CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **00**:1–15
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Adaptive Loss-Less Data Compression Method Optimized for GPU decompression

Shunji Funasaka, Koji Nakano*, Yasuaki Ito

Department of Information Engineering, Hiroshima University, Kagamiyama 1-4-1, Higashihiroshima, JAPAN

SUMMARY

There is no doubt that data compression is very important in computer engineering. However, most lossless data compression and decompression algorithms are very hard to parallelize, because they use dictionaries updated sequentially. The main contribution of this paper is to present a new lossless data compression method that we call Adaptive Loss-Less (ALL) data compression. It is designed so that the data compression ratio is moderate but decompression can be performed very efficiently on the GPU. This makes sense for applications such as training of deep learning, in which compressed archived data are decompressed many times. To show the potentiality of ALL data compression method, we have evaluated the running time using five images and five text data and compared ALL with previously published lossless data compression methods implemented in the GPU, Gompresso, CULZSS, and LZW. The data compression ratio of ALL data compression is better than the others for eight data out of these 10 data. Also, our GPU implementation on GeForce GTX 1080 GPU for ALL decompression runs 84.0-231 times faster than the CPU implementation on Core i7-4790 CPU. Further, it runs 1.22-23.5 times faster than Gompresso, CULZSS, and LZW running on the same GPU. Copyright © 2010 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: lossless data compression, parallel algorithms, GPGPU, parallel prefix scan

1. INTRODUCTION

1.1. Background

A GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [1]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [2], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [3], since they have thousands of processor cores and very high memory bandwidth.

CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [4]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-96 Kbytes. The global memory is implemented as an off-chip DRAM, and thus, it has large capacity, say, 1.5-12 Gbytes, but its access latency is very large. When we develop CUDA programs, we need to consider *the coalescing* of the global memory access [3, 5]. To maximize the bandwidth between the GPU and the off-chip DRAM, the consecutive addresses of the global memory must be accessed

*Correspondence to: Department of Information Engineering, Hiroshima University, Kagamiyama 1-4-1, Higashihiroshima, JAPAN.

at the same time. Thus, CUDA threads should perform coalesced access when they access the global memory.

There is no doubt that data compression is one of the most important tasks in the area of computer engineering. In particular, almost all image data are stored in files as compressed data formats. There are basically two types of image compression methods: *lossy* and *lossless* [6]. Lossy compression can generate smaller files, but some information in original files are discarded. Hence, decompression of lossy compressed data does not generate files identical to the original data. On the other hand, lossless compression creates compressed data, from which we can obtain the exactly same original data by decompression.

The main purpose of this paper is to present a novel lossless data compression method optimized for efficient decompression using the GPU. Usually, data in a large archive are stored in a compressed format to reduce the space. If each of compressed data is accessed many times, it makes sense to use a data compression method that maximizes performance in terms of decompression time and compression ratio. In particular, if archived data stored in a data center are accessed by users, and they are processed on the GPU for display, such data compression method optimized for GPU decompression should be used. Also, in training phase of deep learning, each of data stored in an archive is repeatedly accessed. If such data are processed on the GPU, a data compression method optimized for GPU decompression should be used. Thus, our goal is to present a novel lossless data compression method with better compression ratio and high GPU decompression speed.

1.2. Lossless data compression and related work

As usual, we assume that an input data to be compressed is a string of 8-bit characters taking value in range [0,255]. The task of data compression is to convert it into a string of codes and that of data decompression is inverse-conversion. One of the simplest lossless compression methods is run-length encoding, in which each run of the same character is replaced by a code (character,count). For example, AAAABBBCC is encoded to (A,4)(B,3)(C,2) by run-length encoding. This encoding is very useful if an input sequence has a lot of long runs such as line drawings. However, it does not achieve good compression ratio for images with few runs such as natural images and plain text. LZSS (Lempel-Ziv-Storer-Szymanski) [7] is a well-known dictionary-based lossless compression method, which replaces a substring appearing before by code (offset, length). LZSS decompression is performed using a buffer storing recently decoded string as a dictionary. Each code (offset,length) is decoded by retrieving the corresponding substring in the dictionary. For example, if the dictionary stores ABCDEFGH, then code (2,5) is decoded to CDEFG. LZ77 (Lempel-Ziv) [8] is a compression method very similar to LZSS. It uses code (offset, length, next unmatched character). LZW (Lempel-Ziv-Welch) [9] is a patented lossless compression method used in Unix file compression utility "compress" and in GIF image format. In LZW compression/decompression, a newly appeared substring is added to the dictionary. Codes of LZW are the indices of dictionary, each of which corresponds to an added substring.

Recent lossless data compression uses several other techniques including arithmetic encoding, Burrows-Wheeler transform(BWT)[10] and move-to-front transform(MTF). Table I summarizes lossless compression methods used these days.

Table I. Lossless data compression methods and tools

lossless compression	encoding method
LZSS, LZ77, LZW	dictionary-based encoding
gzip[11], LZH, LZMA	dictionary-based encoding, Huffman or arithmetic encoding
bzip2[12], ZZIP	BWT, MTF, Huffman-coding, run-length encoding

It is very hard to parallelize LZSS compression, because a newly decoded string is appended to the dictionary every time after a code is decoded and output. To parallelize LZSS compression, the input string is partitioned into equal-sized strips, each of which is compressed sequentially. Parallel LZSS decompression can be done for all strips in parallel, but codes in each strip are decoded sequentially one by one. Such *strip-wise parallel* LZSS decompression called *CULZSS* have been implemented

DOI: 10.1002/cpe

in a GPU [13], but it achieves very small acceleration ratio over the sequential implementation on the CPU due to low parallelism.

LZ77-based [8] data compression called *Gompresso* and the GPU implementation have been shown in [14]. The structure of compressed data of Gompresso is a sequence of codes (unmatched characters of length at most 15, offset, length). The compression ratio and the running time on the GPU are better than CULZSS. The GPU implementation is *code-wise parallel* in the sense that a thread is arranged in each code of a compressed string. Hence, it has very high parallelism and a lot of threads work in parallel. Since memory access latency of the GPU is quite large, higher parallelism can hide large memory access latency and can attain better performance. However, decoding of codes is performed sequentially in the worst case due to the code dependency.

Parallel algorithms for LZW compression and decompression have been presented [15, 16]. However, threads perform compression and decompression with strip-wise low parallelism. Quite recently, we have presented GPU implementation of LZW decompression with code-wise parallel [17]. The idea is to create a dictionary by parallel pointer traversing. Since pointer traversing is code-wise parallel, LZW decompression on the GPU is much faster than CULZSS decompression.

Encoding methods such as Huffman encoding and BWT are useful to obtain better compression ratio, but they are hard to parallelize on a GPU. In [18], a bzip2-like lossless data compression scheme and the GPU implementation have been presented, but the GPU implementation runs slower than the CPU implementation.

1.3. Our contribution

In this paper, we will present a new lossless data compression method called *ALL (Adaptive Loss-Less)* data compression, which is an extension of *LLL (Light Loss-Less)* data compression presented in the conference version [19] of this paper. LLL data compression uses *run-length coding* and *Huffman-based byte-wise coding*. In addition, ALL data compression uses *segment-wise coding* and *adaptive dictionary coding* to improve the data compression ratio and the running time for decompression. Since new techniques are applied only if they improve the performance, the performance of ALL is always better than that of LLL. ALL data compression has five types of codes of length 1, 2, or 3 bytes as follows: *single character code* (SC, 1 byte), *short run-length code* (SRL, 2 bytes), *short interval code* (SI, 2 bytes), *long run-length code* (LRL, 3 bytes), and *long interval code* (LI, 3 bytes). An SC code simply represents a 1-byte uncompressed character. SRL and LRL codes encode a run of the same character with length in ranges $[2, 16]$ and $[18, 3408]$, respectively. Also, SI and LI codes encode a substring in the dictionary of length in ranges $[2, 16]$ and $[18, 3408]$, respectively. Since longer run and substring, which tend to appear less frequently, use more bytes, we can think this encoding is Huffman-based byte-wise coding. To accelerate decompression using the GPU, we use segment-wise coding in which multiple codes use the same dictionary and they can be decoded in parallel using multiple threads in the GPU. However, segment-wise coding may deteriorate the data compression ratio. To compensate this deterioration, we use *adaptive dictionary coding*, in which the prefix of the dictionary is replaced by *a magic string*. A magic string is used to encode substrings appear frequently. It can also be used to encode random substrings which can not be compressed into a smaller sequence of codes. To clarify the performance of ALL data compression method, we have evaluated the data compression ratio and the running time using five images and five text data and compared ALL with the other lossless data compression methods implemented in the GPU, Gompresso, CULZSS, and LZW. The data compression ratio of ALL data compression is better than the others for eight data out of these 10 data. Also, our GPU implementation on GeForce GTX 1080 GPU for ALL decompression runs 84.0-231 times faster than the CPU implementation on Core i7-4790 CPU. Further, it runs 1.22-23.5 times faster than Gompresso, CULZSS, and LZW.

This paper is organized as follows. Section 2 introduces ALL encoding and shows sequential algorithms for ALL compression and decompression. We then go on to show a GPU implementation of ALL decompression in Section 3. Section 4 offers various experimental results including data

DOI: 10.1002/cpe

compression ratio, running time on the CPU and the GPU, and the SSD-GPU loading time. Section 5 concludes our work.

2. ALL: ADAPTIVE LOSS-LESS DATA COMPRESSION

The main purpose of this section is to present *ALL (Adaptive Loss-Less)* data compression. We first introduce outline of ALL coding, and then show the details. After that, we briefly show sequential ALL compression and decompression.

2.1. Outline of ALL coding

ALL coding includes several data compression techniques: *run-length coding*, *segment-wise coding*, *adaptive dictionary coding*, and *Huffman-based byte-wise coding*.

In *run-length coding*, a substring with the same character is encoded in a run-length code with a pair (character, length). For example, run-length codes (A,4)(B,3)(C,2) are decoded to AAAABBBCC. ALL codes include run-length codes. The run-length coding achieves good compression ratio if the input character has many long runs.

The sliding window compression technique is used in LZSS compression. It uses a dictionary buffer of a fixed size and an interval code with a pair (offset,length). For example, suppose that a dictionary buffer of size 8 is used. A compressed codes ABCDEFGHI(1,4)(3,3)B(0,3) is decoded to ABCDEFGHICDEFICDBCDE. The decoding operations for interval codes (1,4), (3,3), and (0,3) are performed as follows. First, when (1,4) is decoded, the dictionary buffer stores the latest decoded 8 characters BCDEFGHI. Thus, (1,4) is decoded to a substring of length 4 starting from offset 1 of the dictionary buffer, that is, CDEF. After that, the dictionary buffer has 8 characters FGHICDEF and interval code (3,3) is decoded to ICD. Similarly, interval code (0,3) is decoded to CDE, because the dictionary buffer is CDEFICDB. The sliding window compression works efficiently if the same substring appears two or more times in close positions. However, it is very hard to parallelize decompression by the sliding window. Since the decoded string of an interval code can be obtained after the dictionary buffer is determined, decoding operations must be performed one by one.

To parallelize decompression for interval codes, we use *segment-wise coding*. In this coding, multiple interval codes use the same dictionary. We show how ABCDEFGHI(1,4)I(1,2)B(1,3) is decoded to ABCDEFGHICDEFICDBCDE. We assume that three interval codes (1,4), (1,2), and (1,3) are in the same segment and use the same dictionary buffer BCDEFGHI. They are decoded to CDEF, CD, and CDE in parallel. In ALL data compression, a group of 16-32 codes called *a segment* uses the same dictionary, which includes the latest 4096 uncompressed characters of the first code in the segment.

In the sliding window compression technique, each code is decoded using the latest decoded substring as a dictionary. On the other hand, in segment-wise coding, multiple codes are decoded using the same dictionary. Thus, the last code in a segment uses a distant substring as a dictionary and the data compression ratio may be deteriorated, because it is likely that the same substring appears again with higher probability in a closer position. To compensate this deterioration, we use *adaptive dictionary coding*. We replace the prefix of the dictionary buffer by *a magic string*, which is defined for each segment of codes. For example, for a compressed string ABCDEFGH(0,4)(4,3)(0,3)(1,3), a magic string IJKL is preserved for a segment of interval codes (0,4)(4,3)(0,3) and (1,3). When they are decoded, the latest 8 characters ABCDEFGH stored in a dictionary are overwritten by the magic string IJKL from the left. We use the resulting string IJKLEFGH as a dictionary. Interval codes (0,4), (4,3), (0,3), and (1,3) are decoded to IJKL, EFG, IJK, and JKL, and we obtain a decompressed string ABCDEFGHIJKLEFGIJKJKL. A magic string should include frequently appeared substrings to achieve better compression ratio.

A magic string can also be used to encode a random string with high entropy, which there is no way to compress in smaller size. Usually, the total size of codes for a random string may be larger than the input uncompressed data due to the overhead of coding. Magic strings can minimize the overhead. An interval code in ALL coding supports length up to 3408. So, a random string of length

DOI: 10.1002/cpe

3408 can be encoded using a magic string of length 3408 and only one interval code (0,3408). Thus, the overhead of the random string of length 3408 is only one additional interval code.

In many compression methods, Huffman coding is used to minimize the average code length. More specifically, fewer bits are assigned to frequently used codes. Since the number of bits in each code of Huffman coding varies and each code is identified by reading bits in a code one by one, decoding operation is very hard to parallelize. Hence, it is not possible to identify the boundary of every code in parallel. Thus, we simply use *Huffman-based byte-wise coding* which enables us to identify every code in parallel. ALL coding uses 1-byte and 2-byte words as atomic elements of coding which constitutes one of the three codes: 1-byte code (1-byte word), 2-byte code (2-byte word), and 3-byte code (2-byte word plus 1-byte word). Basically, a 1-byte code simply encodes a 1-byte character. A 2-byte code is used to represent length from 2 to 16. A 3-byte code represents length from 18 to 3408. Intuitively, longer codes are appeared less frequently and longer strings are encoded in them, byte-wise Huffman coding makes sense.

2.2. ALL codes

In this subsection, we define ALL codes in detail. ALL compresses a string of 1-byte characters into ALL codes, each of which consists of one or two words. It uses two types of words: *1-byte word* and *2-byte word*. A 1-byte word simply represents *8-bit parameter* c, which can be a 1-byte character or 8-bit length. A 2-byte (16-bit) word stores *12-bit parameter* t and *4-bit parameter* l. One or two consecutive words constitute *an ALL code*, in which a sequence of one or more 1-byte characters are encoded. More specifically, an ALL code is a 1-byte word, a 2-byte word, or a 1-byte word plus a 2-byte word as follows:

1-byte code a 1-byte word
If the previous word is not a 2-byte word of a 3-byte code defined below, a 1-byte word is *a 1-byte code* with length $L = 1$.

2-byte code a 2-byte word
If 4-bit parameter l of the 2-byte word is not 15 ($= 1111$ in binary) then it is *a 2-byte code* with length $L = l + 2$. Clearly, $0 \leq l \leq 14$ and so $2 \leq L \leq 16$.

3-byte code a 2-byte word plus a 1-byte word
If parameter l of the 2-byte word is 15 then it is *a 3-byte code* with length L such that $L = c + 18$ if $c + 18 \leq 64$, that is, $c \leq 46$, where c is the parameter of the 1-byte word. If $c \geq 47$ then, $L = 16 \cdot (c - 47) + 80$, that is, $L = 16c - 672$. Since $0 \leq c \leq 255$, length L can be $18, 19, 20, \ldots, 64$ and $80, 96, 112, \ldots, 3408$.

Note that both 2-byte codes and 3-byte codes do not support length 17, because a 2-byte code and a 1-byte code combined can encode a string of length 17.

ALL data compression uses 3 types of codes as follows:

Single Character Code (1-byte code): A 1-byte code simply represents a 1-byte character.

Run-Length Code (2-byte/3-byte code): If parameter t of the 2-byte word in the code is 4095 ($= 111111111111$ in binary), then it is a Run-Length Code representing a run of length L with the previous character.

Interval Code (2-byte/3-byte codes): If t of the 2-byte word is not 4095, then it is an Interval Code representing interval $[t, t + L - 1]$ of length L in the dictionary.

The reader should refer Table II that summarizes five types of ALL codes.

2.3. Adaptive Dictionary

A sequence of ALL codes is partitioned into *segments* of 32 words each. Since each code has 1 or 2 words, each segment involves 16-32 codes. The same dictionary is used for all codes of a segment. A dictionary consists of *a previous string* and *a magic string*. The previous string is a

DOI: 10.1002/cpe

Table II. ALL codes: p is the previous character and $d_0 d_1 \cdots d_{4095}$ are 4096 characters in the dictionary

Codes	words	length L	string to be deocded
SC Single Character	c	1	c
SRL Short Run-Length	111111111111 | l	$l+2$	$pp\cdots p$
SI Short Interval	t | l		$d_t d_{t+1} \cdots d_{t+L-1}$
LRL Long Run-Length	111111111111 | 1111 | c	$c+18$ if $c \leq 46$ $16c-672$ if $c \geq 47$	$pp\cdots p$
LI Long Interval	t | 1111 | c		$d_t d_{t+1} \cdots d_{t+L-1}$

string of uncompressed 4096 characters obtained by decoding codes in previous segments. If it has less than 4096 characters, then it is right justified and 0's are filled to obtain a string of length 4096. In particular, the previous string of the first segment is a run of 4096 0's, because the first segment has no "previous string." Let $p_0 p_1 \cdots p_{4095}$ be the previous string of 4096 characters, and let $w_0 w_1 \cdots w_{v-1}$ be the magic string of v characters. The dictionary can be obtained by overwriting the magic string from the beginning of the previous string, that is, the dictionary is $w_0 w_1 \cdots w_{v-1} p_v p_{v+1} \cdots p_{4095}$. A magic string is a sequence of 8-bit characters determined in the process of compression to attain better compression ratio. Usually, it includes substrings appearing frequently in an uncompressed input.

Figure 1 shows an example of ALL codes and the decoded string. In the figure, we assume that the length of a previous string is 32 and a magic string of length 4 is used. Thus, the first 4 characters of the dictionary is the magic string. The remaining characters of the dictionary are 28 characters of the previous string. Five ALL codes in the figure are decoded as follows.

1. SI code with offset 1 and length $2+1$ is decoded to "wxy."
2. SC code with character "z" is decoded to "z."
3. SI code with offset 0 and length $2+1$ is decoded to "vwx."
4. LI code with offset 7 and length $0+18$ is decoded to "HIJKLMNOPQRSTUVWXY."
5. SRL code with length $2+2$ is decoded to "YYYY."

Thus, we can confirm that these codes are decoded into the uncompressed string as shown in the figure.

2.4. Data block: encoded data for a strip

An input uncompressed string to be compressed is partitioned into *strips* of 65536 bytes each. Each strip is compressed using ALL codes independently. *A data block* contains ALL codes with additional parameters necessary to decode a strip. We will show the data structure of a data block.

Let $a_0 a_1 \cdots a_{m-1}$ denote the words for ALL codes in a strip, where m is the number of words in this strip. We use an array of *word identifiers* $b_0 b_1 \cdots b_{m-1}$ such that $b_i = 0$ $(0 \leq i \leq m-1)$ if a_i is a 1-byte word and $b_i = 1$ if a_i is a 2-byte word. Clearly, the strip has $k = \lceil \frac{m}{32} \rceil$ segments. Magic strings for the k segments are concatenated and stored as a single magic string. To identify each magic string, we use *magic identifiers* $q_0 q_1 \cdots q_{k-1}$ such that $q_i = 0$ $(0 \leq i \leq k-1)$ if the length of the magic string of the i-th segment is 0 and $q_i = 1$ if it is not 0. An array of 12-bit integers is used to specify the length of each non-zero length magic string. If the value stored in a 12-bit integer for i-th magic string is l_i, then this magic string has $l_i + 1$ characters. Since the length of each magic string is in range [1,4096], a 12-bit integer field is sufficient to store the length of a magic string.

Figure 2 illustrates a format of data block. The format has *word counts* (16 bits; the total number of words minus 1), *word byte counts* (16 bits; the total size of words in bytes minus 1), *magic string counts* (11 bits; the number of magic strings) and *predictor option flag* (1 bit). Since the total number of words and the total size of words are in the range [1,65536], their values minus 1 are stored in 16

DOI: 10.1002/cpe

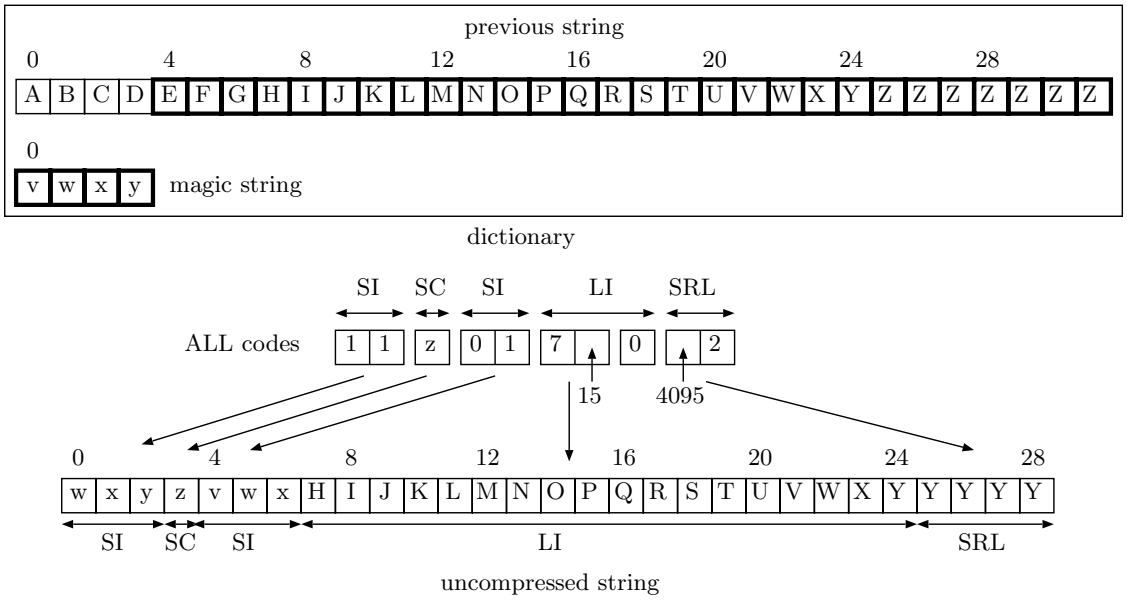

Figure 1. An example of ALL codes and the decoded string

bits each. Also, the number k of strips is less than $\frac{65536}{32} = 2048$, and the number of magic strings is stored in 11 bits. Let $x_0x_1 \cdots x_{65535}$ denote the 65536 characters of an uncompressed strip. If the predictor option flag is set, then $y_0y_1 \cdots y_{65535}$ such that $y_0 = x_0$ and $y_i = (x_i - x_{i-1}) \bmod 256$ $(1 \leq i \leq 65535)$ are used for ALL data compression. Note that $x_i = (y_0 + y_1 + \cdots + y_i) \bmod 256$ for all i. Thus, after $y_0y_1 \cdots y_{65535}$ is obtained by ALL decompression, an original sequence $x_0x_1 \cdots x_{65535}$ can be computed by the prefix-sums for $y_0y_1 \cdots y_{65535}$. Usually, the compression ratio may be better for image data if the predictor is used. On the other hand, the predictor should not be used for most text data.

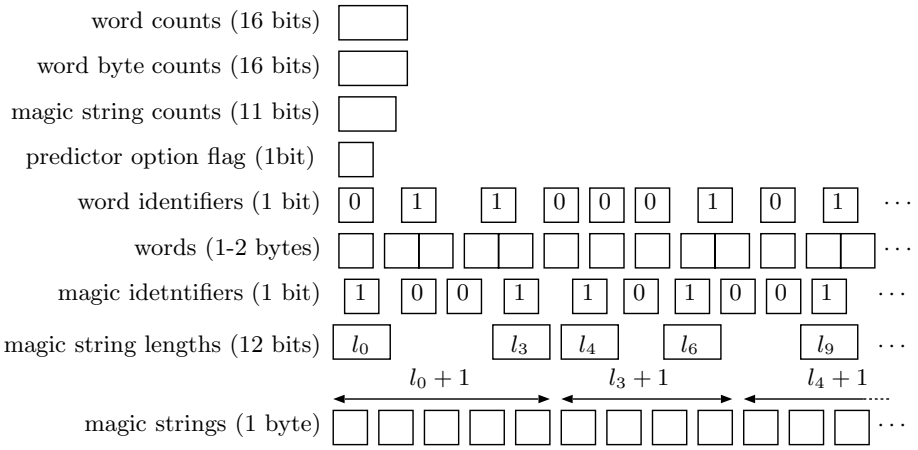

Figure 2. A data block for a strip of 65536 bytes

2.5. ALL file format

ALL file format begins with *the header* followed by *the body*. The body simply contains all data blocks. The header has *strip count* field storing strip count minus 1. It also has an array of *strip*

DOI: 10.1002/cpe

bytes each of which stores the size of the data block for each strip in bytes minus 1. Since a strip has 65536 bytes, it makes no sense that the data block has more than 65536 bytes. Thus, if the value of a strip byte is 65535, then the data block stores uncompressed 65536 characters of a strip as they are without using ALL codes.

Figure 3 illustrates data stored in the header. We assume that the strip count filed has 32 bits and thus the maximum number of strips is 2^{32}. Since each strip encodes $65536 = 2^{16}$ bytes, a single ALL file can encode up to 2^{48} =256T-byte input data in theory.

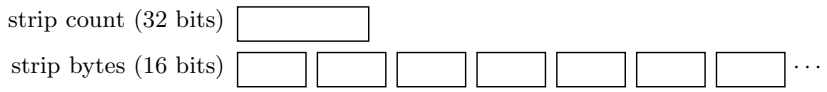

Figure 3. ALL file header

2.6. *Compression and decompression algorithms*

We briefly show sequential algorithms for ALL data compression and decompression. Due to the stringent page limitation, we omit the details, because our main purpose is to show ALL decompression using a GPU.

For sequential ALL compression, we will show how codes of a segment in a strip are generated. We first generate codes for a current uncompressed string in a strip using a previous string of length 4096 as a dictionary. Note that this dictionary has no magic string. This can be done in the same way as LZSS compression [7]. More specifically, we find the longest match and the longest run of the prefix of a current uncompressed string. We select the longest of the two. This procedure is repeated until codes of 32 words are generated. We compute the current compression ratio,

$$\frac{\text{the current total size of generated words and magic strings}}{\text{the current total size of encoded uncompressed characters}}$$

for later reference. After that, we find a magic string as follows. We pick following codes from generated codes:

- the length of codes is less than 3, and
- two or more such codes are consecutive.

We replace such consecutive codes by one interval code using a magic string. For this purpose, we compute a common superstring that contains all consecutive codes as a substring. For example, for strings of consecutive codes BAA , AAB , BBA , ABA , and ABB, a string ABAABBA is a common superstring, because it contains every string as a substring. To achieve better compression ratio, we should use the shortest common superstring as a magic string. However, the problem of finding the shortest common superstring is NP-complete [20] and it is not realistic to solve this problem for compression. Hence, we use a greedy approximation algorithm for the shortest common superstring problem shown in [21]. We use an approximation solution for the shortest common superstring as a magic string and start over generation of codes with 32 words again. In other words, we use the obtained magic string and generate codes with 32 words as before. After that, we evaluate the current compression ratio again. If this current compression ratio of codes is not better than the previous one, we do not use this magic string and use the previous codes computed before. Otherwise, we continue finding a better magic string by repeating the same procedure until no more improvement of the data compression ratio is possible.

Sequential ALL decompression can be done very easily in linear time. We show how codes of 32 words for a segment are decoded and the resulting string is written in the corresponding output buffer. We assume that previous string of the segment has been already written in the output buffer. We first need to compute the dictionary for a segment. Recall that the dictionary can be obtained by replacing the prefix of the previous string of 4096 characters by the magic string. The reader may think that copy operation with a large overhead is necessary for this replacement. However, we do

DOI: 10.1002/cpe

not have to copy the magic string. We can think that the address space of the dictionary is separated such that addresses $[0, v-1]$ are arranged in the magic string and those of $[v, 4095]$ are the previous string, where v is the length of the magic string. Thus, dictionary accesses to address $[0, v-1]$ and $[v, 4095]$ are redirect to the magic string and the previous string, respectively. Hence, each character in the dictionary can be accessed efficiently in $O(1)$ time and the decoded string of every code can be written in the output buffer in time linear to the length.

3. GPU IMPLEMENTATION FOR ALL DECOMPRESSION

We assume that the ALL compressed file is stored in the global memory of the GPU and show how they are decompressed. Our GPU implementation performs decompression and stores the resulting decompressed string of characters in the global memory of the GPU.

3.1. Parallel prefix scan on the GPU

Before showing a GPU implementation for ALL decompression, we briefly explain a well-known technique called *the parallel prefix scan* [19, 22] on the GPU, which computes the prefix-sums of integers. The parallel prefix scan is used to identify the data block of a strip, to identify words of a segment, and to compute the writing offset of each code.

Let $a_0, a_1, \ldots, a_{31}$ be 32 integers of 32 bits each. We assume that 32-bit register $A[i]$ of each thread i stores a_i $(0 \leq i \leq 31)$. The prefix sums $\hat{a}_i = a_0 + a_1 + \cdots + a_i$ for all i $(0 \leq i \leq 31)$ can be computed in 5 iterations as follows:

[Parallel prefix scan]
for $k \leftarrow 0$ to 4 do
  for $i \leftarrow 0$ to 31 do in parallel
    thread i performs $A[i] \leftarrow A[i] + A[i - 2^k]$ if $i \geq 2^k$;

Figure 4 illustrates addition performed in the parallel prefix scan. The reader should have no difficulty to confirm that each $A[i]$ stores the prefix sum $\hat{a}_i$ when this algorithm terminates.

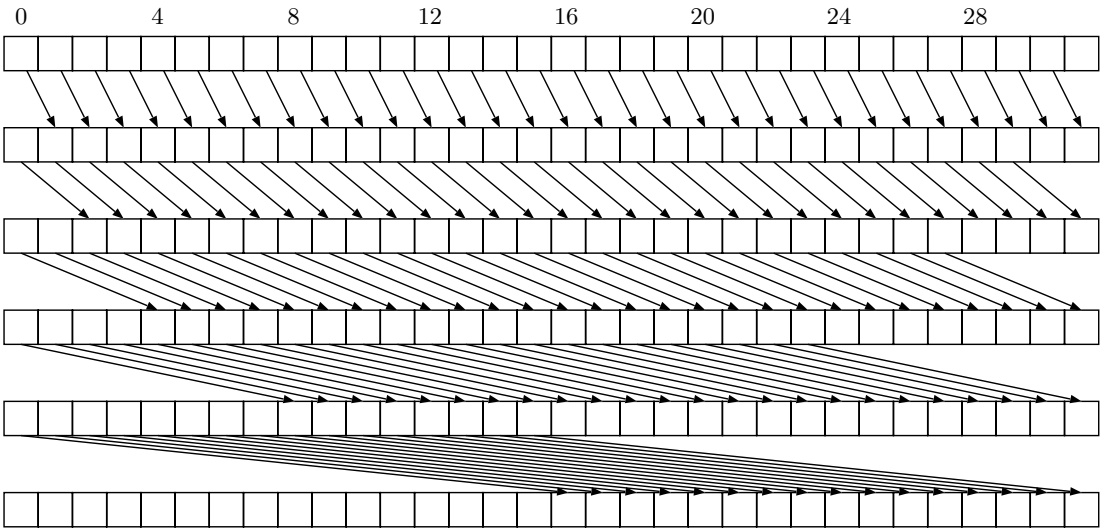

Figure 4. Parallel prefix scan

The parallel prefix scan can be implemented very efficiently using warp shuffle function [2], which exchanges the values stored in 32-bit registers of different threads in the same warp. Using warp shuffle function shfl_up(), the value stored in $A[i - 2^k]$ can be transferred to thread i with very small overhead.

DOI: 10.1002/cpe

Next, we will show that the prefix-sums of 128 8-bit integers $a_0, a_1, \ldots, a_{127}$ under modulo 256 very efficiently using the parallel prefix scan. It should be clear that they can be computed by executing the parallel prefix scan four times in 20 iterations. We show that the number of iterations can be reduced to 6 if we use SIMD addition vadd4(), which computes the addition of four 8-bit integers in a 32-bit word under modulo 256. We partition each 32-bit register $A[i]$ $(0 \leq i \leq 31)$ used in the parallel prefix scan into the four 8-bit integers such that each $A[i]$ stores a_i, a_{32+i}, a_{64+i} and a_{96+i}. Suppose that the parallel prefix scan is executed for $A[0], A[1], \ldots, A[31]$, with addition being done by SIMD addition vadd4(). Let $a[j,k]$ denote the interval sum $a_j + a_{j+1} + \cdots + a_k$. Each $A[i]$ $(0 \leq i \leq 31)$ stores local prefix-sums $a[0,i], a[32,32+i], a[64,64+i]$, and $a[96,96+i]$. In particular, $A[31]$ stores the local sums $a[0,31], a[32,63], a[64,95]$, and $a[96,127]$. By simple additions, we can obtain 32-bit word with four 8-bit integers being $0, a[0,31], a[0,63]$, and $a[0,95]$. By adding this word to every $A[i]$, it stores the prefix-sums $a[0,i], a[0,32+i], a[0,64+i]$, and $a[0,96+i]$ under modulo 256. For later reference, we call this algorithm *parallel 8-bit prefix scan*.

3.2. CUDA kernel for ALL decompression

We show how CUDA kernel for ALL decompression invokes CUDA blocks. A CUDA kernel with CUDA blocks with 64 threads, that is, two warps of 32 threads each, are invoked for decompression. A warp of 32 threads is assigned to a data block, which contains ALL codes for a strip with 4096 characters. Two warps in a CUDA block work independently. Since the compute capability of GeForce GTX 1080 is 6.1, each streaming multiprocessor can have up to 2048 resident threads and 32 resident CUDA blocks [2]. Thus, each CUDA block must have 64 or more blocks to achieve 100% occupancy. A warp of 32 threads assigned to a data block decodes the codes and writes decompressed string in the output buffer in the global memory of the GPU.

We will show how each warp is assigned to a data block. Let k be the number of strips (or data blocks). An integer variable c in the global memory is used as a counter to arrange a warp to a data block. After the CUDA kernel is invoked, the first thread of every warp calls function atomicAdd(&c,1), which increments c as an atomic operation and returns the value of c before addition [2]. If the return value i of atomicAdd satisfies $i < k$, then the warp is assigned to the i-th data block and works for decompressing the block. If $i \geq k$ then the warp terminates. Since each function call atomicAdd(&c,1) returns 0, 1, 2, ... in turn, every data block can be assigned a warp and is decompressed by it. Let $s_0, s_1, \ldots, s_{k-1}$ be the strip bytes stored in the header and $\hat{s}_0, \hat{s}_1, \ldots$ be the prefix sums such that $\hat{s}_i = (s_0 + 1) + (s_1 + 1) \cdots + (s_i + 1)$ for all i $(0 \leq i \leq k-1)$. Since each i-th data block has $s_i + 1$ bytes, it is allocated from offset $\hat{s}_{i-1}$ of the body. Thus, a warp can identify the assigned data block in the global memory if the value of $\hat{s}_{i-1}$ is computed. We will show how $\hat{s}_{i-1}$ is computed by the warp. A warp of 32 threads finds minimum j such that $\hat{s}_{i-j}$ has been already computed. This can be done by checking 32 values of p's in turn and merging the results by ballot function call of CUDA [2]. For such j, the sum $\hat{s}_{i-j} + (s_{i-j+1} + 1) + (s_{i-j+2} + 1) + \cdots + (s_{i-1} + 1)$, which is equal to $\hat{s}_{i-1}$, is computed by the parallel prefix scan (Figure 4). After that, decompression for i-th data block stored in offset $\hat{s}_{i-1}$ is performed. When decompression is completed, the first thread in the warp calls atomicAdd(&c,1) and performs decompression of strip i' again if the return value i' is smaller than k. Otherwise, the warp terminates. When all warps terminate, decompression of all data blocks is completed.

3.3. A CUDA block decompressing a data block

We will show how each data block is decompressed by 32 threads in a warp. Recall that ALL codes are segment-wise and segments are encoded in 32 words each. Thus, 32 threads decode codes for every segment one by one. We show how codes of 32 words for each segment are decoded using 32 threads in a warp and decoded string is written in the corresponding output buffer in the global memory. This decoding procedure has four stages as follows:

Stage 1 Identify all codes of 32 words for the segment.

Stage 2 Determine the code type and the reading offset t, length L, and the writing offset.

DOI: 10.1002/cpe

Stage 3 Write the decoded strings in the output buffer.

Stage 4 If the predictor option flag is 1, then the prefix-sums of the decoded strings are computed

As we have mentioned in sequential ALL decompression, copy operation for magic and previous strings is not necessary. We can think that the address space of the dictionary is separated into the magic string and the previous string.

In Stage 1, 32 threads read 32 word identifiers of a segment. Let $b_0, b_1, \ldots, b_{31}$ be these 32 word identifiers. They compute the prefix sums $\hat{b}_i = (b_0 + 1) + (b_1 + 1) + \cdots + (b_i + 1)$ for all i $(0 \leq i \leq 31)$ by the parallel prefix scan (Figure 4). Basically, each i-th thread $(0 \leq i \leq 31)$ is assigned to the i-th word stored in offset $\hat{b}_{i-1}$, and works for decoding the code corresponding to the word.

In Stage 2, each thread identifies the code type (SC, SRL, SI, LRL, or LI) and the reading offset t and length L of the assigned code. Since the offset of each code has been identified, this can be done in an obvious way using Table II. After that, the writing offset of each code is determined as follows. Since each code is decoded to a string of length L, the prefix-sums of lengths correspond to the writing offset. More specifically, let $L_0, L_1, \ldots, L_{31}$ be the lengths of codes corresponding to 32 words. If a code is a 3-byte code with two words, let the length of the second word be 0 for convenience. The 32 threads in a warp compute the prefix sums $\hat{L}_i = L_0 + L_1 + \cdots + L_i$ for all i $(0 \leq i \leq 31)$ by the parallel prefix scan (Figure 4). Clearly, the writing offset of each i-th code is $\hat{L}_{i-1}$.

In Stage 3, the decoded string of each code is written in the output buffer. This decoding operation is performed for five types of codes in turn as follows:

Step 1: Single Character Codes A thread assigned to a single character code simply copies the parameter c of it to the output buffer.

Step 2: Short Interval Codes An assigned thread simply copies the string of length L (≤ 16) in the dictionary to the output buffer.

Step 3: Long Interval Codes We use all 32 threads to copy the string of length L (≥ 18) in the dictionary to the output buffer.

Step 4: Short Run-length Codes An assigned thread repeats writing previous character p in the output buffer L (≤ 16) times.

Step 5: Long Run-length Codes We use all 32 threads to write a run of previous character p with length L (≥ 18) in the output buffer.

In Steps 3 and 5, at least 18 bytes are written in the output buffer. Hence, 32 threads are used for this writing operation and such codes are decoded in turn if a segment has multiple long codes with long length. On the other hand, Steps 2 and 4 perform writing operation for at most 16 bytes. Thus, we use a single thread for each code and perform writing operation for all such codes in parallel.

Finally, in Stage 4 if predictor option flag is 1, we compute the prefix-sums of the resulting string. For efficient prefix-sum computation, we use the parallel 8-bit prefix scan which computes the prefix-sums of 128 8-bit short integers under modulo 256.

3.4. Performance issues of the GPU implementation for ALL decompression

We will discuss several performance issues of our GPU implementation for ALL decompression.

First, we discuss memory access to the global memory, which is the bottleneck in many GPU implementations due to large latency. In our GPU implementation, 32 words in the global memory are read by 32 threads in a warp at the same time. Since these words are consecutive, memory access is coalesced. Also, the resulting string of each LRL code is written in the global memory by 32 threads. For decoding each LI code, the dictionary (or the previous string and the magic string) stored in the global memory are copied to the output buffer in the global memory by 32 threads. Clearly, these memory access operations are coalesced. Decoding SC, SI, and SRL codes involves

DOI: 10.1002/cpe

few memory accesses. Thus, these codes are decoded using one thread each. Although memory access for these codes are not coalesced, the performance is not so degraded.

To maximize the performance of a GPU implementation, we should invoke as many threads and CUDA blocks as possible. For this purpose, the occupancy must be 100%. In our GPU implementation, a CUDA block with 64 threads uses no shared memory. Each thread uses 31 32-bit registers. Each streaming multiprocessor of GeForce GTX 1080 (compute capability 6.1) has 2048 resident threads, 32 resident CUDA blocks and 64K 32-bit registers [2]. Hence, a CUDA kernel can dispatch 32 CUDA blocks in each streaming multiprocessor, because they use $32 \times 64 = 2048$ resident threads and $2048 \times 31 = 62$K 32-bit resisters. Thus, the occupancy of our implementation is 100%. Also, since GeForce GTX 1080 has 20 streaming multiprocessors, our GPU implementation can dispatch $32 \times 20 = 640$ CUDA blocks with $2048 \times 20 = 40$K threads at the same time. Since each strip is decoded using a warp of 32 threads, 1280 strips can be decoded in parallel. Thus, resident threads of GeForce GTX 1080 are fully utilized if the size of uncompressed data is $1280 \times 65536 = 80$M bytes or larger.

4. EXPERIMENTAL RESULTS

We have implemented the following four compression methods in the GPU and evaluated the performance.

ALL Our Adaptive Loss-Less compression presented in this paper

Gompresso LZ77-based [8] compression implemented in the GPU shown in [14]

CULZSS GPU implementation of LZSS [7] implemented in the GPU [13]

LZW GPU implementation of LZW [9, 23] shown in [17]

For evaluating the performance including the data compression ratio and the running time for decompression, we have used a data set in Table III. We have used five images and five text data. Figure 5 shows three JIS standard images, Crafts, Flowers, and Graph. The pixel value of every pixel of Random image is selected independently at random from $[0, 255]$. Hence, it is not possible to generate a smaller compressed data than this uncompressed Random image. Every pixel of Black image is 0 and so the compression ratio is the minimum. Thus, Random and Black images are extreme for data compression.

Table III. Data set used for evaluating the performance

Data	Size (Mbytes)	Data contents
Crafts	37.6	Standard RGB images (JIS X 9204-2004) of size 4096×3072
Flowers	37.6	
Graph	37.6	
Random	37.6	Random image of size 4096×3072
Black	37.6	Black image of size 4096×3072
wiki	1000	XML dump of a 1G-byte subset of English Wikipedia
matrix	808	Hollywood-2009 sparse matrix in CVS format
linux-2.4.5.tar	114	Source codes of Linux kernel
rectail96	121	Reuters news in XML
w3c2	109	XML documents for w3c

Table IV shows the data compression ratio, the ratio between the uncompressed size and compressed size. The best compression ratios among the four data compression methods are underlined. The data compression ratio of ALL is better than the others for almost all data. For two data, wiki and w3c2, Gompresso is better than ALL, but the difference is quite small. Since Random image cannot be compressed to smaller data, the compression ratio is larger than 1. The

DOI: 10.1002/cpe

Crafts Flowers Graphs

Figure 5. Three standard images used for experiments

overhead of compression of ALL is very small for such data. Also, for data with low entropy such as Crafts and Black images, ALL is much better than the others, because run-length encoding works efficiently for long runs in these images.

Table IV. Data compression ratio by four compression methods

Data	Size (Mbytes)	ALL	Gompresso	CULZSS	LZW
Crafts	37.6	0.644	0.781	0.863	0.808
Flowers	37.6	0.434	0.682	0.768	0.657
Graph	37.6	0.0168	0.0480	0.0710	0.0375
Random	37.6	1.0002	1.11	1.20	1.368
Black	37.6	0.00110	0.0370	0.0421	0.00602
wiki	1000	0.534	0.532	0.555	0.540
matrix	808	0.428	0.431	0.454	0.455
linux-2.4.5.tar	114	0.446	0.485	0.573	0.464
rectail96	121	0.396	0.395	0.437	0.489
w3c2	109	0.292	0.326	0.296	0.463

Table V shows the running time of decompression. The running time on GeForce GTX 1080 is evaluated for four data compression methods. We also evaluated the running time of ALL decompression on Intel i7-4790 (3.67GHz) CPU to see the acceleration ratio over the GPU. Note that, we have implemented a sequential algorithm for ALL using a single core of Intel i7-4790 CPU and we do not use multicores and hyperthread of the CPU. Since the comparison of computation powers of GPU and CPU is out of scope of this paper, we did not implement multithreaded algorithm in the CPU. However, we can say that the CPU implementation cannot be accelerated more than 8 times by multithreaded implementation using 8 hyperthreads of Intel i7-4790 CPU. The table shows the running time of ALL using the CPU and the GPU. It also shows the acceleration ratio of GPU over CPU. From the table, it achieves high acceleration ratio of 84.0-231. The high acceleration ratio implies that our implementation of ALL decompression on GPU is highly parallelized and runs very efficiently. It also shows the ratios of running time between GPU implementation of ALL and those of the others. Since all ratios are larger than 1, ALL GPU implementation always runs faster than the others.

We have evaluated the SSD-GPU loading time for three possible scenarios (Figure 6) as follows:

Scenario A: Uncompressed data in the SSD is transferred to the global memory of the GPU through the CPU.

Scenario B ALL-compressed data is transferred to the CPU, it is decompressed using the CPU, and then the resulting decompressed data is copied to the global memory of the GPU.

Scenario C ALL-compressed data is transferred to the GPU, and decompression is performed by the GPU.

Table VI shows the SSD-GPU loading time which is the time necessary to load uncompressed data in the global memory of the GPU from the SSD. The best total time of the three scenarios for each

DOI: 10.1002/cpe

Table V. The running time in milliseconds for decompression

Data	ALL			Gompresso		CULZSS		LZW	
	CPU	GPU	$\frac{\text{CPU}}{\text{GPU}}$	GPU	$\frac{\text{Gomp}}{\text{ALL}}$	GPU	$\frac{\text{CULZSS}}{\text{ALL}}$	GPU	$\frac{\text{LZW}}{\text{ALL}}$
Crafts	116.2	1.167	99.5	1.418	1.22	4.113	3.52	2.477	2.12
Flowers	92.73	0.8447	110	2.068	2.45	4.096	4.85	1.255	1.49
Graph	56.21	0.4862	116	4.213	8.67	2.343	4.82	1.531	3.15
Random	87.78	0.3792	231	3.872	10.2	8.918	23.5	3.921	10.3
Black	45.96	0.4605	99.8	5.761	12.5	3.231	7.02	3.729	8.10
wiki	2962	28.05	106	50.41	1.80	218.8	7.80	56.92	2.03
matrix	1932	19.57	98.8	27.12	1.39	87.60	4.48	25.79	1.32
linux-2.4.5.tar	273.6	2.344	117	3.387	1.44	11.29	4.82	3.030	1.29
rectail96	252.1	2.244	112	3.354	1.49	11.43	5.09	3.248	1.45
w3c2	165.8	1.974	84.0	2.803	1.42	10.92	5.53	2.536	1.28

data is underlined. Since all images have the same size, the SSD-GPU loading times for Scenario A is proportional to the uncompressed data size. In Scenario B, the time for CPU decompression dominates data transfer time. Hence, it makes no sense to use CPU decompression to load data in the GPU. The total time of Scenario C is smaller than that of Scenario A except Random image, in which the compressed data is larger than the uncompressed data. However, the difference of the total time is very small, because such data can be decompressed on the GPU very efficiently. Hence, we can say that Scenario C should be selected for loading data in the GPU regardless of data. In particular, we should use GPU decompression to load data stored in the SSD, even if the storage capacity is so large that all uncompressed data can be stored.

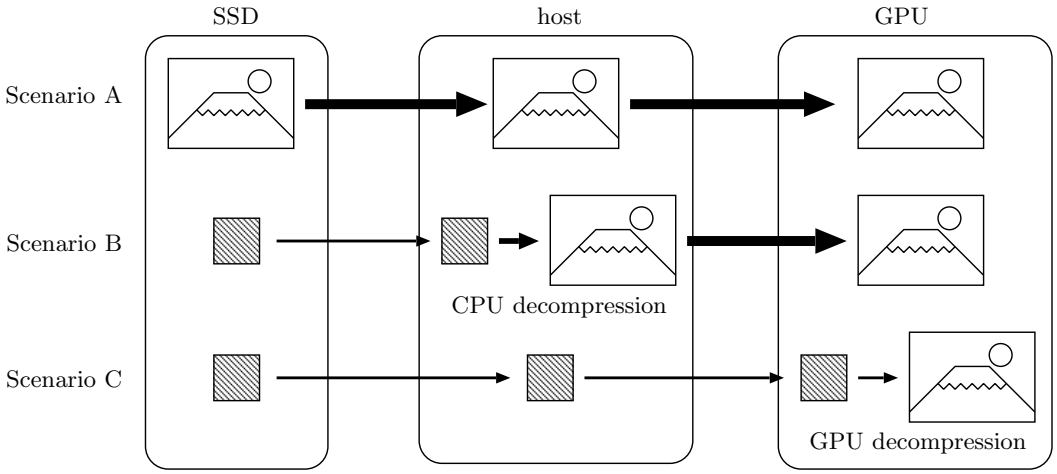

Figure 6. Three scenarios for evaluating the SSD-GPU loading time

5. CONCLUSION

In this paper, we have presented a new data compression method called ALL (Adaptive Loss-Less) compression. Although the compression ratio is comparable, the ALL decompression on the GPU is much faster than previously published Gompresso, CULZSS and LZW decompression. We also provided the SSD-GPU loading time using ALL decompression. The experimental results show that the scenario, which transfers ALL-compressed data stored in the SSD to the GPU through a host and decompresses it in the GPU, is enough fast for practical use.

DOI: 10.1002/cpe

Table VI. The SSD-GPU loading time in milliseconds using ALL decompression for three scenarios

	Crafts	Flowers	Graph	Rand.	Black	wiki	matrix	linux	rectrail	w3c2
Size (MB)	37.6	37.6	37.6	37.6	37.6	1000	808	114	121	109
Scenario A										
SSD→CPU	19.15	19.15	19.13	19.15	19.16	432.0	411.2	57.95	61.09	55.25
CPU→GPU	11.51	11.52	11.52	11.53	11.52	259.8	247.4	34.91	36.28	33.28
Total	30.66	30.67	30.65	30.68	30.68	691.8	658.6	92.86	97.38	88.53
Scenario B										
SSD→CPU	12.33	8.296	0.3210	19.14	0.02082	230.4	175.9	25.84	24.20	16.14
CPU decomp.	116.2	92.73	56.21	87.78	45.96	2962	1932	273.6	252.1	165.8
CPU→GPU	11.52	11.52	11.51	11.52	11.51	259.6	247.6	34.92	36.79	33.27
Total	140.1	112.5	68.04	118.4	57.49	3452	2356	334.3	313.1	215.2
Scenario C										
SSD→CPU	12.32	8.305	0.3230	19.16	0.02080	230.7	176.2	25.86	24.23	16.16
CPU→GPU	7.402	5.001	0.1940	11.52	0.01320	138.7	106.0	15.57	14.58	9.712
GPU decomp.	1.168	0.8447	0.4862	0.3792	0.4605	28.05	19.57	2.344	2.243	1.974
Total	20.88	14.15	1.003	31.06	0.4945	397.5	301.6	43.77	41.05	27.85

REFERENCES

1. Hwu WW. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.
2. NVIDIA Corporation. NVIDIA CUDA C programming guide version 7.5 Sept 2016.
3. Man D, Uda K, Ueyama H, Ito Y, Nakano K. Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs. *International Journal of Networking and Computing* July 2011; **1**(2):260–276.
4. NVIDIA Corporation. NVIDIA CUDA C programming guide version 7.0 Mar 2015.
5. NVIDIA Corporation. NVIDIA CUDA C best practice guide version 3.1 2010.
6. Sayood K. *Introduction to Data Compression, Fourth Edition*. Morgan Kaufmann, 2012.
7. Storer JA, Szymanski TG. Data compression via textual substitution. *Journal of the ACM* Oct 1982; **29**(4):928–951.
8. Ziv J, Lempel A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* May 1977; **23**(3):337–343.
9. Welch T. High speed data compression and decompression apparatus and method. US patent 4558302 Dec 1985.
10. Burrows M, Wheeler DJ. A block-sorting lossless data compression algorithm. *Technical Report*, DEC System Resarch Center May 1994.
11. Gzip. http://www.gzip.org/.
12. bzip2. http://www.bzip.org/.
13. Ozsoy A, Swany M. CULZSS: LZSS lossless data compression on CUDA. *Proc. International Conference on Cluster Computing*, 2011; 403 – 411.
14. Sitaridi E, Mueller R, Kaldewey T, Lohman G, Ross KA. Massively-parallel lossless data decompression. *Proc. of Internatinal Conference on Parallel Processing*, 2016; 242–247.
15. Funasaka S, Nakano K, Ito Y. Fast LZW compression using a GPU. *Proc. of International Symposium on Computing and Networking*, 2015; 303–308.
16. Klein ST, Wiseman Y. Parallel Lempel Ziv coding. *Discrete Applied Mathematics* 2005; **146**:180 – 191.
17. Funasaka S, Nakano K, Ito Y. Fully parallelized LZW decompression for CUDA-enabled GPUs. *IEICE Transactions on Information and Systems* Dec 2016; **99-D**(12):2986–2994.
18. Patel RA, Zhang Y, Mak J, Davidson A. Parallel lossless data compression on the GPU. *Proc. of Innovative Parallel Computing (InPar)*, 2012; 1–9.
19. Funasaka S, Nakano K, Ito Y. Light loss-less data compression, with GPU implementation. *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (LNCS 10048)*, 2016; 281–294.
20. Rih KJ, Ukkonen E. The shortest common supersequence problem over binary alphabet is np-complete. *Theoretical Computer Science* 1981; **16**(2):187–198.
21. Fraser CB, Irving RW. Approximation algorithms for the shortest common supersequence. *Nordic Journal of Computing* 1995; **2**(3):303–325.
22. Harris M, Sengupta S, Owens JD. Chapter 39. parallel prefix sum (scan) with CUDA. *GPU Gems 3*, Addison-Wesley, 2007.
23. Welch TA. A technique for high-performance data compression. *IEEE Computer* June 1984; **17**(6):8–19.

DOI: 10.1002/cpe