Light Loss-Less Data Compression, With GPU Implementation

Shunji Funasaka, Koji Nakano, and Yasuaki Ito

Department of Information Engineering, Hiroshima University
Kagamiyama 1-4-1, Higashihiroshima 739-8527, Japan

{funasaka,nakano,yasuaki}@cs.hiroshima-u.ac.jp

Abstract. 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 Light Loss-Less (LLL) compression. It is designed so that decompression can be highly parallelized and run very efficiently on the GPU. This makes sense for many applications in which compressed data is read and decompressed many times and decompression performed more frequently than compression. We show optimal sequential and parallel algorithms for LLL decompression and implement them to run on Core i7-4790 CPU and GeForce GTX 1080 GPU, respectively. To show the potentiality of LLL compression method, we have evaluated the running time using five images and compared with well-known compression methods LZW and LZSS. Our GPU implementation of LLL decompression runs 91.1-176 times faster than the CPU implementation. Also, the running time on the GPU of our experiments show that LLL decompression is 2.49-9.13 times faster than LZW decompression and 4.30-14.1 times faster that LZSS decompression, although their compression ratios are comparable.

Keywords: data compression, parallel algorithms, GPGPU

1 Introduction

A GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [6]. 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 [7, 11]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [12], 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 [10], since they have thousands of processor cores and very high memory bandwidth.

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* [15]. Lossy compression can generate smaller files, but some information in original files are discarded. On the other hand, lossless compression creates compressed files, from which we can obtain the exactly same original files by decompression. The main contribution of this paper is to present a novel lossless data compression method, in which decompression can be done very fast using the GPU.

Usually, data to be compressed is a sequence of 8-bit numbers (or a string of characters). LZSS (Lempel-Ziv-Storer-Szymanski) [16] is a well-known dictionary-based lossless compression method, which replaces a substring appearing before by a pair of offset and length. For example, ABCDEBCDEF is encoded into ABCDE(1,4)F, where (1,4) is a code representing a substring of length 4 from offset 1. LZSS decompression is performed using a buffer storing recently decoded substring. We can think that the buffer is a dictionary and an offset/length pair is decoded by retrieving the corresponding substring in the dictionary. For example, the dictionary stores ABCDE when (1,4) is decoded and BCDE in a dictionary is read and output. Since the dictionary is updated every time after a code is decoded and output, it is very hard to parallelize LZSS compression. To parallelize LZSS compression, the input string is partitioned into equal-sized strips, each of which is encoded sequentially using one thread. The LZSS decompression is also performed using one thread for each encoded strip. Since every strip is encoded/decoded sequentially, we call this low parallelism *strip-wise*. Strip-wise parallel LZSS compression/decompression have been implemented in a GPU [13], but it achieves very small acceleration ratio over the sequential implementation on the CPU.

LZW (Lempel-Ziv-Welch) is a patented lossless compression method [17] used in Unix file compression utility "compress" and in GIF image format. Also, LZW compression option is included in TIFF file format standard [1], which is commonly used in the area of commercial digital printing. In LZW compression/decompression, a newly appeared substring is added to the dictionary. Hence, it is very hard to parallelize them. Parallel algorithms for LZW compression and decompression have been presented [2, 8]. However, processors perform compression and decompression with strip-wise low parallelism. Quite recently, we have presented GPU implementation of LZW decompression with high parallelism [3]. This parallel algorithm is *code-wise* 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. The experimental results in [3] show that it achieves a speedup factor up to 69.4 over sequential CPU decompression. This code-wise LZW decompression on the GPU is much faster than the strip-wise LZSS decompression, although it performs complicated pointer traversing operations.

We present a simple lossless compression method called *LLL (Light Loss-Less)* data compression that can be implemented in the GPU as code-wise. Basically, it combines run-length and LZSS encoding. In the LLL compression, each strip is partitioned into several segments, say 16 segments of size 4096 bytes. The previous segment is used as a dictionary when a segment is encoded/decoded. We focus on LLL decompression in this paper, because decompression is performed more frequently than compression in many applications. For example, an image compressed and stored in a storage may be read and decompressed every time when it is necessary. Thus, compression is performed once for this image, but decompression may be performed many times. We first show a sequential LLL decompression algorithm that computes and outputs characters corresponding to every code one by one. This algorithm runs $O(n)$ time, where n is the number of output characters. Since $\Omega(n)$ time is necessary, this sequential algorithm is optimal.

Our parallel LLL decompression algorithm has 2 stages. Stage 1 computes some prefix-sums twice, to determine, for each code, reading offsets of the previous segment, writing offsets of the current segment, and the lengths of substrings to be copied. Stage 2 performs, for each code, copy operations from the previous segment to the current segment. We have evaluated the performance of this parallel algorithm using the CREW-PRAM (Concurrent Read and Exclusive Write Parallel Random Access Machine), which is a standard theoretical parallel computing model with a number of processors and the shared memory [4]. Our parallel LLL decompression algorithm runs $O(k \log m)$ time and $O(n)$ total work using $\frac{m}{\log m}$ processors on the CREW-PRAM, where m and k are the number of codes and the maximum length of all codes, respectively, and the total work is the total number of instructions executed by all processors. Since at least $\Omega(n)$ work is necessary, this parallel algorithm is work optimal.

We have implemented our parallel LLL decompression algorithm in the GPU. Since the GPU can compute the prefix-sums very efficiently, our GPU implementation for LLL decompression run much faster than those for LZSS decompression and LZW decompression. The experimental results using five images show that LLL decompression on the GPU runs 91.1-176 times faster than that on the CPU. Also, the LLL compression method achieves comparable compression ratio to the LZW and LZSS compression methods. Despite good compression ratio, LLL decompression is 2.49-9.13 times faster than LZW decompression and 4.30-14.1 times faster than LZSS decompression on the GPU.

As far as we know, there is few published work which aims to design a data compression method to be implemented in the GPU. In [9], a compression method for a sequence of sensing data has been presented. The idea is so simple that it finds the maximum value of a segment and removes unnecessary significant bits from sensing data. Hence, it does not work well for data with high dynamic range and cannot attain good compression ratio. In [14], a bzip2-like lossless data compression scheme and the GPU implementation have been presented, but it did not succeed in GPU acceleration of compression.

This paper is organized as follows. Section 2 introduces LLL encoding and shows sequential algorithm for LLL decompression. We then go on to show that LLL decompression can be done in parallel by computing prefix-sums twice and by copy operations in Section 3. We show the details of GPU implementation of LLL decompression in Section 4. Section 5 offers various experimental results including compression ratio, running time on the CPU and the GPU, the SSD-GPU loading time. Section 6 concludes our work.

2 LLL: Light Loss-Less Data Compression

The main purpose of this section is to present *LLL (Light Loss-Less)* data compression method and efficient sequential algorithms for it.

Non-dictionary Encoding

Non-dictionary has two codes, single character code and run-length code as follows:

Single Character (SC) code: A 1-byte SC code simply represents an 8-bit character .

Run-Length (RL) code: A 2-byte RC code has two fields: an 8-bit character field c and an 8-bit length field l. This code represents a run (or a sequence of the same character) with $l + 2$ characters c.

Dictionary Encoding

Dictionary encoding has five codes: single character code (1-byte word), short run-length code (2-byte word), long run-length code (2-byte word plus 1-byte word), short interval code (2-byte word), and long interval code (2-byte word plus 1-byte word). A 2-byte word has two fields: 12-bit offset field t and 4-bit length field l. Also, let c denote the value of a 1-byte word. The string corresponding a code can be determined by the following five encoding rules:

Single Character (SC) code: If a 1-byte code is not that of a long run-length or long interval code defined next, then it is an SC code, which represents an 8-bit character.

Short Run-Length (SRL) code: If the offset t of the 2-byte word is 4095 (= 111111111111 in binary) and the length l is NOT 15 (= 1111 in binary) then the 2-byte word is *short run-length code* and represents a run of length $l + 2$ with the previous character.

Long Run-Length (LRL) code: If the offset t of the 2-byte word is 4095 and the length l is 15 then a 1-byte word follows, and these two words constitute a *long run-length code*, which represents a run of length $c + 18$ with the previous character.

Short Interval (SI) code: If offset t of a 2-byte word is NOT 4095 and length l is NOT 15 then the 2-byte word is *short interval code* and represents reading offset t and length $l + 2$. The decoded string of this code is a substring of length $l + 2$ in the previous segment from offset t.

Long Interval (LI) code: If offset t of a 2-byte word is NOT 4095 and the length l is 15 then a 1-byte word must follow, and these two words constitute a *long interval code*, which represents reading offset t and length $c+18$. The decoded string of this code is a substring of length $c + 18$ in the previous segment from offset t.

Additional Rule: Two run-length codes should not be consecutive.

The reader should refer to Table 1 that summarizes five codes. Note that, one SI/SRL code and one SC code combined in 3 bytes can represent up to 17 characters. Thus, it is not necessary for a 3-byte LI/LRL code to support length 17.

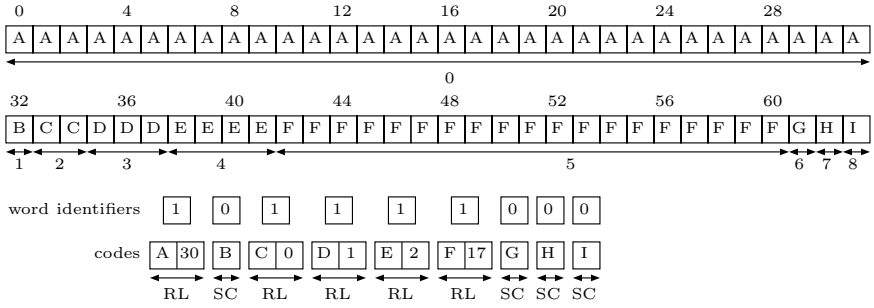
Table 1. Rules of codes: p is the previous character and $x(0)x(1)\cdots x(4095)$ is the previous segment

Codes	words	length	encoded string
Non-dictionary encoding			
SC Single Character	c	1	c
RL Run-Length	c | l	$l+2$	$cc\cdots c$
Dictionary encoding			
SC Single Character	c	1	c
SRL Short Run-Length	111111111111 | l	$l+2$	$pp\cdots p$
LRL Long Run-Length	111111111111 | 1111 | c	$c+18$	$pp\cdots p$
SI Short Interval	t | l	$l+2$	$x(t)\cdots x(t+l+1)$
LI Long Interval	t | 1111 | c	$c+18$	$x(t)\cdots x(t+c+17)$

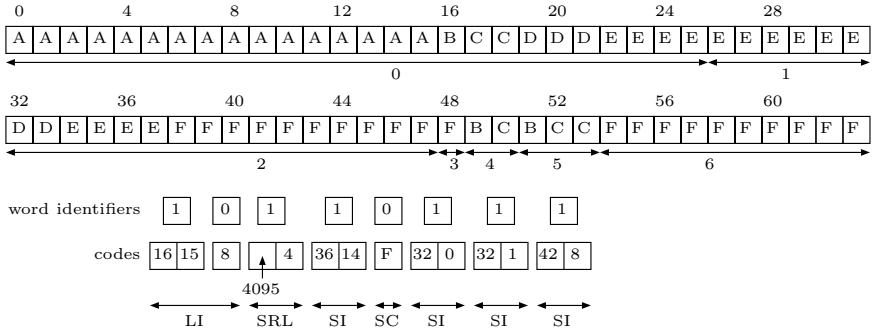
Fig. 1 shows examples of LLL-compressed data for two segments. We assume that each segment has 64 characters each although it is defined to be 4096 characters. The first segment is compressed by non-dictionary encoding. The encoded data has four RL codes and four SC codes. For example, the first RL code with character A and length 30 corresponds to run of $30+2 = 32$ A's. Next SC code with character B corresponds to one B.

The second segment in Fig. 1 is encoded using the first segment as a dictionary. The compressed data has six 2-byte words and two 1-byte words. The first 26 characters appear from offset 16 of the first segment. Hence, they are encoded using one LI code with length $8 + 18 = 26$. After that, six E's follow. Since the previous character is also E, they can be encoded using one SRL code

with length $2 + 4 = 6$. The following string of 16 characters appears in the first segment. It is encoded using one SI code with length $2 + 14 = 16$ and one SC code. Remaining characters can be partitioned into three strings appearing in the first string. Hence, they are encoded using three SI codes.



(1) The first segment and LLL-compressed data



(2) The second segment and LLL-compressed data

Fig. 1. Examples of LLL-compressed data for two segments with 64 characters each

2.1 LLL With Segment Halving

Recall that the first segment with 4096 characters is encoded using non-dictionary encoding, by which, we cannot expect good compression ratio. We introduce *the segment halving technique*, that reduces the length of a segment compressed by non-dictionary encoding.

In the segment halving technique, the first segment is partitioned into subsegments such that subsegment 0 and subsegment 1 have 512 characters, subsegment 2 has 1024 characters, and segment 3 has 2048 characters. Subsegment 0 is compressed using non-dictionary encoding. The remaining subsegments are compressed using dictionary encoding.

2.2 LLL File Format

We will show how a large data are encoded and compressed using LLL data compression. An input sequence is partitioned into segments of 4096 characters. Several, say, 8 consecutive segments constitute *a strip*. Each strip with 32K characters is encoded independently by the LLL compression method. Fig. 2 illustrates an example of LLL-compressed file format. It has *a header*, which contains tags storing several setting data such as the number of segments per strip and the number of strips. It also has *a directory*, which stores an array of addresses pointing the heads of encoded strips. The encoded strip has two blocks: the word identifier block and the word array block. The word identifier block stores all word identifiers in a strip. All word identifiers of all segments are concatenated and stored in the word identifier block. Similarly, the word arrays of all segments are concatenated and stored in the word array block. Using this file format, decompression of each strip can be done independently.

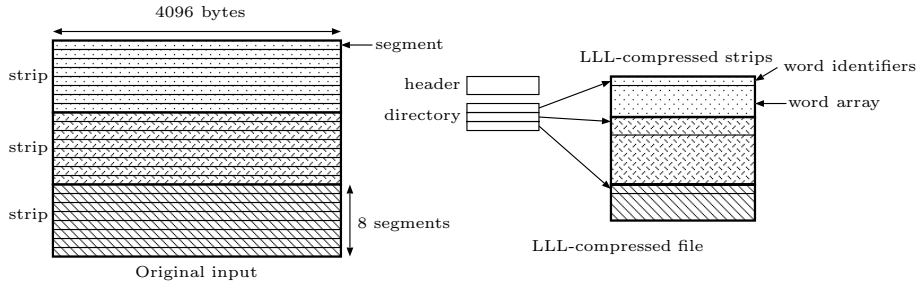


Fig. 2. LLL-compressed file format

2.3 Sequential LLL Decompression Algorithm

We show a sequential LLL decompression algorithm for dictionary encoding. Since those for non-dictionary encoding can be performed easier, we omit to describe it.

Let $x(0)x(1)\cdots x(4095)$ be a sequence of characters in the previous segment. Let $w(0)w(1)\cdots w(m-1)$ be m bits of the word identifiers and $b(0)b(1)\cdots b(m_1+2m_2-1)$ be bytes in the word array, where m, m_1, and m_2 be the numbers of words, 1-byte words, and 2-byte words, respectively. Sequential LLL decompression can be done by reading the word identifiers as follows:

[Sequential LLL decompression algorithm]
$j \leftarrow 0$; $p \leftarrow$ NULL;
for $i \leftarrow 0$ to $m-1$ do
  if($w(i) = 0$) write($b(j)$);$p \leftarrow b(j)$; $j \leftarrow j+1$;// SC code

```
else
    (t, l) ← the offset and the length fields of 2-byte word b(j)b(j + 1);
    if(l = 15)
        c ← b(j + 2);
        if(t = 4095) write(p) is executed c + 18 times; // LRL code
        else
            write(x(t)x(t + 1) · · · x(t + c + 17)); // LI code
            p = x(t + c + 17);
        j ← j + 3;
        i ← i + 1;
    else
        if(t = 4095) write(p) is executed l + 2 times; // SRL code
        else
            write(x(t)x(t + 1) · · · x(t + l + 1)); // SI code
            p = x(t + l + 1);
        j ← j + 2;
    if(t = 4095) p ← NULL;// run-length codes should not be consecutive
```

We will show theoretical analysis of the running time of sequential LLL decompression algorithm. To evaluate the running time using big-O notation, we use parameter n to denote the size of each segment. The number of write operations is m and totally n characters are written. Thus, sequential LLL decompression runs $O(n)$ time and we have,

Theorem 1. *Sequential LLL decompression algorithm for a segment with n characters runs $O(n)$ time.*

Since at least n characters are output, this sequential algorithm is time optimal.

3 Parallel LLL Decompression Algorithm

This section shows a parallel LLL decompression algorithm. We focus on how a segment compressed by dictionary encoding can be decoded. Decoding for non-dictionary encoding can be done in a similar way. Parallel prefix-sums computation is a key ingredient of LLL decompression. For later reference, we use "$\hat{}$" to denote the prefix-sums of a sequence of numbers. For example, the prefix-sums of a sequence $a(0), a(1), \ldots$ are $\hat{a}(0), \hat{a}(1), \ldots$, where $\hat{a}(i) = a(0)+a(1)+\cdots+a(i)$ for all i (≥ 0). For simplicity, let $\hat{a}(-1) = 0$.

We assume that m-bit word identifiers $w(0)w(1)\cdots w(m-1)$, word array $b(0)b(1)\cdots b(m_1+m_2-1)$, and the previous segment $x(0)x(1)\cdots x(4095)$ are given. Our goal is to compute the decoded string $y(0)y(1)\cdots y(4095)$ from these given data.

Let code-type(i) ($0 \leq i \leq m-1$) be a function returning *the code type* of the i-th word, SC, SI, LI, SRL, or LRL. It also returns NULL if the i-th word is the 1-byte word of a LI/LRL code. Also, let code-length(i) be a function returning *the code length*, or the number of characters corresponding to the code. Let

$w'(i) = w(i) + 1$ denote the number of bytes in word i. Since the i-th word $(0 \leq i \leq m-1)$ is $b(\hat{w'}(i-1))$ if 1-byte and $b(\hat{w'}(i-1))b(\hat{w'}(i-1)+1)$ if 2-byte, these functions can be computed in $O(1)$ time after the prefix-sums $\hat{w'}$ of w' are computed. Our parallel LLL decompression can be done in two stages using these functions. Let $c(i) =$ code-length(i) be the code length of the i-th word. Clearly, $c(i)$ characters must be written from $y(\hat{c}(i-1))$. If it is a run-length code, a run with $c(i)$ characters is written. If it is an interval code, $c(i)$ characters from $x(t(i))$ are read and written. Stage 1 computes the values of $t(i)$ (*read offset*), $\hat{c}(i-1)$ (*write offset*), and $c(i)$ (*code length*) for all i $(0 \leq i \leq m-1)$. Stage 2 performs reading/writing operations using the values to decode all codes. The details are spelled out as follows:

[Parallel LLL decompression algorithm]
// Stage 1: Compute $t(i)$, $c(i)$, and $\hat{c}(i)$.
Compute the prefix-sums $\hat{w'}(0)\hat{w'}(1)\cdots\hat{w'}(m-2)$ in parallel;
for $i \leftarrow 0$ to $m-1$ do in parallel
  if$(w(i) = 1)$ // 2-byte word
    $(t(i), l(i)) \leftarrow$ the offset and the length fields of $b(\hat{w'}(i-1))b(\hat{w'}(i-1)+1)$
  $c(i) \leftarrow$ code-length(i);
Compute the prefix-sums $\hat{c}(0)\hat{c}(1)\cdots\hat{c}(m-2)$ in parallel;
// Stage 2: Write the decoded string using the values of $t(i)$, $c(i)$, and $\hat{c}(i)$.
for $i \leftarrow 0$ to $m-1$ do in parallel
  if(code-type(i)=SC) $y(\hat{c}(i-1)) \leftarrow b(\hat{w'}(i-1))$;
  else if(code-type(i)=LI or SI)
    for $j \leftarrow 0$ to $c(i)-1$ do $y(\hat{c}(i-1)+j) \leftarrow x(t(i)+j)$;
for $i \leftarrow 0$ to $m-1$ do in parallel
  if(code-type(i)=LRL or SRL)
    for $j \leftarrow 0$ to $c(i)-1$ do $y(\hat{c}(i-1)+j) \leftarrow y(\hat{c}(i-1)-1)$;

We will show theoretical analysis of parallel LLL decompression algorithm. We use the CREW-PRAM (Concurrent Read and Exclusive Write-Parallel Random Access Machine), a standard theoretical model of a parallel machine with a number of processors and the shared memory [4]. Let n and k be the size of a segment and the maximum length of codes, respectively. The prefix-sums of $m-1$ numbers can be computed in $O(\log m)$ time using $\frac{m}{\log m}$ processors on the CREW-PRAM [4]. Also both code-type(i) and code-length(i) for any i can be computed in $O(1)$ time using a single processor. Thus, Stage 1 can be completed in $O(\log m)$ time using $\frac{m}{\log m}$ processors. Suppose that we use m processor and each processor i $(0 \leq i \leq m-1)$ is used to decode a code for the i-th word in Stage 2. Clearly, each processor runs at most $O(k)$ time. Also, since n characters are output, the total work of Stage 2 is $O(n)$. If we use $\frac{m}{\log m}$ processors each of which simulates $\log m$ processors, Stage 2 runs $O(k \log m)$ time and $O(n)$ work. Thus, we have,

Theorem 2. *Parallel LLL decompression algorithm runs in $O(k \log m)$ time and $O(n)$ work using $\frac{m}{\log m}$ processors on the CREW-PRAM.*

At least $\Omega(n)$ work is necessary, this parallel LLL decompression algorithm is work optimal.

4 GPU Implementation of LLL Decompression

This section shows an efficient GPU implementation of parallel LLL decompression algorithm. We assume that a compressed file is stored in the global memory of the GPU. Our goal is to write decompressed data in the global memory. Each CUDA block is assigned to a compressed strip to decode it.

4.1 LLL Decompression on the GPU

We assume that a compressed file is stored in the global memory of the GPU. Our GPU implementation decompresses it and the resulting decoded string of characters is written in the global memory. We use CUDA blocks with 128 threads each. Each CUDA block is assigned a compressed strip to decode it. Thus, the number of CUDA blocks is equal to the number of strip. We will show how a CUDA block decodes an assigned compressed strip.

A CUDA block repeats decompression of 256 words of the compressed segment. It uses the shared memory in a streaming multiprocessor as follows:

Read offsets (512 bytes): an array to store the values of 256 $t(i)$'s
Write offsets (514 bytes): an array to store the values of 257 $\hat{c}(i)$'s
Work space (6 bytes): three short integers used for prefix-sums computation
Current segment (4096 bytes): an array to store decoded characters of a segment with 4096 characters
Previous segment (4096 bytes): an array to store decoded characters of the previous segment with 4096 characters

Basically, the parallel LLL decompression algorithm is used for this decompression. Stage 1 computes read offsets and write offsets, and write them in the shared memory. Since the length can be computed from write offsets by formula $c(i) = \hat{c}(i) - \hat{c}(i-1)$, it is not necessary to write the code lengths in the shared memory. Note that, 257 values of $\hat{c}(i)$'s are necessary for computing 256 $c(i)$'s and thus we use 514 bytes for the write offsets. Stage 2 writes out decoded characters to the current segment using read and write offsets and the previous segment. If the computation of the current segment is completed, they are written out in the global memory.

We will show the details of Stage 1. We focus on the k-th ($k \geq 1$) iteration of 256-word decompression and assume the previous segment have been already computed. A CUDA block reads 256 word identifiers $w(256k), w(256k+1), \ldots, w(256k+255)$ in the global memory and computes the prefix-sums $\hat{w'}(256k)$, $\hat{w'}(256k+1)$, ..., $\hat{w'}(256k+255)$ by the prefix-sums computation [5] on the shared memory. By the values of the prefix-sums, it reads words in the word array and determine code-type(i), code-length(i)($= c(i)$), and read offset $t(i)$ for all i ($256k \leq i \leq 256k+255$). The prefix-sums $\hat{c}$ of c are computed on the shared

memory, and write offsets $\hat{c}(256k-1), \hat{c}(256k), \ldots \hat{c}(256k+255)$ are written in the shared memory. Also, read offsets $t(256k), t(256k+1), \ldots, t(256k+255)$ are written in the shared memory.

We should note that we have optimized the prefix-sums computation for w's and c's using the fact that these numbers are 16-bit unsigned short integers. The idea is to store two 16-bit unsigned short integers in one unsigned 32-bit integer and compute the sum of two pairs of 16-bit integers by one addition for 32-bit integers.

Since read offsets and write offsets are stored in the shared memory, we can execute Stage 2, which writes out decoded characters in the global memory. We use one thread assigned to a word writes out decoded characters if the code is SC, SRL, or SI. If the code is LRL or LI, then the code consists of two words and two threads are assigned. Thus, two threads are used to writes out decoded characters for LRL and LI codes in the current segment of the shared memory. Each of them writes out a half of decoded characters of the LRL/LI codes. If all 256 words are in the same segment, a CUDA block writes out all decoded characters for the 256 words. If they are separated into two or more segments, we need to perform this writing operation for each segment in turn. If all words of a segment are obtained in the current segment of the shared memory, they are written in the global memory. The pointers for the heads of the current segment and the previous segment are swapped to avoid copy operation between the current segment and the previous segment.

Let us evaluate the occupancy of a streaming multiprocessor in GeForce GTX 1080, which has 96K-byte shared memory, 64K 32-bit registers, 2048 resident threads, and 32 resident CUDA blocks. Since a CUDA block uses 9224 bytes in the shared memory, it can have up to $\lfloor \frac{98304}{9224} \rfloor = 10$ CUDA blocks from the shared memory capacity. From the compiler report, each thread uses 41 32-bit registers. Hence, $41 \times 128 \times 10 = 52480$ 32-bit registers are sufficient to arrange 10 CUDA blocks in a streaming multiprocessor at the same time, and the occupancy is $\frac{1280}{2048} = 62.5\%$, which is reasonably high to maximize the memory access throughput.

5 Experimental Results

We have used GeForce GTX 1080 GPU and Core i7-4790(3.6GHz) CPU to evaluate the performance. GeForce GTX 1080 has 20 streaming multiprocessor with 128 cores each. We have used five gray scale images to evaluate the performance. Three of them shown in Fig. 3 are converted from JIS X 9204-2004 standard color image data of size 4096×3072. We also use two gray scale images, Random and Black with the same size. Each pixel value of Random is selected from the range $[0, 255]$ independently at random. Every pixel in Black takes value 0.

Each strip has 64k pixels for LLL and LZW compressions. Thus, each image has 192 strips and 192 CUDA blocks are invoked for decompression. LLL and LZW compression uses CUDA blocks with 128 and 1024 threads each, respectively. To maximize the performance of LZSS, the number of strips must

Crafts Flowers Graph

Fig. 3. Gray scale images with 4096×3072 pixels used for experiments

be larger. Hence, we partition each image into 3072 strips for LZSS. Also, 48 CUDA blocks with 64 threads are invoked for LZSS decompression. Further, we use 7-bit offset and 7-bit length for LZSS, because CULZSS [13] also uses these parameters.

Table 2 shows the compression ratios for the five images. The size of compressed image of Random is larger than the original, and that of Black is very small. We can see that the compression ratios obtained by LZW and LLL are almost the same. Those by LZSS are slightly worse than the others.

Table 2. Compression ratios for five images using three compression methods

Images	Crafts	Flowers	Graph	Random	Black
LZSS	84.7%	80.3%	6.78%	111%	1.81%
LZW	78.3%	63.9%	3.22 %	137%	0.643 %
LLL	77.5%	65.9%	4.54%	112 %	1.21%

Table 3 shows the running time of three decompression methods on the CPU and the GPU. The running time on the CPU are not so different. On the other hand, the LLL decompression on the GPU attains higher acceleration ratio and runs faster than the other decompression methods. Actually, LLL decompression is 2.49-9.13 times faster than LZW decompression and 4.30-14.1 times faster that LZSS decompression.

Table 4 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. We have evaluated the SSD-GPU loading time for the following three scenarios:

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

Scenario B LLL-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 LLL-compressed data is transferred to the GPU, and decompression is performed by the GPU.

Table 3. The running time for decompression on the CPU and on the GPU in milliseconds and the speed-up ratios

Images		Crafts	Flowers	Graph	Random	Black
LZSS	CPU	53.0	35.9	22.2	61.5	19.1
	GPU	3.00	2.98	1.67	3.19	1.90
Speed-up	GPU/CPU	17.6	12.1	13.3	19.3	10.1
LZW	CPU	63.1	52.1	26.0	76.9	27.1
	GPU	1.81	0.912	1.09	2.00	1.23
Speed-up	GPU/CPU	34.8	57.1	23.9	38.5	22
LLL	CPU	56.7	40.6	28.3	67.6	23.8
	GPU	0.521	0.366	0.284	0.741	0.135
Speed-up	GPU/CPU	109	111	99.8	91.1	176
Speed-up	LLL/LZSS	5.76	8.14	5.88	4.30	14.1
	LLL/LZW	3.48	2.49	3.84	2.70	9.13

Since all images have the same size, the SSD-GPU loading times for Scenario A are almost the same. 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 original image. Hence, it makes sense to use GPU decompression to load data, even if the storage capacity is so large that all uncompressed data can be stored.

Table 4. The SSD-GPU loading time in milliseconds using LLL decompression for three scenarios

Images		Crafts	Flowers	Graph	Random	Black
Scenario A	SSD→CPU	6.38	6.39	6.38	6.39	6.39
	CPU→GPU	3.84	3.85	3.84	3.85	3.91
	Total	10.2	10.2	10.2	10.2	10.3
Scenario B	SSD→CPU	4.92	4.19	0.290	7.06	0.152
	CPU decompression	56.8	40.7	28.1	67.2	23.6
	CPU→GPU	3.83	3.87	3.81	3.82	3.93
	Total	65.5	48.7	32.2	78.1	27.7
Scenario C	SSD→CPU	4.92	4.18	0.289	6.99	0.150
	CPU→GPU	2.95	2.53	0.174	4.25	0.149
	GPU decompression	0.520	0.366	0.284	0.741	0.135
	Total	8.40	7.08	0.748	12.0	0.434

6 Conclusion

In this paper, we have presented a new data compression method called LLL (Light Loss-Less) compression. Although the compression ratio is comparable,

the LLL decompression on the GPU is much faster than previously published LZW decompression and LZSS decompression. We also provided the SSD-GPU loading time using LLL decompression, which shows that our GPU LLL decompression can be useful for many applications.

References

1. Adobe Developers Association: TIFF Revision 6.0 (June 1992), http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf
2. Funasaka, S., Nakano, K., Ito, Y.: Fast LZW compression using a GPU. In: Proc. of International Symposium on Computing and Networking. pp. 303–308 (Dec 2015)
3. Funasaka, S., Nakano, K., Ito, Y.: A parallel algorithm for LZW decompression, with GPU implementation. In: Proc. of International Conference on Parallel Processing and Applied Mathematics (LNCS9573). pp. 228–237. Springer (2015)
4. Gibbons, A., Rytter, W.: Efficient Parallel Algorithms. Cambridge University Press (1988)
5. Harris, M., Sengupta, S., Owens, J.D.: Chapter 39. parallel prefix sum (scan) with CUDA. In: GPU Gems 3. Addison-Wesley (2007)
6. Hwu, W.W.: GPU Computing Gems Emerald Edition. Morgan Kaufmann (2011)
7. Kasagi, A., Nakano, K., Ito, Y.: Parallel algorithms for the summed area table on the asynchronous hierarchical memory machine, with GPU implementations. In: Proc. of International Conference on Parallel Processing (ICPP). pp. 251–250 (Sept 2014)
8. Klein, S.T., Wiseman, Y.: Parallel Lempel Ziv coding. Discrete Applied Mathematics 146, 180 – 191 (2005)
9. Lok, U.W., Fan, G.W., Li, P.C.: Lossless compression with parallel decoder for improving performance of a GPU-based beamformer. In: Proc. of International Ultrasonics Symposium. pp. 561 – 564 (July 2014)
10. 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 1(2), 260–276 (July 2011)
11. Nishida, K., Ito, Y., Nakano, K.: Accelerating the dynamic programming for the matrix chain product on the GPU. In: Proc. of International Conference on Networking and Computing. pp. 320–326 (Dec 2011)
12. NVIDIA Corporation: NVIDIA CUDA C programming guide version 7.0 (Mar 2015)
13. Ozsoy, A., Swany, M.: Culzss: Lzss lossless data compression on cuda. In: Proc. International Conference on Cluster Computing. pp. 403 – 41 (Sept 2011)
14. Patel, R.A., Zhang, Y., Mak, J., Davidson, A.: Parallel lossless data compression on the GPU. In: Proc. of Innovative Parallel Computing (InPar). pp. 1–9 (May 2012)
15. Sayood, K.: Introduction to Data Compression, Fourth Edition. Morgan Kaufmann (2012)
16. Storer, J.A., Szymanski, T.G.: Data compression via textual substitution. Journal of the ACM 29(4), 928–951 (Oct 1982)
17. Welch, T.: High speed data compression and decompression apparatus and method. US patent 4558302 (Dec 1985)