SEP-Graph: Finding Shortest Execution Paths for Graph Processing under a Hybrid Framework on GPU

Hao Wang†, Liang Geng†§, Rubao Lee‡, Kaixi Hou¶, Yanfeng Zhang§, Xiaodong Zhang†*

†Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA, wang.2721@osu.edu, geng.161@osu.edu, zhang@cse.ohio-state.edu

§Department of Computer Science and Engineering, Northeastern University, China, zhangyf@cc.neu.edu.cn

‡United Parallel Computing Corporation, DE, USA, lirb@unipacc.com

¶Department of Computer Science, Virginia Tech, Blacksburg, VA, USA, kaixihou@vt.edu

Abstract

In general, the performance of parallel graph processing is determined by three pairs of critical parameters, namely synchronous or asynchronous execution mode (Sync or Async), Push or Pull communication mechanism (Push or Pull), and Data-driven or Topology-driven traversing scheme (DD or TD), which increases the complexity and sophistication of programming and system implementation of GPU. Existing graph-processing frameworks mainly use a single combination in the entire execution for a given application, but we have observed their variable and suboptimal performance.

In this paper, we present SEP-Graph, a highly efficient software framework for graph-processing on GPU. The hybrid execution mode is automatically switched among three pairs of parameters, with an objective to achieve the shortest execution time in each iteration. We also apply a set of optimizations to SEP-Graph, considering the characteristics of graph algorithms and underlying GPU architectures. We show the effectiveness of SEP-Graph based on our intensive and comparative performance evaluation on NVIDIA 1080, P100, and V100 GPUs. Compared with existing and representative GPU graph-processing framework Groute and Gunrock, SEP-Graph can reduce execution time up to 45.8 times and 39.4 times.

CCS Concepts • **Computing methodologies** → **Parallel programming languages**; • **Software and its engineering** → *Parallel programming languages*; Massively parallel systems.

*Hao Wang and Liang Geng contributed equally to this work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPoPP '19, February 16–20, 2019, Washington, DC, USA

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6225-2/19/02…$15.00

https://doi.org/10.1145/3293883.3295733

Keywords Graph Algorithms, GPU, Hybrid

1 Introduction

Graph analysis along with many data mining and machine learning algorithms are playing important roles in different disciplines of science, from social networks to Internet of Things, from biology to bioinformatics, etc. The research and development in this area have driven the design and implementation of parallel graph-processing frameworks on multicore CPUs and distributed environments, e.g., Pregel [31], GraphLab [30], PowerGraph [16], PowerLyra [9], GraphX [17], Ligra [42], Galois [36], Gemini [54], etc. On the other hand, due to the massive parallelism, high memory bandwidth, and energy-efficiency of GPU over CPU, GPUs have also become important platforms for graph-processing frameworks, e.g., Medusa [53], CuSha [25], Gunrock [46], Groute [3], Frog [41], Gluon [10], etc.

Many aforementioned software frameworks adopt the "think like a vertex" philosophy [31] to implement the vertex-centric programming model. In this way, programmers implement the computation on vertices, while the frameworks implement communication between vertices along edge and process vertices in parallel at runtime. In general, a parallel graph-processing framework must consider three basic components for the purpose of high performance and scalability. First, the execution of a graph algorithm can be either in a synchronous mode or in an asynchronous mode, a.k.a. Sync and Async. In the sync mode, the whole execution of a program is divided into multiple iterations. An explicit barrier between two successive iterations is required to synchronize the execution. A vertex can see the updates from others only after the end of previous iteration. In contrast, the async mode doesn't need any explicit barrier. A vertex is allowed to see the updates from others as soon as possible. Second, the communication between vertices can be implemented either in a push way or in a pull way. Once a vertex has updates for destination vertices, the vertex can actively push the updates to the destinations, or passively let the destination vertices pull the updates. Third, the vertex traversing can use either a data-driven mechanism or a topology-driven mechanism, a.k.a. DD and TD. DD distinguishes active vertices from inactive vertices, and only those active vertices

will be traversed. TD treats active and inactive nodes equally, and all vertices will be traversed.

In practice on GPU, a combination of these three parameters can be used in an implementation, aiming to best utilize the massive parallelism of GPU and the high memory bandwidth, to achieve load balancing and other high performance goals. However, the combination variations may cause execution and performance dynamics, increasing the complexity and sophistication of programming and system implementation for high performance. In addition, in each of the existing systems [3, 25, 46, 53], a single and fixed combination is used in the entire execution for a given application, which may lead to variable and suboptimal performance. In order to address our uncertainty and doubt, we have looked into the execution of two graph-processing systems by tracing and measuring the execution time at each iteration.

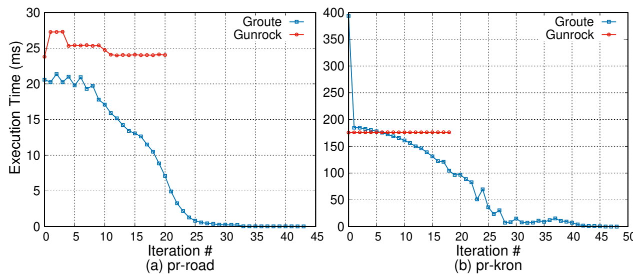

Figure 1. The execution time (in milliseconds) of PageRank from Gunrock and Groute running on an NVIDIA 1080 GPU for different datasets.

Fig. 1 illustrates the per iteration execution time of PageRank from Gunrock [46] and Groute [3] on an NVIDIA 1080 GPU. Both Gunrock and Groute are high-performance graph-processing frameworks on GPUs. The major difference between them is Gunrock uses the sync model and Groute uses the async model. Additionally, Gunrock PageRank adopts Push for the communication and TD for vertex traversing (Sync + Push + TD), while the equivalent of Groute adopts Push and DD (Async + Push + DD). [1] As shown in Fig. 1(a) for the road_usa dataset, although requiring more iterations to converge, Groute has better overall performance than Gunrock (356.5 ms vs. 523.4 ms), due to much less execution time in each iteration. However, in Fig. 1(b) for the kron dataset, Gunrock shows better overall performance than Groute (3346.8 ms vs. 3931.1 ms). These figures also show that Gunrock has consistent execution time in each iteration, because TD traverses all vertices in each iteration. In contrast, Groute has significantly variable execution time, primarily because DD maintains and updates the worklist of active vertices in each iteration. DD also leads to higher execution time in the first several iterations of Groute, as shown in Fig. 1(b).

We have shown a fixed combination in the entire execution may be problematic because execution time in each iteration can be dramatically different. This motivates us to develop an adaptive and hybrid software framework for graph processing, where the execution mode is adaptively and automatically changed with different combinations, aiming for the best overall performance. We will first answer the following two questions to lay a foundation for our system framework.

- Can we find the root causes that lead to variable execution time for a given graph algorithm and its dataset, considering the combinations of execution mode, communication mechanism, and traversing scheme on GPUs?
- After we reveal the root causes and obtain the insights to switch from one alternative to another one, can we implement a lightweight mechanism to switch the solution at runtime, for ensuring the system-level overhead will not offset the benefit of hybrid?

In this paper, we present the design and implementation of SEP-Graph, a highly efficient software framework for graph processing on GPU. The hybrid execution mode is switched among three pairs of parameters (Sync/Async, Pull/Push, and DD/TD), with an objective to minimize the execution time in each iteration. We also apply a set of optimizations to SEP-Graph to optimize performance of algorithms, considering the characteristics of graph algorithms and underlying GPU architectures. We evaluate SEP-Graph with Gunrock and Groute on three types of GPUs with a set of graph algorithms and datasets. The experimental results show the effectiveness of our framework.

2 Background

2.1 Sync vs. Async

Fig. 2(a) compares the different execution modes of Sync and Async. In this figure, we assume thread 0 is scheduled to process vertices 0, 1, 2; and thread 1 is responsible for vertices 3, 4, 5. With Sync, the execution of a program is divided into multiple iterations. To coordinate the multiple threads and ensure the updates on vertices detectable at the end of each iteration, the Bulk Synchronous Parallel (BSP) mode [44] is usually adopted, introducing a barrier between any two successive iterations. With Async, the update on a vertex can be seen by others as soon as possible. As shown in the bottom half of this figure, vertex 0 is scheduled twice in the schedule sequence (0, 1, 0, 1) of thread 0, and the updates are sent to vertex 4 twice. Note that, an algorithm that can be executed synchronously may not be executed correctly in the async mode. Several studies [15, 52] have provided the theoretical foundations for the conditions that sync graph algorithms can be transformed to async ones. These studies guarantee each sync algorithm used in our

[1] The asynchronous framework Groute has iterations, because its data-driven model is implemented on top of a worklist for active vertices and the worklist is updated by a GPU kernel function, as an implicit barrier. An iteration corresponds to an update, and inside an iteration, a vertex can be scheduled multiple times in an asynchronous way.

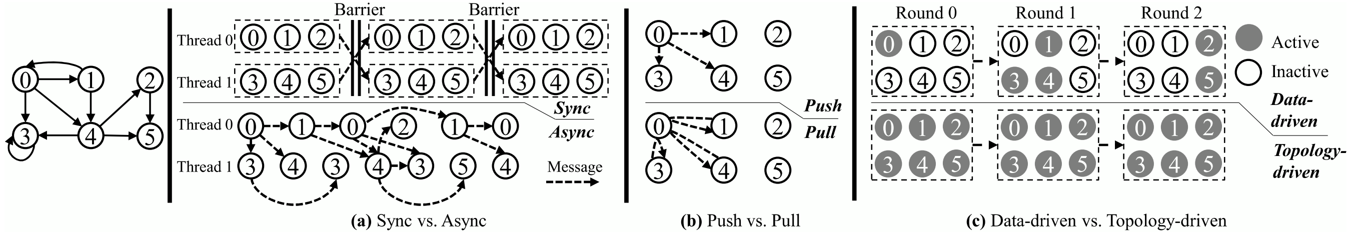

Figure 2. Comparisons of Sync and Async, Push and Pull, and DD and TD. The left-most subfigure is the input graph dataset.

work has an async equivalent with a correctness proof. The transformable problem of sync-to-async is out of the scope of this paper.

There are two major reasons that Async is believed to converge faster than Sync. First, since Async can propagate the update of a vertex to its neighbors as soon as possible, the most up-to-date updates can be used in computations on neighbors. Second, because the fast worker threads can move ahead without waiting for the stragglers, there is no waiting cost. However, on GPUs, there are three structural limits of Async. First, compared to Sync, where a vertex only generates the update once in an iteration, Async can generate updates of a vertex multiple times in a nondeterministic way, incurring more irregular data communication. On GPUs, this may significantly reduce the performance of memory accesses. Second, although Async doesn't require fast threads to wait for the stragglers, this potential benefit may be offset on GPU, because a group of GPU threads are intended to execute the same code at the same time for high performance. Third, Async may incur excessively stale computations [19], because many up-to-date messages can make a computation become stale soon. The redundant stale computations lead to unnecessary computations and higher communication costs.

2.2 Push vs. Pull

Fig. 2(b) compares Push and Pull in graph processing. When vertex 0 has the update for its neighbors, i.e., vertices 1, 3, 4, vertex 0 can push updates to the neighbors; or let vertex 1, 3, 4 pull updates from vertex 0.

Push may lead to the write-write conflict on a destination vertex from multiple sources. For example, vertex 0 and vertex 1 can write updates to vertex 4 simultaneously. Such a write-write conflict exists in both Sync and Async mode. In contrast, Pull doesn't have the write-write conflict when combined with Sync; while it has the read-write conflict when combined with Async: when a source vertex is updating its local buffer for updates, a destination vertex is pulling the update in the same address. Modern hardware architectures provide instructions, e.g., Fetch-and-Add (FAA) and Compare-and-Swap(CAS), to implement atomic read and write. On CPUs, the variable performance of using atomics in graph algorithms has been explored [4, 18, 54]. On GPUs, the overhead of using Push and Pull is also discussed with a few graph algorithms [1, 29]. Most GPU graph-processing frameworks use Push for communication.

2.3 Data-driven vs. Topology-driven

Fig. 2(c) shows the process of vertex traversing in Breadth-First Search (BFS) starting from vertex 0 when using DD and TD, respectively. The shadowed cycles represent the active vertices and the bright cycles represent those inactive. DD will traverse vertex 0 in the first round, then vertices 1, 3, 4 in the second round, and vertices 2 and 5 in the third round; while TD will traverse all vertices in each round no matter they are active or inactive.

DD is usually implemented with a worklist- or queue-based method. Some graph-processing frameworks with Sync [25, 46, 53] use two worklists, one for active vertices in the current round and the other for active vertices in the next round. In each iteration, the active and inactive worklists are swapped. The active vertices are also called "frontiers" in some proposals using DD [29, 33, 47]. When integrated with Async, DD only needs one worklist to store active vertices. Each thread will independently manipulate the worklist to dequeue a vertex and enqueue new discovered vertices. However, such a combination of Async and DD is opposite to the GPU preference, which prefers all threads to do the same thing at the same time. As a result, a better solution for GPUs is to schedule GPU threads together for the worklist management and allow threads to execute asynchronously between two times of worklist updates. We adopt this method to combine Async and DD.

In contrary, TD doesn't distinguish active vertices from inactive ones, simplifying the implementation of systems. Although TD has unnecessary computations on inactive vertices, TD doesn't always underperform, when compared to DD. First, the performance of DD is affected by the overhead of worklist management and the saved computation after getting rid of inactive vertices. Second, the redundant computation induced by TD may be offset by the large number of GPU threads.

3 Rationale of Performance Dynamics

In this section, we design and implement two typical graph algorithms by using different combinations of Sync or Async, Push or Pull, and DD or TD with a set of optimizations on GPU. Through analyzing the variable performance of

algorithms on different datasets, we provide the insights of when a specific combination can obtain good performance.

3.1 Algorithms and Datasets

Based on their behaviors, graph algorithms can be put into two categories [4]. **Iterative** algorithms try to derive some properties of the whole graph and satisfy: (1) starting from all vertices of the graph; (2) processing vertices in iterations until some convergence condition is hit. PageRank and Triangle Counting belong to this category. **Traversal** algorithms try to derive some properties for a subgroup of graph and usually require: (1) starting from one or few vertices; (2) processing vertices along a direction until the end condition is hit. BFS and SSSP belong to this category. We select PageRank and SSSP to explore the reasons of variable performance.

PageRank: PageRank is an algorithm to rank websites based on the number and quality of their links. Since the original algorithm [7] cannot be executed correctly with Async [52], we use the *Delta-based* PageRank [48, 52] that has both sync and async implementations. This algorithm allocates buffers PR_j and ΔR_j on a vertex j to record the PR value and receive updates. The computation on a vertex is to accumulate ΔR_j. When ΔR_j is added to PR_j, the delta message $d\frac{\Delta R_j}{|N(j)|}$ (d is the damping factor and $|N(j)|$ is the outdegree of j) is sent to neighbors of j and ΔR_j is reset to 0. With Sync, a vertex will receive updates produced in the previous iteration, accumulate updates, and send delta messages to neighbors. Async will do these steps independently.

SSSP: SSSP is an algorithm to find a shortest path from a given vertex s to all other vertices $j \in \mathbb{V} - \{s\}$ in the graph. Δ-stepping SSSP [34] is a generalization of the Dijkstra's algorithm and the Bellman-Ford algorithm. Δ-stepping SSSP allocates buffer D_j and ΔD_j on a vertex j to record the current shortest distance and the received smallest distance from its neighbors. A vertex will send its current shortest distance to the outgoing neighbors. The ΔD_j of its neighbors will be updated with the smallest received distance, and the D_j of its neighbors will be updated as $D_j = min\{D_j, \Delta D_j\}$. Δ-stepping SSSP can be optimized by processing a set of vertices with shortest distances (i.e., smallest D_j) in a mini-batch (where the batch size is Δ). The vertices with the updated shortest distances are enqueued in the mini-batch, and the vertices in the mini-batch are processed again and again until no vertex's D_j changes. Δ-stepping SSSP can be implemented synchronously where the synchronization is required after a round of computation on each mini-batch; while in Async, the threads process mini-batches independently. We use the *Near-Far* optimization [11], which is a practical design of Δ-stepping SSSP on GPU.

Datasets: We use road_usa and kron_21 datasets in the experiments of this section. The former is a typical sparse graph having the high diameter; while the latter is a graph whose in- and out-degree obey the power-law distribution.

3.2 Analysis of Iterative Algorithms

We analyze PageRank, a representative iterative algorithm, by profiling all possible combinations of execution variables, as shown in Fig. 3(a, b) and Fig. 4(a, b).

Data-driven or Topology-driven: Overall, TD outperforms DD, because iterative algorithms often start from all vertices and only a few vertices might satisfy convergence conditions at the beginning stages. DD has the worklist/queue management overhead. Such overhead cannot be offset by removing the converged vertices in the first several iterations when very few vertices can converge. This observation holds for different datasets. Fig. 4 (a, b) show TD is always preferred over DD when the execution mode and communication mechanism stay the same.

Push or Pull: The choice depends mainly on the graph structure. For example, Push works well for high-diameter graphs, e.g., road_usa, and Pull is better for scale-free graphs, e.g., kron_21, when combined with Sync. For high-diameter graphs, Push generates less overhead in its atomics and shows consistently better performance, thanks to the relatively low in/out degree of all vertices. In contrast, the high in-degree vertices in scale-free graphs could form a bottleneck for such an atomic-based push mechanism. Pull, however, is favorable in this case by assigning working threads to actively pull the updates from in-neighbors one by one without the write-write conflict in Push. Note that the updates should be synchronized; otherwise, the pull will cause the read-write conflict.

Sync or Async: Async is slightly better than Sync, if they exhibit the similar performance trends. One main reason is the faster convergence speed of Async. Thus, in our hybrid implementation for iterative algorithms, Async is given a higher priority than Sync.

Considering all these factors, we choose "sync-pull-td", "async-push-td", and "async-push-dd" as the candidates for iterative algorithms in our experiments, expecting the runtime system can switch these methods adaptively and rapidly.

3.3 Analysis of Traversal Algorithms

We have the following observations for the representative traversal graph algorithm SSSP.

Data-driven or Topology-driven: DD outperforms TD in most cases, because DD matches the traversal nature of this type of algorithm. Specifically, DD only tracks a small fraction of vertices for high-diameter graphs.

Push or Pull: Push and Pull have their own best scenarios, especially for the scale-free graph, as shown in Fig. 3(c). We have to include both Push and Pull as the candidates for the hybrid solution. The intuition of switching is related with the number of current active vertices, the total out-edges of current active vertices, and the number of untouched edges.

Sync or Async: By using DD, Async and Sync show similar trends of performance dynamics, regardless of input graphs.

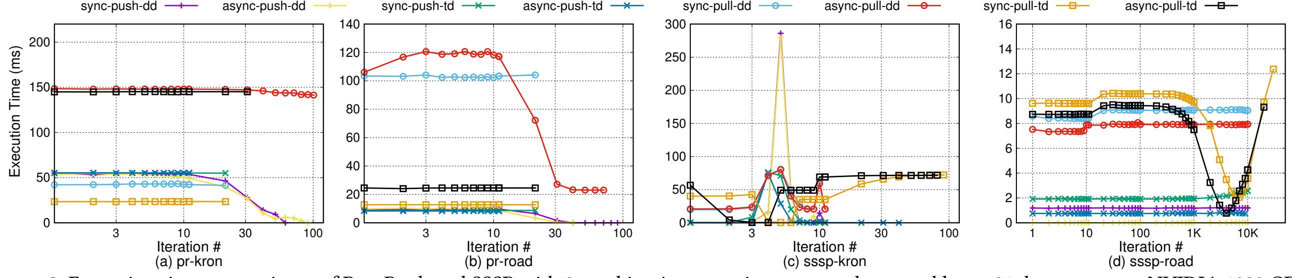

Figure 3. Execution time comparisons of PageRank and SSSP with 8 combinations running over road_usa and kron_21 datasets on an NVIDIA 1080 GPU.

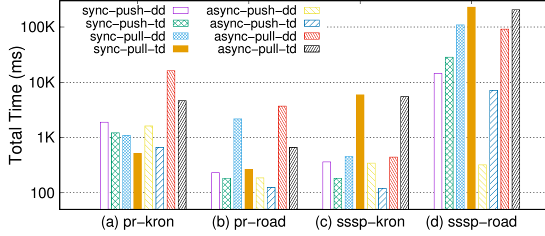

Figure 4. Accumulated execution time comparisons of PageRank and SSSP with 8 combinations for Fig. 3.

One reason is the async implementation with DD on GPU. As mentioned in Sec. 2.3, Async-DD needs to swap the worklists for active vertices, as an implicit barrier. Another reason for the close performance is the characteristics of traversal algorithms. For example, SSSP traverses the graph along multiple paths and updates distances along them. The threads working on the same path will touch the same vertices and do the same computations, no matter in which implementation.

Considering all these observations, traversal algorithms on GPU will benefit from Async-DD-based implementations with Push and Pull hybrid for scale-free graphs. However, for the high-diameter graph, as shown in Fig. 3 (d), the traversal algorithm will execute over 10,000 iterations. That indicates the GPU kernel fusion [45] and GPU utilization in each iteration is more important for such scenarios.

4 System Design and Implementation

4.1 Overview of System Design

SEP-Graph is a runtime system that executes graph algorithms in a hybrid way on GPU. From the point of view of programmers, SEP-Graph is yet another vertex-centric programming model. SEP-Graph provides a set of interfaces for programmers. Once the programmer defines the necessary computation function, communication function, and convergence function on vertices, SEP-Graph can schedule the program to be running on GPU in a hybrid way.

The core of SEP-Graph includes a hybrid engine running on GPU and a controller running on CPU. Fig. 5 shows the GPU part. At runtime, the controller running on CPU will iteratively collect the monitoring data on GPU and switch the execution path of a graph algorithm on GPU. The hybrid engine on GPU includes the basic implementations of eight execution paths of Sync or Async, Push or Pull, and DD or TD. These implementations are built on top of built-in sparse data structures, e.g., Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC). The runtime monitor inside the hybrid engine monitors and records the changes of parameters, e.g., the number of active vertices, the numbers of accumulated in- and out-degree of active vertices, etc. These parameters will be used in Alg. 1 and Alg. 2 by the controller running on CPU to determine the execution path. The hybrid engine also includes a set of optimizations to optimize algorithm themselves and the mapping on different GPU architectures.

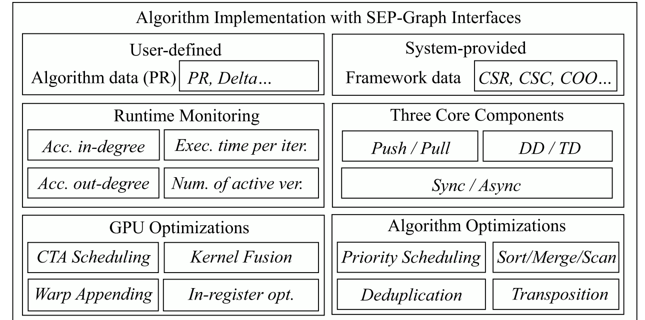

Figure 5. Overview of SEP-Graph hybrid engine on GPU.

4.2 Programming Interfaces

Tab. 1 describes some programming interfaces of SEP-Graph that programmers need to implement. In our model, each vertex has two buffers: one is the value buffer of the vertex and the other is the update(message) buffer of the vertex. Programmers implement the functions `InitValue()` and `InitBuffer()` to initialize this pair of buffers on each vertex. For example, for PageRank, one can do "`return 0;`" in `InitValue()` and "`return (1 - ALPHA);`" in `InitBuffer()`, where ALPHA is the damping factor. The framework will call these functions to initialize the graph at the beginning of execution.

The computation function `ComputeBuffer()` is to compute updates in the message buffer and apply to the value of the vertex. For PageRank, one can do "`buf = atomicExch (buffer, 0);`" and "`*value += buf;`". After that, one

Table 1. SEP-Graph Programming Interfaces

API	Summary
Initialization APIs	
`TValue InitValue(...)`	Return an initial value for every vertices. If a vertex id is given, operate on a vertex.
`TBuffer InitBuffer(...)`	Return an initial value for the buffer on every vertices. If a vertex id is given, operate on a vertex.
Computation and Communication APIs	
`Pair<TBuffer, bool> ComputeBuffer(...)`	Compute updates in the buffer and apply to the vertex. Return a new TBuffer for the communication.
`Int AccumulateBuffer()`	Pass messages from the source to the destination, and accumulate the update in the source buffer to the destination. If the source and destination are not given, operate on all vertices with their neighbors.
Convergence APIs	
`bool IsActive(...)`	Determine if vertices are converged or not (active or inactive). If a vertex id is given, operate on a vertex.

should set the temporary variable "buf", as "`buf = ALPHA * buf / out_degree;`", which will split the Δ value of a vertex for neighbors; and return "buf" to the framework as "`return utils::pair<TBuffer, bool>(buf, true);`". After the computation function, the framework will use the pull or push for the user defined communication function `AccumulateBuffer()` to accumulate updates, as "`atomic Add(buffer, buf);`". With Push, "buffer" is the message buffer of each destination vertex and "buf" is the return value of the current vertex; while with Pull, "buffer" is the message buffer of the current vertex and "buf" is the return value of each source vertex. No matter in which mode, one provides the same implementation of this communication function and the framework will handle the communication correctly.

Note that there is no data-race problem. First, the accesses on the global variable "buffer" are atomic. Second, our shared queue design guarantees only one thread is scheduled to process one vertex once a time, and thus only that thread can access the global variable "value". We also would like to mention that instead of implementing one for each combination of Sync or Async, Push or Pull, and DD or TD, users only need to implement these interfaces once and the system will call the corresponding low-level implementation at runtime.

4.3 Runtime Switch

We design two switching mechanisms for two types of graph algorithms. Once a programmer points out the type of a graph algorithm, SEP-Graph will run Alg. 1 for the iterative type and Alg. 2 for the traversal type.

For iterative algorithms, any combination may lead to the max efficiency in the next round. Therefore, we run the candidates one by one at the beginning to get their execution time; and for the remaining rouds, we predict the execution time and choose the one resulting in the shortest execution time for the next iteration. Alg. 1 shows how SEP-Graph makes decisions for iterative graph algorithms over a graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$. The candidates are contained in **GPUGA**, e.g., sync_push_dd(·), sync_pull_td(·), etc. outDegree(·) and inDegree(·) calculate the out degree and in degree of a given set of vertices, respectively. normalizeTime(·) attempts to normalize the execution time of a given graph algorithm based on the statistics of active vertices **A** in current round, and predictTime(·) predicts the execution time of next round according to the normalized execution time and the statistics of latest active vertices. It is worth noting that the implementations of normalizeTime(·) and predictTime(·) for high accuracy vary towards different graph algorithms. For example, a reliable implementation of normalizeTime(·) for a DD-Push-based algorithm could be $T^n = \frac{t}{\text{outDegree}(\mathbf{A})}$, where t is the measured execution time over **A**, and predictTime(·) can be implemented as $T^p = \text{outDegree}(\mathbf{A}') \cdot T^n$, where $\mathbf{A}'$ is the active vertices for the next round. At the beginning of Alg. 1, the active vertices **A** are composed of **V**. In the first N rounds, SEP-Graph executes the available algorithms in turn to initialize T_i^n. Until the program ends, which is indicated by "isConverged(**G**) ≠ true", SEP-Graph always predicts how long a candidate needs for the next round based on the normalized execution time T^n and the statistics of active vertices, and selects the shortest one to execute. After each round, T_i^n is updated based on the real execution time.

Algorithm 1 SEP-Graph for iterative graph algorithms

```
 1: procedure SEP-GRAPH-ITERATIVE(G(V, E))
 2:     Input: outDegree(·), inDegree(·), isConverged(·)
 3:     GPUGA ← { sync_push_dd(·), sync_pull_td(·), . . . }
 4:     N ← |GPUGA|
 5:     A ← V
 6:     for i in 1 . . . N do
 7:         out_cur ← outDegree(A)
 8:         in_cur ← inDegree(A)
 9:         ▹ A is updated by GPUGA(·)
10:         t ← GPUGA_i(G, A)
11:         T_i^n ← normalizeTime(t, out_cur, in_cur,)
12:     end for
13:     while isConverged(G) ≠ true do
14:         out_cur ← outDegree(A)
15:         in_cur ← inDegree(A)
16:         T^p ← predictTime(T^n, out_cur, in_cur)
17:         i ← indexOf(min(T^p))
18:         t ← GPUGA_i(G, A)
19:         T_i^n ← normalizeTime(t, out_cur, in_cur)
20:     end while
21: end procedure
```

For traversal algorithms, the framework only selects the policy between Push and Pull with Async-DD (See the analysis in Sec. 3.3). Alg. 2 illustrates how the framework makes the decision, in which v_s is the source vertex in the input graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$. α, β, γ, and δ are constants, and all of them are predefined to assist making the determination. neighbor(·) is the function to calculate all neighbors of a given set of vertices. For a specific algorithm like SSSP, an edge is likely to be visited more than once, so that E' is scaled by a constant $\gamma \geq 1$.

This algorithm is initialized with Push, because there is only one active vertex v_s in the worklist, and no duplicate messages are generated. Before the algorithm stops (there are no active vertices in the worklist), the determination

will be made that if the current policy should switch to the alternative. The best opportunity switching to Pull is when the framework discovers the out degree of the vertices in the worklist accounts for a large proportion of the untouched edges, which is defined as $\frac{E'}{\alpha}, \alpha \geq 1$. The basic idea of switching from Push to Pull is that: a large amount of out-degree of the active nodes implies Push is likely to generate plenty of duplicate messages, while a small number of untouched edges indicates relatively small overhead of using Pull. Only if the active vertices lowers than a small number represented by $\frac{|V|}{\beta}$, where $\beta \geq 1$, the framework switches back to Push without worrying the duplicate messages. This idea of switching between Pull and Push is similar to the direction-optimizing BFS [1]. Moreover, the switch only occurs in a dense graph (with the scale-free characteristics); and SEP-Graph always use Push for sparse graphs, if the average edges to a vertex $\frac{|E|}{|V|}$ is less than a threshold δ.

Algorithm 2 SEP-Graph for traversal graph algorithms

1: **procedure** SEP-GRAPH-TRAVERSAL(G(V, E), v_s, α, β, γ, δ)
2: Input: outDegree(·), neighbor(·)
3: GPUGA ← { push(·), pull(·) }
4: A ← $\{v_s\}$
5: $E' \leftarrow |E| \cdot \gamma$
6: $policy$ ← push
7: **if** $\frac{|E|}{|V|} < \delta$ **then**
8: GPUGA(G, A, $policy$)
9: **else**
10: **while** A ≠ ∅ **do**
11: ▷ A is updated by GPUGA(·)
12: GPUGA(G, A, $policy$)
13: **if** $policy$ = push **then**
14: $E' \leftarrow E' -$ outDegree(A)
15: **if** outDegree(A) $> \frac{E'}{\alpha}$ **then**
16: $policy$ ← pull
17: **end if**
18: **else if** $policy$ = pull **and** $|A| < \frac{|V|}{\beta}$ **then**
19: $policy$ ← push
20: **end if**
21: **end while**
22: **end if**
23: **end procedure**

4.4 Optimizations

A set of optimizations have been implemented in SEP-Graph. We introduce three of them.

4.4.1 CTA Scheduling

SEP-Graph adopts Cooperative Thread Array (CTA) scheduling [23, 26] to ensure the load balance and improve the data locality in the communication. When a working thread pushes the update of a vertex to its out-neighbors or pulls the updates from its in-neighbors, the out-degree and in-degree may be significantly diverse, thus resulting in load imbalance between GPU threads. This is observed in most scale-free graphs. SEP-Graph uses a warp of threads or a block of threads to keep balance: if the degree is larger than the block size, the work on the vertex will be processed by a CTA and via the shared memory; if the degree is larger than the warp size, the work will be processed by a warp and via the register; and otherwise, threads will process different vertices independently.

In addition, CTA scheduling can improve the data locality, if thread blocks working on the same group of vertices are scheduled on the same SM to share L1 cache. We can achieve this goal by manipulating the CTA indices and leveraging the round-robin manner of GPU hardware scheduling [28]. With it, SEP-Graph can obtain better data locality in the push- and pull-based communication.

4.4.2 Warp Appending

Because multiple threads may write active vertices to the shared queue for the next iteration, the atomic write is usually required and may harm the performance. The warp appending [21] is used to reduce the number of atomic operations as below.

First, a warp of threads call the warp voting instruction `__ballot()` and population counting instruction `__popc()` to count the number of threads that have updates, i.e., active vertices, and then call the data shuffle instruction `__shfl()` to broadcast the counting result in the warp. Second, the first thread of each warp coordinates to get the warp offset in the shared queue via atomic operations. Third, threads write their updates to the queue in parallel with the offsets. This process is called iteratively until all updates are written to the shared queue. We also notice that a recent study [43] can reduce the overhead of atomics on GPU with the relaxed atomics. We will investigate this new memory consistency model and integrate it into our framework.

4.4.3 Priority Scheduling

Priority scheduling can optimize graph algorithms by prioritizing some vertices to be executed first after distinguishing these vertices from others. In Δ-based PageRank, the condition to determine a vertex active or inactive is based on the Δ value. Scheduling the vertices having larger Δ values first can accelerate the convergence speed, since the Δ value is decreased by the damping factor in the propagation.

SEP-Graph has different implementations of priority scheduling for iterative algorithms and traversal algorithms. The difference is in the method of calculating a proper threshold. Because iterative algorithms operate on the whole graph through all iterations, the threshold of the priority scheduling, e.g., the Δ value of PageRank, should consider all active vertices. We use the random sampling to get current values of priority scheduling variable on sampled vertices, and then calculate the threshold. In PageRank as an example, we sample 1000 vertices, sort them on the Δ value, and use the

200th Δ value as the threshold. This method has been successfully adopted by distributed graph-processing systems, e.g., Maiter [52] and PowerSwitch [50], and has much less overhead on GPU. Calculating the threshold for a traversal algorithm is simpler, because we only need to consider the traversed vertices along the path, instead of the whole graph. For example, in Δ-stepping SSSP, we can pre-define a value of distance as the threshold and put the active vertices having shorter distances to the source to be scheduled first.

5 Evaluation

In this section, we evaluate SEP-Graph with four algorithms. Besides PageRank and SSSP, other two algorithms are:

- **Breadth-First Search (BFS):** BFS traverses a graph from a given source vertex, and outputs the hops from the source to traversed vertices.
- **Betweenness Centrality (BC):** BC measures the centrality of a graph by calculating the centrality of each vertex. The centrality of a vertex is the number of the shortest paths that pass through the vertex. We use the algorithm proposed by Brandes [6] due to its fast speed and generality. In the evaluation, we compute the centrality of a given vertex.

We compare SEP-Graph with two GPU graph-processing frameworks. Gunrock (version 0.4) [14] is a sync-based framework. In contrast, Groute [13] is an async-based framework. We carry out our experiments on three types of compute nodes.

The first server has an NVIDIA GTX 1080 GPU, which has 20 SMs (2560 CUDA cores) and 8 GB GDDR5 memory. This server also has an Intel i7-3770K CPU running on 3.5GHz (4 cores) and 32 GB memory. The second server has two Intel Xeon E5-2680v4 CPUs running on 2.4GHz (28 cores in total), 512 GB memory, and two NVIDIA P100 GPUs. Each P100 GPU has 60 SMs (3840 CUDA cores) and 12 GB HBM2 memory. The third server has two Intel Xeon Gold 6136 CPUs running on 3.0GHz (24 cores in total), 256 GB memory, and two NVIDIA V100 GPUs. Each V100 GPU has 5120m CUDA cores and 16 GB HBM2 memory. The GPUs are connected to the host via PCIe Gen3 on all systems. The software environments, e.g., CUDA, GCC, Linux kernel, etc., will be introduced in the artifact evaluation. We compare SEP-Graph with Gunrock and Groute on a single GPU on these systems.

Tab. 2 lists out the used datasets. "kron_g500-logn21", "soc-LiveJournal1", and "road_usa" are downloaded from the Suite Sparse Matrix Collection [12], and "soc-orkut" and "soc-twitter-2010" are downloaded from Network Data Repository [40]. As mentioned earlier, "road_usa" is a typical high-diameter graph. The threshold is set to 5, which means if the average degree of a graph ($\#Edges/\#Vertices$) is smaller than 5, the graph is detected as a high-diameter graph. We also use this value for Gunrock and Groute for the fair comparison. Other graphs in the table are scale-free graphs. In the evaluations, we run each experiment multiple times and report the average time, and the pre- and post-processing time is excluded.

Table 2. The list of graph datasets

Name	#Vertices	#Edges	Avg. Degree	Max Degree
kron_g500-logn21	2.1M	182.1M	86.7	213.9K
soc-LiveJournal1	4.8M	68.9M	14.4	20.3K
soc-orkut	3M	106.3M	35.4	27.5K
soc-twitter-2010	14.8M	265.0M	17.9	302.7K
road_usa	23.9M	57.7M	2.4	9

5.1 Comparisons On NVIDA 1080 GPU

Fig. 6 shows the performance comparisons of SEP-Graph, Groute, and Gunrock on the node having an NVIDIA 1080 GPU. As shown in the figure, SEP-Graph is the winner, except Gunrock BFS on the scale-free graphs.

5.1.1 PageRank

Fig .6(a) shows the performance comparisons of PageRank. SEP-Graph can get up to 2.9x and 1.8x speedups over Groute and Gunrock for scale-free graphs, both on kron; and obtain 1.9x and 2.5x speedups over Groute and Gunrock for the high-diameter graph road_usa.

For this iterative algorithm, SEP-Graph runs Alg. 1 to switch the execution path. The log file shows SEP-Graph switches the execution path of PageRank from "sync-pull-td" to "async-push-dd" for the scale-free graphs, including kron, liveJournal1, orkut, and twitter; and switch the path from "async-push-td" to "async-push-dd" for the high-diameter graph usa_road. On the contrary, the PageRank implementations of Groute and Gunrock always work on "async-push-dd" and "sync-push-td", respectively. As mentioned earlier in Fig. 3(a), compared to "sync-pull-td", neither "async-push-dd" nor "sync-push-td" is a good candidate for the iterative algorithm running on the scale-free graphs. First, compared to Pull, using the atomic operation in Push leads to non-negligible performance penalty on the scale-free graph. Second, the overhead of worklist management makes DD not suitable for the iterative algorithm, until most vertices of the graph are converged. SEP-Graph can identify "sync-pull-td" to be the best start in the inspection stage of Alg. 1, and switch to "async-push-dd" when active vertices become very few. On the high-diameter graph, SEP-Graph detects "async-push-td" is best in the inspection and will switch to "async-push-dd" in the final iterations.

Fig. 7 shows the switched execution path and execution time of SEP-Graph PageRank on kron and usa_road, respectively. The monitoring parameters, including the number of active vertices, the number of in-degree of all vertices, and the number of out-degree of active vertices, are also shown in the figure at the switching points. As we choose "sync-pull-td", "async-push-td", and "async-push-dd" as the candidates for the iterative type algorithm, SEP-Graph will

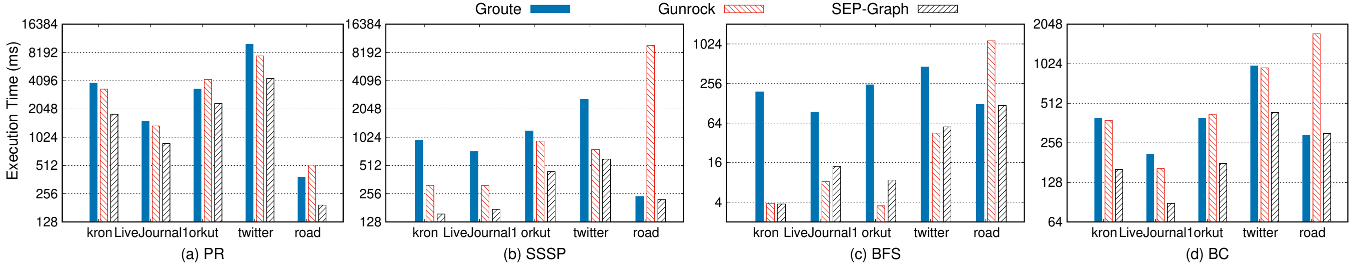

Figure 6. Performance comparisons of SEP-Graph with Groute and Gunrock on an NVIDIA GTX 1080 GPU.

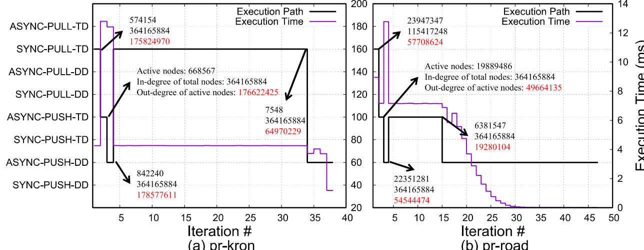

Figure 7. SEP-Graph PageRank execution path on an NVIDIA 1080 GPU.

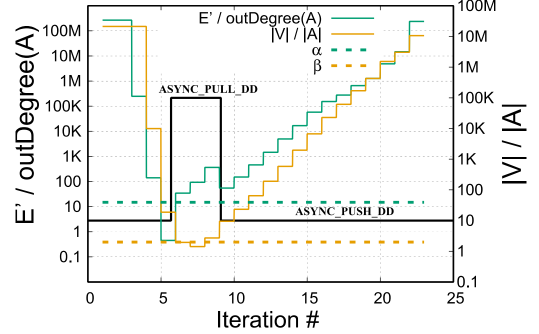

Figure 8. SEP-Graph SSSP execution path for the twitter dataset on an NVIDIA 1080 GPU.

run the candidates one by one in the first three iterations, and record the execution time per iteration (78 ms, 182 ms, 180 ms). At the end of third iteration, SEP-Graph observes "sync-pull-td" has the shortest execution time, and then switch to it at the forth iteration. At the end of each following iteration, SEP-Graph updates the monitoring data and predicts the execution time of these three methods. Because the execution time of "sync-pull-td" and "async-push-td" is nearly stable in the iterative algorithm, SEP-Graph only needs to predict the execution time of "async-push-dd" with Alg. 1. As mentioned in Sec. 4.3, the predicted execution time of a Push-DD-based iterative algorithm is linear with the number of out-degree of active vertices. At the end of iteration 34, the predicted execution time of "async-push-dd" can be simplified as $its_inspect_time * (curr_out - degree/inspect_out - degree)$, 180 * (64970229 / 176622425) = 66.2 ms, which is smaller than the inspection time of "sync-pull-td" and "async-push-td", i.e., 78 ms and 182 ms. The execution path will switched to "async-push-dd" after the iteration 34. We skip the detailed analysis of switched execution path of PageRank on usa_road in Fig. 7(b) for the space limitation.

5.1.2 Single-Source Shortest Path

Before running SSSP algorithms, we use the pre-processing tool from Gunrock to assign the weight to each edge in the graph. Like SEP-Graph, we also enable the priority scheduling in Groute and Gunrock for their best performance.

In traversal algorithms, SEP-Graph running with Alg. 2 can switch the execution path between "async-push-dd" and "async-pull-dd" for the scale-free graphs, and always use "async-push-dd" for the high-diameter graphs. In contrast, Groute and Gunrock work on "async-push-dd" and "sync-push-dd" for the entire execution of SSSP, respectively. Although Gunrock provides a hybrid implementation using Push and Pull for BFS [39], so far it is not enabled for other algorithms like SSSP. Fig. 6(b) shows the performance comparisons of SSSP. For the scale-free graphs, SEP-Graph can get up to 6.5x and 2.1x speedups over Groute and Gunrock on kron and orkut datasets, respectively. For the high-diameter graph road_usa, SEP-Graph can deliver 1.1x and 39.4x speedups over Groute and Gunrock.

Fig. 8 illustrates the switching points of SEP-Graph SSSP on twitter. In this figure, the y1 axis and y2 axis represent the variation of $\frac{E'}{\text{outDegree(A)}}$ and $\frac{|V|}{|A|}$ along iterations. The black solid line indicates the execution path between "async-push-dd" (lower) and "async-pull-dd" (upper). The dash lines mean the constants of α (green) and β (yellow), which are set to 15 and 2, respectively. During execution, SEP-Graph compares $\frac{E'}{\text{outDegree(A)}}$ to α to determine if switching from push to pull and compares $\frac{|V|}{|A|}$ to β to determine if switching from pull to push. The rules are shown in Alg. 2. We can clearly observe that as $\frac{E'}{\text{outDegree(A)}}$ (the green solid line) becomes lower than α at the end of the 5th round, and SEP-Graph changes to the pull mode in the next iteration for higher performance. Once $\frac{|V|}{|A|}$ (the yellow solid line) is higher than β at the end of the 8th round, SEP-Graph switches back to the push mode in the 9th round.

For the high-diameter graph road_usa, SEP-Graph uses "async-push-dd" for the entire execution as Groute does. The better performance of SEP-Graph comes from the optimizations of sorting and de-duplication in SEP-Graph for the Push-DD mode, and the additional overhead of worklist

management in Groute for supporting multiple GPUs. The significant overhead of Gunrock is related with the GPU SM utilization and the kernel fusion. The kernel fusion in Gunrock only fuses the advance operator and the filter operator in successive iterations [39], leading to multiple times of kernel launch. While, in Groute and SEP-Graph, the SSSP kernel is only launched once for the high-diameter graph. The GPU SM utilization will be discussed in Sec. 5.2 with the performance numbers on NVIDIA P100 and V100 GPUs.

5.1.3 Breadth-First Search

Fig. 6(c) shows the performance comparisons of BFS. SEP-Graph has 7.0x speedup over Gunrock on the high-diameter graph road_usa; while Gunrock has the best performance for the scale-free graphs, having up to 2.4x speedup over SEP-Graph. There are two major reasons for the best performance of Gunrock. First, Gunrock optimizes the forward traversal of BFS by using bitmasks in the recent work [38], which is not implemented in SEP-Graph yet. Second, because Gunrock has implemented the Push and Pull optimization for BFS on scale-free graphs, Gunrock BFS has the same execution path of SEP-Graph. Therefore, compared to Gunrock, SEP-Graph has additional overhead of running the switching algorithm, which is relatively higher in BFS (3.8 56 ms, occupying 6% to 20% of total execution time) than other algorithms.

Compared to Groute, SEP-Graph has up to 45.8x speedup (kron) on the scale-free graphs, and very similar performance on the high-diameter graph road_usa. This also indicates the importance of Push and Pull hybrid for the traversal algorithms on scale-free graphs.

5.1.4 Betweenness Centrality

The BC implementation includes two stages. The first stage is to run the BFS algorithm from the source vertex, and the second stage is a trace back that computes the centrality from traversed vertices to the source. As Groute doesn't provide a BC implementation, we use the Groute BFS and the trace back of SEP-Graph to implement Groute BFS. Therefore, the BC performance comparisons between SEP-Graph and Groute illustrates the same trend of BFS comparisons. Fig. 6(d) shows the SEP-Graph has 2.2x to 2.5x speedups over Groute on the scale-free graphs, and very close performance on the high-diameter graph road_usa. Compared to Gunrock, SEP-Graph has 1.8x to 2.4x speedups on the scale-free graphs, and 5.8x speedup on the high-diameter graph.

5.2 Comparisons on NVIDIA P100 and V100 GPU

We also conduct the same experiments on NVIDIA P100 and V100 GPUs, where the same trend of performance comparisons has been observed. On P100, SEP-Graph has up to 2.8x, 4.1x, 20.2x, and 2.6x speedups over Groute when running PageRank, SSSP, BFS and BC; and the speedup numbers change to 1.6x, 12.7x, 2.4x and 2.1x when comparing to Gunrock. As Groute doesn't support the newly released V100 yet, so that we only measure the performance of Gunrock and SEP-Graph on V100. The speedups of our framework over Gunrock are 2.8x (PageRank), 8.4x (SSSP), 3.9x (BFS) and 3.5x (BC).

We observe that compared to SEP-Graph and Groute, Gunrock exhibits more variable performance on different GPUs. For example, for SSSP with the road_usa dataset, Gunrock needs 8699.5 ms, 4216.4 ms, 1676.5 ms on the nodes having 1080, P100, and V100 GPUs, respectively; while SEP-Graph needs 220.8 ms, 332 ms, and 199.9 ms. The significantly variable performance of Gunrock is related with the numbers of SMs of different GPUs. Once we set the grid size of Gunrock on V100 to 20, which is the number of SMs of 1080, the execution time of Gunrock SSSP changes to 5955.8 ms. Another reason is related with the kernel launch overhead. Once SEP-Graph detects a high-diameter graph in a traversal algorithm, SEP-Graph will launch the kernel once with "async-push-dd". In contrast, Gunrock still launches the kernel multiple times. For this case, Gunrock uses 6791 iterations to launch GPU kernels. The kernel launch overhead and the consequent throughput drop in the launch can not be ignored [37, 51].

6 Related Work

A lot of studies focus on accelerating a single graph algorithm on GPU. Bisson et al. [5] use the sparse linear algebra for implementing the PageRank algorithm on GPU, and further optimize it by overlapping the kernel computation and the data communication between CPU and GPU. Merrill et al. [33] optimize BFS on GPU by using a fast scan implementation for the frontier computation and a multi-granularity scheduling for load balancing. Liu and Huang [29] optimize BFS on GPU by implementing the direction optimization [1]. Pan et al. [38] optimize BFS on multiple GPUs by combining the direction optimization and the efficient data communication between multiple GPUs. Davidson et al. [11] propose the *Near-Far* optimization for SSSP on GPU, which is a practical implementation of Δ-stepping SSSP. McLaughlin and Bader [32] propose a hybrid betweenness centrality on GPUs, which can switch between the work-efficient mode and edge-parallel mode based on the change of vertex frontiers.

Many efforts have been devoted to the study of graph algorithms on GPU. Che et al. [8] propose Pannotia, a GPU benchmark suite for graph algorithms. They also conduct the benchmark characterization on different GPUs. Wu et al. [49] analyze the impact of GPU performance factors, e.g., synchronization, load balancing, etc., on graph algorithms. Kaleem et al. [22] investigate several synchronization strategies for graph algorithms on GPUs, and conclude there is no "one-size-fits-all" solution for different datasets and GPU architectures. Li et al. [27] propose the warp consolidation to improve the data locality and reduce the synchronization overhead on GPU. They show the effectiveness of this

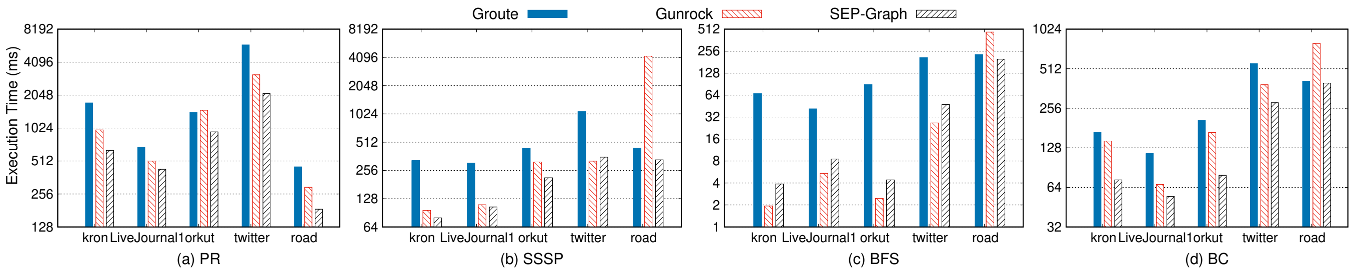

Figure 9. Performance comparisons of SEP-Graph with Groute and Gunrock on an NVIDIA Tesla P100 GPU.

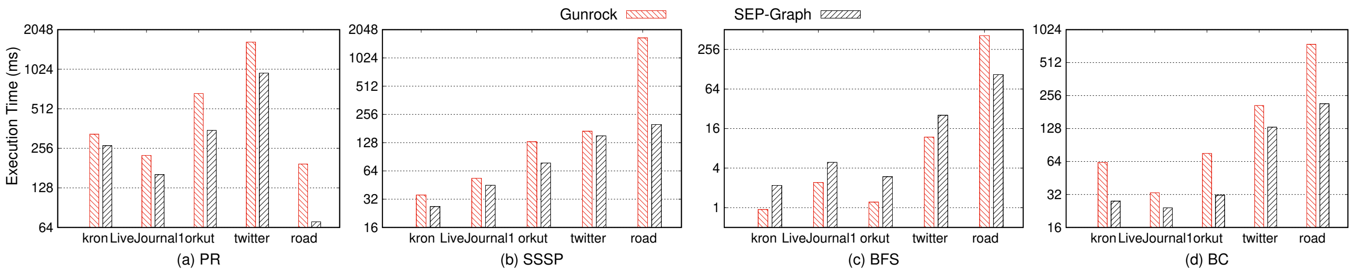

Figure 10. Performance comparisons of SEP-Graph with Groute and Gunrock on an NVIDIA Tesla V100 GPU.

technique on several graph algorithms. Nasre et al. [35] investigate the impact of vertex traversing mechanisms on graph algorithms on GPUs. Ben-Nun et al. [2] propose MAPS-Multi, an automatic multi-GPU partitioning framework for different memory access patterns in graph algorithms.

There are also graph-processing frameworks for single-GPU, multi-GPU, and GPU clusters. Zhong et al. [53] propose Medusa, a BSP-model-based graph analytics framework on GPU. Khorasani et al. [25] propose CuSha, a GAS model-based GPU graph-processing framework, focusing on resolving the load imbalance and GPU underutilization problems. Khorasani et al. [24] propose the warp segmentation mechanism to compact graph representations with the vertex refinement method to extend CuSha to multiple GPUs. Wang et al. [46] propose Gunrock, a synchronous graph-processing system on single-GPU. Pan et al. [39] extend Gunrock to multi-GPU with a set of optimizations, e.g., the kernel fusion and direction optimizing traversal. Ben-Nun et al. [3] propose Groute, an asynchronous graph-processing framework on multiple GPUs. Hong et al. [20] propose MultiGraph, which uses multiple graph representations and trades off the data movement and load balancing among GPU threads. Shi et al. [41] propose Frog, another asynchronous graph-processing framework on GPU, which uses graph coloring to determine the access sequence on neighbors of each vertex and alleviate the overhead of conflict in the push-base communication. Dathathri et al. [10] propose Gluon, a communication-optimizing substrate for graph analytics on GPU clusters. Gluon allows programmers to write a graph algorithm in a shared-memory system, and automatically enables it on GPU clusters.

Compared to these systems, SEP-Graph is a hybrid framework on GPU that can switch the execution path of graph algorithms with Sync or Async, Push or Pull, and DD or TD, to achieve the shortest execution time in each iteration. There exist several CPU-based systems that can switch the execution path of graph algorithms. Ligra [42] can switch between Push and Pull, based on the density of active vertices. PowerSwitch [50] can switch between Sync and Async, based on the vertex computation throughput. Besides targeting on a more complicated problem space (Sync or Async, Pull or Push, and DD or TD) on GPU, SEP-Graph adopts an inspection-execution method to determine when and how to switch, by monitoring a set of graph parameters. On GPU, our method is more efficient than those systems that use a single factor, i.e., the density of active vertices or the overall vertex throughput, to determine the switch.

7 Conclusion

We present SEP-Graph, a highly efficient GPU graph processing framework by adaptively switching execution path based on a selection in each of the three pairs of parameters, namely, Sync or Async, Push or Pull, and DD or TD. This approach is necessary to achieve the shortest execution time in each iteration, and consequently and significantly improve overall performance. Our intensive experiments show the high effectiveness of SEP-Graph.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and suggestions. This work has been partially supported by the National Science Foundation under grants CCF-1513944, CCF-1629403, and CCF-1718450, and by the National Natural Science Foundation of China under grants 61672141 and 61433008.

References

[1] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-optimizing Breadth-first Search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC ’12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 12, 10 pages.

[2] Tal Ben-Nun, Ely Levy, Amnon Barak, and Eri Rubin. 2015. Memory Access Patterns: The Missing Piece of the multi-GPU Puzzle. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’15)*. ACM, New York, NY, USA, Article 19, 12 pages.

[3] 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 (PPoPP ’17)*. ACM, New York, NY, USA, 235–248.

[4] Maciej Besta, MichałPodstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC ’17)*. ACM, New York, NY, USA, 93–104.

[5] Mauro Bisson, Everett Phillips, and Massimiliano Fatica. 2016. A cuda implementation of the pagerank pipeline benchmark. In *Proceedings of the 2016 IEEE High Performance Extreme Computing Conference (HPEC ’16)*. IEEE, IEEE Computer Society, Washington, DC, USA, 1–7.

[6] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology* 25, 2 (2001), 163–177.

[7] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *Proceedings of the Seventh International Conference on World Wide Web (WWW7)*. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 107–117.

[8] Shuai Che, Bradford M Beckmann, Steven K Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC ’13)*. IEEE, IEEE Computer Society, Washington, DC, USA, 185–195.

[9] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys ’15)*. ACM, New York, NY, USA, Article 1, 15 pages.

[10] 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 (PLDI ’18)*. ACM, New York, NY, USA, 752–768.

[11] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. 2014. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *Proceedings of the 2014 IEEE International Parallel and Distributed Processing Symposium (IPDPS ’14)*. IEEE Computer Society, Washington, DC, USA, 349–359.

[12] Tim Davis, Yifan Hu, and Scott Kolodziej. 2018. SuiteSparse Matrix Collection. Retrieved 08/20/2018 from https://sparse.tamu.edu/

[13] Groute Deveklopers. 2018. Groute: An Asynchronous Multi-GPU Programming Framework. Retrieved 08/20/2018 from https://github.com/groute/groute

[14] Gunrock Developers. 2018. Gunrock: GPU Graph Analytics. Retrieved 08/20/2018 from https://github.com/gunrock/gunrock

[15] Wenfei Fan, Ping Lu, Xiaojian Luo, Jingbo Xu, Qiang Yin, Wenyuan Yu, and Ruiqi Xu. 2018. Adaptive Asynchronous Parallelization of Graph Algorithms. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD ’18)*. ACM, New York, NY, USA, 1141–1156.

[16] 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 Conference on Operating Systems Design and Implementation (OSDI ’12)*. USENIX Association, Berkeley, CA, USA, 17–30.

[17] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI ’14)*. USENIX Association, Berkeley, CA, USA, 599–613.

[18] Samuel Grossman, Heiner Litz, and Christos Kozyrakis. 2018. Making Pull-based Graph Processing Performant. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’18)*. ACM, New York, NY, USA, 246–260.

[19] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1 (NIPS’13)*. Curran Associates Inc., USA, 1223–1231.

[20] Changwan Hong, Aravind Sukumaran-Rajam, 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 (PACT ’17)*. IEEE, IEEE Computer Society, Washington, DC, USA, 27–40.

[21] Kaixi Hou, Weifeng Liu, Hao Wang, and Wu-chun Feng. 2017. Fast Segmented Sort on GPUs. In *Proceedings of the International Conference on Supercomputing (ICS ’17)*. ACM, New York, NY, USA, Article 12, 10 pages.

[22] Rashid Kaleem, Anand Venkat, Sreepathi Pai, Mary Hall, and Keshav Pingali. 2016. Synchronization trade-offs in gpu implementations of graph algorithms. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS ’16)*. IEEE, IEEE Computer Society, Washington, DC, USA, 514–523.

[23] Onur Kayiran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. 2013. Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT ’13)*. IEEE Press, Piscataway, NJ, USA, 157–166.

[24] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. 2015. Scalable SIMD-Efficient Graph Processing on GPUs. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilations (PACT ’15)*. IEEE Computer Society, Washington, DC, USA, 39–50.

[25] 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 (HPDC ’14)*. ACM, New York, NY, USA, 239–252.

[26] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA-20)*. IEEE, IEEE Computer Society, Washington, DC, USA, 260–271.

[27] Ang Li, Weifeng Liu, Linnan Wang, Kevin Barker, and Shuaiwen Leon Song. 2018. Warp-Consolidation: A Novel Execution Model for GPUs. In *Proceedings of the 2018 International Conference on Supercomputing (ICS ’18)*. ACM, New York, NY, USA, 53–64.

[28] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. 2017. Locality-Aware CTA Clustering for Modern GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’17)*. ACM, New York, NY, USA, 297–311.

[29] Hang Liu and H. Howie Huang. 2015. Enterprise: Breadth-first Graph Traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 68, 12 pages.

[30] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment* 5, 8 (April 2012), 716–727.

[31] 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 (SIGMOD '10)*. ACM, New York, NY, USA, 135–146.

[32] Adam McLaughlin and David A. Bader. 2014. Scalable and High Performance Betweenness Centrality on the GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 572–583.

[33] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU Graph Traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 117–128.

[34] Ulrich Meyer and Peter Sanders. 2003. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms* 49, 1 (Oct. 2003), 114–152.

[35] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Data-Driven Versus Topology-driven Irregular Computations on GPUs. In *Proceedings of the 2013 IEEE International Symposium on Parallel and Distributed Processing (IPDPS '13)*. IEEE Computer Society, Washington, DC, USA, 463–474.

[36] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 456–471.

[37] Sreepathi Pai and Keshav Pingali. 2016. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '16)*. ACM, New York, NY, USA, 1–19.

[38] Yuechao Pan, Roger Pearce, and John D Owens. 2018. Scalable Breadth-First Search on a GPU Cluster. In *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS '18)*. IEEE, IEEE Computer Society, Washington, DC, USA, 1090–1101.

[39] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D Owens. 2017. Multi-GPU graph analytics. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS '17)*. IEEE, IEEE Computer Society, Washington, DC, USA, 479–490.

[40] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI '15)*. AAAI Press, Palo Alto, CA, USA, 4292–4293.

[41] Xuanhua Shi, Xuan Luo, Junling Liang, Peng Zhao, Sheng Di, Bingsheng He, and Hai Jin. 2018. Frog: Asynchronous graph processing on GPU with hybrid coloring model. *IEEE Transactions on Knowledge and Data Engineering* 30, 1 (Jan. 2018), 29–42.

[42] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 135–146.

[43] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. 2017. Chasing Away RAts: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 161–174.

[44] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111.

[45] Mohamed Wahib and Naoya Maruyama. 2014. Scalable Kernel Fusion for Memory-bound GPU Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 191–202.

[46] 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 (PPoPP '16)*. ACM, New York, NY, USA, Article 11, 12 pages.

[47] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Transactions on Parallel Computing* 4, 1, Article 3 (Aug. 2017), 49 pages.

[48] Joyce Jiyoung Whang, Andrew Lenharth, Inderjit S Dhillon, and Keshav Pingali. 2015. Scalable data-driven pagerank: Algorithms, system issues, and lessons learned. In *Proceedings of the 21st International European Conference on Parallel and Distributed Computing (Euro-Par '15)*. Springer, Berlin, Germany, 438–450.

[49] Yuduo Wu, Yangzihao Wang, Yuechao Pan, Carl Yang, and John D. Owens. 2015. Performance Characterization of High-Level Programming Models for GPU Graph Analytics. In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization (IISWC '15)*. IEEE Computer Society, Washington, DC, USA, 66–75.

[50] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. 2015. SYNC or ASYNC: Time to Fuse for Distributed Graph-parallel Computation. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '15)*. ACM, New York, NY, USA, 194–204.

[51] Jing Zhang, Ashwin M. Aji, Michael L. Chu, Hao Wang, and Wu-chun Feng. 2018. Taming Irregular Applications via Advanced Dynamic Parallelism on GPUs. In *Proceedings of the 15th ACM International Conference on Computing Frontiers (CF '18)*. ACM, New York, NY, USA, 146–154.

[52] 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 (Aug. 2014), 2091–2100.

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

[54] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-centric Distributed Graph Processing System. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 301–316.

A Artifact Appendix

A.1 Abstract

This artifact contains all executable of SEP-Graph on github. The shell scripts of running four graph algorithms and comparing to Gunrock and Groute are also included. The artifact will report the execution time of the graph algorithms evaluated in the PPoPP'19 paper, which is entitled "SEP-Graph: Finding Shortest Execution Paths for Graph Processing under a Hybrid Framework on GPU".

A.2 Artifact check-list (meta-information)

- **Algorithm:** PageRank, Breadth-First Search, Single-Source Shortest Path, Betweenness Centrality.
- **Program:** CUDA and C/C++ code
- **Compilation:** Use nvcc for CUDA kernels and use gcc and g++ for host code. The compilation flags, e.g., -O3, -std=c++11, etc., are set in CMakeLists.txt.
- **Binary:** One for each algorithm.
- **Data set:** Publicly available matrix market (.mtx) files.
- **Run-time environment:** The first server having NVIDIA GTX 1080 GPU is installed Ubuntu 18.04 with CMake 3.10.2, GCC 5.4.0, and CUDA 10.0. The second and third have NVIDIA Pascal P100 GPU and NVIDIA Volta V100 GPU, respectively. Each has CentOS Linux release 7.4.1708 with CMake 3.8.2, GCC 5.2.0, and CUDA 9.1.
- **Hardware:** CUDA-capable GPUs with compute capability of at least 3.5, e.g., NVIDIA GTX 1080, NVIDIA Pascal P100, and NVIDIA Volta V100.
- **Output:** Program execution time in millisecond, and execution path and execution time in each iteration of algorithms with SEP-Graph.
- **How much disk space required (approximately)?:** 28 GB (most for the datasets).
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 1.5 hour
- **Publicly available?:** Yes
- **Code/data licenses (if publicly available)?:** Apache License
- **Archived?:** Yes (The tarballs for the source code and artifact evaluation are placed at Zenodo https://zenodo.org/, with the DOIs 10.5281/zenodo.2008655 and 10.5281/zenodo.2008653).

A.3 Description

A.3.1 How delivered

SEP-Graph is an open source framework under Apache license version 2.0. It is hosted with the codes, build instructions, running scripts, and documentations at Github: https://github.com/sep-graph. The tarballs are also placed at Zenodo https://zenodo.org/, with the DOIs 10.5281/zenodo.2008655 and 10.5281/zenodo.2008653.

A.3.2 Hardware dependencies

SEP-Graph requires NVIDIA GPU with the compute capability of no less than 3.5.

A.3.3 Software dependencies

SEP-Graph has been tested on Ubuntu 18.04 and CentOS Linux release 7.4.1708, and is expected to run correctly under other Linux distributions. The tested CUDA versions include CUDA 9.1 and 10.0. The tested GCC versions include GCC 5.2.0 and 5.4.0. The scripts in the artifact use Python 2.7.

A.3.4 Data sets

All datasets are either publicly available, including *road_usa*, *soc-LiveJournal*, *soc-orkut*, and *soc-twitter*, or generated using standard graph generation software, e.g., *kron_g500-logn21*. Users can run a script provided in the artifact to download these datasets (Check A.5 Experiment workflow for more details).

A.4 Installation

To install, follow the instructions below:

- Clone the Git repository recursively from https://github.com/sep-graph/ppopp19-artifact.git

```
$ git clone --recursive \
  https://github.com/sep-graph/ppopp19-artifact.git
```

- Automatically build SEP-Graph by running the script "setup_sep.py" under "./bin"

```
$ cd bin
$ export PATH=${CUDA_HOME}/bin:${GCC_PATH}/bin:$PATH
$ export LD_LIBRARY_PATH=${CUDA_HOME}/lib64: \
           ${GCC_PATH}/lib64:$LD_LIBRARY_PATH
$ ./setup_sep.py
```

- (Optional) Manually build SEP-Graph with the command lines below:

```
$ git clone --recursive \
  https://github.com/sep-graph/ppopp19-artifact.git
$ cd sep-graph
$ mkdir build && cd build
$ cmake .. -DCMAKE_CXX_COMPILER=${G++_PATH} \
           -DCMAKE_C_COMPILER=${GCC_PATH} \
           -DCUDA_TOOLKIT_ROOT_DIR=${CUDA_HOME}
$ make -j 8
```

A.5 Experiment workflow

To run the experiments, follow the instructions below:

- Clone and build SEP-Graph as mentioned above. The SEP-Graph's executables are generated under "/path/to/ppopp19-artifact/sep-graph/build".

- Prepare the datasets.

```
$ cd /path/to/ppopp19-artifact/bin
$ ./download.py
```

  The "download.py" script will download datasets to "/path/to/ppopp19-artifact/dataset", including the .mtx and .gr files for each dataset. Users can also download .mtx files publicly and generate corresponding .gr files (Check A.7 Notes (1)).

- Run the script run_sep_${algo}.py for an algorithm:

```
$ cd /path/to/ppopp19-artifact/bin
$ ./run_sep_${algo}.py
$ ./run_all.py
```

  ${algo} can be pr, bfs, sssp, and bc. Each "run_sep_${algo}.py" script will run an algorithm with SEP-Graph and write its output to a log file under "/path/to/ppopp19-artifact/log". The script "run_all.py" calls "run_${arch}_${algo}.py" scripts to run all algorithms with three frameworks respectively, where ${arch} can be Gunrock, Groute, and SEP-Graph, and report corresponding execution time in millisecond with the CSV format, under "/path/to/ppopp19-artifact/output". These numbers are used in Figures 6-10 of the paper. Check A.7 Notes (3) for the setup of Gunrock and Groute.

- (Optional) Users can run the executables of SEP-Graph separately to check the detail of how SEP-Graph switches the execution paths of an algorithm at runtime:

```
$ cd /path/to/ppopp19-artifact/sep-graph/build
$ ./hybrid_${algo} -trace \
                  -graphfile ${path-to-dataset.gr}
```

  The executable will report the execution path and execution time in each iteration of an algorithm with SEP-Graph.

- (Optional) SEP-Graph has provided the validation functionality to check the correctness of algorithms. Users need to use "-check" as the command line parameter in the script "run_sep_${algo}.py" to enable it.

A.6 Evaluation and expected result

The scripts under "/path/to/ppopp19-artifact/bin" are expected to report the total execution time of a graph algorithm. The script "run_all.py" will also report the execution path and runtime in each iteration of an algorithm with SEP-Graph, by calling the executable under "/path/to/ppopp19-artifact/sep-graph/build".

A.7 Notes

(1) A .gr file is generated from a .mtx file with the tool "graph-convert" from the Galois project, which can be downloaded from https://github.com/IntelligentSoftwareSystems/Galois/tree/master/tools/graph-convert.

(2) The SEP-Graph binaries at "/path/to/ppopp19-artifact/sep-graph/build" needs the command line parameters for the tuning purpose, e.g., using "-wl_alloc_factor=0.4" for PageRank when running "hybrid_pr". The scripts under "/path/to/ppopp19-artifact/bin" have been configured properly.

(3) The script "setup_gunrock.py" and "setup_groute.py" under "/path/to/ppopp19-artifact/bin" can setup Grunrock and Groute. Alternatively, these two projects can be downloaded from github at https://github.com/gunrock/gunrock and https://github.com/groute/groute. The Boost library version 1.58.0 is used to run Gunrock in this paper.