2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems

G-match: A Fast GPU-Friendly Data Compression Algorithm

Li Lu
School of Computer Science and Technology
University of Science and Technology of China
Hefei, Anhui, China
luly9527@mail.ustc.edu.cn

Bei Hua
School of Computer Science and Technology
University of Science and Technology of China
Hefei, Anhui, China
bhua@ustc.edu.cn

***Abstract*—Data compression plays an important role in the era of big data; however, such compression is typically one of the bottlenecks of a massive data processing system due to intensive computing and memory access. In this paper, we propose a high-speed GPU-friendly data compression algorithm called G-match that takes full advantage of the GPU parallel computing power to speed up the compression process. The greatest challenge here is to solve the contradiction between the high data dependency inherent in the compression algorithm and the GPU single-instruction multiple-thread operating model. G-match achieves a high parallel degree by eliminating fine-grained data dependency and all path divergences in the algorithm. Compared with other, similar work on GPUs, G-match is the first thoroughly parallelized data compression algorithm. Experiments comparing other GPU compression algorithms show that G-match achieves approximately 33% speedup over the current fastest implementation and the highest compression ratio.**

***Keywords*-Data Compression Algorithm; Graphics Processing Unit(GPU); Algorithm Parallelization;**

I. Introduction

The era of big data is already here. While the volume of data stored by mankind until the year 2000 was only approximately 12 EB, today, the estimated production of data around the world is a massive 2 EB per day. To keep pace with the explosive growth of data, operators of data centers are forced to heavily and ceaselessly invest in their storage and network systems. One effective way of saving resources is compression, which allows reduction of data volume so that one can store more data in a limited storage space, expend less time to access disks, and consume less bandwidth when transferring data. Owing to these advantages, data compression has been widely used.

For today's big data storage and analysis systems (such as Hadoop) to benefit from compression, data need to be frequently compressed and decompressed in most cases. In a typical scenario of serving a data query, the compressed data are first read from the disks into memory and then decompressed and analyzed to serve the query; finally, the data are recompressed and written back onto the disks. Unfortunately, data compression is a computation-intensive and memory-access-intensive operation that consumes precious computing power and memory bandwidth and adds considerable processing latency. It was reported that compression and decompression represent the most time-consuming tasks in the database compaction process when a solid-state drive (SSD) is used as the storage device [1]. Therefore, data compression, if not carefully designed, may impose negative impacts on system performance.

Currently, one of the most widely used data compression software packages used in most of Google's projects (e.g., BigTable, MapReduce) and many open-source databases (e.g., Cassandra, LevelDB, MongoDB) is Snappy. Snappy is an open-source library developed by Google, the core of which is an LZ77-based data compression algorithm that is referred to as the Snappy algorithm in this paper. LZ77, a classical lossless data compression algorithm proposed in 1977, has provided the foundation for multiple derived variants, including LZSS, LZW, GZIP, and 7Zip, which have been widely used to compress text documents, source code and data files in databases. All the LZ77 family algorithms use dictionary-based methods where the main idea is to replace the repeated occurrence of a string with reference to its already occurred duplicate, hoping that the reference is shorter than the original string. The main difference among various variants exists in the different tradeoffs achieved between compression ratio and speed. As an LZ77 variant used in big data storage and analyzing systems, Snappy focuses on very high compression speed and a reasonable compression ratio by means of simplifying the search of duplicate strings.

Instead of modifying algorithms to achieve high speed, one can speed up data compression by exploiting parallel computing [2]. The input data are divided into blocks, and each block is assigned to a processor to compress. With the advent and popularity of heterogeneous computing, modern computer systems are often equipped with multicore processors and a variety of coprocessors that incorporate specialized capabilities for particular tasks. Graphics processing units (GPUs), possessing massively parallel computing power and high memory bandwidth, are commonly used as coprocessors to offload computation-intensive or memory access-intensive tasks from the CPU. For example, the LZSS algorithm has been parallelized on GPUs by dividing input data into blocks and assigning each block to a group of GPU

978-1-7281-2058-4/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCC/SmartCity/DSS.2019.00116

cores to compress [4] [5] [6].

However, the unique architecture of GPUs imposes a great challenge on most of the parallelization works. GPU cores are organized into a two-tiered structure composed of streaming processors (SPs) and streaming multiprocessors (SMs) as per Nvidia terminology. Each SM is made up of a set of SPs, and all SPs in an SM execute in single-instruction multiple-thread (SIMT) mode. For data compression, although different data blocks can be compressed independently, compression within a data block is data-dependent, which makes it challenging to mine intra-block parallelism. Thus far, all work reported on parallelization of data compression on GPU mainly achieve inter-block parallelism, leaving the massive computing power of GPUs under-utilized.

In this paper, we propose a GPU-based compression algorithm called G-match that follows the basic doctrine of LZ77 family algorithms but achieves very high intra-block parallelism. G-match is so far the first thoroughly parallelized algorithm on a GPU that eliminates path divergence and load imbalance throughout the process. G-Match is implemented on the Nvidia's Compute Unified Device Architecture (CUDA) framework and is evaluated on a series of Nvidia GPU cards. Experimental results show that an average compression speed of 3.7 GB/s is achieved on 6 test files with Nvidia GTX 1080, corresponding to approximately 33% speedup over the current fastest GPU compression implementation.

The rest of the paper is organized as follows. Section 2 introduces GPU architecture, dictionary-based compression algorithms and related works on accelerating data compression on GPUs. Section 3 analyzes the Snappy algorithm and explores the design space. Section 4 proposes the G-match algorithm that thoroughly parallelizes the data compression process. Section 5 discusses implementation issues, and Section 6 presents experimental results. Section 7 concludes the paper.

II. Background and Related Work

This section presents a survey of related literature on GPU architecture and relevant algorithms.

A. GPU Architecture

GPUs, originally designed for graphics rendering tasks, have now evolved into many-core processors suitable for general-purpose computing. A single GPU commonly possesses hundreds or even thousands of processing cores.

The basic computing unit in the GPU of the Nvidia series is the streaming processor (SP). A single GPU can have more than hundreds of SPs; for example, the latest Nvidia GTX 900 series comprises thousands of SPs. A certain number of SPs are integrated into a streaming multiprocessor (SM). An SM can be viewed as an SIMD processor which handles several data streams. In addition, each SM has a set of storage elements of its own, including thousands of registers and shared memory.

CUDA provides programmers with application programming interfaces (APIs) to manage theoretically unlimited threads on GPUs. With the CUDA runtime system, a compiled CUDA program can automatically schedule its threads on any number of cores without the knowledge of their states. In CUDA programming, the programmer needs to first divide a task into subtasks and determine the number of threads required to serve a subtask. The number of subtasks (*griddim*) and the number of threads per subtask (*blockdim*) are then passed on to the GPU. During execution, a group of *blockdim* threads are organized into a thread block, and each thread block is mapped to an SM where the threads are scheduled on SPs. Note that *blockdim* is often larger than the number of SPs in an SM (typically 48); therefore, only a part of the threads can be executed in one SM at a time. To hide memory access latency, CUDA organizes a group of 32 threads into warps such that only one warp is executed at a time. When a running warp pauses to wait for memory access, another warp is scheduled to run. As the threads in a warp run in SIMT mode, code divergence caused by conditional branches should be avoided as far as possible.

The memory hierarchy of GPU consists of off-chip memory (such as global memory and constant memory) and on-chip memory (such as shared memory and registers).The off-chip memory is larger but has a longer access latency, while the reverse is true for the on-chip memory. The off-chip memory can be accessed by all the threads; the on-chip memory, on the other hand, is SM-private. As the contexts of the warps are saved during their suspension, the threads belonging to different warps are completely isolated despite being executed on the same SM.

B. Traditional Dictionary-based Compression Algorithms

The family of Lempel-Ziv-77 (LZ77) [7] algorithms have been a cornerstone of lossless data compression for over three decades. In this section, we briefly review the general idea of the LZ77 family rather than describe a specific algorithm.

LZ77 family algorithms maintain a dictionary buffer to store the most recently scanned strings and operate on the input data by repeatedly searching for duplicate strings in the dictionary. If a duplicate string is found, a reference to the previous occurrence of the string is provided as output. This dictionary can be a sliding window or a fixed-sized buffer of the visited text. A reference is usually an *offset-length* pair, where *offset* indicates the distance between the two matched strings (or sometimes the absolute position of the previous duplicate string in the dictionary), and *length* indicates the size of the duplicate string in bytes. To achieve compression, some algorithms require a minimum match size, i.e., a duplicate string is not recorded if its length is shorter than the threshold, and characters are outputted directly with a

flag indicating that these characters are literals. Most of the algorithms comprise three major steps: (1) search in the dictionary for the longest string that matches the prefix of current position; (2) encode the scanned characters in the first step as literals or an *offset-length* pair based on the search result; (3) shift the scanned characters into the dictionary buffer. As per [4] [8], search of duplicate strings consumes the highest percentage of the algorithm time.

C. Snappy as a Modern Compression Algorithm

Snappy was originally designed for Google's BigTable system and has witnessed widespread application in databases and datacenters. Snappy makes an important tradeoff between the compression speed and compression ratio. Relative to LZW or Gzip, it is reported that Snappy achieves 2-5x speedup with a 10-20% loss in compression ratio. Snappy is chosen as a typical example of modern compression algorithms in this paper.

Snappy mainly runs in a *while* loop that comprises three steps. In the first step, the 4-byte prefix of the current input is hashed to find a slot, where the position of a possible duplicate string might be contained, and the slot is read and updated with the current position. In the second step, the current input is compared with the string whose position was just read from the slot to find the longest common prefix. In the third step, if the length of the longest common prefix is not less than 4, an *offset-length* pair is output, and the current position is increased by *length*; otherwise, the first byte of the current input is output as a literal, and the number of current positions is increased by 1. According to the process, a newly position always overwrites the previous position in the slot; i.e., the conflict chain size of the hash table is 1. This condition may result in the loss of some matches but greatly simplifies the search of duplicate strings.

D. Related Works

Many previous studies have attempted to accelerate data compression on GPUs. A floating-point data compression work achieved an extremely high speed of 75 GB/s by taking advantage of the specific data type [9]. To compress the transferring stream of the database, [10] implemented nine lightweight compression on GPU and designed a compression planner to find their best combination. These works aim at providing seamless compression service during data transferring to reduce communication cost in distributed database and thus do not directly address the compression ratio. Moreover, the compression objects in these works are mainly arrays, structures or tables rather than text files.

A series of reports called CULZSS [4] [11] [12] implemented the LZSS algorithm on CUDA with the primary idea of splitting data into chunks and assigning each chunk to a thread block to compress. The original CULZSS algorithm was reported in [4], while [11] and [12] focused on a GPU-CPU pipeline intended to maximize system throughput. However, the LZSS algorithm that it used is directly ported and had not been sufficiently optimized for the GPU parallel architecture, which severely affected the system performance (for the details of the drawbacks of analyzing in CULZSS, see [8]).

CULZSS-bit [13], a bit version of CULZSS, implemented the LZSS algorithm using a bit-vector approach and turned the longest prefix match into a nondeterministic finite automaton by precomputing the incoming alphabet strings and maintaining their bit positions. From its design principles, CULZSS-bit mainly aims at compressing unreadable binary files. As the alphabet strings need to be recalculated frequently, the system scores poorly with text files.

GLZSS [8] redesigned and parallelized LZSS on GPU and has achieved the highest throughput thus far. This approach adopts several methods to simplify duplicate string searching in LZSS, such as using a hash table instead of sliding window and addressing hash conflict by simply overwriting the slot. These methods are similar to those of Snappy. GLZSS also eliminated path divergence in code execution, which shares the same goal with this paper. However, the degree of parallelization of GLZSS was not particularly high; as only reading and writing of continuous characters was performed by multiple threads, duplicate string searching was still performed by a single thread in a thread block.

III. Problem Analysis

Because Snappy (and most of the other compression algorithms) divides the input data into fixed-length blocks and because each block is compressed independently, it is simple to obtain inter-block parallelism. We focus our discussion on intra-block parallelism. As indicated above, compression algorithms are strongly data-dependent. All the visited data are used as the dictionary by all the unvisited data; therefore, it is not feasible to further divide a block into smaller nonoverlapping chunks and assign each chunk to a thread. In addition, compression algorithms always scan the data byte by byte, making it difficult to employ multiple threads to process data simultaneously. We hypothesize that an innovative way exists to solve this problem.

The flow of Snappy(and other fast compression algorithm) broadly consists of two parts. (1) Hash table lookup and updating. The position of a possible duplicate string is read, and position of current 4-byte prefix is then saved in the hash table. (2) Longest common prefix search in two strings starting at *cur* and *mat*. The first part is order-dependent as different writing orders result in different hash tables. To achieve parallelism, the order dependence in the algorithm must be relaxed. The second part is independent of the first part and can be easily parallelized if a set of (*cur*, *mat*) pairs to be compared are found in the first part. The problem in the second part is thread synchronization, as threads must run in lock-step, but they may work on strings of different sizes.

Based on the above analysis, we design a GPU-friendly compression algorithm and refer to the new design as G-match. We divide the process of the existing compression algorithm into two separate passes. In the first pass, 32 threads are launched that scan the data block and look up and update the hash table in parallel. A set of (*cur*, *mat*) pairs are obtained and saved in this pass. In the second pass, the data block is divided into 32 chunks and each chunk along with its (*cur*, *mat*) pairs is assigned to a thread to perform the longest common prefix search. Organizing the algorithm into two separate passes rather than interleaving hash table lookup and common prefix search in a single pass is to maximize the parallel degree, as not every thread is required to perform common prefix search after hash table lookup.

IV. Design of G-match

A. Hash Table Lookup and Updating

To scan the input data in parallel, we relax the order-preserving granularity from bytes in the original algorithm to a slightly larger unit. To be specific, the data block is divided into 32-byte units, and 32 threads are launched to scan from different locations in a unit. They execute line 3 - line 5 of Algorithm 1 from the current unit and then move on to the next unit. To record the results of hash table lookup (i.e., *mat* in line 4), an *index* array is introduced, whose *ith* slot corresponds to the *ith* location of the data block. *Index*[*i*] = *m* means that the 4-byte prefix at position *i* and 4-byte prefix at position *m* are hashed to the same slot, and therefore, the two strings must be compared in the second pass. Table 1 shows a fragment of an *index* array, where *index*[22] = 2 implies that the string at position 22 should be compared with the string at position 2 in the second pass.

With this modification, unit order rather than byte order is preserved, which may cause some deviation from the original algorithm. First, if a string pattern appears twice within a unit but has never occurred before, then the latter one fails to find the previous one, causing a slight loss of compression ratio. Second, if multiple threads need to write the same hash slot, the actual write order becomes unpredictable. However, this uncertainty does not affect the correctness of the algorithm.

B. Finding Duplicate Strings

In the *index* array computed by the first pass, *index*[*i*] = 0 implies that the string at location *i* does not have a duplicate string in the dictionary, and *index*[*i*] = *m* (*m* > 0) indicates that the string at location *i* probably has a duplicate string at location *m* and needs further comparison. As order dependence has been eliminated within the *index* array, we can divide the input data block and its corresponding *index* array into smaller nonoverlapping chunks and assign each chunk with its corresponding *index* array chunk to a thread to perform duplicate string match.

Table I
INDEX ARRAY USED IN COMPRESSION

Char Seq	21	22	23	24	25	26	27	28
Index array	0	2	8	0	0	0	0	17
Preprocessed *Index* array	-22	2	8	-28	-28	-28	-28	17
Result array	0	4	0	0	0	0	0	5

However, the issue of path divergence may occur here. First, threads with *index*[*cur*] = 0 need to do nothing, while threads with *index*[*cur*] > 0 need to enter a *while* loop to match the common prefix. Second, the completion time of each thread depends on the length of the common prefix it is working on.

To solve the first problem, threads must skip over idle positions and directly obtain their working positions. To solve the second problem, threads must make progress in any circumstance without waiting for other threads to complete. We discuss these two problems in the following sections.

1) Finding the Next Position to Compare: To allow threads to directly locate the next nonzero-valued slot in the *index* array, the *index* array is preprocessed as follows. A slot with value 0 is filled with the index of the next nonzero-valued slot. For example, assuming that *index*[*i*] and *index*[*i* + *k*] have nonzero values and slots between them (i.e., *index*[*i* + 1] to *index*[*i* + *k* - 1]) are all 0s, then *index*[*i* + 1] to *index*[*i* + *k* - 1] are all filled with value - (*i* + *k*); the minus sign here is used to distinguish them from normal *cur* values. Therefore, when a thread reaches a slot with a minus value, say *index*[*i*] = - *k*, it has the knowledge that the next working position is contained in *index*[*k*] and then jumps to *index*[*k*] and sets *cur* = *k*). The following code line implements the skip operation:

$$cur = (index[cur] < 0)?\ \text{-}\ index[cur] : cur$$

As shown in the second and third lines of Table 1, *index*[28] is nonzero, and therefore, the preceding zero slots (i.e., *index*[24] to *index*[27]) are all filled with "- 28".

To accelerate the preprocessing, we split the *index* array into 32 chunks and assign each chunk to a thread. Each thread scans its chunk from rear to front, filling each zero-valued slot with the index of the last visited slot with a nonzero value.

As the boundaries of a chunk may contain zero-valued slots, the boundary problem is addressed as follows. When a thread begins to scan a chunk from rear to front, it simply skips over all zero-valued slots before it reaches the first nonzero-valued slot. For the zero-valued slot at the front boundary, a thread continues scanning into the previous chunk, filling all the zero-valued slots until it reaches a nonzero-valued slot.

Algorithm 2 shows the pseudocode that preprocesses an *index* array chunk. Variable *i* records the index of the last visited nonzero-valued slot and is initialized to 0 (line 1). All the zero-valued slots at the rear part of the chunk are

Algorithm 1 Preprocessing an *index* array chunk

1: $now =$ end of a data chunk, $i = 0$
2: **while** now is in the data chunk **or** $index[now] == 0$ **do**
3: $index[now] = (index[now] == 0)?$ *-i* : $index[now]$
4: $i = (index[now] == 0)?i : now$
5: $now = now - 1$
6: **end while**

ignored (line 3), and variable *i* remains zero (line 4). When a nonzero-valued slot is encountered, i.e., *index*[*now*] $\neq 0$, *i* is set to *now* (line 4), and then, all zero-valued slots following it are filled with - *i* (line 3) until another nonzero-valued slot is encountered. Line 2 gives the termination conditions of the *while* loop. Note that the thread does not stop scanning if it encounters a zero-valued slot at the front boundary; instead, it continues to move ahead and fill slots until it encounters a nonzero-valued slot.

2) Longest Common Prefix Match: Each thread is assigned a data chunk and a corresponding *index* array chunk to perform duplicate string match. For each *index*[*i*]=*m* (*m* > 0), the thread needs to find the longest common prefix for the two strings starting at position *i* and *m*. Due to different sizes of common prefixes, threads finishing earlier have to wait for other threads to finish, leading to a lowered parallel degree.

To make all threads continue progressing, we remove the *while* loop (line 6 - line 8) in Algorithm 1 and rewrite the algorithm to prevent threads from waiting for each other. Each thread is required to compare four bytes at a time (to be consistent with the minimum match size of Snappy) and then moves forward according to the comparison result (i.e., variable *tmp*). If *tmp* is less than 4 in the first comparison, i.e., fails to find a duplicate string, the thread moves one slot ahead in the *index* array to find the next position to compare (case 1). If *tmp* is 4, a duplicate string is found but needs further comparison to find its end; the thread then moves four bytes forward in the data chunk (case 2). In subsequent comparisons, whenever *tmp* is less than 4, a complete duplicate string is found, its length is recorded, and the thread moves *tmp* bytes in the *index* array to find the next position to compare (case 3). To sum up, when *tmp* is less than 4 (case 1 and case 3), the *index* array is used to find the next position to compare; otherwise, the current match process is continued.

In the original Snappy algorithm, an *offset-length* pair and all the bytes between two duplicate strings are encoded whenever a duplicate string is found. However, this process may cause path divergence in G-match; therefore, a *result* array is introduced to record the length of duplicate strings during the scan. Similar to the *index* array, the *ith* slot of the *result* array corresponds to the *ith* location of the data chunk. Table 1 shows a fragment of a *result* array, where *result*[28] $= 5$ implies that a 5-byte string at position 28 has a duplicate string.

Algorithm 2 Finding Duplicate Strings

1: $cur =$ start of chunk, $fin = 0, tmp = 0, mat = 0, result = (\sim 0)$
2: $cur = (index[cur] < 0)$? *-*$index[cur] : cur$
3: $mat = index[cur]$
4: **while** $cur \neq$ end of chunk **do**
5: $tmp =$ GET_PREFIX_LEN (cur, mat)
6: $fin = fin + tmp$
7: $mat = (fin < 4)$? $mat : mat + tmp$
8: $cur = (fin < 4)$? $cur + 1 : cur + tmp$
9: $result[cur - fin] = (tmp < 4 \&\& fin \geq 4)$?fin : 0
10: $cur = (index[cur] < 0 \&\& tmp < 4)?$*-*$index[cur]$: cur
11: $mat = (tmp < 4)$? $index[cur] : mat$
12: $fin = (tmp < 4)$? $0 : fin$
13: **end while**

Algorithm 3 shows the pseudocode of finding duplicate strings in a data chunk. For simplicity, we omit the boundary processing code. Line 2 - line 3 locate the first pair of strings to be compared, designated by *cur* and *mat*, respectively. In the main *while* loop (line 4 - line 13), the thread compares 4 bytes from the data block and dictionary, and the length of the common prefix (0 - 4) is calculated (line 5). *Fin* accumulates the length of the common prefix so far (line 6).

If *tmp* < 4 (case 1: no duplicate string is found), *cur* is increased by 1 (line 8), and *index* array is used to find the next position to compare in the data chunk (line 10) and in the dictionary (line 11); *fin* is reset to zero (line 12).

If *tmp* $= 4$ (case 2: need further comparison), both *mat* and *cur* are increased by 4 (line 7 - line 8), and *cur*, *mat*, and *fin* shall not change in line 10 - line 12.

If *tmp* < 4 and *fin* > 4 (case 3: a complete duplicate string is identified), the length of the duplicate string is recorded in the *result* array (line 9), *cur* is increased by *tmp* (line 8), and the *index* array is used to find the next position to compare in the data chunk (line 10) and in the dictionary (line 11); *fin* is reset to zero (line 12).

After finishing the scan, the *result* array and the *index* array are transferred to the CPU for final encoding. The *result* array records the lengths and positions of all duplicate strings, and the *index* array provides dictionary entries. The CPU generates *offset-length* pairs using information from the two arrays and then encodes *offset-length* pairs and literals according to algorithm specific encoding rules. Considering Table 1 as an example, given *index*[22] $= 2$ and *result*[22] $= 4$, the CPU has the knowledge that the string at position

22 has a 4-byte duplicate string at position 2. It therefore encodes this string as an *offset-length* pair (2, 4).

V. Implementation Issues

A. Result Output

A complete dictionary-based data compression algorithm includes two steps: duplicate string search and output encoding. After a stream of literals and *offset-length* pairs are obtained, an encoding method is applied to the output stream to produce the final compressed data. To achieve a high compression ratio, classical data compression software packages, such as GZip, usually apply statistical compression (e.g., Huffman coding) to the output stream. However, statistical compression requires intensive computing and memory access. To achieve high compression speed, most modern data compression solutions use simple encoding methods. For example, Snappy encodes the output stream by simply attaching an indicator prefix to a literal string or an *offset-length* pair describing information such as type and length. In this paper, G-match adopts exactly the same method with Snappy. The output file of G-match can be directly decompressed by any decompression applications that support Snappy.

As the simple encoding method consumes minimal time, it is suitable to run output encoding on the CPU. Therefore, we form a simple CPU-GPU-CPU pipeline for data compression, where the CPU sends original data to the GPU, the GPU performs a duplicate string search using G-match, and the CPU performs output encoding. The data transmission delay between the CPU and GPU can be overlapped with computing on the CPU and GPU.

B. Parameter Selection of Hash Tables

Both the hash table size and conflict chain size affect the compression performance. Adjusting these two parameters of the hash tables is essentially a tradeoff between compression ratio and compression speed. Shrinking the hash table size or conflict chain size reduces the possibility of spotting a duplicated string, thus decreasing the compression ratio, while expanding them increases the maintenance overhead of hash tables, thus decreasing the compression speed. In the original Snappy, the hash conflict chain was one, and the hash table size could be customized by users.

Note that since the G-match inherits the same matching and encoding mechanism of Snappy, if the hash table size and hash conflict chain size are also set to the same value, G-match and Snappy will behave nearly identically and output almost exactly the same compressed files. The slight difference between them is caused by the parallel execution feature discussed in Section 4.1.

In the next section, we evaluate the performance of G-match with different parameters of hash tables. To evaluate the influence of the hash conflict chain size, we implement a variant of G-match with a length n hash table conflict chain ($n > 1$). Each hash entry has n slots, which are initialized to zeros. The position of a 4-byte string can be written into any zero-valued slot in the target hash entry. If all slots are occupied, a slot is randomly chosen to overwrite. To maintain consistency with the n-slot hash entry structure, the entry of the index array also has n slots. Whenever a hash entry is located by a 4-byte string, the whole hash entry is saved in the corresponding entry of index array. During the preprocessing of the index array, only entries with all 0s are filled with the index of the next possible matching position. To find the longest common prefix, all nonzeroed positions in an index entry are searched successively, and the longest match is returned.

VI. Evaluations

A. Experimental Setup

Six files of different types and sizes are used in the experiments: NO.1 is a binary exe file, NO.2 is a DNA sequence text file, NO.3 is an English novel text file, NO.4 are source code files, NO.5 is a bmp image file and NO.6 is a LevelDB data file, which is commonly compressed by Snappy. NO.1 - NO.5 can be acquired on Internet(No.2 - NO.4 are from Project Gutenberg [14]), No.6 is randomly generated.

Several existing works of GPU compression are selected for comparison, which are CULZSS, CULZSS-bit and GLZSS. For each work, we choose the best performance configuration/version reported in their papers. CULZSS and GLZSS code are obtained from their authors, and CULZSS-bit is reimplemented. The original Snappy code is obtained online on its official website.

We use 4 Nvdia GPU cards in the scalability evaluation experiment. The hardware parameters of these GPUs are as follows: Nvdia GTX680 with frequency of 1536 MHz and 3.09 TFLOPs, Nvdia GTX780 with frequency of 2304 MHz and 4 TFLOPs, Nvdia GTX980 with frequency of 2048 MHz and 4.6 TFLOPs and Nvdia GTX1080 with frequency of 2560 MHz and 9 TFLOPs.

B. Influence of the Hash Table Size on the Compression Ratio

This experiment is performed to evaluate the influence of the hash table size on the compression ratio. Duplicate strings found by compression algorithms are redundant data that can be compressed. The more redundant data are found, the higher is the resulting compression ratio. We use the ratio of redundant data, defined as the ratio between the total length of duplicate strings found in a file and the original file size, to approximately evaluate the compression effect. This experiment is run on CPU E5-1620 and GPU GTX980.

To evaluate the influence of the hash table size on the compression ratio, we increase the hash table size (in entries) from 2K to 64K according to the power of 2, and the conflict chain size is set to one. G-match is applied to all six test

Table II
RATIOS OF REDUNDANT DATA WITH DIFFERENT HASH TABLE SIZES

Test Files	Hash Table Size					
	2K	4K	8K	16K	32K	64K
File 1	2.2%	2.8%	3.5%	3.6%	3.6%	3.6%
File 2	60.5%	61.8%	62.1%	62.2%	62.4%	62.5%
File 3	36.3%	40.5%	43.3%	45.3%	46.3%	46.8%
File 4	74.5%	76.3%	77.2%	77.8%	78.1%	78.3%
File 5	26.5%	28.1%	28.8%	29.5%	30.3%	30.4%
File 6	28.1%	31.1%	33.0%	33.5%	34.4%	34.6%

Table III
RATIOS OF REDUNDANT DATA WITH DIFFERENT CONFLICT CHAIN SIZES

Test Files	Hash Conflict Chain Size			
	1	2	3	4
File 1	3.5%	3.6%	3.6%	3.6%
File 2	62.1%	63.7%	64.2%	64.3%
File 3	43.3%	46.3%	46.7%	46.7%
File 4	77.2%	79.2%	80.0%	80.0%
File 5	28.8%	29.9%	30.2%	30.2%
File 6	33.0%	34.6%	34.7%	34.7%

Table IV
COMPRESSION SPEEDS WITH DIFFERENT CONFLICT CHAIN SIZES (MB/S)

Test Files	Hash Conflict Chain Size			
	1	2	3	4
File 1	2,222.5	1,555.8	1,377.0	1,155.7
File 2	1,937.4	1,356.3	1,201.2	1,007.5
File 3	1,797.5	1,258.3	1,114.4	934.7
File 4	1,707.5	1,195.3	1,058.6	887.9
File 5	2,117.2	1,482.2	1,312.9	1,101.1
File 6	1,970.0	1,379.0	1,221.4	1,024.4

files, and the ratio of redundant data obtained in each file with a specific hash table size is counted and shown in Table 2.

According to Table 2, increasing the hash table size only slightly improves the compression ratio, and the effect vanishes rapidly. Therefore, in the following experiments, we set the hash table size to 8K.

C. Influence of the Conflict Chain Size on the Compression Performance

Longer conflict chain may save more occurrences of visited strings, increasing the probability of finding duplicate strings but taking up more memory space and degrading the compression speed. To evaluate the influence of the conflict chain size on the compression performance, we set the conflict chain size to 2, 3, and 4 and run a G-match variant on all six test files. The ratios of redundant data with different conflict chain sizes are shown in Table 3, and the corresponding compression speeds are shown in Table 4. This experiment is run on CPU E5-1620 and GPU GTX980.

According to Table 3, a longer conflict chain size only slightly increases the ratio of redundant data (less than 3% in most cases) but greatly decreases the compression speed (Table 4) by at least 510 MB/s when increasing the conflict chain size from 1 to 2. Therefore, the conflict chain size of G-match is set to one.

Table V
COMPRESSION PERFORMANCE

Test Files	Compression Ratio			Compression Speed (MB/s)		
	Snappy	G-match	Loss	Snappy	G-match	Imp.
File 1	1.033	1.033	<0.01%	446.2	2,222.5	5.0x
File 2	2.166	2.165	0.04%	267.7	1,937.5	7.2x
File 3	1.626	1.626	0.01%	217.9	1,797.5	8.3x
File 4	2.702	2.701	0.03%	237.2	1,707.5	7.2x
File 5	1.321	1.321	<0.01%	302.3	2,117.5	7.0x
File 6	1.365	1.365	<0.01%	230.6	1,970.0	8.5x

Table VI
COMPRESSION SPEED OF G-MATCH ON DIFFERENT GPUS (MB/S)

Test Files	GTX 680	GTX 780	GTX 980	GTX 1080
File 1	1310	2101	2222	4410
File 2	1131	1672	1937	3650
File 3	1090	1444	1797	3479
File 4	1009	1472	1707	3120
File 5	1139	1719	2177	4010
File 6	1111	1802	1970	3679

D. Compression Performance

To evaluate the compression performance of G-match and compare it with that of Snappy, we encode the output stream of G-match with the same encoding method used by Snappy. The output files can be decompressed directly by the decompression software of Snappy. The data compression ratio is defined as the ratio between the uncompressed and compressed data volume, the higher the better. Compression ratios and compression speeds on all test files are shown in Table 5. This experiment is run on Intel CPU E5-1620 and Nvdia GPU GTX980.

According to Table 5, the loss of compression ratio introduced by G-match is incredibly small: less than 0.05% in our experiments. This difference is mainly caused by parallel execution of 32 threads in hash table lookup and updating, as duplicate strings within a 32-byte unit may be lost in this process (Section 4.1).

It can be calculated from Table 5 that the average compression speed of Snappy on the six test files is 283 MB/s, and the average compression speed of G-match on the six test files is approximately 1.96 GB/s, which is approximately 6.9x larger than that of Snappy. Although compression speeds vary with files, the overall speedup values are stable and reasonable.

E. Scalability of G-match

We run G-match on 4 different GPU-cards, namely, GTX 680, GTX 780, GTX 980 and GTX 1080, to evaluate the scalability of G-match. All the experiments are tested on CPU i7-5930K, and each GPU card is plugged into the PC bus slot one at a time. The results are shown in Table 6.

Table VII
COMPRESSION PERFORMANCE OF DIFFERENT GPU IMPLEMENTATIONS- SPEED(MB/S) | RATIO

Test Files	G-match	GLZSS	CULZSS	CULZSS-bit
File 1	2222|1.03	1536|1.03	1078|1.01	825|1.02
File 2	1937|2.17	1325|2.09	721|1.79	679|2.02
File 3	1797|1.61	1338|1.8	846|1.2	780|1.79
File 4	1707|2.69	1479|2.73	880|1.5	790|1.80
File 5	2177|1.29	1779|1.30	979|1.10	802|1.13
File 6	1970|1.33	1525|1.25	825|1.15	745|1.19

It can be seen that G-match is efficient on different GPUs with different architectures. On the most advanced GPU GTX 1080, G-match achieves the highest speed of approximately 3.7 GB/s.

F. G-Match vs other GPU Implementations

The core indicators for measuring a compressor are the compression speed and compression ratio. In this section, we compare G-match with GLZSS, CULZSS and CULZSS-bit on GPU GTX 980 with all 6 test files. The compression performance is shown in Table 7.

Table 7 shows that G-match achieves the highest compression speed in all test files. The speedups over GLZSS, CULZSS and CULZSS-bit are approximately 33%, 140% and 160%, respectively. The compression ratios between G-match and GLZSS are close because the two algorithms adopt a similar hash table lookup mechanism. Relative to CULZSS and CULZSS-bit, G-match improves the compression ratio by 22% on average.

VII. CONCLUSION

In this paper, we propose a GPU-friendly data compression algorithm that is the first thoroughly parallelized algorithm of this type. The most notable contribution is to divide the whole process of scanning a byte into two separate passes and introduce an intermediate data structure to interface with them. This design allows the two passes to be parallelized independently, eliminating the main source of path divergence in the algorithm. Concurrent hash table lookup and updating achieves the maximum gain of speed at the cost of a slightly decreased compression ratio. The path divergence in the longest common prefix match is eliminated by decomposing a long string match into multiple 4-byte string matches with an intermediate data structure saving the match results. This design allows each thread to make progress in all circumstances. Experiments of performance breakdown verify the effectiveness of our design and optimizations. Compared with other existing GPU compression implementations, G-match achieves the highest speed with almost no compression ratio loss.

REFERENCES

[1] Zigang Zhang, Yinliang Yue, Bingsheng He, Jin Xiong, Mingyu Chen, Lixin Zhang, and Ninghui Sun. Pipelined compaction for the lsm-tree. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 777–786. IEEE, 2014.

[2] Jeff 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.

[3] S Sandeep Pradhan, Julius Kusuma, and Kannan Ramchandran. Distributed compression in a dense microsensor network. *Signal Processing Magazine, IEEE*, 19(2):51–60, 2002.

[4] Adnan Ozsoy and Martin Swany. Culzss: Lzss lossless data compression on cuda. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 403–411. IEEE, 2011.

[5] Ritesh Patel, Yao Zhang, Jason Mak, Andrew Davidson, John D Owens, et al. *Parallel lossless data compression on the GPU*. IEEE, 2012.

[6] L Erdodi. 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.

[7] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[8] Yuan Zu and Bei Hua. Glzss: Lzss lossless data compression can be faster. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, page 46. ACM, 2014.

[9] Molly A O'Neil and Martin Burtscher. Floating-point data compression at 75 gb/s on a gpu. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 7. ACM, 2011.

[10] Wenbin Fang, Bingsheng He, and Qiong Luo. Database compression on graphics processors. *Proceedings of the VLDB Endowment*, 3(1-2):670–680, 2010.

[11] Adnan Ozsoy, Martin Swany, and Anamika 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.

[12] Adnan Ozsoy, Martin Swany, and Arun Chauhan. Optimizing lzss compression on gpgpus. *Future Generation Computer Systems*, 30:170–178, 2014.

[13] Adnan Ozsoy. Culzss-bit: a bit-vector algorithm for lossless data compression on gpgpus. In *Proceedings of the 2014 International Workshop on Data Intensive Scalable Computing Systems*, pages 57–64. IEEE Press, 2014.

[14] Project Gutenberg. https://en.wikipedia.org/wiki/Project_Gutenberg.