Gunrock: A Fast and Programmable Multi-GPU Graph Processing Library

Yangzihao Wang and Yuechao Pan with Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Andy Riffel and John D. Owens
University of California, Davis
{yzhwang, ychpan}@ucdavis.edu

Why use GPUs for Graph Processing?

Graphs

- Found everywhere
  - Road & social networks, web, etc.
- Require fast processing
  - Memory bandwidth, computing power and GOOD software
- Becoming very large
  - Billions of edges
- Irregular data access pattern and control flow
  - Limits performance and scalability

GPUs

- Found everywhere
  - Data center, desktops, mobiles, etc.
- Very powerful
  - High memory bandwidth (288 GBps) and computing power (4.3 Tflops)
- Limited memory size
  - 12 GB per NVIDIA K40
- Hard to program
  - Harder to optimize

Scalability

Performance

Programmability

Current Graph Processing Systems

Single-node CPU-based systems: Boost Graph Library

Multi-CPU systems: Ligra, Galois

Distributed CPU-based systems: PowerGraph

Specialized GPU algorithms

GPU-based systems: CuSha, Medusa, Gunrock...

Why Gunrock?

- Data-centric abstraction is designed for GPU
- Our APIs are simple and flexible
- Our optimizations achieve high performance
- Our framework enables multi-GPU integration

What we want to achieve with Gunrock?

Performance

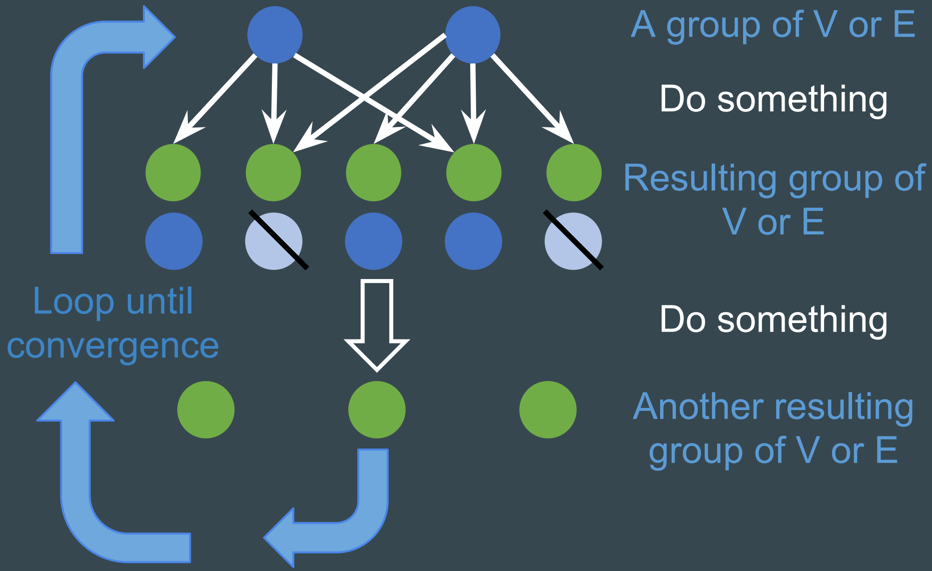
- High performance GPU computing primitives
- High performance framework
- Optimizations
- Multi-GPU capability

Programmability

- A data-centric abstraction designed specifically for the GPU
- Simple and flexible interface to allow user-defined operations
- Framework and optimization details hidden from users, but automatically applied when suitable

Idea: Data-Centric Abstraction & Bulk-Synchronous Programming

A generic graph algorithm:



Data-centric abstraction

- Operations are defined on
  a group of vertices or edges $\stackrel{\text{def}}{=}$ a frontier

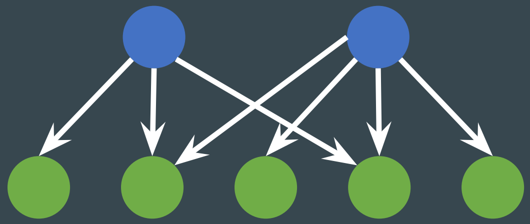
=> Operations = manipulations of frontiers

Bulk-synchronous programming

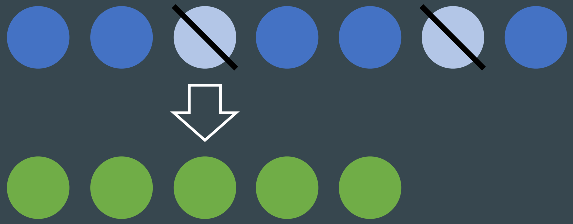
- Operations are done one by one, in order
- Within a single operation, computing on multiple elements can be done in parallel, without order

Gunrock's Operations on Frontiers

Generation

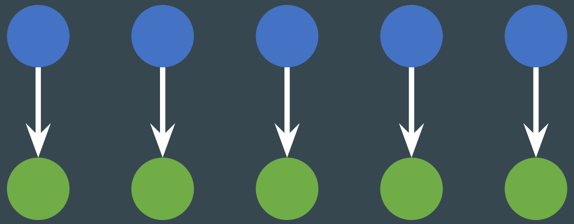


Advance: visit neighbor lists



Filter: select and reorganize

Computation



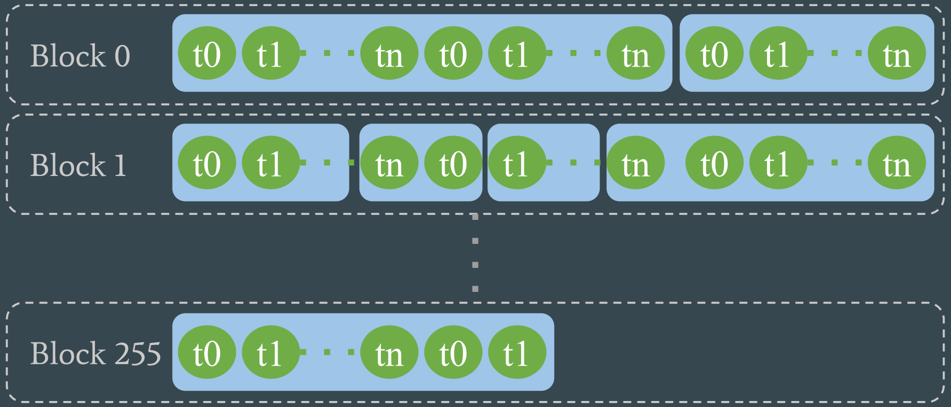
Compute: per-element computation, in parallel
can be combined with advance or filter

Optimizations: Workload mapping and load-balancing

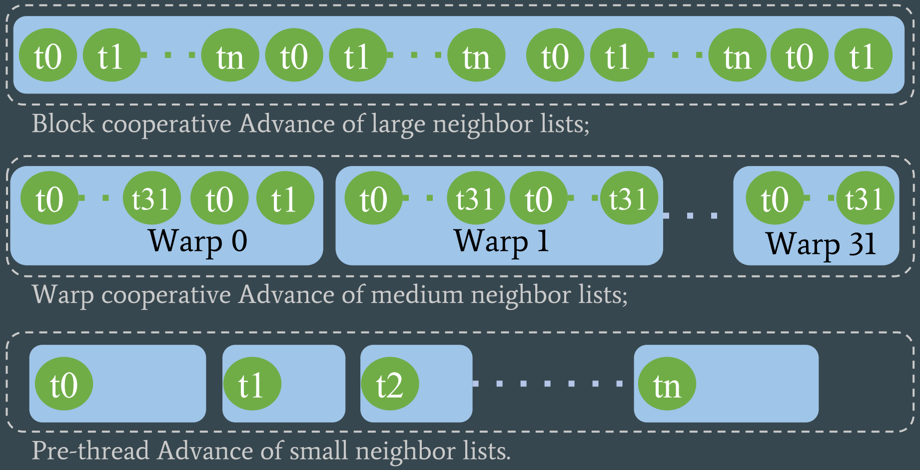
P: uneven neighbor list lengths

S: trade-off between extra processing and load balancing

First appeared in various BFS implementations, now available for all advance operations



Load-Balanced Partitioning [3]



Per-thread fine-grained, Per-warp and per-CTA coarse-grained [4]

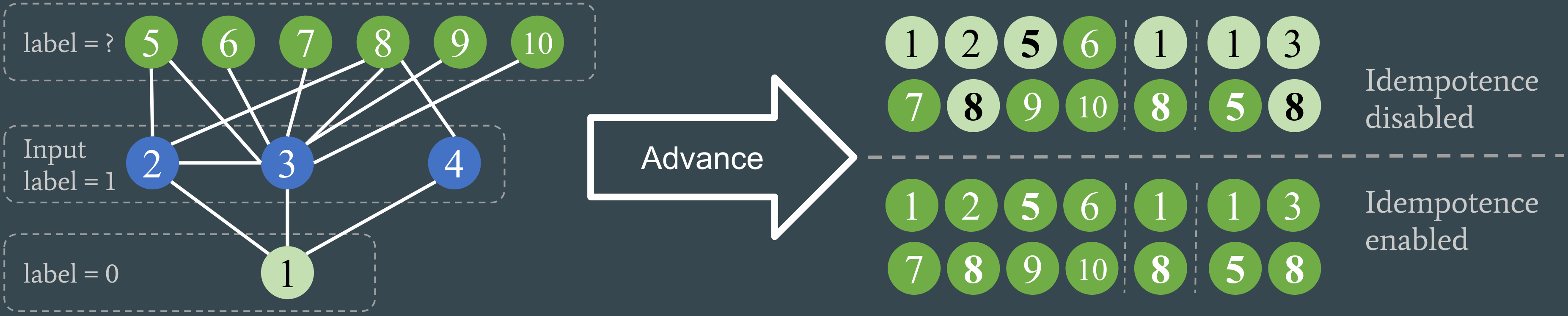
Optimizations: Idempotence

P: Concurrent discovery conflict (v5,8)

S: Idempotent operations (frontier reorganization)

- Allow multiple concurrent discoveries on the same output element
- Avoid atomic operations

First appeared in BFS [4], now available to other primitives



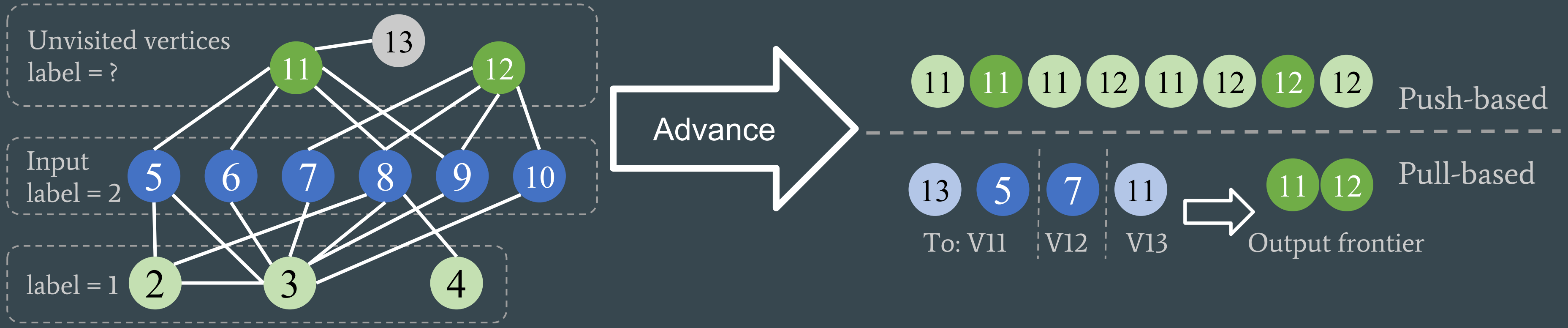
Optimizations: Pull vs. push traversal

P: From many to very few (v5,6,7,8,9,10 -> v11, 12)

S: Pull vs. push operations (frontier generation)

- Automatic selection of advance direction based on ratio of undiscovered vertices

First appeared in DO-BFS [5], now available to other primitives



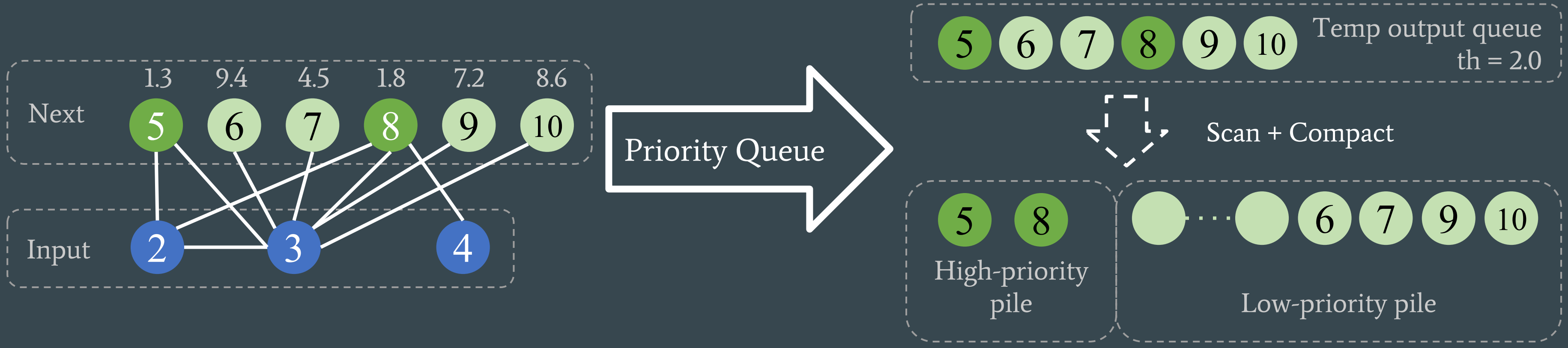
Optimizations: Priority queue

P: A lot of redundant work in SSSP-like primitives

S: Priority queue (frontier reorganization)

- Expand high-priority vertices first

First appeared in SSSP[3], now available to other primitives



Idea: Multiple GPUs

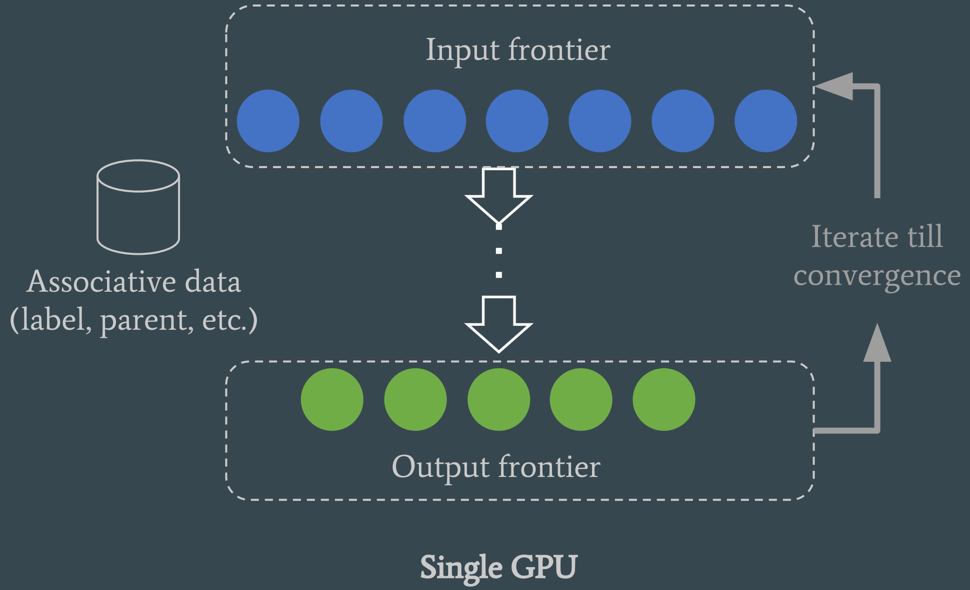
P: Single GPU is not big and fast enough
S: use multiple GPUs
-> larger combined memory space and computing power

P: Multi-GPU program is very difficult to develop and optimize
S: Make algorithm-independent parts into a multi-GPU framework
-> Hide implementation details, and save user's valuable time

P: Single GPU primitives can't run on multi-GPU
S: Partition the graph, renumber the vertices in individual sub-graphs
and do data exchange between super steps
-> Primitives can run on multi-GPUs as it is on single GPU

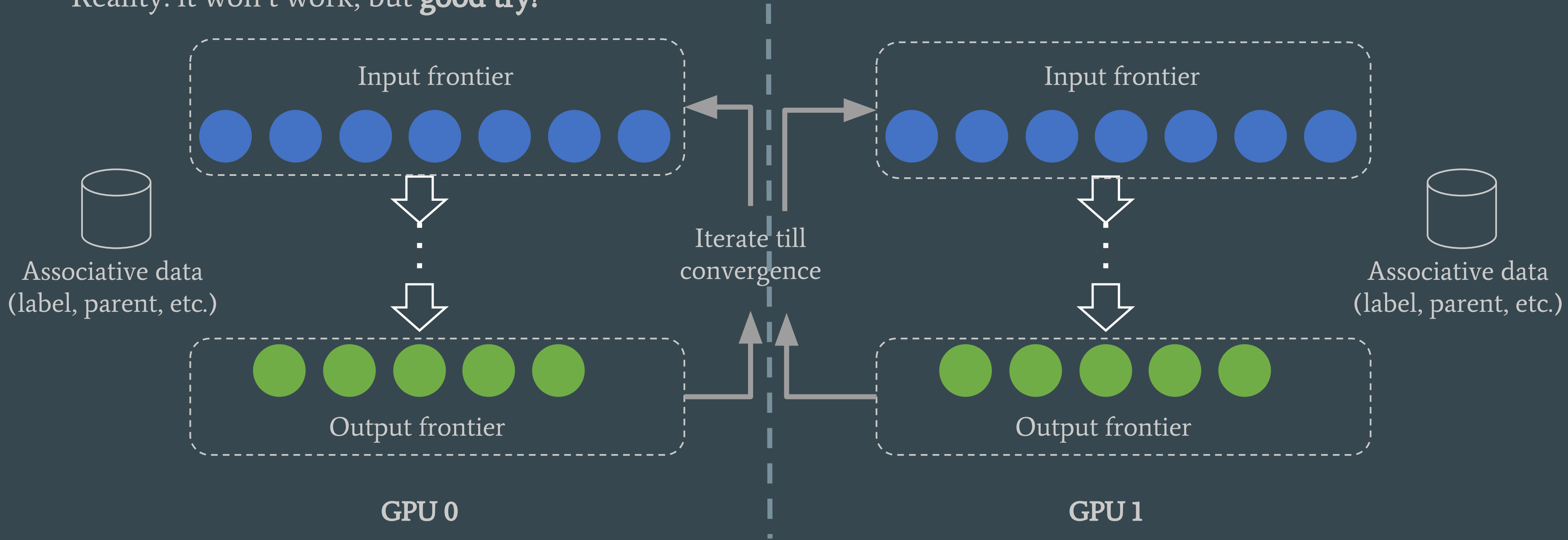
Multi-GPU Framework (for programmers)

Recap: Gunrock on single GPU



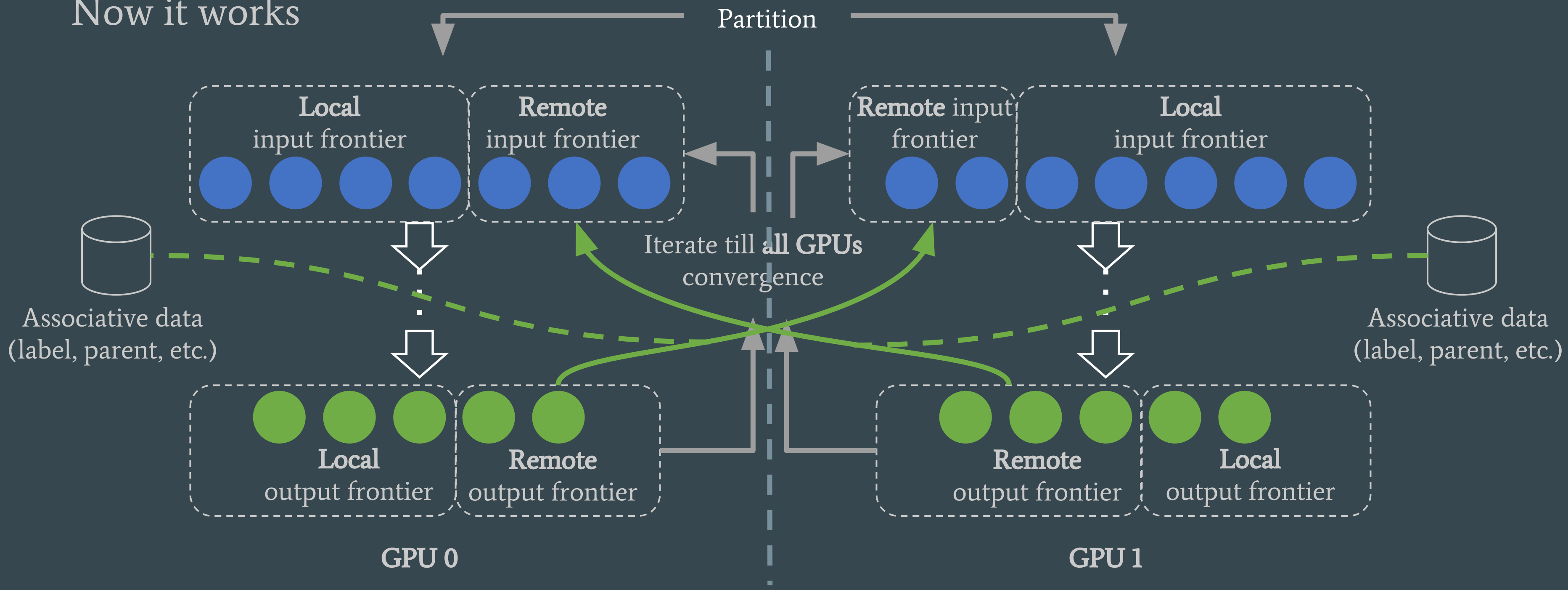
Multi-GPU Framework (for programmers)

Dream: just duplicate the single GPU implementation
Reality: it won't work, but **good try!**



Multi-GPU Framework (for programmers)

Now it works



Multi-GPU Framework (for end users)

```
gunrock_executable input_graph --device=0,1,2,3 other_parameters
```

Graph partitioning

- Distribute the vertices
- Host edges on their sources' host GPU
- Duplicate remote adjacent vertices locally
- Renumber vertices on each GPU (optional)

-> Primitives no need to know peer GPUs
-> Local and remote vertices are separated
-> Partitioning algorithm not fixed

P: Still looking for good partitioning algorithm /scheme

Optimizations: Multi-GPU Support & Memory Allocation

P: Serialized GPU operation dispatch and execution

S: Multi CPU threads and multiple GPU streams

$\geq$1 CPU threads with multiple GPU streams to control each individual GPUs

-> overlap computation and transmission

-> avoid false dependency

P: Memory requirement only known after advance / filter

S: Just-enough memory allocation

check space requirement before every possible overflow

-> minimize memory usage

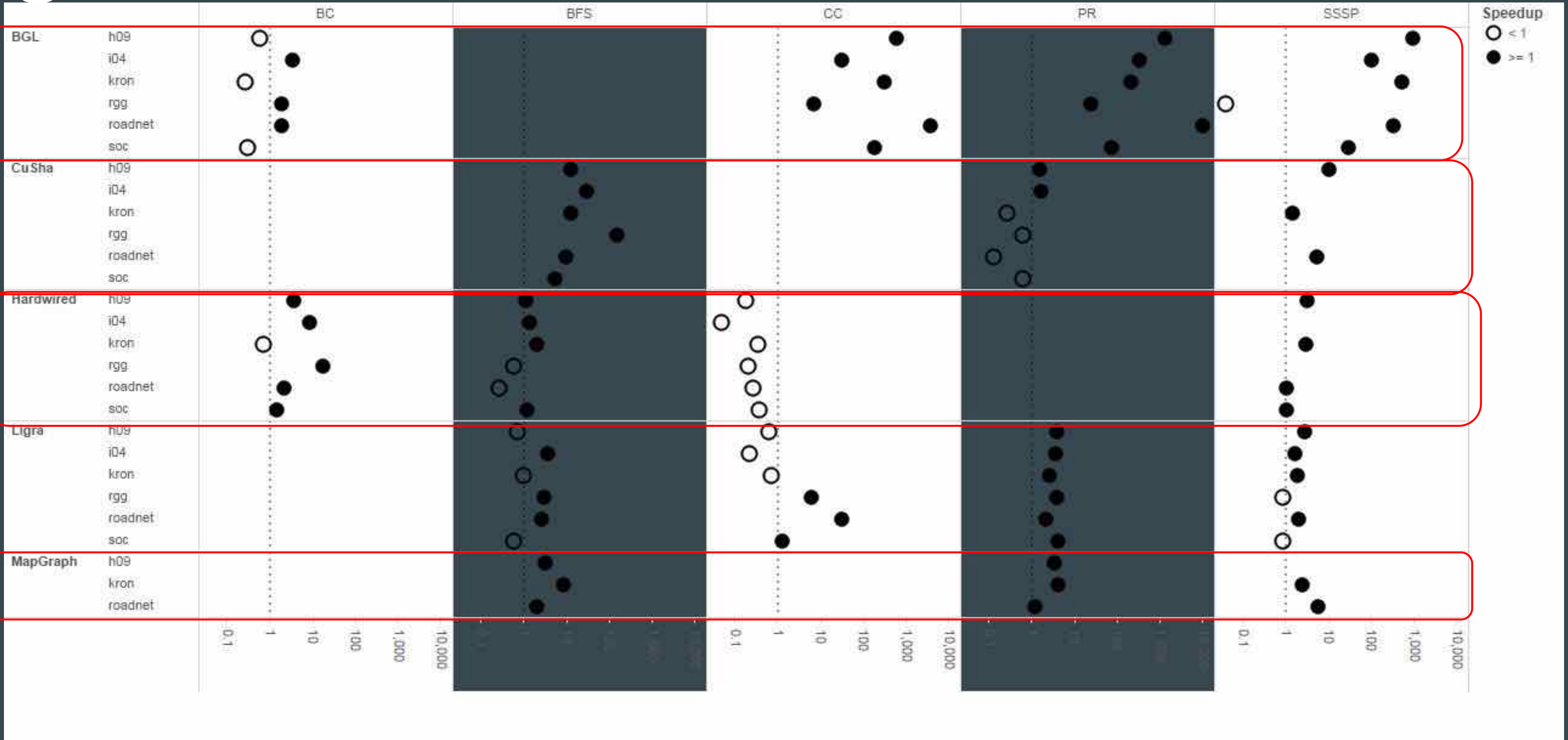
-> can be turned off for performance, if requirements are known (e.g. from previous runs on similar graphs)

Results:
Single GPU Gunrock vs. Others

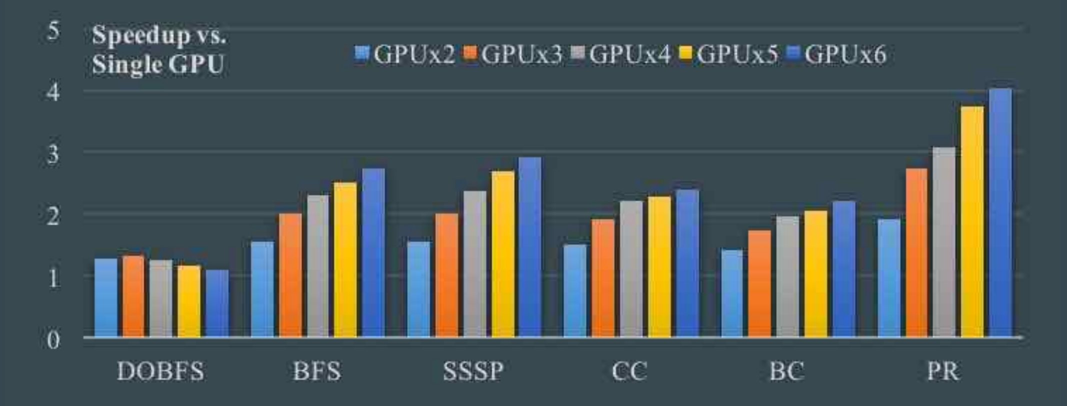
6x-337x speedup on avg over all primitives compared to BGL and PowerGraph.

Outperforms both CuSha and MapGraph.

Comparable to hardwired implementation.



Results: Multi-GPU Scaling



* Primitives (except DOBFS) get good speedups (averaged over 16 datasets of various types)
BFS: 2.74x, SSSP: 2.92x, CC: 2.39x, BC: 2.22x, PR: 4.03x using 6 GPUs

* Peak DOBFS performance: 514 GTEPS with rmat_n20_512

* Gunrock is able to process graph with 3.6B edges (full-friendster graph, undirected, DOBFS in 339ms, 10.7 GTEPS using 4 K40s), 50 PR iterations on the directed version (2.6B edges) took ~51 seconds

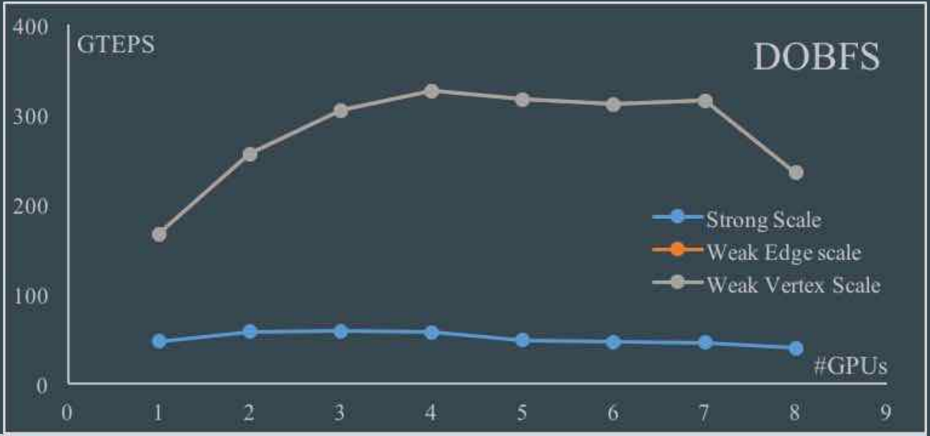
Results: Multi-GPU Scaling

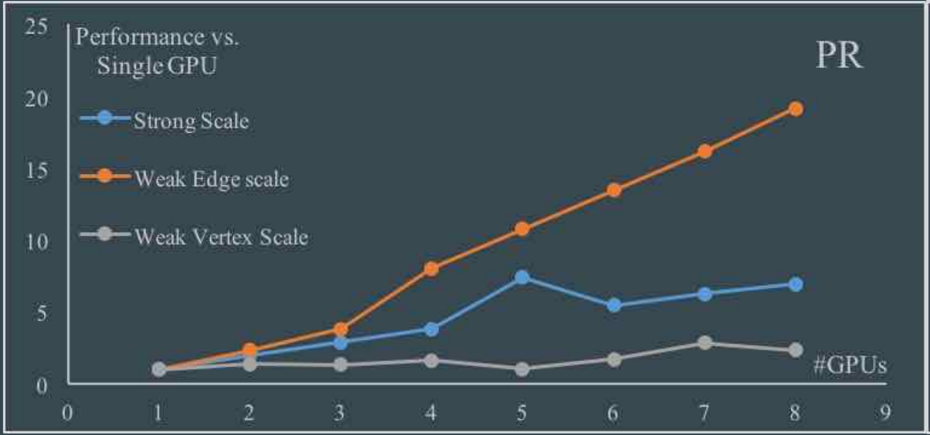
*Strong: Rmat_n24_32

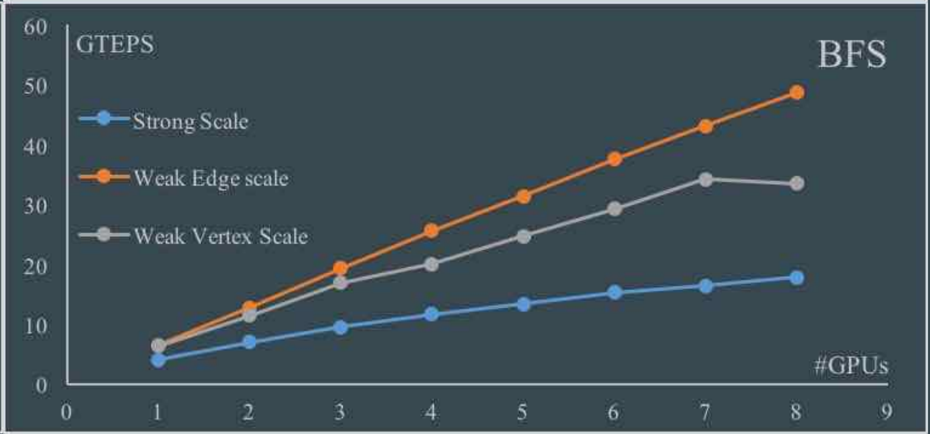
*Weak edge: Rmat_n19_256 * #GPUs

*Weak vertex: Rmat_2^{19} * #GPUs_256

Mostly linear, except for DOBFS strong scaling







Results: Multi-GPU Gunrock vs. Others (BFS)

graph	algo	ref.	ref. hw.	ref. perf.	our hw.	our perf.	comp.
com-orkut (3M, 117M, UD)	BFS	Bisson [5]	1×K20X×4	2.67 GTEPS	4×K40	14.22 GTEPS	5.33X
com-Friendster (66M, 1.81B, UD)	BFS	Bisson [5]	1×K20X×64	15.68 GTEPS	4×K40	14.1 GTEPS	0.90X
kron_n23_16 (8M, 256M, UD)	BFS	Bernaschi [4]	1×K20X×4	~1.3 GTEPS	4×K40	30.8 GTEPS	23.7X
kron_n25_16 (32M, 1.07G, UD)	BFS	Bernaschi [4]	1×K20X×16	~3.2 GTEPS	6×K40	31.0 GTEPS	9.69X
kron_n25_32 (32M, 1.07G, D)	BFS	Fu [13]	2×K20×32	22.7 GTEPS	4×K40	32.0 GTEPS	1.41X
kron_n23_32 (8M, 256M, D)	BFS	Fu [13]	2×K20×2	6.3 GTEPS	4×K40	27.9 GTEPS	4.43X
kron_n24_32 (16.8M, 1.07G, UD)	BFS	Liu [23]	2×K40	15 GTEPS	2×K40	77.7 GTEPS	5.18X
kron_n24_32 (16.8M, 1.07G, UD)	BFS	Liu [23]	8×k40	18.4 GTEPS	4×K80	40.2 GTEPS	2.18X
twitter-mpi (52.6M, 1.96G, D)	BFS	Bebee [3]	1×K40×16	0.2242 sec	3×K40	94.31 ms	2.38X

* graph format: name (|V|, |E|, directed (D) or undirected (UD))
* ref. hw. format: #GPU per node x GPU model x #nodes
* Gunrock out-performs or close to small GPU clusters using 4 ~ 64 GPUs, on both real and generated graphs
* a few times faster than Enterprise (Liu et al., SC15), a dedicated multi-GPU DOBFS implementation

Current Status

Open source, available @

http://gunrock.github.io/

It has over 10 graph primitives

* traversal-based, node-ranking, global (CC, MST)
* LOC ≤ 10 to use a primitive
* LOC ≤ 300 to program a new primitive
* Good balance between performance and programmability

Multi-GPU framework going to support multi-node GPU cluster

* use circular-queue for better scheduling and smaller overhead
* extendable onto multi-node usage

More graph primitives are coming

* graph coloring, maximum independent set, community detection, subgraph matching

Future Work

* Multi-node support with NVLink
* Performance analysis and optimization
* Graph BLAS
* Asynchronized graph algorithms
* Fixed partitioning / 2D partitioning
* Global, neighborhood, and sampling operations
* More graph primitives
* Dynamic graphs
* ...

Acknowledgment

The Gunrock team

Onu Technology and Royal Caliber team

Erich Elsen, Vishal Vaidyananthan, Oded Green and others

For their discussion on library development and dataset generating code

All code contributors to the Gunrock library

NVIDIA

For hardware support, GPU cluster access, and all other supports and discussions

The Gunrock project is funded by

* DARPA XDATA program under AFRL Contract FA8750-13-C-0002
* NSF awards CCF-1017399 and OCI-1032859
* DARPA STTR award D14PC00023

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”. CoRR, abs/1501.05387(1501.05387v4) (Oct. 2015, http://arxiv.org/abs/1501.05387), **to appear at PPoPP 2016**;

[2] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens. “Multi-GPU Graph Analytics”. CoRR, abs/1504.04804(1504.04804v1) (Apr. 2015, http://arxiv.org/abs/1504.04804);

[3] A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel GPU methods for single source shortest paths. In Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium, pages 349–359, May 2014;

[4] D. Merrill, M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’12, pages 117–128, Feb. 2012;

[5] S. Beamer, K. Asanovic, and D. Patterson. Direction-optimizing ´ breadth-first search. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC ’12, pages 12:1–12:10, Nov. 2012;

[6] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley, Dec. 2001;

[7] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. CuSha: Vertexcentric graph processing on GPUs. In Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing, HPDC ’14, pages 239–252, June 2014;

[8] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’13, pages 135–146, Feb. 2013;

[9] Z. Fu, M. Personick, and B. Thompson. MapGraph: A high level API for fast development of high performance graph analytics on GPUs. In Proceedings of Workshop on GRAph Data Management Experiences and Systems, GRADES ’14, pages 2:1–2:6, June 2014;

[10] J. Zhong and B. He. Medusa: Simplified graph processing on GPUs. IEEE Transactions on Parallel and Distributed Systems, 25(6):1543-1552, June 2014;

[11] Z. Fu, H. K. Dasari, B. Bebee, M. Berzins, and B. Thompson. Parallel breadth first search on GPU clusters. In IEEE International Conference on Big Data, pages 110-118, Oct. 2014.

Questions?

Q: How can I find Gunrock?
A: http://gunrock.github.io/

Q: Is it free and open?
A: Absolutely (under Apache License v2.0)

Q: Papers, slides, etc.?
A: https://github.com/gunrock/gunrock#publications

Q: Requirements?
A: CUDA ≥ 7.5, GPU compute capability ≥ 3.0, Linux || Mac OS

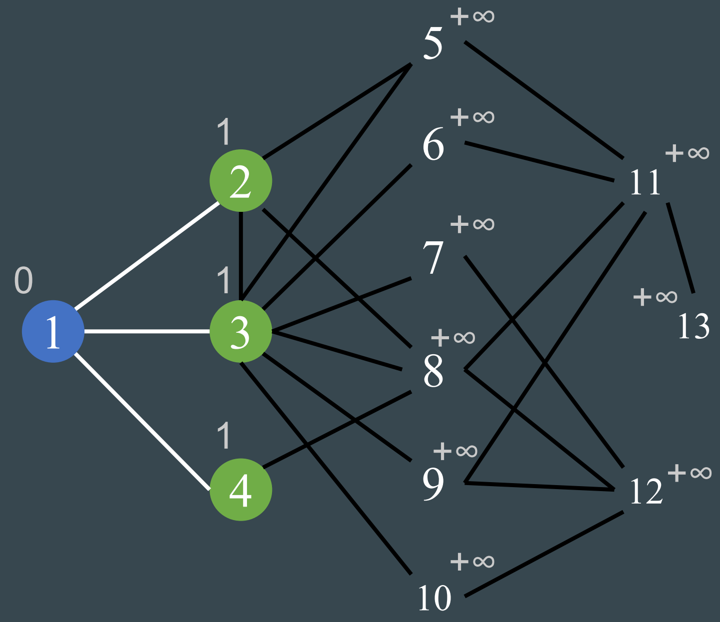
Q: Language?
A: C/C++, with a simple wrapper connects to Python

Q: ... (continue)

Example python interface - breadth-first search

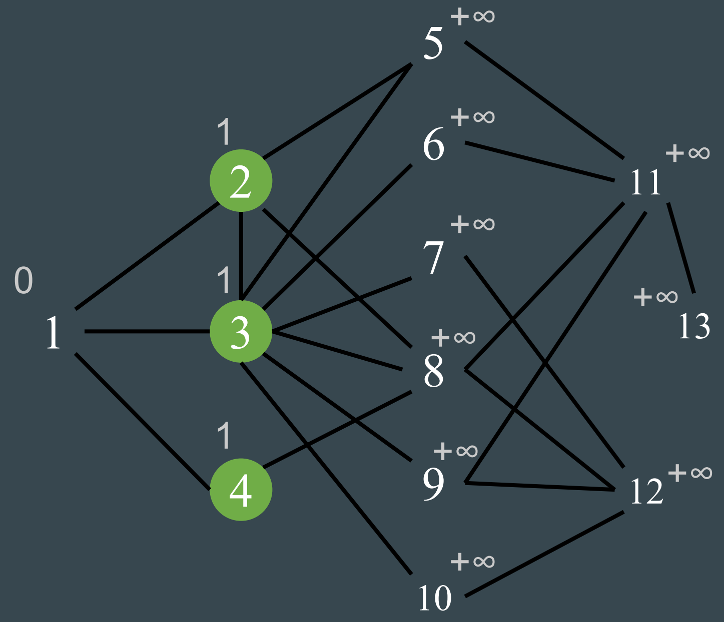
```
from ctypes import *
### load gunrock shared library - libgunrock
gunrock = cdll.LoadLibrary('../../build/lib/libgunrock.so')

### read in input CSR arrays from files
row_list = [int(x.strip()) for x in open('toy_graph/row.txt')]
col_list = [int(x.strip()) for x in open('toy_graph/col.txt')]

### convert CSR graph inputs for gunrock input
row = pointer((c_int * len(row_list))(*row_list))
col = pointer((c_int * len(col_list))(*col_list))
nodes = len(row_list) - 1
edges = len(col_list)

### output array
labels = pointer((c_int * nodes)())

### call gunrock function on device
gunrock.bfs(labels, nodes, edges, row, col, 0)

### sample results
print ' bfs labels (depth):',
for idx in range(nodes): print labels[0][idx],
```

Example: BFS with Gunrock



Advance + Compute (+1, AtomicCAS)

Example: BFS with Gunrock



1

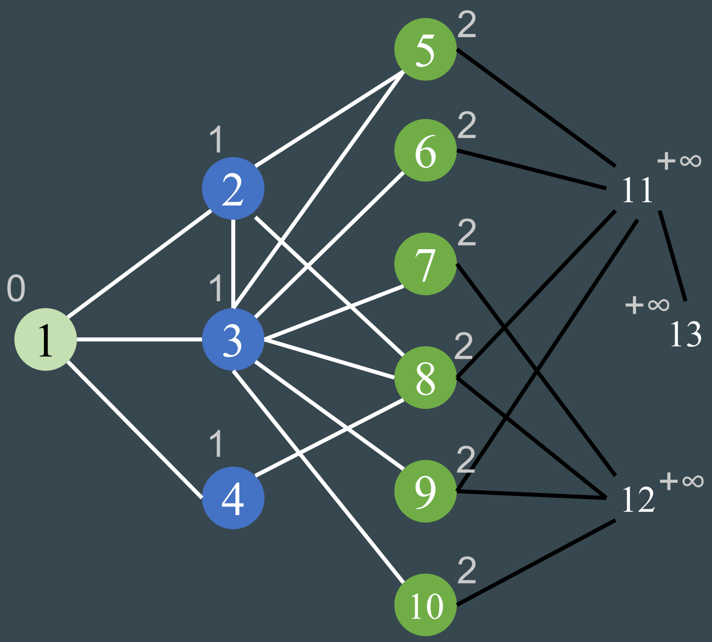
Advance + Compute (+1, AtomicCAS)

3 4 2

Filter

3 4 2

Example: BFS with Gunrock



1

Advance + Compute (+1, AtomicCAS)

3 4 2

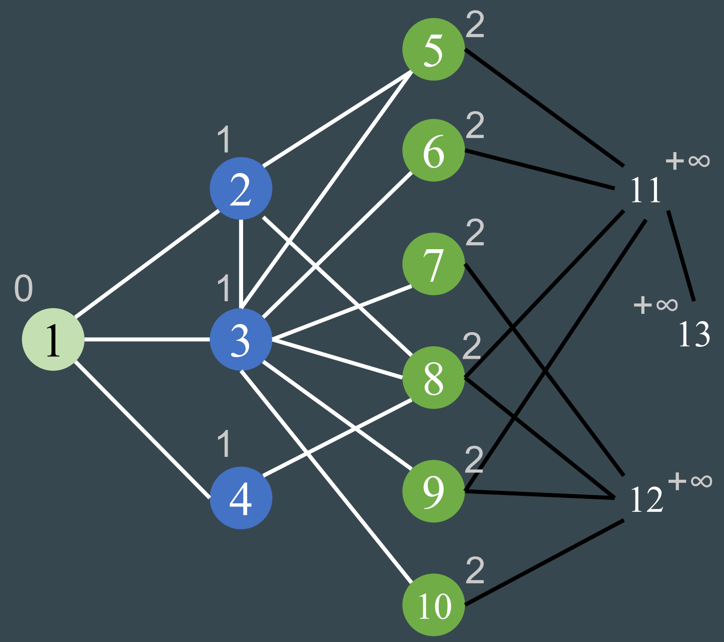
Filter

3 4 2

Advance + Compute (+1, AtomicCAS)

1 2 5 6 7 8 9 10 | 1 8 | 1 3 5 8

Example: BFS with Gunrock



1

Advance + Compute

3 4 2

Filter

3 4 2

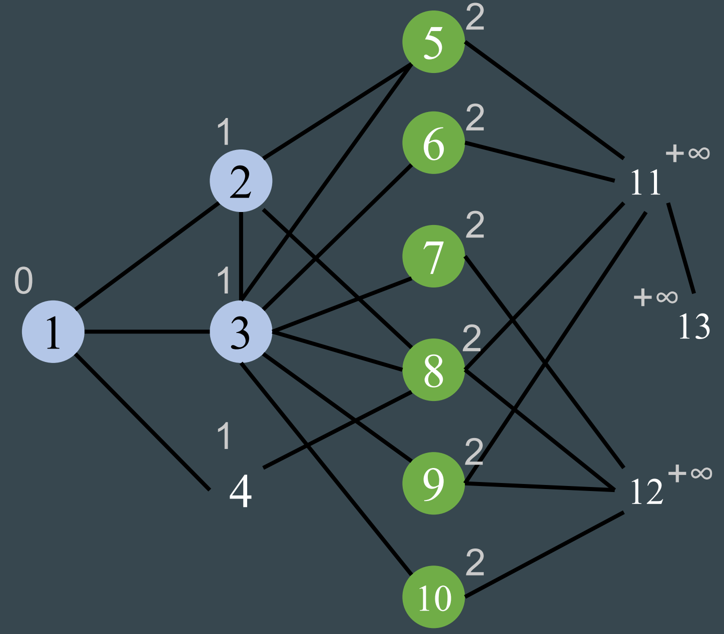
Advance + Compute (+1, AtomicCAS)

1 2 5 6 7 8 9 10 | 1 8 | 1 3 5 8

P: uneven neighbor list lengths (v4 vs. v3)

P: Concurrent discovery conflict (v5,8)

Example: BFS with Gunrock



1

Advance + Compute

3 4 2

Filter

3 4 2

Advance + Compute (+1, AtomicCAS)

1 2 5 6 7 8 9 10 | 1 8 | 1 3 5 8

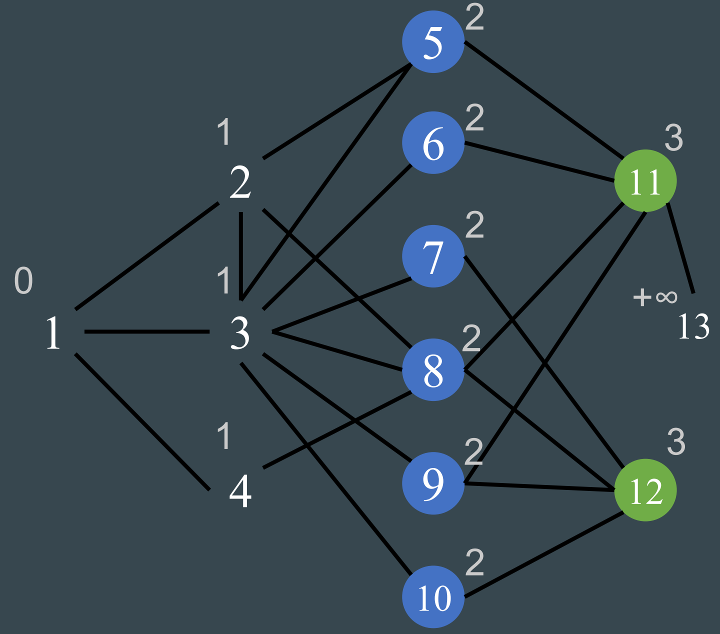
Filter

6 7 9 10 8 5

P: uneven neighbor list lengths (v4 vs. v3)

P: Concurrent discovery conflict (v5,8)

Example: BFS with Gunrock



1

Advance + Compute

3 4 2

Filter

3 4 2

Advance + Compute (+1, AtomicCAS)

1 2 5 6 7 8 9 10 | 1 8 | 1 3 5 8

Filter

6 7 9 10 8 5

Advance + Compute, Filter

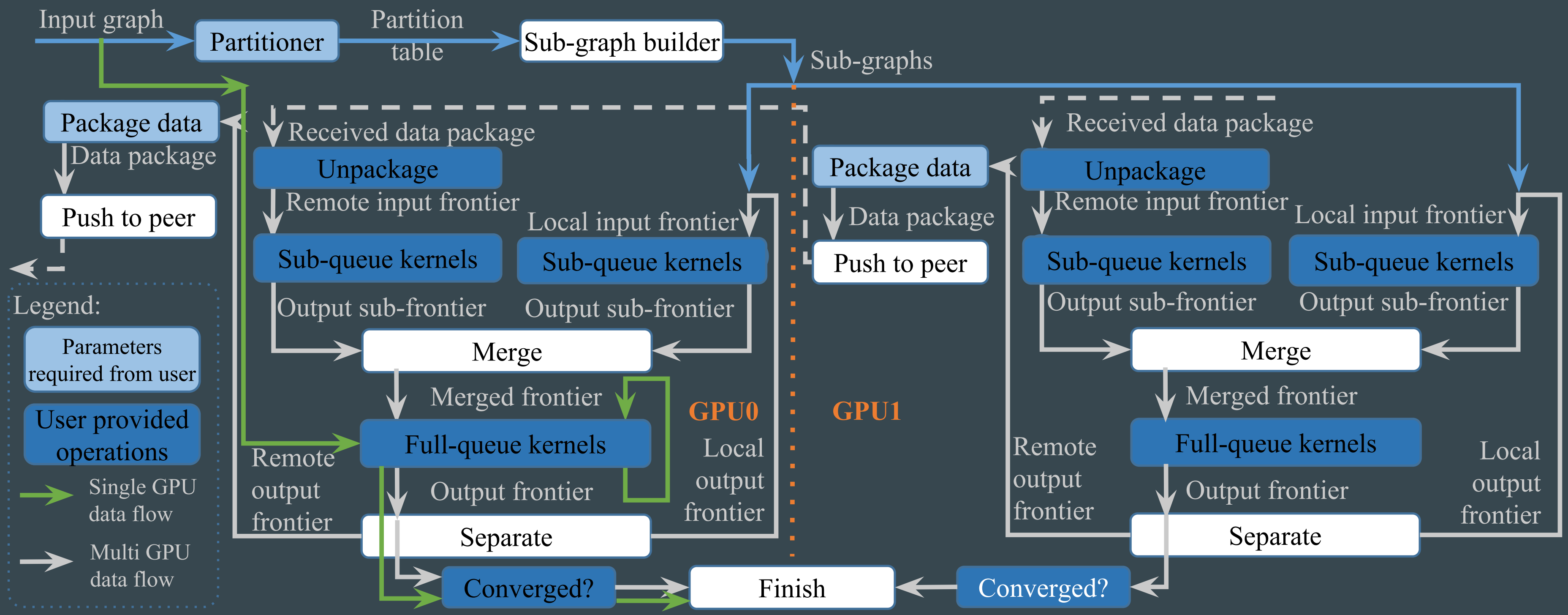
11 12

P: uneven neighbor list lengths (v4 vs. v3)

P: Concurrent discovery conflict (v5,8)

P: From many to very few (v5,6,7,8,9,10 -> v11, 12)

Multi-GPU Framework (for programmers)



Graph partitioning