

# CS344 Introduction to Parallel Programming

## Lesson 6.2: Parallel Computing Pattern Part B

### L6.2-6.1-Quadratic GPU vs Serial CPU Implementation

COMPARING QUADRATIC GPU IMPL. WITH SERIAL CPU

N<sup>2</sup> GPU

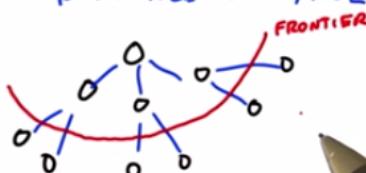
N<sup>2</sup> WORK

VISITS EVERY EDGE  
MANY TIMES BUT ONLY  
SETS DEPTH ONCE

CPU

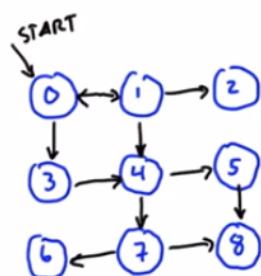
N WORK

MAINTAINS FRONTIER  
TO MINIMIZE VISITS/NODE



### L6.2-6.2-How to Represent Graph Data Structure

GRAPH DATA STRUCTURE



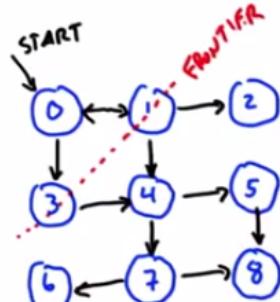
As a function of e and v...

how big is C [ ]  
R [ ] ?

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 1
	0 1 2 3 4 5 6 7 8 9

### L6.2-6.3-How Does This Algorithm Work Part 1

GRAPH DATA STRUCTURE



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 1
	0 1 2 3 4 5 6 7 8 9

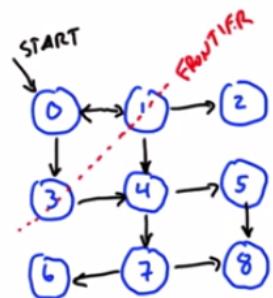
D:	0 -1 -1 -1 -1 -1 -1 -1 -1 -1
	0 1 2 3 4 5 6 7 8 9

- ① IN PARALLEL FOR EACH NODE ON THE FRONTIER,  
FIND THE STARTING POINT OF ITS NEIGHBORS.

R[v]

#### L6.2-6.4-How Does This Algorithm Work Part 2

GRAPH DATA STRUCTURE



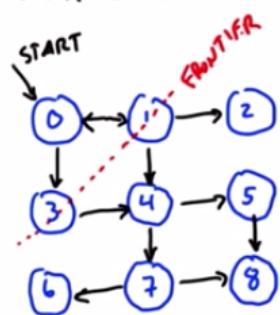
C:	1 3 0 2 4 4 5 7 8 6 8
R:	0 2 5 5 6 8 9 9 11 11
D:	0 -1 -1 1 -1 -1 -1 -1 -1

- ② FOR EACH FRONTIER NODE, CALCULATE  
# OF ITS NEIGHBORS.



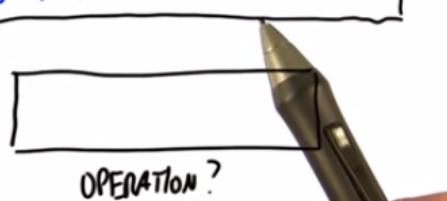
#### L6.2-6.5-How Does This Algorithm Work Part 3

GRAPH DATA STRUCTURE



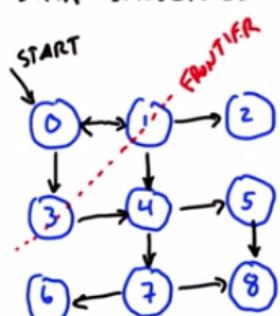
C:	1 3 0 2 4 4 5 7 8 6 8
R:	0 2 5 5 6 8 9 9 11 11
D:	0 -1 -1 1 -1 -1 -1 -1 -1

- ③ ALLOCATE SPACE TO STORE  
THE NEW FRONTIER.



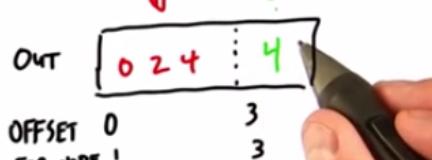
#### L6.2-6.6-How Does This Algorithm Work Part 4

GRAPH DATA STRUCTURE

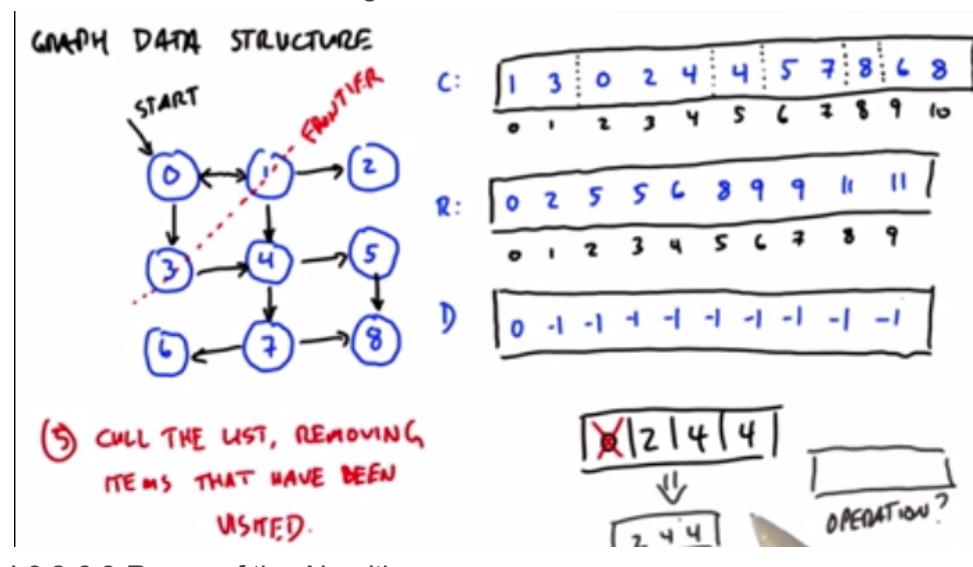


C:	1 3 0 2 4 4 5 7 8 6 8
R:	0 2 5 5 6 8 9 9 11 11
D:	0 -1 -1 1 -1 -1 -1 -1 -1

- ④ COPY EACH ACTIVE'S EDGE  
LIST TO THIS ARRAY.



### L6.2-6.7-How Does This Algorithm Work Part 5

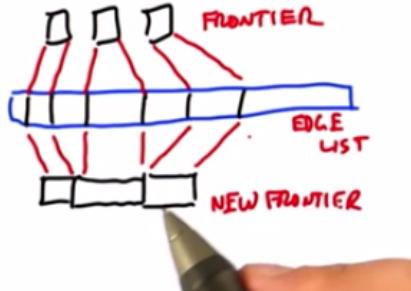


### L6.2-6.8-Recap of the Algorithm

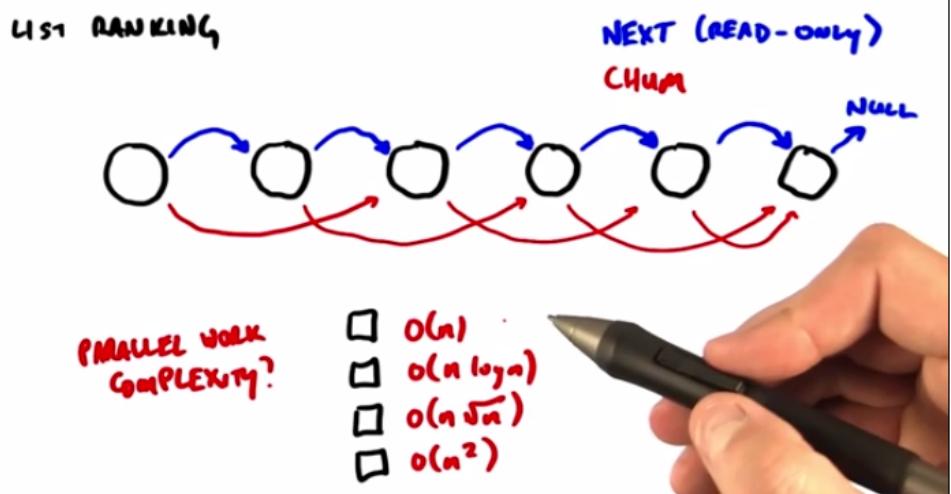
ALGORITHM SUMMARY

INITIALIZE: STARTING NODE'S DEPTH = 0  
STARTING FRONTIER = NEIGHBORS OF STARTING NODE

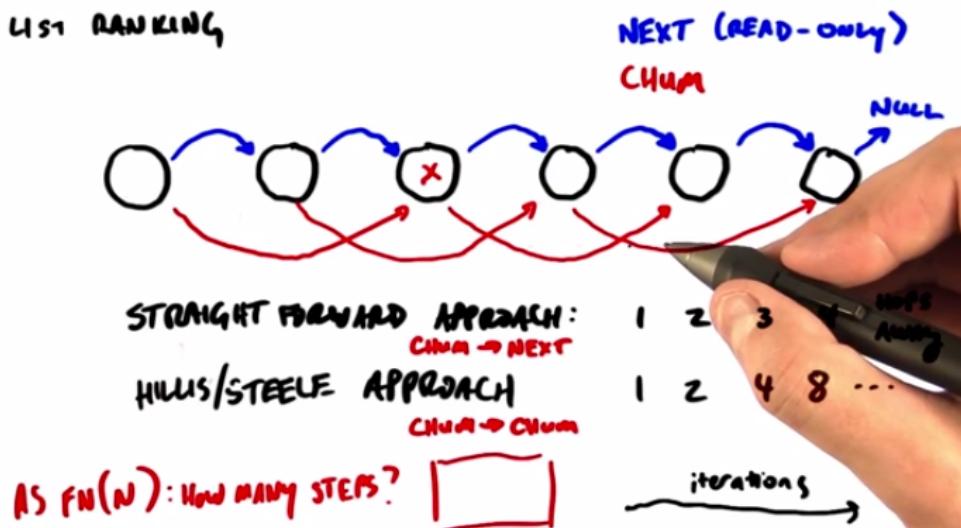
- 1 FRONTIER: FIND NEIGHBOR START : HOW MANY NEIGHBORS?
- 2
- 3 ALLOCATE SPACE FOR NEW FRONTIER
- 4 COPY EDGE LIST TO NEW ARRAY
- 5 CULL VISITED ELEMENTS



### L6.2-6.9-Merrills Linear-Complexity BFS on GPUs Part 1



## L6.2-6.10-Merrills Linear-Complexity BFS on GPUs Part 2



## L6.2-6.11-Merrills Linear-Complexity BFS on GPUs Part 3

```
find_start_node (const int *next, int *chum) {
    int k = blockDim.y * blockIdx.x + threadIdx.x;
    chum[k] = next[k];
    while ((chum[k] != NULL) && (chum[chum[k]] != NULL)) {
        chum[k] =  ; ;
    }
}
```

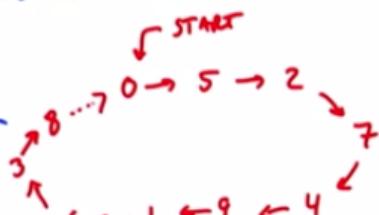
## L6.2-6.12-List Ranking Part 1

### LIST RANKING

**INPUT:** - EACH NODE KNOWS ITS SUCCESSOR  
- STARTING NODE

**OUTPUT:** - ALL NODES IN ORDER

**INPUT:**  $\text{NODE\#}$   $\begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ +1 & 5 & 6 & 7 & 8 & 9 & 2 & 3 & 4 & 0 \end{matrix}$



### L6.2-6.13-List Ranking Part 2

INPUT:      *SELECT*  
 ↓

NODE #	0	1	2	3	4	5	6	7	8	9
+1	5	6	7	8	9	2	3	4	0	1
+2										
+4										
+8										

### L6.2-6.14-List Ranking Part 3

### L6.2-6.15-List Ranking Part 4

INPUT:      *SELECT*  
 ↓

NODE #	0	1	2	3	4	5	6	7	8	9
+1	5	6	7	8	9	2	3	4	0	1
+2	2	3	4	0	1	7	8	9	5	6
+4	4	0	1	2	3	9	5	6	7	8
+8	3	4	0	1	2	8	9	5	6	7

HOW MUCH WORK TO COMPUTE  
 THIS TABLE FOR N NODES?

- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(n^3)$

### L6.2-6.16-List Ranking Part 5

INPUT:      *SELECT*  
 ↓

NODE #	0	1	2	3	4	5	6	7	8	9	ANSWER	
+1	5	6	7	8	9	2	3	4	0	1	0 5 2 7 4	
+2	2	3	4	0	1	7	8	9	5	6	9 1 6 3 8	
+4	4	0	1	2	3	9	5	6	7	8		
+8	3	4	0	1	2	8	9	5	6	7		
OUTPOS	0					1						

INPUT:      *SELECT*  
 ↓

NODE #	0	1	2	3	4	5	6	7	8	9	ANSWER	
+1	5	6	7	8	9	2	3	4	0	1	0 5 2 7 4	
+2	2	3	4	0	1	7	8	9	5	6	9 1 6 3 8	
+4	4	0	1	2	3	9	5	6	7	8		
+8	3	4	0	1	2	8	9	5	6	7		
OUTPOS	0					1						

INPUT:      *start*

NODE #	0	1	2	3	4	5	6	7	8	9	ANSWER
+1	5	6	7	8	9	2	3	4	0	1	0 5 2 7 4
+2	2	3	4	0	1	7	8	9	5	6	9 1 6 3 8
+4	4	0	1	2	3	9	5	6	7	8	
+8	3	4	0	1	2	8	9	5	6	7	
OUTPOS	0	2		1	3						

INPUT:      *start*

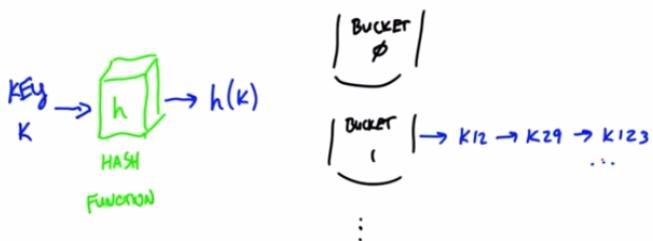
NODE #	0	1	2	3	4	5	6	7	8	9	ANSWER
+1	5	6	7	8	9	2	3	4	0	1	0 5 2 7 4
+2	2	3	4	0	1	7	8	9	5	6	9 1 6 3 8
+4	4	0	1	2	3	9	5	6	7	8	
+8	3	4	0	1	2	8	9	5	6	7	
OUTPOS	0	6	2	8	4	1	7	3	9	5	
	↓										
	0	5	2	7	4	9	1	6	3	8	

HOW MANY STEPS?

- $\sqrt{n}$
- $\log n$
- $n$
- $n^2$

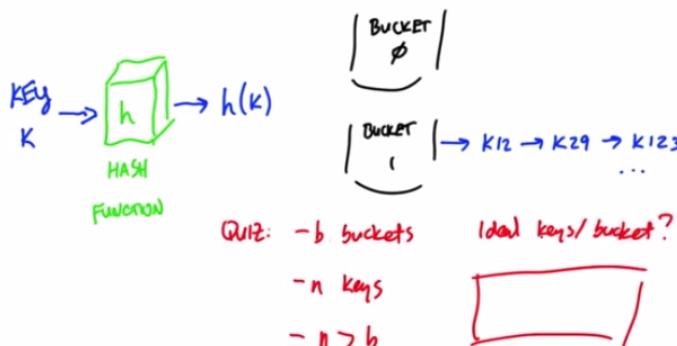
### L6.2-6.17-Hash Table on CPUs

#### CPU HASH TABLE CONSTRUCTION



### L6.2-6.18-Ideal Keys Per Bucket

#### CPU HASH TABLE CONSTRUCTION

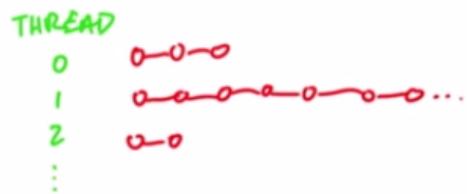


### L6.2-6.19-Chaining Is Bad

### L6.2-6.20-Why Chaining is Bad

#### CHAINING IS BAD

- 1 LOAD IMBALANCE



QUIZ: 32 THREADS, EACH LOOKING UP A DIFFERENT KEY IN THE SAME 32-ITEM BUCKET

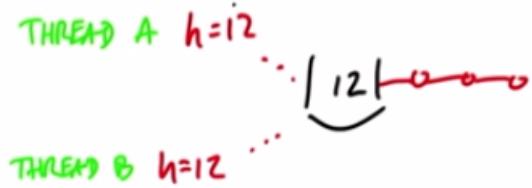
WHAT FRACTION OF TIME DO THREADS DO USEFUL WORK?



### L6.2-6.21-Chaining Is Bad for Hash Table Construction

#### CHAINING IS BAD

- 1 LOAD IMBALANCE
- 2 CONTENTION IN CONSTRUCTION



### L6.2-6.22-Cuckoo Hasing Part 1

#### CUCKOO HASHING

##### KEY IDEAS

- MULTIPLE HASH TABLES (NOT 1)
- MULTIPLE HASH FUNCTIONS (1 PER HASH TABLE)

ONLY ONE ITEM PER BUCKET!  
NO CHAINING

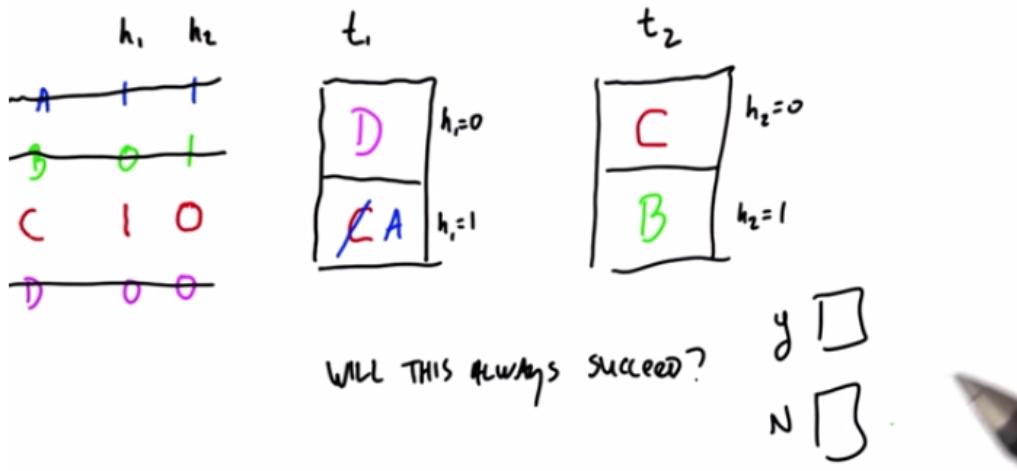
##### PROCEDURE

- 1ST ITERATION: ALL ITEMS USE  $h_1$  AND WRITE INTO  $t_1$ ,  
SOME FAIL!
- 2ND : THOSE THAT FAIL USE  $h_2, t_2$
- REPEAT



## L6.2-6.23-Cuckoo Hashing Part 2

### CUCKOO EXAMPLE



## L6.2-6.24-Cuckoo Hashing Part 3

### IMPLEMENTATION NOTES

- CONSTANT TIME LOOKUPS — NO DIVERGENCE

- CONSTRUCTION IS SIMPLE & FAST

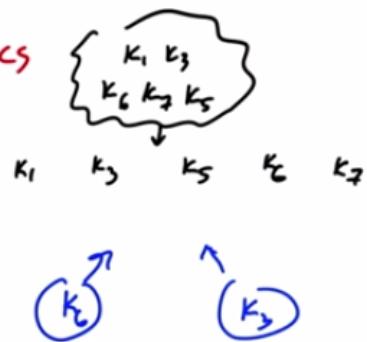
— SHARED MEM: REALLY FAST

— GLOBAL MEM: REQUIRES ATOMICS

- ALTERNATE ALGORITHM:

— CONSTRUCTION: SET

— LOOKUP: BINARY SEARCH



## L6.2-6.25-Conclusion

### CONCLUSION

- DENSE N-BODY

• MINIMIZE GLOBAL MEM BW

• REDUCE PARALLELISM BY INCREASING WORK/THREADS

---

### Conclusion

- DENSE N-BODY
- SPARSE MATRIX X VECTOR
  - DATA STRUCTURE
  - REDUCE LOAD IMBALANCE
  - KEEP GPU BUSY



### Conclusion

- DENSE N-BODY
- SPARSE MATRIX X VECTOR
- BFS
  - CHOOSE EFFICIENT ALGORITHMS
  - IRREGULAR EXPANSION / CONTRACTION : SCAN



### Conclusion

- DENSE N-BODY
- SPARSE MATRIX X VECTOR
- BFS
- LIST RANKING,
  - TRADE MORE WORK FOR FEWER STEPS
  - POWER OF SCAN

### Conclusion

- DENSE N-BODY
- SPARSE MATRIX X VECTOR
- BFS
- LIST RANKING,
- HASH TABLE
  - PARALLEL-FRIENDLY DATA STRUCTURE

