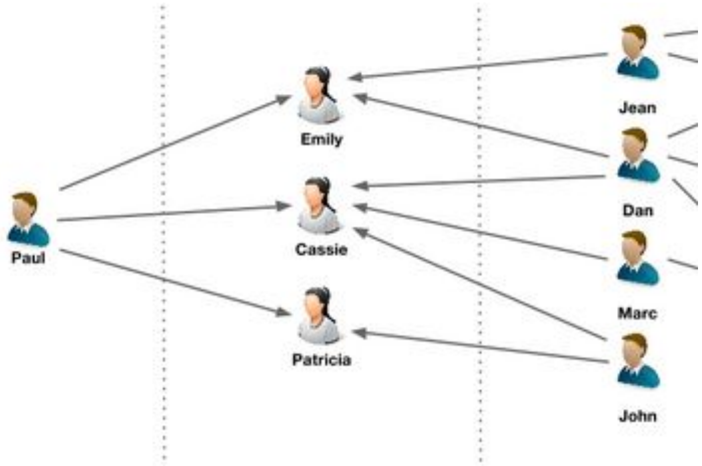
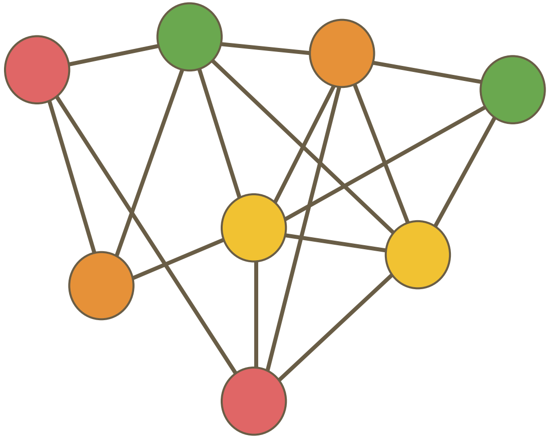
Dynamic Graph Data Structures on the GPU

Graph Problems

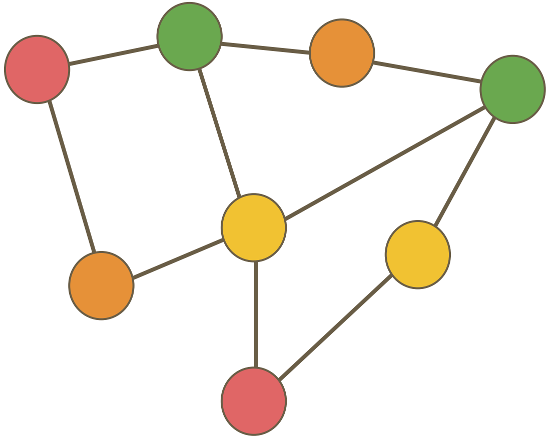


Graph Density

Dense Graphs



Sparse Graphs



Static vs. Dynamic Graphs

Static Graphs

- Queries

Dynamic Graphs

- Queries
- Add Vertex
- Delete Vertex
- Insert Edge
- Remove Edge

Data Structures: Vertices and Edges

Vertices: Stored in an array, often just a pointer to edge lists

Edges: Many possible data structures

Dense Graphs

The edge list of dense graphs is often stored in an **Adjacency Matrix**

Pros:

- Efficient use of memory

Cons:

- Large memory requirement as number of vertices grow

	v_0	v_1	v_2	v_3	v_4	v_5
v_0		1	1		1	1
v_1	1		1	1	1	1
v_2	1	1		1		1
v_3		1	1		1	
v_4	1	1		1		1
v_5	1	1	1		1	

Sparse Graphs

This doesn't work well for sparse graphs since there are few edges per vertex

Large graphs tend to be sparse

	v_0	v_1	v_2	v_3	v_4	v_5
v_0		1			1	
v_1	1			1		
v_2				1		1
v_3		1	1			
v_4	1					1
v_5			1		1	

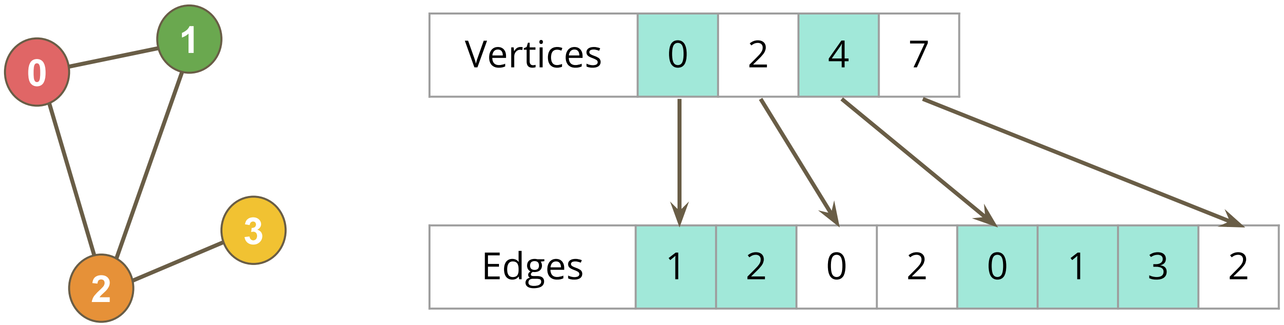
Dataset Statistics

Some relevant data sets.

Data Sets	Vertices	Edges	Max Degree	Avg Degree
luxembourg_osm	114K	239K	6	2.1
germany_osm	11.5M	24.7M	13	2.1
road_usa	23.9M	57.71M	9	2.4
delaunay_n20	1M	6.3M	23	6.0
hollywood-2009	1.1M	112.8M	11,000	98.9

Static Implementation: Compressed Sparse Row

Static graphs can make extremely efficient use of memory



Inserting a new edge or deleting a vertex would require a complete rebuild of this structure.

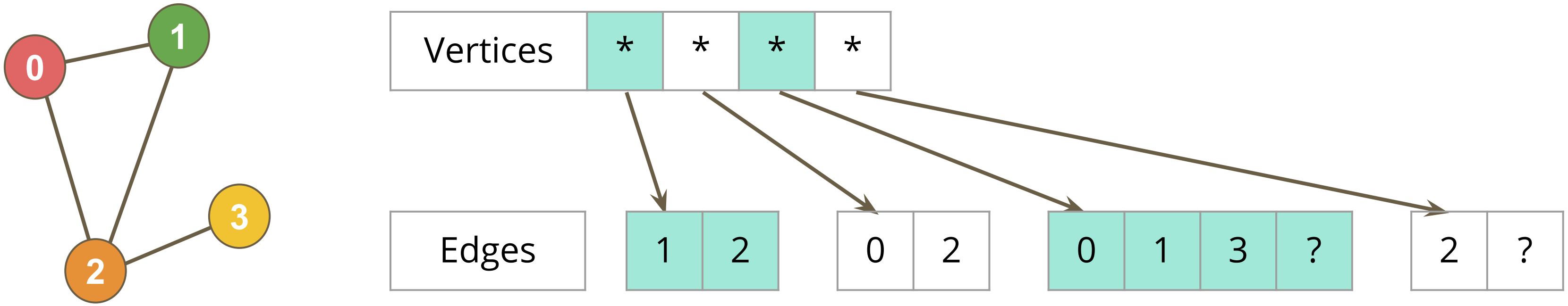
Dynamic Graph Data Structures

Two alternate GPU dynamic graph data structures:

- Hornet
- faimGraph

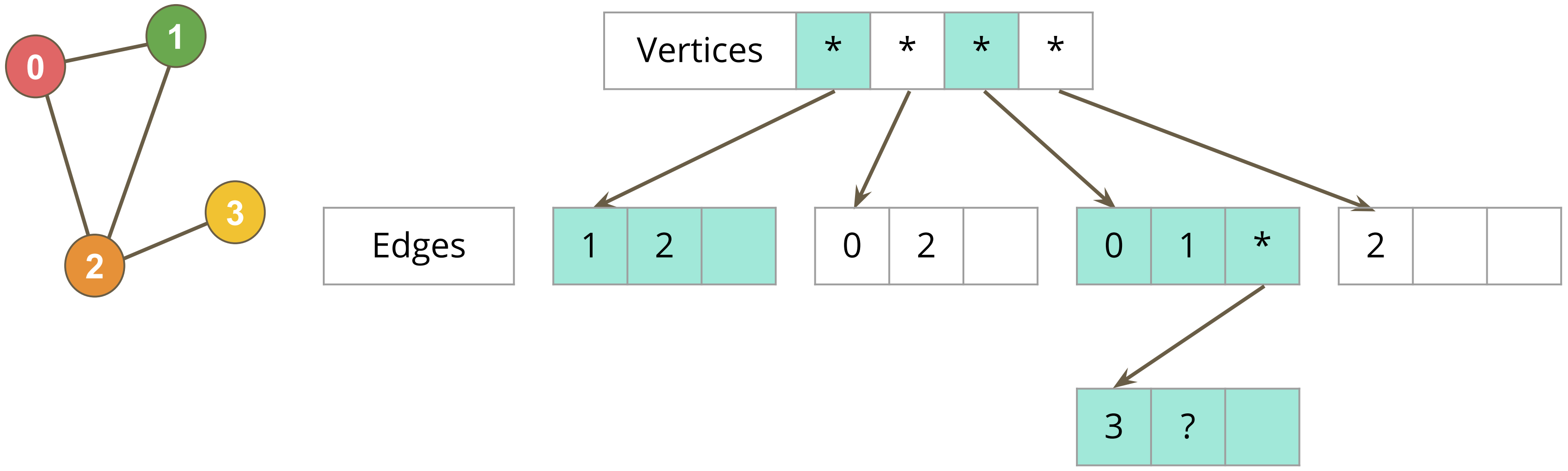
Hornet

Hornet stores an edge list for each vertex
Each edge list is a variable sized array



faimGraph

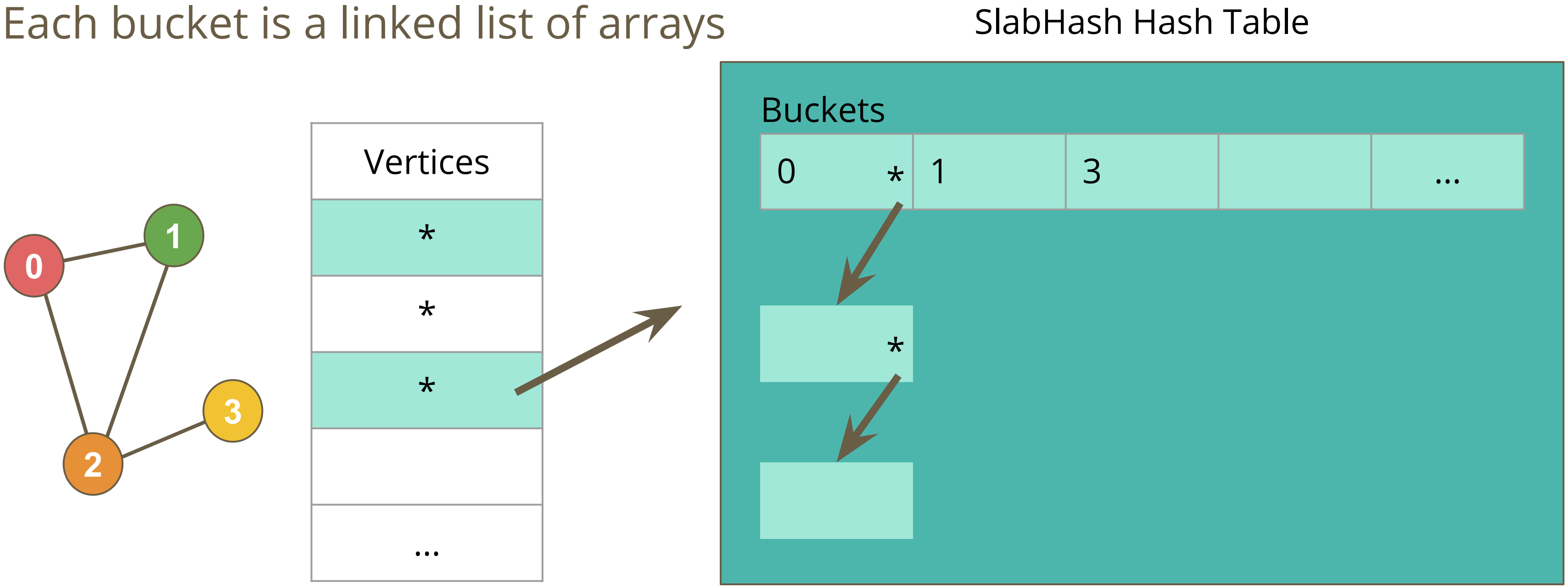
faimGraph stores edges in a linked list of fixed sized arrays



GPU Hash Tables

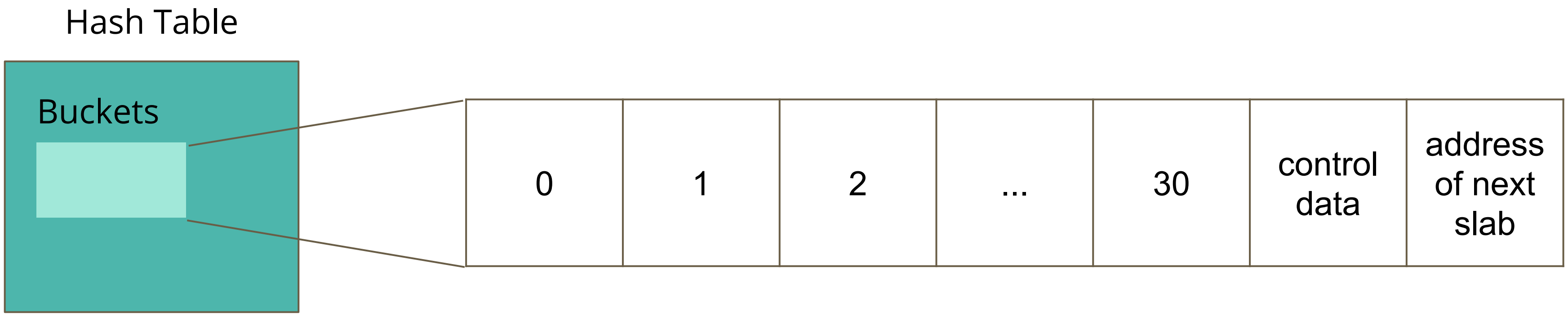
Edges are stored in a hash table

Each bucket is a linked list of arrays



Hash Table Buckets

Each bucket is a linked list of **slabs**
Each slab holds 30 edge indices along with some other control data



If edges have a weight, we would only store 15 values per slab

Why Hash Tables?

With a good hash distribution, they can be very fast

Operation	Complexity	
	Average Case	Worst Case
Query	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

Implementation Details

Dynamic Graph Setup

Initial Allocations:

- Large block of memory for Vertices, each a hash table
- Large block of memory for Slabs
- Large block of memory for operation data (Edges to insert, delete, etc)

Initial Setup

- Each vertex is assigned a configurable number of empty buckets

2 Approaches for Inserting Edges

- Each thread inserts one edge

- Groups of 32 threads work cooperatively to do the work assigned to that group of threads

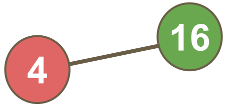
Common "Insert Edge" Kernel Setup

Regardless of which approach is taken, the following setup is done on the CPU

- A list of edges to insert are copied to the GPU
- A CUDA Kernel is run with 1 thread per entry in the list

"Insert Edge": 1 per Thread

- Calculate the hash for the edge being inserted to pick a bucket

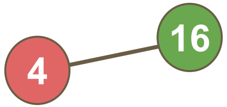


Hash = 16 % numBuckets;

* simplest possible hash
assuming 5 buckets per vertex

"Insert Edge": 1 per Thread

- Calculate the hash for the edge being inserted to pick a bucket
- **Look up bucket address based on the vertex number and the hash**



Hash = 16 % numBuckets;
BucketAddr = baseAddr + Hash;

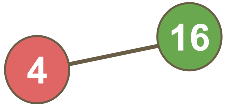
V4 Bucket	31	1	-1	...	-1	*	*

* simplest possible hash
assuming 5 buckets per vertex

Step 3/6

"Insert Edge": 1 per Thread

- Calculate the hash for the edge being inserted to pick a bucket
- Look up bucket address based on the vertex number and the hash
- **Loop through the slots in the bucket**
    - If edge was already found, exit loop
    - If empty slot found:
        - Use AtomicCAS to store the edge



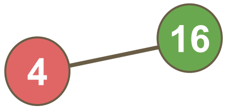
V4 Bucket	31	1	-1	...	-1	*	*

Hash = 16 % numBuckets;
BucketAddr = baseAddr + Hash;

* simplest possible hash
assuming 5 buckets per vertex

"Insert Edge": 1 per Thread

- Calculate the hash for the edge being inserted to pick a bucket
- Look up bucket address based on the vertex number and the hash
- **Loop through the slots in the bucket**
    - If edge was already found, exit loop
    - If empty slot found:
        - Use AtomicCAS to store the edge



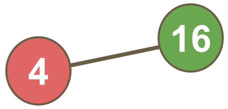
V4 Bucket	31	1	-1	...	-1	*	*

```
Hash = 16 % numBuckets;
BucketAddr = baseAddr + Hash;
```

* simplest possible hash
assuming 5 buckets per vertex

"Insert Edge": 1 per Thread

- Calculate the hash for the edge being inserted to pick a bucket
- Look up bucket address based on the vertex number and the hash
- Loop through the slots in the bucket
    - If edge was already found, exit loop
    - **If empty slot found:**
        - Use AtomicCAS to store the edge



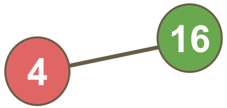
V4 Bucket							
	31	1	-1	...	-1	*	*

Hash = 16 % numBuckets;
BucketAddr = baseAddr + Hash;

* simplest possible hash
assuming 5 buckets per vertex

"Insert Edge": 1 per Thread

- Calculate the hash for the edge being inserted to pick a bucket
- Look up bucket address based on the vertex number and the hash
- Loop through the slots in the bucket
    - If edge was already found, exit loop
    - If empty slot found:
        - **Use AtomicCAS to store the edge**



Hash = 16 % numBuckets;
BucketAddr = baseAddr + Hash;

V4 Bucket

31	1	16	...	-1	*	*

* simplest possible hash
assuming 5 buckets per vertex

Atomics

Atomic Compare and Swap

oldValue **atomicCAS(** address, value, newValue **);**

This will do the following in one operation:

```
oldValue = *address;
if( *address == value )
    *address = newValue;
return oldValue;
```

Concurrency Issues During Insert

2 Threads inserting in the same bucket at the same time will cause issues

Locking with AtomicCAS avoids those concurrency issues at the cost of performance

In sparse graphs, collisions during insert isn't common enough to dramatically impact performance

Warp Cooperative Work Sharing

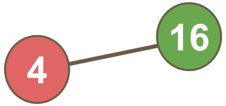
This technique involves all threads of the Warp working together to insert edges into the hash table

Terminology:

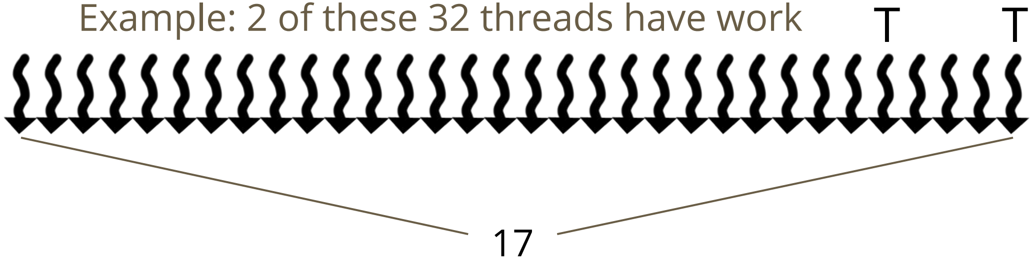
- A **Warp** is a group of 32 threads on the GPU
- Each thread has an ID from 0-31, known as a **Lane ID**
- All threads run the same code in sync

"Insert Edge": Warp Cooperative Work Sharing

- **Use __ballot_sync to query which lanes of the warp have work**



Example: 2 of these 32 threads have work

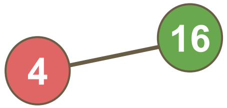


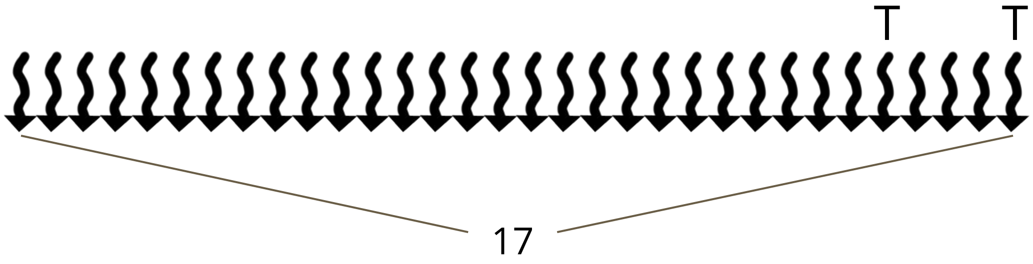

```
bool hasWork = ?
uint queue = __ballot_sync( hasWork )
```

__ballot_sync will create a 32-bit int using a bool from each thread

"Insert Edge": Warp Cooperative Work Sharing

- Use __ballot_sync to query which lanes of the warp have work
- **Use __ffs to find the first lane with work**





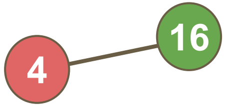

```
bool hasWork = ?
uint queue = __ballot_sync( hasWork )
int laneID = __ffs( queue )
```

__ffs will find the index of the first bit set in the ballot
i.e. the next Lane ID with work to do

Step 3/5

"Insert Edge": Warp Cooperative Work Sharing

- Use __ballot_sync to query which lanes of the warp have work
- Use __ffs to find the first lane with work
- **Use __shfl_sync to send edge info to all threads in the warp**

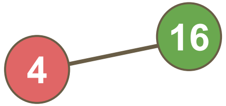


```
bool hasWork = ?
uint queue = __ballot_sync( hasWork )
int laneID = __ffs( queue )
__shfl_sync( edgeInfo, laneID )
```

Now all threads in the warp know which edge they are working together to insert

"Insert Edge": Warp Cooperative Work Sharing

- Use __ballot_sync to query which lanes of the warp have work
- Use __ffs to find the first lane with work
- Use __shfl_sync to send edge info to all 32 threads in the warp
- **All threads in the warp each check one of the slots of the slab for duplicates and for an empty slot**



V4 Bucket	31	1	-1	...	-1	*	*

bool isEmpty = ?
bool edgeExists = ?

"Insert Edge": Warp Cooperative Work Sharing

- Use __ballot_sync to query which lanes of the warp have work
- Use __ffs to find the first lane with work
- Use __shfl_sync to send edge info to all 32 threads in the warp
- All threads in the warp each check one of the slots of the slab for duplicates and for an empty slot
- **ballots are used to communicate the status and the edge is inserted if needed**

V4 Bucket	31	1	16	...	-1	*	*

```
bool isEmpty = ?
bool edgeExists = ?
uint exists = __ballot_sync( edgeExists )
uint empty = __ballot_sync( isEmpty )
```

Additional Details In Brief

- Bidirectional edges
- Adding/Deleting Vertices
- Deleting Edges
- Phase-Concurrent Operations

Memory Usage

Each bucket in our hash table holds a minimum of 30 edges using 128 bytes

Graphs with a low average degree, such as a road graphs, will make very inefficient use of available memory

Data Sets	Max Degree	Avg Degree
luxembourg_osm	6	2.1
germany_osm	13	2.1
road_usa	9	2.4
delaunay_n20	23	6.0
hollywood-2009	11,000	98.9

Results

Luxembourg data set 114599 vertices

Technique	Edges per Second (extrapolated)	
	119666	239332 (bidirectional)
1 per thread	14 million	24 Million
Warp Cooperative	9 million	14 Million

* Results from an NVIDIA GTX 970

Follow up Questions

What is the difference between a static graph and a dynamic graph?

Follow up Questions

What is the biggest advantage of using a hash table over a single array of edges per vertex?

Follow up Questions

What's a major disadvantage to using the SlabHash hash table for dynamic graphs?