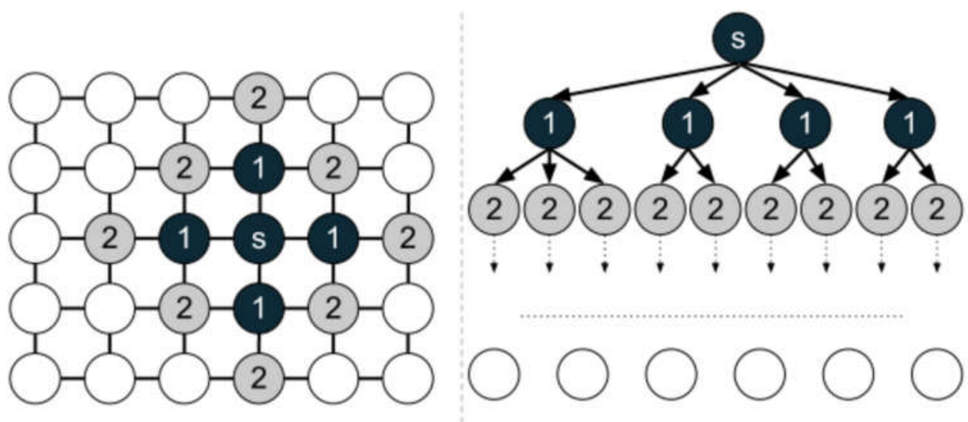
CUDA 6
Breadth-First Search

CS121 Parallel Computing

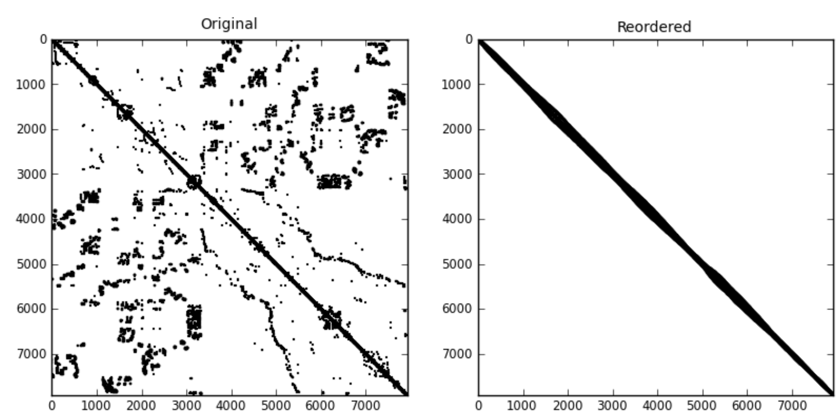
Spring 2020

Breadth-first search

- Given a graph, explore it layer by layer.
  - Go wide, then go deep.
- Large number of applications.
  - Connected components, path finding, Ford-Fulkerson max flow algorithm, Cuthill-McKee ordering, bipartiteness testing, search engine crawlers, garbage collection, etc.
- Used in benchmarks such as Graph500 and Parboil to test parallel computer's memory performance.

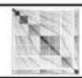
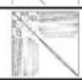
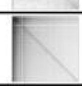
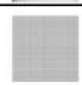


Source: http://www.stoimen.com/blog/2012/10/08/computer-algorithms-shortest-path-in-a-graph/



Source: http://dpo.github.io/pyorder/_images/commanche_dual_rcmk.png

Real world graphs

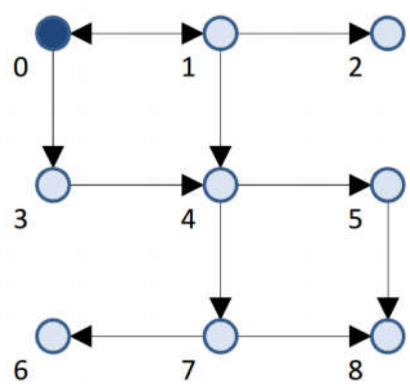
Name	Sparsity Plot	Description	n (10^6)	m (10^6)	$\bar{d}$	Avg. Search Depth
europe.osm		European road network	50.9	108.1	2.1	19314
grid5pt.5000		5-point Poisson stencil (2D grid lattice)	25.0	125.0	5.0	7500
hugebubbles-00020		Adaptive numerical simulation mesh	21.2	63.6	3.0	6151
grid7pt.300		7-point Poisson stencil (3D grid lattice)	27.0	188.5	7.0	679
nlpkkt160		3D PDE-constrained optimization	8.3	221.2	26.5	142
audikw1		Automotive finite element analysis	0.9	76.7	81.3	62
cage15		Electrophoresis transition probabilities	5.2	94.0	18.2	37
kkt_power		Nonlinear optimization (KKT)	2.1	13.0	6.3	37
coPapersCiteseer		Citation network	0.4	32.1	73.9	26
wikipedia-20070206		Links between Wikipedia pages	3.6	45.0	12.6	20
kron_g500-logn20		Graph500 RMAT (*A*=0.57, *B*=0.19, *C*=0.19)	1.0	100.7	96.0	6
random.2Mv.128Me		*G*(*n*, *M*) uniform random	2.0	128.0	64.0	6
rmat.2Mv.128Me		RMAT (*A*=0.45, *B*=0.15, *C*=0.15)	2.0	128.0	64.0	6

- Hundreds of millions of nodes and edges.
  - Some graphs have billions or trillions of edges. But these don't fit into the memory of a single GPU.
- Low average degree (sparse), but high variation in degree.
  - Some nodes have a few neighbors, some nodes 100K's.
- "Small world" graphs have low diameter (~10).
- Grids and maps have high diameter (~1-10K).

Source: *High Performance and Scalable GPU Graph Traversal*, Merrill, Garland, Grimshaw

Sequential algorithm

- Assume graph is sparse, and stored in compressed sparse row format.
  - R[i] indicates index where node i's neighbors start in C.
  - Ex R[1] = 2 means node 1's neighbors (0, 2, 4) are listed starting at C[2].
- Maintain a queue of unvisited nodes.
  - Dequeue a node, add its unvisited neighbors to the queue.
- Running time O(|V|+|E|).



C:

1	3	0	2	4	4	5	7	8	6	8
0	1	2	3	4	5	6	7	8	9	10

R:

0	2	5	5	6	8	9	9	11	11
0	1	2	3	4	5	6	7	8	9

Traversal from source vertex v_0		
BFS Iteration	***Vertex frontier***	***Edge frontier***
1	{0}	{1,3}
2	{1,3}	{0,2,4,4}
3	{2,4}	{5,7}
4	{5,7}	{6,8,8}
5	{6,8}	{}

```
      if (dist[j] == ∞)
        dist[j] := dist[i] + 1;
        Q.Enqueue(j)
```

Parallelizing BFS

- First BFS algorithms for GPUs focused on data parallelism.
- Initially set source distance to 0.
- Run for D rounds, where D is the diameter from s.
  - In round i, distance i nodes are marked.
  - Iterate through all the nodes. If a node is marked, mark its unvisited neighbors as distance i+1 nodes.
- Works well in small diameter graphs, e.g. social networks.
- Very inefficient for large diameter graphs, e.g. maps, since only a few nodes marked per round.
- $O(|V||E|)$ running time.

```
parallel for (i in V) :
  dist[i] := ∞
dist[s] := 0
iteration := 0
do :
  done := true
  parallel for (i in V) :
    if (dist[i] == iteration)
      done := false
      for (offset in R[i] .. R[i+1]-1)
        j := C[offset]
        dist[j] = iteration + 1
  iteration++
while (!done)
```

should add "if dist[j] == ∞" here

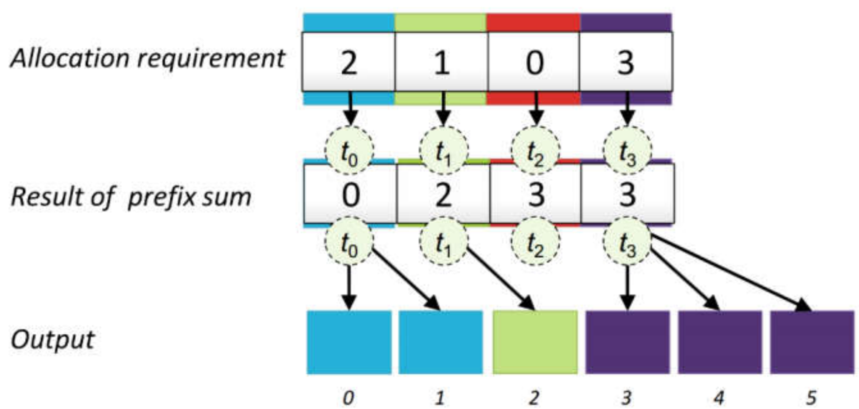
Parallelizing BFS

- Linear, i.e. $O(|V| + |E|)$ work parallel BFS algorithms follow the sequential algorithm.
- Two main bottlenecks
  - Maintaining explicit queue of unvisited nodes requires expensive LockedEnqueue operations.
  - If nodes have very different degrees (e.g. power law graphs), there's high load imbalance in the main parallel for loop.

```
parallel for (i in V) :
  dist[i] := ∞
dist[s] := 0
iteration := 0
inQ := {}
inQ.LockedEnqueue(s)
while (inQ != {}) :
  outQ := {}
  parallel for (i in inQ) :
     for (offset in R[i] .. R[i+1]-1)
        j := C[offset]
        if (dist[j] == ∞)
          dist[j] = iteration + 1
          outQ.LockedEnqueue(j)
  iteration++
  inQ := outQ
```

Gathering neighbors

- We use two queues, one for nodes in current layer of BFS, other for nodes in next layer.
  - After every phase of BFS we swap the queues, to reuse memory.
  - To synchronize the layers, use a separate kernel for each layer.
- For each node in first queue, we first add all its neighbors into the second queue (gather).
  - Some of the neighbors don't belong in the next BFS layer because they've already been visited.
    - Testing each node explicitly is inefficient.
  - Also, we may add duplicates into the second queue.
  - We'll address both problems later.
- To add neighbors of a node into the queue without expensive locks, we use prefix sum, which is much faster.
  - If node has n_i neighbors, we reserve n_i queue spots for them by adding n_i into the prefix sum.



Load balanced gathering

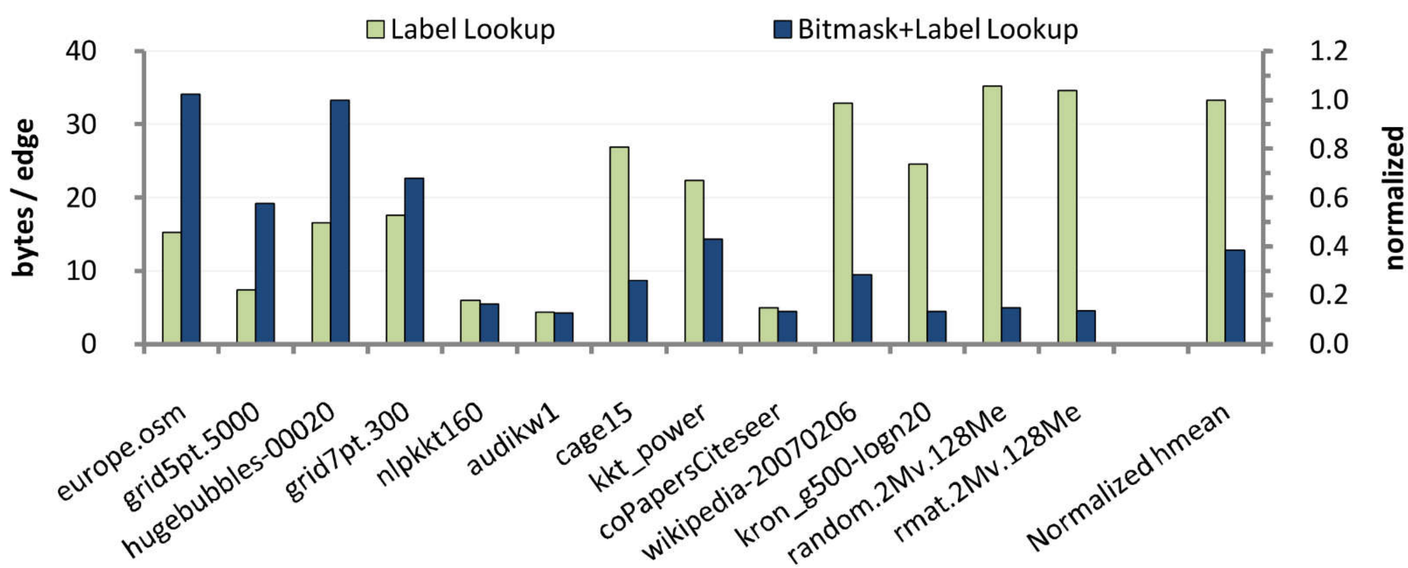
- To load balance, we assign different numbers of threads to gather the neighbors of a node in parallel.
- If node has moderate number of neighbors, assign a warp of threads to gather its neighbors.
  - Each thread in the warp might initially want to gather neighbors of a different node.
  - The warp votes to find a common node to gather.
    - All threads in warp write to a common location, then read it. The last write “wins”. Other threads help gather its node’s neighbors.
- If node has large number of neighbors, use entire thread block for gather.
- For remaining nodes, use prefix sum based method.
  - There’s load imbalance, but only for low degree nodes.
- The size of “moderate” and “large” need to be tuned.
- Eliminates most, but not all load imbalance.

Visited status lookup

- A node should only be added into the frontier queue if it hasn't been visited.
  - Before adding a node, look up its visited status.
- To reduce memory traffic, use an integer bitmask to store status of 32 nodes.
- But then two threads might "clobber" each other by setting (different) bits in the same integer.
  - Can avoid using atomics, but they're slow.
  - Instead, use normal read and write ops, but treat bitmask conservatively.
    - For each node, maintain both a shared bitmask bit, and a private integer label.
    - Usually only access bitmask, saving memory traffic. Occasionally access the label.
    - If bit for a node is set, it's definitely visited.
    - If bit is unset, then not sure about node's visited status, so do another lookup on node's label.

Visited status lookup

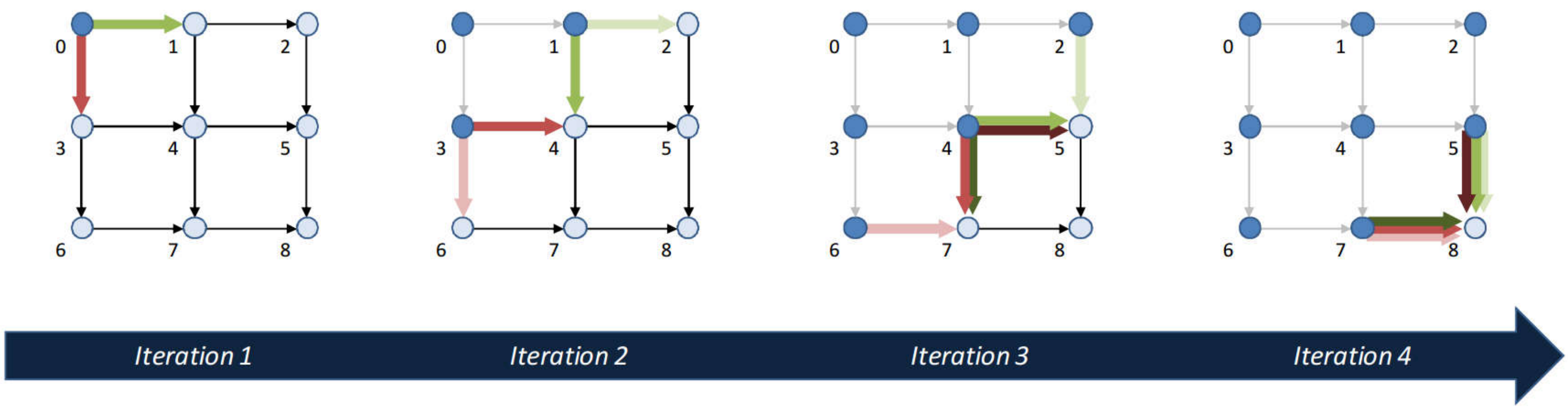
- Bitmasks are cached in texture caches.
- This is effective for low diameter graphs.
- Works less well for high diameter graphs, because each layer is processed in separate kernel, and cache flushed after each kernel launch.
- Also doesn't work well for small frontiers, since cached values aren't reused.
- Graphs on left side have high diameter; right ones are low diameter.



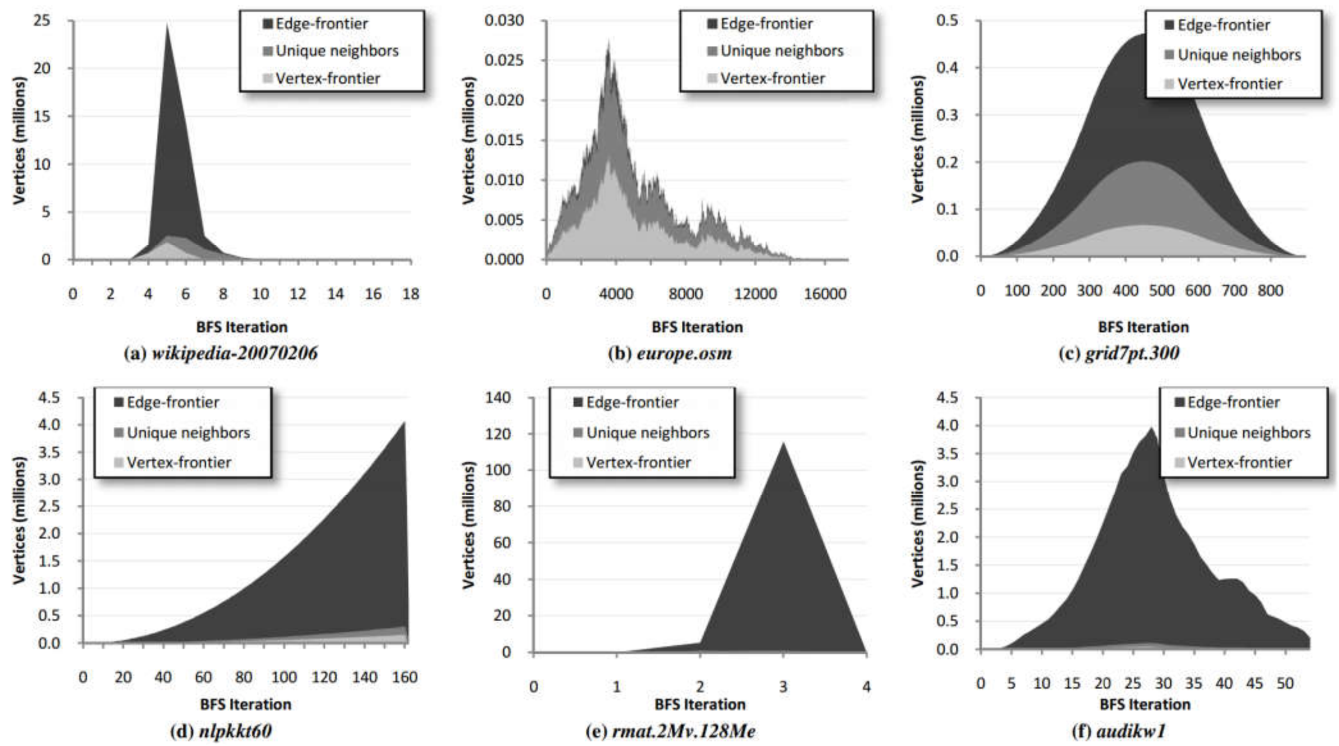
Duplicates in frontier

- May add same node into frontier multiple times, due to concurrent discovery.
- Problem especially severe in GPU because of SIMD and high parallelism.
- If duplicates aren't removed, the vertex frontier can grow exponentially.

BFS Iteration	*Actual Vertex-frontier*	*Actual Edge-frontier*
1	0	1,3
2	1,3	2,4,4,6
3	2,4,4,6	5,5,7,5,7,7
4	5,5,7,5,7,7	8,8,8,8,8,8,8



Duplicates in frontier



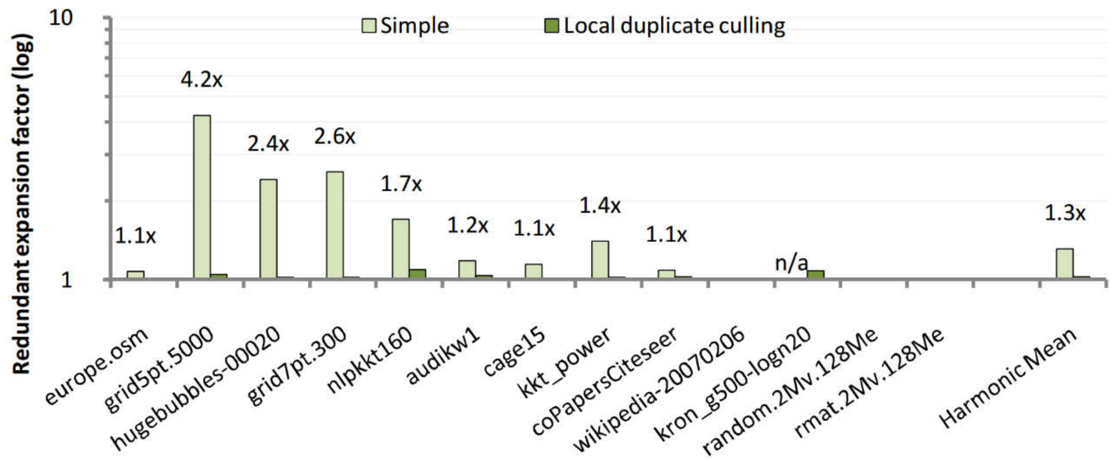
- Edge frontier: Number of nodes added to queue, allowing duplicates.
- Unique neighbors: Number of nodes added to queue, removing duplicates, but allowing visited nodes.
- Vertex frontier: Unique neighbors which haven't been visited.
- Allowing duplicates can lead to huge amount of redundant work.

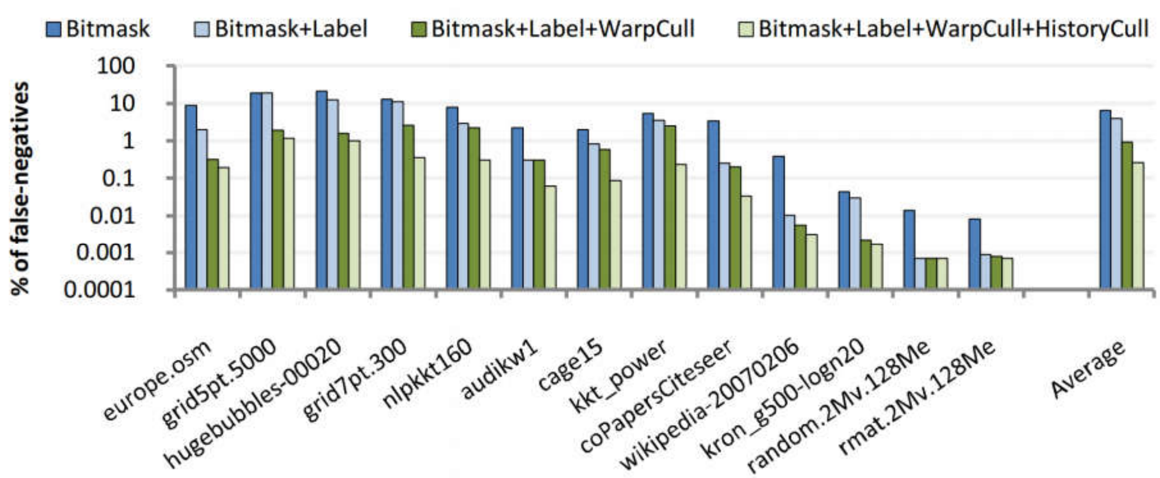
Duplicate culling

- Try to remove duplicates using hash table.
  - Won't remove all duplicates, but quite effective.
- Warp culling
  - Each warp allocates a hash table (with 128 entries) in shared memory.
  - When inserting a node, hash it into hash table.
    - If table entry empty, store the node in entry, and add node to queue.
    - If table entry filled, then if entry equals the node, don't add node to queue. Otherwise, add it.
- History culling
  - Same idea, but use the SM's L1 cache.

Duplicate culling

- Despite small hash table, culling surprisingly effective.

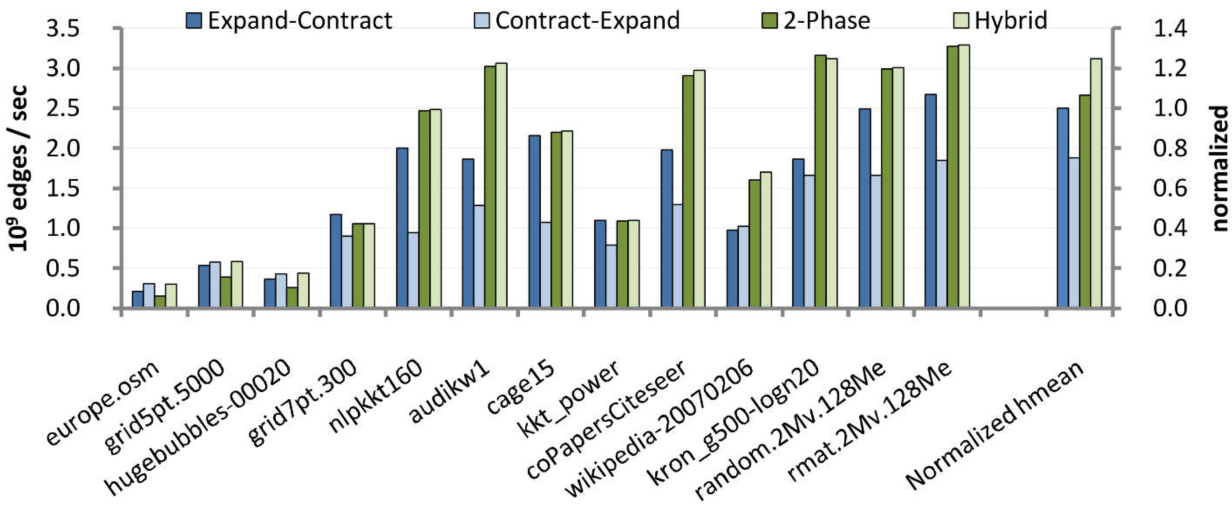




Putting it together

- Each kernel expands one layer of the BFS.
  - Input is queue containing last BFS layer (possibly with duplicate nodes).
- Threads assigned nodes from queue.
- A thread first uses warp and history culling to determine if its vertex is a duplicate.
- If not, thread gathers node's neighbors.
  - Based on neighbor list size, use a block, warp, or prefix sum gather.
    - Each thread wants to gather neighbors of a different node, and tries to "enlist" a block or warp of threads to help it.
    - Each thread writes into a variable shared by warp or block, then reads it.
    - One thread from the warp / block "wins". All other threads help it.
- Before adding a gathered node to (current layer's) queue, check if it's already visited.
- If not, the thread contributes 1 to a thread block-wide prefix sum.
- Synchronize the block and do a block-wide prefix sum to get number of enqueued nodes for block.
- First thread in block atomically adds sum to global queue index, then shares old global index with block.
- Using old global offset and prefix sum offset, each thread adds its gathered neighbor into queue.

Performance



8 core CPU

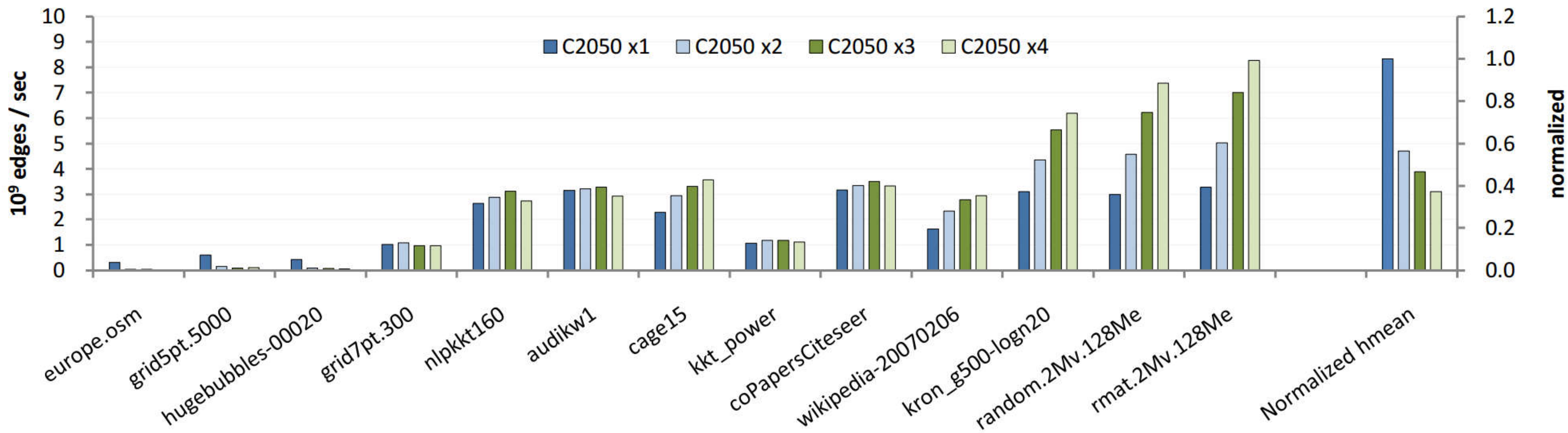
Graph Dataset	CPU Sequential†	CPU Parallel	NVIDIA Tesla C2050 (*hybrid*)			
			Label Distance		Label Predecessor	
	10^9 TE/s	10^9 TE/s	10^9 TE/s	Speedup	10^9 TE/s	Speedup
europe.osm	0.029		0.31	11x	0.31	11x
grid5pt.5000	0.081		0.60	7.3x	0.57	7.0x
hugebubbles-00020	0.029		0.43	15x	0.42	15x
grid7pt.300	0.038	0.12††	1.1	28x	0.97	26x
nlpkkt160	0.26	0.47††	2.5	9.6x	2.1	8.3x
audikw1	0.65		3.0	4.6x	2.5	4.0x
cage15	0.13	0.23††	2.2	18x	1.9	15x
kkt_power	0.047	0.11††	1.1	23x	1.0	21x
coPapersCiteseer	0.50		3.0	5.9x	2.5	5.0x
wikipedia-20070206	0.065	0.19††	1.6	25x	1.4	22x
kron_g500-logn20	0.24		3.1	13x	2.5	11x
random.2Mv.128Me	0.10	0.50†††	3.0	29x	2.4	23x
rmat.2Mv.128Me	0.15	0.70†††	3.3	22x	2.6	18x

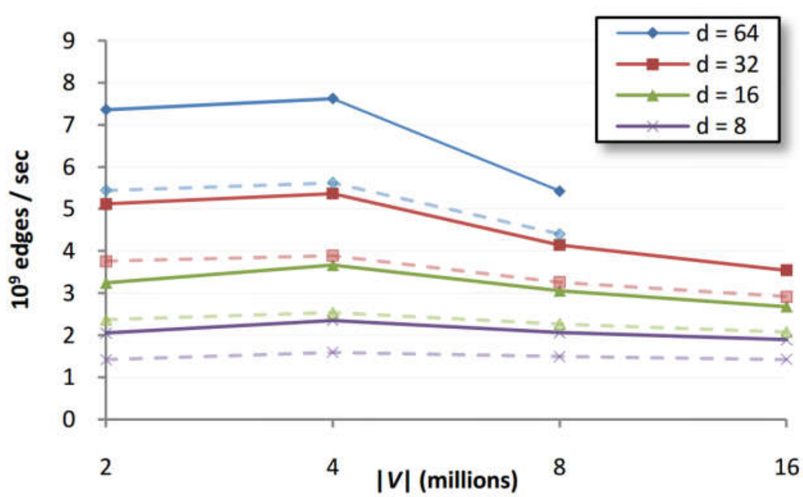
- Previous algorithm called “contract-expand”, because it first takes current layer’s edge frontier, contracts it (removes duplicates), then expands into next layer’s edge frontier (containing duplicates).
- “Expand-contract”, algorithm expands current vertex frontier, then contracts it (removes duplicates) to next layer’s vertex frontier.
- 2-phase expands then contracts in two kernels.
- Hybrid combines contract-expand with 2-phase, using 2-phase for iterations with large frontiers.
- Variants differ in amount of memory traffic, latency and parallelism.
- Hybrid’s performance is mostly determined by average degree (which generally increases moving down the dataset).

Multi-GPU BFS

- Multiple GPUs can use a single logical address space.
  - Communicate through PCI-e 2.0 (6.6 GB/s).
- Given p GPUs, assign n/p vertices and corresponding edges per GPU.
  - Vertices assigned in round robin order for load balancing.
  - Poor locality if p large.
- Each GPU expands / contracts its own vertex queue, as in the single GPU algorithm (*).
- Then sort the new frontier into p bins, corresponding to vertices from different GPUs.
- Barrier across all GPUs.
- Run p-1 kernels, where in i'th kernel, the i'th GPU collects bin i from each other GPU.
- Then go back to step (*) to form the next layer. Continue until all nodes visited.

Performance





Performance on uniform random graph. Higher average degree (d) results in better duplicate culling and higher performance.

- Only achieved speedup on graphs with small diameters and large average degrees.
  - Smaller diameter requires less synchronization.
  - Larger degree makes duplicate culling more effective.
  - Max speedups 1.5X, 2.1X and 2.5X on 2, 3, 4 GPUs.
  - Sometimes parallel algorithm performed much worse.