Decoupling Schedule, Topology Layout, and Algorithm to Easily Enlarge the Tuning Space of GPU Graph Processing

Shinnung Jeong
Yonsei University
Seoul, Republic of Korea
shin0403@yonsei.ac.kr

Yongwoo Lee
Yonsei University
Seoul, Republic of Korea
dragonrain96@yonsei.ac.kr

Jaeho Lee
Yonsei University
Seoul, Republic of Korea
ejaho0103@yonsei.ac.kr

Heelim Choi
Yonsei University
Seoul, Republic of Korea
heelim@yonsei.ac.kr

Seungbin Song
Yonsei University
Seoul, Republic of Korea
seungbin@yonsei.ac.kr

Jinho Lee
Seoul National University
Seoul, Republic of Korea
leejinho@snu.ac.kr

Youngsok Kim
Yonsei University
Seoul, Republic of Korea
youngsok@yonsei.ac.kr

Hanjun Kim
Yonsei University
Seoul, Republic of Korea
hanjun@yonsei.ac.kr

ABSTRACT

Only with a right schedule and a right topology layout, a graph algorithm can be efficiently processed on GPUs. Existing GPU graph processing frameworks try to find an optimal schedule and topology layout for an algorithm via iterative search, but they fail to find the optimal configuration because their schedules and topology layouts are tightly coupled in their processing models. Moreover, their tightly coupled schedules and topology layouts make it difficult for developers to extend the tuning space. To easily enlarge the tuning space of GPU graph processing, this work proposes a new GPU graph processing abstraction scheme that fully decouples schedules, topology layouts, and algorithms from each other with abstraction interfaces. Moreover, this work proposes GRAssembler, a new GPU graph processing framework that efficiently integrates the decoupled schedule, topology layout, and algorithm without abstraction overhead. Thanks to the efficient decoupling and integration, GRAssembler increases the tuning space from 336 to 4,480 and achieves 30.4% higher performance on geomean average, compared to the state-of-the-art GPU graph processing framework.

CCS CONCEPTS

• **Software and its engineering → Development frameworks and environments**; **Software libraries and repositories**.

KEYWORDS

Graph Processing, Compiler Optimization, GPUs, Auto-tuning

ACM Reference Format:
Shinnung Jeong, Yongwoo Lee, Jaeho Lee, Heelim Choi, Seungbin Song, Jinho Lee, Youngsok Kim, and Hanjun Kim. 2022. Decoupling Schedule, Topology Layout, and Algorithm to Easily Enlarge the Tuning Space of GPU Graph Processing. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '22), October 10–12, 2022, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages.

PACT '22, October 10–12, 2022, Chicago, IL, USA
© 2022 Copyright held by the owner/author(s).
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *International Conference on Parallel Architectures and Compilation Techniques (PACT '22), October 10–12, 2022, Chicago, IL, USA*, .

1 INTRODUCTION

Graph processing is widely used in various fields such as social science, chemistry, and neuroscience [31]. For example, social network services recommend friends by analyzing their social graphs, and search engines rank related web pages via the PageRank graph algorithm. As the size of a graph increases to improve the analysis quality, its processing time increases. Since a graph has a numerous number of vertices and edges, and each vertex is connected to a set of other vertices, graph processing is a highly parallel workload and fits well to graphics processing units (GPUs) that have large parallel core counts. Thus, to reduce the processing time, optimizing graph processing on GPUs is crucial.

Graph processing frameworks take as input a graph algorithm, a topology layout, and a schedule. One must choose a right topology layout and a right schedule for a given graph algorithm as they greatly affect the processing efficiency of the graph algorithm on GPUs. Unfortunately, the existing graph processing frameworks heavily focus on the schedules, and thus overlook the high importance of the topology layouts and narrow the graph processing tuning space. First, the frameworks [3, 5, 20, 23, 28, 34, 36] support only up to two kinds of topology layouts such as COO and CSR. For example, GraphIt [5] provides 7 different schedule optimizations, but supports only one topology layout such as CSR. Their limited coverage on topology layouts limits their tuning spaces, and thus the frameworks cannot fully optimize the graph processing.

Moreover, the limited composability between schedules and topology layouts of the existing frameworks [3, 5, 20, 23, 28, 34, 36] limits the tuning spaces and graph processing performance. To find the optimal topology layout and schedule, the existing frameworks examine a few pre-defined schedules; topology layouts are tightly coupled with the schedules, and thus get determined once the frameworks select a schedule. Although different topology layouts can be optimal for different datasets for a schedule as shown

in Figure 3b, the frameworks support a fixed topology layout for a schedule, thus losing optimization opportunities.

To make things worse, the tightly coupled schedules and topology layouts severely limit the extendability of graph processing space. Ideally, developers should be able to explore all the possible " × # pairs by implementing " different schedules and # independent topology layouts (i.e., " + # implementations). However, the tightly coupled nature of schedules and topology layouts incurs significant development costs as developers must write " × # different implementations. That is, if developers wish to evaluate their new topology layout, they must write " different implementations of the new topology layout, each of which is tightly coupled with one of the " existing schedules. Such significant development costs make it difficult for developers to easily extend the tuning space of GPU graph processing. Therefore, to easily enlarge the tuning space, we need a new framework that achieves high tuning coverage, composability, and extendability with lowers development costs by decoupling schedules and topology layouts.

Aimed at easily enlarging the tuning space of GPU graph processing, this work first proposes **a new graph processing abstraction scheme that fully decouples schedules and topology layouts**. We make a key finding that, by developing extensible and interference-free interfaces, schedules and topology layouts can be decoupled. Our detailed characterization of the three components of graph processing reveals the following properties of the components. First, schedules can be classified into two types depending on their graph data access orderings. Second, we identify two abstract types for topology layouts based on whether both the source and destination vertices are explicitly specified within them. Third, we find that the components of the same type share a few commonly-used operations, allowing us to define abstract interfaces which can be used to implement all the existing schedules and topology layouts. The abstract interfaces fully decouple schedules and topology layouts, and can be used to easily enlarge the tuning space.

This work also prototypes GRAssembler, a new GPU graph processing framework that supports the fully decoupled graph processing abstraction interfaces to easily enlarge the tuning space. GRAssembler consists of tuner, graph builder, and runtime for exploring tuning spaces, building various topology layouts, and executing graph programs using the abstract interfaces, respectively. To reduce the potential performance overheads due to the abstraction interfaces, GRAssembler reduces argument passing and uses function templates instead of function pointers. GRAssembler explores the easily extended tuning space and finds the optimal topology layout and schedule which the existing frameworks cannot derive due to their limited tuning spaces. Furthermore, GRAssembler further expands the tuning space with a new GPU-friendly optimization. By exploiting the abstract interfaces and optimizations, GRAssembler identifies the optimal graph processing model from the enlarged tuning space which fully includes the entire tuning spaces of the existing frameworks.

This work evaluates GRAssembler on nine graph datasets with seven schedules, five topology layouts, and four graph algorithms. For the evaluation, this work employs various GPU-friendly optimizations including direction optimization, active set data structure selection, active set deduplication, and active set ordering. Our detailed evaluation using an NVIDIA RTX 3090 GPU shows that GRAssembler obtains 1.30x and 2.21x geomean speedups over GraphIt [5] and GSwitch [23], the state-of-the-art graph processing frameworks. The large speedup gains clearly demonstrate the high effectiveness of enlarging the GPU graph processing tuning space by the efficient decoupling and integration through the abstract interfaces and GRAssembler's optimizations.

The contributions of this paper are:

- the characterization of schedule and topology layout by data access ordering and explicitness of edge data (§4).
- the interfaces that fully decouple schedules, topology layouts and algorithms, and the abstract processing model that assembles the abstract interfaces (§4).
- the prototype of GRAssembler that supports the newly proposed graph processing abstract interfaces with compiler optimizations (§5).
- the extension of the graph processing tuning space with topology layouts (§5).
- the case study of extending topology layout library that shows the high extensibility of GRAssembler (§6).

2 BACKGROUND

GPU graph processing frameworks [3, 5, 9, 13, 16, 20, 23, 28, 34] take a graph, an algorithm, and tuning options as input, and process the graph according to the algorithm and tuning options. The graph = (+•) is a pair of a set of vertices + and a set of edges . The algorithm describes how to process the graph, and the tuning options describe how the framework should process the algorithm on GPUs. This work categorizes the tuning options into **schedule** and **topology layout** which describe in what order to execute the graph processing tasks on GPUs and how to store the graph topology in memory, respectively.

2.1 Algorithm

An algorithm describes how to process a graph = (+•). Typically, a graph processing iteratively executes four operations: *6OC 4A*, *BD<*, *O??;~* and *B2OCC4A*. The *6OC 4A* operation gathers the data of neighbor edge *4* ∈ of an active vertex *E* ∈ +. The *BD<* operation performs user-defined reduction (e.g., fetch-and-add, compare-and-swap) on the neighbor edge data from the *6OC 4A* operation. The *O??;~* operation updates the value of each vertex to the computation result from *6OC 4A* and *BD<*. The *B2OCC4A* operation marks and adds the updated vertex to the new active vertex set [14].

2.2 Schedule

The schedule describes which GPU thread processes which part of a graph in what order. There are two basic scheduling schemes. Vertex Mapping (VM) scheme simply maps each vertex to each thread, and Edge Mapping (EM) maps each edge to each thread.

The skewed characteristics of real-world graphs incur a severe workload imbalance on the Single Instruction Multi Thread (SIMT) architecture of GPUs [12, 14]. For example, since each vertex has different numbers of edges, VM suffers from the workload imbalance as shown in Figure 1. Although EM minimizes the workload imbalance, the write-back from *BD<* operation causes synchronization

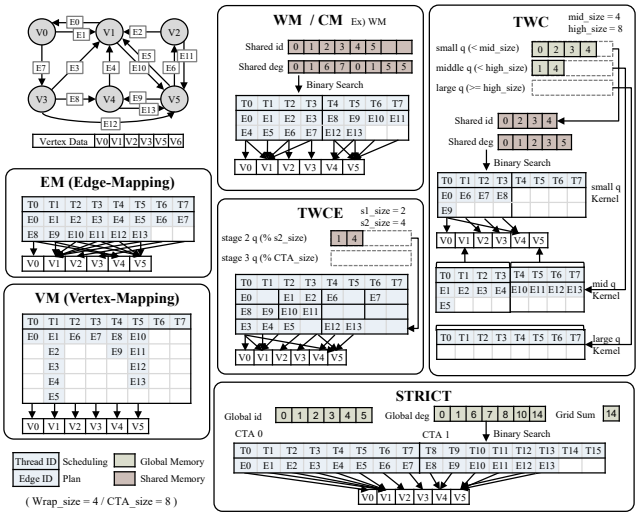


Figure 1: Schedules

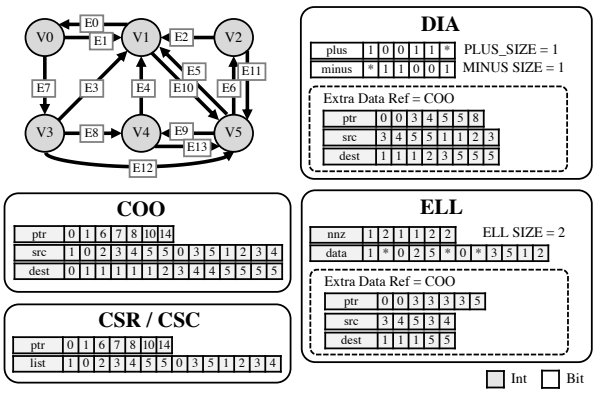


Figure 2: Topology layouts

overhead as shown in Figure 1. To mitigate such problems, various schedules in Figure 1 provide different mappings between the workloads and the resources available within GPUs [5, 10, 23, 25].

Warp Mapping (WM) and Cooperative Thread Array Mapping (CM) share vertices among the threads in a warp and a Cooperative Thread Array (CTA) on shared memory respectively, and distribute their neighbor edges to each thread. Since WM and CM share workloads in a warp or CTA granularity, they reduce synchronization overhead while balancing the workloads.

Thread, Warp and CTA method (TWC) [25] classifies each vertex based on its degree (i.e., the number of edges), and allocates them to low, middle, and high queues on global memory. Then, for the low, middle, and high queues, TWC launches three different kernels that are optimized to thread, warp, and CTA granularity, respectively. Since TWC globally balances its workloads, TWC achieves better workload balance than WM and CM but suffers from additional kernel launch and global memory access overheads.

TWC based on Edge (TWCE) [5] is similar to TWC, but constructs and executes the queues on shared memory within a single kernel. STRICT [10] rearranges active edges on global memory which should be processed at current graph processing iteration and distributes the active edges across CTAs.

2.3 Topology Layout

The topology layout describes how to store a graph topology in GPU memory. Since edges connected to a vertex are very tiny parts of entire edges in a graph, a graph topology is sparse data. Thus, compressing the sparse topology while reducing irregular memory accesses is one of the major challenges in the topology layout design [2, 13, 16, 17, 20, 21, 24, 27], and there are several topology layouts proposed as illustrated in Figure 2.

Coordinate storage format (COO) has a pair of arrays that represent edges as source vertex ids (*BA2*) and destination vertex IDs (*34BC*), and an offset array (*?CA*) that points to a starting index of the *BA2* and *34BC* arrays for each vertex. For example, since *?CA*[1] is 1 and *?CA*[2] is 6 in Figure 2, the edges with destination *E*1 are stored from (*BA2*[1], *3BC*[1]) to (*BA2*[6 − 1], *3BC*[6 − 1]).

Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats remove one of the edge arrays (*34BC* in Figure 2) from the COO format because the index of the offset array (*?CA*) has the information. For example, since *?CA*[2] is 6 in Figure 2, source and destination ids of the edge at *;8BC*[6] are 5 (value of *;8BC*[6]) and 2 (index of *?CA*). CSR/CSC formats use less memory than COO by removing an edge array, but an expensive binary search is required to reconstruct the removed part only from an edge id.

Diagonal (DIA) [2] stores subdiagonal edges in regular arrays such as *?;DB* and *<8=DB*, and non-subdiagonal edges in another layout such as COO format. For example in Figure 2, for a vertex +4, *?;DB*[4] = 1 indicates an upper subdiagonal edge of which the destination is 5, and *<8=DB*[2] = 1 indicates a lower subdiagonal edge of which the destination is 1.

Ellpack (ELL) [2] allocates a constant number (*!!_(/)* of edges in a regular array (*30C0*) for each vertex, and then stores the extra edges in another layout such as COO format. For example in Figure 2, neighbor edges of a vertex +4 are stored from *30C0*[*!!_(/ ∗4*] to *30C0*[*!!_(/ ∗5−1*], because the number of elements for each vertex is constant. The number of edges can be found in *==I*[4] = 2.

3 MOTIVATION

To find the optimal tuning options for a graph algorithm, the existing graph processing frameworks [3, 5, 9, 13, 16, 20, 23, 28, 34] explore their tuning options including schedule and topology layout listed in Table 1. However, the frameworks fail to find the optimal tuning options due to their limited exploration coverage, composability and extendability.

Problem 1: Limited exploration coverage. The existing graph processing frameworks do not consider various schedules and topology layouts in their tuning space exploration. Table 1 shows schedules and topology layouts that each framework considers in its tuning space. Although the frameworks support various schedules while extending the options, they keep leaving one or two options such as CSR and COO for the topology layout. For example, the frameworks do not explore topology layouts that recent

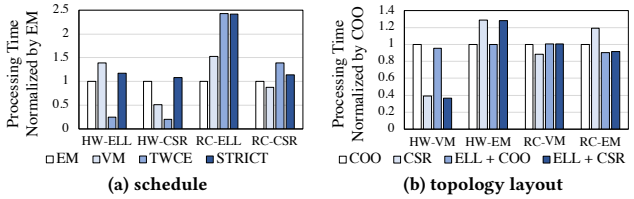


Figure 3: Processing time comparison for different schedules and topology layouts on hollywood-2009 and road-central datasets [11] with PageRank algorithm.

work [16, 27] proposes. Since the topology layout also largely affects the graph processing performance as Figure 3b illustrates, the limited exploration coverage, especially on the topology layout, makes the frameworks miss the further optimization opportunities.

Problem 2: Limited composability. Figure 3 shows how different schedules and topology layouts affect the processing time on two datasets [11]. The best schedule is different depending on the topology layout and dataset, and the best topology layout is different depending on the schedule and dataset. However, the existing graph processing frameworks only decouple the algorithm from their processing model while leaving their schedule and topology layout options tightly coupled in the processing model as Figure 4a shows. Thus, the frameworks can explore only limited schedule and topology layout combinations that are implemented in the frameworks, and may miss the optimal combination if not implemented.

Problem 3: Limited exploration extendability. The tightly coupled schedule and topology layout in the existing frameworks also limit their extendability. Since the schedule and topology layout are tightly coupled, if there exist " schedules and # topology layouts in a framework, the developers should develop " × # options to support all the combinations. For example, to fully support VM and EM schedules and COO, CSR, ELL topology layouts, the framework should develop 6 (2×3) options like Figure 4b. Moreover, if a schedule (or topology layout) developer wants to add a new option, the developer should develop # (or ") options together to support all the combinations. Thus, extending a new option in the existing frameworks requires a huge amount of development cost.

Solution: A new graph processing abstraction model. To solve the problems, a new abstraction model for graph processing is necessary that decouples not only the algorithm but also the schedule and topology layout from the processing model like Figure 6. This work designs the graph processing abstraction model with the design goals such as high coverage, composability, extendability and efficiency. First, the new abstraction model should be general to cover a wide range of schedules and topology layouts (**high coverage**). Second, scheduling graph vertices and edges according to a schedule, accessing topology data from a topology layout and executing an algorithm should be fully decoupled and independently operate in the abstraction model, so all the possible combinations are composable (**high composability**) and adding a new option can operate with the existing options without additional modification (**high extendability**). For example, if there exist " schedules and # topology layouts, the developers only need to develop " + # options to support all the " × # combinations. In the Figure 6 example, developers need to develop only 5 (2 + 3) options to fully support VM and EM schedules and COO, CSR, ELL topology layouts. Finally, the decoupled schedule, topology layout and algorithms should be assembled without a huge abstraction overheads (**high efficiency**), and thus allowing users to achieve the optimal performance by finding the optimal schedule and topology layout.

4 GRAPH PROCESSING ABSTRACTION

This work proposes a new abstract graph processing model. As Figure 5 shows, a graph processing framework iteratively executes an algorithm until its results become saturated. Each iteration, which is called a *super-step*, largely consists of two steps such as *gather step* and *apply step*. In the gather step, the framework gathers incoming data from neighbors of each vertex by iterating its incoming edges. In the apply step, the framework updates each vertex data reflecting the gathered data and the algorithm. After each super-step, the framework analyzes a set of vertices called *active set* that the framework will process in the next super-step.

The gather step consists of four sub-steps such as *edge schedule*, *topology layout access*, *gather* and *sum*. In the edge schedule sub-step, the framework schedules how to access the incoming edges or outgoing edges of the vertices in the active set according to the schedule, and assigns the scheduled edges into GPU worker threads. In the topology layout access sub-step, the framework loads the edge information such as source and destination vertex IDs and its weight from the topology layout. In the gather and sum sub-steps, the framework collects and accumulates the incoming data of the vertices according to the algorithm.

The apply step consists of two sub-steps such as *vertex schedule* and *apply*. In the vertex schedule sub-step, the framework assigns each vertex in the active set to the GPU worker threads. In the apply sub-step, the framework updates each vertex data reflecting its accumulated incoming data.

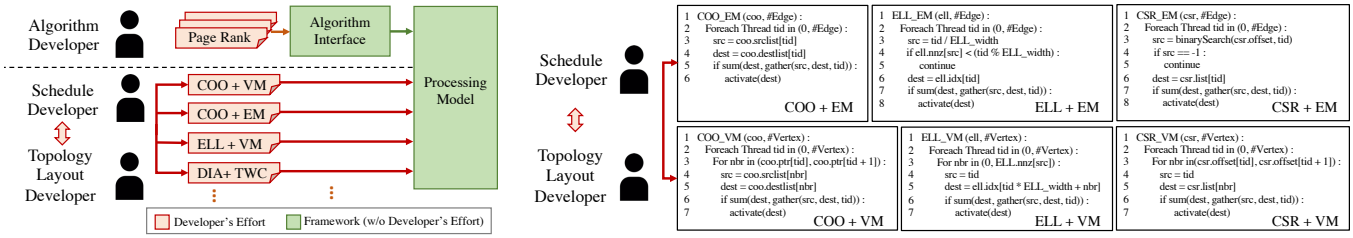
Unlike existing graph processing models [3, 5, 9, 13, 16, 20, 23, 34], edge schedule, vertex schedule and topology layout access are separate sub-steps in the newly proposed abstract graph processing model. In other words, a graph processing framework can schedule edges and vertices, access topology layout and execute the algorithm as independent steps. Thus, if abstraction interfaces are well designed for schedule, topology layout and algorithm, the abstract graph processing model can decouple the schedule, topology layout and algorithm from each other.

4.1 Schedule Abstraction

The schedule basically determines an edge and a vertex for each GPU thread to execute at each edge and vertex schedule sub-step. Since the gather step processes all the imbalanced tasks by manipulating incoming edges of each vertex, the apply sub-step is well balanced without interacting with other vertices or edges. Thus, vertex scheduling is simply mapping a vertex id to a GPU thread, and graph processing schedules [5, 10, 23, 25] focus on how to schedule imbalanced edges in a balanced way. This schedule abstraction also focuses on designing a common interface for all the schedules in §2.2 to schedule the next edge at the edge schedule sub-step.

Table 1: Coverage comparison among existing frameworks about schedules, topology layouts and interfaces

Framework	Schedule	Topology Layout	Algorithm Interface	Schedule Interface	Topology Layout Interface
Gunrock [36]	VM, EM, TWC	COO, CSR	O	X	X
Gswitch [23]	VM, EM, WM, CM, TWC, STRICT	COO, CSR	O	X	X
Graphit [5]	VM, EM, WM, CM, TWC, TWCE, STRICT	CSR	O	X	X
GRAssembler	VM, EM, WM, CM, TWC, TWCE, STRICT	COO, CSR, ELL, DIA, Gshard	O	O	O



(a) Development model of the existing frameworks

(b) Example codes for different schedules and topology layouts

Figure 4: Development model and its example codes used in the existing frameworks [5, 23, 36]. (a) shows the development model of the existing frameworks that only decouples algorithms from its processing model, and (b) shows their example codes about schedules (VM, EM) and topology layouts (COO, CSR, ELL). Since the schedule and topology layout are tightly coupled in the frameworks, the developers should develop M x N combinations to support M schedules and N topology layout.

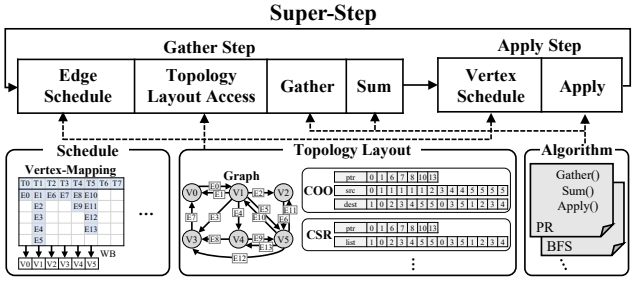


Figure 5: Graph processing model of GRAssembler. The processing model decouples the schedule and topology layout access from the others.

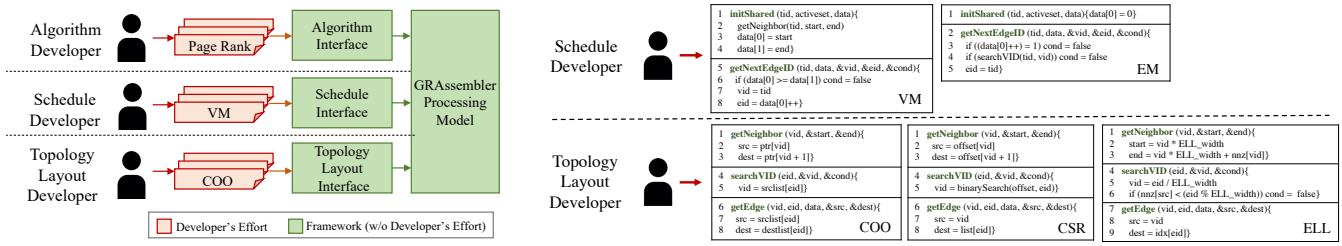
To design a generally applicable schedule interface, this work first analyzes all the schedules in §2.2 and designs the schedule interface like Table 2. Since the schedules schedule edges while balancing workloads across GPU threads, this work abstracts the schedules with two kinds of interfaces such as *edge scheduling* and *load balancing*. Moreover, to increase the coverage, extendability and efficiency, this work introduces several arguments about *active set* and *direction* in the proposed interfaces.

Abstraction for edge scheduling: Since all the schedules schedule the next edge for each GPU thread, the schedule abstraction interface should have a method such as `getNextEdgeID` with *C83* and *483* arguments that returns an edge id (*483*) for a given GPU thread id (*C83*). Moreover, to notify the framework about the end of scheduling or the skipped iteration, `getNextEdgeID` returns a variable (*BCOC4*) about its scheduling state.

Abstraction for load balancing: Schedules such as WM, CM, TWC, TWCE and STRICT share topology information across threads using global or shared memory on a GPU to mitigate the synchronization overheads of EM and the imbalanced schedule of VM. At the beginning of each super-step, the schedules collect neighbor edges of vertices among shared threads within a warp, CTA or kernel depending on the schedule, and equally distribute the collected edges. Since the schedules store the collected and distributed edges on global or shared memory, this schedule abstraction requires interfaces for the distribution such as `initGlobal` and `initShared`, and arguments (*B OA43 D5* and *4364!8BC*) that deliver distribution results to the `getNextEdgeID` method. For example, via the `initGlobal` method, TWC can generate global queues for small, middle and high degree vertices on global memory, and STRICT can collect edges of vertices in the active set and equally distributes edges to the entire CTAs. via the `initShared` method, WM, CM and TWC can generate shared_id and shared_deg as shown in Figure 1, and TWCE can generate the small, middle and high degree queues on shared memory. Here, via `initShared`, VM and EM can also configure the beginning and end indexes of their edge id set, and store the results at *4364!8BC*.

Active set argument: For efficient graph processing, the graph processing framework analyzes a set of vertices called *active set* at each super-step, and only processes vertices in the active set instead of the entire vertices in a graph. To support the active set, this work introduces the active set argument (*02C8E4(4C*) to the `initGlobal` and `initShared` methods. Moreover, to further optimize the graph processing, this work provides various data structures for the active set such as queue, bitmap, bytemap and counter, and allows the graph processing framework to find the optimal active set data structure. Here, *02C8E4(4C* provides a common interface for schedules to manipulate active sets regardless of their data structure.

Direction argument: The two basic schedules are EM and VM. While EM directly maps edges to GPU threads, VM maps incoming



(a) Development model of this work

(b) Example codes for different schedules and topology layouts

Figure 6: Development model and its example codes used in the GRAssembler framework that this work proposes. (a) shows the newly proposed development model that fully decouples the schedule, topology layout and algorithm from its processing model. (b) shows the example codes about schedules (VM, EM) and topology layouts (COO, CSR, ELL) that are used in Figure 4. Since the schedule and topology layout are fully decoupled, the developers need to develop M + N options to support M x N combinations for M schedules and N topology layout.

or outgoing edges of each vertex to GPU threads, so the schedule abstraction requires *direction* information (`8B =`) such as pull or push.

4.2 Topology Layout Abstraction

To design a generally applicable topology layout interface, this work first analyzes all the topology layouts in §2.3 and designs the topology layout interface like Table 2. Since a graph topology is a very sparse data, various compression schemes are used in the topology layouts. To efficiently support the various compression schemes while providing the fundamental topology layout access and the complex topology layout access, this work designs the topology layout interfaces with three kinds of abstraction such as topology data access, fast data access and virtual topology access.

Abstraction for topology data access: Since topology layout is about graph topology, the topology layout interface is basically about returning incoming and outgoing edges for a given vertex and returning source and destination vertices for a given edge. Thus, this work designs two basic topology layout interfaces such as `getNeighbor` and `getEdge`. `getNeighbor` returns incoming or outgoing edge lists of a given vertex id (*E83*) as beginning and end indices (*1468=* and *4=3*) of edge arrays. To indicate the incoming or outgoing edges, `getNeighbor` also takes the direction information (`8B =`) as an input. `getEdge` returns the edge information such as the source and destination vertex ids (*BA2* and *3BC*), and the weight (*F486 C*) of the edge for a given edge id (*483*).

Abstraction for fast data access: To reduce memory usage, some topology layouts like CSR and CSC keep only one of the source or destination vertex ids for edges. To achieve full edge information such as the source and destination ids, a graph processing framework should reconstruct the missing information via an expensive binary search on the *?CA* array. However, since most schedules (all the schedules except EM in §2.2) generate edge lists by invoking `getNeighbor` for a base vertex, the framework already has the missing information in the entire super-step, and can simply use the base vertex instead of executing the binary search. Therefore, to avoid the unnecessary binary search operation, this work divides the `getEdge` operation into two steps such as `searchVID` and `getEdge`. `searchVID` executes the binary search on the *?CA* array and returns the base vertex id (*E83*) for a given edge id (*483*), and `getEdge` receives the base vertex id (*E83*) with its direction (`8B =`) as inputs. If the base vertex id is known in the super-step, the framework can skip the `searchVID` operation with a help of compiler optimization.

Abstraction for virtual topology access: To support different access patterns on a topology layout such as noncontinuous access on edge lists, this topology layout abstraction supports virtual topology access via `initTopology` and `topologyGather`. `initTopology` initializes the virtual topology for each super-step and generates a new active set. `topologyGather` wraps the `gather` method in the algorithm interface for the virtual topology. For example, a blocked layout that has multiple sub-graphs maintains multiple virtual vertices across the sub-graphs for one real vertex. With the `initTopology` method, a topology layout developer can generate virtual vertices and their edge lists, and thus allow multiple virtual vertices in multiple sub-graphs to sequentially access their edges that are not sequentially stored in the real topology. For another example, `initTopology` and `topologyGather` allow the framework to access a continuous edge list with temporary data like Cusha [16]. After generating a virtual topology layout for the temporary data via `initTopology`, the framework can gather the neighbor edge data by continuously accessing the virtual edge list using `topologyGather`.

4.3 Algorithm Abstraction

A graph processing algorithm can consist of four operations such as *60C 4A*, *BD<*, *0??;~* and *B20CC4A* as described in §2.1. This work provides the algorithm interfaces such as `gather`, `sum` and `apply` for the operations except *B20CC4A*. `gather` collects data of an edge (*BA2_83*, *34BC_83*) with its weight value (*F486 C*) for the destination vertex (*34BC_83*), and `sum` integrates the gathered result (*30C0*) for the destination vertex (*34BC_83*). Here, to freely define types of edge weight and gathered data, this work uses template types for the edge weight (*)*) and the gathered results (*)*). `apply` updates the vertex value for a given vertex id (*E83*) with the integrated result from `sum`. To notify the graph processing framework about the updated vertex, `sum` and `apply` return a boolean result. If `sum` and `apply` return a boolean result, the framework executes `filter` for the vertex,

Table 2: GRAssembler Interface Specifications. Here, ET and TD are types for edge weight and temporary data between gather and sum. ActiveSetType is a GRAssembler-tunerable data structure for active sets like a queue, bitmap, bytemap and counter.

Method	Method Description
Schedule Interface	
void **initGlobal** (ActiveSet& activeSet, bool isIn)	Distribute active set to global memory for GPU kernel ∗ activeSet: active set of the current super-step, isIn: PUSH/PULL direction
void **initShared** (int tid, ActiveSet& activeSet, bool isIn, int* sharedBuf, int* edgeList)	Distribute active set to shared memory for warp and CTA ∗ sharedBuf: pointer to queues in the shared memory, edgeList: a edge list for each thread to execute
state **getNextEdgeID** (int tid, bool isIn, int* sharedBuf, int* edgeList, int& vid, int& eid)	Schedule an edge (eid) for a given GPU thread (tid)
Topology layout Interface	
void **getNeighbor** (int vid, bool isIn, int& begin, int& end)	Return beginning and end positions of an edge list for a vertex id (vid) considering direction
void **getEdge** (int eid, int vid, bool isIn, int& src, int& dest, ET& weight)	Return source and destination vertex ids and edge weight for an edge id (eid) considering direction
bool **searchVID** (int eid, bool isIn, int& vid)	Return a vertex id (vid) for a given edge id (eid) considering direction
void **initTopology** (bool isIn, ActiveSet &old, ActiveSet &new)	Initialize virtual topology layout for each super-step
TD **topologyGather** (int src_id, int dest_id, ET weight)	Return pre-gathered data at the virtual topology layout
Algorithm Interface	
void **initVertexValue** (int vid)	Initialize vertex value for a vertex (vid)
TD **gather** (int src_id, int dest_id, ET weight)	Collect information from an edge (src_id, dest_id) for a destination vertex (dest_id)
bool **sum** (int dest_id, TD data)	Integrate gathered data for a destination vertex (dest_id)
bool **apply** (int vid)	Update a vertex value with the integrated information for a vertex (vid)
bool **filter** (int vid)	Filter a vertex (vid) from the active set if not updated
bool **checkDirection** (int numOfVertices, int numOfActiveSet)	Decide the direction of the super-step among pull and push
ET **getNewGlobalThreshold** (ET oldThreshold)	Return a new threshold for the edge weight

and adds the filtered results into the active set. In addition to the operations, this work provides `initVertexValue` that initializes the vertex value at the beginning of the graph processing.

Removing redundant operations in *B2OCC4A*: The *B2OCC4A* operation consists of activation, filtering and fan-out edge processing operation. By checking return values of the *60C 4A* and *O??;~* functions, this work can embed the activation operation into the graph processing framework and remove redundant activation execution in *60C 4A*, *O??;~* and *B2OCC4A*. Moreover, since the fan-out edge processing at the current iteration and the gathering operation at the next iteration have the same semantics, this work makes the *60C 4A* function fan-out the updated values at the next super-step iteration. Thus, only the filter function is necessary.

Abstraction for flexible execution: Some algorithms like BFS and Delta-SSSP dynamically switch their processing direction between pull and push, or update a vertex value only if the value is larger than dynamically changing threshold. To support the flexible execution, this work provides two methods such as `checkDirection` and `getNewGlobalThreshold`. `checkDirection` allows an algorithm to dynamically decide the processing direction based on the active set size at each super-step. `getNewGlobalThreshold` an algorithm to change the threshold value from the old value.

4.4 Assembling Abstract Interfaces

The newly proposed abstract graph processing model assembles the newly proposed schedule, topology layout and algorithm interfaces, and processes a graph. As Figure 6a illustrates, this work allows the schedule, topology layout and algorithm developers to separately and independently implement their schedule, topology layout and algorithm using the interfaces in Table 2. Using a given schedule, topology layout and algorithm via the interfaces, a super-step iteration of the abstract graph processing model presented in Algorithm 1 processes a given graph with an active set. The graph processing model consists of the gather step (Line 2 to 25) and the apply step (Line 26 to 30). To handle the topology layouts like DIA and ELL which require an extra topology layout, the graph processing model iterates over the multiple topology layouts (Line 2). The edge process first initializes topology layout and the global memory (Line 3 to 4), and then launches processing GPU kernels. The GPU threads collaboratively initialize shared memory only once per kernel invocation (Line 7). After initializing global and shared memory, the model executes edge schedule, topology layout access, gather and apply sub-steps in the gather step by iterating over the while loop. Interacting with schedule, the model receives an edge id (Line 10), and decides whether to terminate or skip the iteration. Interacting with topology layout, the model achieves source and destination vertex id and weight value for the edge (Line 16). By checking if the source vertex is active, the model skips the iteration if not (Line 17). Interacting with algorithm, the model executes `gather` and `sum` operations (Line 20). If the `sum` operation updates the destination vertex value, the thread allocates the destination vertex in the output active set (Line 21). After finishing the gather step, the model launches GPU kernel for the apply step, executes the `apply` operation, and allocates the destination vertex in the output active set if the vertex is updated (Line 26 to 30).

5 GRAssembler FRAMEWORK

Figure 7 shows the overall structure of GRAssembler that consists of a tuner, a graph builder and a runtime. The GRAssembler tuner searches for optimal combination of tuning options and evaluates the combination with the GRAssembler runtime. The GRAssembler graph builder converts the input graph data in a raw format to the selected topology layout. The GRAssembler runtime executes the graph processing interacting with the GRAssembler library that the schedule, topology layout and algorithm developers develop.

GRAssembler tuner iteratively searches the optimal tuning options such as schedules and topology layouts for a given algorithm and a graph data. The tuner consists of *manager, topology layout selector, scheduling selector*, and *compiler*. The *manager* determines tunable tuning options for an algorithm. For example, the

Algorithm 1: Super-step of the Abstract Processing Model

```
Input: Sche : Schedule
       Layouts : Topology layouts
       Alg : Algorithm
       ASet_in : Input Active Set
Output: ASet_out : Output Active Set
Function Super-Step (Sche, Alg, Layouts, ASet_in, ASet_out ) :
    foreach Layout ∈ Layouts do
        Layout.initTopology (...)
        Sche.initGlobal (...)
        foreach kernel ∈ Sche.KernelSet do
            foreach thread ∈ kernel do
                Sche.initShared (...)
                while true do
                    // Edge Schedule Step
                    (cond, eid, vid) ← Sche.getNextEdgeID (thread, ...)
                    if cond = terminate then
                        break
                    else if cond = skip then
                        continue
                    // Topology layout Access Step
                    (src, dest, weight) ← Layout.getEdge (eid, vid ...)
                    if Aset_in.check(src) == false then
                        continue
                    data ← Alg.gather (src, dest, weight)
                    if Alg.sum (dest, data) then
                        ASet_out.add(dest)
                end
            end
        end
    end
    foreach thread ∈ vertexKernel do
        vid ← thread
        if Alg.apply (vid) then
            ASet_out.add(vid)
    end
end
```

BFS algorithm dynamically changes the direction (push/pull) with `checkDirection` depending on the number of active vertices. The manager notices the presence of `checkDirection` in the algorithm, generates two sets of tuning options while opting out the direction option from the tunable tuning options, and then separately explores tuning options for each direction. Among the tunable tuning options, the *topology layout selector* chooses tuning options about topology layouts, and the *scheduling selector* chooses tuning options about schedules including active set type and direction of the graph processing. Finally, the *compiler* generates and optimizes the assembled graph processing program for the tuning options from the given algorithm.

GRAssembler graph builder generates a topology layout from an input raw graph for a given topology layout option. Here, the *GRAssembler graph builder* partitions the graph into subgraphs and merges subgraphs into a blocked graph to improve its locality.

GRAssembler runtime executes the assembled graph processing program according to the super-step described in Algorithm 1. The GRAssembler runtime initializes the vertex values and active set data structure, executes the super-step algorithm in Algorithm 1, executes `checkDirection` and `getNewGlobalThreshold` to control the next super-step iteration. If the output active set is empty, the runtime terminates the execution.

GRAssembler library consists of separate implementations of schedules, topology layouts, and algorithms.

5.1 Compiler Optimization

Useless interface elimination: While the proposed abstract interfaces cover all the schedules, topology layouts and algorithms in §2, some of the interfaces are useless for some combination of schedule, topology layout and algorithm. If the implementer leaves the function body in Table 2 empty, GRAssembler analyzes its emptiness by checking if its function body has only a terminator instruction, and eliminates its callsites. For example, the VM schedule does not use `initGlobal`, the COO topology layout does not use `searchVID`, and the BFS algorithm does not use `apply`. To reduce the graph processing latency, the GRAssembler compiler analyzes library functions, and eliminates the useless function calls.

Atomic operation elimination on vertex value: Since graph processing concurrently loads multiple edges and updates a vertex value, a graph processing framework needs to use atomic operation in `sum` for its correct processing. However, if each vertex value is updated by only a thread, the atomic operation is not necessary. GRAssembler conservatively removes the synchronization if all the following three constraints are satisfied. First, the `initTopology` method does not access its active set argument. This constraint is necessary to avoid accessing aliased vertices or edges. Second, the `getNextEdgeID` method maps thread ID to vertex ID. This constraint limits only one thread to access a vertex. Third, the direction of the super-step is PULL. This makes only the mapped thread write the vertex. For example, the COO topology layout using the PULL direction with the VM schedule does not make multiple GPU threads update a single vertex, thus allowing to use non-atomic operations. The GRAssembler compiler analyzes the library to find removable synchronization, and transforms the atomic operations to non-atomic operations only for legal cases.

5.2 Tuning Space of GRAssembler

Table 3 illustrates the tuning options of GRAssembler and the existing frameworks [5, 23, 34, 36]. Compared to the existing frameworks, GRAssembler supports the largest number of schedules and topology layouts, and newly provides topology layout autotuning and CTA size optimization. Thus, while GraphIt [5] supports 336 tuning combinations which was the largest number before this work, **GRAssembler supports 4480 combinations for tuning options**. Considering that some options such as CTA Size is numeric, and this work counts the number of the numeric tuning options as two, the actual number of tuning combination is much larger than 4480. Followings are the tuning options used in GRAssembler in addition to the schedules and topology layouts.

CTA size affects the ratio of active warps. A memory-intensive program can hide its GPU memory latency by controlling the CTA size [8]. This tuning option is newly introduced in this work.

Blocking is the policy to improve locality of graph with tiled graph. Gluon [9] suggests diverse partitioning policies depending on a traverse direction (axis and dimension) and standard (edge or vertex) and we integrate the partitioning policies to GRAssembler.

Active set data structure is a data structure for the active vertex set. This paper supports a queue, bitmap, bytemap and counter for the active set data structure. The queue keeps active vertex ids, and the map keeps activation as boolean value. The counter, which is newly proposed in this paper, only keeps the size of the active set

Table 3: Available tuning space in state-of-the-art GPU graph processing framework

	GRAssembler	**GraphIt**[5]	**Gswitch**[23]	**Gunrock**[36]	**SEP-Graph**[34]
Topology layout Auto-tuning	O	X	X	X	X
Topology layout	COO, CSR, ELL, DIA, Gshard	CSR	COO, CSR	COO, CSR	COO, CSR
CTA Size Optimization	O	X	X	X	X
Blocking	O	O	X	X	X
Schedule	VM, EM, WM, CM, TWC, TWCE, STRICT	VM, EM, WM, CM, TWC, TWCE, STRICT	WM, CM, TWC, STRICT	VM, EM, TWC,	CM
Active Set Data Structure	Queue, BitMap, ByteMap, Counter	Queue, Bitmap, ByteMap	Queue, Bitmap	Queue, Bitmap	Queue
Direction Optimization	O	O	O	O	O
Active Set Deduplication	O	O	X	O	O
Active Set Ordering	O	O	O	O	O
Number of Available Options	**4480**	**336**	**68**	**96**	**64**

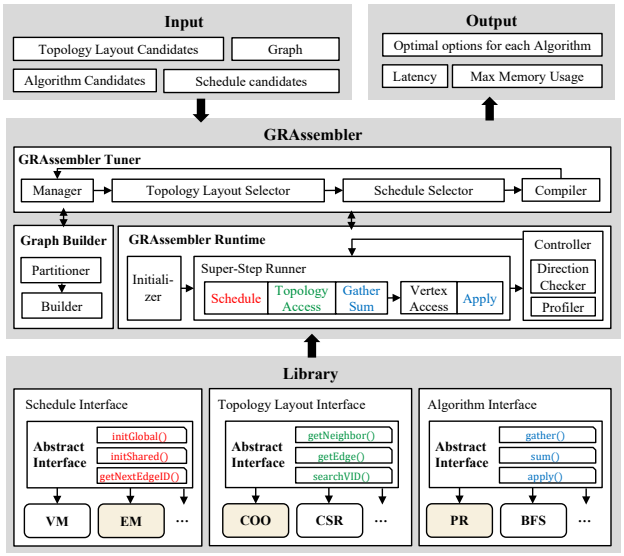


Figure 7: The overview of GRAssembler

by increasing its value by one for $ASet_{out}$.add(vid). The counter is effective if the active set is only used for checking emptiness.

Direction determines whether to access the neighbor edge list based on the source (push) or based on the destination (pull) [3].

Active set deduplication reduces redundant computation [5]. The graph processing can activate the same vertex multiple times. Deduplicating the duplicated active vertices removes redundant computation by paying deduplication costs. If the duplication affects the correctness, this option is mandatory.

Active set ordering determines whether to process an active vertex at the next iteration or later by the global threshold. For example, Delta-SSSP can benefit from this option [15, 26].

6 EVALUATION

This section evaluates the performance of GRAssembler with four algorithms on nine graphs comparing with the existing graph frameworks [5, 23], and shows its extendability with the two case study examples. The evaluation uses NVIDIA GeForce RTX 3090 that has 10,496 CUDA cores, 82 streaming multiprocessors, 6MB L2 cache and 24GB memory for the GPU, and Intel(R) Core(TM) i7-8700 for the host CPU. This evaluation uses the same nine graphs used in

Table 4: The optimal options of Figure 8

Alg	Dataset	Topology	Schedule	Direction	Active set	Thread #
PR	LJ, HW	Block-COO	EM	Push	Counter	512
	OK, TW	Block-COO	EM	Push	Counter	512
	RC	Block-COO	EM	Push	Counter	1024
	RN, RU	CSR	VM	Pull	Counter	512
	IC	CSR	TWC	Push	Counter	1024
	UK	CSR	TWCE	Push	Counter	512
BFS	RC, RU	CSR	TWCE	Push	Queue	512
	RN	CSR	VM	Push	Queue	512
	LJ, OK	ELL + CSR	VM	Pull	RQnB	512
		ELL + CSR	TWCE	Push	Queue	512
	IC	CSR	VM	Pull	QnB	512
		CSR	TWCE	Push	Queue	512
	TW	CSR	CM	Pull	RQnB	512
		CSR	TWCE	Push	Queue	512
	UK	CSR	TWCE	Pull	RQnB	512
		CSR	TWCE	Push	Queue	512
	HW	CSR	CM	Pull	RQnB	512
		CSR	STRICT	Push	Queue	512
CC	HW, RC, LJ,	Block-COO	EM	Push	Counter	512
	TW, RU, OK	Block-COO	EM	Push	Counter	512
	RN	CSR	VM	Pull	Counter	512
	UK	CSR	TWCE	Pull	Counter	512
	IC	ELL + COO	EM	Pull	Counter	512
DS	RN	COO	TWCE	Push	Queue	512
	HW, RC	CSR	TWCE	Push	Queue	512
	LJ, OK	CSR	TWCE	Push	Queue	512
	TW, RU, UK	CSR	TWCE	Push	Queue	512
	IC	CSR	CM	Push	QnB	1024

GraphIt [5] and Gunrock [36] such as soc-orkut (OK) [29], uk-2005 (UK) [29], soc-twitter-2010 (TW) [29], soc-LiveJournal (LJ) [11], indochina-2004 (IC) [11], hollywood-2009 (HW) [11], roadNetCA (RN) [11], road_usa (RU) [?], and road_central (RC) [11]. This evaluation uses the four different algorithms such as PageRank (PR) [6], Connected Components (CC) [32], Breadth-First-Search (BFS) [1] and Delta-SSSP(DS) [26]. Each algorithm has different operation features. PR and CC always update the vertex value at each superstep. BFS can switch the processing direction according to the ratio of the active set, thus evaluating the impact of `checkDirection`. Delta-SSSP can use a dynamically changing delta by defining the global threshold in the algorithm interface. For comparison, this work chooses GraphIt [5] and Gswitch [23] that explore various graph processing options such as direction, schedule, active set data structure and active vertex ordering.

Figure 8: Performance comparison with the state-of-the-art graph processing frameworks such as GraphIt and Gswitch. Each graph shows speedups over Gwsitch. Four algorithms on nine graphs are used on NVIDIA GeForce RTX 3090 GPU.

6.1 Overall Performance

Figure 8 shows that GRAssembler improves performance for most applications. GRAssembler achieves 2.21x and 1.30x speedups compared to Gswitch and GraphIt, respectively. In detail, GRAssembler achieves 2.33x, 1.84x, 1.66x, and 3.38x speedups over Gswitch, and 1.52x, 1.77x, 0.99x, and 1.07x speedups over GraphIt for PR, CC, BFS, and DS, respectively.

By using function templates and fewer function arguments, GRAssembler reduces the performance overhead coming from assembling the abstract interfaces. First, this work reduces the number of function arguments by using global symbols. Second, GRAssembler uses class templates and function templates instead of function pointers. Passing device function pointer to GPU kernel especially complicates the compiler analysis, hence preventing useful compiler optimizations such as inlining. Using function template binds the polymorphic function call at compile-time, enhancing the compiler optimizations.

In detail, GRAssembler signi cantly outperforms all the other frameworks on PR and CC, thanks to the CTA size optimization. For coalesced memory accesses, a GPU performs better for a speci c thread-size and block-size. For example, using 1024 threads is better than 512 threads for IC and RC datasets for PR. The CTA size optimization is e ective especially for the EM schedule due to its coalesced memory accesses to edge data. For example, for the PR algorithm and the UK dataset, the CTA size optimization improves the performance of EM and Push with COO, CSR, ELL with COO, and Block-COO by 71%, 8%, 73%, and 68%, respectively.

GRAssembler also outperforms the other frameworks on the CC algorithm, by nding di erent optimal solutions for topology layout such as CSR and Block-COO depending on its dataset. Moreover, the CC algorithm repeats the super-step until there is no update. Since GraphIt only explores two active set options such as a bitmap and a bytemap, it should check if every vertex is active or not, and also check if there exists an active vertex in the active set. On the other hand, since GRAssembler supports a counter as an active set, GRAssembler does not need to check all the vertices to terminate the super-step.

GRAssembler shows di erent optimal options for BFS depending on its dataset as illustrated in Table 4. Unlike the other algorithms, for BFS, GRAssembler changes the direction from pull to push during the execution, so the optimal tuning options consists of two sets of directions, schedules, and active set data structures on LJ, OK, IC, TW, UK and HW datasets. GRAssembler su ers from higher overheads than the other frameworks for BFS due to its complex optimal tuning option and simple computation algorithm. Since the apply function of BFS is empty, the abstraction overheads relatively largely a ect the overall execution time for BFS. Thus, GRAssembler that abstracts graph processing more than the other frameworks su ers from higher abstraction overheads, and shows slower performance than them for BFS. Here, GRAssembler dynamically changes the schedule and active set structures when the return value of checkDirection function is changed. This work can improve the performance of graph processing programs like BFS, by adopting intensive online tuning [18, 23] that supports more than two dynamically changing options.

Finally, GRAssembler shows better results than the other frameworks on DS. Table 4 shows tuning options used for the best performance. QnB means using a queue for the input active set structure and using a map for the output active set structure, and RQnB means using a reverse queue for the input active set structure.

6.2 Tuning Performance and Cost

GRAssembler nds the optimal tuning options that have the shortest execution time. Compared to the second optimal solution, GRAssembler achieves 2.04<4B maximum and 1.16<4B geomean speedups. 52.78% of the second optimal tuning results adopt topology layouts that are di erent from the optimal ones. This shows that trying di erent tuning options is required to nd the optimal solution. The execution times of the optimally tuned applications in the evaluation section range from 0.19 milliseconds to 1.17 seconds (33.75 milliseconds on average), which are 1.36x to 450.33x speedups (21.48x geomean speedup, 401.63 milliseconds on average and up to 2.7 seconds reduction) compared to the worst cases. Ÿ6.5 shows detailed performance result of di erent tuning options as a case study.

GRAssembler takes up to 2 hours for tuning each application and dataset while GraphIt takes 10 minutes. It is because GRAssembler explores 14 times more candidates than GraphIt. The tuning time gap is less than 14 times because GRAssembler reduces the tuning overheads by terminating its candidates without full execution if

(a) Block-COO (b) Cusha

Figure 9: Extending topology layout library. (a) shows speedup of Block-COO over COO using GRAssembler interface. (b) shows speedup of GShard topology layout over original GShard implementation of Cusha [16].

its latency exceeds the optimal one. Although the tuning cost is relatively large compared to its execution time, tuning the graph processing options is still important because a graph application is executed multiple times at runtime. If the auto-tuner can exploit the previous tuning results and prune ine cient tuning candidates, GRAssembler can reduce the tuning cost.

6.3 Line of Code Analysis

GRAssembler separates schedule and topology layout to the separated interfaces. The total lines of codes of schedule implementation are 1540 Lines (VM: 82, EM: 84, CM: 185, WM: 141, TWC: 292, TWCE: 238, STRICT: 395, Interface Utilities: 123). The total lines of topology layout implementation are 644 lines (COO: 114, ELL: 118, CSR: 131, Gshard: 166, Interface Utilities: 115). GraphIt implements its processing model with 1044 lines of codes, and the lines of codes of the device functions that access and operate on the graph data take 64% (669 Lines). To add a new topology layout, GraphIt requires a deep understanding of its processing model code and modi cation on 64% of processing model codes.

6.4 Abstraction Overhead

To shows the cost of the GRAssembler interface abstraction and integration, this work compares the execution times of GRAssembler and GraphIt with the same tuning options. The comparison uses the tuning options that are used for the optimal GraphIt execution. The result shows that GRAssembler has 21.1% longer geomean processing time than GraphIt. More speci cally, PR, CC, BFS and DS take 15.6%, 15.2%, 34.0%, and 20.0% longer processing time on geomean, respectively. The evaluation results show that GRAssembler su ers from the higher overhead for BFS than other applications due to its complex optimal tuning options and simple computation algorithm.

6.5 Case Study 1 : Extendability

This section demonstrates the extendability of GRAssembler by describing implementations of two example topology layouts, and shows that it exhibits equal or better performance compared to the original implementations.

Block-COO: Blocking is a common technique for improving the locality of the accesses to vertex values. As a case study, we described how a blocked version of COO has been implemented with the abstractions of GRAssembler. When the neighbor edges of a vertex are blocked into multiple edge lists, a virtual vertex substitutes the vertex of an edge. Using the concept of virtual vertex, `getNeighbor`, `getEdge`, and `searchVID` method are implemented for a blocked graph. The `getNeighbor` method is written such that it returns a sublist of the blocked edge list, and the edges in the sublist are only connected to the virtual vertex assigned to the sublist. Figure 9a shows that the Block-COO implementation successfully achieves speedsup over COO up to 4.53x using Push and EM without CTA optimization. This example shows that the proposed interface can seamlessly integrate a blocked graph implementation. On the other hand, GraphIt [5] allows blocking optimization but cannot provide a block-wise connected neighbor list for a vertex, limiting the blocking optimization to the EM scheduling.

Figure 10: Performance comparison among di erent tuning options for PR with LJ. Four topology layouts (COO, CSR, ELL+COO, Block-COO), two schedules (EM, TWCE), and two directions (Push, Pull) are compared. The baseline tuning option is (Block-COO, TWCE, Push).

Cusha: Cusha [16] proposes a unique topology layout, Gshard, which reorders and blocks edge data. The implementation of Gshard on GRAssembler can be done in a similar way to Block-COO. The separated `gather` and `sum` operation for the processing model in Cusha is performed in `apply` step.

GRAssembler interface supports `topologyGather` that allows Cusha implementation on GRAssembler. GRAssembler compiler checks the presence of `topologyGather`, substitutes `gather` operation in processing model by `topologyGather`, and integrates the original `gather` operation to `initTopology` implementation. Figure 9b shows that the implementation of GRAssembler performs equal to or better than the original Cusha implementation. One reason is that GRAssembler implementation does not use asynchronous update that may incur unnecessary synchronization between atomic updates, improving the processing time on TW, IC, and HW dataset.

6.6 Case Study 2 : Performance Impact of Tuning Options

Figure 10 compares the performance di erent tuning options. Four topology layouts (COO, CSR, ELL+COO, Block-COO), two schedules (EM, TWCE), and two directions (Push, Pull) are used as tuning options in the comparison. Figure 10 shows a sorted performance across di erent tuning options. The comparison results show three notable features in graph processing.

First, extending its tuning space is crucial to achieve better performance. The Block-COO topology layout is used for the best performed tuning option, showing almost two times better performance than the second-performed tuning option (CSR, TWCE and Pull). Since the existing frameworks do not support Block-COO, they cannot nd this tuning option.

Second, the tuning algorithm should consider the synergistic performance e ect of tuning options. The Block-COO topology layout is used not only for the best, but also for the worst performed tuning option, showing 4.7 times performance gap. The processing time can be remarkably changed depending on how to combine topology layout, schedule, and other options, and thus ignoring one tuning option may cause a great tuning opportunity miss.

Third, various tuning options beyond schedule and topology layout are also important. In the results, the direction largely a ects the performance. GRAssembler integrates various tuning options including the direction, and achieves better performance results.

7 RELATED WORK

Con gurable graph processing frameworks . Existing work [5, 23, 36] on con gurable graph processing frameworks separates graph algorithms and graph processing models to support various algorithms and provide con gurable graph processing optimizations to control the scheduling plan of graph processing model. Gswitch [23] de nes ve con gurations: direction, active-set data-structure, load balance (scheduling plan of GRAssembler), stepping, and kernel fusion, considering the generality and signi cance of graph program. Gunrock [36] additionally supports an active set deduplication tuning and adds VM, EM, TWC scheduling plans to the load balancing. GraphIt [5] supports vertex blocking tuning knob and proposes TWCE scheduling plan. Existing graph processing frameworks also support auto-tuning of the proposed options. However, existing graph processing frameworks do not consider topology layout as an tuning space. GRAssembler supports the tuning spaces of the existing frameworks and extends the tuning spaces to support topology layout.

Graph processing models . Prior work [14, 22] introduces graph processing models decoupling algorithms from graph program. Pregel [22] adopts Bulk Synchronous Parallel (BSP) model [33] and proposes computation and communication models. Gunrock [36] extends the BSP model focusing on the frontier and regroups graph processing models as Advance, Filter, Compute. On the other hand, PowerGraph [14] introduces Gather-Apply-Scatter (GAS) model to separate algorithms and graph processing. An algorithm implements gather operation for edge access computation, apply operation for vertex data update, and scatter operation for vertex activation. GraphQ [35] and Wonderland [38] also propose a graph processing model for big graphs that do not t in memory and targets out-of-core graph processing. These graph processing models only separate algorithms and graph processing, but GRAssembler separates topology layout, schedules, and algorithms.

Auto-tuner for topology layout. Selecting topology layout for graph algorithms impacts the overall performance of the graph processing. Thorsten et al. [4] provide comprehensive study of topology layout and show that topology layout in uences the performance of graph processing algorithms, but they miss scheduling plans and other optimizations. Existing work on topology layout auto-tuner of sparse matrix [19, 30, 37] proposes auto-tuning methods to optimize computation on sparse matrix such as sparse matrix-vector and matrix-matrix multiplication.

8 CONCLUSION

Our insight about decoupling schedule and topology layout from graph processing eases to enlarge graph tuning space. Based on the insight, this work proposes graph processing abstraction of schedule, topology layout, and algorithm based on characterization of graph program. Then, we propose GRAssembler that implements our processing model and abstract interfaces with various graph processing optimizations.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. We also thank the CoreLab members for their support and feedback during this work. This work is supported by IITP-2020-0-01847, IITP-2020-0-01361, IITP-2021-0-00853, and IITP-2022-0-00050 through the Institute of Information and Communication Technology Planning and Evaluation (IITP) funded by the Ministry of Science and ICT. This work is also supported by Samsung Electronics. (Corresponding author: Hanjun Kim)

REFERENCES

[1] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing Breadth-First Search. In SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. 1 10. https://doi.org/10.1109/SC.2012.50

[2] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. https://doi.org/10.1145/1654059.1654078

[3] Maciej Besta, Micha Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoe er. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '17). 93 104.

[4] Thorsten Blaÿ and Michael Philippsen. 2019. Which Graph Representation to Select for Static Graph-Algorithms on a CUDA-Capable GPU. In Proceedings of the 12th Workshop on General Purpose Processing Using GPUs (Providence, RI, USA) (GPGPU '19). Association for Computing Machinery, New York, NY, USA, 22 31. https://doi.org/10.1145/3300053.3319416

[5] Ajay Brahmakshatriya, Yunming Zhang, Changwan Hong, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2021. Compiling Graph Applications for GPU s with GraphIt. In 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 248 261. https://doi.org/10.1109/CGO51591.2021.9370321

[6] Sergey Brin and Lawrence Page. 2012. Reprint of: The anatomy of a large-scale hypertextual web search engine. Computer networks 56, 18 (2012), 3825 3833.

[7]]paper:dataset:dimacs A. Goldberg C. Demetrescu and D. Johnson. [n. d.]. DIMACS implementation challenge - shortest paths. http://www.dis.uniroma1.it/challenge9/

[8] Thanh Tuan Dao and Jaejin Lee. 2018. An Auto-Tuner for OpenCL Work-Group Size on GPUs. IEEE Transactions on Parallel and Distributed Systems 29, 2 (2018), 283 296. https://doi.org/10.1109/TPDS.2017.2755657

[9] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A Communication-Optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 752–768. https://doi.org/10.1145/3192366.3192404

[10] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. 2014. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 349–359. https://doi.org/10.1109/IPDPS.2014.45

[11] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[12] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. 1999. On Power-Law Relationships of the Internet Topology. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (Cambridge, Massachusetts, USA) *(SIGCOMM '99)*. Association for Computing Machinery, New York, NY, USA, 251–262. https://doi.org/10.1145/316188.316229

[13] Zhisong Fu, Michael Personick, and Bryan Thompson. 2014. 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* (Snowbird, UT, USA) *(GRADES'14)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2621934.2621936

[14] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[15] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. 2011. Ordered vs. Unordered: A Comparison of Parallelism and Work-Efficiency in Irregular Algorithms. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. 3–12.

[16] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: Vertex-Centric Graph Processing on GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing* (Vancouver, BC, Canada) *(HPDC '14)*. Association for Computing Machinery, New York, NY, USA, 239–252. https://doi.org/10.1145/2600212.2600227

[17] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. 2011. CSX: An Extended Compression Format for Spmv on Shared Memory Systems. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (San Antonio, TX, USA) *(PPoPP '11)*. Association for Computing Machinery, 247–256.

[18] Da Li and Michela Becchi. 2013. Deploying Graph Algorithms on GPUs: An Adaptive Solution. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 1013–1024. https://doi.org/10.1109/IPDPS.2013.101

[19] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: An Input Adaptive Auto-Tuner for Sparse Matrix-Vector Multiplication. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 117–126. https://doi.org/10.1145/2491956.2462181

[20] Hang Liu and H. Howie Huang. 2019. SIMD-X: Programming and Processing of Graph Algorithms on GPUs. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 411–427.

[21] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, 339–350.

[22] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[23] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. 2019. A Pattern Based Algorithmic Autotuner for Graph Processing on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 201–213. https://doi.org/10.1145/3293883.3295716

[24] Duane Merrill and Michael Garland. 2016. Merge-Based Sparse Matrix-Vector Multiplication (SpMV) Using the CSR Storage Format. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain) *(PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article 43, 2 pages. https://doi.org/10.1145/2851141.2851190

[25] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU Graph Traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA) *(PPoPP '12)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/2145816.2145832

[26] Ulrich Meyer and Prashanthan Sanders. 2003. Delta-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms* 49 (10 2003), 114–152. https://doi.org/10.1016/S0196-6774(03)00076-2

[27] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. *Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing*. Association for Computing Machinery, New York, NY, USA, 622–636. https://doi.org/10.1145/3173162.3173180

[28] Sreepathi Pai and Keshav Pingali. 2016. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 1–19. https://doi.org/10.1145/2983990.2984015

[29] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com

[30] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. 2015. Automatic Selection of Sparse Matrix Representation on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 99–108. https://doi.org/10.1145/2751205.2751244

[31] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph Processing on GPUs: A Survey. , Article 81 (jan 2018), 35 pages.

[32] Jyothish Soman, Kothapalli Kishore, and PJ Narayanan. 2010. A fast GPU algorithm for graph connectivity. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 1–8.

[33] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (aug 1990), 103–111. https://doi.org/10.1145/79173.79181

[34] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. 2019. SEP-Graph: Finding Shortest Execution Paths for Graph Processing under a Hybrid Framework on GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 38–52. https://doi.org/10.1145/3293883.3295733

[35] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. 2015. GraphQ: Graph Query Processing with Abstraction Refinement: Scalable and Programmable Analytics over Very Large Graphs on a Single PC. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA) *(USENIX ATC '15)*. USENIX Association, USA, 387–401.

[36] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain) *(PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article 11, 12 pages. https://doi.org/10.1145/2851141.2851145

[37] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2019. IA-SpGEMM: An Input-Aware Auto-Tuning Framework for Parallel Sparse Matrix-Matrix Multiplication. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/3330345.3330354

[38] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. 2018. *Wonderland: A Novel Abstraction-Based Out-Of-Core Graph Processing System*. Association for Computing Machinery, New York, NY, USA, 608–621. https://doi.org/10.1145/3173162.3173208