ECE 508
Manycore Parallel Algorithms

Lecture 8: Triangle Counting for Graph Analytics

Started Simple: with BFS

- In the last two lectures, we explored BFS,
  - a quick introduction to dynamic extraction and graph problems,
  - but not the most commonly used algorithm
  - (perhaps that might be shortest path?).

Objective

- to become familiar with parallel graph analytics algorithms

- to understand triangle counting on undirected graphs,
  - a building block for community detection,
  - consider algorithm alternatives, and
  - discuss multi-GPU parallelization

- to understand a use case: truss decomposition

Graphs Used to Represent Many Things

Graphs are used to describe a myriad of relationships.

- computational science relevance (example: bipartite graph between grid points and atoms within cutoff)
- physical path/location connectivity
- temporal relations between road use, movies, products, web pages
- social connections and relationships
- causal relations between events
- and many more…

Can Obtain Information from Graph Structure

One can mine a graph to get high-level information.

- Which communities does an individual X belong to?
- Who are the leaders of community X?
- Which products did people buy after viewing product X?
- Which freeway sections are bottlenecks in X area?

And so forth.

Graph Analysis Enables Insights

Graph analysis produces "value" of many companies.

- "Which web pages include <keyword>?" becomes "Which web page about <keyword> am I likely to want to read now?"
- "What smartphones are available?" becomes
  - "What's the best deal on the phones I like?", and
  - "Which screen protector should I buy," and
  - "Do I need a portable battery?" and …

Companies that answer the **transformed questions attract more customers**.

© Steven S. Lumetta, ECE508/CS508/CSE508, 2021

Want to Find Communities in Graphs

- Some **graphs** already **have structured data**.
- Social networks, for example,
  - often have groups that individuals can join.
  - Group members are tagged as such.
- But **informal communities** are
  - often **as important or more important**, and
  - these **must be derived** from graph structure.
- Keep in mind that **a community is a set of nodes**—products, roads, movies, or anything else—not just people.

Triangle Counting is Example and Building Block

We use **triangle counting as an example** of analysis.

- Triangles are the **foundation for finding communities**.
- Show how we can do more work on a graph in order to extract higher-level information.
- And talk about a use case: truss decomposition.

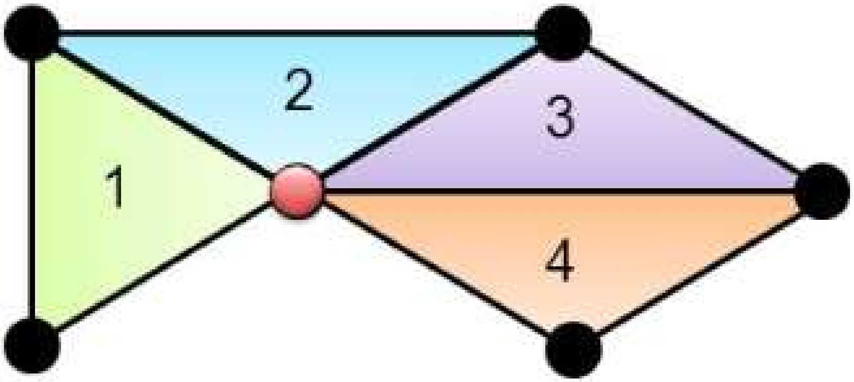
Count Triangles to Measure Community Strength

- Count the number TC of triangles in a graph.

What's a triangle?

A clique on 3 nodes.

- A foundational function for community analysis:
  - **small TC means** the **community** is **weak**, while
  - **large TC means** the **community** is **strong**.



© Wen-mei W. Hwu , David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021

Examine Two Approaches to Counting

Approaches to obtaining a graph's TC:

- **linear algebra**: use matrices to count triangles, and
- **neighbor intersection**: measure intersections between edge neighbor lists.

These are **equivalent**, so

- we can **mix** the approaches
- **to find** an **optimal** strategy.

Other approaches exist, such as counting graph isomorphisms, but we discuss only these two.

© Wen-mei W. Hwu , David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021

Counting on Undirected Graph is Inefficient

Start with a simple, undirected graph.

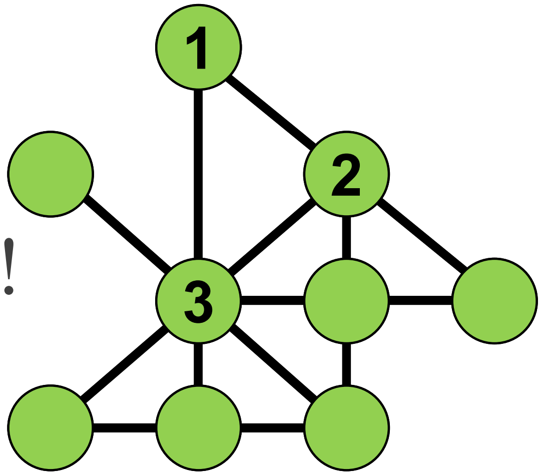
What happens if we count triangles as discussed?

Let **N(n)** be the set of neighbors of node **n**.

For the triangle 1-2-3,

- **3 ∈ N(1) ∩ N(2)**, so **count** it!
- **2 ∈ N(1) ∩ N(3)**, so **count** … **again**!
- **1 ∈ N(2) ∩ N(3)**, a **third time**!

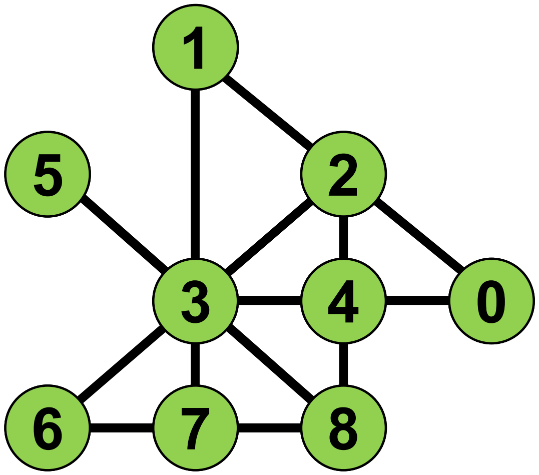
When done, divide by three.



© Steven S. Lumetta, ECE508/CS508/CSE508, 2021

Use Total Order to Transform to Directed

- Instead, **choose a total order** on nodes.
  - Any order will do.
  - For example, node number:
  - dst > src.
- **Keep** only **edges that obey** the **rule**.



Transform into directed graph.

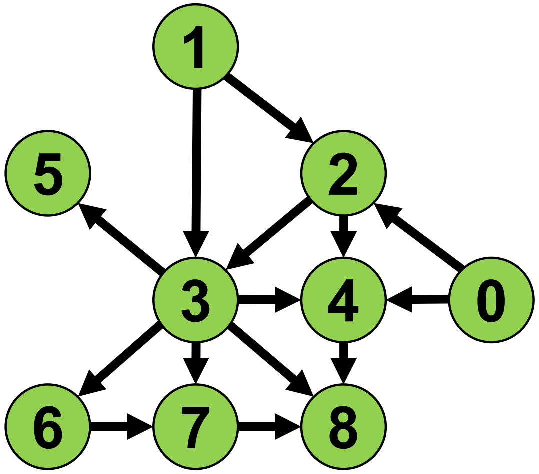
© Steven S. Lumetta, ECE508/CS508/CSE508, 2021

Resulting Graph Avoids Triple Counting

Now count triangles again.

For the triangle 1-2-3,

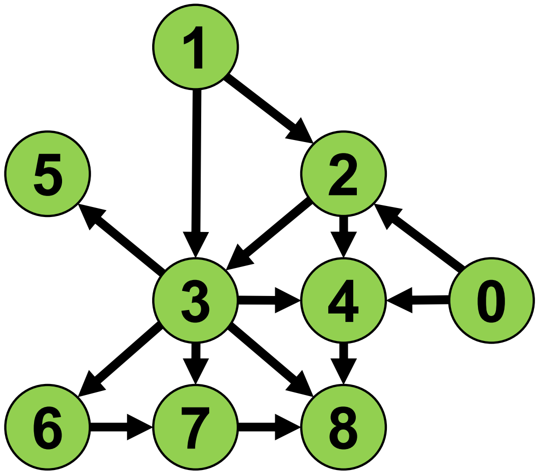
- **3 ∈ N(1) ∩ N(2)**, so **count** it!
- **N(1) ∩ N(3)** is now empty.
- And **N(2) ∩ N(3) = {4}** (no 3).



Each triangle counted once!

Graph Rewriting Not Strictly Necessary

Do we need to rewrite the graph?

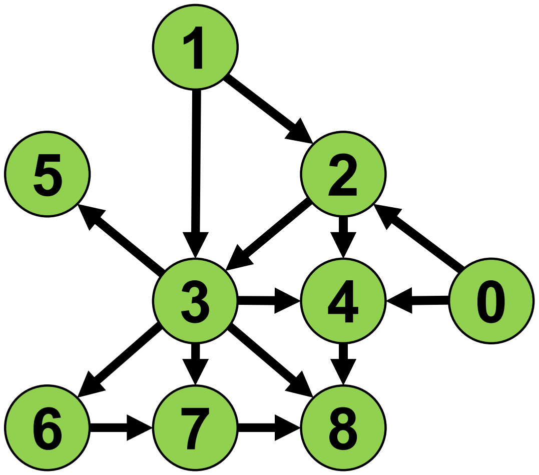


Not necessarily.

For example,

- if we pick node number as the total order,
- we **can ignore edges** with dst < src
- **while counting** on the original graph.

Graphs Represented as Adjacency Matrices



destination

source	0	1	2	3	4	5	6	7	8
0			1		1				
1			1	1					
2				1	1				
3					1	1	1	1	1
4									1
5									
6								1	
7									1
8									

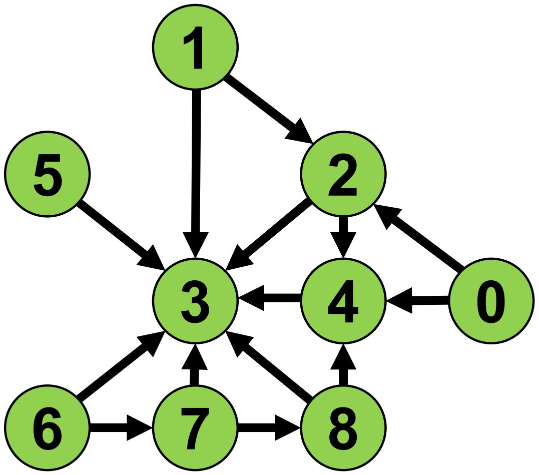
Practically,

- **use** the **adjacency matrix**.
- Here it’s upper diagonal.

Many Options Possible for Total Order

- We saw use of node number:
  - dst > src : upper-triangular matrix, and
  - dst < src : lower-triangular matrix.
- Alternatively, we could use node degree:
  - d(dst) > d(src), or by node number for ties.

Adjacency Matrix with Degree as Total Order



Using node degree,
we obtain this graph.

destination

source	0	1	2	3	4	5	6	7	8
0			1		1				
1			1	1					
2				1	1				
3									
4				1					
5				1					
6				1				1	
7				1					1
8				1	1				

© Wen-mei W. Hwu , David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021

Side-by-Side Adjacency Matrix Comparison

destination

source	0	1	2	3	4	5	6	7	8
0			1		1				
1			1	1					
2				1	1				
3					**1**	**1**	**1**	**1**	**1**
4									**1**
5									
6								1	
7									1
8									

dst > src

destination

source	0	1	2	3	4	5	6	7	8
0			1		1				
1			1	1					
2				1	1				
3									
4				**1**					
5				**1**					
6				**1**				1	
7				**1**					1
8				**1**	**1**				

d(dst) > d(src)

© Wen-mei W. Hwu , David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021

Different Orders Give Different Properties

- What's the **advantage of** using **node numbers?**
  - No need for graph-wide properties, so can
  - **stream through graph** and write out in one pass.

- What's the **advantage of** using **degree?**
  - **Fewer high-degree nodes**, thus
  - potentially **better load balancing**, but
  - need whole graph to do conversion.

Counting Triangles with Linear Algebra

Given adjacency matrix **A**,

- **compute (A×A)·A**, where
- × is matrix multiplication, and
- · is element-wise multiplication.

Let's do a couple of examples…

Element (6,3) Counts One Triangle

		1		1				
		1	1					
			1	1				
			1					
			1					
			1				1	
			1					1
			1	1				

×

		1		1				
		1	1					
			1	1				
			1					
			1					
			1				1	
			1					1
			1	1				

Compute element **(6,3)**.

- Inner product sums to 1.
- **A(6,3) = 1**, so per-element multiplication gives **1**.

Element (0,3) Counts Zero Triangles

		1		1				
		1	1					
			1	1				
			1					
			1					
			1				1	
			1					1
			1	1				

×

		1		1				
		1	1					
			1	1				
			1					
			1					
			1				1	
			1					1
			1	1				

Compute element **(0,3)**.

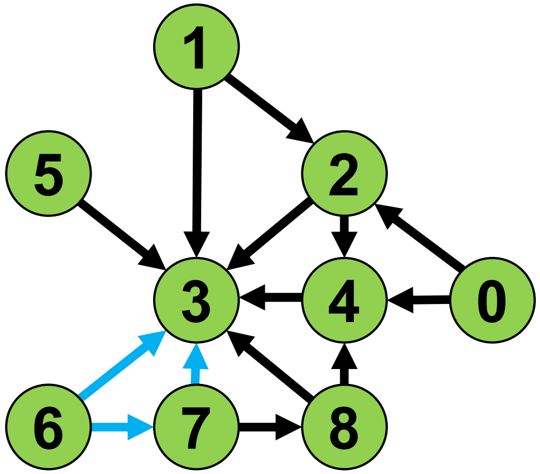
- Inner product sums to 2.
- **A(0,3) = 0**, so per-element multiplication gives **0**.

© Wen-mei W. Hwu , David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021

How Does the Computation Work?

Element (6,3)

- produced **one triangle**.
- It’s shown to the right.



Element (0,3)

produced **no triangles**.

Does the computation work?

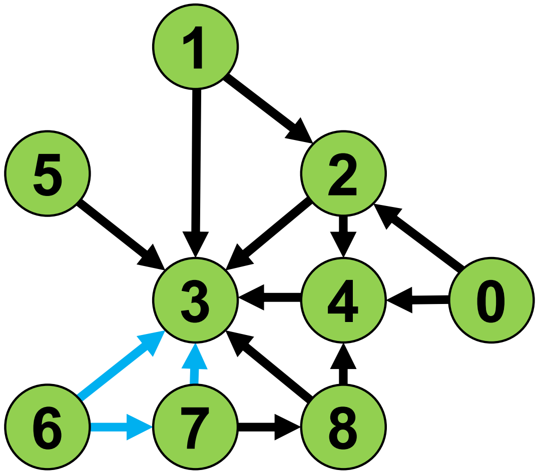
Combine Two-Hop Paths with Edge Information

Element **(u,v)** from matrix multiply

$$\mathbf{P}_{\mathbf{uv}} = \sum_{w=1}^{N} A_{uw} A_{wv}$$

computes the **number of two-hop paths** from **u** to **v**.

The **element-wise multiplication retains** only those **elements for which (u,v) is in** the **graph**.



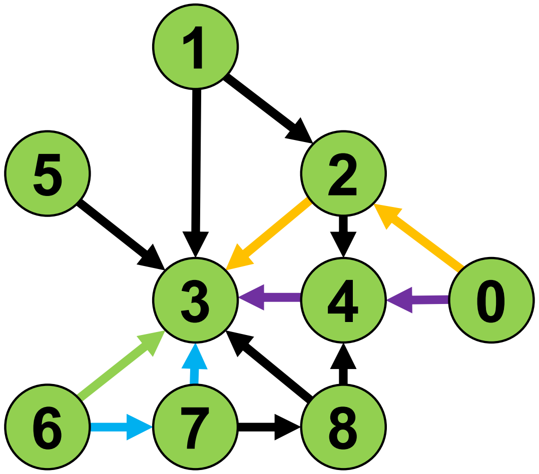
Each Element Counts One Edge's Triangles

Element (6,3)

- one 2-hop path: **6→7→3**,
- and **(6,3)** is in graph, so
- one triangle.

Element (0,3)

- two 2-hop paths
  - **0→2→3**
  - **0→4→3**
- but (0,3) is not in the graph!



© Wen-mei W. Hwu , David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021

Complete Computation Gives Six Triangles

$$\left(\begin{array}{|c|c|c|c|c|c|c|c|c|}\hline
 & & 1 & & 1 & & & & \\ \hline
 & & 1 & 1 & & & & & \\ \hline
 & & & 1 & 1 & & & & \\ \hline
 & & & & & & & & \\ \hline
 & & & 1 & & & & & \\ \hline
 & & & 1 & & & & & \\ \hline
 & & & 1 & & & & 1 & \\ \hline
 & & & 1 & & & & & 1 \\ \hline
 & & & 1 & 1 & & & & \\ \hline
\end{array}\right)^{2} = \begin{array}{|c|c|c|c|c|c|c|c|c|}\hline
 & & & 2 & 1 & & & & \\ \hline
 & & & 1 & 1 & & & & \\ \hline
 & & & 1 & & & & & \\ \hline
 & & & & & & & & \\ \hline
 & & & & & & & & \\ \hline
 & & & & & & & & \\ \hline
 & & & 1 & & & & & 1 \\ \hline
 & & & 1 & 1 & & & & 1 \\ \hline
 & & & 1 & & & & & \\ \hline
\end{array}$$

Squared adjacency matrix shown on right.

Highlighted terms retained in element-wise multiplication.

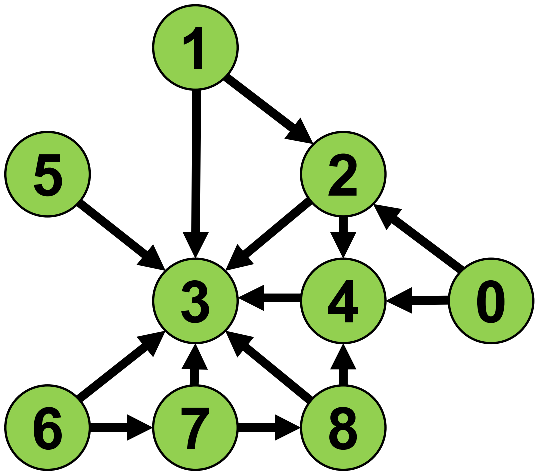
Six triangles total.

Algebraic Approach Slow for Sparse Graphs

- The idea is **a starting point**:
  - reasonable for dense matrices, but
  - highly **inefficient for sparse matrices**.

- In practice, for example,
  - only compute matrix elements that matter
  - (no edge, no computation).

Can Also Count Using Neighbor Set Intersection



destination

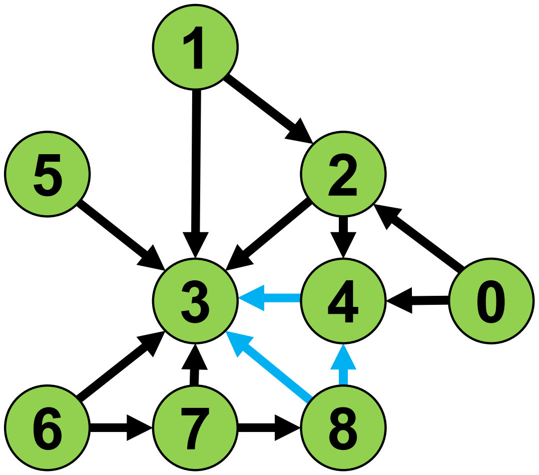
source	0	1	2	3	4	5	6	7	8
0			1		1				
1			1	1					
2				1	1				
3									
4				1					
5				1					
6				1				1	
7				1					1
8				1	1				

For each edge **(u,v)**,

count | N(u) ∩ N(v) | triangles

(cardinality of intersection of neighbors of **u** and **v**).

Edge (8,4) Produces One Triangle



destination

source	0	1	2	3	4	5	6	7	8
0			1		1				
1			1	1					
2				1	1				
3									
4				1					
5				1					
6				1				1	
7				1					1
8				1	1				

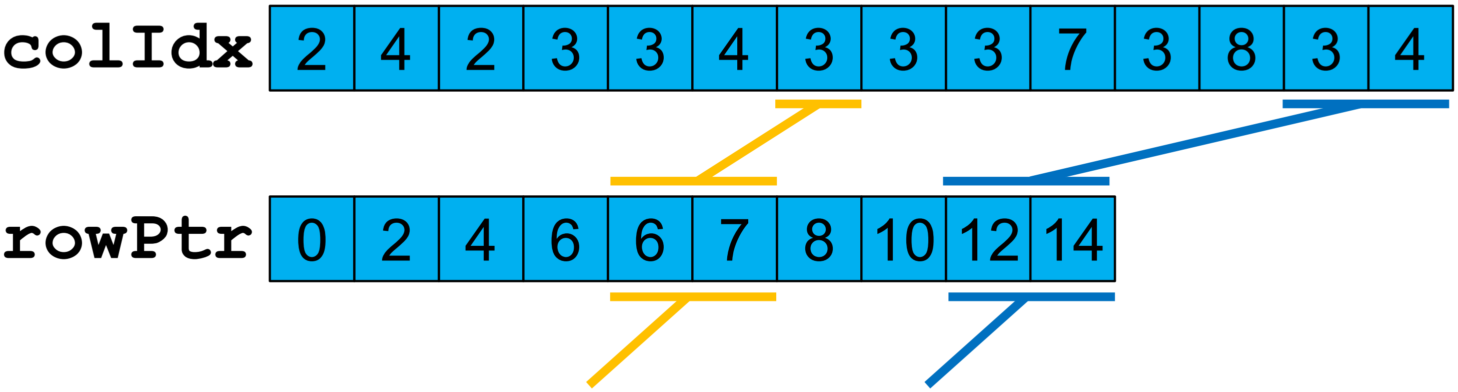
Consider edge **(8,4)**:

N(8) ∩ N(4)

= {3,4} ∩ {3} = {3} (**one triangle**)

© Wen-mei W. Hwu , David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021

Execute Intersection on CSR Format



colIdx	2	4	2	3	3	4	3	3	3	7	3	8	3	4
rowPtr	0	2	4	6	6	7	8	10	12	14				

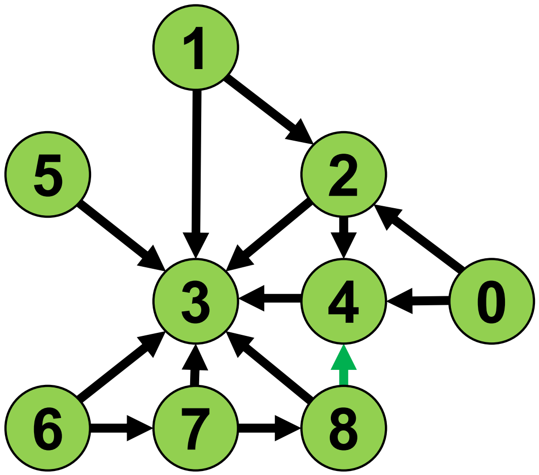
Let’s look at our graph in CSR form.

Consider edge **(8,4)**:

Node 8 … {3, 4}

Node 4 … {3}

Intersect to find one triangle.



© Wen-mei W. Hwu , David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021

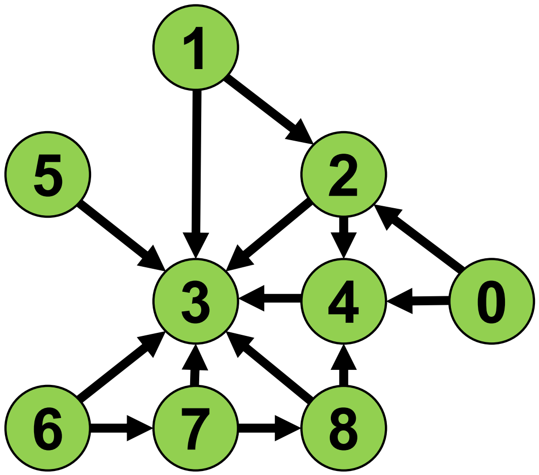
Add COO-Style Row Indices to Find Edges

`rowIdx`	0	0	1	1	2	2	4	5	6	6	7	7	8	8
`colIdx`	2	4	2	3	3	4	3	3	3	7	3	8	3	4
`rowPtr`	0	2	4	6	6	7	8	10	12	14				

Parallelize over **edges** (**`colIdx`** array), but … **`colIdx`** gives **v** from **(u,v)**.

Where is u?

Add an array of row indices!



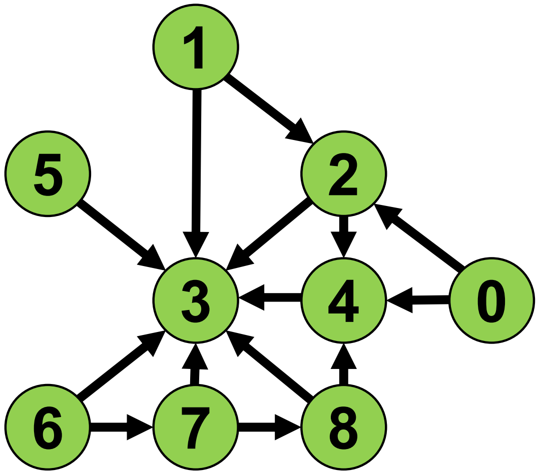
Still Need rowPtr Array to Find Neighbors

`rowIdx`	0	0	1	1	2	2	4	5	6	6	7	7	8	8
`colIdx`	2	4	2	3	3	4	3	3	3	7	3	8	3	4

`rowPtr`	0	2	4	6	6	7	8	10	12	14

Why do we still need rowPtr?

To find N(u) and N(v)!



© Wen-mei W. Hwu , David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021

How Does One Find an Intersection?

Now we have two neighbor lists…

How do we compute set intersection?

Let $\mathbf{U} \equiv |\mathbf{N(u)}|$, $\mathbf{V} \equiv |\mathbf{N(v)}|$.

Assume w.l.o.g. that $\mathbf{U} \leq \mathbf{V}$.

Are the two lists sorted?

Intersecting Sorted Lists is Fast

(**sorted lists**, **U ≤ V**)

- **linear search**
  - similar to merge sort
  - **complexity O(U + V) = O(V)**
  - parallelization a bit tricky—discussed later
- **binary search** (find elements of **U** in **V**)
  - **complexity O(U log V)**
  - parallelizes more easily,
    but better with dynamic parallelism

Intersecting Unsorted Lists is Less Fast

(**unsorted lists**, **U ≤ V**)

- **linear search**
  - **complexity O(UV)**
  - easy to parallelize
- **hash**
  - **complexity O(U + V) = O(V)**
  - requires **O(U) extra memory**
  - easy to parallelize lookups

Unsorted Lists Can Be Sorted

(**unsorted lists**, **U ≤ V**)

- **sort both** lists **first**,
  - then **use linear search**
  - **complexity O(U log U + V log V) = O(V log V)**
- **sort (N(U)) first**,
  - then **use binary search**
  - **complexity O(U log U + V log U) = O(V log U)**

Sorting is a completely different kernel!

Pseudo-Code for a Triangle Counting Kernel

(**thread T** executes the following)

```
TC ← 0
uPtr ← rowPtr[rowIdx[T]]
uEnd ← rowPtr[rowIdx[T] + 1]
vPtr ← rowPtr[colIdx[T]]
vEnd ← rowPtr[colIdx[T] + 1]
$w_1$ ← colIdx[uPtr]
$w_2$ ← colIdx[vPtr]
```

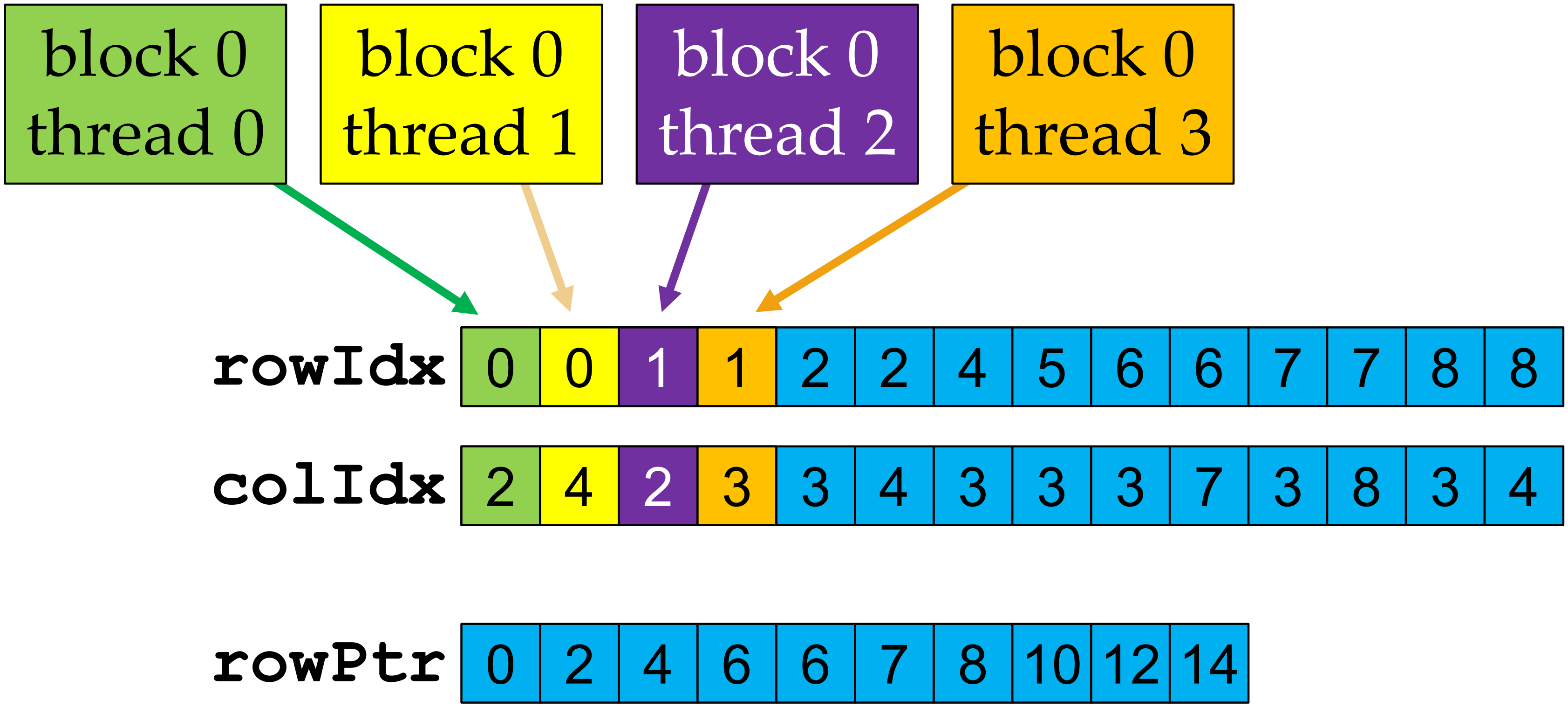
Warning: includes a bug.

Pseudo-Code for a Triangle Counting Kernel

(**thread T** executes the following)

```
while (uPtr < uEnd) ^ (vPtr < vEnd)
    if $w_1 < w_2$ then $w_1 \leftarrow$ colIdx[++uPtr]
    else if $w_1 > w_2$ then $w_2 \leftarrow$ colIdx[++vPtr]
    else if $w_1 = w_2$ then
        $w_1 \leftarrow$ colIdx[++uPtr]
        $w_2 \leftarrow$ colIdx[++vPtr]
        ++TC
    end if
end while
```

Each Thread Handles One Edge



block 0 thread 0

block 0 thread 1

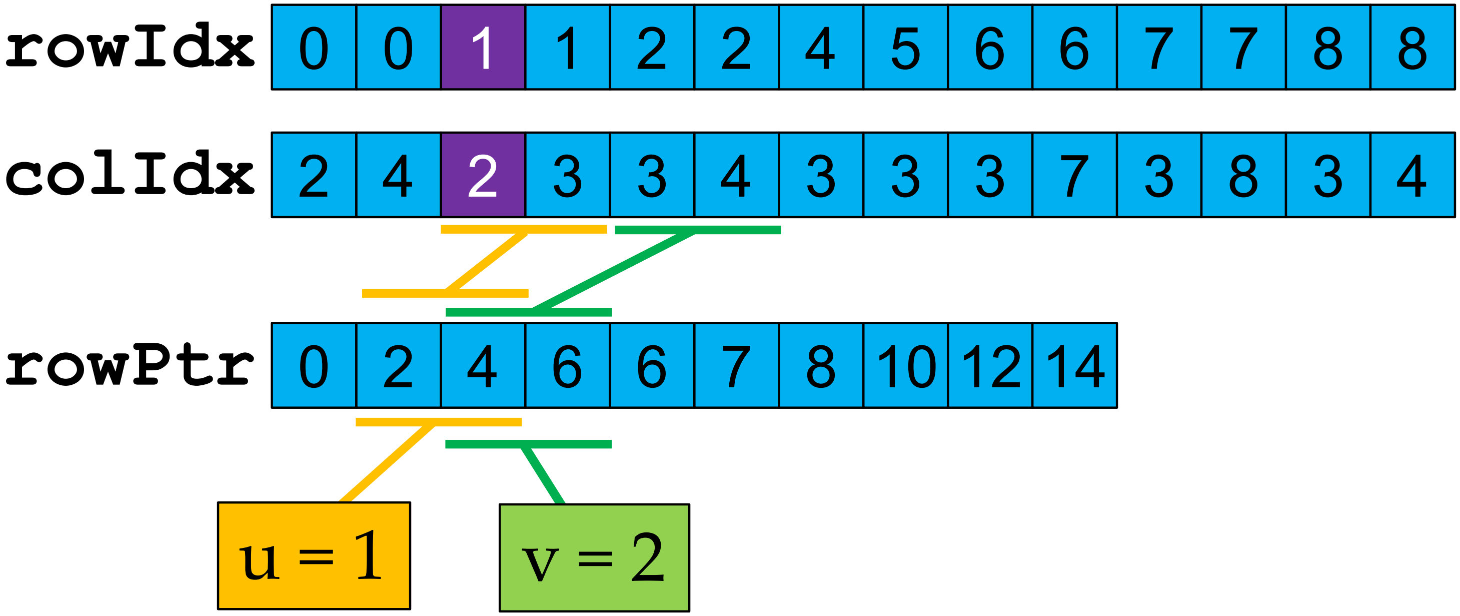
block 0 thread 2

block 0 thread 3

rowIdx	0	0	1	1	2	2	4	5	6	6	7	7	8	8
colIdx	2	4	2	3	3	4	3	3	3	7	3	8	3	4
rowPtr	0	2	4	6	6	7	8	10	12	14				

© Wen-mei W. Hwu , David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021

Thread 2 Looks Up Neighbor List Indices

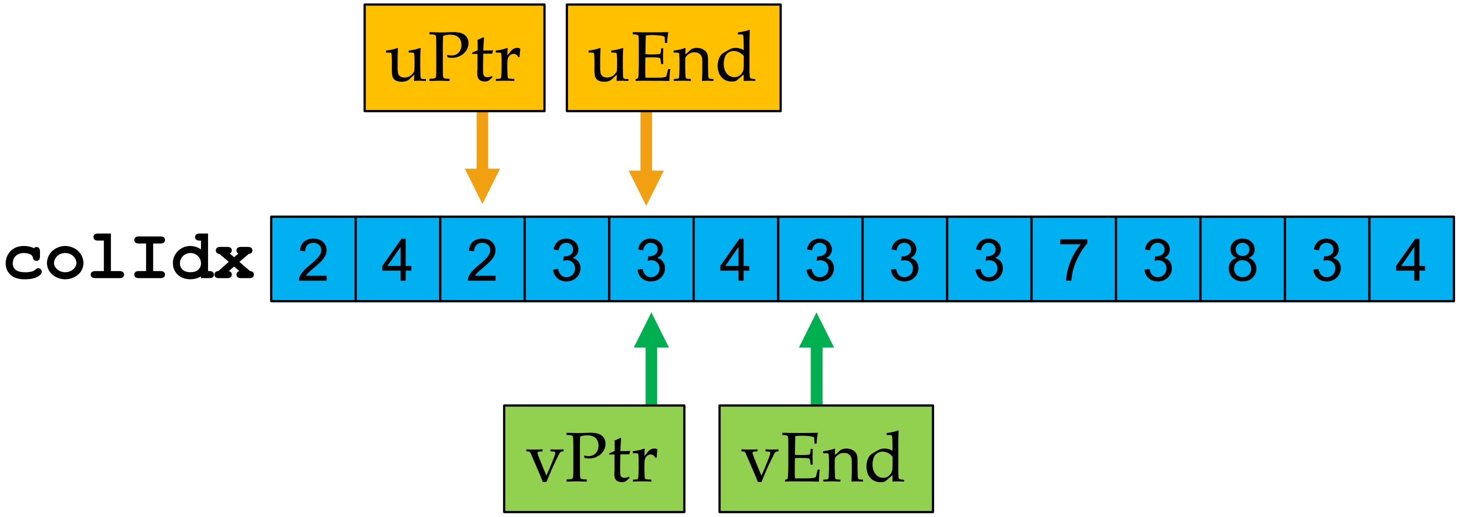


rowIdx	0	0	1	1	2	2	4	5	6	6	7	7	8	8
colIdx	2	4	2	3	3	4	3	3	3	7	3	8	3	4
rowPtr	0	2	4	6	6	7	8	10	12	14				

u = 1

v = 2

Thread 2 Executes the Search Loop (Iter. #1)



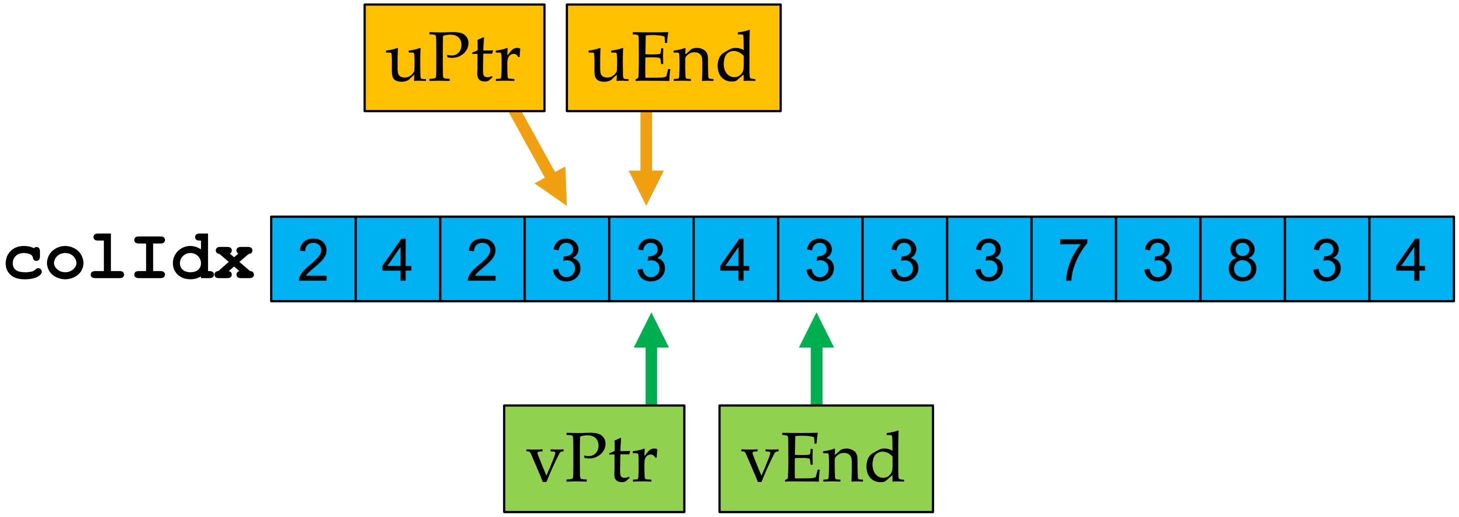
colIdx	2	4	2	3	3	4	3	3	3	7	3	8	3	4

TC = 0, **w_1 = 2**, **w_2 = 3**

$w_1 < w_2$

Advance uPtr and **set w_1 to 3**.

Thread 2 Executes the Search Loop (Iter. #2)



uPtr uEnd

colIdx													
2	4	2	3	3	4	3	3	3	7	3	8	3	4

vPtr vEnd

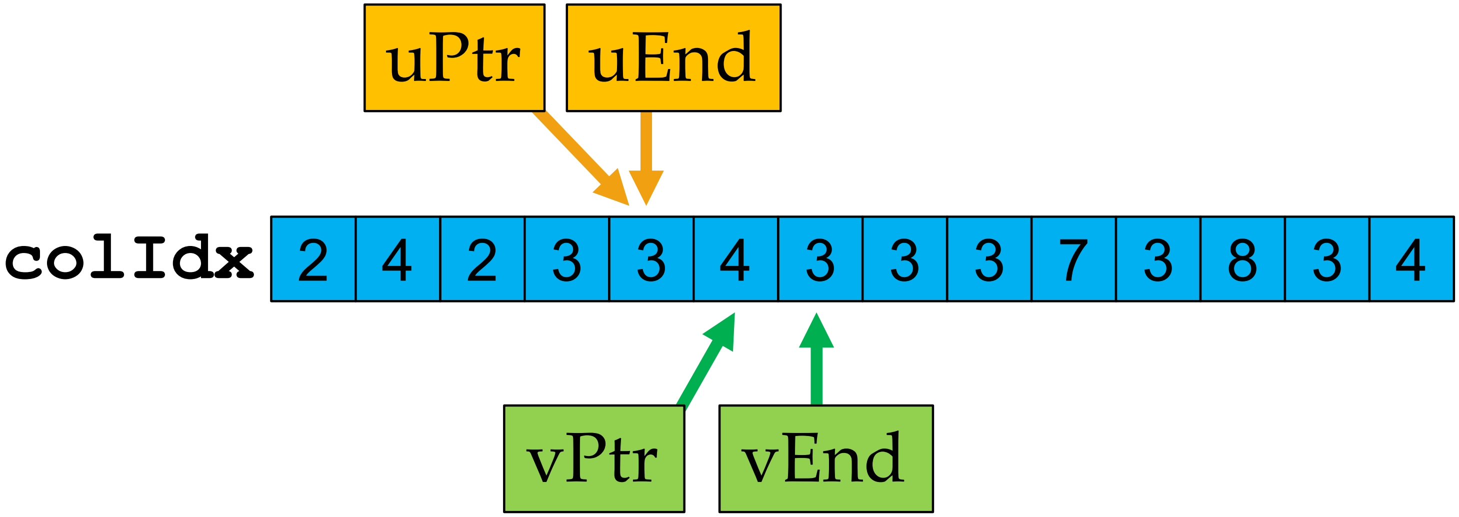
TC = 0, w_1 = 3, w_2 = 3

A match!

Advance uPtr and vPtr, and **increment TC**.

© Wen-mei W. Hwu , David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021

Thread 2 Executes the Search Loop (Ends)



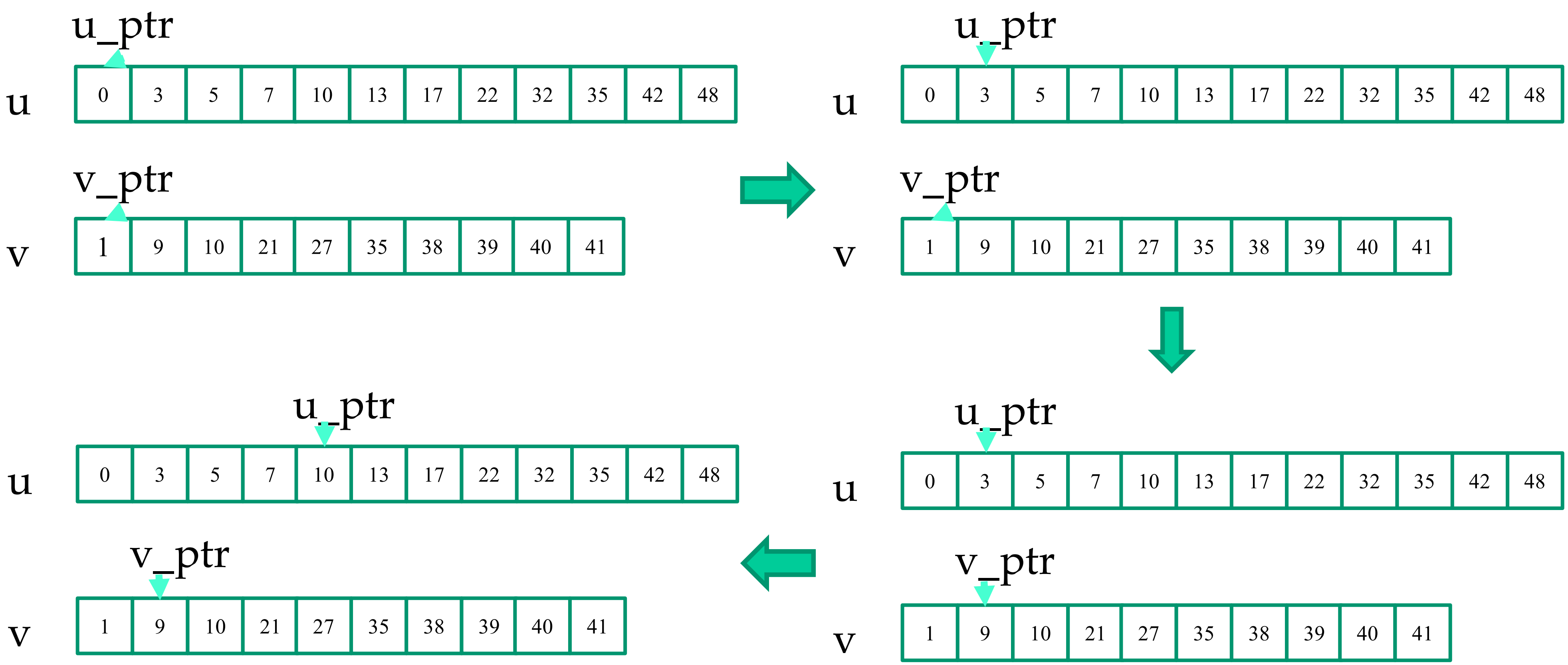
colIdx	2	4	2	3	3	4	3	3	3	7	3	8	3	4

TC = 1, uPtr = uEnd

All done! Found 1 triangle.

© Wen-mei W. Hwu , David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021

Longer Example of Intersection (1 of 4)



u	0	3	5	7	10	13	17	22	32	35	42	48
v	1	9	10	21	27	35	38	39	40	41		

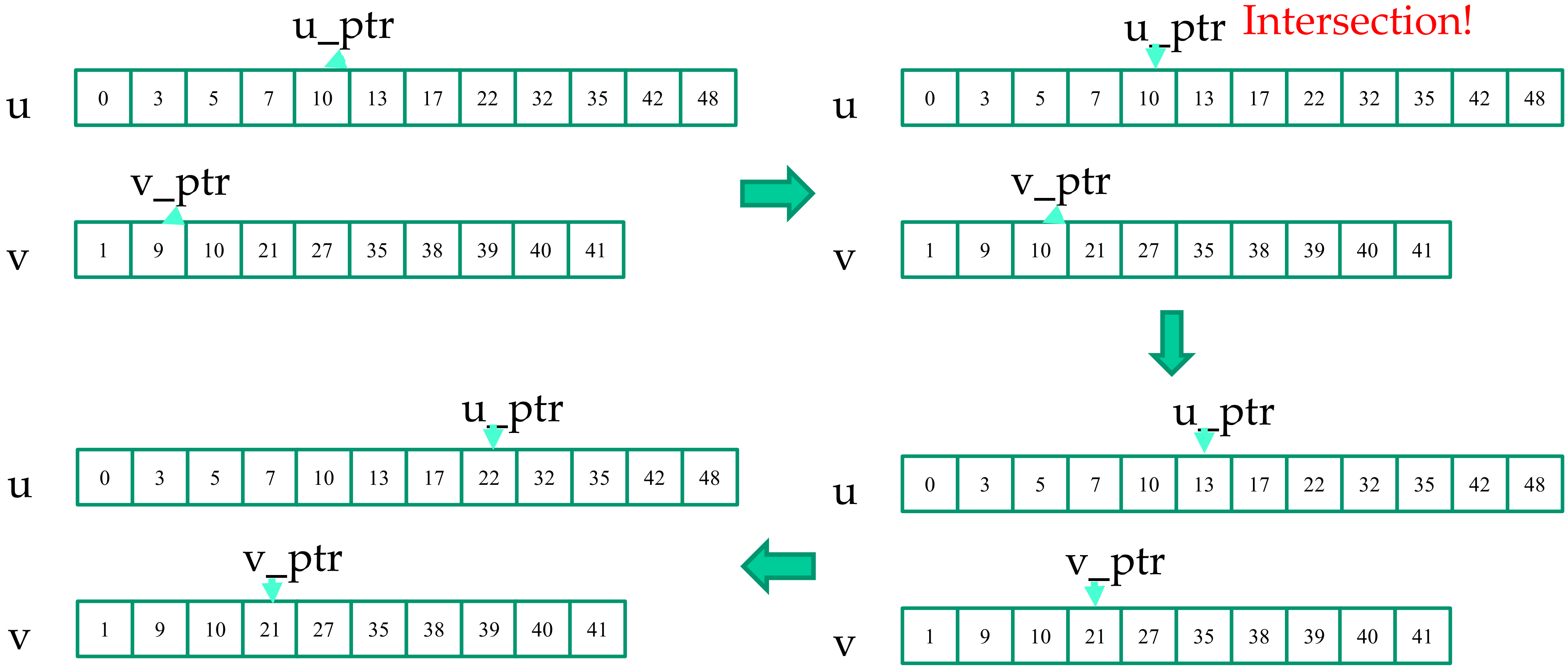
u	0	3	5	7	10	13	17	22	32	35	42	48
v	1	9	10	21	27	35	38	39	40	41		

u	0	3	5	7	10	13	17	22	32	35	42	48
v	1	9	10	21	27	35	38	39	40	41		

u	0	3	5	7	10	13	17	22	32	35	42	48
v	1	9	10	21	27	35	38	39	40	41		

© Wen-mei W. Hwu , David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021

Longer Example of Intersection (2 of 4)



u_ptr

u	0	3	5	7	10	13	17	22	32	35	42	48

v_ptr

v	1	9	10	21	27	35	38	39	40	41

u_ptr Intersection!

u	0	3	5	7	10	13	17	22	32	35	42	48

v_ptr

v	1	9	10	21	27	35	38	39	40	41

u_ptr

u	0	3	5	7	10	13	17	22	32	35	42	48

v_ptr

v	1	9	10	21	27	35	38	39	40	41

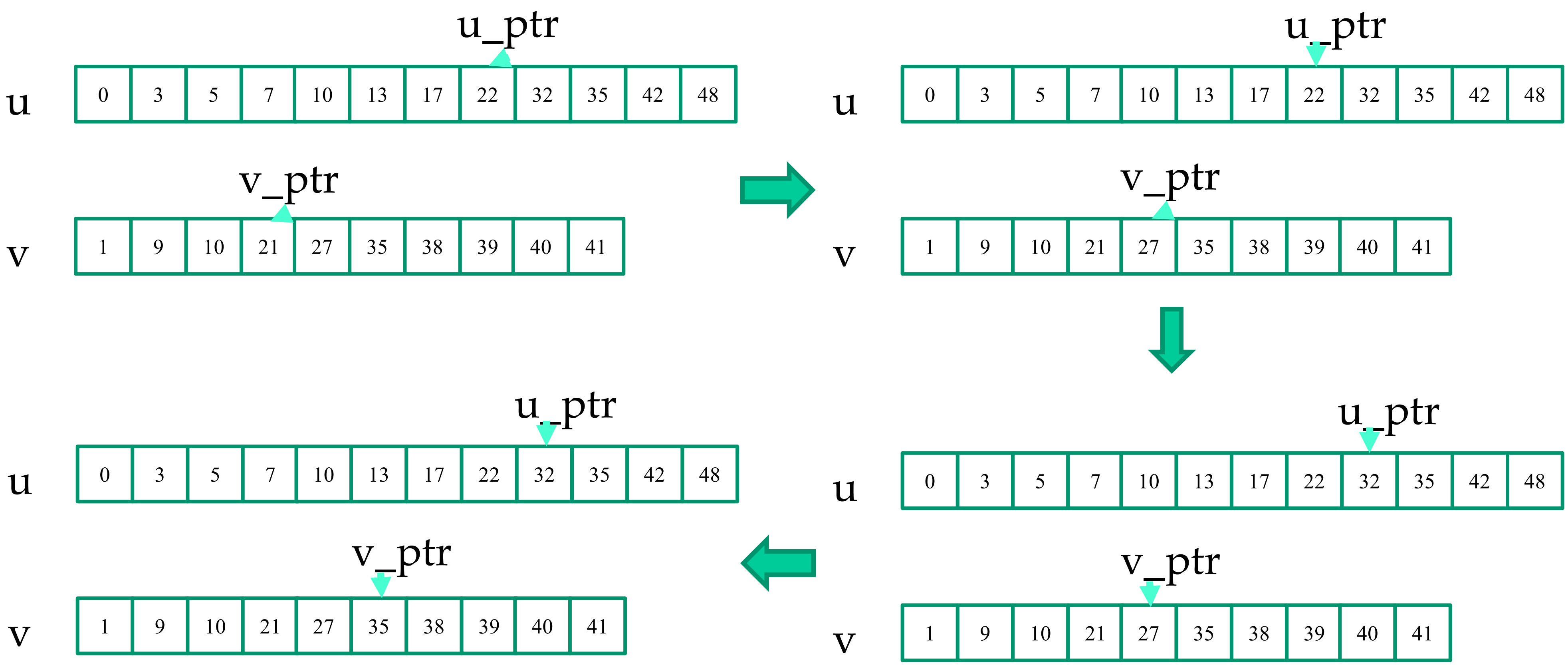
u_ptr

u	0	3	5	7	10	13	17	22	32	35	42	48

v_ptr

v	1	9	10	21	27	35	38	39	40	41

Longer Example of Intersection (3 of 4)

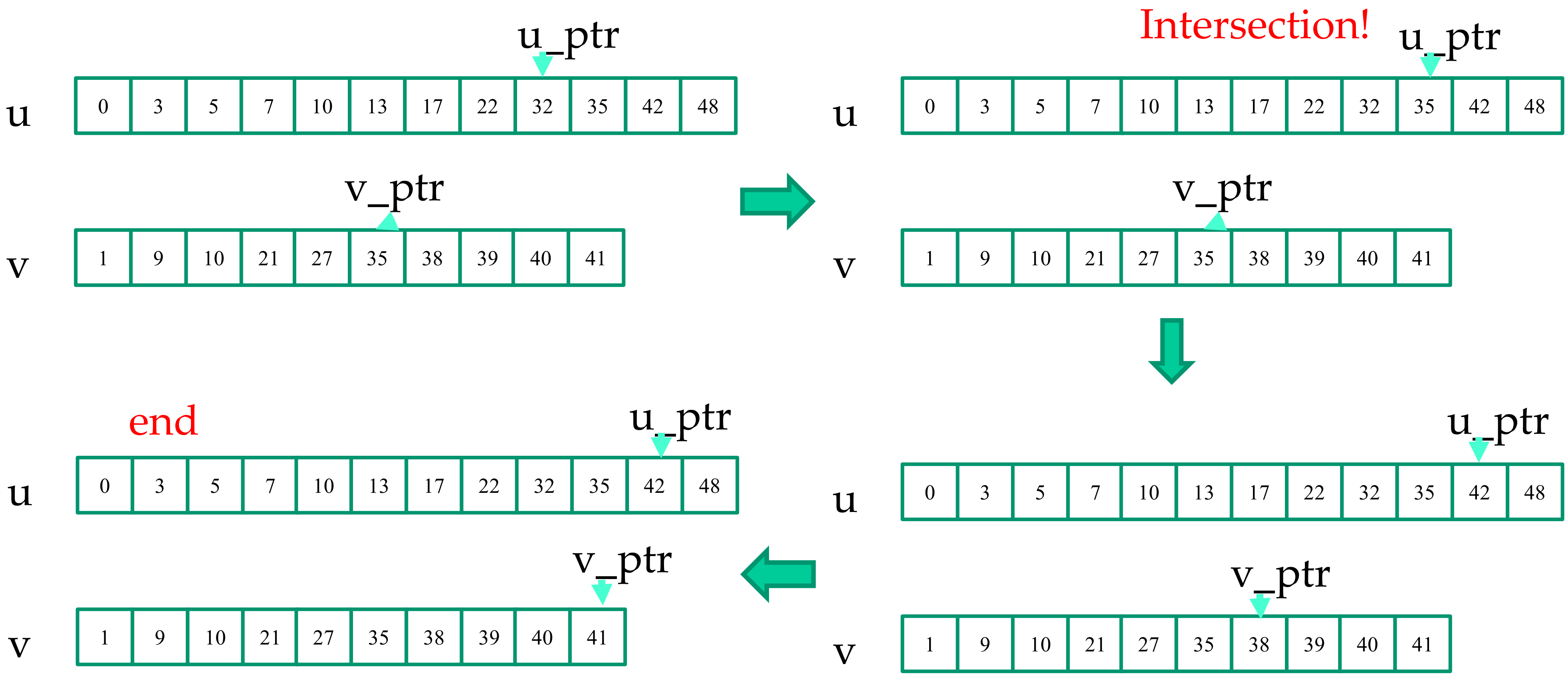


u: 0 3 5 7 10 13 17 22 32 35 42 48

v: 1 9 10 21 27 35 38 39 40 41

© Wen-mei W. Hwu , David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021

Longer Example of Intersection (4 of 4)



u_ptr

u 0 3 5 7 10 13 17 22 32 35 42 48

v_ptr

v 1 9 10 21 27 35 38 39 40 41

Intersection! u_ptr

u 0 3 5 7 10 13 17 22 32 35 42 48

v_ptr

v 1 9 10 21 27 35 38 39 40 41

u_ptr

u 0 3 5 7 10 13 17 22 32 35 42 48

v_ptr

v 1 9 10 21 27 35 38 39 40 41

end

u_ptr

u 0 3 5 7 10 13 17 22 32 35 42 48

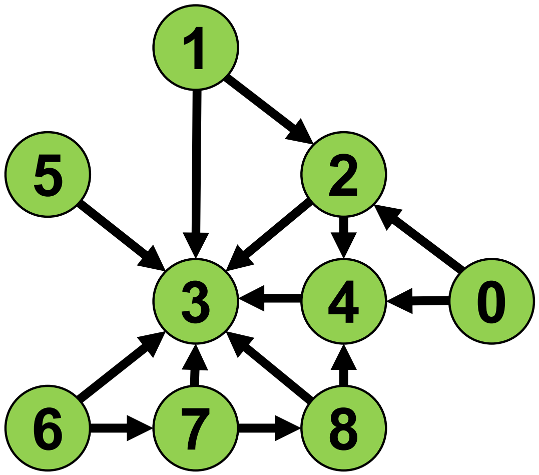
v_ptr

v 1 9 10 21 27 35 38 39 40 41

Need to Reduce Per-Thread TC to Graph TC

- When done,
  - **reduce TC over threads and blocks**
  - to find total TC for graph.

- **Works reasonably well** for small, balanced neighbor lists.
- That's the basic **approach for Lab 6**.



Control Divergence May be a Problem

Each thread

- ping-pongs between
- advancing uPtr and
- advancing vPtr.

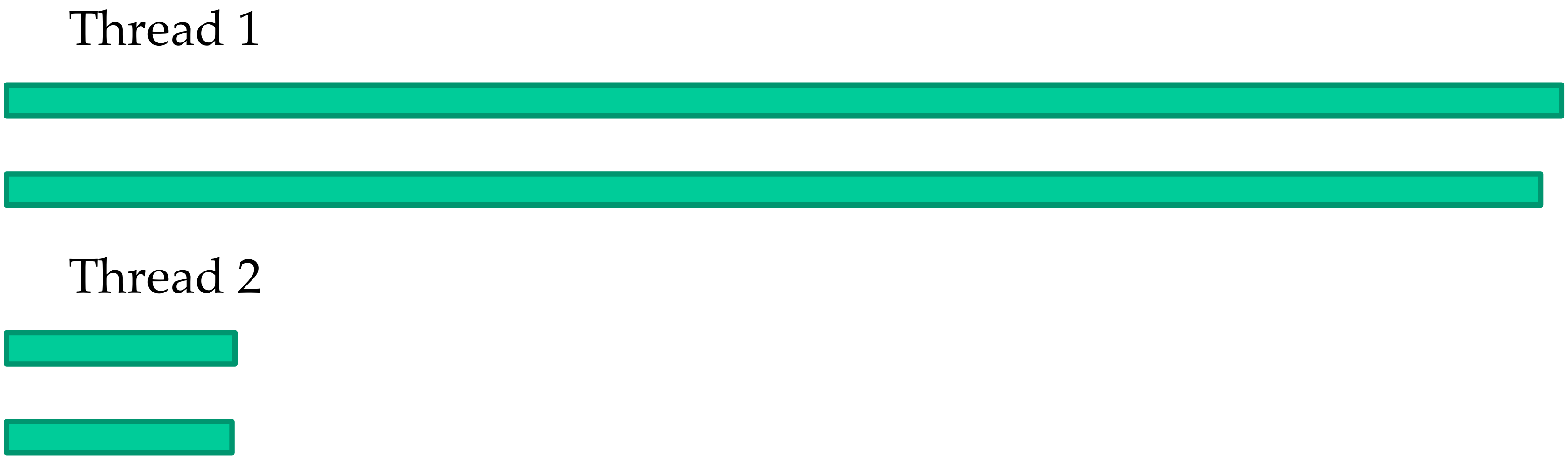
May lead to **significant branch divergence!***

*Take a look at the **`__syncwarp`** function,
which may be useful on Titan V (Volta) GPUs.

Load Imbalance Also a Problem

Neighbor list **intersection parallelized** across edges.

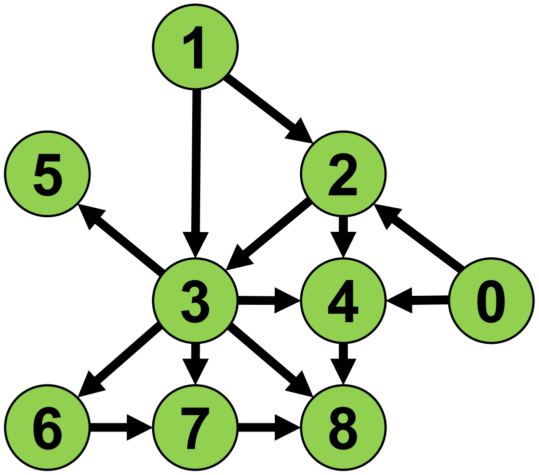
- **Variable** neighbor **list length creates load imbalance**.
- Some threads take much more time to finish.



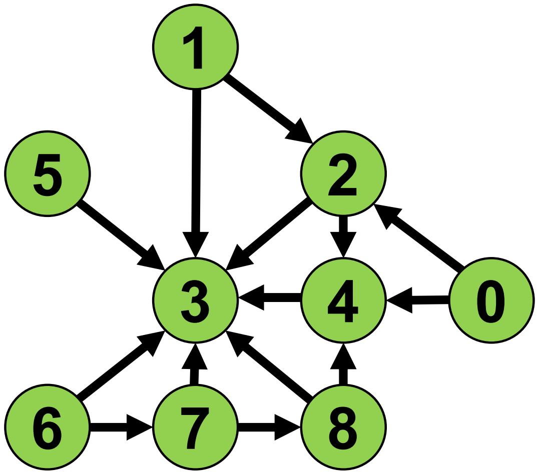
Total Order Selection Affects Load Balance

Load imbalance is why the total order chosen can matter.

Remember our two versions?



Node degrees are
2,2,2,5,1,0,1,1,0



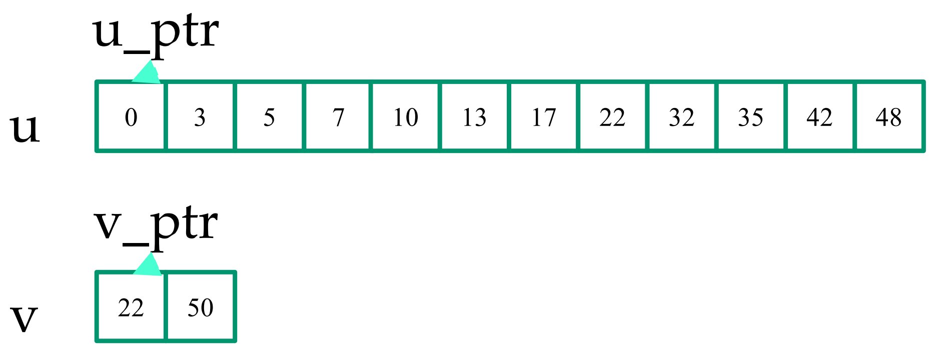
Node degrees are
2,2,2,0,1,1,2,2,2

Complexity O(U+V) Requires TWO Short Lists

One short list

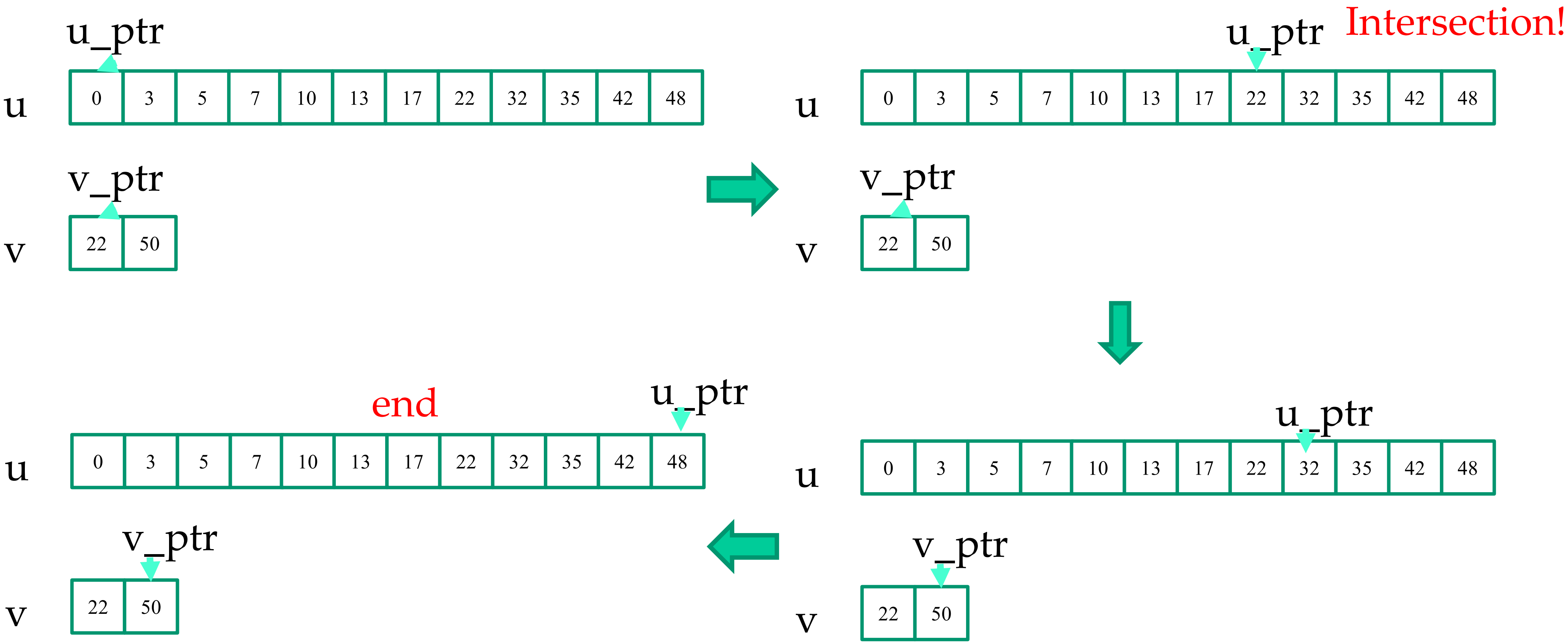
- **does not** imply that
- a thread **finish**es **quickly**.

Consider the lists below.



u	0	3	5	7	10	13	17	22	32	35	42	48
v	22	50										

Long vs. Short List Can Run Long



Overheads May Outweigh Gains in Balanacing Load

What can we do about long-running threads?

- Analogous to exam grading:
  - 5 staff, 5 problems
  - One problem per staff
  - But one problem is hard to grade…
- Can other staff help? Fairly and consistently?
- Or does the time to "train" them (coordinate, launch kernels, change algorithm, and so forth) cost more than just waiting for the thread to finish?

First Option: Switch to Binary Search

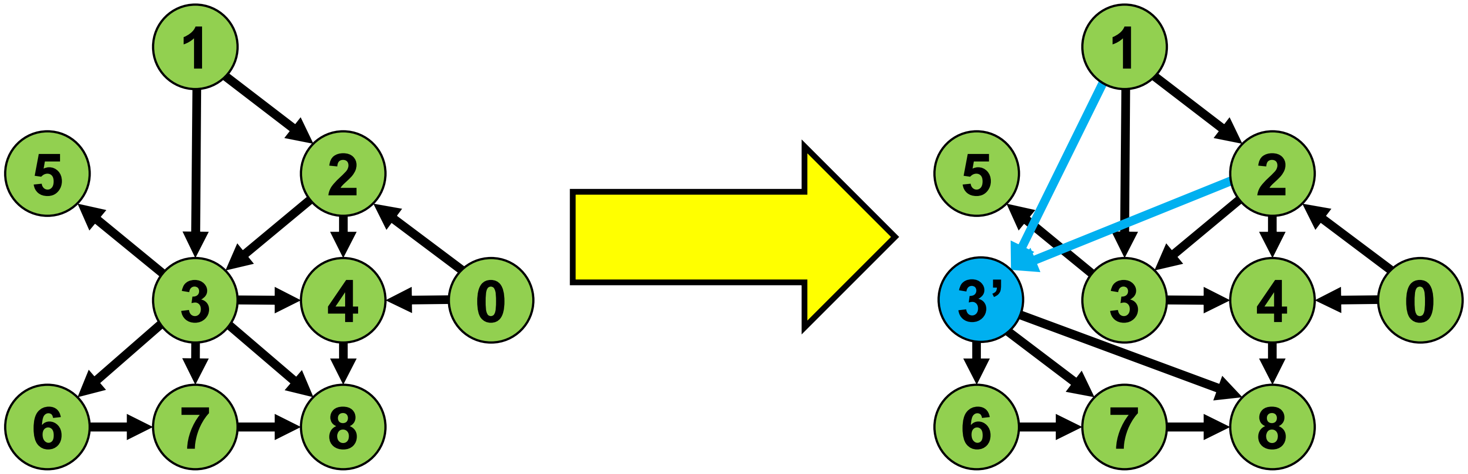
One option: switch to binary search.

- **Find** elements of **short list** (**U**) **in long** list (**V**)
- Be careful:
  - complexity **O(U log V)** instead of **O(V)**, so
  - **need U < V / log V** to be competitive.
  - But remember that one thread running by itself means 31 other resources unused.
- **Can parallelize N(u) across threads**; doing so does not reduce complexity.

Second Option: Break into Pieces

Alternatively, break long list into pieces.

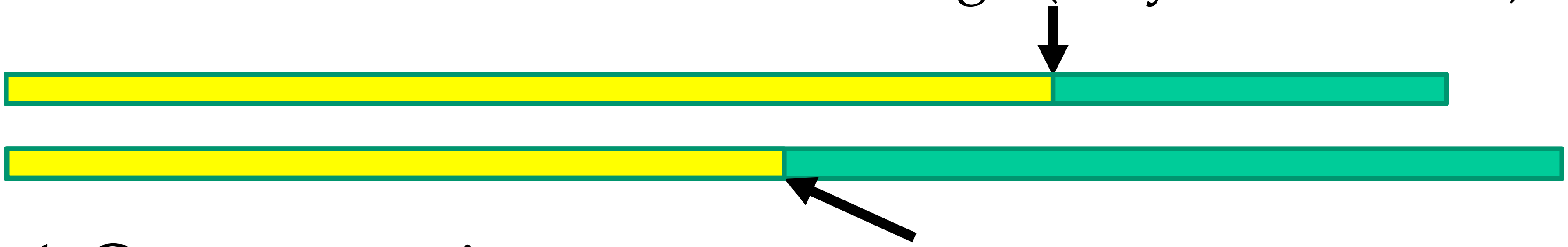
- Can **parallelize** search **over pieces**.
- Be careful: **complexity is O(UV)!**
- Equivalently, can split nodes statically.



Use Binary Search to Partition

What if both lists are long?

- **Splitting both** lists can be **expensive**!
- But maybe **not so bad if done thoughtfully**…



3. Repeat as necessary.

2. Find where splitter should go (may not be even).

4. Compare sections.

1. Pick midpoint as splitter.

Lots of Room for Tuning and Techniques

Lots of Ph.Ds written (and more available!) trying to optimize graph representations and algorithms.

The **solutions discussed**

- **require new kernels** (to get more threads).
- **Next week**, we'll talk about **dynamic parallelism** in CUDA, which allows one to launch kernels from kernels.

Performance Depends on Input Graph

- Let’s take a look at some examples:
  - SuiteSparse Matrix Collection* (https://sparse.tamu.edu/).
  - Many types, many irregular connections.

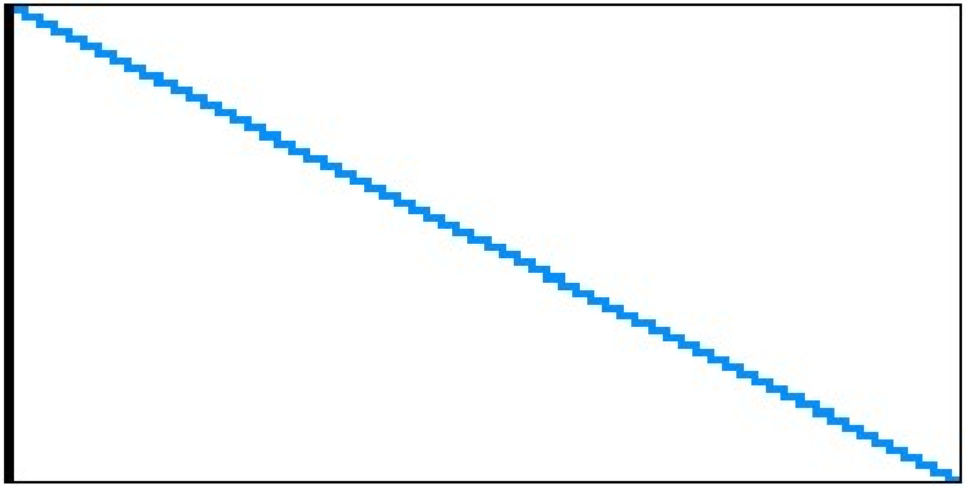
- One architecture does not fit all.

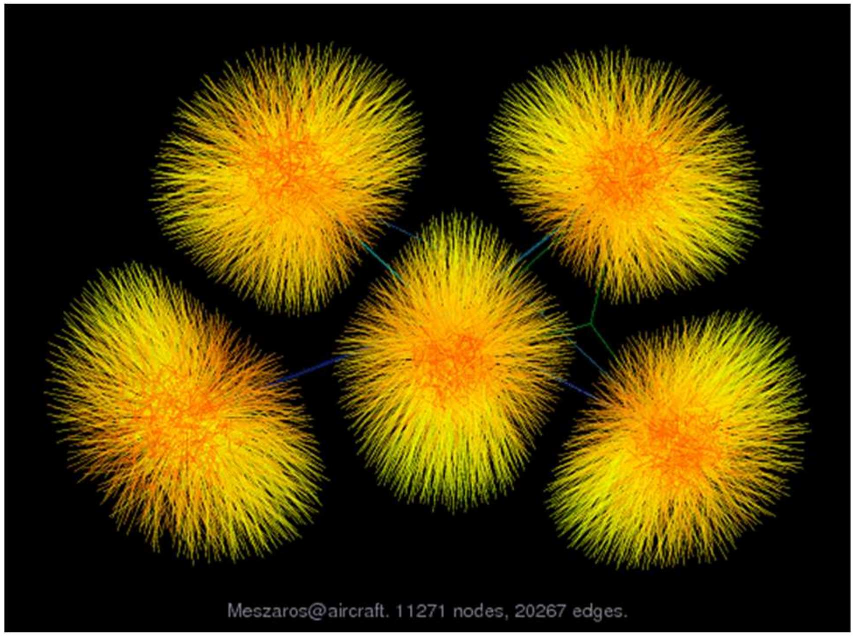
*T.A. Davis, Y. Hu, “The University of Florida Sparse Matrix Collection,” ACM Transactions on Mathematical Software 38(1), Article 1, Dec. 2011.

Example 1: Linear Programming

A linear programming problem:

- C. Meszaros' test set
- ID: Meszaros/aircraft

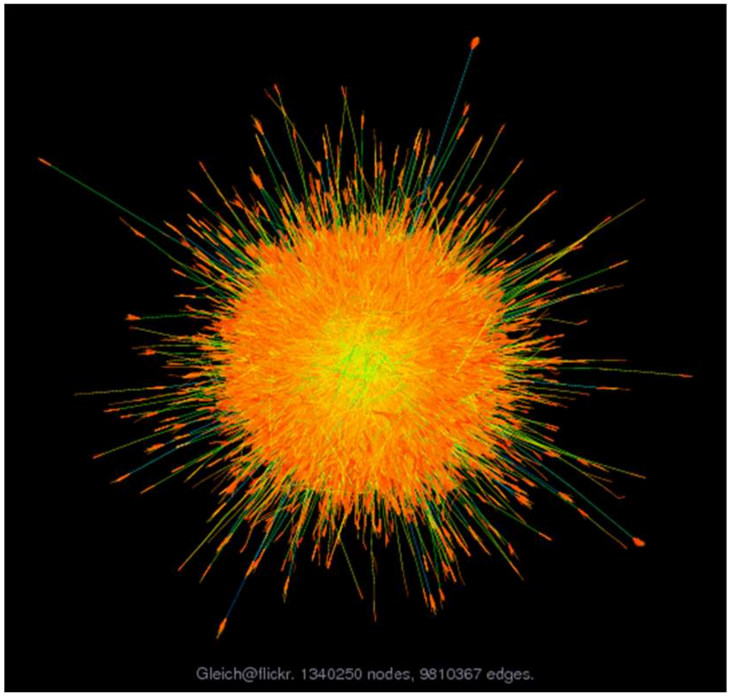




Example 2: Photo Management

Photo management app:

- David Gleich’s 2005 crawl of flickr.com
- ID: Gleich/flickr.html

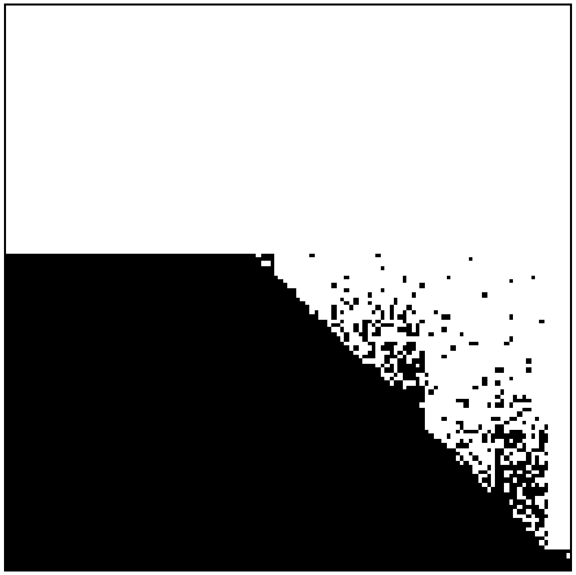


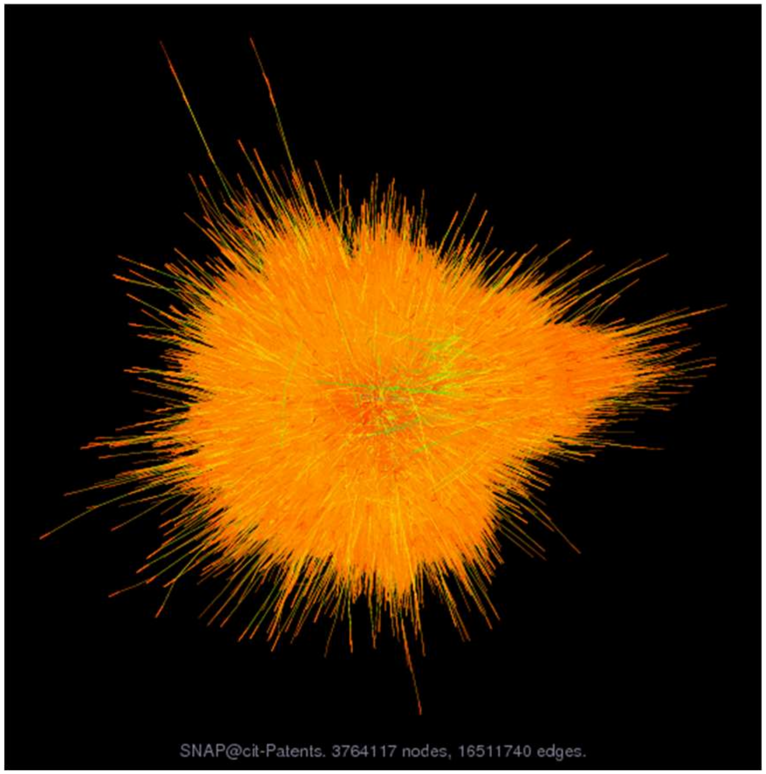
© Wen-mei W. Hwu , David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021

Example 3: Patent Citations

Relationships between patents:

- citations amongst US patents
- ID: SNAP/cit-Patents.html

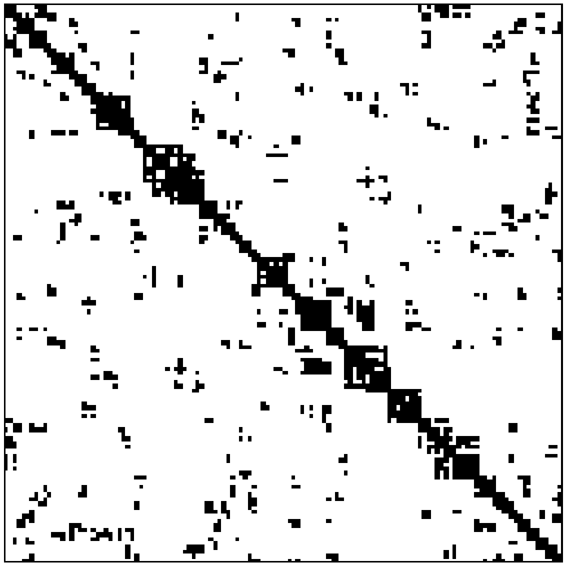


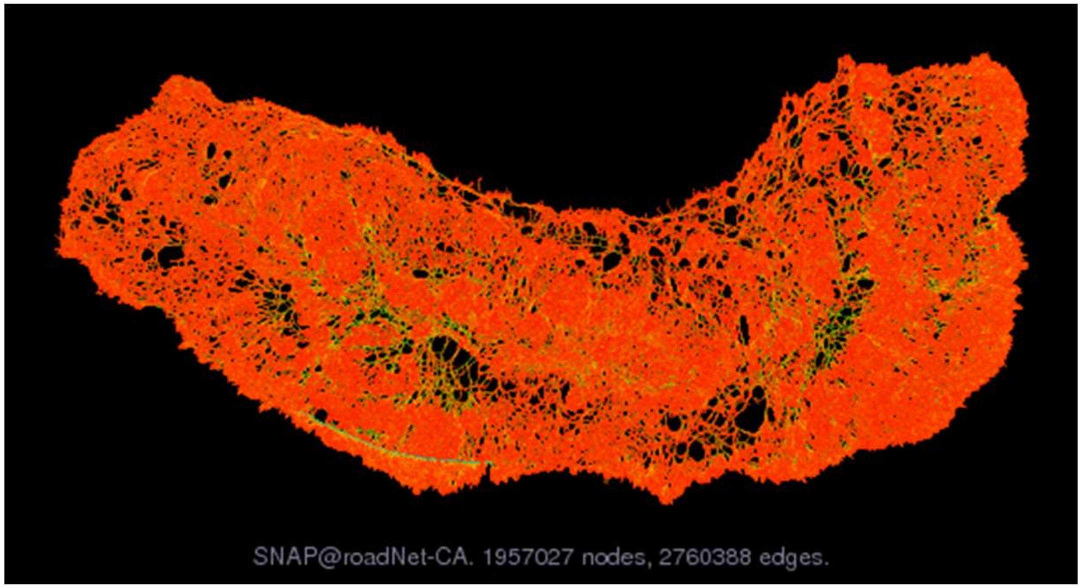


Example 4: California Road Network

Relationships between roads:

- road network in California
- ID: SNAP/roadNet-CA.html

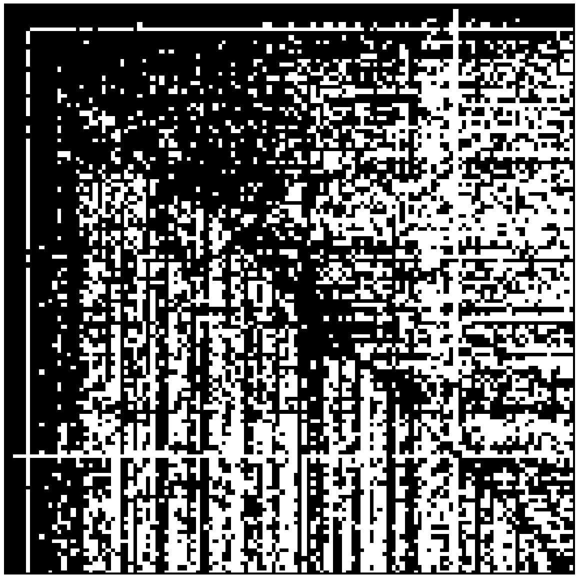


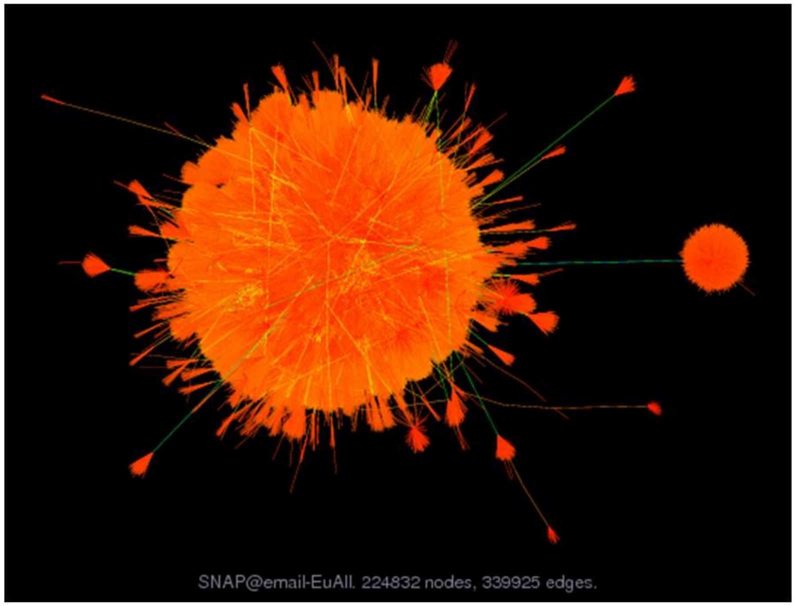


Example 5: E-Mail Social Network

Relationships between e-mail correspondents:

- E-mail network from an EU research institution
- ID: SNAP/email-EuAll.html

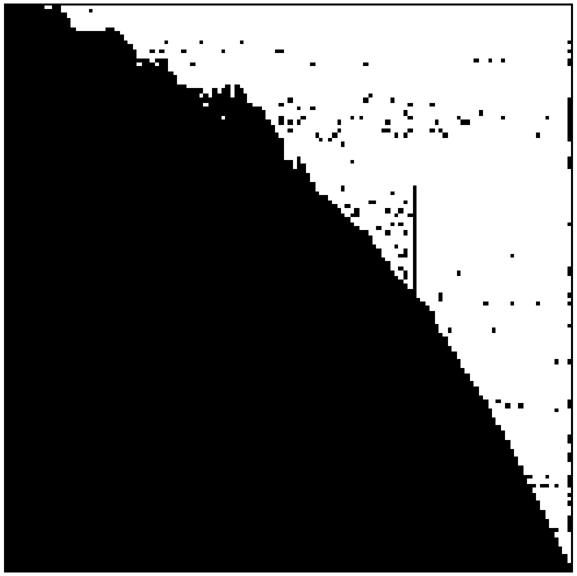


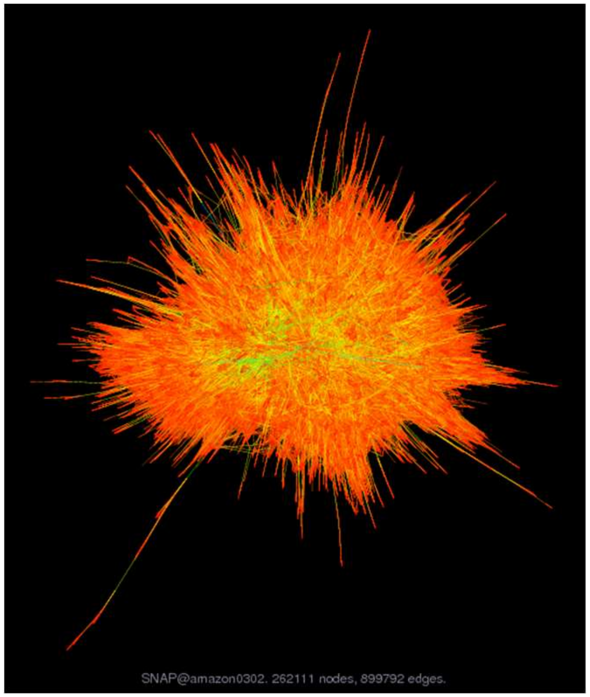


Example 6: Online Shopping!

Relationships between purchases:

- Amazon co-purchasing product network from 2003
- ID: SNAP/amazon0302.html





GPUs Set the Record for Triangle Counting

An example from 2012:*

- Twitter graph: 41M nodes, 1.4B edges, 34.8B triangles
- Hadoop: 1536 machines → 423 minutes (7 hours)
- GraphLab on 64 machines (1024 cores) → 1.5 minutes

Achieving that speed requires multiple GPUs.

How do we do that?

*J.E. Gonzalez, Y. Low, H. Hu, D. Bickson, C. Guestrin, “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs,” OSDI, 2012.

cudaMemcpy Copies Data Across PCIe

Why use multiple GPUs?

In 2012, Twitter graph

- did not fit in a GPU's memory, but
- today it would.
- Bigger graphs exist, but
- didn't need 64 GPUs in 2012.

Sometimes, **need more throughput**.

That's why we want **data scalability**, remember?

cudaMemcpy Copies Data Across PCIe

Let's review CPU-GPU system architecture.

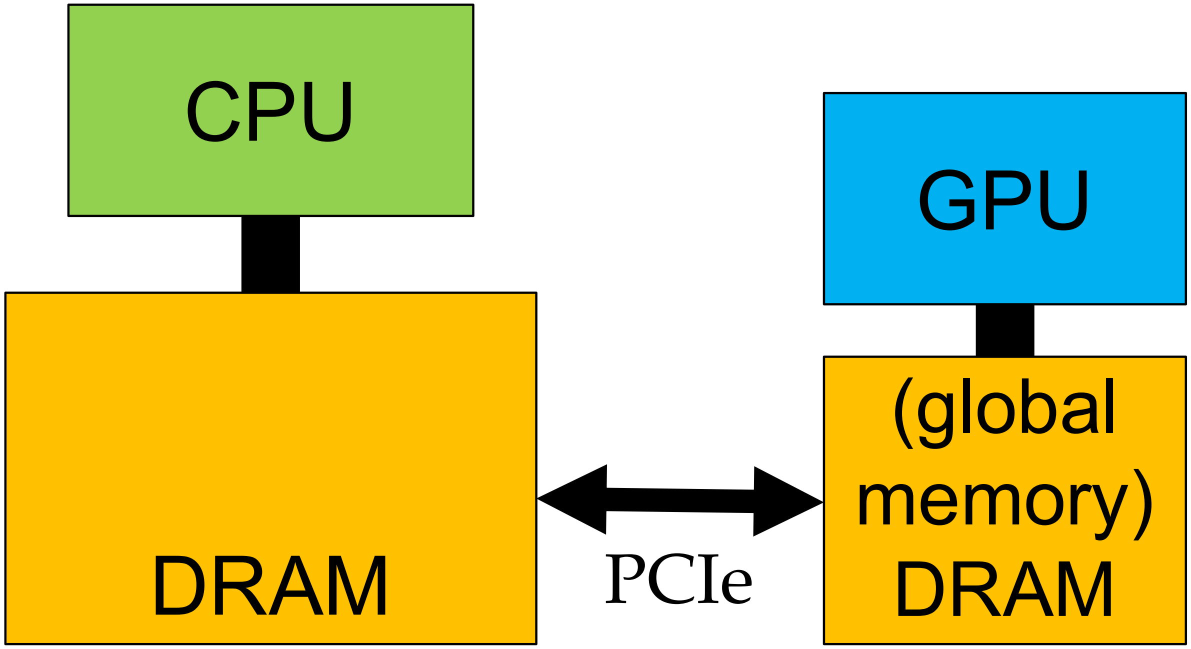
Host (CPU) **and device** (GPU)

- **have** associated **DRAM memories**,
- which are separate.

`cudaMemcpy` uses

- Direct Memory Access (**DMA**) **engines** on GPU to
- **move data between** these **memories** (over PCIe).

Illustration of System Architecture



`cudaMemcpy` used to move data back and forth.

Pinned Host Memory Makes Copying Faster

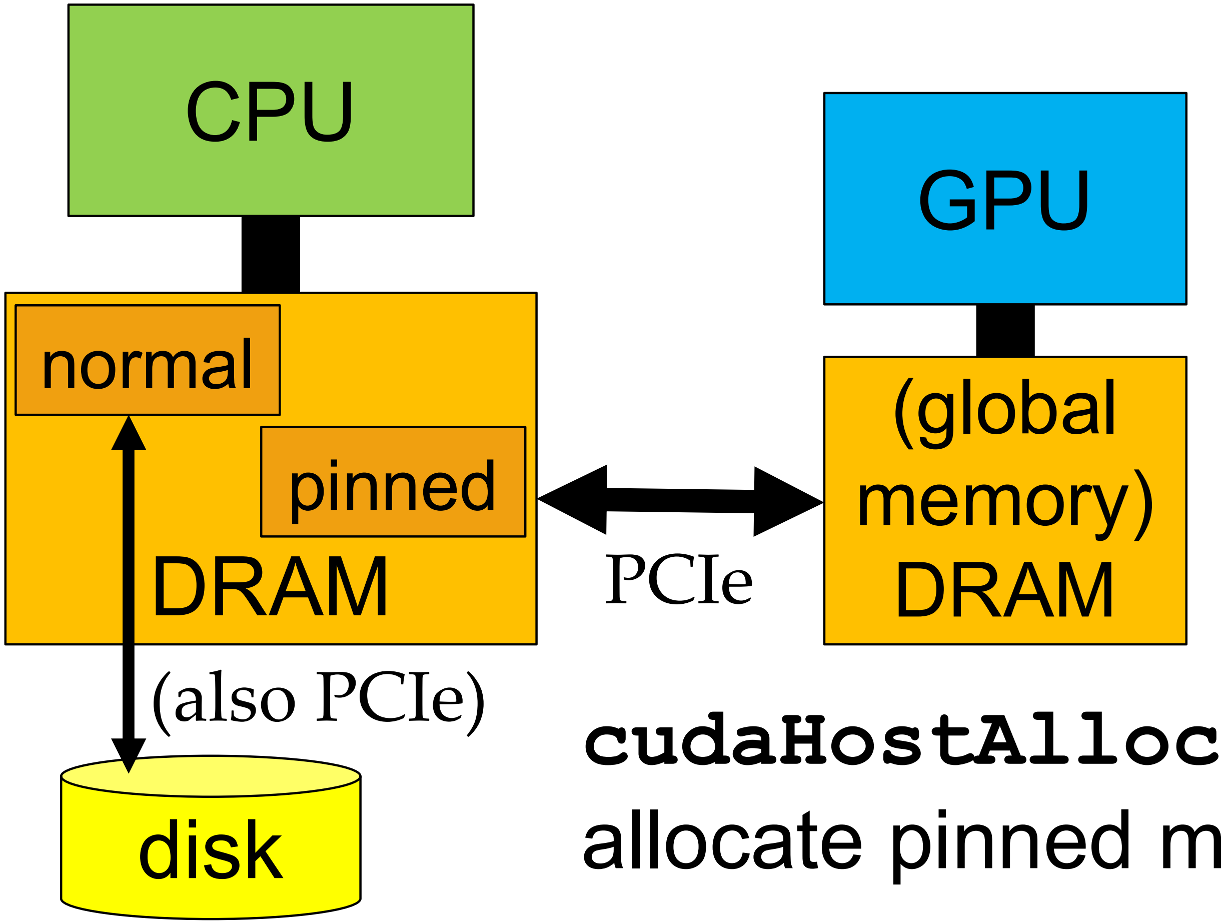
CPU memory managed by OS:

- **virtualized, and** sometimes
- **swapped** transparently **onto disk**.

DMA engines require non-virtualized addresses.

- Can **“pin” memory** in OS to avoid swapping.
- **Use `cudaHostAlloc`** to allocate pinned memory,
- but **be careful**: too much pinned memory makes system extremely slow!
- Pinned memory is **faster to copy** to device memory.

Illustration of Pinning Host Memory



`cudaHostAlloc` used to allocate pinned memory.

Zero-Copy Access from GPU to Host Memory

Since CUDA 2.2 (2009),

- **GPU** has been **able to access pinned host memory**
- **directly** over PCIe!*

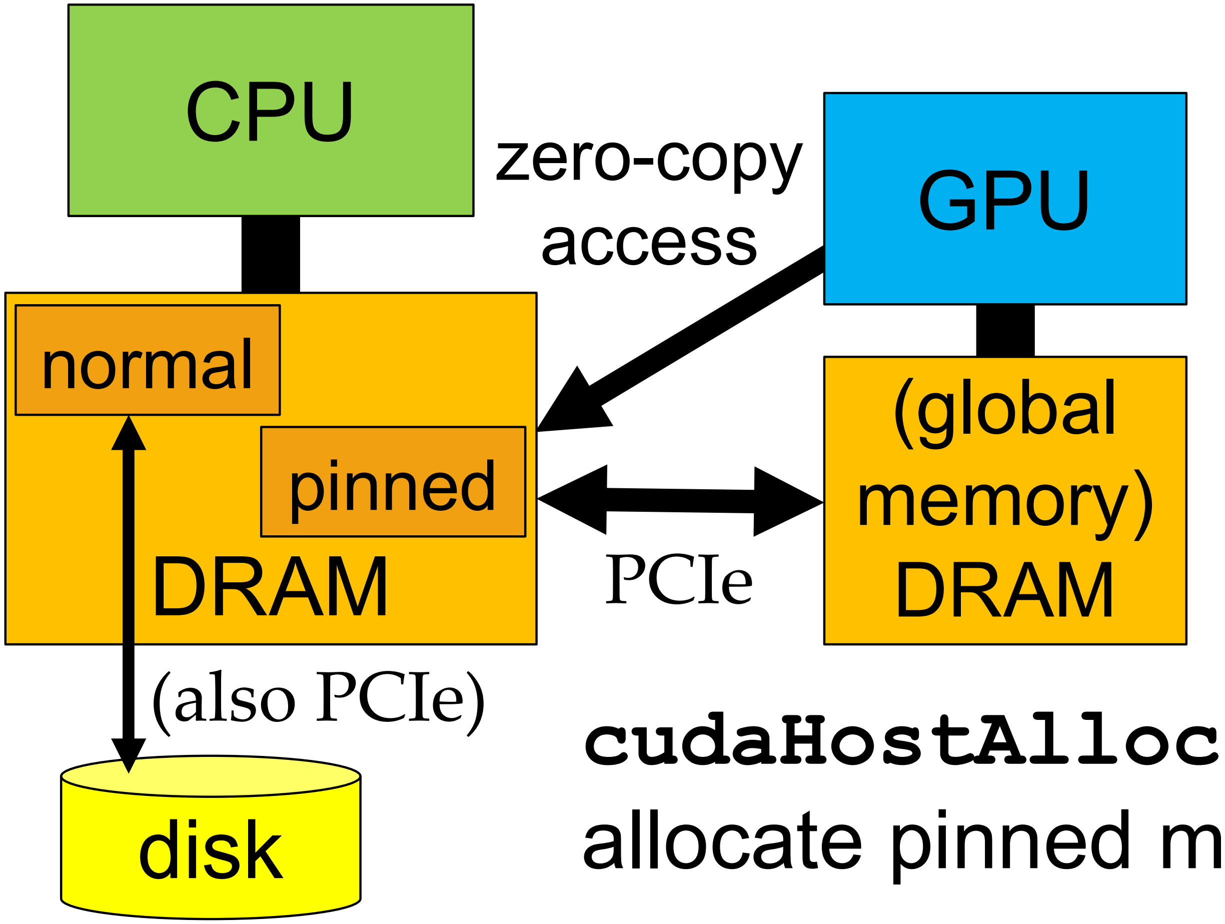
To do so, translate pinned host address to GPU with

```
cudaError_t cudaHostGetDevicePointer
(void** devPtr, void* hostPtr, unsigned int flags);
```

- (flags should be 0).
- GPU can then **use zero-copy access** to *`devPtr`.

*See Section 9.1.3 of https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/ for more detail.

Illustration of Zero-Copy Access



`cudaHostAlloc` used to allocate pinned memory.

Zero-Copy Access Also Works GPU to GPU

In early 2014, NVIDIA announced **NVLink**,

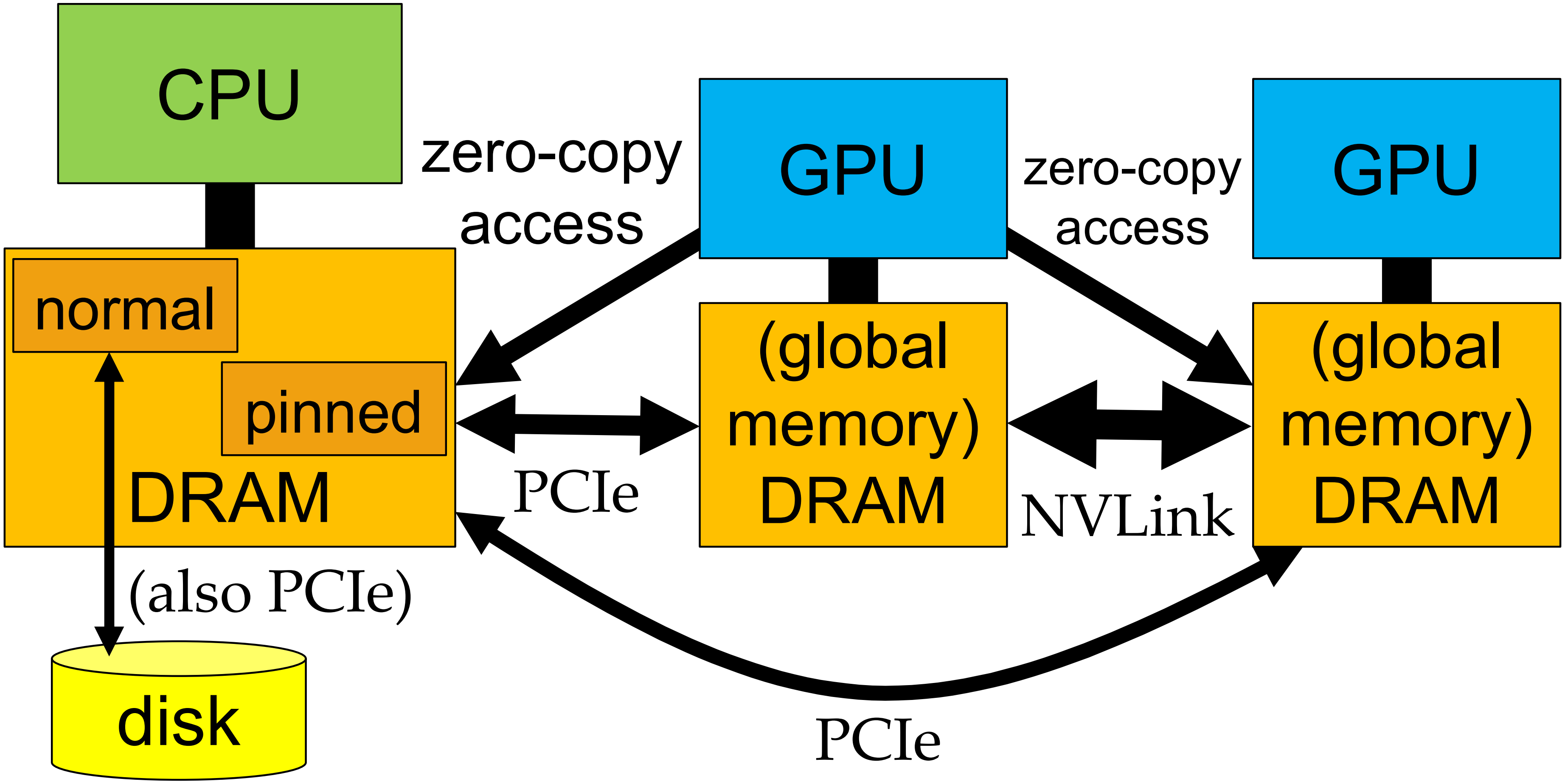
- a **proprietary GPU-GPU interconnect**
- for multi-GPU systems.

NVLink provides

- **lower latency and higher throughput than PCIe**
- for GPUs in the same system to move data between device memories.

In GPU kernel, **use zero-copy access**!

Illustration of Multi-GPU Connectivity



Set Device Context with **`cudaSetDevice`**

How? Takes some setting up…*

Remember this function?

```
cudaError_t cudaGetDeviceCount (int* count);
```

- Turns out it might not always return 1!
- **Device IDs are 0 to N-1** where **N** is **`*count`**.

To switch contexts

- (per "host thread"—consider using Posix),
- **call `cudaSetDevice`** with the desired ID.

*Managing multiple devices is covered in Section 6.1 of https://docs.nvidia.com/cuda/cuda-runtime-api/.

CUDA Calls Use Selected Device

CUDA calls occur in a context.

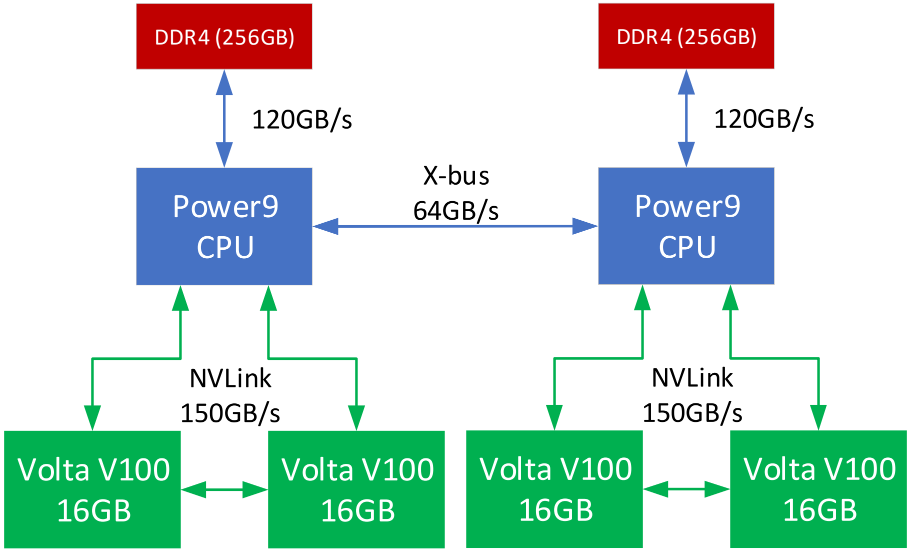
- Allocate memory … on the selected device.

- Copy memory … to/from the selected device.

- Launch a kernel … on the selected device.

Bigger Systems May Have >1 CPU and >1 GPU

You may have access to more powerful machines!



- For example, in Spring 2019, IBM provided the Newell machine:
- 2 IBM Power9 CPUs, each 10 cores/80 threads@4.02GHz, 256GB RAM.
- 4 NVIDIA Volta V100 GPUs (16GB)

NUMA Machines May Not Give Full Accessibility

Be warned: typically, **GPUs cannot use**

- **zero-copy access** to another GPU's memory
- **if** the **other GPU** is "**connected**"
- **to a different CPU**, even in a NUMA system like the Newell machine.

To check accessibility, use (on a CPU)

```
cudaError_t cudaDeviceCanAccessPeer
 (int* canAccessPeer, int device, int peerDevice);
```

If the CPU can't see the target GPU to make this call, the answer is "no."

Zero-Copy Access Still a Distributed Memory Model

Zero-copy access does require programmer effort.

Allocations

- separate for each GPU,
- so **pointers are separate**.

Can we **split** an "**array**" **across** GPU **memories**? Yes …

1. Use indirection: `float* arr[numGPU];`

   Extra instructions and a **little complexity; or**

2. use different names: `float* arr1, * arr2;`

   Much more complicated.

Unified Memory Simplifies Data Structures

In **CUDA 6.0** (also 2014), NVIDIA released

- a primitive version of Distributed Shared Memory*
- called “**unified memory**.”

Allocate with **`cudaMallocManaged`**.

- CPU and GPUs can access.
- GPUs leverage zero-copy access if available.

*K. Li, P. Hudak, “Memory Coherence in Shared Virtual Memory Systems,” 5th Annual Symposium on Principles of Distributed Computing, pp. 229-239, 1986.

Still Need to Think About Access Patterns

Unified memory keeps **one copy of data** (only)

- **Coherence** is **not supported**,
- so be careful with write accesses.

Kernel-level synchronization does work,

- so CPU can read GPU results safely
- after a kernel completes.

Unified Memory Moves Pages Automatically

Unified memory

- **uses page migration** from memory to memory.
- similar to that used in Sun Enterprise servers in early 2000s.

Pages

- matching OS page size,
- typically **4 kB** to **64 kB**.
- **migrate on-demand**.

Unified Memory Tricky to Use Well

Latency of migration has a huge performance impact.*

- **Prefetching is supported**, but
- be sure that kernels have high data locality and reuse.

Poorly-designed access patterns

- **lead to pages thrashing** between GPUs.
- Unified memory also **suffers from false sharing**:
  - threads making **independent use** of data
  - that are mapped to the **same page**.

*N. Sakharnykh, "Maximizing Unified Memory Performance in CUDA, “https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/”

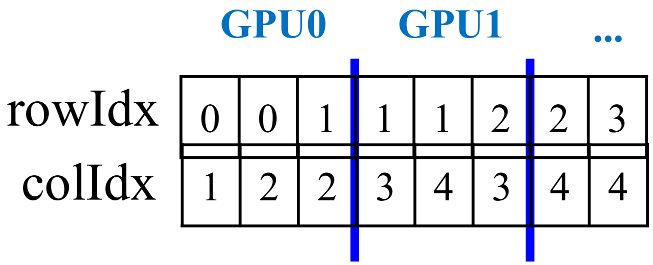
My Take: Same Old DSM

Uniform memory

- is a lot like the 28 previous years
- of SVM/DSM systems:
- looks great at first, but
- can be **really hard to use** in practice
- **if you care about performance**.

Partitioning Strategies: Edge List Partition

What about our approach?



	GPU0			GPU1			...	
rowIdx	0	0	1	1	1	2	2	3
colIdx	1	2	2	3	4	3	4	4

Each thread:

1. Looks up **u** and **v**.

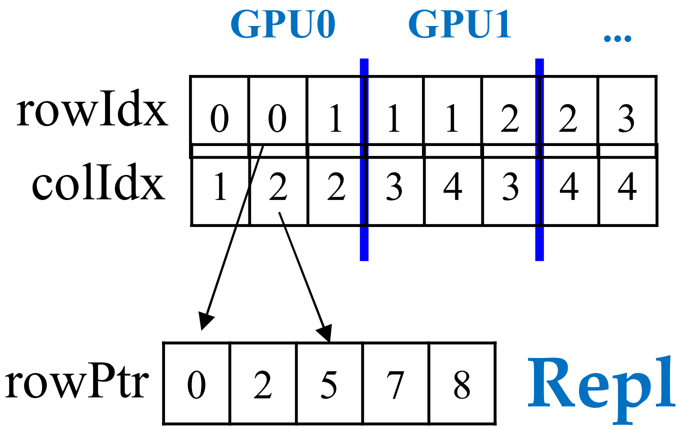
Easy to partition across GPUs!

Partitioning Strategies: Edge List Partition

What about our approach?

Each thread:

1. Looks up **u** and **v**.
2. Retrieves neighbor indices.



rowIdx	0	0	1	1	1	2	2	3
colIdx	1	2	2	3	4	3	4	4

rowPtr	0	2	5	7	8

Replicate?

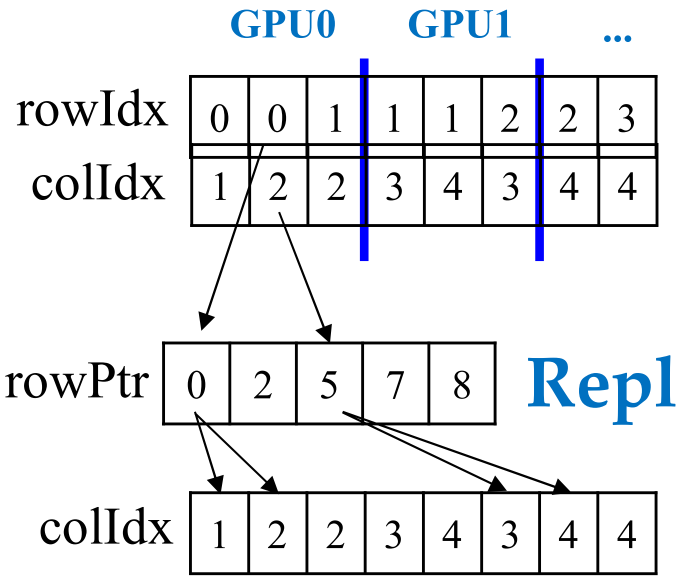
Partitioning Strategies: Edge List Partition

What about our approach?

Each thread:

1. Looks up **u** and **v**.
2. Retrieves neighbor indices.
3. Accesses neighbor lists.

Also **replicate colIdx?**



rowIdx	0	0	1	1	1	2	2	3
colIdx	1	2	2	3	4	3	4	4

rowPtr	0	2	5	7	8

Replicate?

colIdx	1	2	2	3	4	3	4	4

Big Data Require Distribution

That strategy replicates more than half of the data.

Not much room for problem growth.

For throughput, that's fine.

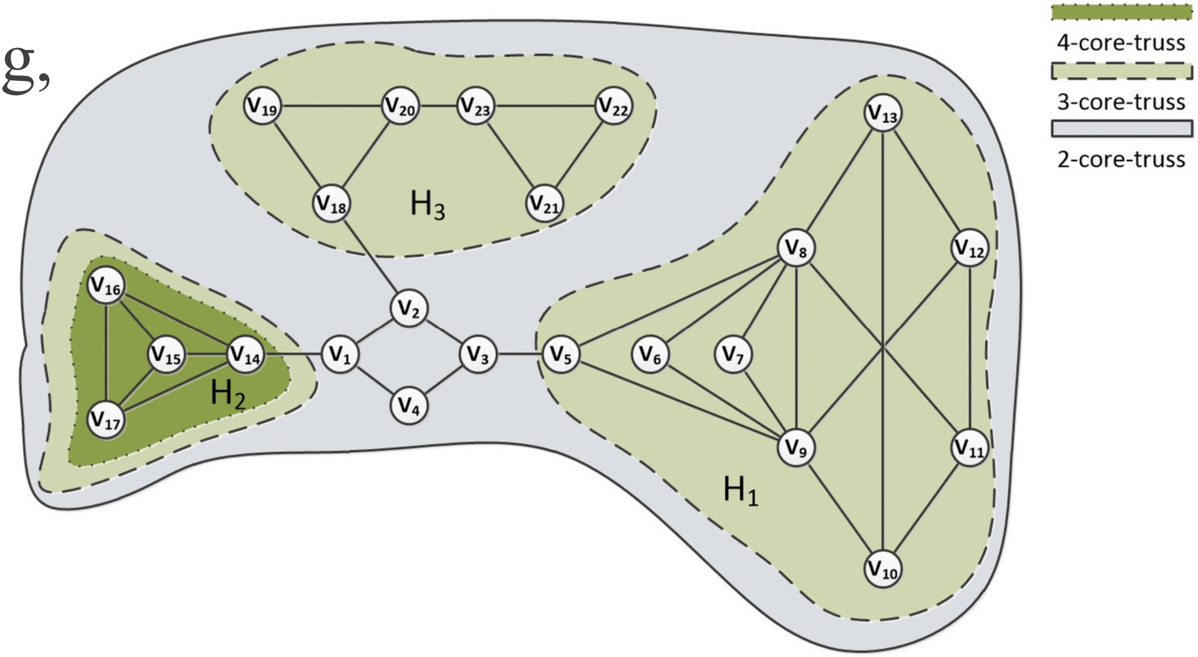
For bigger graphs,

- need to use zero-copy access
- or unified memory.

Eventually need to **write as distributed memory** code (using MPI, for example).

Use Case: Truss Decomposition

From triangle counting, we can keep building up to **find trusses in graphs**.



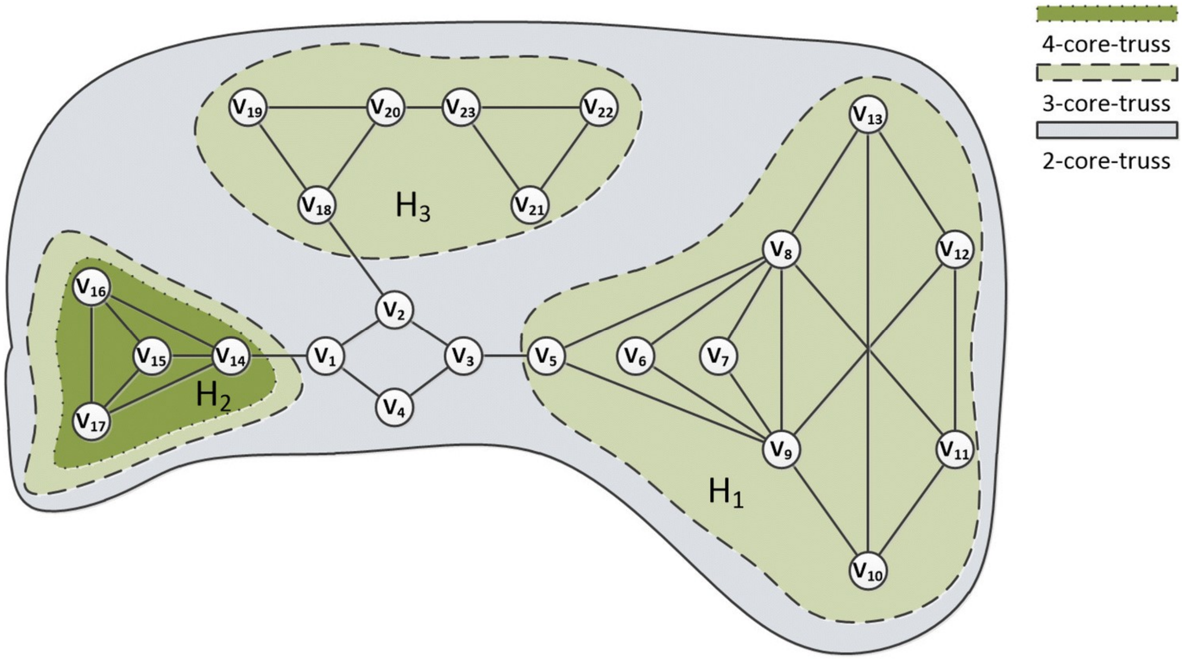
[Image taken from Z. Li, Y. Lu, W.-P. Zhang, R.-H. Li, J. Guo, X. Huang, R. Mao, "Discovering Hierarchical Subgraphs of K-Core-Truss," Data Science and Engineering, 3, pp. 136–149, 2018.]

© Wen-mei W. Hwu , David Kirk/NVIDIA, John Stratton, Izzat El Hajj, Carl Pearson, ECE508/CS508/CSE508/ECE598HK, 2010-2021

Trusses Measure Community Strength

What's a truss?

An **N**-truss is a subgraph in which **each edge** is **part of at least (N – 2) triangles**.



[Image taken from Z. Li, Y. Lu, W.-P. Zhang, R.-H. Li, J. Guo, X. Huang, R. Mao, "Discovering Hierarchical Subgraphs of K-Core-Truss," Data Science and Engineering, 3, pp. 136–149, 2018.]

Trusses are Not Cliques

Is a truss a clique?

Not quite.

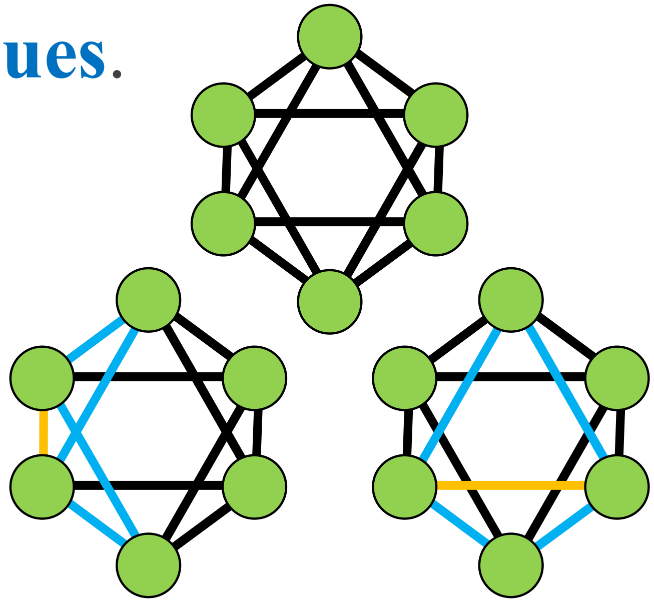
The smallest **N**-truss is the clique on **N** elements.

But **larger N-trusses need not be cliques**.

For example, this graph is a **4**-truss…

It's fully symmetric, but easier to see as two cases:

- side edges (two triangles), and
- middle edges (two triangles).



Finding 3-Trusses is Just a Triangle Count

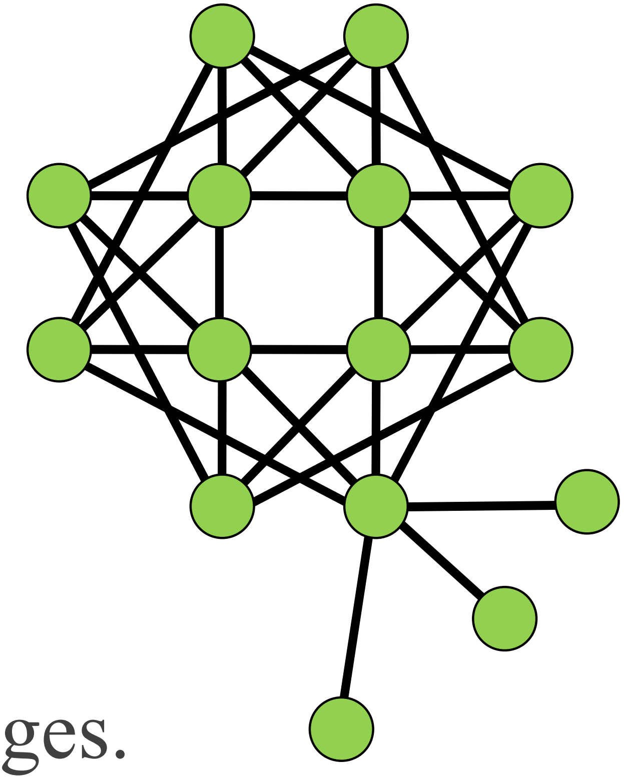
How can we find N-trusses?

For N=3, it's easy:

- **find the triangles**, then
- **get rid of non-triangle edges**.

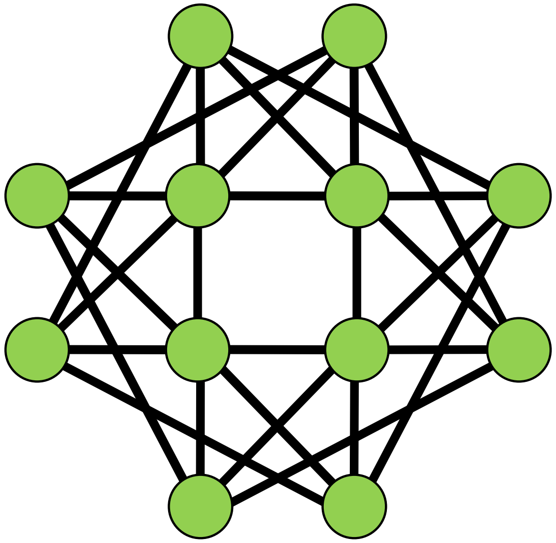
Since edges not in triangles …

- aren't in triangles …
- the triangle count per edge is unaffected by removing those edges.



Stronger Trusses Require Iteration

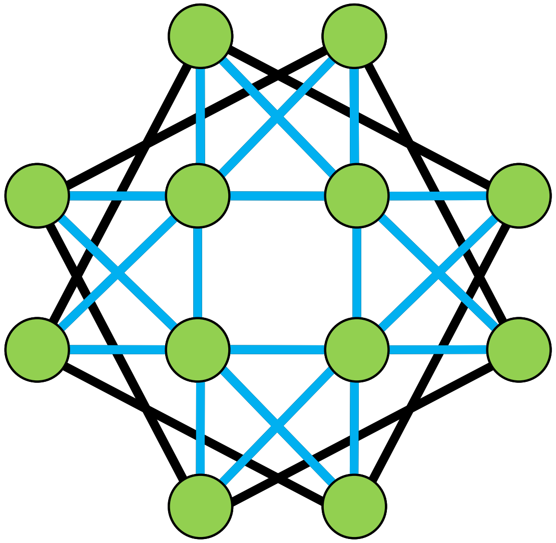
- **For** N-truss with **N > 3**, must **iteratively remove** until all remaining edges are part of at least **N – 2** triangles.
- Consider the graph to the right, for example…
  any 4-truss parts?



© Steven S. Lumetta, ECE508/CS508/CSE508, 2021

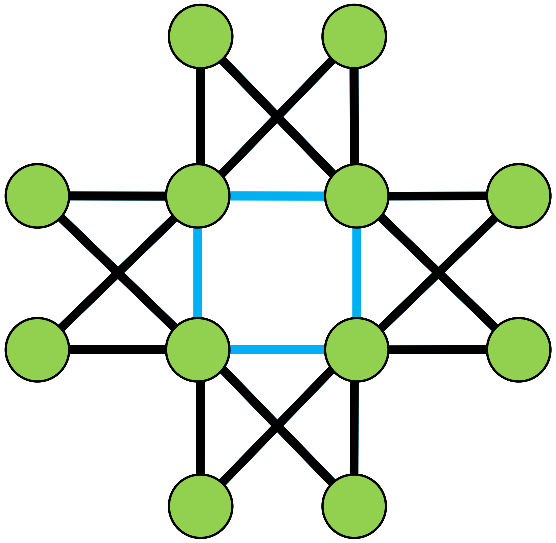
Start with a Triangle Count

- All edges in this graph contribute to triangles, but no edge contributes to more than 2 triangles.
- Let's mark the edges in 2 triangles in blue.
- Black edges contribute to only one triangle.
- Next, let's remove all edges that are not in 2 triangles.



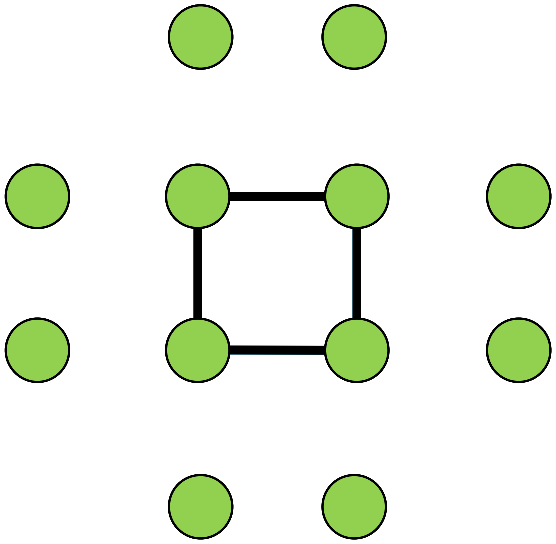
Removing Edges Removes Triangles

- But now some of the edges are no longer in 2 triangles!
- Let’s make those edges black.
- Now we can remove more black edges, as they do not contribute to enough triangles.



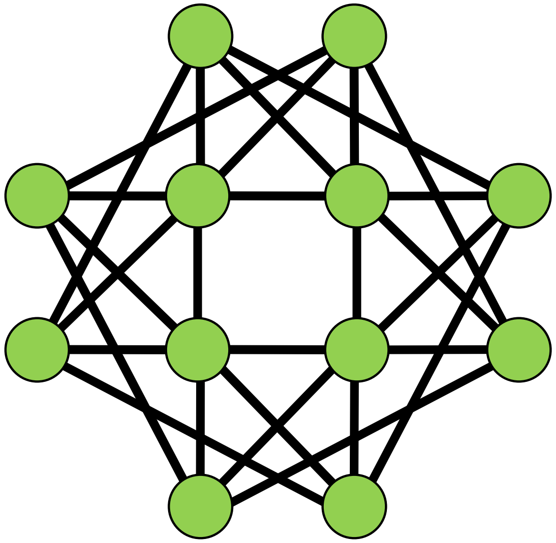
Each Removal Leads to More Removals

- Again, some of the edges are no longer in 2 triangles!
- Let's make those edges black.
- Finally, all edges are black!



Example Graph has no 4-Truss Portions

So the graph shown has no 4-truss portions!



ANY QUESTIONS?