G-CRS: GPU Accelerated Cauchy Reed-Solomon Coding

Chengjian Liu, Qiang Wang, Xiaowen Chu, and Yiu-Wing Leung

Abstract—Recently, erasure coding has been extensively deployed in large-scale storage systems to replace data replication. With the increase in disk I/O throughput and network bandwidth, the performance of erasure coding becomes a major bottleneck of erasure-coded storage systems. In this paper, we propose a graphics processing unit (GPU)-based implementation of erasure coding named G-CRS, which employs the Cauchy Reed-Solomon (CRS) code, to overcome the aforementioned bottleneck. To maximize the coding performance of G-CRS, we designed and implemented a set of optimization strategies, such as a compact structure to store the *bitmatrix* in GPU constant memory, efficient data access through shared memory, and decoding parallelism, to fully utilize the GPU resources. In addition, we derived a simple yet accurate performance model to demonstrate the maximum coding performance of G-CRS on GPU. We evaluated the performance of G-CRS through extensive experiments on modern GPU architectures such as Maxwell and Pascal, and compared with other state-of-the-art coding libraries. The evaluation results revealed that the throughput of G-CRS was 10 times faster than most of the other coding libraries. Moreover, G-CRS outperformed PErasure (a recently developed, well optimized CRS coding library on the GPU) by up to 3 times in the same architecture.

Index Terms—Cauchy Reed-Solomon Code, Graphics Processing Unit, Erasure Coding, Distributed Storage System

1 Introduction

The past decade has witnessed the rapid growth of data in large-scale distributed storage systems. For example, the European Centre for Medium-Range Weather Forecasts [1] reached a data amount of 100 PB and experienced an annual growth rate of 45%. Traditionally, data replication scheme such as triplication is used by distributed storage systems to protect users' data due to its simplicity. Currently, with the rapid increase of data volume, the reduction of storage overhead has become an unavoidable task in large-scale storage systems. Consequently, erasure codes have been introduced in many storage systems because they can provide a higher storage efficiency and fault tolerance than data replication. Examples of such storage systems include Microsoft cloud service Azure [2] and Facebook's warehouse [3] and Web service storage system f4 [4]. In addition, distributed file systems such as Hadoop [5] and Ceph [6] have begun to support erasure coding to yield a higher reliability and lower storage overhead.

A general erasure coding system works as follows. Initially, the user data to be protected is divided into k equal sized data chunks. The encoding operation gathers all k data chunks and generates m equal sized parity chunks according to an encoding algorithm. In a distributed storage system, the set of n=k+m data and parity chunks are usually stored at different hardware devices to prevent data loss due to device failures. When no more than m devices fail out of these n devices, the chunks in the failed devices become unavailable. To recover the lost data, a decoding operation gathers k available chunks and reproduces the missing chunks according to a decoding algorithm. The erasure codes that can restore m missing chunks from the remaining k alive chunks have the highest error correction capability and are called *Maximum Distance Separable* (MDS) codes [7]. Reed-Solomon (RS) coding [8] and its variant Cauchy Reed-Solomon (CRS) coding [9] are the two well-known general MDS codes that can support any values of k and m.

In erasure coded storage systems, an encoding operation is triggered when data is written on the storage system, whereas decoding is performed when some lost data is required to be recovered. Thus, the encoding and decoding performance is crucial for the quality of service and user experience. The coding performance is usually inversely proportional to m. Modern data centers begin to deploy 40-100Gb/s Ethernet or even InfiniBand FDR/QDR/EDR to improve the network speed [10], and disk arrays based on Solid-State Drives (SSD) to improve the disk input-output (I/O) performance [11]. Such technology trend pushes the computationally expensive erasure coding into a potential performance bottleneck in erasure-coded storage systems.

Recently, graphics processing units (GPUs) have been used in some storage systems to perform different computationally expensive tasks. Shredder [12] is one framework used for leveraging GPUs to efficiently chunk files for data deduplication and incremental storage. GPUstore [13] is another framework for integrating GPUs into storage systems for file-level or block-level encryption and RAID 6 data recovery. Another GPU-based RAID 6 system has been developed in [14], which uses GPUs to accelerate two RAID 6 coding schemes, namely Blaum-Roth and Liberation codes. This system achieves a coding speed of up to 3GB/s. However, RAID 6 only supports up to two disk failures

- *Chengjian Liu, Qiang Wang, Xiaowen Chu and Yiu-Wing Leung are with the Department of Computer Science, Hong Kong Baptist University, Kowloon, Hong Kong.*
- *Xiaowen Chu is also with HKBU Institute of Research and Continuing Education, Shenzhen, China.*
  E-mail: {cscjliu,qiangwang,chxw,ywleung}@comp.hkbu.edu.hk

and is not suitable for large-scale systems. To date, Gibraltar [15], which employs table lookup operations to implement Galois field multiplications, is the most successful GPU-based Reed-Solomon coding library; notably, the system has much higher speed over the single-thread Jerasure [16], which is the most popular erasure coding library for CPUs. PErasure [17] is a recent CRS coding library for GPUs and its performance is even better than Gibraltar. However, PErasure does not fully utilize the GPU memory system and results in sub-optimal performance. With the rapid improvement of networking speed and aggregated disk I/O throughput, there is a demand to further improve the coding performance.

In this paper, we aim to design a new CRS coding library for GPUs, namely G-CRS, that can fully utilize the GPU resources and deliver high coding performance that can saturate the state-of-the-art network speed. To this end, we have designed new data structures and a set of optimization strategies. G-CRS can achieve more than 50GB/s of raw coding performance on a modern GPU Nvidia Titan X for the case of $m = 16$ (i.e., the system can withstand up to 16 device failures)[1]. Our major contributions are summarized as follows:

1) We present a step-by-step optimization analysis to reveal the method of utilizing GPUs to accelerate CRS coding.
2) We present a simple yet accurate performance model to understand the major factors that affect the coding performance.
3) We provide a pipelined mechanism to enable our G-CRS to achieve a peak performance by efficiently overlapping data copy operations and coding operations.
4) We conduct extensive experiments to validate the effectiveness of our proposed G-CRS, compare with other state-of-the-art coding libraries, and analyze the dominating factor.
5) G-CRS is open-source and freely available to the public.

The remainder of this paper is organized as follows. Section 2 introduces the background of GPU computing, CRS coding, and some related work. Section 3 gives the details of our optimization and implementation of G-CRS. Section 4 presents a performance model of G-CRS to understand the impact of computational power, memory bandwidth, and coding parameters on the coding performance. Section 5 presents a pipelined mechanism that can further enhance the performance of G-CRS. Section 6 presents the experimental results of G-CRS and other state-of-the-art libraries. We summarize the paper in Section 7.

2 Background and Related Work

In this section, we first briefly introduce the key concepts of GPU computing and CRS coding. Then, we identify the challenges of applying erasure coding in modern distributed storage systems. Finally, we introduce some related work.

2.1 GPU Computing and CUDA

Modern GPUs are typically equipped with hundreds to thousands of processing cores evenly distributed on several streaming multiprocessors (SMs). For example, the Nvidia GTX 980 with Maxwell architecture contains 16 SMs and 4GB off-chip GDDR memory. Each SM has 128 Stream Processors (SPs, or cores) and a 96-KB on-chip memory named *shared memory*, which has much higher throughput and lower latency than the off-chip GDDR memory [18]. Besides the 2MB L2-cache shared by the whole GPU, each SM also has a small amount of on-chip cache to speed up the data access of read-only *constant memory*.

Currently, CUDA is the most popular programming model for GPUs [19]. A typical CUDA program comprises host functions, which are executed on the central processing unit (CPU), and kernel functions, which are executed on the GPU. Each kernel function runs as a grid of threads, which are organized into many equal sized *thread blocks*. Each thread block can include a set of threads distributed in a number of *thread warps*, each of which has 32 threads that execute the same instruction at a time. Threads in a thread block can share data through their shared memory and perform barrier synchronization.

2.2 Cauchy Reed-Solomon Coding

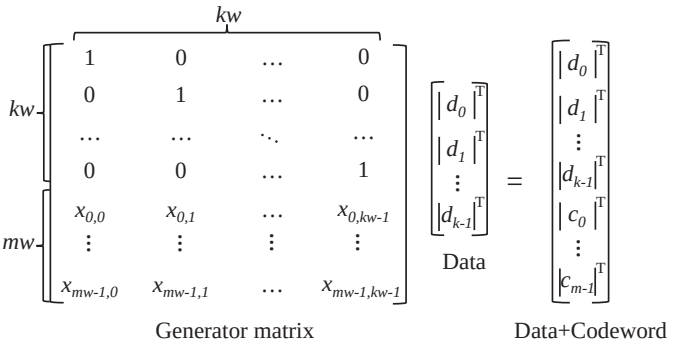


Fig. 1. Illustration of CRS encoding. $|d_i|^T$ is the w-row format of data chunk d_i. $|c_i|^T$ is the w-row format of parity chunk c_i.

As illustrated in Fig. 1, the encoding procedure of CRS takes k equal sized data chunks $d_0, d_1, ..., d_{k-1}$ as input, and generates m parity chunks $c_0, c_1, ..., c_{m-1}$ as output. To perform the coding, it needs to select an integer parameter w that is no less that $log_2(k + m)$. Hence a CRS code can be defined by a triple (k, m, w). CRS first defines an $m \times k$ Cauchy distribution matrix over Galois Field $GF(2^w)$, and then expands it into a $(k + m)w \times kw$ *generator matrix* over $GF(2)$ whose elements are either 1 or 0 [9]. Notice that the top kw rows of the generator matrix is an identity matrix.

Each data chunk d_i needs to be transformed into w rows, denoted by $D_{i,0}, D_{i,1}, ..., D_{i,w-1}$. The w-row format of data chunk d_i is denoted by $|d_i|^T$. Then all the k data chunks can be combined into a data matrix with kw rows. By multiplying the generator matrix and data matrix over $GF(2)$, we can get $k+m$ output chunks that include the k original data chunks (due to the $kw \times kw$ identity sub-matrix) and the m parity chunks. Notice that the multiplication in $GF(2)$ can be implemented by efficient bit-wise XOR operations, which is the major property of CRS.

Fig. 2 presents a concrete example of the CRS encoding process where *k*=2, *m*=2 and *w*=2. Data chunk d_i consists of two rows: $D_{i,0}$ and $D_{i,1}$, $i = 0, 1$. In the actual encoding

1. The source code and experimental results of our G-CRS are available at http://www.comp.hkbu.edu.hk/~chxw/gcrs.html.

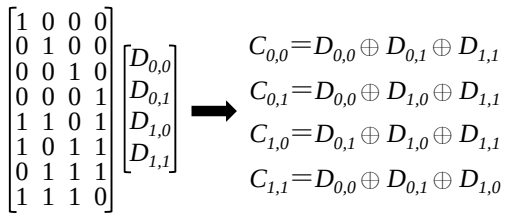


Fig. 2. A concrete example of CRS encoding. $k = 2, m = 2, w = 2$.

process, we only need to calculate the m parity chunks using the bottom mw rows of the generator matrix. For each parity chunk, the values from the corresponding row vector in the generator matrix determine which data chunks will be involved in the XOR operations. For example, the fifth row vector from the generator matrix in Fig. 2 $< 1, 1, 0, 1 >$ determines that $C_{0,0}$ is generated from $D_{0,0}, D_{0,1}$ and $D_{1,1}$ with XOR operations.

When either a data chunk or a parity chunk becomes unavailable due to device failure, a decoding operation is triggered to restore the missing chunk. To recover a parity chunk, w row vectors from the generator matrix together with all data chunks serve as the input. By contrast, to recover a data chunk, an inverse matrix is generated from the generator matrix (which can be done offline in advance), and k alive chunks serve as the input with w row vectors from the inverse matrix. The encoding and decoding operations are essentially the same in terms of data access pattern and computation. Therefore, we use the term coding to represent both encoding and decoding.

2.3 Opportunities and Challenges in Erasure Coding

Data reliability and availability are critical requirements for data storage systems. Although coding-based RAID 5/6 have become the industry standards for decades, replication is still the de facto data protection solution in large-scale distributed storage systems. With the increase in the amounts of data and the deployment of expensive SSDs, there is a great opportunity for erasure coding because it can provide much lower storage overhead and higher reliability compared with data replication. In [20], a comprehensive comparison of erasure coding and replication was presented. However, erasure coding is a compute- and data-intensive task which brings practical challenges to its adoption in distributed storage systems.

In an erasure coded distributed storage system, there could be three potential performance bottlenecks: the aggregated disk I/O throughput, the network bandwidth, and the coding performance. Modern data centers have started to deploy high-speed network switches with more than 40Gb/s of bandwidth per network port [10]. Facebook and LinkedIn are already working on 100Gb/s network for their data centers [21] [22]. Meanwhile, the sequential I/O throughput of a single SSD has been improved to more than 4Gb/s [23], and the aggregated I/O throughput of a disk array can easily match the network bandwidth. However, the throughput of software implemented erasure coding is inversely proportional to the number of parity chunks, and is typically less than 10 Gb/s on multi-core CPUs [17], which makes erasure coding impractical for large-scale distributed systems. Modern GPUs have tens of TFlops computation power and an internal memory bandwidth of a few hundreds of GB/s, providing an opportunity to speed up erasure coding to saturate the disk I/O and network bandwidth. This motivates us to design and implement G-CRS to fully utilize the GPU power and achieve a high coding throughput.

2.4 Related Work

Coding Performance: Many research and industrial studies have focused on improving the performance of erasure coding. One pioneer study optimizes the Cauchy distribution matrix that results in better coding performance for CRS coding on the CPU by performing less XOR operations [24]. Jerasure [16] is a popular library that implements various kinds of erasure codes on the CPU, including optimized CRS coding. Optimization with efficient scheduling of XOR operations on CPU is presented in [25]. These sequential CRS algorithms are designed for CPUs and are not suitable for GPUs due to their complicated control flows. Another thread of research aims to exploit parallel computing techniques to speed up erasure coding. For multi-core CPUs, CRS codes have been parallelized in [26], [27]; EVENODD codes have been parallelized in [28]; and RDP codes have been parallelized in [29]. A fast Galois field arithmetic library for multi-core CPUs with the support of Intel SIMD instructions has recently been presented in [30]. Although these works have achieved great improvement, their coding performance is still not comparable to the throughput of today's high speed networks, especially when a large number of parity data chunks are required for higher data reliability. These parallel algorithms cannot be directly applied to GPUs either, due to their different hardware architectures. For many-core GPUs, the Gibraltar library [15] implements the classical Reed-Solomon coding and outperforms many existing coding libraries on CPUs. PErasure [17] is a recent CRS coding library for GPUs and its performance is better than Gibraltar. However, PErasure does not fully utilize the GPU memory system and results in sub-optimal performance.

Offloading Tasks to GPUs: Modern GPUs are embedded with hundreds to thousands of arithmetic processing units that provide tremendous computing power, and attracts many work to port computationally intensive applications from CPUs to GPUs. For example, G-Blastn [31], which is a nucleotide alignment tool for the GPU, achieves more than 10 times of performance improvement over its CPU version NCBI-BLAST. In [32], regular expression matching on GPUs can be 48 times faster than that on CPUs. SSLShader [33] and AES [34] both demonstrate great performance improvement when data encryption algorithms are offloaded to GPUs. Network coding on GPUs, such as those in [35], [36] and Nuclei [37], are the most closely related work in addition to the aforementioned Gibraltar [15] and PErasure [17]. These studies aim to improve the performance of network coding to match the throughput of high speed networks for both encoding and decoding.

3 Design of G-CRS

In this section, we first present a high-level view of G-CRS and define our terminologies. Next, we provide a baseline implementation of CRS coding on GPUs that directly migrates from the CPU version. Subsequently, we analyze the potential drawbacks of this basic design and provide a set of optimization strategies to accelerate CRS coding on GPUs. Our G-CRS is implemented by applying all optimization strategies described in this section.

Fig. 3 illustrates the system architecture of G-CRS that implements a (k, m, w) CRC code. The *bitmatrix* stores the bottom mw rows of the generator matrix of the CRS code. The input data is divided into k equal sized *data chunks*, and the output includes m parity chunks with the same size. We use s to represent the number of bytes of a *long* data type on the target hardware platform, i.e., $s = sizeof(long)$. We define a *packet* as s consecutive bytes. The XOR of two packets can then be efficiently carried out by a single instruction. We define a *data block* as w consecutive packets, where w is the parameter of CRC and should be no less than $log_2(k + m)$. The number of data blocks in a chunk is denoted by N.

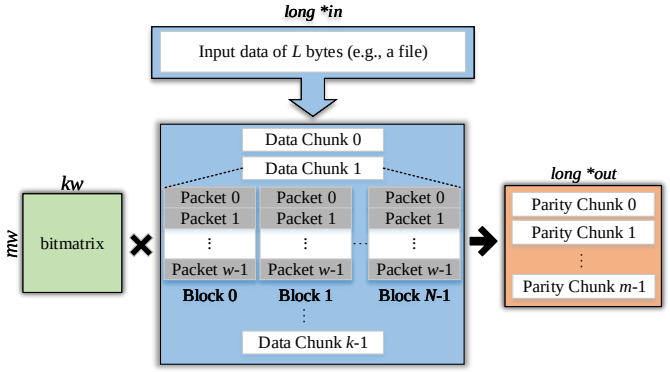


Fig. 3. Input and Output for Coding

We summarize the high level workflow of G-CRS in Algorithm 1. When the size of user data is greater than the available GPU memory, we will encode the data in different rounds.

Algorithm 1 High Level Workflow of G-CRS

1: **Input:** *k, m, w, bitmatrix, dataSize*
2: Compute *round* from *k, m, w, dataSize*
3: Construct *bitmatrix*
4: Allocate GPU memory
5: Copy *bitmatrix* to GPU
6: **for** each $i \in [1, round]$ **do**
7: Copy k data chunks of the i-th round to GPU
8: Launch the encoding kernel function to generate m coding chunks
9: Copy back m parity chunks of the i-th round to main memory
10: **end for**
11: Free memory resources

3.1 Baseline Implementation

Listing 1 presents the baseline implementation of the CRS coding kernel written in CUDA, which is directly migrated from a CPU version. The kernel function defines the behavior of a single GPU thread. In this implementation, each thread is responsible for encoding a single packet. When the kernel function is launched, a total of mwN threads will be created to generate the m parity chunks in parallel. Each element of *bitmatrix* is represented by an integer. This kernel function works as follows: (1) The input buffer *in* and the output buffer *out* are located in the GPU global memory. (2) The mw bottom row vectors from the generator matrix are stored in the *bitmatrix*, which is also located in the global memory. (3) Each thread calculates the initial index of its assigned packet in each data block (line 13). Then, each thread iterates its corresponding row in the *bitmatrix* to determine the data required to perform the XOR operation (lines 16-24). (4) Each thread writes a packet, which is stored in the variable *result*, to the output buffer (line 26).

```
1  __global__ void crs_coding_kernel_naive
2  (long *in, long *out, int *bitmatrix,
3  int size, int k, int m, int w) {
4    int blockUnits = blockDim.x / w;
5    int blockPackets = blockUnits * w;
6    int tid = threadIdx.x + blockIdx.x *
         blockPackets;
7    int unit_id_offset = tid / w * w;
8    int unit_in_id = tid % w, i, j;
9
10   if (threadIdx.x >= blockPackets) return;
11   if (tid >= size ) return;
12
13   int index = threadIdx.y*k*w*w+unit_in_id*k*w;
14   long result = 0, input;
15
16   for (i = 0; i < k ; ++i) {
17     for (j = 0; j < w ; ++j) {
18       if (bitmatrix[index] == 1) {
19         input = *(in+size*i+unit_id_offset+j);
20         result = result^input;
21       }
22       ++index;
23     }
24   }
25
26   *(out + size*threadIdx.y + tid) = result;
27 }
```

Listing 1. A baseline implementation of CRS coding on GPU

Some severe performance penalties exist in this baseline implementation, implying that the GPU resources are substantially under-utilized. To design a fully optimized version of CRS coding on GPU, a thorough understanding of the GPU architecture is required, including its memory subsystem and its method of handling branch divergence. We have identified three major performance penalties of the baseline implementation:

Inefficient memory access: The memory access pattern of the baseline kernel implementation in its global memory causes a considerable performance penalty. From lines 18-20 of Listing 1, we can observe that each XOR operation requires two memory reads, one from the *bitmatrix* and another one from the input buffer. Since the baseline implementation stores the *bitmatrix* and input buffer in global memory, each memory access to the *bitmatrix* or input buffer is very likely to generate a global memory read because

TABLE 1
Features of different types of GPU memory space

Memory	Access Latency	Cached	Scope
Global	Around 600 cycles	Yes	All threads
Constant	Around 400 cycles	Yes	All threads
Shared	Around 20 cycles	N/A	All threads in the same thread block
Register	Around 4 cycles	N/A	Single thread

the GPU L2 cache is small and shared by the whole global memory. Accesses to global memory have long latency and low throughput as compared to the XOR operations, hence the performance of the baseline implementation will be limited by the global memory throughput.

Underutilization of memory bandwidth: Since an integer is used to store the binary value of 1 or 0 in the *bitmatrix*, the memory bandwidth utilization is very low when accessing the *bitmatrix*. Regarding the input data, each data packet is in fact accessed by mw threads, implying an opportunity of temporal reuse. But due to the small size of GPU L2 cache and large data size, the L2 cache hit rate is not satisfactory.

Penalty of Control Flow: Finally, the GPU architecture is inefficient at handling control flow. Specifically, a constant performance penalty is caused by control flow. This is because the threads in a warp can only execute the same instruction simultaneously; if threads in a warp execute different instructions due to control divergence, the execution of these instructions are serialized. The penalty is often incurred by *if-then-else* and *loop* statements. In the baseline implementation, we have two *for loop* statements at lines 16 and 17, and an *if* statement at line 18. Although some threads do not require the execution of the code inside the *if* statement, given a warp, they are required to wait for other threads to finish the calculation in that same warp before proceeding with their execution.

3.2 Optimization Strategies

In this subsection, we present a set of optimization strategies for overcoming the major performance penalties discussed in the previous subsection. Here we first investigate how to improve the performance of accessing *bitmatrix*. Next, we achieve efficient access to data blocks by loading them to on-chip shared memory before accessing and reducing the thread dimension. We also present our strategy of removing control divergence. At last, we discuss how to make decoding performance the same as encoding.

3.2.1 Efficient access to the bitmatrix

GPU memory system has a very complicated structure [18]. Table 1 presents three major features of four types of commonly used GPU memory. Global memory has the longest access latency, and registers have the shortest access latency. Although the access latency of physical constant memory is close to that of global memory, each SM offers an on-chip cache for it, which makes it perfect to store a small amount of read-only data that are frequently accessed.

Since *bitmatrix* is frequently accessed by all threads, we can store it in the constant memory to reduce access latency because each SM offers a small on-chip cache for read-only data in the constant memory. The *bitmatrix* contains kmw^2 elements. However, the size of the constant cache is limited to the KB-level. If the size of the *bitmatrix* exceeds the size of the constant cache in each SM, the access will eventually go to the global memory and result in long access latency. One possible solution is to use the $char$ data type (which has a size of 1 byte) to store each element of the *bitmatrix* instead of using the int type (which has a size of 4 bytes). This approach reduces the size of the *bitmatrix* to one-quarter of its original size, enabling it to more easily fit in the constant cache.

Although storing the *bitmatrix* in constant memory reduces the access latency, it may still suffer from the following performance penalty. If multiple threads in the same warp access the same memory address in the constant cache, the access can proceed in parallel. However, if multiple threads in the same warp access different memory addresses in the constant cache, these accesses will be serialized [38], incurring substantial performance penalty. We thus further investigate how w successive threads from the same warp access the *bitmatrix* in each iteration. The access pattern to the *bitmatrix* in our implementation is illustrated in Fig. 4. In each iteration, w threads simultaneously access w different elements. If the access latency of the on-chip memory takes one time slot, w time slots may be required to obtain elements from the *bitmatrix* for each iteration.

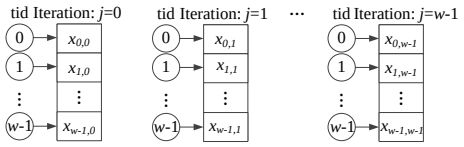


Fig. 4. Access pattern to bitmatrix. tid refers to thread ID.

To reduce the access latency to the read-only cache, we need a strategy that allows threads in each warp to access only one memory address to obtain elements from the *bitmatrix* in each iteration. One possible solution is to assign the same packet to w successive threads in a warp by increasing the size of a packet. This solution enables the threads in a warp to access just one element in the *bitmatrix* at a time. However, it changes the access pattern to the input data. In addition, each thread will be responsible for coding w packets, which may reduce the opportunity for parallelism with the same workload.

Although one memory address is used to store an element in the *bitmatrix* (e.g., each element occupies one byte), each element can be represented by one bit because the value of an element is either 1 or 0. This suggests that w bits can be used to store w elements of the *bitmatrix*. In practice, the value of w can be set to no more than 8 when $k + w \leq 256$, which implies that we can use a byte to store w elements of the *bitmatrix*. If wm is not greater than 32, an integer can be used to carry a column vector from the *bitmatrix*. For this solution, the mapping for each element from the *bitmatrix* to each memory address is illustrated in Fig. 5. The access to one memory location to fetch elements from the *bitmatrix* for a thread warp in each iteration is presented in Fig. 6. The access latency is reduced from w

time slots to one time slot in each iteration. In addition, only $4kw$ bytes are required to store the entire *bitmatrix* if $mw \leq 32$. For mw greater than 32, the $\lfloor 32/w \rfloor w$ elements of a column vector can be stored in a 32-bit integer. The access pattern is similar to the case of $mw \leq 32$.

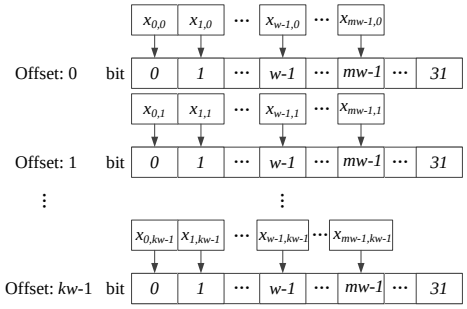


Fig. 5. Mapping of each bitmatrix element to memory

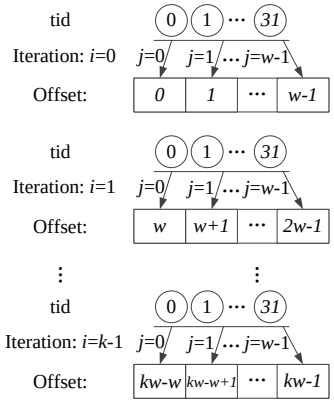


Fig. 6. Memory access pattern for a single warp

After w elements are fetched from one memory address, we need to extract the exact element from the *bitmatrix* for each thread. We use a variable named $unit\_id\_offset$ to mark the index of the packet that is assigned to each thread for each block. Using this variable, its start index in the *bitmatrix* can be located. When each column vector is stored in one value of *int* type, $unit\_id\_offset$ can locate the bit for its row vector elements. To detect whether the value is 1 or 0, a shift operation and XOR operation is adopted. Then, the *if* operation is used to detect whether the XOR operation should be performed. The problem of serialized access to the *bitmatrix* is completely eliminated by our proposed solution.

3.2.2 Efficient memory access

After removing performance penalty introduced by accessing the *bitmatrix*, the major inefficient memory access comes from the access to the input data referenced by *long *in*. Fig. 7 illustrates the access pattern of w successive threads to access wk packets from k data chunks. In the i-th iteration, the memory request is on the i-th data chunk. This access pattern is applicable to all threads in the baseline implementation.

From Table 1, we can see that shared memory is a block of on-chip memory that can be accessed by all threads from the same block. This indicates that we can load data blocks

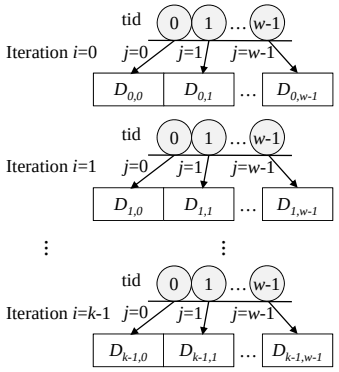


Fig. 7. Access pattern of w threads on the data blocks

to shared memory for frequent access by threads in the same block. To reduce the access latency of the global memory for input data chunks, threads in each unit cooperate with each other to load the required data into the shared memory at the outside iteration. The allocated shared memory for each thread block can be reused for k times. Each thread then accesses the data from shared memory for calculation in the inner iteration.

```
1  extern __shared__ long sh_data[];
2  for(int i = 0; i < k; ++i){
3    if(threadIdx.y == 0)
4      sh_data[threadIdx.x] = (in+size*i) + tid;
5    __synchronized__();
6
7    for(int j = 0; j < w; ++j){
8      if(bitmatrix[index] == 1){
9        result ^= sh_data[unit_in_id + j];
10       ++index;
11     }
12   }
13   __synchronized__();
14 }
```

Listing 2. Use shared memory to reduce the data access latency

The code for loading input data to shared memory is illustrated in Listing 2. The array *sh_data* is used for storing data from the global memory. If a thread block contains 128 threads in one dimension, only 1KB of shared memory is required for each block. Because each packet takes only 8 bytes, up to 128 packets can be loaded from the global memory at a time. By using this optimization strategy, each packet of input data is accessed only once from the global memory.

3.2.3 Improve the utilization of memory bandwidth

Now the performance of input data access can be improved by loading data to the shared memory. However, the global memory bandwidth is still underutilized. When the threads whose *threadIdx.y* = 0 issue an access request to the global memory (line 3 in Listing 2), other threads in the same block remain idle. Notice that the value of the variable *threadIdx.y* ranges from 0 to $m - 1$. Because each SM has a limited number of active blocks and active threads, the memory bandwidth is underutilized due to the inadequate number of memory requests to saturate the global memory bandwidth. If m is equal to 1, the memory bandwidth has an opportunity to be fully utilized because each SM may have more active threads to concurrently issue memory

requests. However, with the increase of m, the number of active blocks markedly decreases, implying substantial underutilization of the global memory bandwidth due to reduced concurrent memory requests.

To conquer this issue, we assign the tasks of coding m packets to each thread to improve memory bandwidth utilization. Each thread is now responsible for coding a column of packets from m parity blocks. In addition to improving the utilization of the global memory bandwidth, this approach can also remove the conditional branch at line 3 in Listing 2, which leads to better performance by avoiding the execution of additional instructions and removing branch divergence.

3.2.4 Optimization of control divergence

Branch divergence introduces a severe performance penalty because threads in a warp can execute the same instruction at a time. The execution of different instructions must be serialized for threads in a warp. In our baseline implementation, branches exist in three places and may introduce performance degradation (lines 16-18 in Listing 1).

We first investigate the *if* statement of the *bitmatrix* operation. The purpose of this statement is to perform an XOR operation with a packet where the corresponding element from the *bitmatrix* is 1; otherwise, no XOR operation is performed. We use the following solution to remove this branch. First, we set all bits of a *long* type variable named *fullOnebit* to 1. Then, *fullOnebit* performs a multiplication with the element extracted from the *bitmatirx*, denoting an operation to copy the element from the *bitmatirx* (either 1 or 0) to all bits of *fullOnebit*. Finally, *fullOnebit* performs an AND operation with the packet obtained from the shared memory. Using this approach, a branch divergence is removed.

For the inner loop in the baseline implementation, the size of the loop depends on w. Since w is small in practice, we can manually apply loop unrolling to remove the inner loop for different values of w.

3.2.5 Parallelism of decoding

The encoding process can be fully parallelized because each thread reads kw packets from the input data to code a packet. However, achieving complete parallelism may not be straightforward for the deocding. When both a data block and parity block become unavailable, the available k blocks must be used together with an inverse matrix to restore the missing data block. Subsequently, w row vectors of the generator matrix are used to restore the missing parity block. Because all data blocks are required to serve as the input for restoring the missing parity block, it is necessary to wait until all data blocks become available to restore any missing parity block. In this case, the decoding process is serialized, which reduces the decoding performance.

To reduce the performance penalty of decoding caused by this serialization, a strategy for complete parallelism of the decoding process must be adopted. This suggests that a matrix with mw row vectors must be used to restore any m missing blocks with any k available blocks. Moreover, for decoding, each thread can restore a packet with input data and a row vector. Consequently, decoding can be completely parallelized for any case, and the decoding performance is the same as encoding.

We first investigate how to generate a matrix that can restore both the data blocks and parity blocks. According to Fig 1, each packet from the parity blocks can be represented by the XOR results of kw packets from data blocks. We can translate this problem into how to represent a packet from a parity block with an XOR operation of kw input packets. Each packet can be represented by the XOR of packets from the kw input packets. Subsequently, the missing data packet in the representation of each missing packet from parity blocks can be replaced. After replacement and calculation, a representation of XOR operations can be obtained from the kw input packets. Finally, a row vector can be written to generate the missing packets from parity blocks with kw input packets.

We present a simple example herein to demonstrate how our proposed method works. Consider the generator matrix from Fig. 8 (a) as an example, where k=2, w=2, and m=2. We assume that the first data block and the first parity block are missing. To restore packets $D_{0,0}$ and $D_{0,1}$, $D_{1,0}$, $D_{1,1}$, $C_{1,0}$ and $C_{1,1}$ are taken as the input. We then use the existing application programming interface to generate two row vectors for restoring $D_{1,0}$, $D_{1,1}$ as shown in Fig. 8 (a). Therefore, $D_{0,0}$ can be represented as $D_{1,1} \oplus C_{1,0} \oplus C_{1,1}$ and $D_{0,1}$ can be represented as $D_{1,0} \oplus D_{1,1} \oplus C_{1,0}$. Moreover, $C_{0,0}$ can be represented as $D_{0,0} \oplus D_{0,1} \oplus D_{1,1}$ and $C_{0,1}$ can be represented as $D_{0,0} \oplus D_{1,0} \oplus D_{1,1}$. By replacing $D_{0,0}$ and $D_{0,1}$ with their representation of input packets in the representations of $C_{0,0}$ and $C_{0,1}$, we can get that $C_{0,0}$ can be represented with $D_{1,0} \oplus D_{1,1} \oplus C_{1,1}$ and $C_{0,1}$ can be represented as $D_{1,0} \oplus C_{1,0} \oplus C_{1,1}$. Subsequently, two row vectors can be written to restore the packets from the missing parity blocks, as illustrated in Fig .8 (b). The four row vectors can function with the input packets to decode the missing blocks in parallel.

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} D_{1,0} \\ D_{1,1} \\ C_{1,0} \\ C_{1,1} \end{bmatrix} = \begin{bmatrix} D_{0,0} \\ D_{0,1} \end{bmatrix} \qquad \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} D_{1,0} \\ D_{1,1} \\ C_{1,0} \\ C_{1,1} \end{bmatrix} = \begin{bmatrix} C_{0,0} \\ C_{0,1} \end{bmatrix}$$

(a) Recover Packets From Data Blocks (b) Recover Packets From Parity Blocks

Fig. 8. An example of decoding missing blocks

We provide a function to generate the matrix that can decode the missing blocks that contain both data blocks and parity blocks. The whole matrix is placed in the *bitmatrix* to perform the decoding process, and the entire process becomes the same as the encoding.

4 Performance Model

The performance of a GPU application is usually bounded by either the computation capability or memory throughput. For GPU applications with low compute-to-memory ratio, the GPU's memory bandwidth limits its performance, while the computational power limits the performance of applications with high compute-to-memory ratios.

Given different combinations of k, m and w, the compute-to-memory ratio of G-CRS could be different,

which indicates that the performance of G-CRS may be limited to either memory bandwidth or computational power of a GPU. To elucidate the utilization of a GPU's computational resources and memory bandwidth by G-CRS under different compute-to-memory ratios, we present a performance model of G-CRS in this section.

4.1 Kernel Analysis

CRS coding with G-CRS includes encoding and decoding processes, which essentially perform the similar operations. To analyze the performance of encoding or decoding, we first categorize the operations in the kernel function into two types, namely SM and dynamic random-access memory (DRAM). SM operations include computational instructions, shared memory operations, constant cache operations, and synchronization instructions that are executed in SM; whereas DRAM operations include global memory read-/write transactions.

To determine the performance of G-CRS, six key operations from different phases are summarized in Table 2.

TABLE 2
Categorize the operation type by SM and DRAM

Phase	Main operation	Domain
Step 3	Load data from global memory	DRAM
Step 3	Store data to shared memory	SM
Step 5	Load data from constant memory	SM
Step 6	Load data from shared memory	SM
Step 6	Computation	SM
Step 9	Store results to global memory	DRAM

We build our G-CRS performance model based on the smallest scheduling unit of thread warp. First, to obtain the i-th data chunk, each warp launches a global memory request to DRAM. Since each thread in a warp loads one long integer of the i-th chunk, totally 32 long integers with length 2048 bits are fetched from global memory. Such memory access should take $\frac{2048}{BW}$ transactions where BW denotes the memory interface width of the GPU. Then the data is stored in shared memory for later usage. Thus, k data chunks need totally $\frac{2048k}{BW}$ global memory transactions.

Second, each thread loads part of the temporary data chunk in shared memory as well as one coding bit from *bitmatrix*. Then they do m times of coding work to produce m integers which contribute to the final m parity chunks. Each warp executes mw shared memory loading and data coding for each data chunk. Thus, each warp executes kmw SM-type operations.

Finally, each warp writes their coding results back to global memory, which launches totally $\frac{2048m}{BW}$ global memory transactions since each thread writes back one long integer.

In summary, each warp has respectively $\frac{2048(k+m)}{BW}$ DRAM data transactions and kmw SM-type operations. Thus, the ratio of DRAM data transactions to SM-type operations varies with different values of (k, m, w), and causes the performance to be dominated by either DRAM data transactions or SM-type operations.

4.2 Dominant Factor Analysis

We define the memory-to-compute ratio r as Eq. (1) to better illustrate the performance of G-CRS. Since $\frac{2048}{BW}$ is a constant that depends on the GPU hardware, we remove it from the r definition.

$$r = \frac{k+m}{kmw} = \frac{1/m + 1/k}{w}. \tag{1}$$

Each thread in a kernel issues $k+m$ global memory requests and conducts kmw XOR operations. Notably, r will decrease with the increase of k, m, w. Moreover, different values of (k, m, w) lead to different memory bandwidth consumption, which is calculated by the total size of data divided by the time spent on the execution. Fig. 9a presents the memory throughput changing with r of two modern GPUs, whose physical parameters are listed in Table 3. We define C_{SM} by Eq. (2) to describe the computational capability with respect to memory bandwidth. f_{core} and f_{mem} represent the GPU core frequency and memory frequency, respectively. $\#SM$ and $\#coresPerSM$ represent the number of SMs and the number of cores per SM, respectively. We have two observations from Fig. 9 (a). First, memory throughput continuously increases with a larger r and reaches an upper bound with a certain r. Second, two GPUs have different r thresholds, denoted by ρ, to reach the maximum memory throughput, which can be explained using C_{SM}. A larger C_{SM} implies that the hardware has a more powerful computational ability, and a smaller r indicates that the computation part of the kernel scarcely limits the performance of the GPU.

$$C_{SM} = \frac{f_{core} \times \#SM \times \#coresPerSM}{f_{mem} \times BW} \tag{2}$$

A GPU schedules its threads in rounds. In each round, the number of concurrent warps (a.k.a active warps) is denoted by $\#Aw$. The number of rounds is denoted by $\#Rs$. Then the execution time of a kernel, denoted by T_{kl}, can be calculated as $T_{kl} = T_{act} \times \#Rs$, where T_{act} denotes the execution time of one round on one SM. Next we discuss how to calculate T_{act}.

As mentioned before, a kernel could be either compute-bound or memory-bound. More specifically, for our G-CRS kernel, we use ρ to help differentiate those two cases. If $r \leq \rho$, we consider the kernel to be compute-bound and the DRAM transaction latency can be hidden by SM-type operations. In this case, we only focus on the kmw SM-type operations. The execution time can be calculated by $kmw \times T_{SM} \times \#Aw$, where T_{SM} denotes the average number of cycles needed for one SM-type operation. On the other hand, if $r > \rho$, we consider the kernel to be memory-bound and the SM-type operations can be hidden by DRAM transactions. Thus, we can focus on the $\frac{2048(k+m)}{BW}$ DRAM operations, whose total time is given by $2048(k+m)/BW \times T_{DRAM} \times \#Aw$, where T_{DRAM} denotes the average number of cycles needed for one DRAM transaction. To summarize, T_{act} can be calculated by Eq. 3.

We also list some key parameters of our performance model in Table 3. T_{DRAM} and T_{SM} can be obtained by microbenchmarks.

TABLE 3
Metrics and parameter settings of different GPUs

Metrics/Parameter	GTX980	Titan X(Pascal)
Compute capability	5.2	6.1
SMs * cores per SM	16 * 128	28 * 128
Memory interface width	256-bit	384-bit
Memory size	4GB	12GB
Base core frequency	1126MHz	1417MHz
Base memory frequency	3500MHz	5000MHz
C_{SM}	2.57	3.21
ρ	0.06	0.05
T_{DRAM}	3.516	2.6
T_{SM}	1.506	1.5

$$T_{act} = \begin{cases} kmw \times T_{SM} \times \#Aw, & \text{if } r \leq \rho \\ \frac{2048(k+m)}{BW} \times T_{DRAM} \times \#Aw, & \text{if } r > \rho \end{cases} \quad (3)$$

To validate the accuracy of our performance model, we adopt various (k, m, w) combinations in which k ranges from 1 to 45, m ranges from 1 to 4 and w ranges from 4 to 8, subject to the constraint of $w \geq log_2(k+m)$. We apply a large number of blocks for each kernel execution to ensure that the GPUs keep as busy as possible. For each group of (k, m, w), we run the kernel for ten times and calculate the average execution time. We use mean absolute precision error (MAPE) to evaluate the accuracy of our model. The MAPE is defined as $MAPE = |T_{model} - T_{exp}|/T_{exp}$, where T_{model} refers to the execution time derived from the performance model while T_{exp} refers to the average time measured by real experiments.

The accuracies of the performance estimation of the two GPUs are shown in Fig. 9 (b). Most of performance estimations reveal a MAPE within 7%. The average MAPE is 2.79% for GTX980 and 3.37% for Titan X respectively. To address the reproducibility of our performance model, we also conduct significance test with bilateral t-distribution for the average MAPE. With 95% confidence interval, our model can achieve no more than 3.9% average MAPE on these two GPUs.

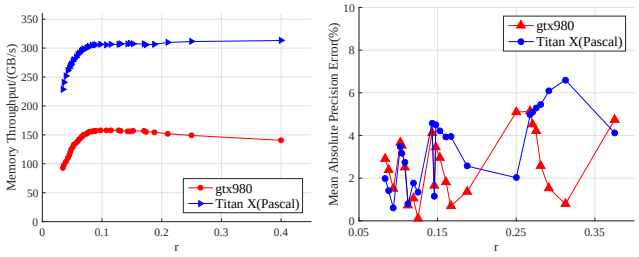


Fig. 9. (a) Memory throughput with different r (b) Model accuracy

5 Pipelined G-CRS

For applications with high computational intensity and a large amount of data involved, one of the major difficulties of obtaining high performance is to balance the cost of computational intensity and I/O operations. For erasure coding, we must move data to the specified GPU device and write the result back to main memory or network interface through I/O transfer. To achieve high performance for G-CRS with both I/O and computation involved, we provide a pipelined mechanism in this section that overlaps data transfer and computation.

Efficient utilization of both the computational resources of GPUs and I/O bandwidth between the main memory and GPUs' global memory is critical for achieving peak performance to offload the coding task to GPUs. Fig. 10 (a) illustrates the execution of two tasks. Conventionally, serialized execution is adopted, and Task 2 waits until the execution of Task 1 is completed. For pipelined execution, both the computational resources and I/O bandwidth are more efficiently utilized. Our G-CRS resembles the execution pattern of these tasks, because data must be copied from main memory for coding and then the parity data must be copied back to the main memory.

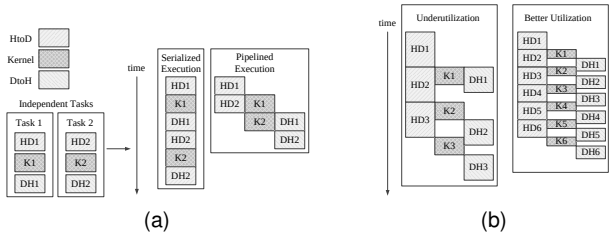


Fig. 10. Cases for Pipelined I/O and Computation

To efficiently utilize both the GPU computational resources and I/O bandwidth, we design a pipelined mechanism for G-CRS to overlap I/O operations and computation tasks. We divide the coding work into independent tasks and use different GPU streams to perform each independent task in order to simultaneously run data transfer and computation.

Algorithm 2 illustrates how our pipeline mechanism works. We launch a number of GPU streams to achieve asynchronous execution of each task. The execution for Pipelined G-CRS is similar with the pipelined execution illustrated in Fig. 10 (a). We can observe that both GPU computational resources and I/O bandwidth get better utilization with Pipelined G-CRS by overlapping the execution of I/O tasks and computational tasks.

Algorithm 2 Pipelined G-CRS

1: **Input:** k, w, m, bitmatrix and numStream
2: **for** each $i \in [0, numStream - 1]$ **do**
3: Copy the i-th data chunk of stream s to GPU asynchronously
4: **end for**;
5: **for** each $i \in [0, numStream - 1]$ **do**
6: Launch the kernel of stream i asynchronously
7: **end for**;
8: **for** each $i \in [0, numStream - 1]$ **do**
9: Copy back the i-th parity chunk of stream s to main memory asynchronously
10: **end for**;

TABLE 4
Major Parameters for Measuring Different Workloads

Parameter	Value Range
k	10
m	1-4
w	4-8
Thread Blocks	2^4-2^{13}
Threads	128 Per Thread Block
Input data size	$8kw\lfloor 128/w \rfloor$ Bytes Per Thread Block

An example of executing three independent coding tasks is illustrated on the left side of Fig. 10 (b). The total execution time is $3t_{HD} + t_{DH} + t_K$, which reveals that the time for executing all tasks comprises three parts. The first part is the time required to transfer all data to the GPU, the second part is the time required for the kernel execution, and the third part is the time required to transfer the data from the GPU back to the main memory. Overall, dividing each task into smaller tasks can reduce the total execution time. The right side in Fig. 10 (b) depicts the execution of each independent task divided into two equal sized tasks, for which the total execution time is reduced to $3t_{HD} + \frac{t_{DH}}{2} + \frac{t_K}{2}$. Assuming that the total time for transferring all of the data from main memory to GPU memory is T_{HD}, the total time for coding is T_K, and the total time for transferring the entire parity chunks from GPU memory to main memory is T_{DH}. We divide the work into x tasks so that each task requires $\frac{T_{HD}}{x}$ time to transfer its data to the GPU memory, $\frac{T_K}{x}$ to produce the parity chunks, and $\frac{T_{DH}}{x}$ to transfer the parity chunks back to the main memory. The total time for completing the entire task is calculated as follows:

$$T_{total} = T_{HD} + \frac{T_K}{x} + \frac{T_{DH}}{x} \tag{4}$$

Equation (4) reveals that the entire coding task can be divided into many smaller tasks that can effectively utilize both the I/O bandwidth and computation resources to achieve better performance.

6 Performance Evaluation

In this section, we conduct a set of experiments to evaluate G-CRS from various aspects on two modern GPU architectures: Maxwell and Pascal. First, we evaluate the coding performance of G-CRS with various workloads. Next, we analyze the peak raw coding performance of G-CRS and compare it with the performance of other major state-of-the-art coding libraries: Jerasure, Gibraltar, gferasure [39] and PErasure [17]. We also analyze how our different optimization strategies improve the performance of CRS coding.

6.1 Throughput Under Different Workloads

The coding performance of G-CRS depends on the level of parallelism on a given GPU for the specified k, m, and w values. The level of parallelism that can be achieved depends on the workload assigned to each task. In other words, an adequate workload must be assigned to a given GPU to achieve a high throughput.

We measure the coding performance of G-CRS under different workloads on two modern generations of GPUs: Maxwell and Pascal. The major parameters for our experiments are presented in Table 4. First, we set the number of threads in each thread block to 128 because this yields the maximum number of active threads and the maximum number of thread thread blocks in each SM for Maxwell and Pascal. Notably, the input data size is slightly different for different w values due to memory alignment. The number of working threads in each thread blocks is $w\lfloor 128/w \rfloor$, and the input data size is the total number of thread blocks multiplied with $8kw\lfloor 128/w \rfloor$ bytes.

The experimental results are presented in Fig. 11. Considering the coding performance with minimum thread blocks for a kernel, the throughput of the Maxwell GTX 980 is around 11 GB/s when m equals 1. We can see that the performance is doubled when the workload is doubled for $m = 1$ or 2. But for the cases of $m = 3$ or 4, there is no big performance improvement. This is because larger value of m requires more computation resources. The situation is similar for Pascal Titan X GPU. But it has a higher throughput with minimum number of chunks for a kernel, because it has higher memory bandwidth than Maxwell GTX 980.

At maximum workload, in which each kernel has 2^{13} thread blocks, the throughput reaches the peak for each GPU. Moreover, when computational power is not the bottleneck of the coding performance, the peak throughput is limited by the GPU memory bandwidth; therefore, the throughput of the Maxwell GTX 980 is about half of that of the Pascal Titan X when $m = 1$. However, when each thread is responsible for coding more packets, the computational power is gradually saturated and hence limits the coding throughput. With maximum workload, the coding throughput decreases to around two third of its original peak when m is increased from 1 to 4. For Pascal Titan X, which has a higher memory bandwidth and computational power, the decrease is not so obvious as the Maxwell GTX 980. Specifically, the peak throughput of the Pascal Titan X is approximately 230 GB/s when each thread is assigned with four packets.

The value of w also has a great impact on the coding performance. Specifically, a GPU performs optimally when it has the smallest value of w, which is 4, than with any other values. Especially when computation is the limitation, smaller w leads to better performance because the increase in w indicates an increase in computational intensity for each working thread. Furthermore, when bandwidth is the limitation, w with values of 4 and 8 may outperform w with other values because of the increased number of working threads in each thread block; thus, a more efficient bandwidth utilization is achieved. Based on our experimental results, we conclude that when the computation is the bottleneck, w with a smaller value outperforms w with larger values. On the other hand, when GPU memory bandwidth is the bottleneck, more working threads in each thread block can lead to better performance.

We present only the encoding performance to represent the overall coding performance of G-CRS herein because the difference between encoding and decoding throughput is less than 1/1,000 in our measurement. The evaluation result-

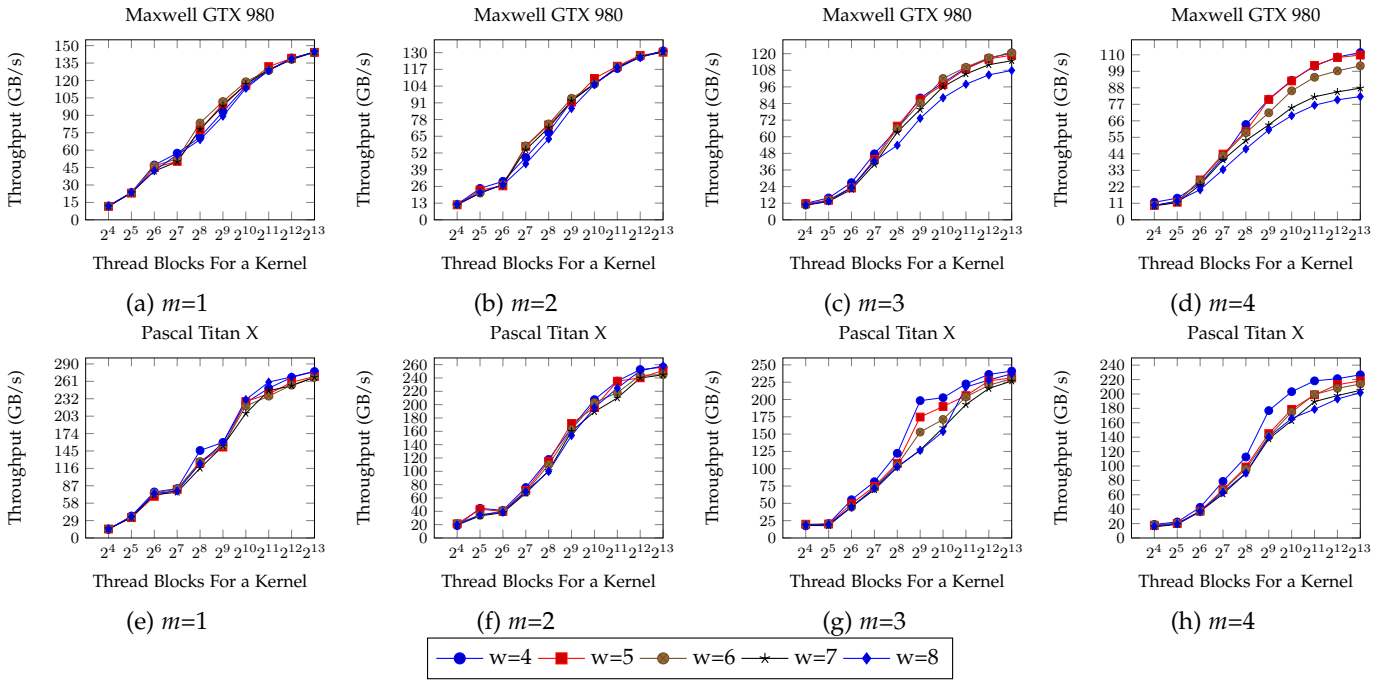


Fig. 11. Throughput under Different Workloads

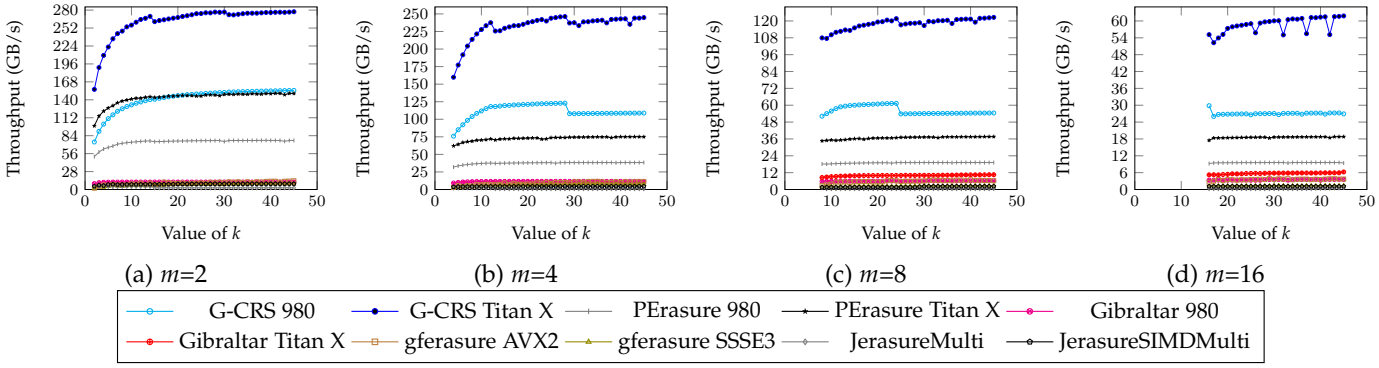


Fig. 12. Comparison of raw encoding performance

s also reveal that more workloads typically indicate a more efficient coding performance because of higher parallelism. At minimum workload, the throughput is usually sufficient to saturate 40G/100G-bps of Ethernet.

6.2 Peak Raw Coding Performance

After investigating the coding performance under different workloads, we measure the peak raw coding performance of our G-CRS and compare it with other state-of-the-art coding libraries, namely Jerasure, Gibraltar, gferasure and PErasure. Jerasure is a well-known erasure coding library on CPU [40]. We measure two versions of Jerasure: one with CRS coding and one with accelerated Reed-Solomon coding that uses Intel SIMD instructions presented in [30]. Gibraltar is a recently developed open-source Reed-Solomon coding library for GPUs. gferasure is an on-going erasure code library accelerated with a high-performance Galois field arithmetic [39]; we measure its performance using SSSE3 and AVX2, which achieve the fastest coding performance of gferasure.

The CPU used for evaluating Jerasure and gferasure is an Intel i7-6800K model with six cores running at 3.4GHz. To maximize the coding performance of the CPU, we used OpenMP to parallel Jerasure and gferasure. We set the values of m to 2, 4 and 8, and the values of k range from m to 45. For Jerasure with CRS coding, we set the packet size to 8 KB to achieve maximum coding performance. The size of each input data chunk is fixed to 10 MB. Finally, we selected the smallest available value of w for erasure coding. Notably, the size of each data chunk varies slightly different due to the necessary alignment for w with values of 5, 6 and 7.

The encoding performance is presented in Fig. 12. As depicted, our G-CRS is more than ten times faster than the Gibraltar, Jerasure and gferasure, and up to 3 times faster than PErasure on the same architecture.

Jerasure shows the lowest coding throughput in all experiments. Although its Reed-Solomon coding outperforms CRS coding with a single thread, the CRS coding achieves 3.5x higher performance with four threads, but the Reed-

Solomon coding only achieves 1.5x higher performance when using four threads. By contrast, our G-CRS outperforms Jerasure by more than 10x in all situations.

Gibraltar achieves a coding throughput of approximately 11 GB/s for m with values of 2 and 4. The performance of Gibraltar is limited by the PCIe bandwidth because it uses zero copy to manage the memory buffer, which indicates that it accesses the host memory directly for coding. However, when m is equal to 8, the coding throughput was only 8 GB/s on the Maxwell GTX 980, where the PCIe bandwidth no longer limited its performance.

The coding performance of gferasure is markedly fast on the CPU. In addition, gferasure with AVX2 outperforms that with SSSE3, because AVX2 can process a 256-bit word which is double of the length of SSSE3. We also achieves a 2x speedup with multi-threads in our measurement. It also outperforms Gibraltar when $m = 2$ and 4. Still, our G-CRS is five to ten times faster than gferasure because G-CRS exhibits a better utilization of GPUs's computational power and memory bandwidth.

In short, G-CRS can effectively exploit GPU's computational power and memory bandwidth, and exhibits a more efficient coding performance compared with other state-of-the-art coding libraries. In all cases, G-CRS's coding performance can saturate the state-of-the-art disk I/O and network bandwidth.

6.3 Optimization Analysis

We have proposed a set of optimization strategies to overcome every possible performance penalty to build G-CRS. In this subsection, we evaluate the effectiveness of our optimization strategies and then investigate the dominating factor that contributes the most to the performance improvement.

6.3.1 Effectiveness of optimization strategies

We evaluate the effectiveness of our optimization strategies by gradually applying them to the baseline version in Listing 1. We fix k as 10 and set m in the range of 1 to 4. The value of w varies from 4 to 8 for different computation workloads. The size of each data chunk is set at 10 MB to achieve the peak coding performance.

The evaluation results are presented in Fig. 13, wherein only the encoding performance is shown to represent the entire coding performance. The Base version is the baseline implementation presented in Listing 1. The Bitmatrix version stores each column vector from the *bitmatrix* in one memory address. The one named EfficientAccess is the version that we apply shared memory optimization in Listing 2 on the Bitmatrix version. For the Bandwidth version, we reduce the thread dimension of y to 1, and each working thread is assigned with m packets. G-CRS is our final version with all optimization strategies applied.

It is obvious that the Base version has the lowest coding performance, about 10% of the highly optimized G-CRS. When we change the way how *bitmatrix* is stored and accessed, we observe a big performance improvement from the Base version to Bitmatrix version. But the degree of improvement decreases greatly when m is increasing. This is because more working threads are launched with the increase of m, which increases the access to data chunks and brings more performance penalties.

We can see that the version named EfficientAccess has much better performance than the Bitmatrix when $m = 2, 3, 4$, and the one named Bandwidth further improves the performance of EfficientAccess. Our G-CRS obtains the best coding performance in all experiments. The one named Bandwidth has the closest performance to our G-CRS. From this we first can see that the better utilization of bandwidth can bring better performance. And we also can see that the control divergence brings the least performance penalty compared with others.

We can see an obvious performance improvement after adding our optimization strategy one by one to form G-CRS. Thus, we can conclude that each of our optimization is effective to improve the coding performance.

6.3.2 Dominating factor

We have shown that our optimization strategies can improve the performance of the Base version greatly. To find out the dominating factor for the performance improvement, we remove one optimization strategy from the G-CRS and keep the remaining ones to observe its impact on the performance.

Fig. 14 illustrates the experimental results. We use the minus symbol to represent that the corresponding optimization strategy is not used. For example, -Bitmatrix means we remove the optimization of *bitmatrix* from G-CRS.

It is obvious to see that without the optimization of *bitmatrix*, the performance degrades significantly. Even though other optimizations are used, its performance is only slightly better than the Base version in Fig. 13. There are two performance penalties for accessing *bitmatrix* from global memory directly. One is the long access latency to fetch an element from global memory. Another one is the serialized access to different elements due to non-coalescing access. From our experimental results, we can conclude that the optimization of the *bitmatrix* is the dominating factor to the performance improvement.

For others with the optimization of *bitmatrix*, we can see the performance penalty is reduced. Without loading data from global to on-chip shared memory, which is the version of -EfficientAccess, the performance decreases obviously compared with G-CRS.

The performance of the -Bandwidth is similar with G-CRS when m is 1. But when m increases, there is a performance gap between -Bandwidth and G-CRS because the memory bandwidth is underutilized with the increase of m in the version of -Bandwidth. The version of -ControlDivergenceOpt is the one that is closest to the performance of our G-CRS. However, the performance gap increases with the increase of m and w.

In a word, the optimization to *bitmatrix* is the dominating strategy that can improve the performance by 4-7 times, while other optimization strategies can further improve the performance by 30-100%.

6.4 Overall Performance

For both encoding and decoding, data must be transferred from other sources to the device memory via PCIe. We

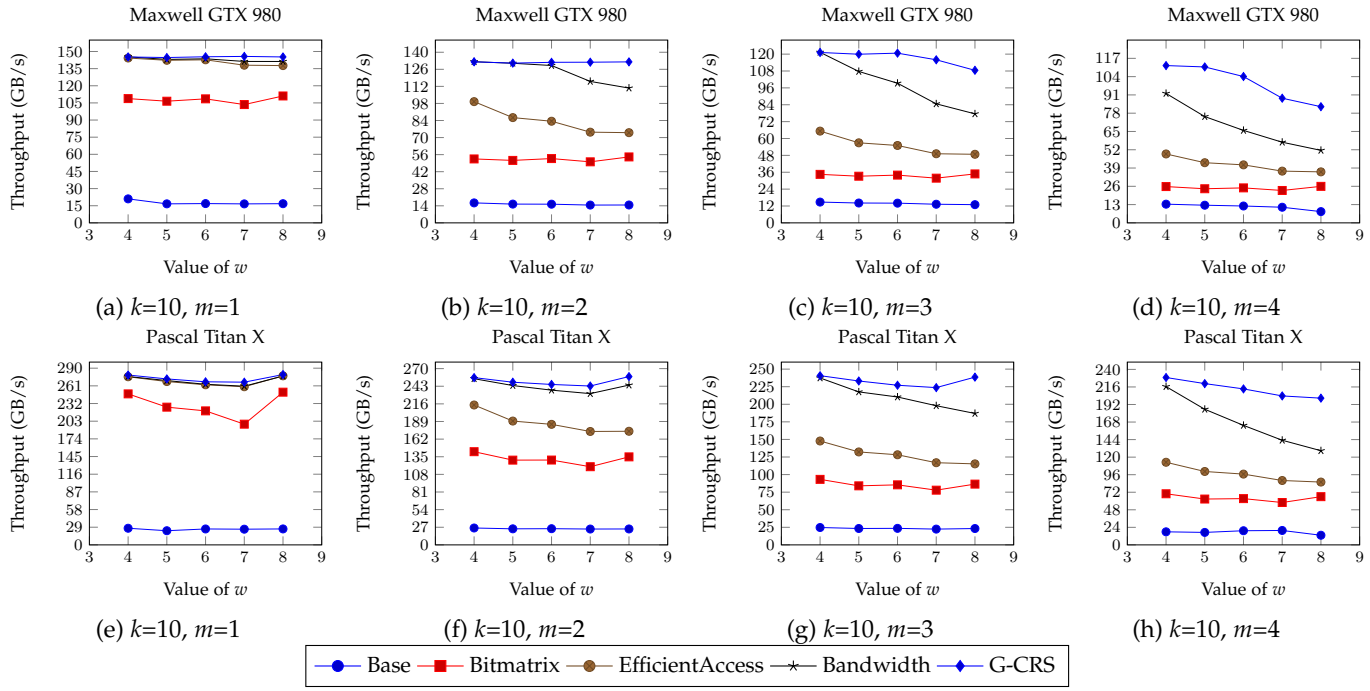


Fig. 13. Optimization analysis

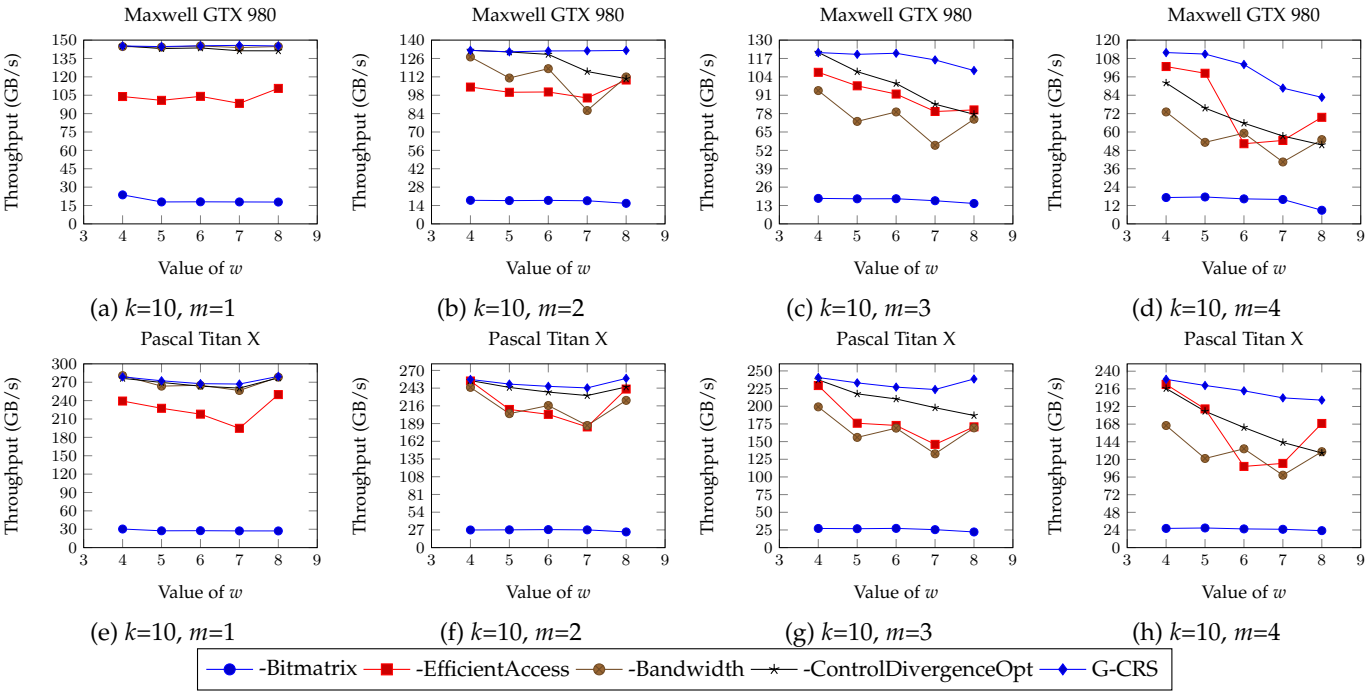


Fig. 14. Dominating factor analysis

evaluate the coding performance of our pipelined G-CRS with PCIe transfer involved. The PCIe used in our experiment is PCIe 3.0, which supports bi-directional PCIe communications. Our benchmarking experiments show that the highest effective PCIe bandwidth on our testbed is 12 GB/s. We set each data chunk size to 10 MB, divide the entire coding task into 10 small tasks, and use 10 CUDA streams to implement pipelined G-CRS.

The encoding performance is presented in Fig. 15. Since no obvious differences have been observed between encoding and decoding performance, the decoding performance is not presented. By using pipelined G-CRS, the coding speed reaches up to 11 GB/s, which is approximately 90% of the bandwidth limitation. This overall coding throughput can already saturate the network bandwidth. With the recent progress on the standardization of PCIe 4.0 and PCIe 5.0 [41], the overall performance of G-CRS can be naturally boosted in the future.

7 Conclusion

In this paper, we presented the design of a GPU accelerated CRS coding and evaluated its performance on two current generations of GPUs. Our evaluation results indicate that G-CRS can operate with various sizes of data and code block to obtain a favorable coding throughput. The coding performance of G-CRS can be up to ten times faster than other state-of-the-art coding libraries such as Jerasure, Gibraltar and gferasure. Moreover, our optimization strategies can effectively utilize both GPU memory bandwidth and computation resources. The overall performance of G-CRS is only limited by PCIe bandwidth. Our optimization strategies can also be applied to other applications with similar compute and data access patterns. Examples include regenerating codes that are built on top of Cauchy Reed-Solomon codes [42], and also circular-shift linear network coding [43].

Acknowledgement

We would like to thank the anonymous reviewers for their valuable comments. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan X Pascal GPU used for this research. This work is partially supported by Shenzhen Basic Research Grant SCI-2015-SZTIC-002 and Hong Kong ITF Grant ITS/443/16FX.

References

[1] M. Grawinkel, Nagel *et al.*, "Analysis of the ECMWF storage landscape," in *Proc. of USENIX FAST'15*, 2015.
[2] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in Windows Azure Storage," in *Proc. of USENIX ATC'12*, 2012.
[3] K. Rashmi, N. B. Shah *et al.*, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster," 2013.
[4] S. Muralidhar, W. Lloyd *et al.*, "F4: Facebook's warm blob storage system," in *Proc. of USENIX SOSP'14*, 2014.
[5] B. Fan, W. Tantisiriroj *et al.*, "Diskreduce: Replication as a prelude to erasure coding in data-intensive scalable computing," *Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh*, 2011.
[6] S. A. Weil, S. A. Brandt *et al.*, "Ceph: A scalable, high-performance distributed file system," in *Proc. of USENIX OSDI'06*, 2006.
[7] M. Blaum and R. M. Roth, "On lowest density MDS codes," *IEEE Transactions on Information Theory*, vol. 45, no. 1, pp. 46–59, 1999.
[8] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.
[9] J. Bloemer, M. Kalfane *et al.*, "An XOR-based erasure-resilient coding scheme," *ICSI Technical Report No. TR-95-048*, 1995.
[10] C. Binnig, A. Crotty *et al.*, "The end of slow networks: It's time for a redesign," vol. 9, no. 7, 2016, pp. 528–539.
[11] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "A large-scale study of flash memory failures in the field," in *Proc. of ACM SIGMETRICS'15*, 2015.
[12] P. Bhatotia, R. Rodrigues, and A. Verma, "Shredder: GPU-accelerated incremental storage and computation." in *Proc. of USENIX FAST'12*, 2012.
[13] W. Sun, R. Ricci, and M. L. Curry, "GPUstore: Harnessing GPU computing for storage systems in the OS kernel," in *Proc. of ACM SYSTOR'12*, 2012.
[14] A. Khasymski, M. M. Rafique *et al.*, "On the use of GPUs in realizing cost-effective distributed RAID," in *Proc. of IEEE MASCOTS'12*, 2012.
[15] M. L. Curry, A. Skjellum *et al.*, "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.
[16] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2," Citeseer, Tech. Rep., 2008.
[17] X. Chu, C. Liu *et al.*, "Perasure: a parallel cauchy reed-solomon coding library for GPUs," in *Proc. of IEEE ICC'15*, 2015.
[18] X. Mei and X. Chu, "Dissecting GPU memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2017.
[19] N. Corporation, "NVIDIA CUDA programming guide," *http://docs.nvidia.com/cuda/cuda-c-programming-guide*.
[20] W. Lin, D. M. Chiu, and Y. Lee, "Erasure code replication revisited," in *Peer-to-Peer Computing*, 2004, pp. 90–97.
[21] B. Jasmeet, M. Hany, and Y. Zhiping, "Opening designs for 6-pack and wedge 100," *https://code.facebook.com/posts/203733993317833/opening-designs-for-6-pack-and-wedge-100/*.
[22] M. Yuval Bachar, "The linkedin data center 100g transformation," *https://engineering.linkedin.com/blog/2016/03/the-linkedin-data-center-100g-transformation*.
[23] W. Wang, T. Xie, and A. Sharma, "SWANS: An interdisk wear-leveling strategy for RAID-0 structured SSD arrays," *Trans. Storage*, vol. 12, no. 3, Apr. 2016.
[24] J. S. Plank and L. Xu, "Optimizing cauchy reed-solomon codes for fault-tolerant network storage applications," in *Proc. of IEEE NCA'06*, 2006.
[25] J. Luo, M. Shrestha, L. Xu, and J. S. Plank, "Efficient encoding schedules for XOR-based erasure codes," *IEEE Transactions on Computers*, vol. 63, no. 9, pp. 2259–2272, 2014.
[26] H. Klein and J. Keller, "Storage architecture with integrity, redundancy and encryption," in *Proc. of IEEE IPDPS'09*, 2009.
[27] P. Sobe, "Parallel reed/solomon coding on multicore processors," in *Proc. of IEEE SNAPI'10*, 2010.
[28] J. Feng, Y. Chen, and D. Summerville, "EEO: An efficient MDS-like RAID-6 code for parallel implementation," in *Proc. of IEEE Sarnoff Symposium*, 2010.
[29] J. Feng, Y. Chen *et al.*, "An extension of RDP code with parallel decoding procedure," in *Proc. of IEEE CCNC'12*, 2012, pp. 154–158.
[30] J. S. Plank, K. M. Greenan, and E. L. Miller, "Screaming fast galois field arithmetic using intel SIMD instructions." in *Proc. of USENIX FAST'13*, 2013.
[31] K. Zhao and X. Chu, "G-BLASTN: Accelerating nucleotide alignment by graphics processors," *Bioinformatics*, vol. 30, no. 10, pp. 1384–1391, 2014.
[32] G. Vasiliadis, M. Polychronakis *et al.*, "Regular expression matching on graphics hardware for intrusion detection," in *Proc. of International Workshop on Recent Advances in Intrusion Detection*, 2009.
[33] K. Jang, S. Han *et al.*, "SSLShader: Cheap SSL acceleration with commodity processors." in *Proc. of USENIX NSDI'11*, 2011.
[34] Q. Li, C. Zhong *et al.*, "Implementation and analysis of AES encryption on GPU," in *Proc. of IEEE HPCC-ICESS*, 2012.
[35] X. Chu, K. Zhao, and M. Wang, "Practical random linear network coding on GPUs," in *Proc. of IFIP Networking'09*, 2009.

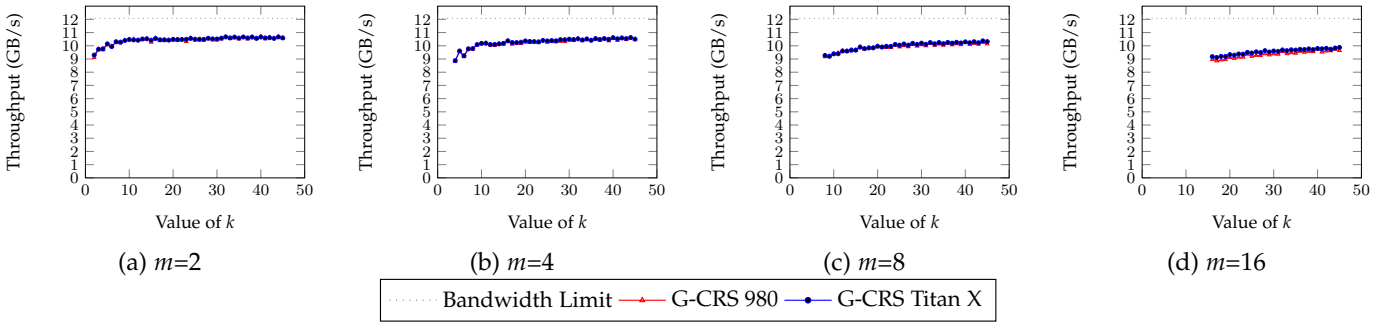


Fig. 15. Overall encoding performance

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

[37] H. Shojania, B. Li, and X. Wang, "Nuclei: GPU-accelerated many-core network coding," in *Proc. of IEEE INFOCOM'09*, 2009.

[38] N. Corporation, "Kepler tuning guide," *http://docs.nvidia.com/cuda/kepler-tuning-guide*.

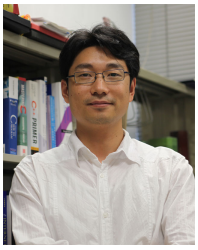
[39] L. T. Ethan Miller, Thomas Schwarz and A. Kwong, "High-performance galois field arithmetic," *http://www.crss.ucsc.edu/proj/galois.html*.

[40] J. S. Plank, J. Luo *et al.*, "A performance evaluation and examination of open-source erasure coding libraries for storage." in *Proc. of USENIX FAST'09*, 2009.

[41] E. Born, "Pcie 4.0 specification finally out with 16 gt/s on tap," *https://techreport.com/news/32064/pcie-4-0-specification-finally-out-with-16-gt-s-on-tap*, 2017.

[42] K. Rashmi, N. B. Shah *et al.*, "A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers," in *Proc. of ACM SIGCOMM'14*, 2014.

[43] H. Tang, Q. T. Sun, Z. Li, X. Yang, and K. Long, "Circular-shift linear network coding," *arXiv preprint arXiv:1707.02163*, 2017.



Xiaowen Chu received the B.E. degree in computer science from Tsinghua University, P.R. China, in 1999, and the Ph.D. degree in computer science from the Hong Kong University of Science and Technology in 2003. Currently, he is an associate professor in the Department of Computer Science, Hong Kong Baptist University. His research interests include distributed and parallel computing and wireless networks. He is serving as an Associate Editor of IEEE Access and IEEE Internet of Things Journal. He is a senior member of the IEEE.

Chengjian Liu received his MS degree in College of Computer Science & Software Engineering, SZU. He has been working in Guangdong Province Key Laboratory of Popular High Performance Computers & Shenzhen Key Laboratory of Service Computing and Applications High Performance Computing during his study in SZU. Currently he is a Ph.D. candidate at Dept of Computer Science, Hong Kong Baptist University. His research interests include Distributed Storage, General-Purpose GPU Computing.

Yiu-Wing Leung received his B.Sc. and Ph.D. degrees from the Chinese University of Hong Kong. He has been working in the Hong Kong Baptist University and now he is Professor of the Computer Science Department and Programme Director of two MSc programmes. His research interests include three major areas: 1) network design, analysis and optimization, 2) Internet and cloud computing, and 3) systems engineering and optimization. He has published many papers in various IEEE journals and conferences.

Qiang Wang received his B.Sc. degree from South China University of Technology in 2014. He is currently a Ph.D. candidate at Department of Computer Science, Hong Kong Baptist University. His research interests include General-Purpose GPU Computing and power-efficient computing. He is a recipient of Hong Kong PhD Fellowship.