18-344 Recitation 8

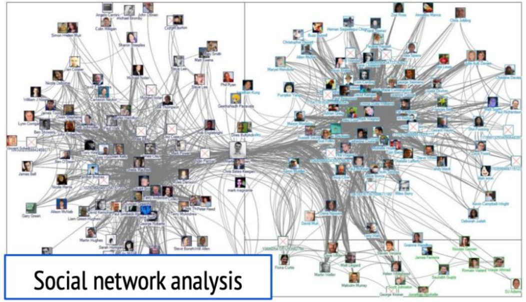
Lab4 - Graph Processing Optimization

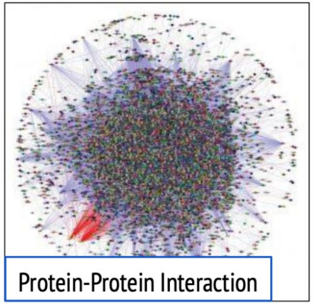
About Sparse Problems

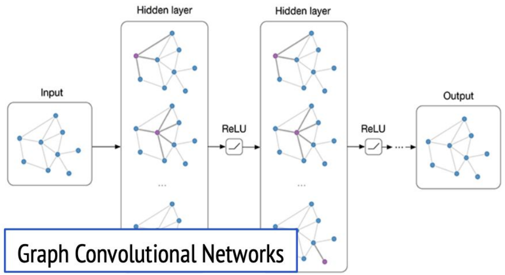
Sparse Problems

- What is a sparse problem? Why are they called “sparse”?
  - Graph Processing Problems are Sparse Problems
  - Machine Learning Problems are Sparse Problems
- What makes sparse problems hard?


Path Planning

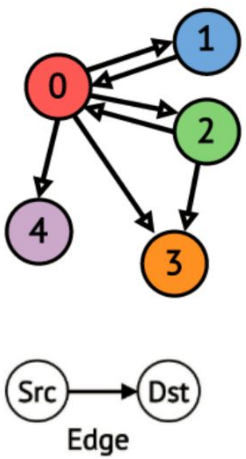

Social network analysis


Protein-Protein Interaction



Graph Convolutional Networks

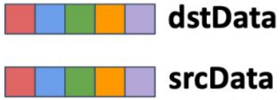
What does a graph processing program look like?



0	1
2	0
1	0
0	2
2	3
0	4
0	3

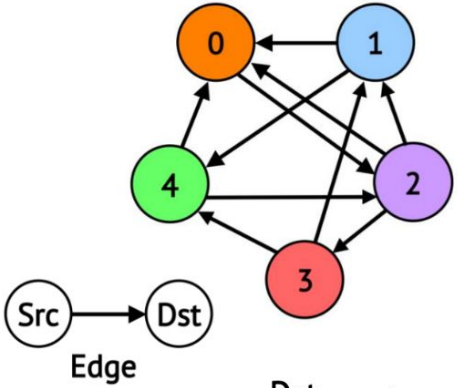
COO
(EdgeList)

```
for e in EL:
  dstData[e.dst] =
     f(srcData[e.src],dstData[e.dst])
```



stores vertex property information
if srcData == dstData, updating in-place;
often “swap” srcData & dstData from 1 iteration to the next iteration

Nobody EVER uses the adjacency matrix!



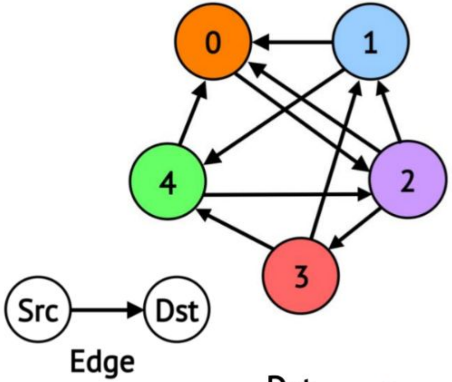
Dst →

Src ↓

	D_0	D_1	D_2	D_3	D_4
S_0			1		
S_1	1				1
S_2	1	1		1	
S_3		1			1
S_4	1		1		

Why would the Adjacency Matrix not be used?

Compressed Sparse Data Structures for Feasible Memory Size



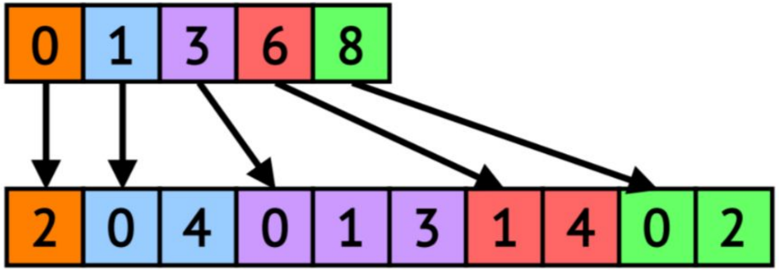
Dst →

Src ↓

	D_0	D_1	D_2	D_3	D_4
S_0			1		
S_1	1				1
S_2	1	1		1	
S_3		1			1
S_4	1		1		

Offsets Array (OA)

Neighbors Array (NA)



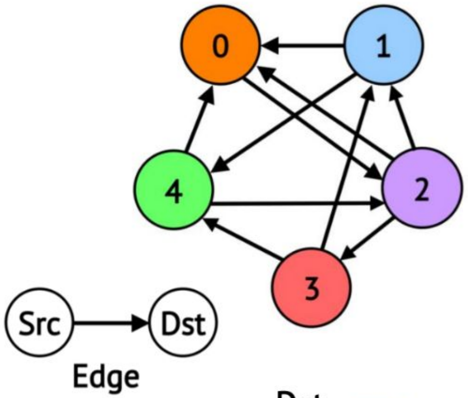
Compressed Sparse Row (CSR)
Outgoing Neighbors

Vertex Property Array
i.e., srcData / dstData

2	1	1	2	1

Often we will leave the vertex property array implicitly defined when we talk about sparse structures, but it is always there

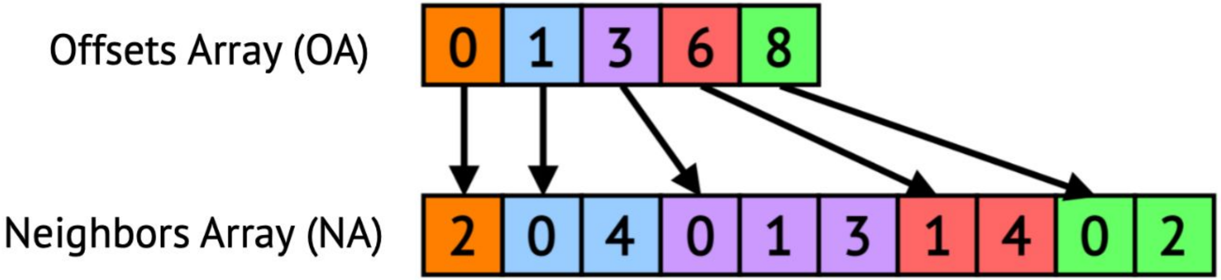
Compressed Sparse Data Structures for Feasible Memory Size



Dst →

Src ↓

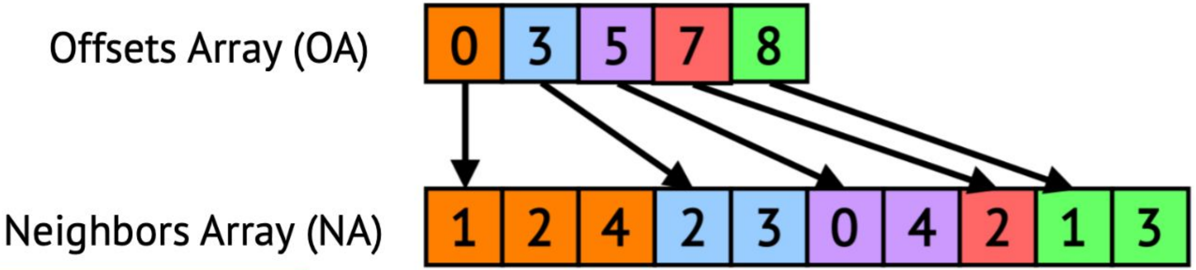
	D_0	D_1	D_2	D_3	D_4
S_0			1		
S_1	1				1
S_2	1	1		1	
S_3		1			1
S_4	1		1		



EdgeList sorted by SrcIDs

Compressed Sparse Row (CSR)
Outgoing Neighbors

The CSC is the *transpose* of the CSR



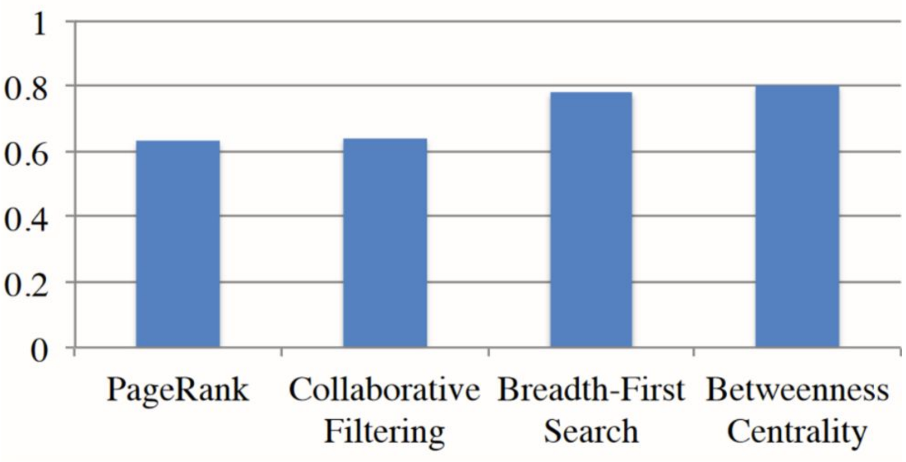
EdgeList sorted by DstIDs

Compressed Sparse Column (CSC)
Incoming Neighbors

Irregular Accesses Lead to Poor Locality

LLC Miss Rate (%)

Cycles stalled on DRAM / Total Cycles

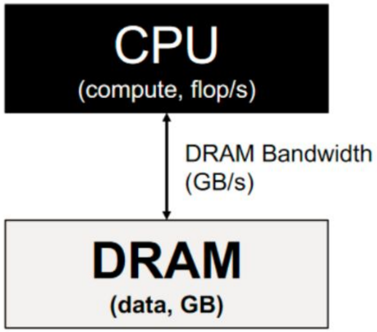


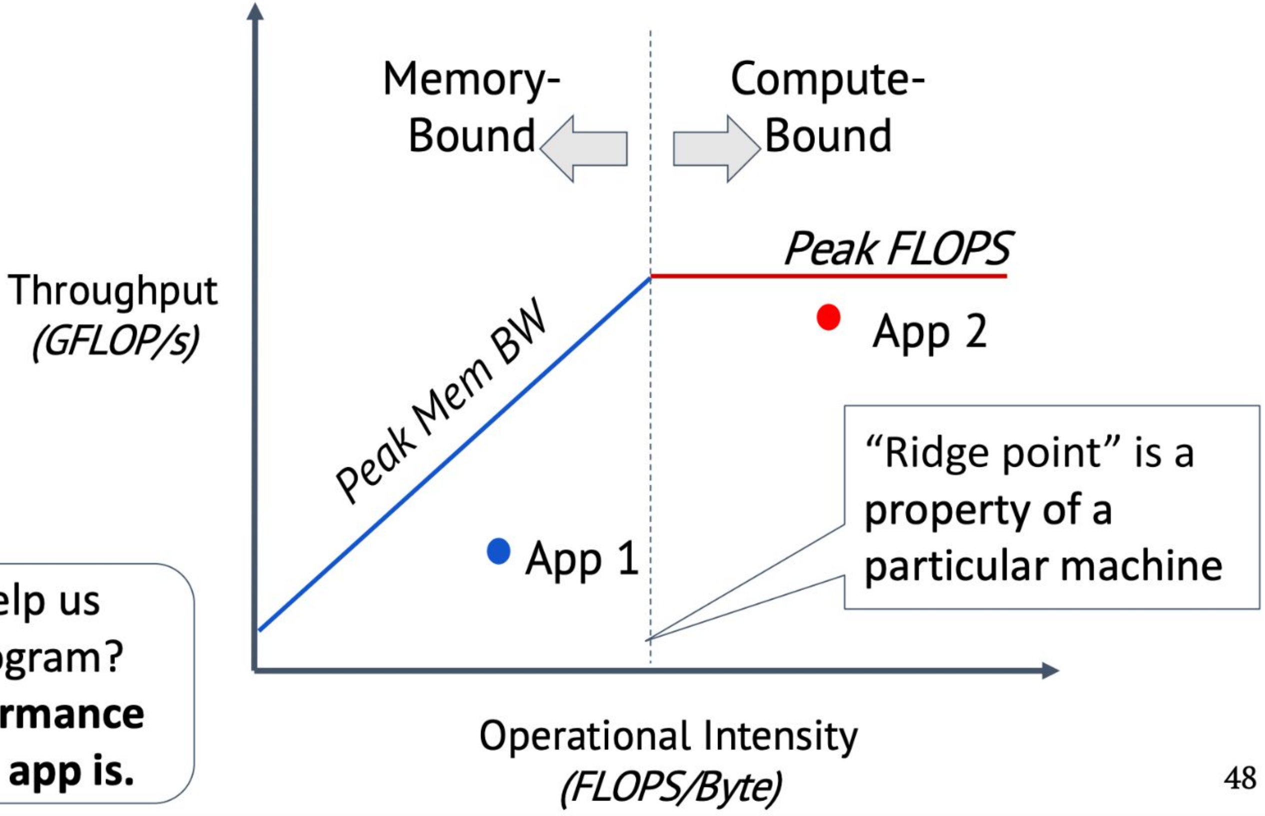
Problem: Sparse representations make processing large graphs feasible, but graph processing still entails a large working set with poor locality

Cache miss latency *cannot be hidden by anything else in the program*. Each miss incurs DRAM latency!

Right Figure from "Optimizing Cache Performance for Graph Analytics" ArXiv v1;

The Roofline Model





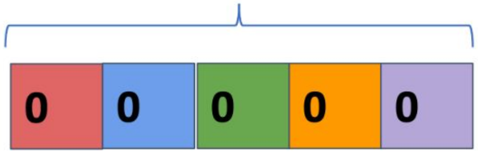
What does Roofline help us understand about a program?
Tell us what limits performance & how close to peak an app is.

Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache

0	1
2	0
1	0
0	2
2	3
0	4
0	3

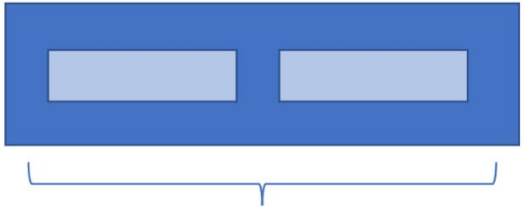
COO
(EdgeList)

|Domain| = |V| = 5 vertices



Recall: irregular accesses into vertex data array based on e.dst *which are essentially random*

Bad for the cache: the size of the *domain* of vertex data array entries is |V|, but the cache holds only |C| << |V| entries

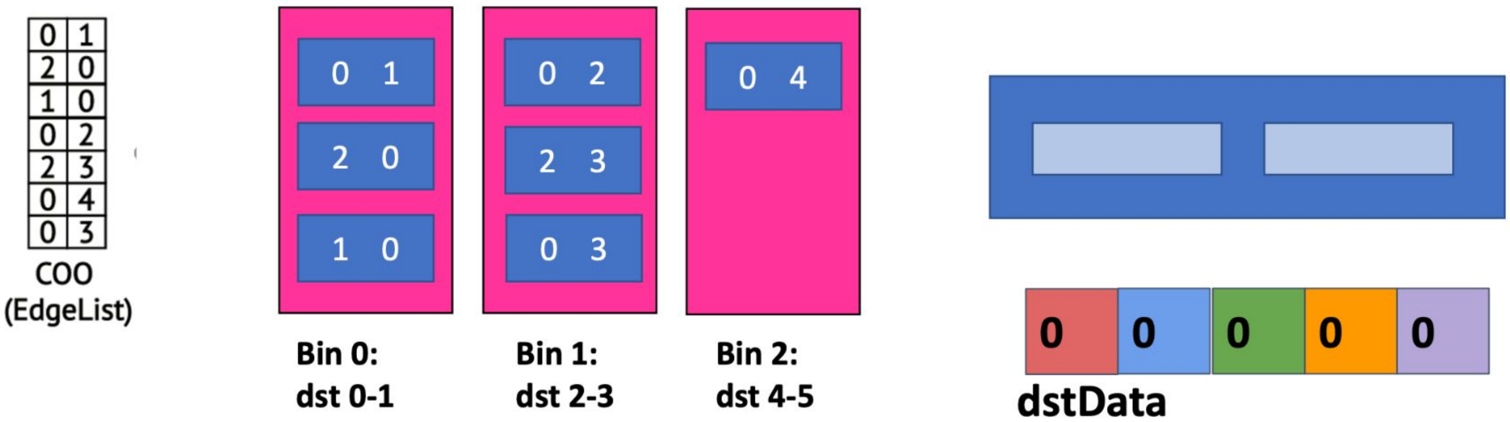


|Cache| = 2 vertices

Key idea in propagation blocking: Limit the domain of updates to a *sub-space* of vertices, **V***, so that |V*| <= |C| and do multiple sub-spaces of V*s, so that all V*s together = V

Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache

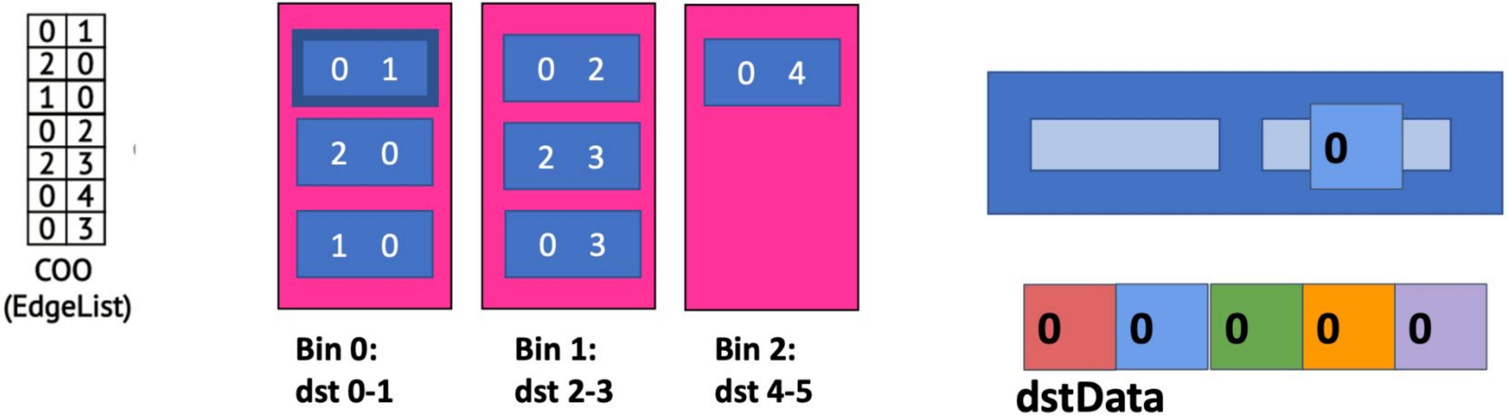
Create "Bins" that hold input elements (edges from the edge list)



Remember: dstData[e.dst] ++
and e.dst is random, from edge list

Execute the kernel for one bin at a time

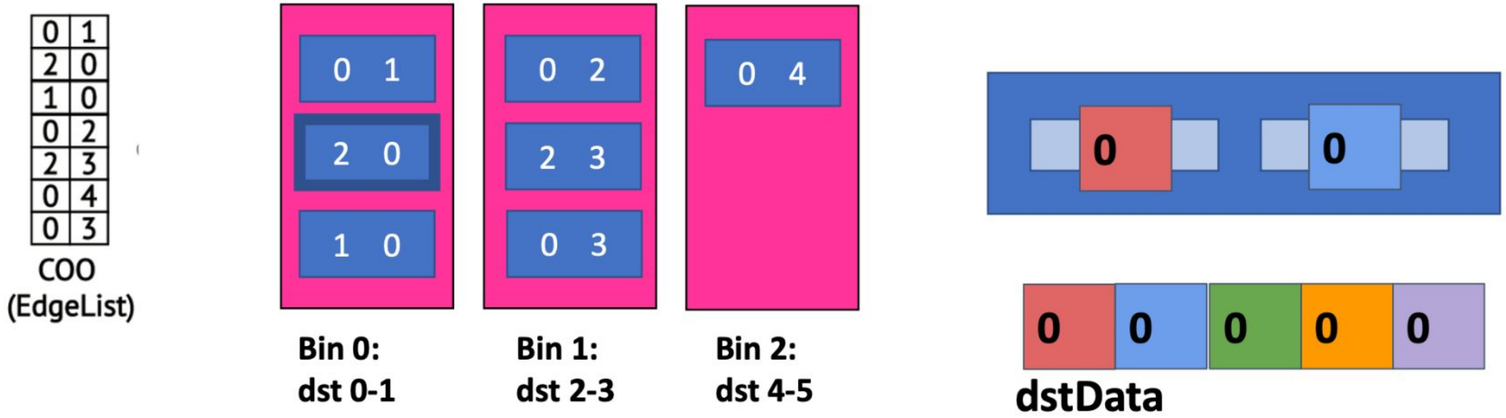
Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache



Execute the kernel for one bin at a time

Remember: dstData[e.dst] ++
and e.dst is random, from edge list

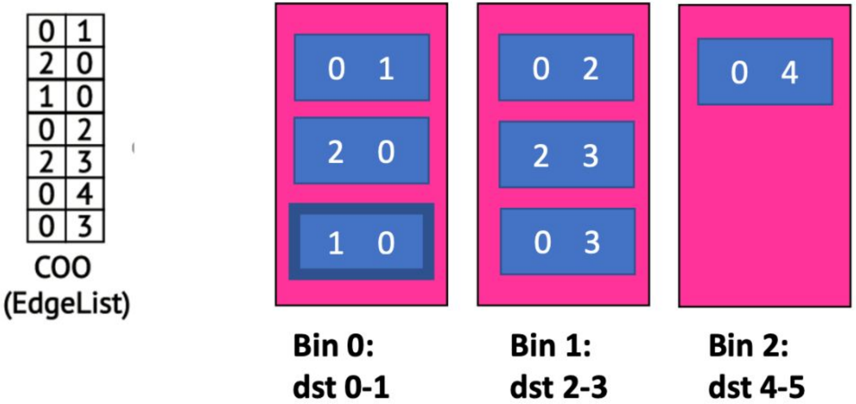
Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache



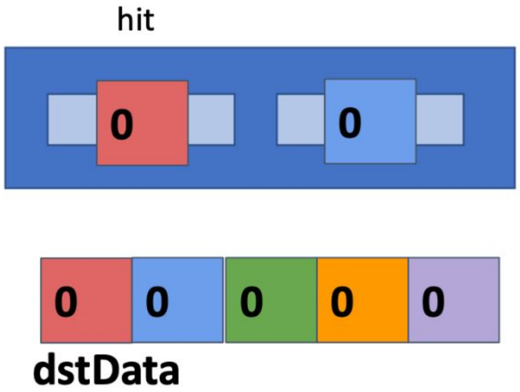
Execute the kernel for one bin at a time

Remember: dstData[e.dst] ++
and e.dst is random, from edge list

Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache

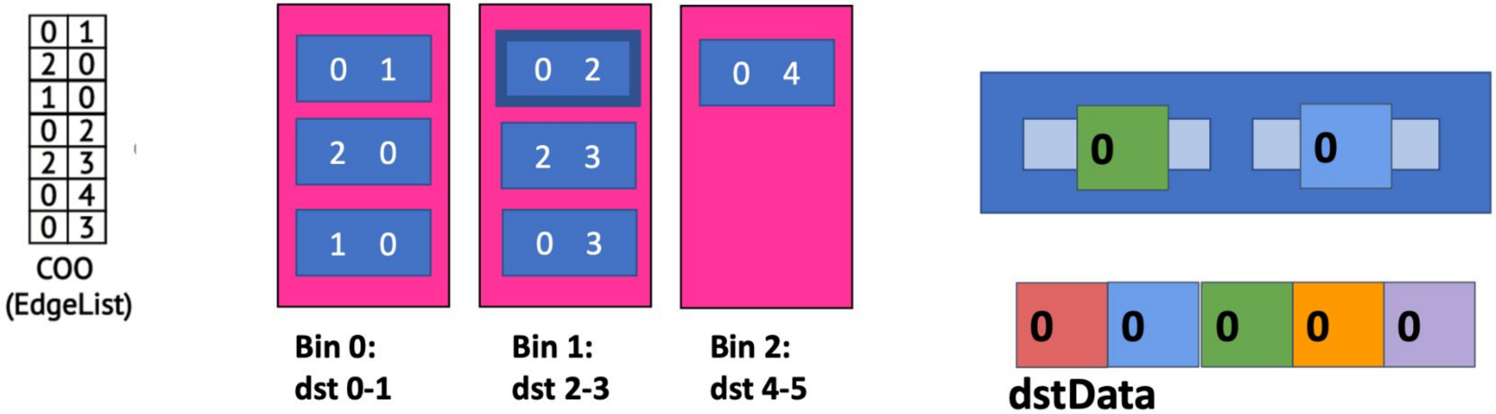


Execute the kernel for one bin at a time



Remember: dstData[e.dst] ++
and e.dst is random, from edge list

Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache



Remember: dstData[e.dst] ++
and e.dst is random, from edge list

Execute the kernel for one bin at a time

…

example continues in lecture slides

…

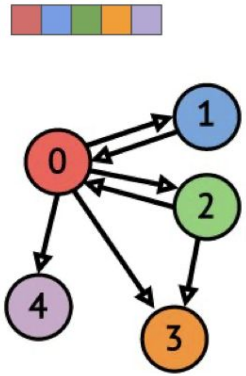
Lab Details

Overview

- Task: Rewrite a Graph Processing Kernel to be more cache-friendly
- Kernel: Converting edge-list to CSR
- Evaluation Metric: Cache metrics
  - Use your lab2 cache simulator pintool to measure metrics
  - If you prefer, we have provided you with a cache simulator that you can use for this lab
    - `memory-hierarchy.so` in `/afs/ece.cmu.edu/class/ece344/assign/`
- Study sensitivity to bin size, graph size and cache configurations

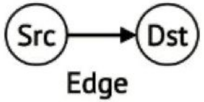
Edge List to CSR Conversion

Functions in csr.c



0	1
2	0
1	0
0	2
2	3
0	4
0	3

COO
(EdgeList)



CSR_EL_count_neigh()

Step 1:

```
for e in EL:
  neigh_count[e.src]++; /*e.src*/
```

2	1	1	2	1

neigh_count

CSR_cumul_neigh_count()

Step 2:

```
OA[0] = 0
for vtx in [1, MAX_VTX]:
  OA[ vtx ] = OA[vtx-1] + neigh_count[vtx-1]
```

Step 3:

CSR_EL_neigh_pop()

```
for e in EL:
  NA[ OA[e.src]++ ] = e.dst
```

0	2	3	4	6

OA

1	2	0	0	0	2	0

NA

Completed Result

Your Task - Propagation Blocking

You will replace Steps 1 and 3 with a binned version

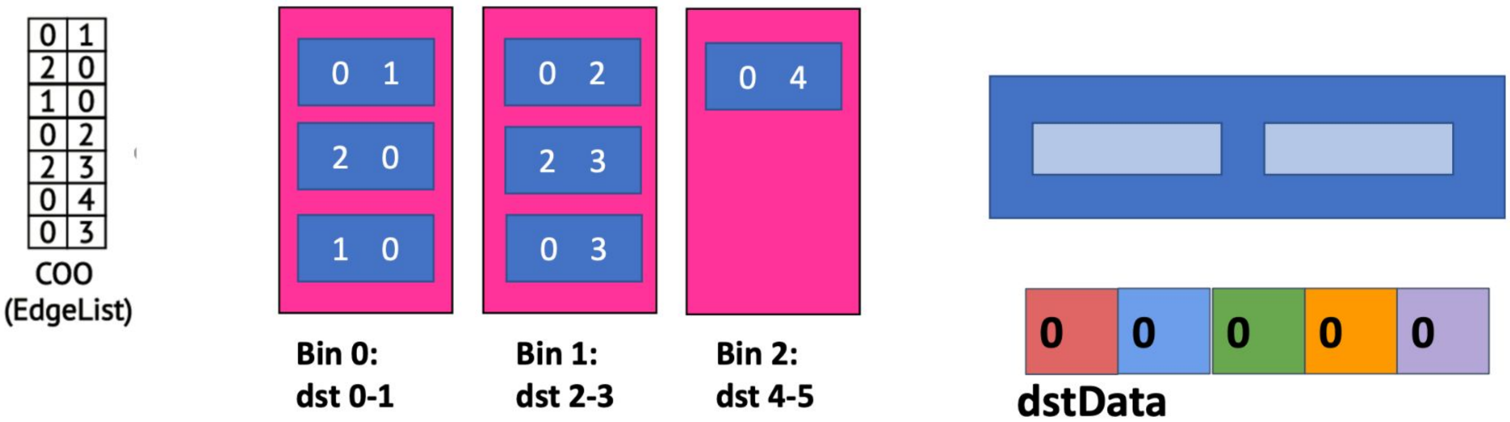
New Steps for EL2CSR conversion:

- Step 1: Traverse EL, populate bins
  - choice of src/dest vtx for binning up to you
- Step 2: Generate neigh_count array, a bin at a time
- Step 3: Sequential Accumulation to generate OA
- Step 4: Generate NA, a bin at a time

Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache

RECALL

Create "Bins" that hold input elements (edges from the edge list)



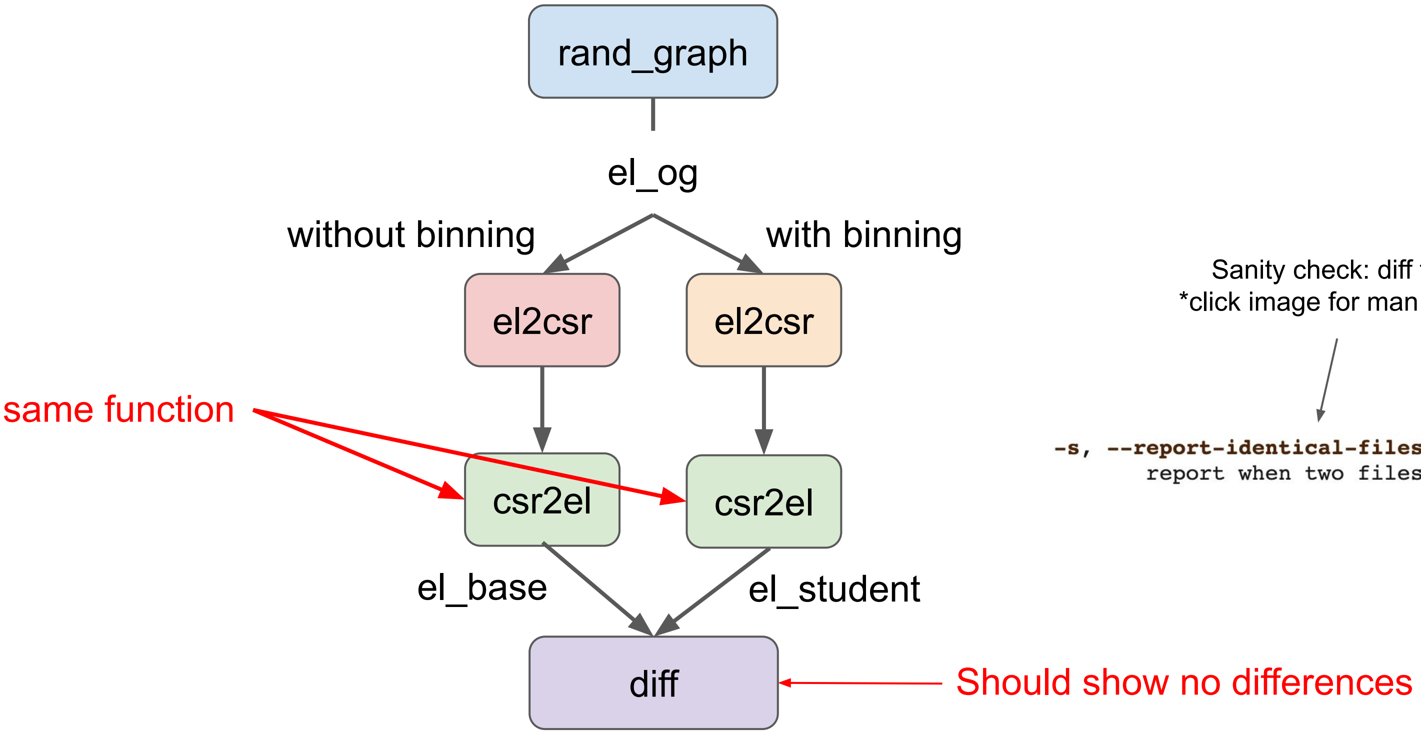
Execute the kernel for one bin at a time

Remember: dstData[e.dst] ++
and e.dst is random, from edge list

Testing Correctness

- Step 1: Generate a graph with some edges using rand_graph, output is an edgelist -- el_og
- Step 2: Use base implementation (provided by default) to convert edgelist to CSR, using el2csr
- Step 3: Convert CSR back to EL using the csr2el program (this sorts the edgelist → use this output to compare your implementation) -- el_base
- Step 4: Run your PB implementation to convert edgelist to CSR
- Step 5: Convert your CSR to EL (using csr2el) program -- el_student
- Step 6: diff el_base el_student
  - This should show no difference between the two EL

Testing Correctness | Flow Chart



Sanity check: diff flag
click image for man page

```
-s, --report-identical-files
       report when two files are the same
```

Evaluation

- You will use the Cache Simulator pintool you developed in Lab 2
  - Or the pintool provided in assign folder
- Your implementation of el2csr will be the input binary to the Cache pintool.
- You will measure appropriate metrics to report cache performance.
- Test different cache configurations, bin sizes and graph sizes (#edges and/or #vertices)
- Recommend checking out Stanford Large Network Dataset Collection

NOTE: If you want to test graphs with different vertices, you should change MAX_VTX in graph.h and rebuild everything

Impromptu Office Hours