AsynGraph: Maximizing Data Parallelism for Efficient Iterative Graph Processing on GPUs

YU ZHANG, XIAOFEI LIAO, LIN GU, HAI JIN, KAN HU, and HAIKUN LIU, 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

Recently, iterative graph algorithms are proposed to be handled by GPU-accelerated systems. However, in iterative graph processing, the parallelism of GPU is still underutilized by existing GPU-based solutions. In fact, because of the power-law property of the natural graphs, the paths between a small set of important vertices (e.g., high-degree vertices) play a more important role in iterative graph processing's convergence speed. Based on this fact, for faster iterative graph processing on GPUs, this article develops a novel system, called *AsynGraph*, to maximize its data parallelism. It first proposes an efficient *structure-aware asynchronous processing way*. It enables the state propagations of most vertices to be effectively conducted on the GPUs in a concurrent way to get a higher GPU utilization ratio through efficiently handling the paths between the important vertices. Specifically, a graph sketch (consisting of the paths between the important vertices) is extracted from the original graph to serve as a fast bridge for most state propagations. Through efficiently processing this sketch more times within each round of graph processing, higher parallelism of GPU can be utilized to accelerate most state propagations. In addition, a *forward-backward intra-path processing way* is also adopted to asynchronously handle the vertices on each path, aiming to further boost propagations along paths and also ensure smaller data access cost. In comparison with existing GPU-based systems, i.e., Gunrock, Groute, Tigr, and DiGraph, AsynGraph can speed up iterative graph processing by 3.06–11.52, 2.47–5.40, 2.23–9.65, and 1.41–4.05 times, respectively.

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

Additional Key Words and Phrases: GPU, graph processing, data parallelism, convergence speed

This paper is supported by National Key Research and Development Program of China under grant No. 2018YFB1003500, National Natural Science Foundation of China under grant No. 61832006, 61825202, 61702202, and 61929103. This work is also supported by Science and Technology on Parallel and Distributed Processing Laboratory (PDL).

Authors' addresses: Y. Zhang, X. Liao (corresponding author), L. Gu, H. Jin, K. Hu, and H. Liu, 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, lingu, hjin, hukan, hkliu}@hust.edu.cn; B. He, National University of Singapore, Singapore; email: hebs@comp.nus.edu.sg.

1544-3566/2020/09-ART29
https://doi.org/10.1145/3416495

ACM Reference format:
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 Trans. Archit. Code Optim.* 17, 4, Article 29 (September 2020), 21 pages.
https://doi.org/10.1145/3416495

1 INTRODUCTION

Graphs widely exist in real-world applications, hence many iterative algorithms [6, 22, 36] are designed to handle these graphs iteratively until the user given condition is met. In order to speed up these time-consuming iterative graph algorithms, many GPU-based systems are designed aimed at ensuring coalesced memory accesses [15], better data locality [15, 35], balanced load [12, 20], fewer updates [4, 35], and less communication cost [4], and so on.

Despite of many previous studies on GPU-based graph processing, there are still two challenges in existing GPU-based solutions [4, 12, 15, 29, 35] because there are many irregular dependencies between graph vertices. First, when these vertices are concurrently processed by existing solutions, stale vertex state may be read by many GPU threads to conduct useless updates. Second, many vertices are also inactive and incur low parallelism of GPU threads. Consequently, the high parallelism of GPUs is underutilized for faster convergence of iterative graph processing. This article investigates whether and how we can efficiently and fully exploit the high parallelism of GPUs to accelerate the propagations of vertices' states in iterative graph processing for faster convergence speed.

In practice, due to the power-law property [8] of natural graph, a small set of high-degree vertices are connected with much more vertices than the others and thus the paths between a few vertices play a more important role on vertex state propagations. It allows us to boost iterative graph processing's convergence speed on GPUs through boosting state propagations on a small set of paths, i.e., the paths between some important vertices (e.g., high-degree vertices). Based on this fact, this article develops an efficient system *AsynGraph* for iterative graph algorithms' faster convergence speed on the GPU. Different from the prior solutions, it uses an efficient *structure-aware asynchronous execution approach* to fully exploit the high parallelism of the GPU to accelerate most state propagations. Specifically, this execution approach represents the graph into paths. Those paths between important vertices are put together as a small important subgraph, called *graph sketch*, so as to open an opportunity to efficiently accelerate most state propagations on the GPU. Through efficiently processing the graph paths in the small graph sketch multiple times within each round of graph processing, most vertices are able to propagate their states to the others in a faster way and also enable more vertices to be active quickly to use higher parallelism of the GPU to conduct vertex state updates.

For faster intra-path state propagations, based on the order of each path's vertices, these vertices are asynchronously processed in a forward-backward way. Then, for each graph path, the state of its each vertex is efficiently propagated to its other vertices with fewer useless updates and better data locality. To validate the effectiveness of our proposal, AsynGraph is compared with four cutting-edge GPU-based systems, i.e., Gunrock [29], Groute [4], Tigr [12], and DiGraph [35], and achieves improvements of 3.06–11.52, 2.47–5.40, 2.23–9.65, and 1.41–4.05 times, respectively.

The rest of this article is organized as follows: Section 2 discusses the background and the challenges of existing solutions. Section 3 and Section 4 present our proposed asynchronous execution approach and describe the details of our system AsynGraph, respectively, followed by comprehensive experimental evaluation in Section 5. The related work is depicted in Section 6. We conclude this article in Section 7.

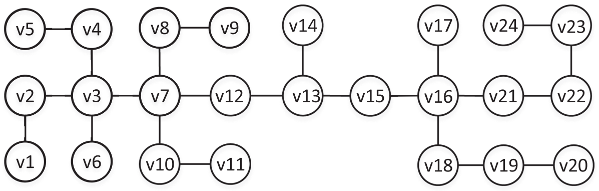


Fig. 1. A graph example used in the whole article.

2 BACKGROUND AND MOTIVATIONS

The new generation of computer usually has one or multiple GPUs in the recent years. To use these GPUs, the program owns device code to be concurrently executed over *Streaming Multiprocessors* (SM) of GPUs and host code to be executed on CPU. The device code, called the *GPU kernel*, should be a single-instruction multiple-threads program. The CPU launches the GPU kernel by dispatching related commands and its parameters to GPU. An SM usually owns several warps, which are assigned by its warp scheduler to be handled. A warp's threads execute the same instruction, which may incur low utilization ratio of SM when the load of these threads is skewed. Recently, many systems [4, 12, 15, 29, 35] were proposed to handle iterative graph processing over GPUs to exploit its high memory bandwidth and computing power.

Iterative graph algorithm usually consists of many rounds and each vertex updates its state within each round according to its neighbor' states. The graph algorithm's execution ends when all vertices' states converge. However, existing GPU-based solutions [4, 12, 15, 29, 35] suffer from low utilization ratio of the GPU because of slow state propagation. It is because most vertices on graph paths (especially the paths between some important vertices) are processed by different GPU threads concurrently. These threads may use their neighbors' stale states to update their own states within each round before the updating of their neighbors in the same round. As a result, based on stale states of their neighbors, most vertices have updated their states and require further updates within the next several rounds, incurring long time to repeatedly deal with the same partitions and low utilization ratio of the GPU.

For example, the path $v_1 - v_2 - v_3 - v_7 - v_{12} - v_{13} - v_{15} - v_{16} - v_{17}$ of Figure 1 may be put into several partitions, and each partition is handled on one SM. Vertex states in a partition, e.g., p_0, need to traverse several SMs to reach the vertices of the other partitions. Consequently, many partitions' vertices may have been updated based on the stale vertex states in the partition p_0 for several rounds before receiving the most recent vertex states from p_0, and need to be reprocessed. In addition, when a partition is assigned to a SM by existing solutions, its each path's vertices are also usually concurrently processed by its different threads. As a result, lots of cross-core propagations are generated, which lead to slow convergence speed because of frequent reprocessing of many vertices. For example, assume a partition p_0 has the vertices v_1, v_2, v_3, v_7. However, with existing solutions, the vertices in p_0 may be processed in the order of v_7, v_3, v_2, v_1 by a warp's different GPU threads. Thus, before receiving v_1's new state, v_2, $v_{3,}$ and v_7 have been updated based on the stale state of v_1 and need further reprocessing in latter rounds, where the new state of v_1 needs to traverse several GPU cores to reach them. The above slow state propagation not only incurs many unnecessary updates but also makes many vertices be inactive during several rounds.

3 OVERVIEW OF OUR APPROACH

To tackle these challenges, an efficient *structure-aware asynchronous execution approach* is proposed in this article to fully exploit the high parallelism of GPU for faster convergence speed of iterative graph processing.

3.1 Structure-Aware Asynchronous Execution Approach

We first define a concept, i.e., *Importance On Propagation (IOP)*, to evaluate a vertex's importance on state propagation, which is allowed to be defined by users. Usually, v_h plays a more important role in state propagations when its degree is higher. Thus, we approximately calculate *IOP* for each vertex v_h via its degree by default. v_h is identified as a *hub-vertex* when $IOP(v_h)$ is higher than a threshold T.

The value of T is set by AsynGraph to ensure that the ratio of hub-vertices to all vertices is λ ($0 \leq \lambda \leq 1$), because the value of T may be significantly different for different graphs and is difficult to be provided by users. The user controls the number of hub-vertices through providing the value of λ, instead of the value of T. The proper value of λ is usually small due to power-law property [8].

For runtime to quickly set T according to λ, it takes $\beta \cdot n$ number of vertex samples to reflect the original graph (which has n number of vertices), where $0 < \beta \leq 1$. Then, it gets a set S by arranging them based on their *IOP* in descending order, and can approximately get $T = IOP(v_s)$, where v_s is the $(\lambda \cdot \beta \cdot n)$th vertex of S.

For fast state propagations on GPU, it first extracts a set of graph paths between the hub-vertices (or called *hub-paths*) from the graph to construct a small *graph sketch*, i.e., G_s. Note that the remaining edges of the graph G are also represented as a set of disjoint paths (i.e., without intersected edges), or called *cold-paths*, i.e., $G_c = \cup c_l$, so as to enable efficient asynchronous intra-path state propagations along graph path, where $G = G_s \cup G_c$.

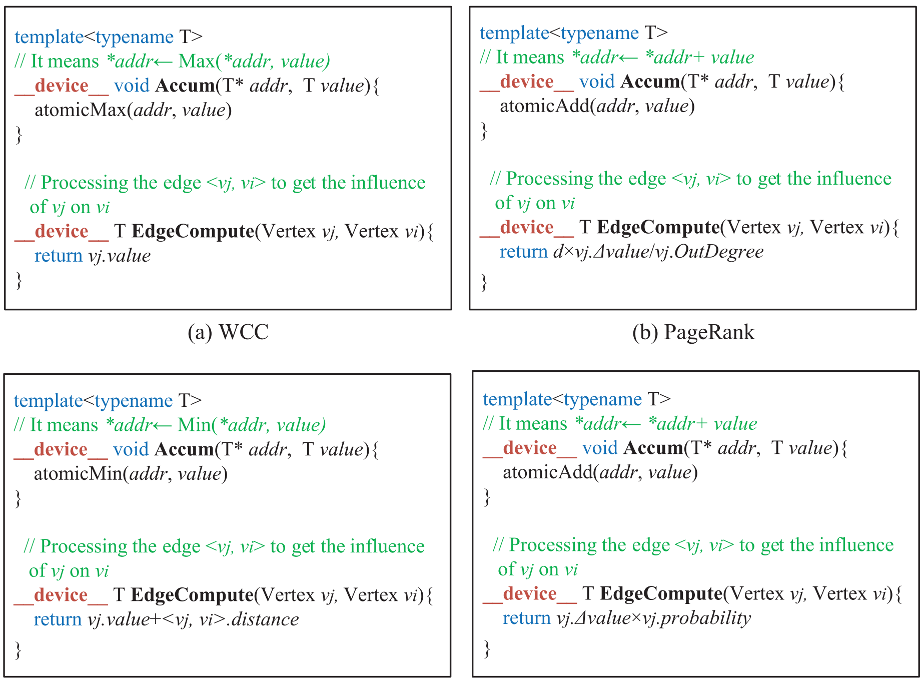
With existing solutions, hub-paths are handled for the same number of times as cold-paths in each round. To propagate vertex state across the graph sketch, it may need several rounds of graph processing to cross several hub-paths. Because most vertices' states need to be propagated through the graph sketch, most vertices need several rounds to converge and may be inactive during the execution, incurring low utilization of GPUs.

Thus, after that, at execution time, the paths in the graph sketch are handled multiple times within each round of graph processing, while the cold-paths are only handled once. By such means, the graph sketch can serve as a fast bridge for most state propagations and higher parallelism of GPU can be utilized to accelerate these propagations. Note that this processing way does not incur incorrect results as demonstrated in previous studies [8, 27, 28, 32].

However, asynchronous intra-path propagations along each path may be inefficiently conducted when this path's vertices are only asynchronously updated in one direction. For example, with this default round-robin way [4, 29, 35], the vertices on the path $v_7 - \cdots - v_{16}$ are only asynchronously processed along $v_7, \cdots, v_{15}, v_{16}$. Thus, v_{15}'s new state may need several rounds (e.g., three rounds) to reach the ones (e.g., v_7) located before v_{15}.

For efficient asynchronous intra-path propagation along each path, a *forward-backward (FB) path processing way* is also implemented. According to the storing order of each path's vertices, these vertices are first handled in a forward direction then in a backward direction until convergence. For example, for the path $v_7 - \cdots - v_{16}$, its vertices are asynchronously handled along the order of $v_7, \ldots, v_{15}, v_{16}$ at the forward stage, and it begins the backward stage to asynchronously handle the vertices from v_{16} towards v_7 when v_{16} is handled. Then, the new state of its each vertex, e.g., v_{15}, can affect all its other vertices, e.g., v_7 and v_{16}, only in one forward and backward stage along the graph path.

Besides, the FB path processing way enables better temporal locality due to shorter reuse distance. For example, it is unnecessary to load v_{15} and v_{16} for the processing of the edge $<v_{15}, v_{16}>$ at the backward stage, because they have been loaded for their processing at the forward stage. It enables a higher utilization ratio of loaded paths.



```
template<typename T>
// It means *addr← Max(*addr, value)
__device__ void Accum(T* addr, T value){
    atomicMax(addr, value)
}

  // Processing the edge <vj, vi> to get the influence
   of vj on vi
__device__ T EdgeCompute(Vertex vj, Vertex vi){
    return vj.value
}
```

(a) WCC

```
template<typename T>
// It means *addr← *addr+ value
__device__ void Accum(T* addr, T value){
    atomicAdd(addr, value)
}

  // Processing the edge <vj, vi> to get the influence
   of vj on vi
__device__ T EdgeCompute(Vertex vj, Vertex vi){
    return d×vj.Δvalue/vj.OutDegree
}
```

(b) PageRank

```
template<typename T>
// It means *addr← Min(*addr, value)
__device__ void Accum(T* addr, T value){
    atomicMin(addr, value)
}

  // Processing the edge <vj, vi> to get the influence
   of vj on vi
__device__ T EdgeCompute(Vertex vj, Vertex vi){
    return vj.value+<vj, vi>.distance
}
```

(c) SSSP

```
template<typename T>
// It means *addr← *addr+ value
__device__ void Accum(T* addr, T value){
    atomicAdd(addr, value)
}

  // Processing the edge <vj, vi> to get the influence
   of vj on vi
__device__ T EdgeCompute(Vertex vj, Vertex vi){
    return vj.Δvalue×vj.probability
}
```

(d) adsorption

Fig. 2. Examples to illustrate the implementation of iterative graph algorithms on AsynGraph.

4 OVERVIEW OF ASYNGRAPH

For efficient iterative graph processing on GPU, AsynGraph is designed to implement the above-described structure-aware asynchronous execution approach. Note that AsynGraph uses the popular GAS model [8, 32] and provides the same APIs. Figure 2 gives some examples to show how to initialize these APIs to implement graph algorithms.

4.1 Graph Preprocessing on CPU

4.1.1 Graph Partitioning Based on Graph Sketch. To identify the graph sketch G_s, a simple parallel method is used. It only reads the graph twice, which is much fewer than the rounds required by an iterative graph algorithm to converge. In detail, each CPU thread is assigned with a part of the graph, e.g., G^m. This CPU thread first calculates a set H^m, which contains all hub-vertices in G^m and also the boundary vertices of G^m which are also connected with the hub-vertices of the remaining graph. After that, as shown in Algorithm 1, the related CPU thread traverses G^m in a depth-first order by taking the hub-vertex of G^m as the root until all edges are visited. Each traverse's visited edges are recursively added into the same queue h_l one-by-one to generate each path (Lines 5 and 10), until a vertex belonging to H^m is found (Line 6) or no other vertices can be further explored. A hub-path is found when a vertex belonging to H^m is found (Line 6), and then is used to construct the graph sketch, i.e., G_s (Line 7). Otherwise, the edges of the path are recursively dequeued from h_l to be stored in the queue c_l as a cold-path (Lines 11-18). Then, it begins a new traverse to find the next path.

The identified hub-paths, e.g., $v_3 - v_7$ and $v_7 - \cdots - v_{16}$, constitute the graph sketch and are put together into several *hub chunks*, e.g., p_s, while the remaining paths (i.e., the cold-paths) of the graph are put into many *cold chunks*, e.g., p_1, p_2, and p_3, as shown in Figure 3, where each chunk

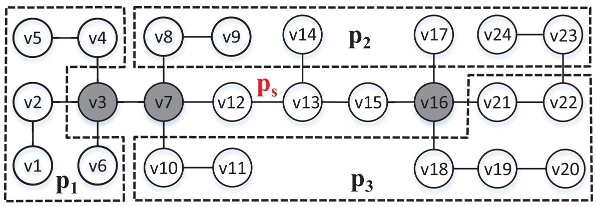


Fig. 3. An example to illustrate the graph sketch, where v_3, v_7, and v_{16} are assumed to be three important vertices in the graph example.

ALGORITHM 1: Efficient Graph Partitioning on CPU

```
1:  procedure GraphPre($G^m$, $G_s$, $G_c$, $v_h$, $h_l$, $c_l$)
2:      $S_N$←GetN($G^m$, $v_h$) /*Get $v_h$'s local neighbors in $G^m$.*/
3:      for each vertex $v_k \in S_N \wedge$ the edge $\langle v_h, v_k \rangle$ is unvisited do
4:          The edge $\langle v_h, v_k \rangle$ is set as visited.
5:          $h_l \leftarrow h_l \cup \langle v_h, v_k \rangle$ /*Insert $\langle v_h, v_k \rangle$ into the queue $h_l$.*/
6:          if $v_k \in H^m$ then /*It means that the path stored in $h_l$ is a hub-path.*/
7:              $G_s \leftarrow G_s \cup h_l$ /*Insert the hub-path into the set $G_s$.*/
8:              $h_l \leftarrow \emptyset$ /*Begin to use it to store a new hub-path.*/
9:          else
10:             GraphPre($G^m$, $G_s$, $G_c$, $v_k$, $h_l$, $c_l$)
11:             if $h_l \neq \emptyset$ then /*It means that it cannot reach any hub-vertex.*/
12:                 $h_l \leftarrow h_l - \langle v_h, v_k \rangle$ /*Dequeue the edge $\langle v_h, v_k \rangle$ from the queue $h_l$.*/
13:                 $c_l \leftarrow c_l \cup \langle v_h, v_k \rangle$ /*Insert $\langle v_h, v_k \rangle$ into a queue $c_l$ storing the cold-path.*/
14:                 if $S_N$ has unvisited vertices or $h_l = \emptyset$ then
15:                     $G_c \leftarrow G_c \cup c_l$ /*The cold-path stored in $c_l$ is inserted into the set $G_c$.*/
16:                     $c_l \leftarrow \emptyset$ /*Begin to use it to store a new cold path.*/
17:                 end if
18:             end if
19:         end if
20:     end for
21: end procedure
```

has almost the same number of edges and is to be handled by a warp. To ensure that the data to be handled by the warps on an SM can be retained in its shared memory as much as possible for a better locality, the size of each chunk is $\frac{1}{N_t}$ of that of the shared memory of this SM, where N_t is the number of warps on an SM. Note that it also tries to put the short paths with the replicas of the same high-degree vertex together to be handled by the same GPU thread for lower synchronization cost. A long path may be divided into several contiguous chunks and sequentially handled by the threads, because each chunk can only store a certain number of edges.

4.1.2 Graph Storage Scheme on GPU. Four arrays are used to store the set of disjoint paths on GPU. Figure 4(a) shows how to store the chunk p_2 of Figure 3 on GPU. It uses E_{Idx} and S'_{val} to store the indexes and the states of the vertices, respectively. For E_{Idx} (as well as S'_{val}), as described in Figure 4(b), for efficient state propagation along each path, the related information of the vertices on this path is tried to be maintained in the GPU memory following their order on this path. Each edge is represented by two items of E_{Idx} for less storage cost. For the edge $<v_j, v_i>$ handled by the tth thread (i.e., T_t), $j = E_{Idx}[d * r + t]$ and $i = E_{Idx}[d * (r + 1) + t]$. There, d is the number of GPU threads in a warp and $d = 32$ by default because a warp consists of 32 GPU threads. r is the row

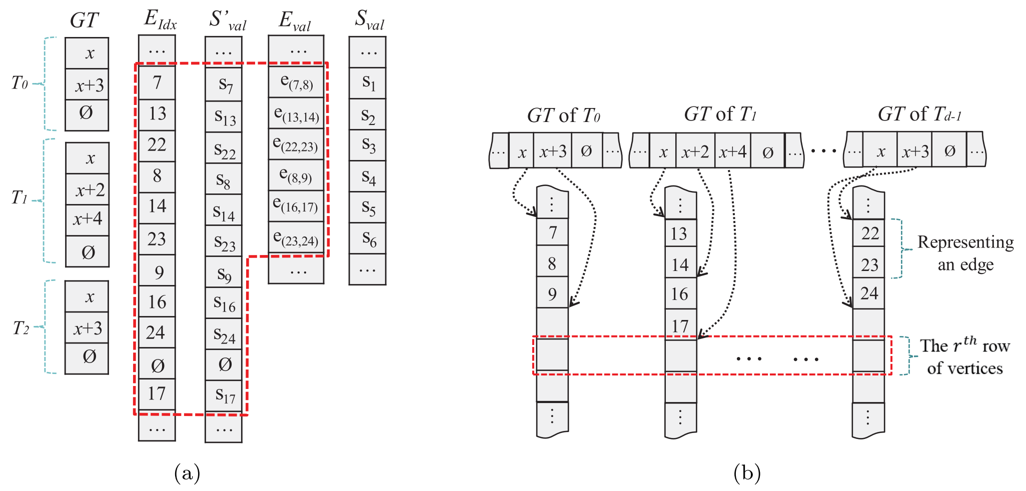


Fig. 4. Illustration of graph storage on GPU: (a) the storage of the chunk p_2 on GPU, where the number of GPU threads in a warp is assumed to be three, and p_2 contains four paths, i.e., $v_7 - v_8 - v_9$, $v_{13} - v_{14}$, $v_{16} - v_{17}$, and $v_{22} - v_{23} - v_{24}$ to be handled by three threads (i.e., T_0, T_1, and T_2); (b) the way to store the arrays E_{Idx} and S'_{val}.

number and t is the thread number in a warp. Note that the values of the edges on each path are stored in E_{val} accordingly. In this way, when loading a contiguous range of E_{Idx}, S'_{val}, and E_{val} into an SM for a warp, each GPU thread of this warp has the related data of its own path to handle. It enables coalesced accesses for parallel processing of graph paths over these GPU threads.

Meanwhile, for the above three arrays, the related data for hub-paths is stored together at the head of the related array for efficient sequential access of them over GPU because no cold-path (which may become inactive soon) is stored among them, while that of the remaining paths is stored following them in a contiguous way. For quick access of each vertex's state, a separate array, i.e., S_{val}, is also maintained. The information of the paths to be processed by each GPU thread is stored in its local table, i.e., *GT*, which has a field to store the row number (i.e., r) of its each path's first vertex. A path's range is indicated by two successive items of *GT*.

4.2 Efficient Processing of Graph Sketch on GPU

After that, for low traffic of CPU-GPU memory copy, the chunks are dispatched to the GPU in batches. It also creates multiple streams for this transfer to overlap kernel execution and memory copy. Then, the paths contained in these chunks begin to be processed on the GPU as the following discussed for faster convergence speed.

As described above, to quickly propagate vertex state across the graph sketch, in each round, the hub chunks are expected to be handled multiple times when the cold chunks are handled only once. However, it may suffer from load imbalance because processing behavior of hub-paths and cold-paths is different. Note that it does not suffer from heavy warp divergence, because the hub-paths are put together at preprocessing stage to be processed by the same warps' GPU threads.

To efficiently get our goal, as depicted in Figure 5, long-running persistent GPU threads [3] are generated over the SMs to repeatedly load chunks from global circular hot/cold queue and asynchronously handle the related active chunks until the queue is empty, i.e., no more active chunks. The hot queue contains all active hub chunks, which are handled by the GPU threads of

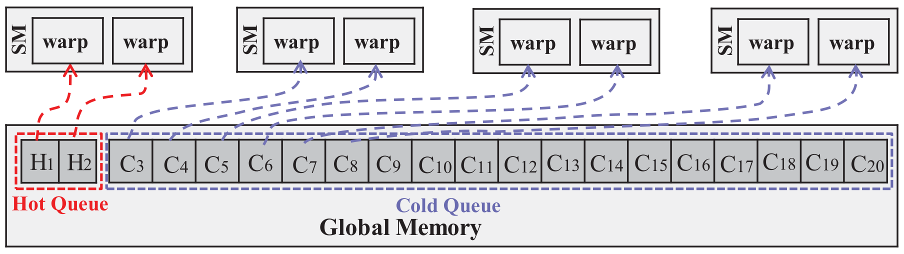


Fig. 5. An example to show concurrent processing of a graph, where N_s is assumed to be 2, the graph has two hub chunks (i.e., H_1 and H_2), and the other chunks are cold ones.

N_s number of warps, while the active cold chunks in the cold queue are processed by the remaining warps. It enables the following optimizations.

For faster propagation on graph sketch, for each loaded hub chunk, GPU threads of the related warp handle it for N_p number of times or until all its vertices are inactive, before it is swapped out of the related SM. The active cold chunks are loaded into SMs in turn, and each cold chunk is handled once each time. When a cold chunk has been handled, another one of the remaining cold chunks is loaded into a free SM. N_s is set as $N_s = \frac{\lambda \cdot \tau \cdot N_p}{1-\lambda+\lambda \cdot N_p}$ to ensure that $\frac{\lambda \cdot N}{N_s} = \frac{(1-\lambda) \cdot N}{(\tau - N_s) \cdot N_p}$, where N is the number of chunks and τ is the number of warps on all SMs. In this way, hub chunks can be handled for N_p number of times in each round, when all cold chunks are handled for exactly once. For a balanced load between SMs, the chunks assigned to warps of overloaded SMs can be stolen by warps of the free SMs. When there is no chunk to be assigned to warps of the idle SMs, it also allows chunks to begin the next round of processing, although the current round is not ended.

4.3 Forward-Backward Intra-Path Processing on SM

When vertex state is propagated across GPU cores, it incurs high data access cost and synchronization cost. For efficient implementation of our FB way, for each chunk loaded by a warp, as shown in Figure 6, in AsynGraph, vertices of the same path are asynchronously handled by the same GPU thread to ensure the intra-path state propagation only occurs on the same core. Different paths in

ALGORITHM 2: Forward-backward Intra-path Processing

```
1:  procedure PathProcessing(Thread $T_t$, Path $L_l$)
2:      $R_l \leftarrow$ Row number of $L_l$'s first vertex
3:      $R_n \leftarrow$ Row number of the next path's first vertex
4:      $r \leftarrow R_l$
5:      $flag \leftarrow 1$ /*Indicates the processing direction.*/
6:      while IsActive($L_l$) do /*The path $L_l$ is active.*/
7:          while $R_l \leq r < R_n$ do
8:              $j \leftarrow E_{Idx}[d * r + t]$
9:              $r \leftarrow r + flag$
10:             $i \leftarrow E_{Idx}[d * r + t]$
11:             $f_{(v_j, v_i)}(s_j) \leftarrow$ EdgeCompute($v_j$, $v_i$)
12:             Accum($s_i$, $f_{(v_j, v_i)}(s_j)$)
13:         end while
14:         $flag \leftarrow flag \times (-1)$
15:     end while
16: end procedure
```

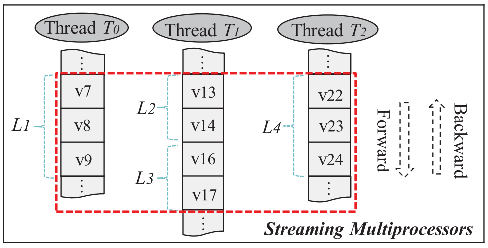


Fig. 6. Illustration of concurrent processing of four paths in the chunk p_2 by three threads on an SM.

a chunk are concurrently handled by its GPU threads for parallel processing. For a balanced load between these threads, each thread is tried to be assigned with almost the same number of edges.

Algorithm 2 describes how to handle a path L_l on a GPU thread T_t in a FB way, where $d = 32$. The vertices of L_l are handled asynchronously based on their order on L_l (Lines 6–15). It reverses the vertices' processing order when the last vertex of L_l has been handled (Line 14). For each vertex v_j, AsynGraph uses the user provided $EdgeCompute()$ to handle its state to get its influence on its local neighbor v_i (Line 11), which is located on the same path. The calculated influence, i.e., $f_{(v_j,v_i)}(s_j)$, is then used to update v_i's state via $Accum()$ (Line 12). When a chunk's processing is finished, for the vertex with multiple replicas, the new state of its each mirror replica (S'_{val} of Figure 4(a)) is sent to its master replica (S_{val} of Figure 4(a)) for state synchronization, getting the final state (stored in its master replica) of this vertex at this round.

4.4 Case Study: WCC Algorithm

This section takes WCC algorithm to explain how to realize graph algorithms on AsynGraph. To implement WCC on AsynGraph, $Accum()$ and $EdgeCompute()$ first need to be instantiated by the programmer. As shown in Figure 2(a), $EdgeCompute()$ directly outputs the component ID of a vertex v_j for the update of the component ID of its neighbor, e.g., v_i. Then, $Accum()$ is used for a vertex (e.g., v_i) to accumulate its current component ID with all component IDs received from its neighbors, e.g., v_j, through calculating the maximum value of all these component IDs, and the accumulated result is then used to update the current component ID of this vertex v_i. Finally, the programmer programs $IsActive()$ to give the termination criterion of WCC, i.e., the component ID of any vertex has no change between two successive rounds.

After that, the WCC algorithm can be executed on our AsynGraph with the parameters (i.e., N_p, λ, and β) provided by users. As shown in Section 5.3, the programmer is expected to provide suitable parameters for better performance. Note that it is similar to implement other popular iterative graph algorithms [32] on AsynGraph.

5 EXPERIMENTAL EVALUATION

This section evaluates AsynGraph against existing systems on GPU. In our hardware platform, there are four 12-core Intel Xeon E5-2670 v3 CPUs on the host side with 128-GB main memory, where each CPU has two 9.6 GT/s QPI link and PCI Express 3.0 lanes operates at 16x speed. The device side is a NVIDIA TESLA V100 GPU with 5120 cores and 32-GB on-board memory. The program is compiled by Boost 1.70.0, GCC 7.3, and CUDA 10.1 with the -O3 flag.

Four popular graph algorithms are used as benchmarks: (1) pagerank [32]; (2) adsorption [32]; (3) SSSP [32]; (4) WCC [32]. Note that the source vertex is randomly selected for SSSP as in many previous studies [4, 12, 29, 35]. Six graphs [1], i.e., ego-Gplus (gplus), com-Amazon (amazon), soc-Pokec (pokec), com-Orkut (orkut), com-LiveJournal (livejournal), and com-Friendster (friendster), are used. Both gplus and pokec are directed graphs, and the others are undirected graphs. Their

Table 1. Characteristic Statistics of Datasets ($\bar{D}$ is the Average Vertex Degree and d is the Graph Diameter)

Datasets	$\#Vertices$	$\#Edges$	$\bar{D}$	d
gplus	107,614	13,673,453	127	6
amazon	334,863	925,872	6	44
pokec	1,632,803	30,622,564	19	11
orkut	3,072,441	117,185,083	76	9
livejournal	3,997,962	34,681,189	17	17
friendster	65,608,366	950,652,916	29	32

characteristics are given in Table 1, where the graph diameter is the longest length of the shortest paths between any two vertices.

The previous studies [29] have shown that GPU-based frameworks outperform CPU-based ones. Thus, we focus on the comparison among GPU-based graph processing systems. AsynGraph is compared with four existing systems, i.e., Gunrock v1.1 [29], Groute v1.0 [4], Tigr [12], and DiGraph [35]. Besides, AsynGraph is also compared with SymGraph-G, TP-G, FBSGraph-G, and HATS-G, which are the versions of the approaches proposed in SymGraph [37], TP-X [17], FBSGraph [34], and HATS [25], and are extended by us for GPU-based graph processing, respectively. We also extend Enterprise [19] to support PageRank and SSSP based on its supported BFS. Their performance is tried to be evaluated with their best-performance settings, respectively. For the best performance of AsynGraph, its parameters are also tried to be tuned. When tuning the parameters for AsynGraph, we find that its parameters are only sensitive to the graphs instead of graph algorithms. For the six graphs, i.e., gplus, amazon, pokec, orkut, livejournal, and friendster, the suitable values of $< \lambda, \beta >$ are <0.5%, 0.01>, <2.5%, 0.01>, <0.5%, 0.001>, <0.5%, 0.001>, <2.5%, 0.001>, and <0.5%, 0.0001>, respectively, while N_p is 2, 4, 3, 2, 4, and 4, respectively.

To understand AsynGraph, two other versions of AsynGraph, i.e., *AsynGraph-w* and *AsynGraph-t*, are also evaluated. The difference between AsynGraph-w and AsynGraph is that AsynGraph-w asynchronously handles each path in a default round-robin way instead of our FB way, yet using our sketch-based approach. The difference between AsynGraph-t and AsynGraph is that AsynGraph-t only asynchronously handles each path using our FB way, however, does not use our sketch-based approach. The results are the average value of ten runs.

5.1 Convergence Speed

Figure 7 evaluates the number of vertex state updates for different graph algorithms to converge. We can find that AsynGraph needs fewer updates than the other systems. It is because the new states of most vertices are able to be propagated by AsynGraph in a faster way by fully utilizing high parallelism of GPU. Therefore, fewer vertices use the stale states of their neighbors to update their own states in each round. Besides, as shown in this figure, AsynGraph usually achieves more reduction on updates for high-diameter graphs. For example, for WCC, AsynGraph can reduce that of Groute by 70.9% for amazon, while 63.2% for gplus, because the diameter of amazon is longer than that of gplus.

5.2 Performance Comparison

The speedups of various systems against Gunrock are given in Figure 8. As we can observed, Gunrock, Groute, and Tigr need more graph processing time than AsynGraph, because they need to deal with more updates and also require a higher cost to transfer unnecessary data into SMs. DiGraph suffers from a low GPU utilization ratio due to the inherent synchronization cost of its

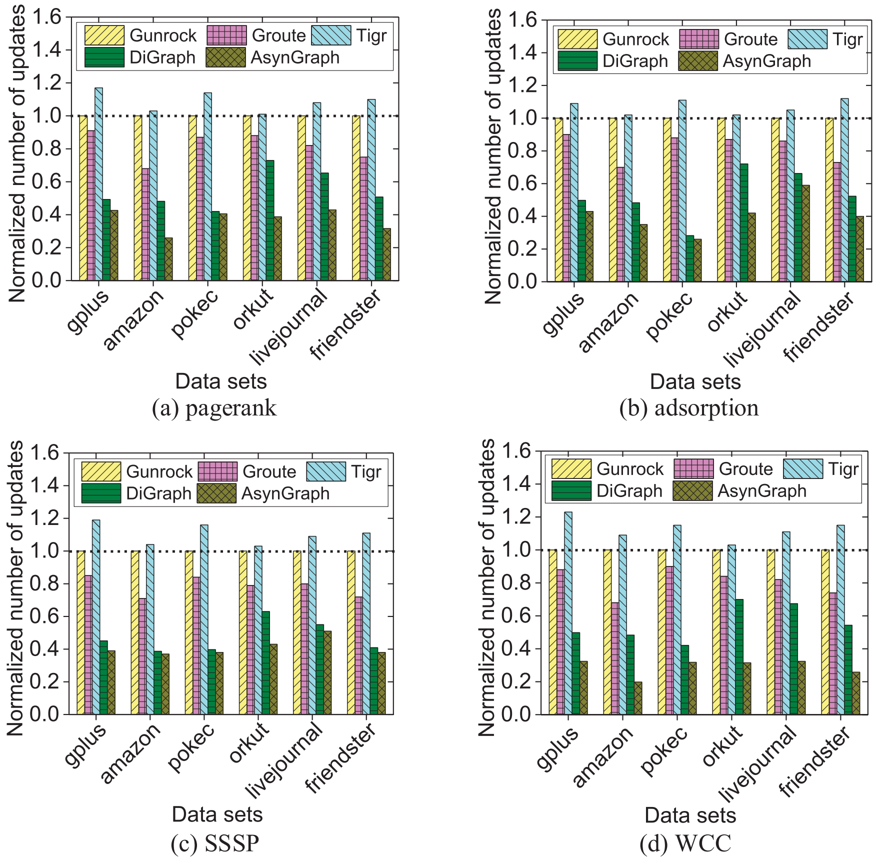
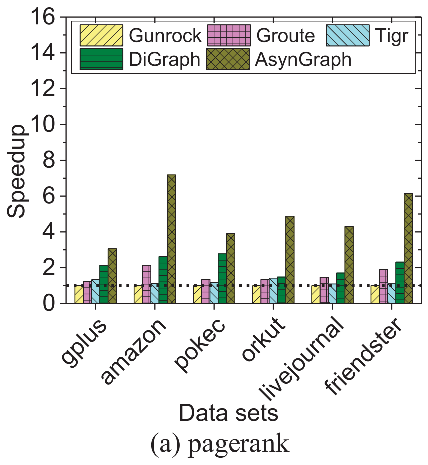


Fig. 7. Number of updates of different systems normalized to that of Gunrock.

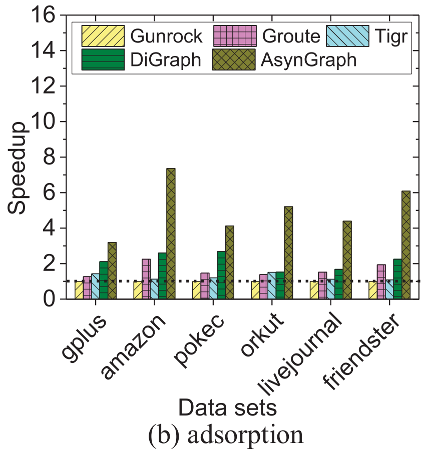
execution model and inefficient state propagation (especially for undirected graphs due to slow backward propagation), although it often performs better than Gunrock, Groute, and Tigr. AsynGraph outperforms DiGraph thanks to more efficient utilization of high parallelism of GPU. For different cases, AsynGraph achieves performance improvements of 3.06–11.52, 2.47–5.40, 2.23–9.65, and 1.41–4.05 times in comparison with Gunrock, Groute, Tigr, and DiGraph, respectively. Note that AsynGraph can finally get the same results as the other systems as shown in Figure 9, yet getting faster convergence speed.

Existing systems, including AsynGraph, usually tradeoff additional preprocessing time for much smaller total execution time. Table 2 gives the breakdown of the total execution time of pagerank. The preprocessing time includes the time to preprocess graph and transfer data from CPU to GPU before execution. At the graph preprocessing stage, Tigr needs to divide the graph and generate a virtual graph based on the original graph. DiGraph also needs to divide the graph and also constructs its required DAG sketch. AsynGraph needs to divide the graph, identify hub-vertices and extract the graph sketch. Note that both Gunrock and Groute only need to transfer the graph to GPU before graph processing. For Gunrock, Groute, Tigr, DiGraph, and AsynGraph, the graph processing time is the time to handle the graph until convergence.

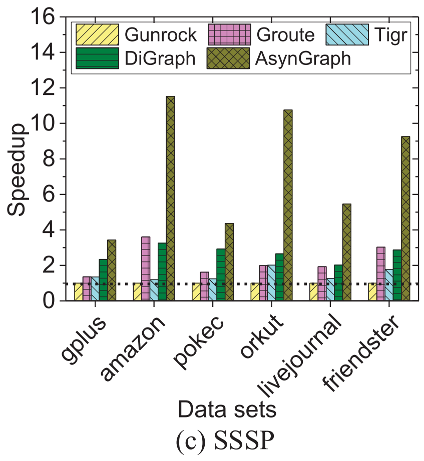
Although AsynGraph needs additional preprocessing cost to find the graph sketch and to divide the graph, the total execution time is smaller in iterative graph algorithms on AsynGraph than that



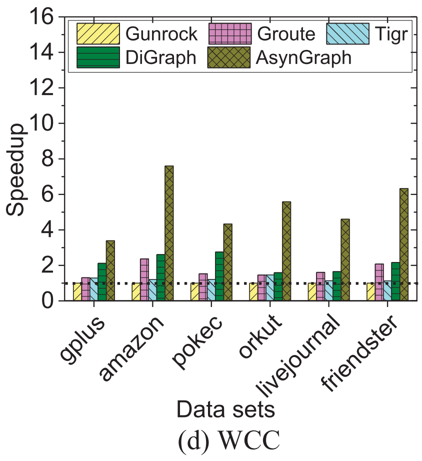
(a) pagerank



(b) adsorption



(c) SSSP



(d) WCC

Fig. 8. Speedups of different systems against Gunrock for various graph algorithms.

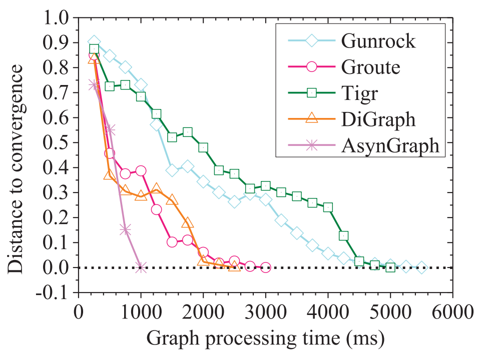


Fig. 9. Average difference between the intermediate results of different systems and the final results of Gunrock for PageRank over friendster.

Table 2. Preprocessing Time and Graph Processing Time of Pagerank Over Different GPU-Based Graph Processing Systems (ms)

Datasets	Gunrock	Groute	Tigr	DiGraph	AsynGraph
gplus	1.8 (51.0)	1.9 (41.1)	5.2 (38.2)	8.6 (23.8)	5.6 (16.4)
amazon	0.2 (9.6)	0.3 (4.5)	0.6 (8.6)	1.8 (3.6)	0.8 (1.2)
pokec	3.8 (141.0)	4.0 (104.3)	11.2 (120.8)	20.8 (50.7)	14.2 (36.0)
orkut	14.7 (376.8)	15.2 (280.1)	48.4 (266.3)	128.7 (254.3)	60.5 (77.3)
livejournal	4.4 (186.0)	4.7 (126.4)	14.4 (170.5)	38.0 (109.2)	19.8 (43.1)
friendster	122.6 (5,327.4)	129.4 (2,818.9)	326.1 (4,821.0)	618.4 (2,301.5)	363.0 (865.8)

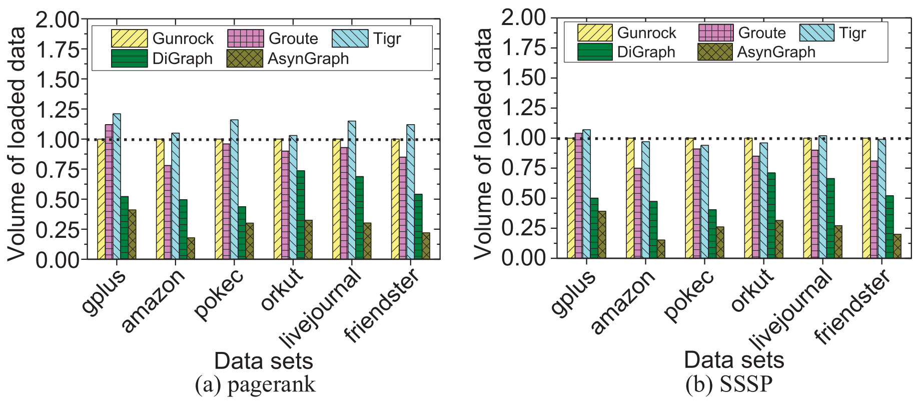


Fig. 10. Normalized volume of graph data loaded into SM by pagerank and SSSP.

of other systems for fewer updates, smaller data access cost, and higher GPU utilization ratio. We can also observe that AsynGraph performs better for the graphs with longer diameter, yet still gets better performance than other systems for low-diameter real-world graphs, such as gplus with the diameter of only 6.

Figure 10 shows the volume of graph data loaded from the global memory into SMs by pagerank and SSSP on different systems normalized to that of Gunrock, respectively. As observed in the figure, AsynGraph loads smaller graph data than other systems for all cases. We observe two reasons for this phenomenon. First, fewer updates are handled by AsynGraph than them for faster propagation of vertex's new state. Second, AsynGraph has better data locality by efficiently storing the graph and also processing the graph along the paths in our proposed FB way.

Figure 11 also studies the impact of individual techniques on the performance improvement of AsynGraph. As observed in the figure, AsynGraph-t gets better performance than FBSGraph-G and HATS-G only using forward-backward intra-path processing way. AsynGraph-w performs better than SymGraph-G and TP-G only using our sketch-based approach. Both AsynGraph-t and AsynGraph-w perform better than Enterprise. We can also observe that AsynGraph outperforms AsynGraph-w under most conditions, i.e., for undirected graphs. It is because of higher utilization of the loaded paths and fewer updates as shown in Figure 12. Note that, for directed graphs, i.e., gplus and pokec, AsynGraph gets no benefits from the FB way because the vertex state only needs to be propagated in the forward direction. Under such circumstances, AsynGraph even needs more graph processing time than AsynGraph-w, because AsynGraph needs little cost to determine whether it needs to handle the path in the backward direction. For example, for

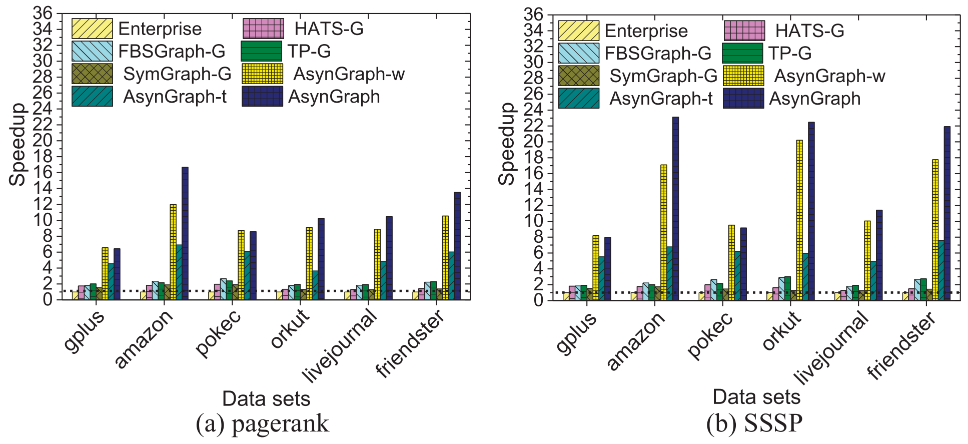


Fig. 11. Graph processing time of different systems normalized to that of Enterprise.

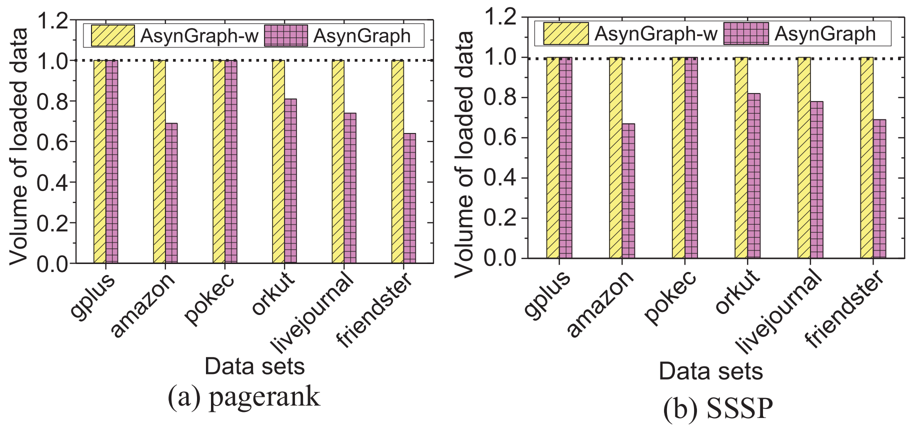


Fig. 12. Volume of graph data loaded into SM by AsynGraph normalized to that of AsynGraph-w.

SSSP over gplus, the graph processing time of AsynGraph is more than that of AsynGraph-w by 2.9%.

Figure 13 shows their GPU utilization ratios on pagerank and SSSP, respectively. As observed, Groute, Tigr, and DiGraph have higher ratio than Gunrock, because Gunrock needs high synchronization cost, while others use the asynchronous way. AsynGraph has the highest ratio. For example, for pagerank on different graphs, the ratio of AsynGraph is 1.74–1.98 and 1.43–2.30 times that of Groute and Tigr, respectively, due to faster propagation of vertex's new state and smaller data access cost via handling the graph along paths in a FB way for better temporal locality. Although DiGraph uses an asynchronous model, its GPU utilization ratio is also not high as AsynGraph, because it handles the graph according to a topological order for fewer updates.

Figure 14 also shows the performance of pagerank on five randomly constructed power-law graphs (see Table 3) with fixed ten-million vertices yet with various values of α, which is generated in the same way as the previous work [8]. A synthetic graph with lower α means it has higher vertex degree skewness [8]. We can find that AsynGraph performs much better when the value of α is lower, because more vertex state propagations can be accelerated through our sketch-based approach.

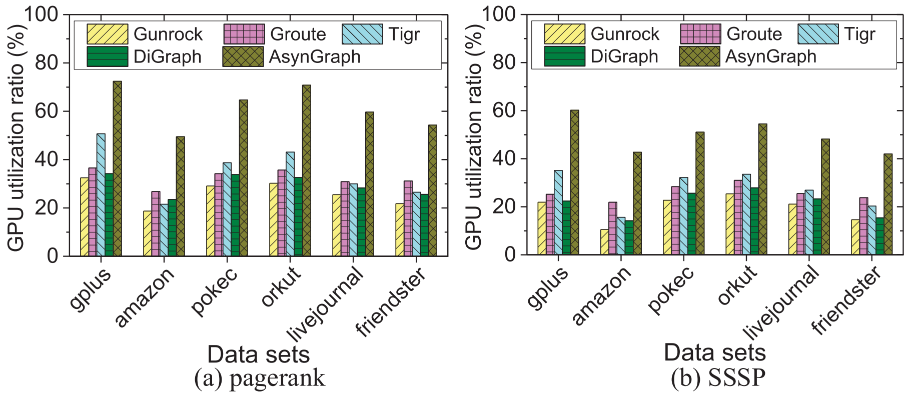


Fig. 13. Average GPU utilization ratio of pagerank and SSSP over different systems.

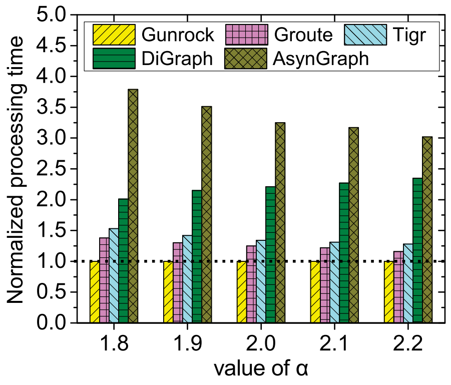


Fig. 14. Sensitivity to various synthetic graphs.

Table 3. Description of the Five Synthetic Graphs

α	1.8	1.9	2.0	2.1	2.2
#Edges	667 M	246 M	104 M	56 M	37 M

5.3 Impacts of Parameters

Table 4 and Table 5 give the total execution time of pagerank and SSSP over AsynGraph with various values of λ ($0.02\% \leq \lambda \leq 62.5\%$) and β ($0.00001 \leq \beta \leq 0.1$), respectively. There, λ is the ratio of hub-vertices to all vertices, and β is the sample rate. λ and β are used by users to control the ratio of the number of hub-vertices to the total number of all vertices. The suitable value of λ is usually between 0.5% and 2.5%, while the suitable value of β (usually no more than 0.01) is smaller when the graph size is larger. From Table 4 and Table 5, we can observe that too small ratio of hub-vertices may miss some hub-paths which are helpful to accelerate state propagations. A large ratio of hub-vertices also may undermine its efficiency due to the waste of the GPU resource to handle the unimportant paths multiple times within each round. Besides, the suitable ratio of hub-vertices is usually small.

Figure 15 depicts the impact of N_p (the number of processing times of the sketch in each round) on the total execution time of pagerank and SSSP. We have two observations from this figure,

Table 4. Impacts of λ and β on the Execution Time of Pagerank Over AsynGraph (ms)

Graphs	Parameters	$\lambda = 0.02\%$	$\lambda = 0.1\%$	$\lambda = 0.5\%$	$\lambda = 2.5\%$	$\lambda = 12.5\%$	$\lambda = 62.5\%$
gplus	β=0.00001	24.2	23.8	23.6	24.1	24.6	25.4
	β=0.0001	23.9	23.3	23.2	23.6	24.3	24.8
	β=0.001	23.4	22.7	22.4	23.1	23.7	24.4
	β=0.01	22.8	22.3	**22.0**	22.7	23.5	24.1
	β=0.1	23.5	22.9	22.5	22.9	23.9	25.6
amazon	β=0.00001	2.45	2.36	2.32	2.23	2.29	2.42
	β=0.0001	2.36	2.28	2.24	2.12	2.24	2.35
	β=0.001	2.27	2.22	2.15	2.09	2.18	2.28
	β=0.01	2.16	2.11	2.07	**2.00**	2.17	2.24
	β=0.1	2.25	2.19	2.13	2.05	2.22	2.31
pokec	β=0.00001	56.9	54.1	53.6	54.3	55.6	57.2
	β=0.0001	55.0	52.5	51.7	52.5	53.8	55.9
	β=0.001	53.8	51.9	**50.2**	51.6	52.7	54.5
	β=0.01	54.3	52.4	51.8	52.9	53.5	55.4
	β=0.1	55.9	54.2	53.0	53.8	54.9	56.7
orkut	β=0.00001	154.9	152.5	144.2	151.1	155.5	160.3
	β=0.0001	151.5	149.0	143.9	145.4	150.4	156.0
	β=0.001	149.2	147.4	**137.8**	142.6	147.5	153.1
	β=0.01	152.6	150.2	139.6	144.2	149.6	155.8
	β=0.1	155.3	151.7	146.5	150.7	153.2	159.5
livejournal	β=0.00001	71.6	70.4	69.1	67.7	69.5	72.0
	β=0.0001	69.5	67.3	65.9	64.2	66.9	69.5
	β=0.001	67.9	66.5	64.3	**62.9**	65.7	67.2
	β=0.01	69.4	68.7	66.1	64.5	68.8	68.6
	β=0.1	71.8	70.1	68.4	65.3	70.4	71.2
friendster	β=0.00001	1,331.5	1,256.3	1,251.9	1,264.6	1,309.5	1,390.2
	β=0.0001	1,279.9	1,246.5	**1,228.8**	1,236.6	1,274.2	1,325.1
	β=0.001	1,285.9	1,247.7	1,235.4	1,241.3	1,282.5	1,332.4
	β=0.01	1,306.1	1,250.9	1,242.2	1,255.8	1,301.8	1,358.5
	β=0.1	1,314.4	1,258.4	1,247.5	1,262.6	1,315.7	1,413.9

which help us to efficiently select the suitable value of N_p. First, when increasing the value of N_p, the performance of AsynGraph always first increases and it decreases when N_p is larger than a value. In other words, the performance degrades when N_p is too small or too large. The execution in this case wastes GPU resource to traverse many inactive vertices when N_p is too large, and incurs useless updates due to slow state propagation when N_p is too small. Second, the suitable value of N_p is usually small and is even much smaller when the graph diameter is much shorter. Nevertheless, in the experiments, AsynGraph still performs better than existing GPU-based solutions without making effort to choose the above parameters.

Finally, Figure 16 also evaluates other definition ways of IOP (e.g., betweenness [10] and closeness [5]) of AsynGraph. We can observe that more execution time is required when the betweenness centrality and the closeness centrality is used to evaluate the IOP of a vertex due to much longer preprocessing time. The user can define IOP in the other ways as required. It is obvious a tradeoff between shorter graph processing time and longer preprocessing time. The user can use

Table 5. Impacts of λ and β on the Total Execution Time of SSSP Over AsynGraph (ms)

Graphs	Parameters	λ = 0.02%	λ = 0.1%	λ = 0.5%	λ = 2.5%	λ = 12.5%	λ = 62.5%
gplus	β=0.00001	16.4	16.1	15.8	16.2	16.3	16.5
	β=0.0001	16.0	15.5	15.4	15.7	15.8	15.9
	β=0.001	15.6	15.2	14.9	15.3	15.4	15.6
	β=0.01	15.2	15.0	**14.8**	15.1	15.3	15.5
	β=0.1	15.5	15.3	15.0	15.4	15.8	16.9
amazon	β=0.00001	1.40	1.33	1.31	1.27	1.38	1.46
	β=0.0001	1.35	1.29	1.25	1.22	1.35	1.39
	β=0.001	1.31	1.25	1.21	1.19	1.30	1.37
	β=0.01	1.29	1.23	1.19	**1.18**	1.26	1.34
	β=0.1	1.32	1.25	1.21	1.20	1.29	1.45
pokec	β=0.00001	30.5	29.2	28.4	29.1	30.2	31.6
	β=0.0001	29.1	28.5	27.9	28.3	28.7	30.5
	β=0.001	28.5	27.7	**27.4**	27.9	28.2	30.1
	β=0.01	28.7	28.0	27.7	28.1	28.6	30.3
	β=0.1	29.4	28.5	28.2	29.5	30.3	31.5
orkut	β=0.00001	77.1	75.2	74.6	76.5	81.9	87.6
	β=0.0001	75.0	74.5	73.2	74.8	77.2	83.9
	β=0.001	74.3	73.0	**72.5**	73.8	75.7	80.5
	β=0.01	75.9	73.4	72.9	74.1	77.9	81.6
	β=0.1	80.5	79.7	74.4	75.2	78.1	82.2
livejournal	β=0.00001	35.9	34.8	34.3	34.1	36.0	37.2
	β=0.0001	35.0	34.1	33.8	33.5	35.6	36.7
	β=0.001	34.6	33.7	33.3	**33.2**	35.4	36.1
	β=0.01	34.9	34.5	33.7	33.6	35.9	37.0
	β=0.1	35.7	35.2	34.6	34.0	36.5	37.4
friendster	β=0.00001	872.0	839.7	820.5	839.2	855.9	880.3
	β=0.0001	837.8	820.4	**813.9**	830.5	841.5	873.5
	β=0.001	841.2	835.2	816.7	835.9	849.3	870.4
	β=0.01	846.7	839.9	821.5	841.6	856.8	889.7
	β=0.1	855.4	842.1	824.4	847.1	870.0	904.1

a more intelligent way to evaluate IOP if the preprocessed results can be repeatedly used multiple times. For example, the graph is static in many applications, such as protein interaction network. In the future, we will research other definition ways of IOP.

6 RELATED WORK

6.1 CPU-Based Graph Processing

Lots of CPU-based systems are designed for graph processing. Pregel [24] executes graph algorithms in a supper-step way to naturally express them. For lower synchronization cost, GraphLab [21] is latterly designed to asynchronously execute graph algorithm. For load balancing, PowerGraph [8] deals with vertex-programs over edges. For better performance of disk-based graph processing via exploiting the value of loaded data, CLIP [2] uses beyond-neighborhood accesses and loaded data reentry. FBSGraph [34] also proposes to handle graph along paths in a forward-backward way as our proposed approach. However, because the GPU is based on SIMT,

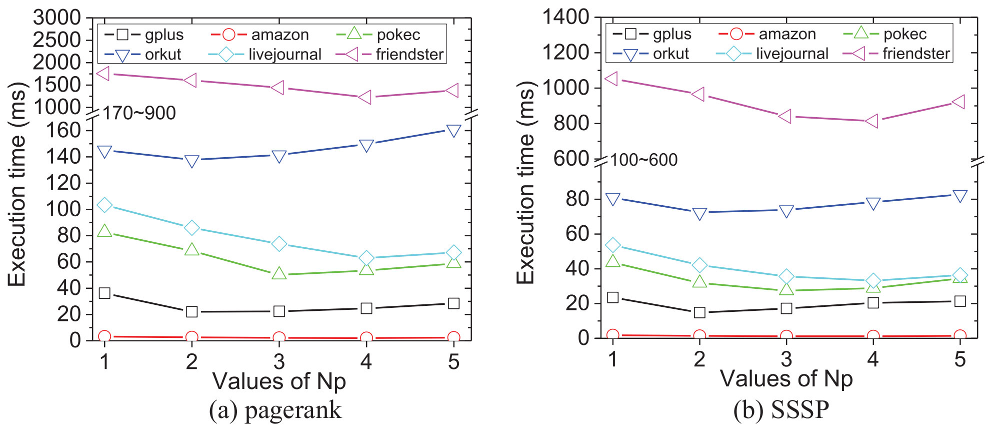


Fig. 15. Impacts of N_p on pagerank and SSSP over AsynGraph.

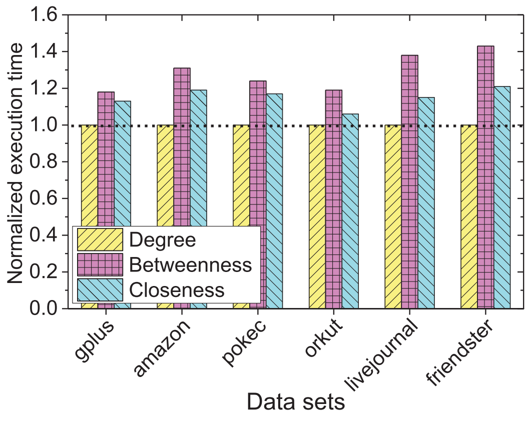


Fig. 16. Performance of pagerank over AsynGraph with various definition ways of IOP.

using FBSGraph, many cross-core state propagations will be generated and incur high runtime cost and also need lots of useless rounds to converge. Similar to FBSGraph, HATS [25] prefetches vertices along paths for better temporal locality with hardware support, however, also suffers from the above problems as FBSGraph.

Hub^2-Labeling method [14] is proposed to reduce the search space for the query of k-degree shortest path. SymGraph [37] tries to overlap the communication delay between nodes and the computation on the nodes through generating unknown symbols for the vertices without available states of their neighbors. However, when using SymGraph for iterative graph processing on the GPU, high runtime overhead is incurred to generate many unknown symbols and it also needs high reprocessing cost to update all related vertices according to the arrived new vertex state.

TP-X [17] transforms the original graph into a smaller graph to get better initial states for some vertices by first handling this small graph. It aims to accelerate iterative processing of the original graph by avoiding unnecessary traversals of some vertices and edges. However, with the transformation approach proposed in TP-X, the hops between connected vertices may increase, which may induce slow state propagation.

ReGraph [18] tries to reduce problem size by alternately shrinking and repartitioning the graph. HotGraph [33] proposes a novel hot graph based approach to efficiently propagate vertex states.

Wonderland [31] uses a similar approach as HotGraph, while allows the user to define a graph abstraction and then use this abstraction to guide the processing of the graph, such as handling the graph abstraction multiple times within each round of graph processing as our system. Compared with these solutions, the techniques proposed in this paper are specifically for fully exploiting high parallelism of GPUs for faster state propagation, rather than the relatively low parallelism of CPU.

6.2 GPU-Based Graph Processing

Because the GPU has more powerful capacity than the CPU, Medusa [38] proposes to take GPU as an accelerator for graph processing. After that, several GPU-based graph processing systems are developed to satisfy different conditions or to get better performance. GTS [16] stores graphs in PCI-E SSDs to support large-scale graphs. TOTEM [7] proposes to process graphs using the parallelism of both CPU and GPU. Garaph [23] further uses a scheduling algorithm to adaptively dispatch low-value/high-value subgraphs to CPU/GPU and then employing different execution engines to handle these subgraphs on CPU and GPU, respectively. Graphie [9] is further proposed to hide the communication cost between CPU and GPU by asynchronously streaming the edge data into the GPU, while vertex states are stored on the GPU.

Meanwhile, some systems are designed to optimize GPU-based graph processing. CuSha [15] develops new graph representation method for high GPU utilization ratio. For better performance, MultiGraph [11] uses various graph representation and execution schemes based on the ratio of active vertices. Lux [13] uses the aggregated memory bandwidth of the distributed platform with multiple GPUs. Gunrock [26, 29, 30] uses a frontier-based synchronous execution model incorporated with several optimization schemes. Enterprise [19] tries to accelerate BFS on GPUs by selectively caching hub vertices to spare data accesses. However, it is demonstrated to perform worse than Gunrock.

For a balanced load, SIMD-X [20] develops task management scheme to intelligently map the tasks and also filter out inactive vertices, and Tigr [12] iteratively splits the high-degree vertices until their degrees reach a predefined limit. For lower synchronization cost, Groute [4] proposes a GPU-based asynchronous graph processing model. For fewer updates of directed graph processing, DiGraph [35] handles graph along paths and uses a DAG (Directed Acyclic Graph) sketch to schedule these paths' processing order. However, because of many dependencies between vertices, the high performance of GPU is underutilized in existing solutions. This article focuses on the investigation about how to unlock the high parallelism of GPUs to get faster convergence speed for iterative graph processing.

7 CONCLUSION

Thisarticle proposes an effective structure-aware asynchronous execution approach to fully exploit the high parallelism of GPUs for faster convergence speed of iterative graph processing. It is a significant advance over existing GPU-based graph processing solutions because it not only enables most vertex state propagations to quickly traverse much fewer GPU cores to reach other vertices, but also enables vertex state propagations to be effectively conducted along the graph paths with much higher parallelism. Experimental results show that AsynGraph obtains better performance than existing GPU-based solutions, especially for the power-law graphs with longer diameter. In the future, we will research how to extend AsynGraph to heterogeneous platforms for large-scale graph processing and also extend AsynGraph to efficiently support evolving graph processing over GPUs using an efficient incremental way. The proposed approach also can be extended to enable graph algorithms to fully exploit the high parallelism of hardware graph processing accelerators. Thus, we will also research new software/hardware cooperative approaches for high-performance graph processing, including evolving graph processing and streaming graph processing.

REFERENCES

[1] 2019. Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/index.html.

[2] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2017. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk I/O. In *Proceedings of the 2017 USENIX Annual Technical Conference*. 125–137.

[3] Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the 2009 Conference on High Performance Graphics*. 145–149.

[4] 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.

[5] Elisabetta Bergamini, Michele Borassi, Pierluigi Crescenzi, Andrea Marino, and Henning Meyerhenke. 2019. Computing top-k closeness centrality faster in unweighted graphs. *ACM Transactions on Knowledge Discovery from Data* 13, 5 (2019), 1–40.

[6] Hanhua Chen, Hai Jin, and Xiaolong Cui. 2017. Hybrid followee recommendation in microblogging systems. *Science China Information Sciences* 60, 012102 (2017), 1–14.

[7] 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.

[8] 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.

[9] 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.

[10] Loc Hoang, Matteo Pontecorvi, Roshan Dathathri, Gurbinder Gill, and Vijaya Ramachandran. 2019. A round-efficient distributed betweenness centrality algorithm. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 272–286.

[11] 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.

[12] 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.

[13] 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.

[14] Ruoming Jin, Ning Ruan, Bo You, and Haixun Wang. 2013. Hub-accelerator: Fast and exact shortest path computation in large social networks. In *arXiv*. 1–12.

[15] 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.

[16] 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.

[17] Amlan Kusum, Keval Vora, Rajiv Gupta, and Iulian Neamtiu. 2016. Efficient processing of large graphs via input reduction. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. 245–257.

[18] Xue Li, Mingxing Zhang, Kang Chen, and Yongwei Wu. 2018. ReGraph: A graph processing framework that alternately shrinks and repartitions the graph. In *Proceedings of the 2018 International Conference on Supercomputing*. 172–183.

[19] Hang Liu and H. Howie Huang. 2015. Enterprise: Breadth-first graph traversal on GPUs. In *Proceedings of the 2005 International Conference for High Performance Computing, Networking, Storage and Analysis*. 68:1–68:12.

[20] 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.

[21] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A framework for machine learning in the cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012), 716–727.

[22] 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.

[23] 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.

[24] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. 135–146.

[25] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*. 1–14.

[26] 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.

[27] Shuang Song, Xu Liu, Qinzhe Wu, Andreas Gerstlauer, Tao Li, and Lizy K. John. 2018. Start late or finish early: A distributed graph processing system with redundancy reduction. *Proceedings of the VLDB Endowment* 12, 2 (2018), 154–168.

[28] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. 2017. CoRAL: Confined recovery in distributed asynchronous graph processing. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. 223–236.

[29] 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*. 11:1–11:12.

[30] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, and John D. Owens. 2017. Gunrock: GPU graph analytics. *ACM Transactions on Parallel Computing* 4, 2 (2017), 39:1–39:50.

[31] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. 2018. Wonderland: A novel abstraction-based out-of-core graph processing system. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. 608–621.

[32] 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.

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

[34] 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.

[35] 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 Architectural Support for Programming Languages and Operating Systems*. 601–614.

[36] 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.

[37] Long Zheng, Xiaofei Liao, and Hai Jin. 2018. Efficient and scalable graph parallel processing with symbolic execution. *ACM Transactions on Architecture and Code Optimization* 15, 1 (2018), 3:1–3:25.

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

Received April 2020; revised July 2020; accepted August 2020