OpenCL-Based Erasure Coding on Heterogeneous Architectures

Guoyang Chen, Huiyang Zhou, Xipeng Shen
North Carolina State University
Raleigh, NC
{gchen11, hzhou, xshen5}@ncsu.edu

Josh Gahm, Narayan Venkat, Skip Booth, John Marshall
Cisco Systems
RTP, NC
{jgahm, narvenk2, ebooth, jwm}@cisco.com

***Abstract*—Erasure coding, Reed-Solomon coding in particular, is a key technique to deal with failures in scale-out storage systems. However, due to the algorithmic complexity, the performance overhead of erasure coding can become a significant bottleneck in storage systems attempting to meet service level agreements (SLAs). Previous work has mainly leveraged SIMD (single-instruction multiple-data) instruction extensions in general purpose processors to improve the processing throughput. In this work, we exploit state-of-art heterogeneous architectures, including GPUs, APUs, and FPGAs, to accelerate erasure coding. We leverage the OpenCL framework for our target heterogeneous architectures and propose code optimizations for each target architecture. Given their different hardware characteristics, we highlight the different optimization strategies for each of the target architectures. Using the throughput metric as the ratio of the input file size over the processing latency, we achieve 2.84 GB/s on a 28-core Xeon CPU, 3.90 GB/s on an NVIDIA K40m GPU, 0.56 GB/s on an AMD Carrizo APU, and 1.19 GB/s (5.35 GB/s if only considering the kernel execution latency) on an Altera Stratix V FPGA, when processing a 836.9MB zipped file with a 30x33 encoding matrix. In comparison, the single-thread code using the Intel's ISA-L library running on the Xeon CPU has the throughput of 0.13 GB/s.**

Keywords—Erasure coding, heterogeneous architecture, OpenCL, GPU, CPU, APU, FPGA

I. INTRODUCTION

A key challenge facing storage systems is failures. Various failures may happen to disk sectors, entire disks or storage sites. To prevent data loss resulting from failures, erasure codes are widely used. Simple erasure codes like replication have high storage cost and limited error correction capability. More complex erasure codes including the well-known Reed-Solomon codes can tolerate more errors while incurring much less storage cost. The overhead of such complex erasure codes, however, lies in their computational complexity. To improve the throughput of erasure coding, prior work [1][7][8] has mainly focused on leveraging SIMD (single-instruction multiple-data) or streaming instruction extensions of general-purpose processors. Such low-level implementations are then utilized extensively in the libraries like Intel's intelligent storage acceleration library (ISA-L) [15].

In this paper, we explore using various heterogeneous architectures, including graphics processor units (GPUs), accelerated processing units (APUs), and Field Programmable Gate Arrays (FPGAs), to accelerate erasure coding. This work serves two purposes. The first is to carefully optimize the code for each accelerator architecture and evaluate which architecture best benefits erasure coding, carefully considering the trade-offs between the performance and cost of ownership. The second is to analyze the different hardware characteristics and reveal the effective code optimization strategies for these heterogeneous architectures.

We make the following contributions in this paper.

First, based on the ISA-L implementation of erasure coding, we add the multithreading support to take advantage multi-core processors to improve the throughput of erasure coding on CPUs.

Second, leveraging its inherent data-level parallelism, we optimize erasure coding for GPUs. At the algorithm-level, we explore different implementations to trade off memory footprints for computations. We then apply various bandwidth optimizations to achieve high-throughput erasure coding on GPUs. We also apply similar optimization strategies on APUs and evaluate shared virtual memory (SVM) [16] to minimize the data communication cost between system memory and device memory.

Third, for FPGAs, we leverage the recent advances on OpenCL for FPGAs [1] and develop the OpenCL kernels for erasure coding on FPGAs. The common OpenCL framework, used in the kernel code for both GPUs and FPGAs, helps to reveal the insight on code optimizations for different accelerator architectures. We highlight that although some code optimizations are universal, different hardware characteristics necessitate architecture-specific code optimization strategies.

Finally, based on the performance we achieved on these heterogeneous architectures, we conclude that FPGAs provide the highest throughput at relatively low cost of ownership, while APUs are a viable alternative to the Intel Xeon processors.

The remainder of the paper is organized as follows. In Section II, we analyze the computational complexity of the Reed-Solomon code and present different algorithms to implement the Galois Field (GF) multiplication, the basic arithmetic operation used in the Reed-Solomon codes. Section III addresses the implementation for computing the Reed-

The research is a result of faculty visits to Cisco Systems and is also partially supported by the National Science Foundation (NSF) (under Grants No. 1216569, 1455404, 1525609).

Solomon code on CPUs, including both single-core and multi-core ones. The code and the optimizations for GPUs/APUs are discussed in Section IV. Section V develops and optimizes the OpenCL code for the FPGA platform. The experimental methodology is presented in Section VI and the results are analyzed in Section VII. The related work is discussed in Section VIII. Section IX concludes the paper.

II. Computational Complexity of Reed-Solomon Coding

Reed-Solomon codes are block-based linear codes [6]. An input file is partitioned into multiple blocks and parity blocks are generated using linear combinations, as shown in Equation 1. In the equation, the input is partitioned into a number ('srcs') of blocks with a block size of 'length' bytes. With the data organized into the 2D matrix format, Equation 1 essentially becomes a matrix multiplication operation, 'Dest = V x Src', where 'Dest', 'V', and 'Src' are the output matrix, the encoding matrix, and the input matrix, respectively, as shown in Figure 1.

$$Dest[l][i] = \sum_{j=0}^{srcs-1} V[l][j] \times Src[j][i] \qquad \text{Eq.(1)}$$

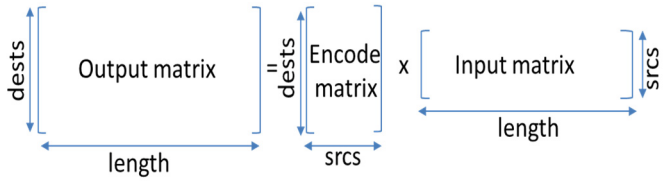

Figure 1. Block-based Parity Encoding.

As shown in Figure 1, the output matrix, 'Dest', has 'dests' parity blocks. The encode matrix, 'V', has the dimension of 'dests' by 'srcs'. Compared to regular matrix multiplication, the only difference is that the Galois Field (GF) arithmetic is used in Equation 1, where a sum is an 8-bit XOR operation and a multiplication is a GF(2^8) multiplication.

The computational complexity of Reed-Solomon codes is mainly from the GF(2^8) multiplications. To generate the output matrix, the number of GF(2^8) multiplications required is (dests*length*srcs). With the input data size as (srcs*length) bytes, the arithmetic intensity (AI) [12] of the algorithm is computed as Equation 2.

AI = overall computations / data transfers

$$= src * dests / (srcs + dests) \qquad \text{Eq. (2)}$$

From Equation 2, we can see that when 'srcs' or 'dests' is small (e.g., 5), the arithmetic intensity of the algorithm is low (e.g., 2.5 GF(2^8) multiplications per byte), which implies that it is more likely to be bandwidth bounded. Increasing 'srcs' and 'dests' (e.g., 30) turns the algorithm to be compute bounded (e.g., 15 GF(2^8) multiplications per byte). Different techniques can be used to implement GF(2^8) multiplications. One technique, referred to as the Russian Peasant algorithm, computes the product between two bytes of data through a loop, as shown in List 1, where a particular field-defining primitive polynomial $x^8+ x^4+ x^3+ x^2+1$ (0x11d) is used. The same primitive polynomial is also used in Intel ISA-L. From List 1, one can see that each GF(2^8) multiplication requires 8 iterations and each iteration contains a few logic operations. A limitation of this approach is that the loop cannot be parallelized, because of loop-carried dependence.

To reduce the computational overhead of the Russian Peasant algorithm, pre-computed tables can be used for small Galois Fields, like GF(2^8).

```
unsigned char gmul(unsigned char x, unsigned char y) {
        unsigned int a = x; unsigned int b = y;
        unsigned int p = 0; int counter;
        for (counter = 0; counter < 8; counter++) {
                if (b & 1)
                   p ^= a;
                a <<= 1;
                if (a & 0x100)
                   a ^= 0x11d;
                b >>= 1;
        }
        return (unsigned char) p;
}
```

List 1. The Russian Peasant Algorithm for GF(2^8) multiplication.

The first approach using precomputed tables leverages the properties of exponential and logarithmic computations. It uses two small pre-computed tables and the resulting code is shown in List 2. As shown in List 2, both pre-computed tables, 'gflog_base' and 'gff_base', have 256 bytes and each GF(2^8) multiplication requires 3 table-lookups, 1 add, 1 subtract, and 1 condition check.

```
unsigned char gmul(unsigned char x, unsigned char y) {
        int l;
        if ((x == 0) || (y == 0))
                return 0;
        return gff_base[(i = gflog_base[x] +
        gflog_base[y]) > 254 ? i - 255 : i];
}
```

List 2. GF(2^8) multiplication using two small pre-computed tables. (From ISA-L[15])

The second approach pre-computes the product for all the possible input values and uses one table lookup to retrieve the product, as shown in List 3. For the GF(2^8) field, since the product is between two bytes,, the pre-computed table, 'gf_mul_table_base', has 256x256 entries, and is of size 64kB. For this implementation, each GF(2^8) multiplication requires 1 table lookup, 1 add, and 1 shift operation.

```
unsigned char gmul(unsigned char x, unsigned char y) {
        return gf_mul_table_base[x * 256 + y];
}
```

List 3. GF(2^8) multiplication using one large pre-computed table. (From ISA-L[15])

In summary, among these three implementations, since the Russian Peasant algorithm does not require any precomputed tables, it has the smallest memory footprint but requires the highest number of computations. The approach with the large pre-computation table is shown in List 3 has the fewest number

of computations but requires a 64kB precomputation table. The small precomputation table approach shown in List 2 requires much smaller storage (two 256B-tables) but more computations as well as extra table lookups compared to the approach in List 3.

III. Reed-Solomon Coding on CPUs

State-of-art Reed-Solomon coding exploits streaming or SIMD instruction extensions to compute multiple GF multiplications concurrently [7]. The key enabler is the special instruction support to perform multiple (e.g., 16) table lookups using a single SIMD instruction. In this approach, a GF(2^8) multiplication is decomposed into multiple GF(2^4) multiplications, which are readily generated using the 16-element table lookup instruction. This implementation is discussed in detail by Plank et al. [7].

In our work, we used the SIMD-based GF(2^8) multiplication functionality in ISA-L as the baseline CPU implementation for Reed-Solomon coding. Since modern CPUs comprise of multiple cores and each core is hyper-threaded, we added multithreading support to utilize the entire CPU. To do so, we divided the output matrix into multiple partitions in a column-wise manner (the data is stored in the row-major order): where each partition has the same number of rows as the original output matrix but only a portion of columns, as shown in Figure 2. Such partitioning is done virtually by only adding pointers and does not require any data copy. Subsequently, multiple threads are generated with each one computing an independent partition.

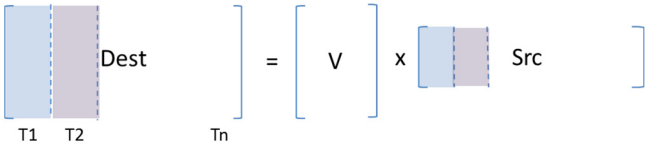

Figure 2. Coarse-grain partition for multithreaded implementation on CPUs.

IV. Reed-Solomon Coding on GPUs

From Equation 1, it can be clearly seen that the computation of each element in the output matrix is independent from others. Such abundant data-level parallelism can be leveraged by many-core architecture such as GPUs to achieve high throughput, as pointed out in prior works [4][10]. We use the fine-grain parallelization in our baseline GPU implementation, meaning that each workitem computes one byte in the output matrix and both local and global workgroups are organized in two dimensions. We then improve code performance through a series of optimizations carefully designed to take advantage of the latest hardware features of GPUs. Next, we will describe in detail the optimizations that have been considered and evaluated.

A. Vectorization to Optimize GPU Memory Bandwidth

To maximize the GPU memory access bandwidth, the memory accesses from the workitems in a wavefront (or a warp) needs to satisfy the memory coalescing requirements, i.e. the memory requests from the workitems in a wavefront can be coalesced into few transactions at the cache-line granularity (e.g., 128 Bytes). In our baseline implementation, the local workgroup is organized such that each workitem computes one byte and all workitems in a wavefront are in the same row. This way, the accesses to both the 'dest' matrix and the 'src' matrix are coalesced. Besides the coalescing requirement, it has been shown [9] that a coalesced vector-type access (e.g., uint2 or uint4) from each workitem can achieve higher bandwidth than the word-type access (e.g., uint) or the byte-type access. As in OpenCL, vector data types are defined with the type name followed by a literal value defining the number of elements in the vector. For example, the vector data type int*n*/uint*n* means a vector of *n* signed/unsigned integers. We let each workitem read a uint4 type of data (i.e., 16 bytes from a single row) from the input matrix 'Src' and compute 16 bytes in a row in the output matrix 'Dest', as shown in Equation 3. In Equation 3, the two-dimension identifiers of a workitem are labeled as 'idx' and 'idy'. From this equation, it can be seen that such vectorization transformation also enables the data reuse for the encode matrix 'V'. For each byte from V, it will be used 16 times to compute 16 output elements. This achieves the same effect as the thread merge [14] or thread coarsening optimization [5].

$$Dest[idy][idx*16 : idx*16+15] = \sum_{j=0}^{srcs-1} V[idy][j] * Src[j][idx*16 : idx*16+15] \quad \text{Eq. (3)}$$

B. Exploration of Different Implementations for GF(2^8) Multiplications

As discussed in Section 2, different implementations for GF(2^8) multiplications have different computation and memory bandwidth requirements. Our target GPU, Nvidia K40m, has a peak bandwidth of 288GB/s and a peak computational throughput of 4290GFLOP/s. As the Reed-Solomon coding uses GF(2^8) multiplication and add instead of the regular integer or floating-point multiplication and add, the effective peak computational throughput of the GPU is halved, i.e., 2145GOP/s, since the peak throughput is based on the fused multiplication and add instructions. As a result, depending on the implementation for GF(2^8) multiplication, the device may become either compute bound or bandwidth bound.

C. Overcoming Memory Bandwidth Limit Using Texture Caches, Tiling, and Prefetching

As shown in Equation 3, the key operation in Reed-Solomon coding is the GF(2^8) multiplication. For the two implementations for GF(2^8) multiplication using lookup tables, each GF(2^8) multiplication requires one or more table accesses. Therefore, we first resort to using the texture cache to reduce the memory bandwidth consumption due to the table lookups. On our target GPU, there is a 48kB texture cache in each compute unit. While its size is sufficient for the two small tables used in List 2, the texture cache is not large enough to hold the large precomputation table used in List 3. Although we can leverage the symmetric property of the precomputed table (since A * B = B * A) to halve the table size, it introduces additional computations required to determine the right table indices. Instead, we make the observation that since each workitem computes 16 consecutive bytes in the same row as shown in Equation 3, the 16 products (i.e., V[idy][j]) * Src[j][idx*16 : idx*16+15]) share the same operand V[idy][j]. In other words,

when accessing the large GF(2^8) multiplication table, the indices used for these 16 products only vary in the lower 8 bits. Furthermore, all workitems in the same row (idy) also share the same operand V[idy][j]. Such strong spatial locality would enable a 256-byte cache work well. Therefore, although the texture cache cannot accommodate the entire large lookup table, it achieves very high hit rates and effectively reduces the bandwidth requirement for accessing off-chip memory.

Besides using the texture cache for precomputed tables, we use it to hold both the encoding matrix 'V' as well as the input matrix 'Src'. Due to its reuse, the accesses to 'V' are much less frequent in comparison to accessing the precomputed tables. Therefore, the performance impact is minor. We also use local memory to enable the tiling optimization for the 'Src' matrix so as to reduce the associated bandwidth utilization. The performance impact is negligible and sometimes there are marginal gains. This is because the bottleneck now becomes computation as the texture caches eliminate the bandwidth limit associated with table lookups.

D. Hiding Data Transmission Latency Over PCIe

We use the same partitioning scheme as described in Figure 2 to generate multiple streams. Once a stream finishes data transmission, the corresponding computation can start. This allows for the data transmission latency over the PCIe bus to be mostly hidden as long as the computation time is longer than the data transmission latency.

E. Shared Virual Memory to Eliminate Memory Copying

Shared virtual memory (SVM) is supported in OpenCL 2.0 and eliminates the need to explicitly copy data between the CPU system memory and the GPU device memory. Instead, an SVM buffer can be mapped to either the CPU or GPU address space and be accessed using pointers from either side. OpenCL 2.0 supports both fine-grained and coarse-grained SVM, which differ in the way how updates are visible to other devices. We use coarse-grained SVM in our work, as the output needs to be visible to the CPU only at the end of a kernel's execution. As SVM is supported in AMD APUs, we explore this feature and compare it with the multi-streaming solution to hide the data movement latency.

V. Reed-Solomon Coding on FPGAs

Compared to CPUs or GPUs, FPGAs have different characteristics. First, FPGAs feature abundant on-chip logic that can be (re)configured for computation. As a result, it is more suitable to computation bound algorithms. Second, FPGAs rely on pipelined parallelism to achieve high computational throughput. In Altera's OpenCL for FPGA framework, the kernel code is compiled into a pipeline. The thread-level (or data-level) parallelism embodied in workgroups is converted to independent workitems sequencing through the pipeline. Third, current FPGA-based accelerators have relatively low memory access bandwidth. As a result, it is important to efficiently and fully utilize the available memory bandwidth. Otherwise, the device may be quickly starving for data to be processed resulting in expensive stalls in the FPGA pipeline.

As analyzed in Section II, with relatively large 'srcs' or 'dests', Reed-Solomon coding is a compute bound problem, making it a good candidate for FPGAs. Taking advantage of the OpenCL compiler for FPGAs, we use the same baseline kernel code as the one used for GPUs, i.e., each workitem computes 1 element (1 byte) in the output 'Dest' matrix. Besides aggressive loop unrolling to deepen the pipelines for higher pipelined parallelism, we apply the following optimizations, mainly targeting at memory bandwidth, since the large amount of computing logic available on FPGAs satisfies the computational requirements.

A. Vectorization to Optimize FPGA Memory Bandwidth

Similar to GPU (or CPU) off-chip memory, high bandwidth is achieved for FPGAs if each access reads/writes a large number of bytes (e.g., 64 bytes). Therefore, vectorization remains to be a key optimization to improve the FPGA performance.

Different from GPUs, on FPGAs, the workitems in the same local workgroup will not form wavefronts. Instead, they will go through the pipeline one by one in a pipelined manner. As a result, the memory access is based on each individual workitem rather than a wavefront of workitems on GPUs. With our target FPGA requiring a granularity of 64 bytes to achieve high DRAM access bandwidth, each workitem needs to compute 64 elements along the same row in the output 'Dest' matrix. This way, when a workitem accesses the input 'Src' matrix, it can use the uint16 vector data type (i.e., 16 consecutive unsigned integers or 64 bytes). As discussed in Section IV, for Reed-Solomon coding, such vectorization also achieves the same effect of the thread merge/coarsening optimization and enables the data reuse of the encoding matrix 'V' by 64 times.

B. Exploration of Different Implementations for GF(2^8) Multiplications

Our target FPGA has a large amount of on-chip resource available for implementing computational logic with a peak theoretical DRAM bandwidth of 25.6 GB/s. As a result, the performance bottleneck tends to be in bandwidth. Based on our analysis in Section II, the Russian Peasant Algorithm for GF(2^8) multiplication does not involve any memory access and can be fully pipelined. Therefore, it is a suitable candidate for FPGAs. On the other hand, for GF(2^8) multiplication based on precomputed lookup tables, we can take advantage of the constant memory, a type of on-chip memory introduced for read-only data, to store these tables. This way, the constant tables will be synthesized into on-chip ROM resource such that no off-chip memory accesses are needed.

C. Overcoming Memory Bandwidth Limit Using Tiling and Prefetching

Since FPGAs largely meets the computational requirements, we need to be more aggressive in optimizing the memory bandwidth utilization as compared to the optimizations that were done for GPUs. In addition, FPGAs also have more on-chip memory resources than the local memory on GPUs. Therefore, we use the local memory structure for the entire encoding matrix 'V'. We then partition the output matrix into tiles, similar to Figure 2, and assign one local workgroup to each tile. With this tiling optimization, we prefetch a tile of input data into local memory, which is shared among all workitems in the local

workgroup. A large tile size results in high data reuse and reduces off-chip memory bandwidth requirements.

D. Kernel Replication to Fully Utilize FPGA Logic Resource

Since the computation kernel for the Reed-Solomon coding does not fully utilize the logic resource available on our FPGA chip, we manually replicate the kernel such that two pipelines are synthesized on the FPGA chip. We also try to use the 'num_compute_units' attribute. This attribute tells the Altera OpenCL compiler to create multiple pipelines in order to support execution of multiple workgroups in parallel. Due to insufficient resources and routing pressure, the Altera OpenCL offline compiler fails to synthesize the hardware for this particular option.

We further constrain each kernel such that it only accesses one DRAM bank so that we eliminate the contention between the two kernels for memory bandwidth.

VI. Experiment Methodology

In our experiments, the input file is a gzip archive with a size of 836.9MB. We vary the number of the blocks that the file will be divided into, i.e., the variable '*srcs*' in Figure 1, and choose to add a fixed number (3) of redundant blocks into the output. In other words, the variable '*dests*' in Figure 1, is equal to (*srcs* + 3). To compare the performance across different acceralator architectures, we use the encoding throughput, which is defined as: *(the original file size / latency)* in the unit of GB/s, as the performance metric. The data in the two dimensional arrays are laid out in the row-major order.

We evaluate our CPU implementations on an Intel(R) Xeon(R) CPU E5-2697 v3 server, which runs at 2.6GHz and has 2 Xeon CPUs with either having 14 cores. So there are 28 cores total on this server. With hyperthreading, the Linux OS reports that it has 56 processors.

We use an NVIDIA Tesla K40m GPU with the CUDA 7.0 driver for our experiments on the GPU platform. It uses a 16-lane PCI-Express Gen 3 to interface with the host CPU. For the APU platform, we use an AMD Carrizo APU system with OpenCL C 2.0.

For FPGAs, we perform our experiments on a Nallatech PCIe-385N FPGA accelerator card, which features an Altera Stratix V A7 FPGA chip and two DDR3 DRAM banks with a theoretical peak bandiwidth of 25.6 GB/s. It uses one 8-lane PCI-Express Gen 3 bus to communicate with the host. The Altera Offline Compiler (aocl) V15.0 is used to compile the OpenCL kernels into the bit-streams for FPGAs.

VII. Experimental Results

In this section, we first present the performance impact of the code optimizations that we discuss earlier for each accelerator architecture. We then compare the performance across these accelerators.

A. Performance Optimizations for CPUs

As discussed in Section III, the baseline CPU implementation is from the Intel ISA-L library, which leverages the SSE instructions for fast GF(2^8) multiplications. We extend it for multithreading and explore the number of threads to improve the performance. The results based on the configuration 'srcs =30; dests = 33' are shown in Figure 3.

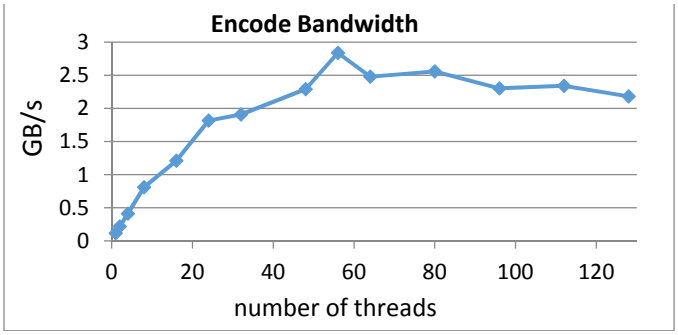

Figure 3. The CPU encoding bandwidth for different numbers of threads on a multi-core Xeon server.

From Figure 3, we can see that multithreading highly improves the throughput of Reed-Solomon encoding on CPUs. With a single thread, the ISA-L version has a throughput of 0.12 GB/s. More threads enable more concurrent processing and as one would expect, the best throughput of 2.83GB/s is achieved when all 56 threads are used.

B. Performance Optimizations for GPUs

For GPUs, we explore different implementations for GF(2^8) multiplications and propose various optimizations as discussed in Section IV. The performance impacts are shown in Figure 4. These results are also based on the configuration 'srcs =30; dests = 33'.

As shown in Figure 4, for each of the three implementations for GF(2^8) field discussed in Section II, we start with a workitem computing one byte (i.e.,char-type) of data in the output matrix, labeled as 'char w/o tx cache' as in the figure. Then, we use the texture cache to store the precomputation tables for the GF(2^8) multiplication using small and large tables, or the encoding matrix and input matrix for the Russian Peasant algorithm for GF(2^8) multiplications, labeled as 'char' in the figure. We further vectorize the data access to achieve the impact of thread coarsening such that each thread computes 4-byte (i.e., one int type, labeled 'int') and 16-byte of data (i.e., one int4-type data, labeled 'int4). As shown in the figure, vectorization consistently improves the performance. In comparison, the texture cache has significant impact on the large table implementation while relatively smaller impact on the small table implementation and the Russian Peasant implementation. The texture cache essentially eliminates the off-chip DRAM bandwidth consumed for accessing the precomputation tables. According to the CUDA profiler, the effective bandwidth of the texture cache in our large table implementation is 1.09TB/s (= the amount of data loaded from the texture cache/overall kernel execution latency).

Among the three implementations for GF(2^8) multiplications, the large table implementation as shown in List 3 achieves the highest performance for the GPU. The reason is that the GPU performance is actually bounded by the number of computations required for Reed-Solomon coding. The large table implementation trades off the large precomputation table for the minimal number of computations required for GF(2^8) multiplications, as discussed in Section II. The Russian Peasant algorithm, on the other hand, has the lowest performance as it incurs the highest number of computations.

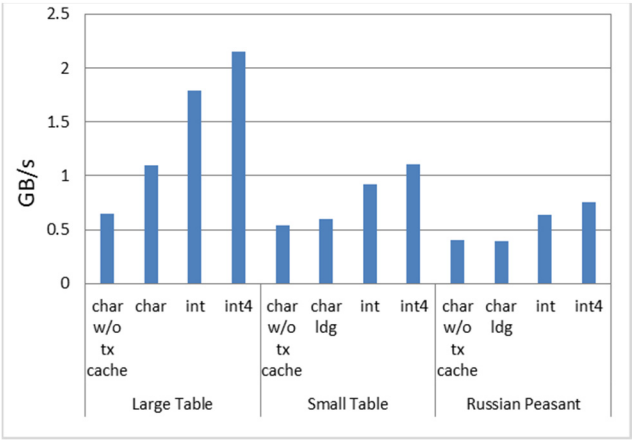

Figure 4. The performance impact of GPU code optimizations. (srcs = 30 and dests = 33)

We perform additional experiments to confirm the GPU performance bottleneck. Based on our best performing code, i.e., large table with int4 type of vector data access, the profiler reports that the majority GPU pipeline stalls are due to execution dependency, as shown in Figure 5. A detailed inspection of the compiler generated assembly code shows that besides the logic computations, table lookups also incur arithmetic instructions to generate array indices. Further bandwidth optimizations, including tiling and putting the encode matrix as well as the input matrix into the texture cache, do not benefit performance. We then modify the code to constrain the accesses of the input 'Src' matrix to one cache line. In this case, the code does not generate the correct outputs but uses the same number of computations while eliminating the bandwidth requirement associated with the input matrix. The performance improvements compared to the large-table int4 version are small. However, if we modify the code to eliminate some of the logic operations (e.g., the XOR operations in Equation 1), the performance improves significantly, confirming that the current performance bottleneck is the computation rather than the bandwidth.

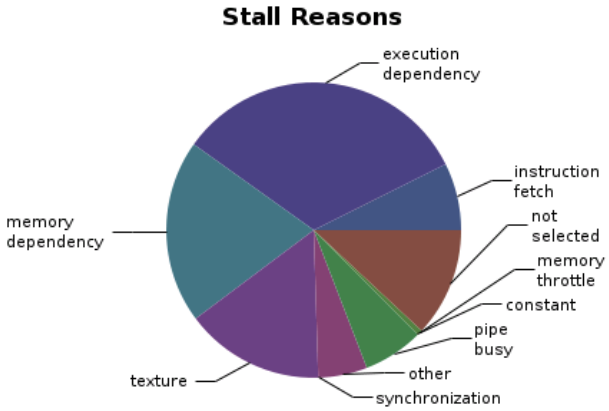

Figure 5. Analysis of GPU pipeline stalls.

The above mentioned encode bandwidth results include the latency for moving the data between the CPU and GPU through the PCIe bus. As discussed in Section IV, multi-streaming can be used to hide such data transmission latency as the GPU kernel computation time is longer than the transmission latency. In our experiments, we use different number of streams and find that using 8 streams can hide most of the transmission latency as shown in Figure 6. With 8 streams, the overall throughput of the large-table int4 version increases from 2.15 GB/s to 3.90 GB/s.

Figure 6. Multi-streaming (8 streams) to hide CPU-GPU data transmission latency through the PCIe bus. (light blue color: kernel execution; dark brown: data transmission latency).

Shared virtual memory (SVM) in APUs eliminates the needs for data transmission between CPU and GPU memory over the PCIe bus. We compare SVM with the streaming approach to hide the data transmission latency. The results are shown in Figure 7. Like in the case of GPUs, the APU in our work also favors the large table implementation for GF(2^8) multiplications. Therefore, we focus on this implementation in Figure 7. As shown in the figure, vectorization is also effective in improving the performance. However, the texture cache is not as effective in improving the performance and we choose not to use it for the APU.

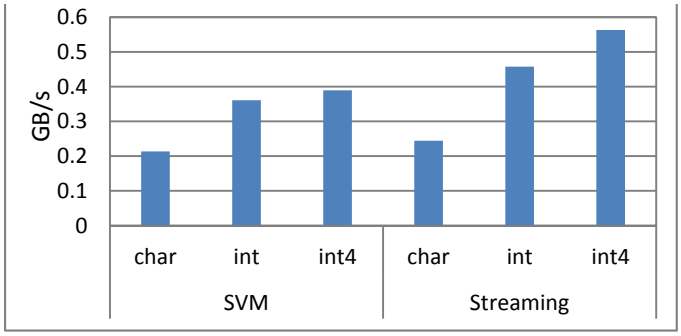

Figure 7. Performance comparison between SVM and multi-streaming on APUs. (srcs = 30 and dests = 33)

Comparing SVM and multi-streaming, one can see from Figure 7 that although SVM eliminates the code to explicitly move data between CPU and GPU memory, the performance is not as competitive as the multi-streaming solution. The reason is mainly due to the overhead associated with the blocking function calls to map and un-map the SVM buffers.

C. Performance Optimizations for FPGAs

We evaluate the performance optimizations that we proposed for Reed-Solomon coding for FPGAs in Section V. The FPGA card under evaluation uses an 8-lane PCI-Express Gen 2 bus. In addition, the current FPGA DMA engine does not support simultaneous bi-directional communication and can only achieve DMA throughput 2.35 GB/s write and 3.05 GB/s of read bandwidth. As a result, there is a high latency overhead associated with the data movement between the CPU system memory and FPGA device memory through this PCIe bus. With a PCIe interface similar to the one available with the GPU card, we expect such overhead will be largely reduced. Given that FPGAs use soft-logic for the DMA engine, we can reasonably assume the DMA engine performance can be easily increased. Therefore, we focus on the kernel execution latency for the

FPGA platform and the results are shown in Figure 8. Also, as the encoding bandwidth results achieved from different optimizations vary significantly, we use the logarithmic scale to highlight the impact of each optimization step. From the figure, we can make the following observations. First, the vector-type data accesses are critical to utilize the DRAM bandwidth. As highlighted in Section V, FPGAs mainly leverage pipelined parallelism. Therefore, the memory accesses in each workitem need to be 64-byte wide. By increasing the granularity from 1 byte (i.e., the char data type) to 64 bytes (i.e., the int16 data type), the throughput is improved by around 50 times.

Second, optimizations that help with further reducing the bandwidth use are also helpful. The tiling optimization on the input matrix, 'Src', improves the performance by 33% on average.

Third, among the three implementations for GF(2^8) multiplications used for Reed-Solomon coding, the Russian Peasant algorithm only involves computations. In comparison, for the other two implementations using precomputation tables, we use constant memory to eliminate communication to the off-chip DRAM. As a result, a ROM block is synthesized for each access to the precomputation table. This incurs high cost on hardware resources, which limits the degrees of loop unrolling.

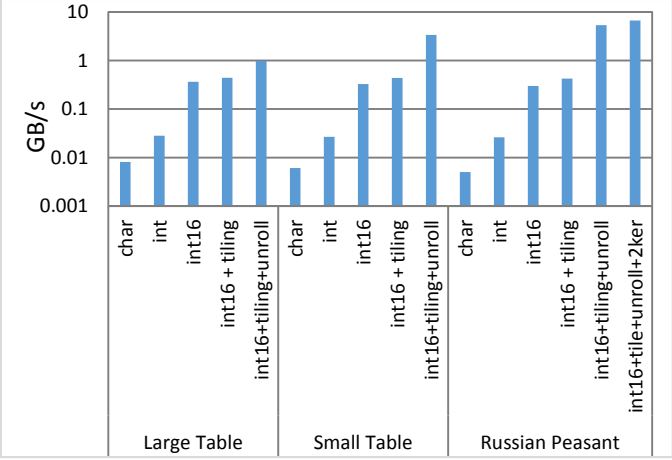

Figure 8. Performance impact of OpenCL code optimizations for FPGAs. (srcs = 30 and dests = 33)

Fourth, unrolling the loop which computes the dot-product as shown in Equation 3, results in deep pipelines, which increases the number of workitems that can be sequenced through them concurrently. As discussed above, the code for the Russian Peasants algorithm can be unrolled very aggressively while the code using precomputation tables can only be unrolled a few times due to resource utilization. With 'srcs = 30', we were able to fully unroll (i.e., 30 times) the dot-product loop for the Russian Peasant algorithm. In comparison, we could only unroll the loop 11 times and 5 times for the implementations using the small and large pre-computed tables, respectively. Since our bandwidth optimizations already eliminate the performance bottlenecks associated with data movement, deeper pipelines show higher performance. The Russian Peasant algorithm achieves a throughput of 5.35 GB/s. The one using the small and large precomputation tables has a throughput of 3.38GB/s and 0.98 GB/s, respectively.

Fifth, as Reed-Solomon coding based on the Russian Peasant algorithm does not use much on-chip resource, 45% logic utilization on the Stratix V FPGA chip, we manually add an additional kernel and partition the data as described in Section V. The logic utilization becomes 68%. The throughput is increased to 6.69 GB/s, labeled as 'int16+tile+unroll+2ker' in Figure 8.

D. Performance Comparison amongst Different Accelerator Architectures

In this experiment, we vary the number of blocks, i.e., the parameter 'srcs', and evaluate the throughput achieved on different accelerator architectures. The results are shown in Figure **9**. For all architectures except the FPGA platform we use the end-to-end latency, which includes the data transmission latency if the data need to be transmitted between the host and device. For FPGAs, the kernel latencies are used instead for the same reason as discussed before. Although this may seem to favor the FPGA platform, the data transmission latencies for GPUs and APUs are largely hidden using the multi-streaming approach. Multi-streaming does not hide the transmission latency for our FPGA platform, as the latency of our optimized kernel is much smaller than the current PCIe transmission latency. With an improved PCIe interface, we expect such latency can be hidden similar to our GPU platform. Furthermore, in architectures where the FPGA is integrated with the CPU they share the system memory, thereby eliminating the need for explicit data transfers. Therefore, we think using the kernel computation latency for FPGAs is a reasonable choice.

From Figure **9**, we can see that with higher 'srcs', Reed-Solomon coding requires more computations and generates more parity data, as discussed in Section II. As a result, the CPU, the GPU, and the APU, given their fixed amount of computational logic and bandwidth, have lower throughput for a larger value of 'srcs'. The interesting exception is the FPGAs. As we fully unroll the dot-product loop of Equation 3, more computational logic units are synthesized for a larger value of 'srcs'. Therefore, when 'srcs' increases from 15 to 25, the overall throughput actually increases. Such a trend is not consistent for all value of 'srcs' since the memory bandwidth stays fixed on the FPGA platform as opposed to scaling with the computational units.

Comparing the different accelerator architectures, Figure **9** shows that the K40m GPU achieves the highest throughput when value of 'srcs' is less than 15, with FPGA having performance close to that of the GPU. For larger values of 'srcs', the computational bound impact affects the throughput of the GPU. The FPGA platform, in comparison, has the superior performance.

Considering the overall cost of ownership of each platform, we estimate that the cost of the Xeon CPU and the K40m GPU having a similar range of thousands of dollars while the FPGA and the APU have a cost ranging in hundreds of dollars. As a result, we conclude that for Reed-Solomon coding in distributed storage systems, the FPGA platform is most promising but needs to improve its current PCIe DMA interface. Though the APU is another low-cost alternative, its performance is actually lower than the Xeon CPU server or the high-end discrete GPUs.

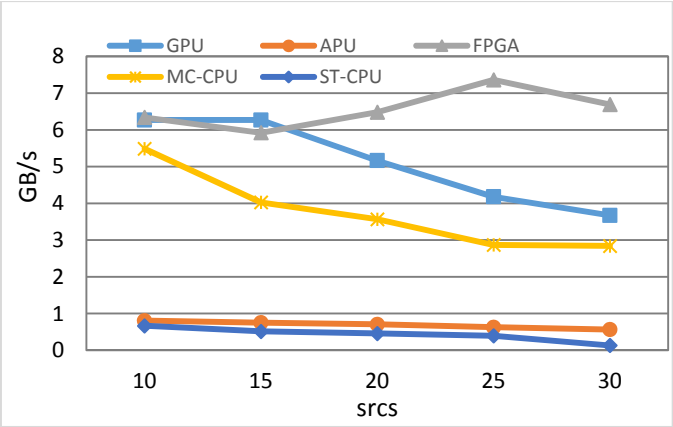

Figure 9. Performance comparison among different accelerator architecture for the input file being divided into different numbers of blocks (srcs).

VIII. Related Work

On the CPU platform, the research on high speed Reed Solomon coding focuses on leveraging the SIMD instructions [1][7], as discussed in Section III. A multi-threaded scheme, similar to the one we describe in Section III, has been proposed for multi-core based network coding [11]. In comparison to this work [11], we show that multithreading remains highly effective in improving the throughput even its single-thread implementation is highly optimized with the SSE instructions.

In order to utilize the massive computational capability and high memory bandwidth of GPUs, prior works [4][10] leverage the fine-grain data-level parallelism and propose code optimizations to improve the performance. In comparison, our proposed GPU optimizations are based on state-of-art GPUs, whose architecture characteristics demand different optimization strategies from those described in the prior works. In particular, Shojania et al. [10] proposes using an implementation based on the two small precomputation tables for GF(2^8) multiplications. In contrast, as highlighted in Section IV and Section VII, the working set of the implementation using the large precomputation table is actually small. Since the large table implementation reduces the computational instructions, it achieves the best performance for both GPUs and APUs. Besides this, our optimizations also differ in the aggressiveness in vectorization and texture cache utilization from prior works on GPUs. The small table implementation for GF(2^8) multiplications is also proposed for APUs [13]. In comparison to this work, we also study the impact of using SVM to eliminate the explicit data movement between CPU and GPU memory.

FPGA implementations for Reed-Solomon coding [2][3] is mainly done through direct hardware design captured in hardware design languages such as Verilog or VHDL. To the best of our knowledge, our work is the first on optimizing the OpenCL code for FPGA implementation of erasure coding.

IX. Conclusions

In this work, we explore different computing devices, including multi-core CPUs, GPUs, APUs, and FPGAs, for high performance erasure coding. Based on the characteristics of each architecture, we propose different code optimization strategies and explore various implementations for GF(2^8) multiplications. Our achieved performance on these platforms well exceeds the highly tuned Intel ISA-L library code running on a single core of Xeon processors. Our results show that the FPGA platform is the most promising device for erasure coding in storage systems. Furthermore, although the support for OpenCL to FPGA enables similar programmability to processors, careful code optimizations at the OpenCL level is critical to achieve high performance.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments to improve the paper.

References

[1] H. P. Anvin. The mathematics of RAID- 6. http://kernel.org/pub/linux/kernel/ people/hpa/raid6.pdf, 2011

[2] A. Azad and M. Shahed, A Compact and Fast FPGA based Implementation of Encoding and Decoing Algorithm Using Reed Solomon Codes, Int'l Journal of Future Compuiter and Communication, Feb. 2014.

[3] V. Babrekar and S. Sakhare, Review of FPGA Implementation of Reed-Solomon Encoder –Decoder, Int'l Journal of Computer Applications, 2014.

[4] S. Kalcher and V. Lindenstruth. Accelerating Galois Field arithmetic for Reed-Solomon erasure codes in storage applications. In IEEE International Conference on Cluster Computing, 2011.

[5] D. B. Kirk and W. W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach, 2010

[6] J. S. Plank, Erasure Code for Storage Systems: A Breif Primer, Login: The USENIX Magzine, www.usenix.org, December 2013, Vol. 38, No. 6, pp. 44-50.

[7] J. S. Plank, K. M. Greenan, and E. L. Miller, Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions, FAST'13, 2013.

[8] J. S. Plank, K. M. Greenan, E. L. Miller, and W. B. Houston. GF-Complete: A comprehensive open source library for Galois Field arithmetic. Technical Report UT-CS-13-703, University of Tennessee, January 2013.

[9] T. Scodiero, GPU Memory Bootcamp II: beyond best practices, GPU Technology Conference, 2015.

[10] H. Shojania and B. Li, Pushing the Envelope: Extreme Network Coding on the GPU, ICDCS'09, 2009.

[11] H. Shojania and B. Li, Parallelized Network Coding with Hardware Acceleration, IWQoS, 2007.

[12] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. Commun. ACM, 52(4):65–76, 2009.

[13] R. Wyrzykowski, M. Wozniak, and L. Kuczynski, Efficient Execution of Erasur Codes on AMD APU Architecture, Parallel processing and applied mathematics, LNCS, 2014.

[14] Y. Yang, P. Xiang, J. Kong, M. Mantor and H. Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. PLDI, 2010

[15] Intel Sorage Acceleration Library (open source version). https://01.org/intel%C2%AE-storage-acceleration-library-open-source-version.

[16] OpenCL 2.0 Shared Virtual Memory. http://developer.amd.com/community/blog/2014/10/24/opencl-2-shared-virtual-memory/

[17] Altera SDK for OpenCL. https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html