PErasure: a Parallel Cauchy Reed-Solomon Coding Library for GPUs

Xiaowen Chu, Chengjian Liu, Kai Ouyang, Ling Sing Yung, Hai Liu, and Yiu-Wing Leung
Department of Computer Science, Hong Kong Baptist University
Kowloon Tong, Hong Kong
Email: {chxw, cscjliu, kouyang, timyung, hliu, ywleung}@comp.hkbu.edu.hk

***Abstract*—In recent years, erasure coding has been adopted by large-scale cloud storage systems to replace data replication. With the increase of disk I/O throughput and network bandwidth, the speed of erasure coding becomes one of the key system bottlenecks. In this paper, we propose to offload the task of erasure coding to Graphics Processing Units (GPUs). Specifically, we have designed and implemented PErasure, a parallel Cauchy Reed-Solomon (CRS) coding library. We compare the performance of PErasure with that of two state-of-the-art libraries: Jerasure (for CPUs) and Gibraltar (for GPUs). Our experiments show that the raw coding speed of PErasure on a $500 Nvidia GTX780 card is about 10 times faster than that of multithreaded Jerasure on a quad-core modern CPU, and 2-4 times faster than Gibraltar on the same GPU. PErasure can achieve up to 10GB/s of overall encoding speed using just a single GPU for a large storage system that can withstand up to 8 disk failures.**

I. Introduction

Replication is currently the most popular way to achieve high data reliability in large-scale cloud storage systems, such as GFS [1], Hadoop [2], and Azure [3]. To reduce the high cost of data replication and further improve the data reliability, erasure coding has attracted attention from both academia and industry in recent years [4]–[8]. Due to the limitation and high cost of hardware RAID controllers, software based erasure coding becomes more attractive and flexible for such storage systems [9]. However, recent technology trends bring new challenges for erasure coding. Firstly, more and more storage systems start to use Solid-State Drives or even huge volume of RAM to improve the I/O performance [10], [11]. Secondly, with the explosive increase of data volume, more redundant disk drives are required to guarantee high data reliability which creates an increasing demand of higher erasure coding speed. Thirdly, with the popularity of 40G/100G Ethernet and InfiniBand FDR/QDR in data centers, network bandwidth will not be the system bottleneck. Therefore, how to improve the speed of erasure coding becomes an important research issue.

With the proliferation of multi-core CPUs and many-core GPUs, parallelization becomes a major approach to improving the speed of erasure coding. For multi-core CPUs, Cauchy Reed-Solomon (CRS) codes [12] have been parallelized in [13], [14]; EVENODD codes have been parallelized in [15]; and RDP codes have been parallelized in [16]. A fast Galois field arithmetic library for multi-core CPUs has recently been presented in [17]. Recently, GPUs have been used in some storage systems to perform different computationally expensive tasks. Shredder is a framework to leverage GPUs to efficiently chunk files [18]. GPUstore [19] is another framework to integrate GPUs into storage systems for file-level or block-level encryption and RAID6 data recovery. Another GPU-based RAID6 system has been developed in [20], which uses GPUs to accelerate two RAID6 coding schemes, namely Blaum-Roth and Liberation codes, and achieves up to 3GB/s of coding speed. Gibraltar is so far the most successful GPU-based Reed-Solomon (RS) coding library [21] that uses table lookup operations to implement Galois Field multiplications and achieves 10-fold speedup over single-thread Jerasure [22], the most well-known erasure coding library.

In this paper, we propose to use GPUs to perform CRS coding [12], which is the state-of-the-art Maximum Distance Separable (MDS) code. CRS coding uses XOR operations only and is more efficient than RS coding. This paper makes the following contributions:

(1) We designed and implemented a new CRS coding library for GPUs named PErasure, which tailors the parallel coding algorithms for GPU architecture.
(2) To overcome the bottleneck of PCI Express (PCIe) bus, we designed a pipeline mechanism for PErasure which can maximize the overlap between memory copy operations and coding operations.
(3) We showed through real experiments that PErasure outperforms two state-of-the-art libraries: Jerasure and Gibraltar.

The remainder of this paper is organized as follows. Section II introduces the background of CRS and GPUs. Section III presents the design of PErasure. Experimental results are presented in Section IV. Section V concludes the paper.

II. Background

A. Cauchy Reed-Solomon Coding

A general erasure coding storage system consists of k data devices and m coding devices. When using an MDS code, such storage system can withstand the failure of any m devices. Although there exist many MDS codes for $m = 2$, RS codes are the only known general MDS codes that support any values of k and m. To generate a single coding word, RS code requires k Galois Field multiplications and $k-1$ XORs, which are computationally expensive. CRS coding is a variant of RS coding that uses XOR operations only and results in better coding performance [12].

CRS coding first selects a value w that is no less than $log_2(k+m)$, and then defines an $m \times k$ Cauchy distribution matrix over Galois Field $GF(2^w)$. Next, the Cauchy distribution matrix is expanded into a $w(k+m) \times wk$ binary distribution

matrix (BDM). Each of the k data devices is partitioned into w rows with the same size. So the k data devices form a matrix with wk rows. The CRS encoding process is similar to matrix multiplication, as illustrated in Figure 1, in which I denotes a $w \times w$ identity matrix, D represents a data device formatted into w rows, and C represents a coding device formatted into w rows. Upon the failure of any $p \leq m$ devices, CRS decoding can recover all failed devices by multiplying the corresponding decoding bit-matrix with the surviving devices. In practice, the decoding bit-matrices can be precalculated offline.

$$\overbrace{\begin{bmatrix} I & 0 & \cdots & 0 \\ 0 & I & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & I \\ X_{0,0} & X_{0,1} & \cdots & X_{0,k-1} \\ X_{1,0} & X_{1,1} & \cdots & X_{1,k-1} \\ \vdots & \vdots & \ddots & \vdots \\ X_{m-1,0} & X_{m-1,1} & \cdots & X_{m-1,k-1} \end{bmatrix}}^{wk} * \begin{bmatrix} D_0 \\ D_1 \\ \vdots \\ D_{k-1} \end{bmatrix} = \begin{bmatrix} D_0 \\ D_1 \\ \vdots \\ D_{k-1} \\ C_0 \\ C_1 \\ \vdots \\ C_{m-1} \end{bmatrix}$$

(rows of BDM: wk identity rows, wm coding rows)

BDM Data Data+Coding

Fig. 1: Illustration of Cauchy Reed-Solomon Coding

B. GPU Computing

GPUs are highly parallel many-core processors. A GPU consists of a number of Streaming Multiprocessors (SMs), and each SM has many Scalar Processors (SPs). The design of the SMs is based on the Single-Instruction Multiple-Data (SIMD) architecture, i.e., at any given clock cycle, all SPs in the same SM must execute the same instruction, but can operate on different data. GPUs have a complex memory system which includes global memory, constant memory, texture memory, register memory, and shared memory [23]. Register and shared memory are very fast but limited in size; both texture and constant memory support cache mechanism; global memory is large and has very high bandwidth, but its access latency is very long.

Currently, CUDA is the most popular programming model for GPUs [24], under which the GPU is regarded as a coprocessor capable of executing a large number of parallel threads. A typical CUDA program consists of the host functions to be executed on the CPU and the kernel functions to be executed on the GPU. Each kernel function is running as a grid of threads, which are organized into many thread blocks. Each thread block can include a set of threads, which can share data through shared memory and perform barrier synchronization. GPUs have found many successful coding-related applications, such as network coding [25]–[27] and data encryption [28], [29].

III. Parallel CRS Coding on GPUs

For any given k and m, PErasure can be used to generate m coding devices from k data devices, and recover up to m failed devices. If the total size of $k+m$ devices cannot fit into GPU memory, each device can be split into smaller chunks. PErasure then process the data chunk by chunk. In each round, it copies k data chunks to GPU, each from a different data device, then encodes them into m coding chunks by launching a GPU kernel function for m times, and finally transfers the m coding chunks back to main memory. The high level workflow of PErasure encoding is shown in Algorithm 1. The workflow of decoding is basically the same as encoding, yet there are two differences. Firstly, CRS decoding uses decoding bit-matrices instead of BDMs. To save time, the set of decoding bit-matrices can be precalculated. Secondly, decoding needs to determine the source of surviving devices and hence introduces more branches in the kernel function.

Algorithm 1: High Level Workflow of PErasure

Input: $k, m, w, bitmatrix, dataSize$
Compute $round$ from $k, m, w, dataSize$
Copy $bitmatrix$ to GPU's texture or constant memory
Allocate GPU memory
for $i \leftarrow 1$ **to** $round$ **do**
  Copy k data chunks of the i-th round to GPU
  for $j \leftarrow 1$ **to** m **do**
    Launch the kernel function to generate a coding chunk
  end
  Copy back m coding chunks of the i-th round to main memory
end
Free memory resources

PErasure supports two different versions of kernel function, namely GMPE (Global Memory PErasure) and SMPE (Shared Memory PErasure). GMPE uses global memory and can support very large values of k and m. SMPE exploits shared memory and constant memory to improve the coding performance, but has a limitation on the value of km due to the limited size of constant memory. We describe these two kernel functions in the following two subsections.

A. GMPE: Global Memory PErasure

The GMPE kernel function specifies the behaviour of GPU threads, each of which generates w coding words from k data words. A word is the basic unit of XOR instruction on GPU, which is a *long*-type integer in CUDA. When PErasure launches a GMPE kernel function, a grid of GPU threads are created to collectively generate a complete coding chunk. The pseudo code of GMPE kernel is given in Algorithm 2.

A coding word can be calculated as the dot product of a bit vector from BDM and a word vector from k data chunks. Since BDM is small in size and invariant, we store BDM in GPU texture memory to reduce the latency of accessing BDM by exploiting the caching effect. Within each thread, **idx**s are first computed to locate related data words and coding words. A boundary check follows immediately after basic index calculations. GMPE then prepares the data address involved and computes the output address in the coding chunk. The calculation of dot products for w coding words is completed by three levels of *for*-loop. The middle and inner-most two loops together compute a coding word from k data words. In each repetition, GMPE prepares a base index for performing fast index shifting within inner loops, which can significantly reduce the clock cycles of index calculation.

B. SMPE: Shared Memory PErasure

CRS coding is a typical memory-intensive application with a memory-to-compute ratio greater than 2: generating a single coding word requires $2kw$ memory accesses and no more than kw XOR operations. Hence PErasure's coding performance is bounded by the bandwidth of GPU global memory. SMPE is

Algorithm 2: GMPE: Global Memory PErasure

Input: $k, w, bitmatrix, destId, dataPtrs, codingPtrs$
$dataSize, numOfLong$
Compute idx from $blockIndex$,$blockDim$,$threadIndex$
Compute $stripShift$ from idx,$numOfLong$,w
if $stripShift > dataSize$ **then**
  Return
end
$longIndex = idx\%numOfLong$
Shift $outPtr$ to the target coding pointer
for $i \leftarrow 0$ **to** $w-1$ **do**
  Shift $outPtr$ by i packets
  Compute $index$ from $destId$ and i
  $temp \leftarrow 0$
  for $dataIndex \leftarrow 0$ **to** $k-1$ **do**
    Shift $srcPtr$ to the target data pointer
    for $j \leftarrow 0$ **to** $w-1$ **do**
      if $bitmatrix[index]$ **then**
        Shift $srcPtr$ by j packets
        $temp = temp \oplus srcPtr[longIndex]$
      end
      $index++$
    end
  end
  $outPtr[longIndex] = temp$
end

Algorithm 3: SMPE: Shared Memory PErasure

Input: $k, w, bitmatrix, destId, dataPtrs, codingPtrs$
$dataSize, numOfLong$
Compute idx from $blockIndex$,$blockDim$,$threadIndex$
if $idx * sizeof(long) > dataSize$ **then**
  Return
end
$temp \leftarrow 0$
$rowIndex = threadIndex/numOfLong$
$colIndex = threadIndex\%numOfLong$
Shift $outPtr$ to the target coding pointer
for $dataIndex \leftarrow 0$ **to** $k-1$ **do**
  Threads collectively load data to $sharedData[]$
  Synchronize threads
  for $j \leftarrow 0$ **to** $w-1$ **do**
    Compute $sdIndex$ from $dataIndex$, j, $colIndex$
    $temp = temp \oplus$
    $(bitmatrix[index] \wedge sharedData[sdIndex])$
    $index++$
  end
  Synchronize threads
end
$outPtr[longIndex] = temp$

designed to reduce the frequency of global memory access by using the fast shared memory as buffers. Due to the limited size of shared memory, each SMPE thread is designed to encode a single coding word, such that a block of threads can load a vector of data words into shared memory collectively and reuse them during the coding operations. The pseudo code of SMPE kernel function is given in Algorithm 3. Major variables like indices are prepared for input data addressing, output coding addressing and fast index shifting. A coding word is generated by two levels of *for*-loop. The outer loop cooperates to load required k data words to shared memory. The shared data words can be reused (i.e., without accessing the global memory) in the inner loop during the coding process. Furthermore, we reduce the *if*-branch penalties by using AND operations in the inner loop. To match the fast speed of shared memory, SMPE chooses to store BDM on constant memory for the best performance, which brings a side effect that km cannot be larger than a threshold (e.g., 1024 on Nvidia GTX780) due to limited constant memory size.

C. Pipelined PErasure

Both GMPE and SMPE need to transfer data between CPU and GPU through PCIe bus. The bandwidth of PCIe bus could be a limiting factor to the overall performance of PErasure. To minimize the impact of data transfer, we design a pipeline mechanism for PErasure such that kernel executions can be overlapped with data transfers between CPU and GPU. The key idea is to launch many GPU kernels simultaneously. We use asynchronous memory copy such that the data transfer operation of one kernel can overlap with another kernel execution. For high-end GPUs such as Nvidia Quadro and Tesla, our pipelined design can achieve even better performance due to the bi-directional data communications between CPU and GPU. An illustration of the pipeline scheme is shown in Figure 2, and the pseudo code of pipelined PErasure is given in Algorithm 4.

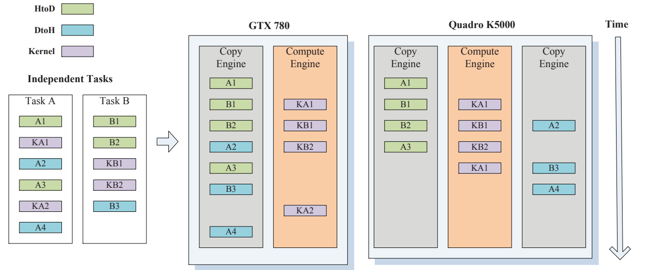


Fig. 2: Pipelined PErasure on Nvidia GTX 780 (sinlge copy engine) and Quadro K5000 (dual copy engines)

Algorithm 4: Pipelined PErasure

Input: $k, m, w, bitmatrix, numStream$
$data, coding, dataSize$
Compute $round$ from $k, m, w, dataSize$
Copy $bitmatrix$ to device's texture or constant memory
Initialize $cudaStreams$
Allocate GPU memory
for $i \leftarrow 0$ **to** $(round-1)/numStream$ **do**
  for $s \leftarrow 0$ **to** $numStream-1$ **do**
    for $j \leftarrow 1$ **to** k **do**
      Copy the j-th data chunk of stream s to GPU asynchronously
    end
  end
  for $s \leftarrow 0$ **to** $numStream-1$ **do**
    for $j \leftarrow 1$ **to** m **do**
      Launch the GMPE/SMPE kernel of stream s asynchronously
    end
  end
  for $s \leftarrow 0$ **to** $numStream-1$ **do**
    for $j \leftarrow 1$ **to** m **do**
      Copy back the j-th coding chunk of stream s to main memory asynchronously
    end
  end
end
Synchronize and wait for the last stream to complete
Free memory resources

D. Discussion: GMPE vs. SMPE

GMPE is designed as a general CRS coding function that can support any values of k and m. It is expected to work

TABLE I: Hardware Configuration of Coding Experiments

Motherboard	Gigabyte Z77-D3H
CPU	Intel i7-3770 (3.4 GHz)
Ram	Kingston DDR3 (8GB /1600 MHz)
Hard disk	Hitachi HDS721050CLA362
Graphic Card	GeForce GTX 780

well for future generations of GPUs because it uses the most primitive features of GPUs. On the contrary, SMPE is designed to optimize the coding performance by exploiting current GPU hardware features and trading off program simplicity and flexibility. Although SMPE has a constraint on the value of km, it can be used for a broad range of CRS codes in practice.

IV. Performance Evaluation

We implement PErasure using C++ and CUDA. The generation of BDMs and decoding bit-matrices is done by Jerasure and hence not included into PErasure. In this section, we conduct experiments to evaluate the coding performance of PErasure, and make a comparison with Gibraltar, CRS in Jerasure, and multi-thread CRS in Jerasure. The hardware configuration of our experiments is summarized in Table I. The coding performance is evaluated using two metrics: *Raw Speed* and *Overall Speed*. The raw speed measures the potential coding capability of GPUs; the overall speed measures how PErasure performs in real systems by taking into account the data transfer overhead. We run experiments on a broad range of (k, m) pairs. Due to limited space, we only present the results for $m = 2, 4, 8$ and k ranging from m to 45. All the results are the average of 5 runs. When evaluating the decoding performance, we consider the worst case by randomly selecting m failed devices out of the $k + m$ devices.

A. Raw Coding Performance

Figures 3 and 4 show the raw encoding and decoding speeds of PErasure (GMPE and SMPE), Gibraltar, single-thread Jerasure, and multi-thread Jerasure, respectively. In all testing cases, PErasure outperforms Gibraltar and Jerasure significantly in both encoding and decoding. The zigzag shape of PErasure and Jerasure curves is due to the increase of w in order to satisfy the condition of $w \geq log_2(k + m)$. The performance of CRS coding is inversely proportional to the values of w and m. Nevertheless, SMPE can achieve an impressive 12.5-15GB/s of raw encoding speed even for the case of $m = 8$, which is about 1.5 times of GMPE, 2.5-3 times of Gibraltar, and 10-12 times of multi-thread Jerasure. The decoding performance of PErasure shows more randomness and is slightly lower than the encoding performance. This is because the decoding procedure involves some extra work in identifying the survival devices.

B. Overall Coding Performance

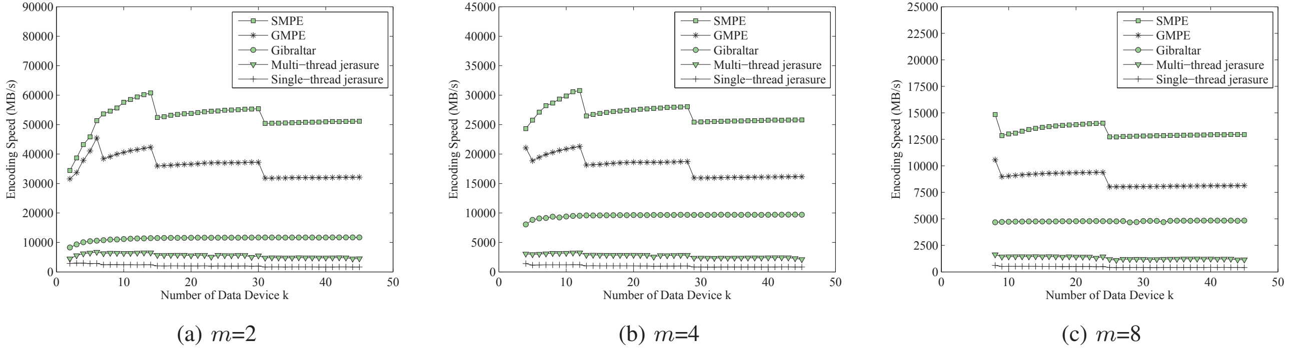
Different from Jerasure, PErasure and Gibraltar must transfer data between CPU and GPU which introduces considerable overhead. Figures 5 and 6 show the overall coding and decoding speeds of PErasure and Gibraltar, respectively. The bandwidth limit of PCIe bus (measured as 11.6GB/s on our testing platform) is also shown in the figures as a reference. Compared with the raw coding speeds, it is obvious that PCIe bandwidth is a critical limiting factor to the overall performance of PErasure. Since Nvidia GTX 780 only supports a single direction of memory copy, the theoretical upper bound of PErasure's overall coding speed is $k/(k + m)$ of the PCIe bandwidth. This explains why the coding performance of PErasure rises up with the increase of k. SMPE achieves the theoretical upper bound for all test cases. GMPE achieves the theoretical upper bound for all cases of $m = 2, 4$ and all $k \leq 24$ when $m = 8$. On more expensive high-end GPUs that support bi-directional PCIe communications, PErasure is able to achieve the full bandwidth of PCIe bus, as verified by our experiments on GTX Quadro K5000. We also notice that the overall performance of Gibraltar is slightly better than that of PErasure when k is small and $m = 2, 4$. This is because Gibraltar uses zero-copy approach for data transfer between CPU and GPU, which doesn't work well for CRS. The benefit of zero-copy will drop with the increase of k. On the contrary, PErasure performs better for larger k. For large values of m, PErasure always outperforms Gibraltar significantly.

C. Experiments on a Storage Cluster

Besides showing the overall coding performance, we also evaluate how our PErasure performs in a real storage cluster by considering both disk I/O and network I/O. The cluster is consisted of a server node and seven client nodes, connected by a 10-Gbps Ethernet switch. The server node is equipped with one 10-Gbps NIC and one NVIDIA GTX780 card. Each client node is equipped with a 1-Gbps NIC and 4 hard disk drives. We use the Iperf tool [30] to test the aggregate TCP throughput between the server and the 7 clients, which fluctuates between 300MB/s and 400MB/s. The aggregate network bandwidth for the server to receive data from the 7 client nodes fluctuates between 600MB/s and 700MB/s.

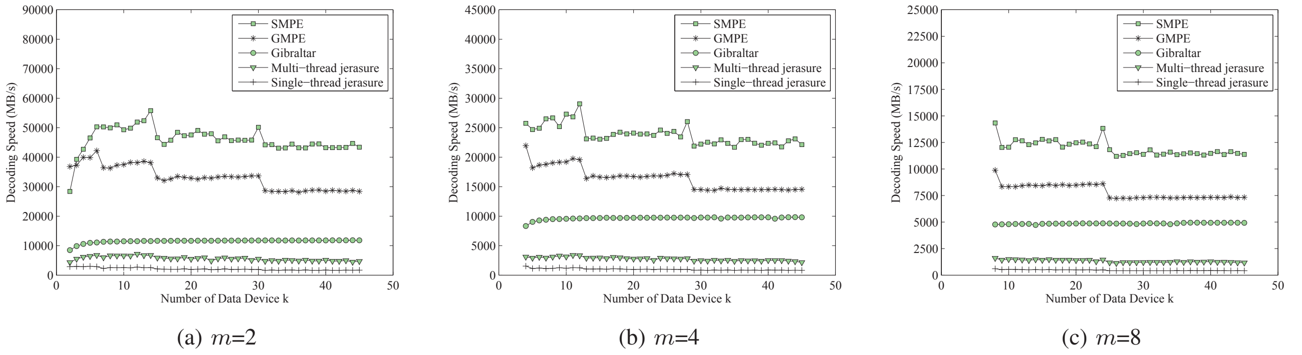
In our experiments, we fix the value of $k+m$ to 28 so as to use all the 28 hard disks, while changing m from 2 to 8 with a step size of 2. To evaluate the encoding performance, we use PErasure to encode a certain amount of data in the memory and distribute the encoded data blocks to the 28 hard disk drives. This is to simulate the scenario that the GPU server receives user data through the network and encodes them directly. We set the data size to $160k$MB such that the total size of encoded data is fixed at 4480MB. To evaluate the decoding performance, we test the worst case that m random disk drives simultaneously fail. The system works in a pipelined manner, and the encoding/decoding, network communications, and disk I/O can overlap with each other. We measure the total time for each single experiment and also the encoding/decoding time. Our experimental results for GMPE and SMPE are shown in Table II.

From the results we can see that both encoding and decoding take a small portion of the total time. This is because in our test bed, the aggregate data transmission rate from the GPU server to the client nodes is the system bottleneck (i.e., between 300-400MB/s). For decoding experiments, the total time is much shorter than that of encoding experiments, due to the fact that the aggregate data transmission rate from the client nodes to GPU server is much higher (i.e., between 600-700MB/s) than the opposite direction. Our experimental results verify that PErasure can be easily used to support erasure coding based data storage systems, and remove the computational bottleneck with a reasonable cost.



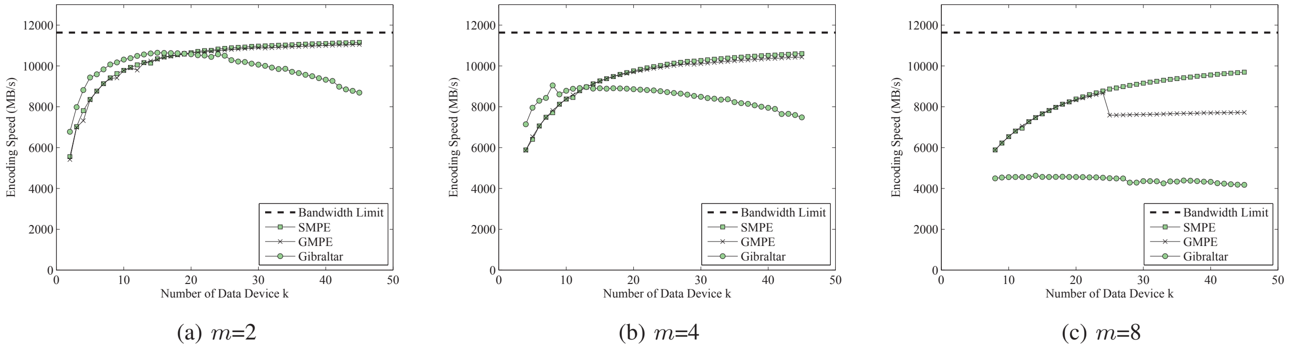
(a) m=2 (b) m=4 (c) m=8

Fig. 3: Comparison of raw encoding speed: k is the number of data devices and m is the number of coding devices



(a) m=2 (b) m=4 (c) m=8

Fig. 4: Comparison of raw decoding speed: k is the number of data devices and m is the number of coding devices



(a) m=2 (b) m=4 (c) m=8

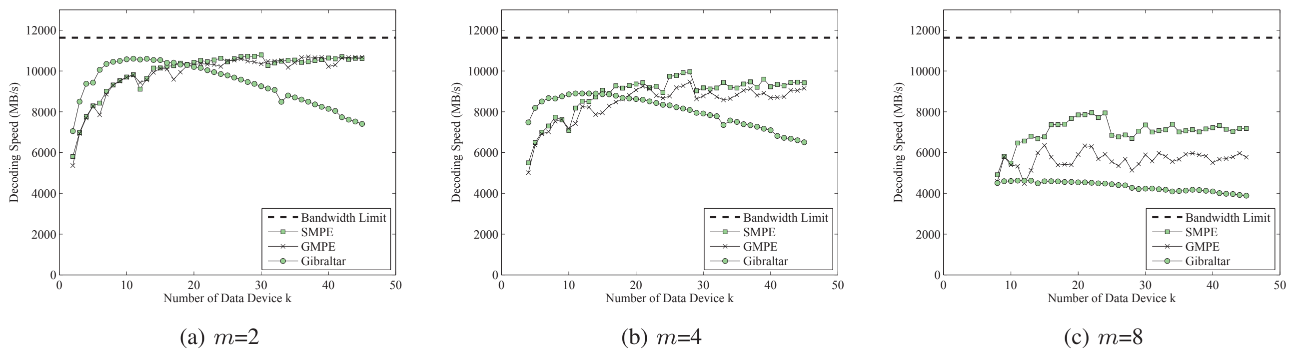
Fig. 5: Comparison of overall encoding speed: k is the number of data devices and m is the number of coding devices

V. Conclusions

In this paper, we presented PErasure, a parallel CRS coding library for GPUs. We evaluated its performance on a contemporary GPU and obtained the following results: (1) When compared with multi-thread CRS coding on a modern quad-core CPU, PErasure is about 10-fold faster; (2) When compared with the most successful GPU implementation of RS coding, PErasure is still 2-4 times faster; (3) The current PCIe 3.0 bandwidth is not fast enough to release the power of PErasure. We use a pipelined design to fully utilize the current PCIe bandwidth. We believe that with the release of PCIe 4.0 in the near future, PErasure can become more attractive to the community of cloud storage systems.

VI. Acknowledgements

This work is supported in part by HONG KONG GRF grant HKBU 210412 and HKBU grant FRG2/13-14/052. We thank all the reviewers for their insightful comments and suggestions.



(a) m=2 (b) m=4 (c) m=8

Fig. 6: Comparison of overall decoding speed: k is the number of data devices and m is the number of coding devices

TABLE II: Encoding and Decoding Time on a Storage Cluster

m	2	4	6	8
Data Size (MB)	4160	3840	3520	3200
GMPE Encoding: Total Time (second)	14.60	13.39	15.31	14.95
GMPE Encoding: Encoding Time (second)	0.96	1.14	1.24	1.34
GMPE Decoding: Total Time (second)	5.83	5.91	5.83	5.82
GMPE Decoding: Decoding Time (second)	1.02	1.19	1.34	1.44
SMPE Encoding: Total Time (second)	14.32	13.79	13.85	14.04
SMPE Encoding: Encoding Time (second)	0.85	0.93	0.98	1.03
SMPE Decoding: Total Time (second)	5.82	5.85	5.79	5.80
SMPE Decoding: Decoding Time (second)	0.92	1.00	1.08	1.16

REFERENCES

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The google file system,” in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, 2003.

[2] D. Borthakur, “The hadoop distributed file system: architecture and design,” 2007.

[3] B. Calder, J. Wang *et al.*, “Windows azure storage: a highly available cloud storage service with strong consistency,” in *Proceedings of the 23rd ACM SOSP*, 2011, pp. 143–157.

[4] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson, “DiskReduce: RAID for data-intensive scalable computing,” in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, 2009, pp. 6–10.

[5] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang, “Rethinking erasure codes for cloud file systems: Minimizing i/o for recovery and degraded reads,” in *Proceedings of USENIX FAST*, 2012.

[6] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin *et al.*, “Erasure coding in windows azure storage,” in *Proceedings of USENIX ATC*, 2012.

[7] M. Sathiamoorthy, M. Asteris *et al.*, “XORing elephants: Novel erasure codes for big data,” in *Proceedings of the 39th International Conference on Very Large Data Bases*. VLDB Endowment, 2013, pp. 325–336.

[8] X.-W. Chu, H. Liu, Y.-W. Leung, Z. Li, and M. Lei, “User-assisted cloud storage system: Opportunities and challenges,” *IEEE MMTC E-Letter*, vol. 8, no. 1, 2013.

[9] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, Z. Wilcox-O’Hearn *et al.*, “A performance evaluation and examination of open-source erasure coding libraries for storage,” in *Proceedings of USENIX FAST*, 2009.

[10] F. Chen, T. Luo, and X. Zhang, “CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives,” in *Proceedings of USENIX FAST*, 2011.

[11] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan *et al.*, “The case for RAMcloud,” *Communications of the ACM*, vol. 54, no. 7, pp. 121–130, 2011.

[12] J. Bloemer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, “An XOR-based erasure-resilient coding scheme,” *ICSI Technical Report No. TR-95-048*, 1995.

[13] H. Klein and J. Keller, “Storage architecture with integrity, redundancy and encryption,” in *Proceedings of IEEE IPDPS*, 2009.

[14] P. Sobe, “Parallel Reed/Solomon coding on multicore processors,” in *Proceedings of International Workshop on SNAPI*, 2010, pp. 71–80.

[15] J. Feng, Y. Chen, and D. Summerville, “EEO: An efficient MDS-like RAID-6 code for parallel implementation,” in *Proceedings of IEEE Sarnoff Symposium*, 2010.

[16] J. Feng, Y. Chen *et al.*, “An extension of RDP code with parallel decoding procedure,” in *Proceedings of IEEE Consumer Communications and Networking Conference*, 2012, pp. 154–158.

[17] J. S. Plank, K. M. Greenan, and E. L. Miller, “Screaming fast Galois field arithmetic using Intel SIMD instructions,” in *Proceedings of USENIX FAST*, 2013.

[18] P. Bhatotia, R. Rodrigues *et al.*, “Shredder: GPU-accelerated incremental storage and computation,” in *Proceedings of USENIX FAST*, 2012.

[19] W. Sun, R. Ricci, and M. L. Curry, “GPUstore: harnessing GPU computing for storage systems in the OS kernel,” in *Proceedings of SYSTOR*, 2012.

[20] A. Khasymski, M. M. Rafique *et al.*, “On the use of GPUs in realizing cost-effective distributed RAID,” in *Proceedings of IEEE MASCOTS*, 2012, pp. 469–478.

[21] M. L. Curry, A. Skjellum, H. Lee Ward, and R. Brightwell, “Gibraltar: A Reed-Solomon coding library for storage applications on programmable graphics processors,” *Concurrency and Computation: Practice and Experience*, vol. 23, no. 18, pp. 2477–2495, 2011.

[22] J. S. Plank, S. Simmerman, and C. D. Schuman, “Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2,” *University of Tennessee, Tech. Rep. CS-08-627*, vol. 23, 2008.

[23] X. Mei, K. Zhao, C. Liu, and X.-W. Chu, “Benchmarking the memory hierarchy of modern GPUs,” in *Proceedings of IFIP NPC*, 2014.

[24] “NVIDIA CUDA programming guide,” *http://docs.nvidia.com/cuda/cuda-c-programming-guide*.

[25] X.-W. Chu, K. Zhao, and M. Wang, “Massively parallel network coding on GPUs,” in *Proceedings of IEEE IPCCC*, 2008.

[26] ——, “Practical random linear network coding on GPUs,” in *Proceedings of IFIP Networking*, 2009.

[27] X.-W. Chu and Y. Jiang, “Random linear network coding for peer-to-peer applications,” *IEEE Network*, vol. 24, no. 4, pp. 35–39, 2010.

[28] Q. Li, C. Zhong, K. Zhao, X. Mei, and X.-W. Chu, “Implementation and analysis of AES encryption on GPU,” in *Proceedings of the 3rd International Workshop on Frontier of GPU Computing*, 2012.

[29] K. Zhao and X.-W. Chu, “GPUMP: A multiple-precision integer library for GPUs,” in *Proceedings of the 1st International Workshop on Frontier of GPU Computing*, 2010.

[30] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs, “Iperf: The TCP/UDP bandwidth measurement tool,” *https://iperf.fr/*, 2005.