A Parallel Algorithm for LZW decompression, with GPU implementation

Shunji Funasaka, Koji Nakano, and Yasuaki Ito

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

Abstract. The main contribution of this paper is to present a 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, its parallelization is not an easy task. We first present a parallel LZW decompression algorithm on the CREW-PRAM. We then go on to present an efficient implementation of this parallel algorithm on a GPU. The experimental results show that our parallel LZW decompression on GeForce GTX 980 runs up to 69.4 times faster than sequential LZW decompression on a single CPU. We also show a scenario that parallel LZW decompression on a GPU can be used for accelerating big data applications.

Keywords: 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 [4]. 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) [7], 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.

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* [9]. 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 far more than images. In this paper, we focus on LZW compression, which is one of the most well known patented lossless

compression method [11] 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. However, LZW compression and decompression are hard to parallelize, because they use dictionary tables created by reading input data one by once. In [10], 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 [6, 8]. As far as we know, no parallel LZW decompression using GPUs has not been presented. In particular, decompression may be performed more frequently than compression; each image is compressed and written in a file once, but it is decompressed whenever the original image is used. Hence, we can say that LZW decompression is more important than the compression.

The main contribution of this paper is to present a parallel algorithm for LZW decompression and the GPU implementation. We first show that a parallel algorithm for LZW decompression on the CREW-PRAM [2], which is a traditional theoretical parallel computing model with a set of processors and a shared memory. We will show that LZW decomposition of a string of m codes can be done in $O(L_{\max} + \log m)$ time using $\max(k, m)$ processors on the CREW-PRAM, where $L_{\max}$ is the maximum length of characters assigned to a code. 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 Xeon CPU X7460 processor show that our implementation on a GPU achieves a speedup factor up to 69.4 over a single CPU.

Let us consider the following scenario to use LZW compression and decompression. Suppose that we have a set of bulk data such as images or text stored in a storage of a host computer with a GPU. A user gives a query to the set of bulk data and all data must be processed to answer the query. To accelerate the computation for the query, data are transferred to the GPU through the host computer and they are processed by parallel computation on the GPU. For the purpose of saving storage space and data transfer time, data are stored in the storage as LZW compressed format. If this is the case, compressed data must be decompressed using the host computer or using the GPU before the query processing is performed. We will show that, since LZW decompression can be done very fast in the GPU by our parallel algorithm, it makes sense to store compressed data in a storage and to decompress them using the GPU.

2 LZW compression and decompression

The main purpose of this section is to review LZW compression/decompression algorithms. Please see Section 13 in [1] for the details.

The LZW (Lempel-Ziv & Welch) [12] compression algorithm converts an input string of characters into a string of codes using a string table that maps strings into codes. If the input is an image, characters may be 8-bit integers. It

reads characters in an input string one by one and adds an entry in a string table (or a dictionary). In the same time, it writes an output string of codes by looking up the string 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. For simplicity of handling the boundary case, we assume that an input is a string of 4 characters a, b, c, and d. Let S be a string table, which determines a mapping of a string to a code, where codes are non-negative integers. Initially, $S(a) = 0$, $S(b) = 1$, $S(c) = 2$, and $S(d) = 3$. By procedure AddTable, new code is assigned to a string. For example, if AddTable(cb) is executed after initialization of S, we have $S(cb) = 4$. The LZW compression algorithm is described as follows:

[LZW compression algorithm]
1 for $i \leftarrow 0$ to $n-1$ do
2 if($\Omega \cdot x_i$ is in S)
3 $\Omega \leftarrow \Omega \cdot x_i$;
4 else
5 Output($S(\Omega)$); AddTable($\Omega \cdot x_i$); $\Omega \leftarrow x_i$;
6 Output($S(\Omega)$);

In this algorithm, Ω is a variable to store a string. Also, "$\cdot$" denotes the concatenation of strings/characters.

Table 1 shows how the compression process for an input string *cbcbcbcda*. 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($S(c)$) and AddTable(cb) are performed. More specifically, $S(c) = 2$ is output and we have $S(cb) = 4$. Also, $\Omega \leftarrow x_1 = b$ is performed. It should have no difficulty to confirm that 214630 is output by this algorithm.

Table 1. String table S, 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
S	-	$cb:4$	$bc:5$	-	$cbc:6$	-	-	$cbcd:7$	$da:8$	-
Y	-	2	1	-	4	-	-	6	3	0

Next, let us show LZW decompression algorithm. Let C be *the code table*, the inverse of string table S. For example if $S(cb) = 4$ then $C(4) = cb$. Initially, $C(0) = a$, $C(1) = b$, $C(2) = c$, and $C(3) = d$. Also, 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 a code table. In 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 ← 1 to n − 1 do
3      if(y_i is in C)
4          Output(C(y_i)); AddTable(C(y_{i−1}) · C_1(y_i));
5      else
6          Output(C(y_{i−1}) · C_1(y_{i−1})); AddTable(C(y_{i−1}) · 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.

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

3 Parallel LZW decompression

This section shows our parallel algorithm for LZW decompression.

Again, let $X = x_0x_1 \cdots x_{n-1}$ be a string of characters. We assume that characters are selected from an alphabet (or a set) with k characters $\alpha(0), \alpha(1)$, ..., $\alpha(k-1)$. 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 $Y = y_0y_1 \cdots y_{m-1}$ 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 code table Y as follows:

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

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 $p^{L(i)-1}(i) \neq \text{NULL}$. Also, let $p^*(i) = p^{L(i)-1}(i)$. Intuitively, $p^*(i)$ corresponds to the dead end from code i along pointers. Further, let $C_l(i)$ $(0 \leq i \leq k+m-2)$ be a character defined as follows:

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

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 $Y = 214630$.

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	-
$C_l(i)$	a	b	c	d	b	c	c	d	a	-
$C(i)$	a	b	c	d	cb	bc	cbc	$cbcd$	da	-

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 $(0 \leq i \leq k-1)$. Also, we use m processors and assign one processor to each i $(k \leq i \leq 2k+m-1)$, 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 ← 0 to k − 1 do in parallel // Initialization
2      p(i) ← NULL; L(i) = 1; C_l(i) ← α(i);
3  for i ← k to k + m − 1 do in parallel // Computation of L and p*
4      p(i) ← y_{i−k}; p*(i) ← y_{i−k};
5      while(p(p*(i)) ≠ NULL)
6          L(i) ← L(i) + 1; p*(i) ← p(p*(i));
7  for i ← k to k + m − 2 do in parallel // Computation of C_l
8      C_l(i) ← α(p*(i + 1));
```

Step 2 of the parallel LZW decompression algorithm uses m threads to compute $C(y_0) \cdot C(y_1) \cdots C(y_{m-1})$, which is equal to $X = x_0 x_1 \cdots x_{n-1}$. For this

purpose, we compute the prefix-sums of $L(y_0), L(y_1), \ldots, L(y_{m-2})$ using $m-1$ processors. In other words, $s(i) = L(y_0)+L(y_1)+\cdots+L(y_i)$ is computed for every i $(0 \leq i \leq m-1)$. For simplicity, let $s(-1) = 0$. After that, for each i $(0 \leq i \leq m-1)$ $L(y_i)$ 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 $x_{s(i)-1}$. Note that, the values of $p^0(y_i), p^1(y_i), \ldots, p^{L(i)-1}(y_i)$ can be obtained by traversing pointers from code i. Hence, it makes sense to perform the copy operation from $x_{s(i)-1}$ down to $x_{s(i-1)}$.

Table 4 shows the values of . By concatenating them, 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

Let us evaluate the computing time. Let $L_{\max} = \max\{L(i) \mid 0 \leq i \leq k+m-1\}$. The for-loop in line 1 takes $O(1)$ time using k processors. Also, while-loop in line 5 is repeated at most $L(i) \leq L_{\max}$ times for each i. Hence, for-loop in line 3 can be done in $O(L_{\max})$ time using m processors. It is well known that the prefix-sums of m numbers can be computed in $O(\log m)$ time using m processors [2]. Hence, every $s(i)$ is computed in $O(\log m)$ time using $m-1$ 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 using m processors. Therefore, we have

Theorem 1. *The LZW decomposition of a string of m codes can be done in $O(L_{\max} + \log m)$ time using $\max(k, m)$ processors on the CREW-PRAM, where k is the number of characters in an alphabet.*

4 GPU implementation

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. 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 [1]. 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.

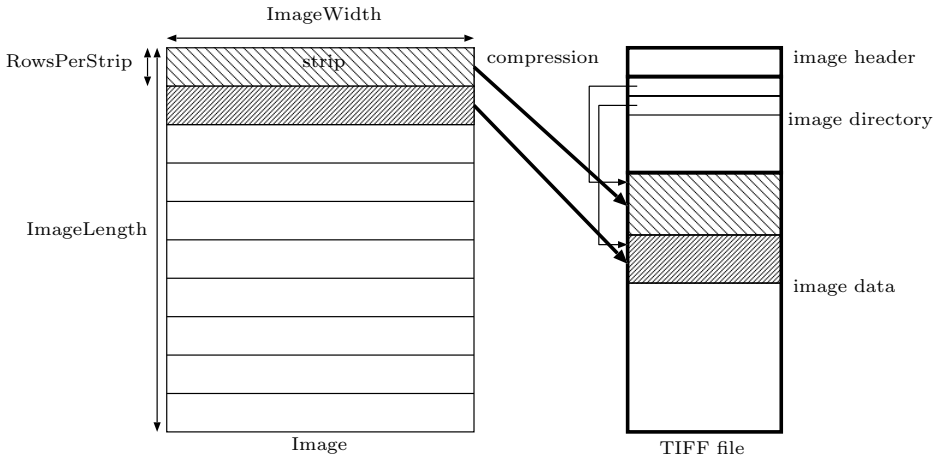

Fig. 1. An image and TIFF image file

Next, we will show how each strip is compressed. Since every pixel has an 8-bit intensity level, we can think that 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 $4094 - 511 + 1 = 3584$ codes. The last code segment for a strip may have codes less than that.

In our implementation, a CUDA block with 1024 threads is assigned a strip. A CUDA block decompresses each code segment in the assigned strip one by

one. More specifically, a CUDA block copies a code segment of a strip stored in the global memory to a shared memory. After that, it performs Step 1 of the parallel LZW decompression. Tables for p, p^*, L, and C_l are created in the shared memory. We use 16-bit unsigned short integer for each element of the tables. Since each table has at most 4096 entries, the total size of tables is $4 \times 4096 \times 2 = 32$Kbytes. Since the capacity of the shared memory is 48Kbytes [7], this is possible. Since the table has 4095 entries, 1024 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 ($0 \leq i \leq 1023$) are computed. After that, these values for every i ($0 \leq i \leq 1023$) 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 as $p^*(i) < 1024$ is satisfied.

After the tables are obtained, the prefix-sums of s is computed in the shared memory for Step 2. Finally, the strings of characters of each code are written in the global memory. The prefix-sums can be computed by parallel algorithm for GPUs [3, 5].

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 Xeon CPU X7460 (2.66GHz) to evaluate the running time of sequential LZW decompression.

We have used three gray scale images with 4096×3072 pixels (Figure 2), which are converted from JIS X 9204-2004 standard color image data. 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. Table 5 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 5 also shows the running time of LZW decompression using a CPU and a GPU. In the table, T_1 and T are the time for constructing tables and the total computing time, respectively. To evaluate time T_1 of sequential LZW decompression, OUTPUT in lines 4 and 6 are removed. Also, to evaluate time T_1 of parallel LZW decompression on the GPU, the CUDA kernel call is terminated without computing the prefix-sums and writing resulting characters in the global memory. Hence, we can think that $T - T_1$ corresponds to the time for for generating the original string using the tables. Clearly, sequential/parallel LZW decompression algorithms take more time to create tables for images with small compression ratio because they have many segments and need to create tables many times.

Also, the time for creating tables dominates the computing time of sequential LZW decompression, while that for writing out characters dominates in parallel LZW decompression. This is because the overhead of the parallel prefix-sums computation is not small. From the table, we can see that LZW decompression for "Flowers" using GPU is 69.4 times faster than that using CPU.

"Crafts" "Flowers" "Graph"

Fig. 2. Three gray scale image with 4096×3072 pixels used for experiments

Table 5. Experimental results (milliseconds) for three images

images	compression ratio	sequential(CPU)			parallel(GPU)			Speedup ratio
		T_1	$T - T_1$	T	T_1	$T - T_1$	T	
"Crafts"	1.67 : 1	90.4	18.0	108	0.847	1.30	2.15	50.2 : 1
"Flowers"	2.34 : 1	70.9	16.6	87.5	0.541	0.719	1.26	69.4 : 1
"Graph"	38.0 : 1	27.2	19.3	46.5	0.202	1.59	1.79	26.0 : 1

Let us discuss the performance of three practical scenarios as follows:
Scenario 1: Non-compressed images are stored in the storage. They are transferred to the GPU thorough the host computer.
Scenario 2: 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.
Scenario 3: LZW compressed images are stored in the storage. They are transferred to the GPU through the host computer, and decompressed in the GPU.

The throughput between the storage and the host computer depends on their bandwidth. For simplicity, we assume that their bandwidth is the same as that between the host computer and the GPU. Note that since the bandwidth of the storage is not larger than that of the GPU in many cases, this assumption does not give advantage to Scenario 3. Table 6 shows the data transfer time for non-compressed and compressed files of the three images. Clearly, the time for non-compressed files is almost the same, because they have the same size. On

the other hand, images with higher compression ratio take fewer time, because their sizes are smaller. It also evaluates the time for three scenarios. Clearly, Scenario 3, which uses parallel LZW decompression in the GPU, takes much fewer time than the others.

Table 6. Estimated running time (milliseconds) of three scenarios

images	compression ratio	Data Transfer		Scenario 1	Scenario 2	Scenario 3
		non-compressed	compressed			
"Crafts"	1.67 : 1	3.79	2.44	7.58	110	4.59
"Flowers"	2.34 : 1	3.83	1.73	7.66	89.2	2.99
"Graph"	38.0 : 1	3.80	0.167	7.60	46.7	1.96

6 Conclusion

In this paper, we have presented a parallel LZW decompression algorithm and implemented in the GPU. The experimental results show that, it achieves a speedup factor up to 69.4. Also, LZW decompression in the GPU can be used to accelerate the query processing for a lot of compressed images in the storage.

References

1. Adobe Developers Association: TIFF Revision 6.0 (June 1992), http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf
2. Gibbons, A., Rytter, W.: Efficient Parallel Algorithms. Cambridge University Press (1988)
3. Harris, M., Sengupta, S., Owens, J.D.: Chapter 39. parallel prefix sum (scan) with CUDA. In: GPU Gems 3. Addison-Wesley (2007)
4. Hwu, W.W.: GPU Computing Gems Emerald Edition. Morgan Kaufmann (2011)
5. Nakano, K.: Simple memory machine models for GPUs. In: Proc. of International Parallel and Distributed Processing Symposium Workshops. pp. 788–797 (May 2012)
6. Nicolaisen, A.L.V.: Algorithms for Compression on GPUs. Ph.D. thesis, Tecnical University of Denmark (Aug 2015)
7. NVIDIA Corporation: NVIDIA CUDA C programming guide version 6.5 (Aug 2014)
8. Ozsoy, A., Swany, M.: Culzss: Lzss lossless data compression on cuda. In: Proc. of International Conference on Cluster Computing. pp. 403–411 (Sept 2011)
9. Sayood, K.: Introduction to Data Compression, Fourth Edition. Morgan Kaufmann (2012)
10. Shyni, K., Kumar, K.V.M.: Lossless LZW data compression algorithm on CUDA. IOSR Journal of Computer Engineering pp. 122–127 (2013)
11. Welch, T.: High speed data compression and decompression apparatus and method. US patent 4558302 (Dec 1985)
12. Welch, T.A.: A technique for high-performance data compression. IEEE Computer 17(6), 8–19 (June 1984)