Exploring Data Analytics without Decompression on Embedded GPU Systems

Zaifeng Pan, Feng Zhang, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, Xiaoyong Du

Abstract—With the development of computer architecture, even for embedded systems, GPU devices can be integrated, providing outstanding performance and energy efficiency to meet the requirements of different industries, applications, and deployment environments. Data analytics is an important application scenario for embedded systems. Unfortunately, due to the limitation of the capacity of the embedded device, the scale of problems handled by the embedded system is limited. In this paper, we propose a novel data analytics method, called G-TADOC, for efficient text analytics directly on compression on embedded GPU systems. A large amount of data can be compressed and stored in embedded systems, and can be processed directly in the compressed state, which greatly enhances the processing capabilities of the systems. Particularly, G-TADOC has three innovations. First, a novel fine-grained thread-level workload scheduling strategy for GPU threads has been developed, which partitions heavily-dependent loads adaptively in a fine-grained manner. Second, a GPU thread-safe memory pool has been developed to handle inconsistency with low synchronization overheads. Third, a sequence-support strategy is provided to maintain high GPU parallelism while ensuring sequence information for lossless compression. Moreover, G-TADOC involves special optimizations for embedded GPUs, such as utilizing the CPU-GPU shared unified memory. Experiments show that G-TADOC provides 13.2$\times$ average speedup compared to the state-of-the-art TADOC. G-TADOC also improves performance-per-cost by 2.6$\times$ and energy efficiency by 32.5$\times$ over TADOC.

Index Terms—TADOC, Embedded GPU Systems, Compression, Data Analytics.

1 Introduction

In daily lives, there is an increasing need for light-weight convenient embedded devices to facilitate data analytics tasks, and embedded systems, such as Nvidia Jetson XAVIER NX [1], have integrated GPUs with CPUs on the same chip, bringing super performance to the edge [2]. We show a Jetson XAVIER NX platform in Figure 1, which contains the whole computer system components including a GPU. Importantly, its length is less than the length of a pen, and its price is low. However, the current embedded GPUs are still less powerful and have limited storage space compared to the discrete GPUs. Fortunately, a recent technology, text analytics directly on compression (TADOC) [3], [4], [5], [6] 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 [3], [4]. Therefore, it is greatly beneficial to apply TADOC on embedded GPU systems.

Enabling TADOC on embedded GPU systems introduces three advantages. First, the storage capacity of current embedded GPU systems is quite limited, so processing on compression can greatly enlarge its capability. For example, the latest Nvidia Jetson XAVIER NX [1] embedded platform has only 8GB memory with microSD enabled for storage. Second, the GPU device is much powerful than the CPU on embedded systems, and utilizing TADOC on embedded GPUs can greatly enhance GPU data processing ability towards massive data. For example, the GPU device on Nvidia Jetson XAVIER NX brings 21 TOPs computing capacity, which is much higher than its CPU [1]. Third, intelligent edge solutions require advanced reasoning capabilities to solve complex problems, and many advanced data analytics applications, such as *term frequency-inverse document frequency* (TFIDF) [7], have been provided to be suitable with TADOC. Hence, enabling TADOC on embedded GPUs can

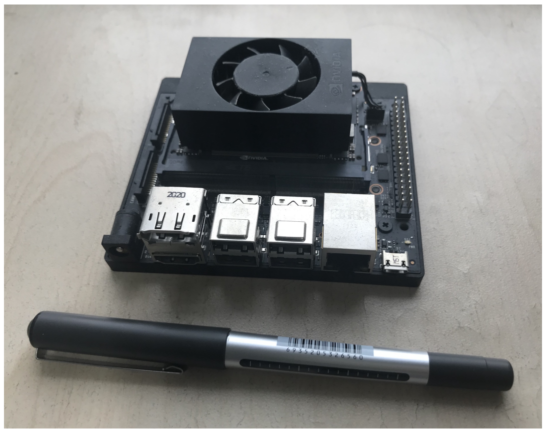


Fig. 1. Nvidia Jetson XAVIER NX. Mechanical 103 mm $\times$ 90.5 mm $\times$ 34.66 mm.

- *Z. Pan, F. Zhang, Y. Zhou, and X. Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China, Beijing 100872, China.*
- *J. Zhai is with the Department of Computer Science and Technology, Tsinghua University, Beijing, 100084, China.*
- *X. Shen is with the Computer Science, North Carolina State University, Raleigh, NC 27695, US.*
- *O. Mutlu is with Department of Computer Science, ETH Zurich, Zurich 8092, Switzerland.*

remove the last barrier to apply embedded systems to wide application scope.

Although enabling TADOC on embedded GPUs brings significant benefits, developing efficient GPU-enabled TADOC is extremely challenging. First, TADOC transforms data into rules, which can be further 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. Moreover, special optimizations, such as utilizing the CPU-GPU shared unified memory towards embedded GPUs, need to be designed.

There is a large amount of TADOC literature, but unfortunately, none of the current TADOC solutions solve the challenges of enabling TADOC on GPUs mentioned above. Zhang *et al*. [4] 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 [3], and further realized random accesses on compressed data [5]. However, the parallelism problems still exist. Zhang *et al*. [6] then provided a parallel TADOC design, which provides much better performance than the sequential TADOC. However, 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, not to mention the embedded heterogeneous systems.

To solve the aforementioned challenges, we develop 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 4.2). 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 4.3). 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 4.4). Our preliminary work has been presented in [8], which only provides a simple design without embedded GPU optimizations and analysis. Compared to [8], we provide new insights and optimizations. Moreover, we add new benchmarks and datasets with extra experiments.

We evaluate G-TADOC on currently the most powerful embedded GPU platform, Nvidia JETSON AGX XAVIER, and three discrete GPU platforms, which cover three generations of Nvidia GPUs (Pascal, Volta, and Turing micro-architectures). We use six real-world datasets of varying lengths, structures, and content. Compared to TADOC on CPUs, G-TADOC achieves 13.2× speedup. In detail, TADOC can be divided into two phases: initialization and DAG traversal. For the initialization phase, G-TADOC achieves 25.9% time saving, while for the DAG traversal phase, G-TADOC achieves 78.6% time saving. Moreover, performance per cost and energy efficiency are also important in data centers, and G-TADOC achieves 2.6× performance-per-cost benefits and 32.5× energy efficiency over TADOC.

As far as we know, this is the first work enabling efficient text analytics on heterogeneous platforms, including embedded GPU platforms and discrete GPU platforms, 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 exhibit our optimizations of G-TADOC on embedded GPU platforms, and show the benefits from both power efficiency and cost efficiency perspectives.
- We evaluate G-TADOC on four GPU platforms, and demonstrate its significant benefits compared to the state-of-the-art TADOC.

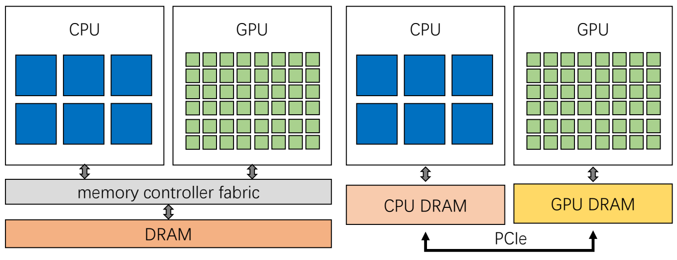
2 Background and Premises

We introduce the background and premises of embedded GPU systems and TADOC in this section.

2.1 Embedded GPUs

CPU-GPU embedded systems are becoming increasingly popular. These small but advanced "supercomputers" are widely used in industry and daily applications, such as autonomous robotics and edge computing [2]. These embedded GPU systems usually have extremely low power, but with much higher computing capacity compared to CPUs. For example, Nvidia Jetson Xavier NX integrates an ARM CPU with an Nvidia Volta GPU together, delivering 14 TOPs performance under only 10W power [1].

Difference from discrete GPUs. We show an embedded GPU and discrete GPU comparison in Figure 2. First, they have different architectural designs. The heterogeneous embedded systems integrate a CPU and a GPU together. They share the same physical unified memory via the memory controller fabric without PCIe communication, as shown in Figure 2 (a), which is different from discrete GPUs shown in Figure 2 (b). Second, the embedded GPUs have much lower power than the discrete GPUs, so the embedded systems are more energy-efficient. Third, the embedded heterogeneous systems are usually less powerful than the discrete GPUs. For example, embedded GPUs have fewer computing cores and lower frequencies than discrete GPUs. The memory and storage size of embedded systems are also smaller.



(a) Embedded GPU system. (b) Discrete GPU system.

Fig. 2. Comparison between embedded and discrete GPU platforms.

Advantages. The embedded GPU systems have the following advantages. First, the embedded GPU systems have more advanced architecture designs: the GPU and the CPU share the same memory buffer, so they can have more fine-grained cooperation without PCIe data transmission. Second, the embedded GPUs have much lower cost and power than discrete GPUs, which are power- and cost-efficient. Third, the embedded systems can have a much smaller size. Based on these advantages, the embedded GPU systems can be applied in various scenarios.

Limitation and opportunity. Unfortunately, though embedded GPU systems have the above advantages, their restricted memory and storage capacities limit their solvable problem scale and applicabilities. Fortunately, a recent promising technology, TADOC, has been proposed that can perform data processing directly on compression. If we can compress large data into the memory of embedded systems, the embedded systems can process a much larger workload, even larger than its memory and storage size. We next introduce the concepts and technologies of TADOC.

2.2 Text Analytics Directly on Compression

Text analytics directly on compression (TADOC) has been proved to be a promising technology that can save both time and space [3], [4], [5], [6]. In detail, TADOC is a novel lossless compression technique that enables data analytics directly on compressed data without decompression. 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. In TADOC, a rule is a symbol substitution pattern with the form of $A \rightarrow \alpha$, where A is a rule symbol and α is a sequence consisting of subrule symbols and word symbols. We can recursively substitute the repeated sequences of rule and word symbols with a substitution rule symbol to generate new 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.

Compression illustration. We use an example from [4] to illustrate how TADOC compresses data into CFG representation. Figure 3 (a) shows the original input data, which consists of two files: file A and file B, and "wi" represents a unique word. Figure 3 (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 3 (c) shows the TADOC compressed data, which are sequences of numbers. The TADOC compressed data can be viewed as CFG shown in Figure 3 (d), which can be further organized as a DAG shown in Figure 3 (e) for traversals. Accordingly, common data analytics are then transformed to a graph traversal problem.

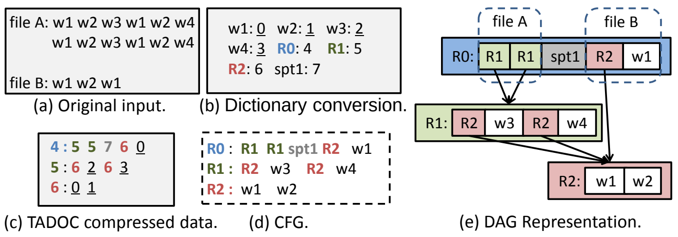


Fig. 3. A compression example with TADOC.

Case study of data analytics. We use *word count* as a case study to show how to perform data analytics on TADOC compressed data, as shown in Figure 4. 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. Other data analytics tasks can be conducted similarly [6].

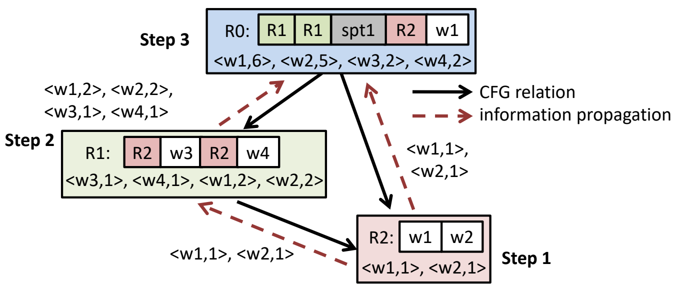


Fig. 4. Word count example using TADOC.

3 Motivation of G-TADOC

In this section, we show the motivation of our work and discuss the challenges.

3.1 Why do we need TADOC on embedded GPU systems?

We discuss the reason of enabling TADOC on embedded GPU systems in this part, which is also the motivation of this work.

First, enabling TADOC on embedded heterogeneous systems is of significant benefit. Data analytics is important in big data era. Due to the big data characteristics of four Vs (**v**olume, **v**elocity, **v**ariety, and **v**eracity) [9], it is necessary to utilize heterogeneous accelerators to facilitate CPUs for these various applications. However, as discussed in Section 2.1, the capacity of embedded heterogeneous systems is limited. If we can enable data analytics in embedded systems, we can greatly enhance the workload scale that can be handled.

Second, current GPU-based BFS traversal does not apply, due to the special features of the DAG representation of TADOC. On one hand, many data analytics tasks require sequence maintenance of words, where BFS cannot be used directly [4]. On the other hand, 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. Finally, 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.

Third, although we transform data analytics tasks to graph traversals, existing DAG traversals on GPUs cannot be used, due to the uniqueness of the DAG representation of TADOC. 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. Hence, we do not use existing designs.

3.2 Challenges

Enabling efficient TADOC on embedded GPU systems needs to address the following challenges.

The first challenge is GPU parallelism for TADOC. Although embedded GPUs have fewer computing cores compared to discrete GPUs, they still have much more cores providing massive parallelism compared to CPUs. For example, the embedded GPU JETSON AGX XAVIER integrates 512 light-weight Volta GPU cores [2]. The high performance of embedded GPU systems relies on the high throughput from thread-level parallelism. Unfortunately, as presented in [4], 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 [6]. 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. Even worse, 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 at warp level, and the threads within a warp have to release resources simultaneously. The workload imbalance problem decreases the parallelism degrees. Additionally, we cannot simply decide the number of threads for rules, because of the various rule length.

The second challenge comes from TADOC final result update conflict of massive GPU threads. The update conflict is a serious problem when we develop TADOC on GPUs. On the one hand, 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. Note that 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. On the other hand, 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. Even worse, the amount of memory required by TADOC is unknown until runtime. In developing TADOC on GPUs, the memory sizes of different threads are also various, which makes the update problem with thread conflicts more difficult.

The third challenge is 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. To keep the sequence information, TADOC originally traverses the DAG in a DFS order [4], which is hard to be parallel. Worse still, 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. Last but not least, 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.

Besides, embedded GPU systems integrate the CPU and the GPU together with the unified memory. Special optimizations need to be designed. Overall, enabling TADOC on embedded GPU systems is very rewarding, but full of challenges.

4 G-TADOC Design

We show our G-TADOC design in this section. G-TADOC targets GPU systems, and involves optimizations for embedded GPU systems.

4.1 General Design

We show the general design of G-TADOC in Figure 5. The input includes TADOC compressed data and user-defined programs, while the output is the required result.

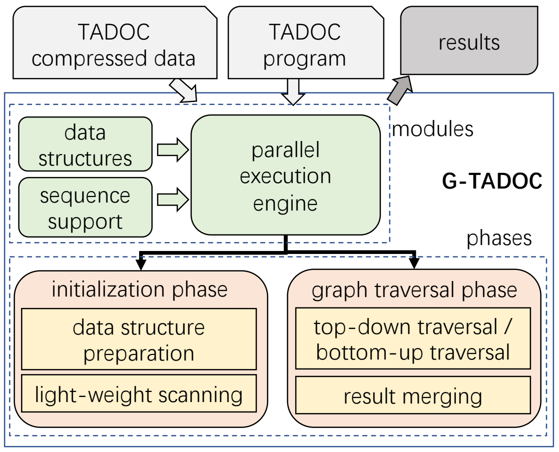


Fig. 5. G-TADOC overview.

Modules. G-TADOC is composed of three modules. The first module is the parallel execution module, which is responsible for G-TADOC parallel execution on GPUs. This module decides how to partition workloads for thread parallelism. The second module is the data structure module, which 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 third module is the sequence support module, which is used for applications that are sensitive to sequence orders. These three modules work together to complete G-TADOC tasks.

Phases. The execution workflow of G-TADOC can be divided into two phases. The first phase is the initialization phase. In this phase, G-TADOC prepares necessary data structures according to the user program after receiving the TADOC compressed data and program, and launches a light-weight scanning to fulfill related values. The second phase is the graph traversal phase. In this phase, G-TADOC analyzes different traversal strategies and chooses the most suitable one based on both data and tasks. Finally, G-TADOC performs a merging process for final results before the end of the graph traversal.

Solutions to challenges. G-TADOC can address the challenges mentioned in Section 3.2. 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 4.2). 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 unified memory objects with consistency guaranteed (Section 4.3). To address the third challenge of sequence sensitivities on GPUs, G-TADOC scans the DAG for recording the cross-rule content in a light-weight manner in the initialization phase. Then, G-TADOC performs a rule-level processing and a result merging process in the graph traversal phase (Section 4.4).

4.2 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 6. The first design is to partition the DAG vertically from the root: different parts are traversed by different threads, as shown in Figure 6 (a). This design can leverage the GPU parallelism, but at the same time, 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 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 [6], 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 [6] for such decisions.

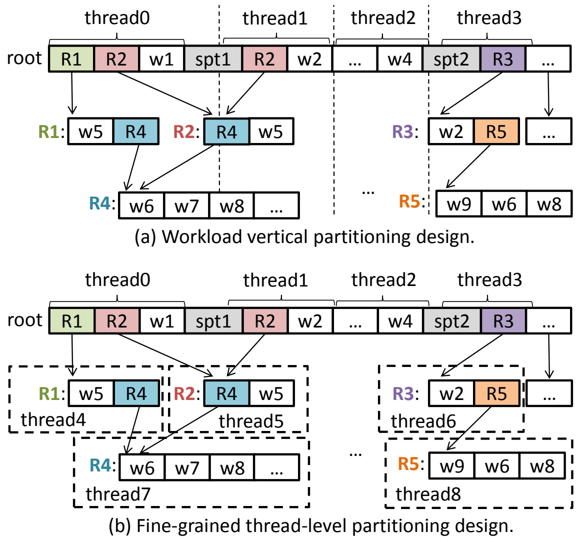


Fig. 6. Workload partitioning design exploration.

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, 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, we develop a self-maintained memory pool, detailed in Section 4.3. 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.

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

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 $devStopFlag \leftarrow true$
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:    $rule \leftarrow rules[tid]$
13:    if $rule.mask$ is $false$ then
14:        return
15:    for each $subRuleId, subRuleFreq$ in $rule.subRules$ do
16:        $subRule \leftarrow rules[subRuleId]$
17:        atomicAdd($subRule.weight, subRuleFreq * rule.weight$)
18:        atomicAdd($subRule.curInEdge, 1$)
19:        if $subRule.curInEdge$ is full then
20:            $subRule.mask \leftarrow true$ ▷ Sub-rule then can be traversed
21:            $devStopFlag \leftarrow false$
22:    $rule.mask \leftarrow false$
```

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 [4]) 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.

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 $devStopFlag \leftarrow true$
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 $devStopFlag \leftarrow true$
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
```

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). Similar to 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 [10]).

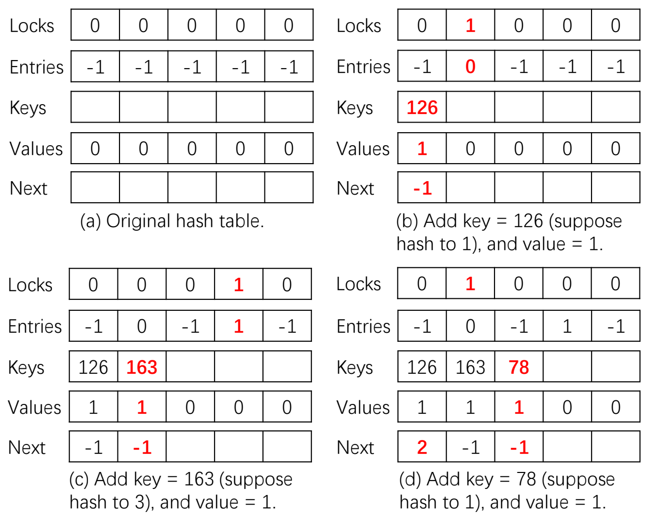
4.3 G-TADOC Data Structures

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 4.2, 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 3, 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 [4], 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 7. The original state of the hash table is shown in Figure 7 (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 7 (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 7 (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 7 (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 into 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.



Locks	0	0	0	0	0
Entries	-1	-1	-1	-1	-1
Keys					
Values	0	0	0	0	0
Next					

(a) Original hash table.

Locks	0	1	0	0	0
Entries	-1	0	-1	-1	-1
Keys	126				
Values	1	0	0	0	0
Next	-1				

(b) Add key = 126 (suppose hash to 1), and value = 1.

Locks	0	0	0	1	0
Entries	-1	0	-1	1	-1
Keys	126	163			
Values	1	1	0	0	0
Next	-1	-1			

(c) Add key = 163 (suppose hash to 3), and value = 1.

Locks	0	1	0	0	0
Entries	-1	0	-1	1	-1
Keys	126	163	78		
Values	1	1	1	0	0
Next	2	-1	-1		

(d) Add key = 78 (suppose hash to 1), and value = 1.

Fig. 7. Illustration for thread-safe hash tables.

Head and tail structures for sequence support. The head and tail structures are used to support sequence sensitive applications, such as *sequence count* [4]. Because G-TADOC traverses the DAG in parallel, rules can 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 8. 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. 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 6, 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 4.4.

4.4 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 [4] 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,

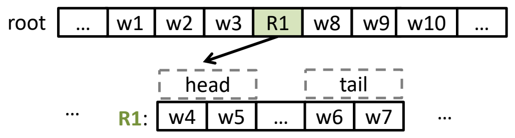


Fig. 8. Head and tail data structures for sequences.

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 4.2 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 [4] is inefficient and thus shall be abandoned. Hence, 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. Besides, the upper limit of memory space for the raw local sequence table in each rule is also calculated in this light-weight scanning.

The raw local sequence table is used to fit the top-down/bottom-up algorithms. The raw local sequence table only contains the word sequences that satisfy one of the following conditions: 1) the full word sequence resides within a rule, and 2) partial word sequence resides in a rule, while the remaining part residing in a subrule. In Figure 9, we use two rules with length $l = 3$ as an example for the construction of the raw local sequence table. For subrule *R2*, we only need to consider its head and tail, and the word sequences are <*w1,w2,w3*>, <*w2,w3,w4*>, <*w5,w6,w7*>, <*w6,w7,w8*>, and <*w7,w8,w9*>, with their beginning words labeled with triangles in Figure 9. Hence, when computing the upper limit in Equation 1, we consider that the sub-rule contains $l-1$ words that can be used as the beginning words of the sequences, excluding the $l-1$ words at the end.

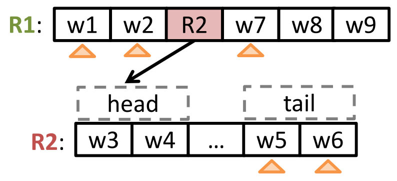


Fig. 9. Fig. 9. Construction of a raw local sequence table. The beginning words of the sequences are labelled with triangles.

Hence, the memory upper limit of the raw local sequence table can be calculated by Equation 1, where *wordNum* denotes the number of the words, *l* denotes the length of sequence, and *subRuleNum* denotes the number of subrules.

$$upperLimit = wordNum + (l-1) \times subRuleNum - (l-1) \quad (1)$$

The detailed process to generate the head and tail buffers of each rule is shown in Figure 10. The CPU side uses a do-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.

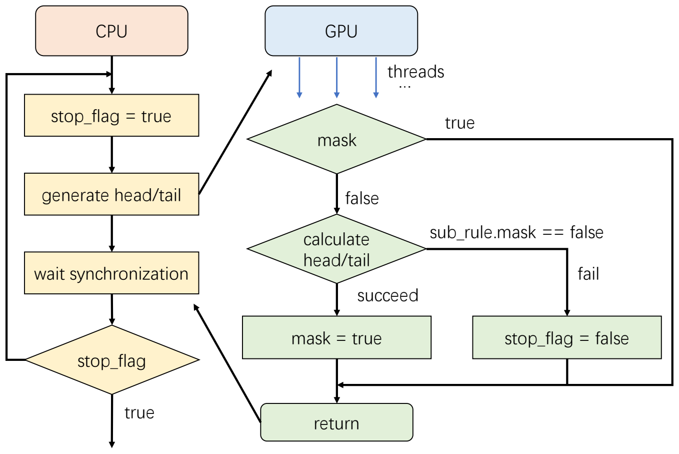


Fig. 10. Phase 1: initialization for head and tail buffers.

Graph traversal phase. The second phase of sequence support is shown in Figure 11. Similar to the first phase shown in Figure 10, the CPU part uses a do-while-loop to control the DAG traversal process. We use *sequence count* [4] as an example, which uses the hash tables described in Figure 7. 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 4.3. 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*.

5 EMBEDDED GPU OPTIMIZATION

In this section, we show our explorations and optimizations, especially for embedded GPU systems.

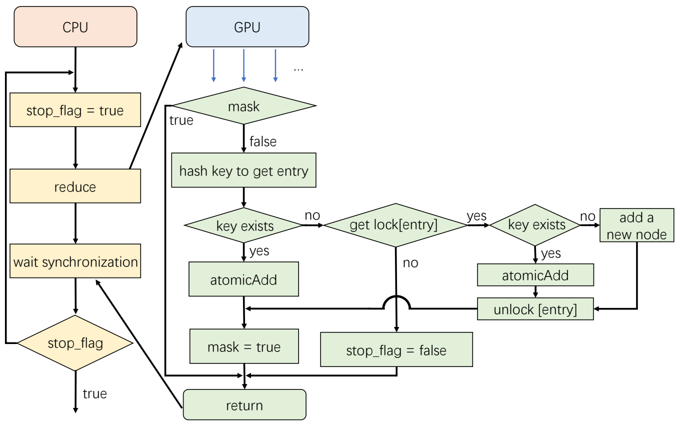


Fig. 11. Phase 2: graph traversal with sequence support.

5.1 Advantages over Discrete GPUs

We first compare the embedded GPU systems with discrete GPUs to highlight its advantages, and then show our optimization.

Detailed comparison. We show a detailed comparison of our embedded GPU system, Jetson AGX Xavier, and three discrete GPU systems, GTX 1080, GTX 2080Ti, and V100, in Table 1. First, the embedded GPU system exhibits high performance-per-cost potentials. The price of the whole embedded system is only 19.2% of the single discrete GPU on average. Second, the power of the embedded GPU system is much lower than that of the discrete GPU. In today's increasingly serious energy consumption situation, low-power devices are a good choice. Third, although the performance and bandwidth of embedded GPU systems are lower than those of the discrete systems, the latency has been proved to be shorter in certain circumstances with the help of unified memory [11], [12].

TABLE 1
Embedded GPU systems vs. discrete GPU systems.

	Integrated Architectures	Discrete Architectures		
Architecture	Jetson AGX Xavier	GTX 1080	RTX 2080 Ti	V100
# cores	512+8	2560	4352	5120
TFLOPS	11 (FP16)	8.873	13.45	14
bandwidth (GB/s)	136.5	320	616	900
price ($)	699	699	1199	8999
TDP (W)	30	180	260	250

The number of cores for the embedded platform includes eight CPU cores. For the discrete GPU platform, we only show the GPU device.

Unified memory. As discussed in Section 2.1, unified memory is a distinct feature between embedded GPU systems and discrete GPUs. Unified memory is a single address space accessible for both CPU and GPU in the embedded system [13], [14]. With the further demand paging technology from Pascal and later architectures, we can allocate unified memory objects beyond GPU memory limits. Unified memory is operated by a page migration engine: faults on non-resident memory accesses can cause the missing unified memory objects to be reallocated and migrated from other space in the system [13]. However, on discrete GPUs, memory migration between CPU and GPU can incur severe performance overhead. Fortunately, the integrated design on embedded systems solves this problem: both CPU and

GPU access the same memory, without extra performance cost on PCIe transmission.

Unified memory utilization. G-TADOC utilizes the unified memory on embedded GPU systems. For example, on our Jetson platform, CPU and GPU share the same memory, which means that after G-TADOC allocating unified memory space, both CPU and GPU can access the shared memory objects directly.The detailed process is as follows. First, G-TADOC allocates a unified memory buffer by using the CUDA API *cudaMallocManaged()*. Second, G-TADOC loads the compressed data, such as TADOC rules, into the unified memory. Third, G-TADOC processes the compressed data according to the user defined program without decompression. In the whole process of embedded GPU systems, G-TADOC avoids PCIe memory copy of discrete GPU systems.

5.2 Large Datasets

In big data era, the large input data can exceed the GPU memory, including the embedded GPU platforms. Although the unified memory has been utilized and TADOC alleviates this contradiction to some extent, we still need to develop effective design for processing large datasets that cannot be stored in memory after compression.

Basic idea. The general idea is that G-TADOC partitions data to different parts stored on disk, and processes different parts sequentially. Then, G-TADOC merges the results to generate final output.

General design. The process of handling large datasets consists of three steps. The first step is to transform the dataset into multiple individual DAGs, rather than a single one. Although this process incurs space cost due to the underutilized redundancy between different DAGs, this design is significantly beneficial for processing large data with limited memory. Second, G-TADOC processes different parts of the input data in turn, keeping the intermediate results. In detail, G-TADOC processes one DAG in memory in one iteration, and frees the memory object before processing the next DAG. Third, G-TADOC merges the intermediate results to generate the final output.

Detailed design. In detail, the process for handling large datasets involves two optimizations. First, the final output can exceed the memory space. For large result, if the results obtained from separate files are unrelated, such as *term vector*, we can store the results on disk in one iteration to avoid memory overflow. Second, for large intermediate results, we store them on disk and then perform a map-reduce process [15] to obtain the final result. For example, for *ranked inverted index*, when processing small datasets, G-TADOC uses a global hash table in memory to store word sequences' corresponding files and counts. In contrast, when handling large datasets, as Figure 12 shows, G-TADOC stores the items with the same hash value into one file on disk during an iteration.

5.3 Other Exploration

In this part, we show our other explorations, including workload balancing, shared memory, and CPU utilization on embedded systems.

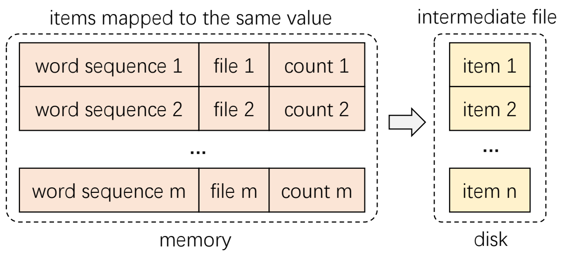


Fig. 12. Mapping items to files according to their hashed keys.

Workload balancing. One of the potential problems of our top-down and bottom-up G-TADOC design is workload imbalance. In each iteration of the do-while loop, we assign each rule a thread, and use a mask to check whether it should be executed or not. The workload for each thread can be different. For example, if a rule has a large number of subrules, this rule needs to transmit more times in a top-down traversal. For this problem, we consider using more threads for long rules.

Shared memory utilization. Shared memory are faster than the global memory, which should be utilized. In our design, when handling large datasets, we only assign threads to rules that should be processed during iteration. To obtain the rules in the next iteration, we consider storing the subrules that are active in the next iteration in the shared memory first, and then merge the subrule buffer to the global memory after block-level synchronization [13].

CPU utilization. On the embedded platform, both CPU and GPU share the same unified memory, so they can have more fine-grained cooperation, which means that operations that are not suitable for the GPU part can be ported to the CPU for execution. In our design, we use the CPU to check whether the loop should be stopped, since this process cannot be parallelized. Additionally, the data stored on disk can be loaded into memory by the CPU for GPU processing, without PCIe overhead.

6 Implementation

We integrate our G-TADOC into the CompressDirect (CD) [4] 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. Moreover, we have integrated the optimizations for embedded GPU systems.

7 Evaluation

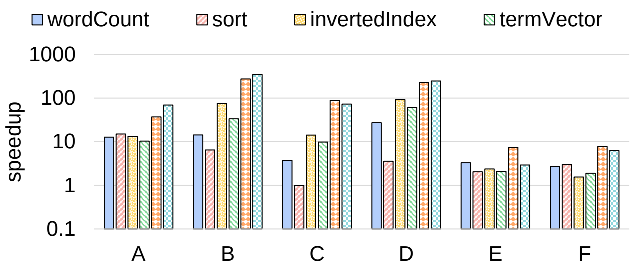
We measure the performance of G-TADOC and compare it with TADOC [4] for evaluation in this section.

7.1 Experimental Setup

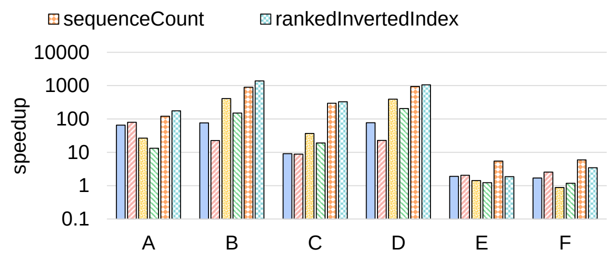
We show our experimental setup in this part.

TABLE 2
Platform configuration.

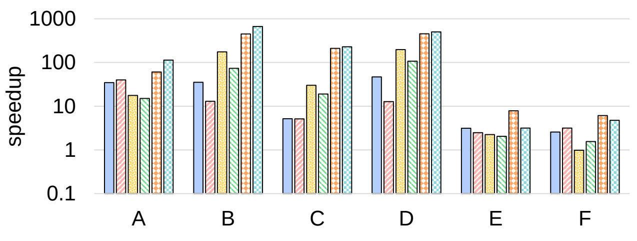
Platform	Jetson AGX Xavier	Pascal	Volta	Turing	10-node cluster
GPU	Volta GPU	GTX 1080	V100	RTX 2080 Ti	NULL
GPU Memory	LPDDR4x	GDDR5X	HBM2	GDDR6	DDR3
GPU TFLOPS	11 (FP16)	8.873	13.45	14	NULL
CPU	Carmel ARM v8.2	i7-7700K	E5-2670	i9-9900K	E5-2676v3
CPU TFLOPS	0.029	0.115	0.238	0.256	0.230
OS	Ubuntu 18.04.4	Ubuntu 16.04.4	Ubuntu 16.04.4	Ubuntu 18.04.5	Ubuntu 16.04.1
Compiler	CUDA 10.2	CUDA 8	CUDA 10.1	CUDA 11.0	GCC 5.4.0



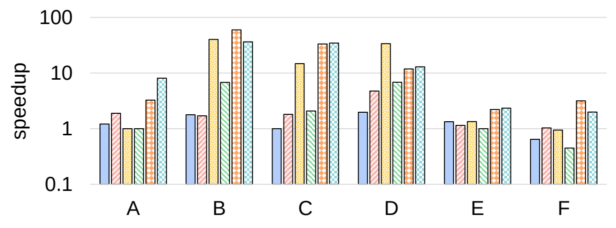
(a) Pascal (GeForce GTX 1080).



(b) Volta (Tesla V100).



(c) Turing (GeForce RTX 2080 Ti).



(d) Jetson AGX Xavier.

Fig. 13. Performance speedups.

Methodology. We compare our method with the baseline, TADOC [4]. TADOC is the state-of-the-art data analytics directly on compression, which is denoted as "TADOC". Our method that enables TADOC on GPUs is denoted as "G-TADOC". In our evaluation, we measure TADOC [4] performance and G-TADOC performance for comparison. Moreover, we assume that small datasets can be stored and processed in memory directly and large datasets are stored on disk.

Platforms. Four GPU platforms, including an embedded GPU platform Jetson AGX Xavier and three discrete GPU platforms, and a 10-node Amazon EC2 cluster are used in our evaluation, as shown in Table 2. 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] used to evaluate TADOC against large datasets.

Datasets. The datasets used in our evaluation are shown in Table 3, which include various real-world workloads. These datasets are widely used in previous research [3], [4], [5], [6], [8]. 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 [10]. To increase the diversity of test data, dataset C is COVID-19 data from Yelp [18], and dataset D is a collection of DBLP web documents [19]. Datasets E and F are large Wikipedia datasets [10], which are evaluated on the 10-node cluster.

TABLE 3
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	62MB	1	36,882	240,552
D	2.9GB	1	8,821,630	23,959,913
E	50GB	109	57,394,616	99,239,057
F	150GB	309	160,891,324	102,552,660

7.2 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 [4] in six datasets in Figure 13. In detail, Figure 13 (a) shows the speedups on Pascal platform, Figure 13 (b) shows the speedups on Volta platform, Figure 13 (c) shows the speedups on Turing platform, and Figure 13 (d) shows the speedups on the embedded GPU platform. We have the following observations.

First, G-TADOC achieves significant performance speedups over TADOC in all cases. On average, G-TADOC achieves 13.2× 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 performance benefit of G-TADOC on the embedded GPU platform is lower than that on the discrete platforms. The reason is that the hardware configuration of the embedded platform is closer to the CPU host part of the discrete platforms. For example, the memory used on the embedded GPU platform is DDR4, while the discrete GPUs use more advanced types of memory, such as DDR6 and HBM2. Moreover, the embedded GPU has much fewer GPU cores, so it has relatively low computing capacity.

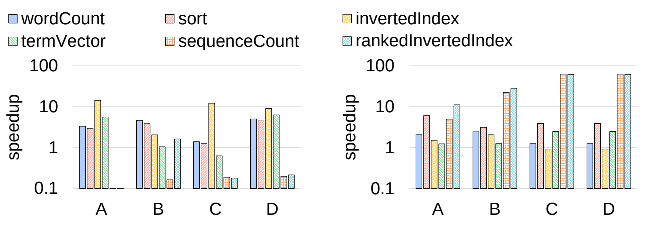
Third, the speedups of G-TADOC over TADOC on single nodes for processing small datasets are higher than the speedups of G-TADOC over TADOC on clusters for processing large datasets. The average speedup of G-TADOC over TADOC on a single node is 32.5×, while the average speedup of G-TADOC over TADOC on the ten-node cluster is 2.2×. The reason is that when processing large datasets, we need to consider the large IO time as well. Moreover, TADOC adopts coarse-grained parallelism in distributed environments to improve the data processing efficiency. However, because of the high performance of GPUs and our fine-grained parallelism, G-TADOC still achieves clear performance advantages.

Fourth, 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 116.9× and 145.8× on the small datasets, which are much higher than the full range average speedup. The reason is that *sequence count* and *ranked inverted index* of TADOC in [4] is of low performance: as described in [4], 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.

7.3 Optimization Analysis

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

Speedups in different phases. To analyze the performance benefits of G-TADOC in different phases, we show the separate speedups for different phases on Jetson AGX Xavier over the CPU in Figure 14. We do not show the results for the large datasets because the baseline is conducted on the Spark cluster, which involves IO interactions that cannot be simply divided into the two phases. We have the following findings. First, G-TADOC achieves clear performance benefits in both phases on the embedded GPU platform. In the first phase, the average speedup is 1.4×, and in the second phase, the average speedup is 4.7×. Second, the performance speedups in different phases have various behavior. For example, for *sequence count* and *ranked inverted index*, their performance benefits mainly come from the second phase. Third, compared to [8], the speedup on the embedded GPU platform is limited. The reason relates to its architecture design. As shown in Table 2, Jetson AGX Xavier uses DDR4 memory, which is much slower than those of the other discrete GPU platforms. Moreover, the embedded GPU is also less powerful than the discrete GPUs.



(a) Phase 1: initialization. (b) Phase 2: traversal.

Fig. 14. Separate speedups for different phases on Jetson AGX Xavier.

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 on the Pascal platform, 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 4.2, which can help select the optimal traversal strategy.

7.4 Detailed Analysis

We further explore the hardware metrics of G-TADOC for detailed performance analysis. Moreover, we analyze the benefits of G-TADOC from the performance-per-power and performance-per-cost perspectives.

Performance profiling. We use the Nvidia performance analysis tool, *nvprof*, to analyze the *DRAM throughput* and *achieved occupancy* of G-TADOC on the embedded GPU and Pascal platforms, as shown in Figure 15. DRAM throughput represents the sum of read and write throughputs of GPU memory, while achieved occupancy represents the average active warp ratio. The embedded GPU platform does not provide DRAM hardware counters, so we only show the DRAM throughput on GTX 1080 in Figure 15 (a). Figure 15 shows that most applications exhibit good metrics, and the embedded GPU has been better utilized than the discrete GPUs. For *inverted index* and *term vector* in dataset A, because of the large number of small files, G-TADOC utilizes a bottom-up traversal strategy. However, there still exist tremendous dependencies in DAG traversal, and hence they have relatively low performance metrics. For dataset C, due to its small input size, the GPU DARAM cannot be fully utilized. However, even in this case, G-TADOC still achieves clear performance benefits.

Energy efficiency. Power is an important consideration in big data management systems. We explore the power benefits of G-TADOC by analyzing the performance-per-power ratio, which is defined as the performance per power of G-TADOC divided by the performance per power of TADOC. We show the performance-per-power ratio of the embedded GPU platform on the large datasets in Figure 16 (a) for illustration. The average performance-per-power ratio is 32.5, which is significant and shows high energy efficiency benefits of G-TADOC. The reason is that the G-TADOC performance on GPU is much higher than the TADOC performance on CPU and thus G-TADOC has much lower execution time, though the GPU has higher power than the CPU.

Cost effectiveness. Price is also an important factor in parallel and distributed systems. We analyze the performance-per-cost ratio to show the benefits of G-TADOC from the price perspective. The performance-per-cost ratio is defined as the performance-per-device price of G-TADOC divided by the performance-per-device price of TADOC. We show the results on the Pascal and embedded GPU platforms for the large datasets in Figure 16 (b) for illustration. On average, the performance-per-cost ratio is 2.6, which implies that G-TADOC exhibits significant performance-per-cost benefits, due to the high performance of G-TADOC.

Comparison with GPU-accelerated uncompressed analytics. In our evaluation for the six data analytics tasks with the six datasets, G-TADOC reaches 13.2$\times$ 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 of 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$\times$ speedup.

Applicability. G-TADOC has the same applicability as TADOC [6]. In general, G-TADOC targets the analytics tasks that can be expressed as a DAG traversal problem, which involves scanning the whole DAG.

7.5 Technical Contributions

We summarize our contributions in this section, and discuss the potentials of enabling efficient GPU-based text analytics without decompression.

First, we find that GPUs, including embedded 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, although the GPU memory is limited, our work can help put much larger content directly in GPU memory. This is extremely useful to embedded GPU systems. 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.

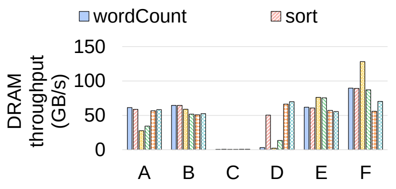
Third, the GPU platforms, especially the embedded GPU platforms, are both cost-effective and energy-efficient, which can be applied to a wide range of data analytics applications directly on compression, especially in large data centers. Moreover, experiments show that a GPU server can have much higher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

8 Related Work

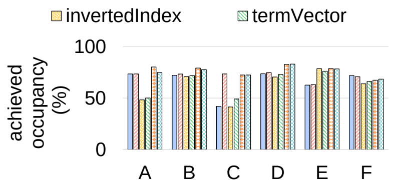
As far as we know, G-TADOC is the first work that enables efficient GPU-based text analytics without decompression. In this section, we show the related work of grammar compression, compression-based data analytics, GPU data analytics, and embedded GPU platforms.

Grammar compression. There are plenty of works on grammar compression [3], [4], [5], [6], [20], [21], [22], [23], [24], [25], [26], [27]. The closest work to G-TADOC is TADOC, which is the text analytics directly on compression in single-node and distributed environments [4]. TADOC extends Sequitur [28], [29], [30] as its compression algorithm for data analytics. After TADOC being proposed, Zhang *et al*. [3] proposed Zwift, which is the first TADOC programming framework, including a domain specific language, TADOC compiler and runtime, and a utility library. Then, Zhang *et al*. [6] applied TADOC as the storage to support advanced document analytics. Furthermore, Zhang *et al*. [5] 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.

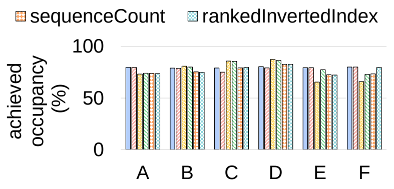
Index compression. The compression-based data analytics is an active research domain in recent years. However, typical approaches mainly use suffix trees and indexes [31], [32], [33], [34], [35], [36], [36], [37], [38], [39]. Suffix trees are traditional representations for data compression [31], [40] but incur huge memory usage [41], [42]. Suffix arrays [43]



(a) DRAM throughput on GTX 1080.

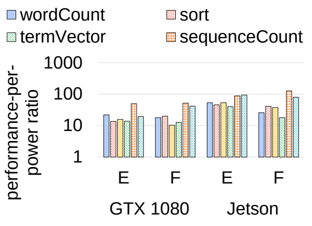


(b) Achieved occupancy on GTX 1080.

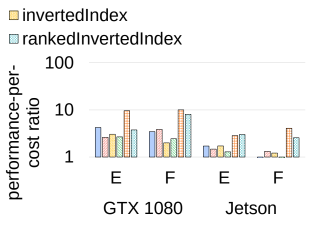


(c) Achieved occupancy on Jetson AGX Xavier.

Fig. 15. Analysis of performance metrics.



(a) Energy efficiency analysis.



(b) Cost effectiveness analysis.

Fig. 16. Analysis of energy efficiency and cost effectiveness.

and Burrows-Wheeler Transform [33], [34] are the development of these compression formats, but still generate high memory consumption [41]. Compressed suffix arrays [44], [45], [46], [47], [48] and FM-indexes [34], [49], [50], [51], [52] are more efficient than the previous compression techniques. Furthermore, Agarwal *et al.* proposed Succinct [32], which targets queries on compressed data. Moreover, there are many works about inverted index compression [53], [54], [55], [56], [57], [58], [59]. For example, Petri and Moffat [53] 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. For example, MapD (Massively Parallel Database) [60] is a popular big data analytics platform powered by GPUs. Most current analytics frameworks, such as Spark, have supported GPUs [61]. SABER [62] is a stream system that schedules queries on both CPUs and GPUs, and Zhang *et al.* [11] further developed FineStream, which enables fine-grained stream analytics on CPU-GPU integrated architectures. Gunrock [63] is an efficient graph library for graph analytics on GPUs, and for large graphs, multi-GPU graph analytics have been explored [64]. For machine learning data analytics, parallel technologies have been extensively applied to various aspects, especially for deep learning applications [65]. Currently, most machine learning frameworks, such as TensorFlow [66], support GPU.

Embedded GPU platform. Embedded GPU systems are popular in recent days, due to their features of low power and low cost. A large amount of literature has investigated the utilization of embedded GPU systems. Davidson *et al.* [67] analyzed the performance of their error resilient image processing application on a low power embedded GPU platform. Ukidave *et al.* [68] evaluated the performance of the unified memory structure of Nvidia TK1, as well as the power-performance ratio and energy effectiveness. Mittal [69] provided a survey for works that evaluate and optimize neural network applications on Jetson platforms. Lee *et al.* [70] used AlexNet on the Jetson TX1 board to recognize the license plate number of vehicles effectively. Rungsuptaweekoon *et al.* [71] evaluated the power efficiency of image recognition on Jetson TX2 with YOLO algorithm, as well as its throughput. Amert *et al.* [72] showed the important aspects of Nvidia TX2's GPU in fully autonomous vehicles' scheduling through experiments. Jose *et al.* [73] developed an intelligent multicamera face recognition based surveillance system on Jetson TX2.

9 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 13.2× speedup on average compared to the state-of-the-art TADOC. In addition, experiments show that G-TADOC achieves 2.6× performance-per-cost benefits and 32.5× energy efficiency over TADOC.

Acknowledgment

This work is supported by

References

[1] "JETSON XAVIER NX," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/, 2020.

[2] "Jetson AGX Xavier Developer Kit," https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit, 2020.

[3] F. Zhang, J. Zhai *et al.*, "Zwift: A Programming Framework for High Performance Text Analytics on Compressed Data," in *ICS*, 2018.

[4] F. Zhang, J. Zhai *et al.*, "Efficient document analytics on compressed data: Method, challenges, algorithms, insights," *PVLDB*, 2018.

[5] F. Zhang, J. Zhai *et al.*, "Enabling efficient random access to hierarchically-compressed data," in *ICDE*, 2020.

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

[7] 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.

[8] F. Zhang, Z. Pan *et al.*, "G-TADOC: Enabling Efficient GPU-Based Text Analytics without Decompression," in *ICDE*, 2021.

[9] K. Zhou, C. Fu, and S. Yang, "Big data driven smart energy management: From big data to big insights," *Renewable and Sustainable Energy Reviews*, vol. 56, pp. 215–225, 2016.

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

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

[12] F. Zhang, C. Zhang *et al.*, "Fine-grained multi-query stream processing on integrated architectures," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[13] D. Guide, "CUDA C programming guide," *NVIDIA, July*, 2013.

[14] F. Zhang, J. Zhai *et al.*, "Understanding co-running behaviors on integrated CPU/GPU architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 905–918, 2016.

[15] L. Chen, X. Huo, and G. Agrawal, "Accelerating MapReduce on a coupled CPU-GPU architecture," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–11.

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

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

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

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

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

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

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

[35] 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.

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

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

[38] 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.

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

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

[41] 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.

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

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

[44] 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.

[45] 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.

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

[47] 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.

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

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

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

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

[52] 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.

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

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

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

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

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

[58] 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.

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

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

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

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

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

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

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

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

[67] R. L. Davidson and C. P. Bridges, "Error resilient GPU accelerated image processing for space applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 9, pp. 1990–2003, 2018.

[68] Y. Ukidave, D. Kaeli *et al.*, "Performance of the NVIDIA Jetson TK1 in HPC," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 533–534.

[69] S. Mittal, "A Survey on optimized implementation of deep learning models on the NVIDIA Jetson platform," *Journal of Systems Architecture*, vol. 97, pp. 428–442, 2019.

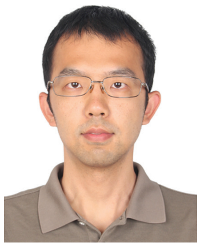
[70] S. Lee, K. Son *et al.*, "Car plate recognition based on CNN using embedded system with GPU," in *2017 10th International Conference on Human System Interactions (HSI)*. IEEE, 2017, pp. 239–241.

[71] K. Rungsuptaweekoon, V. Visoottiviseth, and R. Takano, "Evaluating the power efficiency of deep learning inference on embedded gpu systems," in *2017 2nd International Conference on Information Technology (INCIT)*. IEEE, 2017, pp. 1–5.

[72] T. Amert, N. Otterness *et al.*, "GPU scheduling on the NVIDIA TX2: Hidden details revealed," in *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2017, pp. 104–115.

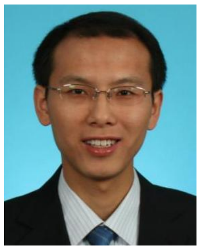
[73] E. Jose, M. Greeshma *et al.*, "Face recognition based surveillance system using facenet and mtcnn on jetson tx2," in *2019 5th International Conference on Advanced Computing & Communication Systems (ICACCS)*. IEEE, 2019, pp. 608–613.

Zaifeng Pan received the bachelor degree from Shanghai Jiao Tong University in 2021, and he is pursuing the master degree in computer science at Renmin University of China, under the supervision of prof. Feng Zhang. His current research interests include parallel computing and heterogeneous computing.



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

Yanliang Zhou received the undergraduate degree from School of Information, Renmin University of China. He joined the Key Laboratory of Data Engineering and Knowledge Engineer (MOE) in 2019, and now works as a research assistant. His current research interests include high performance computing and parallel accelerating.



Jidong Zhai received the BS degree in computer science from University of Electronic Science and Technology of China in 2003, and PhD degree in computer science from Tsinghua University in 2010. He is an associate professor in 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.

Xipeng Shen received the PhD degree in computer science from the University of Rochester, in 2006. He is a professor in Computer Science at the North Carolina State University. He is a receipt of the DOE Early Career Award, NSF CAREER Award, Google Faculty Research Award, and IBM CAS Faculty fellow Award. He is an ACM distinguished member, ACM distinguished speaker, and a senior member of the IEEE. His interest is in Programming Systems and Machine Learning.

Onur Mutlu Onur Mutlu received the BS degrees in computer engineering and psychology from the University of Michigan, Ann Arbor, and the MS and PhD degrees in ECE from the University of Texas at Austin. He is a professor of computer science at ETH Zurich. He is also a faculty member with Carnegie Mellon University, where he previously held the Strecker Early Career Professorship. His current broader research interests include computer architecture, systems, hardware security, and bioinformatics.

Xiaoyong Du obtained the B.S. degree from Hangzhou University, Zhengjiang, China, in 1983, the M.E. degree from Renmin University of China, Beijing, China, in 1988, and the Ph.D. degree from 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.