Enabling Efficient Random Access to Hierarchically Compressed Text Data on Diverse GPU Platforms

Yihua Hu, Feng Zhang, Yifei Xia, Zhiming Yao, Letian Zeng, Haipeng Ding, Zhewei Wei, Xiao Zhang, Jidong Zhai, Xiaoyong Du, and Siqi Ma

***Abstract*—The tremendous computing capacity of GPU offers significant potential in processing hierarchically compressed text data without decompression. However, current GPU techniques offer only traversal-based text data analytics; random access is exceedingly inefficient, limiting their utility significantly. To address this issue, we develop a novel and widely applicable solution that prompts random access to hierarchically compressed text data without decompression in GPU memory. We address three main challenges for enabling efficient random access to compressed text data on GPUs. The first challenge is designing GPU data structures that facilitate random access. The second challenge is efficiently generating data structures on GPU. The CPU is inefficient when generating data structures for random access, and this inefficiency increases considerably when PCIe transmission is incorporated. The third challenge is query processing on compressed text data in GPU memory. Random accesses, such as data updates, cause massive conflicts among countless threads. In order to address the first challenge, we develop several compressed GPU data structures, including indexing within the intricate GPU memory hierarchy. To handle the second challenge, we propose a two-phase process for producing these data structures on GPU. For the third challenge, a double-parsing design is proposed as a solution to avoid conflicts. We evaluate our solution on three platforms, two server-grade GPU platforms and one edge-grade GPU platform, using five real-world datasets. Experimental results show that random access operations on GPU achieve an average speedup of 52.98× compared to the state-of-the-art solution.**

***Index Terms*—Big data applications, data compression, parallel architectures, query processing, text analysis.**

Manuscript received 30 November 2022; revised 8 May 2023; accepted 27 June 2023. Date of publication 11 July 2023; date of current version 11 August 2023. This work was supported in part by the National Natural Science Foundation of China under Grants 62172419 and 62322213, in part by Beijing Nova Program, and Public Computing Cloud, Renmin University of China. This work was also supported by funds for building world-class universities (disciplines) at Renmin University of China. Recommended for acceptance by M. Si. *(Corresponding author: Feng Zhang.)*

Yihua Hu, Feng Zhang, Yifei Xia, Zhiming Yao, Letian Zeng, Haipeng Ding, Zhewei Wei, Xiao Zhang, and Xiaoyong Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, Beijing 100872, China (e-mail: chronomia@ruc.edu.cn; fengzhang@ruc.edu.cn; xiayifei0101@ruc.edu.cn; 2020201366@ruc.edu.cn; 2019201413@ruc.edu.cn; dinghaipeng@ruc.edu.cn; zhewei@ruc.edu.cn; zhangxiao@ruc.edu.cn; duyong@ruc.edu.cn).

Jidong Zhai is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: zhaijidong@tsinghua.edu.cn).

Siqi Ma is with the School of Engineering and Information Technology, University of New South Wales, ADFA, Sydney, NSW 1466, Australia (e-mail: siqi.ma@unsw.edu.au).

Digital Object Identifier 10.1109/TPDS.2023.3294341

I. Introduction

BEING one of the most powerful parallel accelerators [1], [2], [3], [4], [5], [6], GPUs have been effectively deployed to text analytics directly on compression (TADOC) [7], [8], [9]. By utilizing data redundancy, the GPU-based TADOC (G-TADOC) can achieve both time and space savings. Experiments indicate that G-TADOC, which combines TADOC and the extraordinary computing capacity of GPU, is capable of handling cluster-level text data processing [8]. However, G-TADOC supports only traversal-based operations that require traversing the whole compressed text data. Random access, another fundamental set of operations, is not well supported by G-TADOC as it involves only partial data and does not necessitate a full dataset scan. Moreover, with the prevalence of embedded GPU platforms, edge GPU has been deployed in diverse situations, such as object detection [10], [11], [12], [13], emotion recognition [14], [15], autonomous driving [16], genomic analysis [17], and traffic control [18], [19]. Besides, with the continuously surging demand for flexibility and scalability in data processing, edge devices are also playing an increasingly important role in the data processing field [20], [21], [22]. Therefore, it is critical and necessary to enable random access to enhance text data processing for diverse GPU platforms.

Enabling random access to compressed text data on GPU can yield considerable benefits. First, many query operations are based on random access. Consequently, enhancing random access on GPU enables the efficient execution of a variety of queries on compressed text data employing the high GPU compute capacity. Second, GPU offers massive parallelism. The involvement of GPU provides abundant possibilities in the time reduction for batches of random access operations. Third, text analytics consists of two fundamental data processing operations: traversal-based operations and random access operations. G-TADOC has already supported traversal-based operations, so optimizing random access on GPU can complete the functionality of G-TADOC. Moreover, since the GPU memory is limited, this technology can greatly alleviate the limitation induced by the lack of storage space on the GPU.

Many studies have explored the data processing of compressed data in diverse environments. Hierarchical compression refers to the compression using rules to represent repeated text data to reduce the amount of data [23]. The text data after using hierarchical compression are called hierarchically-compressed data. Sequitur [24], [25], [26], [27] is a representative of the hierarchical compression method and TADOC [28], [29]

extends it to traversal-based analytics on compressed text data. Zhang et al. [30] also studied random access to compressed text data on CPU. Zhang et al. [8] then extended TADOC in a heterogeneous environment, and exhibited significant potential in applying GPUs in text analytics. Because the technology of compressed data direct processing can greatly expand the amount of text data that the GPU can manage, Zhang et al. [9] applied TADOC to the embedded GPU environment. The compression-based data processing shows unprecedented opportunities in resource-constrained environments. However, to the best of our knowledge, no work explores the possibility of supporting efficient random access to hierarchically-compressed text data on GPU, not to mention the edge-grade GPU platforms.

Although facilitating random access without decompression on GPU can bring significant benefits, enabling these tactics confronts the following challenges. 1) In data structure generation, traditional methods [8], [9] generate the GPU data structures from the CPU side, and then copy these structures from CPU to GPU via PCIe, which is time-consuming. A more efficient solution is to generate the data structures directly on GPU. However, according to the studies [8], [9], distributing tasks of data structure construction to threads on GPU requires both element counting and offset recording. It is difficult for GPUs to allocate space to store these data for each vertex during graph traversal dynamically. 2) In query processing, random access operations on compressed data can result in severe data conflicts in parallelism. For instance, given the issue of data dependency, *insert* operations should be conducted exactly in the addressed sequence within the same file, since any changes made to the operation order can lead to misconceptions about the insert location. Moreover, the edge-grade GPU platform usually employs a CPU-GPU integrated architecture, which requires targeted optimizations.

To solve these critical challenges, we propose a series of novel solutions. 1) In data structure generation, we develop a novel GPU-based generation process at the rule-level during DAG traversal. To meet diverse requirements for random access, we develop a hybrid strategy of light-weight breadth-first traversal and sequence-guaranteed traversal to generate all required data structures. Then, we parallelize the traversal tasks by rules to maximize GPU parallel resource utilization. 2) In query processing, we develop a novel double-parsing strategy to minimize data dependencies among different operations on GPU. We develop a fusion of two functions for random access operations with data conflicts. The first function comprehends the parameter *offset* as relevant to previous changes within the same query batch, allowing it to handle nested insertions in the same location. The second function comprehends the *offset* based on the original text data status before changes, supporting query-level parallelism. The combination of the two functions covers practical applications in all cases while assuring good performance. It should be noted that our method is only for text data. Our initial work has been presented in the study [31], which provides only preliminary results without the diversity of GPU platforms. Compared to the previous study [31], we provide adaptation to different GPU platforms and add new insights from various perspectives.

We evaluate our solution on three platforms, two server-grade GPU platforms and one edge-grade GPU platform, using five real-world datasets with diverse characteristics. Compared to the state-of-the-art random access method [30], our solution can provide an average of 52.98× in random access operations and 56.35% time saving in data structure generation. In the detail of five random access operations, we attain the acceleration of 24.59× in *count*, 16.26× in *search*, 44.69× in *extract*, 53.52× in *append*, and 124.86× in *insert*, respectively. Moreover, our solution achieves a compression ratio of 2.92 on average.

We mainly provide the following contributions.

- We unveil the challenges and difficulties of enabling efficient random access to hierarchically-compressed text data on GPUs, and show our insights into common random access operations.
- We provide a novel design of GPU-based data structure generation, and deliver the first solution that can efficiently support random access to compressed text data on GPU.
- We adapt our solution to different GPU platforms and demonstrate the benefits from both power and cost perspectives.
- We evaluate our solution on both server- and edge-grade GPU platforms with five datasets, and prove the performance superiority over the state-of-the-art solution.

II. Background

A. Random Access in TADOC

TADOC [7], [28], [29], [30], [32], [33] is a novel hierarchically compression design that performs text analytics directly on compressed data. Different from traditional compression schemes, it does not pursue an extremely high compression ratio, but rather a compromise with performance. TADOC represents the text data in the form of context-free grammar (CFG) by describing words as different terminators and summarizing the repeated content fragments of the text as rules. By using the same rule to represent repeated strings, TADOC can achieve significant space savings. TADOC is a lossless compression method and we can restore the original uncompressed text data by parsing the hierarchical rules. To support cases with multiple files, TADOC joins unique file splitters into boundaries between files to indicate file separation. Moreover, the rule-based representation of TADOC can be regarded as a DAG, so common operations on TADOC-compressed data can be mapped to a DAG traversal problem.

Rule-Based Representation: Fig. 1 illustrates the working principle of TADOC on an example of two files. Fig. 1(a) shows the original text data, including two consecutive files with abstract words marked as wi. The CFG transformation of the input data is shown in Fig. 1(b). The root rule $R0$ implies the overall content of the input data. By recursively replacing the rules with its production on the right, we can restore the full scope of the primary text. Such representation delivers a succinct description of the raw data by concluding the repetitive strings such as "$w1w2$" in both files into one rule. The CFG representation can be viewed in its equivalent form of DAG organization, as displayed in Fig. 1(c). Edges in DAG indicate the hierarchical reference relation from parents to children.

Random Access Data Structures: We show the major data structures used in random access [30] in Fig. 1(d). We assume

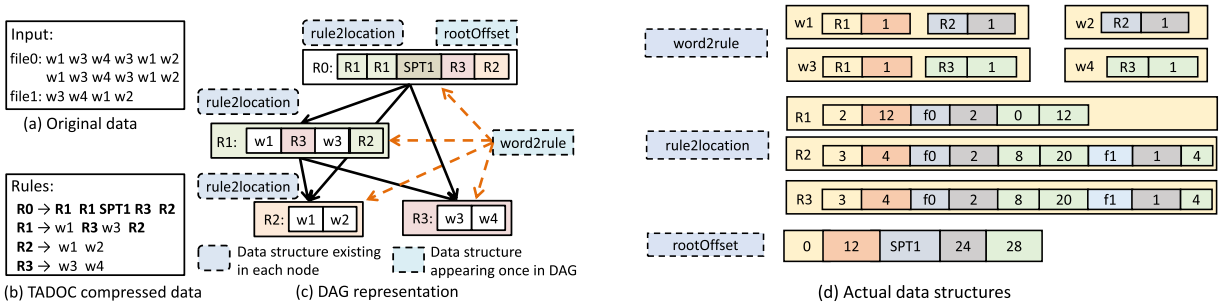


Fig. 1. TADOC illustration.

that the length of each word is 2. For *word2rule*, the entry for each word contains the pairs of rules and frequencies that the word occurs in the related rules. For *rule2location*, as to each rule, the locations in the original uncompressed files are represented as $\langle total, length, \langle file_1, num_1, start_{11}, start_{12}, \cdots \rangle, \ldots, \langle file_i, num_i, start_{i1}, start_{i2}, \cdots \rangle \rangle$, where $total$ is the total number of locations, $length$ is the length of the rule, $file_i$ is the ith file the rule appears, num_i is the number of the rule in $file_i$, and $start_{ij}$ is the jth start location of the rule in $file_i$. For *rootOffset*, it stores the offsets for different elements in the root.

Random Access Example: We use *search(file1,w1)* as an example to show the workflow of random access operations in Fig. 1(c). Assume that the length of a word is 2. In this example, the operation searches for all occurrences of $w1$ in $file1$ and returns the corresponding offsets. In detail, first, we fetch all appeared rules of $w1$ in the data structure *word2rule*, which are $R1$ and $R2$. Second, we obtain all the appearing locations in all files for each rule from the data structure *rule2location*, and then filter out the specific records of the target file, which is $file1$. We obtain the rule location $\langle R2, 4 \rangle$ in this step. Third, we search through $R2$ and record all $w1$ offsets within the rules. By merging the word offset $\langle w1, 0 \rangle$ with the rule location $\langle R2, 4 \rangle$, our search function returns the result *offset 4*.

B. GPU

GPUs are common heterogeneous accelerators in HPC domain [1], [2], [3], [4], [5], [6], [34], [35] and its application scope is gradually expanding from scientific data analytics to data science domain [36], [37], [38], [39], [40], [41], [42], including text analytics. These data science problems can be turned into HPC problems to solve. In detail, GPU assists different kinds of assignments in improving their processing throughput in parallel, and has already been demonstrated to be successful in accelerating traversal-based text analytics in TADOC [7], [8], [9]. Random access queries, such as counting for a specific word, can have low data dependency in the same batch, making them more compatible with GPU working styles. Moreover, since the GPU memory is limited, the compressed data direct processing technology can greatly alleviate the limitation caused by the fixed discrete GPU memory. Therefore, we consider it a promising solution during the conceptualization phase of our work.

Diversified GPU Platforms: With the development of parallel systems, GPUs are also integrated into embedded edge devices. Accordingly, we now have server-grade GPU platforms and edge-grade GPU platforms. The server-grade GPU platform usually adopts a discrete architecture, which is connected to CPU via PCIe. In contrast, the edge-grade GPU platform adopts an integrated architecture, which integrates both CPU and GPU on the same chip, sharing the same memory. More optimization details are discussed in Section V.

GPU Utilization: To optimize applications on the GPU, we need to consider the unique GPU architecture design. GPU involves drastically more arithmetic logic units (ALUs) for parallel data computation and consequently less space for cache systems, which is different from CPU. The large number of streaming multi-processors (SM) consisting of massive lightweight cores provides GPU with the ability to work on multiple pieces of data simultaneously. However, dependencies and data conflicts exist among tremendous threads. Besides, fully utilizing various kinds of APIs contributes to promoting the performance of GPU applications [43], [44]. All these factors need to be considered in enabling random access to hierarchically-compressed data on GPUs.

III. Motivation

A. Revisiting Random Access in Compressed Data

With the development of Big Data technology, the amount of data to be processed becomes very large [45], [46], [47], and the demand for high-throughput text random access has gradually emerged [48]. Efficient random access can help to uncover valuable information in text data with high information density, which includes but is not limited to news [49], legal affairs [50], [51], webpages [52], [53], medical records [54], [55], and logs [56].

Under this circumstance, our work enabling random access to compressed text data well supports the construction of online text analytical processing with five random access operation interfaces. Besides, with the assistance of GPU, we can use only one heterogeneous server to meet the needs of a large number of users, which greatly reduces the burden of the platform.

We revisit previous random access in TADOC [30], which involves the following five common operations to support various random accesses to compressed data. As for deletion, it is not common in the domains we examine since this technology targets datasets with long-term value [30], [32].

- *extract(f,pos,len):* The operation *extract* returns the string in file f at location pos with length len.

- *search(f,w):* The operation *search* returns the offsets of word w from file f.
- *count(f,w):* The operation *count* returns the number of frequencies of word w in file f.
- *insert(f,pos,str):* The operation *insert* puts string str at location pos in file f.
- *append(f,str):* The operation *append* adds string str at the end of file f.

These common random access operations are essential for text analytics on GPUs. Given that G-TADOC generates 31.1× performance speedup compared to TADOC [7], [8], [9], we believe that enabling efficient random access on GPU also provides significant performance benefits. However, before we show the challenges of developing these operations on GPU, we need to first clarify why previous GPU-based traversal is inapplicable.

Why Previous GPU-Based Traversal Does not Work? The specialties in random access require complicated data manipulations between rules, which have not been considered by previous works [8], [9]. In detail, first, each node in the DAG represents a rule, which includes additional data structures such as subrules and words. Previous GPU-based traversals do not involve these contents. Second, the majority of previous works [57], [58], [59], [60], [61] rely on BFS, which is simple for parallelization. To obtain the offsets of each element in the DAG on the GPU, the data structure generation process needs to perform sequential guaranteed DAG traversal, which is different from BFS. Third, random access necessitates fast locating the data that need to be accessed, which requires indexing. There is no previous literature on building GPU indexes based on grammatical rule compression.

B. Challenges

Developing efficient random access to hierarchically-compressed data on GPU needs to handle the following challenges.

Random Access Data Structure Generation: Distributing the DAG traversal process to parallel GPU threads for data structure generation has three challenges.

- Most data structures of random access operations in TADOC are constructed dynamically on the fly, which cannot be calculated in advance, meaning that the size of the used space is unknown until the building process is complete. Different from the CPU, the GPU cannot handle dynamic memory allocation efficiently, especially among multiple parallel threads.
- The generation process of data structures requires the traversal of the entire DAG, but different structures have distinct construction requirements. For example, the data structure *ruleFreq* needs to calculate the frequency of rules that appear in different files, while *rule2location* is designated to record the offset of each rule's appearance. The differences in objectives lead to the diversion of the traversal method choices.
- The sequence-guaranteed depth-first traversal is essential for the validity of offset calculation. However, the depth-first traversal is hard to parallel due to the dependencies among vertices, and can cause unbalanced thread load.

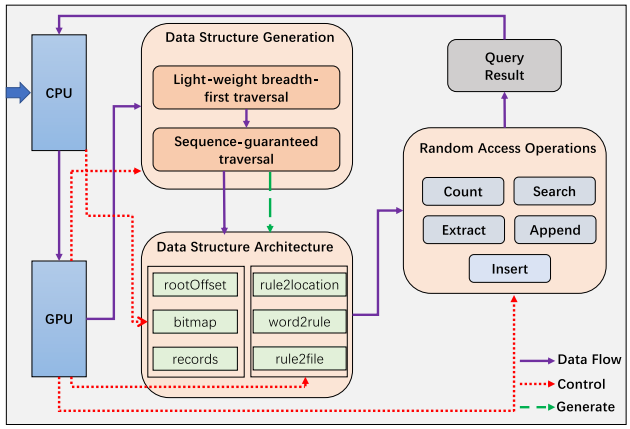


Fig. 2. Solution overview.

Data Conflicts in Random Access Operations: Executing random access operations in parallel among massive GPU threads undoubtedly leads to performance acceleration. However, operations that specify the modification location by file indexes and offsets can cause severe data conflicts. We consider two exemplary insert queries, $insert_1$ and $insert_2$. The operation $insert_1$ puts the string "a" at *offset 6* in $file1$, and $insert_2$ puts the string "b" at the beginning of $file1$. The queries come as "$insert_1, insert_2$", meaning that $insert_1$ is expected to perform first with the insertion of "a" at *offset 6*. However, the reversal in order can cause the string "a" inserted one byte before the expected position. Therefore, if two threads are allocated to execute the two insertions, the result can be different for the conflicts of accessing the offset and uncertainty of the processing sequence. Applying locks in the parallel environment can solve the problem of data conflicts, but at the cost of significant performance degradation.

Moreover, for the diverse GPU platforms, additional optimizations and adaptations are required.

IV. Our Solution

A. Overview

We show the overview of our solution in Fig. 2. Our solution consists of three modules: GPU-based data structure architecture module, data structure parallel generation module, and random access operation module. All relevant data structures are generated and stored directly in the GPU, which avoids unnecessary data transmission overhead between the CPU and GPU. The input is TADOC compressed data, and the output is the result of random access operations.

Workflow Between Different Modules: The three modules work together to enable random access to compressed data on the GPU. In the data structure architecture module, we cover all data structures in the study [30], and develop the GPU-based CSR buffer and Hash table to record the frequency of words and rules, which can achieve significant space savings while assuring fast indexing performance. They are capable of supporting operations in the random access module. The data structures are generated by the data structure generation module, in which we develop a GPU-based two-phase traversal for preprocessing.

The preprocessing calculates the space occupancy in advance, which is a prerequisite for allocating data structures. In the random access operation module, we utilize parallel threads provided by GPU to process multiple random access operations simultaneously. For output generation, the results from separate threads are to be merged and copied back to the CPU.

Solution to Challenges: Our solution can address the challenges mentioned in Section III-B. The two-phase traversal can manage the counting and offset calculation to address the challenges in the generation of random access data structures on GPU. Since the sub-DAG of the same rule is identical, we can avoid processing the same content in different working threads by parallelizing at the rule level from the root. Besides, parallelism on the rule level provides high utilization of thread computing resources. To handle data conflicts in random access, we develop a double-parsing design. In the first parsing, we assume that offsets provided by the same batch of update operations are relevant to all former operations, implying the restriction on the sequence of updates within the same file. In the second parsing, we regard the batch of updates come at the same time. The given offsets are based on the same text status and are independent of other concurrent insert operations. For the edge-grade GPU platform, we provide adaptation detailed in Section V.

Difference From Previous Work: Our solution has significantly different application scenarios compared to the previous G-TADOC [8]. First, G-TADOC necessitates the scanning of the whole dataset. In contrast, our solution aims to involve the minimum memory footprint, focusing more on data locality on GPU. Second, previous G-TADOC does not involve data changes, while our solution involves updates to the compressed data on GPU. Third, previous G-TADOC launches massive threads for the same traversal-based analytics task, which is not referable due to the task diversity of threads in our work. Besides, our solution is significantly different from previous random access on CPU [30]. Although the previous random access work [30] adopts similar data structures, it lacks fine-grained large-scale thread parallelism and cannot address the challenges mentioned in Section III-B. Therefore, the previous work [30] cannot be executed efficiently on GPUs.

Next, we explain in detail about these modules, including data structure architecture module in Section IV-B, data structure generation module in Section IV-C, and random access operation module in Section IV-D.

B. Data Structure Architecture

We show the detailed data structure architecture in this part. Different from the previous work [30], we propose a new data structure *rule2file* and use GPU-friendly storage formats to store our data structures. In detail, *rule2file* is used in *count* and *search* to achieve better performance, and we use thread-friendly buffers and GPU-based Hash tables to fully utilize the GPU. We cover the main data structures mentioned in Section II-A, which are well optimized on GPU.

Analysis: We can infer the sizes of the data structures *rootOffset*, *bitmap*, and *records* from the characteristics of the compressed data and the operations. These three data structures are stored in the form of one-dimensional arrays and can be allocated directly from the CPU. For the other data structures, including *rule2location*, *word2rule*, and *rule2file*, their space occupations keep changing during the generation process. Since GPU has limited ability to manage dynamic memory allocation between threads, we develop a preprocessing procedure, detailed in Section IV-C, to calculate the buffer size before data generation. We also extend the Hash table and CSR storage format to attain efficient indexing while ensuring space savings. We introduce the detailed data structures of *rule2location*, *word2rule*, *rule2file*, and *records* as follows.

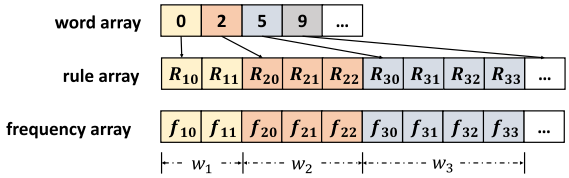


Fig. 3. Illustration of the $word2rule$ storage form.

word2rule: This data structure is useful for word indexing. It is stored in the CSR format in GPU memory. When receiving rules, our solution generates the *word2rule* index in ascending order. Fig. 3 shows an example of the storage form of *word2rule* on the GPU. It consists of three arrays: *wordArrary*, *ruleArray*, and *freqArray*. In detail, *wordArrary* provides the mapping from each word to a range in *ruleArray*, which stores all rules it appears; *ruleArray* records the specific rules that each word appears; *freqArray* records the appearing frequency for each word-rule pair. The frequency stored in *freqArray* corresponds to the rule index at the same location in *ruleArray*. To be specific, R_{ij} represents the $(j+1)$th rule where the ith word resides, and so does to *freqArray*. Given a random word, we can efficiently fetch rules with its presence and its frequency. For instance, when conducting the *search(*f_i, w_j*)* operation, we need to locate rules containing the word we are looking for. That is, we need to use *wordArray* and *ruleArray* in the *word2rule* CSR buffer: we first fetch *wordArray[j-1]* as l_j in the CSR buffer and then fetch *wordArray[j]* as h_j. After that, we can iterate through *ruleArray[*l_j, h_j*)* to obtain all rules containing wj. Performing *count(*f_i, w_j*)* is similar.

rule2location: This data structure is useful for offset-related operations. We record the relative offsets in the root rule due to the parallel generation process mentioned in Section IV-C. The rule-location mappings are stored in two arrays: *ruleArray* and *locationArray*. The first is *ruleArray*, which provides the mapping from each rule to its corresponding range in *locationArray*. The second is *locationArray*, which specifies the appearing locations of each rule. The number of occurrences of each rule in the original file, as well as the total number of occurrences of all rules, are unknown. To generate *rule2location*, in our two-phase traversal, we construct *ruleArray* in the first phase, and then construct *locationArray* with the assistance of *ruleArray* in the second phase.

rule2file: This data structure is constructed in the form of a GPU-based hash table to facilitate counting and searching. It

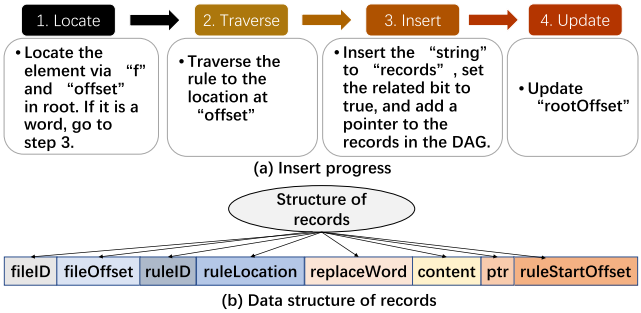


Fig. 4. Illustration of *records* and *insert*.

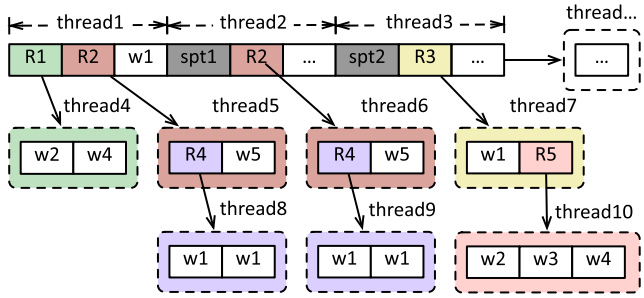


Fig. 5. Conventional GPU-based DAG traversal.

records the frequencies of rules in files and returns the number of occurrences for each rule-file pair. In detail, the key of this hash table is built featured with a <*file, rule*> pair while its value is the frequency of the <*file, rule*> pair. For *count* and *search* operations, it can work with *word2rule* to realize the indirect mapping from words to files. The hash table takes the rule index and the file index as parameters and supports multi-threaded *insert* and *search*.

records: This data structure is used for *insert* and *append*. The insertion progress is shown in Fig. 4(a) and the content of *records* is shown in Fig. 4(b). When performing *insert*, we need to integrate the inserted data into *records*, where *fileID* represents the exact file we insert, *fileOffset* represents the offset we insert in this file, *ruleID* is the rule we insert, *ruleLocation* is the inserted location of the rule, *replaceWord* is the original word we insert, *content* is the string we insert, *ptr* is the list pointer to the *recordID* inserted at the same place (default as *null*), and *ruleStartOffset* is the starting offset of the rule to insert. An example of *insert* using *records* is illustrated in Section IV-D.

C. Data Structure Generation

Analysis: As discussed in Section IV-B, to generate the data structures for random access, we need to calculate the space sizes for different data structures in the first phase, and then allocate the memory space according to the determined sizes in the second phase. We show an example of traditional BFS-based traversal in Fig. 5, where each node is associated with a thread. However, it has three limitations. First, it treats the rules that appear multiple times as distinct rules, which cannot utilize the data redundancy. Second, it associates each rule with one thread, which involves large data transmission overhead between

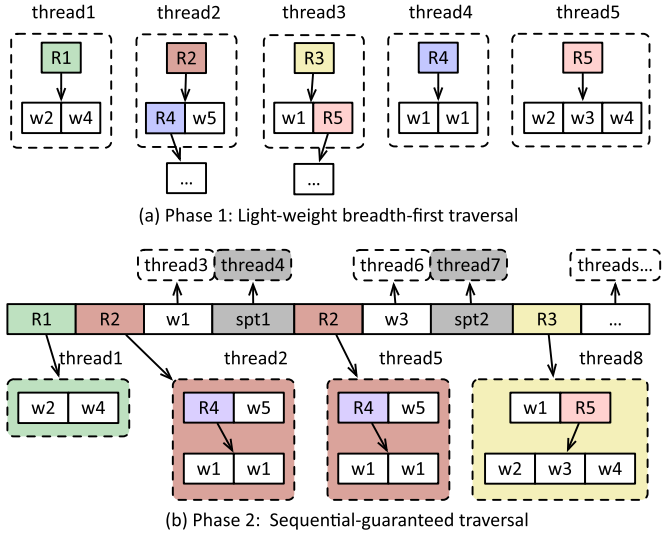


Fig. 6. Two-phase DAG traversal on GPU.

threads. Third, to calculate offsets, a sequence-guaranteed DFS traversal is required. Based on these reasons, we abandon the conventional DAG traversal design.

Two-Phase Traversal Design: We develop a two-phase traversal that is capable of efficiently generating the random access data structures. Fig. 6 illustrates our two-phase traversal. The first phase is a light-weight breadth-first traversal, as shown in Fig. 6(a). It needs to output the rule frequencies for memory allocation. The second phase is the sequence-guaranteed traversal, as shown in Fig. 6(b), in which we generate data structures based on the results from the first phase. This design utilizes the computing resources of threads to the maximum, attaining substantial time savings compared with the conventional design. We show the detailed design of the two phases as follows.

Phase 1: Light-Weight Breadth-First Traversal: The first phase is used for calculating rule frequencies.

1) Design: We utilize the GPU kernel to perform probing at each DAG depth in this phase. During the first phase traversal, we traverse through the DAG from top to bottom for rules, and record the frequencies of all rules of the text data to a temporary array *ruleFreq*.

2) Algorithm: We show the pseudo-code of the first phase in Algorithm 1. The function *breadthFirstTraversal* sets the variable *stopFlag* to false at line 2 and initializes the two arrays *currFreq* and *nextFreq* at line 3. The array *currFreq* records the rule frequency from the higher depth, while the array *nextFreq* records the rule frequency in the processing depth. The do-while loop of lines 4-8 repeatedly counts the rule frequency of the current DAG depth. The variable *stopFlag* is responsible for judging the cease time for the do-while loop. In each iteration of the loop, the *stopFlag* is first set to $true$, and the GPU kernel *getFreq* is then invoked to calculate the rule frequencies in the current depth and update *stopFlag*. After the call to the kernel, elements in the *currFreq* shall be set to 0, and the references to *currFreq* and *nextFreq* shall be swapped in line 7. Therefore, *nextFreq* generated in the current depth can be directly utilized in the next iteration.

Algorithm 1: Phase 1: Light-Weight Breadth-First Traversal.

```
1: function breadthFirstTraversal(ruleFreq)
2:   stopFlag ← false
3:   initialize currFreq and nextFreq
4:   do
5:     stopFlag ← true
6:     getFreq(currFreq, nextFreq, ruleFreq, stopFlag)
       with at least rules.size threads
7:     swap currFreq and nextFreq
8:   while stopFlag is false
9: end Function
10: function getFreq(currFreq, nextFreq, ruleFreq,
    stopFlag)
11:   ruleIdx ← tid                          ▷GPU thread ID
12:   if ruleIdx is not in the range of rule indexes then
13:     return
14:   end if
15:   if currFreq[ruleIdx] is 0 then
16:     return
17:   end if
18:   updateFreq ← currFreq[ruleIdx]
19:   atomicAdd(ruleFreq[ruleIdx], updateFreq)
20:   for each subEleIdx in rules[ruleIdx] do
21:     if subEleIdx is in the range of rule indexes then
22:       atomicAdd(nextFreq[subEleIdx], updateFreq)
23:     end if
24:   end for
25:   stopFlag ← false
26:   currFreq[ruleIdx] ← 0
27: end Function
```

The function *getFreq* calculates the frequency for each rule based on *currFreq*. Line 11 stores the thread ID in *ruleIdx*. Then, if *ruleIdx* is not in the range of rule indexes, that is, between 0 and *rules.size*, the kernel returns at line 13. For rules that do not appear in the last depth, the kernel directly returns in line 16. Line 18 sets the variable *updateFreq* as the rule frequency in the last iteration of *ruleIdx*. Line 19 updates the rule frequency for *ruleIdx*. For each directly derived element *subEleIdx* of *ruleIdx*, we first check if the element is a rule in line 20, and add the rule frequency of *ruleIdx* at the relative position in the array *nextFreq* in line 22 if the element is a rule. In this way, we distribute the tasks to threads on the rule level. Line 25 sets *stopFlag* to $false$, indicating searching necessity in deeper depth. At the end of the kernel, the frequencies in *currFreq* are refreshed in line 26 for future updates at the next depth.

3) Complexity analysis: We analyze the complexity of Algorithm 1 in this part. The traversal of the DAG takes D rounds, where D denotes the diameter of the DAG. Each iteration finishes with the swapping of the arrays *currFreq* and *nextFreq*. In each iteration, we allocate one single thread to process the jth rule. We define the variables e_{ij} and c_{ij} as the existence and the processing cost of the jth rule in the ith level of the DAG respectively, and ℓ_j as the length of the jth rule. Note that $currFreq[j]$ differs in each level, even though this property is not explicitly exhibited in the pseudo-code:

$$e_{ij} = \begin{cases} 0, & currFreq[j] = 0 \\ 1, & currFreq[j] \neq 0 \end{cases}, \tag{1}$$

$$c_{ij} = e_{ij}\ell_j \vee O(1). \tag{2}$$

The theoretical time complexity of each level of DAG is bounded by two folds: 1) the average workload of all threads, and 2) the maximum cost of particular rules with extremely large rule length. In this term, the time complexity can be naturally inferred in (3):

$$Tb_i = O\left(\frac{1}{N_c}\sum_{j=1}^{N_r} c_{ij}\right). \tag{3}$$

where Tb_i denotes the time complexity of traversing the ith level with balanced workloads, N_r is the number of rules except for the root rule, and N_c is the number of processing cores of GPU.

Most rules can only be processed on one processing core, which introduces another time bottleneck, as shown in (4):

$$Tu_i = O\left(\bigvee_{j=1}^{N_r} c_{ij}\right). \tag{4}$$

In (4), Tu_i denotes the time complexity of traversing the ith level with unbalanced workloads, depending on the rule with the highest cost. In real-world circumstances, lengthy rules rarely appear, whose frequency can be approximated to a constant value or even omitted. Hence, the overall theoretical time complexity of the first-phase traversal can be written in the following form:

$$T = O\left(\sum_{i=1}^{D} Tb_i \vee Tu_i\right) \tag{5}$$

$$= O\left(\sum_{i=1}^{D}\left(\frac{\sum_{j=1}^{N_r} c_{ij}}{N_c}\right) \vee \left(\bigvee_{j=1}^{N_r} c_{ij}\right)\right) \tag{6}$$

$$= O\left(\frac{D}{N_c}\sum_{j=1}^{N_r}\ell_j\right). \tag{7}$$

Phase 2. Sequence-Guaranteed Traversal: The second phase is used to allocate and generate the data structures for random access.

1) Design: We utilize the GPU kernel to distribute the work on each root element to different threads in this phase. During the second phase traversal, we record the offset, file ID, and the ancestor in the root rule for each appeared rule to the data structure *rule2location*. Additionally, we keep track of the offsets within root elements in *rootOffset*.

2) Algorithm: We show the pseudo-code of the second phase in Algorithm 2. The *seqTraversal* function takes the root rule, which is stored in *rules[0]*, as the parameter. *seqTraversal* first generates *rootOffset* in line 2 to record the accumulative offsets for each root element. Then, it calls the GPU kernel *seqLaunch* in line 3 to execute sequence-guaranteed probing on each element

Algorithm 2: Phase 2: Sequence-Guaranteed Traversal.

```
1: function seqTraversal(rules[0])
2:   initialize rootOffset
3:   seqLaunch(rules[0], rootOffset) with at least
     rules[0].size threads
4:   aggregate rootOffset
5: end Function
6: function seqLaunch(rules[0], rootOffset)
7:   rootIdx ← tid                          ▷GPU thread ID
8:   if rootIdx is not in 0 to rules[0].size then
9:     return
10:  end if
11:  fileIdx ← getFile(rootIdx)
12:  eleIdx ← rules[0][rootIdx]
13:  seqScan(eleIdx, fileIdx, rootIdx, rootOffset)
14: end Function
15: function seqScan(eleIdx, fileIdx, rootIdx,
    rootOffset)
16:  if eleIdx is in the range of word indexes then
17:    rootOffset[rootIdx] += wordLength[eleIdx]
18:  end if
19:  if eleIdx is in the range of rule indexes then
20:    rule2file.append(eleIdx, fileIdx)
21:    newLoc.fileIdx ← fileIdx
22:    newLoc.rootIdx ← rootIdx
23:    newLoc.ruleOffset ← rootOffset[rootIdx]
24:    rule2location[eleIdx].append(newLoc)
25:    for each subEleIdx in rules[eleIdx] do
26:      seqScan(subEleIdx, fileIdx, rootIdx,
         rootOffset)
27:    end for
28:  end if
29: end Function
```

of the root rule simultaneously. After the invocation of the kernel, *rootOffset* stores the size of each root element. Line 4 aggregates the element sizes in the array to obtain the global offsets.

The GPU kernel *seqLaunch* invokes the device kernel *seqScan* to search through the root element identified by thread ID and sets *rootOffset* for this element. Line 7 stores the thread ID in *rootIdx*. Then, if *rootIdx* is not in the range of element indexes in the root rule (between 0 and *rules[0].size*), the kernel returns in line 9. Lines 11-12 store the processing root element index in *eleIdx* and the corresponding file index in *fileIdx*. Line 13 invokes the device kernel *seqScan* to perform sequence-guaranteed traversal for the root element identified by *rootIdx*.

The device kernel *seqScan* recursively probes the given rule or word in the sequence-guaranteed strategy. The processing element is identified by the parameter *eleIdx*. If *eleIdx* is within the range of indexes of word elements, *seqScan* adds the length of the word to *rootOffset* for the original root element in *seqLaunch*, as shown in lines 16-18. If *eleIdx* is within the range of indexes of rule elements, *seqScan* executes as in lines 19-28. Line 20 appends the ⟨*ruleIdx, fileIdx*⟩ pair to *rule2file*. Lines 21-24 record the rule occurrence with *fileIdx*, *rootIdx*, and *ruleOffset* in *rule2location*. Then, lines 25-27 probe each element within the rule by calling the same kernel *seqScan*. The invoked *seqScan* then searches derivative elements of this rule successively.

3) Complexity analysis: Algorithm 2 splits the root rule into multi-threads. For the ith thread, it traverses the whole sub-DAG of the i th element in the root rule. To ensure the sequence, we need to traverse the DAG in the depth-first order. The theoretical time complexity of Algorithm 2 is bounded by the following two folds: 1) the average workload of all threads, and 2) the maximum size of all the sub-DAGs we process. According to the statistics of all the datasets in the experiments, the length of the root rule dwarfs the processing cores in GPU. Meanwhile, the real-world data rarely appear repetitive patterns with excessive length, which bounds the maximum size of a single rule (except for the root rule). These two features of datasets guarantee the workload balance. In the i th thread, we traverse the entire sub-DAG of the ith element in the root rule and append a new token in the array *rule2location* for each rule in the sub-DAG. Then, the time complexity of Algorithm 2 is derived in (8), where N_{root} signifies the number of elements in the root rule, N_c signifies the number of processing cores, and s_i signifies the tree size of the ith element:

$$T = O\left(\frac{1}{N_c}\sum_{i=1}^{N_{root}} o_i \cdot s_i\right). \tag{8}$$

D. Random Access Operations

We assume that random access operations are performed in batches of thousands of operations. Our work utilizes the GPU to perform operations concurrently within the same batch. For operations with no data dependency, such as *count*, *search*, and *extract*, we arrange different threads to handle them separately. Besides, we develop special strategies to solve the data conflict in parallelism for *append* and *insert* operations. Additionally, our system supports mixed operations. Our detailed design of random access operation processing is as follows.

Count(file, word): We arrange each thread to process one count operation in the same batch of operations. The counting work utilizes the data structures of *word2rule* and *rule2file* to build frequency mapping from words to files using the intermediate rules. For the cases where *word2rule[word]* has a large content size, we allocate additional threads to help accelerate the counting processing. By default, we empirically set the number of threads to $\lceil$*word2rule[word]*/64$\rceil$.

Search(file, word): We schedule one thread to handle one search operation within the same batch. The searching process uses *word2rule* and *rule2location* to map words to their specific locations in the DAG with the transitional rules. Extra threads are allocated in operation to handle the cases when the word has substantial occurrent rules.

Extract(file, offset, length): We also assign one thread for each extract operation in a batch. During the extracting process, we first search *rule2location* for the word that corresponds to the *offset*. Then, if the size of the current output is less than *length*, we add the pointing word by character to the output. After the

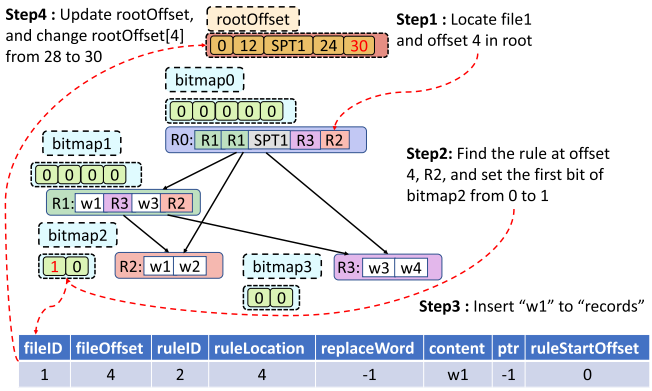


Fig. 7. Example of *insert*.

completion of adding the current word, we locate the following word in *rule2location* and repeat the process described above.

Insert(file, offset, string): Insert operations cause varied results if processed in different orders. We develop a novel double-parsing strategy to resolve the conflict between the processing order and parallelism. In detail, we devise two different *insert* functions, *insert in sequence* and *insert in batch*, which parse the argument *offset* in different ways in order to accommodate a variety of situations. The first function comprehends *offset* as the offset of the text data modified by the last *insert*. It inserts strings in the order specified by the batch, and has a comparatively lower speed because of the restrictions on parallelism. The second function regards the *offset* as the corresponding location of text data before this batch of *insert*s happens. It inserts strings with full parallelism and attains high performance. The primary distinction between the two functions is the parsing style of the parameter *offset* in the *insert* operation. Therefore, data conflict is resolved between insertions in the second case, which supports full parallelism. The two functions have different practical application scenarios. Although the first function has a comparatively low processing speed, it is indispensable when handling nested insertions within a batch. Meantime, the second function is capable of handling concurrent insertions efficiently. It should be noted that we do not recompute the DAG and generate new rules until the inserted entries reach the recomputation threshold we set in the system configuration. Once we reach the threshold, we merge *records* with the previous data, perform recompression to construct a new DAG, and recognize new rules. The detailed design is as follows.

- *Insert in sequence:* In this case, we assume the insertions in the batch are supposed to execute in the given order. We first allocate one thread for each file because of the sequence restriction in each file. Among these threads, insertions are processed in sequence. Every insertion includes three phases, the insertion locating phase, the recording phase, and the data structure updating phase, in which the last phase takes the majority of the time. The detailed process is shown in Fig. 4(a) and we take the example of Fig. 1 to perform *insert*(f_1, *4*, w_1) for illustration, assuming that the size of each word is 2 bytes. The *insert* progress is shown in Fig. 7. First, we locate *file1* and *offset 4* in the root. Second, we find the related rule $R2$. Third, we add the new content $w1$ to $records$ and update related entries. Fourth, we update *rootOffset*.
  During the data structure updating phase, we timely reflect changes in the text on data structures *rootOffset* and *records*. Updates to *rootOffset* increase the offsets behind the *insert* location by the inserted string length, and updates to *records* increase the offset of the previous records located behind the new insertion position. Both types of updates lead to modification on a large scale, so we invoke new kernels to process the update. Experiments show that such updates account for more than 90% of the operation time. Despite the limitation on sequence, this function still attains a preferable performance because the most heavy-load phase is parallelized with efficiency.
- *Insert in batch:* In this case, we assume the insertions in the batch are supposed to execute simultaneously. We partition the work on the operation level and assign each insertion to different threads. Since the parameter *offset* for each insertion is based on the text data before the operation batch happens, we keep the data structure constant in each thread. We integrate all updates to data structures in one round after the insertions. As in the function, *insert in sequence*, we invoke kernels to parallelize the increment on *rootOffset* and *records*. This function accomplishes parallelism on a higher level and exhibits impressive speedup.

Append(file, string, offset): We parallelize the *append* operations by arranging one thread responsible for each of the *append* operations. After filling into the data structure *records* in threads, we post-process *records* to solve data conflicts for appends at the same location. We assume that the operation appears earlier in the batch executes first, so we increase the offsets of the following records by the previous appending length. The post-processing is also performed on the GPU kernel. With the same number of operations, the performance of the *append* function is much better than *insert* because it does not have to deal with *rootOffset*. Similar to *insert*, the increased data size can be obtained from the string in the user-defined operation. Both *insert* and *append* store the increased data in a separate data structure of *records*, so the system only needs to update the corresponding offset positions.

Mixed Operations: We develop an offset-snapshot strategy to parallelize processing mixed operations while guaranteeing the operation execution order. On receiving a batch of mixed operations, we first collect all *insert* and *append* operations respectively, and send them to single-operation subsystems described above for parallel execution. These two operations can be processed in full-parallelism when performed on the same state of the text data, as *append* does not invoke offset changes to the same file, allowing parameters of the *insert* operations to be parsed correctly. During the execution process, we record the operations' order in batch to the data structure *records* for the rest non-updating operations to refer to. Additionally, as both operations cause offset changes in the data structures that will be accessed by the rest of operations such as *count*, our system will not finalize offset updates immediately after execution. Instead, it will create an array to record the update snapshots in the order

of execution and finalize all offset updates after the execution of all operations. Next, after performing all updates to the text data, we perform the rest read-only querying operations in parallel, including *count*, *search*, and *extract*. For the *count* operation that does not involve offset, it checks the hash tables to obtain the word number in the unmodified text data, and counts the word in the added context by searching through entries with a smaller value of execution order in *records*. For the operations taking *offset* as the parameter, including *search* and *extract*, they first use the offset update snapshot array to obtain the certain offset changes of the target file and then access data structures with offset changes to process queries. They will also make use of entries with a smaller value of execution order in *records* to complete the results.

E. Optimization Summary

We summarize our optimizations in data structure generation and random access operation as follows.

Thread Level versus Warp Level: When designing the two-phase traversal in Section IV-C, we can choose the parallelism granularity of processing rules, at thread level or warp level. We find that rules with elements less than 32 account for more than 99% of the total rules. Because the number of threads within a warp for Nvidia GPU is 32, using the warp level design cannot fully utilize the hardware parallelism. Accordingly, we choose the thread-level design, as shown in Fig. 6.

Load Balance: Load balancing is critical to GPU performance [62]. In designing light-weight traversal, in each layer shown in Fig. 6(a), we associate a thread with each rule. Because each rule contains an approximate number of elements and each thread processes the direct elements of the rule, this strategy can achieve good load balancing.

Locality: Locality plays an important role in GPU throughput [63]. The data structures in Section IV-B help GPU to locate the target data in the GPU memory instead of scanning the entire data for random access queries, alleviating the pressure of memory access for the GPU. For example, the combination of data structures of *word2rule* and *rule2file* manage to build the mapping from the target word to the file for *count* operation, while *word2rule*, *rule2location*, and *rootOffset* together construct the mapping from the target word to all its locations in a file for the *search* operation.

Occupancy: We make use of the CUDA API *cudaOccupancyMaxPotentialBlockSize* to determine the block size and grid size for each kernel to maximize the active warp ratio on the SMs. Moreover, occupancy is influenced by the execution of tasks within blocks and the number of registers utilized by each thread. In our kernel design for random access operations, we allocate extra threads to process *count* and *search* operations involving additional data structure accesses to achieve a more even workload distribution among threads. We also make use of *NCU* to find the optimal number of registers for each thread, which is 32.

Adaption to SM Architecture: We incorporate algorithm designs to make use of the SM architecture in data structure generation and the execution of random access operations. For data structure generation, in the first-phase traversal, we load a portion of the data structure, specifically the set of consecutive rules that are stored in adjacent locations of the $rules$ data structure, into shared memory to accelerate processing, as each thread block consistently handles the same set of rules. In the second-phase traversal, due to the DAG structure randomness as well as the sequential traversal requirement, adaption strategies to the SM architecture are limited. We have explored methods of reusing rules at the root level and bottom-up traversal, but both show performance reduction. Our future work targets managing rules' sizes from the compression stage in order to dynamically balance workloads between threads by assigning subtrees with similar sizes.

For the five random access operations, how to adapt each operation to the GPU SM architecture depends on the storage features of involved data structures. Both *count* and *search* operations utilize hash tables to fetch word-to-rule and rule-to-file frequencies. Additionally, *search* queries *rule2location* with random rules, making memory access inconsistent for each SM. Therefore, we can only utilize the shared memory in SM for these two operations when invoking multiple threads to process one complex query. The *append* operations are fully adaptable to the SM architecture. As *append* operations are expected to perform exactly as the query arrival order, we arrange adjacent queries to the same thread block and load the corresponding part of the linear data structure *records* to the shared memory.

Both *extract* and *insert* operations are applicable to certain SM optimization by sorting queries with offset, fetching the smallest subtree for each SM in advance, and then loading the subtree content to the shared memory. This optimization brings about 10.47% and 3.22% performance improvement in *extract* and *insert* respectively, but results in 75.94% and 107.56% more time consumption (including preprocessing) compared with the current realization. In this case, we consider that the processing overhead outweighs the benefits of certain SM adaption in these two operations. Notably, we have incorporated the shared memory utilization in the *insert* operation's after-insert update part as both $records$ and $rootOffset$ follow a linear update pattern.

V. Adaptation to Different GPU Platforms

With the development of architecture and the growth of various application requirements, GPUs become more diverse and adaptable. In the previous optimizations, we mainly focus on the GPUs deployed on server-grade GPU platforms. In this section, we also explore random access optimizations and adaptation to edge-grade GPU platforms.

A. Server-Grade GPU Platform

Nowadays, a growing number of data analytics tasks adopt heterogeneous hardware for acceleration, and the discrete GPU often serves as an acceleration component of the server-grade platform, enhancing the computing capacity of the whole system. In such heterogeneous systems, CPU, as a host device, transmits the input data to the GPU. The GPU generates data

structures and performs the operations. This is one of the most traditional heterogeneous computing usages.

Analysis of the Discrete Server-Grade GPU Platform: On the discrete server-grade GPU platform, both the CPU and the GPU have their own discrete memories, which are not shared, so communication relies on PCIe to transfer data between them. The host CPU is usually used to send data. The GPU device is responsible for executing computing tasks such as *extract* and then sending back the results to the host. The server-grade GPU has complicated the memory hierarchy, prompting us to design code carefully to fully utilize this hierarchy.

Adaptation of Random Access to the Server-Grade GPU Platform: Our adaptation of random access to the server-grade GPU platform is mainly considered from the programming model perspective. Nvidia GPUs, as one of the most widely used server-grade accelerators, have been applied to diverse data centers and the cloud, and they adopt the CUDA programming model [64]. Accordingly, we use CUDA to demonstrate the benefits of our random access optimizations on GPU. Specifically, we use the CUDA API, *cudaMalloc()*, to allocate the GPU memory buffer, and use *cudaMemcpy()* to transfer data from CPU to GPU. Accordingly, all the optimizations in Section IV are developed in CUDA, and the effectiveness is verified in Section VI.

B. Edge-Grade GPU Platform

With the development of edge computing and IoT [65], [66], [67], [68], [69], [70], edge platforms also integrate GPU devices boosting the application in many fields. For example, Jose et al. [14] applied Nvidia Jetson TX2 edge GPU for a surveillance system. Different from the server-grade GPU platform, the edge GPU is usually integrated with the CPU together, and both the CPU and the GPU on the edge share the same memory, which brings great potential convenience and further optimization opportunities to our random access design on the compressed data.

Analysis of the Edge-Grade GPU Platform: Many vendors have released their edge-grade GPU platforms, which adopt a CPU-GPU integrated architecture, such as Nvidia's Jetson series. The fusion design of the integrated edge platform has two distinct features. First, on the edge platform, GPU and CPU share a unified memory, which is scheduled and allocated by the memory controller fabric. This structural design greatly improves the collaborative processing capability of the GPU and the CPU, avoids repeated memory copies and data transmission via PCIe, and improves program utilization efficiency. Second, due to the cost and energy efficiency constraints, the computing capacity of the edge-grade GPU platform is lower than those of the discrete server-grade GPU platform. For example, the edge GPU platform Nvidia Jetson AGX Xavier has only 512 cores with 854 MHz base clock speed [71], [72], [73], while the discrete GPU platform Nvidia RTX 3090 has 10,496 cores with 1,395 MHz base clock speed [74], [75].

Random Access Adaptation to Edge-Grade GPU Platform: For the edge-grade GPU platform, we still use CUDA to enable random access to the compressed data. It should be noted that although Nvidia's edge-grade GPU platform, like the Jeston series, is different from the discrete GPUs, the edge platform is still compatible with the server-grade GPU syntax; that is, we can still use the CUDA API like *cudaMalloc()* that we use on the server-grade GPU platform. However, this does not utilize the novel features of the integrated design and can drag down the overall performance. Therefore, we use its feature of unified memory for the structures of random accesses. Under this programming model, *cudaMallocManaged()* is used to allocate a section of memory in the unified memory, which can be directly accessed by both CPU and GPU. Hence, we avoid using the combination of *cudaMalloc()* and *cudaMemcpy()* to allocate a section of duplicate memory buffer on the discrete GPU and transfer data through PCIe. Although our solution in Section IV has reduced the data transmission from CPU to GPU as much as possible, we still have unavoidable data transmission such as the input data, which we have to transfer from CPU to GPU in the discrete design. Hence, in processing large data, the limited PCIe bandwidth often drags down the whole executing time. Fortunately, this is solved on the integrated architecture of the edge-grade GPU platform. Meanwhile, due to changes in the memory structure, requirements have been placed on the utilization of edge GPU memory. Therefore, to utilize the sophisticated memory hierarchy on the edge platform, we adopt a more fine-grained strategy to allocate variables. First, we assign all constants in constant memory instead of unified memory to better use its low latency. Second, we put small data structures like *bitmap* into shared unified memory due to their relatively small size. Third, we allocate other huge data structures to global unified memory to avoid two copies of the same data as well as perform better-multithreaded access, thus achieving benefits in both performance and memory.

VI. Evaluation

A. Experimental Setup

Methodology: The baseline we compare to is the random access implementation of TADOC [30], which is the state-of-the-art method identifying and realizing the five common random access operations in compressed text analytics. It accelerates random access to hierarchically-compressed data and eliminates the limitation on compressed data updates. Our solution enables efficient random access in the GPU environment. In our evaluation, we measure the performance of our solution and the state-of-the-art method from multiple dimensions. We generate 100,000 queries for the five different types of random access operations. For *count* and *search*, we choose a word at random from a file's vocabulary. For *extract*, we extract content by selecting random offsets in a file, and we set the average length of the extracted content to be 64 bytes. For *insert* and *append*, the string to be inserted or appended is made up of randomly selected words from the dictionary, and the average length of the inserted string is 64 bytes; the offset is random for *insert*.

Platform: We evaluate the methods on three platforms, as shown in Table I. We use two server-grade GPU platforms with different architectures, Nvidia Geforce RTX 2080 Ti GPU (Turing architecture) and Nvidia Geforce RTX 3090 GPU (Ampere architecture), in our experiments. The RTX 3090 platform

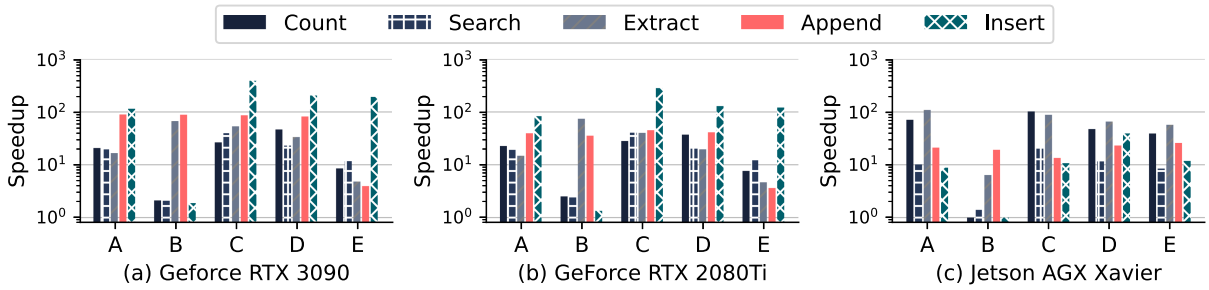


Fig. 8. Speedups of random access operations.

TABLE I
EXPERIMENTAL PLATFORMS

Platforms	Edge-Grade	Server-Grade	
	Jetson AGX Xavier	RTX 2080 Ti	RTX 3090
Cores	8 CPU cores, 512 GPU cores	4352 GPU cores	10496 GPU cores
GFLOPS (float)	1410	13450	35580
Bandwidth (GB/s)	137	616	936.2
Price ($)	699	999	1499
TDP (W)	30	250	390
Programming Model	GPU : CUDA 8.0, CPU : OpenMP	CUDA 8.0	CUDA 8.0

TABLE II
DATASETS ("SIZE" REPRESENTS THE ORIGINAL UNCOMPRESSED SIZE)

Dataset	Size	File #	Rule #	Vocabulary Size
A	2.1GB	4	2,095,573	6,370,437
B	580MB	134,631	2,771,880	1,864,902
C	2.9GB	1	8,821,630	23,959,913
D	62MB	1	36,882	240,552
E	50GB	109	57,394,616	99,239,057

is equipped with an Intel(R) Core(TM) i9-10900X CPU and 128 GB memory, based on the Ubuntu 20.04.3 LTS (GNU/Linux 5.8.0-55-generic x86_64) operating system. The RTX 2080Ti platform is equipped with an Intel(R) Core(TM) i9-9900 K CPU and 64 GB memory, based on the Ubuntu 20.04.2 LTS operating system. The baseline uses the i9-10900X CPU of the RTX 3090 platform. We also adapt our optimization to an edge-grade GPU platform, Nvidia Jetson AGX Xavier. It comprises an integrated 512-core Nvidia Volta GPU and 8-core Carmel ARMv8.2 64-bit CPU. The operating system we use is Ubuntu 18.04.4.

Datasets: The datasets used in our evaluation are shown in Table II. These datasets are available and widely used in previous works [7], [8], [9], [28], [29], [30], [32]. Dataset A is a Wikipedia collection consisting of four files [76]. Dataset B is NSF Research Award Abstracts (NSFRAA) collected from UCI Machine Learning Repository [77]. Dataset C is a DBLP collection of web documents [78]. Dataset D is COVID-19 data collection from Yelp [79]. Dataset E is a large Wikipedia dataset [76].

B. Performance

In our evaluation, the performance speedup of our solution over the baseline is shown in Fig. 8. On average, our solution outperforms the baseline by 52.98×. Specifically, our solution achieves 24.59× speedup for *count*, 16.26× for *search*, 44.69× for *extract*, 53.52× for *append*, and 124.86× for *insert*. We have the following observations.

First, in terms of operations, we achieve relatively high speedups in the *append*, *extract*, and *insert* operations, and moderate speedups in *search* and *count*. The reason for the high performance in the first three operations is that we can perform these operations in nearly full parallelism. Although *count* and *search* perform in parallel on the operation level, the differences between query parameters cause significant differences between workloads in threads. For example, searching for a widespread word in a file is more time-consuming as the word can be distributed in many rules. For *insert*, *insert in batch* function attains a tremendous acceleration of 180.14× for its parallelism on the operation level. As to *insert in sequence*, it achieves a lower acceleration of 19.89× due to its narrow file-level parallelism.

Second, in terms of datasets, we find that most operations on dataset B have low performance. The average speedup for *count*, *search*, and *insert* on dataset B is 1.88×, while their performance on other datasets all attain an average speedup of over 20.0×. This is due to the file size of the text data. Dataset B has an average file size of 4.41 KB, which is much smaller than the file size of the other datasets. In this situation, operations that specify a particular file need to search through or modify the data structures within a narrow region, which the CPU is adept at. Meanwhile, the GPU takes a comparatively great time to invoke kernel functions for the light-weight tasks. However, even in this case, our solution still achieves clear performance benefits.

Third, in terms of the query batch size, when the query batch is relatively low, massive GPU threads do not have enough workload for parallelism. For example, the average performance speedup for counting with a batch size of 500 is 6.08×. In contrast, the speedup of the count operation reaches 23.93× with the query batch size of 10,000. The performance of insertion shows more variances with different query batch sizes, due to the enormous time costs for modification accumulating with massive *insert* operations.

In terms of mixed operations, we use a batch of 100,000 randomly generated mixed operations to evaluate the performance. The experimental result shows that our solution achieves a speedup of 23.56× on the RTX 3090 platform and a speedup of 21.79× on the RTX 2080 platform in throughput compared with the CPU. Although we take extra time in the snapshot construction and in parsing offsets with the snapshot for operations, our solution can still obtain clear performance advantages over the CPU version. We also measure executing the batch in

TABLE III
SPACE OCCUPANCY OF DIFFERENT DATA STRUCTURES

Storage space (MB)	A	B	C	D	E
rule2location	374	303	614	11	17623
rule2file	2.3	68	4.6	0.02	88
word2rule	14	15	43	0.2	378
rootOffset	71	108	342	1.2	6834

full parallelism while applying locks to operations to ensure the execution order. In this case, GPU exhibits throughput with 9.68% slow down compared with the CPU, validating that our offset-snapshot strategy provides an efficient way of avoiding conflicts between threads and attaining parallelism.

We also compare the GPU *insert* with the CPU *insert* with the same data structure. For random access operation *insert in sequence*, we disassemble the insert update into the update to *rootOffset* and update to *records*. The fine-grained update to *rootOffset* enables *insert in sequence* to achieve considerable speedup against CPU even with file-level parallelism on updating *records*. Experiment on dataset A with a number of four files shows that GPU *insert in sequence* achieves speedups of 22.83, 43.74, 47.66, 48.00, and 31.45 over the CPU version with the same data structure on 10, 100, 500, 5000, and 10000 insertions, respectively.

C. Space

We use the compression ratio of the original data size divided by the compressed data size as the metric to evaluate space savings. Our solution achieves a compression ratio of 2.92 on average. In detail, the compression ratios are 4.32, 1.07, 2.55, 4.84, and 1.80 for datasets A, B, C, D, and E respectively. For the data structures used in our experiments, *bitmap* and *records* do not exist until the update happens. Therefore, we exhibit only space occupancy of data structures of *rule2location*, *rule2file*, *word2rule*, and *rootOffset* in Table III. We have the following observations. First, the data structure *rule2location* accounts for most of the space. On average, *rule2location* occupies 66.29% space of all data structures. Second, *rule2location* and *rule2file* of dataset B have higher space occupancy ratios than the original text data size. The reason is that dataset B has much more files, requiring more space to record information distributed in different files. Third, the size of the data structure *word2rule* has a strong relationship with the vocabulary size. Datasets with a larger vocabulary size have a larger size of *word2rule*.

D. Data Structure Generation

Time Savings: We study the time savings in this part. The edge platform does not show clear benefits in time savings due to its architectural features of integrated memory. Hence, we report the time savings of our solution compared to the baseline on discrete architectures in Fig. 9, and we have the following observations.

First, our parallel GPU-based data structure generation attains an average time saving of 56.35% on the RTX 3090 platform and 48.28% on the RTX 2080Ti platform. We can see the general superiority of RTX 3090 over RTX 2080Ti in terms of performance.

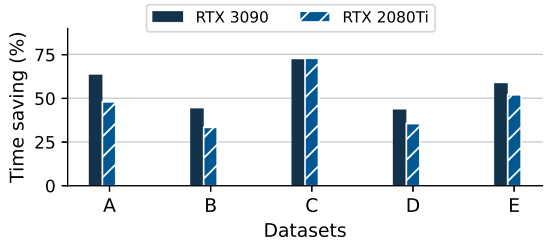


Fig. 9. Speedups of data generation.

TABLE IV
TIME BREAKDOWN OF DATA STRUCTURE GENERATION IN SECONDS

Time breakdown (s)	A	B	C	D	E
Data structure allocation	0.11	0.13	0.30	0.018	16.02
Phase 1 traversal	1.80	3.65	8.31	0.052	47.59
Phase 2 traversal	2.39	2.85	1.19	0.026	44.34

Second, the time savings of different datasets grow with the increment of the data size. Notably, data structure generation of dataset C, the dataset with the largest number of rules and vocabulary size, achieves the highest time saving of 72.14% and 72.75% on the two server-grade platforms respectively, proving the superiority of our GPU-based solution in handling massive text data.

Third, we find that our solution can bring considerable time savings in various cases. The time savings of our solution range from 43.50% to 72.14% on the RTX 3090 platform, and 33.33% to 72.75% on the RTX 2080Ti platform.

Time Breakdown: We show the detailed time breakdown in Table IV. On average, the data structure allocation time accounts for 8.23% of the total data structure generation time. Meanwhile, the phase 1 traversal occupies an average of 55.99%, and the phase 2 traversal occupies an average of 35.77%. For small datasets, the generation time is less than 10 seconds, and for the largest dataset, the generation time is less than two minutes. The preprocessing time is roughly equivalent to executing 60,000 inserts per gigabyte of the original text data, but we still regard it as acceptable because the built data structures are available for all future queries. Similar scenarios are common in the fields such as archive collection [50], [51] and record organization [54], [55].

E. Turnaround Time

We compare the turnaround time of both CPU and GPU on one query. The result shows that the average comparison ratios of the turnaround time with GPU to that with CPU are 0.098, 1.567, 1.873, 12.5, and 1.306 for operations *count*, *search*, *extract*, *append*, and *insert* respectively. When measuring the turnaround time, we assume that all related data structures already exist in the CPU or GPU and choose 1,000 randomly generated operations to compute the average time. Notably, for operations on the GPU, we include the input and output data transmission time in the turnaround time.

The results show that, in general, the CPU outperforms the GPU in terms of turnaround time for most operations. This is because transmitting queries and results between the CPU

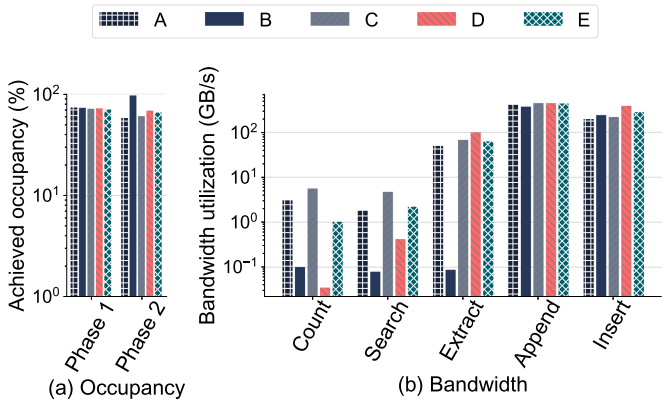


Fig. 10. Performance metric analysis.

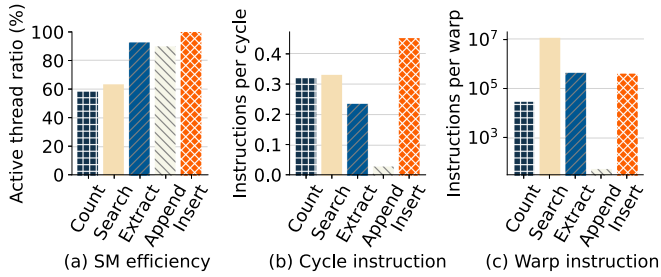


Fig. 11. Efficiency and parallelism metric analysis.

and GPU takes a non-negligible amount of time, and a single thread on the GPU has less computational power compared to the CPU. However, our GPU implementation still attains a very close absolute turnaround time to the CPU for all operations, including *append*, which shows a high time ratio between the platforms due to the short completion time compared with the data transmission time. Additionally, GPU also performs well on queries relating to large scales of the searching range in the text data as we conduct thread adaptation for complex queries to parallelize constituent tasks in a single operation, including *count*, *search*, and *insert*.

F. Hardware Metrics

We further measure the hardware metrics of our solution, including achieved occupancy, bandwidth utilization, SM efficiency, and cycle instruction, utilizing the tool $NsightComputeCLI$ provided by Nvidia [80]. Achieved occupancy reflects the parallelism. Bandwidth utilization shows the bandwidth utilization of different operations. SM efficiency represents the efficiency. Warp instruction illustrates the load differences between different operations. Cycle instruction shows the instruction-level parallelism. We use the GTX 3090 platform for illustration, and the other platforms exhibit similar performance behavior. Besides, we use the Jetson AGX Xavier platform to illustrate the benefits of adaptation to edge GPU.

Achieved Occupancy: The achieved occupancy represents the rate of active warps to the maximum number of warps during kernel processing. We measure the achieved occupancy of our two-phase traversal, and show the results in Fig. 10(a). The first phase traversal has an average of 71.87% achieved occupancy, whereas the second phase traversal attains an average of 69.67% achieved occupancy. The average achieved occupancy of the two-phase traversal is about 70%. The result shows that the achieved occupancy of the first phase slightly transcends that of the second phase. It is due to that the first phase separates the whole traversal into levels, balancing workloads among threads. Considering the fact that the DAG is unbalanced, we believe that the achieved occupancy of this implementation is sufficient to demonstrate the effectiveness and applicability of our traversal strategy.

Bandwidth Utilization: We show the bandwidth utilization in Fig. 10(b). As shown in Fig. 10(b), the bandwidth utilization between random access operations and datasets is varied. Operations involving updates to data structures, such as *append* and *insert*, exhibit high bandwidth utilization of 422.61 GB/s and 263.84 GB/s on average. Although *count* and *search* operations have comparatively lower average bandwidth utilization, they nevertheless provide significant performance improvements. In terms of datasets, our solution also exhibits bandwidth utilization variance among different datasets. Except for the operations of *append* and *insert* in the batch level that concern little about the file size, the bandwidth utilization is lower in datasets with smaller file sizes, and higher in datasets with larger files.

SM Efficiency: SM efficiency is the percentage of time a streaming multiprocessor (SM) has at least one warp working. We measure the metric of the five random access operations, and show the results in Fig. 11(a). In general, the five operations achieve an average of 80.8% efficiency, with the efficiency of operations *extract*, *append*, and *insert* exceeding or close to 90%.

Cycle Instruction: Cycle instruction represents the average number of instructions executed in each cycle. The cycle instruction of each random access operation is displayed in Fig. 11(b). Cycle instruction of *insert* is higher than all the other operations, because it frequently accesses data structures including *rule2location* and *rootOffset*. In contrast, *append* has a rather small number of instructions per cycle, because it mainly involves simple data copy operations.

Warp Instruction: Warp instruction represents the average number of instructions each warp executes. As shown in Fig. 11(c), warp instruction of random access operations illustrates considerable variation between different operations. In detail, the warp instruction of *append* operation is only 54.9 because of its lightweight processing workload, while other operations, such as *search*, *extract*, and *insert*, have relatively high values for the complexity of the procedure.

Adaptation to Edge GPU: We analyze the zero-copy edge optimizations discussed in Section V-B. In detail, we evaluate our solution between the usage of *cudaMalloc()* and *cudaMallocManaged()*. By using *cudaMallocManaged()* on Jetson AGX Xavier, we avoid the data transfer through PCIe and two duplicate data allocations in memory. We evaluate the five operations on different datasets. Due to the intensive data transmission between different nodes of the DAG and computation during the DAG traversal in processing, the data transfer time does not dominate the end-to-end performance. However, we do have

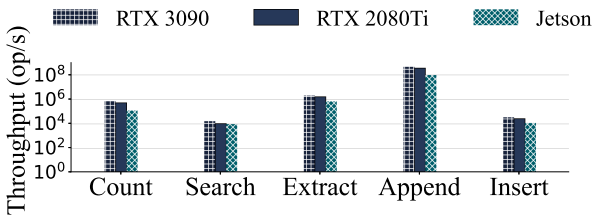


Fig. 12. Performance comparison.

great improvement in space savings because edge architecture provides a unified memory that can share the whole memory by using *cudaMallocManaged()*. In our design, with *cudaMallocManaged()*, we put *rootOffset*, *rule2location*, *word2rule*, and *records* into the global unified memory, and put small data structures like *bitmap* into both shared and unified memory, thus achieving a 43% memory saving compared to using *cudaMalloc()* alone.

G. Comparison With Different GPU Platforms

In this part, we compare the server-grade GPU platform and the edge-grade GPU platform from diverse perspectives.

Average Performance: We exhibit the performance comparison between different platforms in Fig. 12. Fig. 12 shows that the server-grade GPU platforms perform better than the edge-grade GPU platform. The reason is that the server-grade platforms have higher hardware configurations. For example, the RTX 3090 GPU has 10,482 cores with 1,395 MHz, while the Jetson AGX Xavier has only 512 cores with 854 MHz. Moreover, the server-grade GPU adopts discrete memory, which has much higher memory bandwidth than the edge-grade GPU. As for the detailed operations, their performance trends are also similar. For example, the *append* operation can achieve the highest performance on both the server-grade platform and the edge-grade platform. The reason is that it can perform in full parallelism.

Power Efficiency: We next study the power efficiency of our solution on these platforms. Energy consumption is a common concern in edge application scenarios, and we use *performance per power* ratio, shorten as *ppw_ratio* to represent the power efficiency, denoted in (9).

$$ppw\_ratio = \frac{average\ performance}{performance\ power\ of\ whole\ process} \quad (9)$$

In detail, the *average performance* in (9) stands for the average performance of throughput for all datasets with the five random access operations. Meanwhile, to profile the power consumption, we use related analysis tools on different platforms. On the Jetson platform, we use the Nvidia official tool, *tegrastats*, to report power usage. On server GPU platforms like RTX 3090 and RTX 2080Ti, we measure the GPU power consumption using *nvidia-smi* while measuring the CPU power consumption using *s-tui*. In our measurement, the edge-grade GPU platform achieves the highest power efficiency, which is 2.07× and 2.28× over those of the RTX 3090 and RTX 2080 Ti GPU platforms, respectively. This implies that the high performance of server-grade platforms is usually accompanied by high energy consumption, so they are not suitable for low-power application scenarios.

Cost Effectiveness: Besides power efficiency, cost effectiveness is another important factor we consider in HPC systems. We use *performance per price* ratio, shorten as ppc_ratio, to represent the cost effectiveness, denoted as (10).

$$ppc\_ratio = \frac{average\ performance}{price\ of\ the\ platform} \quad (10)$$

The price information is found on the Amazon official website [81], [82] and other authoritative websites [83]. The *average performance* is the same as in power efficiency. In our experiments, the server-grade GPU platforms achieve better cost-effectiveness than the edge-grade platform. The result shows that RTX 3090 achieves 2.3× cost effectiveness, while RTX 2080Ti achieves 2.41× cost effectiveness over Jetson AGX Xavier. That is, the server-grade GPU platforms reach 2.36× cost effectiveness over the edge GPU platform on average.

H. Comparison With the State-of-the-Art

We further evaluate our work by making comparisons with the state-of-the-art works, including G-TADOC and the current solution to perform mixed random access operations.

Comparison With G-TADOC: The original G-TADOC [8] mainly targets data analytics tasks that require a full range scan without considering the locality. To process random access operations, G-TADOC has to scan the entire data with a BFS-based traversal method, which is time-consuming. In contrast, our solution sets up index data structures, such as *word2rule* and *rule2location*, for efficient random access operations, which can help minimize data accesses. To evaluate the locality of our solution, we compare the size of data accessed by a random access operation with the original size of the compressed data. We conclude that one operation accesses data only about an average of 3.43‰ the size of the compressed data. The specific data access ratios range from 0.0002‰ to 17.02‰. In detail, *search* accesses the most amount of data, while the data amount accessed by *append* is at the lowest level. To demonstrate the efficiency of our solution, we compare our solution with G-TADOC, in which we perform *count* for 20,000 random words on datasets A, B, C, D, and E, and our solution achieves 5325.9×, 296310.1×, 2349.2×, 22205.2×, and 50859.2× performance speedups, respectively.

Comparison With the Mixed-Operation Solution: The studies [52], [84] show that for webpages, searching or counting specific words, and extracting certain content are common operations in real-world Web search. Moreover, as new web pages are continuously produced every day, the latest pages could need to be appended to the existing stored datasets. The research [30] also discusses this situation. In such cases, we use the real-world Wikipedia dataset and conduct mixed random access operations of 20,000 *extract*, 20,000 *search*, 20,000 *count*, 20,000 *insert*, and 20,000 *append* for evaluation. Compared to the operations without GPU [30], our solution achieves 22.39× speedup in terms of throughput.

I. Applicability

As discussed in Section III-A, our solution targets high-throughput text random accesses, and can provide extremely high throughput for random accesses to compressed data on GPU. Analytics systems that deal with static data, such as news [49], legal affairs [50], [51], webpages [52], [53], medical records [54], [55], and logs [56], all have such needs, and our solution can use only a GPU machine to meet the task volume that needs to be done by several servers in the past. Our solution handles the same input as previous works [7], [8], [28], [29], [30], so other tasks analyzing hierarchically compressed data can also be performed under our framework.

In terms of real-world applications, our work helps to handle heavy-load occasions such as the online analytical processing system that supports text analytics operations and insertion, and users are able to upload their text files to operate on. When queries from users come in a low-load manner, the system can handle the queries with only the CPU as discussed in the previous work [30] to achieve less turnaround time. On occasions that large amounts of queries arrive within a short time, GPU can be put to work to attain an overall high performance. This type of CPU-GPU collaboration design has been widely applied in OLAP systems. For example, the study [85] introduces a scheduling strategy that helps to balance the task load between CPU and GPU to achieve the optimal task completion time in the OLAP system. To align with the requirements of the text analytics system, our solution first executes the compression to transform text data from users into TADOC form and stores it in the CPU memory. It then loads the compressed data to GPU memory only when needed. Specifically, for datasets that are larger than the GPU memory capacity, we would keep them in the CPU memory and perform operations by loading the text data to GPU memory for processing part by part.

VII. Related Work

Data Analytics on Compression: Plenty of works have been proposed for text analytics directly on compressed data [7], [8], [28], [29], [30], [32], [86]. The closest work to ours is the study [30], which enables efficient random access to TADOC-compressed data on CPU. However, because this work lacks fine-grained large-scale thread parallelism, it cannot be executed on GPUs. TADOC [29] is a novel design for text analytics that runs directly on compression in both single-node and distributed environments. Following that, Zhang et al. [7] applied TADOC as the storage structure to support advanced document analytics. Succint [87] is a novel data store enabling efficient queries directly on the compressed data, and has been applied to other scenarios [88], [89]. Recently, Zhang et al. [8] developed G-TADOC, which enables TADOC on GPUs. However, it does not support random access operations. Our work compensates for the lack of supporting random access on GPU.

Data Processing on GPU: GPUs have been widely used in data science applications. For example, Hu et al. [90] accelerated triangle counting on GPU. Paul et al. [91] explored scalable join on multi-GPU systems. Li et al. [92] developed dynamic Hash tables on GPUs. Floratos et al. [93] studied nested query processing on GPU. Peng et al. [94] developed GPU-based sequence indexing. Rui et al. [95] proposed large table join with multi-GPUs. Li et al. [96] built a CPU-GPU hybrid database product. MapD [36] is a novel GPU-powered database engine, which can provide efficient query processing utilizing massive GPU cores. Gunrock [97] is a novel GPU-based graph analytics framework. For huge graphs, multi-GPU can be applied [98]. SABER [39] is a hybrid CPU-GPU stream processing engine that can handle relational stream queries. Furthermore, FineStream [38], [99] is developed to enable efficient window-based stream processing on CPU-GPU integrated architectures.

Edge-Grade GPU Computing: Edge-grade GPUs are becoming increasingly popular in recent days, due to their attractive features such as power efficiency [100], [101]. Edge GPUs have been applied in diverse applications. For example, Jose et al. [14] used Nvidia Jetson TX2 edge GPU to develop a surveillance system. Amert et al. [102] studied the Nvidia TX2 edge GPU in autonomous-driving systems. Rungsuptaweekoon et al. [103] demonstrated the power efficiency of edge GPU in inference. Lee et al. [104] used the edge GPU for car plate recognition. Davidson et al. [105] applied embedded GPU to process images for space applications. Different from these works, we study compressed data analytics on edge-grade GPU platforms.

VIII. Conclusion

This paper presents our optimization of enabling GPU-based random access to hierarchically-compressed data, which is the first to develop efficient random access operations on the GPU without decompression. We unveil the challenges and difficulties of enabling random access on GPU, and develop a set of novel designs to address them, including data structure architecture, data structure parallel generation, and random access operations. Our solution attains an average speedup of 52.98× in operation acceleration, and 56.35% time saving in data structure generation. Moreover, our solution achieves an average compression ratio of 2.92 in space.

References

[1] T. Allen et al., "In-depth analyses of unified virtual memory system for GPU accelerated computing," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, St. Louis, MO, USA, Nov. 14–19, 2021, pp. 64:1–64:15.

[2] J. Kosaian et al., "Arithmetic-intensity-guided fault tolerance for neural network inference on GPUs," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, St. Louis, MO, USA, Nov. 14–19, 2021, pp. 79:1–79:15.

[3] I. Sakiotis et al., "PAGANI: A parallel adaptive GPU algorithm for numerical integration," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, St. Louis, MO, USA, Nov. 14–19, 2021, pp. 83:1–83:13.

[4] F. Knorr et al., "ndzip-gpu: Efficient lossless compression of scientific floating-point data on GPUs," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, St. Louis, MO, USA, Nov. 14–19, 2021, pp. 93:1–93:14.

[5] K. Ranganath et al., "MAPA: Multi-accelerator pattern allocation policy for multi-tenant GPU servers," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, St. Louis, MO, USA, Nov. 14–19, 2021, pp. 99:1–99:14.

[6] Z. Bian et al., "Online evolutionary batch size orchestration for scheduling deep learning workloads in GPU clusters," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, St. Louis, MO, USA, Nov. 14–19, 2021, pp. 100:1–100:15.

[7] F. Zhang et al., "TADOC: Text analytics directly on compression," *VLDB J.*, vol. 30, no. 2, pp. 163–188, 2021.

[8] F. Zhang et al., "G-TADOC: Enabling efficient GPU-based text analytics without decompression," in *Proc. IEEE Int. Conf. Data Eng.*, 2021, pp. 1679–1690.

[9] Z. Pan et al., "Exploring data analytics without decompression on embedded GPU systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 7, pp. 1553–1568, Jul. 2022.

[10] Y. Kortli et al., "Deep embedded hybrid CNN–LSTM network for lane detection on NVIDIA Jetson Xavier NX," *Knowl.-Based Syst.*, vol. 240, 2022, Art. no. 107941.

[11] H.-H. Nguyen, D. N.-N. Tran, and J. W. Jeon, "Towards real-time vehicle detection on edge devices with NVIDIA Jetson TX2," in *Proc. IEEE Int. Conf. Consum. Electron.-Asia*, 2020, pp. 1–4.

[12] Z. Qiang, W. Yuanyu, Z. Liang, Z. Jin, L. Yu, and L. Dandan, "Research on real-time reasoning based on Jetson TX2 heterogeneous acceleration YOLOv4," in *Proc. IEEE 6th Int. Conf. Cloud Comput. Big Data Analytics*, 2021, pp. 455–459.

[13] A. Wong, M. Famuori, M. J. Shafiee, F. Li, B. Chwyl, and J. Chung, "YOLO Nano: A highly compact you only look once convolutional neural network for object detection," in *Proc. 5th Workshop Energy Efficient Mach. Learn. Cogn. Comput.-NeurIPS Ed.*, 2019, pp. 22–25.

[14] E. Jose, G. M., M. T. P. Haridas, and M. H. Supriya, "Face recognition based surveillance system using FaceNet and MTCNN on Jetson TX2," in *Proc. 5th Int. Conf. Adv. Comput. Commun. Syst.*, 2019, pp. 608–613.

[15] V. Sati et al., "Face detection and recognition, face emotion recognition through NVIDIA Jetson Nano," in *Proc. 11th Int. Symp. Ambient Intell. Ambient Intell.–Softw. Appl.*, Springer, 2021, pp. 177–185.

[16] W. Vijitkunsawat and P. Chantngarm, "Comparison of machine learning algorithm's on self-driving car navigation using NVIDIA Jetson Nano," in *Proc. 17th Int. Conf. Elect. Eng./Electron. Comput. Telecommun. Inf. Technol.*, 2020, pp. 201–204.

[17] P. Grzesik et al., "Metagenomic analysis at the edge with Jetson Xavier NX," in *Proc. 21st Int. Conf. Comput. Sci.*, Springer, Krakow, Poland, Jun. 16–18, 2021, pp. 500–511.

[18] Z.-D. Zhang et al., "CDNet: A real-time and robust crosswalk detection network on Jetson nano based on YOLOv5," *Neural Comput. Appl.*, vol. 34, no. 13, pp. 10 719–10 730, 2022.

[19] M. I. Uddin, M. S. Alamgir, M. M. Rahman, M. S. Bhuiyan, and M. A. Moral, "AI traffic control system based on deepstream and IoT using NVIDIA Jetson nano," in *Proc. 2nd Int. Conf. Robot. Elect. Signal Process. Techn.*, 2021, pp. 115–119.

[20] Y. Wu, "Cloud-edge orchestration for the Internet of Things: Architecture and AI-powered data processing," *IEEE Internet of Things J.*, vol. 8, no. 16, pp. 12 792–12 805, Aug. 2021.

[21] G. Li et al., "Data processing delay optimization in mobile edge computing," *Wireless Commun. Mobile Comput.*, vol. 2018, pp. 1–9, 2018.

[22] J. Koo et al., "Fine-grained data processing framework for heterogeneous IoT devices in sub-aquatic edge computing environment," *Wireless Pers. Commun.*, vol. 116, pp. 1407–1422, 2021.

[23] J. C. Kieffer, "A tutorial on hierarchical lossless data compression," in *Modeling Uncertainty*. Berlin, Germany: Springer, 2002, pp. 711–733.

[24] C. G. Nevill-Manning, "Inferring sequential structure," PhD dissertation, Univ. Waikato, Hamilton, New Zealand, 1996.

[25] C. G. Nevill-Manning and I. H. Witten, "Compression and explanation using hierarchical grammars," *Comput. J.*, vol. 40, no. 2/3, pp. 103–116, Jan. 1997.

[26] C. G. Nevill-Manning et al., "Identifying hierarchical structure in sequences: A linear-time algorithm," *J. Artif. Intell. Res.*, vol. 7, pp. 67–82, 1997.

[27] C. G. Nevill-Manning and I. H. Witten, "Linear-time, incremental hierarchy inference for compression," in *Proc. Data Compression Conf.*, 1997, pp. 3–11.

[28] F. Zhang et al., "Zwift: A programming framework for high performance text analytics on compressed data," in *Proc. Int. Conf. Supercomputing*, 2018, pp. 195–206.

[29] F. Zhang et al., "Efficient document analytics on compressed data: Method, challenges, algorithms, insights," in *Proc. VLDB Endowment*, vol. 11, no. 11, pp. 1522–1535, 2018.

[30] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, "Enabling efficient random access to hierarchically-compressed data," in *Proc. IEEE Int. Conf. Data Eng.*, 2020, pp. 1069–1080.

[31] F. Zhang et al., "Optimizing random access to hierarchically-compressed data on GPU," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2022, Art. no. 18.

[32] F. Zhang, J. Zhai, X. Shen, O. Mutlu, and X. Du, "POCLib: A high-performance framework for enabling near orthogonal processing on compression," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 2, pp. 459–475, Feb. 2022.

[33] F. Zhang et al., "CompressDB: Enabling efficient compressed data direct processing for various databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2022, pp. 1655–1669.

[34] F. Zhang, J. Zhai, B. He, S. Zhang, and W. Chen, "Understanding co-running behaviors on integrated CPU/GPU architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 3, pp. 905–918, Mar. 2017.

[35] F. Zhang, Z. Chen, C. Zhang, A. C. Zhou, J. Zhai, and X. Du, "An efficient parallel secure machine learning framework on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 9, pp. 2262–2276, Sep. 2021.

[36] C. Root et al., "MapD: A GPU-powered big data analytics and visualization platform," in *Proc. ACM SIGGRAPH Talks*, 2016, pp. 1–2.

[37] Y. Yuan, M. F. Salmi, Y. Huai, K. Wang, R. Lee, and X. Zhang, "Spark-GPU: An accelerated in-memory data processing engine on clusters," in *Proc. IEEE Int. Conf. Big Data*, 2016, pp. 273–283.

[38] F. Zhang et al., "FineStream: Fine-grained window-based stream processing on CPU-GPU integrated architectures," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2020, pp. 633–647.

[39] A. Koliousis et al., "SABER: Window-based hybrid stream processing for heterogeneous architectures," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 555–569.

[40] A. Shanbhag et al., "Tile-based lightweight integer compression in GPU," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2022, pp. 1390–1403.

[41] C. Chai et al., "Selective data acquisition in the wild for model charging," in *Proc. VLDB Endowment*, vol. 15, no. 7, pp. 1466–1478, 2022.

[42] C. Chai, J. Wang, Y. Luo, Z. Niu, and G. Li, "Data management for machine learning: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 5, pp. 4646–4667, May 2023.

[43] I. Buck, "GPU computing with NVIDIA CUDA," in *Proc. ACM SIGGRAPH Courses*, 2007, pp. 6-es.

[44] J. Sanders et al., *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Reading, MA, USA: Addison-Wesley Professional, 2010.

[45] C. Arcila-Calderon et al., "Big data techniques: Large-scale text analysis for scientific and journalistic research," *Profesional de la Informacion*, vol. 25, pp. 623–631, 2016.

[46] M. Chowkwanyun, "Big data, large-scale text analysis, and public health research," *Amer. J. Public Health*, vol. 109, pp. S126–S127, 2019.

[47] Q. Yin et al., "An adaptive elastic multi-model big data analysis and information extraction system," *Data Sci. Eng.*, vol. 7, no. 4, pp. 328–338, 2022.

[48] K. Atasu et al., "Hardware-accelerated regular expression matching for high-throughput text analytics," in *Proc. Int. Conf. Field Programmable Log. Appl.*, 2013, pp. 1–7.

[49] B. Zhao and S. Vogel, "Adaptive parallel sentences mining from web bilingual news collection," in *Proc. IEEE Int. Conf. Data Mining*, 2002, pp. 745–748.

[50] A. B. Bepko, "Public availability or practical obscurity: The debate over public access to court records on the internet," *New York Law Sch. Law Rev.*, vol. 49, 2004, Art. no. 967.

[51] P. A. Winn, "Online court records: Balancing judicial accountability and privacy in an age of electronic information," *Washington Law Rev.*, vol. 79, 2004, Art. no. 307.

[52] S. Bao et al., "Method and apparatus for enhancing webpage browsing," U.S. Patent 8 577 900, Nov. 05, 2013.

[53] S. Lawrence and C. L. Giles, "Context and page analysis for improved Web search," *IEEE Internet Comput.*, vol. 2, no. 4, pp. 38–46, Jul./Aug. 1998.

[54] W. Raghupathi et al., "Big data analytics in healthcare: Promise and potential," *Health Inf. Sci. Syst.*, vol. 2, 2014, Art. no. 3.

[55] R. H. Miller et al., "Physicians' use of electronic medical records: Barriers and solutions," *Health Affairs*, vol. 23, pp. 116–126, 2004.

[56] B. Zhang et al., "The cloud is not enough: Saving IoT from the cloud," in *Proc. 7th USENIX Conf. Hot Topics Cloud Comput.*, 2015, Art. no. 21.

[57] D. Merrill et al., "Scalable GPU graph traversal," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 117–128, 2012.

[58] P. Harish et al., "Accelerating large graph algorithms on the GPU using CUDA," in *Proc. Int. Conf. High-Perform. Comput.*, Springer, 2007, pp. 197–208.

[59] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in *Proc. Int. Conf. Parallel Architectures Compilation Techn.*, 2011, pp. 78–88.

[60] Y. Jia et al., "Edge v. node parallelism for graph centrality metrics," in *GPU Computing Gems Jade Edition*. Amsterdam, The Netherlands: Elsevier, 2012, pp. 15–28.
[61] H. Liu et al., "Enterprise: Breadth-first graph traversal on GPUs," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, pp. 1–12.
[62] E. Z. Zhang et al., "On-the-fly elimination of dynamic irregularities for GPU computing," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 369–380, 2011.
[63] B. Wu, E. Z. Zhang, and X. Shen, "Enhancing data locality for dynamic simulations through asynchronous data transformations and adaptive control," in *Proc. Int. Conf. Parallel Architectures Compilation Techn.*, 2011, pp. 243–252.
[64] M. Garland et al., "Parallel computing experiences with CUDA," *IEEE Micro*, vol. 28, no. 4, pp. 13–27, Jul./Aug. 2008.
[65] H. Zhou et al., "Accelerating large scale real-time GNN inference using channel pruning," 2021, *arXiv:2105.04528*.
[66] Y. Zhang et al., "Distributed deep learning on data systems: A comparative analysis of approaches," in *Proc. VLDB Endowment*, vol. 14, no. 10, pp. 1769–1782, 2021.
[67] N. M. Kumar et al., "Distributed energy resources and the application of AI, IoT, and blockchain in smart grids," *Energies*, vol. 13, no. 21, 2020, Art. no. 5739.
[68] O. Debauche et al., "A new edge architecture for AI-IoT services deployment," *Procedia Comput. Sci.*, vol. 175, pp. 10–19, 2020.
[69] A. Koliousis et al., "Crossbow: Scaling deep learning with small batch sizes on multi-GPU servers," 2019, *arXiv: 1901.02244*.
[70] K. Shafique, B. A. Khawaja, F. Sabir, S. Qazi, and M. Mustaqim, "Internet of Things (IoT) for next-generation smart systems: A review of current challenges, future trends and prospects for emerging 5G-IoT scenarios," *IEEE Access*, vol. 8, pp. 23 022–23 040, 2020.
[71] Jetson-AGX-Xavier, 2022. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/
[72] Volta-architecture-whitepaper, 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
[73] NVIDIA Jetson AGX Xavier Delivers 32 TeraOps for New Era of AI in Robotics, 2018. [Online]. Available: https://developer.nvidia.com/blog/nvidia-jetson-agx-xavier-32-teraops-ai-robotics/
[74] GeForce-rtx-3090, 2020. [Online]. Available: https://www.techpowerup.com/gpu-specs/geforce-rtx-3090.c3622
[75] Architecture-Whitepaper-V1, 2017. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf
[76] Wikipedia HTML data dumps, 2017. [Online]. Available: https://dumps.wikimedia.org/enwiki/
[77] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml
[78] DBLP, 2020. [Online]. Available: https://dblp.uni-trier.de/xml/
[79] COVID-19 data from yelp open dataset, 2019. [Online]. Available: https://www.yelp.com/dataset
[80] Nsight compute command line interface, 2019. [Online]. Available: https://docs.nvidia.com/nsight-compute/2019.1/pdf/NsightComputeCli.pdf
[81] NVIDIA RTX 3090 Price, 2023. [Online]. Available: https://www.amazon.com/ZOTAC-Graphics-IceStorm-Advanced-ZT-A30900J-10P/dp/B08ZL6XD9H/
[82] NVIDIA RTX 2080Ti Price, 2023. [Online]. Available: https://www.amazon.com/MSI-GAMING-RTX-2080-TRIO/dp/B07HWW7NCW/?th=1
[83] NVIDIA Jetson AGX Xavier Price, 2023. [Online]. Available: https://elinux.org/Jetson_AGX_Xavier
[84] S. Lawrence and C. L. Giles, "Context and page analysis for improved web search," *IEEE Internet Comput.*, vol. 2, no. 4, pp. 38–46, Jul./Aug. 1998.
[85] L. Riha et al., "Task scheduling for GPU accelerated OLAP systems," in *Proc. Conf. Center Adv. Stud. Collaborative Res.*, 2011, pp. 107–119.
[86] P. Boncz et al., "FSST: Fast random access string compression," in *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 2649–2661, 2020.
[87] R. Agarwal et al., "Succinct: Enabling queries on compressed data," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2015, pp. 337–350.
[88] A. Khandelwal et al., "BlowFish: Dynamic storage-performance tradeoff in data stores," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2016, pp. 485–500.
[89] A. Khandelwal et al., "ZipG: A memory-efficient graph store for interactive queries," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2017, pp. 1149–1164.
[90] L. Hu et al., "Accelerating triangle counting on GPU," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2021, pp. 736–748.
[91] J. Paul et al., "MG-Join: A scalable join for massively parallel multi-GPU architectures," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2021, pp. 1413–1425.
[92] Y. Li, Q. Zhu, Z. Lyu, Z. Huang, and J. Sun, "DyCuckoo: Dynamic hash tables on GPUs," in *Proc. IEEE Int. Conf. Data Eng.*, 2021, pp. 744–755.
[93] S. Floratos et al., "NestGPU: Nested query processing on GPU," in *Proc. IEEE Int. Conf. Data Eng.*, 2021, pp. 1008–1019.
[94] B. Peng, P. Fatourou, and T. Palpanas, "SING: Sequence indexing using GPUs," in *Proc. IEEE Int. Conf. Data Eng.*, 2021, pp. 1883–1888.
[95] R. Rui et al., "Efficient join algorithms for large database tables in a multi-GPU environment," in *Proc. VLDB Endowment*, vol. 14, no. 4, pp. 708–720, 2020.
[96] R. Lee et al., "The art of balance: A RateupDB experience of building a CPU/GPU hybrid database product," in *Proc. VLDB Endowment*, vol. 14, no. 12, pp. 2999–3013, 2021.
[97] Y. Wang et al., "Gunrock: GPU graph analytics," *ACM Trans. Parallel Comput.*, vol. 4, 2017, Art. no. 3.
[98] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens, "Multi-GPU graph analytics," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2017, pp. 479–490.
[99] F. Zhang et al., "Fine-grained multi-query stream processing on integrated architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 9, pp. 2303–2320, Sep. 2021.
[100] S. Mittal, "A survey on optimized implementation of deep learning models on the NVIDIA Jetson platform," *J. Syst. Architecture*, vol. 97, pp. 428–442, 2019.
[101] Y. Ukidave, D. Kaeli, U. Gupta, and K. Keville, "Performance of the NVIDIA Jetson TK1 in HPC," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2015, pp. 533–534.
[102] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU scheduling on the NVIDIA TX2: Hidden details revealed," in *Proc. IEEE Real-Time Syst. Symp.*, 2017, pp. 104–115.
[103] K. Rungsuptaweekoon, V. Visoottiviseth, and R. Takano, "Evaluating the power efficiency of deep learning inference on embedded GPU systems," in *Proc. Int. Conf. Inf. Technol.*, 2017, pp. 1–5.
[104] S. Lee, K. Son, H. Kim, and J. Park, "Car plate recognition based on CNN using embedded system with GPU," in *Proc. 10th Int. Conf. Hum. Syst. Interact.*, 2017, pp. 239–241.
[105] R. L. Davidson and C. P. Bridges, "Error resilient GPU accelerated image processing for space applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 9, pp. 1990–2003, Sep. 2018.

Yihua Hu is a research assistant with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China. She joined the Key Laboratory of Data Engineering and Knowledge Engineering (MOE) in 2020. Her major research interests include parallel and distributed systems.

Feng Zhang received the bachelor's degree from Xidian University, in 2012, and the PhD degree in computer science from Tsinghua University, in 2017. He is an associate professor with DEKE Lab and School of Information, Renmin University of China. His major research interests include database systems, and parallel and distributed systems.

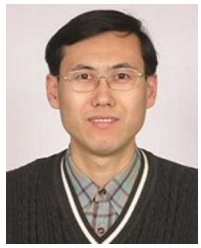
Yifei Xia is a research assistant with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China. He joined the Key Laboratory of Data Engineering and Knowledge Engineering (MOE) in 2020. His major research interests include parallel and distributed systems.

Zhiming Yao is a research assistant with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China. He joined the Key Laboratory of Data Engineering and Knowledge Engineering (MOE) in 2019. His major research interests include parallel and distributed systems.

Letian Zeng is a research assistant with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China. He joined the Key Laboratory of Data Engineering and Knowledge Engineering (MOE) in 2019. His major research interests include distributed systems.

Haipeng Ding received the bachelor's degree from the Renmin University of China, in 2022. He is a research assistant with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), Renmin University of China. He joined the Key Laboratory of Data Engineering and Knowledge Engineering (MOE) in 2020. His major research interests include parallel and distributed systems and machine learning.

Zhewei Wei received the PhD degree in computer science and engineering from the Hong Kong University of Science and Technology. He is currently a professor with the Renmin University of China. His research interests include algorithms for massive data, streaming algorithms, and graph algorithms.



Xiao Zhang received the master's degree in computer science and technology from Renmin University, in 1998, and the PhD degree in computer science and technology from the Institute of Computing Technology, Chinese Academy of Science, in 2001. He is a professor with the School of Information, Renmin University of China. His research interests include database architecture and Big Data management systems.

Jidong Zhai received the BS degree in computer science from the University of Electronic Science and Technology of China, in 2003, and the PhD degree in computer science from Tsinghua University, in 2010. He is an associate professor with the Department of Computer Science and Technology, Tsinghua University. His research interests include performance evaluation for high performance computers, performance analysis, and modeling of parallel applications.

Xiaoyong Du received the BS degree from Hangzhou University, Zhejiang, China, in 1983, the ME degree from the Renmin University of China, Beijing, China, in 1988, and the PhD degree from the Nagoya Institute of Technology, Nagoya, Japan, in 1997. He is currently a professor with the School of Information, Renmin University of China. His current research interests include databases and intelligent information retrieval.

Siqi Ma received the PhD degree from Singapore Management University. She is with the School of Engineering and Information Technology, University of New South Wales, Australia. Her research interests include mobile security, IoT security, and software engineering.