A Study of Loss-Less Data Compression Method Optimized for GPU Decompression

(GPU 上での展開に適した可逆データ圧縮方式に関する研究)

Shunji Funasaka

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Engineering in Information Engineering

Under Supervision of
Professor Koji Nakano

Department of Information Engineering
Graduate School of Engineering
Hiroshima University

March, 2018

Abstract

A GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [1, 2, 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) [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. Most lossless data compression and decompression algorithms are very hard to parallelize, because they use dictionaries updated sequentially. Usually, data in a large archive are stored in a compressed format to save the space. If each of compressed data is accessed many times, it makes sense to use a data compression method that maximizes performance in terms of decompression time and compression ratio. In particular, if archived data stored in a data center are accessed by users, and they are processed on the GPU for display, such data compression method optimized for GPU decompression should be used.

First, we present a work-optimal parallel algorithm for LZW decompression and to implement it in a CUDA-enabled GPU. Since sequential LZW decompression creates a dic-

tionary 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.

Second, we present a new lossless data compression method that we call Adaptive Loss-Less (ALL) data compression. It is designed so that the data compression ratio is moderate but decompression can be performed very efficiently on the GPU. This makes sense for applications such as training of deep learning, in which compressed archived data are decompressed many times. To show the potentiality of ALL data compression method, we have evaluated the running time using five images and five text data and compared ALL with previously published lossless data compression methods implemented in the GPU, Gompresso, CULZSS, and LZW. The data compression ratio of ALL data compression is

better than the others for eight data out of these 10 data. Also, our GPU implementation on GeForce GTX 1080 GPU for ALL decompression runs 84.0-231 times faster than the CPU implementation on Core i7-4790 CPU. Further, it runs 1.22-23.5 times faster than Gompresso, CULZSS, and LZW running on the same GPU.

Contents

1 Introduction **1**

1.1 Background 1

1.2 Contributions 3

1.2.1 A parallel algorithm for LZW decompression, with GPU implementation 3

1.2.2 A new loss-less data compression method: ALL(Adaptive Loss-Less data compression) 5

1.3 Dissertation organization 6

2 GPU and CUDA **7**

3 Loss-less data compression and related work **13**

4 A GPU implementation of LZW compression and decompression **17**

4.1 LZW compression and decompression 17

4.2 GPU implementation of LZW compression for TIFF images 22

4.2.1 GPU implementation 22

4.2.2 Experimental results 26

4.3 Parallel LZW decompression 30

4.4 GPU implementation of LZW decompression for TIFF images 34

4.4.1 TIFF images 34

4.4.2 GPU implementation of parallel LZW decompression 36

4.5 Experimental results 39

5 Adaptive loss-less data compression method optimized for GPU decompression 45

5.1 ALL: Adaptive Loss-Less data compression 45

5.1.1 Outline of ALL coding 45

5.1.2 ALL codes 48

5.1.3 Adaptive Dictionary 50

5.1.4 Data block: encoded data for a strip 51

5.1.5 ALL file format 53

5.1.6 Compression and decompression algorithms 54

5.2 GPU implementation for ALL decompression 56

5.2.1 Parallel prefix scan on the GPU 56

5.2.2 CUDA kernel for ALL decompression 58

5.2.3 A CUDA block decompressing a data block 60

5.2.4 Performance issues of the GPU implementation for ALL decompression 62

5.3 Experimental results 63

6 Conclusion 70

Bibliography 75

Acknowledgement 76

List of publications 77

List of Figures

2.1 GPU architecture . . . 8
2.2 Shared Memory with 32 banks . . . 9
2.3 two types of global memory access . . . 10
2.4 CUDA architecture . . . 10
2.5 warp vote functions . . . 11
2.6 The example of calculating occupancy(GTX1080) . . . 12

4.1 An image and TIFF image file . . . 23
4.2 Three gray scale image with 4096 × 3072 pixels used for experiments . . . 27
4.3 Two scenarios to archive an LZW-compressed image in the SSD . . . 29
4.4 GPU implementation of parallel LZW decompression . . . 37
4.5 Three gray scale image with 4096 × 3072 pixels used for experiments . . . 40
4.6 Scenarios A, B, and C . . . 42

5.1 An example of ALL codes and the decoded string . . . 52
5.2 A data block for a strip of 65536 bytes . . . 53
5.3 ALL file header . . . 54
5.4 Parallel prefix scan . . . 57

5.5 Three standard images used for experiments 65

List of Tables

3.1 Lossless data compression methods and tools 15

4.1 String stored in Ω, code table C, and output string Y for $X = cbcbcbcda$ . . 19

4.2 Code table C and the output string for 214630 20

4.3 The running time (in milliseconds) of LZW compression using a GPU and a CPU for three images 27

4.4 The running time (in milliseconds) of two scenarios using our GPU and CPU implementations and libTIFF library for three images 28

4.5 The running time (in milliseconds) of LZW compression for multiple images of "Crafts" using a GPU 28

4.6 The values of p, p^*, l, C_l, and C for Y = 214630 31

4.7 The values of $L(y_i)$, $s(i)$, and $C(y_i)$ for Y = 214630 33

4.8 The running time (milliseconds) of LZW decompression 41

4.9 The running time (milliseconds) of each Scenario for three images 44

5.1 ALL codes: p is the previous character and $d_0 d_1 \cdots d_{4095}$ are 4096 characters in the dictionary 50

5.2 Data set used for evaluating the performance 64

5.3 Data compression ratio by four compression methods 66

5.4 The running time in milliseconds for decompression 67

5.5 The ratio of the total size of original data compressed using each of five types of code 67

5.6 The SSD-GPU loading time in milliseconds using ALL decompression for three scenarios 69

Chapter 1

Introduction

1.1 Background

A GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [1]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [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.

CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [6]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-96 Kbytes. The global memory is implemented as an off-chip DRAM, and thus, it has large capacity, say, 1.5-12 Gbytes, but its access latency is very large. When we develop CUDA programs, we need to consider *the coalescing* of the global memory

access [7, 8]. To maximize the bandwidth between the GPU and the off-chip DRAM, 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* [9]. Lossy compression can generate smaller files, but some information in original files are discarded. Hence, decompression of lossy compressed data does not generate files identical to the original data. On the other hand, lossless compression creates compressed data, from which we can obtain the exactly same original data by decompression.

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

1.2 Contributions

In this dissertation, we present the GPU implementation of LZW decompression and a new loss-less compression method for very efficient decompression method on the GPU.

1.2.1 A parallel algorithm for LZW decompression, with GPU implementation

In this work, we focus on LZW(Lempel-Ziv-Welch) compression, which is the most well known patented lossless compression method [10] 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 [11], which is commonly used in the area of commercial digital printing. We 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) [12, 13, 14], and 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.

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.

1.2.2 A new loss-less data compression method: ALL(Adaptive Loss-Less data compression)

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

ALL data compression has five types of codes of length 1, 2, or 3 bytes as follows: *single character code* (SC, 1 byte), *short run-length code* (SRL, 2 bytes), *short interval code* (SI, 2 bytes), *long run-length code* (LRL, 3 bytes), and *long interval code* (LI, 3 bytes). An SC code simply represents a 1-byte uncompressed character. SRL and LRL codes encode a run of the same character with length in ranges $[2, 16]$ and $[18, 3408]$, respectively. Also, SI and LI codes encode a substring in the dictionary of length in ranges $[2, 16]$ and $[18, 3408]$, respectively. Since longer run and substring, which tend to appear less frequently, use more bytes, we can think this encoding is Huffman-based byte-wise coding. To accelerate decompression using the GPU, we use segment-wise coding in which multiple codes use

the same dictionary and they can be decoded in parallel using multiple threads in the GPU. However, segment-wise coding may deteriorate the data compression ratio. To compensate this deterioration, we use *adaptive dictionary coding*, in which the prefix of the dictionary is replaced by *a magic string*. A magic string is used to encode substrings appear frequently. It can also be used to encode random substrings which can not be compressed into a smaller sequence of codes. To clarify the performance of ALL data compression method, we have evaluated the data compression ratio and the running time using five images and five text data and compared ALL with the other lossless data compression methods implemented in the GPU, Gompresso, CULZSS, and LZW. The data compression ratio of ALL data compression is better than the others for eight data out of these 10 data. Also, our GPU implementation on GeForce GTX 1080 GPU for ALL decompression runs 84.0-231 times faster than the CPU implementation on Core i7-4790 CPU. Further, it runs 1.22-23.5 times faster than Gompresso, CULZSS, and LZW.

1.3 Dissertation organization

This dissertation is organized as follows. Chapter 2 describes the GPU and CUDA. We then go on to describe loss-less data compression methods and, related works for GPU implementations of loss-less compression and decompression in Chapter 3. In Chapter 4, we present parallel algorithm for LZW decompression and its GPU implementation. Chapter 5 shows a new lossless data compression method called ALL (Adaptive Loss-Less) data compression. Chapter 6 concludes this dissertation.

Chapter 2

GPU and CUDA

In this chapter, we describe the NVIDIA GPU architecture and the CUDA. *A GPU* (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images [1]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [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. Figure 2.1 shows an architecture of CUDA-enabled GPU. A CUDA-enabled GPU has multiple streaming processors(SMs), each of which has execution cores with integer and floating point operations, shared memory, register file. For example, Geforce GTX 1080 (Compute Capability 6.1) has 20 SMs with 128 processor cores, a 96K bytes shared memory, and a register file with 64K 32-bit registers each. CUDA uses two types of memories:the shared memory and

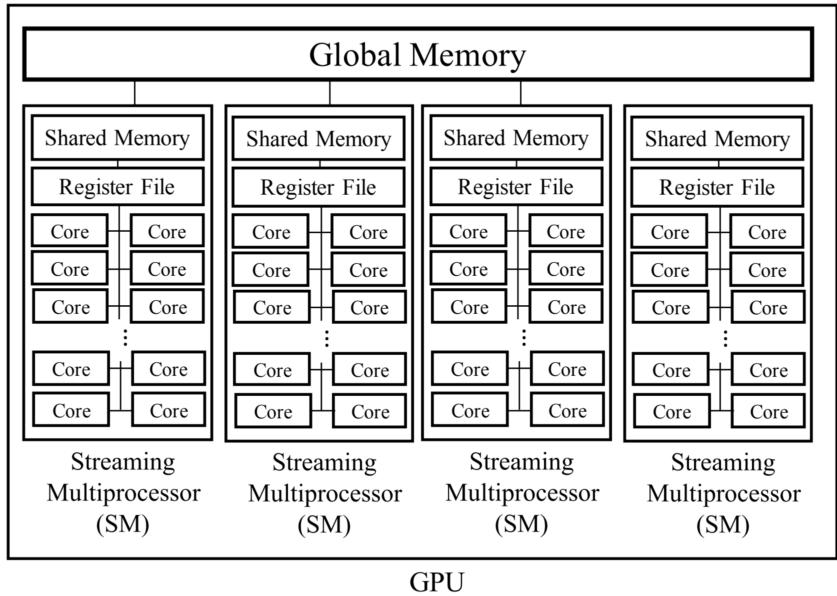


Figure 2.1: GPU architecture

the global memory. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-96 Kbytes. This memory is divided into equally-sized memory modules, called banks. For example, GTX 1080 has 32 banks in the shared memory and each banks which can be accessed simultaneously is 32-bit. Figure 2.2 shows the shared memory and two access patterns. If two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized [6]. In figure 2.2, thread 0 and thread 1 access to bank 0 sequentially, however, thread 2 and thread 3 access to the shared memory simultaneously.

The global memory is implemented as an off-chip DRAM, and thus, it has large capacity, say, 1.5-12 Gbytes, but its access latency is very large. When we develop CUDA programs, we need to consider *the coalescing* of the global memory access [7, 8]. To maximize the bandwidth between the GPU and the off-chip DRAM, the consecutive addresses of the

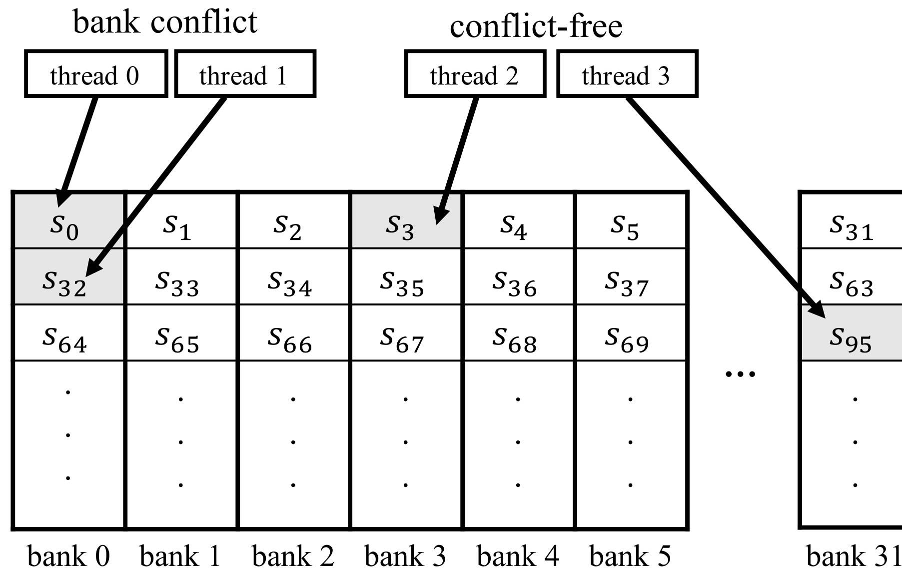


Figure 2.2: Shared Memory with 32 banks

global memory must be accessed at the same time(coalesced access). However, when the distant locations are accessed at the same time(stride access), this access pattern needs a lot of clock cycles. Thus, CUDA threads should perform coalesced access when they access the global memory. Figure 2.3 shows two types of global memory access: coalesced access and stride access.

A CUDA program running on a host PC invokes one or more CUDA kernels executed on a GPU in turn. Each CUDA kernel has one or more CUDA blocks, a set of threads running on a GPU. Figure 2.4 illustrates the CUDA programming model with 128 threads/block. Threads in a CUDA block are partitioned into groups of 32 threads each called warps. All threads in the same warp work completely synchronously, share the program counter, and execute the same instruction. CUDA supports warp shuffle instructions that exchange a variable between threads with in a warp without shared memory. We can shuffle data in a

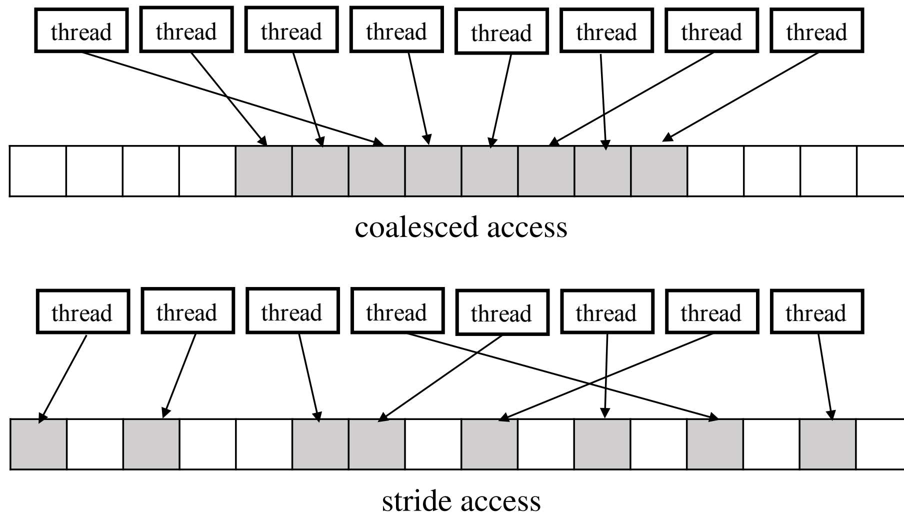


Figure 2.3: two types of global memory access

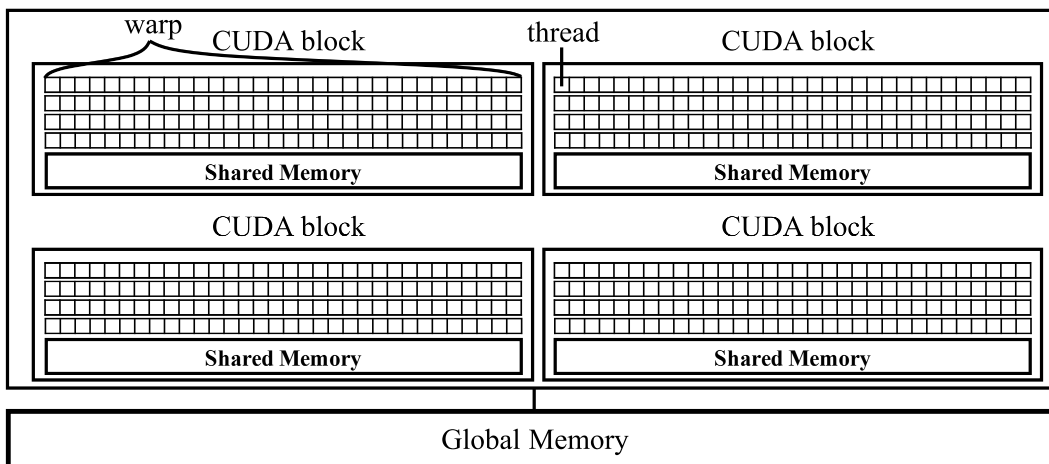


Figure 2.4: CUDA architecture

warp very efficiently. Also, warp vote functions are supported by CUDA. The warp vote functions: __any, __all and __ballot (__all_sync, __any_sync and __ballot_sync in CUDA 9.0) allow the threads of a given warp to perform a reduction-and-broadcast operation [6]. __any(predicate) returns non-zero if and only if predicate evaluates to true for one and more threads in a warp. On the other hand, __all(predicate) returns non-zero if and only if predicate evaluates to true for all threads in a warp. Finally, __ballot(predicate) returns an 32-bit

integer whose n-th bit is 1 if and only if n-th thread in a warp evaluates predicate and its result is true. Figure 2.5 shows three warp vote functions when we assume that a warp is 8-thread and each functions return 8-bit integer. We assume thread 0,1,4 and 6 have integer 3 in register a. Then, the result value of __ballot(a == 3) is $(01010011)_2$ which is that 0-th, 1-th, 4-th and 6-th bits are set.

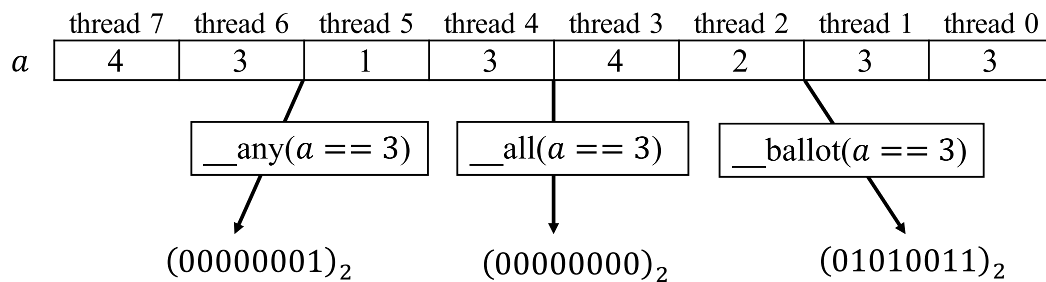


Figure 2.5: warp vote functions

To maximize the performance of a GPU implementation, we should invoke a many threads and CUDA blocks as possible to hide large memory access latency. GPU occupancy as a percentage can be calculated by dividing concurrent threads by a maximum number of threads per multiprocessor. Since there are a maximum number of resident blocks, warps, threads, registers and Maximum amount of shared memory in a streaming multiprocessor, we should write CUDA programming codes carefully such that GPU occupancy is high. For example, Since the compute capability of GeForce GTX 1080 is 6.1, each streaming multiprocessor can have up to 64 resident warps, 2048 resident threads and 32 resident CUDA blocks [6]. If our GPU implementation to GTX 1080 uses 32 registers/thread, 128 threads/block, 32 resisters/thread and 24 Kbytes shared memory/block, a CUDA kernel can dispatch 4 CUDA blocks in each streaming multiprocessor because they

use $128 \times 4 = 256$ resident threads, $24 \times 4 = 96$ Kbytes shared memory and $256 \times 32 = 16$ K registers. Thus, the Occupancy is $\frac{256}{2048} = 25\%$ as illustrated in the figure 2.6. If this im-

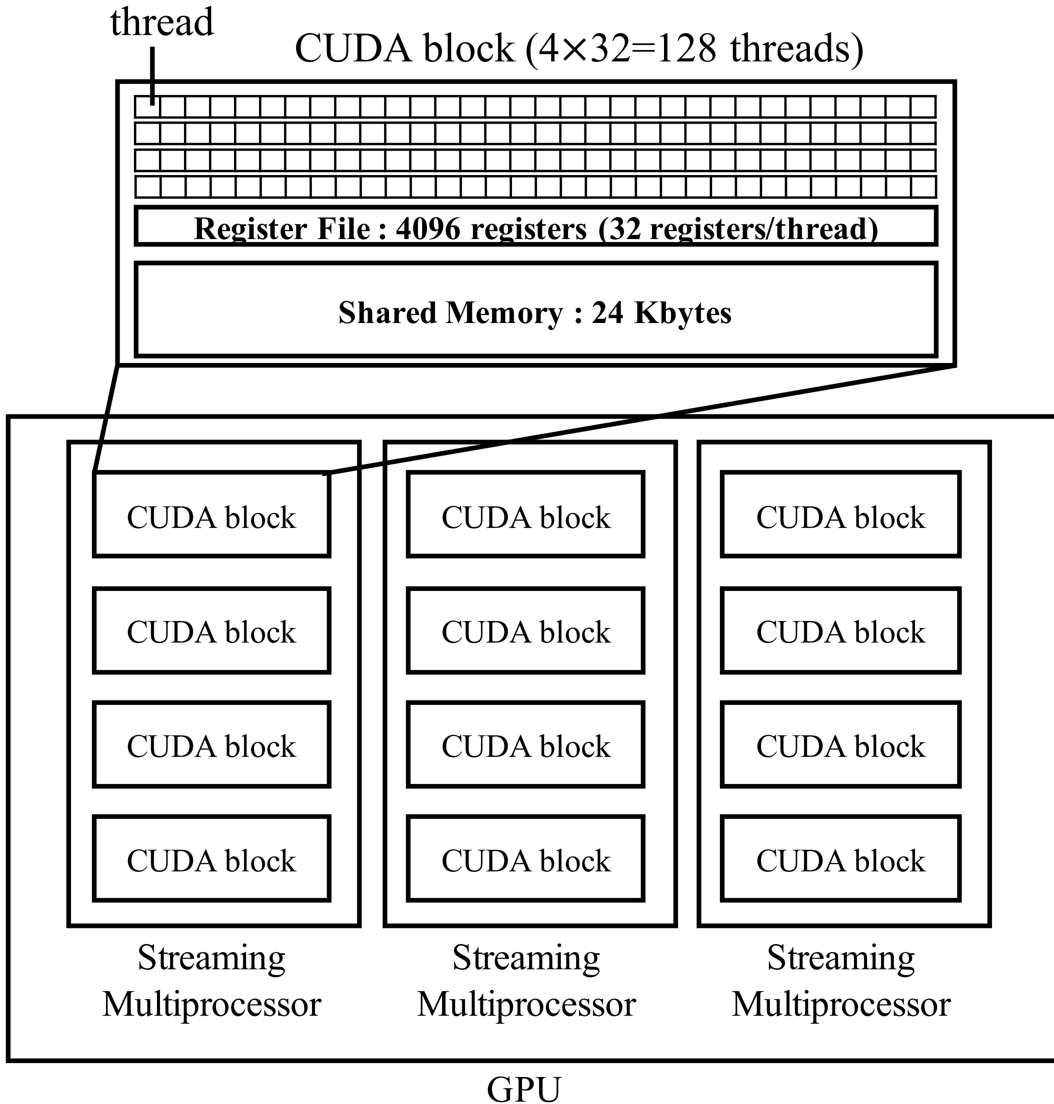


Figure 2.6: The example of calculating occupancy(GTX1080)

plementation uses 1 Kbytes shared memory/block, a CUDA kernel can dispatch 16 CUDA blocks($1 \times 16 = 16$ Kbytes shared memory). Hence, the Occupancy is $\frac{128 \times 16}{2048} = 100\%$.

Chapter 3

Loss-less data compression and related work

In this chapter, we describe loss-less data compression methods and, related works for GPU implementations of loss-less compression and decompression.

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*. Lossy compression can generate smaller files, but some information in original files are discarded. Hence, decompression of lossy compressed data does not generate files identical to the original data. On the other hand, lossless compression creates compressed data, from which we can obtain the exactly same original data by decompression.

As usual, we assume that an input data to be compressed is a string of 8-bit characters taking value in range [0,255]. The task of data compression is to convert it into a string of codes and that of data decompression is inverse-conversion. One of the simplest lossless compression methods is run-length encoding, in which each run of the same character is replaced by a code (character,count). For example, AAAABBBCC is encoded

to (A,4)(B,3)(C,2) by run-length encoding. This encoding is very useful if an input sequence has a lot of long runs such as line drawings. However, it does not achieve good compression ratio for images with few runs such as natural images and plain text. LZSS (Lempel-Ziv-Storer-Szymanski) [15] is a well-known dictionary-based lossless compression method, which replaces a substring appearing before by code (offset, length). LZSS decompression is performed using a buffer storing recently decoded string as a dictionary. Each code (offset,length) is decoded by retrieving the corresponding substring in the dictionary. For example, if the dictionary stores ABCDEFGH, then code (2,5) is decoded to CDEFG. LZ77 (Lempel-Ziv) [16] is a compression method very similar to LZSS. It uses code (offset, length, next unmatched character). LZW (Lempel-Ziv-Welch) [10] is a patented lossless compression method used in Unix file compression utility "compress" and in GIF image format. 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 needs 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 [17]. 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 [18], 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 [19, 20], but they are not LZW compression. In this dissertation, we present GPU implementation of LZW decompression with code-wise parallel in this work. The idea is to create a dictionary by parallel pointer traversing. Since pointer traversing is code-wise parallel, LZW decompression on the GPU is much faster than CULZSS decompression.

Recent lossless data compression uses several other techniques including arithmetic encoding, Burrows-Wheeler transform(BWT)[21] and move-to-front transform(MTF). Table 3.1 summarizes lossless compression methods used these days.

Table 3.1: Lossless data compression methods and tools

lossless compression	encoding method
LZSS, LZ77, LZW	dictionary-based encoding
gzip[22], LZH, LZMA	dictionary-based encoding, Huffman or arithmetic encoding
bzip2[23], ZZIP	BWT, MTF, Huffman-coding, run-length encoding

It is very hard to parallelize LZSS compression, because a newly decoded string is appended to the dictionary every time after a code is decoded and output. To parallelize LZSS compression, the input string is partitioned into equal-sized strips, each of which is compressed sequentially. Parallel LZSS decompression can be done for all strips in parallel,

but codes in each strip are decoded sequentially one by one. Such *strip-wise parallel* LZSS decompression called *CULZSS* have been implemented in a GPU [24],d but it achieves very small acceleration ratio over the sequential implementation on the CPU due to low parallelism.

LZ77-based [16] data compression called *Gompresso* and the GPU implementation have been shown in [25]. The structure of compressed data of Gompresso is a sequence of codes (unmatched characters of length at most 15, offset, length). The compression ratio and the running time on the GPU are better than CULZSS. The GPU implementation is *code-wise parallel* in the sense that a thread is arranged in each code of a compressed string. Hence, it has very high parallelism and a lot of threads work in parallel. Since memory access latency of the GPU is quite large, higher parallelism can hide large memory access latency and can attain better performance. However, decoding of codes is performed sequentially in the worst case due to the code dependency.

Encoding methods such as Huffman encoding and BWT are useful to obtain better compression ratio, but they are hard to parallelize on a GPU. In [26], a bzip2-like lossless data compression scheme and the GPU implementation have been presented, but the GPU implementation runs slower than the CPU implementation.

Chapter 4

A GPU implementation of LZW compression and decompression

In this chapter, we present GPU implementation of LZW compression and a work-optimal parallel algorithm for LZW decompression and to implement it in a CUDA-enabled GPU. Particular, we focus on GPU implementation of LZW decompression. 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 describe sequential LZW compression/decompression algorithm and GPU implementation of parallel LZW compression. Next, we 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.

4.1 LZW compression and decompression

The main purpose of this section is to review LZW compression/decompression algorithms, which are shown in Section 13 in [11]. In addition, we prove several important properties

and evaluate the running time of LZW decompression algorithm.

The LZW (Lempel-Ziv & Welch) [27] 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 in Algorithm 4.1.

Algorithm 4.1 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: **end for**
6: 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 4.1 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.

Table 4.1: 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

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 in Algorithm 4.2.

Table 4.2 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

Algorithm 4.2 LZW decompression algorithm

```
1: Output(C(y_0));
2: for i ← 1 to m − 1 do
3:    if thenC(y_i) has been registered
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}));
7:    end if
8: end for
```

that *cbcbcbcda* is output by this algorithm, and the generated code table is the same as that generated by LZW compression.

Table 4.2: 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

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 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. Parallelizing LZW compression is very hard. So for, the speedup factor of GPU LZW compression over the sequential algorithm can be only 3 shown in section .

4.2 GPU implementation of LZW compression for TIFF images

In this section, we focus on LZW compression of an image into a TIFF image file.

4.2.1 GPU implementation

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 4.1, 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 [18]. It also has an image directory containing pointers to the actual image data. For LZW compression, an original 8-bit gray-scale image

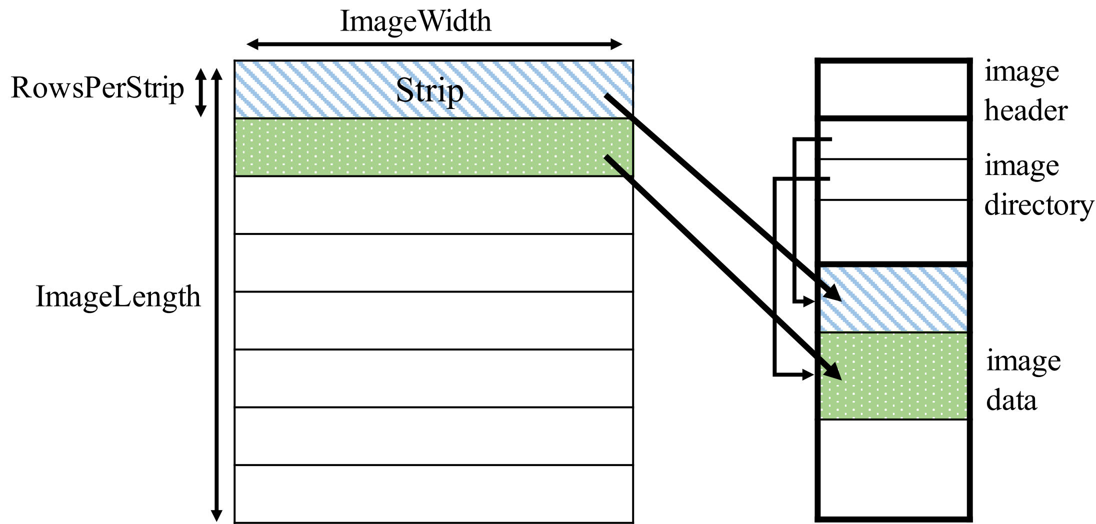


Figure 4.1: An image and TIFF image file

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.

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 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.

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

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 $(h(i, x) + 501i) \wedge \text{0x3FFF}$ is read for i = 1, 2,... 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 $2^{14} \times 2 = 32$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 [28]. 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 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 [29, 30]. 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.

4.2.2 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.

Crafts Flowers Graph

Figure 4.2: Three gray scale image with 4096×3072 pixels used for experiments

Table 4.3: The running time (in milliseconds) of LZW compression using a GPU and a CPU for three images

Images	compression ratio	GPU (transposed) Step 1	Step 2	Step 3	Step 4	All	CPU	Speed-up	GPU (non-transpose) Step2
“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

We have used three gray scale images with 4096×3072 pixels (Figure 4.2), 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 4.3 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 4.3 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

Table 4.4: The running time (in milliseconds) of two scenarios using our GPU and CPU implementations and libTIFF library for three images

(1)Scenario 1

Images	Compress on GPU	Transfer GPU→CPU	Writing CPU→SSD	All
"Crafts"	29.3	2.34	3.85	35.2
"Flowers"	22.23	1.44	2.80	26.0
"Graph"	10.99	0.40	0.38	11.3

(2)Scenario 2

Images	Transfer GPU→CPU	Compress on CPU	Writing CPU→SSD	All	Speed-up	libTIFF
"Crafts"	3.84	3.84	92.8	100.4	2.9	118.6
"Flowers"	3.82	65.4	2.74	71.9	2.8	105.0
"Graph"	3.88	33.3	0.28	37.5	3.3	46.1

Table 4.5: 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

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

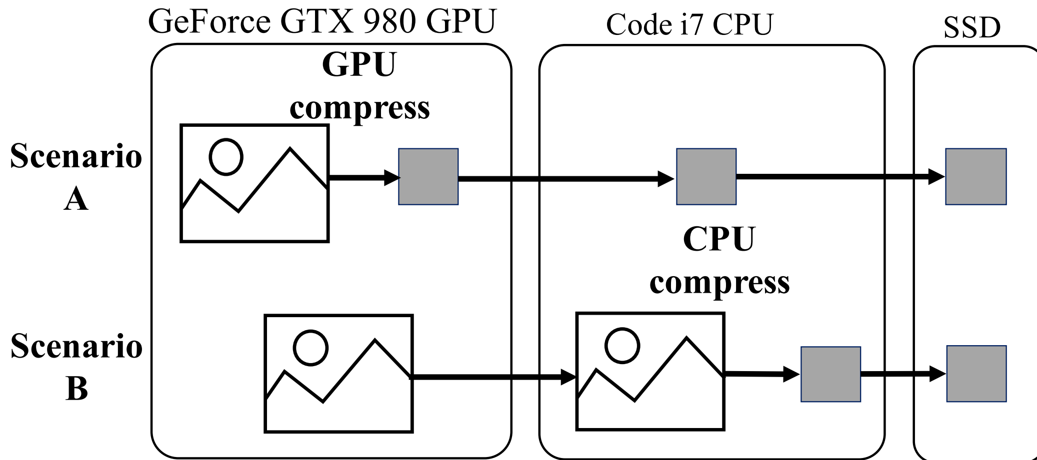


Figure 4.3: Two scenarios to archive an LZW-compressed image in the SSD

(Solid State Drive) connected to the host PC. We compare the following two scenarios as shown in Figure 4.3.

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 4.4 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 [31]. 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 4.5 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 [6] 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.

4.3 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. Table 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:

$$p(i) = \begin{cases} \text{NULL} & \text{if } 0 \le i \le k-1 \\ y_{i-k} & \text{if } k \le i \le k+m-1 \end{cases} \tag{4.1}$$

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:

$$C_l(i) = \begin{cases} \alpha(i) & \text{if } 0 \le i \le k-1 \\ \alpha(p^*(i+1)) & \text{if } k \le k+m-2 \end{cases} \tag{4.2}$$

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:

$$C(i) = C_l(p^{L(i)-1}(i)) \cdot C_l(p^{L(i)-2}(i)) \cdots C_l(p^0(i)). \tag{4.3}$$

Table 4.6 shows the values of p, p^*, L, C_l, and C for $Y = 214630$.

Table 4.6: 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*	-

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 $(k \leq i \leq k+m-1)$, 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 in Algorithm 4.3.

Algorithm 4.3 Step 1 of the parallel LZW decompression algorithm

1: **for do in parallel**$i \leftarrow 0$ to $k-1$
2: $p(i) \leftarrow$ NULL; $L(i) = 1$; $C_l(i) \leftarrow \alpha(i)$;
3: **end for**
4: **for do in parallel**$i \leftarrow 0$ to $k-1$
5: $p(i) \leftarrow y_{i-k}$; $p^*(i) \leftarrow y_{i-k}$;
6: while$(p(p^*(i)) \neq$ NULL$)$ $L(i) \leftarrow L(i) + 1$; $p^*(i) \leftarrow p(p^*(i))$;
7: **end for**
8: **for do in parallel**$i \leftarrow k$ to $k+m-2$
9: $C_l(i) \leftarrow \alpha(p^*(i+1))$;
10: **end for**

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 [12].

Step 2-B: Using one processor 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 $x_{s(i)-1}$ in Algorithm 4.4.

Algorithm 4.4 Step 2B of the parallel LZW decompression algorithm

```
1: for do in parallel i ← 0 to m − 1
2:     y(i) ← y_i;
3:     for do j ← 1 to L(y_i)
4:         x_{s(i)−j} ← C_l(y(i));
5:         y(i) ← p(y(i));
6:     end for
7: end for
```

Table 4.7 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 4.7: 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 $p(i) = \text{NULL}$, $L(i) = 1$ 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 [12]. 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.

4.4 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.

4.4.1 TIFF images

In this subsection, we will review TIFF format file. 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 4.1, 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 [11]. 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 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.

4.4.2 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 4.3 to decompress this image stored in the global memory.

Figure 4.4 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

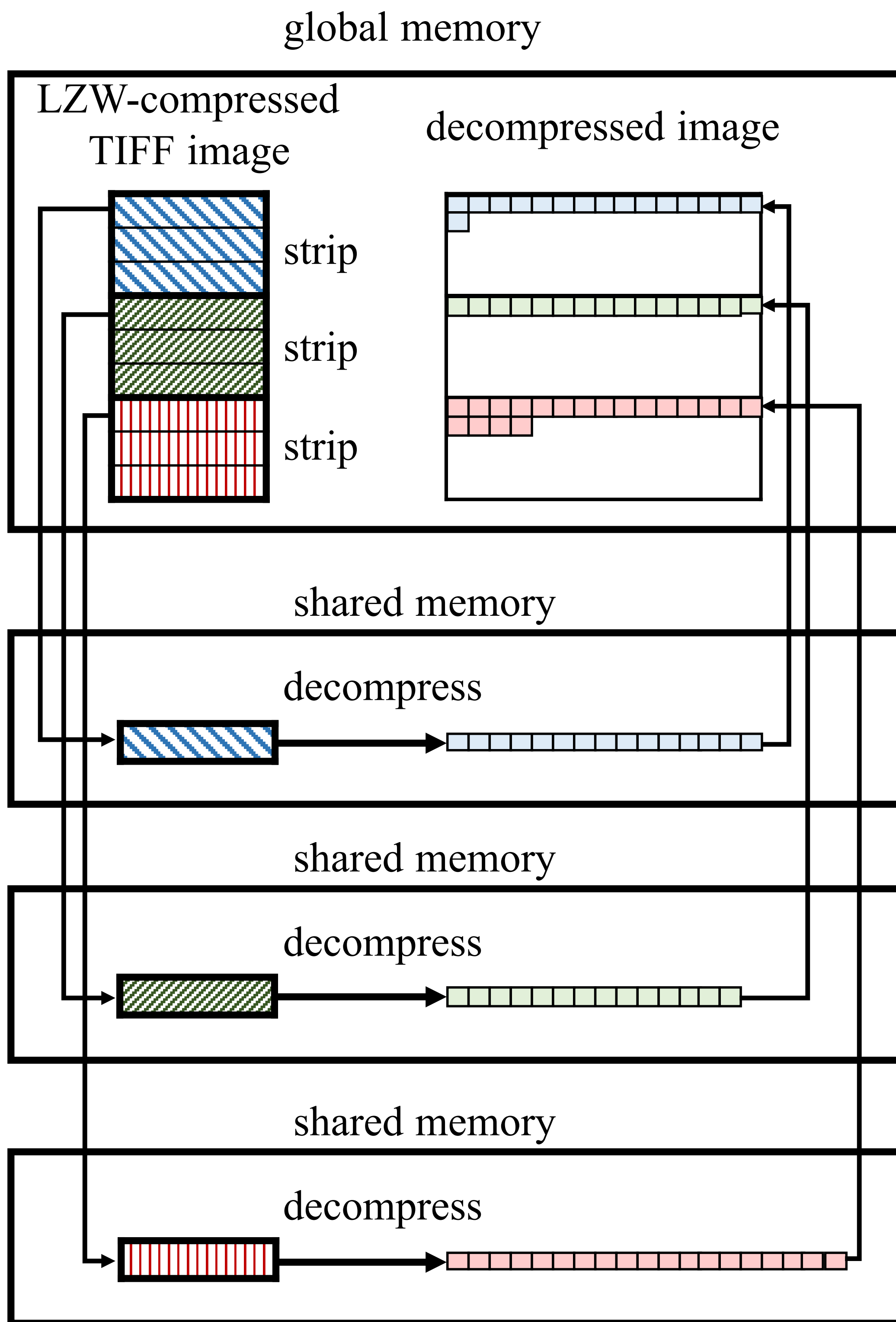


Figure 4.4: GPU implementation of parallel LZW decompression

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 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 $(1024 \leq i \leq 2047)$ 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 as $p^*(i) < 1024$ 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 [30, 32].

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%.

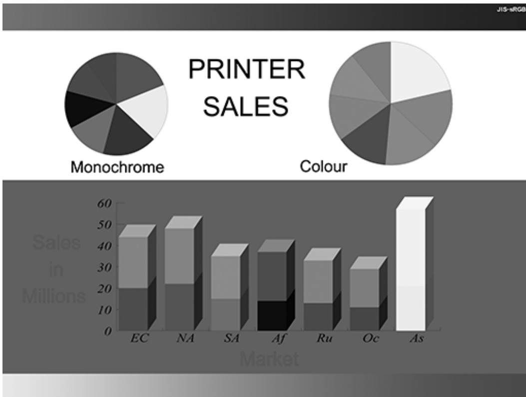
4.5 Experimental results

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.

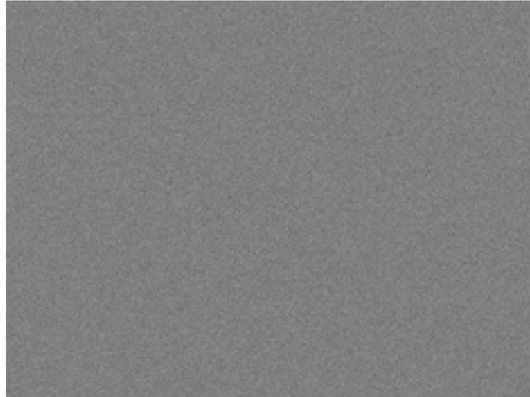
We have used gray scale images, Crafts, Flowers, and Graph with 4096×3072 pixels in Figure 4.5, which are converted from JIS X 9204-2004 standard color image data. We also use two gray scale images, Random and Black with the same size. The intensity level of each pixel of Random is selected from $[0, 255]$ 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.

Crafts:59.9%

Flowers:42.7%



Graph:2.6%


Random:136.6%

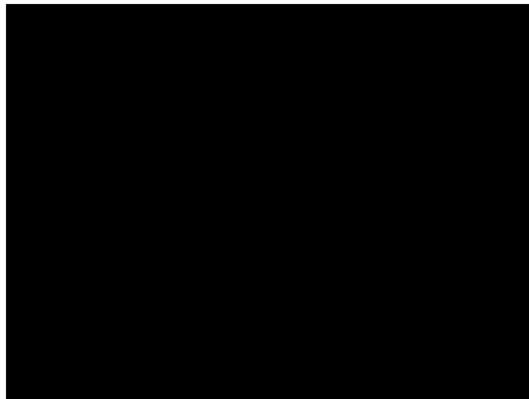

Black:0.6%

Figure 4.5: Three gray scale image with 4096 × 3072 pixels used for experiments

We first evaluated the running time of GPU decompression using Core i7 CPU and GeForce GTX 980 GPU. Table 4.8 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 4.8: 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 4.6 illustrating the three scenarios.

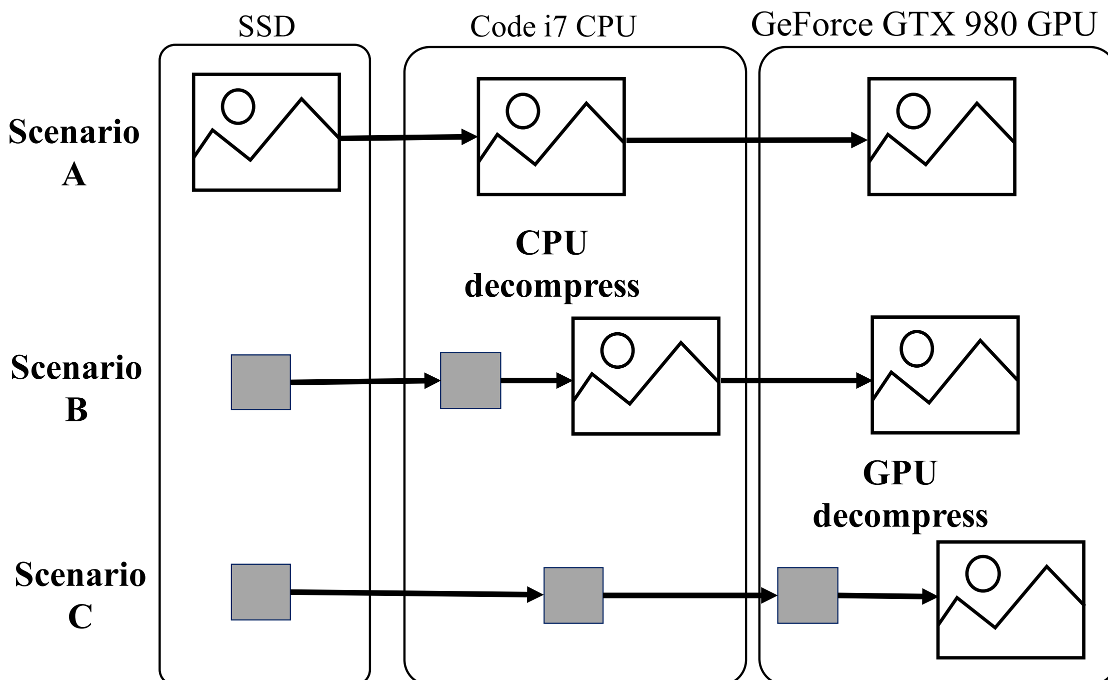


Figure 4.6: Scenarios A, B, and C

Table 4.9 shows the running time of three scenarios. From Table 4.9 (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 4.9 (2) and (3), the time necessary to load a non-compressed image is much larger than 10 milliseconds. Table 4.9 (2) shows the running time using our LZW decompression sequential algorithm. Table 4.9 (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 4.9 (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.

Table 4.9: 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

Chapter 5

Adaptive loss-less data compression method optimized for GPU decompression

In this chapter, we present a new lossless data compression method that we call Adaptive Loss-Less (ALL) data compression. It is designed so that the data compression ratio is moderate but decompression can be performed very efficiently on the GPU.

5.1 ALL: Adaptive Loss-Less data compression

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

5.1.1 Outline of ALL coding

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

In *run-length coding*, a substring with the same character is encoded in a run-length code with a pair (character, length). For example, run-length codes (A,4)(B,3)(C,2) are decoded

to AAAABBBCC. ALL codes include run-length codes. The run-length coding achieves good compression ratio if the input character has many long runs.

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

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

uncompressed characters of the first code in the segment.

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

A magic string can also be used to encode a random string with high entropy, which there is no way to compress in smaller size. Usually, the total size of codes for a random string may be larger than the input uncompressed data due to the overhead of coding. Magic strings can minimize the overhead. An interval code in ALL coding supports length up to 3408. So, a random string of length 3408 can be encoded using a magic string of length 3408 and only one interval code (0,3408). Thus, the overhead of the random string of length 3408 is only one additional interval code.

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

5.1.2 ALL codes

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

1-byte code a 1-byte word

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

2-byte code a 2-byte word

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

3-byte code a 2-byte word plus a 1-byte word

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

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

ALL data compression uses 3 types of codes as follows:

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

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

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

The reader should refer Table 5.1 that summarizes five types of ALL codes.

Table 5.1: ALL codes: p is the previous character and $d_0 d_1 \cdots d_{4095}$ are 4096 characters in the dictionary

<table>
<tr><th>Codes</th><th>words</th><th>length $\boldsymbol{L}$</th><th>string to be decoded</th></tr>
<tr><td>SC
Single Character</td><td>[c]</td><td>1</td><td>c</td></tr>
<tr><td>SRL
Short Run-Length</td><td>[111111111111][l]</td><td rowspan="2">$l+2$</td><td>$pp \cdots p$</td></tr>
<tr><td>SI
Short Interval</td><td>[t][l]</td><td>$d_t d_{t+1} \cdots d_{t+L-1}$</td></tr>
<tr><td>LRL
Long Run-Length</td><td>[111111111111][1111] [c]</td><td rowspan="2">$c+18$ if $c \leq 46$
$16c-672$ if $c \geq 47$</td><td>$pp \cdots p$</td></tr>
<tr><td>LI
Long Interval</td><td>[t][1111] [c]</td><td>$d_t d_{t+1} \cdots d_{t+L-1}$</td></tr>
</table>

5.1.3 Adaptive Dictionary

A sequence of ALL codes is partitioned into *segments* of 32 words each. Since each code has 1 or 2 words, each segment involves 16-32 codes. The same dictionary is used for all codes of a segment. A dictionary consists of *a previous string* and *a magic string*. The previous string is a string of uncompressed 4096 characters obtained by decoding codes in previous segments. If it has less than 4096 characters, then it is right justified and 0's are filled to obtain a string of length 4096. In particular, the previous string of the first segment is a run of 4096 0's, because the first segment has no "previous string." Let $p_0 p_1 \cdots p_{4095}$ be the previous string of 4096 characters, and let $w_0 w_1 \cdots w_{v-1}$ be the magic string of v characters. The dictionary can be obtained by overwriting the magic string from the beginning of the previous string, that is, the dictionary is $w_0 w_1 \cdots w_{v-1} p_v p_{v+1} \cdots p_{4095}$. A magic string is a sequence of 8-bit characters determined in the process of compression to attain better compression ratio. Usually, it includes substrings appearing frequently in an uncompressed input.

Figure 5.1 shows an example of ALL codes and the decoded string. In the figure, we assume that the length of a previous string is 32 and a magic string of length 4 is used. Thus, the first 4 characters of the dictionary is the magic string. The remaining characters of the dictionary are 28 characters of the previous string. Five ALL codes in the figure are decoded as follows.

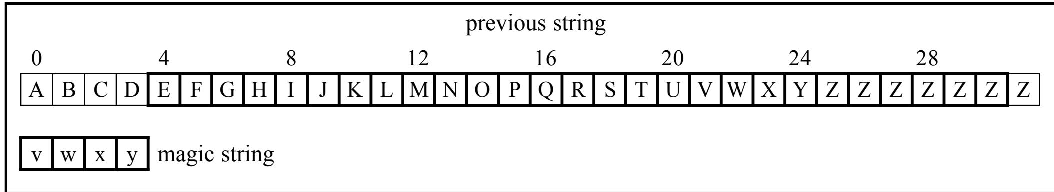
1. SI code with offset 1 and length 2 + 1 is decoded to "wxy."
2. SC code with character "z" is decoded to "z."
3. SI code with offset 0 and length 2 + 1 is decoded to "vwx."
4. LI code with offset 7 and length 0+18 is decoded to "HIJKLMNOPQRSTUVWXY."
5. SRL code with length 2 + 2 is decoded to "YYYY."

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

5.1.4 Data block: encoded data for a strip

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

Let $a_0a_1 \cdots a_{m-1}$ denote the words for ALL codes in a strip, where m is the number of words in this strip. We use an array of *word identifiers* $b_0b_1 \cdots b_{m-1}$ such that $b_i = 0$ $(0 \leq i \leq m-1)$ if a_i is a 1-byte word and $b_i = 1$ if a_i is a 2-byte word. Clearly, the strip



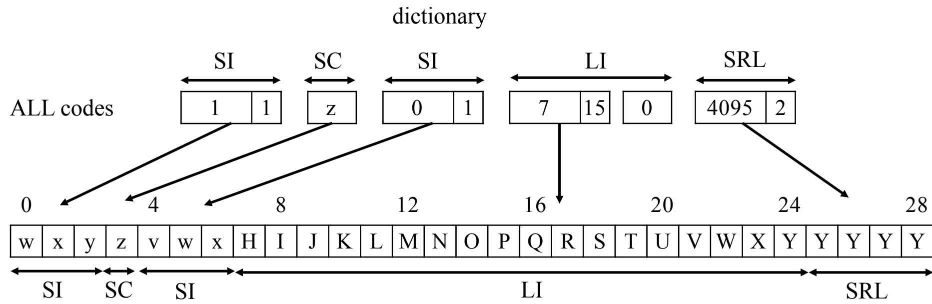


Figure 5.1: An example of ALL codes and the decoded string

has $k = \lceil \frac{m}{32} \rceil$ segments. Magic strings for the k segments are concatenated and stored as a single magic string. To identify each magic string, we use *magic identifiers* $q_0 q_1 \cdots q_{k-1}$ such that $q_i = 0$ $(0 \leq i \leq k-1)$ if the length of the magic string of the i-th segment is 0 and $q_i = 1$ if it is not 0. An array of 12-bit integers is used to specify the length of each non-zero length magic string. If the value stored in a 12-bit integer for i-th magic string is l_i, then this magic string has $l_i + 1$ characters. Since the length of each magic string is in range [1,4096], a 12-bit integer field is sufficient to store the length of a magic string.

Figure 5.2 illustrates a format of data block. The format has *word counts* (16 bits; the total number of words minus 1), *word byte counts* (16 bits; the total size of words in bytes minus 1), *magic string counts* (11 bits; the number of magic strings) and *predictor option flag* (1 bit). Since the total number of words and the total size of words are in the range [1,65536], their values minus 1 are stored in 16 bits each. Also, the number k of strips is less than $\frac{65536}{32} = 2048$, and the number of magic strings is stored in 11 bits. Let $x_0 x_1 \cdots x_{65535}$

denote the 65536 characters of an uncompressed strip. If the predictor option flag is set, then $y_0 y_1 \cdots y_{65535}$ such that $y_0 = x_0$ and $y_i = (x_i - x_{i-1}) \bmod 256$ $(1 \leq i \leq 65535)$ are used for ALL data compression. Note that $x_i = (y_0 + y_1 + \cdots + y_i) \bmod 256$ for all i. Thus, after $y_0 y_1 \cdots y_{65535}$ is obtained by ALL decompression, an original sequence $x_0 x_1 \cdots x_{65535}$ can be computed by the prefix-sums for $y_0 y_1 \cdots y_{65535}$. Usually, the compression ratio may be better for image data if the predictor is used. On the other hand, the predictor should not be used for most text data.

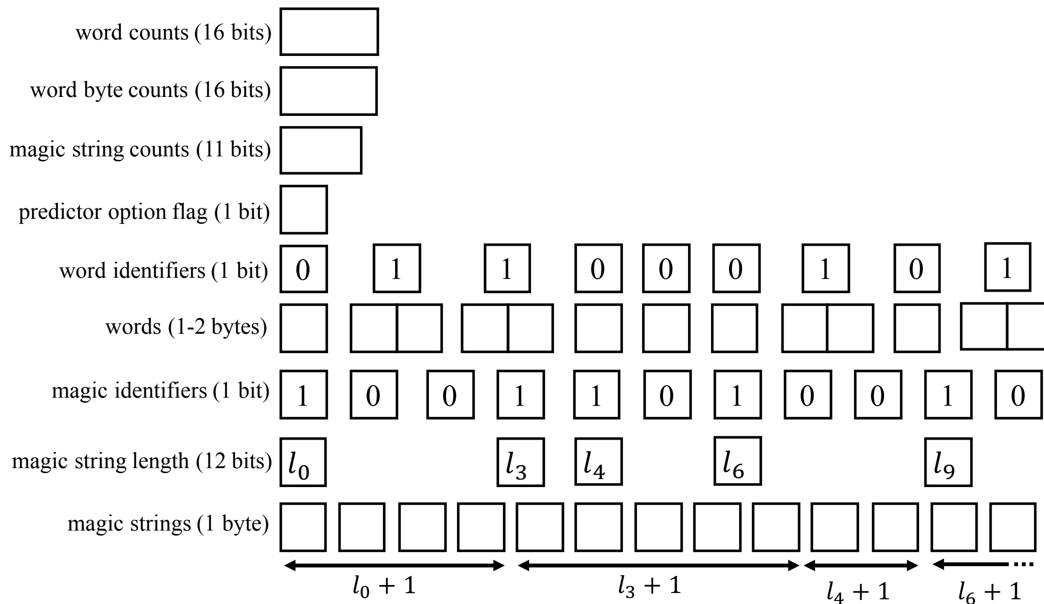


Figure 5.2: A data block for a strip of 65536 bytes

5.1.5 ALL file format

ALL file format begins with *the header* followed by *the body*. The body simply contains all data blocks. The header has *strip count* field storing strip count minus 1. It also has an array of *strip bytes* each of which stores the size of the data block for each strip in bytes minus 1. Since a strip has 65536 bytes, it makes no sense that the data block has more

than 65536 bytes. Thus, if the value of a strip byte is 65535, then the data block stores uncompressed 65536 characters of a strip as they are without using ALL codes.

Figure 5.3 illustrates data stored in the header. We assume that the strip count filed has 32 bits and thus the maximum number of strips is 2^{32}. Since each strip encodes $65536 = 2^{16}$ bytes, a single ALL file can encode up to 2^{48} =256T-byte input data in theory.

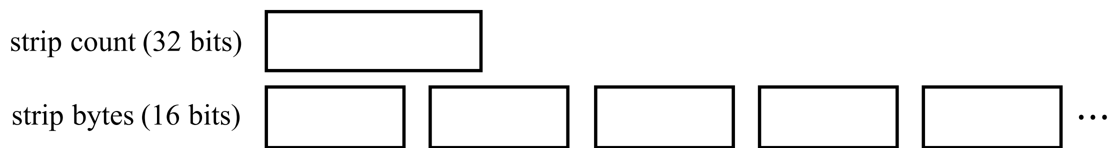


Figure 5.3: ALL file header

5.1.6 Compression and decompression algorithms

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

For sequential ALL compression, we will show how codes of a segment in a strip are generated. We first generate codes for a current uncompressed string in a strip using a previous string of length 4096 as a dictionary. Note that this dictionary has no magic string. This can be done in the same way as LZSS compression [15]. More specifically, we find the longest match and the longest run of the prefix of a current uncompressed string. We select the longest of the two. This procedure is repeated until codes of 32 words are generated. We compute the current compression ratio,

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

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

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

We replace such consecutive codes by one interval code using a magic string. For this purpose, we compute a common superstring that contains all consecutive codes as a substring. For example, for strings of consecutive codes BAA , AAB , BBA , ABA , and ABB, a string ABAABBA is a common superstring, because it contains every string as a substring. To achieve better compression ratio, we should use the shortest common superstring as a magic string. However, the problem of finding the shortest common superstring is NP-complete [33] and it is not realistic to solve this problem for compression. Hence, we use a greedy approximation algorithm for the shortest common superstring problem shown in [34]. We use an approximation solution for the shortest common superstring as a magic string and start over generation of codes with 32 words again. In other words, we use the obtained magic string and generate codes with 32 words as before. After that, we evaluate the current compression ratio again. If this current compression ratio of codes is not better than the previous one, we do not use this magic string and use the previous codes computed before. Otherwise, we continue finding a better magic string by repeating the same procedure until no more improvement of the data compression ratio is possible.

Sequential ALL decompression can be done very easily in linear time. We show how codes of 32 words for a segment are decoded and the resulting string is written in the

corresponding output buffer. We assume that previous string of the segment has been already written in the output buffer. We first need to compute the dictionary for a segment. Recall that the dictionary can be obtained by replacing the prefix of the previous string of 4096 characters by the magic string. The reader may think that copy operation with a large overhead is necessary for this replacement. However, we do not have to copy the magic string. We can think that the address space of the dictionary is separated such that addresses $[0, v-1]$ are arranged in the magic string and those of $[v, 4095]$ are the previous string, where v is the length of the magic string. Thus, dictionary accesses to address $[0, v-1]$ and $[v, 4095]$ are redirect to the magic string and the previous string, respectively. Hence, each character in the dictionary can be accessed efficiently in $O(1)$ time and the decoded string of every code can be written in the output buffer in time linear to the length.

5.2 GPU implementation for ALL decompression

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

5.2.1 Parallel prefix scan on the GPU

Before showing a GPU implementation for ALL decompression, we briefly explain a well-known technique called *the parallel prefix scan* [35, 30] on the GPU, which computes the prefix-sums of integers. The parallel prefix scan is used to identify the data block of a strip, to identify words of a segment, and to compute the writing offset of each code.

Let $a_0, a_1, \ldots, a_{31}$ be 32 integers of 32 bits each. We assume that 32-bit register $A[i]$ of

each thread i stores a_i $(0 \le i \le 31)$. The prefix sums $\hat{a}_i = a_0 + a_1 + \cdots + a_i$ for all i $(0 \le i \le 31)$ can be computed in 5 iterations described in Algorithm 5.1. Figure 5.4 illustrates addition

Algorithm 5.1 Parallel prefix scan

1: **for do** $k \leftarrow 0$ to 4
2:     **for do in parallel** $i \leftarrow 0$ to 31 do in parallel
3:         thread i performs $A[i] \leftarrow A[i] + A[i - 2^k]$ if $i \ge 2^k$;
4:     **end for**
5: **end for**

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

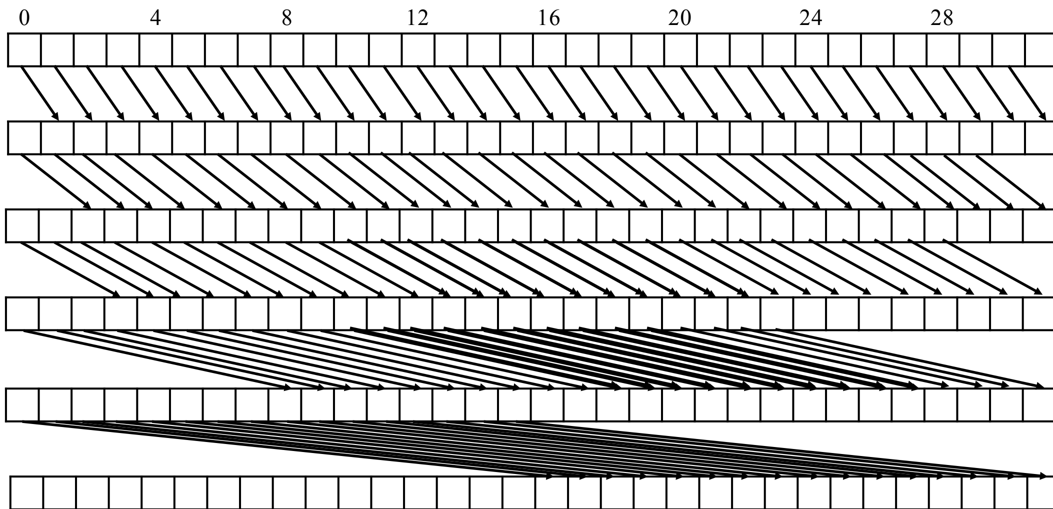


Figure 5.4: Parallel prefix scan

The parallel prefix scan can be implemented very efficiently using warp shuffle function [6], which exchanges the values stored in 32-bit registers of different threads in the same warp. Using warp shuffle function shfl_up(), the value stored in $A[i - 2^k]$ can be transferred to thread i with very small overhead.

Next, we will show that the prefix-sums of 128 8-bit integers $a_0, a_1, \ldots, a_{127}$ under modulo 256 very efficiently using the parallel prefix scan. It should be clear that they can be

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

5.2.2 CUDA kernel for ALL decompression

We show how CUDA kernel for ALL decompression invokes CUDA blocks. A CUDA kernel with CUDA blocks with 64 threads, that is, two warps of 32 threads each, are invoked for decompression. A warp of 32 threads is assigned to a data block, which contains ALL codes for a strip with 4096 characters. Two warps in a CUDA block work independently. Since the compute capability of GeForce GTX 1080 is 6.1, each streaming multiprocessor can have up to 2048 resident threads and 32 resident CUDA blocks [6]. Thus, each CUDA block must have 64 or more blocks to achieve 100% occupancy. A warp of 32 threads

assigned to a data block decodes the codes and writes decompressed string in the output buffer in the global memory of the GPU.

We will show how each warp is assigned to a data block. Let k be the number of strips (or data blocks). An integer variable c in the global memory is used as a counter to arrange a warp to a data block. After the CUDA kernel is invoked, the first thread of every warp calls function atomicAdd(&c,1), which increments c as an atomic operation and returns the value of c before addition [6]. If the return value i of atomicAdd satisfies $i < k$, then the warp is assigned to the i-th data block and works for decompressing the block. If $i \geq k$ then the warp terminates. Since each function call atomicAdd(&c,1) returns 0, 1, 2, ... in turn, every data block can be assigned a warp and is decompressed by it. Let $s_0, s_1, \ldots, s_{k-1}$ be the strip bytes stored in the header and $\hat{s}_0, \hat{s}_1, \ldots$ be the prefix sums such that $\hat{s}_i = (s_0 + 1) + (s_1 + 1) \cdots + (s_i + 1)$ for all i ($0 \leq i \leq k - 1$). Since each i-th data block has $s_i + 1$ bytes, it is allocated from offset $\hat{s}_{i-1}$ of the body. Thus, a warp can identify the assigned data block in the global memory if the value of $\hat{s}_{i-1}$ is computed. We will show how $\hat{s}_{i-1}$ is computed by the warp. A warp of 32 threads finds minimum j such that $\hat{s}_{i-j}$ has been already computed. This can be done by checking 32 values of p's in turn and merging the results by ballot function call of CUDA [6]. For such j, the sum $\hat{s}_{i-j} + (s_{i-j+1} + 1) + (s_{i-j+2} + 1) + \cdots + (s_{i-1} + 1)$, which is equal to $\hat{s}_{i-1}$, is computed by the parallel prefix scan (Figure 5.4). After that, decompression for i-th data block stored in offset $\hat{s}_{i-1}$ is performed. When decompression is completed, the first thread in the warp calls atomicAdd(&c,1) and performs decompression of strip i' again if the return value i' is smaller than k. Otherwise, the warp terminates. When all warps terminate, decompression

of all data blocks is completed.

5.2.3 A CUDA block decompressing a data block

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

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

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

Stage 3 Write the decoded strings in the output buffer.

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

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

In Stage 1, 32 threads read 32 word identifiers of a segment. Let $b_0, b_1, \ldots, b_{31}$ be these 32 word identifiers. They compute the prefix sums $\hat{b}_i = (b_0 + 1) + (b_1 + 1) + \cdots + (b_i + 1)$ for all i ($0 \leq i \leq 31$) by the parallel prefix scan (Figure 5.4). Basically, each i-th thread

($0 \leq i \leq 31$) is assigned to the i-th word stored in offset $\hat{b}_{i-1}$, and works for decoding the code corresponding to the word.

In Stage 2, each thread identifies the code type (SC, SRL, SI, LRL, or LI) and the reading offset t and length L of the assigned code. Since the offset of each code has been identified, this can be done in an obvious way using Table 5.1. After that, the writing offset of each code is determined as follows. Since each code is decoded to a string of length L, the prefix-sums of lengths correspond to the writing offset. More specifically, let $L_0, L_1, \ldots, L_{31}$ be the lengths of codes corresponding to 32 words. If a code is a 3-byte code with two words, let the length of the second word be 0 for convenience. The 32 threads in a warp compute the prefix sums $\hat{L}_i = L_0 + L_1 + \cdots + L_i$ for all i ($0 \leq i \leq 31$) by the parallel prefix scan (Figure 5.4). Clearly, the writing offset of each i-th code is $\hat{L}_{i-1}$.

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

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

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

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

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

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

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

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

5.2.4 Performance issues of the GPU implementation for ALL decompression

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

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

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

To maximize the performance of a GPU implementation, we should invoke as many threads and CUDA blocks as possible. For this purpose, the occupancy must be 100%. In our GPU implementation, a CUDA block with 64 threads uses no shared memory. Each thread uses 31 32-bit registers. Each streaming multiprocessor of GeForce GTX 1080 (compute capability 6.1) has 2048 resident threads, 32 resident CUDA blocks and 64K 32-bit registers [6]. Hence, a CUDA kernel can dispatch 32 CUDA blocks in each streaming multiprocessor, because they use $32 \times 64 = 2048$ resident threads and $2048 \times 31 = 62\text{K}$ 32-bit resisters. Thus, the occupancy of our implementation is 100%. Also, since GeForce GTX 1080 has 20 streaming multiprocessors, our GPU implementation can dispatch $32 \times 20 = 640$ CUDA blocks with $2048 \times 20 = 40\text{K}$ threads at the same time. Since each strip is decoded using a warp of 32 threads, 1280 strips can be decoded in parallel. Thus, resident threads of GeForce GTX 1080 are fully utilized if the size of uncompressed data is $1280 \times 65536 = 80\text{M}$ bytes or larger.

5.3 Experimental results

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

ALL Our Adaptive Loss-Less compression presented in this paper

Gompresso LZ77-based [16] compression implemented in the GPU shown in [25]

CULZSS GPU implementation of LZSS [15] implemented in the GPU [24]

LZW GPU implementation of LZW [10, 27] shown in [36]

For evaluating the performance including the data compression ratio and the running time for decompression, we have used a data set in Table 5.2. We have used five images and five text data. Figure 5.5 shows three JIS standard images, Crafts, Flowers, and Graph. The pixel value of every pixel of Random image is selected independently at random from $[0, 255]$. Hence, it is not possible to generate a smaller compressed data than this uncompressed Random image. Every pixel of Black image is 0 and so the compression ratio is the minimum. Thus, Random and Black images are extreme for data compression.

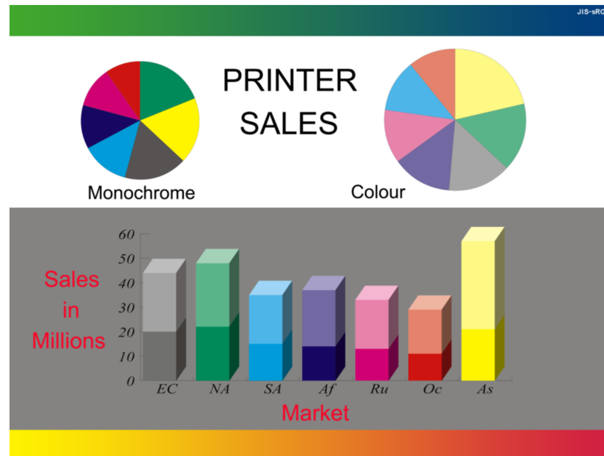
Table 5.2: Data set used for evaluating the performance

Data	Size (Mbytes)	Data contents
Crafts	37.6	
Flowers	37.6	Standard RGB images (JIS X 9204-2004)
Graph	37.6	of size 4096×3072
Random	37.6	Random image of size 4096×3072
Black	37.6	Black image of size 4096×3072
wiki	1000	XML dump of a 1G-byte subset of English Wikipedia
matrix	808	Hollywood-2009 sparse matrix in CVS format
linux-2.4.5.tar	114	Source codes of Linux kernel
rctail96	121	Reuters news in XML
w3c2	109	XML documents for w3c

Table 5.3 shows the data compression ratio, the ratio between the uncompressed size and compressed size. The best compression ratios among the four data compression methods are underlined. The data compression ratio of ALL is better than the others for almost all data. For two data, wiki and w3c2, Gompresso is better than ALL, but the difference is quite small. Since Random image cannot be compressed to smaller data, the compression

Crafts

Flowers



Graphs

Figure 5.5: Three standard images used for experiments

ratio is larger than 1. The overhead of compression of ALL is very small for such data. Also, for data with low entropy such as Crafts and Black images, ALL is much better than the others, because run-length encoding works efficiently for long runs in these images.

Table 5.4 shows the running time of decompression. The running time on GeForce GTX 1080 is evaluated for four data compression methods. We also evaluated the running time of ALL decompression on Intel i7-4790 (3.67GHz) CPU to see the acceleration ratio over

Table 5.3: Data compression ratio by four compression methods

Data	Size (Mbytes)	ALL	Gompresso	CULZSS	LZW
Crafts	37.6	<u>0.644</u>	0.781	0.863	0.808
Flowers	37.6	<u>0.434</u>	0.682	0.768	0.657
Graph	37.6	<u>0.0168</u>	0.0480	0.0710	0.0375
Random	37.6	<u>1.0002</u>	1.11	1.20	1.368
Black	37.6	<u>0.00110</u>	0.0370	0.0421	0.00602
wiki	1000	0.534	<u>0.532</u>	0.555	0.540
matrix	808	<u>0.428</u>	0.431	0.454	0.455
linux-2.4.5.tar	114	<u>0.446</u>	0.485	0.573	0.464
rctail96	121	0.396	<u>0.395</u>	0.437	0.489
w3c2	109	<u>0.292</u>	0.326	0.296	0.463

the GPU. Note that, we have implemented a sequential algorithm for ALL using a single core of Intel i7-4790 CPU and we do not use multicores and hyperthread of the CPU. Since the comparison of computation powers of GPU and CPU is out of scope of this paper, we did not implement multithreaded algorithm in the CPU. However, we can say that the CPU implementation cannot be accelerated more than 8 times by multithreaded implementation using 8 hyperthreads of Intel i7-4790 CPU. The table shows the running time of ALL using the CPU and the GPU. It also shows the acceleration ratio of GPU over CPU. From the table, it achieves high acceleration ratio of 84.0-231. The high acceleration ratio implies that our implementation of ALL decompression on GPU is highly parallelized and runs very efficiently. It also shows the ratios of running time between GPU implementation of ALL and those of the others. Since all ratios are larger than 1, ALL GPU implementation always runs faster than the others. We also evaluated the ratio of the total size of original data compressed using each of five types of code showed in table 5.5. Since all the data blocks in Random image stores uncompressed original data, Random does not have ALL codes. From this table, text data generate Run-Length codes fewer than image data. This

Table 5.4: The running time in milliseconds for decompression

Data	ALL			Gompresso		CULZSS		LZW	
	CPU	GPU	$\frac{CPU}{GPU}$	GPU	$\frac{Gomp}{ALL}$	GPU	$\frac{CULZSS}{ALL}$	GPU	$\frac{LZW}{ALL}$
Crafts	116.2	1.167	99.5	1.418	1.22	4.113	3.52	2.477	2.12
Flowers	92.73	0.8447	110	2.068	2.45	4.096	4.85	1.255	1.49
Graph	56.21	0.4862	116	4.213	8.67	2.343	4.82	1.531	3.15
Random	87.78	0.3792	231	3.872	10.2	8.918	23.5	3.921	10.3
Black	45.96	0.4605	99.8	5.761	12.5	3.231	7.02	3.729	8.10
wiki	2962	28.05	106	50.41	1.80	218.8	7.80	56.92	2.03
matrix	1932	19.57	98.8	27.12	1.39	87.60	4.48	25.79	1.32
linux-2.4.5.tar	273.6	2.344	117	3.387	1.44	11.29	4.82	3.030	1.29
rctail96	252.1	2.244	112	3.354	1.49	11.43	5.09	3.248	1.45
w3c2	165.8	1.974	84.0	2.803	1.42	10.92	5.53	2.536	1.28

Table 5.5: The ratio of the total size of original data compressed using each of five types of code

	compression ratio	Five types of codes				
		SC	SI	LI	SRL	LRL
Crafts	64.40 %	26.3 %	61.0 %	8.51 %	1.59 %	2.65 %
Flowers	43.50 %	6.96 %	89.1 %	0.44 %	1.44 %	2.03 %
Graph	1.68 %	1.09 %	1.61 %	91.10 %	0.102 %	6.05 %
Random	100.02 %	0 %	0 %	0 %	0 %	0 %
Black	0.11 %	0 %	0 %	100 %	0 %	0 %
wiki	53.40 %	12.1 %	81.4 %	5.48 %	1.00 %	0.0801 %
matrix	42.80 %	5.23 %	94.0 %	0.171 %	0.562 %	0.0431 %
linux-2.4.5.tar	44.60 %	14.7 %	61.6 %	21.6 %	1.21 %	0.940 %
rctail96	39.60 %	14.1 %	60.2 %	25.1 %	0.66 %	0.0202 %
w3c2	29.20 %	9.86 %	60.6 %	28.9 %	0.523 %	0.112 %

results imply that SRL and LRL codes have a negligible effect on the compressed ratio, but may generate divergences when decompression on the GPU and affect the running time of decompression adversely.

We have evaluated the SSD-GPU loading time for three possible scenarios (Figure 4.6) as follows:

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

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

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

Table 5.6 shows the SSD-GPU loading time which is the time necessary to load uncompressed data in the global memory of the GPU from the SSD. The best total time of the three scenarios for each data is underlined. Since all images have the same size, the SSD-GPU loading times for Scenario A is proportional to the uncompressed data size. In Scenario B, the time for CPU decompression dominates data transfer time. Hence, it makes no sense to use CPU decompression to load data in the GPU. The total time of Scenario C is smaller than that of Scenario A except Random image, in which the compressed data is larger than the uncompressed data. However, the difference of the total time is very small, because such data can be decompressed on the GPU very efficiently. Hence, we can say that Scenario C should be selected for loading data in the GPU regardless of data. In particular, we should use GPU decompression to load data stored in the SSD, even if the storage capacity is so large that all uncompressed data can be stored.

Table 5.6: The SSD-GPU loading time in milliseconds using ALL decompression for three scenarios

	Crafts	Flowers	Graph	Rand.	Black	wiki	matrix	linux	rctail96	w3c2
Size (MB)	37.6	37.6	37.6	37.6	37.6	1000	808	114	121	109
Scenario A										
SSD→CPU	19.15	19.15	19.13	19.15	19.16	432.0	411.2	57.95	61.09	55.25
CPU→GPU	11.51	11.52	11.52	11.53	11.52	259.8	247.4	34.91	36.28	33.28
Total	30.66	30.67	30.65	30.68	30.68	691.8	658.6	92.86	97.38	88.53
Scenario B										
SSD→CPU	12.33	8.296	0.3210	19.14	0.02082	230.4	175.9	25.84	24.20	16.14
CPU decomp.	116.2	92.73	56.21	87.78	45.96	2962	1932	273.6	252.1	165.8
CPU→GPU	11.52	11.52	11.51	11.52	11.51	259.6	247.6	34.92	36.79	33.27
Total	140.1	112.5	68.04	118.4	57.49	3452	2356	334.3	313.1	215.2
Scenario C										
SSD→CPU	12.32	8.305	0.3230	19.16	0.02080	230.7	176.2	25.86	24.23	16.16
CPU→GPU	7.402	5.001	0.1940	11.52	0.01320	138.7	106.0	15.57	14.58	9.712
GPU decomp.	1.168	0.8447	0.4862	0.3792	0.4605	28.05	19.57	2.344	2.243	1.974
Total	20.88	14.15	1.003	31.06	0.4945	397.5	301.6	43.77	41.05	27.85

Chapter 6

Conclusion

In this dissertation, we have presented efficient GPU implementations of parallel LZW decompression algorithm and a new data compression method for decompression performed very efficiently on the GPU.

In Chapter 4, we have presented a work-optimal parallel LZW decompression algorithm on the CREW-PRAM and implemented in the GPU. The experimental results show 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.

In Chapter 5, we have presented a new data compression method called ALL (Adaptive Loss-Less) compression. Although the compression ratio is comparable, the ALL decompression on the GPU is much faster than previously published Gompresso, CULZSS and LZW decompression. We also provided the SSD-GPU loading time using ALL decompression. The experimental results show that the scenario, which transfers ALL-compressed data stored in the SSD to the GPU through a host and decompresses it in the GPU, is enough fast for practical use.

Bibliography

[1] Wen-mei W. Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[2] Duhu Man, Kenji Uda, Yasuaki Ito, and Koji Nakano. A GPU implementation of computing Euclidean distance map with efficient memory access. In *Proc. of International Conference on Networking and Computing*, pp. 68–76, Dec. 2011.

[3] Yuji Takeuchi, Daisuke Takafuji, Yasuaki Ito, and Koji Nakano. ASCII art generation using the local exhaustive search on the GPU. In *Proc. of International Symposium on Computing and Networking*, pp. 194–200, Dec. 2013.

[4] Akihiko Kasagi, Koji Nakano, and Yasuaki 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)*, pp. 251–250, Sept. 2014.

[5] Kazufumi Nishida, Yasuaki Ito, and Koji Nakano. Accelerating the dynamic programming for the matrix chain product on the GPU. In *Proc. of International Conference on Networking and Computing*, pp. 320–326, Dec. 2011.

[6] NVIDIA Corporation. NVIDIA CUDA C programming guide version 9.0, Mar 2017.

[7] Duhu Man, Kenji Uda, Hironobu Ueyama, Yasuaki Ito, and Koji 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] NVIDIA Corporation. NVIDIA CUDA C best practice guide version 3.1, 2010.

[9] Khalid Sayood. *Introduction to Data Compression, Fourth Edition*. Morgan Kaufmann, 2012.

[10] Terry Welch. High speed data compression and decompression apparatus and method. US patent 4558302, Dec. 1985.

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

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

[13] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karyapis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. The Benjamin/Cumming Publishing, 1994.

[14] Michael J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, 1994.

[15] James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, Vol. 29, No. 4, pp. 928–951, Oct. 1982.

[16] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, Vol. 23, No. 3, pp. 337–343, May 1977.

[17] Shmuel Tomi Klein and Yair Wiseman. Parallel Lempel Ziv coding. *Discrete Applied Mathematics*, Vol. 146, pp. 180 – 191, 2005.

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

[19] Anders L. V. Nicolaisen. *Algorithms for Compression on GPUs*. PhD thesis, Tecnical University of Denmark, Aug. 2015.

[20] Adnan Ozsoy and Martin Swany. CULZSS: LZSS lossless data compression on CUDA. In *Proc. of International Conference on Cluster Computing*, pp. 403–411, Sept. 2011.

[21] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. Technical report, DEC System Resarch Center, May 1994.

[22] Gzip. `http://www.gzip.org/`.

[23] bzip2. `http://www.bzip.org/`.

[24] Adnan Ozsoy and Martin Swany. CULZSS: LZSS lossless data compression on CUDA. In *Proc. International Conference on Cluster Computing*, pp. 403 – 411, Sept. 2011.

[25] Evangelia Sitaridi, Rene Mueller, Tim Kaldewey, Guy Lohman, and Kenneth A. Ross. Massively-parallel lossless data decompression. In *Proc. of Internatinal Conference on Parallel Processing*, pp. 242–247, Aug. 2016.

[26] Ritesh A. Patel, Yao Zhang, Jason Mak, and Andrew Davidson. Parallel lossless data compression on the GPU. In *Proc. of Innovative Parallel Computing (InPar)*, pp. 1–9, May 2012.

[27] Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, Vol. 17, No. 6, pp. 8–19, June 1984.

[28] Koji Nakano. Simple memory machine models for GPUs. *International Journal of Parallel, Emergent and Distributed Systems*, Vol. 29, No. 1, pp. 17–37, 2014.

[29] Koji 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.

[30] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Chapter 39. parallel prefix sum (scan) with CUDA. In *GPU Gems 3*. Addison-Wesley, 2007.

[31] libTIFF. `http://www.remotesensing.org/libtiff/`.

[32] Koji Nakano. Simple memory machine models for GPUs. In *Proc. of International Parallel and Distributed Processing Symposium Workshops*, pp. 788–797, May 2012.

[33] Kari-Jouko Rih and Esko Ukkonen. The shortest common supersequence problem over binary alphabet is np-complete. *Theoretical Computer Science*, Vol. 16, No. 2, pp. 187–198, 1981.

[34] C. B. Fraser and R. W. Irving. Approximation algorithms for the shortest common supersequence. *Nordic Journal of Computing*, Vol. 2, No. 3, pp. 303–325, 1995.

[35] Shunji Funasaka, Koji Nakano, and Yasuaki Ito. Light loss-less data compression, with GPU implementation. In *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (LNCS 10048)*, pp. 281–294, 2016.

[36] Shunji Funasaka, Koji Nakano, and Yasuaki Ito. Fully parallelized LZW decompression for CUDA-enabled GPUs. *IEICE Transactions on Information and Systems*, Vol. 99-D, No. 12, pp. 2986–2994, Dec. 2016.

Acknowledgement

First and foremost, I would like to show my deepest gratitude to my supervisor, Professor Koji Nakano for his continuous encouragement, advice and support. He is a respectable and resourceful scholar. His knowledge and research experience are in value through the whole period of my study. As a supervisor, he taught me skills and practices that will benefit my future research career.

I shall express sincere appreciation to my thesis committee members, Professor Satoshi Fujita and Associate Professor Yasuaki Ito for reviewing my dissertation.

I would also like to express my thanks to Assistant Professor Daisuke Takafuji for his support and continuous guidance in every stage of my study. My heartiest thanks go to all members of computer system laboratory. They were always kind and very keen to help.

I would express thanks to all the faculty members of the Department of Information Engineering of Hiroshima University.

Last but not least, I wish to express my thanks to my parents who have always supported me.

List of publications

Journals

J-1: Shunji Funasaka, Koji Nakano, and Yasuaki Ito, Fully Parallelized LZW decompression for CUDA-enabled GPUs, IEICE Transactions on Information and Systems, Vol. E99-D, No. 12, pp. 2986-2994, December 2016.

- Chapter 4: A GPU implementation of LZW compression and decompression

J-2: Shunji Funasaka, Koji Nakano, and Yasuaki Ito, Adaptive loss-less data compression method optimized for GPU decompression, Concurrency and Computation: Practice and Experience, Vol. 29, No. 24, e4283, November 2017.

- Chapter 5: Adaptive loss-less data compression method optimized for GPU decompression

International Conferences

C-1: Shunji Funasaka, Koji Nakano, and Yasuaki Ito, A Parallel Algorithm for LZW decompression, with GPU implementation, Proc. of 11th International Conference of Parallel Processing and Applied Mathematics (PPAM 2015, LNCS 9573), pp. 228-237, Krakow, Poland, September 2015.

- Chapter 4: A GPU implementation of LZW compression and decompression

C-2: Shunji Funasaka, Koji Nakano and Yasuaki Ito Fast LZW compression using a GPU, Proc. of International Symposium on Computing and Networking (CANDAR), pp. 303 - 308, Sapporo, Japan, December 2015.

- Chapter 4.2: GPU implementation of LZW compression for TIFF images

C-3: Shunji Funasaka, Koji Nakano and Yasuaki Ito Light Loss-Less Data Compression, with GPU implementation, Proc. of the 16th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP LNCS 10048), pp. 281-294, Granada, Spain, December 2016.
(Selected as best student paper)

- Chapter 5: Adaptive loss-less data compression method optimized for GPU decompression

Others

C-4: Shunji Funasaka, Koji Nakano and Yasuaki Ito, Single Kernel Soft Synchronization Technique for Task Arrays on CUDA-enabled GPUs, with Applications, Proc. of International Symposium on Computing and Networking (CANDAR), Aomori, Japan, November 2017.

C-5: Takuma Wada, Shunji Funasaka, Koji Nakano and Yasuaki Ito, A Hybrid Architecture for the Approximate String Matching on an FPGA, Proc. of International

Symposium on Computing and Networking (CANDAR), Aomori, Japan, November 2017.