Liberator: A Data Reuse Framework for Out-of-Memory Graph Computing on GPUs

Shiyang Li, Ruiqi Tang, Jingyu Zhu, Ziyi Zhao, Xiaoli Gong, Wenwen Wang, Member, IEEE, Jin Zhang, and Pen-Chung Yew, Fellow, IEEE

Abstract—Graph analytics are widely used including recommender systems, scientific computing, and data mining. Meanwhile, GPU has become the major accelerator for such applications. However, the graph size increases rapidly and often exceeds the GPU memory, incurring severe performance degradation due to frequent data transfers between the main memory and GPUs. To relieve this problem, we focus on the utilization of data in GPUs by taking advantage of the data reuse across iterations. In our studies, we deeply analyze the memory access patterns of graph applications at different granularities. We have found that the memory footprint is accessed with a roughly sequential scan without a hotspot, which infers an extremely long reuse distance. Based on our observation, we propose a novel framework, called Liberator, to exploit the data reuse within GPU memory. In Liberator, GPU memory is reserved for the data potentially accessed across iterations to avoid excessive data transfer between the main memory and GPUs. For the data not existing in GPU memory, a Merged and Aligned memory access manner is employed to improve the transmission efficiency. We also further optimize the framework by parallel processing of data in GPU memory and data in the main memory. We have implemented a prototype of the Liberator framework and conducted a series of experiments on performance evaluation. The experimental results show that Liberator can significantly reduce the data transfer overhead, which achieves an average of 2.7x speedup over a state-of-the-art approach.

Index Terms—Graph Computing, GPU memory oversubscription, Partition-based method, Data Reuse, Zero-copy

1 INTRODUCTION

Graph analytics has become one of the major workloads in data centers, such as genome analysis [1], recommender systems [2], social network analysis [3], graph databases [4], etc. In these applications, graphs are often used to represent relationships between entries or certain states, and the analysis of graphs tends to be highly intensive in data access because the size of graph data grows dramatically. Generally, there are millions of nodes and even hundreds of millions of edges in the graph in the real production environment [5]. Meanwhile, the access patterns of graph algorithms are usually very irregular, which makes it a challenging task for traditional computation platforms.

In recent years, GPUs have been widely used to accelerate graph computation due to their turbo performance. However, as the size of graph data continues to grow, the limited memory resources of GPUs greatly limit their effectiveness in graph computing. A recent survey [5] conducted that a wide of domains use large graph data whose size is tens or even hundreds of gigabytes. But the GPUs commonly used in data centers generally have only a dozen or tens of gigabytes of memory. For example, a maximum of 16GB for NVIDIA P100 [6], 32GB for NVIDIA V100 [7], and 80GB for NVIDIA A100 [8]. Consequently, the performance of graph computing on GPUs is often confined due to the excessive data operations filling the working set into GPU memory.

A straightforward solution is graph partitioning, which divides the graph data into subsets fitting the GPU memory, and loads the subset when requested. As shown in Figure 1, the example graph shown at the bottom has been divided into two partitions before being processed: $Pa$ and $Pb$. In most graph analytic applications, graph data are often traversed many times [9], and data partitions are brought into GPUs when it is needed, such as shown at the top: $Pa \rightarrow Pb \rightarrow Pb \rightarrow Pb \rightarrow Pa$.

Improvements to the partition-based method are feasible by enhancing data utilization. Redundant data movements exist in a single transfer. For example, in the second transfer of iteration 1, only $V7$ is needed but the entire subgraph $Pb$ is transferred. Furthermore, data reuse may occur across iterations. The 2nd, 3rd, and 4th transfers use the same subgraph $Pb$, and it is possible to avoid redundant data transfer if the GPU can exploit the data left in the previous iteration.

We also prove our idea with a real workload. By running PageRank [10] with friendster-konect [11] on a GPU with 11GB GPU memory. It takes 43 iterations and the data transfer from main memory to GPU is about 1,306GB while the dataset is only about 11GB. This reveals the shortfalls of data management in partition-based approaches, i.e. the
Recently, Unified Virtual Memory (UVM) is provided as a new technology to extend the memory size of GPUs with main memory [12], [13], [14], [15]. The data in the main memory requested by GPU is loaded on demand via Page Fault with the granularity of a page, i.e., 4K bytes. In fact, it can be considered as a variant of partition-based schemes where each partition is a page.

In our studies, we find that in graph computing the data can be reused frequently across iterations. Also, the redundant data in a certain transfer might be reused in the next iterations. For example, in Figure 1, the second transfer of iteration 1 transfers $P_b$, where $V_4$, $V_5$, and $V_6$ are the redundant data in this transfer. But they can be reused in the next two iterations. We run PageRank with gsh-2015-host [16] in the UVM scheme. It runs 62 iterations and every active vertex will be visited an average of 26 times, which means in graph computing data is reused quite frequently.

We also find that there is a roughly linear scan of sparse accesses on a large dataset for every iteration in graph computing. Therefore, it is an extremely long reuse distance without obvious hot regions. It is challenging to exploit data reuse for traditional data management mechanisms, such as LRU in UVM [17].

In this work, we proposed a novel data management strategy in GPU memory to utilize the discovered data-accessing behavior of graph computing. We reserve some data in the GPU memory, called Static Data, for data reuse across iterations. For the requested data missing from Static Data, called On-demand Data, it is loaded from the main memory with minimal redundancy. Based on the general idea, the GPU memory region is reserved and managed as a data cache for graph computing. To make this approach effective and avoid data thrashing and redundant data transfer, we need to address the following challenges:

1. We need an efficient way to look up the Static Data and determine cache hit or not.
2. We need to reduce the overhead of On-demand data transmission, which is considered as the penalty of Static Data cache misses.
3. We need to exploit the design space of prefetch and replacement strategy for the Static Data since there can be potential to further improve the performance.

In this paper, we propose an efficient graph framework called Liberator. It automatically fills up GPU memory with the data for cross-iterations data efficiency. A bitmap called Static Map is set as a cache look-up table, which can be retrieved with AND or XOR operations. When cache miss happens, the requested graph data is accessed with a zero-copy mechanism to minimize data transfer and avoid thrashing. The Liberator also improves the performance by parallel processing of Static Data and On-demand Data. In summary, we have made the following contributions:

1. We explore the potential of data reuse in large graph computing workloads that usually exceed the GPU memory size. We provide a comprehensive analysis of the access patterns of graph analytic applications and addressed the limitations of conventional locality-based memory management approaches.
2. We propose Liberator, a novel graph computing framework. Liberator exploits the temporal locality across iterations and minimizes data transfers. It also improves GPU efficiency by maximizing CPU-GPU concurrency and PCIe bandwidth.
3. We have implemented a prototype of Liberator with CUDA. Comprehensive experiments have been conducted and the results show that Liberator can achieve 5.1x speedup in the best case and 2.7x speedup on average over a state-of-the-art graph processing approach.

The rest of this paper is organized as follows. Section 2 provides some background in graph computing and motivates our proposed approach. Section 3 describes some design and implementation details of our approach. Section 4 presents the experimental setup and evaluation analysis. Section 5 discusses the related work. Section 6 concludes the paper.
2 BACKGROUND AND MOTIVATION

2.1 Access Pattern Analysis of Graph Processing

It is intuitive that the memory access pattern of graph processing is irregular. Traditionally, the adjacency matrix is used to store graph data, while the graph data is normally sparse, leading to sparse adjacency matrices. To improve the data efficiency, previous research has proposed to the CSR format [18] to compress the graph data. An example graph G(8,9) is shown in Figure 2(a) and the CSR format of it is shown in Figure 2(b). CSR format uses two arrays to represent a graph, VertexList and EdgeList. The EdgeList stores the neighbors of each node sequentially starting from vertex 0, while the VertexList stores the starting position of each node’s neighbor list in the edge array, indexed by the vertex ID, as shown in the bottom part of Figure 2(b).

In large-scale graph computing applications, the number of edges can be incredibly high compared to the limited number of vertices. For instance, in the Friendster-konect dataset, there are about 2.4 billion edges occupying approximately 11G bytes of memory, while the vertices account for only 65.1 million, taking up less than 1G bytes. Consequently, when using GPUs to accelerate such applications, it is typical to keep the vertices in GPU memory while transferring the EdgeList between the GPU and the main memory consumes most of the effort. This paper primarily focuses on the access pattern of EdgeList.

The access patterns to graph data are generally characterized by significant irregularity when using the adjacency matrix representation. However, this is not the case when using CSR. During the preprocess of CSR, the vertices are organized in sequential order, and edges connecting to the vertices are also sequential. In one iteration of graph processing, if a thread accesses the VertexList sequentially, the corresponding EdgeList is also retrieved sequentially.

When using the GPU to accelerate graph processing, multiple threads are created according to the cores provided by the GPU hardware, and the graph data is divided among the threads to maximize the benefit of parallelism. To optimize the load balance and thread management, a common approach is to divide the VertexList according to the number of available threads.

Figure 3 illustrates an example of graph traversal on GPUs. There are 3 threads in the example, and the threads handle the active elements (colorful items in the list) in the VertexList in a loop manner. The VertexList is accessed sequentially with a step size relative to the thread numbers, and in each loop iteration, each thread picks up an element in VertexList based on the order of its threadID. And then, each thread accesses the EdgeList based on the index in the VertexList element. As the EdgeList is organized sequentially according to the VertexList, and the EdgeList is accessed roughly sequentially. It is evident that there are inactive elements in the VertexList (the white item in the example), which makes the VertexList not accessed exactly continuously sequential. Accordingly, the access to EdgeList is not continuously sequential. However, the memory area accessed by the second loop must be after the area accessed by the first loop, which makes the access pattern to the EdgeList a roughly sequential scan.

To observe the ID of the element in Edgelist among the steps of loops in one iteration of graph traversal, we present the results in Figure 4. Notably, this observation does not imply a significant memory access pattern for graph processing, which is counterintuitive. In fact, when considering the data stored in the accessed elements of the EdgeList, specifically the IDs of vertices connecting to the active vertices in the current iteration, the results have little to do with the discipline, as shown in Figure 5. By focusing solely on the access of EdgeList, the results presented in Figure 4 indicate rare data reuse on the large footprint in one iteration of graph traversal.

We conducted experiments to analyze the access pattern across iterations. To this end, we retained all vertices in GPU memory and placed the Edgelist in UVM. We divided the EdgeList into large chunks, where each data chunk consisted of 4 million edges, equivalent to about 15MB, and monitored the edge access by tracing page migration using nvprof [19]. We recorded the order in which each data chunk was accessed over time and illustrated the results in Figure 6. Our findings reveal that there is considerable overlap between iterations, indicating the potential for data reuse. Furthermore, we observed no significant hot spots in any of the iterations.

According to our observations, there is a similar access pattern of the EdgeList among different graph traversal iterations. This presents an opportunity to optimize data utilization. However, in each iteration, there is a nearly sequential scan of the vast Edgelist without any identifiable hot spots, rendering conventional data management strategies such as LRU ineffective in capturing reusability. In our proposed Liberator strategy, we aim to exploit data
reuse across iterations by designating a memory region on the GPU for EdgeList and avoiding thrashing through conservative replacement.

### 2.2 Partitioning-Based Graph Processing on GPUs

Graph partitioning is a traditional solution for out-of-memory graph processing. The graph is divided into partitions, i.e., sub-graphs, which can fit into GPU memory. The sub-graphs are processed in turns and the data required in the processing procedure are transferred into GPU memory before being accessed. There are many schemes, such as GraphReduce [20] and Graphie [21] that use this approach to allow efficient out-of-memory graph processing on GPUs.

However, due to the sparse access in each iteration, lots of data movement is redundant. Table 1 shows the percentage of active edges processed in each iteration using four graph traversal algorithms, breadth-first search (BFS), single source shortest path (SSSP), connected component (CC), and pagerank (PR) on the friendster-konect [11] and uk-2007-04 [22] datasets. We keep all edge data in UVM and record all the active edges of every iteration. It shows that the active edges in each iteration only account for a tiny fraction of the total edges, even below 5% in some cases.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>BFS</th>
<th>SSSP</th>
<th>CC</th>
<th>PR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Friendster-konect</td>
<td>4.3%</td>
<td>3.1%</td>
<td>14.1%</td>
<td>28.7%</td>
</tr>
<tr>
<td>UK-2007-04</td>
<td>0.8%</td>
<td>3.1%</td>
<td>3.0%</td>
<td>25.1%</td>
</tr>
</tbody>
</table>

Unified Virtual Memory (UVM) was introduced by NVIDIA in Pascal architecture [6]. It provides an easier programming interface to deal with memory oversubscription. A unified memory address space is provided including both GPU memory and the main memory. It allows pages to be migrated between the main memory and GPU memory transparently with a demand paging mechanism.

The approach based on UVM can be viewed as a variation of the partitioning-based method, where each page serves as a partition. Instead of using explicit swap opera-
tions, data to be accessed is migrated to GPU memory on demand. However, the handling of page faults can cause considerable overheads [17]. Furthermore, page alignment may exacerbate the irregularity and sparsity of memory access, thereby aggravating the situation.

2.3 Fine-grained Memory Management

A novel GPU memory management scheme named Subway [23] has been proposed recently to reduce the amount of data transferred between CPU and GPU by more accurately selecting the required data to be transferred. In the proposed scheme, the vertexes are stored in both the main memory and GPU memory, while the edges are located in the main memory. There are 3 steps in each iteration:

(a) GPUs are employed to locate a fine-grained sub-graph required for the current iteration and send it to the CPU in a compact format.
(b) CPU picks up the required edge data from the main memory with multiple threads. After that, the data is organized in CSR [18] compacted format and transferred to GPU.
(c) GPU performs the graph processing on the transferred data for this iteration.

Obviously, there is no redundant edge transfer in every iteration, which significantly improves the data communication between GPU and the main memory. However, there are still several limitations. Firstly, these steps have to be processed sequentially. For example, the GPU has to wait for the data from CPU in each iteration. Second, the GPU memory is not fully utilized as the active edges are only a limited proportion of GPU memory. We have re-implemented the Subway platform on a workstation equipped with 128GB memory and an NVidia P100 with 16GB memory. We have evaluated the framework with breadth-first search (BFS), single source shortest path (SSSP), connected component (CC), and pagerank (PR) on the friendster-konect [11] and uk-2007-04 [22] datasets.

As shown in Table 2, we measured the average GPU memory usage in each iteration by recording the size of all the subgraphs transferred to GPU memory. It can be seen that in most cases, on average only 8% of GPU memory (total 16GB) is utilized in each iteration. We also set two timers to record the time using in CPU and GPU separately. Our study shows that on average 54% of GPU time is idle in the four algorithms on Friendster-konect dataset.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>BFS</th>
<th>SSSP</th>
<th>CC</th>
<th>PR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Friendster-konect</td>
<td>0.45GB</td>
<td>0.64GB</td>
<td>1.64GB</td>
<td>2.97GB</td>
</tr>
<tr>
<td>UK-2007-04</td>
<td>0.11GB</td>
<td>0.94GB</td>
<td>0.46GB</td>
<td>3.80GB</td>
</tr>
</tbody>
</table>

Last but not the least, Subway does not exploit potential temporal locality across iterations. Subway discards the data after each iteration and re-organizes a new subgraph even if it is similar to the subgraph in the previous iterations. As we have observed in Section 2.1, there are lots of overlaps between the data required across iterations, while it is possible to further improve the data efficiency.

2.4 Zero-copy memory access between GPU and the main memory

The GPUs can only access the internal memory in the traditional architecture before Pascal is announced [6]. Therefore, the data requested in the graph processing has to be copied into GPU memory via PCIe buses before being accessed, just as the mechanism employed in partitioning-based solutions [21] [20].

Since the CUDA Toolkit version 2.2 was released by NVIDIA, the programmers are allowed to access the external memory, i.e. the main memory, without page migration, which is also named as zero-copy memory access in CUDA [24].

In this way, the address space can be significantly extended. However, there is a non-negligible overhead when touching the address space mapped from the main memory. Further investigation shows that the main reason for such a high overhead is due to the long latency and low bandwidth utilization of PCIe bus [25]. To further improve the performance of zero-copy memory access, it has been proposed that the access should be wrapped and aligned [26]. In EMOGI [26], it has been shown that when GPU threads from the same warp access the zero-copy memory in a 128-byte aligned manner, the bandwidth of zero-copy memory is almost the same as explicitly transferring data via cudamemcpyp.

As there is rarely data reuse in one iteration shown in Section 2.1, which means the data copied into GPU memory is accessed only once, it is not a wise choice to migrate the data into GPU memory. On the other hand, there is an opportunity to fully utilize the bandwidth if the program access zero-copy memory with a Merged and Aligned scheme. In the proposed Liberator framework, we try to optimize the out-of-GPU-memory access with this scheme to further improve the performance.

3 Design of Liberator

3.1 Design Overview

Based on the above observations, we propose a new framework called Liberator. In Liberator, we use a push-based vertex-centric model by keeping all vertices in the GPU memory. For the edge data, which exceeds the GPU memory capacity, Liberator fills up GPU memory with potentially reusable data across iterations, called Static Data, and for other required data that are not in GPU memory, called On-demand Data, it will be loaded on-demand via zero-copy memory. Liberator further improves the parallelism between CPU and GPU by overlapping the data transfer and GPU computation. Figure 7 shows the workflow of Liberator.

In each iteration, the vertices that need to be accessed are identified and marked as Active in GPU. The edges connected to these active vertices are then loaded into the GPU. To efficiently locate the edges in the Static Data region, we create a bitmap index called StaticBitmap that indicates the present state of each vertex. If all edges connected to a vertex are present in the Static Data region, the corresponding bit in StaticBitmap is set to 1; otherwise, it is set to 0. Similarly, we create a bitmap index called ActiveBitmap that marks the active vertices in the current iteration. We then use
as an AND operation to obtain the StaticMap, which represents the edges present in GPU memory. To locate the requested data located in the main memory, we use an XOR operation to obtain the OndemandMap. Although the bitmap can be large, hundreds of megabytes in size, these operations can be conducted in parallel, making them highly suitable for the GPU architecture.

After locating the data, Liberator conducts graph processing computation on the edge data separately in parallel. For the data indexed by StaticMap, as the vertices and edges are located in GPU memory, using the traditional vertex-to-thread computing mechanism [18], and we majorly focus on the OndemandMap part.

As the data indexed by OndemandMap is located in the main memory, they have to be transferred into GPU via PCIe. Instead of copying them to GPU memory, Liberator accesses the requested data via the zero-copy mechanism. As there is rarely data reuse in one iteration, it is critical to fully utilize the PCIe bandwidth during the memory access. In Liberator, the data request is reorganized and issued in a Merged and Aligned manner proposed in EMOGI [26], which can maximize the PCIe bandwidth utilization. More detail is explained in Section 3.3.

Taking Breadth-First Search (BFS) as an example, as shown in Figure 8. Initially, the CPU fills up the GPU memory with part of edge data as Static Data. Upon the start of graph processing, the CPU notifies the GPU that the source vertex is active in the first iteration, and the GPU sets the index of the source vertex in the ActiveBitmap to 1. Subsequently, parallel AND and XOR operations between StaticBitmap and ActiveBitmap are performed to generate StaticMap and OndemandMap. If the index of the source vertex in StaticMap is 1, then all its edges are already in the GPU memory, and processing can begin immediately. In contrast, if the index of the source vertex in OndemandMap is 1, it implies that the edges must be transferred via PCIe. During each iteration, all adjacency vertices of the currently visited vertex are marked as 1 in ActiveBitmap, and the indices of visited vertices are set to 0. This prepares the ActiveBitmap for the next iteration. The above process is iterated until there are no more elements with a value of 1 in the ActiveBitmap, indicating that the algorithm has converged.

### 3.2 Parallel Processing of Data in GPU memory and the Main Memory

As presented in Section 2.3, GPU is the major provider of computing resources, while it has to wait for the CPU to prepare and transfer the data and thus can incur serious overhead. In Liberator, as the data is always available to GPUs, it provides us with an opportunity to improve the performance with parallelism. Since parts of the data are in the GPU memory, i.e. Static Data, while the other parts are in the zero-copy memory area, i.e. OndemandData, we use two CUDA streams to process them separately. As there is no conflict between the computation of the Static Data and all the works in the OndemandData, they can be executed concurrently.

Each vertex is handled with a GPU thread for the edges available in GPU memory to exploit the turbo performance of parallel computing and HBM memory in GPUs. For the edges in the zero-copy area, we assign a warp to each vertex for proper bandwidth utilization, which is explained in Section 3.3. However, this scheme will launch a large number of light computing tasks, which far exceed the available SM and cores in GPU.
There have been many studies on how to perform effective SM scheduling when GPU resources are limited [27], [28], [29]. Due to most of the computing tasks in graph computing being short tasks, we have tried four different lightweight scheduling approaches to achieve the parallelism effect shown in Figure 9. The performance of different scheduling approaches are discussed in Section 4.3. Eventually, we choose to launch the Static Data computing kernel and On-demand Data computing kernel at the same time, to avoid the busy wait between two computing kernels, i.e., the GatherData and Transfer time in the Baseline or Ascetic scheme.

Fig. 9. The final parallelism effects in Liberator. Two kernels are put in two different CUDA streams and then launched at the same time. SM scheduling is totally controlled by GPU runtime automatically.

3.3 Merged and Aligned Access of On-demand Data

In the zero-copy mechanism provided by CUDA, access to the main memory issued by GPUs is conducted via PCIe. Based on the observation of EMOGI [26], the memory access to the zero-copy area should be extended to 128-byte aligned blocks to get the best performance and bandwidth utilization. As shown in Figure 10, the data of edges are organized sequentially based on the index of vertices. Therefore, the access to edges during the graph processing should be merged into 128-byte sequential blocks correspondingly, in order to maximize bandwidth utilization. In the multiple-thread programming model of GPUs, the edge data is divided among the threads for parallelism, with one thread accessing an edge in each step of the loop iteration. Thus, it is crucial to managing data assignment to GPU threads to ensure edge data access is conjunct and mergeable by the hardware.

Figure 10 shows the vertex-to-warp design employed in Liberator. It can be seen that the edges adjacent to a vertex are assigned to the available threads simultaneously, and therefore accessed in one iteration. As there are thousands of threads in the GPUs, and the warp is the minimal resource scheduling unit for the threads management on GPUs, we assign the edges adjacent to one vertex to one warp, which is named vertex-to-warp mapping. As there are generally 32 threads in one warp, and the edge data is normally a four-byte unsigned integer indicating the index number of the target vertex, it is straightforward to access the continuous 32 edges with the threads in the warp simultaneously for the best performance.

However, as there can be an arbitrary number of edges adjacent to one vertex, it is not always available to issue the memory request in an aligned manner as required. For the misaligned situations shown in Figure 11, a padding mechanism is employed to insert dummy access threads shown as the pink squares in Figure 11. These dummy threads(t0,t1,t2, and t32) fetched unnecessary data are then turned off in the subsequent procedure to avoid errors. Even though this increases the branch instructions in the CUDA program, it keeps the memory access in the Merged and Aligned manner which is critical to performance. Based on the survey of previous research [26], the average degree per vertex in large-scale graphs is about 71, which is enough to fill up the warp and the padding mechanism is not frequently required.

Inspired by the idea of EMOGI [26], we collect the vertices corresponding to the Ondemand Map and assign them to warps separately. As shown in the bottom part of Figure 7, the edge data correspondingly to these vertices are then accessed with the optimized memory access manner. Compared to the previously proposed solution in Ascetic [30], which copies the request data into GPU memory, the zero-copy-based access in Liberator avoids data pollution in GPU memory and also reduce the memory latency, which is similar to the idea of read-through cache [31].

3.4 Data pre-fetch and Replacement

For better utilization of GPU memory, we should pre-fetch those data that could be reused across iterations to fill up GPU memory. As presented in Section 2.1, most of the time the graph computing algorithms just scan the data set sequentially. Therefore, we use the one-shot strategy to fill up GPU memory. For example, if there is 10GB available GPU memory, we pre-fetch the first 10GB of the data set before iterations.

We also need to consider that some Static Data could be “stale” after some iterations. To keep the reusable data “fresh” in GPU memory, we gradually replace the “stale”
data when GPU is processing the On-demand Data. According to the access patterns, we divide the graph dataset into 16KB chunks, which are also amenable to the PCIe burst transfer mechanism. For each chunk, a counter is assigned to record the number of accesses in the previous iterations. If the counter is below a threshold, it means the chunk is stale. Different replacement policies can be implemented for different graph applications. For example, in BFS, CC, and SSSP can record the number of accesses in all of the past iterations to determine if the chunk is stale. While the counter in PageRank determines the status of a chunk by the number of accesses in the last iteration. A server thread handles the data replacement by evicting the stale data chunks in the Static Data and replacing them with new data chunks from the CPU memory. The related metadata will be updated accordingly after the new data chunks are swapped in.

4 EXPERIMENT AND EVALUATION

4.1 Experimental Setup and Methodologies

Extensive experiments have been conducted to evaluate the performance and design tradeoff of Liberator. The experiment platform configuration is shown in Table 3. All of the programs are compiled with -O3 optimization.

![Figure 11. The final implementation of Liberator's Merged and Aligned access manner. The On-demand Data are in blue squares, while the static data are in red squares. Four red threads which fetch padding data are turned off in computing.](image)

The graph datasets used in the experiments are all obtained from the publicly available online graph datasets taken from the real world [16], [32], [33], as shown in Table 4. The parentheses after the dataset name indicate a directed graph (labeled D) or an undirected graph (labeled U). The size shown in the table is calculated without weights of edges, and when the weights of edges are necessary (such as SSSP), it will double the CSR file size. In addition, for the PageRank algorithm, it is necessary to store the out-degree of each node additionally, and the dataset size will also be expanded. Since not all of the used data sets exceed the GPU memory size of our testbed platform, we have manually limited the memory that the programs can use to 10G in the experiment.

### TABLE 4
Dataset used in the experiment

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Name</th>
<th>Vertices</th>
<th>Edges</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>GS</td>
<td>gsh-2015-host(d) [16]</td>
<td>68660142</td>
<td>1802747600</td>
<td>14G</td>
</tr>
<tr>
<td>FK</td>
<td>friendster-konect(u) [32]</td>
<td>68349467</td>
<td>2586147869</td>
<td>11G</td>
</tr>
<tr>
<td>FS</td>
<td>friendster-snap(u) [33]</td>
<td>124836180</td>
<td>1806067135</td>
<td>15G</td>
</tr>
<tr>
<td>UK</td>
<td>uk-2007-04(d) [16]</td>
<td>10686791</td>
<td>790305474</td>
<td>15G</td>
</tr>
</tbody>
</table>

For comparison, we also implemented a traditional partition-based GPU graph processing system (PT) [20], a zero-copy based system EMOGI [26], a fine-grained memory management system SubWay [23], and a partitioning GPU memory system Ascetic [30], which is proposed in our previous research. The PT, SubWay, and Ascetic use multi-thread on the CPU side to accelerate the graph partitioning, subgraph generation, and on-demand data organizing respectively. EMOGI does not rely on the CPU during graph processing. Each program is run 10 times and we take the arithmetic mean in our evaluation.

4.2 Performance Analysis

Table 5 shows the overall performance results. For simplicity, we have normalized the execution time to the partition-based implementation (PT) [20]. It is notable that the open-source release of EMOGI can not achieve a correct result on some applications with some dataset (such as PageRank on GS and FS, CC on directed graphs) and we cannot fix the issues after our best effort. Therefore, we have to ignore these results. It can be seen that compared to PT, Liberator can achieve 165.4X speedup for the best case and 42.5X speedup on average. And compared to Ascetic, Liberator can achieve 2.7X speedup on average and 4.0X speedup in the best case.

### TABLE 5
Performance results. The values of SubWay, Ascetic, EMOGI, and Liberator are normalized to the value of PT. The highest number among them is in bold.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>PT</th>
<th>SubWay</th>
<th>Ascetic</th>
<th>EMOGI</th>
<th>Liberator</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSSP</td>
<td>GS</td>
<td>279.9s</td>
<td>9.4X</td>
<td>15.2X</td>
<td>12.8X</td>
</tr>
<tr>
<td></td>
<td>FK</td>
<td>145.2s</td>
<td>10.9X</td>
<td>10.9X</td>
<td>10.7X</td>
</tr>
<tr>
<td></td>
<td>FS</td>
<td>177.9s</td>
<td>6.5X</td>
<td>8.6X</td>
<td>7.3X</td>
</tr>
<tr>
<td></td>
<td>UK</td>
<td>595.41s</td>
<td>16.5X</td>
<td>23.7X</td>
<td>10.9X</td>
</tr>
<tr>
<td>PageRank</td>
<td>GS</td>
<td>249.1s</td>
<td>1.9X</td>
<td>2.5X</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>FK</td>
<td>97.9s</td>
<td>3.1X</td>
<td>4.3X</td>
<td>4.6X</td>
</tr>
<tr>
<td></td>
<td>FS</td>
<td>198.3s</td>
<td>2.8X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>UK</td>
<td>393.6s</td>
<td>4.6X</td>
<td>3.0X</td>
<td>-</td>
</tr>
<tr>
<td>BFS</td>
<td>GS</td>
<td>49.2s</td>
<td>39.8X</td>
<td>48.2X</td>
<td>58.8X</td>
</tr>
<tr>
<td></td>
<td>FK</td>
<td>59.2s</td>
<td>28.0X</td>
<td>39.5X</td>
<td>70.0X</td>
</tr>
<tr>
<td></td>
<td>FS</td>
<td>84.7s</td>
<td>15.2X</td>
<td>47.0X</td>
<td>77.0X</td>
</tr>
<tr>
<td></td>
<td>UK</td>
<td>281.2s</td>
<td>50.2X</td>
<td>59.8X</td>
<td>165.4X</td>
</tr>
<tr>
<td>CC</td>
<td>GS</td>
<td>40.5s</td>
<td>9.4X</td>
<td>10.7X</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>FK</td>
<td>36.4s</td>
<td>4.4X</td>
<td>9.5X</td>
<td>13.1X</td>
</tr>
<tr>
<td></td>
<td>FS</td>
<td>59.4s</td>
<td>10.4X</td>
<td>14.1X</td>
<td>33.0X</td>
</tr>
<tr>
<td></td>
<td>UK</td>
<td>595.4s</td>
<td>20.9X</td>
<td>-</td>
<td>58.4X</td>
</tr>
</tbody>
</table>

Table 6 summarizes the actual amount of explicit data transferred during the graph processing in each implementation. The results are normalized to the size of the original dataset. Note that they include data transferred during the initial filling of the Static Data. In certain scenarios,
the amount of data transfer could be below 1.0X, as the processing of certain algorithms such as BFS, SSSP, and CC does not necessitate the processing of the entire graph. This is primarily due to the possibility of the graph being incompletely connected and the application not requiring access to isolated vertices. As a result, it is unnecessary to transfer these vertices to the GPU. It can be seen that Liberator can reduce 88% of data transferred during graph processing compared to PT, 45% of that compared to SubWay, and 22% of that compared to Ascetic. EMOGI’s performance was poor in the data transfer experiments, which is attributed to its focus on enhancing PCIe bandwidth rather than reducing data transfer, and therefore, the comparison with other approaches may be considered unfair.

4.3 Breakdown of the Optimizations

The most significant performance improvement of Liberator compared to Ascetic [30] is the reduction in memory transfer thanks to the zero-copy mechanism. To gain a deeper understanding of this performance boost, we evaluated the memory savings for each iteration during graph processing. A detailed analysis of the reduction in data transfer for Liberator in comparison to Ascetic is presented in Figure 12, which utilizes the UK dataset as an example. As demonstrated in the figure, Liberator consistently outperforms Ascetic by transferring less data during each iteration, primarily due to the expansion of available memory for the Static Data.

Another performance improvement is achieved from the parallel execution of the two processing tasks for Static Data and On-demand Data. Massive threads can be created and a straightforward solution is dispatching them to the GPU runtime for thread management and concurrent execution(normal scheme). However, dispatching threads that exceed the available physical cores will incur additional context switch overhead. To address this issue, we have implemented three lightweight scheduling approaches. The first approach involves launching the on-demand kernel after the static kernel has finished, allowing for serial execution while avoiding context switches. The second approach merges the two kernels into a single kernel and schedules thread blocks using the sync API provided by CUDA. This approach also avoids context switches but incurs synchronization overhead. Lastly, we attempted to schedule every warp by polling a bitmap in shared memory. This allows a warp to begin processing On-demand Data immediately after completing the computation of Static Data, without waiting for other warps in the block.

We tested the four scheduling methods on the UK dataset in Table 4 with BFS and PR. Table 7 shows the performance comparison of different scheduling mechanisms with the normal scheme.

<table>
<thead>
<tr>
<th>Workload</th>
<th>方案</th>
<th>NM</th>
<th>Serial</th>
<th>Block sync</th>
<th>Warp sync</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>1398ms</td>
<td>-10.44%</td>
<td>-7.58%</td>
<td>-34.9%</td>
<td></td>
</tr>
<tr>
<td>PR</td>
<td>71.58s</td>
<td>-10.25%</td>
<td>-7.77%</td>
<td>-35.14%</td>
<td></td>
</tr>
</tbody>
</table>

It can be seen that, compared to the normal scheme, the serial execution leads to a 20% performance drop, and the synchronization inside one block or one warp also incurs more serious overheads than context switches, especially the polling to a bitmap, which can lead to a 35% performance drop. Obviously, any synchronization and serial execution in GPU will cause a worse impact than context switches, therefore, we lastly choose the normal scheme in Liberator.

The workload assigned to each thread can be different based on the topology of the graph. It is better to manage each thread with a fine-grained scheduling strategy considering the memory footprint and computation intensity. Unfortunately, there are no such fine-grained scheduling APIs in CUDA. Some research proposed some optimization schemes relying on modifications on hardware [27], [28], [29]. We leave it as our future work.

**Table 6**

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
<th>PageRank</th>
<th>SSSP</th>
<th>BFS</th>
<th>CC</th>
<th>GEOMEAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>GS</td>
<td>28.0G</td>
<td>86.5X</td>
<td>41.9X</td>
<td>4.2X</td>
<td>2.3X</td>
<td>1.2X</td>
</tr>
<tr>
<td>FK</td>
<td>20.0G</td>
<td>30.0X</td>
<td>18.3X</td>
<td>2.1X</td>
<td>1.3X</td>
<td>0.8X</td>
</tr>
<tr>
<td>FS</td>
<td>29.0G</td>
<td>23.7X</td>
<td>0.67X</td>
<td>1.6X</td>
<td>1.3X</td>
<td>0.5X</td>
</tr>
<tr>
<td>UK</td>
<td>30.0G</td>
<td>217.9X</td>
<td>133.3X</td>
<td>12.1X</td>
<td>2.2X</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 7**

<table>
<thead>
<tr>
<th>Workload</th>
<th>NM</th>
<th>Serial</th>
<th>Block sync</th>
<th>Warp sync</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>1398ms</td>
<td>-10.44%</td>
<td>-7.58%</td>
<td>-34.9%</td>
</tr>
<tr>
<td>PR</td>
<td>71.58s</td>
<td>-10.25%</td>
<td>-7.77%</td>
<td>-35.14%</td>
</tr>
</tbody>
</table>

Fig. 12. Liberator’s reduced data transfer in each iteration, using the UK data set as an example. The results of BFS, SSSP and CC are counted in MB, and the result of PR is counted in GB.
In Table 4. Table 8 shows the results, the X represents the situations in which the available memory is not sufficient to handle the memory overhead introduced by the corresponding framework.

It can be seen that Ascetic’s performance has dropped significantly as the available memory dwindles. EMOGI seems not memory sensitive because most data access does not go through GPU memory, which leads to bad performance. However, Liberator always maintains a better performance than both.

<table>
<thead>
<tr>
<th></th>
<th>8G</th>
<th>6G</th>
<th>4G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E</td>
<td>A</td>
<td>L</td>
</tr>
<tr>
<td>FK</td>
<td>1.486</td>
<td>3.085</td>
<td>0.796</td>
</tr>
<tr>
<td>FS</td>
<td>1.712</td>
<td>2.193</td>
<td>0.499</td>
</tr>
<tr>
<td>GK</td>
<td>1.314</td>
<td>2.563</td>
<td>1.086</td>
</tr>
</tbody>
</table>

We also use some synthesized graphs generated by RMAT [36](a widely-used graph generator) because there is fewer larger public dataset than the UK on the Internet. These synthesized datasets’ sizes are dozens of GB, i.e. 20GB-40GB in CSR format, and we did not limit available GPU memory to the application in this experiment. Still using BFS as an example, the results show that Liberator can achieve on average 4.84X speedup over Ascetic and 1.91X speedup over EMOGI on these larger datasets.

### 4.4 Comparison with UVM-based scheme

As Unified-Virtual-Memory [34] is a straightforward way to deal with memory oversubscription, we also implement a UVM-based system and compare the execution performance and memory transfers. Figure 13 shows the time performance improvement of Liberator over the UVM-based scheme and the data transfer comparison. It can be seen that the UVM-based scheme is on average 12.5X slower and transfers 16.7X more data than Liberator.

We deeply analyze the UVM-based scheme with nvprof [19]. Our results revealed three shortcomings of applying UVM to graph computing. First, there are many redundant data transfers when requesting data in page generally 4K bytes granularity, and it will get worse considering the irregularity and sparsity of data access. Second, the page faults mechanism incurs significant overhead. Our experiments show that about 61% of program time on average is spent on handling page faults, which is consistent with other research [17]. Third, the eviction mechanism of UVM negatively impacts performance when there is memory pressure. We observed that the UVM system fetches about 4K-1M into memory as the prefetcher, but swaps out about 32K-2M to create vacancies for potential page requests, aligning with the pattern proposed in [35]. However, the inaccuracy of the prefetcher and radicalness of the evictor can result in data thrashing and lead to poor performance.

### 4.5 Impact of GPU Memory Size

Compared to EMOGI [26], which only uses zero-copy memory with limited bandwidth and long access latency, the Static Data in Liberator works as a data cache on the HBM internal memory. As the Static Data catches the data reuse across iterations, it is intuitive that the size of the cache is critical to the performance. In Ascetic [30], the On-demand Region is reserved in the GPU memory, which limits the memory left for data reuse. The size of On-demand Region varies based on the structure of the graphs, which further impacts the performance.

To further understand the performance impact of GPU memory size, we take the BFS algorithm as an example and evaluate the performance of EMOGI, Ascetic, and Liberator. To simulate this scenario, we limit the available GPU memory to 8G, 6G, and 4G and still use the data set shown in Table 4. Table 8 shows the results, the X represents the situations in which the available memory is not sufficient to handle the memory overhead introduced by the corresponding framework.

It is evident that the bitmap scheme utilized in Liberator incurs memory and time overheads. The memory overhead can be easily calculated, where the size of all bitmaps and auxiliary arrays is relative to the number of vertices in the graph, denoted as \(N\). For BFS, CC, and SSSP, the memory overhead amounts to eight unsigned integer arrays and four boolean arrays, each comprising \(N\) elements, resulting in a total memory overhead of \(36N\) bytes. For PageRank, the addition of two double-precision arrays, each with \(N\) elements, increases the total memory overhead to \(52N\) bytes. Specifically, for GS and FK, the memory overhead is approximately 2.6GB for PageRank and 2.2GB for the other algorithms, while for FS and UK, it is about 4.8GB for PageRank and 4.1GB for the other algorithms. If the number of vertices becomes excessively high, the memory overhead of the bitmaps may exceed the capacity of the GPU memory, resulting in issues such as the X illustrated in Table 8.

We have also evaluated the time overhead in experiments, incurred during look-up for Static Data and On-demand Data, as shown in Table 9. The results show that this processing consumes only 0.7% of program execution time in the best case and 6.55% on average. This is because the AND and XOR operations between ActiveBitmap and StaticBitmap are in parallel which is considerably fast on the GPU. This result is intuitive and easy to comprehend. Some of the results presented in Table 9 appear to be subpar, particularly for BFS. This is due to that the application occasionally selects an isolated vertex as the source, resulting in
no computation being necessary but the lookup cannot be passed.

### TABLE 9
Time overhead incurred during processing bitmaps in Liberator. Results are normalized to the percentage of the total program execution time

<table>
<thead>
<tr>
<th></th>
<th>BFS</th>
<th>SSSP</th>
<th>CC</th>
<th>PR</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSH</td>
<td>10.11%</td>
<td>8.53%</td>
<td>2.47%</td>
<td>1.1%</td>
</tr>
<tr>
<td>FK</td>
<td>6.49%</td>
<td>4.46%</td>
<td>1.5%</td>
<td>0.81%</td>
</tr>
<tr>
<td>FS</td>
<td>18.25%</td>
<td>7.58%</td>
<td>8.52%</td>
<td>2.46%</td>
</tr>
<tr>
<td>UK</td>
<td>17%</td>
<td>5.78%</td>
<td>7.07%</td>
<td>0.7%</td>
</tr>
<tr>
<td>GEOMEAN</td>
<td>6.35%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In general, the memory overhead is a little considerable, while the time overhead is negligible. Despite this, taking into account the time advantage and the limited capacity of GPU memory to handle all edges, it is justifiable to trade off some memory for faster graph processing. Further optimization of our bitmap scheme will be left for future work.

### 4.7 Prefetch and Replacement of Static Data

Based on the observation that most graph processing applications exhibit near sequential and evenly access of data chunks in the dataset, it is straightforward that the Static Data can be randomly selected across the graph dataset. We have conducted a series of experiments by filling up the Static Data with the front portion, the rear portion, and randomly selected data chunks in the dataset. In our measurement, the initial dataset in Static Data has negligible impact on the performance (less than 5%). Therefore, we always fill the Static Data with the front part of the dataset.

Our experiments also show that the replacement of the dataset in Static Data does not significantly improve the performance for three reasons. First, there is no obvious “hot data” in graph computing because the program just scans the data set sequentially most of the time, which leads to no obvious victim for the replacement. Second, the overhead of the replacement of CSR data is unacceptable. Due to the overly compact way it organizes data, modifications for each location in VertexList or EdgeList will result in the reorganization of the total list. If the CPU spends too much time organizing the replacement data, GPU has to wait idly, which also hurts the performance. Last but not least, there are no convenient atomic operations between CPU and GPUs. The procedure of updating Static Data should be protected and GPU threads have to avoid visiting it, similar to the traditional producer-consumer model. However, the synchronization between CPU and GPU is so heavyweight that it may offset the potential performance gain.

The available time for replacement varies depending on the situation, as illustrated in Figure 14. Due to the absence of a fine-grained atomic lock scheme between CPU and GPU in the current CUDA toolkit and our hardware platform, the update process must start after the completion of the Static Data Computing to avoid GPU accessing incorrect data. When the Static Data Computing takes longer than the On-demand Data Computing, as shown in Figure 14(a), the GPU must be in a busy wait state (the blue area). Given the complexity of updating CSR and the large size of the graph, this overhead may negate any potential benefit from replacement. However, when the On-demand Data Computing takes longer, there is an opportunity to overlap the Static Data Update with it. Nevertheless, based on our experimental measurements, after excluding the time overlapped by our overlapping mechanisms, only a small fraction (3%-5%) of GPU time remains available for replacement (the yellow block in Figure 14(b)). During this time, only around 2% of the total data transfer (about 200-300M) can be completed. If we were to force the GPU to wait idly for the replacement, the busy waiting and synchronization overhead would hurt system performance.

If a fine-grained atomic lock scheme between the CPU and GPU like CXL [37] were available to GPUs, the Static Data Update could be processed in parallel with GPU data computing, resulting in significant benefits for the system. Additionally, if the graph is large enough and the On-demand Data Computing takes sufficient time to overlap with the replacement overhead (i.e., the yellow area in Figure 14(b) is long enough), the system could benefit from replacement as well. Due to hardware and data scale limitations, we have to cancel the replacement mechanism described in Section 3.4 in our final experiments for better performance on current datasets with our hardware platform.

### 5 Related Works

A number of graph processing frameworks have emerged in this era of big data, including Grasp [38] and Wonderland [39] for the single-machine environment, and Pregel [40], GraphLab [16] for clustered environments. As the graph data exceeds the size of the main memory, out-of-core graph processing systems such as Graphchi [41] and Grid-Graph [42] have been proposed that partition the graph datasets into smaller sub-graphs and load them from large data storage devices. Similar to the CPU-based graph
processing frameworks, there are many graph-processing systems based on GPUs, such as Cuspa [43], Enterprise [44], and GunRock [45]. However, GPU memory is also a limited resource for large graph datasets. To address such GPU memory constraints, a lot of research has been conducted recently [46]. There are three major challenges that need to be addressed. The first challenge is that the partition-based approaches can have significant overheads due to redundant data transfer. The second challenge is that the data locality is difficult to exploit in graph analytic applications. The third challenge is that UVM or zero-copy has high latency and low bandwidth utilization in graph traversal.

Subway [23] is a fine-grained memory management graph processing system proposed recently. It uses a fine-grained data transfer scheme by accurately organizing the subgraph required in each iteration. In this way, the data that needs to be transferred in each iteration can be significantly reduced. However, it ignores the data reuse across iterations of graph processing. Ascetic [30] is the state-of-the-art graph processing system proposed in our recent research. It partitions GPU memory into two regions. Static Region is for data reuse. On-demand Region, managed with the approach of Subway, transfers other data on demand. Compared to Ascetic, we further improve the performance by removing the On-demand Region to extend the data reuse area. And then, we improve the memory access which is absent from the data reuse area in a Merged and Aligned manner, which is inspired by EMOGI [26].

EMOGI [26] focuses on the utilization of the existing hardware, which is orthogonal to Liberator. The Merged and Aligned memory access is critical to GPU memory oversubscription, which is also applicable in our framework. However, it neglects the utilization of GPU memory, which has lower latency and extremely higher bandwidth compared to the PCIe bus.

In Liberator, we fully exploit the memory hierarchy of GPU memory and the main memory. Firstly, the GPU memory works as the high-performance data cache by holding the static Data, which is accessed by multiple GPU threads and shared across iterations in graph computing. Second, the Merged and Aligned access optimization is employed for the “cache miss events”, which improves the memory transfer efficiency.

6 Conclusion

Processing large graph datasets that exceed GPU memory is a fundamental challenge in graph analytic applications. In our studies, we focus on data efficiency in GPUs by taking advantage of data reuse. By deeply analyze of memory access patterns, we found that there is a long reuse distance across iterations. Based on this observation, we propose Liberator, to exploit the data reuse across iterations with a cache mechanism in GPU memory. With two bitmaps, GPU could determine cache hit or not by fast operations. With judicious use of zero-copy optimizations and CUDA stream mechanism, the penalty of cache misses is significantly reduced and the parallelism of GPU is also improved. We have implemented the graph framework and evaluated its performance with 4 conventional graph applications running on different datasets. The results show that Liberator can outperform traditional partition-based graph processing schemes, Ascetic and EMOGI, and archive a great speed up over them.

The source code of Liberator is publicly available at https://github.com/NKU-EmbeddedSystem/Liberator

Acknowledgments

This work was supported in part by the Key Research and Development Program of Guangdong, China (2021B0101310002), Natural Science Foundation of China (62172239), and Shandong Provincial Natural Science Foundation, China (ZR2022LZH009).

References
