Mini-Gunrock: A Lightweight Graph Analytics Framework on the GPU

Yangzihao Wang
University of California, Davis
yzhwang@ucdavis.edu

Sean Baxter
moderngpu@gmail.com

John D. Owens
University of California, Davis
jowens@ece.ucdavis.edu

***Abstract*—Existing GPU graph analytics frameworks are typically built from specialized, bottom-up implementations of graph operators that are customized to graph computation. In this work we describe Mini-Gunrock, a lightweight graph analytics framework on the GPU. Unlike existing frameworks, Mini-Gunrock is built from graph operators implemented with generic transform-based data-parallel primitives. Using this method to bridge the gap between programmability and high performance for GPU graph analytics, we demonstrate operator performance on scale-free graphs with an average 1.5x speedup compared to Gunrock's corresponding operator performance. Mini-Gunrock's graph operators, optimizations, and applications code have 10x smaller code size and comparable overall performance vs. Gunrock.**

***Keywords*-GPU computing; Graph analytics; Programming model; Runtime system**

I. Introduction

The high-performance, highly parallel, fully programmable modern Graphics Processing Unit's (GPU) high memory bandwidth, computing power, excellent peak throughput, and energy efficiency bring acceleration to regular applications that have extensive data parallelism, regular memory access patterns, and modest synchronization requirements.

However, for graph analytics, the inherent irregularity of graph data structures leads to irregularity in data access and control flow, making efficient graph analytics on GPUs a significant challenge. Promising specialized GPU graph algorithm implementations can achieve high performance, but usually at the price of poor portability and programmability. And more generalized parallel graph analytics systems with better programmability may sacrifice performance.

Our Gunrock framework [1] is built on an abstraction of operations on frontiers of vertices or edges. Like most GPU graph frameworks, Gunrock's implementation is bottom-up and focuses on custom, graph-specific, highly efficient implementations of these operators. In this work we describe Mini-Gunrock, a lightweight graph analytics framework on the GPU that implements Gunrock's abstraction on top of generic data-parallel primitives from *moderngpu 2.0* [2]. With this work we hope to bridge the gap between programmability and high performance by reducing both the application code size and operator development code size. Our thesis in this paper is that moderngpu 2.0's generic data-parallel primitives can implement the necessary operators for a graph library using parallel transforms, a type of operation which applies an arbitrary functor to each item in the input. We demonstrate that our Mini-Gunrock implementation successfully leverages moderngpu's high performance to deliver high-performance graph primitives.

Mini-Gunrock's operator performance on scale-free graphs is on average 1.5x faster than Gunrock's corresponding operator performance, and graph primitive performance of Mini-Gunrock is comparable with Gunrock's corresponding graph primitive performance. Mini-Gunrock implements most of the graph operators, optimizations, and applications from Gunrock with 10x smaller code size (not counting moderngpu) and comparable overall performance. Mini-Gunrock and its bottom-up method for building a graph analytics system show the flexibility of Gunrock's data-centric abstraction. Our implementation enables rapid prototyping of both new graph operators and new graph primitives.

II. Overview

The purpose of Gunrock is to develop a programmable graph processing system with high performance. Its implementation is based on a data-centric programming model that increases flexibility and programmability over other GPU graph processing libraries and achieves comparable performance to specialized GPU graph algorithms. Our work on Gunrock led us to the following question: What is the right level of abstraction of a graph analytics programming model on the GPU? Gunrock's data-centric programming model focuses on evolving frontiers when running a graph primitive. One iteration could be generalized as an equation:

$$f_{\text{out}} = Op(f_{\text{in}}, G, P) \tag{1}$$

where f_{in} and f_{out} are the input and output frontiers respectively, $Op(f, G, P)$ is a graph operation on frontier f using data from graph G and per-node or/and per-edge data P. Gunrock implements each graph operator with specific low-level optimizations. Gunrock uses several common data-parallel primitives in different graph operators, which can be categorized into three groups: **Per-Item Computation:** simple regular kernels; **Prefix-Sum Based:** prefix-sum and streaming compaction; and **Irregular Computation:** load-balanced search and segmented reduction.

In this work, we take another approach to implementing Gunrock's data-centric programming model: building graph operators on top of lower-level common data-parallel primitives. We call the resulting lightweight graph analytics framework "Mini-Gunrock". We have three design goals for Mini-Gunrock:

Flexibility: To have the same features as in Gunrock;

Simplicity: To enable fast development of new graph operators and primitives with less and cleaner code; and

Performance: To have comparable performance to Gunrock.

To achieve the above goals, we need to use a set of lower-level data-parallel primitives that allow sufficient flexibility while offering state-of-the-art performance. Moderngpu 2.0 has implemented a set of data-parallel primitives based on transforms; each of them can take user-specified functors but otherwise inherit the high performance offered by moderngpu 1.0. We implement Mini-Gunrock on top of moderngpu's transform-based primitives.

III. Related Work

Generic GPU Primitive Libraries: Popular primitive libraries on the GPU include the CUDA Data Parallel Primitive Library (CUDPP), CUDA Unbound (CUB), and the moderngpu library we use in this work. We choose moderngpu because its family of transform operators provide both the generality we require to implement Gunrock operators as well as an implementation prioritizing the load balancing of irregular workloads.

GPU Graph Processing Libraries: Several works target the construction of a high-level GPU graph processing library that delivers both good performance and good programmability. Wang et al. [3] gave a detailed survey of GPU graph libraries. They observe that when designing graph analytics libraries on the GPU, rather than starting with generic primitives and building the graph operators on top of them, most recent work instead descends from a highly optimized implementation that was narrowly graph-specific, with little of their implementation focus on generic data-parallel primitives. For instance, CuSha [4] generally does not rely on any generic library, but uses one customized reduce kernel in the traversal process. Medusa [5] used CUDPP's segmented scan primitive in its message combiner operator. The generic primitives in these two libraries were not part of their programming model, but rather served as utility functions. VertexAPI2 [6], MapGraph [7], and Gunrock [1] all used moderngpu 1.0's primitives in their neighbor list expansion operator. However, the use of generic primitives in these libraries is limited to the low-level optimizations and does not influence their programming models. No current GPU-based graph library has built its architecture atop generic primitives.

IV. Transform-based Data-Parallel Primitives

The transform algorithm is one of the mutating sequence operations in the C++ STL. A transform applies the same operation to every element in a range of input iterators. If the operation takes two input iterators, it is a *binary* operation; if one, a *unary* operation. In the context of this paper, "transform" refers to the unary transform because moderngpu 2.0 does not currently support a binary transform. Transform is a mutating algorithm; it could modify input items, and it returns an iterator to the end of the output range. In this section, we discuss Gunrock's graph operator design and moderngpu 2.0's generic primitive design and show how they connect from the perspective of transform-based data-parallel primitives.

A. Gunrock's Graph Operators

Gunrock's data-centric programming model defines graph operators as operations that manipulate frontiers (arrays of vertex or edge IDs). There are four graph operators in Gunrock:

Advance takes an input frontier, visits the neighbor list of each item in the input frontier, does per-item processing, and outputs the neighbor lists as the output frontier.

Filter takes an input frontier, visits each item in the input frontier, does per-item processing, compacts the input frontier according to some user-defined predicate, and generates the output frontier.

Segmented Intersection takes two input frontiers with the same length, visits the neighbor lists of each pair, computes the intersection and does per-item processing, and generates the output frontier.

Compute takes an input frontier, visits each item, and does per-item processing. In Gunrock it is usually combined with other operators and serves as a user-defined functor.

To generalize, Gunrock's graph operator can be presented in the following way: a Gunrock graph operator takes one or more input frontiers,[1] processes a user-defined compute functor in parallel, and outputs a new frontier. While implementing these operators in parallel is not particularly difficult, implementing them with high performance in the face of highly irregular workloads (data-dependent branches and high variance in the size of neighbor lists) is a significant challenge.

[1]For segmented intersection, this input frontier can be either two vertex frontiers with the same length or one edge list frontier where each item has two end vertex IDs.

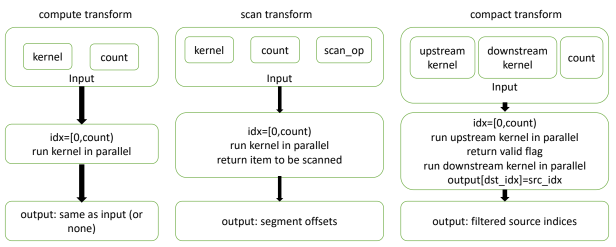

Figure 1: *Compute, scan and compact transforms in Mini-Gunrock.*

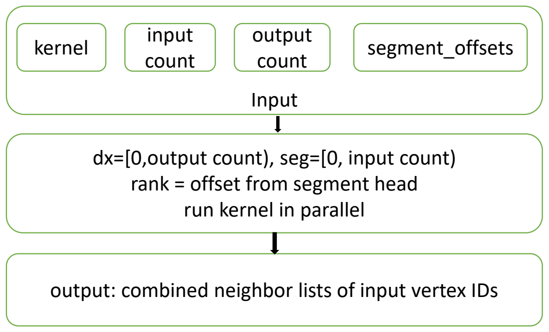
kernel
input count
output count
segment_offsets
Input
dx=[0,output count), seg=[0, input count)
rank = offset from segment head
run kernel in parallel
output: combined neighbor lists of input vertex IDs

Figure 2: *Lbs transforms in Mini-Gunrock.*

B. Moderngpu 2.0's Generic Primitives

Moderngpu 2.0's transforms include *scan_transform* (parallel transform for prefix-sum), *lbs_transform* (parallel transform for merge-based load balanced search), *segreduce_transform* (parallel transform for segmented reduction), and *compact_transform* (parallel transform for streaming compaction). *lbs_transform* is a parallel search of an array of sorted input items in an array of sorted output items, so that the input items can then be coordinated with their output items. It essentially turns an offset array and a neighbor list array into an edge list in parallel, which in practice implements the same strategy as Gunrock's merge-based load-balanced partitioning workload mapping functionality.

Figure 1, Figure 2, and Figure 3 show the data flow of five transforms we use in Mini-Gunrock. For every transform, users can define a named functor or a lambda function. Different transforms have different default auxiliary parameters for the functor. For the regular transform, scan transform, and compact transform, there is only one single auxiliary parameter *idx*, the index of the input element. For lbs transform and segreduce transform, there are three auxiliary parameters *idx*, *seg*, and *rank*, where *idx* is the index of the output frontier, *seg* is the index of the input frontier, and *rank* is the local offset value of the neighbor item. Using these auxiliary parameters and different combinations of these transforms, we can implement most of our graph operators: scan_transform and lbs_transform share similar goals to advance, compact_transform is similar to filter, and general transform is similar to compute. More importantly, by implementing our operators using moderngpu transforms, we can both leverage moderngpu's excellent performance on irregular inputs as well as incorporate numerous Gunrock optimizations. As we will show, using transform-based primitives achieves comparable performance to Gunrock.

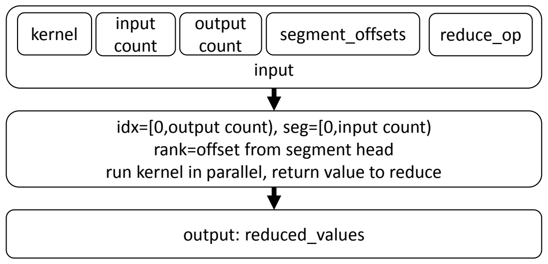
kernel
input count
output count
segment_offsets
reduce_op
input
idx=[0,output count), seg=[0,input count)
rank=offset from segment head
run kernel in parallel, return value to reduce
output: reduced_values

Figure 3: *Segreduce transforms in Mini Gunrock.*

Moreover, using transform-based primitives gives us additional benefits compared with the graph operators we defined in Gunrock. Transform is more flexible as it can define both regular and irregular tasks. Using transform enables a smaller code size for the implementation of graph operators.

V. Graph Operator Implementation Using Transforms

Built on top of moderngpu transforms, our graph operator implementations in Mini-Gunrock benefit from the high performance of moderngpu transforms while maintaining a simple and clean interface.

Filter Operator Mini-Gunrock's filter operator is implemented using *compact_transform*. The transform-based implementation has two phases: 1) an upsweep phase to mark flags (via a user-defined functor) for input items to be filtered, and 2) a downsweep using prefix-sum to generate the compacted output indices and write outputs in parallel.

Advance Operator The advance operator can be implemented using *scan_transform* and *lbs_transform* (as shown in Figure 4). The *scan_transform* on the neighbor list lengths of all the input items will generate an offsets array, which serves as the beginning pointer of each input item's neighbor list in the output frontier. This array is then sent into *lbs_transform* to do the load-balanced search. Instead of having to implement sorted search, allocate shared memory, and handle communications between threads, like what Gunrock has implemented in its advance operator, *lbs_transform* automatically handles the workload mapping among all threads once given the offsets array, and generates the necessary information for the computation on each thread. As a major operator for all traversal-based

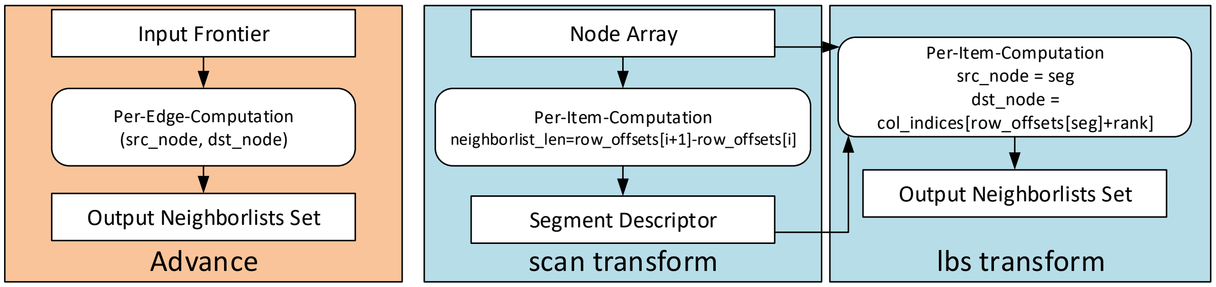
Input Frontier
Per-Edge-Computation (src_node, dst_node)
Output Neighborlists Set
Advance
Node Array
Per-Item-Computation neighborlist_len=row_offsets[i+1]-row_offsets[i]
Segment Descriptor
scan transform
Per-Item-Computation src_node = seg dst_node = col_indices[row_offsets[seg]+rank]
Output Neighborlists Set
lbs transform

Figure 4: *The left block shows the input, output, and parallel operation of an advance operator in Mini-Gunrock; the right block shows its implementation using transform-based primitives.*

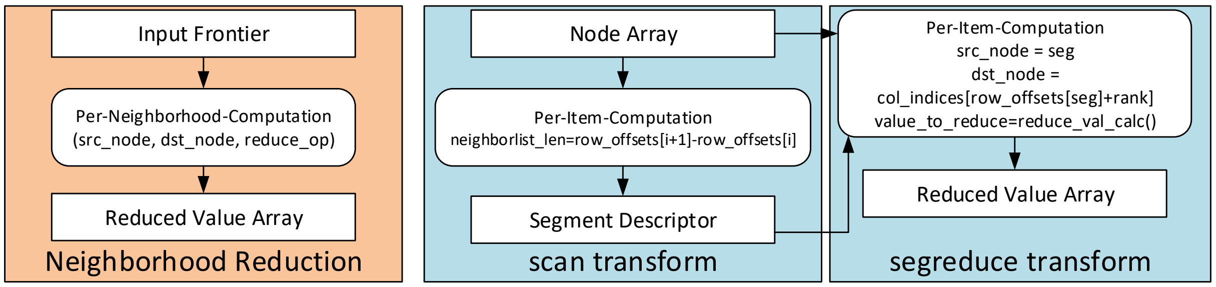
Input Frontier
Per-Neighborhood-Computation (src_node, dst_node, reduce_op)
Reduced Value Array
Neighborhood Reduction
Node Array
Per-Item-Computation neighborlist_len=row_offsets[i+1]-row_offsets[i]
Segment Descriptor
scan transform
Per-Item-Computation src_node = seg dst_node = col_indices[row_offsets[seg]+rank] value_to_reduce=reduce_val_calc()
Reduced Value Array
segreduce transform

Figure 5: *The left block shows the input, output, and parallel operation of a neighborhood reduction operator in Mini-Gunrock. The right block shows its implementation using transform-based primitives.*

graph primitives and several other graph primitives, making Mini-Gunrock's advance efficient is one of the most critical aspects of its high performance. The lbs transform allows us to achieve that goal.

Neighborhood Reduction Operator Our neighborhood reduction operator can be implemented using *scan_transform* and *segreduce_transform* (as shown in Figure 5). In this operator, one pass of scan serves the same purpose as in the advance operator. The following segmented reduction transform internally fuses the merge-based load-balanced search with the reduction computation. It has the same level of efficiency as the advance operator. We can easily choose whether to output the visited neighbor nodes, or to output a list of nodes computed by the reduction operator, depending on our next operation in the graph primitive. Also, we can set the neighborhood to be either out-degree neighbors (for a push-style neighborhood reduction) or in-degree neighbors (for a pull-style neighborhood reduction).

Compute Operator Our compute operator is merely a regular computation task on each input item. It is a basic *transform* with a user-defined functor as the computation step. The difference between *transform* and a specialized regular GPU kernel is that for *transform*, the algorithm will launch kernels with optimal settings that maximize utilization according to a set of GPU-architecture- and input-length-dependent pre-defined conditions.

A. *Mini-Gunrock vs. Gunrock*

This section compares Mini-Gunrock and Gunrock at the graph operator level.

The goal for Mini-Gunrock is to make graph operator implementation easy and flexible. Such goal can be achieved by either a wrapper of a combination of transform-based primitives or a general transform with a simple kernel function. As a more general type of primitive, transform-based primitives and combinations of them separate low-level optimizations from higher-level operator/primitive design. They enable the developers who want to add new graph primitives in the system to focus on the higher-level design of graph operators and primitives. Most of Gunrock's current graph operators can be inplemented using transform-based primitives and their combinations, with the exception of segmented intersection, which we discuss in Section IX.

Compared to Mini-Gunrock, Gunrock's graph operator implementation is more flexible. Gunrock users have control of both launch settings and shared memory and local memory allocation. Gunrock's implementation uses advanced GPU techniques like persistent threads. With such flexibility, users can also switch the optimization strategies used underneath the operator abstraction as needed. For neighborhood reduction, Gunrock's current implementation requires a separated segmented reduction pass after its advance, which is not optimal. However, it is possible to fuse this additional segmented reduction pass with the current advance kernel to increase performance. The downside of Gunrock's implementation is that it mixes its optimizations with graph operator design, exposing too many interfaces and parameters to users. This adds to the complexity of building any graph operator or graph primitive.

VI. Optimizations in Mini-Gunrock

A. *Advance-based Optimizations Using Transforms*

Pull-based Traversal A push-based advance expands the neighbor lists of the current input frontier; this is normally supported in all GPU graph libraries. A pull-based advance [8] instead intersects the neighbor lists of the unvisited node frontier with the current frontier. This approach is beneficial when the number of unvisited vertices drops below the size of the current frontier and shows from 5x to 30x speedup for BFS on certain scale-free graphs, but vertex-centered GPU frameworks have found it challenging to integrate this optimization into their abstraction. The capability of keeping two active frontiers differentiates Gunrock from other GPU graph processing programming models. Gunrock's implementation first generates a unvisited frontier with all the unvisited nodes, then uses the unvisited frontier in the advance step, visiting all unvisited nodes that have visited predecessors, and generates both a new active frontier and a new unvisited frontier. Mini-Gunrock's pull-based implementation also has two phases: data structure preparation and the actual pull-based advance. For data structure preparation, we use two transforms: one regular transform to generate a dense bitmap frontier representation from the sparse compacted frontier representation, then one *compact_transform* to generate a compacted frontier of unvisited nodes. For the pull-based advance, we use *scan_transform* and *lbs_transform* as in a push-based advance operator, but with a new functor to update the bitmap and process the computation on edge or destination vertices.

Flexible Features Compared to Gunrock, Mini-Gunrock has various flexible features in advance and neighborhood reduction. These include: 1) for an input frontier that contains all of the edges in the graph, save the *scan_transform* pass to directly use the row offsets array; 2) for an advance operator whose output frontier is not used anywhere, skip writing to the output frontier; and 3) for a neighborhood reduction, set direction to forward or backward to process the reduction on either out-degree neighbors or in-degree neighbors. While Gunrock also supports these optimizations, its implementations of optimizations of this type would typically require either additional kernels or various interface changes. Building optimizations on top of transform-based primitives allows Mini-Gunrock's implementation to simply port existing optimizations in Gunrock and add additional optimizations that had not been implemented in Gunrock due to their complexity.

B. *Filter-based Optimizations Using Transforms*

Idempotence The idempotence optimization allows arbitrary heuristic filtering functors to be added in the upsweep pass of the *compact_transform*. An additional benefit of having this optimization in a filter operator is the avoidance of atomic operations in an advance operator when the computation is idempotent, with a tradeoff of a potentially larger frontier size per iteration.

Two-level Priority Queue In Mini-Gunrock, a two-level priority queue can be implemented with a *compact_transform* with a user-defined priority score function and threshold value.

C. *Mini-Gunrock vs. Gunrock*

This section compares Mini-Gunrock and Gunrock from the optimization perspective.

In Mini-Gunrock, we have implemented several optimizations that parallel those in Gunrock. The advance-based optimizations include pull-based traversal and flexible features such as optimization for all-edge input frontiers, output frontier skipping, and neighborhood reduction on either in-degree neighbors or out-degree neighbors. The filter-based optimizations include idempotence and a two-level priority queue.

However, due to the different way Mini-Gunrock and Gunrock process items within operators, the performance of these two optimizations differs between the two implementations. The redundancy-removal heuristics in Mini-Gunrock are less effective compared to Gunrock, thus causing an explosion of frontier size, which slows down the convergence when used. This is because in Gunrock, after we decide the workload for each block, we use a smaller tile to move the value of these items from global memory to local memory and within each block sequentially process each tile. In contrast, in Mini-Gunrock, we can only parallelize over all items globally, which reduces the chance of collisions in hash tables. Another such difference is in pull-based traversal, where each unvisited vertex visits all its parents until it finds a parent in the active frontier. In Mini-Gunrock, just as in Gunrock, multiple threads may be cooperatively working together on one unvisited vertex's list of parents. If a Mini-Gunrock thread finds a parent in the frontier, it can exit its kernel immediately; but (unlike in Gunrock) it cannot also cause a group early exit of all other threads working on that vertex. As a result, the performance gain of pull-based traversal in Mini-Gunrock is limited compared to the same optimization implemented in Gunrock.

Several optimization implementations are similar in both Gunrock and Mini-Gunrock, including kernel-fusion optimizations (e.g., the fusing of compute kernel and traversal operators) and the two-level priority queue (splitting a frontier into two subfrontiers, useful in, e.g., SSSP's delta-stepping formulation).

In general, any optimization that 1) only relies on input frontier and output frontier indices and 2) operates on one item only once can be implemented within the current transform-based framework in moderngpu 2.0.

VII. Graph Primitives using Transforms

One goal of Mini-Gunrock is to quickly build new graph primitive prototypes with minimal code size. We discuss various graph primitives that can be implemented using Mini-Gunrock.

Just as in Gunrock, the three main components of a graph primitive in Mini-Gunrock are: 1) Problem, which provides graph topology data and an algorithm-specific data management interface; 2) Functor, which contains user-defined computation code as a device function; and 3) Enactor, which serves as the entry point of the graph primitive and defines the running process by a series of graph operators. To show how Mini-Gunrock implements primitives in Gunrock, we show the source code of the SSSP enactor and functor in Mini-Gunrock (the SSSP problem code is very simple):

Listing 1: *Graph operator interfaces.*

```
template<typename Problem, typename Functor, bool
     idempotence, bool has_output>
int advance_forward_kernel(std::shared_ptr<Problem>
     problem,
std::shared_ptr<frontier_t<int> > &input,
std::shared_ptr<frontier_t<int> > &output,
int iteration,
standard_context_t &context)

template<typename Problem, typename Functor>
int filter_kernel(std::shared_ptr<Problem> problem,
std::shared_ptr<frontier_t<int> > &input,
std::shared_ptr<frontier_t<int> > &output,
int iteration,
standard_context_t &context)

template<typename Problem, typename Functor, typename
     Value, typename reduce_op, bool has_output>
int neighborhood_kernel(std::shared_ptr<Problem>
     problem,
std::shared_ptr<frontier_t<int> > &input,
std::shared_ptr<frontier_t<int> > &output,
Value  *reduced,
Value identity,
int iteration,
standard_context_t &context)
```

In Mini-Gunrock, we keep the operator interface to the minimum. As in Gunrock, each operator interface also contains its graph primitive data structure type (Problem), its functor structure (Functor), and some auxiliary variables. But for function arguments, Mini-Gunrock only contains the problem object, the input and output frontier, the current iteration number, and a context object needed by all transform-based primitives.

Listing 2: *SSSP Enactor.*

```
void enact(std::shared_ptr<sssp_problem_t> sssp_problem,
     standard_context_t &context) {
       init_frontier(sssp_problem);
       int frontier_length = 1;
       int selector = 0;
       // Start the iteration with the initial frontier
            that contains the source node ID.
       for (int iteration = 0; ; ++iteration) {
           frontier_length = advance_forward_kernel<
                sssp_problem_t, sssp_functor_t, false,
                true>
               (sssp_problem, buffers[selector],
                    buffers[selector^1], iteration,
                    context);
           selector ^= 1;
           frontier_length = filter_kernel<
                sssp_problem_t, sssp_functor_t>
               (sssp_problem, buffers[selector],
                    buffers[selector^1], iteration,
                    context);
           // Use advance to relax node distance value
                and discover new frontier, and filter to
                 remove redundant node IDs, until
                frontier is empty.
           if (!frontier_length) break;
           selector ^= 1;
       }
   }
```

Listing 3: *SSSP Functor.*

```
struct sssp_functor_t {
static __device__ __forceinline__ bool cond_filter(int
      idx, sssp_problem_t::data_slice_t *data, int
     iteration) {
    // If idx is not -1, then it's unique, return true
         to keep it.
    return idx != -1;
}
static __device__ __forceinline__ bool cond_advance(
     int src, int dst, int edge_id, int rank, int
     output_idx, sssp_problem_t::data_slice_t *data,
     int iteration) {
    // If source node's distance value plus the edge
         weight is larger than destination node's
         distance value, update destination node's
         distance value.
    float new_distance = data->d_labels[src]+data->
         d_weights[edge_id];
    float old_distance = atomicMin(data->d_labels+dst,
          new_distance);
    return (new_distance < old_distance);
}
static __device__ __forceinline__ bool apply_advance(
     int src, int dst, int edge_id, int rank, int
     output_idx, sssp_problem_t::data_slice_t *data,
     int iteration) {
    // If destination node's distance value is updated
         , set its predecessor to a new source node.
    data->d_preds[dst] = src;
    return true;
}
};
```

For SSSP, the enactor first initializes two buffers for input and output frontiers, and then iteratively calls a series of graph operators until it ends when the frontier is empty. The functor is similar to Gunrock and contains very simple user-defined computation code. Our design reduces the code size needed for both a new graph operator implementation and a new graph primitive using existing graph operators to more than 10 times smaller than Gunrock. The SSSP enactor and functor only contains 208 lines of code (77 for enactor, 39 for functor, 92 for problem). Both the BFS and PR implementations in Mini-Gunrock are under 200 lines of code. The total lines of code of the advance, filter, and neighborhood reduction operator in Mini Gunrock are 47, 21, and 49 respectively. Section VII-B explains the reason that Mini-Gunrock achieves a much smaller code size.

A. *Using Mini-Gunrock to Implement New Graph Primitives*

The neighborhood reduction operator in Mini-Gunrock enables a family of new graph primitive implementations that require this operation. To show this, we implement one typical algorithm in Mini-Gunrock that uses neighborhood reduction as its core component: graph coloring.

Graph coloring finds the minimum number of colors that can be used to color the nodes of a graph such that no two adjacent nodes (connected with an edge) have the same color. It can be used to find the maximal independent set in a task dependency graph, where nodes are tasks and edges are dependencies between tasks. Although the exact problem is NP-complete, several parallel algorithms exist for finding an approximate graph coloring. We chose the Luby-Jones approach [9] for its simplicity and natural parallelism.

We divide Luby-Jones into three steps that can be implemented using Mini-Gunrock operators:

Compute: Assign a hash number to each node;

Neighborhood Reduction: Find an independent set by selecting nodes that have the maximum value among neighbors; and

Filter: Remove the colored nodes and iterate on the remaining nodes.

As Listing 4 shows, using Mini-Gunrock, we can easily implement Luby-Jones. With pull-based neighborhood reduction (described in Section V), Mini-Gunrock has also implemented Gunrock graph primitive with different graph operators, such as pull-based PageRank (PR), where instead of distributing each node's PageRank score to all its out-degree neighbor nodes, we pull the partial PageRank score from one node's in-degree neighbor nodes. However, the current set of graph operators in Mini-Gunrock cannot implement triangle counting in Gunrock (Section IX-B).

Listing 4: *Coloring Enactor.*

```
void enact(std::shared_ptr<coloring_problem_t>
     coloring_problem, standard_context_t &context) {
        std::shared_ptr<frontier_t<int> > full_frontier(
             std::make_shared<frontier_t<int> >(context,
             coloring_problem.get()->gslice->num_nodes));
        // Start the iteration with the initial frontier
             that contains all the nodes in the graph.
        init_frontier(coloring_problem, full_frontier);
        int frontier_length = coloring_problem.get()->
             gslice->num_nodes;

        int selector = 0;
        int iteration = 0;
        int *reduced_max = coloring_problem->
             d_reduced_max.data();
        while (frontier_length > 0 && iteration <
             coloring_problem.get()->max_iter) {
            // Neighborhood reduction chooses the max
                 value among one node's all out-degree
                 neighbor nodes and stores it in
                 reduced_max.
            neighborhood_kernel<coloring_problem_t,
                 reduce_max_t, int, mgpu::maximum_t<int>,
                  false >(coloring_problem, full_frontier
                 , full_frontier, reduced_max, std::
                 numeric_limits<int>::min(), iteration,
                 context);
            // Filter compares each node's hash value,
                 assign new color for those nodes whose
                 hash value is larger than the stored
                 reduced_max.
            frontier_length = filter_kernel<
                 coloring_problem_t, coloring_functor_t>(
                 coloring_problem, buffers[selector],
                 buffers[selector^1], iteration, context)
                 ;
            selector ^= 1;
            ++iteration;
            // Parallel compute kernel to assign new
                 hash value to each uncolored node.
            coloring_problem->reset_hashs(context);
        }
    }
```

B. *Mini-Gunrock vs. Gunrock*

This section compares Mini-Gunrock and Gunrock from the graph primitive perspective.

There are two design choices that help Mini-Gunrock achieves comparable performance with a much smaller code size. The first is a more compact design of basic graph data structures such as the frontier, the problem, and the enactor. The second is the use of transform-based primitives on top of these basic data structures. Although the layers of the system are the same as Gunrock—we put graph topology data into a graph class, add per-node/per-edge data into a problem class, define two queues in frontier class for storing the input frontier and the output frontier when doing graph traversal, then write an enactor function and several functors for our graph operators—for each layer the code is more compact and we replace all the hand-written optimization kernels with moderngpu 2.0 primitive calls or general transform calls with simple kernel functions.

The efficient implementation of neighborhood reduction in Mini-Gunrock makes it capable of implementing several new graph primitives. Such examples include maximal

Dataset	Vertices	Edges	Max Degree	Diameter	Type
soc-orkut	3M	212.7M	27,466	9	rs
hollywood-09	1.1M	112.8M	11,467	11	rs
rmat_s21_e48	2.1M	364.2M	213,904	5	gs
rgg_n_24	16.8M	265.1M	40	2622	gm
roadnet_USA	23.9M	577.1M	9	6809	rm

Table I: *Dataset Description Table. Graph types are: r: real-world, g: generated, s: scale-free, and m: mesh-like. All datasets have been converted to undirected graphs. Self-loops and duplicated edges are removed.*

Algorithm	Datasets	MGR Runtime (ms)	GR Runtime (ms)
BFS	soc-orkut	111.4	212.6
	hollywood	37.32	36.5
	kron	107.3	196.9
	roadnet	7548	622.6
	rgg	3473	485.7
PR	soc-orkut	88.28	176
	hollywood	27	27.31
	kron	93.91	176.2
	roadnet	365.9	91.81
	rgg	199.5	181.6

Table II: *Runtime comparison of Mini-Gunrock and Gunrock for BFS and PR. Note that for BFS, we force Gunrock to use the baseline LB strategy with no idempotence or direction-optimizing traversal, and for PR, we compare the runtime for one iteration.*

independent set, weighted label propagation, and several other primitives whose main operator is neighborhood reduction. The flexibility of doing both in-degree and out-degree neighborhood reduction also enables us to implement node-ranking primitives with rank scores distributed along neighbors using either push-style or pull-style. Such examples include PageRank and betweenness centrality, where switching to a pull-style neighborhood reduction could avoid atomic operations and thus improve overall performance. Because Mini-Gunrock and Gunrock follow the same data-centric abstraction, the features that are missing from current Gunrock are mostly on the implementation side, and can be fixed in future Gunrock releases. For example, neighborhood reduction requires an additional segmented reduction within our current load-balanced search implementation in Gunrock; in-degree and out-degree advance/neighborhood reduction requires some interface changes and additional runtime memory to store neighbor lists for both directions.

VIII. Performance Analysis

We summarize the datasets we used in our experiments in Table I.

A. *Performance Summary*

Table II shows the end-to-end runtime comparison for BFS and PR of Mini-Gunrock and Gunrock on five different datasets. For advance, we compare with Gunrock's baseline implementation, which uses the LB strategy and disables direction-optimizing traversal and idempotence since we have observed that: (1) idempotence will cause different per-iteration frontier sizes and make it difficult to compare Mini-Gunrock and Gunrock; (2) direction-optimizing traversal has a wide performance gap between Mini-Gunrock and Gunrock due to the limitations of pull-based traversal implementation in Mini-Gunrock (Section VI-C). We thus believe the baseline LB strategy in Gunrock is the most appropriate comparison to Mini-Gunrock's transform-based implementation, which also uses merge-based load balanced search.

The geomean speedup of Mini-Gunrock over Gunrock for BFS and PR on three scale-free graphs (soc-orkut, hollywood, and kron) are 1.507 and 1.558 separately. This shows that both our transform-based graph operators in Mini-Gunrock (the *lbs_transform*-based advance operator and the *segreduce_transform*-based neighborhood reduction operator) can achieve comparable performance with Gunrock's baseline advance operator on scale-free graphs. Why? In this comparison, Gunrock's LB advance in BFS and PR uses several separate GPU kernels for getting the length of neighbor lists in the input frontier, doing prefix-sum, doing sorted search, and mapping the workload to threads to cooperatively expand neighbor lists. In contrast, Mini-Gunrock's transform-based implementation fuses these kernels into two transforms, significantly reducing data movement and increasing per-iteration performance.

On the other hand, Mini-Gunrock has an order-of-magnitude performance gap vs. Gunrock on BFS on graphs with large diameter and small average degree (roadnet and rgg). First, the transform-based operators have extra kernel launch overheads, so for graphs with large diameters, if the algorithm has a very large search depth, the accumulated overhead will significantly hurt the overall performance. Second, in Gunrock's implementation, graphs with a small average degree and a small size frontier cannot fully utilize the GPU. Thus Gunrock switches from load-balance on the output frontier (which will have several binary searches for neighbor lists visiting within one block) to load-balance on the input frontier. Since the average degree is small, after the switch, Gunrock's implementation will still have near-perfect load balancing and avoid the overhead of binary searches when the number of the small neighbor lists assigned to one block is far beyond the block dimension. However, Mini-Gunrock's transform-based operator can only perform one strategy of load balancing, on the output frontier, and suffers a performance penalty as a result.

B. Per-Iteration Performance Analysis

To compare per-iteration operation performance between Mini-Gunrock and Gunrock, we collect the runtime and MTEPS for each BFS advance operation on two scale-free graphs and two road network graphs. Without losing generality, we only compare the advance operator, which has the same workload and memory access pattern as the neighborhood reduction operator, and ignore the filter operator, which does not expose much irregularity.

Figure 6 shows the per-iteration advance operation MTEPS for both Mini-Gunrock and Gunrock on two scale-free graphs. From the figure we can see that Mini-Gunrock both shows better peak performance, and stays within the range of the peak performance for more iterations, than Gunrock. This demonstrates the efficiency of the transform-based approach.

Figure 7 shows the per-iteration advance operation MTEPS for both Mini-Gunrock and Gunrock on two road network graphs. Neither shows peak performance due to the small size of frontiers. Both figures show Mini-Gunrock's advance consistently performs worse than Gunrock when the frontier size is too small to fully utilize computing resources. The result also means that for small frontiers, both Gunrock and Mini-Gunrock need to try alternative load balancing strategies, such as Gunrock's TWC strategy.

IX. Limitations

The high-level abstraction and high performance of transform-based primitives allows several benefits when Mini-Gunrock's graph operators use them without modification. However, it also has some limitations if we want more flexibility. The two sources of these limitations are: 1) the transform abstraction and 2) the current transform implementation in moderngpu 2.0.

A. Limitations of the Transform Abstraction

Lack of Multi-Frontier Support: As we noted in section IV, moderngpu supports only two types of transforms—unary transform and binary transform—which means that they cannot be applied to several multi-frontier required operators and optimizations in Gunrock. Such optimizations include multi-level priority queues based on multisplit [10] and integrated pull-and-push-based advance with both a sparse-compact current visiting frontier and a dense-bitmap unvisited frontier [8].

However, we expect that expanding the current transform primitive design to support multiple frontiers would enable more potential optimizations and graph operators. One solution to go beyond the unary and binary transform limitation on the number of input and output arrays is to read from/write to additional arrays in global memory from the kernel function in the operator.

Kernel Launching Overhead: Compared to graph operators in Gunrock, graph operators built on top of transform-based primitives have more kernel launching overhead (see Table II), which makes their performance suffer for graph datasets that have a large diameter. Using transform-based primitives also makes the kernel fusion of two graph operators impossible in Mini-Gunrock.

B. Limitations of the Current Transform Implementation

No Binary Transform Support: One graph operator from Gunrock that is difficult to implement in the current moderngpu 2.0's transform-based framework is segmented intersection, which is a binary transform that takes two input arrays and produces one output array. It can use balanced path (a modified version of merge path in moderngpu 2.0) on the block level to produce an output array that contains all intersected items from two input arrays. However, the current implementation of merge path automatically computes the partition indices of the second array to merge as a way to do load balancing for the first array, so it still only takes one input. General support of binary transform primitives is desirable.

Limited flexibility: The current implementation of moderngpu 2.0 automatically sets up an optimal launch setting and workload scheduling/decomposition phase according to a fixed number of items processed per thread. This can often find the most load-balanced and work-efficient workload mapping for existing primitives in moderngpu 2.0. However, customized optimizations such as a dynamic grouping workload mapping strategy [11] (DG) based on a persistent threads programming style are difficult to implement, as the current unary transform will apply a unary operation to every input item exactly once, but DG requires an input item to take charge of a thread group (either a warp or a block) to perform computation on neighbor lists, in which case each input item may process not only one unary operation more than once but also multiple unary operations. The opacity of the transform-based primitive's set-up and scheduling phase also increases the difficulty of implementing more complicated user-defined functions such as those that use shared memory, complex synchronization, and operations such as early exit on all threads within a block. An implementation with user-specific shared-memory and local-memory allocation capability as well as synchronization interfaces for synchronization among user-specified thread groups is the key feature to make the current transform implementation flexible enough for more optimizations in Gunrock.

Fixed Two-Phase Idiom: Moderngpu 2.0 breaks primitives into two distinct phases: 1) find a coarse-grained partitioning

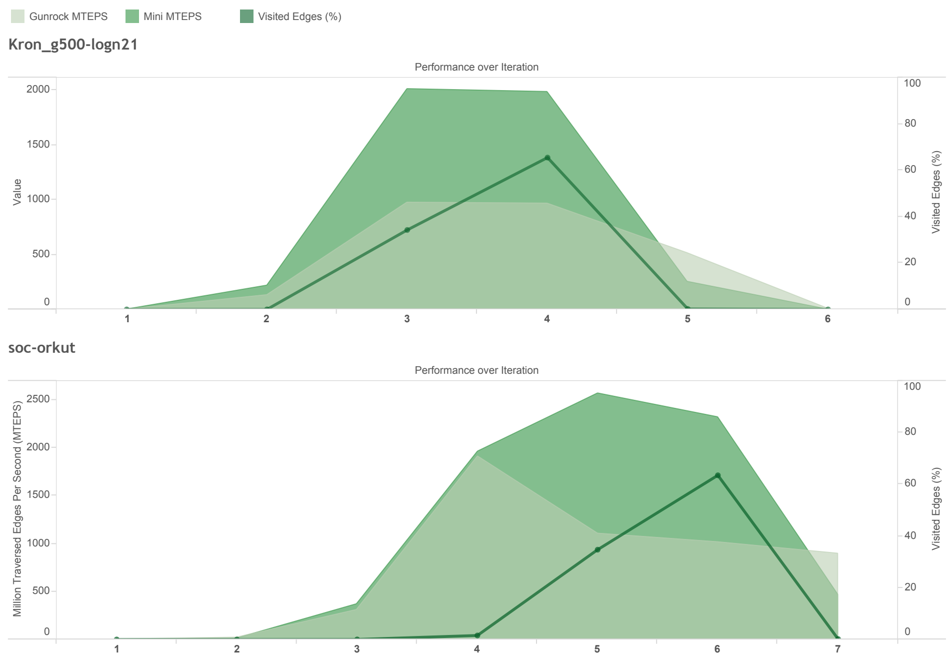

Figure 6: *Area chart that compares per-iteration advance performance between Gunrock and Mini-Gunrock with the percentage of visited edges of that iteration.*

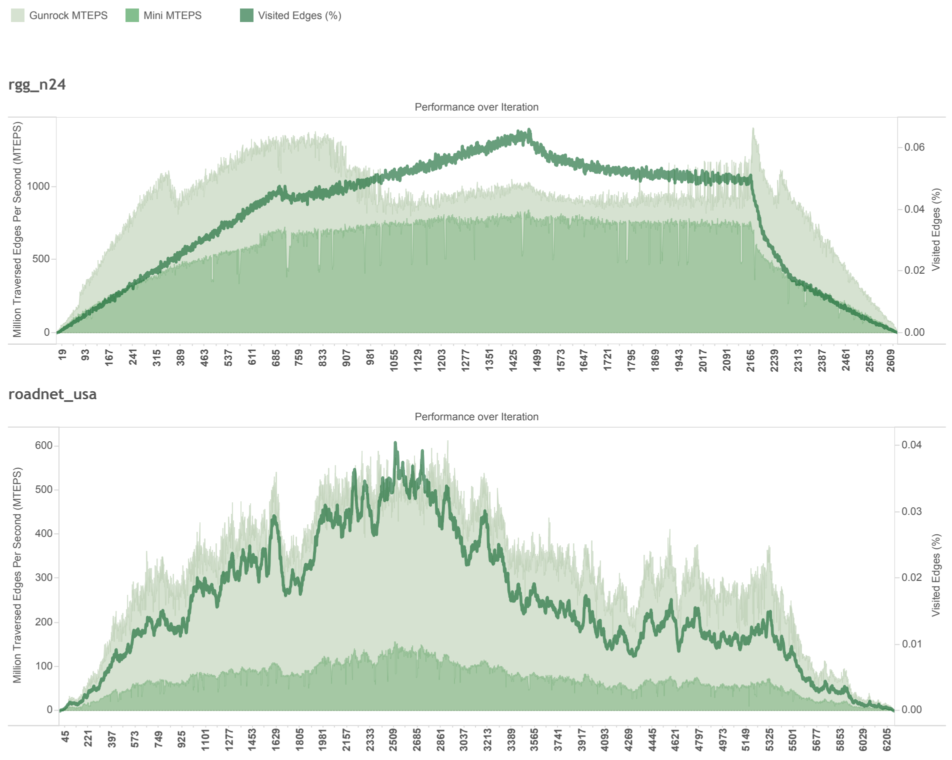

Figure 7: *Area chart that compares per-iteration advance performance between Gunrock and Mini-Gunrock with the percentage of visited edges of that iteration.*

of the problem that exactly load-balances work over each thread, and 2) execute simple, work-efficient, sequential logic that solves the problem. This two-phase idiom always searches for perfect load balance and does not allow user-specified coarse-grained partitioning strategies in the first phase. However, our experiments show that perfect load balance does not necessarily bring the best performance due to the potential overhead it will cause for the second execution phase. Thus a more flexible interface that allows users to try different strategies in both phases will be desirable for achieving better performance for different graph primitives on different datasets.

X. Conclusion

In this paper, we discuss the design and implementation of Mini-Gunrock, a lightweight graph analytics framework on the GPU. It shows the flexibility of the data-centric abstraction by implementing this abstraction with a different implementation. We have also explored different performance tradeoffs, and identified strengths and weaknesses of a transform-based abstraction and implementation. In conclusion, with Mini-Gunrock, we have achieved our three goals of flexibility, simplicity, and comparable performance. It is able to have the same features for graph operators in Gunrock, it has enabled rapid prototyping of both new graph operators and new graph primitives, and it achieves comparable performance with Gunrock.

Acknowledgments

We gratefully acknowledge the support of the DARPA XDATA program (US Army award W911QX-12-C-0059); DARPA STTR awards D14PC00023 and D15PC00010; and NSF awards CCF-1017399, OCI-1032859, and CCF-1629657.

References

[1] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "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*, ser. PPoPP 2016, Mar. 2016.

[2] S. Baxter, "Moderngpu: Patterns and behaviors for GPU computing," 2013–2016, http://moderngpu.github.io/moderngpu.

[3] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, "Gunrock: GPU graph analytics," *CoRR*, vol. abs/1701.01170, no. 1701.01170v1, Jan. 2017.

[4] 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*, ser. HPDC '14, Jun. 2014, pp. 239–252.

[5] J. Zhong and B. He, "Medusa: Simplified graph processing on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, Jun. 2014.

[6] E. Elsen and V. Vaidyanathan, "A vertex-centric CUDA/C++ API for large graph analytics on GPUs using the gather-apply-scatter abstraction," 2013, http://www.github.com/RoyalCaliber/vertexAPI2.

[7] 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 the Workshop on GRAph Data Management Experiences and Systems*, ser. GRADES '14, Jun. 2014, pp. 2:1–2:6.

[8] 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*, ser. SC '12, Nov. 2012, pp. 12:1–12:10.

[9] M. Luby, "A simple parallel algorithm for the maximal independent set problem," in *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, ser. STOC '85. New York, NY, USA: ACM, 1985, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/22145.22146

[10] S. Ashkiani, A. A. Davidson, U. Meyer, and J. D. Owens, "GPU multisplit," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2016, Mar. 2016, pp. 12:1–12:13.

[11] 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*, ser. PPoPP '12, Feb. 2012, pp. 117–128.