G-TADOC: Enabling Efficient GPU-Based Text Analytics without Decompression

Feng Zhang*, Zaifeng Pan‖, Yanliang Zhou*, Jidong Zhai†, Xipeng Shen‡, Onur Mutlu§, Xiaoyong Du*
*Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China
‖School of Mechanical Engineering, Shanghai Jiao Tong University
†Department of Computer Science and Technology, Tsinghua University, BNRist
‡Computer Science Department, North Carolina State University
§Department of Computer Science, ETH Zürich
fengzhang@ruc.edu.cn, 764556762@sjtu.edu.cn, triode-zyl@ruc.edu.cn, zhaijidong@tsinghua.edu.cn,
xshen5@ncsu.edu, onur.mutlu@inf.ethz, duyong@ruc.edu.cn

***Abstract*—Text analytics directly on compression (TADOC) has proven to be a promising technology for big data analytics. GPUs are extremely popular accelerators for data analytics systems. Unfortunately, no work so far shows how to utilize GPUs to accelerate TADOC. We describe G-TADOC, the first framework that provides GPU-based text analytics directly on compression, effectively enabling efficient text analytics on GPUs without decompressing the input data.**

G-TADOC solves three major challenges. First, TADOC involves a large amount of dependencies, which makes it difficult to exploit massive parallelism on a GPU. We develop a novel fine-grained thread-level workload scheduling strategy for GPU threads, which partitions heavily-dependent loads adaptively in a fine-grained manner. Second, in developing G-TADOC, thousands of GPU threads writing to the same result buffer leads to inconsistency while directly using locks and atomic operations lead to large synchronization overheads. We develop a memory pool with thread-safe data structures on GPUs to handle such difficulties. Third, maintaining the sequence information among words is essential for lossless compression. We design a sequence-support strategy, which maintains high GPU parallelism while ensuring sequence information.

Our experimental evaluations show that G-TADOC provides 31.1× average speedup compared to state-of-the-art TADOC.

***Index Terms*—TADOC, GPU, parallelism, analytics on compressed data**

I. Introduction

Text analytics directly on compression (TADOC) [1]–[4] has proven to be a promising technology for big data analytics. Since TADOC processes compressed data without decompression, a large amount of space can be saved. Meanwhile, TADOC reuses both data and intermediate computation results, which results in that the same contents in different parts of original files can be processed only once, thus saving significant computation time. Recent studies show that TADOC can save up to half of the processing time and 90.8% storage space [1], [2]. On the other hand, GPU as a heterogeneous processor shows promising performance in many real applications, such as artificial intelligence. It is popular to use heterogeneous processors, such as GPU, to accelerate data analytics systems [5]–[9]. Therefore, it is essential to enable efficient data analytics on GPUs without decompression.

Applying GPUs to accelerate TADOC brings three key benefits. First, GPU performance is much higher than CPU performance, so applying GPUs with proper designs can greatly accelerate TADOC performance, which means that users can feel no delay in data analytics towards massive data. Second, previous TADOC mainly focuses on distributed systems. If we develop a GPU-based solution on a single HPC server while achieving higher performance, tremendous resources, including equipment cost and electricity cost, can be significantly saved. Third, many data analytics applications, such as *latent Dirichlet allocation* (LDA) [10] and *term frequency-inverse document frequency* (TFIDF) [11], have been ported to GPUs, while TADOC has proven to be suitable for these advanced data analytics applications. Hence, providing a GPU solution would remove the last barrier to apply TADOC to a wide range of applications.

Although it is both beneficial and essential to develop TADOC on GPUs, building efficient GPU-based TADOC is very challenging. Applying GPUs to accelerate TADOC faces three challenges. First, TADOC organizes data into rules, which can further be represented as a DAG. Unfortunately, the amount of dependencies among the rule-structured DAG of TADOC is extremely large, which is unfriendly for GPU parallelism. For example, in our experiments, the generated DAG for each file has 450,704 dependent middle-layer nodes on average, which greatly limits its parallelism. Even worse, a node in the DAG of TADOC can have multiple parents, which makes this problem more complicated. Second, a large number of GPU threads writing to the same result buffer inevitably cause tremendous write conflicts. A straightforward solution is to lock the buffer for threads, but such atomicities lose partial performance. In the worst case, the parallel performance is lower than that of the CPU sequential TADOC. Third, maintaining and utilizing the sequence information on GPUs is another difficulty: the original TADOC adopts a recursive call to complete sequential traversal on compressed data, which is similar to a depth-first search (DFS) and is extremely hard to

solve in parallel.

Currently, none of the TADOC solutions can solve the challenges of enabling TADOC on GPUs mentioned above. Zhang *et al*. [2] first proposed TADOC solution but it is designed in a sequential manner. Although TADOC can be applied in a distributed environment, TADOC adopts coarse-grained parallelism and the processing for each compressed unit is still sequential. Zhang *et al*. next developed a domain specific language (DSL), called Zwift, to present TADOC [1], and further realized random accesses on compressed data [3]. However, the parallelism problems still exist. Zhang *et al*. [4] then provided a parallel TADOC design, which provides much better performance than the sequential TADOC. Unfortunately, such parallelism is still coarse-grained: it only divides the original file into several sub-files, processes different files separately, and then follows a merge process, which cannot be utilized by GPUs efficiently.

We design G-TADOC, the first framework that provides **G**PU-based **t**ext **a**nalytics **d**irectly **o**n **c**ompression, effectively enabling efficient text analytics on GPUs without decompressing input data. G-TADOC involves three novel features that can address the above three challenges. First, to utilize the GPU parallelism, we develop a fine-grained thread-level workload scheduling strategy on GPUs, which allocates thread resources according to the load of different rules adaptively and uses masks to describe the relations between rules (Section IV-B). Second, to solve the challenge of write conflict from multiple threads, we enable G-TADOC to maintain its own memory pool and design thread-safe data structures. We use a lock buffer when multiple threads update the global results simultaneously (Section IV-C). Third, to support sequence sensitive applications in G-TADOC, we develop *head* and *tail* data structures in each rule to store the contents at the beginning and end of the rule, which requires a light-weight DAG traversal (detailed in Section IV-D).

We evaluate G-TADOC on three GPU platforms, which involve three generations of Nvidia GPUs (Pascal, Volta, and Turing micro-architectures), and use five real-world datasets of varying lengths, structures, and content. Compared to TADOC on CPUs, G-TADOC achieves 31.1× speedup. In detail, TADOC can be divided into two phases: initialization and DAG traversal. For the initialization phase, G-TADOC achieves 76.5% time saving, while for the DAG traversal phase, G-TADOC achieves 82.2% time saving.

As far as we know, this is the first work enabling efficient text analytics on GPU without decompression. In summary, we have made the following contributions in this work.

- We present G-TADOC, which is the first framework enabling efficient GPU-based text analytics directly on compressed data.
- We unveil the challenges for developing TADOC on GPUs and provide a set of solutions to these challenges.
- We evaluate G-TADOC on three GPU platforms, and demonstrate its significant benefits compared to state-of-the-art TADOC.

II. Background

In this section, we introduce TADOC and GPUs, which are the background and premises of our work.

A. TADOC

TADOC [1]–[4] is a novel lossless compression technique that enables data analytics directly on compressed data without decompression. In detail, TADOC adopts dictionary conversion to encode original input data with numbers, and then uses context-free grammar (CFG) to recursively represent the numerical transformed data after conversion into rules. Repeated pieces of data are transformed into different rules in CFG, and the data analytics tasks are then represented as rule interpretations. To leverage redundant information between files, TADOC inserts unique splitting symbols for file boundaries. Moreover, the CFG can be represented as a directed acyclic graph (DAG), so the interpretation of the rules for data analytics can be regarded as a DAG traversal problem. Currently, TADOC extends Sequitur [12]–[14] as its core algorithm.

We use Figure 1 to show how TADOC compresses data by CFG representation, which is an example used in [2]. Figure 1 (a) shows the original input data, which consists of two files: file *A* and file *B*, and "wi" represents a unique word. Figure 1 (b) shows the dictionary conversion, which uses an integer to represent an element. Note that the rules "Ri" and file splitters "spti" are also transformed into numerical forms. Figure 1 (c) shows the TADOC compressed data, which are sequences of numbers. The TADOC compressed data can be viewed as CFG shown in Figure 1 (d), which can be further organized as a DAG shown in Figure 1 (e) for traversals.

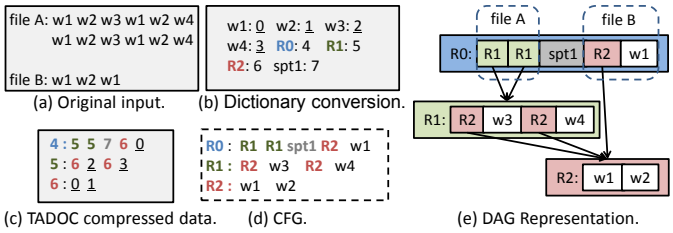


Fig. 1. A compression example with TADOC.

We use Figure 2 to show how to utilize the TADOC compressed data to perform a simple data analytics application – *word count*. In Step 1, *R2* transmits its accumulated local word frequencies to its parents, which are *R0* and *R1*. In Step 2, *R1* receives the word frequencies from *R2* and merges these frequencies to *R1*'s local frequency table. In Step 3, *R1* transmits its accumulated word frequencies to its parent *R0*. After *R0* receives the word frequencies from all its children, which are *R1* and *R2*, *R0* merges all received word frequencies into *R0*'s word count results, which are also the final word counts.

B. GPU

GPU is a specialized device targeting graphics and image processing originally. Due to its high parallelism

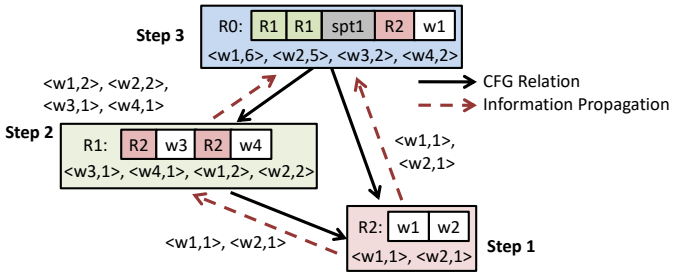


Fig. 2. Word count example using TADOC.

design, GPUs have now been applied to a wide range of applications, including data management systems [5]–[9]. A server equipped with GPUs can offer unprecedented computing power within a single machine. Previous TADOC mainly targets distributed environments without GPUs. If we provide a GPU solution for TADOC, enabling efficient GPU-based data analytics without decompression, not only the number of machines, but also the electricity budget can all be saved.

GPUs are different from CPUs in various aspects. First, different from CPUs, GPUs include a large number of light-weight cores, grouped into different streaming multi-processors (SM). GPUs utilize high throughput to hide memory access latency. Second, each SM has controllable shared memory, which has similar performance to caches. Therefore, good use of shared memory is critical for GPU performance. Third, the execution model on GPUs is also quite different from that on CPUs. The basic thread scheduling unit is a *warp*, which includes 32 threads for Nvidia GPUs. The threads within the same warp execute the same instructions in a lock-step manner, so called single instruction, multiple threads (SIMT). Developing GPU-based TADOC needs to adapt to the GPU execution model.

III. Motivation

A. Revisiting Previous Techniques

In this part, we revisit previous traversal-based techniques to show our motivation of a new GPU-based TADOC design.

Why GPU-based BFS does not apply? The DAG traversal in TADOC is unique and cannot be replaced by a BFS traversal. First, many data analytics tasks require sequence maintenance of words, where BFS cannot be used directly [2]. Second, TADOC involves complicated data processing and complex data structures during traversal. For example, each rule, which is a node in DAG, needs to maintain a local word table and a rule table, and all rules write to the same global buffer, which generates write conflicts in G-TADOC among GPU threads. Third, the DAG traversal in TADOC involves dynamic data transmission. For example, the traversal can transmit accumulated word frequencies among rules. Unfortunately, the amount of data transferred between nodes cannot be obtained in advance, which has not been involved in BFS on GPUs.

Why existing DAG traversal on GPUs does not apply? The uniqueness of TADOC is that each node requires complicated text-related intra- and inter-node operations. This uniqueness does not need to be considered in previous GPU traversal solutions. In detail, within a node, a dynamic buffer needs to be maintained to receive intermediate results from parents and to transmit data to children. Between different nodes, cross-rule sequence needs to be considered.

B. Challenges

In this section, we mainly discuss the challenges of enabling efficient text analytics on GPUs without decompression.

Challenge 1: GPU parallelism for TADOC. The high performance of GPU relies on the high throughput from thread-level parallelism. First, as presented in [2], there exist massive dependencies among the DAG, which leads TADOC difficult to be parallel. Accordingly, TADOC utilizes coarse-grained parallelism that mainly processes different compressed files in parallel: each CPU thread handles a separate file [4]. We cannot apply such coarse-grained parallelism on GPUs because a GPU supports thousands of threads and it is inefficient to split the compressed data into that large number of partitions. Second, if we use one GPU thread for one rule, there is a workload unbalancing problem because the numbers of elements in different rules vary significantly. GPUs launch threads in warp level, and the threads within a warp have to release resources simultaneously. The workload imbalance problem decreases the parallelism degrees. Third, we cannot simply decide the number of threads for rules, because of the various rule length.

Challenge 2: TADOC final result update conflict of massive GPU threads. The update conflict is a serious problem when we develop TADOC on GPUs. First, the update conflict of multiple threads writing to the same result buffer is not a serious problem on CPUs because the number of CPU threads is limited. However, on a GPU server, when a large number of GPU threads write to the same result buffer, we have to use atomic operations to guarantee correctness, which incurs massive conflicts. Second, the complicated data structures used in TADOC cannot be applied in GPU environment. For example, TADOC uses an *unordered map* data structure for results such as word counts; we need to develop our own similar data structures on GPUs with atomicity and consistency considered. Third, the amount of memory required by TADOC is unknown until runtime. Even worse, for TADOC on GPUs, the memory sizes of different threads are also various, which makes the update problem with thread conflicts more difficult.

Challenge 3: sequence maintenance of TADOC compressed data on GPUs. How to keep the sequence information on GPUs is also challenging. Sequence maintenance is essential for sequence sensitive applications, such as counting three continuous word sequences. First, to

keep the sequence information, TADOC originally traverses the DAG in a DFS order [2], which is hard to be parallel. Second, a word sequence can span several rules and these rules can be controlled by different GPU threads. Currently, threads across different GPU blocks have no mechanism for synchronization. Third, TADOC uses *map* data structures to store sequence counts. For these sequence-based applications, we need to develop special data structures in GPU memories to store sequences and perform basic comparisons between threads.

Based on the analysis, designing a GPU-based TADOC is very rewarding, but full of challenges.

IV. G-TADOC

In this section, we show our G-TADOC framework. G-TADOC consists of three components, a module for data structures, a parallel execution module, and a sequence support module. We first show the G-TADOC overview and then the different modules.

A. Overview

We show the overview of G-TADOC in Figure 3. The inputs are TADOC compressed data and user program, and the outputs are the results, which are similar to previous non-GPU TADOC implementations [1], [2].

Modules. G-TADOC consists of three major modules. The parallel execution module is responsible for the G-TADOC parallel execution on GPUs, which decides how to partition workloads for thread parallelism. The data structure module provides necessary data structures for G-TADOC execution, including a self-controlled memory pool, thread-safe data structures, and *head* and *tail* structures for sequences. The sequence support module is used for applications that are sensitive to sequence orders.

Phases. After receiving the TADOC compressed data and program, G-TADOC execution mainly consists of two phases: initialization phase and graph traversal phase. In the initialization phase, G-TADOC prepares necessary data structures according to the user program and launches a light-weight scanning to fulfill related values. In the graph traversal phase, G-TADOC analyzes different traversal strategies and chooses the most suitable one based on both data and tasks. Before the end of the graph traversal, G-TADOC performs a merging process for final results.

Solutions to challenges. G-TADOC can handle the challenges mentioned in Section III-B. To address the first GPU parallelism challenge, G-TADOC adopts a thread-level workload scheduling strategy for GPU threads, which partitions the DAG in a fine-grained manner for parallelism (Section IV-B). To address the second TADOC update conflict challenge, we develop a memory pool on GPUs and maintain necessary data structures so that all threads manage the same memory objects with consistency guaranteed (Section IV-C). To address the third challenge of sequence sensitivities on GPUs, G-TADOC scans the DAG for recording the cross-rule

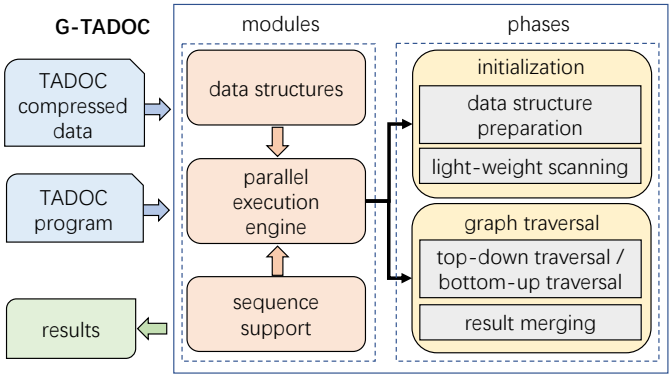


Fig. 3. G-TADOC overview.

content in a light-weight manner in the initialization phase. Then, G-TADOC performs a rule-level processing and result merging process in the graph traversal phase (Section IV-D).

B. Fine-Grained Thread-Level Execution Engine

We show our G-TADOC parallel execution engine in this part. In developing our parallel partitioning strategy, we consider two possible designs, as shown in Figure 4. The first design is to partition the DAG vertically from the root: different parts are traversed by different threads, as shown in Figure 4 (a). This design can leverage the GPU parallelism, but at the same time, some rules can be scanned by different threads. For example, *R2* and *R4* are scanned by both *thread0* and *thread1*. Even worse, when the DAG is very deep and complicated, the problem that massive rules are repeatedly scanned by different threads can be serious. Hence, we abandon this design. The second design is fine-grained thread-level scheduling: we assign a thread for each node except the root; the root rule usually includes a large number of elements so we allocate a group of threads based on the rule length to handle it. Note that when a rule includes a large number of elements (the default threshold is 16 times the average number of elements per thread), such as *R4*, more threads should be allocated for the rule. To traverse the DAG, each rule is associated with a *mask* to indicate whether a rule is ready to be traversed or not. This design ensures the dependency for correctness in the DAG traversal and retains great parallelism simultaneously. Therefore, we adopt this fine-grained design. Moreover, as discussed in [4], the optimal traversal strategy depends on both input data and analytics tasks, so we develop both top-down and bottom-up traversals and use the strategy selector in [4] for such decisions.

Next, we show our detailed top-down and bottom-up designs in G-TADOC.

Top-down traversal. We show our top-down traversal design in this part.

1) General design. The general design of top-down traversal transmits required data, such as file information, from the root to sub-nodes for processing. Then, G-TADOC gathers local results from different nodes as the final result. First, in root,

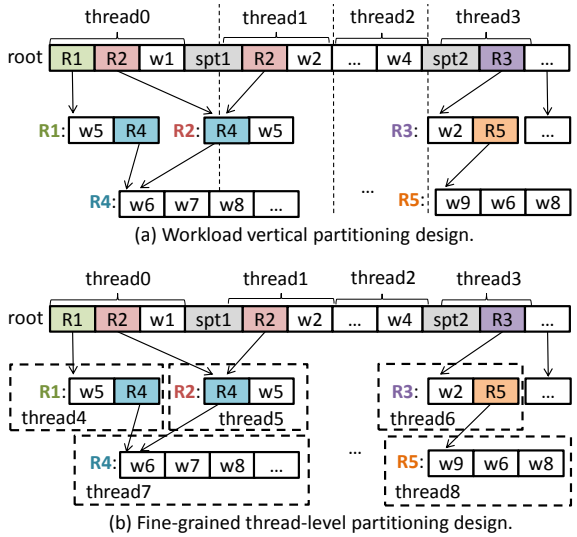


Fig. 4. Workload partitioning design exploration.

different consecutive parts are controlled by different threads, which can be processed in parallel. Second, during traversal, multiple parents can write to the same local buffer of a rule, which relates to data consistency. To handle this problem, a self-maintained memory pool is introduced, detailed in Section IV-C. Third, G-TADOC reduces intermediate results from rules to the final output in a global buffer in parallel.

2) Detailed algorithm. We show our top-down DAG traversal design in Algorithm 1. The general control is performed on the CPU side by the *topDown* function, as shown in Lines 1 to 8, which calls different GPU kernels, including *initTopDownMaskKernel*, *topDownKernel*, and *reduceResultKernel*.

The *initTopDownMaskKernel* is executed on GPUs, which initializes the nodes whose in-edges are from only the root. In detail, we consider the topology of the rules except the root, and accordingly, the initial weights of these rules are their frequencies in the root. Additionally, *rule.numInEdge* stores the number of in-edges of each rule, and only the rules with zero *numInEdge* can start the DAG traversal initially. In G-TADOC, we mark the masks of the rules that can be processed as *true*.

The *topDownKernel* is the main body of the top-down traversal and is executed on GPUs. We set the *devStopFlag* to *true* in Line 4. If the *devStopFlag* is still *true* after the *topDownKernel* execution, which means that the DAG has no update and has been fully traversed, then G-TADOC stops traversal. In *topDownKernel*, for different applications, only the for-loop from Lines 15 to 17 is different. Here, we take *word count* as an example. For a given rule, it first transmits its accumulated weights to all its subrules (Line 17). If the number of current in-edges *subRule.curInEdge* is equal to a subrule's full number of in-edges *subRule.numInEdge*, then we mark the subrule's mask to *true*, indicating that the subrule is ready to be traversed in the next round (Line 20). Note that when any masks are changed, *stopFlag* shall be set to *false*. Moreover, *rule.mask* should be set to *false* in Line 22 so that the rule will not be involved in the next round.

The *reduceResultKernel* merges the word frequencies from all rules multiplied by their corresponding accumulated rule weights on GPUs.

Algorithm 1 Top-Down Traversal

```
1: function topDown(rules)  ▷ Executed by host; use word count as an example
2:     initTopDownMaskKernel(rules) with at least rules.size threads
3:     do  ▷ Repeated top-down traverse until all rules' weight generated
4:         cudaMemSet devStopFlag ← true
5:         topDownKernel(rules, devStopFlag) with at least rules.size threads
6:         cudaMemCpy devStopFlag to stopFlag
7:     while stopFlag is false
8:     reduceResultKernel(rules) with at least rules.size threads  ▷ Reduce results from all rules

9: function topDownKernel(rules, devStopFlag)  ▷ GPU kernel
10:    if tid not in 1 to rules.size − 1 then
11:        return
12:    rule ← rules[tid]
13:    if rule.mask is false then
14:        return
15:    for each subRuleId, subRuleFreq in rule.subRules do
16:        subRule ← rules[subRuleId]
17:        atomicAdd(subRule.weight, subRuleFreq ∗ rule.weight)
18:        atomicAdd(subRule.curInEdge, 1)
19:        if subRule.curInEdge is full then
20:            subRule.mask ← true  ▷ Sub-rule then can be traversed
21:            devStopFlag ← false
22:    rule.mask ← false
```

3) Theoretical analysis. Algorithm 1 can be divided into three stages. The first stage is mask initialization (Line 2), in which each thread checks the corresponding rule's number of in-edges and then sets its mask. Assuming sufficient parallel resources, the complexity is $O(1)$. The second stage is top-down traversal (Lines 3 to 7). Assuming that the DAG has k layers, the number of loops is not greater than k. Then each thread in *topDownKernel* traverses the corresponding rule's sub-rules. Suppose in the i^{th} loop, the maximum number of sub-rules of a rule is $e_{i,max}$, then the total complexity of this stage is $O(\sum_{i=1}^{k} e_{i,max})$, which can be represented as $O(k\bar{e}_{max})$. The third stage is to reduce results (Line 8). Each thread needs to merge the corresponding rule's local words from the local table to the global table, so the complexity is $O(w_{max})$, where w_{max} is the maximum number of local words among all rules. Therefore, the overall complexity of Algorithm 1 is $O(k\bar{e}_{max} + w_{max})$.

Bottom-up traversal. We show our bottom-up traversal design in this part.

1) General design. The bottom-up traversal transmits required data, such as local word counts, from leaves to upper-level nodes. After transmission, the root and its directly connected nodes (called 2nd-layer nodes) store the gathered result. Note that we do not accumulate the results to the root because the root contains file information. In detail, first, each leaf transmits the required data from its local tables to

its parents. Second, during traversal, each node accumulates the transmitted data from children and then transmits the accumulated results to its parents. Note that data consistency needs to be guaranteed since different rules are controlled by different threads. Third, after traversal, G-TADOC analyzes the local buffers in the root and 2nd-layer nodes in parallel to generate the final results.

2) Detailed algorithm. We show our bottom-up DAG traversal design in Algorithm 2. The general control is performed by the function *bottomUp* from the CPU side, which calls different GPU kernels. Different from Algorithm 1, the bottom-up design in Algorithm 2 first generates the pointers from children to parents (Lines 2 to 3), initializes masks (Line 4), and generates the local tables' bound in a light-weight bottom-up manner (Lines 5 to 9), so that the local tables in rules can be allocated (Line 10). Then, it initializes masks again (Line 11) and traverses the graph in a comprehensive bottom-up direction with a result merging process (Lines 12 to 17).

The *initBottomUpMaskKernel* set the rule masks. The leaves are set to *true* so that they can be traversed initially.

The *genLocTblBoundKernel* is used to calculate the memory size limit for local tables, and is called by the *bottomUp* function repeatedly. Its kernel execution is similar to that of *topDownKernel* in Algorithm 1, except the use of out-edge rather than in-edge during traversal. When a rule is traversed, G-TADOC sums the upper limits of its local words and all its children's local tables as the amount of space that should be allocated. Then, the rule increases all its parents' out-edges. When a parent's number of current out-edges is equal to its number of subrules, G-TADOC sets its mask to *true* for the next-iteration execution. After calculating the memory limit of each node, we uniformly allocate the corresponding buffer for each rule in $rules.locTbl$ (Line 10).

The *genLocTblKernel* is used for DAG traversal with the allocated memory space from the *bottomUp* function. Its traversal order is controlled by the traversed out-edges, which is the same as *genLocTblBoundKernel*. However, the kernel's computation task is much heavier. Here, we use the *word count* example for illustration. When a rule is traversed, it first reduces its local word frequencies, and then merges all its subrules' local word frequencies into its own local table.

The *reduceResultKernel* merges the word frequencies from the root and its children where the root is directly connected (called *level-2 nodes* in [2]) on GPUs. In detail, G-TADOC merges 1) the word frequencies in the root, and 2) the frequencies in the local tables of the root's direct children multiplied by their corresponding rule frequencies in the root. This is different from the *reduceResultKernel* in Algorithm 1.

3) Complexity analysis. Different from Algorithm 1, Algorithm 2 consists of five stages. The first stage is to generate the parents of rules (Lines 2 to 3). Each thread in *genRuleParentsKernel* stores the corresponding rule's ID in all its sub-rules' parent table. The complexity is $O(e_{max})$, where e_{max} is the maximum number of sub-rules of all rules. The second stage is mask initialization (Line 4 and 11).

Algorithm 2 Bottom-Up Traversal

```
1: function bottomUp(rules)                                  ▷ word count
2:     allocate device memory to rules.parentIds
3:     genRuleParentsKernel(rules) with at least rules.size threads
4:     initBottomUpMaskKernel(rules) with at least rules.size threads
5:     do
6:         cudaMemSet devStopFlag ← true
7:         genLocTblBoundKernel(rules, devStopFlag) with at least rules.size threads
8:         cudaMemCpy devStopFlag to stopFlag
9:     while stopFlag is false
10:    allocate device memory to rules.locTbl
11:    initBottomUpMaskKernel(rules) with at least rules.size threads
12:    do
13:        cudaMemSet devStopFlag ← true
14:        genLocTblKernel(rules, devStopFlag) with at least rules.size threads
15:        cudaMemCpy devStopFlag to stopFlag
16:    while stopFlag is false
17:    reduceResultKernel(rules) with at least root.size threads
```

Similar with the mask initialization in Algorithm 1, the complexity is also $O(1)$. The third stage is to generate rules' local table bound (Lines 5 to 9). Each thread in *genLocTblBoundKernel* traverses the corresponding rule's sub-rules and parents. Suppose in the i^{th} loop, the maximum numbers of sub-rules and parents of these rules are $e_{i,max}$ and $p_{i,max}$ respectively. Then, the complexity is $O(\sum_{i=1}^{k}(e_{i,max}+p_{i,max})) = O(k(\bar{e}_{max}+\bar{p}_{max}))$, where k is the number of layers in the DAG. The fourth stage is to generate rules' local table (Lines 5 to 9). Besides traversing corresponding rule's sub-rules and parents, each thread in *genLocTblKernel* also merges all sub-rules' local tables and its own words. For a given rule i, suppose its local table size is t_i and its number of words is w_i, then its computation load is $w_i+\sum_{j\in i.subRules} t_j$. The complexity of this stage is $O(\sum_{i=1}^{k} C_{i,max})$, which is $O(k\bar{C}_{max})$. $C_{i,max}$ is the maximum computation load among rules in the i^{th} loop. The fifth stage is to reduce results (Line 8). This stage scans the root and merges all *level-2 nodes*. In detail, each thread is responsible for one *level-2 node*, so the complexity is $O(t_{lv2,max})$, where $t_{lv2,max}$ is the maximum size of *level-2 nodes'* local tables. Therefore, the overall complexity of Algorithm 2 is $O(k(\bar{e}_{max}+\bar{p}_{max}+\bar{C}_{max})+t_{lv2,max})$.

Parameter selection. G-TADOC involves a few parameters to adjust, such as the threshold of GPU thread resources allocated to a rule. The current solution is to extract a sample set of input and then use a greedy strategy to set each parameter in turns. If the input is unavailable until runtime, then the parameters are set according to our training set (a small extracted dataset from Wikipedia [15]).

C. G-TADOC Data Structures

The data structures in G-TADOC include a self-maintained memory pool, thread-safe structures, and sequence support.

G-TADOC maintained memory pool. As discussed in Section IV-B, we need to provide each thread a separate

memory space during DAG traversal. Because 1) the required memory size is unknown until runtime, and 2) allocating memory dynamically for all threads is inefficient, we develop a global memory pool to manage the GPU memory by G-TADOC itself. First, each rule calculates its own required memory size for necessary data structures. Second, with data transmission in the initialization phase in Figure 1, each rule transmits its memory requirement to its parents in a bottom-up traversal, or to its children in a top-down traversal. This memory requirement transmission process can be recursive. Third, after the whole range transmission in the initialization phase, each rule determines its maximum memory requirement and we can allocate related resources of different rules from the memory pool.

Thread-safe data structures. After we introduce the memory pool in G-TADOC, we next describe the thread-safe data structures used in the memory pool for GPU threads. The most important data structure in TADOC is the *hash* structure [2], which can be used to store the results both locally and globally. Hence, we use the hash structure for illustration in G-TADOC thread-safe design, as shown in Figure 5. The original state of the hash table is shown in Figure 5 (a). The lock buffer is for locking entries (1 means locked, and 0 means unlocked). The entry buffer is for hashing (default -1). The key and value buffers are for the key-value pairs. The *next* buffer is for the next entry if multiple key-value pairs are mapped into the same entry. Figure 5 (b) shows the state after inserting <126,1>, assuming the key-value pair is hashed to 1. Because there is no conflict in this insertion, the related value in the *next* buffer is -1. Figure 5 (c) shows the hash table state after inserting <163,1>, assuming the key-value pair is hashed to 3. Accordingly, G-TADOC just stores the key-value pair $<163, 1>$ after the first <126,1>. Figure 5 (d) shows a hash conflict situation: the hash table state after inserting <78,1>, assuming the key-value pair is hashed to 1. Because $<126, 1>$ has already been inserted to the first entry, we update its "next" buffer pointing to a new place for the newly inserted <78,1>. Note that the lock buffer is used only when all threads writing to the same buffer location. Moreover, if the hash table is private and owned by one thread, we do not need to create the locks.

Head and tail structures for sequence support. The head and tail structures are used to support sequence sensitive applications, such as *sequence count* [2]. Because G-TADOC traverses the DAG in parallel, some rules may involve cross-rule sequence (a word sequence spanning multiple nodes in the DAG). We design *head* and *tail* data structures for each rule to store the content of the beginning and end of the rule, which are provided to the parents. We show an example in Figure 6. In the root, the first sequence, *<w1,w2,w3>*, is a sequence that does not span across rules. However, for the next three-word sequence, *<w2,w3,w4>*, it spans across the root and *R1*. For this sequence, we store the partial content of *<w4,w5>* in the head buffer of *R1*, so that this cross-rule sequence can be processed by the parent, which is the root.

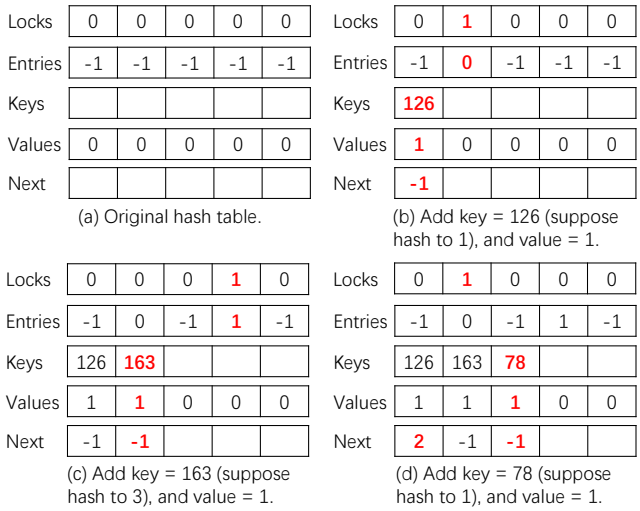


Fig. 5. Illustration for thread-safe hash tables.

Similarly, we store *<w6,w7>* in the tail buffer of *R1* so that *R1*'s parent can quickly process the sequences containing the words in *R1*'s tail buffer. Note that the first few elements and the last few elements in the subrule can also be a rule. For example, in Figure 4, the first element of *R2* is also a rule, so the sequence from the root can span more than two rules, which is complicated. In our design, each rule can be handled by different threads. If we can provide the head and tail buffers of all rules, we can avoid multi-rule scanning by looking into only the head and tail buffers of different subrules directly. In summary, the parents are responsible to process cross-rule sequences, and the problem can be solved by scanning the head and tail buffers of the direct children. More details are presented in Section IV-D.

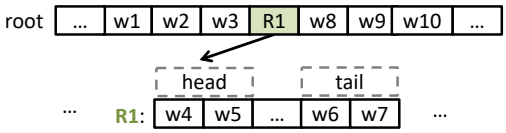


Fig. 6. Head and tail data structures for sequences.

D. Sequence Support in G-TADOC

In this part, we discuss the sequence support in G-TADOC for sequence sensitive applications. The sequence support in TADOC [2] is developed by function recursive calls, which is inefficient and hard to be parallel on GPUs. To improve the sequence support of TADOC and parallelize it on GPUs, we have the following insights. First, to fully parallelize the rule processing, each rule needs to include the head and tail buffers mentioned in Section IV-B to remove the sequence dependency across rules. Second, a first-round initialization phase is required to fulfill the head and tail buffers for all rules. Third, the original recursive design in TADOC [2] is inefficient and thus shall be abandoned; a more efficient parallel graph traversal needs to be developed.

Based on the analysis, we develop a two-phase sequence support design for sequence sensitive applications.

Initialization phase. The first initialization phase is to prepare the head and tail buffers for each rule with a light-weight scanning. The upper limit of memory space for each rule is shown in Equation 1, where *wordSize* denotes the size of the word elements, l denotes the sequence length, and *subRuleSize* denotes the number of subrules.

$$upperLimit = wordSize + (l-1) \times subRuleSize - (l-1) \quad (1)$$

The detailed process to generate the head and tail buffers of each rule is shown in Figure 7. The CPU side uses a while-loop to continuously check whether all the head and tail buffers have been fulfilled. To generate the head buffers, G-TADOC traverses the rules, and puts a given number of continuous words at the beginning of the rule in the head buffer. Within such a process, if G-TADOC encounters a subrule, G-TADOC first checks the related mask. If the mask is set, which implies that the subrule's head buffer is ready, then G-TADOC can put the content from the subrule's head buffer to the current rule's head buffer; otherwise, the calculation fails and needs to be conducted in the next round. The generation of the tail buffers is similar to the generation of the head buffers.

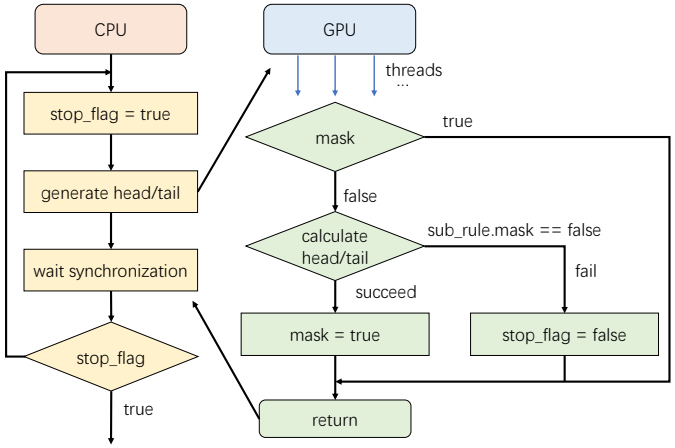


Fig. 7. Phase 1: initialization for head and tail buffers.

Graph traversal phase. The second phase of sequence support is shown in Figure 8. Similar to the first phase shown in Figure 7, the CPU part uses a while-loop to control the DAG traversal process. We use *sequence count* [2] as an example, which uses the hash tables described in Figure 5. For *sequence support*, we need to reduce the intermediate results in the local tables from the rules. We use parallel hash tables to merge these results, as discussed in Section IV-C. First, we distribute each key-value pair a *mask*, and each entry a *lock*. Second, each thread is responsible for one key-value pair. Third, each thread needs to justify whether it is necessary to insert a key-value pair. If not, G-TADOC returns directly; otherwise, G-TADOC obtains the entry based on hash functions, and then verifies if the same key already exists on this entry. If the key exists, G-TADOC uses atomic additions directly, and then sets the mask to *true*; otherwise, G-TADOC tries to obtain the lock of the entry. If the lock is occupied by other threads, G-TADOC sets the stop flag to *false* and returns directly; if G-TADOC obtains the lock, G-TADOC needs to verify whether the same key coexists. If the same key coexists, G-TADOC uses atomic additions to avoid this issue; otherwise, G-TADOC obtains a new node and sets the entry accordingly. Finally, G-TADOC unlocks the table, sets the mask to *true*, and returns. Note that the CPU part continuously launches this process until the stop flag is set to *true*.

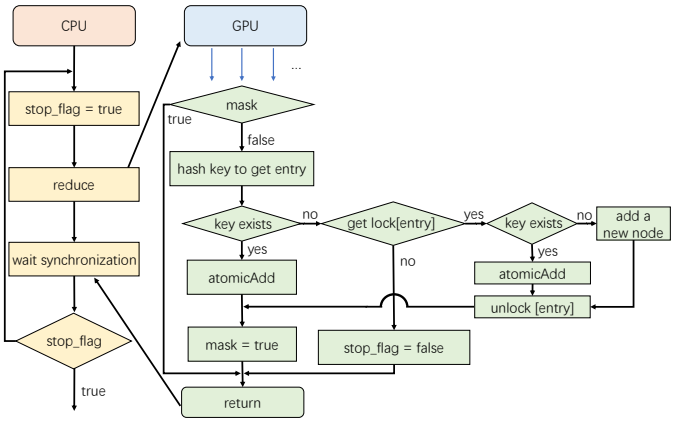


Fig. 8. Phase 2: graph traversal with sequence support.

V. Implementation

We integrate our G-TADOC into the CompressDirect (CD) [2] library, which is an implementation of TADOC. G-TADOC in CD includes two parts: 1) the CPU part that is used to input data and program, and to handle the GPU module, and 2) the GPU part that is used for GPU-based TADOC acceleration. We use the same interfaces as TADOC in CD, including *word count*, *sort*, *inverted index*, *term vector*, *sequence count*, and *ranked inverted index*, so users do not need to change any code in this GPU support.

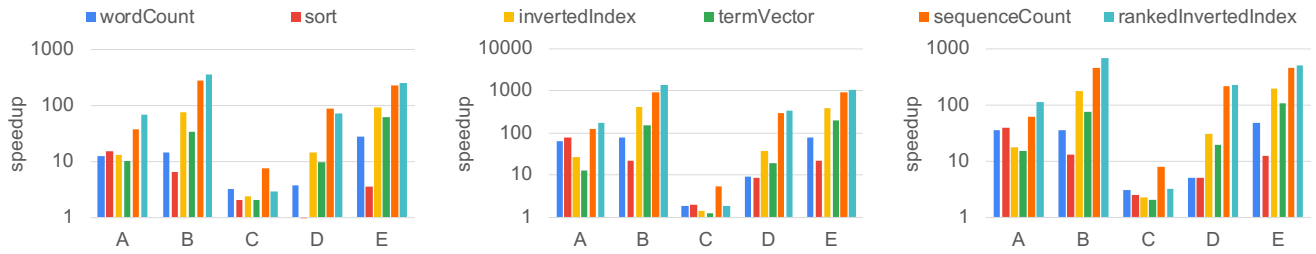
VI. Evaluation

In this section, we measure the performance of G-TADOC and compare it with TADOC [2] for evaluation.

A. Experimental Setup

We show our experimental setup in this part.

Methodology. The baseline in our evaluation is TADOC [2], which is the state-of-the-art data analytics directly on compression, denoted as "TADOC". Our method that enables TADOC on GPUs is denoted as "G-TADOC". In our evaluation, we measure TADOC [2] performance and G-TADOC performance for comparison. Moreover, we assume that small datasets can be stored and processed in GPU memory directly without PCIe data transmission; large datasets are stored on disk with PCIe data transmission required to be involved in time measurement.



(a) Pascal (GeForce GTX 1080). (b) Volta (Tesla V100). (c) Turing (GeForce RTX 2080 Ti).

Fig. 9. Performance speedups.

Platforms. We use three GPU platforms and a 10-node Amazon EC2 cluster in our evaluation, as shown in Table I. We evaluate G-TADOC on three generations of Nvidia GPUs (Pascal, Volta, and Turing micro-architectures), which are used to prove the adaptability of G-TADOC. Since GPU architectures are constantly changing, if we can achieve high performance on all these platforms, then it is very likely that G-TADOC can achieve promising results for future GPU products. The 10-node cluster is a Spark cluster on Amazon EC2 [16] for large datasets of TADOC.

TABLE I
PLATFORM CONFIGURATION.

Platform	Pascal	Volta	Turing	10-node cluster
GPU	GTX 1080	V100	RTX 2080 Ti	NULL
GPU Memory	GDDR5X	HBM2	GDDR6	DDR3
CPU	i7-7700K	E5-2670	i9-9900K	E5-2676v3
OS	Ubuntu 16.04.4	Ubuntu 16.04.4	Ubuntu 18.04.5	Ubuntu 16.04.1
Compiler	CUDA 8	CUDA 10.1	CUDA 11.0	GCC 5.4.0

Datasets. The datasets used in our evaluation are shown in Table II, which include various real-world workloads. Datasets A, B, and C are used in [1]–[4]. Dataset A is NSF Research Award Abstracts (NSFRAA) downloaded from UCI Machine Learning Repository [17], and is composed of a large number of small files. Dataset B is a collection of four web documents downloaded from Wikipedia [15]. Dataset C is a large Wikipedia dataset [15]. To increase the diversity of test data, we add datasets D and E compared to previous works [1]–[4]. Dataset D is COVID-19 data from Yelp [18], and dataset E is a collection of DBLP web documents [19]. Note that only dataset C is evaluated on the 10-node cluster.

TABLE II
DATASETS ("SIZE": ORIGINAL UNCOMPRESSED SIZE).

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

B. Performance

In this part, we measure the speedups of G-TADOC over TADOC and show their time breakdowns.

Overall speedups. We show the speedups that G-TADOC achieves over TADOC [2] in five datasets in Figure 9. In detail, Figure 9 (a) shows the speedups on Pascal platform, Figure 9 (b) shows the speedups on Volta platform, and Figure 9 (c) shows the speedups on Turing platform. We have the following observations.

First, G-TADOC achieves significant performance speedups over TADOC in all cases. On average, G-TADOC achieves 31.1× speedup over TADOC. The reason is that the GPU device for G-TADOC provides much higher computing power and bandwidth than the CPU device for TADOC. For example, on the Pascal platform, the theoretical peak performance of the GPU is about 185.3× over the theoretical peak performance of the CPU. Moreover, the bandwidth provided by the GPU memory is about 8.3× over the memory bandwidth provided by the CPUs. The performance speedups achieved by G-TADOC further prove the effectiveness of our solutions to handle the dependencies in our parallel design for GPUs.

Second, the speedups of G-TADOC over TADOC on single nodes for processing small datasets are higher than the speedups of G-TADOC over TADOC on clusters for processing large dataset C. The average speedup of G-TADOC over TADOC on a single node is 57.5×, while the average speedup of G-TADOC over TADOC on a ten-node cluster is 2.7×. The reason is that when processing the large dataset, TADOC adopts coarse-grained parallelism in distributed environments to improve the data processing efficiency. However, due to the data exchange overhead between nodes in the distributed environment of TADOC, our G-TADOC is still more efficient than TADOC.

Third, the speedups G-TADOC achieves for *sequence count* and *ranked inverted index* are much higher than the speedups of the other applications in most cases. In detail, the average speedups of *sequence count* and *ranked inverted index* are 111.3× and 112.0×, which are much higher than the full range average speedup. The reason is that *sequence count* and *ranked inverted index* of TADOC in [2] is of low performance: as described in [2], the performance behaviors of *sequence count* and *ranked inverted index* of TADOC are close to those of the original implementations on uncompressed data without compression. As to G-TADOC, *sequence count* and *ranked inverted index* reuse the partial results of duplicate data and execute in parallel on GPUs.

C. Optimization Analysis

We analyze G-TADOC acceleration in different phases and the traversal strategies on GPUs in this part.

Speedups in different phases. We show the separate speedups of G-TADOC over TADOC in different phases in Figure 10. First, the average speedup in the second phase is 64.1×, which implies that most G-TADOC performance benefits come from the acceleration in the second phase of DAG traversal. The second phase also has a relatively long execution time in TADOC [2], which provides more parallel optimization opportunities. Second, for the first phase, although the execution time is relatively short, G-TADOC still achieves clear performance speedups: on average, G-TADOC is 9.5× faster than TADOC on CPUs. Third, in large dataset C, the speedups in the first initialization phase are extremely high, which shows that the data structure preparation for massive large files is time-consuming in TADOC [2] and is essential to be accelerated by G-TADOC.

Top-down vs. bottom-up traversals. We develop top-down and bottom-up traversals in G-TADOC, but the optimal traversal strategy for each application can be input dependent. For example, for *term vector* in dataset A, the top-down traversal takes 14.04 seconds, but the bottom-up traversal takes only 1.56 seconds. In contrast, for *term vector* in dataset B, the bottom-up traversal takes 0.43 seconds, but the top-down traversal takes only 0.11 seconds. In detail, dataset B involves only four files. If we traverse the DAG in a top-down strategy, we only need to maintain a small buffer of 16 bytes in each rule indicating its file information, and the transmission for file information in DAG traversal is also marginal. In contrast, for dataset A, which involves a large number of small files, the top-down traversal with file information would be time-consuming and drags down the overall performance. Therefore, we should select the bottom-up traversal strategy in dataset A, and top-down strategy in dataset B. We apply the TADOC adaptive traversal strategy selector on GPUs, as discussed in Section IV-B, which can help select the optimal traversal strategy.

D. Summary of Findings

We summarize our findings and insight as follows.

First, we find that GPUs are very suitable for text analytics directly on compression, but need special optimizations. For example, G-TADOC needs fine-grained thread-level workload scheduling for GPU threads, thread-safe data structures for parallel updates, and head and tail structures for sequence sensitive applications.

Second, the GPU platform is both cost-effective and energy-efficient, which can be applied to a wide range of data analytics applications directly on compression, especially in large data centers. Experiments show that a GPU server can have much higher performance on data analytics directly on compressed data than a ten-node cluster does.

Third, although the GPU memory is limited, our work can help put much larger content directly in GPU memory. The frequent data transmission between the CPU and GPU drags down the performance advantages of GPUs when large workloads fail to be loaded to the GPU memory at once. Our work sheds light on the GPU acceleration design for such big data applications.

E. Discussion

We next show the importance of our paper and future work.

Importance of our work. As the first work enabling efficient GPU-based text analytics without decompression, G-TADOC provides the insights that are of interests to a wide range of readers. Currently, G-TADOC involves only the applications in TADOC [2], but other data analytics tasks can all benefit from G-TADOC. Furthermore, the series of optimizations on GPUs for TADOC can be directly applied to other advanced data analytics scenarios.

Comparison with GPU-accelerated uncompressed analytics. In our evaluation for the six data analytics tasks with the five datasets, G-TADOC reaches 31.1× of the performance of the state-of-the-art TADOC on CPUs. A common question is how the G-TADOC performance differs from the performance of GPU-accelerated uncompressed analytics. Currently, there is no implementation about the six analytics tasks on GPUs, so we develop efficient GPU-accelerated uncompressed analytics for comparison. Experiments show that G-TADOC still achieves an average of 2× speedup.

Applicability. G-TADOC has the same applicability as TADOC [4]. In general, G-TADOC targets the analytics tasks that can be expressed as a DAG traversal problem, which involves scanning the whole DAG.

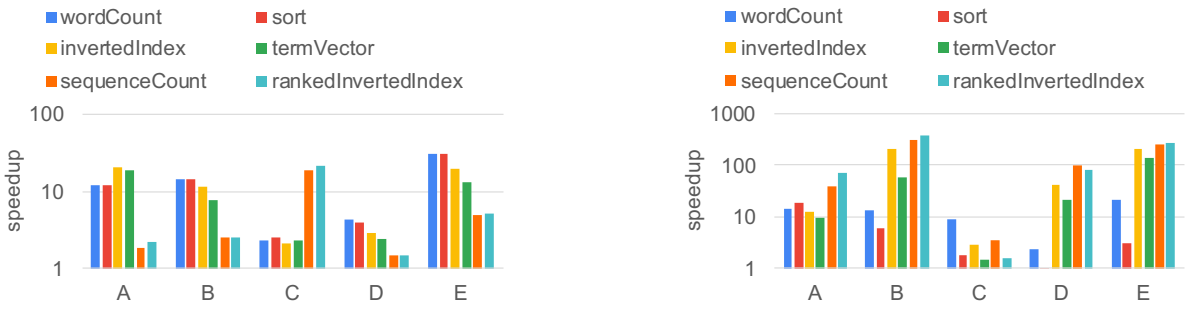
How far is the performance from the optimal? Although G-TADOC already achieves high performance, it still has room for performance improvement. The reasons include 1) dependencies in DAG traversal, 2) random accesses on large memory space, and 3) atomic operations on global buffers. When these issues are solved, G-TADOC can achieve at least 20% extra performance improvement.

Future work. Currently, G-TADOC supports data analytics directly on compression on GPUs. This research is headed for high-performance and efficient data analytics methods. The future possible avenues of exploration include architecture optimizations or multi-GPU environments, which can further accelerate G-TADOC.

VII. Related Work

As far as we know, G-TADOC is the first work that enables efficient GPU-based text analytics without decompression. In this section, we show the related work of grammar compression, compression-based data analytics, and GPU data analytics.

Grammar compression. There are plenty of works on grammar compression [1]–[4], [20]–[28]. The closest work to G-TADOC is TADOC, which is the text analytics



(a) Phase 1: initialization.

(b) Phase 2: traversal.

Fig. 10. Separate speedups for different phases.

directly on compression in single-node and distributed environments [2]. TADOC extends Sequitur [12]–[14] as its compression algorithm for data analytics. After TADOC being proposed, Zhang *et al*. [1] proposed Zwift, which the first TADOC programming framework, including a domain specific language, TADOC compiler and runtime, and a utility library. Then, Zhang *et al*. [4] applied TADOC as the storage to support advanced document analytics, such as *word co-occurrence* [29], [30], *term frequency-inverse document frequency* (TFIDF) [11], *word2vec* [31], [32], and *latent Dirichlet allocation* (LDA) [10]. Furthermore, Zhang *et al*. [3] enabled random accesses to TADOC compressed data, and at the same time, supported *insert* and *append* operations. In this work, we enable TADOC on GPUs, which improves the performance of TADOC significantly.

Index compression. The compression-based data analytics is an active research domain in recent years. However, typical approaches mainly use suffix trees and indexes [33]–[38], [38]–[41]. Suffix trees are traditional representations for data compression [33], [42] but incur huge memory usage [43], [44]. Suffix arrays [45] and Burrows-Wheeler Transform [35], [36] are the development of these compression formats, but still generate high memory consumption [43]. Compressed suffix arrays [46]–[50] and FM-indexes [36], [51]–[54] are more efficient than the previous compression techniques. Furthermore, Agarwal *et al*. proposed Succinct [34], which targets queries on compressed data. Moreover, there are many works about inverted index compression [55]–[61]. For example, Petri and Moffat [55] developed compression tools for compressed inverted indexes. Different from these works, G-TADOC targets text analytics directly on compressed data on GPUs.

GPU data analytics. GPUs have been applied to various aspects of data analytics, including structured data analytics, stream data analytics, graph analytics, and machine learning analytics [5]–[9], [62]–[65]. For example, MapD (Massively Parallel Database) [5] is a popular big data analytics platform powered by GPUs. Most current analytics frameworks, such as Spark, have supported GPUs [6]. SABER [7] is a stream system that schedules queries on both CPUs and GPUs, and Zhang *et al*. [9] further developed FineStream, which enables fine-grained stream analytics on CPU-GPU integrated architectures. Gunrock [8] is an efficient graph library for graph analytics on GPUs, and for large graphs, multi-GPU graph analytics have been explored [62]. For machine learning data analytics, parallel technologies have been extensively applied to various aspects, especially for deep learning applications [63]. Currently, most machine learning frameworks, such as TensorFlow [64], support GPU.

VIII. Conclusion

In this paper, we have presented G-TADOC enabling efficient GPU-based text analytics without decompression. We show the challenges of parallelism, result update conflicts from multi-threads, and sequence sensitivities in developing TADOC on GPUs, and present a series of solutions in solving these challenges. By developing an efficient parallel execution engine with data structures and sequence support on GPUs, G-TADOC achieves 31.1× speedup on average compared to state-of-the-art TADOC.

Acknowledgment

This work is partially supported by National Natural Science Foundation of China (Grant No. U20A20226, 61802412, and 61732014), Beijing Natural Science Foundation (4202031 and L192027), State Key Laboratory of Computer Architecture (ICT,CAS) under Grant No. CARCHA202007, and Beijing Academy of Artificial Intelligence (BAAI), Tsinghua University-Peking Union Medical College Hospital Initiative Scientific Research Program. This work is also supported by National Science Foundation (NSF) under Grants CNS-1717425, CCF-1703487, CCF-2028850, and the Department of Energy (DOE) under Grant DE-SC0013700. Jidong Zhai, Xipeng Shen, and Xiaoyong Du are the corresponding authors of this paper.

References

[1] F. Zhang, J. Zhai *et al.*, "Zwift: A Programming Framework for High Performance Text Analytics on Compressed Data," in *ICS*, 2018.
[2] F. Zhang, J. Zhai *et al.*, "Efficient document analytics on compressed data: Method, challenges, algorithms, insights," *PVLDB*, 2018.
[3] F. Zhang, J. Zhai *et al.*, "Enabling efficient random access to hierarchically-compressed data," in *ICDE*, 2020.

[4] F. Zhang, J. Zhai *et al.*, "TADOC: Text analytics directly on compression," *The VLDB Journal*, 2020.

[5] C. Root and T. Mostak, "MapD: A GPU-powered big data analytics and visualization platform," in *ACM SIGGRAPH 2016 Talks*, 2016.

[6] Y. Yuan, M. F. Salmi *et al.*, "Spark-GPU: An accelerated in-memory data processing engine on clusters," in *Big Data*, 2016.

[7] A. Koliousis, M. Weidlich *et al.*, "SABER: Window-based hybrid stream processing for heterogeneous architectures," in *International Conference on Management of Data*, 2016.

[8] Y. Wang, Y. Pan *et al.*, "Gunrock: GPU graph analytics," *ACM Transactions on Parallel Computing (TOPC)*, 2017.

[9] F. Zhang, L. Yang *et al.*, "FineStream: Fine-Grained Window-Based Stream Processing on CPU-GPU Integrated Architectures," in *USENIX ATC*, 2020.

[10] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, 2003.

[11] T. Joachims, "A Probabilistic Analysis of the Rocchio Algorithm with TFIDF for Text Categorization." Carnegie-mellon univ pittsburgh pa dept of computer science, Tech. Rep., 1996.

[12] C. G. Nevill-Manning, "Inferring sequential structure," Ph.D. dissertation, University of Waikato, 1996.

[13] C. G. Nevill-Manning and I. H. Witten, "Identifying hierarchical structure in sequences: A linear-time algorithm," *J. Artif. Intell. Res.*, 1997.

[14] C. G. Nevill-Manning and I. H. Witten, "Linear-time, incremental hierarchy inference for compression," in *Data Compression Conference*, 1997.

[15] "Wikipedia HTML data dumps," https://dumps.wikimedia.org/enwiki/, 2017.

[16] E. Amazon, "Amazon elastic compute cloud (Amazon EC2)," *Amazon Elastic Compute Cloud (Amazon EC2)*, 2010.

[17] M. Lichman, "UCI machine learning repository," http://archive.ics.uci.edu/ml, 2013.

[18] "COVID-19 Data from Yelp Open Dataset," https://www.yelp.com/dataset, 2019.

[19] "DBLP," https://dblp.uni-trier.de/xml/, 2020.

[20] W. Rytter, "Grammar compression, lz-encodings, and string algorithms with implicit input," in *International Colloquium on Automata, Languages, and Programming*, 2004.

[21] M. Charikar, E. Lehman *et al.*, "The smallest grammar problem," *IEEE Transactions on Information Theory*, 2005.

[22] T. Gagie, P. Gawrychowski *et al.*, "A faster grammar-based self-index," in *International Conference on Language and Automata Theory and Applications*, 2012.

[23] P. Bille, G. M. Landau *et al.*, "Random access to grammar-compressed strings and trees," *SIAM Journal on Computing*, 2015.

[24] P. Bille, A. R. Christiansen *et al.*, "Finger search in grammar-compressed strings," *arXiv preprint arXiv:1507.02853*, 2015.

[25] N. R. Brisaboa, A. Gómez-Brandón *et al.*, "Gract: a grammar-based compressed index for trajectory data," *Information Sciences*, 2019.

[26] M. Ganardi, A. Jeż, and M. Lohrey, "Balancing straight-line programs," in *Annual Symposium on Foundations of Computer Science*, 2019.

[27] Y. Takabatake, H. Sakamoto *et al.*, "A space-optimal grammar compression," in *25th Annual European Symposium on Algorithms*, 2017.

[28] R. Wu, F. Zhang *et al.*, "POSTER: Exploring Deep Reuse in Winograd CNN Inference," in *PPoPP*, 2021.

[29] Y. Matsuo and M. Ishizuka, "Keyword extraction from a single document using word co-occurrence statistical information," *International Journal on Artificial Intelligence Tools*, 2004.

[30] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *EMNLP*, 2014.

[31] X. Rong, "word2vec parameter learning explained," *arXiv preprint arXiv:1411.2738*, 2014.

[32] "word2vec," https://code.google.com/archive/p/word2vec/, 2013.

[33] G. Navarro, *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

[34] R. Agarwal, A. Khandelwal, and I. Stoica, "Succinct: Enabling queries on compressed data," in *NSDI*, 2015.

[35] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," 1994.

[36] P. Ferragina and G. Manzini, "Indexing compressed text," *Journal of the ACM (JACM)*, 2005.

[37] R. Grossi, A. Gupta, and J. S. Vitter, "When indexing equals compression: Experiments with compressing suffix arrays and applications," in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, 2004.

[38] P. Ferragina, R. González *et al.*, "Compressed text indexes: From theory to practice," *Journal of Experimental Algorithmics (JEA)*, 2009.

[39] A. Farruggia, P. Ferragina, and R. Venturini, "Bicriteria data compression: efficient and usable," in *European Symposium on Algorithms*, 2014.

[40] P. Ferragina, I. Nitto, and R. Venturini, "On the bit-complexity of lempel-ziv compression," in *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2009.

[41] S. Gog, T. Beller *et al.*, "From theory to practice: Plug and play with succinct data structures," in *International Symposium on Experimental Algorithms*, 2014.

[42] K. Sadakane, "Compressed suffix trees with full functionality," *Theory of Computing Systems*, 2007.

[43] W.-K. Hon, T. W. Lam *et al.*, "Practical aspects of Compressed Suffix Arrays and FM-Index in Searching DNA Sequences," in *ALENEX/ANALC*, 2004.

[44] S. Kurtz, "Reducing the space requirement of suffix trees," *Software: Practice and Experience*, 1999.

[45] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," *siam Journal on Computing*, 1993.

[46] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, 2003.

[47] R. Grossi and J. S. Vitter, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching," *SIAM Journal on Computing*, 2005.

[48] K. Sadakane, "Compressed text databases with efficient query algorithms based on the compressed suffix array," in *International Symposium on Algorithms and Computation*, 2000.

[49] K. Sadakane, "Succinct representations of lcp information and improvements in the compressed suffix arrays," in *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, 2002.

[50] K. Sadakane, "New text indexing functionalities of the compressed suffix arrays," *Journal of Algorithms*, 2003.

[51] "FM-index," https://en.wikipedia.org/wiki/FM-index, 2018.

[52] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Foundations of Computer Science. Proceedings. 41st Annual Symposium on*, 2000.

[53] P. Ferragina and G. Manzini, "An experimental study of a compressed index," *Information Sciences*, 2001.

[54] P. Ferragina and G. Manzini, "An experimental study of an opportunistic index," in *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, 2001.

[55] M. Petri and A. Moffat, "Compact inverted index storage using general-purpose compression libraries," *Software: Practice and Experience*, 2018.

[56] A. Moffat and M. Petri, "Index compression using byte-aligned ANS coding and two-dimensional contexts," in *WSDM*, 2018.

[57] G. E. Pibiri, M. Petri, and A. Moffat, "Fast dictionary-based compression for inverted indexes," in *WSDM*, 2019.

[58] G. E. Pibiri and R. Venturini, "Techniques for Inverted Index Compression," *arXiv preprint arXiv:1908.10598*, 2019.

[59] G. E. Pibiri, R. Perego, and R. Venturini, "Compressed Indexes for Fast Search of Semantic Data," *TKDE*, 2020.

[60] H. Oosterhuis, J. S. Culpepper, and M. de Rijke, "The potential of learned index structures for index compression," in *Proceedings of the 23rd Australasian Document Computing Symposium*, 2018.

[61] J. Mackenzie, A. Mallia *et al.*, "Compressing inverted indexes with recursive graph bisection: A reproducibility study," in *European Conference on Information Retrieval*, 2019.

[62] Y. Pan, Y. Wang *et al.*, "Multi-GPU graph analytics," in *IPDPS*, 2017.

[63] S. R. Upadhyaya, "Parallel approaches to machine learning—a comprehensive survey," *Journal of Parallel and Distributed Computing*, 2013.

[64] M. Abadi, A. Agarwal *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[65] F. Zhang, Z. Chen *et al.*, "An Efficient Parallel Secure Machine Learning Framework on GPUs," *TPDS*, 2021.