2015 Third International Symposium on Computing and Networking

Fast LZW compression using a GPU

Shunji Funasaka, Koji Nakano and Yasuaki Ito
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

***Abstract*—The LZW compression is a well known patented lossless compression method used in Unix file compression utility "compress" and in GIF and TIFF image formats. It converts an input string of characters (or 8-bit unsigned integers) into a string of codes using a code table (or dictionary) that maps strings into codes. Since the code table is generated by repeatedly adding newly appeared substrings during the conversion, it is very hard to parallelize LZW compression. The main purpose of this paper is to accelerate LZW compression for TIFF images using a CUDA-enabled GPU. Our goal is to implement LZW compression algorithm using several acceleration techniques using CUDA, although it is a very hard task. Suppose that a GPU generates a resulting image generated by a computer graphics or image processing CUDA program and we want to archive it as a LZW-compressed TIFF image in the SSD connected to the host PC. We focused on the following two scenarios. Scenario 1: the resulting image is compressed using a GPU and written in the SSD through the host PC, and Scenario 2: it is transferred to the host PC, and compressed and written in the SSD using a CPU. The experimental results using NVIDIA GeForce GTX 980 and Intel Core i7 4790 show that Scenario 1 using our LZW compression implemented in a GPU is about 3 times faster than Scenario 2. From this fact, we can say that it makes sense to compress images using a GPU to archive them in the SSD.**

Keywords*—*Data compression, big data, parallel algorithms, GPU, CUDA

I. 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. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [4], 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.

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, 16-64K bytes. 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 long. The efficient usage of the shared memory and the global memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *bank conflicts* of the shared memory access and *coalescing* of the global memory access [5]–[14]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access the same memory banks at the same time, the access requests are processed in turn. Hence, to maximize the shared memory access performance, threads of CUDA should access distinct memory banks to avoid the bank conflicts of the memory accesses. To maximize the throughput between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed 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* [15]. 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. In this paper, we focus on LZW (Lempel-Ziv & Welch) [16] compression, which is one of the most well known 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 [18], which is commonly used in the area of commercial digital printing.

The LZW compression algorithm converts an input string of characters into a string of codes using a code table (or a dictionary) that maps strings into codes. In LZW compression in TIFF file format, characters are 8-bit unsigned integers representing intensity levels of gray-scale images, and codes are 12-bit unsigned integers. During the conversion, the code table is generated by adding new substrings. However, LZW compression is hard to parallelize, because they use dictionary tables created by reading input data one by one. In [19], a CUDA implementation of LZW compression has been mentioned, but the paper is very poorly written and it is not possible to understand their results. Also, several GPU implementations of some dictionary based compression methods have been presented [20], [21]. As far as we know, no paper has presented the details of LZW implementations for GPUs.

Quite recently, we have presented a GPU implementation for LZW decompression [22]. The LZW decompression algorithm converts an input string of codes into a string of characters, that is, it is the inverse of the LZW compression. Similarly, during the conversion, the code table is generated one by one. However, unlike the LZW compression, the LZW decompression can be highly parallelized. The idea of parallel

2379-1896/15 $31.00 © 2015 IEEE
DOI 10.1109/CANDAR.2015.20

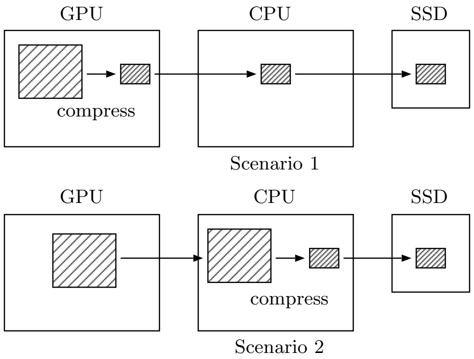


Fig. 1. Two scenarios to archive an LZW-compressed image in the SSD

LZW decompression is to generate the code table from the input string of codes in parallel. After that, the input string of codes are converted into the output string of characters. These two steps can be done in parallel using one thread to each code. On the other hand, it is not possible to generate the code table from the input string of characters in parallel. Hence, it is very hard to parallelize LZW compression.

Our idea for LZW compression is to use an idea of bulk execution of the same sequential computation that have been show in our previous papers [23]–[25]. We have proved and showed that bulk execution of a sequential algorithm can be implemented very efficiently if it is oblivious in the sense that memory access is independent of the values of the input. If each thread performs one execution of bulk executions, we can guarantee that memory access is coalesced. In TIFF LZW compression, an input image to be LZW compressed is partitioned into stripes, each of which consists of one or more rows. Since each strip is LZW compressed independently, we assign one thread to each strip for this task. Our implementation for LZW compression performs bulk execution of LZW compression using one thread for each strip. Using this idea, memory access to the input string of codes is oblivious. However, the memory access to the code table is not coalesced. In particular, reading operations for the code table are performed to random addresses. Thus, we should minimize the space for the code table to reduce the miss rate of the memory cache. Straightforward implementation for the code table needs at least 2Mbytes. We have reduced it to 32Kbytes using a hash table designed carefully.

To show the benefit of LZW compression using the GPU, we have compared two scenarios as illustrated in Figure 1. Suppose that some GPU computation generated an image in the global memory of the GPU and we want to archive it in the SSD. *Scenario 1*: a generated image is LZW compressed using the GPU, and the resulting compressed image is stored in the SSD through the CPU. *Scenario 2*: a generated image is transferred to the main memory of the CPU, and it is compressed and stored in the SSD by the CPU. We will show that Scenario 1 is approximately 3 times faster than Scenario 2. From this fact, we can say that it makes sense to compress images using a GPU to archive them in the SSD.

This paper is organized as follows. Section II reviews LZW compression algorithm. We also present that the code table can be implemented using a pointer-character table efficiently. In Section III, we present our GPU implementation of LZW compression for TIFF images. We show experimental results using GeForce GTX 980 and Core i7 4790. Finally, Section V concludes our work.

II. LZW COMPRESSION ALGORITHM

The main purpose of this section is to review LZW compression/decompression algorithms. Please see Section 13 in [18] for the details.

The LZW compression algorithm converts an input string of characters into a string of codes using a code table (or a dictionary) that maps strings into codes. If the input is an image or plain ASCII text, characters may be 8-bit unsigned integers. It reads characters in an input string one by one and adds an entry in a code table. In the same time, it writes an output string of 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. When we show examples of LZW compression, we use an input string with 4 characters a, b, c, and d, which can be mapped to 2-bit unsigned integers, 0, 1, 2, and 3. For convenience, we assume that characters a, b, c, and d take integer values 0, 1, 2, and 3, if they are used in the context of integers. Let C be a code table, which determines a mapping of a code to a string, where codes are non-negative integers. Initially, $C(0) = a$, $C(1) = b$, $C(2) = c$, and $C(3) = d$. By procedure AddTable, new code is assigned to a string. For example, if AddTable(cb) is executed after initialization of C, we have $C(4) = cb$. We also use symbol C to denote a set of codes in a code table C, that is, $C = \{C(0), C(1), \ldots\}$ if it is clear from the context.

The LZW compression algorithm finds the longest prefix Ω of the current input that is in the code table, and outputs the code of Ω. Let x be the following character of Ω in the current input. Since $\Omega{\cdot}x$ is not in the table, it is added to the code table, where "·" denotes the concatenation of strings/characters. The same procedure is repeated from x. Let $C^{-1}(\Omega)$ denote the index of C where Ω is stored. For example, when $C(3) = d$, $C^{-1}(d) = 3$. The LZW compression algorithm is formally described as follows:

```
[LZW compression algorithm]
$\Omega \leftarrow x_0$;
for $i \leftarrow 1$ to $n-1$ do
    if($\Omega \cdot x_i$ is in $C$)
        $\Omega \leftarrow \Omega \cdot x_i$;
    else
        Output($C^{-1}(\Omega)$); AddTable($\Omega \cdot x_i$); $\Omega \leftarrow x_i$;
Output($C^{-1}(\Omega)$);
```

Table I shows the compression process and the code table C for an input string *cbcbcbcda*. First, $\Omega \leftarrow x_0(= c)$ is performed. Next, since $\Omega{\cdot}x_1 = cb$ is not in C, Output($C^{-1}(c)$) and AddTable(cb) are performed. More specifically, $C^{-1}(c) = 2$ is output and we have $C(4) = cb$. Also, $\Omega \leftarrow x_1(= b)$ is performed. By repeating the same procedure, we can confirm that 214630 is output by this algorithm.

Let us discuss implementations of code table C. The following operations for a string Ω of characters must be supported for LZW compression.

TABLE I. Code table C, string stored in Ω, 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

- determine if $\Omega \cdot x_i$ is in C,
- return the value of $C^{-1}(\Omega)$,
- perform AddTable($\Omega \cdot x_i$).

A straightforward implementation of a code table C, which uses an array such that each i-th ($i \geq 0$) element stores $C(i)$, is not efficient. All values of $C(i)$ may be accessed to compute $C^{-1}(\Omega)$. We may use an associative array with keys $C(i)$ and values i, which can be implemented by a balanced binary tree or a hash table. However, these operations take more than $O(|\Omega|)$ time. If the compression ratio is high, Ω may be a long string. Hence, it is not a good idea to use a conventional associative array to implement C.

We will use a pointer-character table shown in Table II to implement a code table C. In the pointer-character table, a pointer $p(j)$ and a character $c(j)$ are stored for each code j. Also, a back-pointer $q(j,x)$ for every code j and character x is used. Back-pointer table q can be implemented using an associative array. We will discuss implementations of a back-pointer later. We can obtain a string $C(j)$ by traversing p until we reach NULL. More specifically, $C(j)$ can be obtained from p and c by the following definition:

$$\begin{aligned} C(j) &= c(j) \quad \text{if } p(j) = \text{NULL} \\ &= C(p(j)) \cdot c(j) \quad \text{otherwise.} \end{aligned}$$

For example, in Table II, we have $C(6) = C(4) \cdot c = C(2) \cdot bc = cbc$. A back-pointer $q(j,x)$ takes value k if $p(k) = j$ and $c(k) = x$. If there exists no k such that $p(k) = j$, then $q(j,k) = \text{NULL}$. It is used to perform the three operations above efficiently.

We implement procedure AddTable($\Omega \cdot x_i$) for code table C as a procedure AddTable(j, x_i) for the pointer-code table. If AddTable(j, x_i) is performed, a new available entry k with $p(k) = j$ and $c(k) = x_i$ is added to the pointer-character table. Also, the value k is written in $q(j, x_i)$. Using the pointer-character table, we can rewrite LZW compression algorithm as follows:

```
[LZW compression algorithm]
$j \leftarrow c^{-1}(x_0)$;
for $i \leftarrow 1$ to $n-1$ do
    if($q(j, x_i) \neq$ NULL)
        $j \leftarrow q(j, x_i)$;
    else
        Output($j$); AddTable($j, x_i$); $j \leftarrow x_i$;
Output($j$);
```

Note that, when $j \leftarrow x_i$ is executed, x_i represents the integer value of x_i. Let us see how Table II is created by this algorithm. First, $j \leftarrow c^{-1}(x_0) = 2$ is performed. Next, since $q(j, x_1) = q(2, b)$ is NULL, Output(2) and AddTable(2, b) are performed. The pointer-character table has new entry $p(4) = 2$ and $c(4) = b$. Also, 4 is stored in $q(2, b)$. Continuing similarly, we can confirm that the algorithm creates the pointer-character table and outputs 214630.

III. GPU implementation of LZW compression for TIFF images

We focus on LZW compression of an image into a TIFF image file. We assume 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 LZW compression for color image compression.

As illustrated in Figure 2, 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 [18]. 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.

Next, we will show how each strip is compressed. Since every pixel has an 8-bit intensity level, we can think that an input string of an integer in the range $[0, 255]$. Hence, codes from 0 to 255 are assigned to these integers. Code 256 (ClearCode) is reserved to clear the code table. Also, code 257 (EndOfInformation) is used to specify the end of the data. Thus, AddTable operations assign codes to strings from code 258. While the entry of the code table is less than 512, codes are represented as 9-bit integer. After adding code table entry 511, we switch to 10-bit codes. Similarly, after adding code table entry 1023 and 2037, 11-bit codes and 12-bit codes are used, respectively. As soon as code table entry 4094 is added, ClearCode is output. After that, the code table is re-initialized and AddTable operations use codes from 258 again. 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 $4094 - 257 + 1 = 3838$ codes. The last code segment for a strip may have codes less than that.

Let us discuss the implementation of back-pointer q for TIFF LZW compression. Since codes have up to 12 bits and characters are 8 bits, we can implement q as a table which has $2^{12} \times 2^8 = 2^{20}$ entries. Since the value of back-pointer $q(i,x)$ takes value up to 12 bits, each entry can be 2 bytes. Hence, a back pointer can be implemented in $2^{21} = 2$Mbytes. However, this straightforward implementation has large overhead due to

TABLE II. A POINTER-CHARACTER TABLE AND A BACK-POINTER TABLE TO IMPLEMENT CODE TABLE C

j	0	1	2	3	4	5	6	7	8	9
$p(j)$	NULL	NULL	NULL	NULL	2	1	4	6	3	0
$c(j)$	a	b	c	d	b	c	c	d	a	-
$q(j,a)$	NULL	NULL	NULL	8	NULL	NULL	NULL	NULL	NULL	NULL
$q(j,b)$	NULL	NULL	4	NULL	NULL	NULL	NULL	NULL	NULL	NULL
$q(j,c)$	NULL	5	NULL	NULL	6	NULL	NULL	NULL	NULL	NULL
$q(j,d)$	NULL	NULL	NULL	NULL	NULL	7	NULL	NULL	NULL	NULL
$C(j)$	a	b	c	d	cb	bc	cbc	$cbcd$	da	-

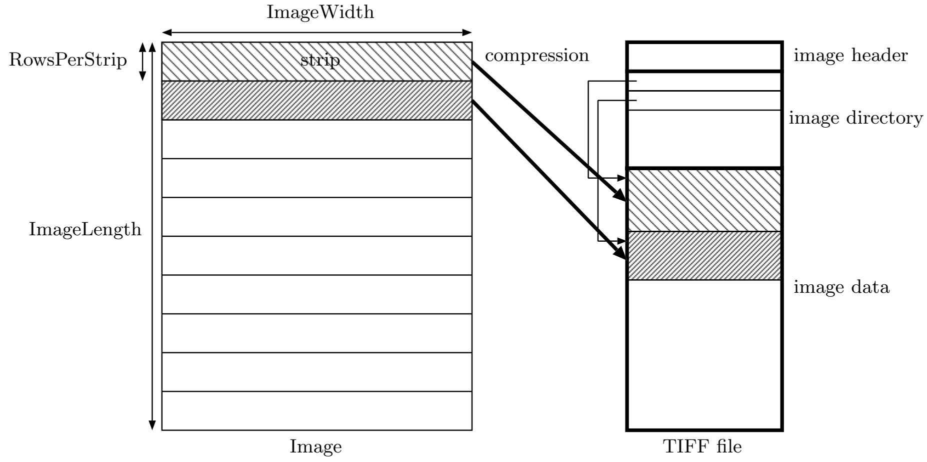


Fig. 2. An image and TIFF image file

the cache miss. Hence we will use a hash table to implement back-pointer q.

Let $h(i,x)$ be a hash function returning a 14-bit number, where i and x are 12 bits and 8 bits, respectively. In the experiment that we will show later, we have used the following hash function h to specify a 14-bit number.

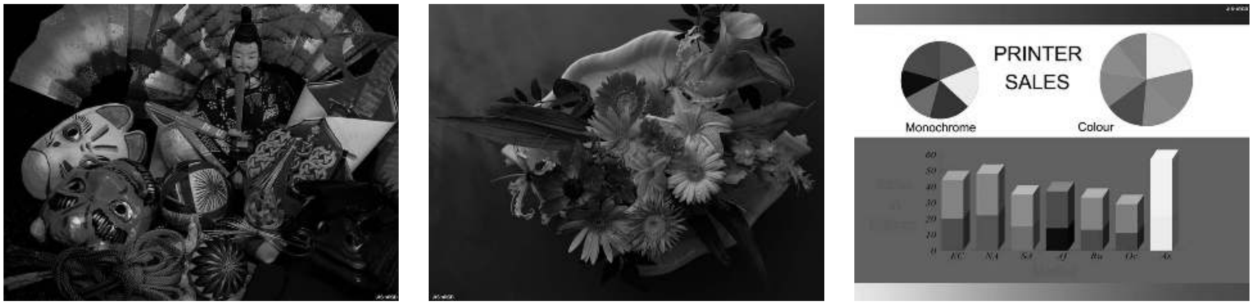
$$h(i,x) \quad = \quad (i \oplus (x << 10) \oplus (x >> 4)) \wedge \mathrm{0x3FFF},$$

We use an array of 2^{14} elements with 2 bytes each to store the 14-bit values of back pointers $q(i,x)$. When we write the value of back pointer in address $h(i,x)$, it may already be used. If this is the case, the current value of each address $(h(i,x) + 501i) \wedge \mathrm{0x3FFF}$ is read for $i = 1, 2, \ldots$ until an unused address is found. Since at most 3838 elements are added, the hash table of size $2^{14} = 16384$ is good enough.

After ClearCode is output, we need to initialize the hash table. However, it is too costly to clear all elements in the hash table. Hence, we use the time-stamp technique as follows: Since the value of each $q(i,x)$ has 12 bits is stored in 2 byte element, the remaining 4 bits are used as a time stamp. The time stamp takes value from 0 to $2^4 - 1 = 15$. Initially, the time stamp is 0 and incremented after ClearCode is output. When the new entry is added to and some value is written in $q(i,x)$, the current time stamp is written with it. Using the time stamp, one can determine if the value stored in each $q(i,x)$ is valid. When the time stamp is incremented 16 times, it is set to 0 and the values of all addresses are initialized by 0. Note that the size of the hash table is $2^{14} \cdot 2 = 32$K bytes, which is much smaller than the straightforward implementation. However, the hash table of size 32K bytes is too large to store it in the shared memory, we use the CUDA local memory, which is arranged in the global memory.

We are now in a position of our implementation of LZW compression using a CUDA-enabled GPU. We assume that an 8-bit gray scale image to be LZW-compressed is stored in the global memory of the GPU. Our implementation performs LZW compression and the resulting image is stored in the global memory using a TIFF format. To maximize parallelism, we set RowsPerStrip= 1, that is, each strip has one row of the gray-scale image. We assign each thread to one strip, which perform LZW compression of it independently. Each thread uses the local memory, which is mapped in the global memory of the GPU, to store the pointer-character table and the hash table. The details of our implementation is spelled out as follows:
[LZW compression using a CUDA-enabled GPU]
Step 1: The gray-scale image is transposed such that each row of the image is in a column.
Step 2: Each thread performs the LZW compression and the resulting sequence of LZW codes is written in the global memory.
Step 3: The prefix-sums of the lengths of the resulting sequences of LZW codes are computed.
Step 4: The resulting LZW codes are concatenated into one to fit a TIFF format using the prefix-sums.

One CUDA kernel is invoked for each of the three steps. Step 1 can be done by an algorithm for matrix transposition [26]. After the transposing, each row of the image is arranged in a column. Since every thread accesses the same position of a column, access to the image performed in Step 2 is coalesced. After Step 2, the resulting sequences of LZW codes generated by all threads are separated. To convert it in



“Crafts” “Flowers” “Graph”

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

a TIFF format, they must be concatenated. For concatenation, the prefix-sums of the lengths of all resulting sequences of LZW codes are computed in Step 3. More specifically, let $l_0, l_1, l_2, \ldots$ be the lengths of all resulting sequences. The prefix-sums $l_0, l_0+l_1, l_0+l_1+l_2, \ldots$ are computed. The prefix-sums can be computed by a GPU very efficiently [9], [27] From the prefix-sums, we can determine the position in the TIFF format where each resulting sequence must be copied. Step 4 performs this copy operation in an obvious way.

IV. Experimental results

We have used NVIDIA GeForce GTX 980 which has 16 streaming multiprocessors with 128 processor cores each to implement our parallel LZW compression algorithm. We also use Intel Core i7 4790 (3.6GHz) to evaluate the running time of sequential LZW compression.

We have used three gray scale images with 4096×3072 pixels (Figure 3), which are converted from JIS X 9204-2004 standard color image data. We set RowsPerStrip$= 1$, and so each image has 3072 strips with 4096 pixels each. We invoked a CUDA kernel with $\frac{4096}{32} = 128$ CUDA blocks of 32 threads each for compression. Table III shows the compression ratio, that is, “original image size: compressed image size.” We can see that “Graph” has high compression ratio because it has large areas with constant intensity levels. On the other hand, the compression ratio of “Crafts” is small because of the small details. Table III also shows the running time for LZW compression using a GPU and a CPU. It shows the running time of each step of GPU LZW compression. Clearly, Step 2 dominates the total computing time. The time for transposition, prefix-sum computation, concatenating LZW codes is negligible. The table also shows the running time of our GPU implementation for all steps. Since the sum of the running times of all steps is a little larger than that for all steps, the running time of each step includes overhead for measuring the running time. We can see that our GPU implementations is about three times faster than the CPU implementation. The last column shows the running time of Step 2 for the case that the input image is not transposed. Note that if memory access to the image is not coalesced if this is the case. Since the running time is rather longer than Step 2 with transpose, we should perform Step 1 beforehand.

We have evaluated the running time of two scenarios that may be used in real life applications. What we want to do is to store it using LZW-compressed TIFF format in the SSD (Solid State Drive) connected to the host PC. We compare the following two scenarios as shown in Figure 1

Scenario 1: The gray-scale image is compressed and converted into a TIFF image by our implementation on the GPU. After that, the resulting LZW-compressed TIFF image is transferred to the host PC and written in the SSD.

Scenario 2: The gray-scale image is transferred to the host PC and compressed using a CPU. After that, the resulting LZW-compressed TIFF image is written in the SSD.

Table IV shows the running time of each scenario. The compression time is much larger than the data transfer time both for Scenarios 1 and 2. Similarly, the time for all procedures is a little smaller than the sum of the running time of three procedures because of the overhead for measuring the running time. We can see that the Scenario 1 is about three times faster than Scenario 2. The readers may think that our CPU implementation is not efficient. Hence, we have also used libTIFF, which is a standard library for handling TIFF images [28]. The last column shows the time of Scenario 2 using libTIFF. Clearly, it is not faster than that of our CPU implementation.

Considering practical cases, some application may LZW-compress multiple images successively. Therefore, we evaluate the running time of LZW compression for multiple images. Table V shows the running time (in milliseconds) of LZW compression for multiple TIFF images of “Crafts” using our proposed GPU implementation. To utilize computation resources of the GPU as possible, we use CUDA stream [4] to execute kernels concurrently. We can see that the running time for multiple images is shorter than that for one image since kernels are invoked asynchronously and overhead due to invoking kernels is hidden. According to the table, when 16 or more images are LZW-compressed, the running time per image does not change. Also, the running time per image for 64 images is 1.38 times shorter than that for one image. Therefore, to increase the throughput of the execution, multiple images should be LZW-compressed.

V. Conclusion

In this paper, we have presented an implementation of LZW compression in a CUDA-enabled GPU. We have compared two scenarios to archive LZW-compressed TIFF images in the SSD. The scenario that uses a GPU for LZW compression is about three times faster than that uses a CPU. From this fact,

TABLE III. The running time (in milliseconds) of LZW compression using a GPU and a CPU for three images

Images	compression ratio	GPU (transposed)					CPU	Speed-up	GPU (non-transpose)
		Step 1	Step 2	Step 3	Step 4	All			Step 2
“Crafts”	$1.23:1$	0.32	29.3	0.015	0.17	29.3	92.8	3.2	40.4
“Flowers”	$1.44:1$	0.40	23.8	0.015	0.16	22.2	65.4	2.9	33.0
“Graph”	$10.8:1$	0.36	11.0	0.017	0.14	11.0	33.3	3.0	13.2

TABLE IV. The running time (in milliseconds) of two scenarios using our GPU and CPU implementations and libTIFF library for three images

Images	Scenario 1				Scenario 2				Speed-up	Scenario 2
	Compress on GPU	Transfer GPU→ CPU	Writing CPU→SSD	All	Transfer GPU→ CPU	Compress on CPU	Writing CPU→SSD	All		libTIFF
“Crafts”	29.3	2.34	3.85	35.2	3.84	92.8	3.84	100.4	2.9	118.6
“Flowers”	22.23	1.44	2.80	26.0	3.82	65.4	2.74	71.9	2.8	105.0
“Graph”	10.99	0.40	0.38	11.3	3.88	33.3	0.28	37.5	3.3	46.1

TABLE V. The running time (in milliseconds) of LZW compression for multiple images of “Crafts” using a GPU

Number of images	1	2	4	8	16	32	64
Running time	29.83	49.98	93.46	176.95	348.32	691.65	1380.68
Running time per image	29.83	24.99	23.37	22.12	21.77	21.64	21.57

we can say that it makes sense to compress images using a GPU to archive them in the SSD.

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,” in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 68–76.

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

[4] NVIDIA Corporation, “NVIDIA CUDA C programming guide version 7.0,” Mar 2015.

[5] A. Kasagi, K. Nakano, and Y. Ito, “Offline permutation algorithms on the discrete memory machine with performance evaluation on the GPU,” *IEICE Transactions on Information and Systems*, vol. Vol. E96-D, no. 12, pp. 2617–2625, Dec. 2013.

[6] ——, “An optimal offline permutation algorithm on the hierarchical memory machine, with the GPU implementation,” in *Proc. of International Conference on Parallel Processing (ICPP)*, Oct. 2013, pp. 1–10.

[7] NVIDIA Corporation, “NVIDIA CUDA C best practice guide version 7.0,” 2015.

[8] 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.

[9] K. Nakano, “Optimal parallel algorithms for computing the sum, the prefix-sums, and the summed area table on the memory machine models,” *IEICE Trans. on Information and Systems*, vol. E96-D, no. 12, pp. 2626–2634, 2013.

[10] K. Nakano, S. Matsumae, and Y. Ito, “The random address shift to reduce the memory access congestion on the discrete memory machine,” in *Proc. of International Symposium on Computing and Networking*, Dec. 2013, pp. 95–103.

[11] A. Kasagi, K. Nakano, and Y. Ito, “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)*, Sept. 2014, pp. 251–250.

[12] Y. Ito and K. Nakano, “A GPU implementation of dynamic programming for the optimal polygon triangulation,” *IEICE Transactions on Information and Systems*, vol. E96-D, no. 12, pp. 2596–2603, Dec. 2013.

[13] H. Kouge, Y. Ito, and K. Nakano, “A GPU implementation of clipping-free halftoning using the direct binary search,” in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (LNCS 8630)*, Aug. 2014, pp. 57–70.

[14] Y. Ito, K. Ogawa, and K. Nakano, “Fast ellipse detection algorithm using Hough transform on the GPU,” in *Proc. of International Conference on Networking and Computing*. IEEE CS Press, Dec. 2011, pp. 313–319.

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

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

[17] T. Welch, “High speed data compression and decompression apparatus and method,” US patent 4558302, Dec. 1985.

[18] Adobe Developers Association, *TIFF Revision 6.0*, June 1992. [Online]. Available: http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf

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

[20] A. L. V. Nicolaisen, “Algorithms for compression on GPUs,” Ph.D. dissertation, Technical University of Denmark, Aug. 2015.

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

[22] S. Funasaka, K. Nakano, and Y. Ito, “A parallel algorithm for LZW decompression, with GPU implementation,” in *to appear in Proc. of International Conference on Parallel Processing and Applied Mathematics*, 2015.

[23] D. Takafuji, K. Nakano, and Y. Ito, “A CUDA C program generator for bulk execution of a sequential algorithm,” in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing*, Aug. 2014, pp. 178–191.

[24] K. Tani, D. Takafuji, K. Nakano, and Y. Ito, “Bulk execution of oblivious algorithms on the unified memory machine, with GPU implementation,” in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2014, pp. 586–595.

[25] T. Fujita, K. Nakano, and Y. Ito, “Bulk gcd computation using a gpu to break weak rsa keys,” in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2015, pp. 385–394.

[26] K. Nakano, “Simple memory machine models for GPUs,” *International Journal of Parallel, Emergent and Distributed Systems*, vol. 29, no. 1, pp. 17–37, 2014.

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

[28] *libTIFF - TIFF Library and Utilities*. [Online]. Available: http://www.remotesensing.org/libtiff/