achieving nearly 1.5 GB/s. For the $13+3$ system and the $29+3$ system, this would imply satisfying near 100 MB/s and 51 MB/s per disk, respectively. By comparison, the CPU-based algorithms averaged around 100 MB/s overall. Such throughput would support less than 7 MB/s for a $13+3$ system, and less than 4 MB/s for a $29+3$ system.

The second experiment shows recovery throughput for three failed disks in each system. Once again, this is from the user

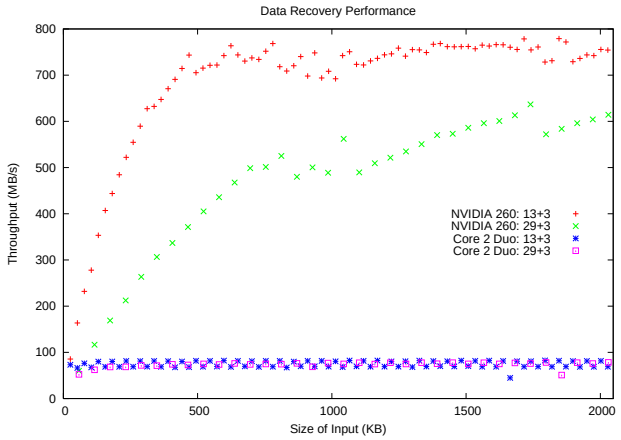

Fig. 3. Throughput of Recovery from M Failures with 2MB Maximum Buffer

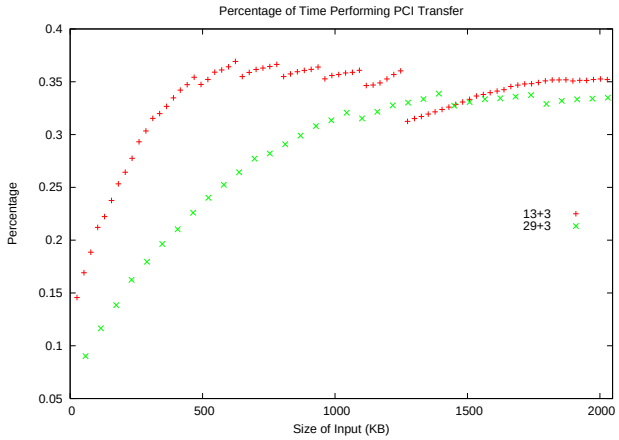

Fig. 4. Percentage of time spent transferring over PCI-Express during computation.

perspective: The measured rate is the rate of data delivered to the user application.

Figure 3 shows the recovery rates. While the overall demonstrated throughput is lower than that of parity generation, the performance seen far exceeds the 100 MB/s demonstrated for software implementations of $n + 3$.

One interesting effect of the computation model of the GPU is demonstrated by the inflection points in the performance graph, especially in Figure 2. The GPU is a SPMD compute architecture which acts much like a vector processor, with an added requirement that groups of threads that start together must all finish before new threads begin execution. A result of this architecture is that a particular workload is expressed as a number of threads which can be specified independently of the number of processors available in the GPU. This eases the task of programming the GPU, but can lead to inefficiencies under certain workloads. For example, the NVIDIA 260 has 192 processing cores. If 4×192 threads are started, all cores can potentially be fully utilized. However, if $4 \times 192 + 1$ threads are started, one thread will ultimately run alone on hardware that can support many more threads. Because four threads must run per processor to keep it fully utilized, this slightly larger load results in a nearly fifty percent drop in parallel efficiency compared to the fully populated device.

Figure 4 shows the possibility of hiding the PCI-Express transfer time to the GPU. At no point does the cost of the transfer exceed the cost of performing parity generation on a batch of the same size.

V. Conclusion

While the case study in our original work [1] shows initial promise for GPUs in RAID systems, this new implementation shows that real RAID systems of production-level sizes are supportable. With further development of this technology, it will become possible for more reliable RAID systems to be developed than are currently on the market. Furthermore, such systems can be built inexpensively with a $300 GPU and a JBOD disk controller. Such a system can support the RAID workload with good performance, often providing near full bandwidth of disks for streaming writes during normal operation, and respectable performance while operating in degraded mode.

Acknowledgments

This material is based upon work supported by the Department of Energy under Award Number DE-FC02-06ER25767.

References

[1] M. L. Curry, A. Skjellum, H. L. Ward, and R. Brightwell, "Accelerating Reed-Solomon coding in RAID systems with GPUs," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008.

[2] B. Schroeder and G. A. Gibson, "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*. Berkeley, CA, USA: USENIX Association, 2007, pp. 1–1.

[3] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2007, pp. 17–28.

[4] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960. [Online]. Available: http://www.jstor.org/stable/2098968

[5] J. S. Plank, "A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems," *Software – Practice & Experience*, vol. 27, no. 9, pp. 995–1012, September 1997.

[6] H. P. Anvin, "The mathematics of RAID-6," http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf, 2007, accessed on April 8, 2008.

[7] R. Bhaskar, P. K. Dubey, V. Kumar, and A. Rudra, "Efficient Galois field arithmetic on SIMD architectures," in *SPAA '03: Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. New York, NY, USA: ACM, 2003, pp. 256–257.

[8] *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, NVIDIA, 2007.