Dynamic Graphs on the GPU

IPDPS 2020

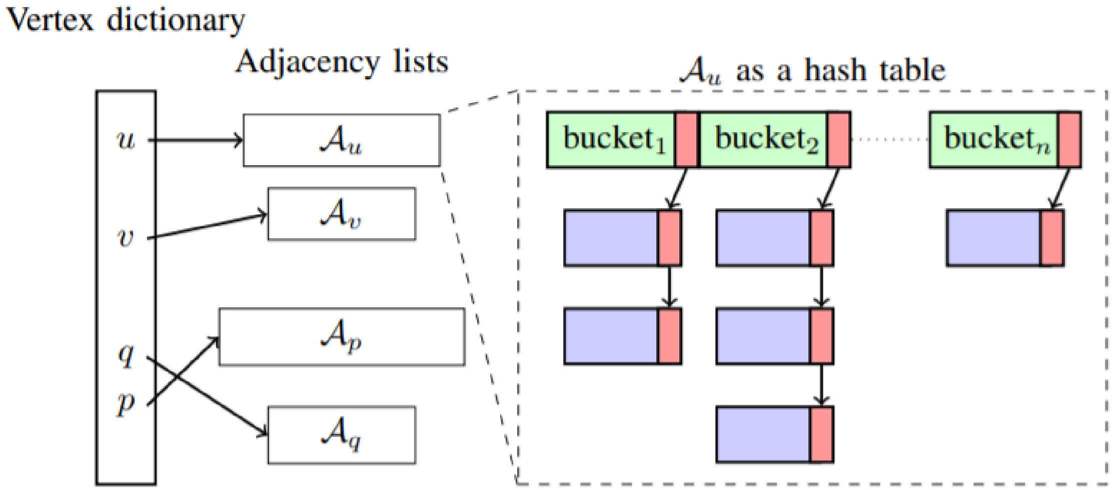
Muhammad A. Awad, Saman Ashkiani, Serban D. Porumbescu, and John D. Owens

IPDPS 2020 New Orleans

UC DAVIS UNIVERSITY OF CALIFORNIA

Dynamic Graphs on the GPU

- **Goal:** High-performance dynamic graph data structure optimized for updates and queries.
- **Approach:** Hash-table-based graph data structure.
- **Argument:** List-based data structures add the complexity of maintaining sorted adjacency lists to optimize queries and updates.



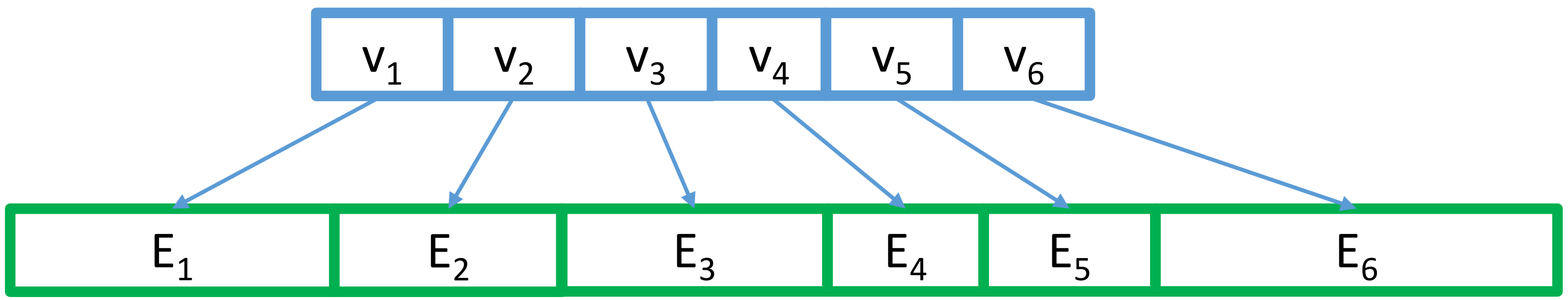
High level schematic of our graph data structure

UCDAVIS

Graph Data Structures

Vertex Dictionary

Edge List (CSR)
Fixed-size array

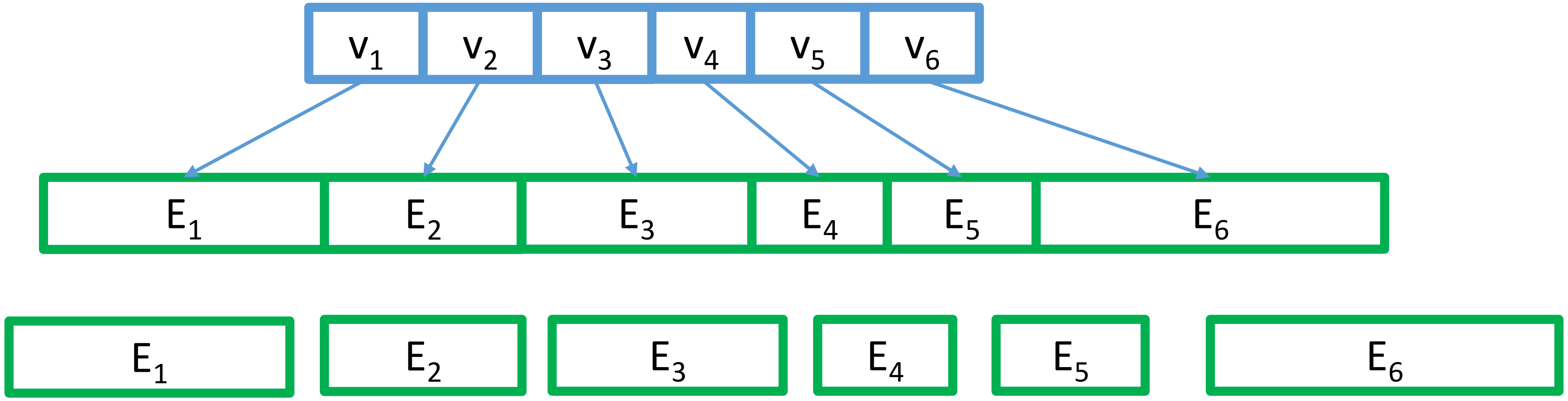


Graph Data Structures

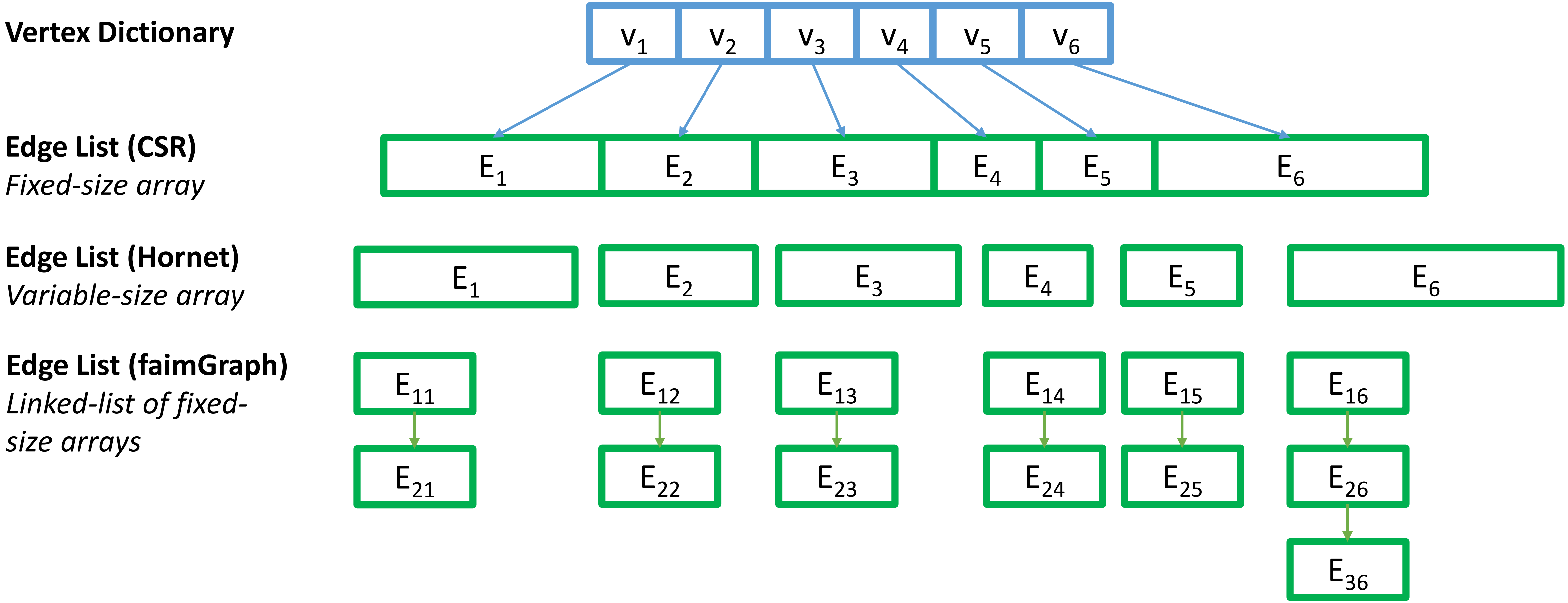
Vertex Dictionary

Edge List (CSR)
Fixed-size array

Edge List (Hornet)
Variable-size array



Graph Data Structures



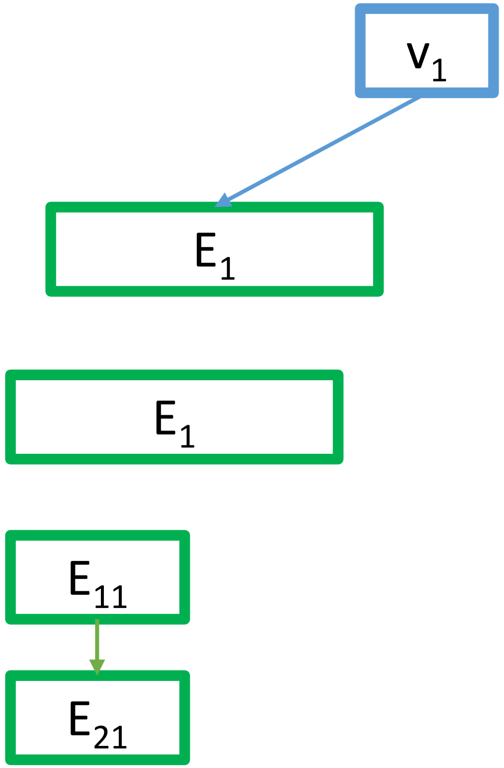
Graph Data Structures

Vertex Dictionary

Edge List (CSR)
Fixed-size array

Edge List (Hornet)
Variable-size array

Edge List (faimGraph)
Linked-list of fixed-size arrays



Edge exist query is an essential query.
Example: insertion while maintaining unique edges per-vertex.

Query cost if the adjacency list is not sorted:

- $O(|E|)$

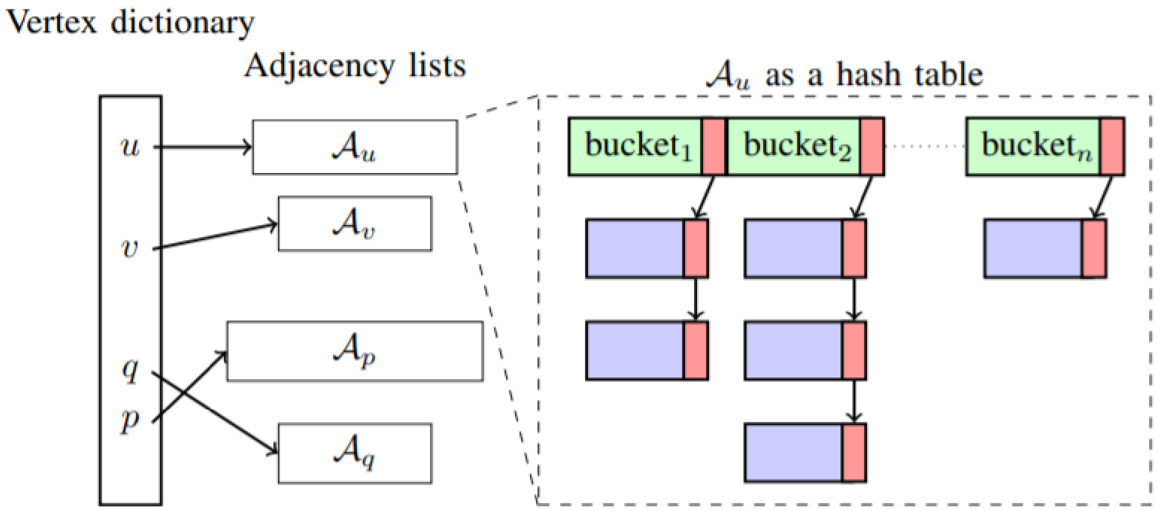
And if sorted:

- $O(\log(|E|))$

… But we must maintain the sorting order during updates.

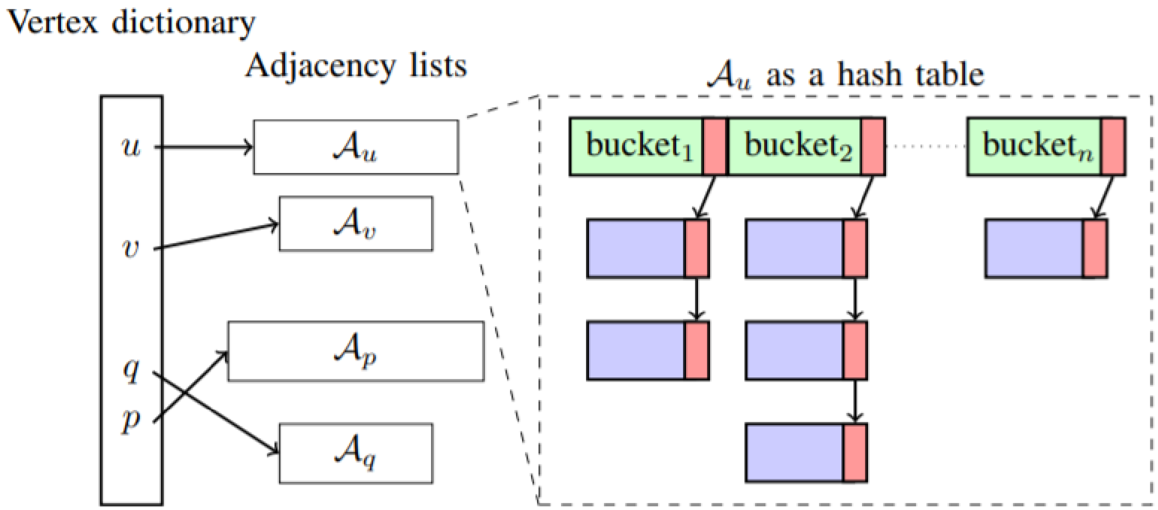
Our Dynamic Graph Data Structure

- Hash-table-based graph data structure.



Our Dynamic Graph Data Structure

- Hash-table-based graph data structure.
- Each vertex has:
  - Pointer to its own hash table (we use Slab Hash*).
  - Additional counters for number of edges and other metrics.



Query Performance:

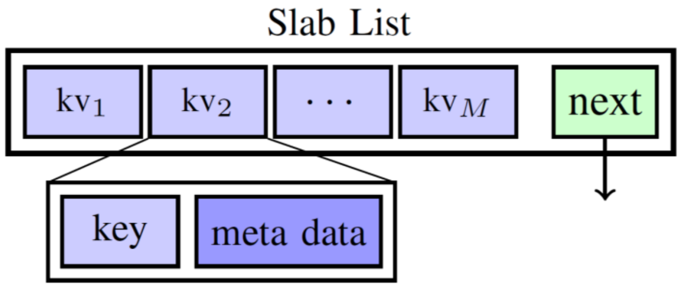
- O(1)

* Saman Ashkiani, Martin Farach-Colton, and John D. Owens. **A Dynamic Hash Table for the GPU**. In *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium*, IPDPS 2018, pages 419–429, May 2018. [bib | DOI | code | http]

UCDAVIS

Slab Hash*

- Load factor defines the initial number of buckets.
- Each bucket is a *128 bytes* slab.
- Collision is resolved using a linked-list of dynamically allocated slabs.
- Offers *concurrent multimap* ***(duplicate keys)*** and our additions:
    - *Concurrent map* **(unique keys)** -> used in weighted graphs
    - *Concurrent set* ***(unique keys and no values)*** -> used in unweighted graphs



* Saman Ashkiani, Martin Farach-Colton, and John D. Owens. **A Dynamic Hash Table for the GPU**. In *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium*, IPDPS 2018, pages 419–429, May 2018. [bib | DOI | code | http]

UCDAVIS

Our Dynamic Graph Data Structure

We support the following operations:

- **Low-level Operations**
  - Edge insertion and deletion
  - Vertex insertion and deletion
- **Bulk build**
- **Queries**
  - Edge exist query
  - Iterator over a vertex's adjacency list

Example Operation: Edge Insertion

- Using warp cooperative work sharing strategy (WCWS).
  - Per-thread assignment.
  - Per-warp processing.

Algorithm 1 Graph edge insertion algorithm.

```
procedure INSERTEDGES(GpuGraph graph, Edges edges)
    thread_edge ← edges[threadIdx]
    to_insert ← thread_edge.src != thread_edge.dst
    while work_queue ← ballot(to_insert) do
        current_lane ← find_first_set_bit(work_queue)
        current_src ← shuffle(thread_edge.src, current_lane)
        same_src ← thread_edge.src == current_src
        success ← graph[current_src].replace(thread_edge, same_src & to_insert)
        added_count ← popc(ballot(success))
        graph[current_src].incrementEdgesCount(added_count)
        if same_src & to_insert then
            to_insert ← false
        end if
    end while
end procedure
```

Example Operation: Edge Insertion

- Using warp cooperative work sharing strategy (WCWS).
  - Per-thread assignment.
  - Per-warp processing.

Build a queue of new edges →

Algorithm 1 Graph edge insertion algorithm.

```
procedure INSERTEDGES(GpuGraph graph, Edges edges)
    thread_edge ← edges[threadIdx]
    to_insert ← thread_edge.src != thread_edge.dst
    while work_queue ← ballot(to_insert) do
        current_lane ← find_first_set_bit(work_queue)
        current_src ← shuffle(thread_edge.src, current_lane)
        same_src ← thread_edge.src == current_src
        success ← graph[current_src].replace(thread_edge, same_src & to_insert)
        added_count ← popc(ballot(success))
        graph[current_src].incrementEdgesCount(added_count)
        if same_src & to_insert then
            to_insert ← false
        end if
    end while
end procedure
```

Example Operation: Edge Insertion

- Using warp cooperative work sharing strategy (WCWS).
  - Per-thread assignment.
  - Per-warp processing.

Build a queue of new edges →

Warp-wide single edge insertion →

Algorithm 1 Graph edge insertion algorithm.

```
procedure INSERTEDGES(GpuGraph graph, Edges edges)
    thread_edge ← edges[threadIdx]
    to_insert ← thread_edge.src != thread_edge.dst
    while work_queue ← ballot(to_insert) do
        current_lane ← find_first_set_bit(work_queue)
        current_src ← shuffle(thread_edge.src, current_lane)
        same_src ← thread_edge.src == current_src
        success ← graph[current_src].replace(thread_edge, same_src & to_insert)
        added_count ← popc(ballot(success))
        graph[current_src].incrementEdgesCount(added_count)
        if same_src & to_insert then
            to_insert ← false
        end if
    end while
end procedure
```

Example Operation: Edge Insertion

- Using warp cooperative work sharing strategy (WCWS).
  - Per-thread assignment.
  - Per-warp processing.

Algorithm 1 Graph edge insertion algorithm.

```
procedure INSERTEDGES(GpuGraph graph, Edges edges)
    thread_edge ← edges[threadIdx]
    to_insert ← thread_edge.src != thread_edge.dst
    while work_queue ← ballot(to_insert) do
        current_lane ← find_first_set_bit(work_queue)
        current_src ← shuffle(thread_edge.src, current_lane)
        same_src ← thread_edge.src == current_src
        success ← graph[current_src].replace(thread_edge, same_src & to_insert)
        added_count ← popc(ballot(success))
        graph[current_src].incrementEdgesCount(added_count)
        if same_src & to_insert then
            to_insert ← false
        end if
    end while
end procedure
```

Build a queue of new edges → while work_queue ← ballot(to_insert) do

Warp-wide single edge insertion → success ← graph[current_src].replace(thread_edge, same_src & to_insert)

Maintain per-vertex edge count → graph[current_src].incrementEdgesCount(added_count)

Example Operation: Edge Insertion

- Using warp cooperative work sharing strategy (WCWS).
  - Per-thread assignment.
  - Per-warp processing.
- WCWS benefits:
  - Eliminates branch divergence.
  - Coalesced memory access.

Evaluating a Dynamic Graph Data Structure

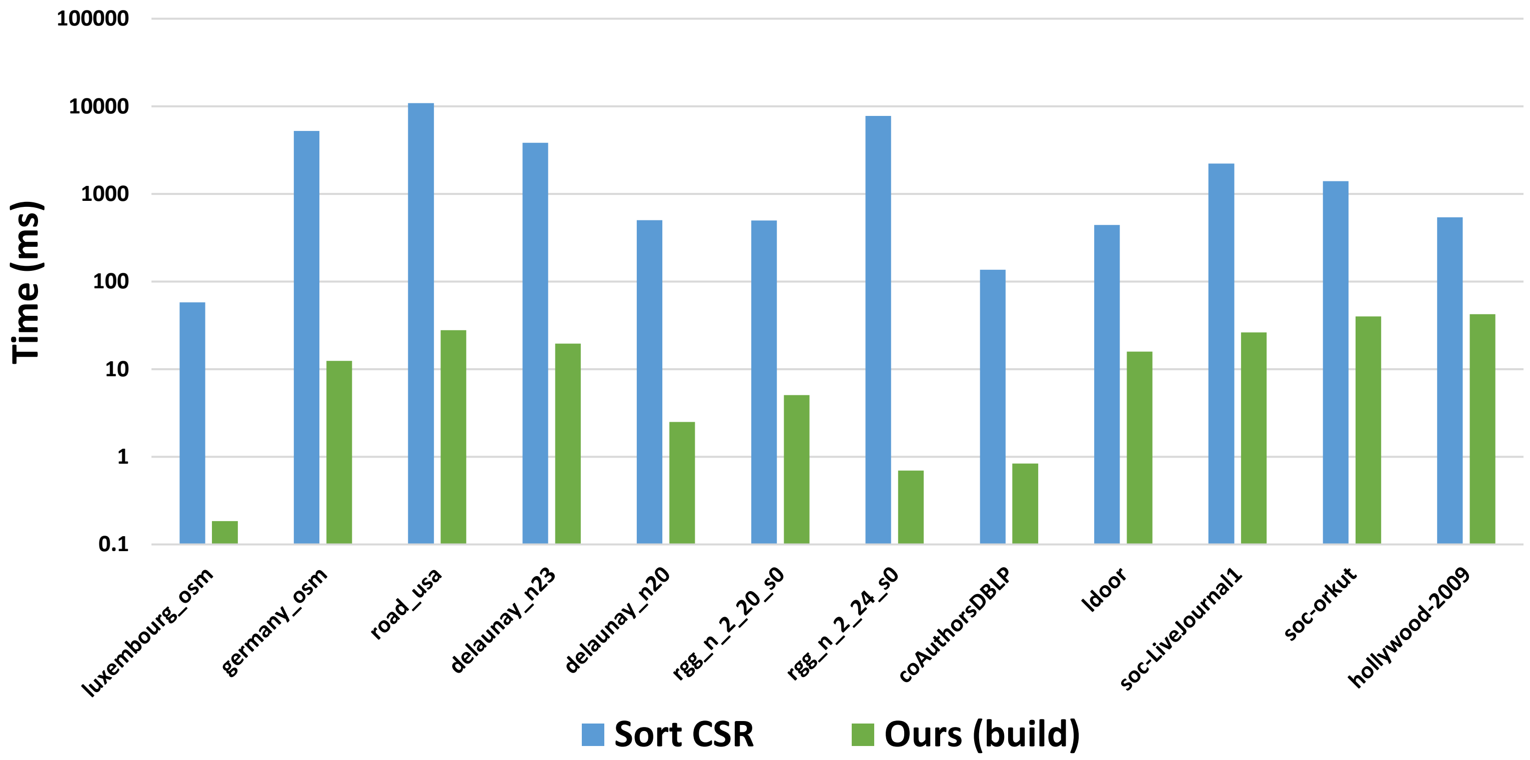
We define a set of benchmarks to evaluate a dynamic graph data structure:

- **Low-level Operations:**
    - Edge insertion and deletion
    - Vertex insertion and deletion
- **Workloads:**
    - Bulk build
    - Incremental build
- **Applications:**
    - Static and dynamic graph application (we use triangle counting)

We evaluate our graph data structure and compare it to Hornet and faimGraph.

Results*

- Sorting CSR† compared to building our data structure (log scale).



* Using an NVIDIA TITAN V (Volta) GPU.

† Using CUB's Segmented Sort

UCDAVIS

Results*

- High throughput low-level operations.

Operation	Rate (MOp/s)	Speedup vs. Hornet	Speedup vs. faimGraph
Edge insertion	646	**5.8-14.8x**	**3.4-5.4x**
Edge deletion	1024.87	**1.0-7.0x**	**3.6-5.7x**
Vertex deletion	26.49	**--**	**8.9-12.2x**

- High throughput in graph building workloads.
  - Incremental build: 993.82 MEdge/s (5x faster than Hornet).
  - Bulk build: 2–30x faster than Hornet.
    - In hollywood-2009 dataset 45% of Hornet’s time is spent in deduplication (same time required to build or data structure).

*** Using an NVIDIA TITAN V (Volta) GPU.**

*** Averaged over different datasets**

UCDAVIS

Results*

- Dynamic triangle counting.
  - Intersection: given two adjacency lists, count the number of common vertices.
  - Two phases: 1) compute the intersections, 2) update the graph.

- Although performing intersection using hash tables is slower than using sorted lists, the update phase makes up for the slowdown.
  - 5 rounds of updates (1M Edge insertion).
  - Hollywood-2009: 56,774 ms (0.91x Hornet).
  - Road_usa: 325.8 ms (1.83x Hornet).

*** Using an NVIDIA TITAN V (Volta) GPU.**

Conclusions and Future Work

- **Hash-table-based dynamic graph data structure offers superior performance compared to alternative list-based graph data structures.**

- **Vertices have different workloads (updates and queries).**
    - Load factor controls this tradeoff and we can have different load factors per-vertex.

Conclusions and Future Work

- **Hash tables are not suited for all graph problems.**
  - A sorted adjacency list is useful for some application, we can replace a hash table with a B-Tree*.
- **Concurrent updates and queries.**
- **Dynamic graph problems and workloads.**

* Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D. Owens. **Engineering a High-Performance GPU B-Tree**. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2019, pages 145–157, February 2019. [bib | DOI | http]

Acknowledgments

- Martín Farach-Colton. *Rutgers University*
- Yuechao Pan, Muhammad Osama, and Kerry A. Seitz. *UC Davis*

Thanks!

- Send us your questions:
  - Muhammad A. Awad: mawad@ucdavis.edu
  - Saman Ashkiani: sashkiani@ucdavis.edu
  - Serban D. Porumbescu: sdporumbescu@ucdavis.edu
  - John D. Owens: jowens@ece.ucdavis.edu
- Our code:
  - In Gunrock: https://github.com/gunrock/gunrock/tree/dynamic-graph
  - Open issues if you have questions about the code or how to use it.