A Generic FPGA Accelerator for Minimum Storage Regenerating Codes

Mian Qin*, Joo Hwan Lee†, Rekha Pitchumani†, Yang Seok Ki†, Narasimha Reddy* and Paul V. Gratz*
*Texas A&M University, USA
{celery1124, reddy}@tamu.edu, pgratz@gratz1.com
†Samsung Semiconductor Inc., USA
{joohwan.lee, r.pitchumani, yangseok.ki}@samsung.com

Abstract— Erasure coding is widely used in storage systems to achieve fault tolerance while minimizing the storage overhead. Recently, Minimum Storage Regenerating (MSR) codes are emerging to minimize repair bandwidth while maintaining the storage efficiency. Traditionally, erasure coding is implemented in the storage software stacks, which hinders normal operations and blocks resources that could be serving other user needs due to poor cache performance and costs high CPU and memory utilizations. In this paper, we propose a generic FPGA accelerator for MSR codes encoding/decoding which maximizes the computation parallelism and minimizes the data movement between off-chip DRAM and the on-chip SRAM buffers. To demonstrate the efficiency of our proposed accelerator, we implemented the encoding/decoding algorithms for a specific MSR code called Zigzag code on Xilinx VCU1525 acceleration card. Our evaluation shows our proposed accelerator can achieve ∼2.4-3.1x better throughput and ∼4.2-5.7x better power efficiency compared to the state-of-art multi-core CPU implementation and ∼2.8-3.3x better throughput and ∼4.2-5.3x better power efficiency compared to a modern GPU accelerator.

I. Introduction

With the explosive growth of data in the era of cloud computing, reliability is a major concern in storage systems as their underlying components are highly susceptible to write induced wear [5]. Traditionally, replication schemes are used to provide fault tolerance. However, as the enormous scale of data volume demands, more sophisticated erasure coding techniques are used to minimize storage overhead. Currently, Maximum Distance Separable (MDS) codes, such as Reed-Solomon codes, are widely employed in both local storage systems [12] and large distributed storage systems [20, 17].

Although MDS codes provide significantly better reliability, while sacrificing the least amount of storage overhead, they impose a huge burden on repair bandwidth when rebuilding data in the event of failure [4]. Recently, a new class of erasure codes called Minimum Storage Regeneration (MSR) codes have been proposed [4, 18, 19] as an alternative to MDS codes. MSR codes minimize the data required for rebuilding while maintaining optimal storage efficiency. Although MSR codes reduce the amount of data required for rebuilding, the computation cost for encode and decode remains high, comparable to MDS codes, which are highly CPU and memory intensive [13, 10, 22]. Table I shows the experimental results for a specific MSR code (Zigzag code) encoding using GF-Complete library [14] on a modern Intel CPU. As shown in the table, the encode throughput doesn't scale well with increased number of threads. This is caused by poor cache performance which saturates the system DRAM bandwidth. Thus, it's worth considering designing more efficient hardware architecture to offload erasure coding computation from CPU.

TABLE I: Zigzag encode performance for 64MB object size using GF-Complete library [14].

# of threads	1	4	8	12	16
Throughput (GB/sec)	2.18	7.67	10.64	10.96	10.98
LLC hit rate	0.4	0.014	0.02	0.007	0.007
DRAM util (GB/sec)	9.53	40.99	59.55	63.22	64.60

Traditional accelerators such as GPUs and FPGAs suffer from extra data movement between host and accelerator memory [2]. However, recent efforts of RDMA NICs [6] and the emerging PCIe peer-to-peer (P2P) communication between PCIe devices [1] (such as NVMe SSDs, NICs and accelerators) enable inter and intra server data movement to be almost free with minimum CPU intervention. With these efforts, the offloaded erasure coding computation can be carried out in the accelerator on the fly without moving data back and forth between the host and the accelerator. This makes offloading erasure coding computation further appealing.

The above observations motivate us to design efficient accelerators for MSR erasure code, which can free the host CPU and memory for supporting other applications; a solution that is both economical (cheap hardware versus expensive server CPU) and power/energy efficient. Considering erasure coding is pure fixed-point computation, FPGA is a more efficient platform compared to floating-point optimized GPU.

In this paper, we describe a generic FPGA accelerator to perform the code construction and data rebuild for Minimum Storage Regenerating Codes. In our design, we leverage the abundant logic and memory resources in FPGA to provide massive parallelism for encode/decode computation and reduce unnecessary data movement between off-chip DRAM and FPGA on-chip BRAM buffer through analyzing the memory access pattern for MSR code construction and data rebuild. We implement our accelerator on a Xilinx VCU1525 board and compare against the state-of-art software MSR code implementation with GF-Complete library [14]. Our proposed design shows superior benefits on both performance and power efficiency.

To summarize, we make the following contributions:

1) A generic hardware architecture to process code construction and data rebuild for MSR codes. This architecture maximizes parallelism for the finite field operations used in erasure codes and minimizes data movement

978-1-7281-4123-7/20/$31.00 ©2020 IEEE

from off-chip memory, to address the problems in traditional CPU implementation.

2) Demonstration of a flexible and easy to maintain OpenCL implementation leveraging Xilinx High Level Synthesis to implement such an accelerator for MSR code construction and data rebuild.

3) Experimental evaluation of the proposed approaches on a state-of-art FPGA accelerator card, comparing performance with CPU and GPU implementation.

II. Background

In this section, we briefly describe the theory of erasure coding and Minimum Storage Regeneration (MSR) codes. Then we demonstrate the code construction and data rebuild algorithms for a specific MSR code called Zigzag [18] code.

A. Erasure Code and MDS codes

In storage systems, erasure codes are exploited to tolerate storage failures with less extra storage. Maximum Distance Separable (MDS) codes achieve ideal storage overhead. Consider an erasure coded system composed with total number of n nodes. We split them into k information nodes and $r = n-k$ parity nodes. We denote the erasure code configuration as $\{n,\ k\}$, and we refer to a node as an independent failure point such as a disk or a storage node in the data center. We stripe the data object (a.k.a. stripe) into k even size information fragments and apply erasure codes to generate r even size parity fragments and store them in the information nodes and parity nodes respectively. MDS codes have the property that they can recover from up to $n-k$ failures of any nodes.

The encode procedure of MDS codes can be generalized as linear arithmetic operations in Galois Field as shown in equations 1 where each element in the matrix is a codeword (minimum data size to operate in Galois Field). The decode procedure for m-node failure ($m \leq n-k$ where $n-k$ is the maximum number of nodes failure that MDS codes can tolerate) can be achieved by solving the linear equation 1 (the coefficients matrix C must be invertible to guarantee the feasibility of decoding).

$$\begin{bmatrix} P_1 \\ P_2 \\ \vdots \\ P_m \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,k} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m,1} & c_{m,2} & \cdots & c_{m,k} \end{bmatrix} * \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_k \end{bmatrix} \quad (1)$$

B. Minimum Storage Regenerating (MSR) codes

Regenerating codes were first introduced by Dimakis *et al.* [4] to reduce the high repair bandwidth of MDS codes in distributed storage systems. Minimum Storage Regenerating (MSR) codes offer the same storage-availability trade-off as MDS codes while minimizing the repair bandwidth. Here we will briefly introduce the specific MSR code used in this paper, Zigzag [18] code, with an example to intuitively illustrate how MSR codes generally work. Other MSR codes [19, 21] follow the same principles.

Zigzag encode. The data object to be stored will be first split into k even fragments. Each fragment is further partitioned into m data elements as shown in Fig 1 (when $m = 1$, it degenerates into to MDS code). In this paper, we will refer $\{n,\ k,\ m\}$ as the configuration parameters for Zigzag code where n is the total number of storage nodes. (For detailed zigzag code parameters, please refer to [18].) The Zigzag code parities are encoded as follows:

1) For each data element in a parity fragment, find a specific data element in each information fragment (the specific data element index is determined by the code design), totally k data elements.

2) Each data element in the parity fragment is generated by the k corresponded information data elements using Galois Field operations with the following formula:
$p_i = \sum_{j=1}^{k} C_j d_{j,i} \quad (1 \leq i \leq N)$
Where N is the number of codewords in each data element.

We generalize several parameters for the above procedure. For each data element in the parity fragments, there is a set of indices $\{I_1, I_2, \cdots, I_k\}$ indicating the location of the data element in each information fragment and a set of coefficients $\{C_1, C_2, \cdots, C_k\}$ for calculating the parity data element. In total there are $(n-k) * m$ sets of those indices/coefficients parameters to finish the entire encode procedure.

To better understand the description above, consider an MSR coded storage system with 4 information nodes and 2 parity nodes as shown in Fig 1. Each data fragment contains 8 data elements. Codewords in the first and third data element of the first parity fragment are calculated as:

$$P1_{r1} = 1 * D1_{r1} + 1 * D2_{r1} + 1 * D3_{r1} + 1 * D4_{r1} \quad (2)$$
$$P2_{r3} = 1 * D1_{r3} + 2 * D2_{r4} + 1 * D3_{r1} + 1 * D4_{r7} \quad (3)$$

The corresponding indices sets are $\{1, 1, 1, 1\}$, $\{3, 4, 1, 7\}$. The coefficients sets are $\{1, 1, 1, 1\}$, $\{1, 2, 1, 1\}$.

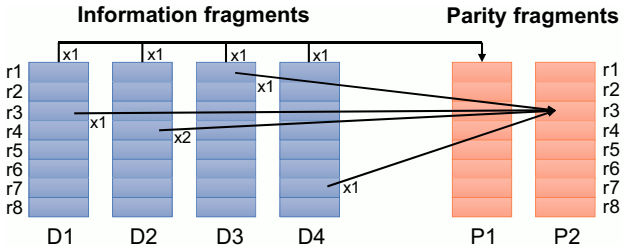


Fig. 1: MSR codes encode example.

Zigzag decode. In this paper, we focus on MSR code rebuild for only the single erasure case, since single node failure is the most common case [16].

The data rebuild formula for single erasure is nearly identical to the code construction formula (linear algebra transformation). Similarly, we define indices set $\{I_1, I_2, \cdots, I_k\}$ to indicate the location of the data elements in surviving information/parity fragments needed for rebuild (each rebuild data element is generated from k information/parity data elements [18]) and coefficients set $\{C_1, C_2, \cdots, C_k\}$ for calculating rebuild data words in each data element. As shown in Fig 2, the rebuild data in the first and third data element in the erased fragment is calculated as:

$$D1_{r1} = 1 * P1_{r1} + 1 * D2_{r1} + 1 * D3_{r1} + 1 * D4_{r1} \quad (4)$$
$$D1_{r3} = 1 * P2_{r3} + 2 * D2_{r4} + 1 * D3_{r1} + 1 * D4_{r7} \quad (5)$$

As illustrated in Fig 2, the rebuild for single erasure case for MSR codes require much less the data compared to conventional MDS codes such as Reed-Solomon codes.

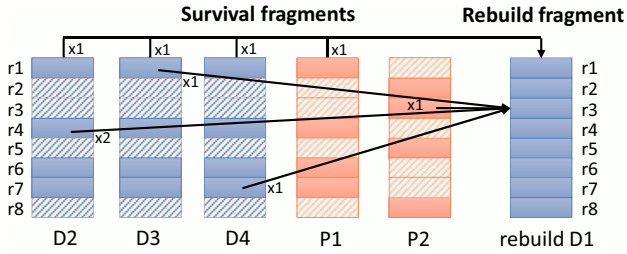


Fig. 2: MSR decode example (The solid filled boxes are the data needed for rebuild.)

III. Proposed Architecture

In this section, we will describe the accelerator architecture for encode/decode offloading for Zigzag code. While it is intended for Zigzag code, this architecture can be easily extended to other MSR codes.

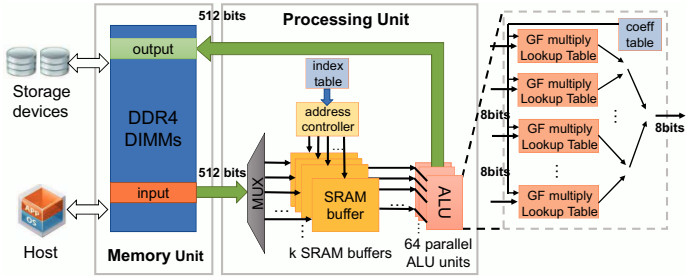


Fig. 3: Overall accelerator architecture.

The overall diagram for our proposed architecture is shown in Fig 3. The architecture is mainly composed of two components. First, the *memory unit* holds the information and parity fragments that are transferred from host memory or storage devices. The memory unit uses the off-chip DDR memory connected to the FPGA. Second, the *processing unit* which process the data from the memory unit and perform the actual encode/decode computation. There can be more than one *processing unit* connected to the *memory unit* to fully utilize the off-chip DRAM bandwidth and hide memory latency, assuming FPGA resources are available.

A. Memory Unit

The memory unit temporarily holds the input data for the encode/decode processing and the output results (parity fragments and rebuild information fragment for encode and decode respectively). For encoding, the information fragments will be transferred to the memory unit from the host. The encoded parity fragments will be written back to the memory unit after processing unit fetches the information fragments and finishes processing. Finally, the information and parity fragments will be transferred to the storage nodes through P2P transfer. For decoding, the data fragments needed for rebuild will be transferred to the memory unit from the surviving storage nodes through P2P transfer. After the processing unit finishes the decoding process, the rebuild data will be stored in the memory unit and transferred back to the host or to a new storage node depending on the recovery process. All the input/output buffers in the memory unit are allocated/deallocated through the OpenCL framework dynamically.

B. Processing Unit

The processing unit consists of mainly three parts. The SRAM buffers which hold all or part of the input data for the encode/decode process. The address calculation controller which manages how the data is fed in the SRAM buffers from the memory unit and how the data is read from the SRAM buffers for encode/decode computation and how the results are written back to the memory unit. The ALU unit which computes the Galois Field multiply-add arithmetic.

SRAM buffers. In each processing unit, we use k separate SRAM buffers where k is the number of information nodes in our Zigzag code configuration to hold partial or all of input data for computation. The SRAM buffers are implemented using the BRAMs in the FPGA. The SRAM buffers are a key design to minimize the traffic to the memory unit. Taking encode process as an example (which is similar to the decode process), remember that each codeword in the parity fragments is generated by operating on k codewords from k different information fragments with different relative offsets. To improve the data reuse rate, we need to buffer all the data elements for every information fragment in the SRAM for future use. Thus, k SRAM buffers will buffer all the codewords required to calculate the codewords for all parity fragments. With the design of k separate SRAM buffers, each byte of the input data only needs to be read **once** from the memory unit to the SRAM buffers once which significantly minimize the data movement between off-chip DDR memory and FPGA logic.

To maximize the memory unit bandwidth utilization and the process throughput, the data are packed to 512 bits when being transferred from or to the memory unit. Each memory buffer is organized as 512 bits width dual-ports RAM. Thus, the data is read, written and processed in 512 bits granularity per cycle in the processing unit.

The detailed illustration for the memory layout of the input and output data in the memory unit and how data is moved into the SRAM buffers will be demonstrated in section III-C.

Address calculation controller. The address calculation controller is the most complex control unit. It has three tasks.

- Read the data from memory unit input buffer to k SRAM buffers. This includes slicing the data elements and read per sliced data in each data element to the SRAM buffers when stripe size is too large.
- Read the data from SRAM buffers in parallel and feed to the ALU units for the encode/decode computation (Galois Field arithmetic).
- Write the results (parity codewords or rebuilt data words) to the output buffer in the memory unit.

Once the Zigzag code configuration $\{n, k, m\}$ is fixed, the indices sets for accessing the information fragments to generate each parity data element are also fixed. We precalculate these indices sets offline and use a table to store these indices sets in the FPGA. These indices sets will be used for the address calculation controller to fetch the data from SRAM buffers to the ALU units to perform the computations.

ALU unit. The ALU unit are the core computation logic to perform the Galois field arithmetic to generate the parity and do data rebuild for Zigzag code. As we discussed in section II, both encode and decode process for the Zigzag

code or any other erasure codes are composed of only Galois Field multiply-add operation. Thus, our ALU unit is designed to perform only Galois Field multiply-add operation. In our implementation, we use lookup tables to implement Galois Field multiply and bitwise XOR to implement Galois Field add which can make the most of the massive LUT resources in FPGA. All Galois Field operations are in 8-bit granularity which is a good parameter for lookup table size. Unlike the "single-instruction-multiple-data" (SIMD) unit in the CPU which only operates on two input operands, we leverage the abundant logic resources in the FPGA and designed a pipelined tree structure to perform multiple inputs gf multiply-add operations in pipeline as shown in Fig 3 on the right side. Similar as the indices sets, we store the fixed coefficients sets as tables in the FPGA to compute parities.

The pipelined tree structure for Galois Field multiply-add operation in our design has two advantages compared to the SIMD unit in the CPU. First, data is processed with better parallelism. Second, to generate each output codeword, each input codeword (operand) only needs to be read once from the SRAM buffers. While in the CPU implementations, this needs to be done in a loop to read the input codewords (operands) from cache iteratively. Since the useful cache lines may be evicted to lower level cache or even DRAM, this will cause stalls in the SIMD pipeline and extra power to move the data.

C. Process Stages

To better demonstrate how our accelerator works, we will describe the process stages for single encode or decode task. Since the computation and data flow for encode and decode are similar, we do not differentiate encode and decode.

The processing unit is able to handle an arbitrary length stripe size. This is important for erasure codes since different storage systems may require different stripe sizes. Thus, the process stages for each encode/decode task may include one or more passes, each process pass contains three phases as follows.

Read phase. In the read phase, the address calculation controller will control the memory read from the off-chip memory unit and write to the SRAM buffers. Each SRAM buffer holds part or all of the input data fragment. If the size of the input data is small enough that can be filled entirely in the SRAM buffers, the whole process will be done in one pass. However, if the size of the input data is larger than the SRAM buffers, the input data will be partitioned properly and read into to the SRAM buffers for further processing. In this way, the whole process will be done in several passes.

To maximize the off-chip memory unit bandwidth performance and reduce energy, the partitioned areas are 4KB to match the internal DRAM page size to improve row locality. If the stripe size is small enough that can be filled entirely in the SRAM buffers, all the data will be fed into the SRAM buffers in one pass (sequentially read for each fragment). If the stripe size is too large the data will be read from memory unit data slice by data slice to the SRAM buffers.

Computation phase. In the computation phase, the processing unit will apply the code construction and data rebuild algorithm described in section II-B. The indices sets and coefficients sets for the data element will be applied here for each data slice. The address calculation controller unit will control the memory read according to the pre-stored indices table and read the correct data slices from the k SRAM buffers simultaneously. The read data will be fed to the ALU units for parity calculation or data rebuild as described in section II-B.

Write phase. Since computation phase is fully pipelined, the output results from the ALU units can be written to the off-chip memory unit immediately. It can be considered as adding one more pipeline stage after the XOR tree. Since the data is partitioned when read into the SRAM buffers, the output results' written back to the memory unit is also partitioned. In the first process pass, the parities generated will be written to the output fragments. In the second process pass, the parities generated will be written to the output fragments.

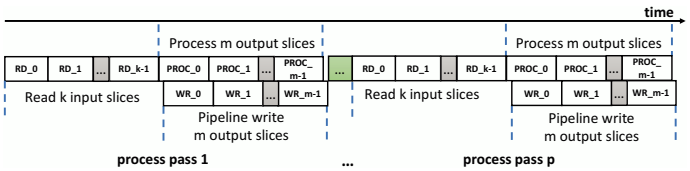


Fig. 4: Timing diagram of the process stages workflow.

Fig 4 illustrates the timing diagram of the three process stages for the accelerator. Consider Zigzag code n, k, m, there are k input data fragments and $r = n - k$ output data fragments. Let's take encode procedure as an example here (for decode is similar). The input data is larger than the internal SRAM buffers size and it needs p passes to process the whole input data and p is equal to the number of data slices partitioned for each input data element.

D. Other Considerations

Multiplexing resources for encode and decode. Since both the data flow and computation for encode and decode are similar, as shown in section II-B. We can multiplex most of the hardware resources (SRAM buffers, ALU units) to conduct both encode and decode procedure. In our design, we have separate tables to store the indices/coefficients sets for encode and decode. The host can setup different kernel parameters to control the kernel launch of different functions (decode or encode).

Batch processing. For processing small size data, the kernel launch overhead and data migration overhead from host to accelerator and vice versa is non-negligible. In our design, we also implement a batch process to process multiple same size input data in single kernel launch. The batch size is also a separate parameter for setting up the kernel. The batch process support is implemented by slightly modifying the address calculation controller to continuously read, compute and write after finishing each encode or decode task.

IV. Implementation and Evaluation

A. System Setup

We implemented our accelerator for a $\{6, 4, 8\}$ Zigzag coding system on a Xilinx Virtex UltraScale+ FPGA VCU1525 acceleration card with 4 DDR4-2400 SDRAMs. The GPU implementation is on a Tesla K80 GPU acceleration card with 240 GB/sec GDDR5 memory. The host machine has a 2.1GHz

Intel Xeon Gold 6152 CPU with 22 cores and a 30.25MB L3 cache. There are 4 DDR4-2666 SDRAMs on the host machine. Although we only implement and evaluate on a local storage system, the results can be also extended to distributed storage systems. Our FPGA accelerator is developed in Xilinx SDaccel toolchain. Software CPU implementation is developed in C++ with GF-Complete library [14]. GPU implementation is developed using CUDA toolkit.

We evaluate Zigzag code encoding/decoding a wide spectrum of object size (stripe size) from few kilobytes to tens of megabytes for potential use cases. Usually RAID systems use smaller (64KB to 256KB) stripe sizes [12, 18], while the cloud storage [8, 15, 19] industry tends to use much larger stripe sizes, on the order of tens of MB.

B. Resource utilization

The FPGA resource utilization and kernel frequency are shown in Table II. This implementation uses all 4 DDR4 channels on board and each channel (memory unit) implements three processing units (PUs). We use 32KB SRAM buffers for each PU (4KB buffer per storage node to maximize the DDR bandwidth utilization). The resource utilization and timing result include platform cost for implementing OpenCL framework and are post route results.

TABLE II: System resource utilization on VCU1525 accel. board.

Resource Type	Used	Available	Util%
CLB Registers	552005	2364480	23.35
CLB LUTs	376287	1182240	31.83
Block RAMs (36Kb)	1050	2160	48.61
Kernel clock frequency	300MHz		
Platform clock frequency*	300MHz		

* Platform clock include the clock domain for OpenCL implementation (memory controllers, PCIe endpoints, interconnect, etc.)

C. Performance of Zigzag encode/decode

Here we compare our FPGA implementation against the state-of-art CPU implementation leveraging SIMD instructions [14] and GPU implementation. For the software CPU implementation we use different numbers of threads to process in parallel (each thread processes a complete encode/decode task). For GPU implementation, each thread processes only a few 32 bits GF multiply-adds for a encode/decode task to fully exploit the "single-instruction-multiple-threads" (SIMT) parallelism. We conduct experiments on a wide range of data object sizes from (tens of kilobytes to tens of megabytes). For software implementation, the CPU runs at 2.1GHz with 85.3 GB/sec memory bandwidth. The FPGA accelerator runs at 300MHz with 76.8 GB/sec memory bandwidth. GPU accelerator runs at 875MHz with 240 GB/sec memory bandwidth. As shown in Fig 5a, compared to peak CPU implementation, our FPGA accelerator achieves similar performance for smaller stripe size and 3.1x better on encode and 2.4x better on decode for larger stripe sizes. Our FPGA accelerator also surpasses the GPU implementation by $\sim$2-3x.

There are two reasons that our accelerator achieves better performance. First, our accelerator design optimizes the data fetch and store from the memory unit to the on-chip SRAM buffers and has much better DRAM bandwidth utilization. We collected the memory traffic for the software implementation via performance counters and compared against our accelerator. Our accelerator can reduce up to 20% of the DRAM traffic compared to CPU and 43% compared to GPU. The extra DRAM traffic in the CPU implementation is caused by poor cache performance and cache thrashing in multicore workloads for large stripe size. Second, our accelerator achieves better computation parallelism by using multi-operand GF multiply-add ALUs compared to two operands SIMD ALUs in CPU architecture. Compared to GPU, the hardware level parallelism in FPGA is much more efficient than SIMT. Thus, even though our accelerator runs at much lower frequency and memory bandwidth ($\sim$3x less than GPU) the performance still surpasses the CPU SIMD and GPU implementation.

D. Power efficiency

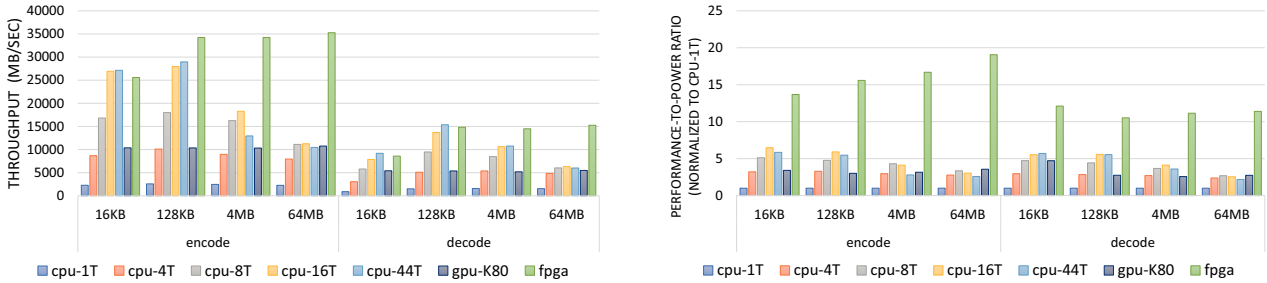
We also did the performance-to-power ratio analysis to estimate power efficiency. We calculate the total power of core (CPU or FPGA) and off-chip DRAMs. For the CPU implementation we obtained the dynamic power consumption through the Intel Performance Counter Monitor. For GPU implementation we obtained the overall power consumption through the GPU driver. We use the Xilinx SDaccel toolchain to estimate the FPGA power (worst case scenario) and a power calculator by Micron to estimate the accelerator DRAM power consumption to get the overall power of our accelerator. Fig 5b shows performance-to-power ratio comparison. Our accelerator achieves up to 19.1x and 11.4x better compared to single thread CPU implementation on encode and decode respectively. Compared to the best CPU implementation results, our accelerator is 5.7x and 4.2x better on encode and decode respectively. Compared to GPU implementation results, our accelerator is 5.3x and 4.1x better on encode and decode respectively. We also analyze the raw power consumption data for the CPU implementation and our accelerator. We found our accelerator consumes less power on both core (worst case) and DRAM since the FPGA runs on a much lower clock frequency and we significantly reduce the DRAM traffic.

V. Related Work

Plank, *et al.* [11] proposes efficient "bitmatrix" representation for Reed-Solomon code to reduce the number of GF multiplications. Plank, *et al.* [14] proposes to leverage SIMD instructions to accelerate erasure coding on CPU platforms. In Section I we show that SIMD optimizations may now scale well for multiple threads due to poor cache performance. Kalcher, *et al.* [9] and Curry, *et al.* [3] introduce to leverage the massive computation and memory bandwidth on GPU to accelerate coding performance of Reed-Solomon code. Chen, *et al.* [7] proposes an OpenCL based FPGA implementation for erasure coding. Prior works focus on accelerating MDS codes. To the best of our knowledge, our work is the first to optimize hardware architecture for accelerating MSR codes.

VI. Concludsion

In this paper, we present a generic FPGA accelerator architecture for Minimum Storage Regenerating (MSR) codes in reliable storage systems. In our design, we leverage the abundant FPGA logic and memory resources to provide massive parallelism for encode/decode computation and optimize the data movement between off-chip DRAM and FPGA. Under evaluation on real systems, we show our proposed



(a) Throughput performance (higher is better).

(b) Performance-to-power ratio (higher is better).

Fig. 5: Encode/decode evaluation results (We enable 4MB batch mode for 16KB and 128KB stripe size for both FPGA and GPU).

accelerator's performance surpasses the state-of-art multi-core CPU implementation on both throughput and power efficiency. The design can be beneficial for storage system acceleration especially with PCIE P2P communication enabled.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and feedback. The project is funded by Memory Solution Lab (MSL) in Samsung Semiconductor Inc. and the National Science Foundation I/UCRC-1439722 and FoMR-1823403.

References

[1] Stephen Bates. *Donard: NVM Express for Peer-2-Peer between SSDs and other PCIe Devices*. 2015.

[2] S. Che et al. "Accelerating Compute-Intensive Applications with GPUs and FPGAs". In: *2008 Symposium on Application Specific Processors*. 2008, pp. 101–107.

[3] M. L. Curry et al. "Accelerating Reed-Solomon coding in RAID systems with GPUs". In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. 2008, pp. 1–6.

[4] A. G. Dimakis et al. "Network Coding for Distributed Storage Systems". In: *IEEE Transactions on Information Theory* 56.9 (2010), pp. 4539–4551.

[5] Sanjay Ghemawat et al. "The Google File System". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: ACM, 2003, pp. 29–43.

[6] Chuanxiong Guo et al. "RDMA over Commodity Ethernet at Scale". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: ACM, 2016, pp. 202–215.

[7] Guoyang Chen et al. "OpenCL-based erasure coding on heterogeneous architectures". In: *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2016, pp. 33–40.

[8] Cheng Huang et al. "Erasure Coding in Windows Azure Storage". In: *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX, 2012, pp. 15–26.

[9] S. Kalcher and V. Lindenstruth. "Accelerating Galois Field Arithmetic for Reed-Solomon Erasure Codes in Storage Applications". In: *2011 IEEE International Conference on Cluster Computing*. 2011, pp. 290–298.

[10] Osama Khan et al. "Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads". In: *10th USENIX Conference on File and Storage Technologies (FAST 12)*. San Jose, CA: USENIX Association, 2012.

[11] J. S. Plank and Lihao Xu. "Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Network Storage Applications". In: *Fifth IEEE International Symposium on Network Computing and Applications (NCA'06)*. 2006, pp. 173–180.

[12] James S. Plank. "The RAID-6 Liberation Codes". In: *Proceedings of the 6th USENIX Conference on File and Storage Technologies*. FAST'08. San Jose, California: USENIX Association, 2008, 7:1–7:14.

[13] James S. Plank et al. "A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage". In: *7th USENIX Conference on File and Storage Technologies (FAST 09)*. San Francisco, CA: USENIX Association, 2009.

[14] James S. Plank et al. "Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions". In: *11th USENIX Conference on File and Storage Technologies (FAST 13)*. San Jose, CA: USENIX Association, 2013, pp. 298–306.

[15] Maheswaran Sathiamoorthy et al. "XORing Elephants: Novel Erasure Codes for Big Data". In: *Proc. VLDB Endow.* 6.5 (Mar. 2013), pp. 325–336.

[16] Bianca Schroeder and Garth A. Gibson. "Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You?" In: *5th USENIX Conference on File and Storage Technologies (FAST 07)*. San Jose, CA: USENIX Association, 2007.

[17] K. Shvachko et al. "The Hadoop Distributed File System". In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 2010, pp. 1–10.

[18] I. Tamo et al. "Zigzag Codes: MDS Array Codes With Optimal Rebuilding". In: *IEEE Transactions on Information Theory* 59.3 (2013), pp. 1597–1616.

[19] Myna Vajha et al. "Clay Codes: Moulding MDS Codes to Yield an MSR Code". In: *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA: USENIX Association, 2018, pp. 139–154.

[20] Sage A. Weil et al. "Ceph: A Scalable, High-performance Distributed File System". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. Seattle, Washington: USENIX Association, 2006, pp. 307–320.

[21] M. Ye and A. Barg. "Explicit Constructions of Optimal-Access MDS Codes With Nearly Optimal Sub-Packetization". In: *IEEE Transactions on Information Theory* 63.10 (2017), pp. 6307–6317.

[22] Tianli Zhou and Chao Tian. "Fast Erasure Coding for Data Storage: A Comprehensive Study of the Acceleration Techniques". In: *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, 2019, pp. 317–329.