

# Gunrock: A High-Performance Graph Processing Library on the GPU

Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens  
University of California, Davis  
{yzhwang, aaldavidson, ychpan, yudwu, atriffel, jowens}@ucdavis.edu

## ABSTRACT

For large-scale graph analytics on the GPU, the irregularity of data access and control flow and the complexity of programming GPUs have been two significant challenges for developing a programmable high-performance graph library. “Gunrock”, our graph-processing system, uses a high-level bulk-synchronous abstraction with traversal and computation steps, designed specifically for the GPU. Gunrock couples high performance with a high-level programming model that allows programmers to quickly develop new graph primitives with only a few hundred lines of code. We evaluate Gunrock on five key graph primitives and show that Gunrock has at least an order of magnitude speedup over Boost and PowerGraph, comparable performance to the fastest GPU hardwired primitives, and better performance than any other GPU high-level graph library.

## Keywords

Graph Processing, GPU, Runtime Framework

## 1. INTRODUCTION

Graphs are ubiquitous data structures that can represent relationships between people (social networks), computers (the Internet), biological and genetic interactions, and elements in unstructured meshes, just to name a few. In this paper, we describe “Gunrock”, our system for graph processing that delivers high performance in computing graph analytics with a high-level parallel programming model. At a high level, Gunrock targets graph primitives that are iterative, convergent processes with the highly-parallel, programmable graphics processing unit (GPU). We believe that the superior performance, price-performance, and power-performance capabilities of the modern GPU over the traditional CPU make it a strong candidate for data-intensive applications like graph processing; however, the irregularity of graph data structures leads to irregularity in data access and control flow, making an efficient implementation on GPUs a significant challenge.

Previous work in large graph analytics lies in one of four areas:

1. Single-node CPU-based systems, which are in common use for these problems today, but whose serial or coarse-grained-parallel programming models are poorly suited for a massively parallel processor like the GPU;
2. Distributed CPU-based systems, which offer scalability advantages over single-node systems but incur substantial communication cost, and whose programming models are also poorly suited to GPUs;
3. GPU “hardwired”, low-level implementations of specific graph primitives, which provide a proof of concept that GPU-based graph analytics can deliver best-in-class performance, but are difficult to develop even by the most skilled programmers, and do not generalize well to a variety of graph primitives; and
4. High-level GPU programming models for graph analytics, which often recapitulate CPU programming models (CuSha and MapGraph use PowerGraph’s GAS programming model, Medusa uses Pregel’s messaging model). Although they all try to apply optimizations to handle load imbalance and make use of fast GPU computing primitives, they still do not compare favorably in performance with hardwired primitives due to different kinds of overhead introduced by maintaining a high-level framework.

Our goal with Gunrock is to deliver the performance of GPU hardwired graph primitives with a high-level programming model that allows programmers to quickly develop new graph primitives. Gunrock achieves this goal by implementing optimization strategies that improve both load balancing and work efficiency and folding them into the core system where they benefit all graph primitives. The Gunrock abstraction centers on an active *frontier* of active vertices or edges in the graph, and supports two high-level bulk-synchronous-parallel (BSP) operations. *Traversal* computes a new frontier from the current frontier; *Computation* performs operations on the vertices or edges in the current frontier. Programmers can assemble complex and high-performance graph primitives from these two high-level operations without knowledge of their internals.

Our contributions are as follows:

1. We present an abstraction for doing graph traversal and computations that simplifies the programming for graph

processing algorithms on the GPU while delivering high performance.

2. We design and implement a set of simple and flexible APIs that significantly reduce the code size and the development time and apply to a wide range of graph processing primitives.
3. We describe several GPU-specific optimization strategies for memory efficiency, load balancing, and workload saving that together achieve high performance. All of our graph primitives achieve comparable performance to their hardwired counterparts.
4. We provide a detailed experimental evaluation of our graph primitives with performance comparisons to several CPU and GPU implementations. The five graph primitives implemented using Gunrock are breadth-first search (BFS), single-source shortest path (SSSP), betweenness centrality (BC), PageRank and connected components (CC).

Gunrock is currently available in an open-source repository at <http://gunrock.github.io/> and is currently available for use by external developers.

## 2. RELATED WORK

### 2.1 Specialized Parallel Graph Algorithms

Recent work has developed numerous best-of-breed, hard-wired implementations of many graph primitives. Merrill et al. [20] presented the first notable linear parallelization of the BFS algorithm on the GPU. They proposed an adaptive strategy for load-balancing parallel work by expanding one node’s neighbor list to one thread, one warp, or a whole block of threads. With this strategy and a memory-access efficient data representation, their implementation achieves high throughput on large scale-free graphs. Beamer et al.’s recent work on a very fast BFS for shared memory machines [1] uses a hybrid BFS which switches between the top-down and bottom-up neighbor list visiting algorithm according to the size of the frontier to save redundant edge visits. The current fastest connected-component algorithm on the GPU is Soman et al.’s work [28] based on two PRAM connected-component algorithms [14]. There are several parallel Betweenness Centrality implementations on the GPU [10, 23, 25] based on the work from Brandes and Ulrik [2]. Davidson et al. [5] proposed a work-efficient Single-Source Shortest Path algorithm on the GPU that explores a variety of parallel load-balanced graph traversal and work organization strategies to outperform other parallel methods. After we discuss the Gunrock abstraction in Section 4.1, we will discuss these existing hard-wired GPU graph algorithm implementations in Gunrock terminology.

### 2.2 Parallel Graph Libraries

Another goal of building parallel graph algorithms is having high-level, programmable, high-performance abstractions. The Boost Graph Library (BGL) is among the first efforts towards this goal, though its serial formulation and C++ focus together make it poorly suited for a massively parallel architecture like a GPU. Designed using the generic programming paradigm, the parallel BGL [13] separates the implementation of parallel algorithms from the underlying data structures

and communication mechanisms. However, the implementation is still specialized for each separate graph algorithm, missing a higher level of abstraction for common operators shared between different graph algorithms. Pregel [18] is Google’s effort at large-scale graph computing. It follows the Bulk Synchronous Parallel (BSP) model. A typical application in Pregel is an iterative convergent process consisting of global synchronization barriers called super-steps. The computation in Pregel is vertex-centric and based on message passing. Its programming model is good for scalability and fault tolerance. However, in standard graph algorithms in most Pregel-like graph processing systems, slow convergence arises from graphs with structure. GraphLab [17] allows asynchronous computation and dynamic asynchronous scheduling. By eliminating message-passing, its programming model isolates the user-defined algorithm from the movement of data, and therefore is more consistently expressive. PowerGraph [12] uses the more flexible Gather-Apply-Scatter (GAS) abstraction for power-law graphs. It supports both BSP and asynchronous execution. For the load imbalance problem, it uses vertex-cut to split high-degree vertices into equal degree-sized redundant vertices. This exposes greater parallelism in natural graphs. Ligra [26] is a graph processing framework for shared memory. It uses a similar operator abstraction for doing graph traversal. Its lightweight implementation is targeted at shared memory architectures and uses CilkPlus for its multithreading implementation. Galois [24, 22] is a graph system for shared memory based on a different operator abstraction. Their set iterators support priority scheduling and dynamic graphs and process on subsets of vertices called active elements. However, their set iterator does not abstract the internal details of the loop from the user. Users have to generate the active elements set directly for different graph algorithms. In Medusa [30], Zhong and He presented their pioneering work on GPU-based programming model for parallel graph processing using message passing model. CuSha [16] implements the parallel-sliding-window (PSW) graph representation on the GPU to avoid non-coalesced memory access. Both frameworks offer a small set of user-defined APIs but have severe load imbalance and thus fail to achieve the same level of performance as low-level GPU graph implementations. MapGraph [8] is another GPU graph library that adopts the GAS abstraction and represents the state-of-the-art for programmable single-node GPU graph processing. With our set of optimizations, Gunrock achieves better performance than all other GPU graph libraries. In Section 4.1, we will provide further details on how Gunrock’s traversal-compute abstraction maps to graph primitives in some of the above parallel graph libraries.

## 3. BACKGROUND & PRELIMINARIES

A graph is an ordered pair  $G = (V, E, w_e, w_v)$  comprised of a set of vertices  $V$  together with a set of edges  $E$ , where  $E \subseteq V \times V$ .  $w_e$  and  $w_v$  are two weight functions that show the weight values attached to edges and vertices in the graph. A graph is undirected if for all  $v, u \in V : (v, u) \in E \iff (u, v) \in E$ . Otherwise, it is directed. In graph processing, a vertex frontier represents a subset of vertices  $U \in V$  and an edge frontier represents a subset of edges  $I \in E$ .

Modern NVIDIA GPUs are throughput-oriented many-core processors that use massive parallelism to get very high peak computational throughput and hide memory latency. Kepler-based GPUs can have up to 15 vector processors,

termed streaming multiprocessors (SMX), each containing 32 parallel processing cores, termed streaming processors (SP). NVIDIA GPUs use the Single Instruction Multiple Thread (SIMT) programming model to achieve data parallelism. GPU programs called *kernels* run on a large number of parallel threads. Each set of 32 threads forms a divergent-free group called a *warp* to execute in lockstep in a Single Instruction Multiple Data (SIMD) fashion. These warps are then grouped into cooperative thread arrays called *blocks* whose threads can communicate through a pool of on-chip shared memory. All SMXs share an off-chip global DRAM.

For problems that require irregular data accesses such as graph problems, in addition to exposing enough parallelism, a successful GPU implementation needs to meet the following requirements: 1) coalesced memory access and effective use of the memory hierarchy, 2) minimizing thread divergence within a warp, and 3) reducing scattered reads and writes.

To achieve these goals, Gunrock represents per-node and per-edge data in structure-of-array (SOA) data structures that allow coalesced memory accesses with minimal memory divergence. The data structure for the graph itself is perhaps even more important. In Gunrock, we use a compressed sparse row (CSR) sparse matrix for vertex-centric operations and an edge list for edge-centric operations. CSR uses a column-indices array,  $C$ , to store a list of neighbor vertices and a row-offsets array,  $R$ , to store the offset of the neighbor list for each vertex. It provides compact and efficient memory access, and allows us to use scan, a common and efficient parallel primitive, to reorganize sparse and uneven workloads into dense and uniform ones in all phases of graph processing [20]. Our edge list uses two arrays to store the source and destination node for each edge in the graph. It provides free load balancing for edge-centric operations at the price of extra memory use [15].

## 4. FRAMEWORK

### 4.1 Abstraction

Gunrock targets graph operations that can be expressed as iterative convergent processes. By “iterative”, we mean operations that may require running a series of steps repeatedly; by “convergent”, we mean that these iterations allow us to approach the correct answer and terminate when that answer is reached. This goal is similar to most high-level graph frameworks.

Where Gunrock differs from other frameworks is in how and what we define as a step. Our goals here are to choose abstractions for steps that provide a high-level programming model to programmers while simultaneously permitting a high-performance implementation on GPUs. We aim to provide expressiveness and productivity to programmers while leveraging the unique strengths of GPU parallelism.

In our abstraction, each step operates on an active set of vertices or edges in the graph (the “frontier”). Steps are *bulk-synchronous parallel*: different steps may have dependencies between them, but individual operations within a step can be processed in parallel. For instance, a computation on each vertex within the frontier can be parallelized across vertices, and updating the frontier by identifying all the vertices neighboring the current frontier can also be parallelized across vertices. BSP operations are well-suited for efficient implementation on the GPU because they exhibit enough parallelism to keep the GPU busy and do not require

expensive fine-grained synchronization or locking operations.

We classify our BSP steps into two categories.

**Traversal** A *traversal* step generates a new frontier from the current frontier. The two most common ways to perform traversal in our programming model are (1) *advance*, which generates a new frontier by visiting the neighbors of the current frontier, and (2) *filter*, which chooses a subset of the current frontier based on programmer-specified criteria. According to the direction of the edges, advance can perform both push style traversal (scatter) and pull style traversal (gather). A frontier can consist of either vertices or edges, and can change datatypes over the entire computation; we can, for instance, generate a new frontier of neighboring edges from an existing frontier of vertices. Advance is generally an irregularly-parallel operation both because different vertices have different numbers of neighbors and because they share neighbors, so an efficient advance is the most significant challenge of a GPU implementation. When necessary, the traversal step uses atomic operations to guarantee each element appears only once in the frontier. No atomic operation is needed for graph primitives that allow idempotent operation; instead performing the same computation multiple times on the same element causes no harm to the final result. This may cause concurrent discovery of child nodes. A series of heuristics are used during the filter phase to remove redundant nodes to reduce the extra workload.

The generality of Gunrock’s efficient traversal operators allows us to use the same traversal implementation across a wide variety of interesting graph primitives. For instance, beyond simple advance and filter operations, Gunrock traversal operators can: 1) visit each element in the current frontier while updating local values and/or accumulating global values; 2) visit the vertex or edge neighbors of all the elements in the current frontier while updating source vertex, destination vertex, and/or edge values; 3) generate edge frontiers from vertex frontiers or vice versa; or 4) pull values from all vertices 2 hops away by starting from an edge frontier, visiting all the neighbor edges, and returning the far end vertices of these neighbor edges. As a result, we can concentrate our effort on implementing an efficient traversal and see that effort reflected in better performance on all graph primitives.

**Computation** A programmer-specified *computation* step defines an operation on all elements (vertices or edges) in the current frontier; Gunrock then performs that operation in parallel across all elements. Because this parallelism is regular, computation is straightforward to parallelize in a GPU implementation. Many simple graph primitives (e.g., computing the degree distribution of a graph) can be expressed as a single computation step.

Gunrock primitives are then assembled from a sequence of BSP steps. These steps are inherently sequential: one step completes all of its operations before the next step begins. Typically, Gunrock graph primitives run to convergence, which on Gunrock usually equates to an empty frontier; as individual elements in the current frontier reach convergence, they can be filtered out of the frontier. Programmers can also use other convergence criteria such as a maximum number of iterations or volatile flag values that can be set in a computation step.

#### 4.1.1 Alternative Abstractions

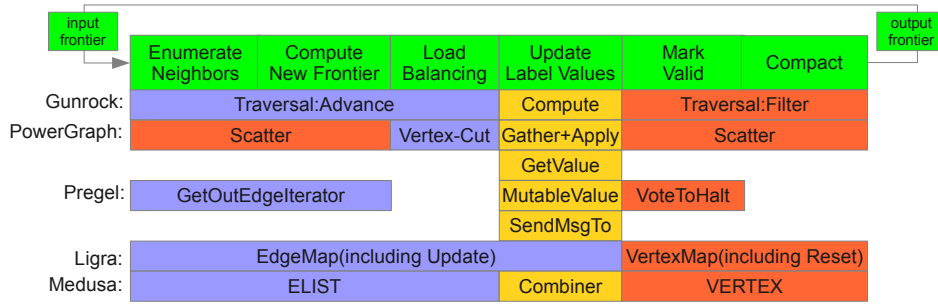


Figure 1: Operations that make up one iteration of SSSP and their mapping to the Gunrock, PowerGraph (GAS) [12], Pregel [18], Ligra [26], and Medusa [30] abstractions.

Previous work contains numerous interesting alternative abstractions that we did not integrate into Gunrock.

**Asynchronous execution** Many CPU frameworks (e.g., Galois and GraphLab) efficiently incorporate asynchronous execution, but the GPU’s expensive synchronization or locking operations would make this a poor choice for Gunrock. We do recover some of the benefits of prioritizing execution through our two-level priority queue (Section 4.4).

**Gather-apply-scatter (GAS) abstraction** GAS abstractions have successfully been mapped to the GPU, first with VertexAPI2 [7] and later with MapGraph [8] and CuSha [16]. GAS offers the twin benefits of simplicity and familiarity, given its popularity in the CPU world [12].

GAS operations can often map directly to Gunrock steps, but the reverse is not true; Gunrock can often combine multiple operations into one for efficiency. For instance:

- The combination of GAS’s gather and apply typically maps to Gunrock’s advance and computation steps. In a hardwired graph primitive, an expert GPU programmer would likely integrate gather and apply into a single kernel. Gunrock also implements that step in a single kernel, but a straightforward GAS implementation would instead split it into gather and apply kernels, and automatically combining them constitutes a significant compiler challenge.
- Gunrock can also merge a load-balancing operation followed by an apply operation by integrating the apply functor into the load-balancing operation. The most efficient existing GAS implementations of this sequence require a load-balance kernel followed by an apply kernel.

As well, we feel Gunrock’s decisions make the most sense as general building blocks for GPU implementation. Gunrock’s advance step encompasses both gather (a pull-based advance) and scatter (a push-based advance). We feel that Gunrock appropriately separates advance and filter, which we see as two distinct operations, with dedicated, high-performance implementations for each, and allowing both edge and vertex frontiers as input to these steps allows both higher performance and more generality.

**Message-passing in Medusa** The Medusa GPU graph-processing framework [30] also implements a BSP model and

allows computation on both edges and vertices. Medusa, unlike Gunrock, also allows edges and vertices to send *messages* to neighboring vertices. The Medusa authors note the complexity of managing the storage and buffering of these messages, and the difficulty of load-balancing when using segmented reduction for per-edge computation. Though they address both of these challenges in their work, the overhead of *any* management of messages is a significant contributor to runtime. Gunrock prefers the less costly direct communication between primitives and supports both push-based (scatter) communication and pull-based (gather) communication during traversal steps.

## 4.2 Example: Comparing Abstractions on Single-Source Shortest Path

SSSP is a reasonably complex graph primitive that computes the shortest path from a single node in a graph to every other node in the graph. We assume weights between nodes are all non-negative, which permits the use of Dijkstra’s algorithm and its parallel variants. Efficiently implementing SSSP continues to be an interesting problem in the GPU world [3, 5, 6]. Here we characterize one iteration of SSSP using the abstractions of four high-level parallel graph frameworks: Gunrock, Pregel, PowerGraph, and Ligra (Figure 1). The iteration starts with an input frontier of active vertices. First, SSSP enumerates the sizes of the frontier’s neighbor list of edges and computes the length of the output frontier. Because the neighbor edges are unequally distributed among the frontier’s vertices, SSSP next redistributes the workload across parallel threads. Next, each edge adds its weight to the distance value at its source value and, if appropriate, updates the distance value of its destination vertex. Finally, SSSP removes redundant vertex IDs, decides which updated vertices are valid in the new frontier, and computes the new frontier for the next iteration. Algorithm 1 provides more detail.

Gunrock maps one SSSP iteration onto three Gunrock steps: (1) *advance*, which computes the list of edges connected to the current vertex frontier and (transparently) load-balances their execution; (2) *compute*, to update neighboring vertices with new distances; and (3) *filter*, to generate the final output frontier. This mapping of traversal and computation is simple and intuitive, allowing the program to fully utilize the GPU’s computing resources in a load-balanced way.

Pregel, as a vertex-centric programming model, only provides data parallelism on vertices. For graphs with significant variance in vertex degree (e.g., power-law graphs), this would

---

**Algorithm 1** Single-Source Shortest Path

---

```
1: procedure SET_PROBLEM_DATA( $G, P, root$ )
2:    $P.labels[1..G.verts] \leftarrow \infty$ 
3:    $P.preds[1..G.verts] \leftarrow -1$ 
4:    $P.labels[root] \leftarrow 0$ 
5:    $P.preds[root] \leftarrow src$ 
6:    $P.frontier.Insert(root)$ 
7: end procedure
8:
9: procedure UPDATE_LABEL( $s\_id, d\_id, e\_id, P$ )
10:   $new\_label \leftarrow P.labels[s\_id] + P.weights[e\_id]$ 
11:  return  $new\_label$  <
12:   $atomicMin(P.labels[d\_id], new\_label)$ 
13: end procedure
14: procedure SET_PRED( $s\_id, d\_id, P$ )
15:   $P.preds[d\_id] \leftarrow s\_id$ 
16:   $P.output\_queue\_ids[d\_id] \leftarrow output\_queue\_id$ 
17: end procedure
18:
19: procedure REMOVE_REDUNDANT( $node\_id, P$ )
20:  return  $P.output\_queue\_id[node\_id] == output\_queue\_id$ 
21: end procedure
22: procedure SSSP_ENACTOR( $G, P, root$ )
23:  SET_PROBLEM_DATA( $G, P, root$ )
24:  while  $P.frontier.Size() > 0$  do
25:    ADVANCE( $G, P, UpdateLabel, SetPred$ )
26:    FILTER( $G, P, RemoveRedundant$ )
27:    PRIORITY_QUEUE( $G, P$ )
28:  end while
29: end procedure
```

---

cause severe load imbalance on GPUs. A Pregel implementation would first run GetOutEdgeIterator to start the traversal process, then GetValue, MutableValue, and SendMsgTo to process computations and send out updated values. The traversal operator in Pregel is general enough to apply to a wide range of graph primitives, but its vertex-centric design only achieves good parallelism when nodes in the graph have small and evenly-distributed neighborhoods. For real-world graphs that often have uneven distribution of node degrees, Pregel suffers from severe load imbalance.

PowerGraph’s vertex-cut operation directly addresses Pregel’s vertex load-imbalance problem: PowerGraph splits large neighbor lists, duplicates node information, and deploys each partial neighbor list to different machines. Working as a load balancing strategy, vertex-cut replaces the large synchronization cost in edge-cut into a single-node synchronization cost. This is a productive strategy for multi-node implementations, but Gunrock can use efficient single-GPU computing primitives such as scan and sorted search to achieve the same load balance without any extra cost of duplicate storage and synchronization.

Unlike the above frameworks, Ligra targets a SSSP formulation that allows negative weights; Ligra implements the Bellman-Ford algorithm with an edge phase (propagating edge weights to vertices and recording if those vertices have been updated) followed by a vertex phase (to reset the visited list of vertices). Ligra’s load-balancing strategy is based on CilkPlus, a fine-grained task-parallel library for CPUs. Despite promising GPU research efforts on task parallelism [4, 29], no such equivalent is available on GPUs, thus we implement our own load-balancing strategies within Gunrock.

### 4.3 System Design and Implementation

Gunrock’s software architecture is divided into two parts. Above the traversal-compute abstraction is the application module. This is where users define different graph algorithms using the high-level APIs provided by Gunrock (Figure 2). Under the abstraction are the utility functions and the implementation of operators used in traversal, including optimization strategies for efficient traversal and workload reorganization.

The application module contains three components: Problem, which provides graph topology data and an algorithm specific data management interface; Functor, which contains user-defined computation code; and Enactor, which serves as the entry point of the graph algorithm and defines the running process by combining the traversal and the computation. Gunrock offers recommended kernel launching settings for existing primitives; for new primitives, we provide interfaces for users to choose their own settings.

Gunrock provides two operators for traversal: advance and filter. The implementation of these two operators integrates several optimization strategies and exposes an interface for sending functors into an operator at compile time.

A typical graph primitive in Gunrock starts with initializing and loading data to the GPU. When running the enactor, operators will traverse the graph, handle the irregular workload by load balancing, assign the workload to threads, then load the functors that are integrated during compilation time to perform computations in parallel for each element in the frontier. This process continues until the frontier is empty or another termination criteria is met. Our runtime model mixes traversal and computation to achieve high performance while keeping the implementation logic simple.

## 4.4 Optimizations

Choosing the right abstraction is one key component in achieving high performance within a graph framework. The second component is optimized implementations of the primitives within the framework. We have implemented both generalized optimizations that apply to all graph operations (Sections 4.4.1 and 4.4.2) as well as optimizations that primarily target one or a small number of operations (Section 4.4.3). The numerous optimizations we incorporate are either new to the GPU (e.g., direction-optimized traversal in Section 4.4.2) or were previously specific to one graph algorithm/implementation (e.g., workload mapping strategies in Section 4.4.1).

### 4.4.1 Workload Mapping for Advance

Gunrock’s advance step generates an irregular workload. Consider an advance that generates a new vertex frontier from the neighbors of all vertices in the current frontier. If we parallelize over input vertices, graphs with a variation in vertex degree (with different-sized neighbor lists) will generate a corresponding imbalance in per-vertex work. Thus, mapping the workload of each vertex onto the GPU so that they can be processed in a load-balanced way is essential for efficiency.

The most significant previous work in this area balances load by cooperating between threads. Merrill et al. [20] map the workload of a single vertex to a thread, a warp, or a cooperative thread array (CTA) according to the size of its neighbor list. Davidson et al. [5] use two load-balanced workload mapping strategies, one that groups input work and the other that groups output work. The first partitions

```

static __device__ __forceinline__ bool
CondEdge(VertexId s_id, VertexId d_id, DataSlice *problem,
          VertexId e_id = 0, VertexId e_id_in = 0)

static __device__ __forceinline__ void
ApplyEdge(VertexId s_id, VertexId d_id, DataSlice *problem,
          VertexId e_id = 0, VertexId e_id_in = 0)

static __device__ __forceinline__ bool
CondVertex(VertexId node, DataSlice *p)

static __device__ __forceinline__ void
ApplyVertex(VertexId node, DataSlice *p)

```

```

gunrock::oprtr::advance::Kernel
<AdvancePolicy, Problem, Functor>
<<<advance_grid_size, AdvancePolicy::THREADS>>>(>
    queue_length,
    graph_slice->ping_pong_working_queue[selector],
    graph_slice->ping_pong_working_queue[selector^1],
    data_slice,
    context,
    gunrock::oprtr::ADVANCETYPE)

gunrock::oprtr::filter::Kernel
<FilterPolicy, Problem, Functor>
<<<filter_grid_size, FilterPolicy::THREADS>>>(>
    queue_length,
    graph_slice->ping_pong_working_queue[selector],
    graph_slice->ping_pong_working_queue[selector^1],
    data_slice)

```

Figure 2: Gunrock’s API set.

the frontier into equally sized chunks and assigns all neighbor lists of one chunk to one block; the second partitions the neighbor list set into equally sized chunks (possibly splitting the neighbor list of one node into multiple chunks) and assigns each chunk of edge lists to one block of threads. We extend these approaches in two ways. Both Merrill et al. and Davidson et al. support frontiers of vertices; Gunrock adds the capability of edge frontiers as well as pull-based traversal (Section 4.4.2). The result is two load-balancing strategies within Gunrock.

**Per-thread fine-grained** Our first strategy maps one frontier vertex’s neighbor list to one thread. It is the easier method to implement. Each thread loads the neighbor list offset for its assigned node, then serially processes edges in its neighbor list. As suggested by Merrill et al. and Davidson et al., we can improve this method in several ways. First, we can load all the neighbor list offsets into shared memory, then use a CTA of threads to cooperatively strip edges off the neighbor list. Simultaneously, we use vertex-cut to split the neighbor list of a node so that it can be processed by multiple threads. Together these improvements balance thread work within a CTA, but not across CTAs. In Gunrock, we have integrated this strategy into one kernel and we found out that this method performs better when used for large-diameter graphs with a relatively even degree distribution. For scale-free social graphs with a more unevenly degree distribution, we turn to a second strategy.

**Per-warp and per-CTA coarse-grained** Significant differences in neighbor list size cause the worst performance with our per-thread fine-grained strategy. We directly address the variation in size by grouping neighbor lists into three categories based on their size, then individually processing each category with a strategy targeted directly at that size. Our three sizes are (1) lists larger than a CTA; (2) lists larger than a warp (32 threads) but smaller than a CTA; and (3) lists smaller than a warp. We begin by assigning a subset of the frontier to a block. Within that block, each thread owns one node. The threads that own nodes with large lists arbitrate for control of the entire block (by writing to a shared memory location and choosing the last one to write as the winner). All the threads in the block then cooperatively process the neighbor list of the winner’s node. This procedure continues until all nodes with large lists have been processed. Next, all threads in each warp begin a similar procedure to process all the nodes whose neighbor lists are medium-sized lists. Finally, the remaining nodes are processed using our per-thread fine-grained workload-mapping strategy (Figure 3).

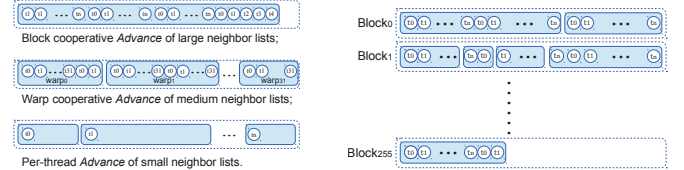


Figure 3: Merrill et al. [20]

Figure 4: Davidson et al. [5]

The specialization of this method allows higher throughput on frontiers with a high variance in degree distribution, but at the cost of higher overhead due to the sequential processing of the three different sizes. Davidson et al., and Gunrock, improve on this method by first organizing groups of edges into equal-length chunks and assigning each chunk to a block. This division requires us to find the starting and ending indices for all the blocks within the frontier. We use an efficient sorted search to map such indices with the scanned edge offset queue. When we start to process neighbor list of a new node, we use binary search to find the node ID for the edges that are going to be processed. Using this method, we ensure load-balance both within a block and between blocks (Figure 4).

We note that our coarse-grained (load-balancing) traversal method works better on social graphs with irregular distributed degrees, while the fine-grained method works better on graphs where most nodes have small degrees. For this reason, in Gunrock we implement a hybrid of both methods on both vertex and edge frontiers, using the per-thread fine-grained strategy for nodes with relatively smaller neighbor lists and the per-CTA coarse-grained strategy for nodes with relatively larger neighbor lists. Gunrock sets a runtime threshold value for the neighbor count of the current frontier that selects between our two strategies. For Gunrock-supplied graph primitives, we set this value to 4096 because it gives the best overall performance on all datasets we tested. Users can also change this value easily in the Enactor module for their own datasets or graph primitives. We have found this hybrid gives consistently high performance with both balanced and unbalanced vertex degree distributions.

#### 4.4.2 Improving Work Efficiency

Work-efficiency optimizations try to save redundant work during the graph processing. Such optimizations may incur an additional memory cost and reduction in parallelism, but still achieve a significant performance increase.

**Push-based and pull-based operations** When Gunrock’s advance operator generates a new frontier of vertices, those



vertices fall into three categories: (1) vertices that have never been visited before; (2) vertices that were previously visited on this iteration; and (3) vertices that were visited on a previous iteration. For graph algorithms like BFS that only want to visit each vertex once, the vertices in the second and third category represent wasted work, and if the graph algorithm requires distinguishing between category 1 and 2, implementations generally must use expensive atomic operations. How can we reduce the cost of this overhead?

The natural way to describe many graph traversals is in terms of *push*: the current frontier of active vertices “pushes” active status to its neighbors. Beamer et al. [1] noted that as a traversal expands along a graph, push makes sense: one step in the traversal will mostly visit new nodes (category 1). But near the end of the traversal, when most nodes have already been visited, push is inefficient: one step in the traversal will primarily visit already-visited nodes (categories 2 and 3). In this scenario, it makes much more sense to *pull* instead of push: begin each step with all *unvisited* nodes and add them to the new frontier if one of their neighbors is in the current frontier. Instead of visiting many category-2 or -3 nodes, we visit none; we have no cost for atomics; instead our only wasted work is unvisited vertices that are disconnected from the current frontier.

Implementing pull-based traversal requires visiting each predecessor of the unvisited vertices, thus we need to store the reverse graph in memory. Gunrock’s pull implementation also needs to store the current frontier in a bitmap. Unlike the work of Beamer et al. and Ligra, which both loop over all vertices, Gunrock prefers a cheap filter step to put all the unvisited nodes into a frontier, and pulls the computation from only these nodes’ predecessors if they are valid in the frontier bitmap. This brings us better resource utilization on the GPU. Once we have the reverse graph and the frontier containing the unvisited nodes, the rest of the traversal can take advantage of our work mapping optimizations according to the frontier size. Pull is a better strategy when the frontier for a push-based advance is large enough so that the number of edges to check from the frontier itself ( $m_f$ ) is larger than the number of edges to check from the unvisited nodes ( $m_u$ ). In practice we use a tuning parameter  $\alpha$  since  $m_u$  is an overly pessimistic upper-bound on the number of edges the pull-based advance will check. We use a similar tuning parameter  $\beta$  to switch back to push-based advance when the frontier is smaller than  $\frac{n}{\beta}$ , where  $n$  is the number of nodes. This optimization is a specific optimization and can only be applied to graph algorithms that do not require visiting all the edges, since it inherently avoids visiting edges that connect visited nodes. In BFS, this optimization brings a 1.6x speedup on two scale-free graph datasets. Finding the optimal  $\alpha$  and  $\beta$  for different graph algorithms and graph topologies is difficult since both can generate very different frontier sizes over iterations. We found out that switching between push-based and pull-based traversal works better on scale-free graphs (the speedup has a geometric mean of 1.52), whereas on the small-degree large-diameter graph, we see less concurrent discovery and the performance benefits are not as significant (the speedup has a geometric mean of 1.28). Currently Gunrock provides an interface for setting these parameters on the command line; we have two sets of parameters for small-degree large-diameter graphs and scale-free graphs separately; we plan to explore automatic

parameter determination in our future research.

**Priority Queue** A straightforward BSP implementation of an operation on a frontier will treat each element in the frontier equally, i.e., with the same priority. Many graph primitives benefit from prioritizing certain elements for computation with the expectation that computing those elements first will save work overall (e.g., delta-stepping for SSSP [21]). Gunrock generalizes the approach of Davidson et al. [5] by allowing user-defined priority functions to organize an output frontier into “near” and “far” slices. This allows the GPU to use a simple and high-performance split operation to create and maintain the two slices. Gunrock then considers only the near slice in the next processing steps, adding any new elements that do not pass the near criterion into the far slice, until the near slice is exhausted. We then update the priority function and operate on the far slice.

Currently Gunrock uses this specific optimization only in SSSP, but with future work, we believe a workload reorganization strategy based on a more general priority queue implementation will enable a semi-asynchronous execution model in Gunrock. This will potentially increase the performance of various types of community-detection and label propagation algorithms and also algorithms on graphs with small “long tail” frontiers.

#### 4.4.3 Primitive-Specific Optimizations

Gunrock’s traversal operators are flexible enough to allow users to implement several graph-primitive-specific optimizations in user code without touching Gunrock’s library code base. For some optimizations that come with the existing Gunrock-supplied graph primitives, users can set command-line parameters to turn them on or off.

**Idempotent Operation** BFS is an idempotent process—visiting the same node multiple times is acceptable and benign. Thus we can allow concurrent discovery of nodes and do not require the advance operator’s atomic operations that serialize node discovery. However, concurrent discovery can potentially result in duplicate nodes in the frontier, which wastes work. We adapt Merrill et al.’s heuristics [20] in Gunrock’s filter phase to mitigate (but not eliminate) duplicate nodes. In general, allowing idempotent traversal avoids expensive atomic operations at the price of a few redundant nodes missed by the heuristics and results in a 2.7x speedup on average over all datasets in our test set. This optimization strategy can also be applied to any graph primitive that allows idempotent operation.

**Optimizing Filter** In Gunrock, we use filtering in two ways. If we only want computation on a subset of the frontier, we use a functor to guarantee that only elements that meet the user-defined criterion are preserved by using a scan and compact pass to generate the new frontier. However, when we know no elements will be removed, we automatically convert the filter operator into a simple parallel apply operator, eliminating the scan-and-compact operation. This optimization is currently used in Gunrock’s connected component primitive and are generally applicable to apply-to-all-elements-in-frontier type operations, including several node-ranking and link-analysis algorithms.

**Output Frontier Storage** Our betweenness-centrality primitive (detailed in Section 5.1.3) has a forward BFS phase

and a backward phase. The backward phase visits nodes generated by the forward BFS phase, but in reverse order. The simplest way to implement this is to use a filter operator to get nodes with specific BFS label values in the backward phase. One optimization is to add an extra array sized to hold all vertices that stores the output frontiers for every iteration in the forward BFS phase and their offsets. With this array and the offsets, we can directly send nodes with a specific BFS label value to the advance operator in the backward phase, thus avoiding the use of a filter operator. This optimization requires no changes in Gunrock’s library; instead, users only need to add a few lines of code in the enactor and functor modules.

## 5. APPLICATIONS

One of the principal advantages of Gunrock’s traversal-compute abstraction is that by reusing Gunrock’s efficient operators and combining different functors, we can build new graph primitives with minimal extra work. In this section we show that our operators not only apply to BFS-based primitives that traverse all the nodes/edges in the graph, but also other primitives that traverse the graph in a more flexible way.

### 5.1 Graph Primitives

For each primitive, we describe the hardwired GPU implementation to which we compare, followed by how we express this primitive in Gunrock. Section 6 compares the performance between hardwired and Gunrock implementations.

#### 5.1.1 Breadth-First Search (BFS)

BFS initializes a vertex frontier with a single source vertex. On each iteration, it adds all unvisited neighbor vertices of the vertices in the current frontier to the new frontier, setting their depth and repeating until all vertices are visited. BFS is one of the most fundamental graph primitives and serves as the basis of several other graph primitives.

**Hardwired GPU Implementation** The well-known BFS implementation of Merrill et al. [20] achieves its high performance through careful load-balancing, avoidance of atomics, and heuristics for avoiding redundant vertex discovery. Its chief operations are expand (to generate a new frontier) and contract (to remove redundant vertices) phases.

**Gunrock Implementation** Merrill et al.’s expand maps nicely to Gunrock’s advance operator, and contract to Gunrock’s filter operator. During advance, we set a label value for each vertex to show the distance from the source, and/or set a predecessor value for each vertex that shows the predecessor vertex’s ID. We implement efficient load-balancing (Section 4.4.1) and both push- and pull-based traversal (Section 4.4.2) for more efficient traversal. Our base implementation uses AtomicCAS in the Cond functor to prevent concurrent vertex discovery. When a vertex is uniquely discovered, we set its label and/or predecessor id in the apply functor. We can avoid the use of atomics and improve performance by enabling idempotent operation with heuristics that reduce the concurrent discovery of child nodes (Section 4.4.3).

With these optimizations, we are able to increase the performance of BFS even more in Gunrock while significantly reducing the code size users have to write. Currently on graphs with large diameter, Gunrock’s performance is 50%

of Merrill et al.’s BFS implementation. One reason is that in their BFS implementation, the highest performing strategy uses kernel fusion to merge their expand and contract kernels into one kernel, which significantly reduces data movement within one iteration. Gunrock’s implementation does not include such optimization now, but we intend to implement a general kernel fusion strategy to integrate into our current traversal-compute abstraction.

#### 5.1.2 Single-Source Shortest Path

Single-source shortest path finds paths between a given source vertex and all other vertices in the graph such that the weights on the path between source and destination vertices are minimized. While the traversal mode of SSSP is identical to BFS, the computation mode differs.

**Hardwired GPU Implementation** Currently the highest performing SSSP algorithm implementation on the GPU is the work from Davidson et al. [5]. They provide two key optimizations in their SSSP implementation: 1) a load balanced graph traversal method and 2) a priority queue implementation that reorganizes the workload. Gunrock integrates both optimization strategies into its abstraction. It generalizes the graph traversal method to other graph primitives, bringing a performance boost to all graph primitives that use the advance operator. We implement Gunrock’s priority queue as an additional filter pass between two iterations.

**Gunrock Implementation** To compute a distance value from the source vertex, we need one advance and one filter operator. We start from a single source vertex in the frontier. On each iteration, we visit all associated edges in parallel for each vertex in the frontier and relax the distance’s value (if necessary) of the vertices attached to those edges. We use AtomicMin in the Cond functor to atomically find the minimal distance value we want to keep and a bitmap flag array to remove redundant vertices. After each iteration, we use a priority queue to reorganize the vertices in the frontier.

#### 5.1.3 Betweenness Centrality

The BC index can be used in social network analysis as an indicator of the relative importance of vertices in a graph. At a high level, the BC for a vertex in a graph is the fraction of shortest paths in a graph that pass through that vertex. Brandes describes a BC formulation [2] that is most commonly used for GPU implementation.

**Hardwired GPU Implementation** Brandes’s formulation has two passes: a forward BFS pass to accumulate sigma values for each node, and a backward BFS pass to compute centrality values. Jia et al. [15] and Sariyüce et al. [25] both use an edge-parallel method to implement the above two passes. We achieve this in Gunrock using two advance operators on an edge frontier with different computations.

**Gunrock Implementation** Gunrock’s implementation also contains two phases. The first phase is a BFS, identical to the original BFS except for a new apply functor that computes the number of shortest paths from source to each vertex. The second phase iterates over the BFS frontier backwards. We can generate the right frontier queue for each iteration in the backward phase using a filter operator or it can be saved directly during the forward BFS phase with extra memory



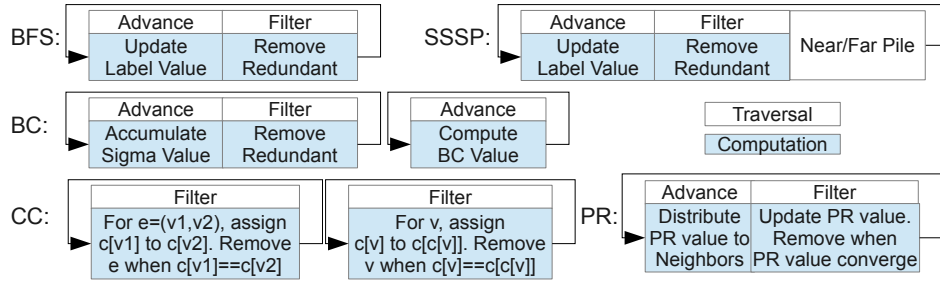


Figure 5: Operation flow chart for selected primitives in Gunrock (a black line with an arrow at one end indicates a while loop that runs until the frontier is empty).

cost. We then use an advance operator with an apply functor to compute the dependency scores.

#### 5.1.4 Connected Component Labeling

The connected component problem labels the vertices in each connected component in a graph with a unique component ID.

**Hardwired GPU Implementation** Soman et al. [28] base their implementation on two PRAM algorithms: hooking and pointer-jumping. Hooking takes an edge as the input and tries to set the component IDs of the two end vertices of that edge to the same. In the odd numbered iteration, the lower vertex writes its value to the higher vertex, and vice versa in the even numbered iteration. This strategy could increase the rate of convergence. Pointer-jumping reduces a multi-level tree in the graph to a one-level tree (star). By repeating these two operators until no component ID changes for all the nodes in the graph, the algorithm will compute the number of connected components for the graph and the connected component to which each node belongs.

**Gunrock Implementation** Gunrock uses a filter operator on edges to implement hooking. During each iteration, one end vertex of each edge in the frontier tries to assign its component ID to the other vertex. In our implementation, the priority of assignment can be switched by using two different functors at different iterations. Hooking repeats until no vertex’s component ID changes. For pointer-jumping, a filter operator on vertices traces the component ID of each vertex to its parent until it reaches the root. Gunrock achieves comparable performance to Soman et al.’s implementation.

#### 5.1.5 PageRank and Other Node Ranking Algorithms

The PageRank link analysis algorithm assigns a numerical weighting to each element of a hyperlinked set of documents, such as the World Wide Web, with the purpose of quantifying its relative importance within the set. The iterative method of computing PageRank gives each vertex an initial PageRank value and updates it based on the PageRank of its neighbors, until the PageRank value for each vertex converges. In Gunrock, we use one advance operator to compute the PageRank value during each iteration, and one filter operator to filter out the vertices whose PageRanks have already converged. For the accumulation of PageRank value, we use AtomicAdd in the functor. We start from a frontier that contains all vertices in the graph and end when all vertices have converged.

Dataset	Vertices	Edges	Max Degree	Diameter
soc-LiveJournal1	4.8M	68.9M	20333	16
bitcoin	6.3M	28M	565991	1041
kron_g500-logn20	1M	44.6M	131503	6
roadNet-CA	2M	5.5M	12	849

Table 1: Dataset Description Table.

**Bipartite graphs** Geil et al. [9] used Gunrock to implement Twitter’s who-to-follow algorithm (“Money” [11]), which incorporated three node-ranking algorithms based on bipartite graphs (Personalized PageRank, Stochastic Approach for Link-Structure Analysis (SALSA), and Hyperlink-Induced Topic Search (HITS)). Their implementation, which the authors cite as the first GPU implementation of bipartite graph operations, demonstrated that Gunrock’s advance operator is flexible enough to encompass all three node-ranking algorithms, including a 2-hop traversal in a bipartite graph.

## 6. EXPERIMENTS & RESULTS

We ran all experiments in this paper on a Linux workstation with 2×3.50 GHz Intel 4-core E5-2637 v2 Xeon CPUs, 528 GB of main memory, and an NVIDIA K40c GPU with 12 GB on-board memory. The GPU programs were compiled with NVIDIA’s nvcc compiler (version 6.0.1) with the -O3 flag. The BGL and PowerGraph code were compiled using gcc 4.6.3 with the -O3 flag. Ligra was compiled using icpc 15.0.1 with CilkPlus. All results ignore transfer time (both disk-to-memory and CPU-to-GPU). All tests were run 10 times with the average runtime used for results.

The datasets used in our experiments are shown in Table 1. We converted all datasets to undirected graphs. The graph topology of these four datasets spans from small-degree large-diameter to scale-free. The soc-LiveJournal1 (soc) and kron\_g500-logn20 (kron) datasets are two scale-free graphs with diameters of less than 20 and unevenly distributed node degrees (90% of nodes have degree less than 128). The roadNet-CA (roadnet) dataset has a large diameter (over 500) with small and evenly distributed node degrees (all nodes have degree less than 12, and 59% of nodes have degree between 2 and 4). The bitcoin dataset has a unique topology: one of its nodes has degree of over 0.5 million, while 94% of the other nodes have degree less than 4. Bitcoin’s diameter is also very large (over 1000).

**Performance Summary** Table 2 compares Gunrock’s performance against several other graph libraries and hardwired GPU implementations. In general, Gunrock’s performance

		Runtime (ms) [lower is better]							Edge throughput (MTEPS) [higher is better]						
Alg.	Dataset	Hardwired							Hardwired						
		BGL	PG	Medusa	MapGraph	GPU	Ligra	Gunrock	BGL	PG	Medusa	MapGraph	GPU	Ligra	Gunrock
BFS	soc	816	—	75.82	84.08	37.87	57.4	24.37	169.1	—	1820	1641	3643	2404	5662
	bitc	480	—	1557	142.4	69.22	94.9	67.79	117.3	—	36.16	395.4	813.4	593.3	830.6
	kron	388	—	46.21	44.29	18.67	13.3	17.28	230.1	—	1932	2016	4782	6713	5167
	roadnet	72	—	223.9	53.44	8.18	51.5	17.16	76.7	—	24.66	103.3	675.1	107.2	321.8
SSSP	soc	5664	1900	—	225.7	236.7	172	361.6	15.8	47.1	—	396.2	377.8	498.3	237
	bitc	2440	1610	7311	250.9	183.6	133	178.8	12.67	19.21	4.231	123.3	168.4	232.6	173
	kron	1268	1000	—	124.8	125.1	16.4	105.2	70.38	89.24	—	714.8	713.1	7531	1174
	roadnet	408	5800	1143	76.48	163.7	62.2	140	13.53	0.952	4.757	72.18	33.73	88.75	39.43
BC	soc	2120	—	—	—	543.8	264	205.3	65.09	—	—	—	253.7	522.7	672.1
	bitc	4840	—	—	—	190.2	271	206.6	11.63	—	—	—	295.9	207.7	272.5
	kron	1456	—	—	—	156.1	52.6	246.9	61.29	—	—	—	571.7	1696	361.4
	roadnet	732	—	—	—	256.3	129	100.1	7.559	—	—	—	21.59	42.89	55.27
PageRank	soc	49568	9500	—	5431	—	<b>265</b>	1927 · <b>175</b>							
	bitc	20400	8600	48156	2471	—	<b>240</b>	651.4 · <b>79.6</b>							
	kron	33432	2500	—	5702	—	<b>114</b>	2766 · <b>212</b>							
	roadnet	2440	2600	532.8	122.7	—	<b>13.1</b>	63.25 · <b>4</b>							
CC	soc	2176	12802	—	803.8	72	498	110							
	bitc	1508	8464	—	612.5	28	6180	58.33							
	kron	716	5375	—	260	48	1890	67.21							
	roadnet	232	9995	—	1935	8	1320	21.33							

Table 2: Gunrock’s performance comparison (runtime and edge throughput) with other graph libraries (Boost Graph Library, PowerGraph, Medusa, MapGraph, Ligra) and hardwired GPU implementations. Ligra runs PageRank for only one iteration; Ligra’s timings for PageRank and Gunrock’s one-iteration PageRank are in **bold**. Hardwired GPU implementations for each primitive are b40c (BFS) [20], deltaStep (SSSP) [5], gpu\_BC (BC) [25], and conn (CC) [28].

Dataset	Runtime (ms)					Edge throughput (MTEPS)		
	BFS	BC	SSSP	CC	PageRank	BFS	BC	SSSP
kron_g500-logn17 ( $v = 2^{17}, e = 10.2\text{M}$ )	4.69	60.38	12	11.76	91.5	2180	169.4	852.3
kron_g500-logn18 ( $v = 2^{18}, e = 21\text{M}$ )	7.34	55.51	20	20.5	270.8	2885	381.3	1058
kron_g500-logn19 ( $v = 2^{19}, e = 43.6\text{M}$ )	12.69	138.7	36	45.99	917.2	3434	314.2	1210
kron_g500-logn20 ( $v = 2^{20}, e = 89.2\text{M}$ )	23.65	337.8	76	128.8	2686	3774	361.4	1174
kron_g500-logn21 ( $v = 2^{21}, e = 182\text{M}$ )	44.53	746.8	200	292.7	6563	4089	243.8	910.4

Table 3: Scalability of 5 Gunrock primitives (runtime and edges traversed per second) on a single GPU on five differently-sized synthetically-generated Kronecker graphs with similar scale-free structure. BFS, BC and SSSP scale linearly; PR and CC show non-ideal scaling because the number of iterations increases with the dataset size.

on BFS-based primitives (BFS, BC, and SSSP) shows comparatively better results when compared to other graph libraries on two scale-free graphs, kron and soc, than on two small-degree large-diameter graphs, bitcoin and roadnet. The primary reason is our load-balancing strategy during traversal, and particularly our emphasis on good performance for highly irregular graphs. As well, graphs with uniformly low degree expose less parallelism and would tend to show smaller gains in comparison to CPU-based methods. Table 3 shows Gunrock has good scalability with increasing dataset size.

**vs. CPU Graph Libraries** We compare Gunrock’s performance with three CPU graph libraries: the Boost Graph Library (BGL) [27], one of the highest-performing single-CPU graph libraries [19]; PowerGraph, a popular distributed graph library [12]; and Ligra, one of the highest-performing multi-core shared-memory graph libraries [26]. Against both BGL and PowerGraph, Gunrock is at least an order of magnitude faster on average on all primitives. Compared to

Ligra, Gunrock’s performance is generally comparable on most tested graph primitives; note Ligra uses both CPUs effectively. The performance inconsistency for SSSP is due to comparing our Dijkstra-based method with Ligra’s Bellman-Ford algorithm. Our SSSP’s edge throughput is smaller than BFS but similar to BC because of similar computations (atomicMin vs. atomicAdd) and a larger number of iterations for convergence.

**vs. Hardwired GPU Implementations and GPU Libraries** Compared to hardwired GPU implementations, depending on the dataset, Gunrock’s performance is comparable or better on BFS, BC, and SSSP; for CC, Gunrock is 1.5–2x slower than the hardwired GPU implementation. While still achieving high performance, Gunrock’s abstraction significantly reduces both the development time and the code size of graph primitives. Our library code base including the operator implementation and other utility functions contains over 15,000 lines of code, but for a new

graph primitive, users only need to write from 133 (simple primitive, BFS) to 261 (complex primitive, SALSA) lines of code. Writing Gunrock code may require parallel programming concepts (e.g., atomics) but neither details of low-level GPU programming nor optimization knowledge. Annotated code for BFS and SALSA is available in the appendix and at [http://gunrock.github.io/gunrock/doc/annotated\\_primitives/annotated\\_primitives.html](http://gunrock.github.io/gunrock/doc/annotated_primitives/annotated_primitives.html).

Gunrock compares favorably to existing GPU graph libraries. MapGraph is faster than Medusa on all but one test [8] and Gunrock is faster than MapGraph on all but two: the geometric mean of Gunrock’s speedups over MapGraph on BFS, PageRank, SSSP, and CC are 2.1, 2.6, 1.1, and 12.8, respectively. All three GPU BFS-based high-level-programming-model efforts (Medusa, MapGraph, and Gunrock) adopt load-balancing strategies from Merrill et al.’s BFS [20]. While we would thus expect Gunrock to show similar performance on BFS-based graph primitives as these other frameworks, we attribute our performance advantage primarily to the improvements to efficient and load-balanced traversal that are integrated into the Gunrock core.

Figure 6 shows how different optimization strategies improve the performance of graph traversal; here we use BFS as an example. As noted in Section 4.4, the load-balancing traversal method works better on social graphs with irregular distributed degrees, while the Thread-Warp-CTA method works better on graphs where most nodes have small degrees. The direction-optimal traversal strategy also works better on social graphs, whereas on the road-network and bitcoin-transactions graph, we see less concurrent discovery and the performance benefits are not as significant. In general, we can predict which strategies will be most beneficial based only on the degree distribution; many application scenarios may allow precomputation of this distribution and thus we can choose the optimal strategies before we begin computation.

## 7. FUTURE WORK & CONCLUSIONS

Gunrock was born when we spent two months writing a single hardwired GPU graph primitive. We knew that for GPUs to make an impact in graph analytics, we had to raise the level of abstraction in building graph primitives. From the beginning, we designed Gunrock with the GPU in mind, and its traversal-compute abstraction has proven to map naturally to the GPU, giving us both good performance and good flexibility. We have also found that implementing this abstraction has allowed us to integrate numerous optimization strategies, including multiple load-balancing strategies for traversal, direction-optimal traversal, and a two-level priority queue. The result is a framework that is general (able to implement numerous simple and complex graph primitives), straightforward to program (new primitives only take a few hundred lines of code and require no parallel programming knowledge), and fast (on par with hardwired primitives and faster than any other programmable GPU graph library).

**More flexible operators** As we have discussed in Section 4.1, we are targeting more traversal operators such as vertex-merge and neighborhood/subset update to support a wider range of graph algorithms as well as mutable graphs.

**Queue sampling** Sampling elements from the frontier and performing graph traversal and computation only on those

samples for each iteration may achieve equivalent results at a fraction of the runtime of the entire graph. A general framework for doing probabilistic sampling on the GPU is a challenging and interesting research direction.

**Graph primitives on external memory and multi-node GPUs** Today, many graph processing algorithms are still compute- or memory-bound when running on large datasets. This makes a multi-CPU/multi-GPU architecture running across multiple nodes a promising solution. Gunrock’s API is also designed with a multi-node architecture in mind and we are currently exploring multi-GPU and out-of-core extensions to Gunrock.

## 8. ACKNOWLEDGMENTS

Thanks to Erich Elsen and Vishal Vaidyanathan from Royal Caliber for their technical discussion. This work was funded by the DARPA XDATA program under AFRL Contract FA8750-13-C-0002, by NSF awards CCF-1017399 and OCI-1032859, and by UC Lab Fees Research Program Award 12-LR-238449.

## 9. REFERENCES

- [1] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC ’12, pages 12:1–12:10, Nov. 2012.
- [2] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.
- [3] M. Burtcher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on GPUs. In *IEEE International Symposium on Workload Characterization*, IISWC 2012, pages 141–151, Nov. 2012.
- [4] D. Cederman and P. Tsigas. On dynamic load-balancing on graphics processors. In *Graphics Hardware 2008*, pages 57–64, June 2008.
- [5] A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel GPU methods for single source shortest paths. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, pages 349–359, May 2014.
- [6] D. Delling, A. V. Goldberg, A. Nowatzky, and R. F. Werneck. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73:940–952, Sept. 2010.
- [7] E. Elsen and V. Vaidyanathan. A vertex-centric CUDA/C++ API for large graph analytics on GPUs using the gather-apply-scatter abstraction, 2013.
- [8] Z. Fu, M. Personick, and B. Thompson. MapGraph: A high level API for fast development of high performance graph analytics on GPUs. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES ’14, pages 2:1–2:6, June 2014.
- [9] A. Geil, Y. Wang, and J. D. Owens. WTF, GPU! Computing Twitter’s who-to-follow on the GPU. In *Proceedings of the Second ACM Conference on Online Social Networks*, COSN ’14, Oct. 2014.
- [10] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *Proceedings*

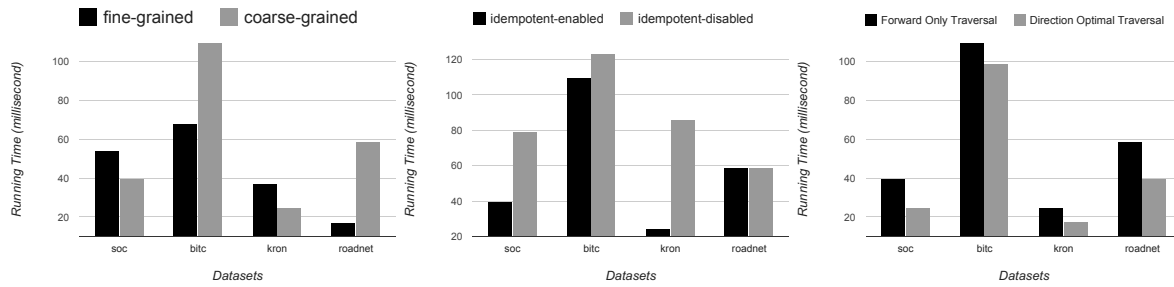


Figure 6: Left: Performance comparison with two different workload mapping optimizations. Middle: Performance comparison on graph traversal with idempotent operations enabled vs. disabled. Right: Performance comparison between forward and direction optimal graph traversal.

- of the Tenth Workshop on Algorithm Engineering and Experiments, ALENEX08, pages 90–100, Jan. 2008.
- [11] A. Goel. The “who-to-follow” system at Twitter: Algorithms, impact, and further research. WWW 2014 industry track, 2014.
  - [12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI '12*, pages 17–30. USENIX Association, Oct. 2012.
  - [13] D. Gregor and A. Lumsdaine. The parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)*, July 2005.
  - [14] J. Greiner. A comparison of parallel algorithms for connected components. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '94*, pages 16–25, June 1994.
  - [15] Y. Jia, V. Lu, J. Hoberock, M. Garland, and J. C. Hart. Edge v. node parallelism for graph centrality metrics. In W. W. Hwu, editor, *GPU Computing Gems Jade Edition*, chapter 2, pages 15–28. Morgan Kaufmann, Oct. 2011.
  - [16] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. CuSha: Vertex-centric graph processing on GPUs. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing, HPDC '14*, pages 239–252, June 2014.
  - [17] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Proceedings of the Twenty-Sixth Annual Conference on Uncertainty in Artificial Intelligence, UAI-10*, pages 340–349, July 2010.
  - [18] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10*, pages 135–146, June 2010.
  - [19] R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader. A performance evaluation of open source graph databases. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications, PPAA '14*, pages 11–18, Feb. 2014.
  - [20] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '12*, pages 117–128, Feb. 2012.
  - [21] U. Meyer and P. Sanders.  $\Delta$ -stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, Oct. 2003. 1998 European Symposium on Algorithms.
  - [22] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of ACM Symposium on Operating Systems Principles, SOSP '13*, pages 456–471, Nov. 2013.
  - [23] P. R. Pande and D. A. Bader. Computing betweenness centrality for small world networks on a GPU. In *HPEC*, 2011.
  - [24] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11*, pages 12–25, June 2011.
  - [25] A. E. Sariyüce, K. Kaya, E. Saule, and U. V. Çatalyürek. Betweenness centrality on GPUs and heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, GPGPU-6*, pages 76–85, Mar. 2013.
  - [26] J. Shun and G. E. Blelloch. 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*, pages 135–146, Feb. 2013.
  - [27] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, Dec. 2001.
  - [28] J. Soman, K. Kishore, and P. J. Narayanan. A fast GPU algorithm for graph connectivity. In *24th IEEE International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum, IPDPSW 2010*, pages 1–8, Apr. 2010.
  - [29] S. Tzeng, B. Lloyd, and J. D. Owens. A GPU task-parallel model with dependency resolution. *IEEE Computer*, 45(8):34–41, Aug. 2012.
  - [30] J. Zhong and B. He. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, June 2014.

## APPENDIX

### A. ANNOTATED GUNROCK CODE

Below is annotated code for the BFS and SALSA graph primitives in Gunrock. These annotations are located in the Gunrock documentation at [http://gunrock.github.io/gunrock/doc/annotated\\_primitives/annotated\\_primitives.html](http://gunrock.github.io/gunrock/doc/annotated_primitives/annotated_primitives.html).

#### bfs\_problem.cuh

This data structure (the "Problem" struct) stores the graph topology in CSR format and the frontier. All Problem structs inherit from the ProblemBase struct. Algorithm-specific data is stored in a "DataSlice".

MARK\_PREDECESSORS sets the predecessor node ID during a traversal for each node in the new frontier.

ENABLE\_IDEMPOTENCE is an optimization when the operation performed in parallel for all neighbor nodes/edges is idempotent, meaning data races are benign.

The DataSlice struct stores per-node or per-edge arrays and global variables (if any) that are specific to this particular algorithm. Here, we store the depth value and predecessor node ID for each node.

The constructor and destructor are ignored here.

"Extract" copies labels and predecessors back to the CPU.

The Init function initializes this Problem struct with a CSR graph that's stored on the CPU. It also initializes the algorithm-specific data, here depth and predecessor.

The Reset function primes the graph data structure to an untraversed state.

Set all depth and predecessor values to invalid. Set the source node's depth value to 0.

Put the source node ID into the initial frontier.

```
template<
    typename VertexId,
    typename SizeT,
    typename Value,
    bool _MARK_PREDECESSORS, // Whether to mark predecessor ID when advance
    bool _ENABLE_IDEMPOTENCE, // Whether to enable idempotence when advance
    bool _USE_DOUBLE_BUFFER>
struct BFSProblem : public ProblemBase<VertexId, SizeT, _USE_DOUBLE_BUFFER>
{
    static const bool MARK_PREDECESSORS = _MARK_PREDECESSORS;

    static const bool ENABLE_IDEMPOTENCE = _ENABLE_IDEMPOTENCE;

    struct DataSlice
    {
        VertexId *d_labels; // BFS depth value
        VertexId *d_preds; // Predecessor IDs
    };

    SizeT nodes;
    SizeT edges;
    DataSlice *d_data_slices;

    cudaError_t Extract(VertexId *h_labels, VertexId *h_preds)
    {
        cudaError_t retval = cudaSuccess;
        if (retval = util::GError(CopyGPU2CPU(data_slices[0]->d_labels, h_labels, nodes))) break;
        if (retval = util::GError(CopyGPU2CPU(data_slices[0]->d_preds, h_preds, nodes))) break;
        return retval;
    }

    cudaError_t Init(
        const Csr<VertexId, Value, SizeT> &graph)
    {
        cudaError_t retval = cudaSuccess;
        if (retval = util::GError(ProblemBase::Init(graph))) break;
        if (retval = util::GError(GPUMalloc(data_slices[0]->d_labels, nodes))) break;
        if (retval = util::GError(GPUMalloc(data_slices[0]->d_preds, nodes))) break;
        return retval;
    }

    cudaError_t Reset(
        const Csr<VertexId, Value, SizeT> &graph, VertexId src)
    {
        cudaError_t retval = cudaSuccess;
        if (retval = util::GError(ProblemBase::Reset(graph))) break;

        util::MemsetKernel<<<BLOCK, THREAD>>>>(data_slices[0]->d_labels, INVALID_NODE_VALUE, nodes);
        util::MemsetKernel<<<BLOCK, THREAD>>>>(data_slices[0]->d_preds, INVALID_PREDECESSOR_ID, nodes);
        if (retval = util::GError(CopyGPU2CPU(data_slices[0]->d_labels+src, 0, 1)))

        if (retval = util::GError(CopyGPU2CPU(g_slices[0]->ping_pong_working_queue, src, 1)));
        return retval;
    }
};
```

## bfs\_functor.cuh

bfs\_functor defines user-specific computations with (1) two per-edge functors, CondEdge and ApplyEdge, which will be used in the Advance operator; and (2) two per-node functors, CondVertex and ApplyVertex, which will be used in the Filter operator.

Set predecessor for each destination node. We set the depth later, because we only want to set the depth for valid nodes.

If we're not keeping track of predecessors, we can immediately set the depth of the destination vertex to one plus the source vertex's depth.

ApplyEdge here increments the depth value.

We know the destination node is valid (from CondEdge), so here we set its depth to one plus the source vertex's depth.

In BFS, CondVertex checks if the vertex is valid in the next frontier.

In BFS, we don't apply any actions to vertices.

## bfs\_enactor.cuh

The enactor defines how a graph primitive runs. It calls traversal (advance and filter operators) and computation (functors).

For BFS, Constructor, Destructor, and Setup functions are ignored

Start with the Setup function to initialize kernel running parameters

Define the graph topology data pointer (g\_slice) and the problem-specific data pointer (d\_slice)

Initialize the queue length (frontier size) to 1.

We ping-pong between old and new frontiers; "selector" picks which one is the current destination.

Here we sequence our operators and functors. For BFS, we alternate between advancing to a new frontier (vertex-to-vertex), calling the BFS functor to set depths along the way, and then filtering out invalid nodes from the new frontier. We repeat until the frontier is empty.

This advance is vertex-to-vertex

The entry point in the driver code to BFS is this Enact call.

Gunrock provides recommended settings here for kernel parameters, but they can be changed by end-users.

```
template<typename Vertexid, typename SizeT, typename Value, typename ProblemData>
struct BFSFunctor {
    typedef typename ProblemData::DataSlice DataSlice;

    __device__ bool CondEdge(Vertexid s_id, Vertexid d_id, DataSlice *p)
    {
        if (ProblemData::MARK_PREDECESSORS)

            return (atomicCAS(&p->d_preds[d_id], INVALID_PREDECESSOR_ID, s_id) == INVALID_PREDECESSOR_ID)
                ? true : false;
        else

            return (atomicCAS(&p->d_labels[d_id], INVALID_NODE_VALUE, s_id+1) == INVALID_NODE_VALUE)
                ? true : false;
    }

    __device__ void ApplyEdge(Vertexid s_id, Vertexid d_id, DataSlice *p)
    {
        if (ProblemData::MARK_PREDECESSORS)

            p->d_labels[d_id] = p->d_labels[s_id]+1;
    }

    __device__ void CondVertex(Vertexid node, DataSlice *p)
    {
        return node != INVALID_NODE_ID;
    }

    __device__ void ApplyVertex(Vertexid node, DataSlice *p)
    {
    }
};
```

```
class BFSEnactor : public EnactorBase {
```

```
    template<
        typename AdvancePolicy,
        typename FilterPolicy,
        typename BFSProblem>
    cudaError_t EnactBFS(CudaContext &context, BFSProblem *problem, Vertexid src)
    {
```

```
        typedef BFSFunctor<
            typename BFSProblem::Vertexid,
            typename BFSProblem::SizeT,
            typename BFSProblem::Vertexid,
            BFSProblem> BFSFunctor;
```

```
        cudaError_t retval = cudaSuccess;
        if (retval == EnactorBase::Setup(problem)) break;
```

```
        typename BFSProblem::GraphSlice *g_slice = problem->d_graph_slices;
        typename BFSProblem::DataSlice *d_slice = problem->d_data_slices;
```

```
        SizeT queue_length = 1;
```

```
        int selector = 0;
```

```
        while (queue_length > 0) {
            gunrock::oprtr::advance::Kernel
            <AdvancePolicy, BFSProblem, BFSFunctor>
            <<<advance_grid_size, AdvancePolicy::THREADS>>>({
                queue_length,
                g_slice->ping_pong_working_queue[selector],
                g_slice->ping_pong_working_queue[selector+1],
                d_slice,
                context,
```

```
                gunrock::oprtr::advance::V2V);
```

```
            selector ^= 1; // Swap selector
```

```
            gunrock::oprtr::filter::Kernel
            <FilterPolicy, BFSProblem, BFSFunctor>
            <<<filter_grid_size, FilterPolicy::THREADS>>>({
                queue_length,
                g_slice->ping_pong_working_queue[selector],
                g_slice->ping_pong_working_queue[selector+1],
                d_slice);
        }
```

```
        return retval;
    }
```

```
template <typename BFSProblem>
cudaError_t Enact(
    CudaContext &context,
    BFSProblem *problem, // Problem data sent in
    typename BFSProblem::Vertexid src) // Source node ID for BFS
{
```

```
    typedef gunrock::oprtr::filter::KernelPolicy<
        BFSProblem,
        300, //CUDA_ARCH
        8, //MIN_CTA_OCCUPANCY
        8> //LOG_THREAD_NUM
        FilterKernelPolicy;
```

```
    typedef gunrock::oprtr::advance::KernelPolicy<
        BFSProblem,
        300, //CUDA_ARCH
        8, //MIN_CTA_OCCUPANCY
        10, //LOG_THREAD_NUM
        32*128> //THRESHOLD_TO_SWITCH_ADVANCE_MODE
```





then atomically update hub ranks.

Forward advance functors for the Authority nodes (reverse graph) Like the Hub forward advance functor, this just sets all predecessors.

For authority graph (reverse graph), set each edge's source node ID

Backward advance functors for the Authority nodes (reverse graph) The backward advance functors distribute ranks to nodes.

Choose nodes with non-zero outgoing degrees ...

... then atomically update authority ranks.

## salsa\_enactor.cuh

The enactor defines how a graph primitive runs. It calls traversal (advance and filter operators) and computation (functors).

For SALSA, Constructor, Destructor, and Setup functions are ignored

This user-defined function swaps current and next rank pointers

copy next to curr and reset next

This enactor defines the SALSA high-level algorithm.

Define SALSA functors.

Load the Setup function.

```
static __device__ __forceinline__ void ApplyEdge(Vertexid s_id, Vertexid d_id, DataSlice *problem, Vertexid e_id = 0, Vertexid e_id_in = 0)
{
    Value hrank_dst = problem->d_hrank_curr[d_id] / (problem->d_in_degrees[s_id] * problem->d_out_degrees[d_id]);
    Vertexid v_id = problem->d_hub_predecessors[e_id_in];
    atomicAdd(&problem->d_hrank_next[v_id], hrank_dst);
}

template<typename Vertexid, typename SizeT, typename Value, typename ProblemData>
struct AFORWARDFunctor
{
    typedef typename ProblemData::DataSlice DataSlice;

    static __device__ __forceinline__ bool CondEdge(Vertexid s_id, Vertexid d_id, DataSlice *problem, Vertexid e_id = 0, Vertexid e_id_in = 0)
    {
        return true;
    }

    static __device__ __forceinline__ void ApplyEdge(Vertexid s_id, Vertexid d_id, DataSlice *problem, Vertexid e_id = 0, Vertexid e_id_in = 0)
    {
        problem->d_auth_predecessors[e_id] = s_id;
    }
};

template<typename Vertexid, typename SizeT, typename Value, typename ProblemData>
struct ABACKWARDFunctor
{
    typedef typename ProblemData::DataSlice DataSlice;

    static __device__ __forceinline__ bool CondEdge(Vertexid s_id, Vertexid d_id, DataSlice *problem, Vertexid e_id = 0, Vertexid e_id_in = 0)
    {
        Vertexid v_id = problem->d_auth_predecessors[e_id_in];
        bool flag = (problem->d_in_degrees[v_id] != 0);
        if (!flag) problem->d_arank_next[v_id] = 0;
        return flag;
    }

    static __device__ __forceinline__ void ApplyEdge(Vertexid s_id, Vertexid d_id, DataSlice *problem, Vertexid e_id = 0, Vertexid e_id_in = 0)
    {
        Value arank_dst = problem->d_arank_curr[d_id] / (problem->d_out_degrees[s_id] * problem->d_in_degrees[d_id]);
        Vertexid v_id = problem->d_auth_predecessors[e_id_in];
        atomicAdd(&problem->d_arank_next[v_id], arank_dst);
    }
};
```

```
class SalsaEnactor : public EnactorBase {
```

```
template<typename ProblemData>
void SwapRank(ProblemData *problem, int is_hub, int nodes)
{
    typedef typename ProblemData::Value Value;
    Value *rank_curr;
    Value *rank_next;
    if (is_hub) {
        rank_curr = problem->data_slices[0]->d_hrank_curr;
        rank_next = problem->data_slices[0]->d_hrank_next;
    } else {
        rank_curr = problem->data_slices[0]->d_hrank_curr;
        rank_next = problem->data_slices[0]->d_hrank_next;
    }

    util::MemsetCopyVectorKernel<<<128, 128>>>>(rank_curr, rank_next, nodes);
    util::MemsetKernel<<<128, 128>>>>(rank_next, (Value)0.0, nodes);
}
```

```
template<
    typename AdvancePolicy,
    typename FilterPolicy,
    typename SALSAProblem>
    cudaError_t EnactSALSA(
        CudaContext &context,
        SALSAProblem *problem,
        int max_iteration) {

    typedef typename SALSAProblem::Vertexid Vertexid;
    typedef typename SALSAProblem::SizeT SizeT;
    typedef typename SALSAProblem::Value Value;

    typedef HFORWARDFunctor<
        Vertexid,
        SizeT,
        Value,
        SALSAProblem> HForwardFunctor;

    typedef AFORWARDFunctor<
        Vertexid,
        SizeT,
        Value,
        SALSAProblem> AForwardFunctor;

    typedef HBACKWARDFunctor<
        Vertexid,
        SizeT,
        Value,
        SALSAProblem> HBackwardFunctor;

    typedef ABACKWARDFunctor<
        Vertexid,
        SizeT,
        Value,
        SALSAProblem> ABackwardFunctor;

    cudaError_t retval = cudaSuccess;
    if (retval = EnactorBase::Setup(problem)) break;
```

Define the graph topology data pointer (g\_slice) and the problem-specific data pointer (d\_slice).

Now let's do some computation.

First we'll do some initialization code that runs just once. Start by initializing the frontier with all node IDs.

Set predecessor nodes for each edge in the original graph.

And set the predecessor nodes for each edge in the reverse graph.

Now we iterate between two Advance operators, which update (1) the hub rank and (2) the authority rank. We loop until we've reached the maximum iteration count.

This Advance operator updates the hub rank ...

and here, the authority rank.

The entry point in the driver code to SALSAs is this Enact call.

Gunrock provides recommended settings here for kernel parameters, but they can be changed by end-users.

```
typename SALSAProblem::GraphSlice *g_slice = problem->d_graph_slices;
typename SALSAProblem::DataSlice *d_slice = problem->d_data_slices;

SizeT queue_length = g_slice->nodes;
int selector = 0;
{

    util::MemsetdxKernel<<<BLOCK, THREAD>>>>(g_slice->ping_pong_working_queue[selector], g_slice->nodes);

    gunrock::oprtr::advance::Kernel
    <<AdvancePolicy, SALSAProblem, HForwardFuncor>
    <<<<advance_grid_size, AdvancePolicy::THREADS>>>>({
        queue_length,
        g_slice->ping_pong_working_queue[selector],
        g_slice->ping_pong_working_queue[selector+1],
        g_slice->d_row_offsets,
        g_slice->d_column_indices, //advance on original graph
        d_slice,
        context,
        gunrock::oprtr::advance::VZE);

    gunrock::oprtr::advance::Kernel
    <<AdvancePolicy, SALSAProblem, AForwardFuncor>
    <<<<advance_grid_size, AdvancePolicy::THREADS>>>>({
        queue_length,
        g_slice->ping_pong_working_queue[selector],
        g_slice->ping_pong_working_queue[selector+1],
        g_slice->d_column_offsets,
        g_slice->d_row_indices, //advance on reverse graph
        d_slice,
        context,
        gunrock::oprtr::advance::VZE);
    }

    int iteration = 0;
    while (true) {
        util::MemsetdxKernel<<<BLOCK, THREAD>>>>(g_slice->ping_pong_working_queue[selector], g_slice->edges);
        SizeT queue_length = g_slice->edges;

        gunrock::oprtr::advance::Kernel
        <<AdvancePolicy, SALSAProblem, ABackwardFuncor>
        <<<<advance_grid_size, AdvancePolicy::THREADS>>>>({
            queue_length,
            g_slice->ping_pong_working_queue[selector],
            g_slice->ping_pong_working_queue[selector+1],
            g_slice->d_column_offsets,
            g_slice->d_row_indices, //advance backward on reverse graph
            d_slice,
            context,
            gunrock::oprtr::advance::EZV);

        SwapRank<SALSAProblem>(problem, 0, g_slice->nodes);

        gunrock::oprtr::advance::Kernel
        <<AdvancePolicy, SALSAProblem, ABackwardFuncor>
        <<<<advance_grid_size, AdvancePolicy::THREADS>>>>({
            queue_length,
            g_slice->ping_pong_working_queue[selector],
            g_slice->ping_pong_working_queue[selector+1],
            g_slice->d_row_offsets,
            g_slice->d_column_indices, //advance backward on original graph
            d_slice,
            context,
            gunrock::oprtr::advance::EZV);

        SwapRank<SALSAProblem>(problem, 0, g_slice->nodes);

        iteration++;
        if (iteration >= max_iteration) break;
    }

    return retval;
}

template <typename SALSAProblem>
cudaError_t Enact(
    CudaContext &context,
    SALSAProblem *problem,
    typename SALSAProblem::SizeT max_iteration)
{

    typedef gunrock::oprtr::filter::KernelPolicy<
        SALSAProblem,
        300, //CUDA_ARCH
        8, //MIN_CTA_OCCUPANCY
        8> //LOG_THREAD_NUM
        FilterKernelPolicy;

    typedef gunrock::oprtr::advance::KernelPolicy<
        SALSAProblem,
        300,
        8, //MIN_CTA_OCCUPANCY
        10, //LOG_THREAD_NUM
        32*128> //THRESHOLD_TO_SWITCH_ADVANCE_MODE
        AdvanceKernelPolicy;

    return EnactSALSA<AdvanceKernelPolicy, FilterKernelPolicy, SALSAProblem>({
        context, problem, max_iteration);
}
};
```