GLZSS: LZSS Lossless Data Compression Can Be Faster

Yuan Zu[†] and Bei Hua[‡]
School of Computer Science and Technology
University of Science and Technology of China
Hefei, Anhui, 230027, China
Suzhou Institute for Advanced Study
University of Science and Technology of China
Suzhou, Jiangsu, 215123, China
[†]wyn@mail.ustc.edu.cn, [‡]bhua@ustc.edu.cn

ABSTRACT

The need for data compression has grown for better utilization of network bandwidth and data storage space. LZ77 is the most widely used data compression method, which has many variants in practical applications. The biggest obstacle that prevents data compression from being used in high-speed applications is its high computation overhead. In this paper, we focus on parallelizing LZSS that is a derivative of LZ77 on GPU using the NVIDIA CUDA framework to improve the compression speed. Based on in-depth understanding of LZSS's dictionary-based compression mechanism and GPU's architectural features, we propose an effective method to parallelize LZSS compression algorithm on GPU. The biggest merit of this method is that it eliminates threads serialization by carefully redesign the algorithm process. Experiments on an NVIDIA GTX 590 machine with 13 benchmark files from real world demonstrate the effectiveness of our method, which achieves 2x speedup over existing work.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms

Algorithm, Design, Experimentation, Performance

Keywords

CUDA, LZSS, Lossless Data Compression, GPU

1. INTRODUCTION

The era of big data is coming; huge volumes of data are being created, stored and transferred every moment. Data compression is a popular idea that helps reduce resources usage, such as storage and bandwidth.

GPGPU-7, March 01, 2014, Salt Lake City, UT, USA.

Data compression can be either lossy or lossless. Lossy compression reduces data by discarding (losing) some of it in order to minimize the amount of data that needs to be held, handled, and/or transmitted by a computer. Lossy compression is most commonly used to compress multimedia data (audio, video and still images), especially in applications such as streaming media and Internet telephony [7]. Lossless compression (e.g. Run-length encoding, Huffman coding, Lempel-Ziv algorithms and Burrows-Wheeler transform) reduces data by identifying and eliminating redundancy, and allows the original data to be perfectly reconstructed from the compressed data. Lossless compression is required for text and data files, such as bank records, text documents and source code. This paper focuses on lossless compression.

For 30 years, Lempel-Ziv-77 (LZ77) [18] family compression algorithms have been a cornerstone of lossless data compression, and have been widely applied in popular compression tools bundled with most Linux distributions, such as GZIP, ZLIB, and 7zip. Compression techniques in the LZ77 family maintain a dictionary to store the most recently scanned substrings, and operate on the input data by repeatedly searching for duplicate substrings in the dictionary and then outputting a series of pointers to the previous strings' occurrence in the dictionary. If the bits denoting the pointers are fewer than the bits of the matched strings, compression is achieved. Once the distance to a duplicate substring is beyond the size of dictionary, the potential match is missed. In many LZ77 applications, duplicate substrings search is the main bottleneck [16].

LZSS (Lempel-Ziv-Storer-Szymanski) is a derivative of LZ77. The main difference between LZ77 and LZSS is that in LZ77 the dictionary reference could actually be longer than the substring it replaces. In LZSS, such reference is omitted if the substring length is less than a "break even" point [8]. Many popular archives like PKZip, ARJ, RAR, ZOO and LHarc use LZSS rather than LZ77 as the primary compression algorithm; the encoding of literal characters and of length-distance pairs varies. In this paper, we focus on LZSS algorithm.

LZ77 relies on a dictionary to store the scanned strings; enlarging the dictionary and doing longest substring search can get higher compression ratio. However, as nothing comes for free, there is always a trade-off between compression ratio and compression speed. The two Genies - compression speed and compression ratio, are inherently hard to accommodate. There are several studies trying to remedy for this

problem by employing parallel processing. PIGZ [14] and Gilchrist [4] implement LZ77 on multi-core CPUs, using the same idea of splitting input strings among processors and having each substring compressed simultaneously.

Given that Graphics Processing Unit (GPU) typically has hundreds of processors in it, GPU is particularly suitable for executing LZ77 [3] [12] [13]. In the literature, the only work about parallelizing LZSS on GPU is given by Ozsoy et al. [12] [13]. The main idea is splitting the input string among GPU processors, and having a group of processors cooperate to compress the substrings independently. In their design much GPU computing power (128 CUDA cores in their implementation) are required to compress each sub block. Consequently, the compression speed they report is a few hundred megabits per second.

In this paper, we redesign the LZSS to make it match the GPU architecture, and then parallelize it on the GPU. The core of this work is a novel method that eliminates code path divergence in the algorithm, which is the biggest obstacle to algorithm parallelization on GPU.

1 Compared with [12] [13], sequential search buffer is replaced by a hash table to reduce the computation complexity of finding duplicate substrings.

2 A novel method that eliminates path divergences is provided to increase compression speed.

3 Compared with improved CULZSS [13], our solution achieves 2x speedup and 20% $\sim$ 196% compression ratio improvement.

The rest of this paper is organized as follows. Section 2 introduces GPU architecture, classic LZSS algorithm, and a GPU implementation of LZSS called CULZSS. Section 3 analyzes the drawbacks of CULZSS, based on which Section 4 presents a redesign of the LZSS called GLZSS-basic that is immune to these problems, and then Section 5 further eliminates the path divergence in GLZSS-basic. Section 6 evaluates our GLZSS algorithm. Section 7 summarizes the related works, and Section 8 concludes the paper.

2. BACKGROUND

2.1 GPU Architecture

We begin with a brief introduction on the micro-architecture of modern graphics processors. We focus on NVIDIA GPUs here, but our techniques are applicable to any similar GPU architecture.

2.1.1 Execution Architecture

GPU is the very kind of many-core processor possessing hundreds of processing cores, for example, NVIDIA GTX590 has 512 CUDA cores in one GPU card, and NVIDIA K20 even has 2048 CUDA cores. Each CUDA core is a scalar processor that GPU thread actually runs on. In the latest two generations of NVIDIA GPU with codename Fermi and codename Kepler, the CUDA cores are organized into Streaming Multiprocessors (SMs). Each SM features 32 CUDA cores in Fermi architecture, and 192 CUDA cores in Kepler architecture.

2.1.2 Scheduling

To hide the hierarchy of processors in GPU, NVIDIA develops CUDA, a programming framework which presents the programmer with the illusion of unlimited logical threads. These threads are sequentially divided into groups of 32 called warps and threads in a warp execute in lock step, which means execution of these threads are naturally synchronized. The warps are further grouped into blocks, and one SM can execute one or more thread blocks.

All threads in a warp share one instruction dispatch unit and run one copy of code (called kernel) following the single-instruction multiple-thread (SIMT) execution model. Threads in a warp may have different data-dependent code paths; however, these divergent paths are executed in serial, greatly degrading performance.

The NVIDIA GPU has two-level, distributed thread scheduler. "At the chip level, a global work distribution engine schedules thread blocks to various SMs, while at the SM level, each warp scheduler distributes warps of 32 threads to its execution units."[11]. The former scheduler runs only once, while the warp scheduler runs more frequently. Whenever a warp is currently stalled for some reason (for example, waiting on memory transactions), the warp scheduler chooses and executes a new available warp. Context-switch between warps is inexpensive as it is done by the hardware schedule engine.

2.1.3 Memory Subsystem

The memory space in GPU can be divided into two types, on-chip memory (i.e. SRAM, such as L2 cache, 64KB configurable L1 cache and shared memory) and off-chip memory (i.e. DRAM, such as global memory, local memory, etc.). L2 cache is shared by all the SMs, while L1 cache and shared memory are private to each SM or a block. Furthermore, the register file is private to each thread. CUDA now provides programmers with flexibility that the programmer can configure the size of shared memory and L1 cache as 48KB/16KB, 16KB/48KB or 32KB/32KB. Users are recommended to maximize the use of shared memory, as non-cached access to global or local memory may incur 400 $\sim$ 600 clock cycles of latency [10]. However, shared memory should not be overused, as excessive use can reduce the number of active warps and finally degrade overall system performance.

2.2 LZSS Algorithm

LZSS encodes data as a string consisting of the original literal characters and pointers to a dictionary. The already scanned data are serving as a dictionary, and prefix of unscanned data is searched in the dictionary for longest substring match. Only the matched substring that exceeds a certain minimum length L will be coded with the form of $(offset, substring\ length)$, where offset is the distance between the two duplicate substrings. If the matched substring is shorter than L, the first literal character is outputted as the form of $(1, literal)$. Successive unmatched characters can be merged into the form of $(length, literals)$. For example, if the input string is $S = abbaabbbaaabab$, and the minimum length L is 3, then the output is $S = [(4, abba), (4, 3), (5, 3), (4, abab)]$.

2.3 CULZSS

CULZSS algorithms [12] [13] are all built upon two data structures: a sliding window and an uncoded data buffer.

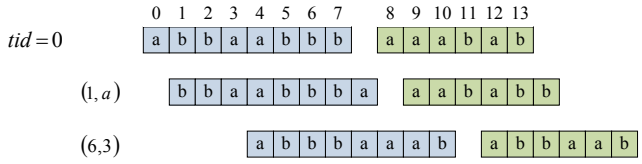

Figure 1: Execution of CULZSS in serial manner.

The sliding window stores the processed data that acts as the dictionary, and the uncoded data buffer contains the unprocessed input data. Take an input stream $S = abbaabbbaaababbaab$ as an example, and use a single thread model to explain the execution of CULZSS in Figure 1. Assume that the sliding window size and uncoded data buffer size are 8 bytes and 6 bytes, respectively, and set $L = 3$. Initially the sliding window contains the first 8 bytes of S as a dictionary, and the uncoded data buffer contains the next 6 bytes of S. Since the first 3 characters in the uncoded buffer (aaa) don't have a matched substring in the sliding window, a copy pointer $(1, a)$ is emitted, and both the sliding window and uncoded buffer shift right one byte. This time, the first 3 characters (aab) in uncoded buffer match a substring in dictionary at position 3, therefore a match pointer $(6, 3)$ is emitted, and both the sliding window and uncoded buffer shift right three bytes. This process repeats until the entire input data shifts out of the uncoded buffer.

CULZSS adopts two levels of parallelization. Firstly the input data is split into equally sized chunks and each chunk is assigned to a thread block to do the compression independently. Secondly, in each block the longest substring search is parallelized by assigning each thread a different starting point in the chunk, as shown in Figure 2.

Four threads ($tid = 0, 1, 2, 3$) in Figure 2 cooperate to do the longest substring search for a chunk $S = aaababbaab$, the offset of each thread's starting point is determined by the thread ID. For example, thread 0 ($tid = 0$) has zero offset, and thread i ($tid = i$) shifts the sliding window and uncoded data buffer i characters right. Each thread has its private buffer and runs just like the single thread model to output a $(offset, length)$ pointer or a $(1, literal)$ pointer. Pointers outputted by different threads may overlap, for example, pointer issued by thread 1 overlaps with pointers issued by thread 2 and thread 3. All the issued pointers are transferred to host memory, and CPU will do the redundancy elimination. After that, the sliding window and uncoded data buffer are updated accordingly. The above process repeats until the whole chunk is visited.

Finally, CPU concatenates each compressed data into a continuous stream.

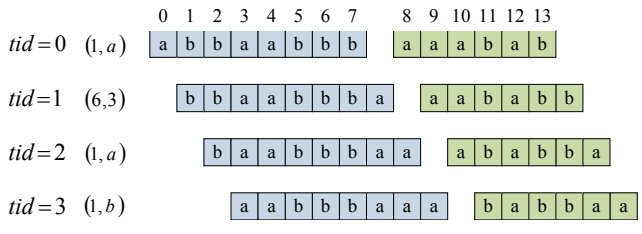

Figure 2: Execution of CULZSS in parallel.

3. PROBLEM ANALYSIS

We evaluated the performance of CULZSS on NVIDIA GTX590 GPU with a few benchmark files, the maximum compression speed achieved in the experiment is 657.68 Mbps, less than 1Gbps (see Section 6).

We identify three major drawbacks of the design of CULZSS. The first drawback is organizing the dictionary as a sequential character buffer, and doing the longest substring match linearly in the dictionary. The search complexity is $O(n)$, where n is the dictionary size. Therefore CULZSS restricts the dictionary size (typically 128 bytes) in order to control the substring search time.

The small dictionary size as well as small chunk size (4KB) is the second drawback, as it impacts the compression efficiency. Two duplicate substrings may miss the match if the distance between them exceeds the dictionary size. A potential longer match separated by two chunks may degrade to one or two shorter matches. The smaller the dictionary size and chunk size are, the more frequently the above cases occur, and the poorer the compression ratio achieves.

CULZSS assigns a chunk to a thread block, which is the third drawback. A thread block consists of multiple warps; having multiple warps to process the same chunk inevitably introduces inter-warp synchronization. Moreover, pointers issued by different threads may overlap, which means many threads search in vain, reducing the resource utilization. The more threads are involved in a data chunk, the more resources are wasted.

4. BASIC DESIGN

Based on the above analysis, we propose an efficient design that is immune to these problems. We call our design *GLZSS-basic*, which has three design points.

1. The dictionary is organized as a hash table to reduce the time complexity of duplicate substring search to $O(1)$.
2. The dictionary can be extended at run time to hold more substrings, and the chunk size is enlarged to 64KB. These two measures help to find more duplicate substrings. The size of 64KB is chosen for efficient coding consideration: an offset smaller than 64K can be coded as a *short* data type.
3. Each chunk is assigned to a warp in order to eliminate the inter-warp synchronization, increase the number of chunks a thread block can process (a block has multiple warps), and reduce the resources wasted on useless substring search.

The hash table is implemented as a flat array, where each slot stores a position of a scanned substring in the chunk. The 4-byte prefix of current uncoded string is hashed to find a slot, and its position in the chunk is stored in the slot. Multiple 4-byte substrings at different positions may be hashed to the same slot. To ease the hash conflict solving, new position simply overwrites the old one in the slot. Therefore, for two substrings with the same 4-byte prefix, position of the later substring is stored, causing smaller offset value for later match. For two substrings with different 4-byte prefixes that are hashed to the same slot, position of the early scanned substring is evicted from the dictionary. Obviously, restricting the length of conflict list to 1 may miss some matches

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
S	a	b	b	a	a	b	b	b	a	a	a	b	a	b	b	a	a	b
tag array	0	0	0	0	0	0	5	5	5	5	0	0	12	12	12	12	12	12

Table 1: Illustration of Tag Array

Algorithm 1 GLZSS-basic Algorithm

Require:
cur: point to current position in the input stream; initially point to the second position of stream;
org: store the *cur* value before searching from *cur*;
out: point to the output buffer;
lit_len: the length of copied literals;
mat_len: the number of matching bytes;

Ensure:
1: $org = cur$ - 1;
2: $lit\_len = 1$;
3: $mat\_len = 0$;
4: **while** the current data stream S not end **do**
5: cur_prefix = read(cur, 4);
6: $index$ = hash(cur_prefix);
7: pre = hash_table[$index$];
8: pre_prefix = read(pre, 4);
9: hash_table[$index$] = cur;
10: **if** $pre\_prefix == cur\_prefix$ **then**
11: mat_len = find_all_matching_bytes(cur, pre);
12: $off = cur$ - pre;
13: output_literal_pointer(org, lit_len, out);
14: output_matching_pointer(off, mat_len, out);
15: $cur = cur + mat\_len$;
16: $org = cur$;
17: $mat\_len = 0$;
18: $lit\_len = 0$;
19: **else**
20: $cur = cur + 1$;
21: $lit\_len = lit\_len + 1$;
22: **end if**
23: **end while**
24: **if** $lit\_len > 0$ **then**
25: output_literal_pointer(org, lit_len, out);
26: **end if**

and reduce the average length of matched substrings; however, this loss may be compensated by enlarged dictionary size. Moreover, the process of duplicate substring search is greatly simplified.

Algorithm 1 presents the pseudo-code of GLZSS-basic design. At the very beginning, the input data is split into chunks of 64KB, and each chunk is compressed independently by a warp. Moreover, each chunk has its private hash table which is initialized with 0s, the first position value of the input stream. In each loop, threads in a warp read 4 bytes from current position (actually each thread reads one byte), perform a hash function on the 4 bytes to get a hash index, and then read the corresponding hash slot. A previous position is then obtained (denoted as *pre*), and threads read 4 bytes from *pre*, compare them with current prefix. If the two substrings match, threads execute match branch (line 11 $\sim$ line 18). If the two substrings don't match, threads execute no-match branch (line 20, 21). Before entering the following branches, the hash table is updated by storing current position in the slot (line 9). In no-match branch, threads just increase current position and number of literal by 1. In match branch, threads find out all the matching characters in the two substrings starting from *cur* and *pre*, respectively, issue copy pointer and match pointer, update current position, and then reset the two variables that count the number of literals and matching bytes, respectively, for the next iteration. Pointers issued by GPU are finally transferred back to host memory for later usage.

On the benchmark file *sources*, the compressing speed obtained by GLZSS-basic is 158.33 MBps, 1.92 times faster than CULZSS.

5. TAGGING COMPRESSION

In GLZSS-basic design, only chunk-level parallelism is used. Within the chunk (warp), code path diverges into two branches. Even worse, the match branch is much longer than the no-match branch. Code path divergence causes serialization of threads, making the massively parallel power of GPU underutilized.

Path diverges after the current 4-byte prefix is searched in the dictionary. If no duplicate substring is found, search process stops; otherwise, more bytes are searched until the longest common substring is found. In order to eliminate path divergence, all threads should execute the same code, i.e., all threads should search 4 bytes in a single loop. However, in this way a long common substring needs several loops to finish the search, and a thread is unable to issue a match pointer before the entire common substring is identified.

To solve this problem, a linear buffer called tag array is introduced to store the match information during the scan. Each slot of the tag array corresponds to a byte in the chunk, and stores an offset that indicates whether the byte is a copied literal ($offset = 0$) or belongs to a match pointer ($offset > 0$). Every time a byte is scanned, its slot in the tag array is filled. After the whole chunk is scanned, tag array is transferred to the host memory for CPU to generate the final copy pointers and match pointers. For a sequence of adjacent bytes with the same offset value, one pointer is issued in the form of ($offset$, $length$).

For example, assume the input string is $S = abbaabbbaaababbaab$, the tag array is shown in Table 1. Substring *bbaa* starting from position 6 has a duplicate substring at position 1; therefore, their corresponding slots in the tag array are filled with 5, distance ($offset$) between the two substrings. Substring *abbaab* starting from position 12 has a duplicate substring at position 0, so their corresponding slots in the tag array are filled with 12.

However, there still remains one problem. In our method, a long common substring is divided into groups of 4 bytes, and each group is searched in one loop. Although they are searched independently in different loops, we require that their offset in the tag array should be the same so that the entire common substring could be issued as one match pointer. To explain the problem more clearly, let's suppose

that two substrings starting from *cur* and *pre* have a long common prefix, we require that 4 bytes from *cur* and *pre* are compared, and 4 bytes from $cur + 4$ and $pre + 4$ are compared, and so on. To this end, we must insert the 4-byte substring from $pre + 4$ in dictionary in the first loop; then when the 4-byte substring from $cur + 4$ is searched in the dictionary in the second loop, hash slot corresponding to the substring from $pre + 4$ is located, and search position of $pre + 4$ is got. Therefore, we let the threads read and compare 8 bytes (two groups) in a loop, 4 bytes starting from *cur* are used to search the dictionary to get *pre*, and 4 bytes starting from $pre+4$ are used to update the dictionary, preparing for the comparison in the next loop. We call this design GLZSS-tagging.

Algorithm 2 presents the pseudo code of GLZSS-tagging design. In each loop, 8 bytes from current position *cur* are loaded into cur_prefix, of which the first 4 bytes are used to locate a hash slot that may contain the position (*pre*) of a possible duplicate substring (line 2 $\sim$ line 4). Then 8 bytes from *pre* are loaded into pre_prefix, of which the last 4 bytes are used to update the hash table (line 6 $\sim$ line 8), preparing for the next loop of process. The trickiest part of the algorithm is the adjustment of *len*, the number of bytes current position *cur* will move forward. Function $find\_common\_prefix()$ returns the length of the common prefix of two substrings (line 9), which indicates three different cases. If $len < 4$, no duplicate substring is found, then *cur* should move forward one byte (adjust in line 11). If $4 \leq len < 7$, the longest common prefix is found, and *cur* should move forward *len* bytes. If $len = 8$, the identification of the longest common prefix must be postponed to the next loop, so *cur* should move forward 4 bytes (adjust in line 10). After adjusting the value of *len*, offset value off that will be filled in the tag array is computed according to *len*. If $len < 4$ (no duplicate substring is found), off is set to 0; otherwise, off is set to the distance between the two substrings (line 12). In line 13, slot(s) corresponding to the scanned byte(s) is (are) filled with off. At last, current position of chunk and tag array are updated, respectively (line 14 $\sim$ line 15).

To parallelize the while loop in Algorithm 2, we observe that the only parts that can be executed in parallel are line 2, line 6, line 9 and line 13 if coalesced access feature of global memory is used. Therefore, we assign every 4 consecutive CUDA threads to compress one data chunk, within each data chunk 4 threads cooperate to read bytes from chunk and write off to tag array. Since a warp consists of 32 threads, we divide a warp into 8 sub-warps, which allows a warp to compress 8 data chunks simultaneously.

Tag array effectively eliminates the path divergence in the while loop, and improves the compression speed of GLZSS-basic by 31% on benchmark file *sources*.

Algorithm 2 GLZSS-tagging Algorithm

Require:
cur: point to current position in the input stream;
op: point to current position in the tag array;
pre: point to matching position in the dictionary;

Ensure:
1: **while** the current data stream S not end **do**
2: cur_prefix = read(*cur*, 8);
3: $index$ = hash($1st\_4B\_c$); //$1st\_4B\_c$ is the first 4 bytes of cur_prefix
4: pre = hash[$index$];
5: hash[$index$] = *cur*;
6: pre_prefix = read(*pre*, 8);
7: $index$ = hash($2nd\_4B\_p$); //$2nd\_4B\_p$ is the last 4 bytes of pre_prefix
8: hash[$index$] = $pre + 4$;
9: len = find_common_prefix(cur_prefix, pre_prefix);
10: $len = (len == 8)\ ?\ 4 : len$;
11: $len = (len < 4)\ ?\ 1 : len$;
12: $off = (len < 4)\ ?\ 0 :$ *cur* - *pre*;
13: $op[cur..(cur + len - 1)] = off$
14: $cur = cur + len$
15: $op = op + len$
16: **end while**

6. PERFORMANCE EVALUATION

This section evaluates the performance of GLZSS, and compares it with an improved version of CULZSS [13]. We got the improved version of CULZSS from its author. This section also compares the performance of GLZSS with GZIP, the most popular LZ77-based compression routine.

6.1 Experiment Setup

Our experiments are conducted on a PC machine equipped with an AMD A8-3870 quad-core APU running at 800MHz, 4GB main memory, and an NVIDIA dual-GPU GTX590 card. Each GPU has 512 CUDA cores and 1.5GB RAM. Only one GPU is used in our experiments. The operating system is Ubuntu Linux 12.04, and the development toolkit is CUDA SDK 5.5.

We use 13 benchmark files in the experiments, all of them are from real world and available online [9] [15]. They are carefully selected to provide us a good span of data size from 32MB to 200MB and a good mix of strings with different properties. The 13 benchmark files can be grouped into 5 sets. The first set is C/Java source code files, including linux kernel tarball and gcc sources. The second set is English dictionary files (*english*, *etext99*) selected from texts collections of Gutenberg Project [5]. The third set is protein sequences and gene DNA sequences (*proteins*, *dna*, *chr22.dna*), which are used in biological analysis. The fourth is XML structured texts (*dblp.xml*, *rctail96*). The fifth is a collection of ordinary textual documents (*sprot32.dat*, *rfc*, *w3c2*). Except for the experiment on hash table size, the hash table size is set to 4K in other experiments.

6.2 Compression Speed

Two primary optimization techniques adopted by our GLZSS algorithm are: 1) organize the dictionary as a hash table to reduce the computing complexity of duplicate substring search; 2) eliminate path divergence to maximize the parallel degree of GPU execution. To evaluate the effectiveness of the two techniques, we measure the performance of improved CULZSS [13], GLZSS-basic design, and GLZSS-tagging design on the 13 benchmark files. The compression speeds are shown in Table 2, where column String gives the benchmark file, Size shows the file length, and Speedup is defined as $Speed_{GLZSS}/Speed_{CULZSS}$

Not surprisingly, GLZSS-tagging achieves the highest compression speed on all the 13 files. Furthermore, the fact that GLZSS-basic outperforms CULZSS, and GLZSS-tagging outperforms GLZSS-basic on all the 13 files indicates that both

String	Size	CULZSS	GLZSS-basic		GLZSS-tagging		LZSS		GZIP	
	(MB)	Speed	Speed	Speedup	Speed	Speedup	1 core	4 cores	1 core	4 cores
sources	200	82.21	158.33	1.92	207.73	2.52	41.96	162.12	23.64	90.52
rfc	111	78.54	148.98	1.89	195.39	2.48	39.10	153.54	20.46	77.43
linux-2.4.5	110	79.49	156.50	1.96	192.28	2.41	41.96	157.36	24.77	93.66
rctail96	109	79.33	165.22	2.08	211.47	2.66	47.68	185.97	27.12	101.97
sprot34.dat	104	77.37	159.27	2.05	192.44	2.48	45.78	171.66	31.74	117.56
dblp.xml	100	76.85	181.53	2.36	224.54	2.92	56.27	212.67	34.86	129.76
english	100	77.14	109.61	1.42	166.47	2.15	26.70	104.90	12.75	49.42
etext99	100	76.32	110.07	1.44	169.46	2.22	26.70	104.90	12.75	48.64
gcc-3.0	82	76.29	150.93	1.97	192.19	2.51	44.82	143.05	25.81	96.44
proteins	63	71.21	131.38	1.84	144.81	2.03	32.42	117.30	22.25	83.73
w3c2	50	78.76	179.47	2.27	183.01	2.32	61.99	187.87	37.65	137.22
dna	50	71.79	130.28	1.81	197.80	2.75	27.66	108.72	6.82	26.81
chr22.dna	32	65.55	126.24	1.92	175.04	2.67	28.61	111.58	7.27	28.54

Table 2: Compression speed (MBps) of different algorithms

the two optimization techniques are effective.

To evaluate the GPU performance advantage over CPU, we take a sequential version of GLZSS (actually an exact copy of GLZSS-basic) and parallelize it with *pthread* on a quad-core x86 CPU. Each file is divided into chunks, which are then distributed equally among four threads (as there are only four cores in the CPU) for compression. The compression speeds of the CPU-version of GLZSS are shown in Table 2 in the column LZSS. Except for file *w3c2*, GLZSS-tagging outperforms the parallelized CPU-version of GLZSS. This experiment shows that GPU is competent to offload the data compression task from CPU.

We also compare our GLZSS with GZIP, the most popular compression routine that is based on LZ77 algorithm. The sequential GZIP is also parallelized with *pthread* on the same quad-core x86 CPU, and the compression speeds are shown in Table 2 in the column GZIP. On all the 13 files, GLZSS-tagging outperforms the parallelized CPU-version of GZIP.

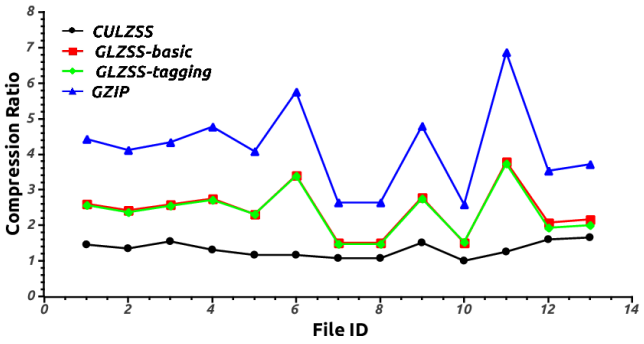

Figure 3: Comparison of compression ratio.

6.3 Compression Ratio

This section evaluates the compression ratio of GLZSS, which is defined as the ratio between the uncompressed size and compressed size, the higher the better. Figure 3 shows the compression ratios of CULZSS, GLZSS-basic, GLZSS-tagging and GZIP on all the 13 files. The horizontal axis denotes the benchmark files, which are numbered in the order they appear in the column String in Table 2.

Compared with CULZSS, GLZSS improves the compression ratio by 20% $\sim$ 196%, verifying our analysis in Section 3 that the limited window size and look-ahead buffer size degrade the CULZSS's compression efficiency. We also notice that compression ratio of GLZSS-basic is not identical to that of GLZSS-tagging, with minor difference less than 4%. The reason lies in the minor different way the hash table is updated. GLZSS-basic updates the hash table only with the first 4 bytes of cur_prefix (line 5 $\sim$ line 9 in Algorithm 1); but in addition to that, GLZSS-tagging further updates the hash table with the last 4 bytes of pre_prefix (line 6 $\sim$ line 8 in Algorithm 2). The extra update may fill in an empty hash slot to increase the chance of finding a duplicate substring; however, the extra update may also replace a recent position with a distant one, resulting in a larger offset in the match pointer.

We also notice that GZIP has the highest compression ratio that is about 69%$\sim$84% higher over GLZSS. The reason lies in the different coding methods that the two algorithms use to code the pointers. GZIP uses high-efficient Huffman coding, while GLZSS simply uses LZSS coding format. Parallelizing Huffman coding on GPU is a hard problem, we will solve it in the future.

6.4 Choose a proper hash table size

Hash table acts as the dictionary in GLZSS. Large dictionary improves the compression ratio, but consumes precious memory space. To get a proper hash table size, we measure the compression ratio and compression speed of GLZSS-tagging on all 13 files with different hash table size. Due to the limited space, we only show the experimental data on file *etex99* in Figure 4. Both the compression ratio and compression speed rise with the increasing hash table size, but the growth rate decreases. In order to have a reasonable performance/cost ratio, we set the hash table size to 4K.

6.5 Decompression

We also implement LZSS decompression on CPU and GPU. During the decompression procedure, each character of com-

String	Size(MB)	Compression ratio	CPU		GPU
			1 core	4 cores	
sources	200	2.57	65.80	168.80	141.14
rfc	111	2.36	68.66	168.75	144.00
linux-2.4.5	110	2.55	67.61	169.75	135.42
rctail96	109	2.72	71.53	167.85	132.56
sprot34.dat	104	2.32	90.60	210.76	158.31
dblp.xml	100	3.39	61.04	145.91	109.67
english	100	1.48	83.92	221.25	188.82
etext99	100	1.48	82.97	219.35	190.73
gcc-3.0	82	2.75	63.90	145.91	109.67
proteins	63	1.53	128.75	294.69	238.41
w3c2	50	3.74	61.99	114.44	98.22
dna	50	1.94	74.39	165.94	137.32
chr22.dna	32	2.01	68.66	178.34	96.32

Table 3: Decompression Speed (MBps) of GLZSS on CPU and GPU.

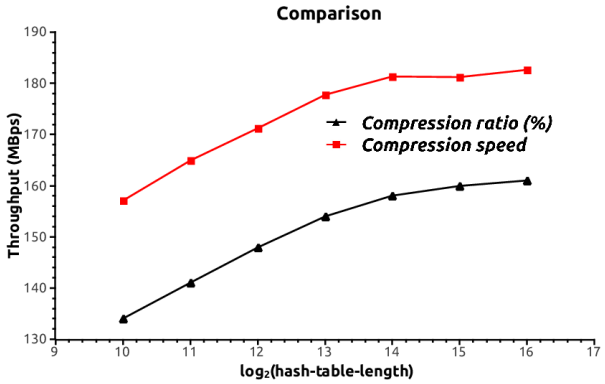

Figure 4: Performance of GLZSS on different hash table sizes.

pressed strings is read and decoded. If it is a literal byte, it is then output directly. If it is identified to be a part of pointer, the whole pointer $(offset, length)$ will be interpreted by the *length* previously decoded bytes at the distance of $offset$. As the uncompressed data are processed in block-level, we can still utilize the block-level parallelism in decompression. To distribute the compressed blocks across the processors, we need to identify the bound of each compressed data chunk. To achieve this, we store the total number of uncompress blocks in the header of compressed file, and record the plain size as well as the compressed size of each block in the header of each compressed block. Therefore, each compressed block is easily to be identified after reading the header information. Besides, the extra information needs only 8 bytes and is tiny compared to compressed block size in our tests, meaning that it does not hurt the compression efficiency.

Both sequential decompression and *pthread*-based parallel decompression are implemented on CPU. In the parallel version, compressed file is split into chunks and then distributed among CPU cores for decompression. The sequential decompression code is also ported to GPU, where the parallelization is achieved by assigning each chunk to a CUDA warp. The decompression speeds are shown in Table 3. The speed of GPU-based decompression is lower than that of *pthread*-based implementation; that's because we only simply port the decompress procedure to GPU without digging into the algorithm itself to exploit more parallelism. The divergences on processing literals and pointers in decompression are not eliminated in GPU-based procession and degrade the performance.

An interesting observation is that lower compression ratio corresponds to higher decompression speed. This is because a file with lower compression ratio has fewer match pointers, and therefore needs less computation to parse the match pointers when it is being decompressed.

7. RELATED WORKS

Researches have been done on parallelizing data compression on CPU and GPU. The parallelizing method adopted in most of the works is splitting the input data into chunks, and then making these chunks to be compressed simultaneously, examples include [2] [4] [13], etc. Our GLZSS also uses this method. As chunks are compressed independently, duplicate substrings in different chunks are missed, causing compression ratio decreased.

In recent years, methods that parallelize data compression without splitting input data are proposed. [17] presents a work-efficient parallel algorithm for LZ-factorization and evaluates it on a 40-core Intel machine. The experimental results show that their algorithms achieve good speedup with respect to best sequential implementations; however, the absolute processing speed is still less than 1Gbps. [2] proposes a parallel implementation of suffix array construction on GPU, which can be used in Burrows-Wheeler transform (BWT)-based data compression, e.g., BZIP2[1]. This work increases the compression speed by 11x over the fastest BWT on GPU.

In the GLZSS design, sub-warp division is used to increase the parallelism within a warp. Sub-warp division has been widely used in modern GPU-accelerated applications, and is fully discussed in [6]. Sub-warp is defined in [6] as virtual warp, and is used to address the problem of workload imbalance during graphic processing.

8. CONCLUSIONS

In this paper, we present an effective method to parallelize LZSS compression algorithm on GPU. We reorganize

the dictionary as a hash table to speed up the locating of duplicate substrings, and then redesign the matching process of duplicate substrings to avoid threads serialization, the most difficult part of the algorithm to be parallelized. We evaluate our algorithm on NVIDIA GTX 590 GPU with benchmark files from real world. Experimental results show that our GLZSS achieves 2x speedup over existing work.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their help in improving the content and presentation of the paper. We additionally would like to thank Shengjie Jiang for invaluable comments.

10. REFERENCES

[1] BZIP2. http://www.bzip.org/.

[2] M. Deo and S. Keely. Parallel suffix array and least common prefix for the GPU. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 197–206. ACM, 2013.

[3] L. Erdodi and L. Erdődi. File compression with lzo algorithm using nvidia cuda architecture. In *Logistics and Industrial Informatics (LINDI), 2012 4th IEEE International Symposium on*, pages 251–254. IEEE, 2012.

[4] J. Gilchrist. Parallel data compression with bzip2. In *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems*, volume 16, pages 559–564, 2004.

[5] Gutenberg Project. http://www.gutenberg.org/dirs/. 2005.

[6] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 267–276. ACM, 2011.

[7] Lossy Compression. http://en.wikipedia.org/wiki/lossy_compression.

[8] LZSS Algorithm. http://en.wikipedia.org/wiki/lempel-ziv-storer-szymanski.

[9] Manzini's Corpus. http://people.unipmn.it/manzini/lightweight/corpus/. 2002.

[10] NVIDIA. CUDA C Best Practice. 2013.

[11] NVIDIA. CUDA C Programming Guide. 2013.

[12] A. Ozsoy and M. Swany. CULZSS: LZSS lossless data compression on CUDA. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 403–411. IEEE, 2011.

[13] A. Ozsoy, M. Swany, and A. Chauhan. Pipelined parallel LZSS for streaming data compression on GPGPUs. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 37–44. IEEE, 2012.

[14] PIGZ. http://zlib.net/pigz/.

[15] Pizzachili's Texts. http://pizzachili.dcc.uchile.cl/texts.html. 2002.

[16] S. Puglisi, D. Kempa, et al. Lempel-Ziv factorization: Simple, fast, practical. In *Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*, 2013.

[17] J. Shun and F. Zhao. Practical parallel Lempel-Ziv factorization. DCC, 2013.

[18] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 23(3):337–343, 1977.