PAPER *Special Section on Parallel and Distributed Computing and Networking*

Fully Parallelized LZW decompression for CUDA-enabled GPUs*

Shunji FUNASAKA[†], ***Student Member***, **Koji NAKANO**[†], ***and*** **Yasuaki ITO**[†], ***Members***

SUMMARY The main contribution of this paper is to present a work-optimal parallel algorithm for LZW decompression and to implement it in a CUDA-enabled GPU. Since sequential LZW decompression creates a dictionary table by reading codes in a compressed file one by one, it is not easy to parallelize it. We first present a work-optimal parallel LZW decompression algorithm on the CREW-PRAM (Concurrent-Read Exclusive-Write Parallel Random Access Machine), which is a standard theoretical parallel computing model with a shared memory. We then go on to present an efficient implementation of this parallel algorithm on a GPU. The experimental results show that our GPU implementation performs LZW decompression in 1.15 milliseconds for a gray scale TIFF image with 4096×3072 pixels stored in the global memory of GeForce GTX 980. On the other hand, sequential LZW decompression for the same image stored in the main memory of Intel Core i7 CPU takes 50.1 milliseconds. Thus, our parallel LZW decompression on the global memory of the GPU is 43.6 times faster than a sequential LZW decompression on the main memory of the CPU for this image. To show the applicability of our GPU implementation for LZW decompression, we evaluated the SSD-GPU data loading time for three scenarios. The experimental results show that the scenario using our LZW decompression on the GPU is faster than the others.

key words: *Data compression, big data, parallel algorithm, GPU, CUDA*

1. Introduction

A GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [1]–[3]. 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 [4], [5]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) (Compute Unified Device Architecture) [6], 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 [7], 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* [8]. Lossy compression can generate smaller files, but some information in original files are discarded. Hence, decompression of lossy compressed images does not generate files identical to the original images. On the other hand, lossless compression creates compressed files, from which we can obtain the exactly same original files by decompression. Hence, lossless compression can be used for more than images. In this paper, we focus on LZW(Lempel-Ziv-Welch) compression, which is the most well known patented lossless compression method [9] used in Unix file compression utility "compress" and in GIF image format. Also, LZW compression option is included in TIFF (Tagged Image File Format) file format standard [10], which is commonly used in the area of commercial digital printing. The LZW compression converts an input string of 8-bit numbers into a string of codes with 9-12 bits. The idea of LZW compression is to create a dictionary to map a substring of 8-bit numbers into a code. The dictionary is created by reading the input string from the beginning. The LZW compression need to check if a current substring is in the dictionary. Thus, implementation of the dictionary is not trivial. Usually, it is implemented using a hash table. On the other hand, LZW decompression checks if a current code is in the dictionary and its implementation is not difficult. Actually, it is easy to write a sequential program for LZW decompression running linear time. On the other hand, LZW compression and decompression are hard to parallelize, because they use dictionary tables created by reading input data one by one. Parallel algorithms for LZW compression and decompression have been presented [11], [12]. However, processors perform compression and decompression in *block-wise* in the sense that the input data is partitioned into blocks of size several Kbytes each and each block is processed sequentially using a single processor. In other words, multiple processors are used but each processor performs sequential LZW compression/decompression independently. Hence, such block-wise parallel algorithms have low parallelism, and they achieved a speed up factor of no more than 3. In [13], a CUDA implementation of LZW compression has been presented. But, it achieved only a speedup factor less than 2 over the CPU implementation using MATLAB. Also, several GPU implementations of dictionary based compression methods have been presented [14], [15], but they are not LZW compression. As far as we know, no parallel LZW decompression using GPUs has not been presented. In this paper, we focus on LZW decompression. LZW decompression may be

Manuscript received February 16, 2010.
Manuscript revised December 16, 2011.

[†]The author is with Department of Information Engineering, Hiroshima University

*The preliminary version of this paper has been presented at the 11th International Conference on Parallel Processing and Applied Mathematics (PPAM 2015)

DOI: 10.1587/transfun.E94.A.1

executed more frequently than compression, because each image/data is compressed and written in a file once, but it is decompressed whenever the original image/data is used. Hence, LZW decompression may be used more frequently and more important than LZW compression.

The main contribution of this paper is to present a parallel algorithm for LZW decompression and the GPU implementation. For the purpose of revealing parallelism of LZW decompression, we use the PRAM (Parallel Random Access Machine) [16]–[18], the most popular abstract parallel computer for designing parallel algorithms. It is well known that the RAM (Random Access Machine) [19] is used to evaluate the performance such as time complexity and space complexity of sequential algorithms. Similarly, the PRAM has been used as a standard theoretical parallel computing model to evaluate the performance of parallel algorithms. Thus, many researchers have been devoted to design parallel algorithms on the PRAM. The PRAM has a set of processors and a shared memory. Every processor works synchronously and can access any address of the shared memory. So, it is possible that two or more processors can access the same address at the same time. We assume that the PRAM is CREW (Concurrent Read Exclusive Write), in which multiple processors can read from the same address at the same time but cannot write a same address simultaneously. The CREW-PRAM is the most popular assumption in terms of limitation of simultaneous access to the shared memory. Since a lot of parallel algorithms have been developed on the CREW-PRAM, we can use these parallel algorithms as sub-algorithms. For example, our LZW decompression parallel algorithm uses a parallel algorithm for computing the prefix-sums of m numbers in $O(\log m)$ time using $\frac{m}{\log m}$ processors on the CREW-PRAM.

Quite surprisingly, our LZW decompression is *fully parallelized* in the sense that each processor is assigned to an input code in the compressed string of codes and converts the assigned code into the corresponding original input string of characters in parallel. The idea of our LZW decompression is to generate a pointer-character table from the input code sequence. We first show that a parallel algorithm for LZW decompression on the CREW-PRAM. More specifically, we show that LZW decompression of a string with m codes can be done in $O(L_{\max} + \log m)$ time using m processors on the CREW-PRAM, where $L_{\max}$ is the maximum length of characters assigned to a code. We also evaluate *the work* of this parallel algorithm, which is the total number of operations performed by processors. We prove that our parallel LZW decompression performs $O(n)$ work, where n is the length of decompressed string, k is the number of characters in the alphabet, and the work is the total number of instructions executed by all processors on the CREW-PRAM. Since optimal sequential LZW decompression takes at least $O(n)$ time, our parallel LZW decompression algorithm is work-optimal.

We then go on to show an implementation of this parallel algorithm in CUDA architecture. The experimental results using GeForce GTX 980 GPU and Intel Core i7-4790 (3.66GHz) CPU show that our implementation on a GPU achieves a speedup factor of 13.8-43.6 over a single CPU. For example, LZW-compressed TIFF (Tagged Image File Format) image "Flowers" with 4096×3072 pixels stored in the global memory of the GPU can be decompressed in 1.15 milliseconds. Note that the data transfer time to the global memory is not included. On the other hand, sequential LZW decompression for the same TIFF image stored in the main memory of the CPU takes 50.1 milliseconds. Thus, our LZW decompression on the GPU is 43.6 times faster than that on the CPU for this image.

We focus on applications that perform some operations for images/data stored in the SSD (Solid State Drive) using the GPU. For example, in the learning process of a deep neural network using the GPU, a lot of images must be transferred to the global memory of the GPU. For this purpose, we should minimize the time to load images stored in the SSD into the global memory of the GPU. We define *the SSD-GPU data loading time* as the time necessary to load images stored in the SSD into the global memory, and evaluate this time for three scenarios as follows:

Scenario A: A Non-compressed image is stored in the SSD. They are transferred to the GPU via the host computer (CPU).

Scenario B: A LZW-compressed image is stored in the SSD. They are transferred to the host computer, and decompressed in it. After that, the resulting non-compressed images are transferred to the GPU.

Scenario C: A LZW compressed images is stored in the SSD. They are transferred to the GPU through the host computer, and decompressed in the GPU.

Our experimental results of the SSD-GPU data loading time for these scenarios show that Scenario C is the best among the three. Hence, our experiment results imply a very strong fact: Even if the SSD is enough large and compression is not necessary to store all images/data in the SSD, we should LZW-compress them and store in the SSD for the purpose of leaning process on the GPU.

This paper is organized as follows. We first review LZW compression/decompression algorithms in Section 2. We also clarify the running time of LZW decompression algorithm. We then go on to show a parallel algorithm for LZW decompression on the CREW-PRAM and prove that it is work optimal in Section 3. In Section 4, we present a GPU implementation of this parallel decompression algorithm for LZW-compressed TIFF images. Section 5 shows experimental results using five LZW-compressed TIFF images. Section 6 concludes our work.

2. LZW compression and decompression

The main purpose of this section is to review LZW compression/decompression algorithms, which are shown in Section 13 in [10]. In addition, we prove several important properties and evaluate the running time of LZW decompression algorithm.

The LZW (Lempel-Ziv & Welch) [20] compression algorithm converts an input string of characters into a string of codes using a code table that maps strings into codes. If the input is an image, characters may be 8-bit integers representing intensity levels of pixels. The algorithm reads characters in an input string one by one and adds an entry in a code table (or a dictionary). At the same time, it writes codes by looking up the code table. Let $X = x_0x_1 \cdots x_{n-1}$ be an input string of characters and $Y = y_0y_1 \cdots y_{m-1}$ be an output string of codes, where n and m are the length of input and output strings. Also, let k be the number of characters in the input alphabet. If 8-bit integers used as characters, then $k = 2^8 = 256$. We will explain the details of the algorithm using an example, in which an input is a string of $k = 4$ characters a, b, c, and d. Let C be *a code table*, which determines a mapping of a code to a string, where a code is a non-negative integer. Initially, $C(0) = a$, $C(1) = b$, $C(2) = d$, and $C(3) = d$. By procedure AddTable, a new code is assigned to a string. For example, if AddTable(cb) is executed after initialization of C, we have $C(4) = cb$. The LZW compression algorithm is described as follows:

```
[LZW compression algorithm]
1 $\Omega \leftarrow$NULL string (i.e. string of length 0);
2 for $i \leftarrow 0$ to $n-1$ do
3    if($\Omega \cdot x_i$ is in $C$) $\Omega \leftarrow \Omega \cdot x_i$;
4    else Output($C^{-1}(\Omega)$); AddTable($\Omega \cdot x_i$); $\Omega \leftarrow x_i$;
5 Output($C^{-1}(\Omega)$);
```

In this algorithm, Ω is a variable to store a string and C^{-1} is the inverse of C. For example, $C^{-1}(a) = 0$ because $C(0) = a$. Also, "$\cdot$" denotes the concatenation of strings/characters. Further, "$\Omega \cdot x_i$ is in C" is true if there exists j such that $C(j) = \Omega \cdot x_i$.

Table 1 shows how an input string *cbcbcbcda* is compressed by LZW compression algorithm. First, since $\Omega \cdot x_0 = c$ is in S, $\Omega \leftarrow c$ is performed. Next, since $\Omega \cdot x_1 = cb$ is not in S, Output($C^{-1}(c)$) and AddTable(cb) are performed. In other words, $C^{-1}(c) = 2$ is output and we have $C(4) = cb$. Also, $\Omega \leftarrow x_1 = b$ is performed. It should have no difficulty to confirm that string 214630 is output by this algorithm.

Next, we will show LZW decompression algorithm. Again, we use a code table C and initialize it as the same as LZW compression. Let $C_1(i)$ denote the first character of code i. For example $C_1(4) = c$ if $C(4) = cb$. Similarly to LZW compression, the LZW decompression algorithm reads a string Y of codes one by one and adds an entry of the code table. At the same time, it writes a string X of characters. The LZW decompression algorithm is described as follows:

```
[LZW decompression algorithm]
1 Output($C(y_0)$);
2 for $i \leftarrow 1$ to $m-1$ do
3    if($C(y_i)$ has been registered)
4       Output($C(y_i)$); AddTable($C(y_{i-1}) \cdot C_1(y_i)$);
5    else
6       Output($C(y_{i-1}) \cdot C_1(y_{i-1})$); AddTable($C(y_{i-1}) \cdot C_1(y_{i-1})$);
```

Table 2 shows the decompression process for a code string 214630. First, $C(2) = c$ is output. Since $y_1 = 1$ is in C, $C(1) = b$ is output and AddTable(cb) is performed. Hence, $C(4) = cb$ holds. Next, since $y_2 = 4$ is in C, $C(4) = cb$ is output and AddTable(bc) is performed. Thus, $C(5) = bc$ holds. Since $y_3 = 6$ is not in C, $C(y_2) \cdot C_1(y_2) = cbc$ is output and AddTable(cbc) is performed. The reader should have no difficulty to confirm that *cbcbcbcda* is output by this algorithm, and the generated code table is the same as that generated by LZW compression.

For later reference, we will prove the following lemma for LZW compression and decompression:

Lemma 1: Let k, n, m be the number of characters in input alphabet, the length of input/output string of characters, and the length of output/input string of codes for LZW compression/decompression, respectively. We have: (1) AddTable is performed $m-1$ times, (2) $m \leq n$ always holds, and (3) the total length $L(k) + L(k+1) + \cdots + L(k+m-2)$ of strings added in code table C is equal to $n+m-2$.

We will prove this lemma for the LZW compression. The readers should have no difficulty to prove for the LZW decompression.

Proof: When Output in line 4 is executed, AddTable is also performed. Further, Output in line 5 is executed once. Since Output is executed m times, AddTable is performed $m-1$ times.

When $i = 0$ in for-loop of line 2, $\Omega \cdot x_i$ $(= x_0)$ in line 3 must be in C. Also, each Output outputs one code in Y. Hence, line 4 is executed at most $n-1$ times. Since AddTable in line 4 is performed $m-1$ times, we have $m \leq n$.

LZW compression algorithm performs Output($C^{-1}(\Omega)$) and AddTable($\Omega \cdot x_i$) in line 4 at the same time. Hence, when y_i $(0 \leq i \leq m-2)$ is output by Output($C^{-1}(\Omega)$), $\Omega = C(y_i)$ and the length of $\Omega \cdot x_i$ is $L(y_i) + 1$. Thus, a string of length $L(y_i) + 1$ is added to code table C by AddTable($\Omega \cdot x_i$). Also, when Output($C^{-1}(\Omega)$) in line 5 is performed, $\Omega = x_{n-1}$ and $L(y_{m-1}) = 1$. Thus, we have,

$$\begin{aligned}\sum_{i=k}^{k+m-2} L(i) &= \sum_{i=0}^{m-2}(L(y_i)+1) \\ &= (\sum_{i=0}^{m-1} L(y_i)) - L(y_{m-1}) + m - 1 = n + m - 2.\end{aligned}$$

□

We will prove that sequential LZW decompression can be done in $O(n)$ time. We assume that code table C is implemented as an array of linked lists. The array has $k+m-1$ elements, each of which is a pointer to a linked lists. For example, if $C(7) = cbcd$ then the 7-th element of the array stores a pointer to a linked list storing $cbcd$. Recall that $C(i) = i$ for all i $(0 \leq i \leq k-1)$. Hence, it is not necessary to initialize the first k elements in C. If the value of $C(i)$ $(0 \leq i \leq k-1)$ is necessary, we can return i without accessing $C(i)$. Thus, we can omit the initializing the first k elements of C. Using the array of linked list for C, "y_i

Table 1 String stored in Ω, code table C, and output string Y for $X = cbcbcbcda$

i	0	1	2	3	4	5	6	7	8	-
x_i	c	b	c	b	c	b	c	d	a	
Ω	-	c	b	c	cb	c	cb	cbc	d	a
C	-	4 : cb	5 : bc	-	6 : cbc	-	-	7 : $cbcd$	8 : da	-
Y	-	2	1	-	4	-	-	6	3	0

Table 2 Code table C and the output string for 214630

i	0	1	2	3	4	5
y_i	2	1	4	6	3	0
C	-	4 : cb	5 : bc	6 : cbc	7 : $cbcd$	8 : da
X	c	b	cb	cbc	d	a

is in C" in line 3 can be determined in $O(1)$ time. Further, Output totally outputs n characters, and totally $n + m - 2$ characters are registered to C by AddTable. Therefore, the running time is $O(n + m - 2) = O(n)$, and we have,

Lemma 2: LZW decompression algorithm runs $O(n)$ time.

Since at least $\Omega(n)$ time is necessary to output n characters, this LZW decompression algorithm is optimal in the sense that no algorithm can perform LZW decompression faster than $O(n)$.

Implementation of LZW compression is not trivial. It is difficult to determine if "$\Omega \cdot x_i$ is in C" in $O(1)$ time. The implementation of code table C must be a hash table, and we need to take care of conflicts. Hence, LZW compression, which is out of scope of this paper, is harder than LZW decompression. Further, parallelizing LZW compression is very hard. So for, the speedup factor of GPU LZW compression over the sequential algorithm can be only 3 [12].

3. Parallel LZW decompression

This section shows our parallel algorithm for LZW decompression.

Again, let $X = x_0 x_1 \cdots x_{n-1}$ be a string of characters. We assume that characters are selected from an alphabet (or a set) with k characters $\alpha(0), \alpha(1), \ldots, \alpha(k-1)$. We use $k = 4$ characters $\alpha(0) = a$, $\alpha(1) = b$, $\alpha(2) = c$, and $\alpha(3) = d$, when we show examples as before. Let $Y = y_0 y_1 \cdots y_{m-1}$ denote the compressed string of codes obtained by the LZW compression algorithm. In the LZW compression algorithm, each of the first $m - 1$ codes $y_0, y_1, \ldots, y_{m-2}$ has a corresponding AddTable operation. Hence, the argument of code table C takes an integer from 0 to $k + m - 2$.

Before showing the parallel LZW compression algorithm, we define several notations. We define pointer table p using input string Y of codes as follows:

$$p(i) = \begin{cases} \text{NULL} & \text{if } 0 \le i \le k-1 \\ y_{i-k} & \text{if } k \le i \le k+m-1 \end{cases} \tag{1}$$

We can traverse pointer table p until we reach NULL . Let $p^0(i) = i$ and $p^{j+1}(i) = p(p^j(i))$ for all $j \ge 0$ and i. In other words, $p^j(i)$ is the code where we reach from code i in j pointer traversing operations. Let $L(i)$ be an integer satisfying $p^{L(i)}(i) = \text{NULL}$ and $p^{L(i)-1}(i) \ne \text{NULL}$. Also, let $p^*(i) = p^{L(i)-1}(i)$. Intuitively, $p^*(i)$ corresponds to the dead end from code i along pointers. Further, let $C_l(i)$ $(0 \le i \le k + m - 2)$ be a character defined as follows:

$$C_l(i) = \begin{cases} \alpha(i) & \text{if } 0 \le i \le k-1 \\ \alpha(p^*(i+1)) & \text{if } k \le k+m-2 \end{cases} \tag{2}$$

It should have no difficulty to confirm that $C_l(i)$ is the last character of $C(i)$, and $L(i)$ is the length of $C(i)$. Using C_l and p, we can define the value of $C(i)$ as follows:

$$C(i) = C_l(p^{L(i)-1}(i)) \cdot C_l(p^{L(i)-2}(i)) \cdots C_l(p^0(i)). \tag{3}$$

Table 3 shows the values of p, p^*, L, C_l, and C for $Y = 214630$.

We are now in a position to show parallel LZW decompression on the CREW-PRAM. Parallel LZW decompression can be done in two steps as follows:

Step 1 Compute L, p^*, and C_l from code string Y.
Step 2 Compute X using p, C_l and L.

In Step 1, we use k processors to initialize the values of $p(i), C_l(i)$, and $L(i)$ for each i $(0 \le i \le k - 1)$. Also, we use m processors and assign one processor to each i $(k \le i \le k + m - 1)$, which is responsible for computing the values of $L(i), p^*(i)$, and $C_l(i)$. The details of Step 1 of parallel LZW decompression algorithm are spelled out as follows:

```
[Step 1 of the parallel LZW decompression algorithm]
1  for $i \leftarrow 0$ to $k-1$ do in parallel
2      $p(i) \leftarrow$ NULL; $L(i) = 1$; $C_l(i) \leftarrow \alpha(i)$;
3  for $i \leftarrow k$ to $k+m-1$ do in parallel
4      $p(i) \leftarrow y_{i-k}$; $p^*(i) \leftarrow y_{i-k}$;
5      while($p(p^*(i)) \ne$ NULL)
6          $L(i) \leftarrow L(i)+1$; $p^*(i) \leftarrow p(p^*(i))$;
7  for $i \leftarrow k$ to $k+m-2$ do in parallel
8      $C_l(i) \leftarrow \alpha(p^*(i+1))$;
```

Step 2 of the parallel LZW decompression algorithm uses m processors to compute $C(y_0) \cdot C(y_1) \cdots C(y_{m-1})$, which is equal to $X = x_0 x_1 \cdots x_{n-1}$ as follows:
[Step 2 of the parallel LZW decompression algorithm]
Step 2-A: Compute the prefix-sums $s(0), s(1), \ldots, s(m-1)$ of $L(y_0), L(y_1), \ldots, L(y_{m-1})$ using m processors by the optimal prefix-sums algorithm on the CREW-PRAM [16].
Step 2-B: Using one processor for each i $(0 \le i \le m - 1)$,

Table 3 The values of p, p^*, l, C_l, and C for $Y = 214630$

i	0	1	2	3	4	5	6	7	8	9
$p(i)$	NULL	NULL	NULL	NULL	2	1	4	6	3	0
$p^*(i)$	-	-	-	-	2	1	2	2	3	0
$L(i)$	1	1	1	1	2	2	3	4	2	1
$C_l(i)$	a	b	c	d	b	c	c	d	a	-
$C(i)$	a	b	c	d	cb	bc	cbc	$cbcd$	da	-

$L(y_i)$ characters $C_l(p^{L(y_i)-1}(y_i)) \cdot C_l(p^{L(y_i)-2}(y_i)) \cdots C_l(p^0(y_i))(= C(y_i))$ are copied from $x_{s(i-1)}$ to $x_{s(i)-1}$ as follows:

```
1  for $i \leftarrow 0$ to $m - 1$ do in parallel
2      $y(i) \leftarrow y_i$;
3      for $j \leftarrow 1$ to $L(y_i)$ do
4          $x_{s(i)-j} \leftarrow C_l(y(i))$;
5          $y(i) \leftarrow p(y(i))$;
```

Table 4 shows the values of $L(y_i)$, $s(i)$, and $C(y_i)$ for $Y = 214630$. By concatenating $C(y_0), C(y_1), \ldots, C(y_5)$, we can confirm that $X = cbcbcbcda$ is obtained.

Table 4 The values of $L(y_i)$, $s(i)$, and $C(y_i)$ for $Y = 214630$

i	0	1	2	3	4	5
y_i	2	1	4	6	3	0
$L(y_i)$	1	1	2	3	1	1
$s(i)$	1	2	4	7	8	9
$C(y_i)$	c	b	cb	cbc	d	a

We can omit for-loop in line 1, which initializes $p(i)$, $L(i)$, $C_l(i)$ for all i $(0 \le i \le k-1)$, because $p(i) =$ NULL, $L(i) = 1$ and $C_l(i) = i$. If these elements are accessed, we can return the value without accessing these elements in an obvious way. For example, when $C_l(i)$ $(0 \le i \le k-1)$ is read, we return i without accessing $C_l(i)$.

Next, we will evaluate the computing time on the CREW-PRAM. Let $L_{\max} = \max\{L(i) \mid 0 \le i \le k+m-2\}$. Also, while-loop in line 5 is repeated at most $L(i) \le L_{\max}$ times for each i. Hence, for-loop in line 3 can be done in $O(L_{\max})$ time using m processors. Also, processor working for value i of for-loop in line 3 repeats while-loop in line 5 $L(i)$ times. Thus, the work for this task is $O(L(k)+L(k+1)+\cdots+L(k+m-2)) = O(n+m-2) = O(n)$ from Lemma 1. It is well known that the prefix-sums of m numbers can be computed in $O(\log m)$ time with $O(m)$ work using $m/\log m$ processors [16]. Hence, every $s(i)$ is computed in $O(\log m)$ time using $m/\log m$ processors. After that, every $C(y_i)$ with $L(y_i)$ characters is copied from $x_{s(i)-1}$ down to $x_{s(i-1)}$ in $O(L_{\max})$ time and $O(n)$ work using m processors. Therefore, we have

Theorem 3: Parallel LZW decompression runs $O(L_{\max} + \log m)$ time with total $O(n)$ work using m processors on the CREW-PRAM.

From Lemma 2, the work of our parallel LZW decompression is equal to the running time of optimal sequential LZW decompression, and our parallel LZW decompression is work optimal.

4. GPU implementation of LZW decompression for TIFF images

The main purpose of this section is to describe a GPU implementation of our parallel LZW decompression algorithm. We focus on the decompression of TIFF image file compressed by LZW compression.

4.1 TIFF images

We assume that a TIFF image file contains a gray scale image with 8-bit depth, that is, each pixel has intensity represented by an 8-bit unsigned integer. Since each of RGB or CMYK color planes can be handled as a gray scale image, it is obvious to modify gray scale TIFF image decompression for color image decompression. As illustrated in Figure 1, a TIFF file has *an image header* containing miscellaneous information such as ImageLength (the number of rows), ImageWidth (the number of columns), compression method, depth of pixels, etc [10]. It also has *an image directory* containing pointers to the actual image data. For LZW compression, an original 8-bit gray-scale image is partitioned into *strips*, each of which has one or several consecutive rows. The number of rows per strip is stored in the image file header with tag RowsPerStrip. Each strip is compressed independently, and stored as the image data. The image directory has pointers to the image data for all strips.

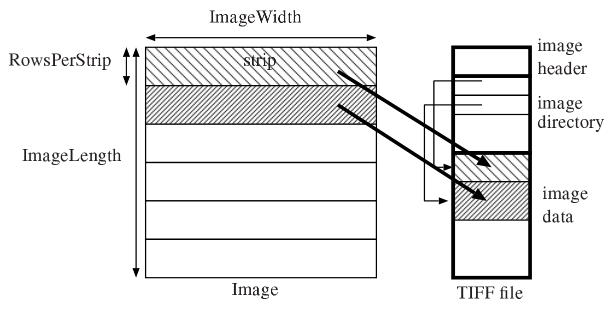


Fig. 1 An image and TIFF image file

Next, we will show how each strip is compressed. Since every pixel has an 8-bit intensity level, we can think that each strip is a string of integers in the range $[0, 255]$. Hence, codes from 0 to 255 are assigned to these integers. Codes 256 (ClearCode) and 257 (EndOfInformation) are reserved to clear the code table and to specify the end of the data. Codes from 256 to 4094 are used to store strings. As soon as entry for Code 4094 is added, ClearCode is output the code table is re-initialized. The same procedure is repeated until all pixels in a strip are converted into codes. After the code for the last pixel in a strip is output, EndOfInformation is written out. We can think that a code string for a particular strip is separated by ClearCode. We call each of them *a code segment.* Except the last one, each code segment has $4094 - 257 = 3837$ codes with $254 \times 9 + 512 \times 10 + 1024 \times 11 + 2047 \times 12 = 43234$ total bits.

We also assume that Differencing Predictor is used in the TIFF images to get better compression rate. If Predictor is used, every pixel value is replaced by the difference with the left neighbor. More specifically, if pixel values of a row are $x_0, x_1, x_2, \ldots$, then they are replaced by $x_0, x_1 - x_0, x_2 - x_1, \ldots$ to obtain better compression rate. Clearly, to obtain the original values, we need to compute the prefix sums $x'_0, x'_0 + x'_1, x'_0 + x'_1 + x'_2, \ldots$ of decompressed pixel values $x'_0, x'_1, x'_2, \ldots$ obtained by LZW decompression. Note that the subtraction and the addition is done for modulo 256. To compute the prefix-sums of 8-bit numbers efficiently, we store three 8-bit numbers in 32-bit unsigned integers, and compute the prefix sums of three lines at the same time. More specifically, suppose that we have three sequences $X' = x'_0, x'_1, \ldots$, $Y' = y'_0, y'_1, \ldots$, and $Z' = z'_0, z'_1, \ldots$ of 8-bit numbers. Each x'_i, y'_i, and z'_i are stored in one 32-bit unsigned integers such that each of them uses 9 bits. Additional 1 bit is used to handle the carry of addition. We can compute the pairwise sums $x'_i + x'_{i+1}$, $y'_i + y'_{i+1}$, and $z'_i + z'_{i+1}$, by computing the addition of two 32-bit numbers and masking out the carry bits using the bitwise AND operation. Using this idea, we can obtain the original pixel values by computing the prefix-sums of 8 bits efficiently.

4.2 GPU implementation of parallel LZW decompression

We assume that an LZW-compressed TIFF image is stored in the global memory of the GPU. We have implemented our parallel LZW decompression algorithm shown in Section 3 to decompress this image stored in the global memory.

Figure 2 illustrates of the outline of our GPU implementation. Recall that each strip consists of one or more code segments. Each block copies the first code segment of the assigned strip in the global memory to the shared memory. After that, it decompresses the code segment by our parallel LZW decompression algorithm. The resulting pixel values are copied to the global memory. The same procedure is repeated for all code segments in the assigned strip.

We will show that how a CUDA block with 1024 threads decompresses a code segment by our parallel LZW decompression algorithm. Steps 1 and 2 of it are implemented in the GPU as follows:

Step 1: Each CUDA block copies a code segment in the global memory to the shared memory by an obvious way. After that, it computes the values of each $p(i)$, $p^*(i)$, $L(i)$, and $C_l(i)$. Since the table has less than 4096 entries, 1024

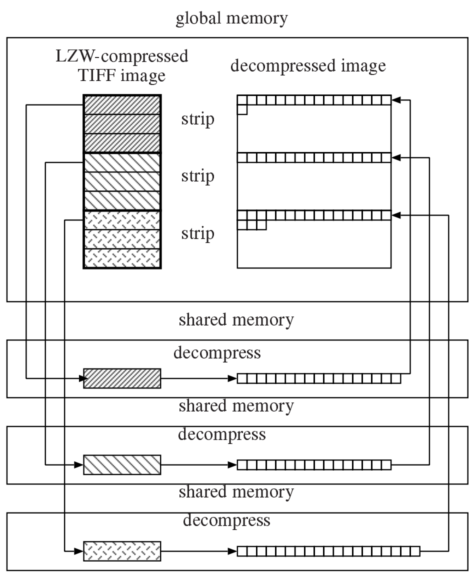


Fig. 2 GPU implementation of parallel LZW decompression

threads compute them in four iterations. In each iteration, 1024 entries of the tables are computed by 1024 threads. For example, in the first iteration, $L(i)$, $p^*(i)$, and $C_l(i)$ for every i $(0 \le i \le 1023)$ are computed. After that, these values for every i $(1024 \le i \le 2047)$ are computed. Note that, in the second iteration, it is not necessary to execute the while-loop in line 5 until $p(p^*(i)) \neq$ NULL is satisfied. Once the value of $p^*(i)$ is less than 1024, the final resulting values of $L(i)$ and $p^*(i)$ are computed using those of $L(p^*(i))$ and $p(p^*(i))$. Thus, we can terminate the while-loop as soon aSs $p^*(i) < 1024$ is satisfied.

Step 2-A: Each CUDA block of 1024 threads computes the prefix sums of $L(y_0), L(y_1), \ldots, L(y_{m-2})$ by parallel prefix-sums algorithm for GPUs shown in [21], [22].

Step 2-B: Each thread writes out $L(y_i)$ characters $C_l(p^{L(y_i)-1}(y_i)) \cdot C_l(p^{L(y_i)-2}(y_i)) \cdots C_l(p^0(y_i))$ in the global memory.

Next we will show that our implementation for GeForce GTX 980 can achieve 100% occupancy of a streaming multiprocessor. The Compute Capability of GeForce GTX 980 is 5.2, in which each streaming multiprocessor can have up to 2048 resident threads, 32 CUDA blocks, a shared memory of size 96Kbytes, and 64K 32-bit registers [6]. Since our implementation uses CUDA blocks of 1024 threads each, it is sufficient to show that two CUDA blocks of 2048 threads uses no more than 96Kbytes in a shared memory and 64K registers. The tables for $p(i)$ and $L(i)$ use 2 bytes each and $p^*(i)$ and $C_l(i)$ use 1 byte each. Since each table has 4096 elements, the tables occupy $4096 \times 6 = 24$K bytes in the shared memory. The prefix-sum computation by a CUDA block uses 32 4-byte integers in the shared memory. Hence, two CUDA blocks uses less than 49K bytes. Also, a thread in a CUDA block uses 28 registers and two CUDA blocks of 2048 threads uses 56K registers. Thus, two CUDA blocks of 2048 threads can be active in a streaming multiprocessor and the occupancy can be 100%.

5. Experimental results

We have used GeForce GTX 980, which has 16 streaming multiprocessors with 128 processor cores each to implement parallel LZW decompression algorithm. We also use Intel Core i7-4790 (3.66GHz) to evaluate the running time of sequential LZW decompression.

We have used gray scale images, Crafts, Flowers, and Graph with 4096×3072 pixels in Figure 3, which are converted from JIS X 9204-2004 standard color image data. We also use two gray scale images, Random and Black with the same size. The intensity level of each pixel of Random is selected from $[0, 255]$ independently at random. Every pixel in Black takes value 0. Thus, the size of compressed image of Random is larger than the original, and that of Black is very small. The figure also shows the compression ratio $\frac{S_C}{S_O}$, where S_O and S_C are the sizes of the original and the compressed images, respectively. They are stored in TIFF format with LZW compression option. We set RowsPerStrip= 16, and so each image has $\frac{3072}{16} = 192$ strips with $16 \times 4096 = 64$k pixels each. We invoked a CUDA kernel with 192 CUDA blocks, each of which decompresses a strip with 64k pixels.

We first evaluated the running time of GPU decompression using Core i7 CPU and GeForce GTX 980 GPU. Table 5 summarizes the running time of LZW decompression. Recall that Step 1 computes the values of $L(i)$, $p^*(i)$, and $C_l(i)$. Step 2A corresponds to the prefix-sums computation for s. Step 2B writes out string of characters to the global memory. Clearly, the running time of Step 1 and Step 2A is small if an input LZW compressed image is small, because the total number of operations is proportional to the total number of codes, that is, the size of input LZW-compressed image. On the other hand, Step 2B takes more time for images with high compression ratio, because each thread need to write out more characters. From the table, the speedup factor of GPU implementation over CPU implementation is 43.6 for image Flower.

Table 5 The running time (milliseconds) of LZW decompression

Images	GPU				CPU	Speedup
	Step 1	Step 2A	Step 2B	Total		
Crafts	0.66	0.80	0.41	1.87	61.0	32.6
Flowers	0.38	0.59	0.18	1.15	50.1	43.6
Graph	0.167	0.067	1.59	1.82	25.1	13.8
Random	1.23	1.59	0.76	3.58	76.9	21.5
Black	0.176	0.034	1.66	1.87	26.0	13.9

Suppose that we have a lot of images are stored in the SSD, and we want to perform some computation such as neural network learning for each of the these images using the GPU. For this purpose, we need to transfer each image in the SSD to the global memory of the GPU as a preprocessing step. We call the time of this processing step *SSD-GPU loading time*. We evaluate the GPU data loading time for three scenarios as follows:

Scenario A: Non-compressed images are stored in the SSD. They are transferred to the GPU via the host computer (CPU). The time for SSD→CPU and CPU→GPU is evaluated.

Scenario B: LZW compressed images are stored in the storage. They are transferred to the host computer, and decompressed in it. After that, the resulting non-compressed images are transferred to the GPU. The time for SSD→CPU, CPU decompression, and CPU→GPU is evaluated.

Scenario C: LZW compressed images are stored in the storage. They are transferred to the GPU through the host computer, and decompressed in the GPU. The time for SSD→CPU, CPU→GPU, and GPU decompression is evaluated. We also evaluated the time for "predictor", which computes the prefix-sums to obtain pixel values of original images LZW-compressed with Differencing Predictor option.

The reader should refer to Figure 4 illustrating the three scenarios.

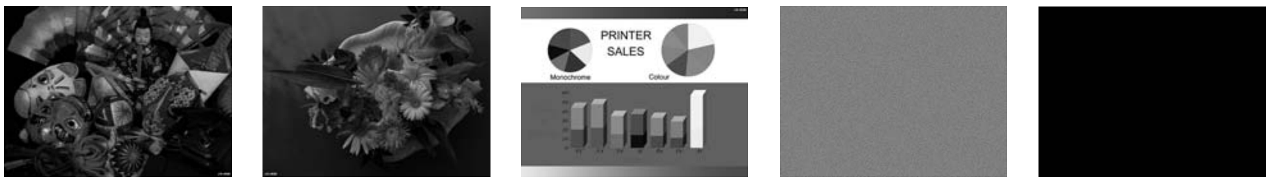
Table 6 shows the running time of three scenarios. From Table 6 (1), we can see that it takes about 10 milliseconds to copy a non-compressed image in the SSD to the global memory of the GPU. If the size of image stored in the SSD is not a big issue, and if our main goal is to accelerate the time for load a non-compressed image in the global memory of the GPU, then other scenarios make sense only if the running time is less than 10 milliseconds.

Unfortunately, from Tables 6 (2) and (3), the time necessary to load a non-compressed image is much larger than 10 milliseconds. Table 6 (2) shows the running time using our LZW decompression sequential algorithm. Table 6 (3) uses libTIFF library, a set of C functions that support the manipulation of TIFF images. We have used a libTIFF function to read a LZW-compressed TIFF image stored in SSD write decompressed data in the memory of the CPU. From these tables, we can see that our sequential LZW decompression is not slower than libTIFF. Hence, it makes sense to compare our GPU implementation and our sequential LZW decompression. Since the running time of Scenario A is much larger than Scenario B, Scenario B makes sense only if we want to reduce the total size of images stored in the SSD.

We can see that, from the Table 6 (4), the running time of Scenario C is smaller than 10 milliseconds except for image Random. Since the LZW-compressed image of Random is larger than the original image, Scenario C for it cannot be faster than 10 milliseconds. The time for image transfer from the SSD to the GPU is more than 10 milliseconds. It is quite surprising for us that Scenario C can be faster than Scenario A, and it makes sense to use our parallel LZW decompression algorithm.

6. Conclusion

In this paper, we have presented a work-optimal parallel LZW decompression algorithm on the CREW-PRAM and implemented in the GPU. The experimental results show



Crafts:59.9% Flowers:42.7% Graph:2.6% Random:136.6% Black:0.6%

Fig. 3 Three gray scale image with 4096 × 3072 pixels used for experiments

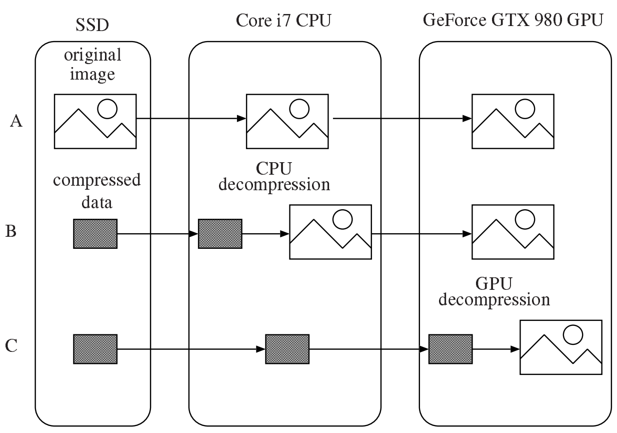


Fig. 4 Scenarios A, B, and C

Table 6 The running time (milliseconds) of each Scenario for three images

(1) Scenario A

Images	SSD→CPU	CPU→GPU	Total
Crafts	6.38	3.53	9.91
Flowers	6.43	3.52	9.95
Graph	6.55	3.50	10.05
Random	6.44	3.50	9.94
Black	6.39	3.50	9.89

(2) Scenario B (Our implementation)

Images	SSD →CPU	CPU decompression	CPU predictor	CPU →GPU	Total
Crafts	3.72	61.0	7.00	3.58	75.3
Flowers	2.55	50.1	6.99	3.56	63.2
Graph	0.19	25.1	7.00	3.56	35.8
Random	8.47	76.9	6.99	3.57	95.9
Black	0.16	26.0	7.00	3.54	36.7

(3) Scenario B (libTIFF)

Images	SSD→CPU LZW decode+predictor	CPU→GPU	Total
Crafts	82.0	3.55	85.6
Flowers	71.0	3.53	74.5
Graph	41.1	3.60	44.7
Random	91.0	3.59	94.6
Black	41.1	3.54	44.6

(4) Scenario C

Images	SSD →CPU	CPU →GPU	GPU decompression	GPU predictor	Total
Crafts	3.79	2.56	1.86	0.079	8.29
Flowers	2.47	1.75	1.13	0.080	5.43
Graph	0.17	0.16	1.83	0.080	2.24
Random	8.43	5.01	3.57	0.080	17.1
Black	0.15	0.099	1.87	0.079	2.20

that, it achieves a speedup factor up to 43.6. Also, our parallel LZW decompression in the GPU can minimize the SSD-GPU data loading time, when images stored in the SSD must be loaded in the global memory of the GPU.

References

[1] W.W. Hwu, GPU Computing Gems Emerald Edition, Morgan Kaufmann, 2011.

[2] D. Man, K. Uda, Y. Ito, and K. Nakano, "A GPU implementation of computing Euclidean distance map with efficient memory access," Proc. of International Conference on Networking and Computing, pp.68–76, Dec. 2011.

[3] Y. Takeuchi, D. Takafuji, Y. Ito, and K. Nakano, "ASCII art generation using the local exhaustive search on the GPU," Proc. of International Symposium on Computing and Networking, pp.194–200, Dec. 2013.

[4] A. Kasagi, K. Nakano, and Y. Ito, "Parallel algorithms for the summed area table on the asynchronous hierarchical memory machine, with GPU implementations," Proc. of International Conference on Parallel Processing (ICPP), pp.251–250, Sept. 2014.

[5] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the matrix chain product on the GPU," Proc. of International Conference on Networking and Computing, pp.320–326, Dec. 2011.

[6] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 7.0," Mar 2015.

[7] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs," International Journal of Networking and Computing, vol.1, no.2, pp.260–276, July 2011.

[8] K. Sayood, Introduction to Data Compression, Fourth Edition, Morgan Kaufmann, 2012.

[9] T. Welch, "High speed data compression and decompression apparatus and method." US patent 4558302, Dec. 1985.

[10] Adobe Developers Association, TIFF Revision 6.0, June 1992.

[11] S.T. Klein and Y. Wiseman, "Parallel Lempel Ziv coding," Discrete Applied Mathematics, vol.146, pp.180 – 191, 2005.

[12] S. Funasaka, K. Nakano, and Y. Ito, "Fast LZW compression using a GPU," Proc. of International Symposium on Computing and Networking, pp.303–308, Dec. 2015.

[13] K. Shyni and K.V.M. Kumar, "Lossless LZW data compression algorithm on CUDA," IOSR Journal of Computer Engineering, pp.122–127, 2013.

[14] A.L.V. Nicolaisen, Algorithms for Compression on GPUs, Ph.D. thesis, Tecnical University of Denmark, Aug. 2015.

[15] A. Ozsoy and M. Swany, "CULZSS: LZSS lossless data compression on CUDA," Proc. of International Conference on Cluster Computing, pp.403–411, Sept. 2011.

[16] A. Gibbons and W. Rytter, Efficient Parallel Algorithms, Cambridge University Press, 1988.

[17] V. Kumar, A. Grama, A. Gupta, and G. Karyapis, Introduction to

Parallel Computing: Design and Analysis of Algorithms, The Benjamin/Cumming Publishing, 1994.

[18] M.J. Quinn, Parallel Computing: Theory and Practice, McGraw-Hill, 1994.

[19] A.V. Aho, J.D. Ullman, and J.E. Hopcroft, Data Structures and Algorithms, Addison Wesley, 1983.

[20] T.A. Welch, "A technique for high-performance data compression," IEEE Computer, vol.17, no.6, pp.8–19, June 1984.

[21] M. Harris, S. Sengupta, and J.D. Owens, "Chapter 39. parallel prefix sum (scan) with CUDA," in GPU Gems 3, Addison-Wesley, 2007.

[22] K. Nakano, "Simple memory machine models for GPUs," Proc. of International Parallel and Distributed Processing Symposium Workshops, pp.788–797, May 2012.

Shunji Funasaka received the BE from the Department of Information Engineering, Hiroshima University in 2013. Currently, he is a master student at the Department of Information Engineering, Hiroshima University.

Koji Nakano received the BE, ME and Ph.D degrees from Department of Computer Science, Osaka University, Japan in 1987, 1989, and 1992 respectively. In 1992-1995, he was a Research Scientist at Advanced Research Laboratory. Hitachi Ltd. In 1995, he joined Department of Electrical and Computer Engineering, Nagoya Institute of Technology. In 2001, he moved to School of Information Science, Japan Advanced Institute of Science and Technology, where he was an associate professor. He has been a full professor at School of Engineering, Hiroshima University from 2003. He has published extensively in journals, conference proceedings, and book chapters. He served on the editorial board of journals including IEEE Transactions on Parallel and Distributed Systems, IEICE Transactions on Information and Systems, and International Journal of Foundations on Computer Science. His research interests includes image processing, hardware algorithms, GPU-based computing, FPGA-based reconfigurable computing, parallel computing, algorithms and architectures.

Yasuaki Ito received B.E. degree from Nagoya Institute of Technology (Japan), M.S. degree from Japan Advanced Institute of Science and Technology in 2003, and D.E. degree from Hiroshima University (Japan), in 2010. From 2004 to 2007 he was a Research Associate at Hiroshima University. Since 2007, Dr. Ito has been with the School of Engineering, at Hiroshima University, where he is working as an Associate Professor. His research interests include reconfigurable architectures, parallel computing, computational complexity and image processing.