LargeGraph: An Efficient Dependency-Aware GPU-Accelerated Large-Scale Graph Processing

YU ZHANG, DA PENG, YUXUAN LIANG, XIAOFEI LIAO, HAI JIN, HAIKUN LIU, and LIN GU, National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, China
BINGSHENG HE, National University of Singapore, Singapore

Many out-of-GPU-memory systems are recently designed to support iterative processing of large-scale graphs. However, these systems still suffer from long time to converge because of inefficient propagation of active vertices' new states along graph paths. To efficiently support out-of-GPU-memory graph processing, this work designs a system *LargeGraph*. Different from existing out-of-GPU-memory systems, LargeGraph proposes a *dependency-aware data-driven execution approach*, which can significantly accelerate active vertices' state propagations along graph paths with low data access cost and also high parallelism. Specifically, according to the dependencies between the vertices, it only loads and processes the graph data associated with dependency chains originated from active vertices for smaller access cost. Because most active vertices frequently use a small evolving set of paths for their new states' propagation because of power-law property, this small set of paths are dynamically identified and maintained and efficiently handled on the GPU to accelerate most propagations for faster convergence, whereas the remaining graph data are handled over the CPU. For out-of-GPU-memory graph processing, LargeGraph outperforms four cutting-edge systems: Totem (5.19–11.62×), Graphie (3.02–9.41×), Garaph (2.75–8.36×), and Subway (2.45–4.15×).

CCS Concepts: • **Computer systems organization → Special purpose systems**; **Single instruction, multiple data;**

Additional Key Words and Phrases: Graph processing, GPU, out-of-GPU-memory, access cost

ACM Reference format:
Yu Zhang, Da Peng, Yuxuan Liang, Xiaofei Liao, Hai Jin, Haikun Liu, Lin Gu, and Bingsheng He. 2021. LargeGraph: An Efficient Dependency-Aware GPU-Accelerated Large-Scale Graph Processing. *ACM Trans. Archit. Code Optim.* 18, 4, Article 58 (September 2021), 24 pages.
https://doi.org/10.1145/3477603

This work was supported by National Natural Science Foundation of China (grants 61832006, 61825202, 62072193, and 61929103). This work was also sponsored by Zhejiang Lab (2021KD0AB01) and the Fundamental Research Funds for the Central Universities (2020kfyXJJS018).
Authors' addresses: Y. Zhang, D. Peng, Y. Liang, X. Liao, H. Jin, H. Liu, and L. Gu, National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei, 430074, China; emails: {zhyu, xfliao, hjin}@hust.edu.cn; B. He, National University of Singapore, Singapore; email: hebs@comp.nus.edu.sg.

1544-3566/2021/09-ART58 $15.00
https://doi.org/10.1145/3477603

1 INTRODUCTION

In real-world applications, large-scale graphs widely exist. Lots of algorithms [6, 24, 28, 33, 38, 39, 44, 46–50] have been designed to iteratively process these graphs to get the convergent results for different purposes. These applications iteratively process large-scale graphs until a user-specified condition is met and thus are usually time consuming. It has become a fruitful research topic to accelerate these large-scale graph algorithms. Many CPU-based graph processing systems [8, 25, 27, 34, 41–43, 51–53] have been proposed in recent years. However, their performance is still limited due to the limited hardware parallelism and memory bandwidth of CPU. Because GPU is much more powerful on the aspects of parallelism (e.g., 6,912 CUDA cores for Tesla A100 GPU) and memory bandwidth (e.g., 1,555 GB/s for Tesla A100 GPU), many GPU-accelerated systems, such as Totem [13, 14], Graphie [16], Garaph [29], and Subway [31], have been recently developed to offload iterative graph processing from the CPU to the GPU. They support large-scale graph processing by sparing data access overhead and redressing imbalanced load, and so on.

However, due to the dependencies (caused by the edges) between the vertices, to reach the other vertices for convergence, each active vertex's state requires to be propagated down the path between them sequentially. With existing out-of-GPU-memory systems [9, 10, 13, 14, 16, 19, 29, 31], vertices and edges on each path may be distributed over CPU and GPU. In each round, these vertices may also be processed on different CPU/GPU cores concurrently. Consequently, for out-of-GPU-memory graph processing's convergence, the active vertices' new states need to sequentially travel many CPU/GPU cores in these systems to be propagated along the paths, incurring high synchronization cost and CPU-GPU data transfer cost.

Taking Subway [31] as an example, in Figure 1, higher ratio of the total execution time is wasted to transfer the graph data between the CPU and the GPU when more powerful GPU is used. For example, for SSSP on uk2007, 85.9% of the total execution time is wasted to transfer graph data for GTX980, whereas the ratios are up to 91.5% and 93.6% for K80 and A100, respectively. This is because long communication time is used to transfer graph data between the CPU and the GPU to sequentially propagate vertex states along graph paths, although the processing of vertex states can be more quickly finished when the GPU has more powerful computational capacity. This means that the GPU utilization is lower when the GPU is more powerful, although the execution time is smaller. In addition, because of the expensive and slow propagation of the new vertex states, more cores become idle and more vertices can only update their states using the other vertices' stale states (i.e., the ones that are not the latest states) when the GPU has more cores. Thus, the ratio of the unnecessary graph transfer time (transferring the graph data associated with the updates based on stale states) to the whole transfer time is higher when the GPU has more cores. For example, for SSSP on uk2007, the ratios of the unnecessary graph transfer time to the whole transfer time are 65.0%, 67.9%, and 70.2% for GTX980, K80, and A100, respectively.

To tackle this challenge, this work develops a system *LargeGraph*, which uses our proposed *dependency-aware data-driven execution approach* to accelerate active vertices' state propagations along the paths. Specifically, following the dependencies between the vertices, LargeGraph dynamically explores and only loads the paths originated from active vertices for processing for smaller data access cost. Because of the real-world graphs' power-law property [15], a few paths are frequently used to propagate new states of most active vertices. Thus, in LargeGraph, these frequently used paths are also efficiently identified on-the-fly and are dispatched to be efficiently handled on the GPU for faster convergence so as to accelerate most propagations, whereas the remaining paths are tried to be handled over the CPU. These frequently used paths are also maintained in the GPU for reusing. Compared with four out-of-GPU-memory systems, namely Totem [13, 14], Graphie [16], Garaph [29], and Subway [31], LargeGraph offers speedups of 5.19–11.62, 3.02–9.41, 2.75–8.36, and 2.45–4.15 times, respectively.

GPUs	Architecture	CUDA Cores	GPU Memory	Bandwidth
GTX980	Maxwell	2,048	4 GB	224 GB/s
K80	Kepler	4,992	24 GB	480 GB/s
A100	Ampere	6,912	40 GB	1,555 GB/s

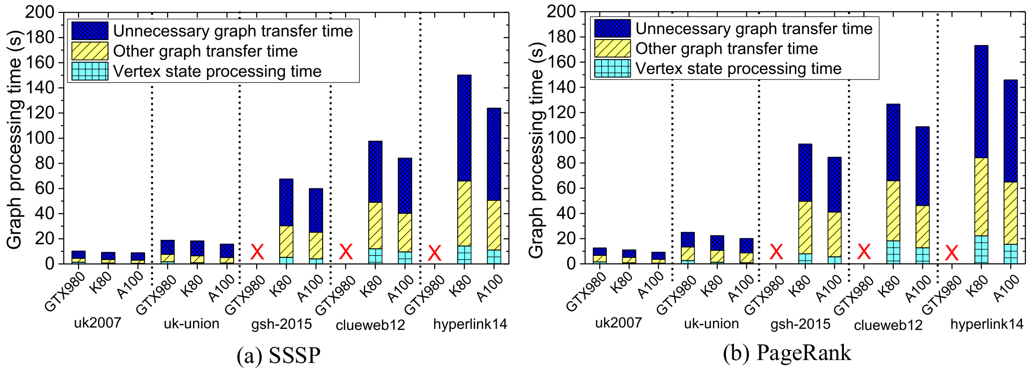


Fig. 1. Graph processing time breakdown of SSSP and PageRank over Subway [31] after plugging three various generations of GPUs into a PC with 48 cores and 256 GB of main memory, respectively, where the graphs' sizes range from 46.2 GB of uk2007 to 793.4 GB of hyperlink14.

This article has three major contributions:

- An effective dependency-aware data-driven execution approach is proposed for out-of-GPU-memory graph processing to achieve faster convergence speed and smaller data access cost.
- An efficient system *LargeGraph* is designed for efficient implementation of our execution approach over the platform with the GPU accelerator.
- Experimental evaluation of LargeGraph is conducted to demonstrate its advantages by comparing with the cutting-edge out-of-GPU-memory systems.

The rest of the article is organized as follows. Section 2 discusses the background and motivations, followed by our LargeGraph in Section 3. Section 4 comprehensively evaluates its performance in comparison with the existing GPU-accelerated graph processing solutions. Section 5 surveys the related work. Section 6 presents our conclusion.

2 BACKGROUND AND MOTIVATIONS

The PC is often equipped with GPUs in the recent years, where each GPU has multiple **Streaming Multiprocessors (SMs)**. Recently, many systems [13, 14, 16, 29, 31] have been developed to use the GPU to accelerate large-scale iterative graph processing.

2.1 Problems of Existing Techniques Based on GPU

In practice, graph processing is the process of iteratively calculating each vertex's state based on the influences of its direct and indirect neighbors' states [15, 38]. However, for iterative graph algorithms, within each round of graph processing, each vertex only updates its state based on its direct neighbors' states according to its edges. Thus, for the convergence of each vertex, the influence of each indirect neighbor's state needs to be iteratively propagated to it sequentially along the graph path between it and this indirect neighbor. To accelerate this process, many systems are designed to use asynchronous graph processing model [5, 38] to efficiently support the asynchronous versions of graph algorithms, such as asynchronous SSSP [38] and asynchronous PageRank [38].

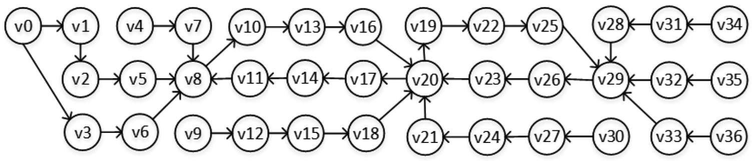


Fig. 2. An example to show the inefficiency of existing systems.

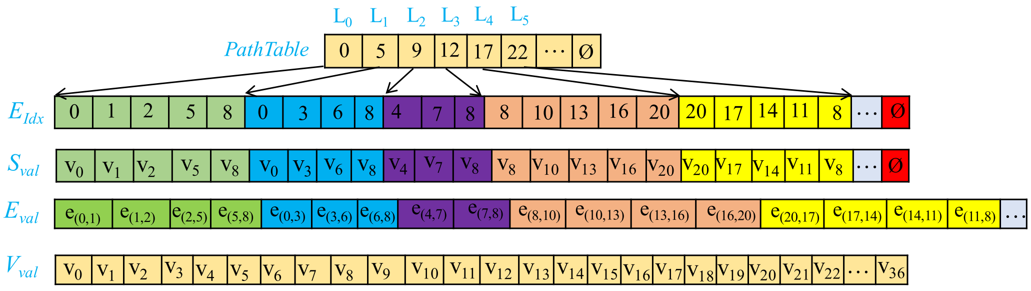


Fig. 3. Illustration of graph representation.

Given a graph $G = (V, E)$, for asynchronous graph processing [5, 38], after processing G for R rounds, the state s_i of each vertex v_i is

$$s_i^R = s_i^0 \oplus \left(\sum_{v_t \in N(i)} \oplus f_{(v_t, v_i)} \left(s_t^{k_t} \right) \right), \tag{1}$$

where

$$\sum_{v_t \in N(i)} \oplus f_{(v_t, v_i)} \left(s_t^{k_t} \right) = f_{(v_{i_1}, v_i)} \left(s_{i_1}^{k_{i_1}} \right) \oplus \cdots \oplus f_{(v_{i_m}, v_i)} \left(s_{i_m}^{k_{i_m}} \right). \tag{2}$$

$\oplus$ is a user-defined generalized sum. $f_{(v_t, v_i)}(s_t^{k_t})$ denotes the influence of v_t's state (i.e., $s_t^{k_t}$) on the state of v_i. $N(i)$ is the direct neighbor set of v_i, $0 \leq k_t \leq R, v_{i_1}, \ldots, v_{i_m} \in N(i)$. We can observe that after R rounds of graph processing, each vertex's new state is the results of the generalized sum of its initial state (e.g., s_i^0) and the influences (e.g., $f_{(v_t, v_i)}(s_t^{k_t})$) of the states of its direct neighbors, where $s_t^{k_t}$ may be stale state.

To further accelerate state propagation along graph paths, DiGraph [44] recently proposes to divide graph into static paths at the preprocessing stage and asynchronously processes these paths. Taking the graph depicted in Figure 2 as an example, DiGraph uses four arrays to efficiently store the generated graph paths as shown in Figure 3. For the edges on each path, the array E_{Idx} stores the indexes of their source vertices, whereas the array S_{val} is used to store the states of the vertices in E_{Idx} accordingly. To help efficient propagation along each path, in the array E_{Idx}, the indexes of each path's vertices are sequentially maintained along this path. Two successive entries of E_{Idx} represent an edge. The array E_{val} stores the weights of the edges accordingly as described in Figure 3. An array (i.e., V_{val}) stores the state values of the vertices. It also uses a *PathTable* to maintain the information of these graph paths. A field of this table is used to store the index of the beginning vertex of each path, where two successive values indicate the range of a path. However, as shown in Section 4.4, when using the cutting-edge approach [10] to extend DiGraph to support out-of-GPU-memory graph processing, it performs worse than the existing system *Subway* [31] due to more data access cost. Although LargeGraph uses a path-based graph decomposition similar to DiGraph, different from DiGraph, LargeGraph proposes a lightweight runtime approach to

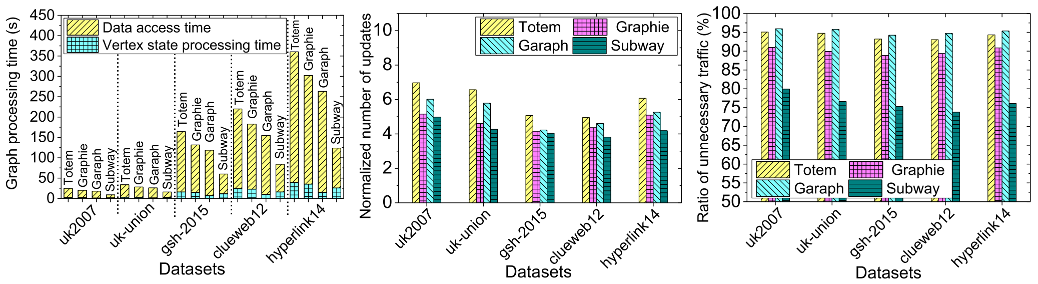
Table 1. Comparison of LargeGraph with the Prior Systems

GPU-Based Systems	**Ratio of Updates Based on Stale States to All Updates**	**Ratio of Unnecessary Data Accesses to All Accesses**
Totem [13, 14]	High	High
Garaph [29]	High	High
Graphie [16]	High	Medium
Subway [31]	High	Low
LargeGraph	Very Low	Very Low

dynamically explore the dependency chains originated from active vertices, and also dynamically maintains the frequently used paths as well as the direct dependencies associated with these paths on the GPU, which enables much smaller data access cost, faster vertex state propagation, and higher GPU utilization.

Recently, to support large-scale graph processing using GPU, many out-of-GPU-memory systems [13, 14, 16, 29, 31] have been developed. When using these systems, large-scale graphs are decomposed into several same-sized subgraphs (or called *chunks*), which can fit into the GPU's global memory to be dispatched to the GPU for processing. Thus, the vertices and edges on each path may be divided into many chunks, which are distributed over the CPU and GPU. To sequentially propagate new vertex states along the paths for large-scale graph processing, existing systems need high CPU-GPU data transfer cost and also suffer from slow propagation of these new vertex states. When the new vertex states are expensively and slowly propagated along the graph paths, many GPU threads become idle because the required new vertex states are unavailable, or the vertices on many GPU cores update their own states using their direct neighbors' stale states within each round, where these updates based on stale states in practice are invalid (i.e., are unnecessary). It eventually incurs a long idle time of the GPU threads or a long time to conduct much unnecessary data accessing and processing associated with the updates based on stale states, until the new states from their indirect neighbors are received. Table 1 shows the inefficiency of the cutting-edge out-of-GPU-memory graph processing systems.

Taking the Bellman-Ford-style SSSP [38, 44] as an example, v_0 of Figure 2 is assumed to be the source vertex. With existing solutions, the graph path $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow v_5 \rightarrow v_8 \rightarrow v_{10} \rightarrow v_{13} \rightarrow v_{16} \rightarrow v_{20}$ may be divided into several chunks, which are distributed over the CPUs and SMs. In addition, when a chunk is handled over a SM of the GPU, different GPU threads of the same warp always concurrently process the vertices contained in this chunk. Consequently, many cross-core state propagations (i.e., the ones propagated between the CPU/GPU cores) are generated in existing systems when they are conducted along the paths, incurring high synchronization cost and communication cost. For example, v_0's state (i.e., the distance from the source vertex) may need high cost to travel several CPU/GPU cores to reach the other vertices (e.g., v_{20}). The update of v_{20}'s state can be conducted only when it has received the state of v_0 along a path. This indicates that the GPU thread (which processes v_{20}) may become idle when no new state is received from v_0. In addition, although v_0 has reached v_{20} through a path (e.g., $v_0 \rightarrow v_3 \rightarrow v_6 \rightarrow v_8 \rightarrow v_{10} \rightarrow v_{13} \rightarrow v_{16} \rightarrow v_{20}$) and the states of v_{20} as well as the vertices following v_{20} (e.g., v_{19}) have been updated accordingly, the states of v_{20} as well as its followers (e.g., v_{19}) may have to be updated again, when the state of v_0 slowly reaches v_{20} along the other path (e.g., $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow v_5 \rightarrow v_8 \rightarrow v_{10} \rightarrow v_{13} \rightarrow v_{16} \rightarrow v_{20}$). This indicates that the expensive and slow vertex state propagation may incur low GPU utilization, because much GPU time may be idle or may be wasted to conduct updates based on stale states.



(a) Graph processing time breakdown of different systems over various graphs

(b) Number of updates normalized to that of the sequential way

(c) Ratio of unnecessary traffic volume to that of all traffic volume

Fig. 4. Performance of SSSP on existing systems.

To demonstrate the preceding discussion, we evaluate a Bellman-Ford-style SSSP [38, 44] on Totem [13, 14], Graphie [16], Garaph [29], and Subway [31]. Section 4 depicts the platform and the benchmarks. As shown in Figure 4(a), these systems suffer from high data access overhead to computation ratio. Although Subway needs to transfer the lowest volume of graph data between the CPU and the GPU, it still needs high data access overhead. This is because the graph data associated with a vertex on a path can be transferred by Subway to the GPU only when the state of this vertex's precursor has been propagated to this vertex to activate it. More importantly, as shown in Figure 4(b), much more updates are conducted in these systems than the sequential way, due to the previously described slow cross-core propagations of active vertices' states along the paths, where the sequential way is the way to asynchronously execute Bellman-Ford-style SSSP and asynchronously handle the vertices on only a core in a depth-first order in each round. So many updates indicate that a major proportion of the vertex state processing time and a major ratio of the data access time are wasted to process and transfer unnecessary graph data (see Figure 4(c)), respectively.

In practice, previous work [45] has shown that only the graph data on dependency chains originated from active vertices require to be processed. Smaller data access cost is required when the vertices are asynchronously handled along these chains. In addition, a path will be accessed more times in a round when more active vertices connect to the beginning vertex of this path in this round. This motivates our design for faster out-of-GPU-memory graph processing.

3 OVERVIEW OF LARGEGRAPH

LargeGraph is proposed to accelerate large-scale graph processing by fully utilizing the powerful capacity of the GPU accelerator. Different from existing solutions, LargeGraph proposes an effective *dependency-aware data-driven execution approach*, which can efficiently load and effectively process the graph data associated with dynamically explored dependency chains originated from active vertices. This enables much smaller data access cost and less unnecessary processing cost (i.e., the processing based on stale vertex states). For faster convergence speed and smaller data access cost, frequently used graph paths are dynamically identified and then are dispatched to be handled and maintained on the GPU accelerator so as to work as a fast bridge for most propagations, whereas the remaining paths are tried to be handled over the CPUs.

3.1 Dependency-Aware Data-Driven Execution Method

To efficiently identify the dependency chains on-the-fly at the execution time, the graph $G = <V, E>$ is first represented as $G = \cup_{L_l \in L} L_l$ in advance. L is a set of disjoint paths. $L_l = < v_x, \ldots, v_y >$ is

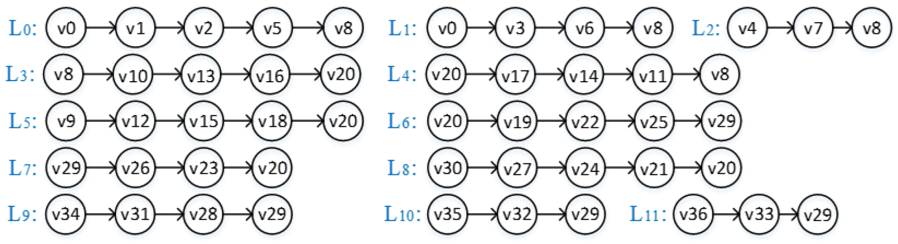


Fig. 5. Illustration of dividing the graph of Figure 2 into paths.

a sequence of connected edges to ensure that $v_x \in V$ and $v_y \in V$. In addition, for any two paths (e.g., $L'_l = < v'_x, \ldots, v'_y >$ and $L''_l = < v''_x, \ldots, v''_y >$), it should ensure that $L'_l \cap L''_l \subseteq \{v'_x, v'_y\} \cap \{v''_x, v''_y\}$. This means that the intersections of any two paths are only the intersections of the beginning and end vertices of these two paths. Figure 5 gives an example. By such means, any dependency chain originated from any active vertex must consist of a subset of paths of L. For example, in Figure 2, the dependency chain $v_1 \rightarrow \cdots \rightarrow v_{20}$ originated from an active vertex v_1 consists of L_0 and L_3 of L. More importantly, through such a graph decomposition, such a subset of paths can be efficiently identified at the execution time according to the dependencies between the paths of L. This enables us to use a lightweight runtime approach to dynamically explore dependency chains originated from active vertices at the beginning of each round of graph processing.

In detail, at the beginning of each round of graph processing, it first takes the paths with active vertices as the initial active paths. Then, it iteratively explores the other active paths (the ones on the chains originated from the active vertex) according to their dependencies until all paths originated from the initial active paths have been explored. Specifically, if the beginning vertex (e.g., v_8) of a path (e.g., L_3) is the end vertex of an active path (e.g., L_0, where v_1 is assumed to be an active vertex), this path (i.e., L_3) also belongs to the dependency chains originated from v_1 and is also identified as an active path, which needs to be processed within this round of graph processing. Otherwise, it is an inactive path (i.e., all its vertices must be inactive) and does not need to be loaded for processing, because no new vertex state can reach its vertices and no update needs to be conducted for its vertices within this round of graph processing. For example, assume that only v_1 is the active vertex at the beginning of a round of graph processing. Then, L_0 will be an initial active path. L_3, L_4, L_6, and L_7 then will be identified as active paths as well. Only these identified active paths are loaded and then put together to dynamically constitute the *logical chunks*, which will then trigger the dispatching and processing of the task (i.e., the processing of a logical chunk) on the CPU/GPU within this round. The graph data associated with the remaining paths are not loaded and processed within the current round of graph processing. Note that the most frequently used active paths are tried to be put together into the same chunks (or called *frequently used chunks*). The remaining active paths are put together as the *rarely used chunks*. Note that the set of frequently used paths changes at the execution time. The frequently used chunks are transferred to the GPU for parallel processing, and the rarely used chunks are concurrently processed on the CPUs. In addition, for smaller data access overhead, the frequently used paths are also dynamically maintained in the GPU. This processing way has two benefits. First, the frequent state updates of the vertices on the frequently used paths can be quickly conducted using the high parallelism and high memory bandwidth of the GPU, and most state propagations can quickly reach the others through these paths. Second, only the frequently used paths are transferred from the CPU to the GPU for repeatedly processing on the GPU, improving the utilization of the scarce GPU memory and the limited CPU-GPU bandwidth.

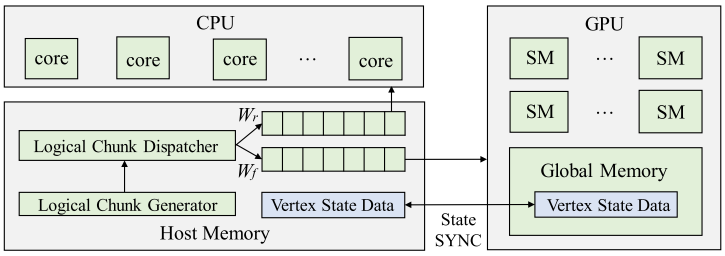


Fig. 6. Architecture of LargeGraph.

3.2 Implementation of LargeGraph

This section first describes the architecture of LargeGraph in Figure 6 and then discusses how to use our proposed execution approach to efficiently support out-of-GPU-memory graph processing.

In detail, the graph is initially stored on the host. At the graph preprocessing stage, each CPU thread is first evenly assigned with a range of vertices associated with their edges (i.e., a subgraph), and then these CPU threads concurrently divide these subgraphs into disjoint paths. In this way, the graph can be concurrently divided into disjoint paths.

After that, at the beginning of each round, active paths are identified (the frequently used paths are also identified) by *Logical Chunk Generator* (which is responsible for the identification of active paths and the generation of logical chunks) from the preceding set of generated paths, and only these active paths are used to constitute the logical chunks, which will then trigger the dispatching and execution of the tasks (each task is the processing of a logical chunk) on the CPU or GPU accordingly within this round. In this way, many graph data—that is, the graph data associated with the inactive paths—do not need to be loaded and processed. When *Logical Chunk Generator* generates the logical chunks, the most frequently used active paths are put together into the same chunks (or called *frequently used chunks*). The remaining active paths are put together as the *rarely used chunks*. Then, *Logical Chunk Dispatcher* assigns the processing of the frequently used chunks to the GPU by storing these chunks in a worklist W_f (which are only processed by the GPU), whereas the rarely used chunks are stored in the other worklist W_r (which are only handled by the CPU). Figure 6 shows that these two worklists are created in the main memory of the host. Note that the set of frequently used chunks and the rarely used chunks change with the time and are dynamically identified and maintained in these two worklists at the beginning of each round. The frequently used chunk in the worklist W_f will be transferred to the GPU for processing when a warp of GPU threads are idle and need to process a logical chunk.

A vertex may have multiple replicas[1] (distributed on the paths of different logical chunks), and thus LargeGraph needs to synchronize vertex state data at the execution time. Note that to quickly access each vertex's new state, both the GPU memory and the host memory maintain an array (i.e., V_{val}) that stores the state values of the vertices' masters. When synchronizing the replicas' states, LargeGraph also takes the chunk as the synchronization unit and uses an approach similar to that of DiGraph [44] for low cost. As shown in Figure 7, for each SM or CPU, a buffer is created for it and this buffer has several entries. When a chunk is being handled on this SM or CPU, each entry of this buffer is used for each vertex of this chunk to temporarily store the accumulated results of the pushed new states of this vertex' local mirrors (step ①). The accumulated results are pushed to update the states of the masters stored in V_{val} of the GPU memory for state synchronization (step ②), only when the processing of the chunk has been finished. Note that the local mirrors

[1]One replica is called *master*. The others are regarded as *mirrors*.

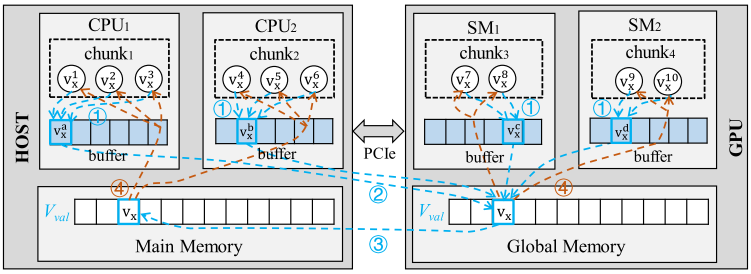


Fig. 7. Example of state synchronization between the replicas of a vertex v_x.

```
template<typename T>
// It means *addr← *addr+ value
__host__ __device__ void Accum(T* addr, T value){
# ifdef _CUDA_ // For GPU
     atomicAdd(addr, value);
# else // For CPU
     *addr+=value;
# endif
}

// Processing the edge <vj, vi> to get the influence
 (which is denoted by f(vj,vi)(sj)) of sj on si
__host__ __device__ T EdgeCompute(Vertex vj, Vertex vi){
    return d×vj.Δvalue/vj.OutDegree;
}
```

(a) PageRank

```
template<typename T>
// It means *addr← Min(*addr, value)
__host__ __device__ void Accum(T* addr, T value){
# ifdef _CUDA_ // For GPU
     atomicMin(addr, value);
# else // For CPU
     Min(addr, value);
# endif
}

 // Processing the edge <vj, vi> to get the influence
 (which is denoted by f(vj,vi)(sj)) of sj on si
__host__ __device__ T EdgeCompute(Vertex vj, Vertex vi){
    return vj.value+<vj, vi>.distance;
}
```

(b) SSSP

Fig. 8. Graph algorithm examples using our APIs.

on the chunk are allowed to update their neighbors in the same round according to the recently accumulated results stored in the buffer for this chunk, enabling fast propagation of the new vertex state. After that, the corresponding vertex states stored in V_{val} of the host memory are also updated accordingly (step ③). Finally, when LargeGraph uses the master's new state to update all its mirrors' states, the updates are arranged to be conducted according to the destination chunks' IDs. The updates for the same destination chunk are conducted together (step ④). Then, fewer chunks need to be loaded, because lots of vertex state updates are successively conducted on the same chunk.

LargeGraph uses the popular *Gather-Apply-Scatter* (GAS) programming model [15, 38]. To implement graph algorithms on LargeGraph, *Accum*() and *EdgeCompute*() need to be instantiated by the users (Figure 8). *EdgeCompute*() calculates the influence of the state (e.g., s_j) of a vertex (e.g., v_j) on the state (e.g., s_i) of its out-neighbor (e.g., v_i). *Accum*() accumulates the influences of the states of in-neighbors (e.g., v_j) for a vertex (e.g., v_i).

The following section first describes the parallel graph preprocessing on the CPU to efficiently divide the graph. Then, it discusses how to efficiently handle the graph over the CPU and the GPU concurrently.

3.2.1 Parallel Graph Partitioning. A parallel approach is employed on the CPU to efficiently decompose the large-scale graph, which accesses the graph for once and partitions the graph into paths. Specifically, each thread is first evenly assigned with a range of vertices associated with their edges (i.e., a subgraph). In other words, it just needs to assign the starting and ending indices of the vertices to be preprocessed by each thread so as to divide the graph into subgraphs. After

ALGORITHM 1: Dependency-Aware Graph Partitioning

```
1:  procedure GraphPartition($v_h$, $L_l$, $L$)
2:      $S_N \leftarrow$ GetNeighbors($v_h$)
3:      for $v_k \in S_N \land \langle v_h, v_k \rangle$ is unvisited do
4:          The edge $\langle v_h, v_k \rangle$ is set as visited.
5:          $L_l \leftarrow L_l \cup \langle v_h, v_k \rangle$
6:          if $v_k$ has in-degree larger than 1 then
7:              $L \leftarrow L \cup L_l$
8:              $L_l \leftarrow \emptyset$
9:          else
10:             GraphPartition($v_k$, $L_l$, $L$)
11:         end if
12:     end for
13: end procedure
```

that, these threads can concurrently divide these subgraphs into disjoint paths, where each thread preprocesses the vertices (and the edges associated with these vertices) within the range of vertices assigned to this thread. For each subgraph, as described in Algorithm 1, a thread travels its edges in a depth-first order, and this thread repeatedly takes its vertex as the beginning vertex until all edges have been visited. It aims to decompose this subgraph into a set of disjoint graph paths that meet the requirements described in Section 3.1. It generates each path L_l by recursively gathering the visited edges of each traverse one by one (see Lines 5 and 10). It begins to explore the next path when it reaches a vertex with in-degree larger than 1 (see Line 6). All these generated paths are inserted into a path set L (see Line 7), which will be used to constitute the logical chunk at the execution time (see Section 3.2.2). These generated paths are also stored using the approach of DiGraph [44] for efficient data accesses.

3.2.2 Dynamic Generation and Dispatching of Logical Chunks. To efficiently generate the logical chunks, it employs a bitmap to indicate whether a path is active at the beginning of the current round of graph processing. This bitmap is updated by LargeGraph at the beginning of this round of graph processing through taking the current active paths as the initial active paths to iteratively explore the other active paths (i.e., the paths on the dependency chains originated from active vertices at the beginning of this round) according to the dependencies between the paths until all paths originated from the initial active paths have been explored (see Section 3.1). Then, through sequentially scanning this bitmap, the active paths can be identified at the beginning of the current round. Only these active paths are loaded and used to constitute the logical chunks for the current round of graph processing (i.e., the other paths are not loaded for processing).

To ensure better locality, in LargeGraph, the size of the logical chunk is different for the ones to be handled on the CPU and the GPU. The size of the chunk is set to half that of a SM's shared memory for the ones to be handled over the GPU, whereas it is half that of the last-level cache of each CPU for the chunks to be handled over this CPU. By such means, not only better locality is ensured but also the next chunk can be loaded in advance to hide the latency of data loading, when a chunk is being processed. When the current chunk has been handled, it can immediately process the next chunk.

To constitute the frequently used chunks, the frequency of the usage (i.e., $f(L_j)$) of each path (i.e., L_j) is evaluated on-the-fly at the beginning of each round, because the frequency of each

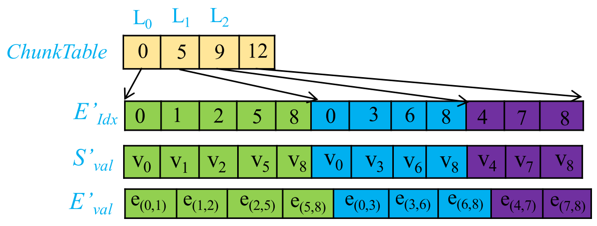

L0
L1
L2
ChunkTable
0 5 9 12
E'Idx
0 1 2 5 8 0 3 6 8 4 7 8
S'val
v0 v1 v2 v5 v8 v0 v3 v6 v8 v4 v7 v8
E'val
e(0,1) e(1,2) e(2,5) e(5,8) e(0,3) e(3,6) e(6,8) e(4,7) e(7,8)

Fig. 9. Illustration of the storing of a logical chunk, which is assumed to contain three active graph paths of Figure 5 (i.e., L_0, L_1, and L_2).

path's usage changes. A path needs to be accessed and processed more number of times, when the beginning vertex of this path has incoming edges from more active vertices. Thus, the usage frequency $f(L_j)$ of each path L_j is approximately evaluated via $f(L_j) = n_j$, where n_j is the number of active vertices connected to the beginning vertex of this path (i.e., L_j). A vector is used to store the frequency of the usage of each path.

Then, at the beginning of each round, the paths with the highest usage frequency are dynamically put together to constitute the frequently used chunks until the user-specified number of the frequently used chunks is met, where these paths with the highest usage frequency are called the *frequently used paths*. Note that a parameter, such as α (which denotes the ratio of the number of the edges of all frequently used paths to that of the whole graph), is provided for the user to specify the number of the frequently used chunks. The suitable value of α is usually small, and we will discuss how to set it in Section 4.4. The total number of the frequently used chunks N_h is determined by the parameter α, namely $N_h = \frac{\alpha \cdot |E|}{|C|}$, where $|C|$ is the number of edges that can be contained in a chunk to be processed on the GPU. $|E|$ is the total number of edges. The number of the frequently used chunks that can be maintained in the global memory is about $N_c = \frac{S_G - S_R}{|C|}$. There, S_G denotes the space size of the GPU's global memory, whereas S_R denotes the global memory's space size reserved by LargeGraph to store the other graph data. When the value of N_h is larger than N_c, the frequently used chunks are swapped into the GPU for processing in batches.

For each logical chunk, similar to DiGraph [44], three arrays are used to efficiently store its graph data. The indexes of the vertices of its each path are sequentially stored in E'_{Idx} along their original order on this path. S'_{val} and E'_{val} are used to store the states of the vertices in E'_{Idx} and the values of the edges in this chunk, respectively. A table *ChunkTable* is also maintained for this chunk. For each path, this table has a field to maintain its beginning vertex's location in the array E'_{Idx} of this chunk. The range of each path is indicated by two successive entries of this field. An example is given in Figure 9 to show the storage of a logical chunk. Note that this storage scheme can ensure coalesced accesses for the parallel processing of the paths of the chunk on the GPU, because the paths to be handled by the threads of a warp are stored sequentially in the same chunk, which can fit into an SM.

After that, the logical chunks are dispatched to be processed over the CPU or GPU before the beginning of the current round. The frequently used chunks are inserted into the worklist W_f to only be processed by the GPU. The rarely used chunks are inserted into the worklist W_r to only be handled by the CPU.

3.2.3 *Dynamically Maintaining of the Frequently Used Chunks.*

The chunks with lower average usage frequency (i.e., the average usage frequency of a chunk's paths) may make the chunks with higher average usage frequency frequently swapped into and out of GPU. It incurs the thrashing of the chunks with higher average usage frequency. In addition, as discussed previously, the average usage frequency of each chunk changes with time. Thus, to further reduce the CPU-GPU data transfer cost, LargeGraph explicitly creates another worklist W_f^G in the GPU memory and tries to store the frequently used paths in W_f^G, although they become inactive at the execution time, because they may become active and need to be frequently accessed soon. An inactive chunk is swapped out of W_f^G of the GPU only when the GPU memory is full and the average usage frequency of its paths is also smaller than the average usage frequency of the paths of the chunk that needs be transferred to GPU to be processed by a warp of GPU threads. Note that only when there is no frequently used chunk on GPU to be handled by a warp of GPU threads (i.e., these GPU threads become idle), will frequently used chunk C_i be transferred to GPU to be processed by these threads. Specifically, after the processing of a frequently used logical chunk on an SM, the GPU threads on this SM will try to get the next frequently used logical chunk from this worklist (i.e., W_f^G) to process it. If no frequently used logical chunk can be obtained from W_f^G, these GPU threads then try to get a suitable frequently used logical chunk from the worklist W_f in the host memory and become idle until a frequently used logical chunk has been obtained from the host for processing.

3.2.4 Data-Driven Processing of Logical Chunks. As shown in Figure 6, the logical chunks in the worklist drive the execution of the CPU threads and the GPU threads to concurrently process them, respectively, in an asynchronous way. The process ends when there is no logical chunk in both of the two worklists. Because the graph paths in the frequently used chunks are repeatedly used to propagate most vertex states, many rounds of graph processing are needed for the convergence of these paths. Thus, in LargeGraph, before swapped out of the SM, each loaded frequently used chunk is repeatedly handled until all its vertices are inactive. By such means, most propagations in each chunk can quickly reach the other chunks through the frequently used chunks. In addition, it can improve the utilization of each loaded frequently used chunk.

3.2.5 Supporting of Unified Virtual Memory. Like Subway [31], LargeGraph can also be used when supporting **Unified Virtual Memory (UVM)**, through little modification of the implementation in comparison with the non-UVM version. In detail, the non-UVM version of LargeGraph explicitly creates another worklist W_f^G in the GPU memory to store the frequently used chunks that have been transferred to the GPU. In this way, it allows the non-UVM version of LargeGraph to efficiently manage these frequently used chunks buffered in the worklist W_f^G to avoid data thrashing. This is because the non-UVM version of LargeGraph tries to maintain the chunk with higher average usage frequency on the GPU, although it becomes inactive at the execution time, because it may become active and need to be frequently accessed soon. An inactive chunk is swapped out of the GPU only when the GPU memory is full and the average usage frequency of its paths is also smaller than the average usage frequency of the paths of the chunk to be swapped into the GPU memory. Compared with it, the UVM version of LargeGraph does not need to create the worklist W_f^G in the GPU memory. For the UVM version of LargeGraph, any frequently used chunk in the worklist W_f will be automatically migrated from the host to the GPU memory when this chunk needs to be processed by a warp of GPU threads. Because LRU-based page replacement policy is used in NVIDIA GPU, some frequently used chunks may be buffered in the GPU memory by default. However, for large-scale graph processing, it may incur data thrashing. Specifically, for the UVM version of LargeGraph, the chunks with lower average usage frequency (i.e., the average usage frequency of a chunk's paths) may make the chunks with higher average usage frequency swapped into and out of GPU frequently. Thus, it suffers from more data transfer cost than the non-UVM version.

Table 2. Dataset Properties

Datasets	*#Vertices*	*#Edges*	*#Graph Size*	A_{Deg}	A_{Dis}
uk2007	105.9 M	3.7 B	46.2 GB	35.3	8.2
uk-union	133.6 M	5.5 B	68.3 GB	41.2	5.8
gsh-2015	988.5 M	33.9 B	418.8 GB	34.3	7.1
clueweb12	978.4 M	42.5 B	522.0 GB	43.5	18.5
hyperlink14	1.7 B	64.4 B	793.4 GB	37.9	31.7

A_{Dis} is average distance between two vertices. A_{Deg} is average vertex degree.

4 EXPERIMENTAL EVALUATION

In the experiments, the platform has four 12-core Intel Xeon E5-2670 v3 CPUs and 256 GB of main memory on the host side, and a NVIDIA TESLA A100 GPU (which supports NVLink 2.0 and UVM) with 6,912 cores and 40 GB of on-board memory on the device side. Each CPU has two 9.6 GT/s QPI link and PCI Express 4.0 lanes operating at 16x speed. It uses four typical graph algorithms [44]: k-core, SSSP, adsorption, and PageRank. The program is compiled by CUDA v11.2.0, GCC v10.2, and Boost v1.75.0 using the -O3 flag. Five real-world graphs (i.e., uk2007 [3], uk-union [3], gsh-2015 [3], clueweb12 [1], and hyperlink14 [2]) are used. Table 2 summarizes the characteristic statistics of these graphs.

LargeGraph is compared with Totem [13, 14], Graphie [16], Garaph [29], and Subway [31]. They are the cutting-edge GPU-based systems for large-scale graph processing, and we have tuned the performance of them to be the best in the following experiments for the fairness of performance comparison. For Totem, Graphie, and Garaph, we directly run the graph algorithms on these systems, because the graph algorithms can directly get their best performance. Subway has different performance when its different strategies are selected for different graph algorithms' execution over various graphs. Thus, we select the setting leading to the best performance (i.e., execution strategy), such as synchronous processing scheme and asynchronous processing scheme. When tuning α for LargeGraph, we observe that α is mainly sensitive to the graphs instead of graph algorithms and its value is set to 35%, 25%, 5%, 5%, and 5% for uk2007, uk-union, gsh-2015, clueweb12, and hyperlink14, respectively, in the experiments. Its impact is evaluated later in Figure 17. To further understand LargeGraph, the other version of LargeGraph (i.e., *LargeGraph-w*) is evaluated as well. LargeGraph-w is the version of LargeGraph without maintaining the frequently used paths in the GPU. In addition, LargeGraph is also finally compared with SIMD-X-E, AsynGraph-E, and DiGraph-E, which are the versions of three cutting-edge GPU-based systems (i.e., SIMD-X [26], AsynGraph [40], and DiGraph [44]) extended to support out-of-GPU-memory graph processing using Gluon-Async [10], respectively. Ten repetitive runs are conducted, and we then obtain the average value for the following reported results.

4.1 Preprocessing Cost

We first compare the preprocessing cost of different systems for various real-world graphs. As described in Figure 10, the preprocessing time of LargeGraph is little more than that of the other systems, because LargeGraph needs a little more additional cost to divide the graph into the paths. However, as shown in the figure, after such a graph preprocessing in LargeGraph, smaller total execution time is required than that in the other systems for the convergence of iterative execution of the graph algorithms. This is because of the much fewer number of updates, much smaller cost to access the graph data, and higher CPU/GPU utilization than other systems, as we will see in the later experimental results. Note that for static graph processing, the preprocessing time is

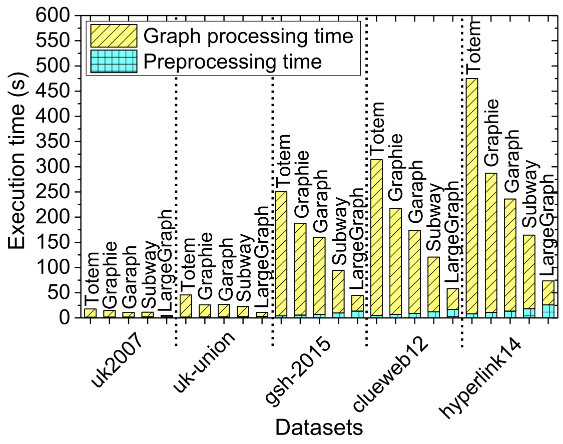


Fig. 10. Execution time breakdown of PageRank.

one-time cost [44] and the graph preprocessing results can be repeatedly used. The storage cost of the graph representation [44] used in LargeGraph is only 95.8%, 97.7%, 95.2%, 98.1%, and 97.5% of that of CSR [31] for uk2007, uk-union, gsh-2015, clueweb12, and hyperlink14, respectively.

4.2 Convergence Speed

To demonstrate the preceding discussions, this section first evaluates the number of vertex state updates needed by various iterative graph algorithms on Totem, Graphie, Garaph, Subway, and LargeGraph. From Figure 11, we have the following three observations.

First, Totem requires much fewer vertex state updates than Graphie for all conditions. Totem even only requires 74.1% of the vertex state updates needed by Graphie for the convergence of SSSP over uk2007, because Totem is able to propagate vertex states in a faster way than Graphie through asynchronously updating each vertex's state based on other vertices' new states.

Second, more vertex state updates still need to be handled by Totem than by LargeGraph, because much unnecessary vertex processing exists in Totem. The number of vertex state updates of LargeGraph is even 22.0% of that of Totem for SSSP over uk2007. It is because LargeGraph achieves faster state propagation through our dependency-aware data-driven execution approach.

Third, LargeGraph usually achieves better performance when A_{Dis} of the graph is larger. For SSSP, LargeGraph only needs 22.0% of the vertex state updates required by Totem on uk2007, whereas it is 38.2% of that of Totem on gsh-2015, where A_{Dis} of uk2007 is larger than that of gsh-2015. This is because that new vertex state in Totem, Graphie, Garaph, or Subway cannot be effectively propagated along graph paths. Note that Totem, Graphie, Garaph, and Subway mainly try to reduce data access cost and ensure balanced load, instead of reducing the number of updates, and thus different runtime characteristics of various graph algorithms do not have much impact on their normalized number of updates.

4.3 Communication and Utilization Ratio

Figure 12 shows that LargeGraph needs the smallest data access time for three reasons. First, fewer updates need to be handled by LargeGraph than the other systems due to faster vertex state propagation. Second, LargeGraph has better locality via processing vertices along graph paths and also efficiently maintains the frequently used paths in the GPU. Third, LargeGraph only needs to load and transfer the graph data associated with active vertices' state propagations.

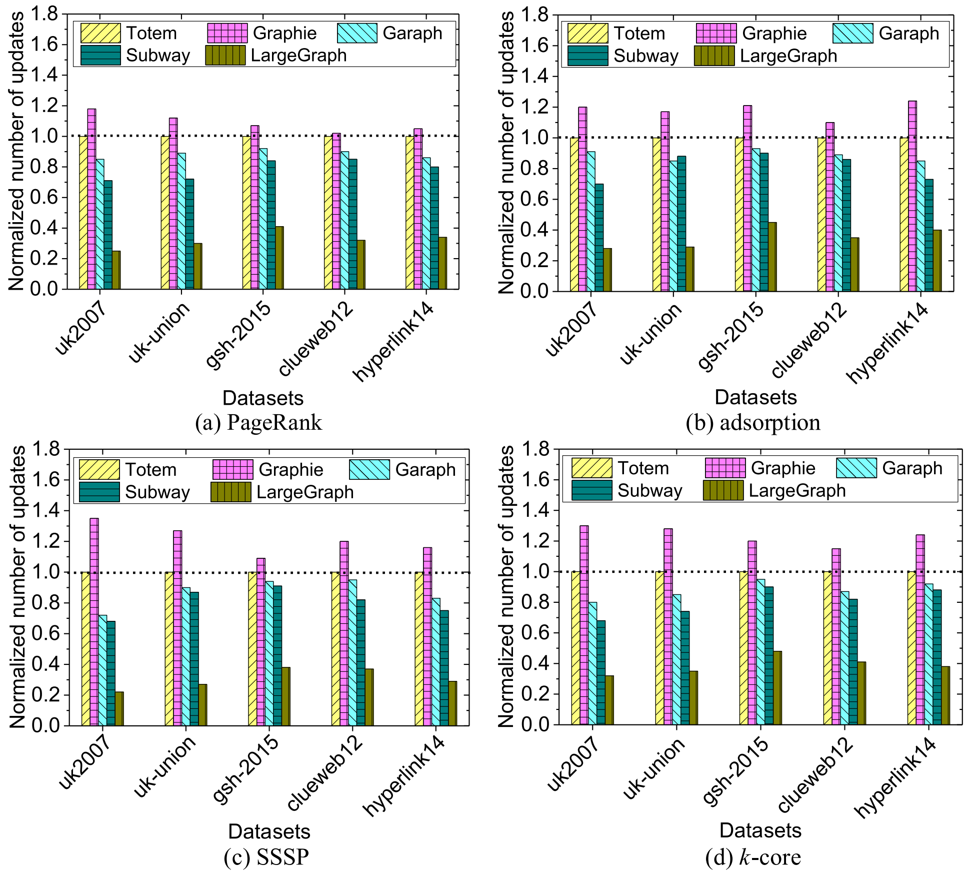


Fig. 11. Number of updates normalized to that of Totem.

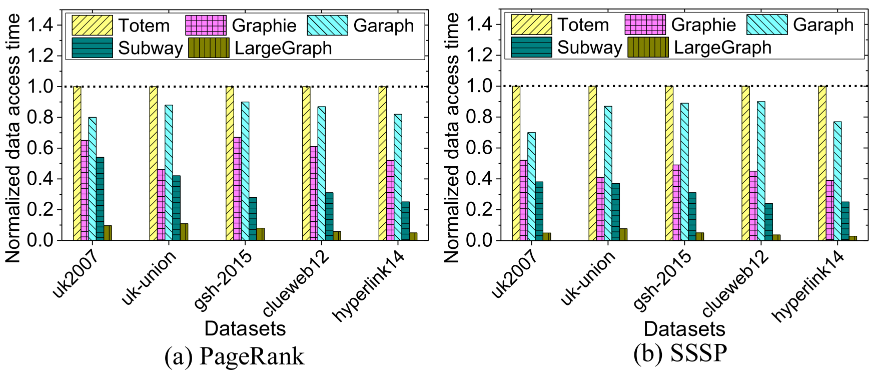


Fig. 12. Normalized data access time on various systems.

To evaluate how much benefit can be gotten by the approach to maintain the frequently used paths in the GPU accelerator, Figure 13 shows the graph processing time of LargeGraph and LargeGraph-w. We can observe that LargeGraph can effectively accelerate LargeGraph-w due to higher utilization of the transferred paths. For example, for PageRank over hyperlink14,

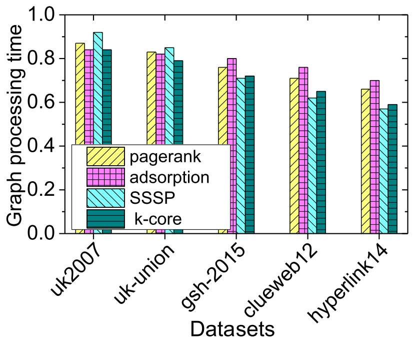


Fig. 13. Processing time of LargeGraph normalized to that of LargeGraph-w.

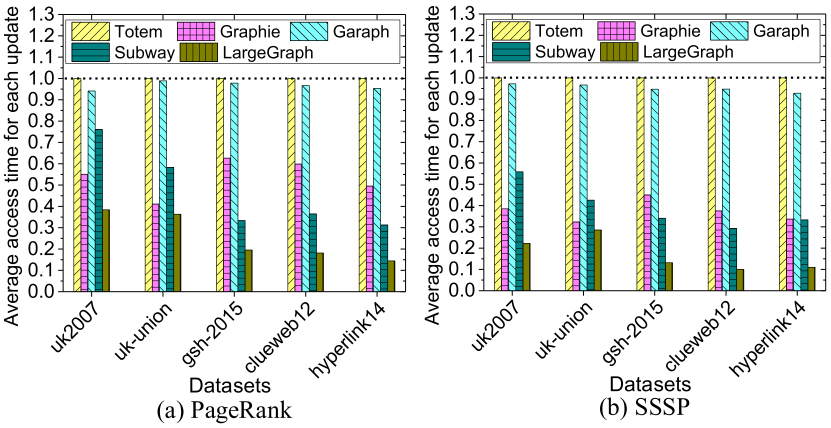


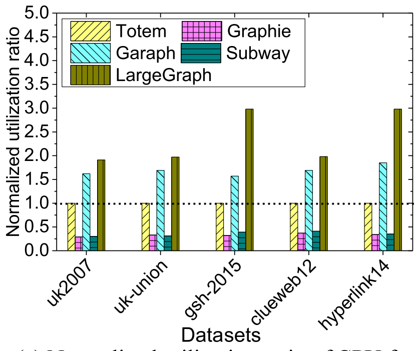
Fig. 14. Data access time of various systems to conduct each update normalized to that of Totem.

LargeGraph spares LargeGraph-w's graph processing time by 34.0%. Figure 14 shows that smaller average data access time is required by LargeGraph to conduct each update than the other systems due to better spatial and temporal locality.

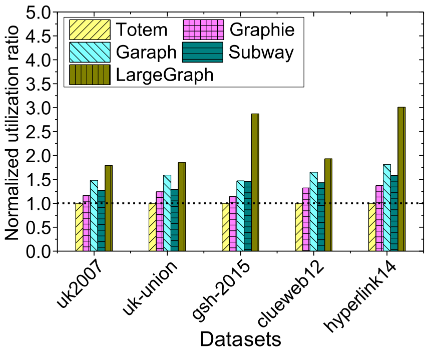
Figure 15 shows the average utilization ratio of all CPU cores and all GPU cores, respectively, by executing PageRank over different systems. From Figure 15, we can observe that both the CPU and GPU utilization ratio of Garaph is higher than that of Totem, Graphie, and Subway. Taking hyperlink14 as an example, the GPU utilization ratio of Garaph is 1.81, 1.32, and 1.15 times as high as that of Totem, Graphie, and Subway, respectively. This is because Garaph can effectively adjust the workload between the CPUs and the GPU. In addition, we can find that both the CPU and GPU utilization ratio of LargeGraph is higher than that of the other GPU-based graph processing sytems, because LargeGraph has smaller data access cost than the other systems.

4.4 Performance Comparison

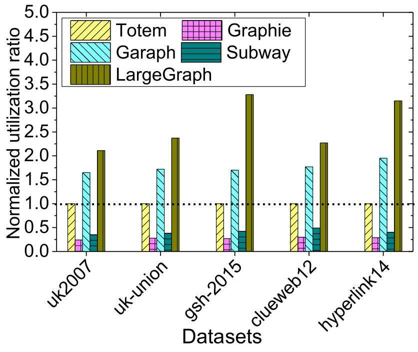
We first evaluate the graph processing time of various iterative graph algorithms over different systems. The speedups of Totem, Graphie, Garaph, Subway, and LargeGraph against Totem are given in Figure 16. Note that over uk2007, uk-union, gsh-2015, clueweb12, and hyperlink14, the graph processing time of the baseline system (i.e., Totem) is 16.3, 43.9, 246.2, 308.8, and 467.0 seconds for PageRank, is 30.9, 89.1, 494.8, 611.4, and 962.4 seconds for adsorption, is 24.9, 33.8, 164.2, 220.0, and 360.3 seconds for SSSP, and is 36.0, 103.1, 539.2, 728.7, and 1,064.7 seconds for *k*-core, respectively.



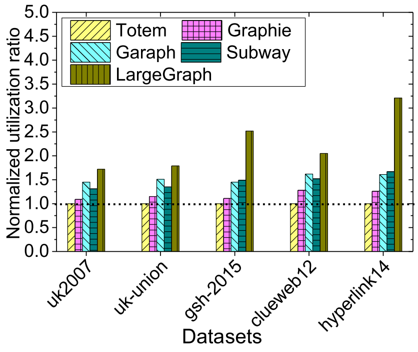
(a) Normalized utilization ratio of CPU for PageRank



(b) Normalized utilization ratio of GPU for PageRank



(c) Normalized utilization ratio of CPU for SSSP



(d) Normalized utilization ratio of GPU for SSSP

Fig. 15. Average utilization normalized to that of Totem.

From Figure 16, we can observe that Garaph usually performs better than both Totem and Graphie, because of more balanced load and higher utilization ratio of the parallelism of both the CPU and the GPU. For example, for k-core over uk2007, Garaph reduces the graph processing time by 55.7% and 42.0% in comparison with Totem and Graphie, respectively. However, Garaph needs to repeatedly transfer many unnecessary graph data between the GPU and CPU. Compared with it, Subway is able to avoid the transfer of many unnecessary graph data between the GPU and CPU. Consequently, better performance is always obtained by Subway than Totem, Graphie, and Garaph.

However, compared with LargeGraph, Subway needs to process more updates and needs more overhead to load and to transfer the graph data required for these updates. Thus, Subway needs more time to converge than LargeGraph. For example, for SSSP over uk2007, LargeGraph can reduce the graph processing time by 75.9% in comparison with Subway. Compared with Totem, Graphie, Garaph, and Subway, LargeGraph achieves performance improvements of 5.19–11.62, 3.02–9.41, 2.75–8.36, and 2.45–4.15 times for different cases, respectively. Note that the runtime cost of LargeGraph occupies 6.92% to 18.15% of its graph processing time for different cases. The space for LargeGraph to store its auxiliary data (e.g., $ChunkTable$) occupies 1.29% to 5.63% of the total storage cost for the tested graphs. In addition, we can find that LargeGraph usually performs much better when A_{Dis} is larger. Note that LargeGraph does not work when A_{Dis} of a graph (e.g., bipartite graph) is 1. In addition, for different cases, our results show that LargeGraph outperforms LargeGraph-only-CPU (i.e., the version of LargeGraph only using CPU) and LargeGraph-only-GPU (i.e., the version of LargeGraph only using GPU) by 1.7–8.2 and 2.4–30.6 times, respectively.

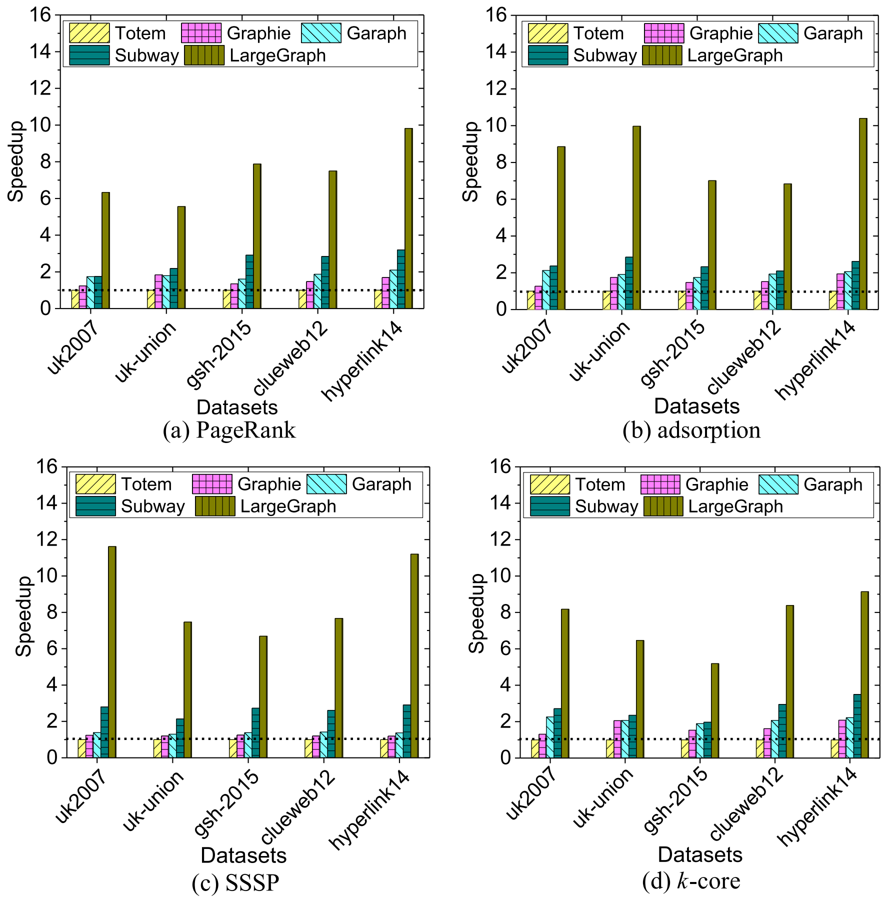


Fig. 16. Speedups of various systems against Totem.

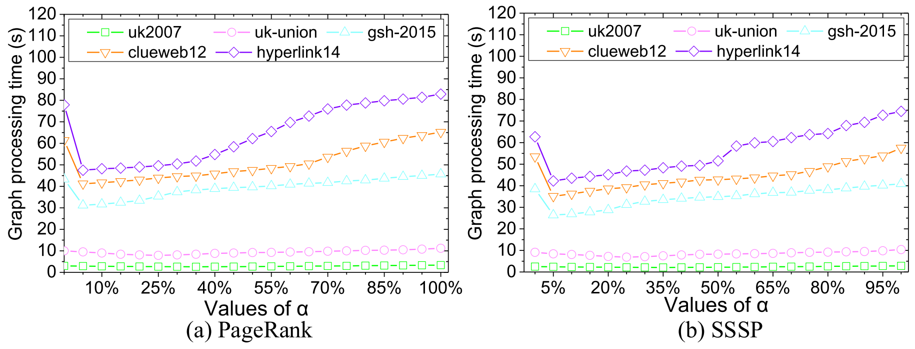
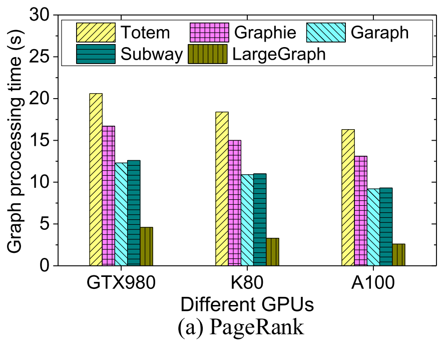


Fig. 17. Impacts of α on LargeGraph.

This means that LargeGraph gets better performance/energy efficiency than the way only using CPU as well as the way only using GPU under most circumstances.

The impacts of α on the performance of PageRank over LargeGraph is depicted in Figure 17. We have two observations from the results. First, LargeGraph always gets better performance with the



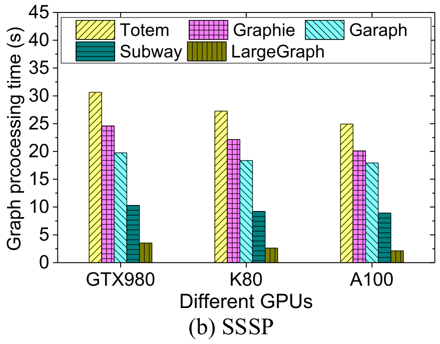


Fig. 18. Performance on various GPUs for uk2007.

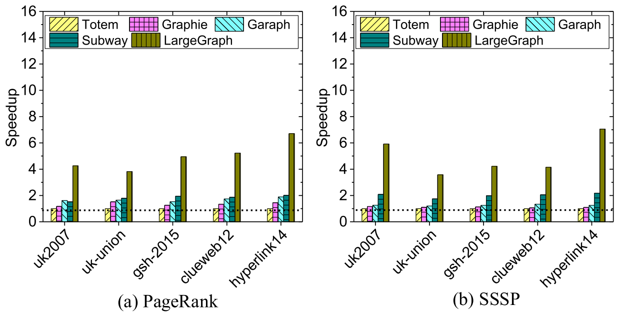


Fig. 19. Speedups against Totem when using NVLink 2.0.

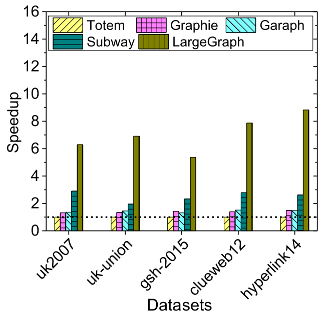


Fig. 20. Performance of BFS.

increasing of α's value. LargeGraph's performance becomes worse as α reaches an upper bound. This means that worse performance is gotten as the value of α is too small or too large. This is because too small α incurs the inefficient processing of many frequently used paths. Too large α also induces high runtime cost. For different cases, the overall runtime overhead occupies 4.9% to 40.4% of the graph processing time, where the overhead of tracking of the frequently used paths occupies 13.5% to 91.1% of the overall runtime overhead. Second, α should be set small. These observations can be used to guide the selection of the suitable value of α. Note that better performance is still achieved by LargeGraph than existing systems without making efforts to select the suitable α. In comparison with Subway, LargeGraph can still achieve performance improvements of 1.66–2.91 times for different graphs at the worst values of α.

Figure 18 describes the performance of PageRank on various systems for uk2007 when various generations of GPUs are plugged into the platform. We can observe that LargeGraph outperforms other systems when more powerful GPU is used. The graph processing time of PageRank is even spared 44.2% by LargeGraph as the plugged GPU is changed from NVIDIA GeForce GTX980 to NVIDIA TESLA A100, whereas Totem, Graphie, Garaph, and Subway can only reduce 20.8%, 21.4%, 23.8%, and 25.9%, respectively. This is because LargeGraph still ensures a higher effective GPU utilization ratio than them when more powerful GPU is used.

We also evaluate the performance of LargeGraph on the platform where the CPUs and GPU are connected with each other through NVLink 2.0. As shown in Figure 19, LargeGraph still performs better than the other solutions. Figure 20 also evaluates the performance of BFS [5] on different systems, and we can observe that LargeGraph still outperforms the other systems.

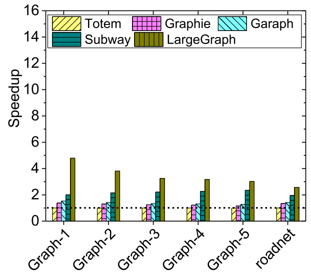


Fig. 21. Performance on the roadnet and five synthetic graphs.

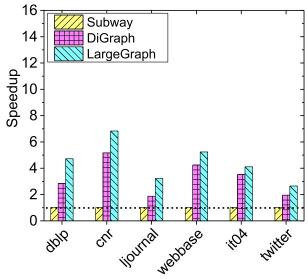


Fig. 22. Performance on the graphs fitting into the GPU memory.

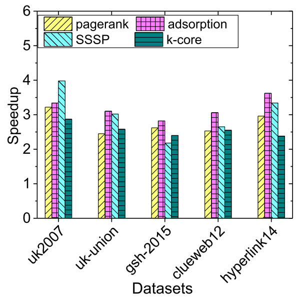


Fig. 23. Speedup against Subway when using UVM.

Table 3. Synthetic Power-Law Graphs with 1 Billion Vertices and Different β [15]

Datasets	Graph-1	Graph-2	Graph-3	Graph-4	Graph-5
β	1.8	1.9	2.0	2.1	2.2
#Edges	66.7B	24.6B	10.4B	5.6B	3.7B

Table 4. Preprocessing and Graph Processing Time (Seconds)

Datasets	SIMD-X-E	AsynGraph-E	DiGraph-E	LargeGraph
uk2007	1.9 (22.0)	2.7 (9.8)	2.6 (8.7)	2.5 (2.6)
uk-union	1.8 (41.3)	3.6 (23.4)	3.4 (20.5)	3.2 (7.9)
gsh-2015	5.1 (212.1)	16.5 (90.8)	16.1 (82.9)	13.6 (31.2)
clueweb12	6.5 (229.6)	20.1 (115.2)	18.9 (108.4)	17.0 (41.2)
hyperlink14	10.4 (381.7)	31.7 (165.5)	29.2 (155.1)	25.9 (47.5)

Figure 21 evaluates the performance of PageRank on roadnet[2] [4] and also synthetic graphs (Table 3). The results show that when β is smaller, LargeGraph has much better performance. In addition, LargeGraph still outperforms other systems for the graph without power-law property (i.e., roadnet) because LargeGraph enables faster state propagation and better locality.

From Figure 22, we can also find that LargeGraph still performs better than the other systems (e.g., Subway and DiGraph) when the graphs (which are the same graphs used in DiGraph [44]) fit into the GPU's global memory. This is because LargeGraph enables faster propagation and higher GPU utilization. Figure 23 shows the performance of the UVM version of LargeGraph in comparison with that of Subway on our platform using UVM. We can find that the UVM version of LargeGraph also performs better than Subway due to efficient state propagation, although its performance is poorer than that of LargeGraph (which is the non-UVM version of LargeGraph by default) due to data thrashing.

Finally, Table 4 also shows the performance of PageRank on LargeGraph in comparison with SIMD-X-E, AsynGraph-E, and DiGraph-E. As shown in this table, LargeGraph performs better than these systems. This is because LargeGraph enables much smaller data access cost than them. Note that DiGraph-E can spare many updates through dividing the graph into static graph paths at the preprocessing stage and asynchronously processing the graph along these static paths. However, due to much more accesses of the graph data associated with inactive vertices than

[2]Roadnet is a graph without power-law property and can fit into the GPU. It has 1,965,206 vertices and 2,766,607, 849 edges.

Subway, DiGraph-E even performs worse than Subway for out-of-GPU-memory graph processing. This is because a static path of DiGraph-E has to be loaded, although this static path may only have an active vertex.

5 RELATED WORK

For higher performance of graph processing, Medusa [54] is proposed as the first graph processing system using the powerful capacity of the GPU and can simplify the programming of iterative graph algorithms over GPUs by providing several simple APIs. For efficient utilization of the GPU's resources, CuSha [20] designs two methods to represent graph, whereas MultiGraph [17] uses suitable graph representation and processing strategies accordingly. For better performance, Gunrock [30, 37] uses a novel frontier-based approach, whereas Groute [5] proposes a GPU-based asynchronous execution and communication approach for lower synchronization cost. For faster convergence of asynchronous iterative graph processing, DiGraph [44] recently proposes a path-centric execution approach, and AsynGraph [40] further proposes a graph-sketch-based approach to optimize DiGraph through maximizing the data parallelism of iterative graph processing on the GPU accelerator. For balanced load over GPU cores, Tigr [18] transforms an irregular graph into a more regular one, and SIMD-X [26] intelligently maps the tasks using just-in-time task management. However, these systems fail to work when the GPU's global memory cannot hold the whole graph.

To efficiently support iterative processing of large-scale graphs that have larger size than the capacity of the GPU's global memory, several GPU-based graph processing systems have been recently designed, yet also following the ideas of out-of-core graph processing [23, 55]. Gluon-Async [10] can be used to extend existing GPU graph processing systems for heterogeneous execution. GTS [22] stores large-scale graphs in PCI-E SSDs and streams graph structure data to GPUs. Totem [13, 14] proposes to deal with large-scale graphs using both the CPU and GPU. Garaph [29] dynamically schedules the works between the CPU and GPU for load balancing. GraphReduce [32] employs asynchronous streams to reduce the negative impacts of data movement between the CPU and the GPU. Meanwhile, for smaller CPU-GPU communication cost, GraphReduce and Graphie [16] also track active partitions and only load these active ones to the GPU for processing. Note that the edge data in Graphie are also streamed into the GPU in an asynchronous way, whereas vertex states are stored in the GPU. To further spare the CPU-GPU data transfer cost, Subway [31] only loads the required data into GPU global memory.

Meanwhile, many hardware and software approaches are proposed for efficient large-scale graph processing in the UVM model. Kim et al. [21] propose a software-hardware cooperation approach to enable a group of GPU page faults efficiently amortize the high CPU-GPU transfer cost for unified memory management in GPUs. Ganguly et al. [11] propose to leverage hardware-based access counters for smaller page thrashing overhead. However, it is based on the information profiled from the previous access behaviors of applications and still suffers from serious page thrashing, because the set of active vertices of graph processing change with time. In addition, it also suffers from high CPU-GPU data transfer cost associated with the updates based on stale states. ATMem [7] is a data placement optimization framework to enable adaptive data placement on a heterogeneous memory system for graph algorithms. It uses hardware-counter based sampling to profile information from the previous access behaviors of graph applications and then identifies and migrates the critical regions inside data structures according to these profiled information. Because the set of active vertices of graph processing change with time and many data accesses are associated with the updates based on stale states, ATMem also suffers from high data access cost as does the preceding work [11]. HALO [12] is a lightweight offline graph reordering algorithm to improve data locality for large-scale graph processing on GPUs with unified memory. However, HALO also suffers from high CPU-GPU transfer cost, because many unnecessary graph data (e.g., the graph data associated with the inactive vertices as well as the updates based on stale states) are transferred.

Although getting high performance for large-scale graph processing, these systems still suffer from slow convergence speed because of slow propagations of active vertices' states and a long time to frequently transfer the graph data between the CPU and GPU to propagate active vertices' states along the graph paths. Note that GpSM [36] proposes an specific approach to optimize the STW method [35] for efficient subgraph matching on GPU via pruning the irrelevant candidate vertices that cannot contribute to subgraph matching so as to reduce intermediate results; however, this is not suitable to general graph algorithms.

6 CONCLUSION

This article proposes a dependency-aware GPU-accelerated system *LargeGraph* for faster convergence speed of large-scale graph processing. LargeGraph efficiently accesses and asynchronously processes the graph data according to the paths originated from the active vertices. By doing so, only the graph data associated with the paths originated from the active vertices are loaded and efficiently transferred between the CPU and GPU, minimizing the data access cost. Through efficiently maintaining and processing the frequently used paths on the GPU, LargeGraph further accelerates the active vertices' state propagations with lower data access cost and higher parallelism. The results demonstrate that LargeGraph converges faster than the cutting-edge out-of-GPU-memory graph processing systems.

REFERENCES

[1] Lemur. 2020. ClueWeb12 Web Graph. Retrieved August 17, 2021 from http://www.lemurproject.org/clueweb12/webgraph.php/.

[2] Web Data Commons. 2020. Hyperlink Graphs. Retrieved August 17, 2021 from http://webdatacommons.org/hyperlinkgraph/.

[3] Laboratory for Web Algorithmics. 2020. Datasets. Retrieved August 17, 2021 from http://law.di.unimi.it/datasets.php.

[4] Stanford. 2020. Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/index.html.

[5] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An asynchronous multi-GPU programming model for irregular computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 235–248.

[6] Hanhua Chen, Hai Jin, and Xiaolong Cui. 2017. Hybrid followee recommendation in microblogging systems. *Science China Information Sciences* 60, 1 (2017), 1–14.

[7] Yu Chen, Ivy B. Peng, Zhen Peng, Xu Liu, and Bin Ren. 2020. ATMem: Adaptive data placement in graph applications on heterogeneous memories. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 293–304.

[8] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. 2016. NXgraph: An efficient graph processing system on a single machine. In *Proceedings of the 2016 IEEE International Conference on Data Engineering*. 409–420.

[9] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 752–768.

[10] R. Dathathri, G. Gill, L. Hoang, V. Jatala, K. Pingali, V. K. Nandivada, H. Dang, and M. Snir. 2019. Gluon-async: A bulk-asynchronous system for distributed and heterogeneous graph analytics. In *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques*. 15–28.

[11] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2020. Adaptive page migration for irregular data-intensive applications under GPU memory oversubscription. In *Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium*. 451–461.

[12] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David A. Bader. 2020. Traversing large graphs on GPUs with unified memory. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1119–1133.

[13] Abdullah Gharaibeh, Lauro Beltro Costa, Elizeu Santos-Neto, and Matei Ripeanu. 2012. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. 345–354.

[14] Abdullah Gharaibeh, Tahsin Reza, Elizeu Santosneto, Lauro Beltrao Costa, Scott Sallinen, and Matei Ripeanu. 2014. Efficient large-scale graph processing on hybrid CPU and GPU systems. arXiv:1312.3018.

[15] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. 17–30.

[16] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. 2017. Graphie: Large-scale asynchronous graph traversals on just a GPU. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques*. 233–245.

[17] Changwan Hong, Aravind Sukumaranrajam, Jinsung Kim, and P. Sadayappan. 2017. MultiGraph: Efficient graph processing on GPUs. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques*. 27–40.

[18] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming irregular graphs for GPU-friendly graph processing. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. 622–636.

[19] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A distributed multi-GPU system for fast graph processing. *Proceedings of the VLDB Endowment* 11, 3 (2017), 297–310.

[20] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: Vertex-centric graph processing on GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*. 239–252.

[21] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. 2020. Batch-aware unified memory management in GPUs for irregular workloads. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 1357–1370.

[22] Min Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. 2016. GTS: A fast and scalable graph processing method based on streaming topology to GPUs. In *Proceedings of the 2016 International Conference on Management of Data*. 447–461.

[23] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. 31–46.

[24] Xiaofei Liao, Yicheng Chen, Yu Zhang, Hai Jin, Haikun Liu, and Jin Zhao. 2019. An efficient incremental strongly connected components algorithm for evolving directed graphs. *Scientia Sinica Informationis* 49, 8 (2019), 988–1004.

[25] Xiaofei Liao, Jin Zhao, Yu Zhang, Bingsheng He, Ligang He, Hai Jin, and Lin Gu. 2021. A structure-aware storage optimization for out-of-core concurrent graph processing. *IEEE Transactions on Computers*. Early access, July 26, 2021. DOI: https://doi.org/10.1109/TC.2021.3098976

[26] Hang Liu and H. Howie Huang. 2019. SIMD-X: Programming and processing of graph algorithms on GPUs. In *Proceedings of the 2019 USENIX Annual Technical Conference*. 411–428.

[27] Wei Liu, Haikun Liu, Xiaofei Liao, Hai Jin, and Yu Zhang. 2019. NGraph: Parallel graph processing in hybrid memory systems. *IEEE Access* 7 (2019), 103517–103529.

[28] Xinqiao Lv, Wei Xiao, Yu Zhang, Xiaofei Liao, Hai Jin, and Qiangsheng Hua. 2019. An effective framework for asynchronous incremental graph processing. *Frontiers of Computer Science* 13, 3 (2019), 539–551.

[29] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. 2017. Garaph: Efficient GPU-accelerated graph processing on a single machine with balanced replication. In *Proceedings of the 2017 USENIX Annual Technical Conference*. 195–207.

[30] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D. Owens. 2017. Multi-GPU graph analytics. In *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium*. 479–490.

[31] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: Minimizing data transfer during out-of-GPU-memory graph processing. In *Proceedings of the 15th European Conference on Computer Systems*. Article 12, 16 pages.

[32] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. 2015. GraphReduce: Processing large-scale graphs on accelerator-based systems. In *Proceedings of the 2015 International Conference for High Performance Computing, Networking, Storage, and Analysis*. Article 28, 12 pages.

[33] Zhiyuan Shao, Lin Hou, Yan Ai, Yu Zhang, and Hai Jin. 2015. Is your graph algorithm eligible for nondeterministic execution? In *Proceedings of the 44th International Conference on Parallel Processing*. 430–439.

[34] Beibei Si, Yuxuan Liang, Jin Zhao, Yu Zhang, Xiaofei Liao, Hai Jin, Haikun Liu, and Lin Gu. 2020. GGraph: An efficient structure-aware approach for iterative graph processing. *IEEE Transactions on Big Data*. Early access, August 26, 2020. DOI: https://doi.org/10.1109/TBDATA.2020.3019641

[35] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment* 5, 9 (2012), 788–799.

[36] Ha-Nguyen Tran, Jung Jae Kim, and Bingsheng He. 2015. Fast subgraph matching on large graphs using graphics processors. In *Proceedings of the 20th International Conference on Database Systems for Advanced Applications*. 299–315.

[37] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Article 11, 12 pages.

[38] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2014. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *IEEE Transactions on Parallel and Distributed Systems* 25, 8 (2014), 2091–2100.

[39] Yu Zhang, Lin Gu, Xiaofei Liao, Hai Jin, Deze Zeng, and Bing Bing Zhou. 2017. FRANK: A fast node ranking approach in large-scale networks. *IEEE Network* 31, 1 (2017), 36–43.

[40] Yu Zhang, Xiaofei Liao, Lin Gu, Hai Jin, Kan Hu, Haikun Liu, and Bingsheng He. 2020. AsynGraph: Maximizing data parallelism for efficient iterative graph processing on GPUs. *ACM Transactions on Architecture and Code Optimization* 17, 4 (2020), Article 29, 21 pages.

[41] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. 2018. CGraph: A correlations-aware approach for efficient concurrent iterative graph processing. In *Proceedings of the 2018 USENIX Annual Technical Conference*. 441–452.

[42] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Guang Tan, and Bing Bing Zhou. 2017. HotGraph: Efficient asynchronous processing for real-world graphs. *IEEE Transactions on Computers* 66, 5 (2017), 799–809.

[43] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, and Bing Bing Zhou. 2018. FBSGraph: Accelerating asynchronous graph processing via forward and backward sweeping. *IEEE Transactions on Knowledge and Data Engineering* 30, 5 (2018), 895–907.

[44] Yu Zhang, Xiaofei Liao, Hai Jin, Bingsheng He, Haikun Liu, and Lin Gu. 2019. DiGraph: An efficient path-based iterative directed graph processing system on multiple GPUs. In *Proceedings of the 2019 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 601–614.

[45] Yu Zhang, Xiaofei Liao, Hai Jin, Ligang He, Bingsheng He, Haikun Liu, and Lin Gu. 2021. DepGraph: A dependency-driven accelerator for efficient iterative graph processing. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*. 371–384.

[46] Yu Zhang, Xiaofei Liao, Hai Jin, Li Lin, and Feng Lu. 2014. An adaptive switching scheme for iterative computing in the cloud. *Frontiers of Computer Science* 8, 6 (2014), 872–884.

[47] Yu Zhang, Xiaofei Liao, Hai Jin, and Geyong Min. 2015. Resisting skew-accumulation for time-stepped applications in the cloud via exploiting parallelism. *IEEE Transactions on Cloud Computing* 3, 1 (2015), 54–65.

[48] Yu Zhang, Xiaofei Liao, Hai Jin, and Guang Tan. 2017. SAE: Toward efficient cloud data analysis service for large-scale social networks. *IEEE Transactions on Cloud Computing* 5, 3 (2017), 563–575.

[49] Yu Zhang, Xiaofei Liao, Hai Jin, Guang Tan, and Geyong Min. 2015. Inc-Part: Incremental partitioning for load balancing in large-scale behavioral simulations. *IEEE Transactions on Parallel and Distributed Systems* 26, 7 (2015), 1900–1909.

[50] Yu Zhang, Xiaofei Liao, Hai Jin, and Bing Bing Zhou. 2014. AsyIter: Tolerating computational skew of synchronous iterative applications via computing decomposition. *Knowledge and Information Systems* 41, 2 (2014), 379–400.

[51] Yu Zhang, Xiaofei Liao, Xiang Shi, Hai Jin, and Bingsheng He. 2018. Efficient disk-based directed graph processing: A strongly connected component approach. *IEEE Transactions on Parallel and Distributed Systems* 29, 4 (2018), 830–842.

[52] Yu Zhang, Jin Zhao, Xiaofei Liao, Hai Jin, Lin Gu, Haikun Liu, Bingsheng He, and Ligang He. 2019. CGraph: A distributed storage and processing system for concurrent iterative graph analysis jobs. *ACM Transactions on Storage* 15, 2 (2019), Article 10, 26 pages.

[53] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, and Yicheng Chen. 2019. GraphM: An efficient storage system for high throughput of concurrent graph processing. In *Proceedings of the 2019 International Conference for High Performance Computing, Networking, Storage, and Analysis*. Article 3, 14 pages.

[54] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1543–1552.

[55] Willy Zwaenepoel, Willy Zwaenepoel, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. 472–488.

Received January 2021; revised June 2021; accepted July 2021