CMU 18-344: Computer Systems and the Hardware/Software Interface

Fall 2024, Prof. Brandon Lucia

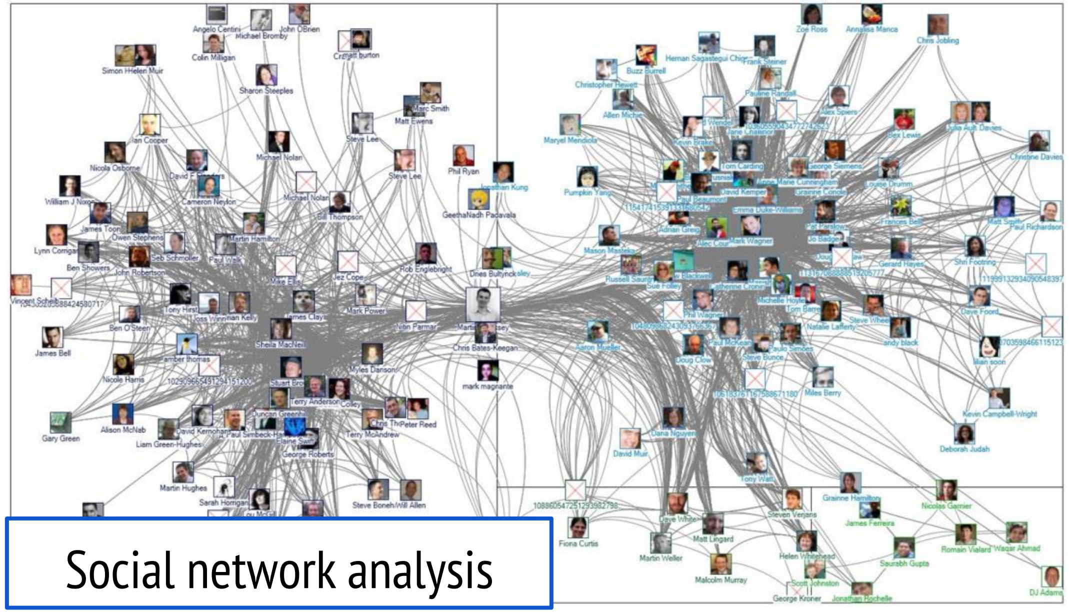
Today: Sparse Problems

- What is a sparse problem? Why are they called “sparse”?
- What makes sparse problems hard?
- Roofline performance modeling
- Hardware and software strategies for optimizing sparse problems

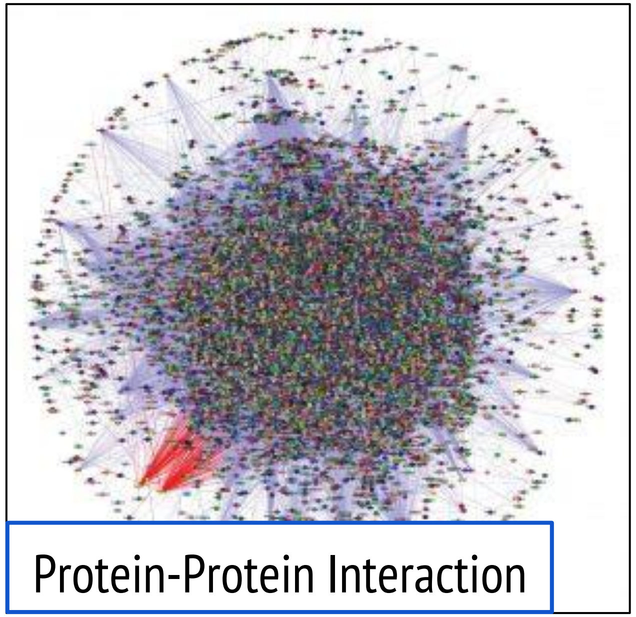
(with acknowledgements to Vignesh Balaji, CMU ECE PhD 2021, now at Nvidia for contributions to this material)

Graph Processing Problems are Sparse Problems

Path Planning



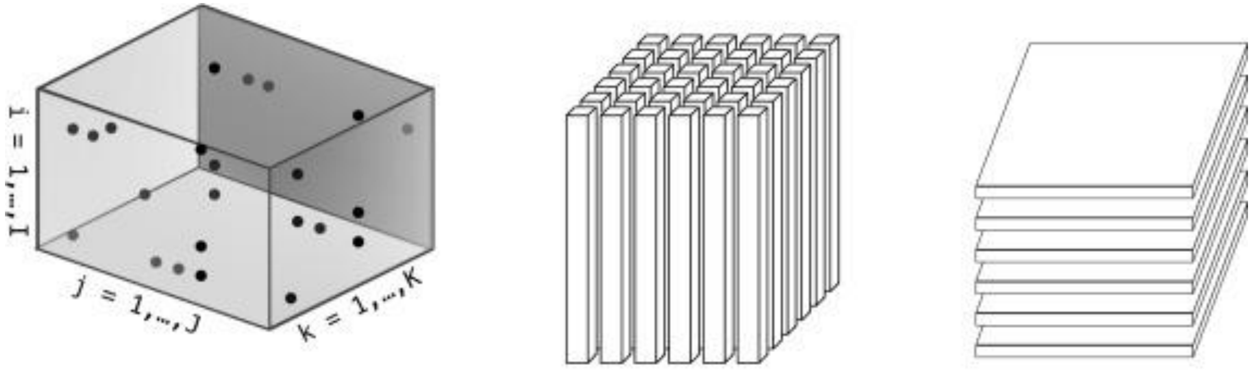
Social network analysis



Protein-Protein Interaction

The canonical examples of sparse problems are graph processing applications.

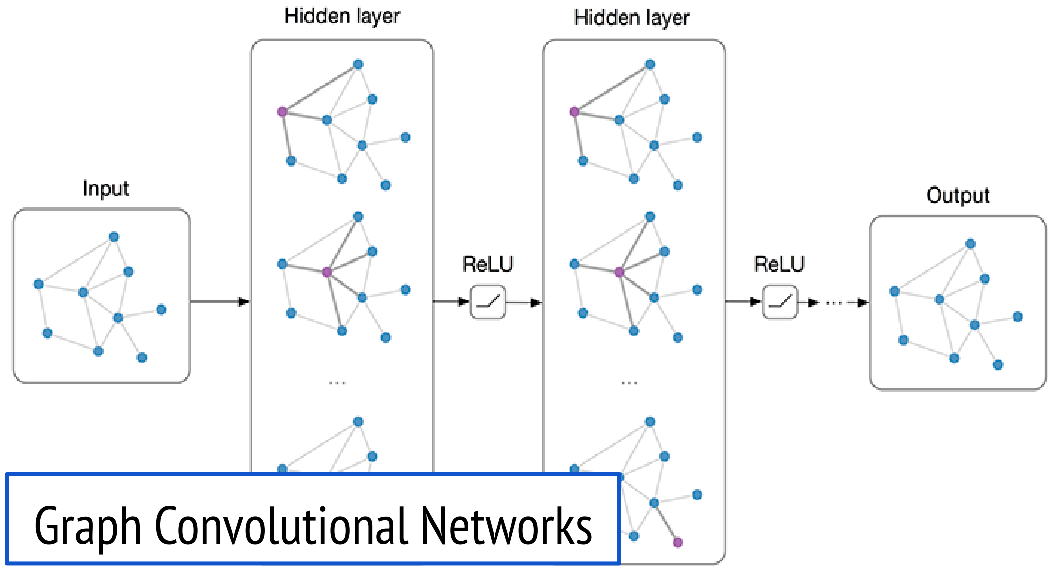
Machine Learning Problems are Sparse Problems



Data Mining

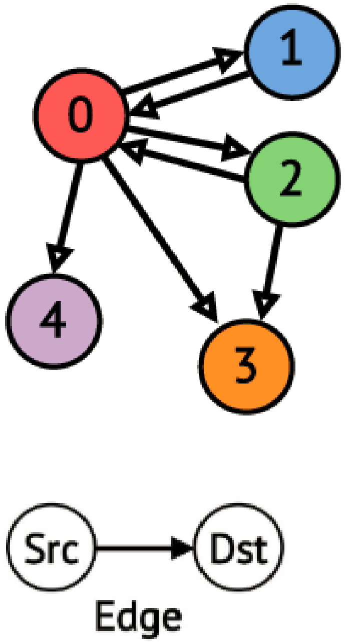
(b) Mode-1 fibers:
$\mathbf{f}_{:jk} = \mathcal{X}(:, j, k)$

(c) Slices:
$\mathbf{S}_{::k} = \mathcal{X}(:, :, k)$



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)

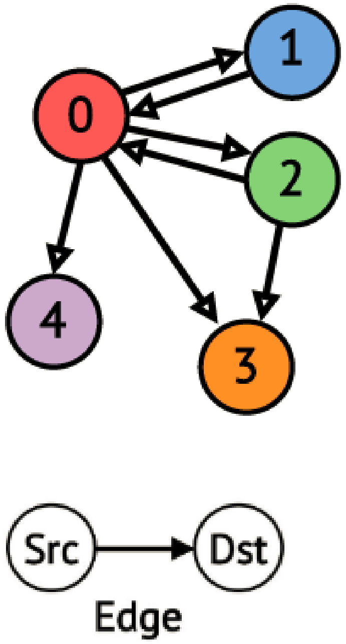
```
for e in EL:
  dstData[e.dst] =
      f(srcData[e.src],dstData[e.dst])
```

🟥🟦🟩🟧🟪 **dstData**

🟥🟦🟩🟧🟪 **srcData**

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

What does a graph processing program look like?



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

COO
(EdgeList)

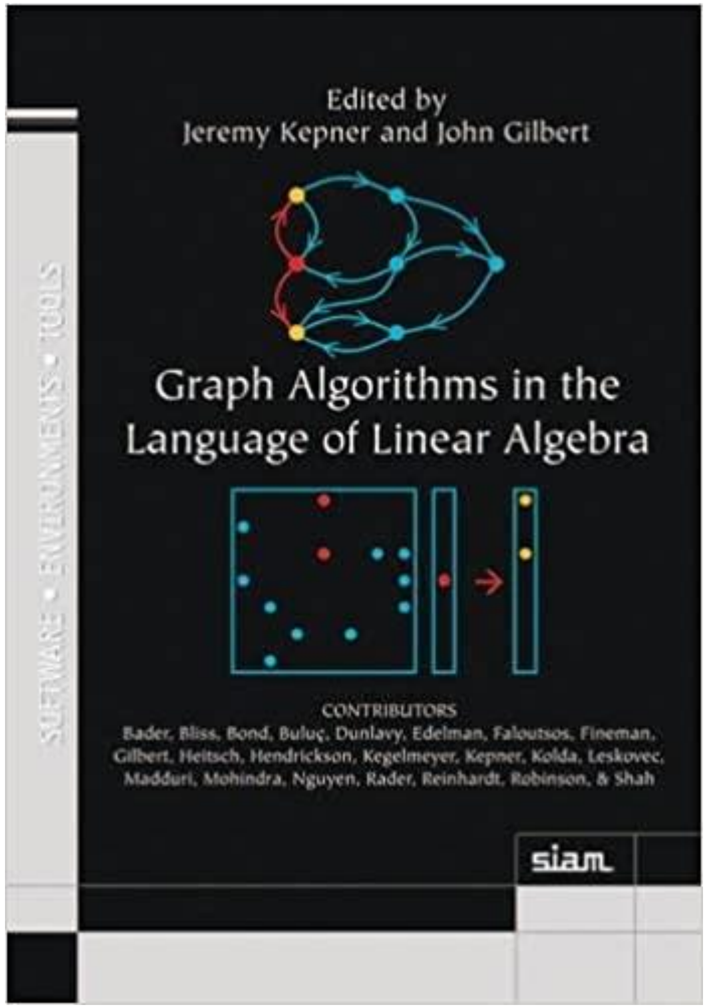
🟥🟦🟩🟧🟪 dstData

🟥🟦🟩🟧🟪 srcData

```
PageRank(-ish){
  for e in EL:
    rank_n[e.dst] =
      rank_nminus1[e.src] + rank_n[e.dst]
}
```

rank_n is a webpage's rank in this iteration,
rank_nminus1 is rank_n from the last iteration

Graph Analytics can be mapped to Sparse Linear Algebra



SOFTWARE • ENVIRONMENTS • TOOLS

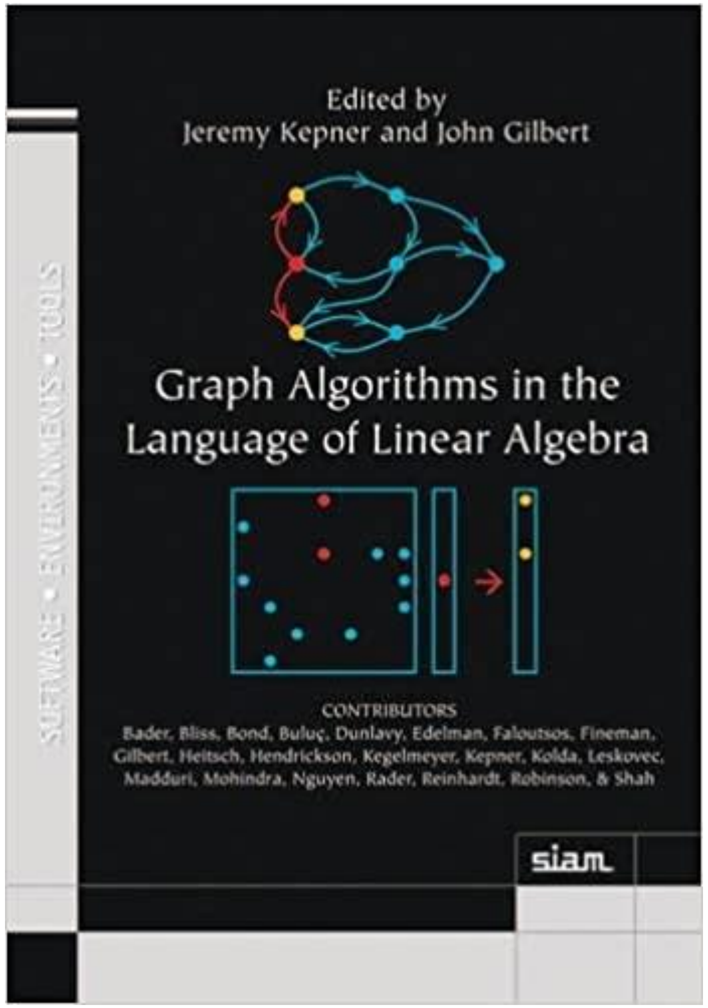
Edited by
Jeremy Kepner and John Gilbert

Graph Algorithms in the
Language of Linear Algebra

CONTRIBUTORS
Bader, Bliss, Bond, Buluç, Dunlavy, Edelman, Faloutsos, Fineman,
Gilbert, Heitsch, Hendrickson, Kegelmeyer, Kepner, Kolda, Leskovec,
Madduri, Mohindra, Nguyen, Rader, Reinhardt, Robinson, & Shah

siam

Graph Analytics can be mapped to Sparse Linear Algebra



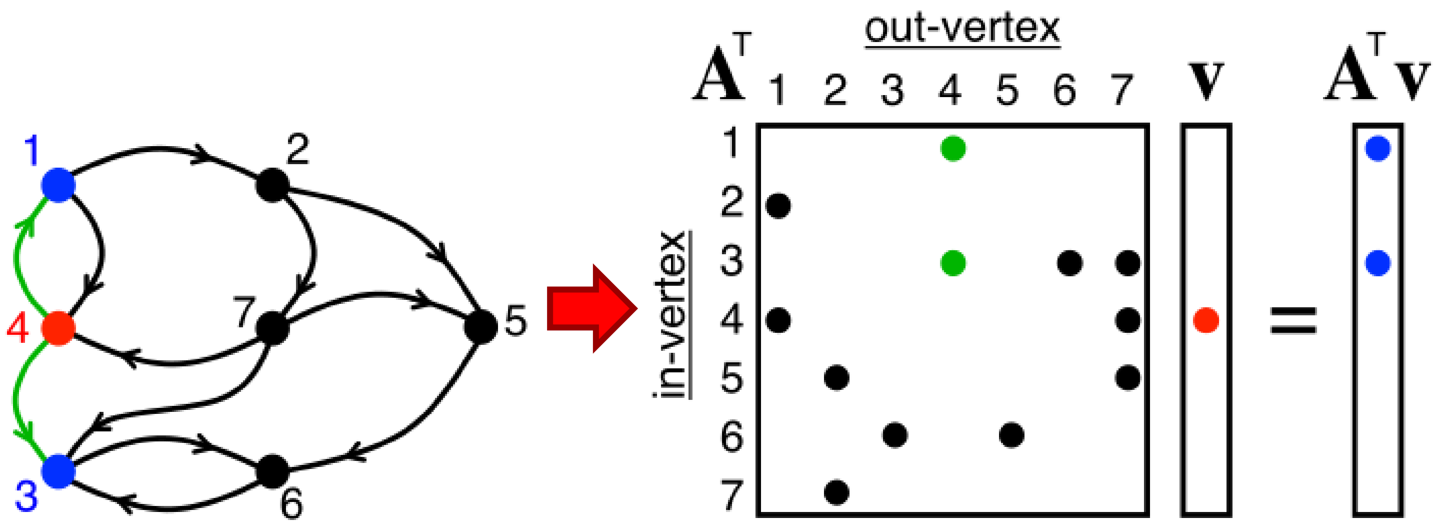
SOFTWARE • ENVIRONMENTS • TOOLS

Edited by
Jeremy Kepner and John Gilbert

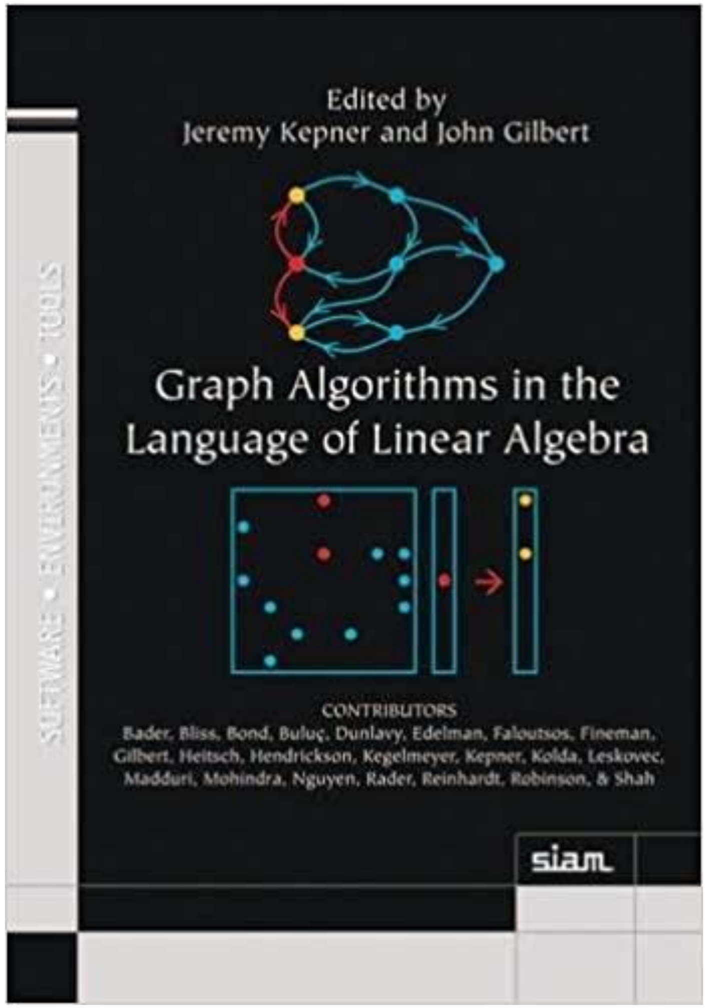
Graph Algorithms in the Language of Linear Algebra

CONTRIBUTORS
Bader, Bliss, Bond, Buluç, Dunlavy, Edelman, Faloutsos, Fineman, Gilbert, Heitsch, Hendrickson, Kegelmeyer, Kepner, Kolda, Leskovec, Madduri, Mohindra, Nguyen, Rader, Reinhardt, Robinson, & Shah

siam



Graph Analytics can be mapped to Sparse Linear Algebra



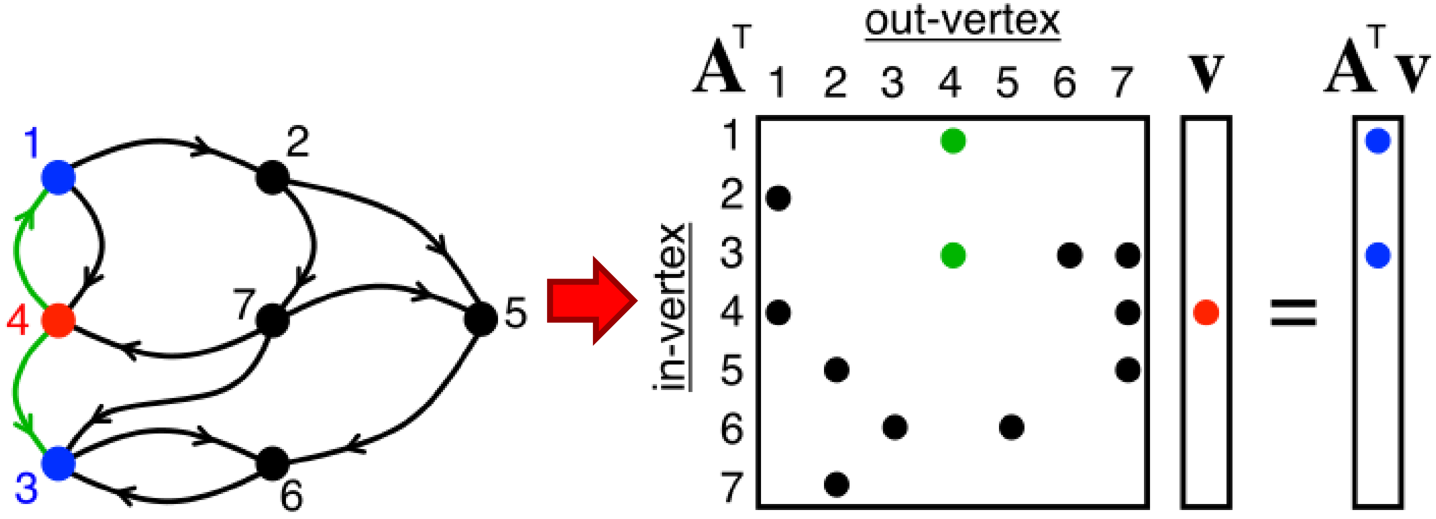
SOFTWARE • ENVIRONMENTS • TOOLS

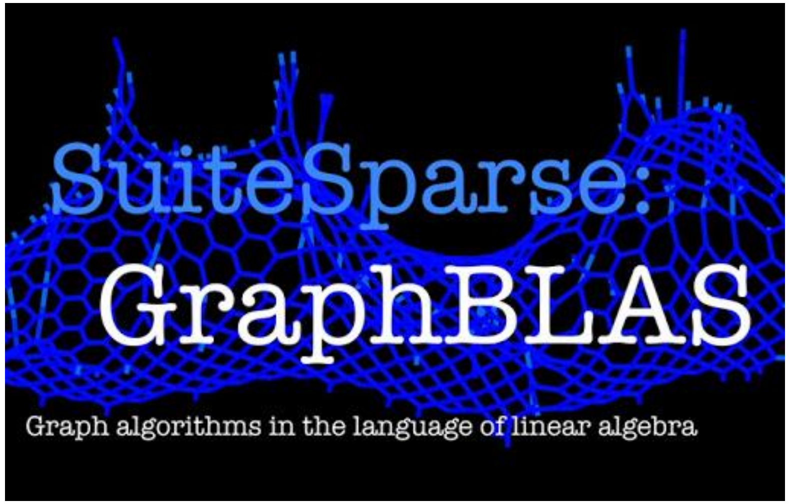
Edited by
Jeremy Kepner and John Gilbert

Graph Algorithms in the Language of Linear Algebra

CONTRIBUTORS
Bader, Bliss, Bond, Buluç, Dunlavy, Edelman, Faloutsos, Fineman, Gilbert, Heitsch, Hendrickson, Kegelmeyer, Kepner, Kolda, Leskovec, Madduri, Mohindra, Nguyen, Rader, Reinhardt, Robinson, & Shah

siam

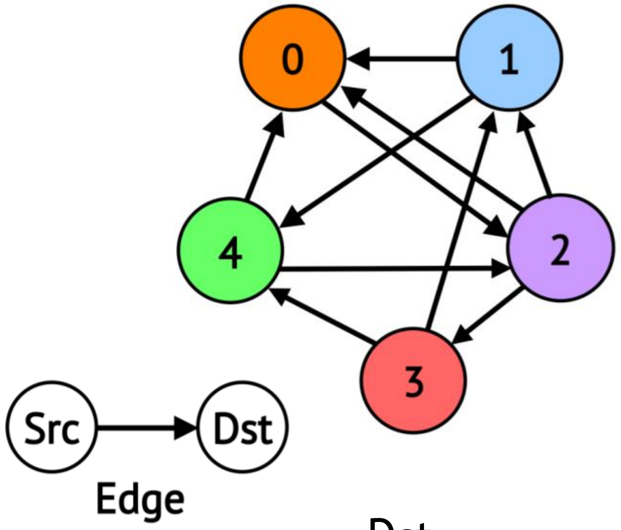




SuiteSparse:
GraphBLAS

Graph algorithms in the language of linear algebra

How do graph applications correspond to linear algebra?

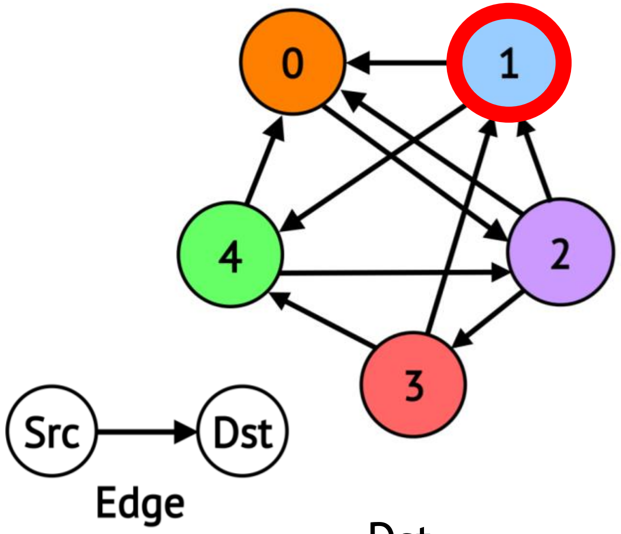


Dst ⟶

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		

How do graph applications correspond to linear algebra?



Matrix-transpose-vector product is one BFS iteration

$$\mathbf{A}^{\mathbf{T}} x_i = x_{i+1}$$

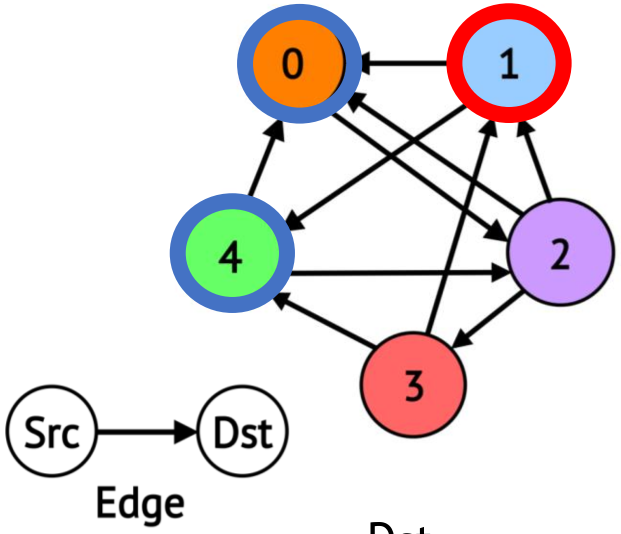
Initial x_i vector is starting vertex for BFS.

Dst ⟶

Src ↓

$$\left(\begin{array}{c|c:c:c:c:c|} & D_0 & D_1 & D_2 & D_3 & D_4 \\ \hline S_0 & & & 1 & & \\ \hdashline S_1 & 1 & & & & 1 \\ \hdashline S_2 & 1 & 1 & & 1 & \\ \hdashline S_3 & & 1 & & & 1 \\ \hdashline S_4 & 1 & & 1 & & \\ \hline \end{array}\right)^{\mathbf{T}} \begin{array}{|c|} \hline \\ \hdashline 1 \\ \hdashline \\ \hdashline \\ \hdashline \\ \hline \end{array} = \begin{array}{|c|} \hline 1 \\ \hdashline \\ \hdashline \\ \hdashline \\ \hdashline 1 \\ \hline \end{array}$$

How do graph applications correspond to linear algebra?



Matrix-transpose-vector product is one BFS iteration

$$\mathbf{A}^T x_i = x_{i+1}$$

A Transpose

Initial x_i vector is starting vertex for BFS.

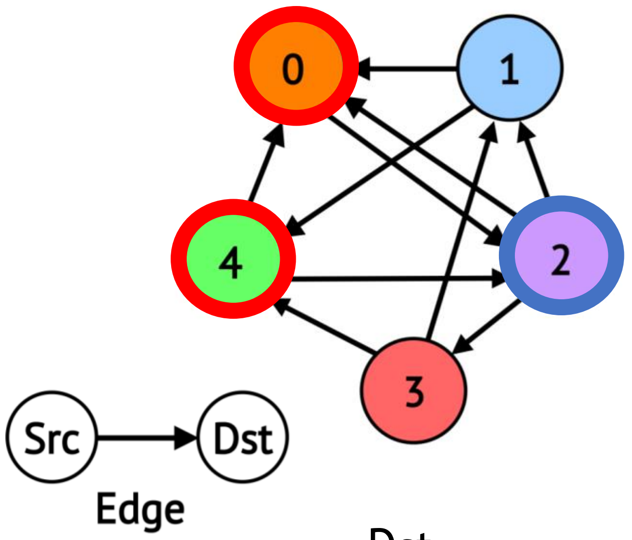
Initial x_{i+1} is vertices reachable from x_i

Dst ⟶

Src ↓

$$\left(\begin{array}{c|ccccc} & D_0 & D_1 & D_2 & D_3 & D_4 \\ \hline S_0 & & & 1 & & \\ S_1 & 1 & & & & 1 \\ S_2 & 1 & 1 & & 1 & \\ S_3 & & 1 & & & 1 \\ S_4 & 1 & & 1 & & \end{array}\right)^{\mathbf{T}} \overset{\mathbf{x_i}}{\begin{bmatrix} \ \\ 1 \\ \ \\ \ \\ \ \end{bmatrix}} = \overset{\mathbf{x_{i+1}}}{\begin{bmatrix} 1 \\ \ \\ \ \\ \ \\ 1 \end{bmatrix}}$$

How do graph applications correspond to linear algebra?



Matrix transpose vector product is one BFS iteration

$$\mathbf{A}^{\mathsf{T}} x_i = x_{i+1}$$

The next iteration is computed by performing the next matrix transpose vector product

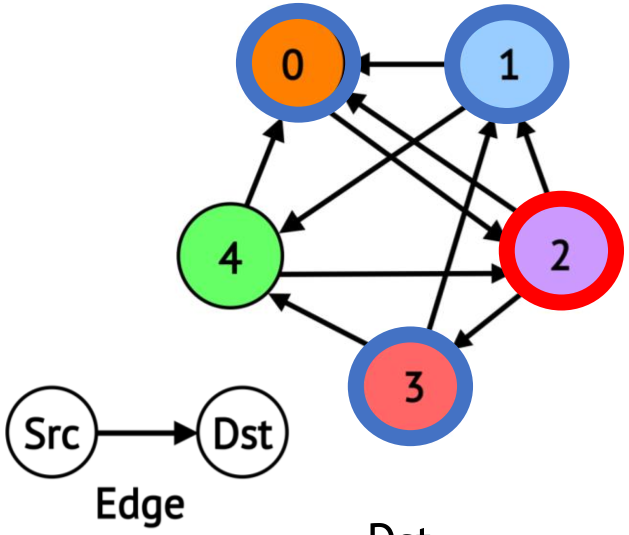
$$\begin{array}{c} 1 \\ 1 \\ \\ \\ \\ \end{array} \qquad \begin{array}{c} 1 \\ \\ \\ \\ 1 \end{array}$$

Dst ⟶

Src ↓

$$\left(\begin{array}{c|ccccc} & D_0 & D_1 & D_2 & D_3 & D_4 \\ \hline S_0 & & & 1 & & \\ S_1 & 1 & & & & 1 \\ S_2 & 1 & 1 & & 1 & \\ S_3 & & 1 & & & 1 \\ S_4 & 1 & & 1 & & \end{array}\right)^{\mathbf{T}} \begin{array}{c} \mathbf{x_i} \\ \hline 1 \\ \\ \\ \\ 1 \end{array} = \begin{array}{c} \mathbf{x_{i+1}} \\ \hline \\ \\ 1 \\ \\ \\ \end{array}$$

How do graph applications correspond to linear algebra?



Matrix transpose vector product is one BFS iteration

$$\mathbf{A}^{\mathbf{T}}x_i = x_{i+1}$$

The next iteration is computed by performing the next matrix transpose vector product

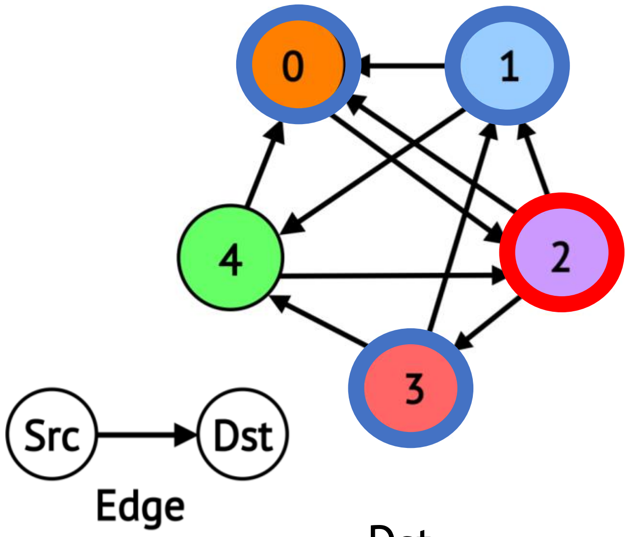
$$\begin{bmatrix} \\ 1 \\ \\ \\ \\ \end{bmatrix} \quad \begin{bmatrix} 1 \\ \\ \\ \\ 1 \end{bmatrix} \quad \begin{bmatrix} \\ \\ 1 \\ \\ \\ \end{bmatrix}$$

Dst ⟶

Src ↓

$$\begin{array}{c|ccccc|} & D_0 & D_1 & D_2 & D_3 & D_4 \\ \hline S_0 & & & 1 & & \\ S_1 & 1 & & & & 1 \\ S_2 & 1 & 1 & & 1 & \\ S_3 & & 1 & & & 1 \\ S_4 & 1 & & 1 & & \\ \hline \end{array}^{\mathbf{T}} \overset{\mathbf{x_i}}{\begin{bmatrix} \\ \\ 1 \\ \\ \\ \end{bmatrix}} = \overset{\mathbf{x_{i+1}}}{\begin{bmatrix} 1 \\ 1 \\ \\ 1 \\ \\ \end{bmatrix}}$$

How do graph applications correspond to linear algebra?



Dst →

Src ↓

$$\begin{array}{c|ccccc} & D_0 & D_1 & D_2 & D_3 & D_4 \\ \hline S_0 & & & 1 & & \\ S_1 & 1 & & & & 1 \\ S_2 & 1 & 1 & & 1 & \\ S_3 & & 1 & & & 1 \\ S_4 & 1 & & 1 & & \end{array}^{\mathbf{T}} \quad \overset{\mathbf{x_i}}{\begin{bmatrix} \\ \\ 1 \\ \\ \\ \end{bmatrix}} = \overset{\mathbf{x_{i+1}}}{\begin{bmatrix} 1 \\ 1 \\ \\ 1 \\ \\ \end{bmatrix}}$$

Matrix transpose vector product is one BFS iteration

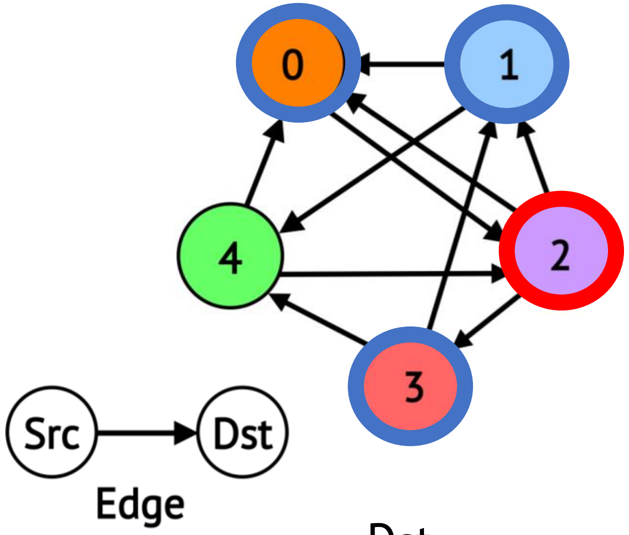
$$\mathbf{A}^{\mathbf{T}} x_i = x_{i+1}$$

The next iteration is computed by performing the next matrix transpose vector product

$$\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \\ 1 \\ \\ \\ \\ \end{bmatrix} \begin{bmatrix} 1 \\ \\ \\ \\ 1 \end{bmatrix} \begin{bmatrix} \\ \\ 1 \\ \\ \\ \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ \\ 1 \\ \\ \end{bmatrix}$$

Search done when no new vertices added (or all visited)

How do graph applications correspond to linear algebra?



Turns out that other graph applications also correspond to roughly this formulation if you change the operations you use (min/+ instead of +/*) or consider weighted edges

$$A^T x_i = x_{i+1}$$

SSSP, BFS, PageRank, Connected-Components, Betweenness-Centrality, triangle counting… BFS is a representative sparse problem.

Dst ⟶

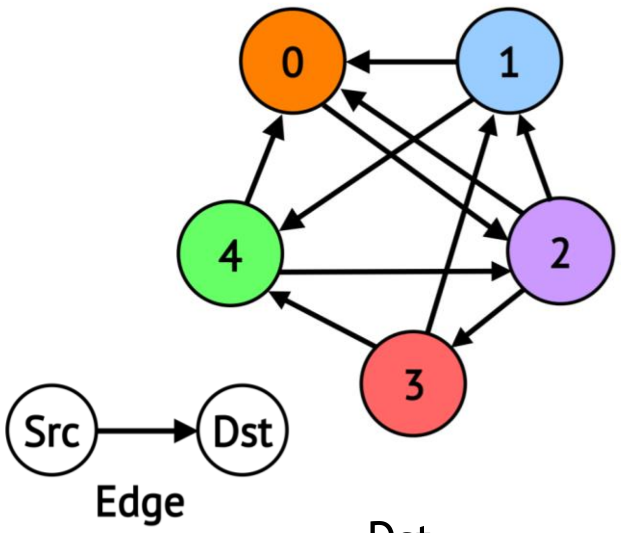
Src ↓

$$\begin{array}{c|ccccc} & D_0 & D_1 & D_2 & D_3 & D_4 \\ \hline S_0 & & & 1 & & \\ S_1 & 1 & & & & 1 \\ S_2 & 1 & 1 & & 1 & \\ S_3 & & 1 & & & 1 \\ S_4 & 1 & & 1 & & \end{array}^{\mathbf{T}} \overset{\mathbf{x_i}}{\begin{bmatrix} \\ \\ 1 \\ \\ \\ \end{bmatrix}} = \overset{\mathbf{x_{i+1}}}{\begin{bmatrix} 1 \\ 1 \\ \\ 1 \\ \\ \end{bmatrix}}$$

$$\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \\ 1 \\ \\ \\ \\ \end{bmatrix} \begin{bmatrix} 1 \\ \\ \\ \\ 1 \end{bmatrix} \begin{bmatrix} \\ \\ 1 \\ \\ \\ \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ \\ 1 \\ \\ \end{bmatrix}$$

Search done when no new vertices added (or all visited)

Nobody EVER uses the adjacency matrix!



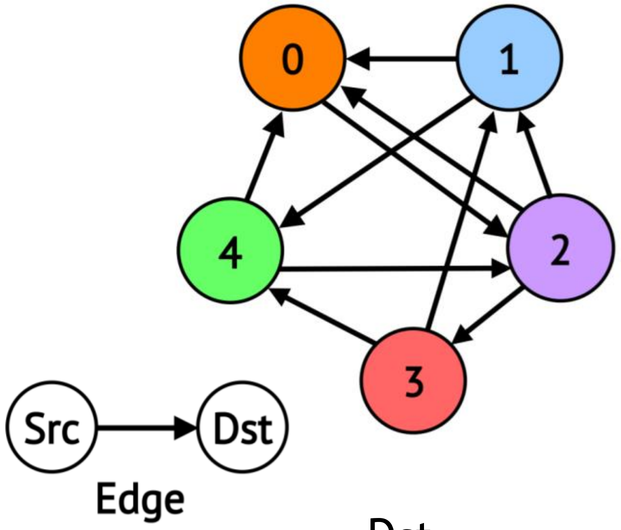
Dst ⟶

Src ↓

	D0	D1	D2	D3	D4
S0			1		
S1	1				1
S2	1	1		1	
S3		1			1
S4	1		1		

Why would the Adjacency Matrix not be used?

Nobody EVER uses the adjacency matrix!



Dst ⟶

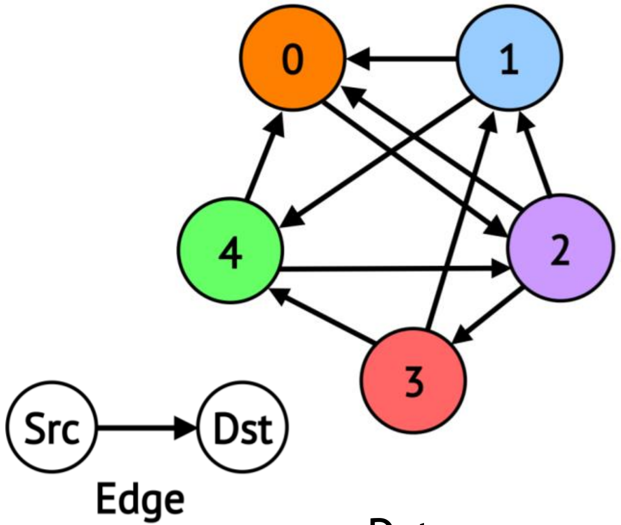
Src ↓

	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		

Reasons Adjacency Matrix is never used:

- **Sparsity:** % of Non-Zero Entries ~ 10^{-5}
- **Total Size:** 32M nodes => (32M * 32M) = 1PB

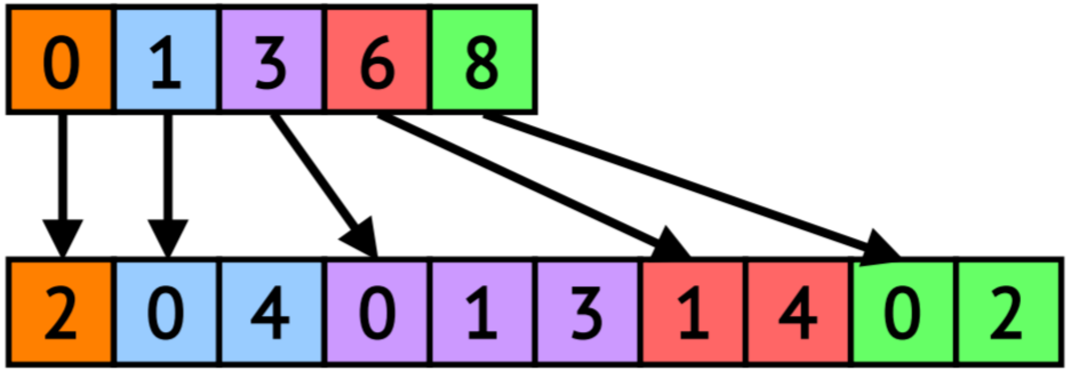
Compressed Sparse Data Structures for Feasible Memory Size



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		



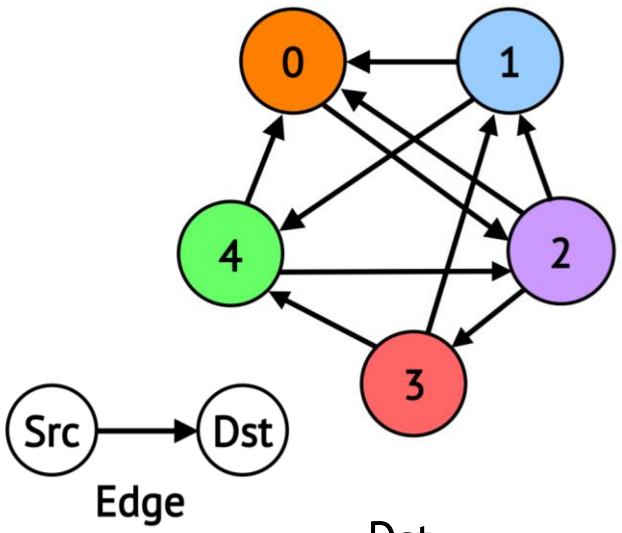
Offsets Array (OA)	0	1	3	6	8					
Neighbors Array (NA)	2	0	4	0	1	3	1	4	0	2

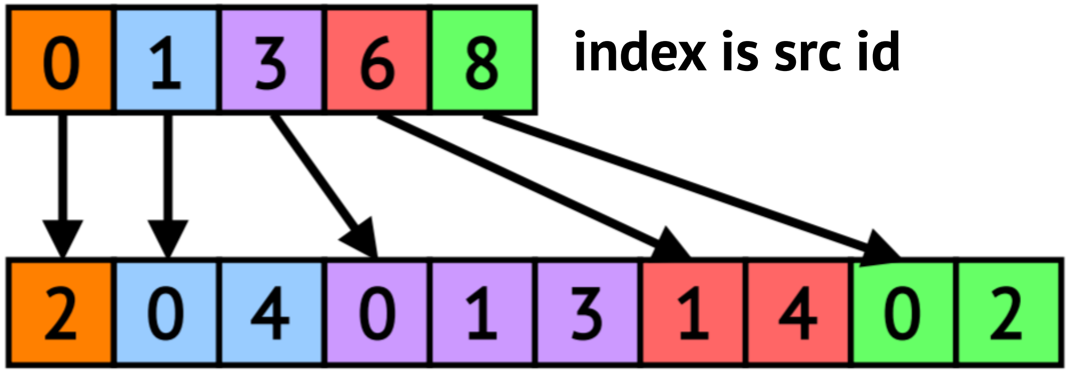
Compressed Sparse Row (CSR)
Outgoing Neighbors

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





Offsets Array (OA) 0 1 3 6 8 **index is src id**

Neighbors Array (NA) 2 0 4 0 1 3 1 4 0 2

EdgeList sorted by SrcIDs

Compressed Sparse Row (CSR)
Outgoing Neighbors

Dst →

Src ↓

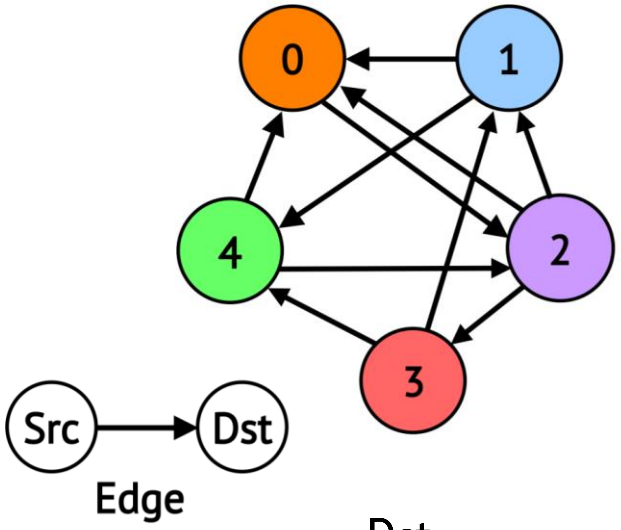
	D_0	D_1	D_2	D_3	D_4
S_0			1		
S_1	1				1
S_2	1	1		1	
S_3		1			1
S_4	1		1		

OA indexed by vertex ID of src of edge
Value in OA is *offset* into NA

start index for edges w/ src == vertex i = OA[i]
#edges with src == vertex i = OA[i+1] – OA[i]

Dense encoding of **sparse** structure

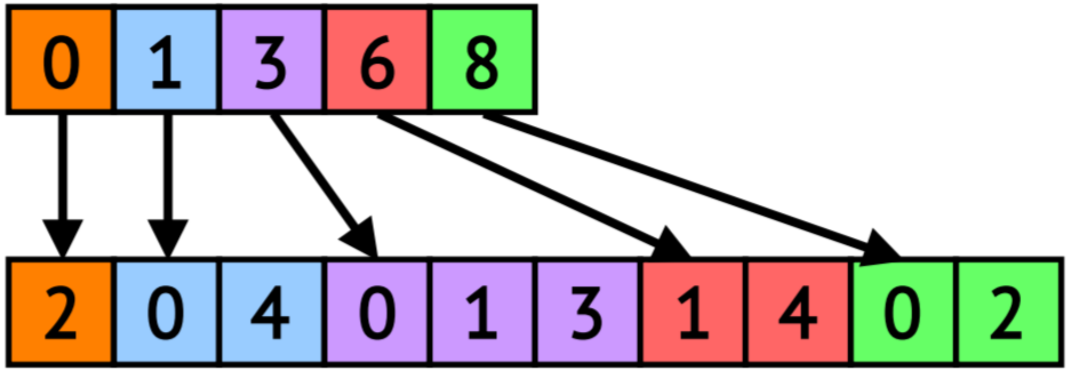
Compressed Sparse Data Structures for Feasible Memory Size



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)

0	1	3	6	8

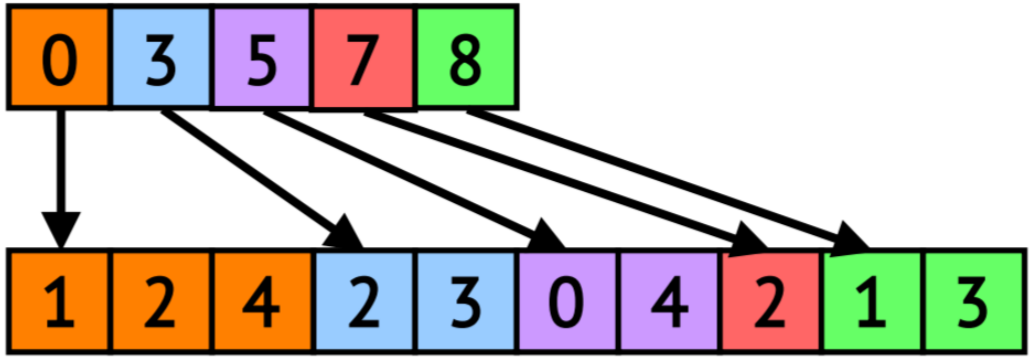
Neighbors Array (NA)

2	0	4	0	1	3	1	4	0	2

EdgeList sorted by SrcIDs

Compressed Sparse Row (CSR)
Outgoing Neighbors

The CSC is the *transpose* of the CSR



Offsets Array (OA)

0	3	5	7	8

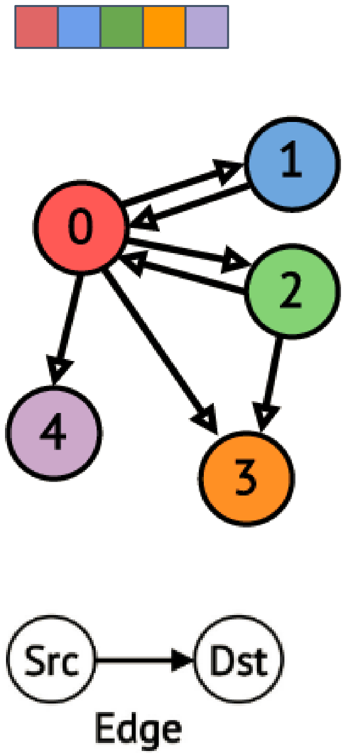
Neighbors Array (NA)

1	2	4	2	3	0	4	2	1	3

EdgeList sorted by DstIDs

Compressed Sparse Column (CSC)
Incoming Neighbors

Building the CSR / CSC from a Graph’s Edge List



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

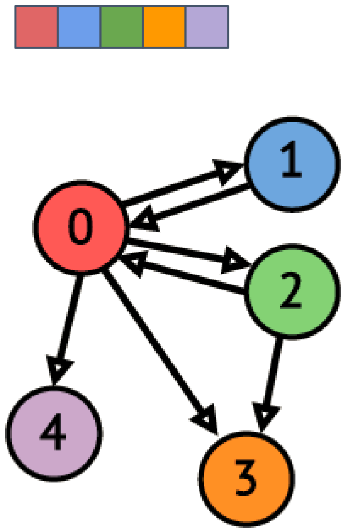
COO
(EdgeList)

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

2	1	1	2	1

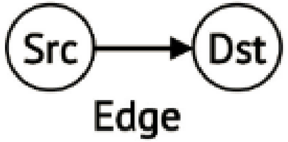
neigh_count

Building the CSR / CSC from a Graph's Edge List



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

COO
(EdgeList)

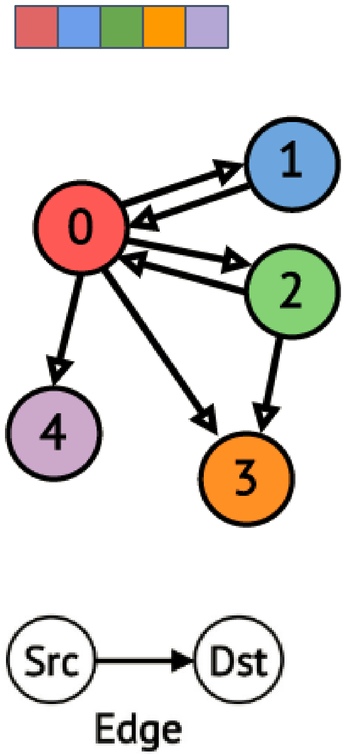


```
for e in EL:
  neigh_count[e.dst]++; /*e.src*/
```

2	1	1	2	1	**neigh_count**
2	1	1	2	1	**neigh_count_dup**

```
sum = 0
for i in 0 .. |V|:
  tmp = neigh_count[i]
  neigh_count[i] = sum;
  neigh_count_dup[i] = sum;
  sum += tmp
```

Building the CSR / CSC from a Graph's Edge List



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

COO
(EdgeList)

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

2	**1**	**1**	**2**	**1**

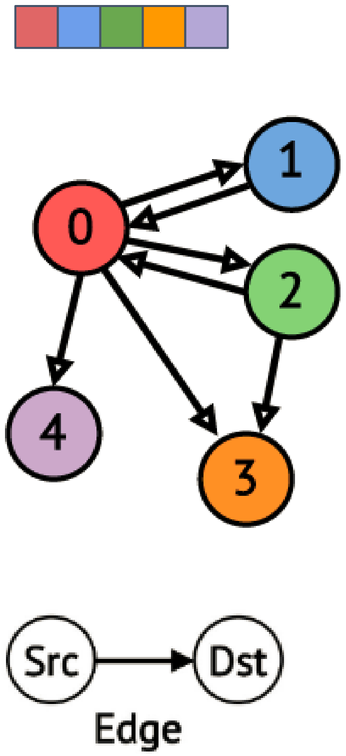
neigh_count

```
sum = 0
for i in 0 .. |V|:
  tmp = neigh_count[i]
  neigh_count[i] = sum; //OA
  neigh_count_dup[i] = sum;
  sum += tmp
```

0	**2**	**3**	**4**	**6**

OA (also OA_dup)

Building the CSR / CSC from a Graph’s Edge List



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

COO
(EdgeList)

0	2	3	4	6

OA (also OA_dup)

```
for e in EL:
  neigh_ind = OA[e.src]
  NA[neigh_ind] = e.dst
  OA[e.src]++ /*sacrificial OA*/
//i.e., NA[ OA[e.src]++ ] = e.dst
```

0	2	3	4	6

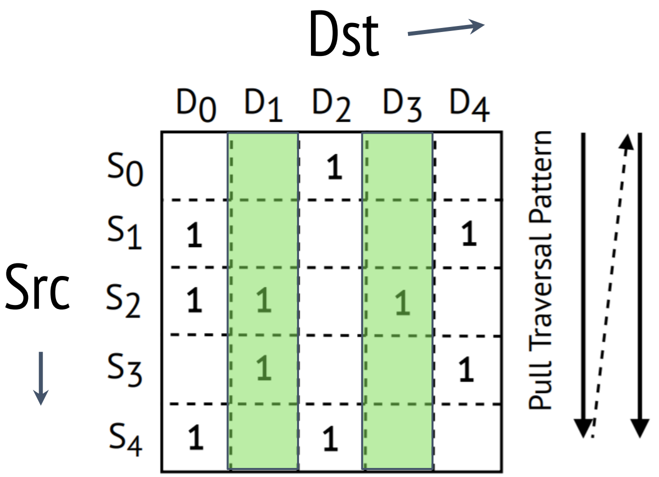
OA_dup

1	2	0	0	0	2	0

NA

Completed CSC

Compressed Representations ⇒ Irregular Memory Accesses

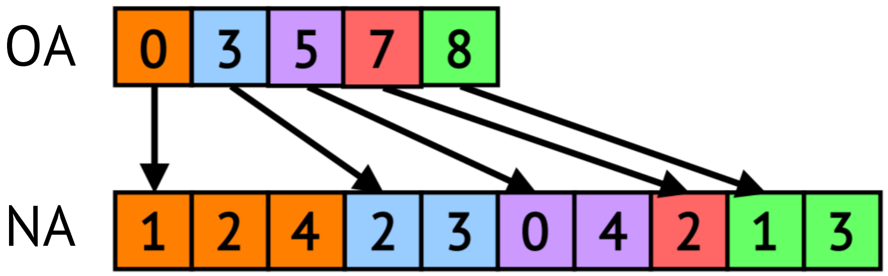


	D_0	D_1	D_2	D_3	D_4
S_0			1		
S_1	1				1
S_2	1	1		1	
S_3		1			1
S_4	1		1		

Pull (CSC Traversal)

```
for dst in G:
     for src in in_neighs(dst):
               dstData[dst] += srcData[src]
```

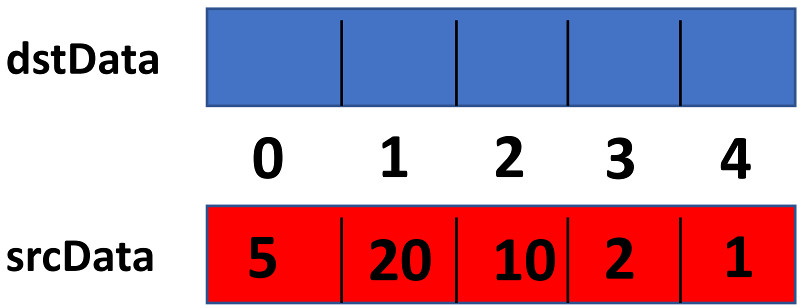
Pull traversal performs *irregular read operations* that lack locality



OA	0	3	5	7	8					
NA	1	2	4	2	3	0	4	2	1	3

CSC

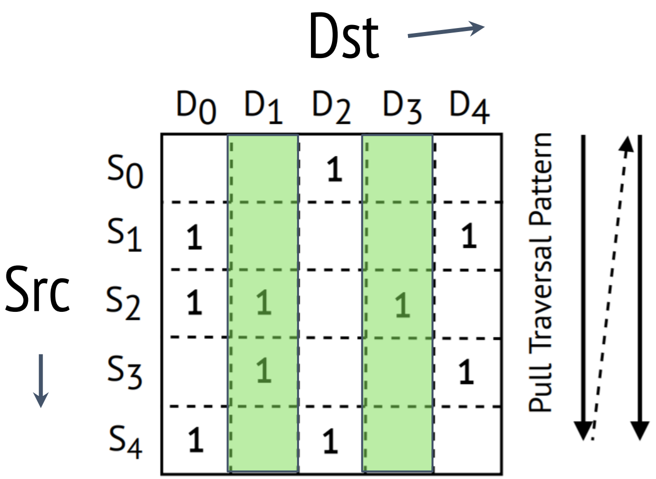
i.e., x_{i+1}



	0	1	2	3	4
dstData					
srcData	5	20	10	2	1

e.g., current rank of page I,
e.g., current shortest path from source vertex

Compressed Representations ⇒ Irregular Memory Accesses

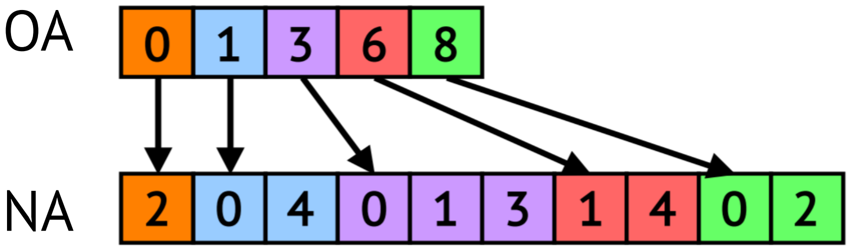


	D0	D1	D2	D3	D4
S0			1		
S1	1				1
S2	1	1		1	
S3		1			1
S4	1		1		

Push (CSR Traversal)

```
for src in G:
    for dst in out_neighs(src):
        dstData[dst] += srcData[src]
```

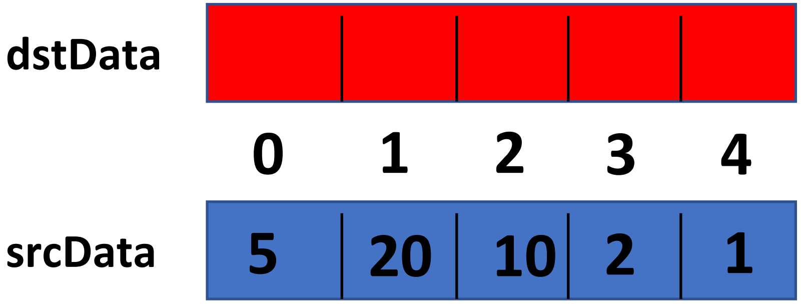
Push traversal performs *irregular write operations* that lack locality



OA	0	1	3	6	8					
NA	2	0	4	0	1	3	1	4	0	2

CSR

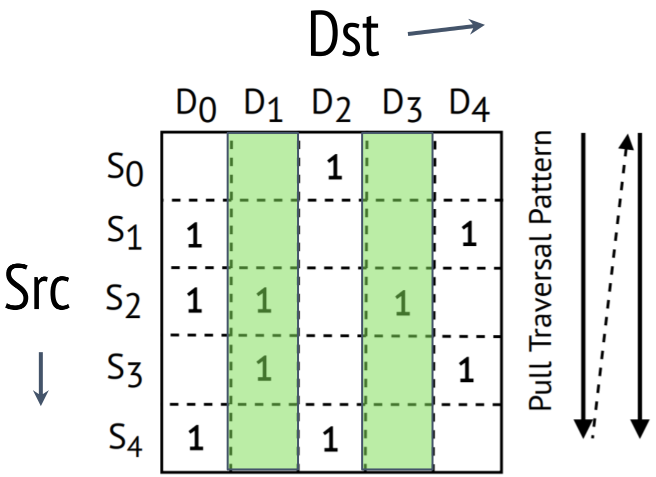
i.e., x_{i+1}



	0	1	2	3	4
dstData					
srcData	5	20	10	2	1

e.g., current rank of page I,
e.g., current shortest path from source vertex

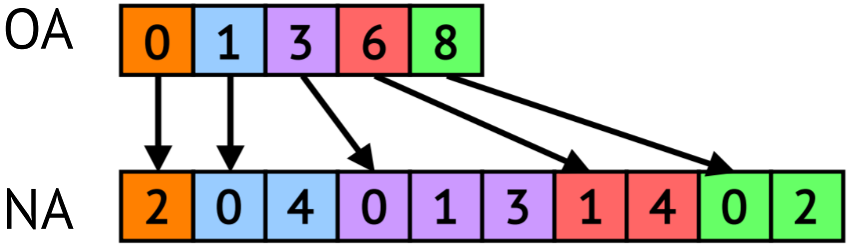
Compressed Representations ⇒ Irregular Memory Accesses



Push (CSR Traversal)

```
for src in G:
     for dst in out_neighs(src):
               dstData[dst] += srcData[src]
```

Push traversal performs *irregular write operations* that lack locality



CSR

i.e., x_{i+1}

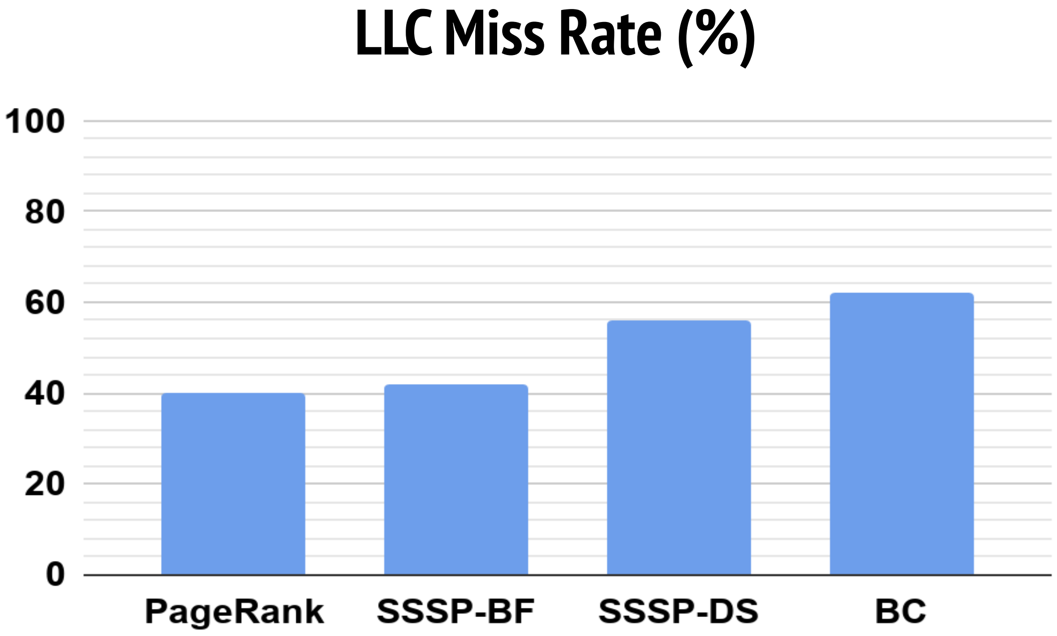
Irregular Data Footprint >> LLC Size

Size of srcData ~ **256MB** (32M vertices * 8B)

e.g., current rank of page i,

e.g., current shortest path from source vertex

Irregular Accesses Lead to Poor Locality

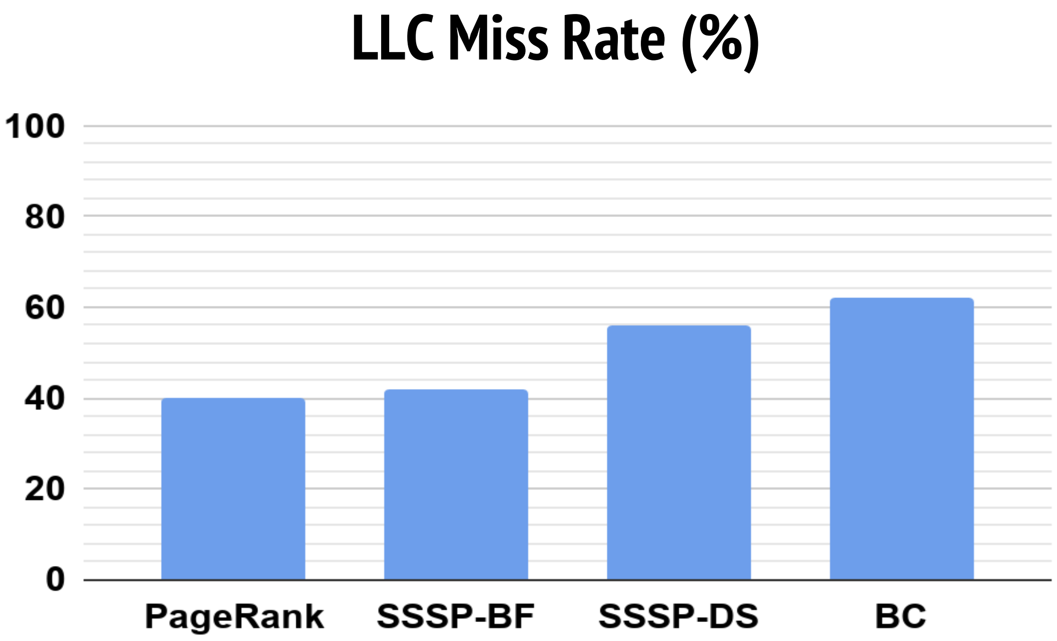


Running on RMAT27
Graph w/ 35MB LLC

Why such bleak cache performance?
Consequence of bleak cache performance?

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

Irregular Accesses Lead to Poor Locality

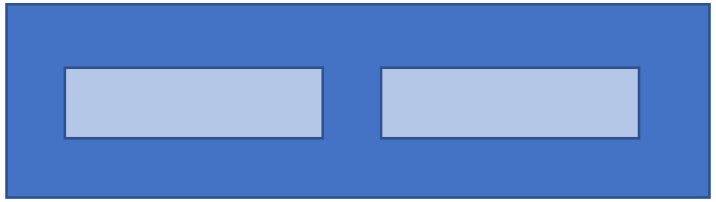


Running on RMAT27
Graph w/ 35MB LLC

Dst coordinate of edge is index in dstData: totally input dependent & random!!!

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

COO
(EdgeList)



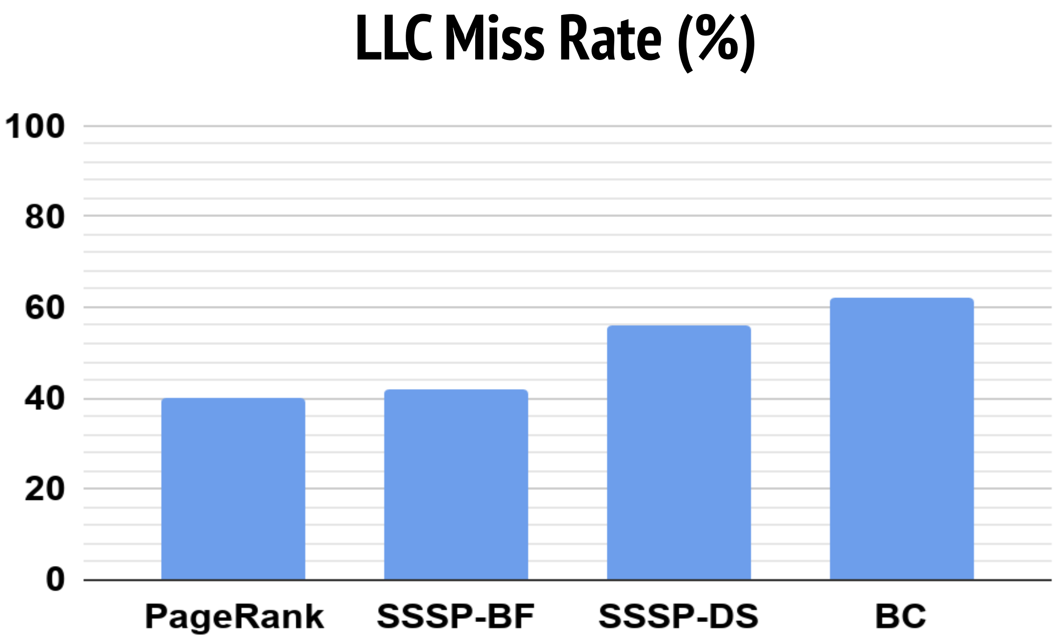
0	0	0	0	0

dstData

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

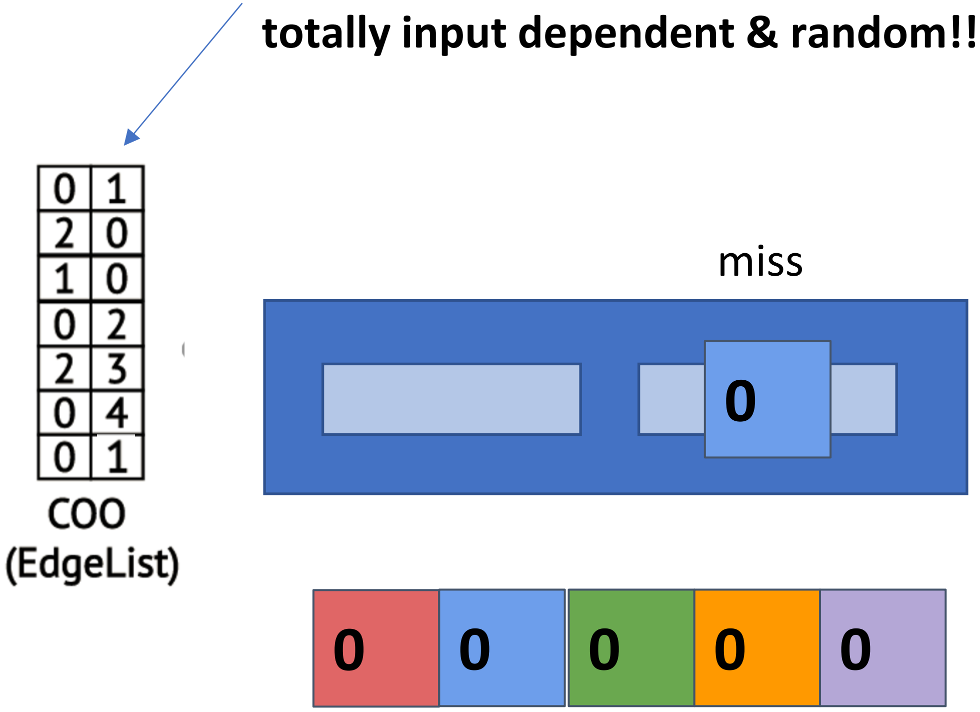
Right Figure from "Optimizing Cache Performance for Graph Analytics" ArXiv v1;

Irregular Accesses Lead to Poor Locality



Running on RMAT27 Graph w/ 35MB LLC

Dst coordinate of edge is index in dstData: totally input dependent & random!!!



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

dstData

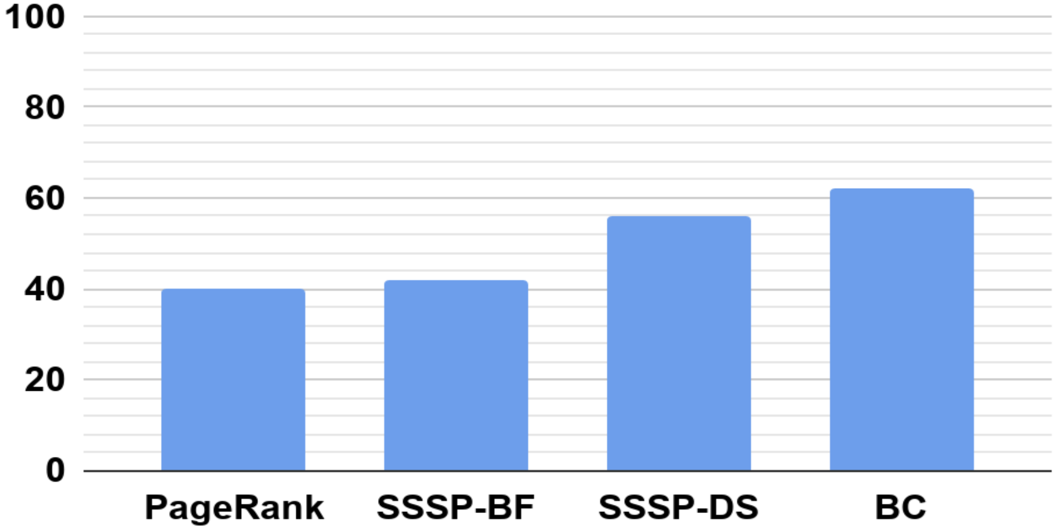
Remember: dstData[e.dst] ++

and e.dst is random, from edge list

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

Irregular Accesses Lead to Poor Locality

LLC Miss Rate (%)

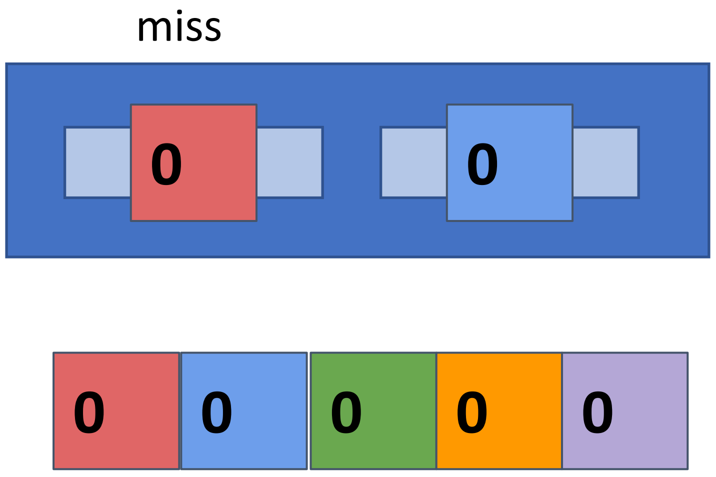


Running on RMAT27
Graph w/ 35MB LLC

Dst coordinate of edge is index in dstData: totally input dependent & random!!!

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

COO
(EdgeList)

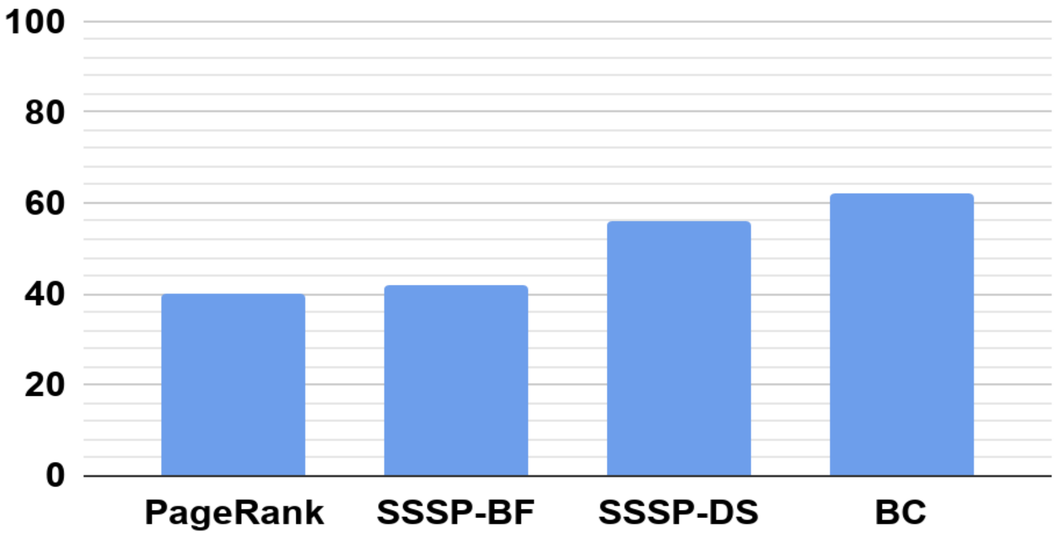


dstData

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

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

Irregular Accesses Lead to Poor Locality



LLC Miss Rate (%)

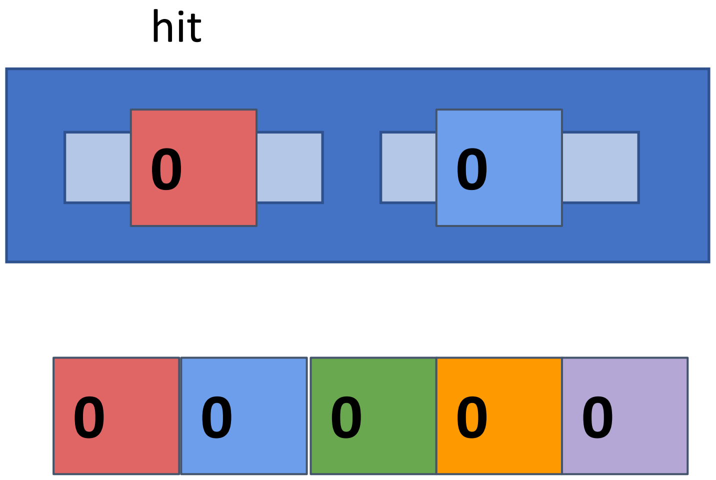
Running on RMAT27 Graph w/ 35MB LLC

Dst coordinate of edge is index in dstData: totally input dependent & random!!!

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

COO (EdgeList)

(You get lucky sometimes)



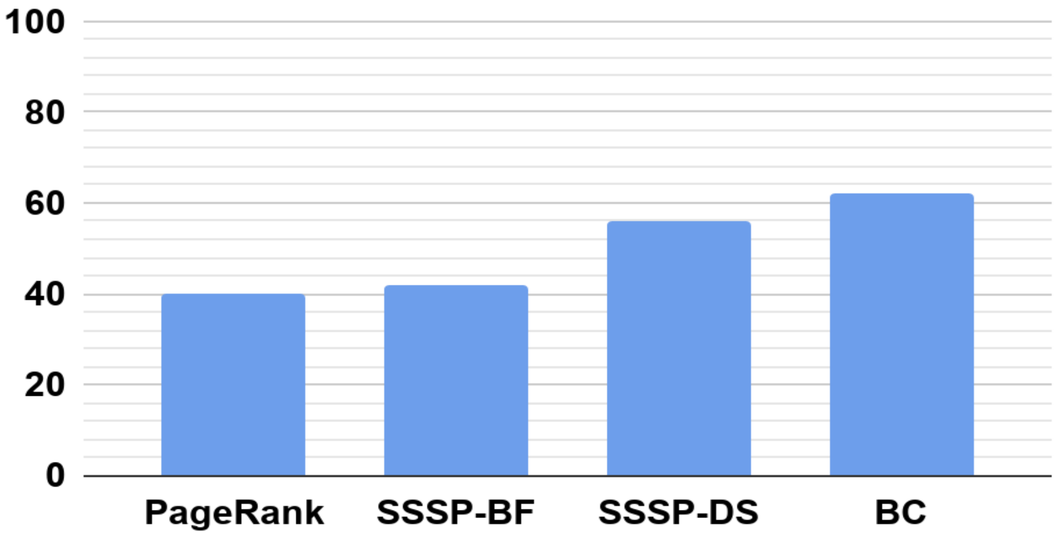
dstData

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

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

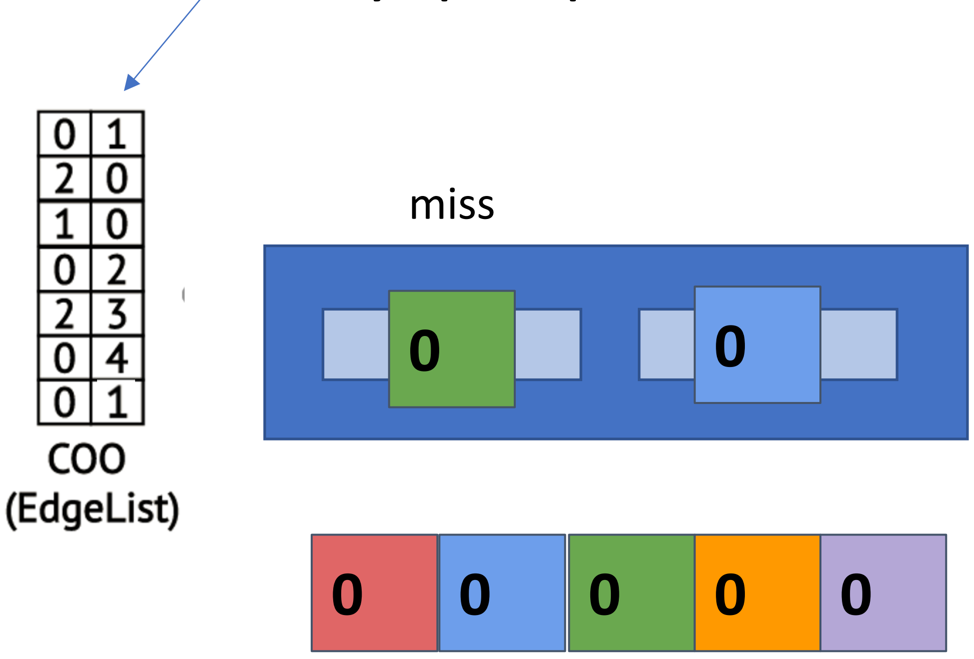
Irregular Accesses Lead to Poor Locality

LLC Miss Rate (%)



Running on RMAT27 Graph w/ 35MB LLC

Dst coordinate of edge is index in dstData: totally input dependent & random!!!



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

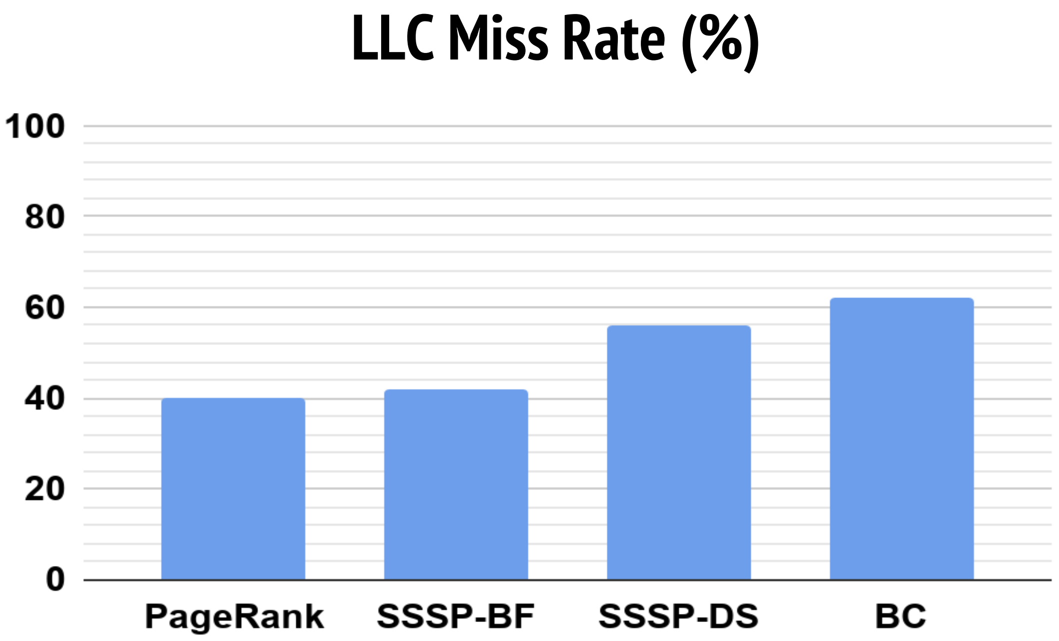
COO (EdgeList)

dstData

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

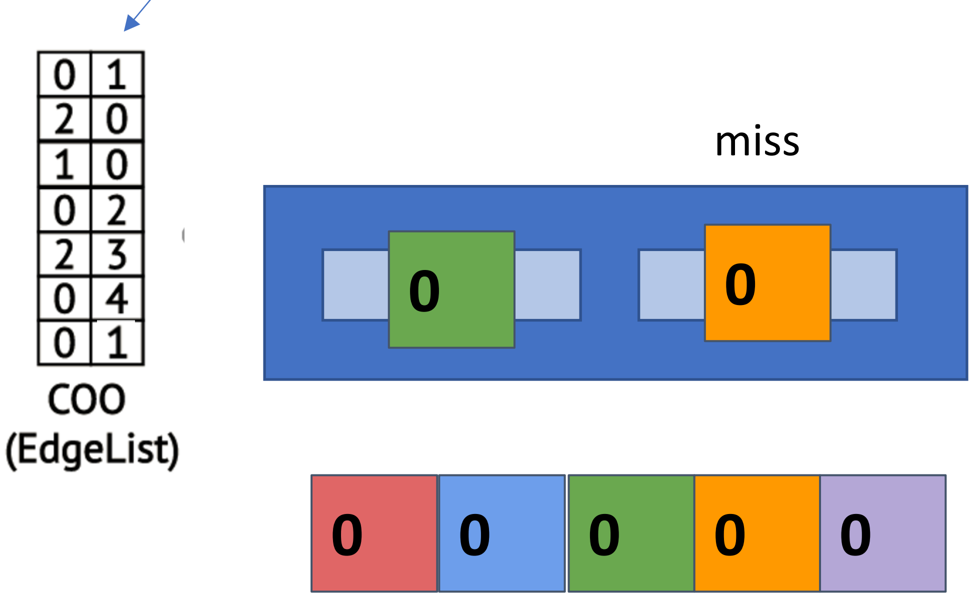
Right Figure from “Optimizing Cache Performance for Graph Analytics” ArXiv v1;

Irregular Accesses Lead to Poor Locality



Running on RMAT27
Graph w/ 35MB LLC

Dst coordinate of edge is index in dstData: totally input dependent & random!!!



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

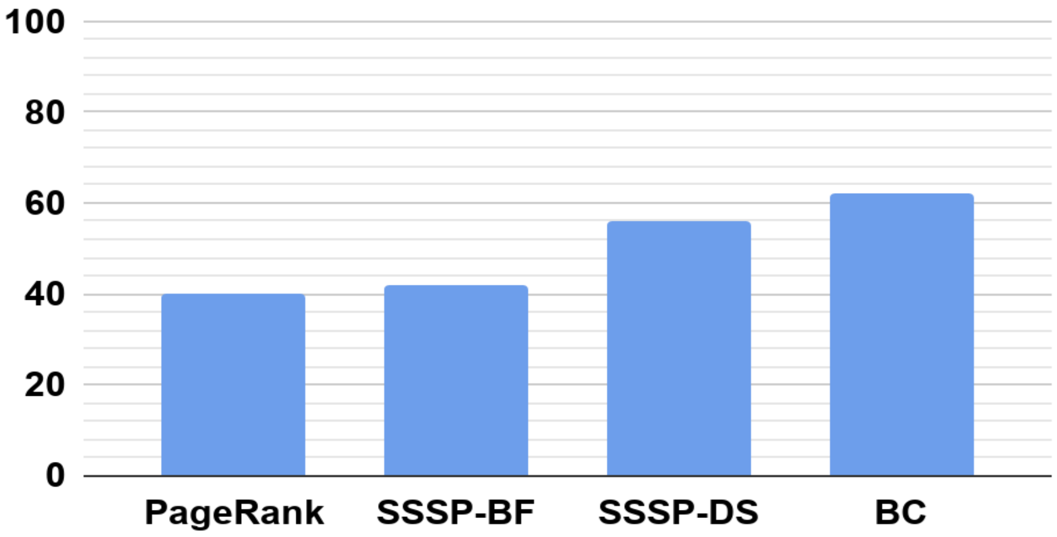
dstData

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

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

Irregular Accesses Lead to Poor Locality

LLC Miss Rate (%)

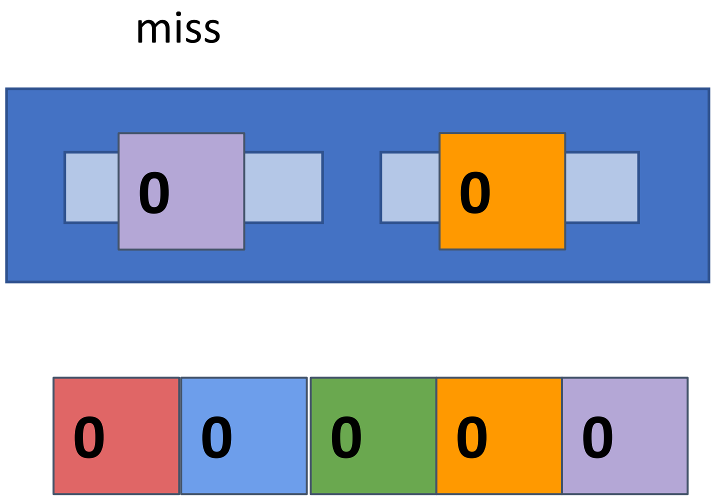


Running on RMAT27
Graph w/ 35MB LLC

Dst coordinate of edge is index in dstData:
totally input dependent & random!!!

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

COO
(EdgeList)

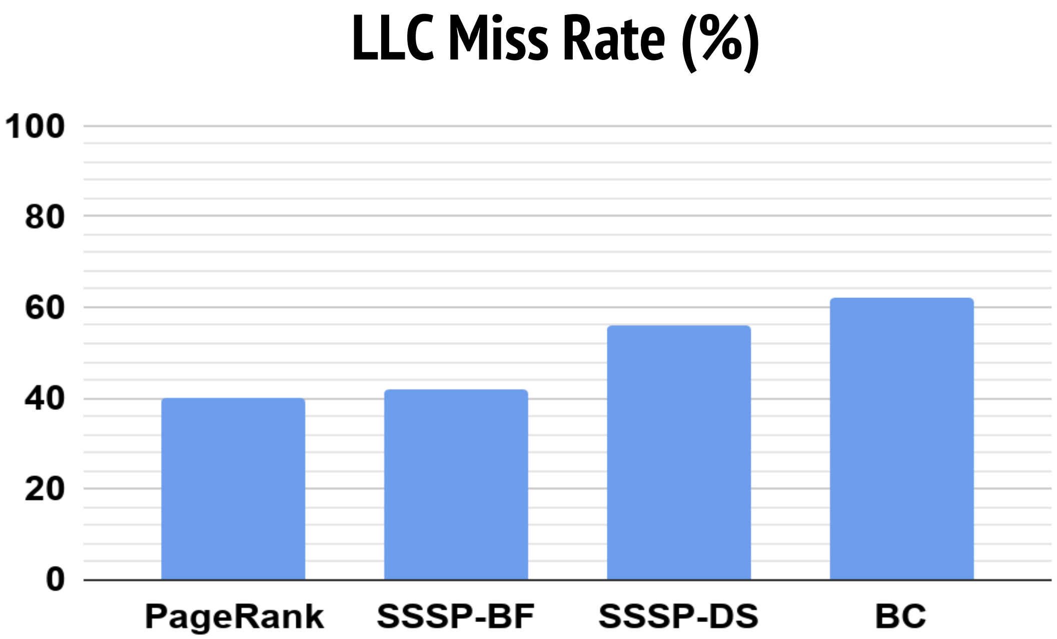


dstData

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

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

Irregular Accesses Lead to Poor Locality

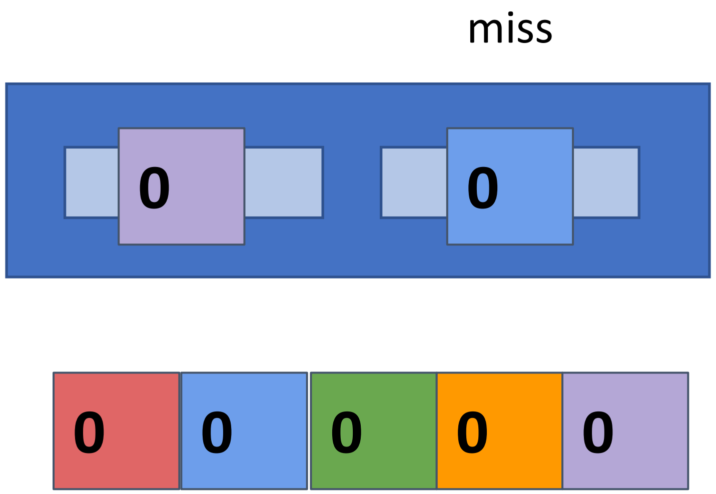


Running on RMAT27
Graph w/ 35MB LLC

Dst coordinate of edge is index in dstData:
totally input dependent & random!!!

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

COO
(EdgeList)



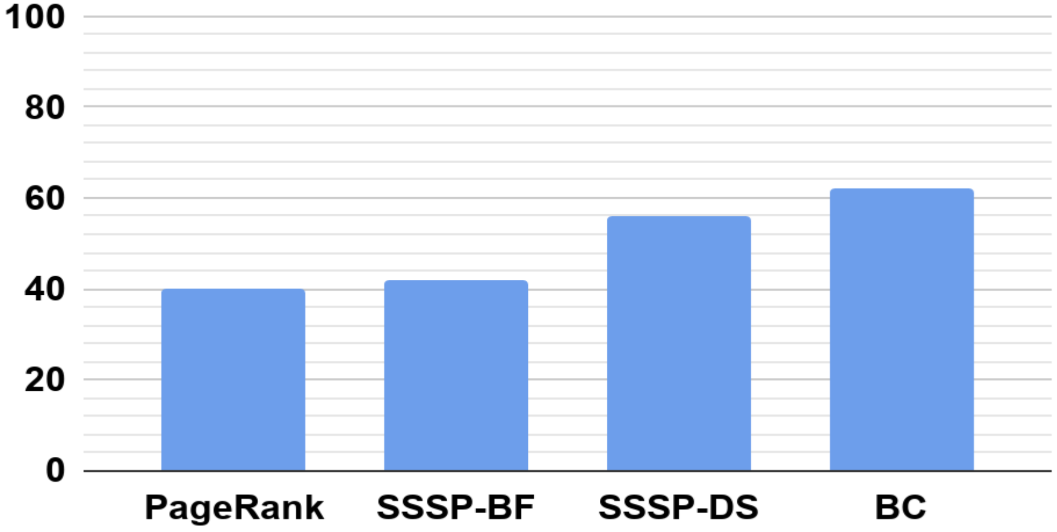
dstData

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

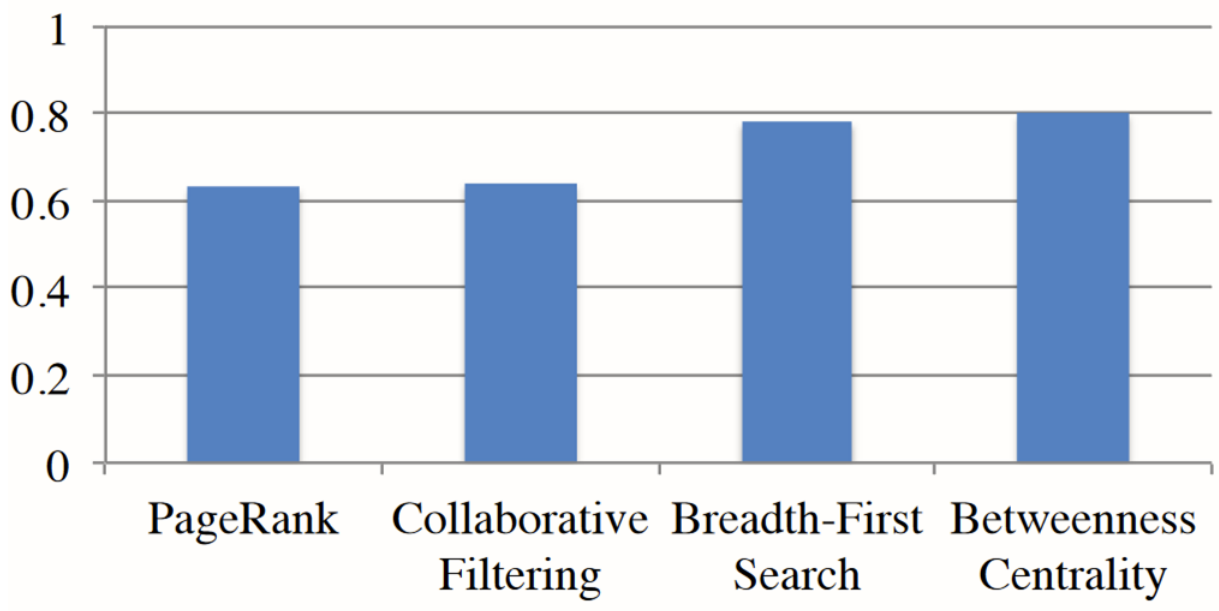
Right Figure from "Optimizing Cache Performance for Graph Analytics" ArXiv v1;

Irregular Accesses Lead to Poor Locality

LLC Miss Rate (%)



Cycles stalled on DRAM / Total Cycles

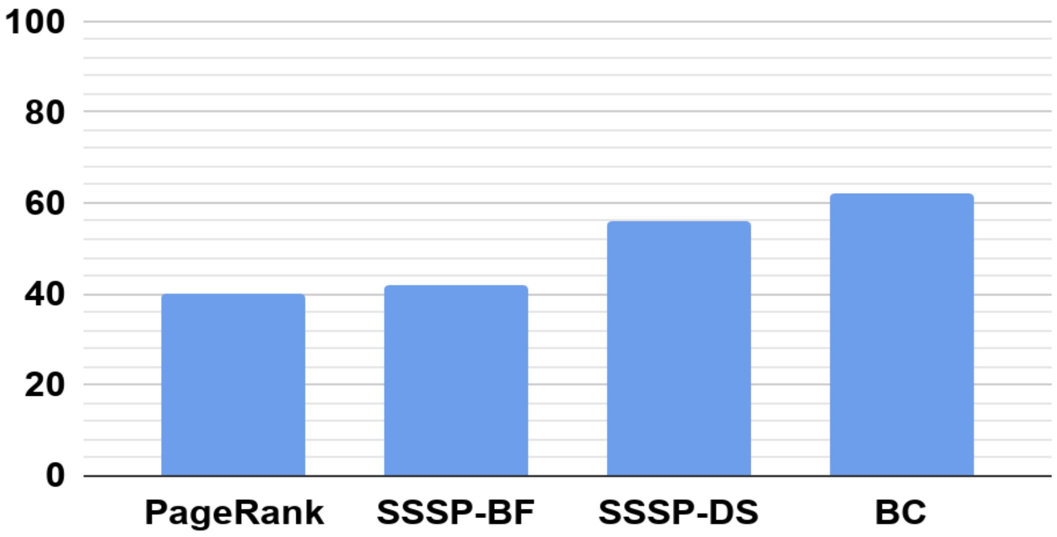


Cache miss latency *cannot be hidden by anything else in the program*. Each miss incurs DRAM latency!

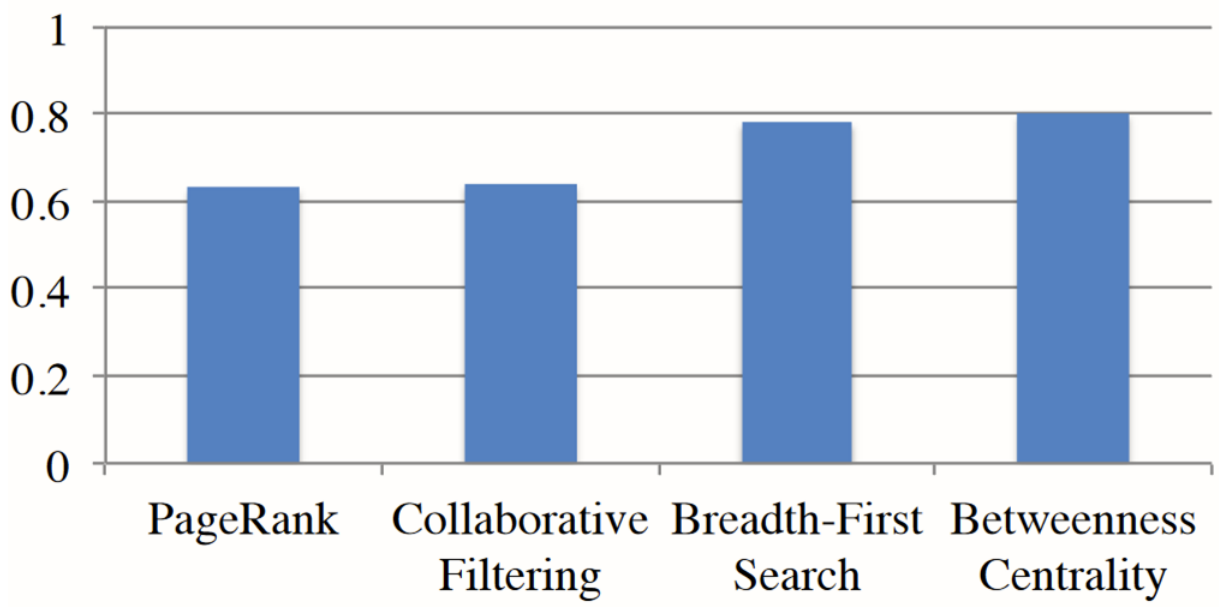
Right Figure from "Optimizing Cache Performance for Graph Analytics" ArXiv v1;

Irregular Accesses Lead to Poor Locality

LLC Miss Rate (%)



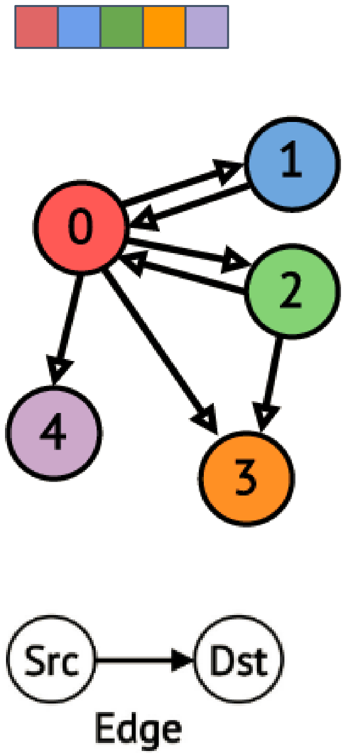
Cycles stalled on DRAM / Total Cycles



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

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

Even Building the CSR / CSC is an Irregular Access Pattern!



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

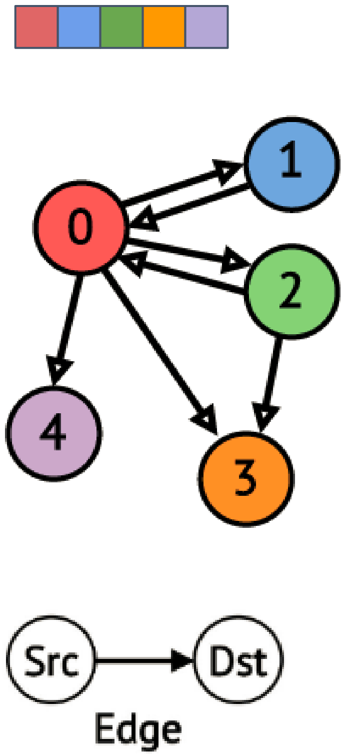
COO
(EdgeList)

```
for e in EL:
  neigh_count[e.dst]++;
```

2	1	1	2	1	**neigh_count**

Why is this irregular?

Even Building the CSR / CSC is an Irregular Access Pattern!



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

COO
(EdgeList)

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

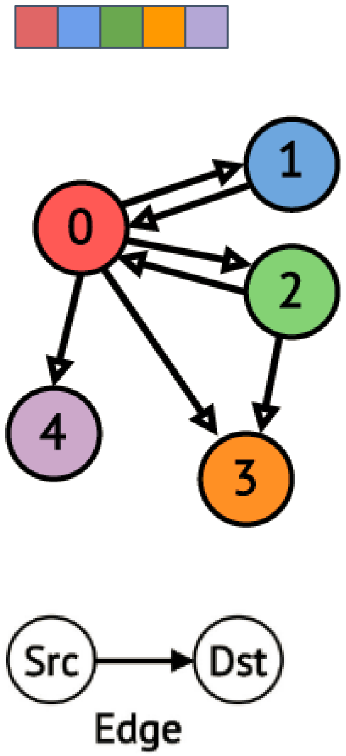
2	1	1	2	1

neigh_count

Updates to the neigh_count array are to random elements determined by order of edges in edge list

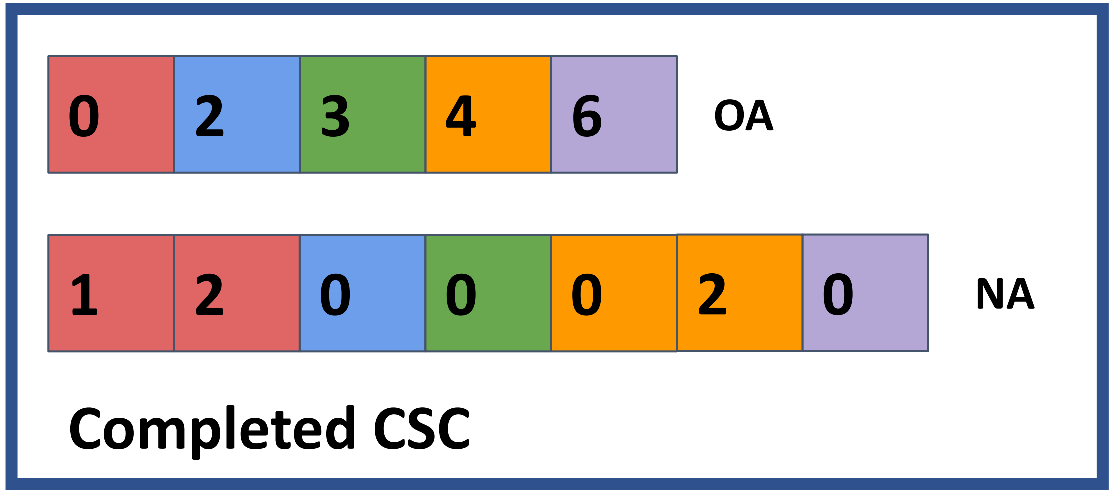
Even Building the CSR / CSC is an Irregular Access Pattern!

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



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

COO
(EdgeList)

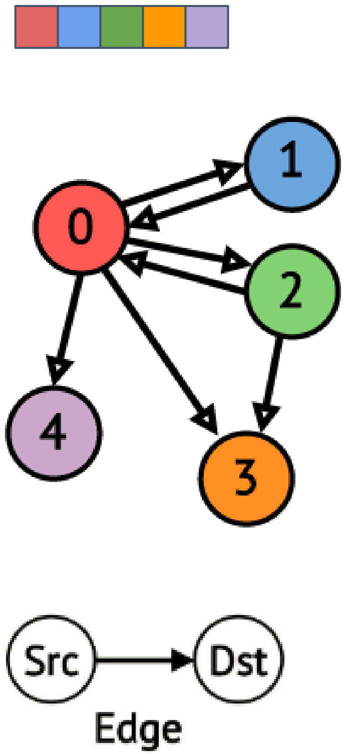


0	2	3	4	6			OA
1	2	0	0	0	2	0	NA

Completed CSC

Why is the NA update part irregular?

Even Building the CSR / CSC is an Irregular Access Pattern!



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

COO
(EdgeList)

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

0	2	3	4	6			OA
1	2	0	0	0	2	0	NA

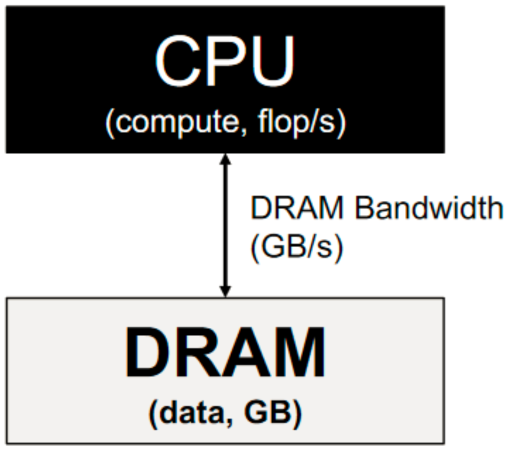
Completed CSC

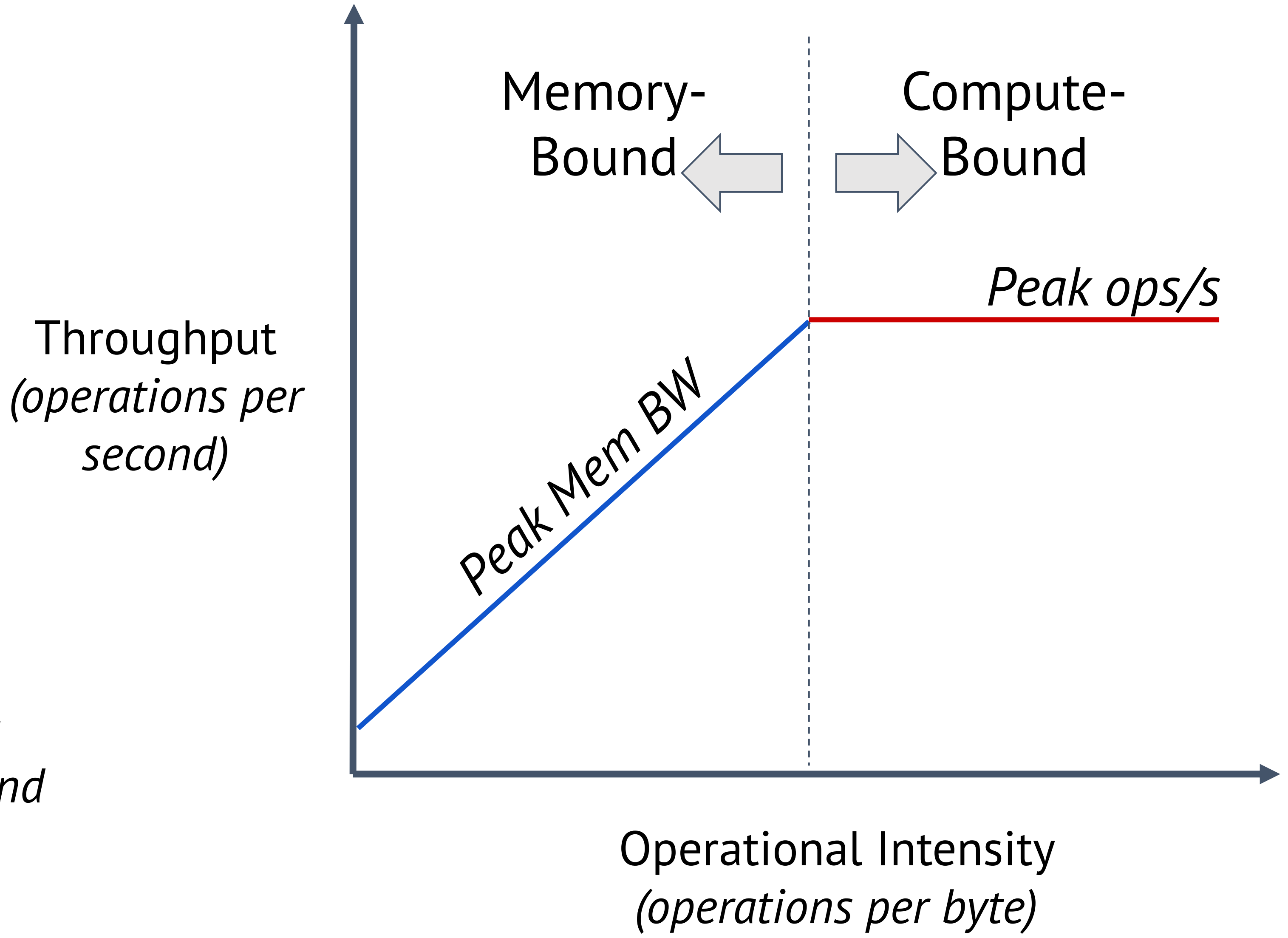
Updates to NA based on EL order & OA[e.src]

NA[OA[e.src]++] = e.dst

Roofline Performance Analysis of Graph Applications

The Roofline Model

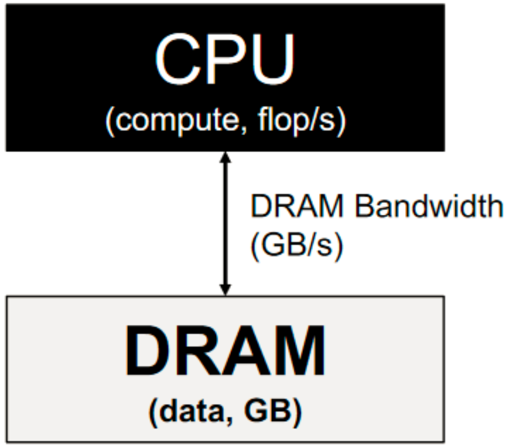




GFLOPS = Giga-Floating Point Operations Per Second

Yes, this is not a proper acronym

The Roofline Model

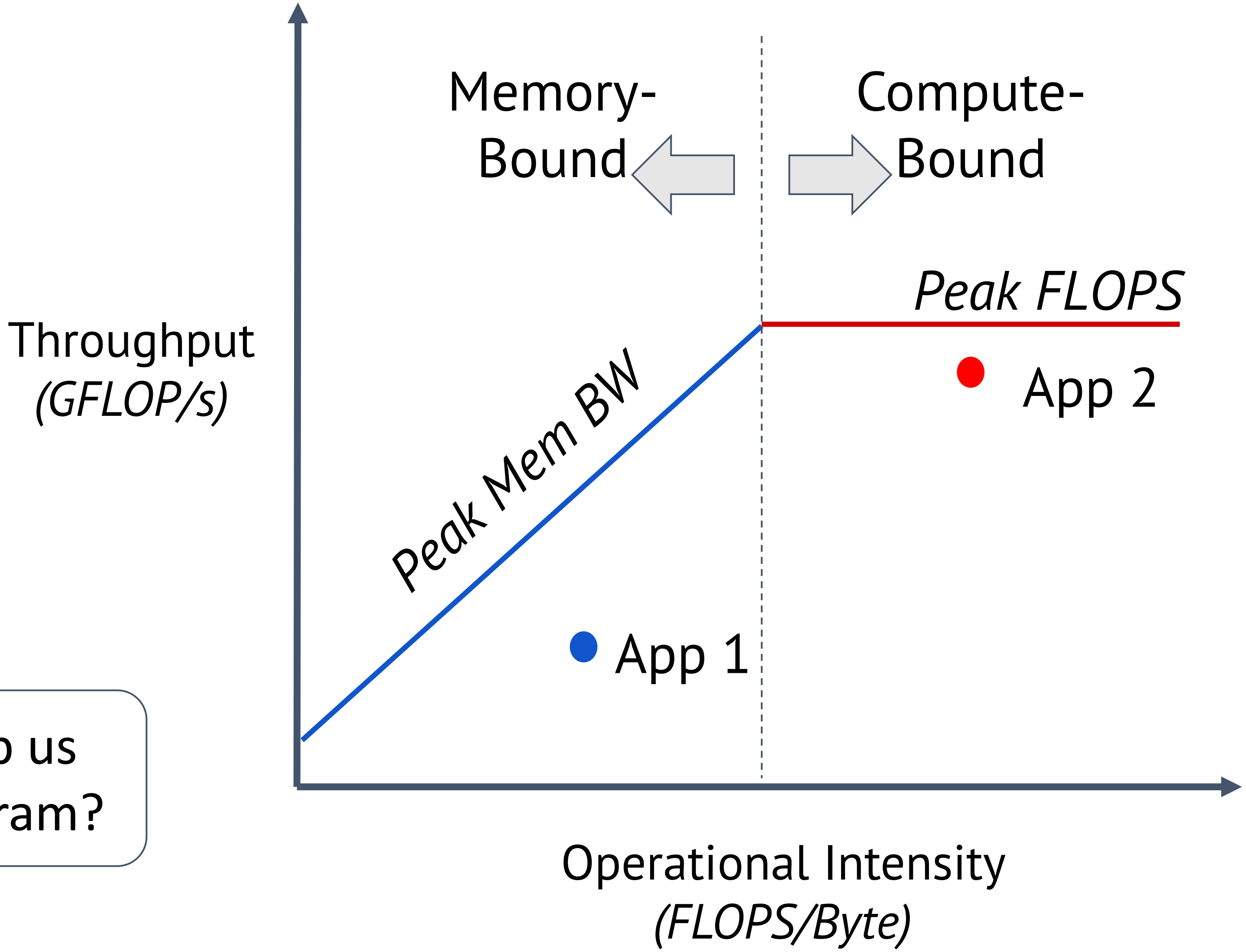

CPU

(compute, flop/s)

DRAM Bandwidth (GB/s)

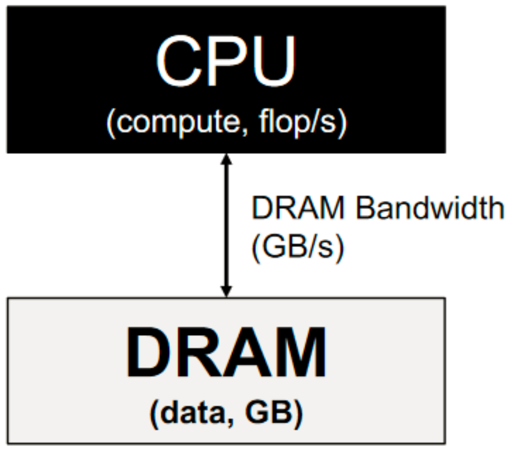
DRAM

(data, GB)



What does Roofline help us understand about a program?

The Roofline Model



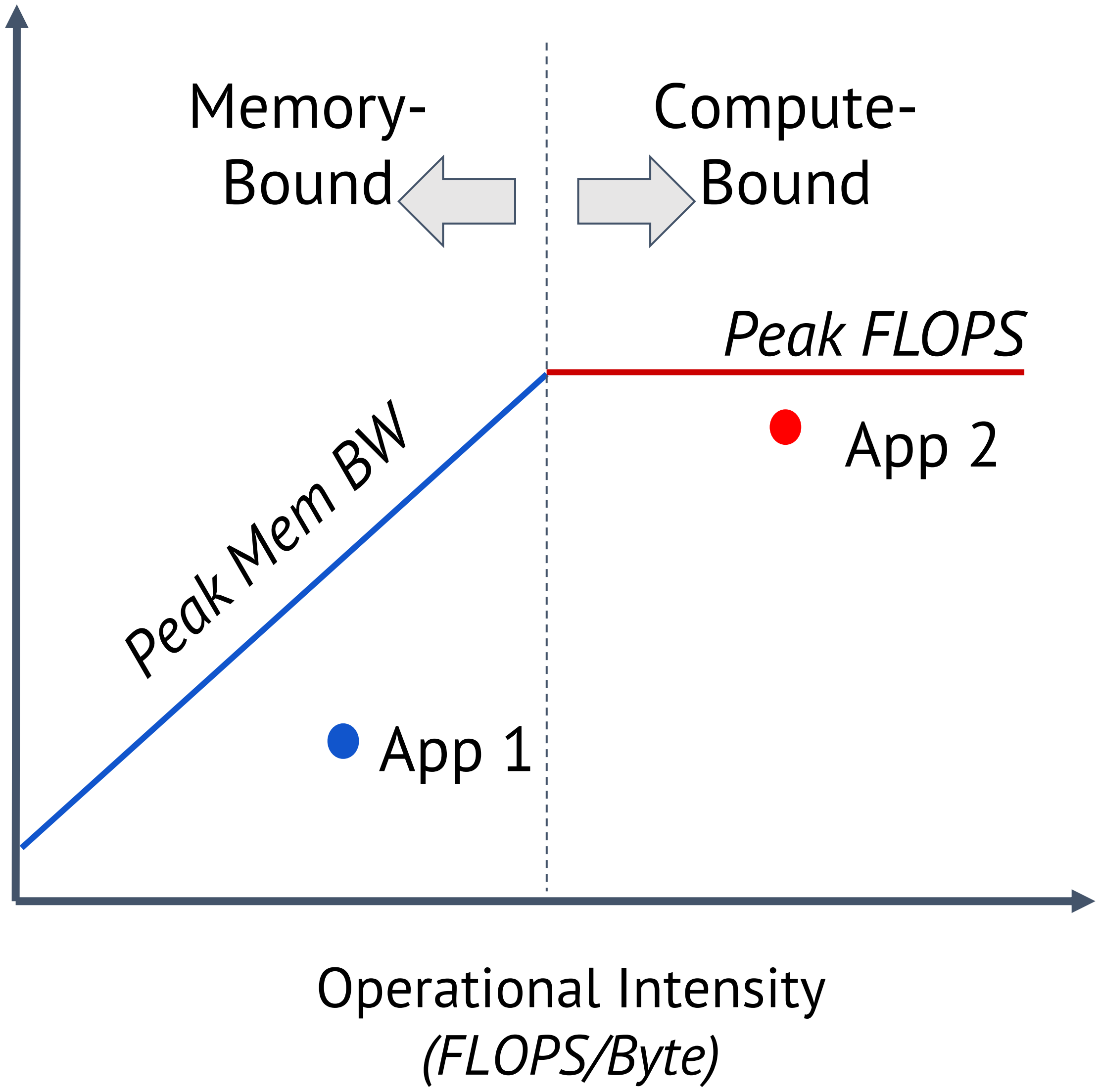
CPU

(compute, flop/s)

DRAM Bandwidth (GB/s)

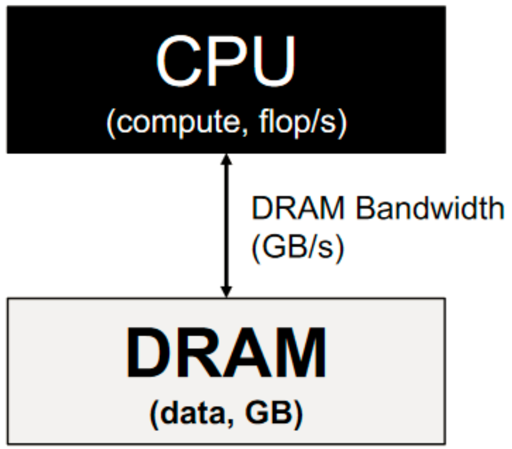
DRAM

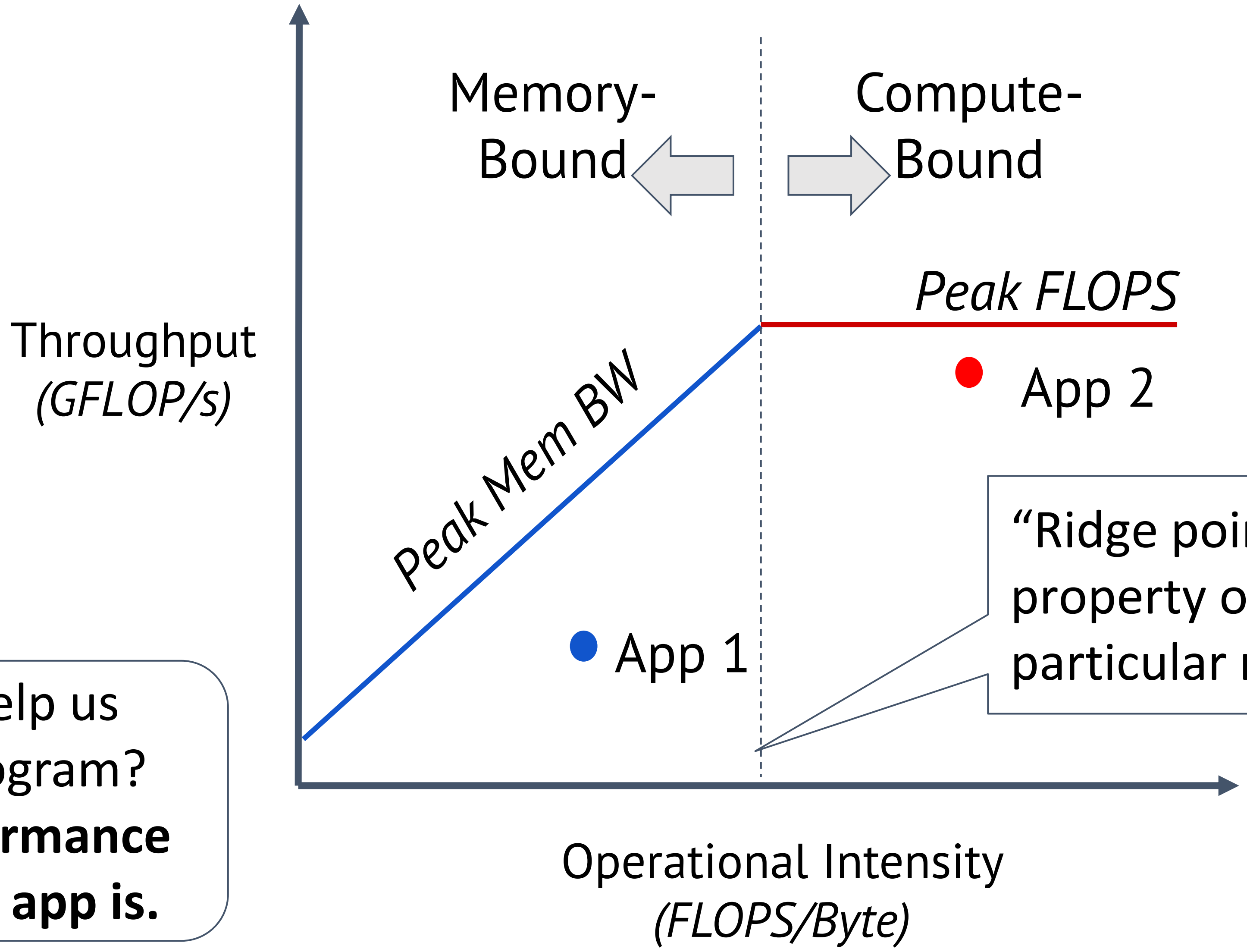
(data, GB)



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

The Roofline Model

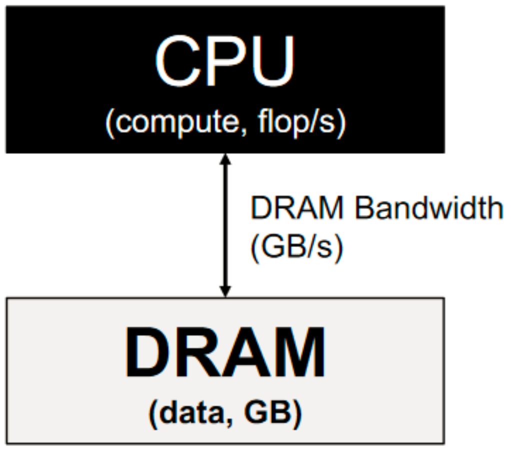


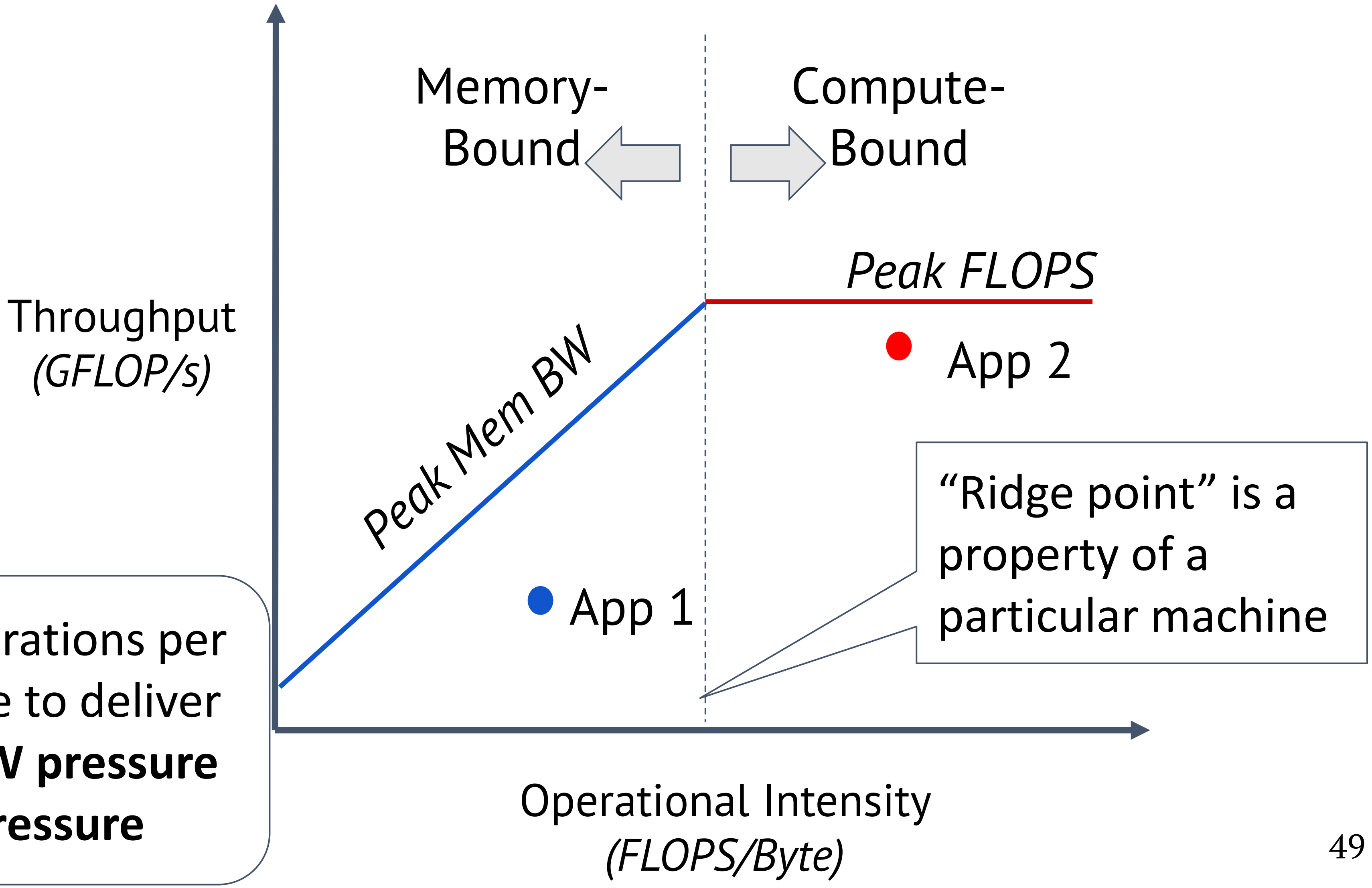


"Ridge point" is a property of a particular machine

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

The Roofline Model

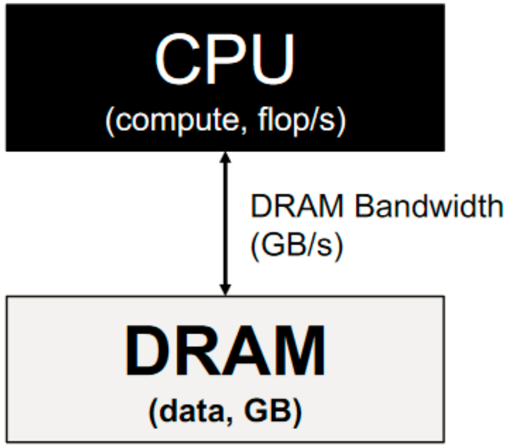


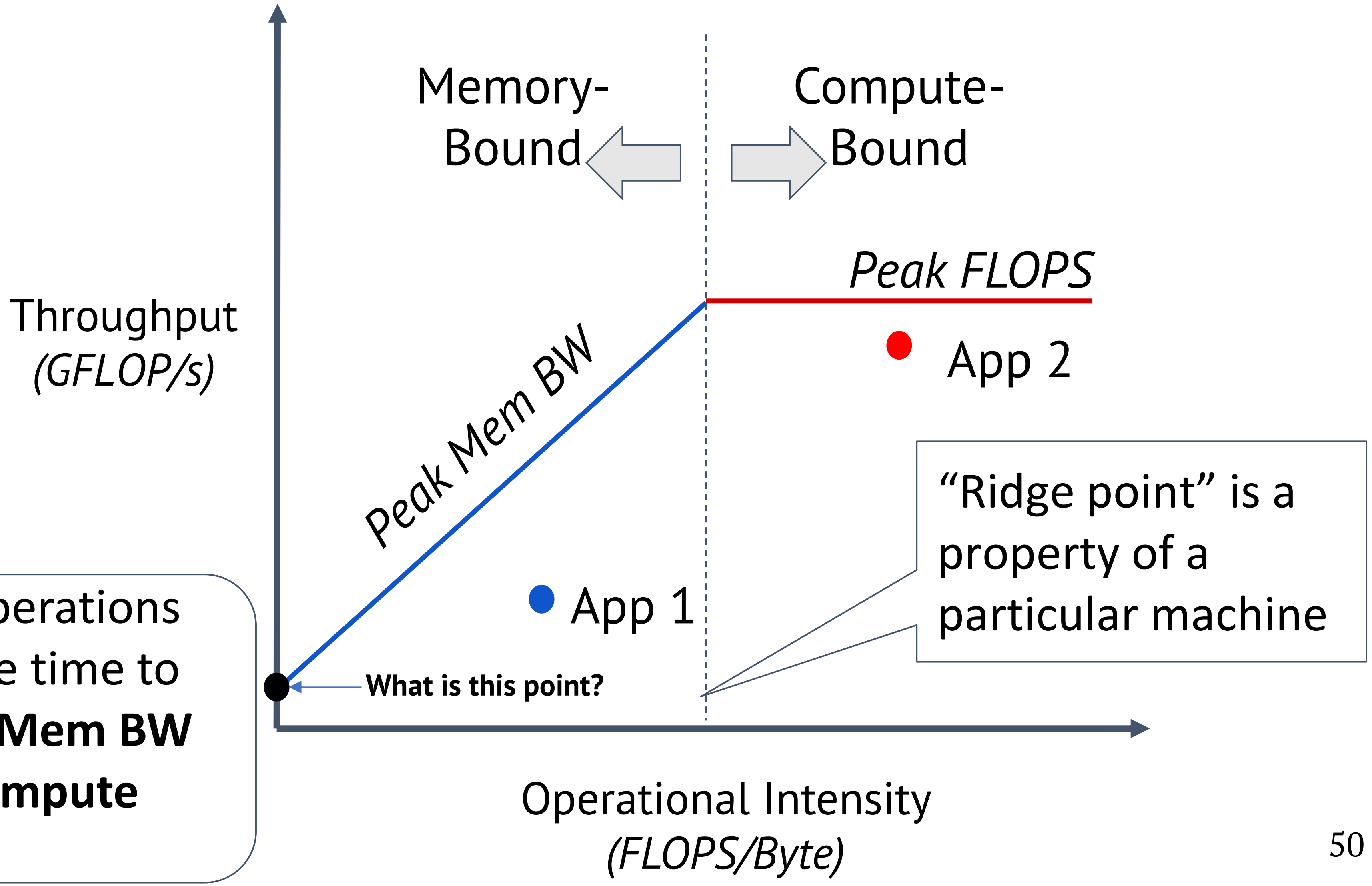


“Ridge point” is a property of a particular machine

As a program does more operations per byte, memory has more time to deliver next byte**, relieving Mem BW pressure & increasing compute pressure**

The Roofline Model

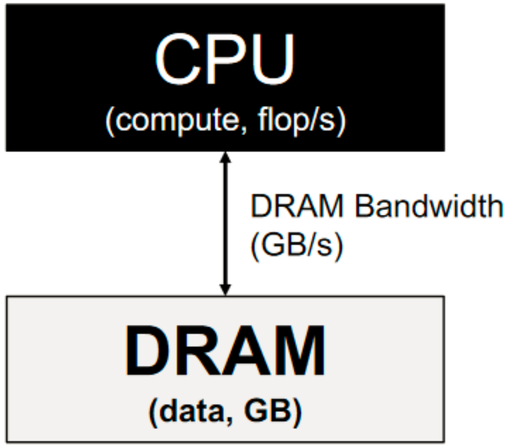




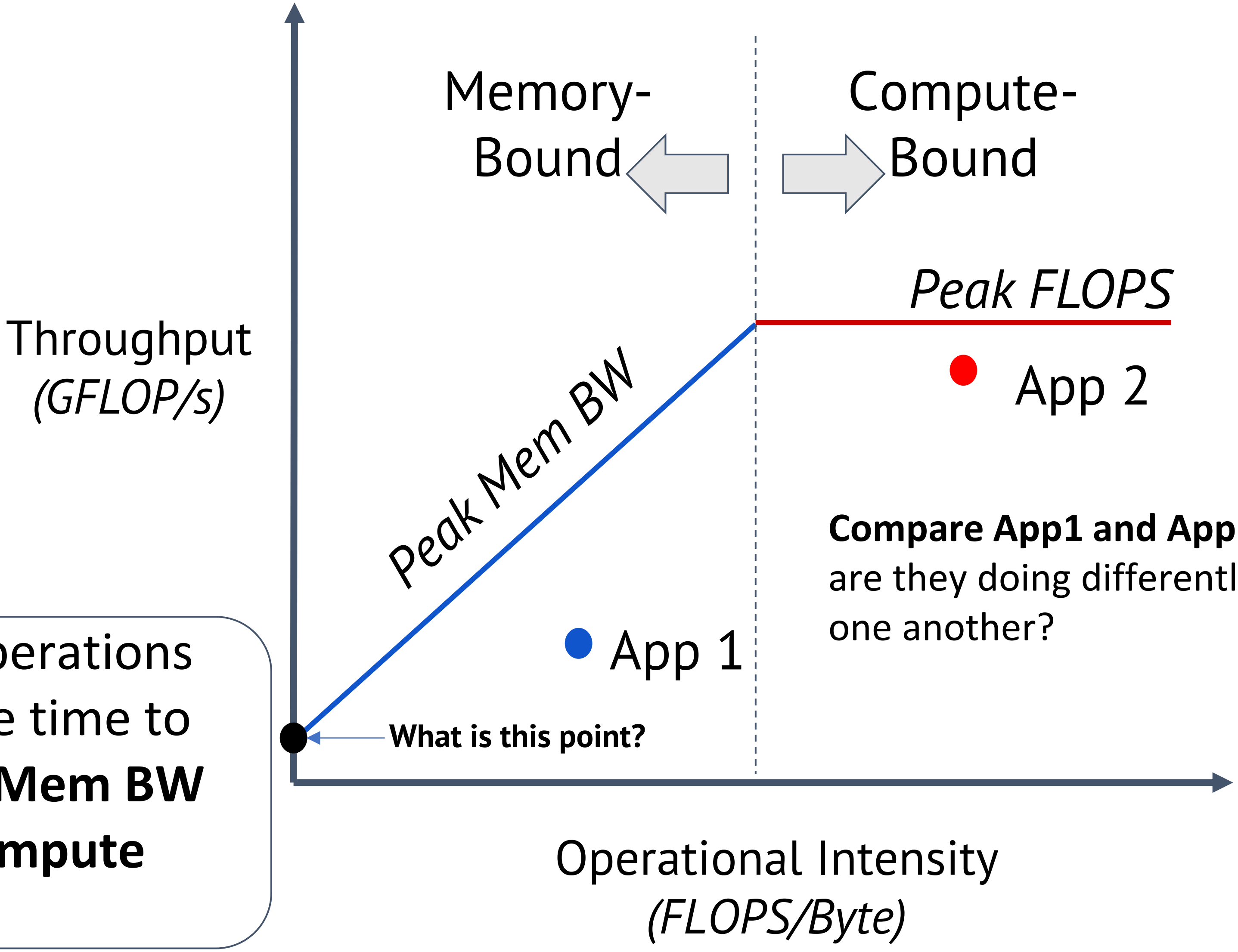
“Ridge point” is a property of a particular machine

As a program does more operations per byte, memory has more time to deliver next byte**, relieving Mem BW pressure & increasing compute pressure**

The Roofline Model

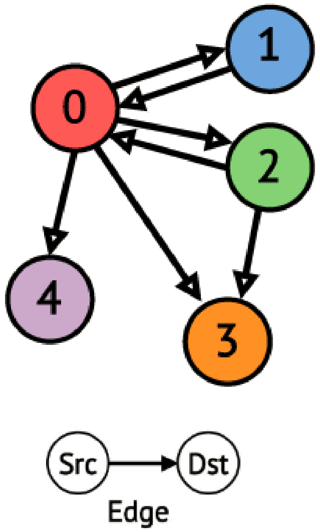


As a program does more operations per byte, memory has more time to deliver next byte**, relieving Mem BW pressure & increasing compute pressure**



Compare App1 and App2. What are they doing differently from one another?

Operational Intensity of Irregular Graph Applications



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

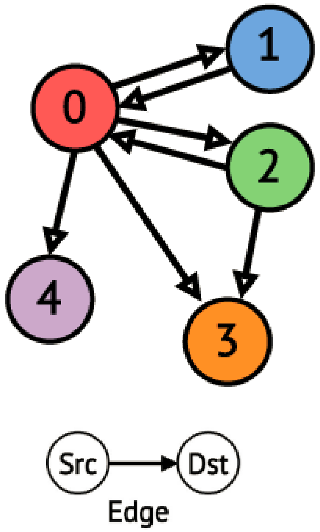
COO
(EdgeList)

🟥🟦🟩🟧🟪

```
for e in EL:
  dstData[e.dst] += srcData[e.src]
```

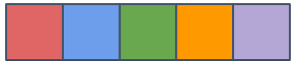
What is the operational intensity of a random update kernel like this one?

Operational Intensity of Irregular Graph Applications



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

COO
(EdgeList)

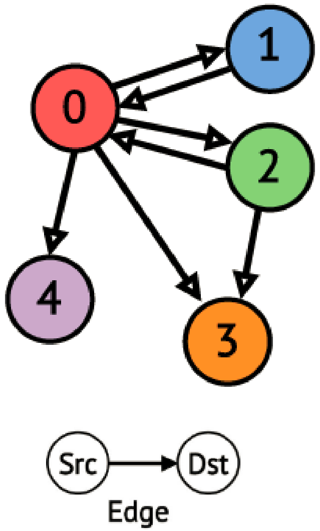


```
for e in EL:
  dstData[e.dst] += srcData[e.src]
```

What is the operational intensity of a random update kernel like this one?

Operations per byte:

Operational Intensity of Irregular Graph Applications



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

COO
(EdgeList)

🟥🟦🟩🟧🟪

```
for e in EL:
  dstData[e.dst] += srcData[e.src]
```

What is the operational intensity of a random update kernel like this one?

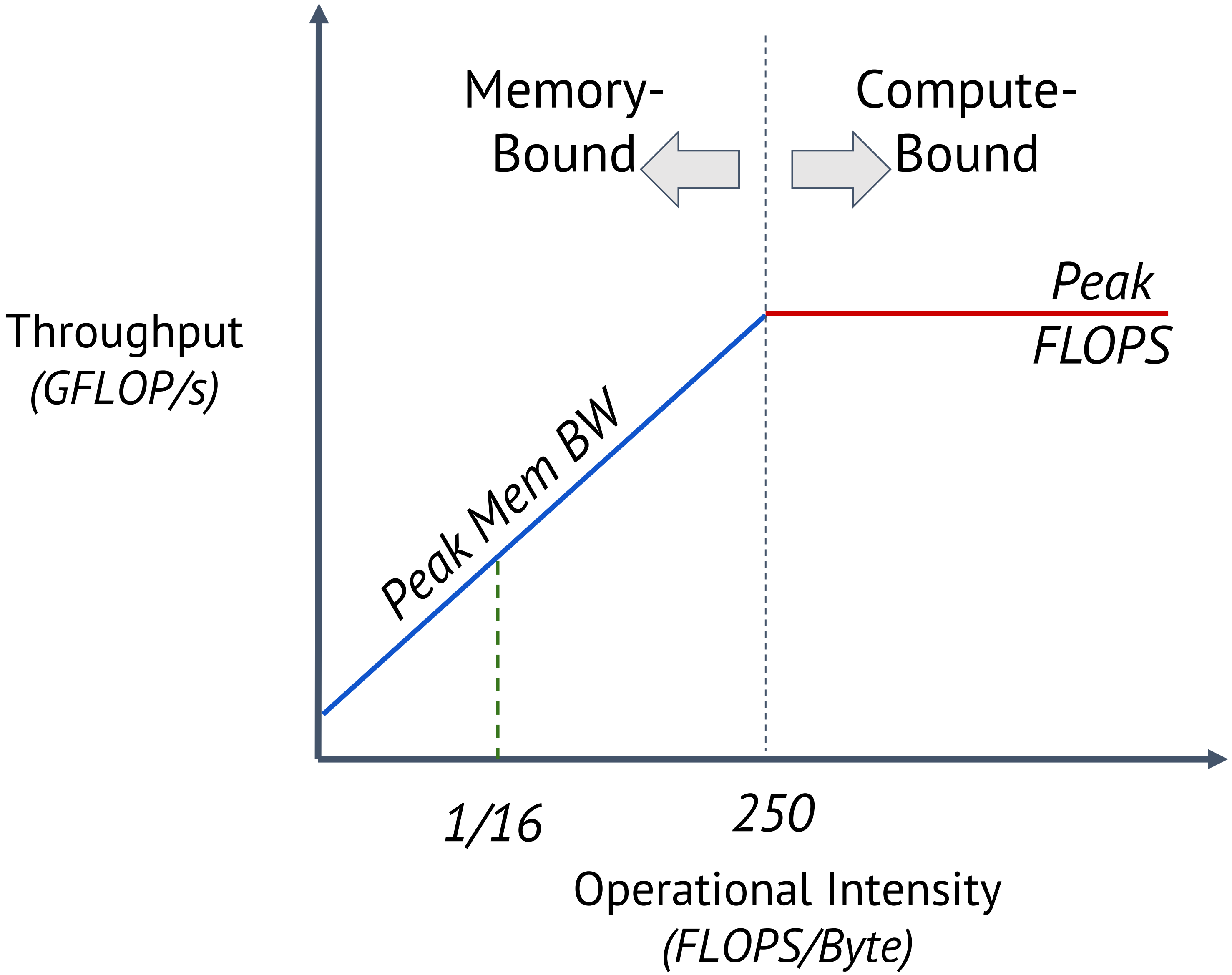
Operations per byte:

Operations: 1 addition

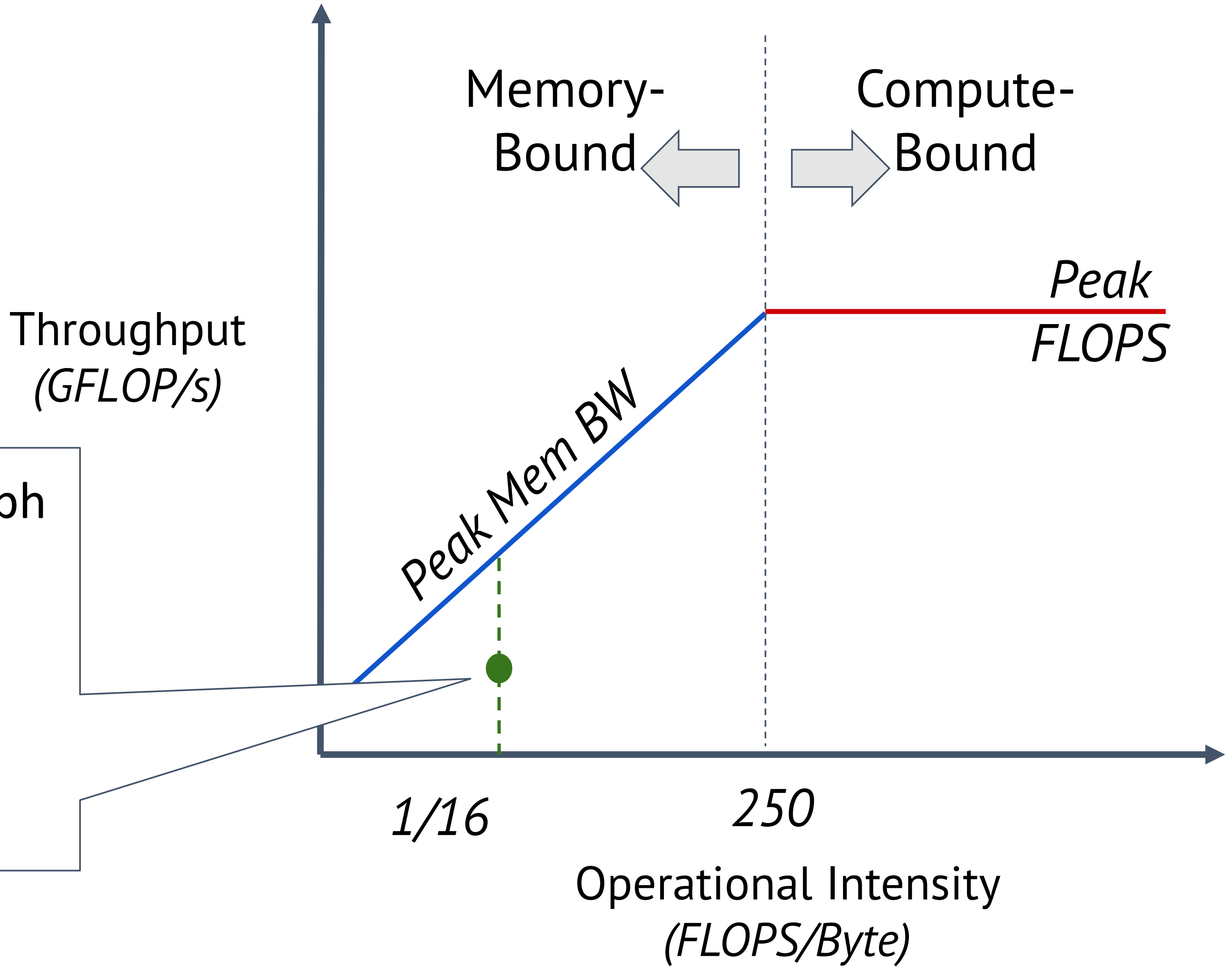
Bytes to Load: 8B for edge, 4B srcData, 4B dstData

Operational Intensity = 1 / (8+4+4) **= 1/16**

Graph Applications are Memory-Bound



Graph Applications are Memory-Bound



DRAM BW utilization in graph apps is ~50%

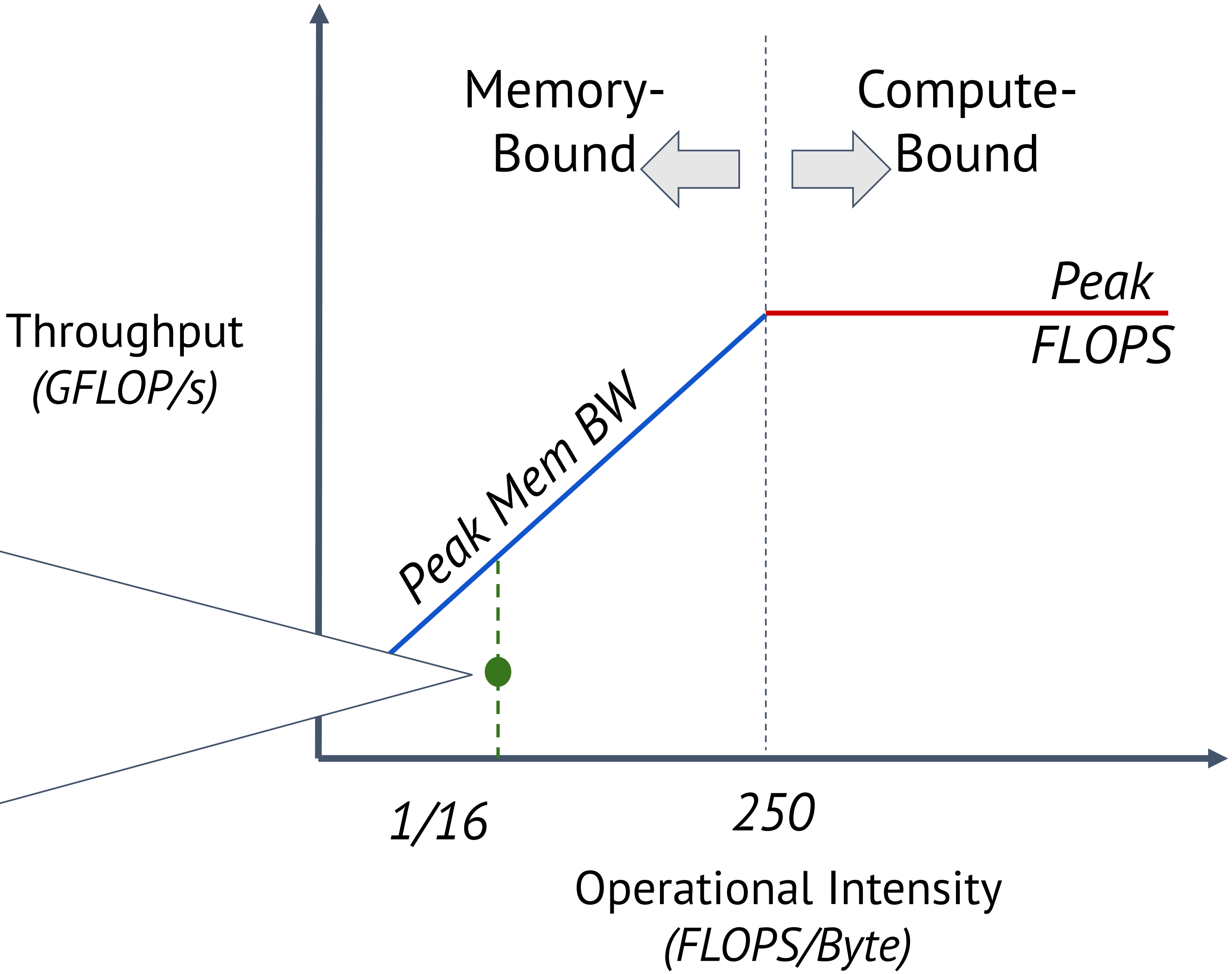
Why would we have spare BW capacity to go to memory and not use it?

Graph Applications are Memory-Bound

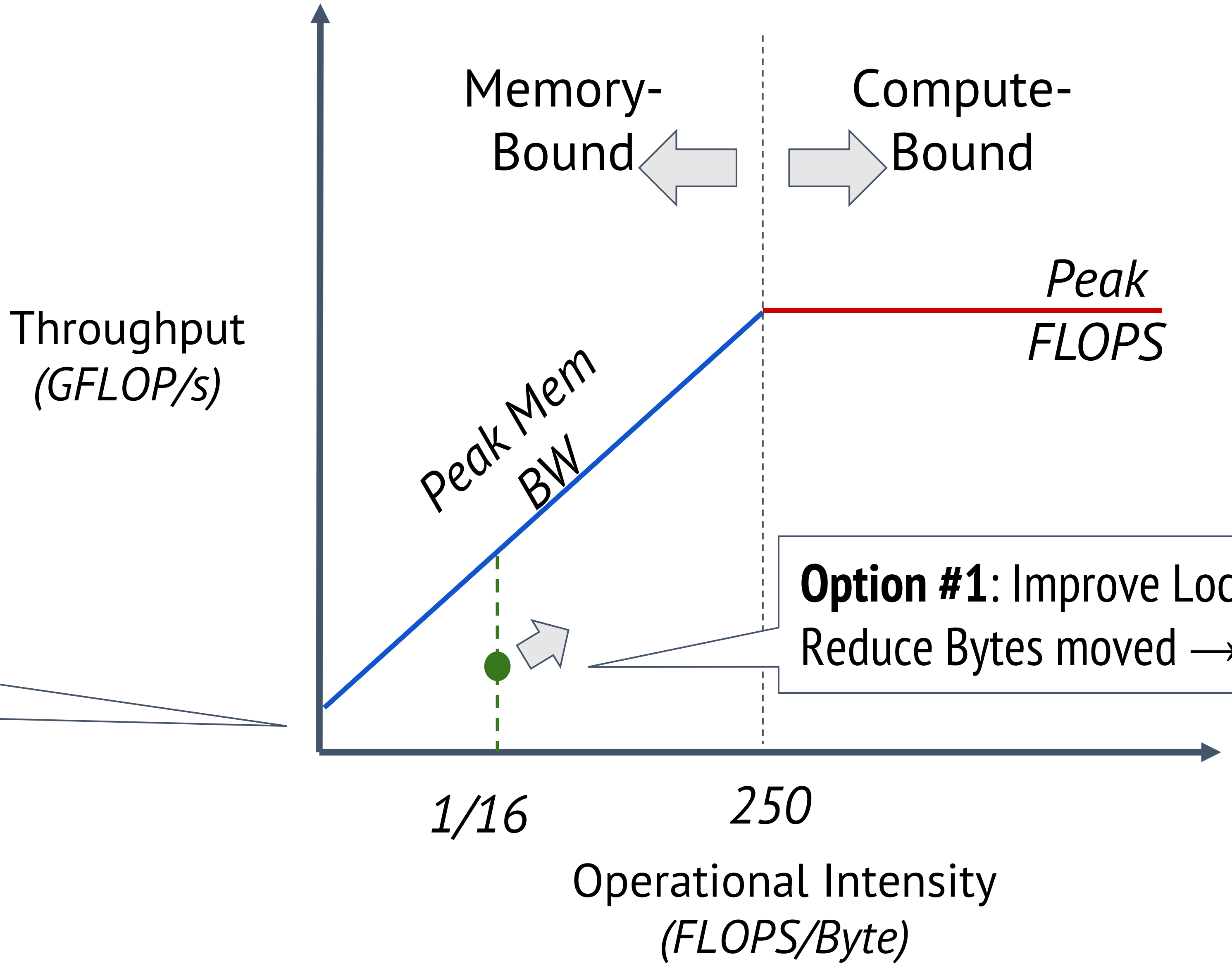
DRAM BW utilization in graph apps is ~50%

Why would we have spare BW capacity to go to memory and not use it?

Don’t know what to fetch next (no temporal locality), can’t use extra stuff we fetch (no spatial locality). Limited ability to send more memory requests (limited mem. parallelism).



Graph Applications are Memory-Bound

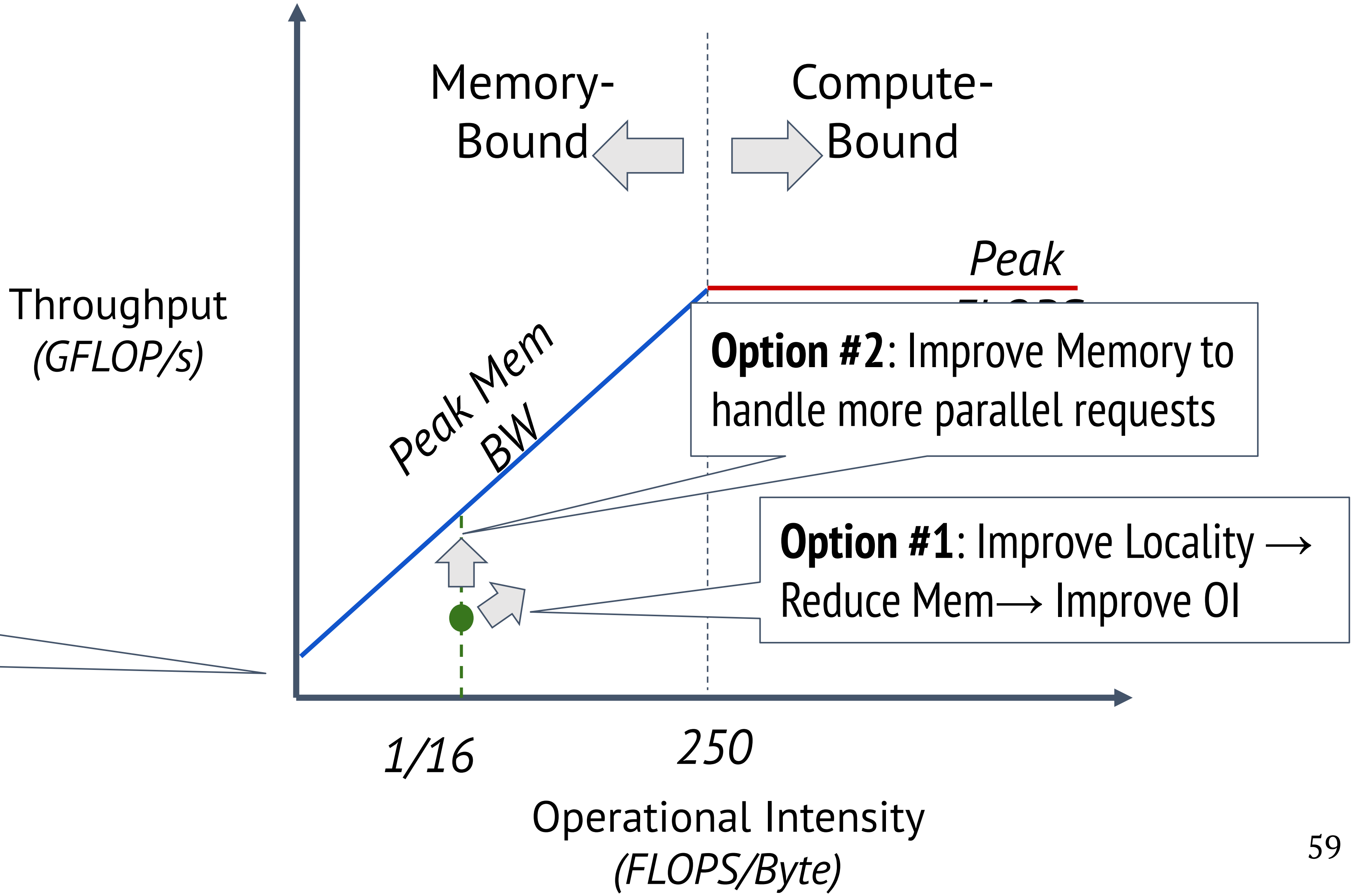


How to improve BW utilization?

Option #1: Improve Locality → Reduce Bytes moved → Improve OI

Graph Applications are Memory-Bound

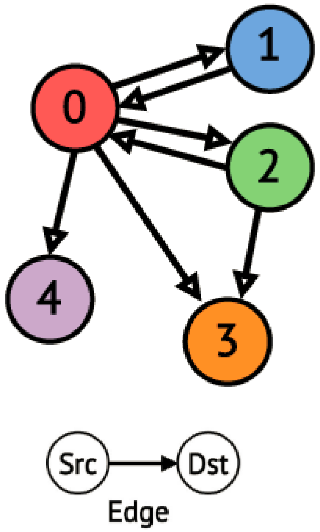
How to improve BW utilization?



Option #2: Improve Memory to handle more parallel requests

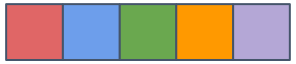
Option #1: Improve Locality → Reduce Mem→ Improve OI

Operational Intensity of Irregular Graph Applications



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

COO
(EdgeList)



```
for e in EL:
  dstData[e.dst] += srcData[e.src]
```

Ideal Best Possible Operational Intensity?

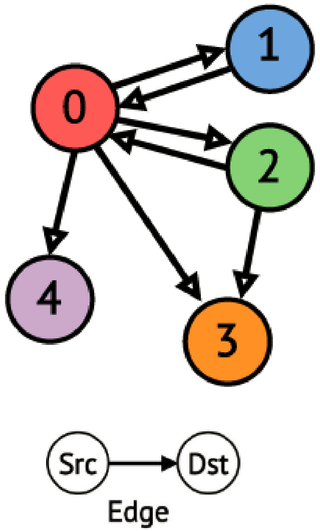
Operations per byte:

Operations: 1 addition

Bytes to Load:

Operational Intensity =

Ideal Operational Intensity of Irregular Graph Applications



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

COO
(EdgeList)

🟥🟦🟩🟧🟪

```
for e in EL:
  dstData[e.dst] += srcData[e.src]
```

Ideal Best Possible Operational Intensity?

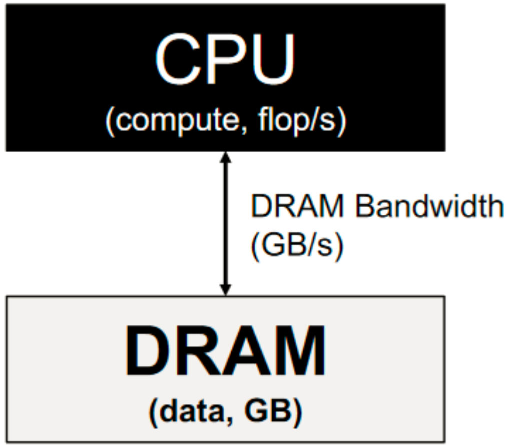
Operations per byte:

Operations: 1 addition

Bytes to Load: 8B for edge, 0B srcData, 0B dstData

Operational Intensity = 1 / (8+0+0) **= 1/8**

Improving Performance by Improving Locality



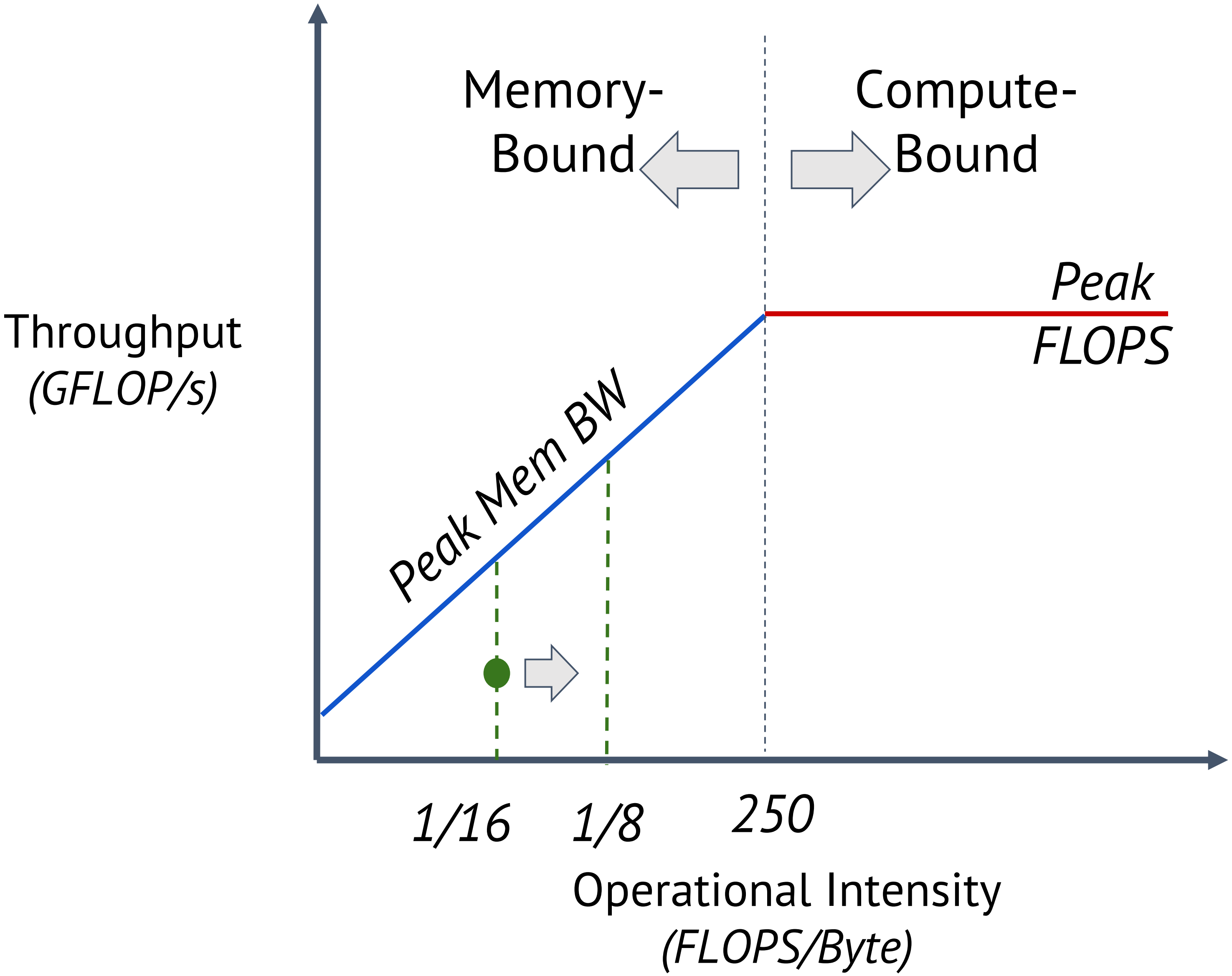
CPU

(compute, flop/s)

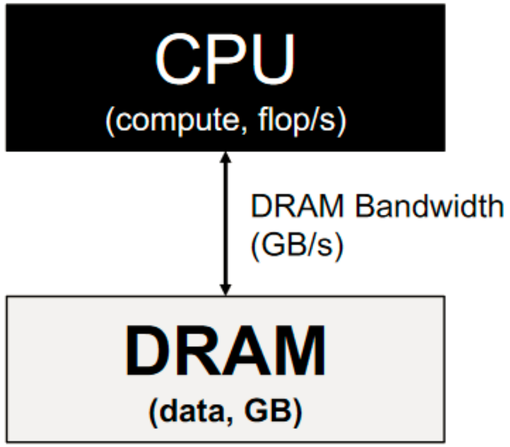
DRAM Bandwidth (GB/s)

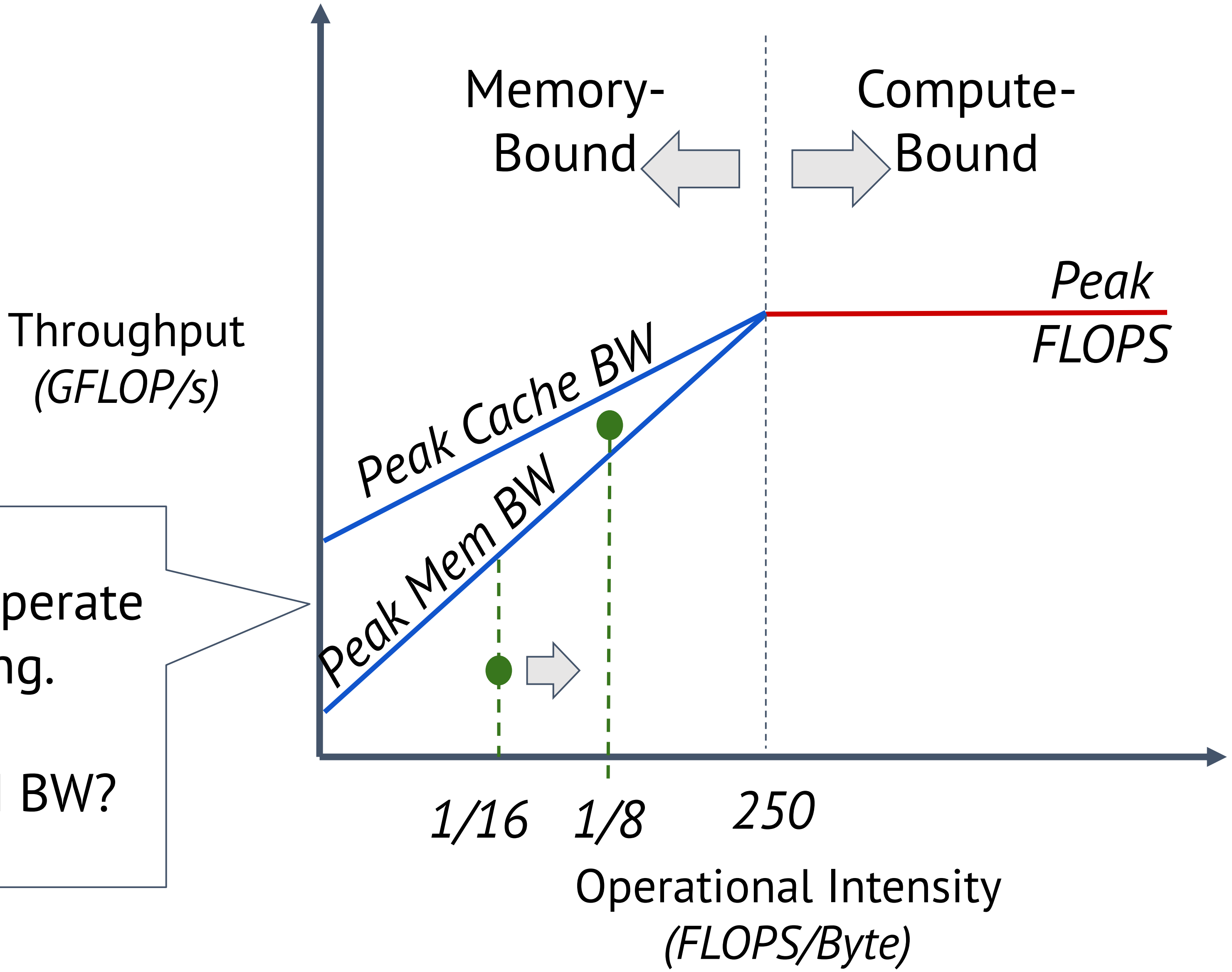
DRAM

(data, GB)



Improving Performance by Improving Locality

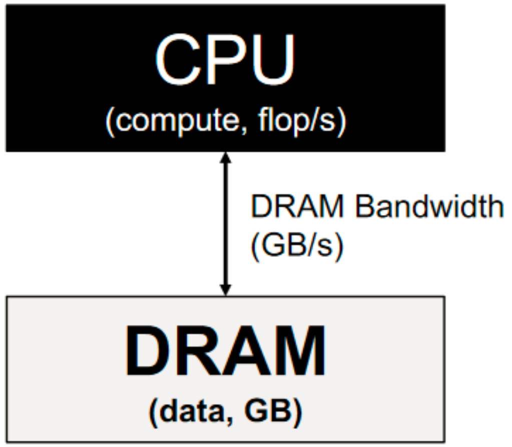




Locality wins: If we can operate out of cache, higher ceiling.

Why is cache BW > DRAM BW?

Improving Performance by Improving Locality



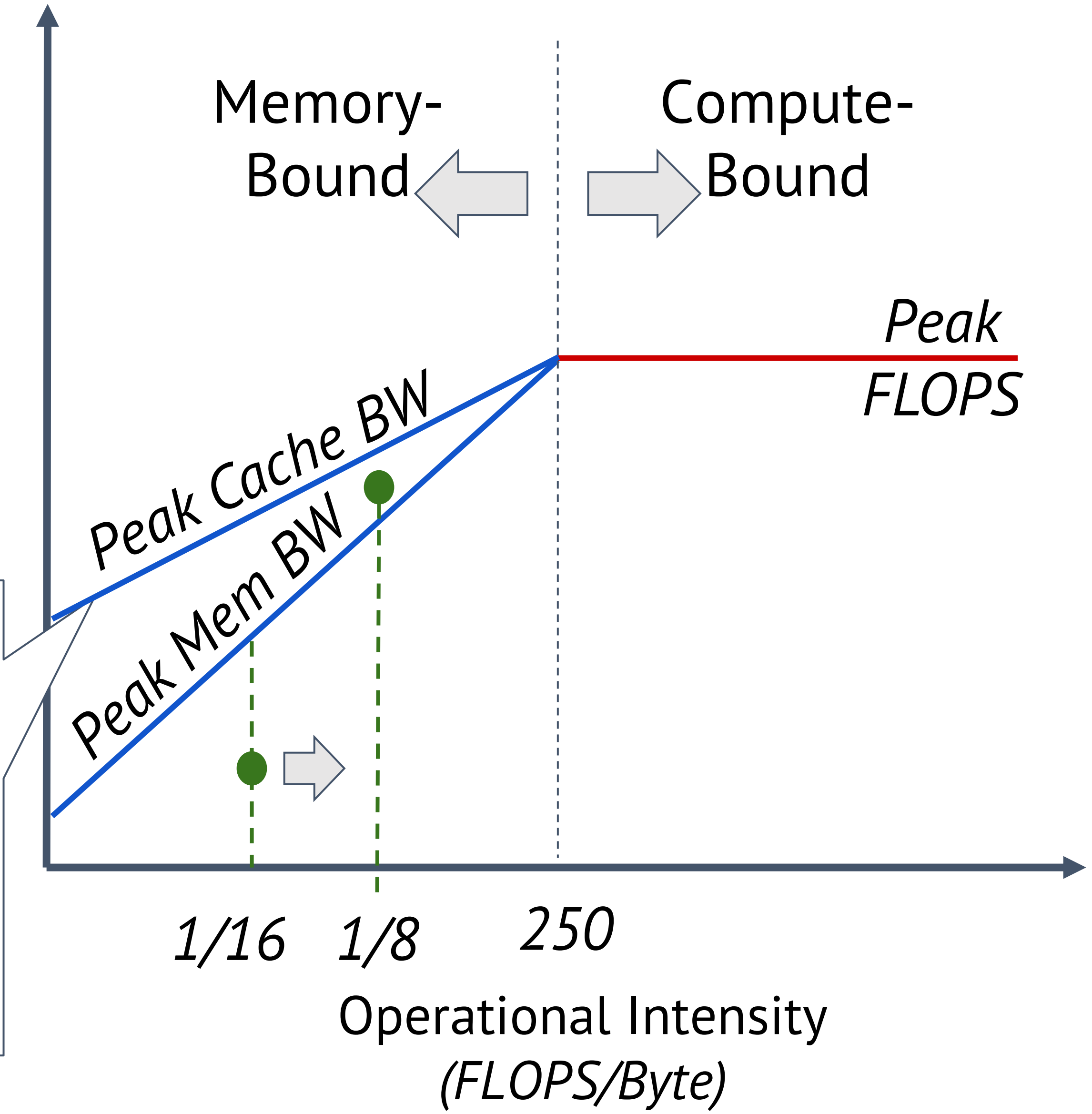
CPU

(compute, flop/s)

DRAM Bandwidth (GB/s)

DRAM

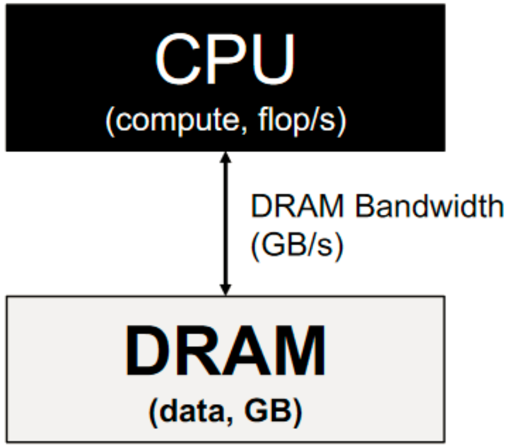
(data, GB)



Locality wins: If we can operate out of cache, higher ceiling & more leftward ridge point.

Why is cache BW > DRAM BW?
Smaller SRAM caches much faster.

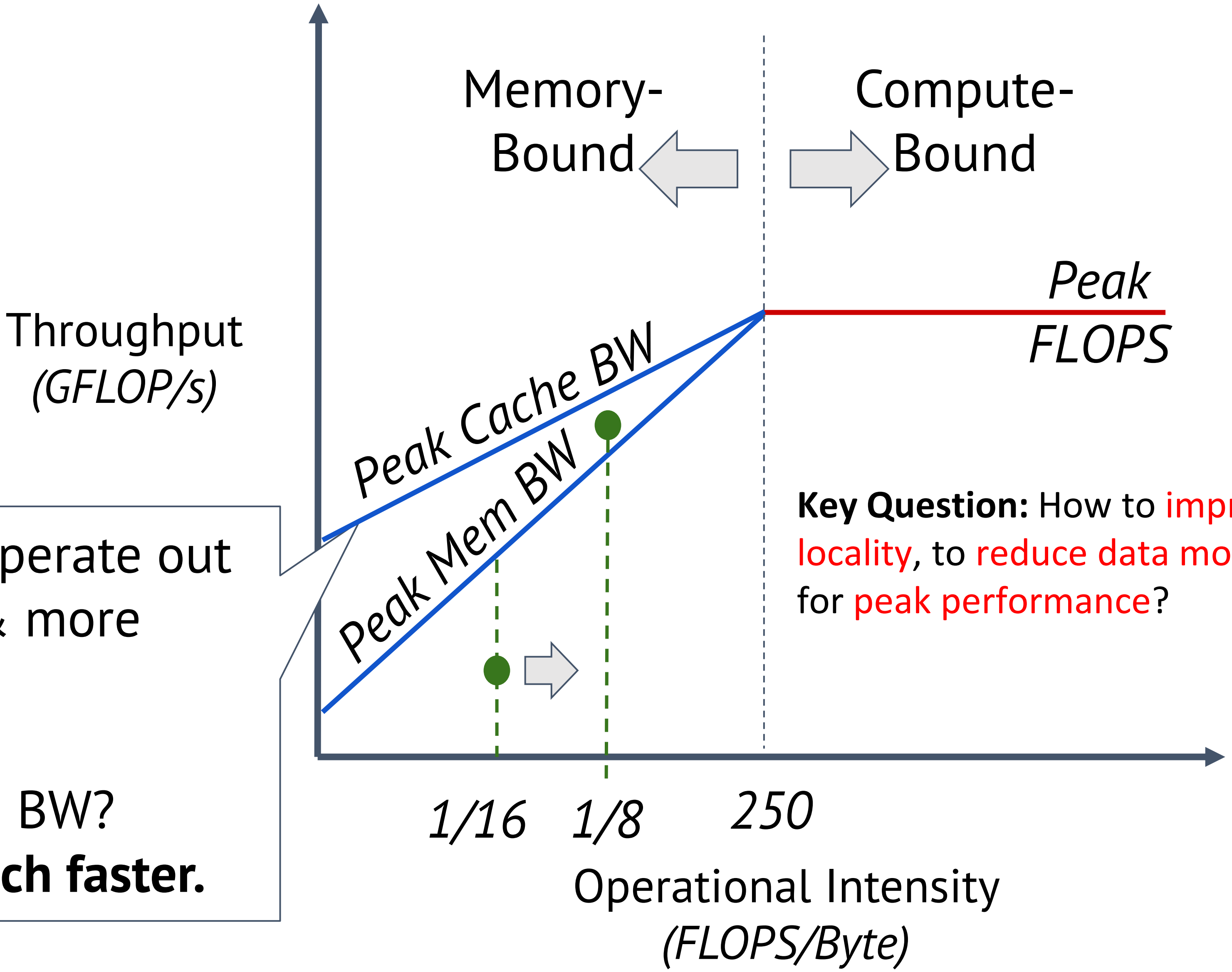
Improving Performance by Improving Locality



CPU
(compute, flop/s)

DRAM Bandwidth
(GB/s)

DRAM
(data, GB)



Key Question: How to improve locality, to reduce data movement, for peak performance?

Locality wins: If we can operate out of cache, higher ceiling & more leftward ridge point.

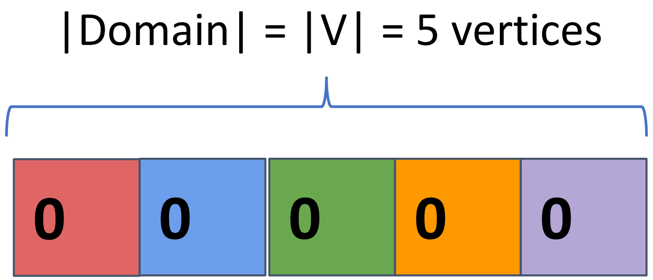
Why is cache BW > DRAM BW?
Smaller SRAM caches much faster.

Propagation Blocking: Optimizing Sparse Irregular Writes to Improve Cache Locality

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

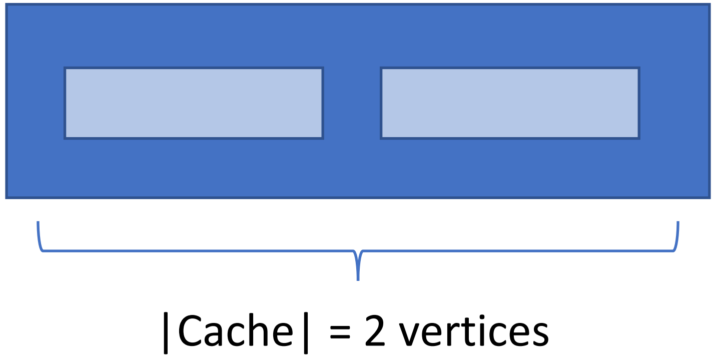
0	1
2	0
1	0
0	2
2	3
0	4
0	3

COO
(EdgeList)



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

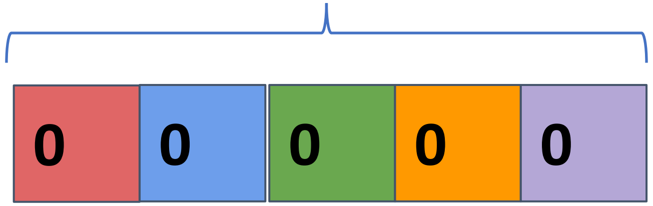


Propagation Blocking: Reorganize Input to Make Memory Being Randomly Written Fit in Cache

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

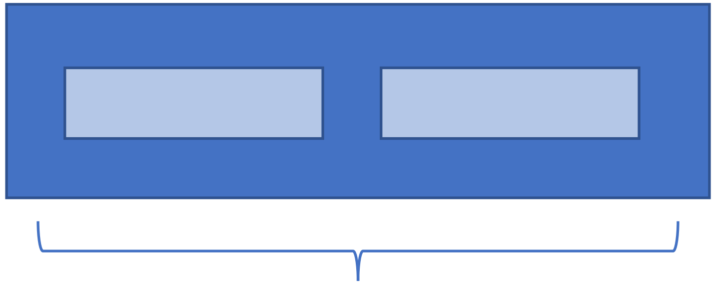
COO
(EdgeList)

|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

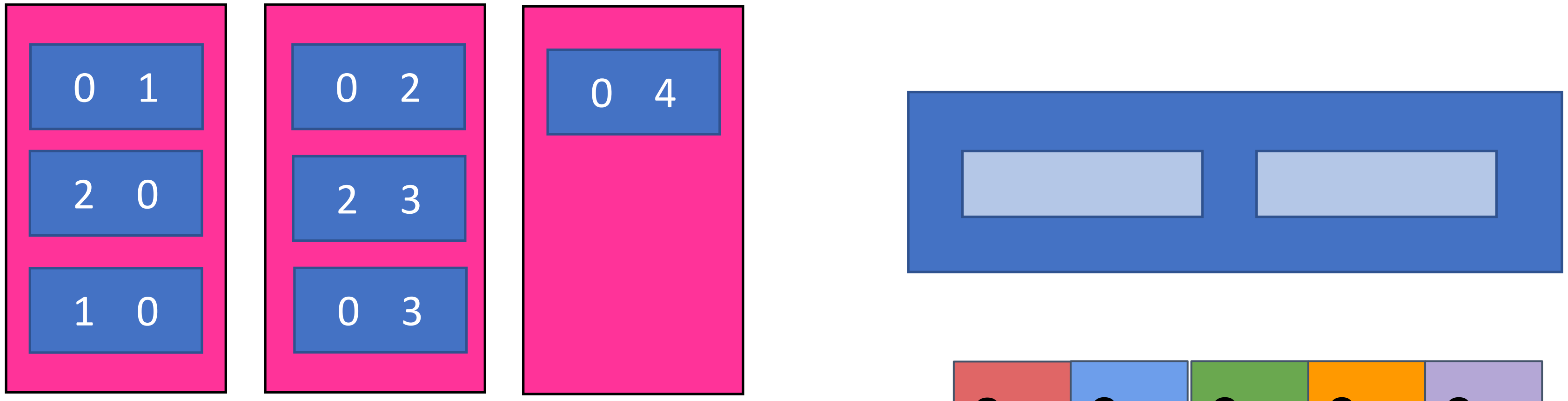
Key idea in propagation blocking: Limit the domain of updates to a *sub-space* of vertices, **V***, so that |V*| <= |C| and do multiple sub-spaces of V*s, so that all V*s together = V

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

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

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

COO
(EdgeList)



Bin 0:
dst 0-1

Bin 1:
dst 2-3

Bin 2:
dst 4-5

0	0	0	0	0

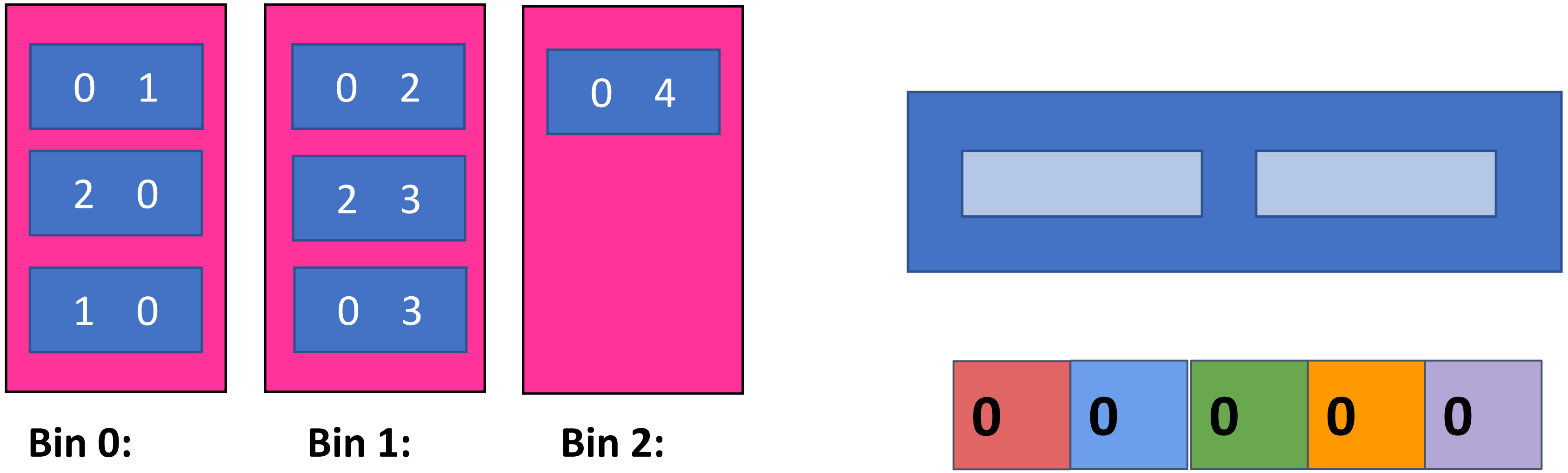
dstData

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

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

COO
(EdgeList)



Bin 0: dst 0-1	Bin 1: dst 2-3	Bin 2: dst 4-5
0 1	0 2	0 4
2 0	2 3	
1 0	0 3	

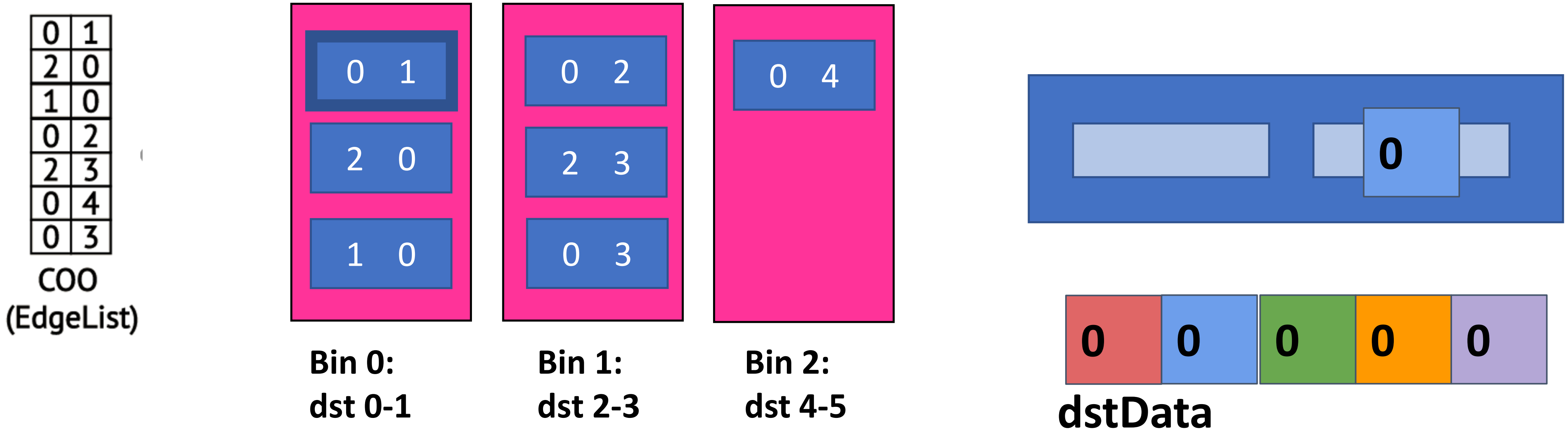
0	0	0	0	0

dstData

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



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

COO (EdgeList)

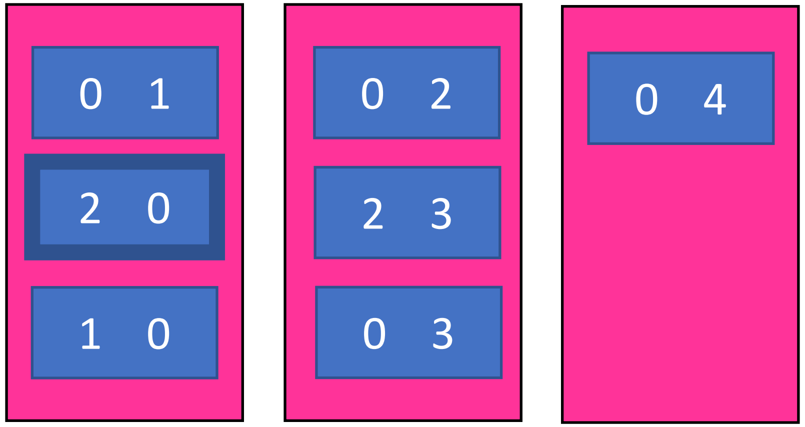
Execute the kernel for one bin at a time

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

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

COO
(EdgeList)

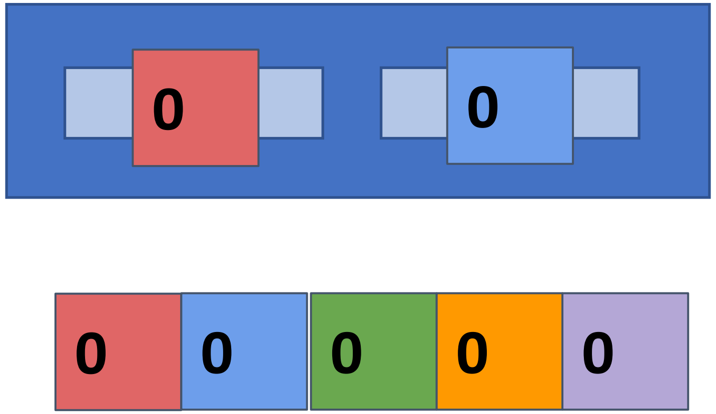


Bin 0:
dst 0-1

Bin 1:
dst 2-3

Bin 2:
dst 4-5

Execute the kernel for one bin at a time



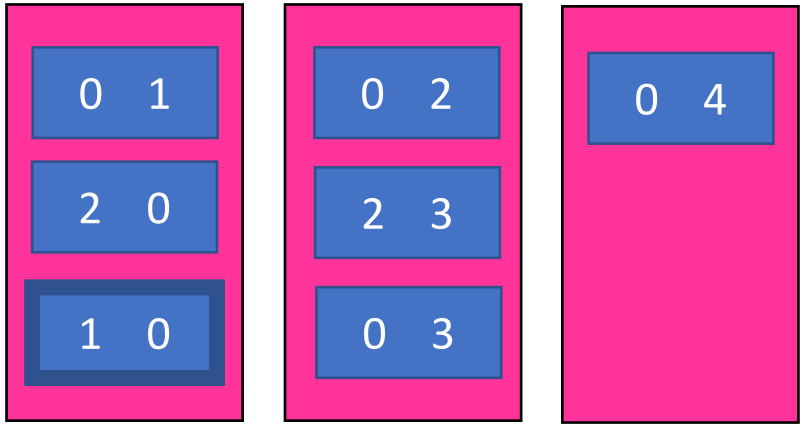
dstData

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

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

COO
(EdgeList)

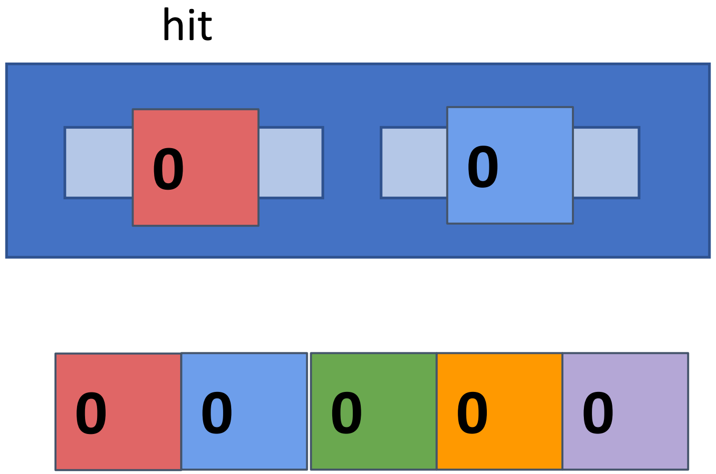


Bin 0:
dst 0-1

Bin 1:
dst 2-3

Bin 2:
dst 4-5

Execute the kernel for one bin at a time



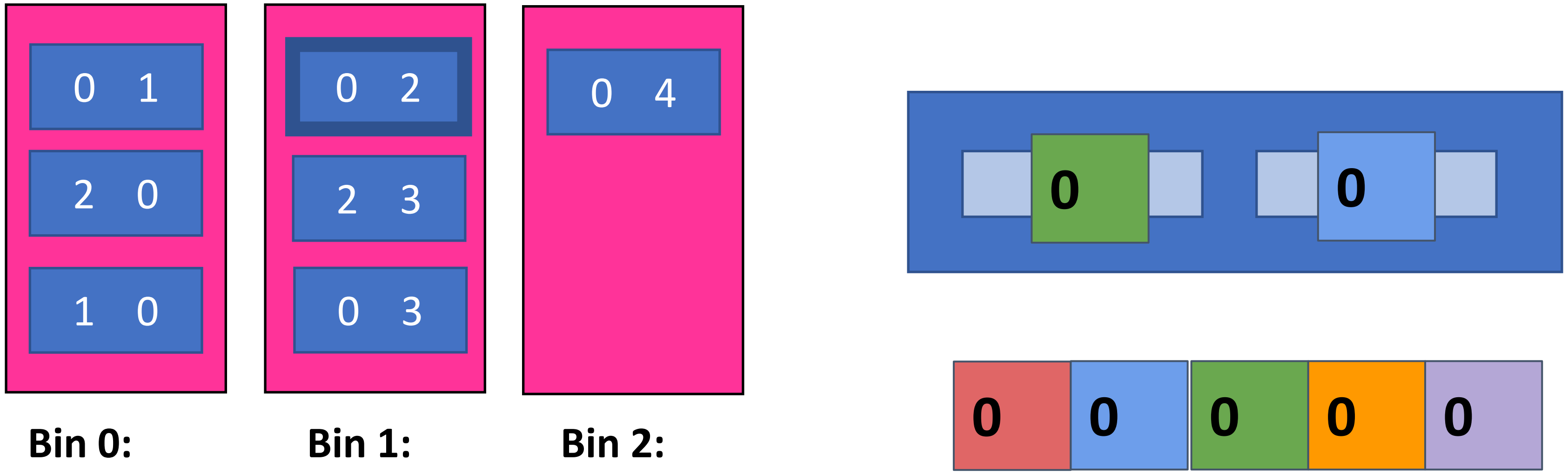
dstData

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

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

COO
(EdgeList)



Bin 0:
dst 0-1

Bin 1:
dst 2-3

Bin 2:
dst 4-5

Execute the kernel for one bin at a time

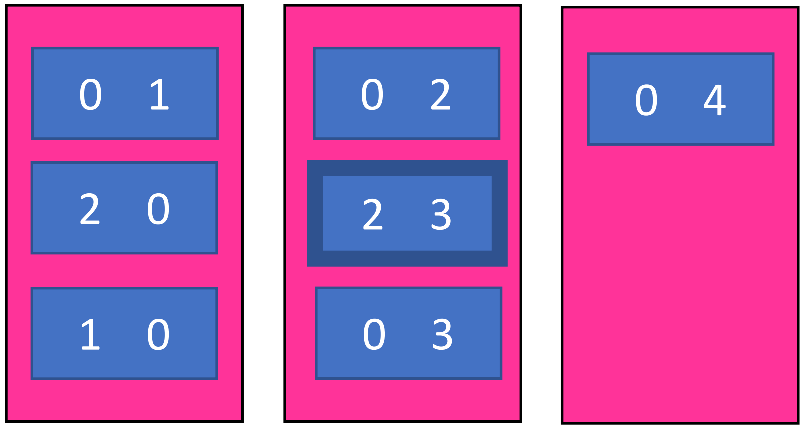
dstData

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

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

COO
(EdgeList)

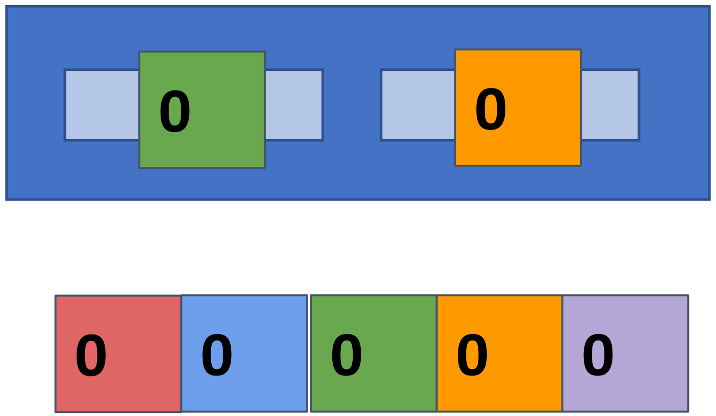


Bin 0:
dst 0-1

Bin 1:
dst 2-3

Bin 2:
dst 4-5

Execute the kernel for one bin at a time

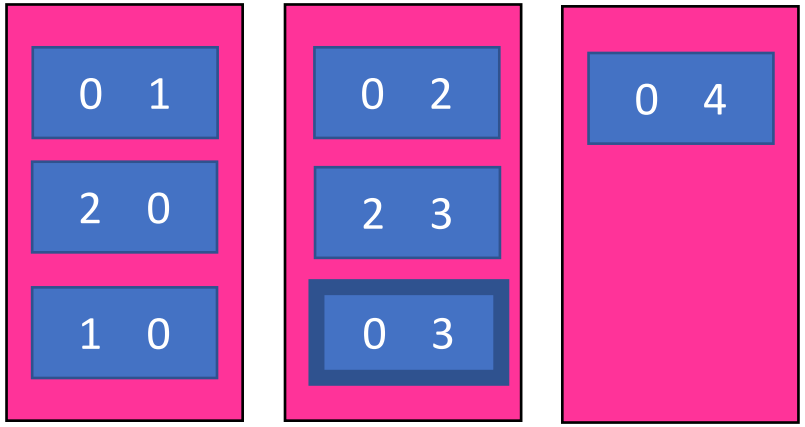


dstData
Remember: dstData[e.dst] ++
and e.dst is random, from edge list

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

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

COO
(EdgeList)

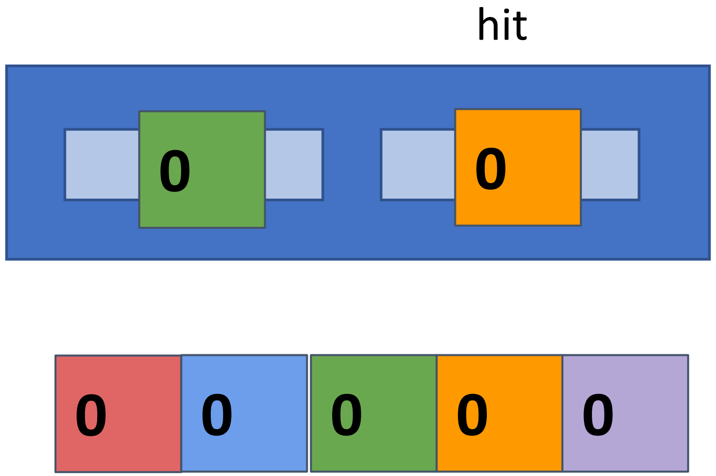


Bin 0:
dst 0-1

Bin 1:
dst 2-3

Bin 2:
dst 4-5

Execute the kernel for one bin at a time

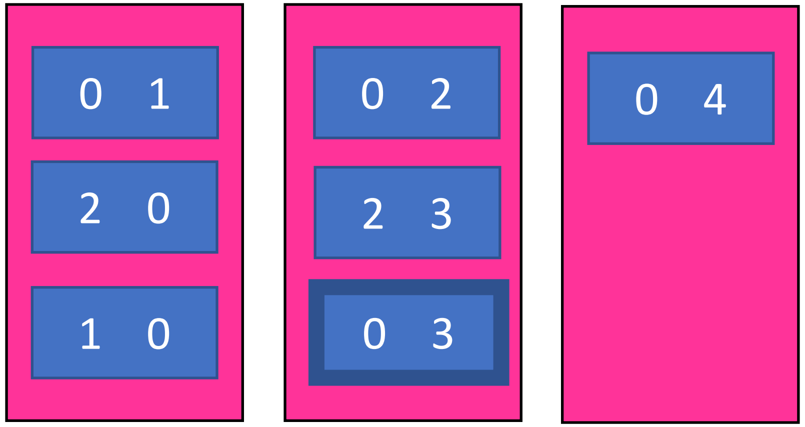


dstData
Remember: dstData[e.dst] ++
and e.dst is random, from edge list

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

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

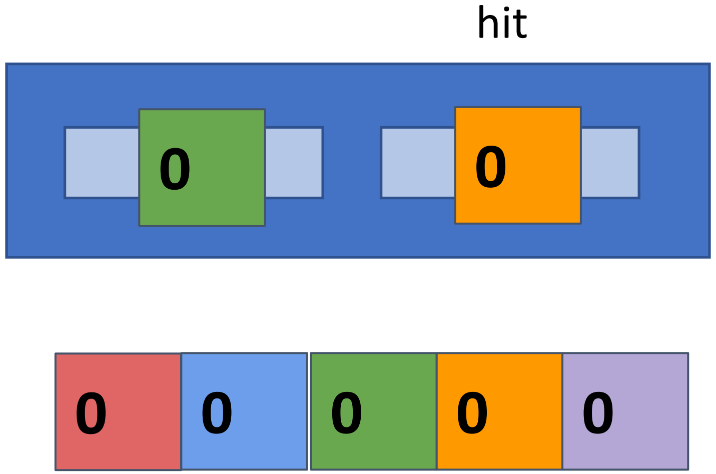
COO
(EdgeList)



Bin 0:
dst 0-1

Bin 1:
dst 2-3

Bin 2:
dst 4-5



dstData

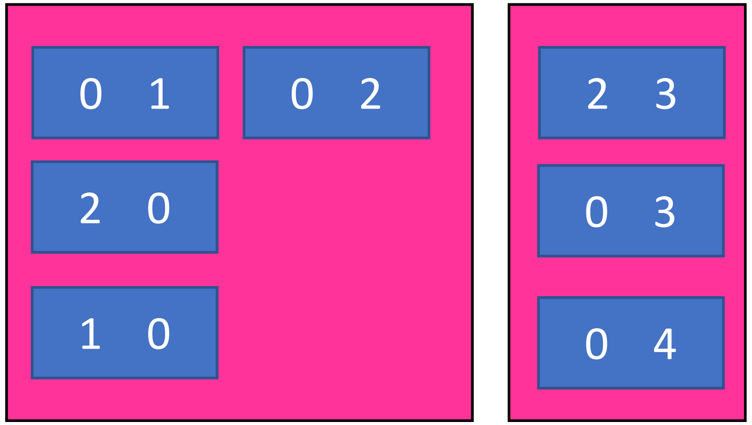
How to decide how many vertices go in each of your Propagation Blocker's bins?

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

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

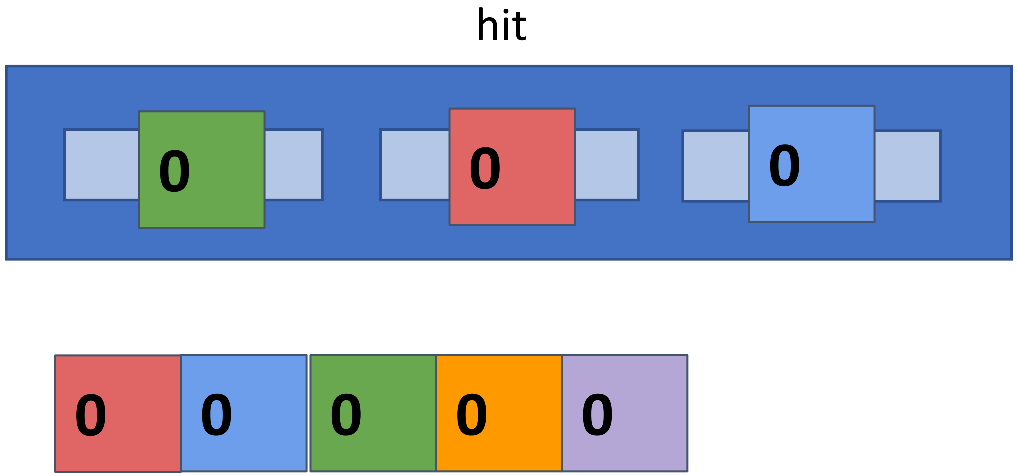
COO
(EdgeList)



Bin 0:
dst 0-2

Bin 1:
dst 3-5

Match destinations per bin to number of vertices worth of dstData that can fit in cache at one time

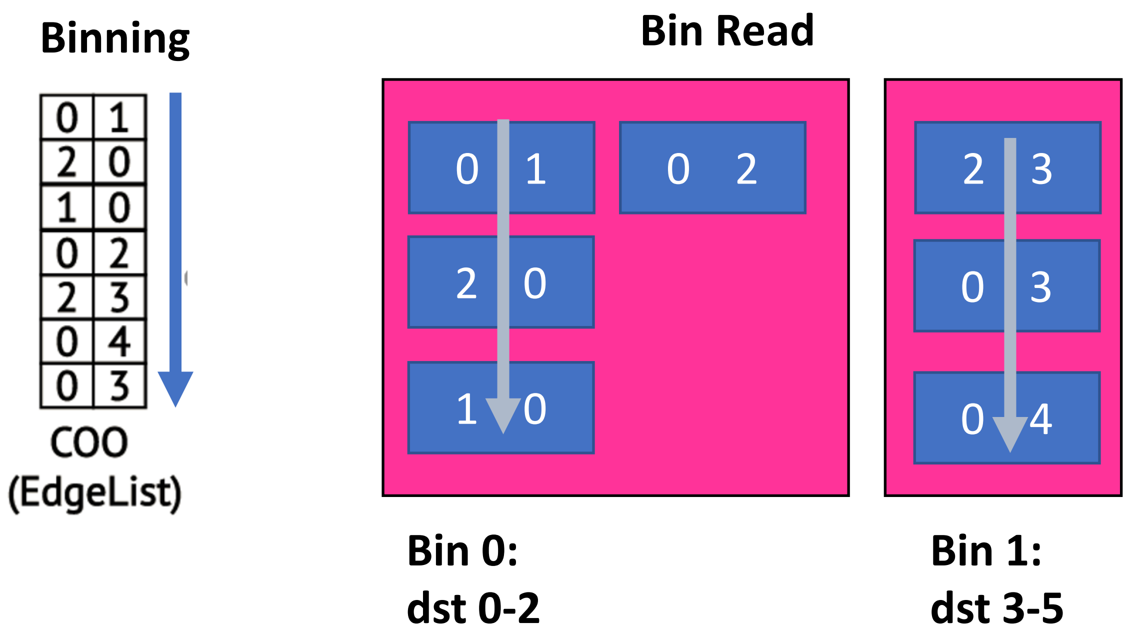


dstData

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

Propagation Blocking: Performance Analysis

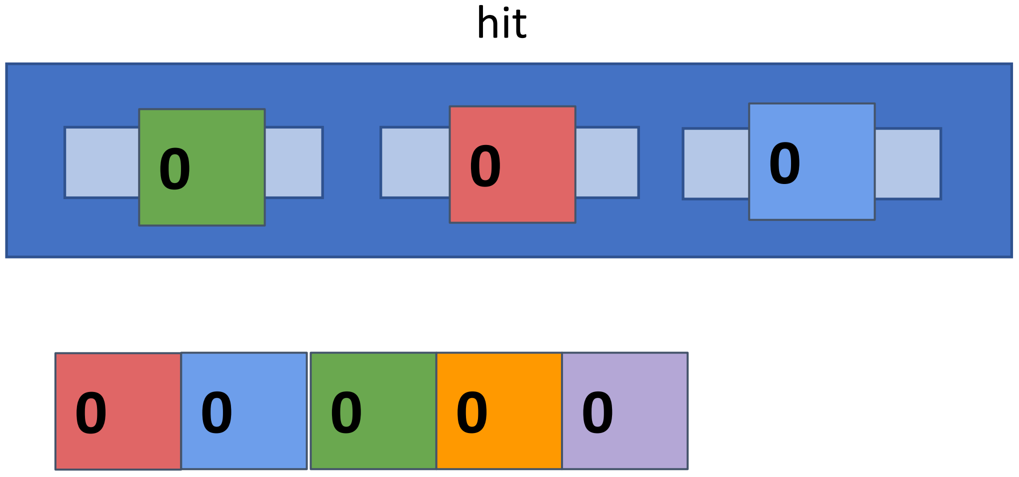
Traverse the edge list twice instead of once



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

COO (EdgeList)

All locations written fit in cache! Compulsory misses on dstData[] only: all the rest hit.



dstData

Remember: dstData[e.dst] ++

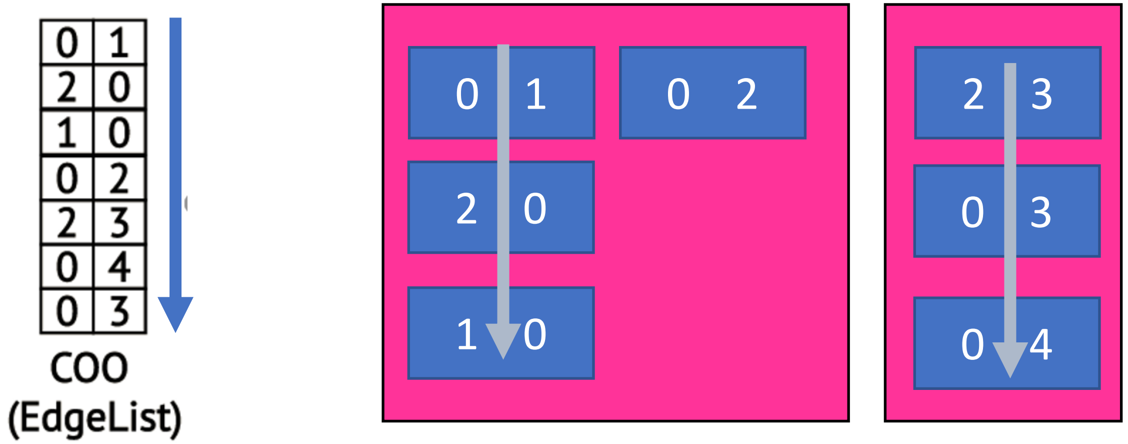
and e.dst is random, from edge list

Propagation Blocking: Performance Analysis

Traverse the edge list twice instead of once

Binning

Bin Read



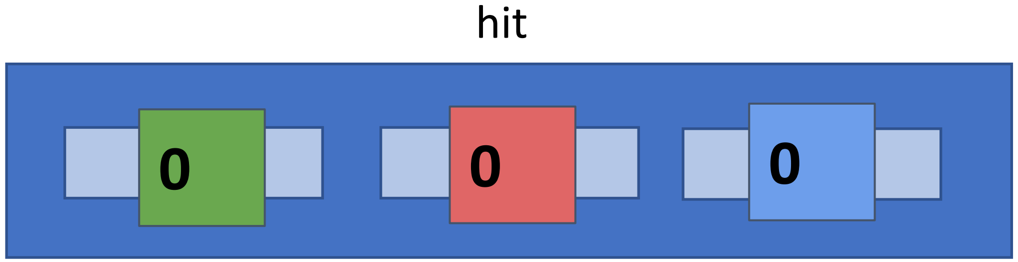
0	1
2	0
1	0
0	2
2	3
0	4
0	3

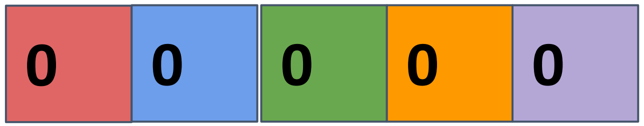
Bin 0:
dst 0-2

Bin 1:
dst 3-5

What about the performance of reading the edge list during binning?

All locations written fit in cache! Compulsory misses on dstData[] only: all the rest hit.





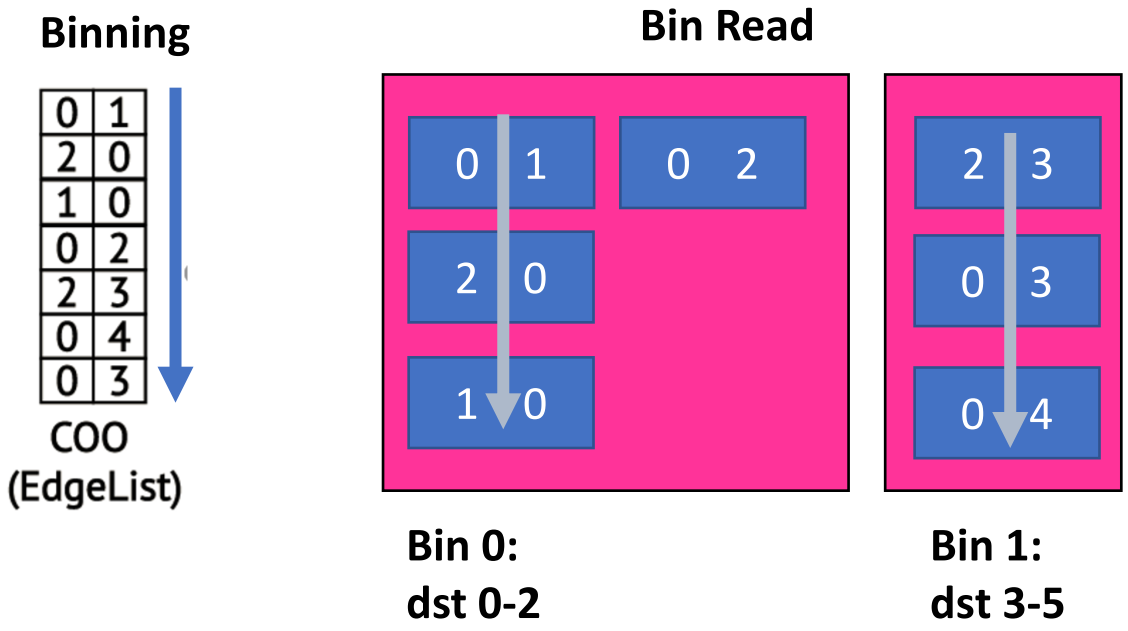
dstData

Remember: dstData[e.dst] ++

and e.dst is random, from edge list

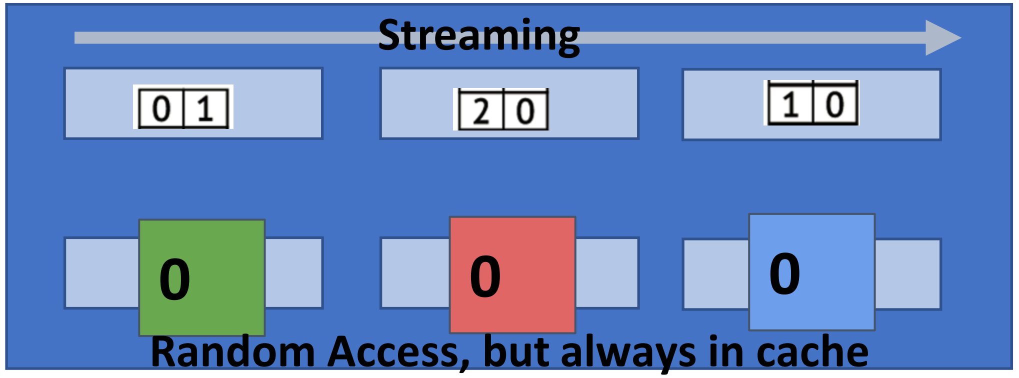
Propagation Blocking: Performance Analysis

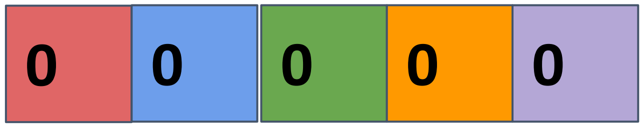
Traverse the edge list twice instead of once



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

Usually save a little space in cache for *streaming edge list* data. Easy to cache.





dstData

Remember: dstData[e.dst] ++

and e.dst is random, from edge list

What about propagation blocking for irregular reads?

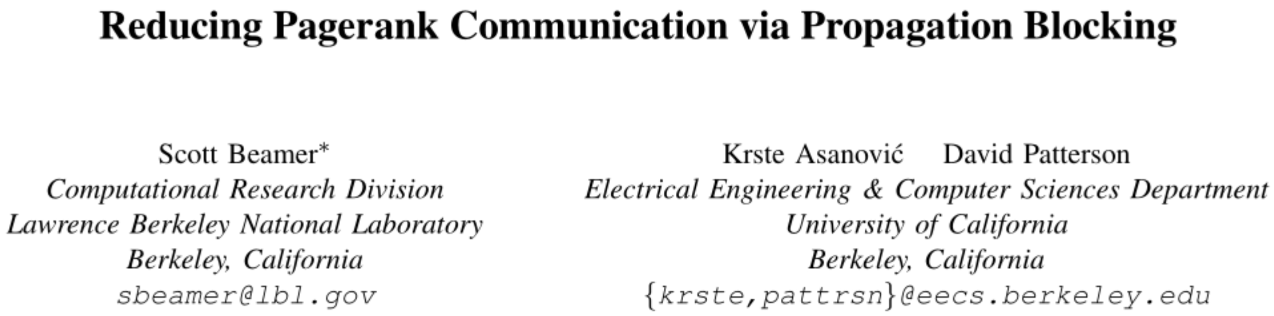
Propagation Blocking

```
PropagationBlocking_EdgeCount(EdgeList E){

  Bins B[];
  for edge in E{
    add_to_bin( find_bin(edge) )
  }

  for bin in B{
    for e in bin{
      dstData[e.dst]++
    }
  }

}
```



Reducing Pagerank Communication via Propagation Blocking

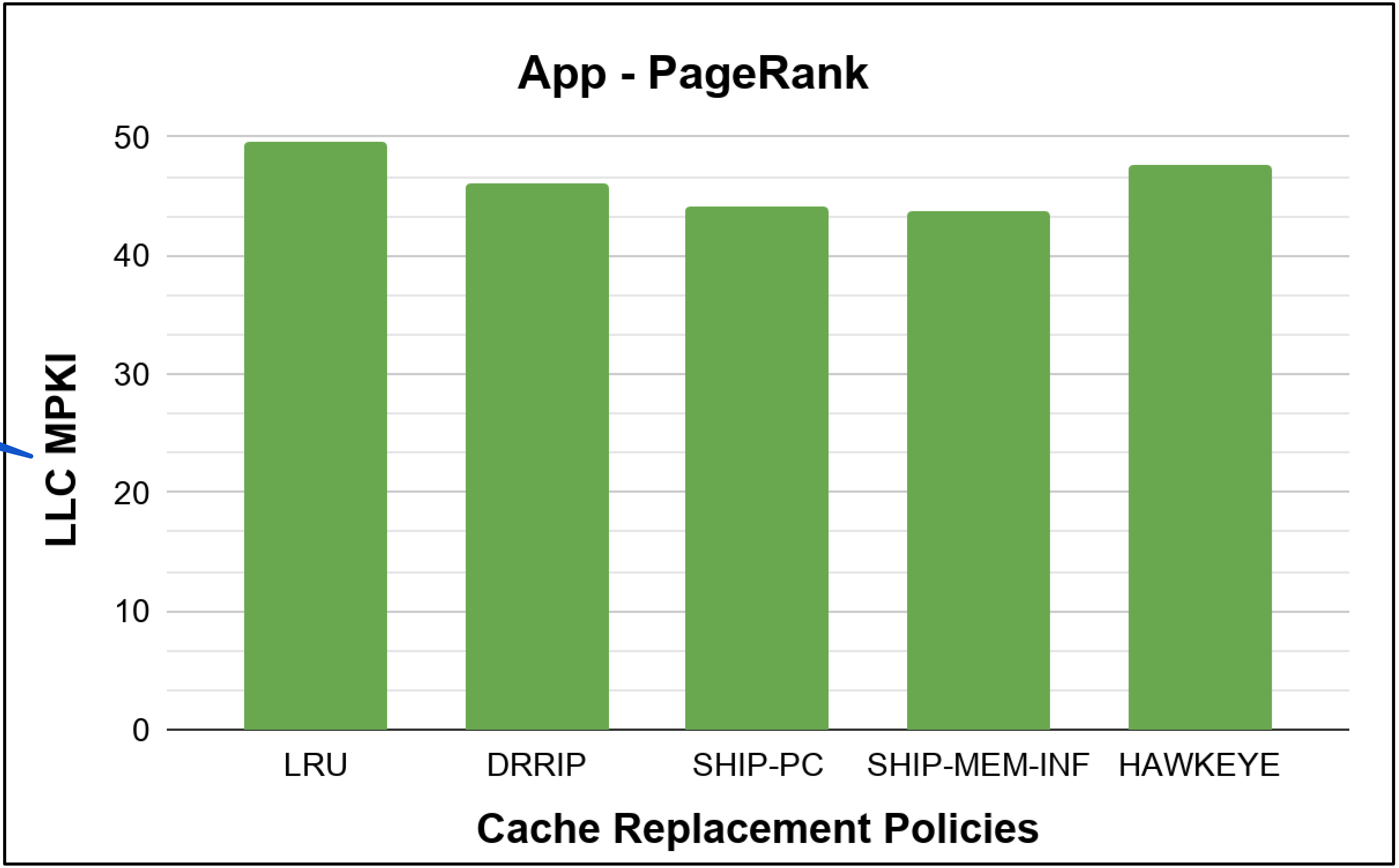
Scott Beamer*
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, California
sbeamer@lbl.gov

Krste Asanović David Patterson
Electrical Engineering & Computer Sciences Department
University of California
Berkeley, California
{krste,pattrsn}@eecs.berkeley.edu

Application of Propagation Blocking for Graph Applications (Page Rank only, at first) discovered in 2017
(Prior work on “radix partitioning” applied the idea to other domains, but not graphs)

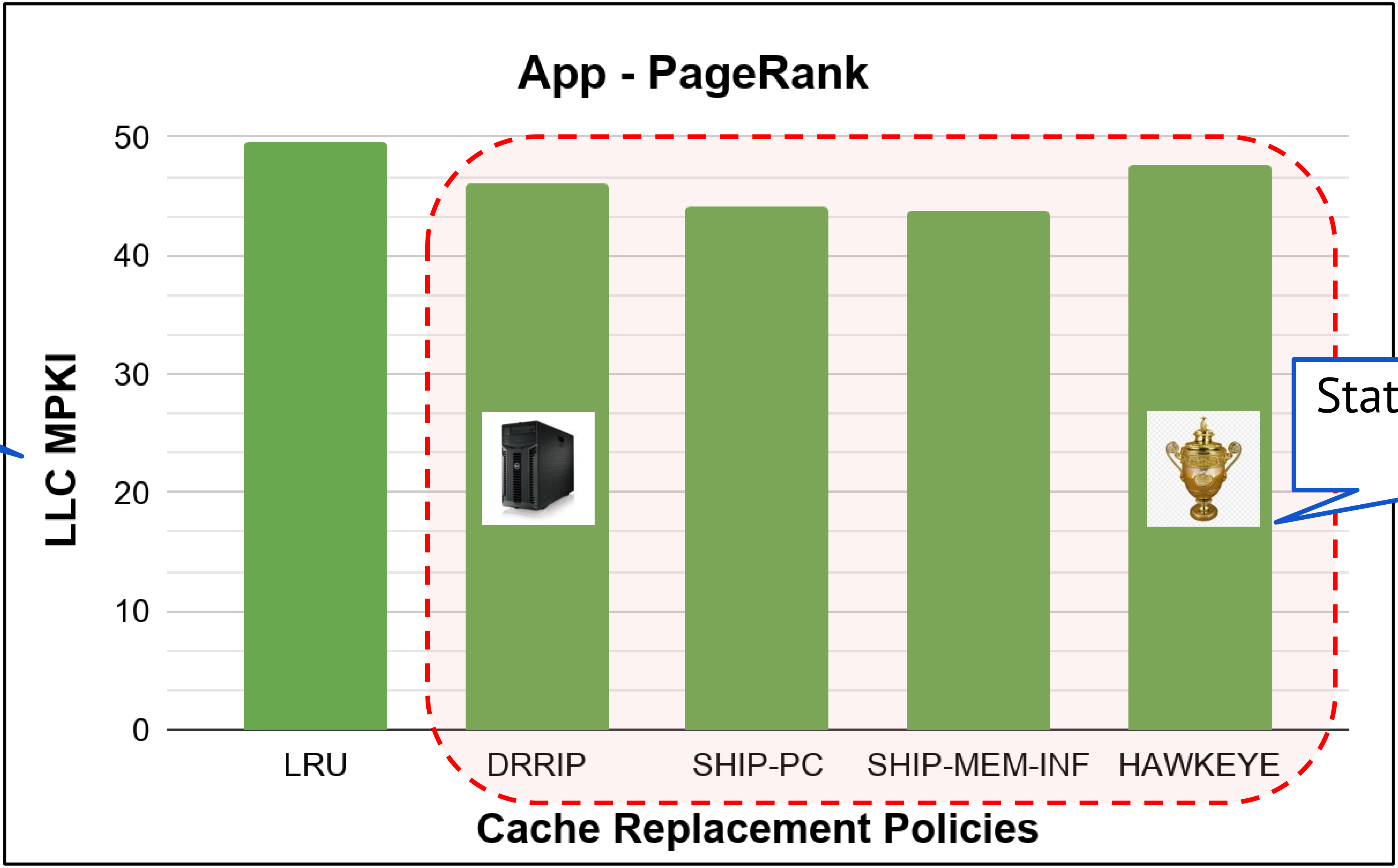
Cache Locality determines Overall Performance
What about better replacement policies?

Existing Replacement Policies Are Insufficient



Lower is Better

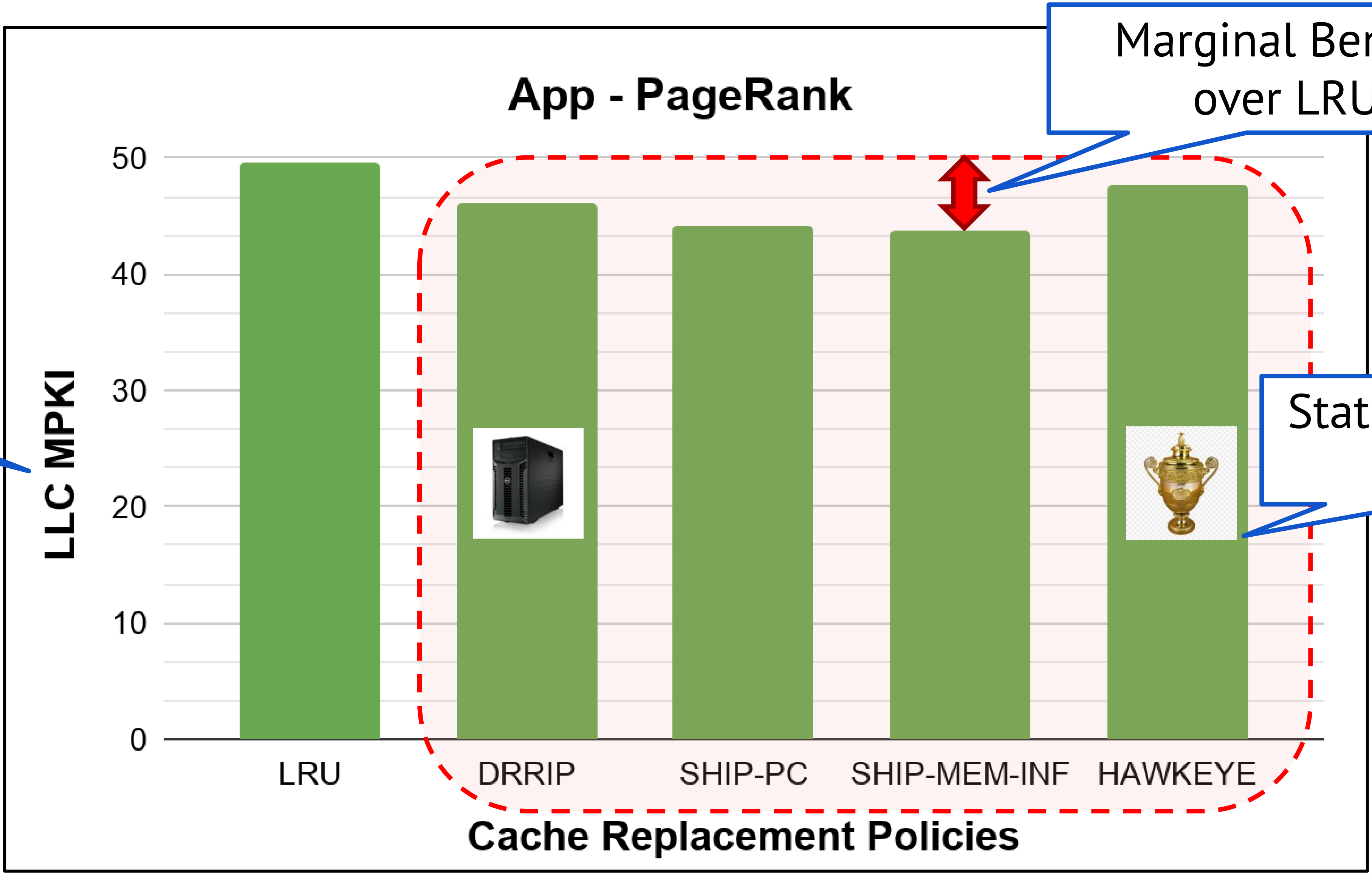
Existing Replacement Policies Are Insufficient



Lower is Better

State-of-the-Art Policies

Existing Replacement Policies Are Insufficient

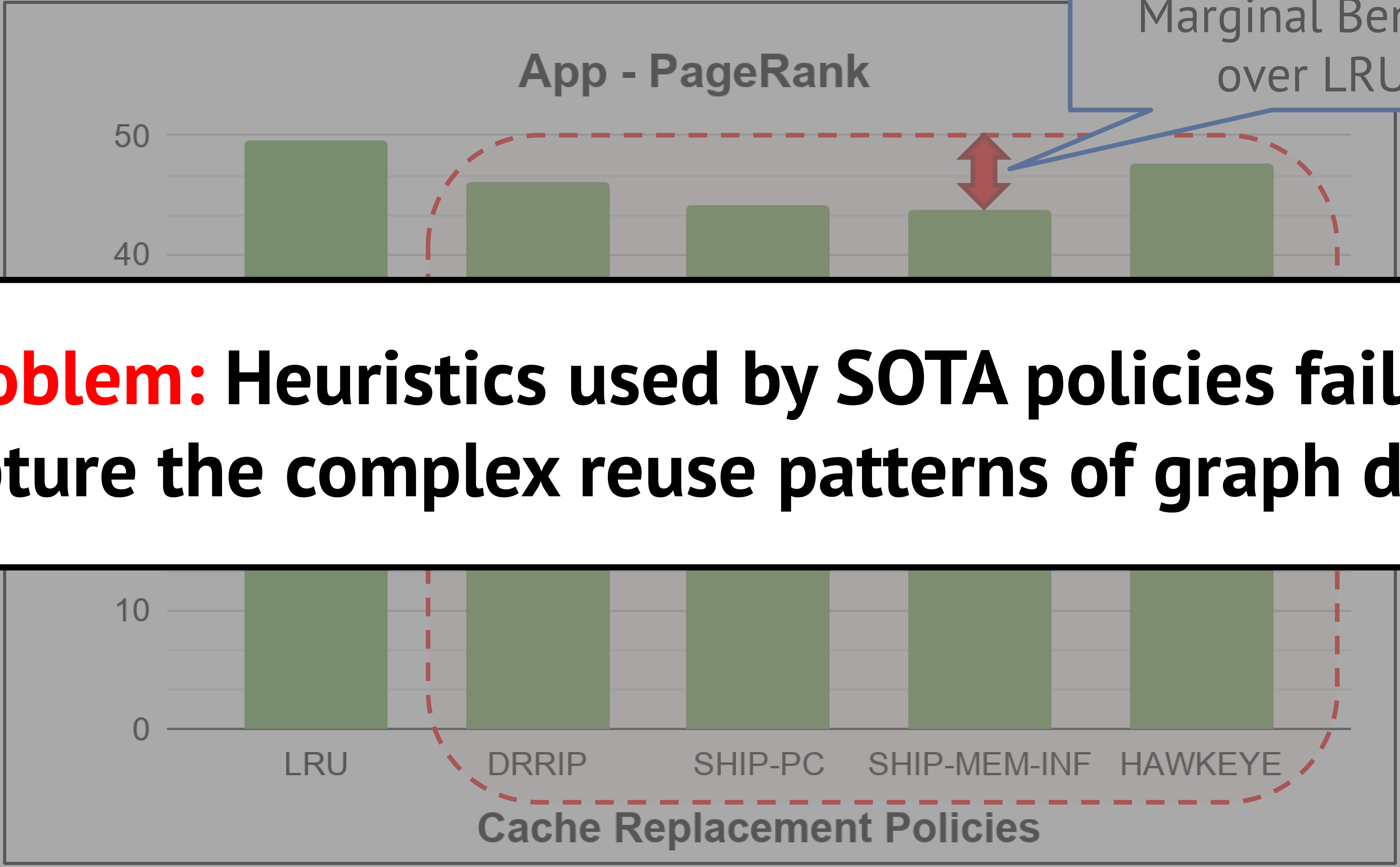


Marginal Benefit over LRU

Lower is Better

State-of-the-Art Policies

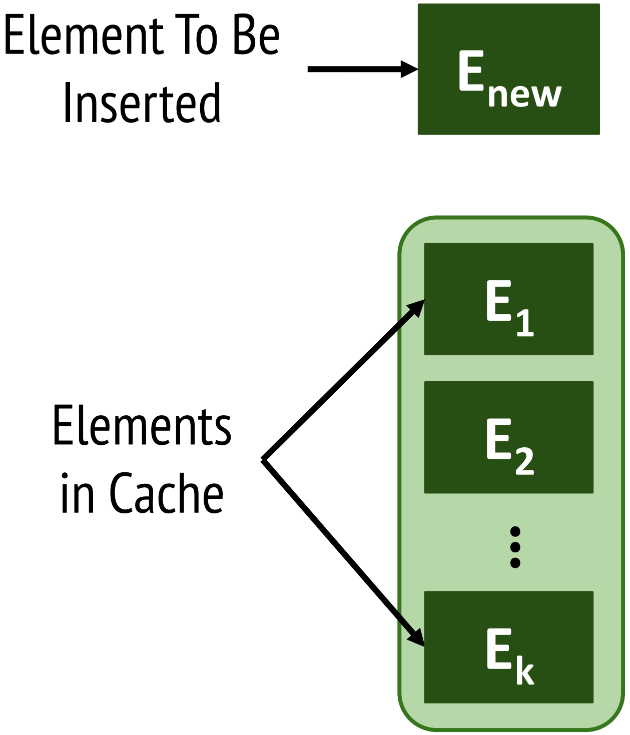
Existing Replacement Policies Are Insufficient



Marginal Benefit over LRU

Problem: Heuristics used by SOTA policies fail to capture the complex reuse patterns of graph data

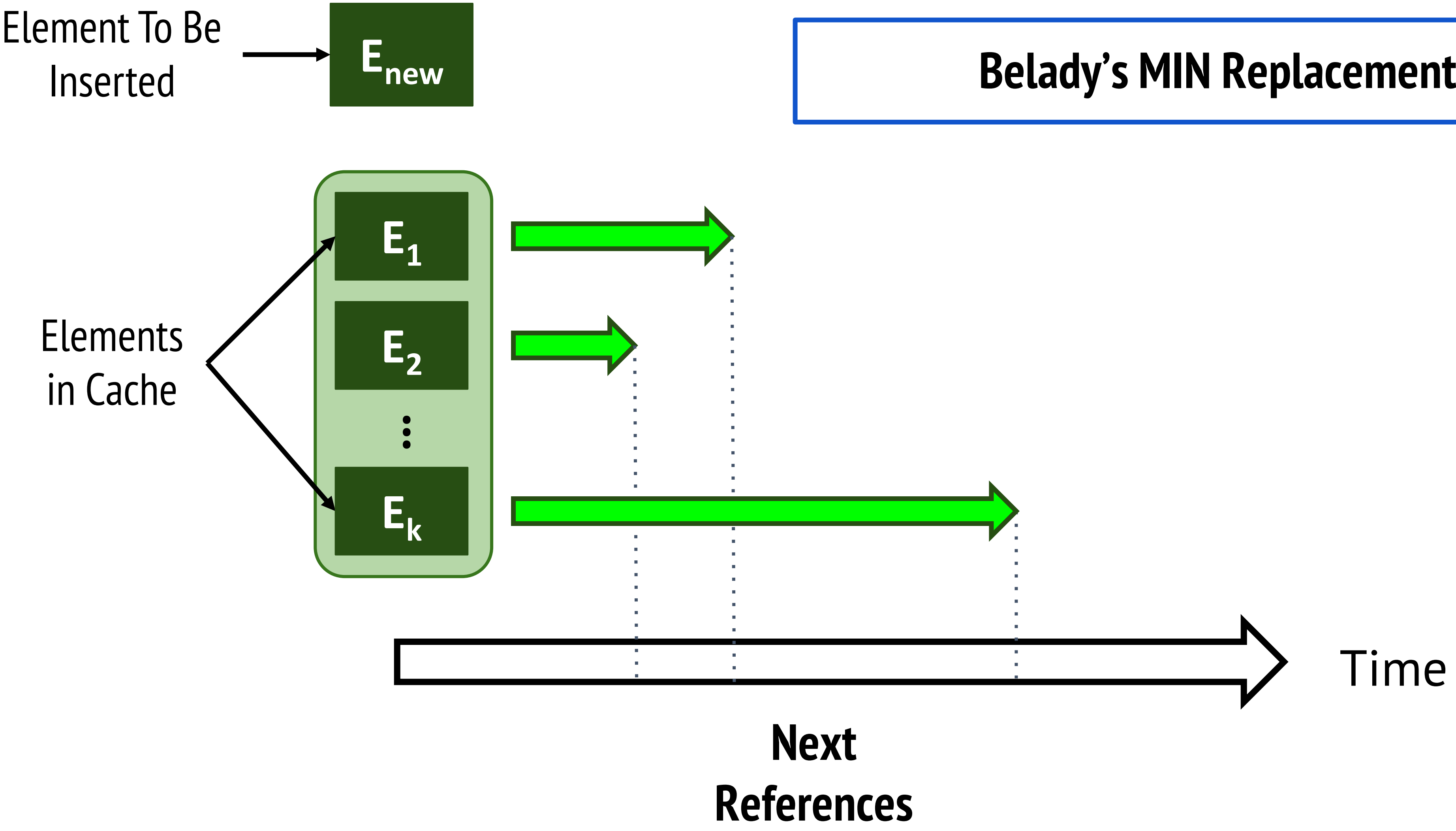
Is It Possible To Do Better Cache Replacement?



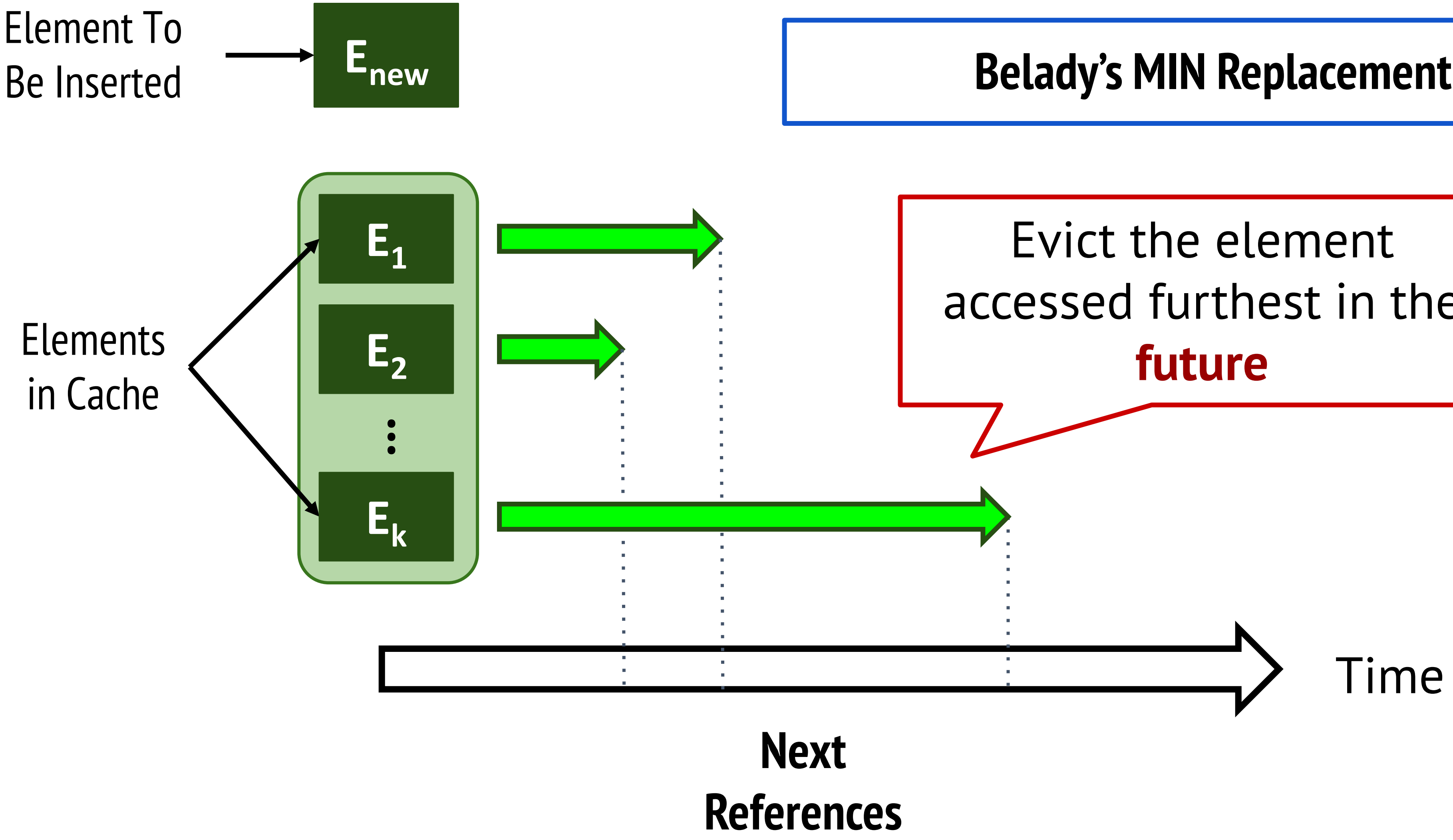
Belady's MIN Replacement Policy

Is It Possible To Do Better Cache Replacement?

Belady's MIN Replacement Policy



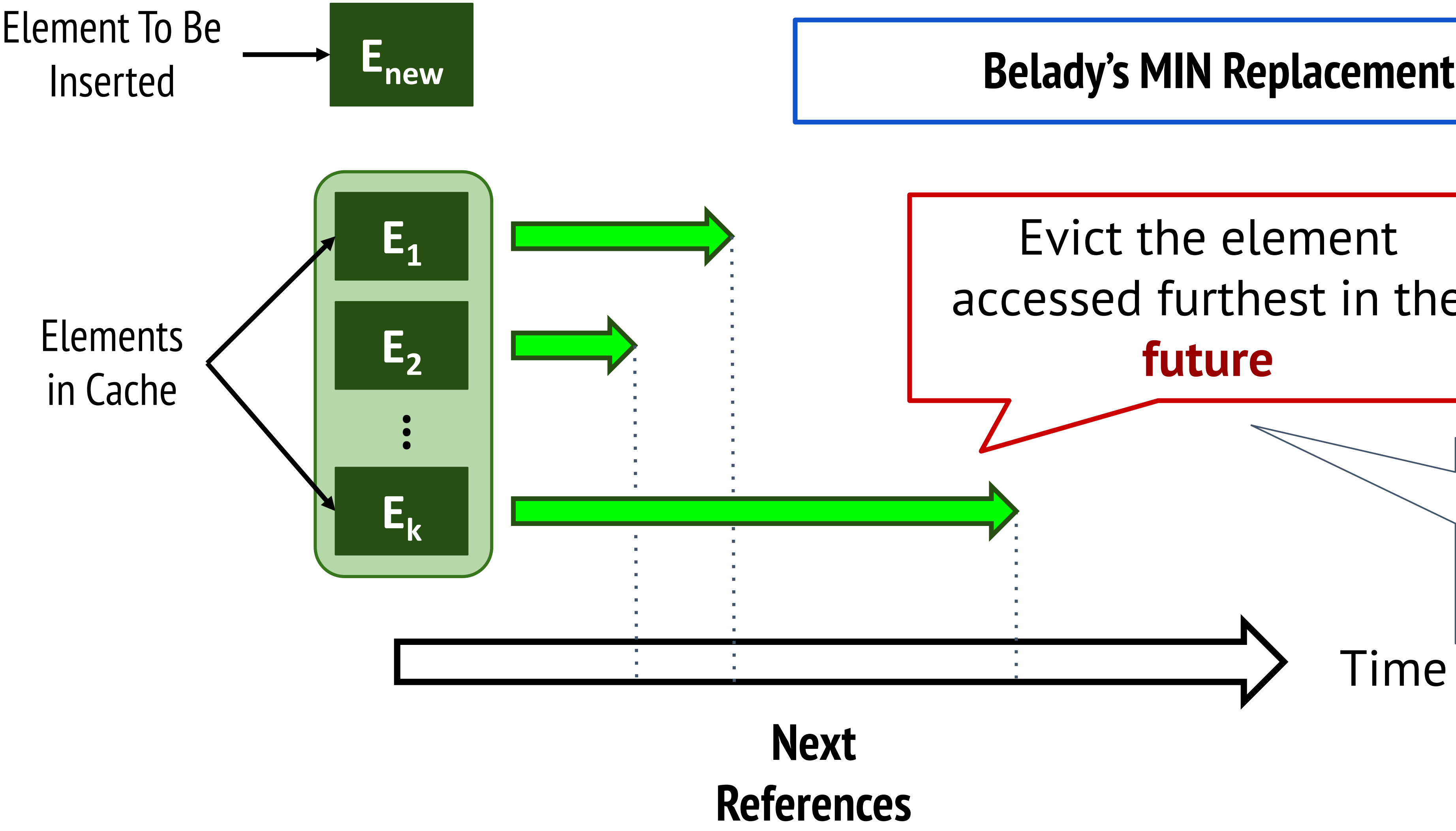
Is It Possible To Do Better Cache Replacement?



Belady's MIN Replacement Policy

Evict the element accessed furthest in the **future**

Is It Possible To Do Better Cache Replacement?

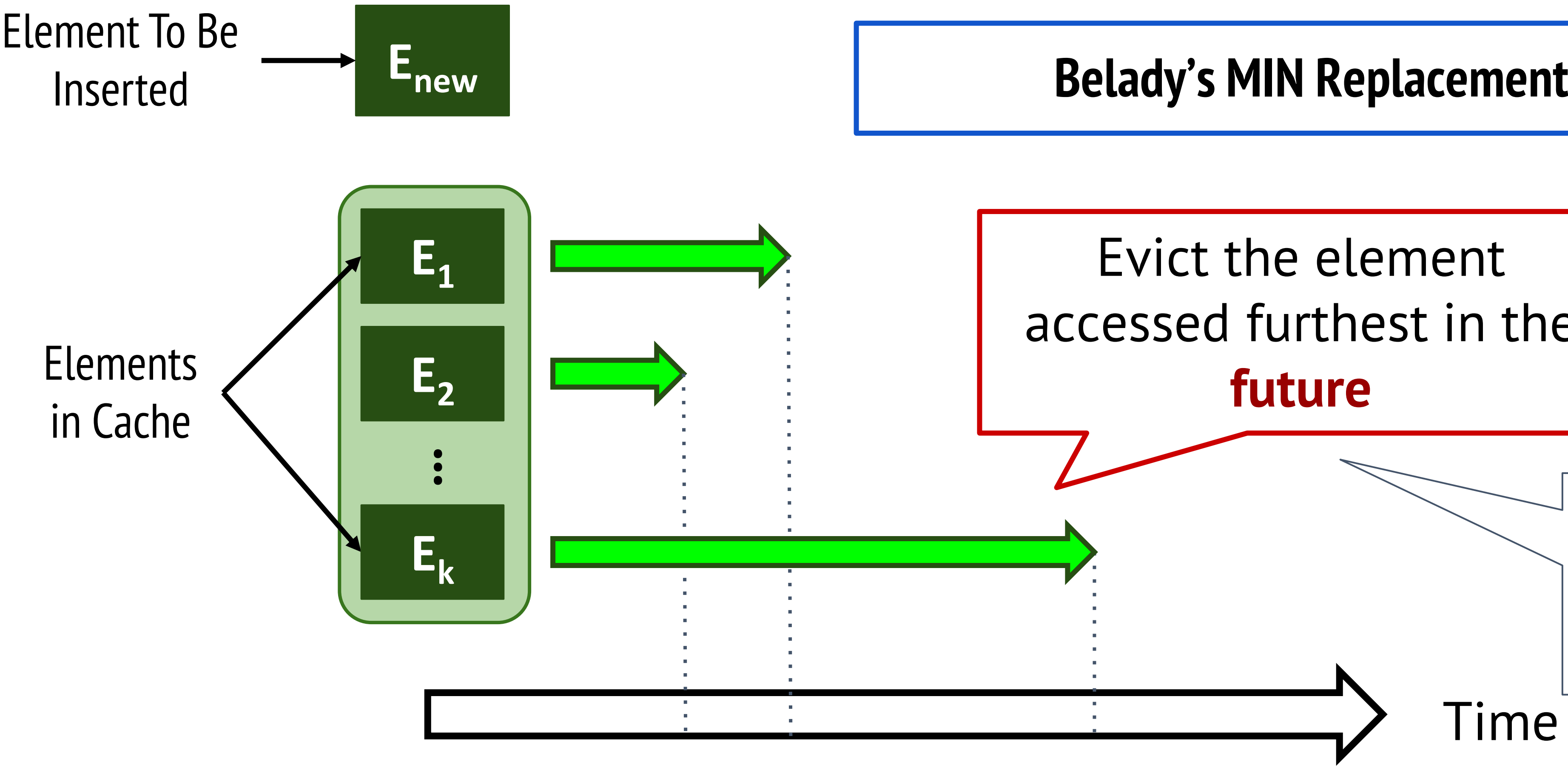


Belady's MIN Replacement Policy

Evict the element accessed furthest in the **future**

Recall: Prescience required. Problem?

Is It Possible To Do Better Cache Replacement? YES!



Belady's MIN Replacement Policy

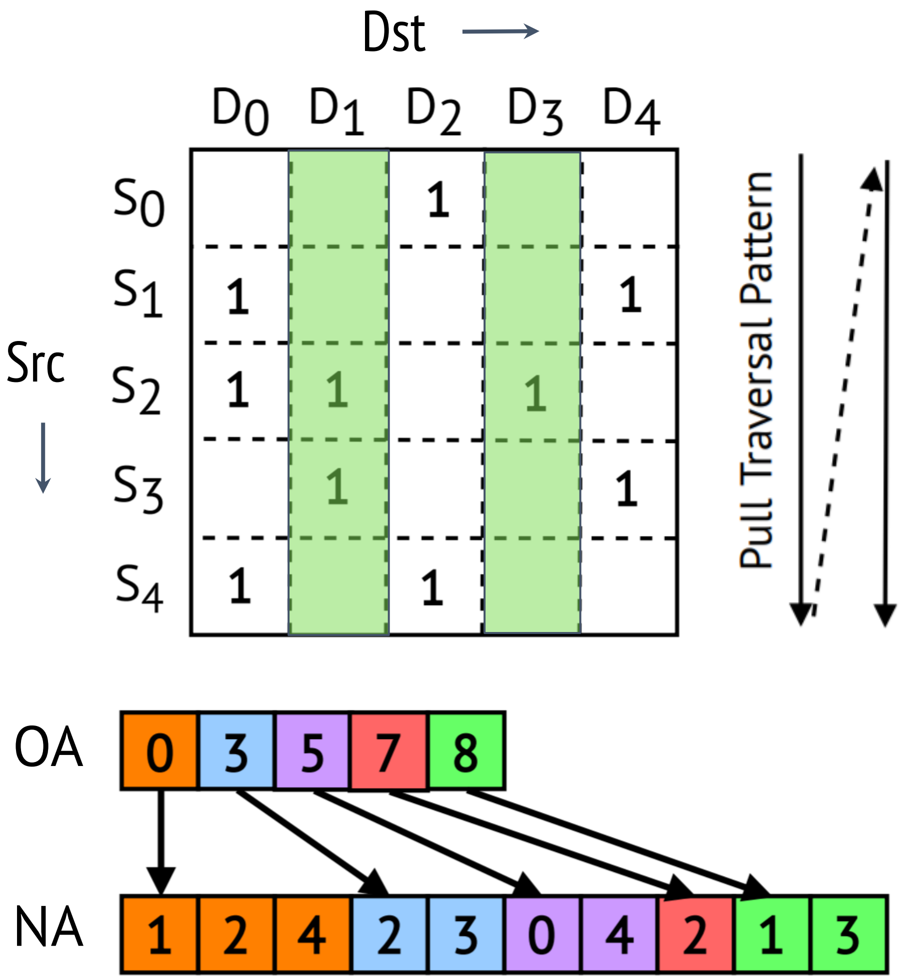
Evict the element accessed furthest in the **future**

Prescience is a flawed requirement: any hope?

Key Observation: The Graph's Transpose Efficiently Encodes Future Accesses

Key Graph Application Property That Enables Belady’s OPT

Key Graph Application Property That Enables Belady's OPT



	D0	D1	D2	D3	D4
S0			1		
S1	1				1
S2	1	1		1	
S3		1			1
S4	1		1		

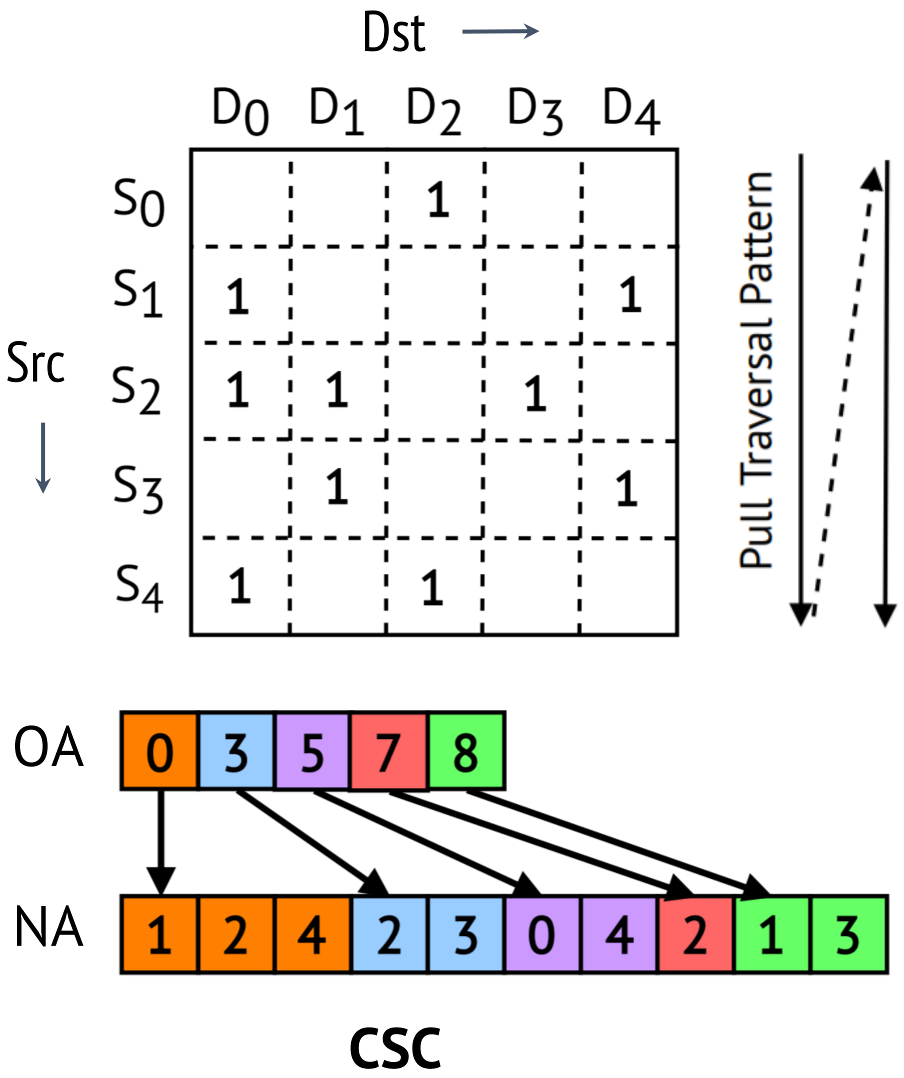
OA	0	3	5	7	8					
NA	1	2	4	2	3	0	4	2	1	3

CSC

Pull Execution (*CSC Traversal*)

```
for dst in G:
     for src in in_neighs(dst):
          dstData[dst] += srcData[src]
```

Key Graph Application Property That Enables Belady's OPT



	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		

OA	0	3	5	7	8					
NA	1	2	4	2	3	0	4	2	1	3

Pull Execution (*CSC Traversal*)

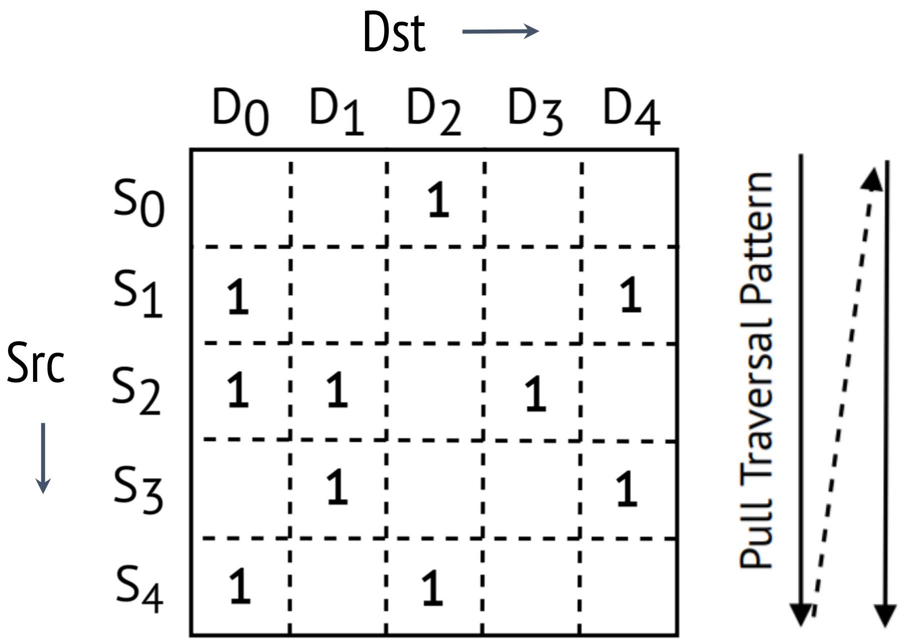
```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

CurrDs	**Irregular Data Stream**
tD_0	srcData[S_1]
D_0	srcData[S_2]
D_0	srcData[S_4]
D_1	srcData[S_2]
D_1	srcData[S_3]

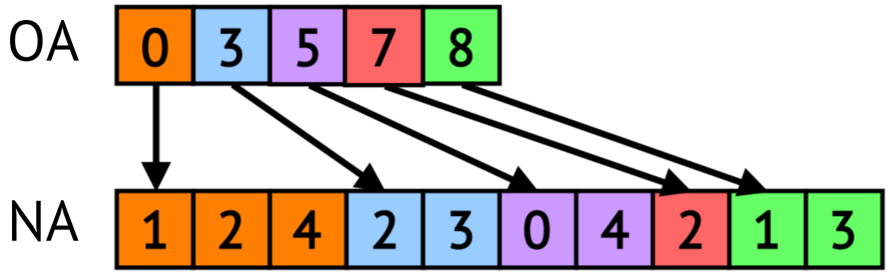
⋮

Time ↓

Key Graph Application Property That Enables Belady's OPT



	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		



OA	0	3	5	7	8					
NA	1	2	4	2	3	0	4	2	1	3

CSC

Pull Execution (*CSC Traversal*)

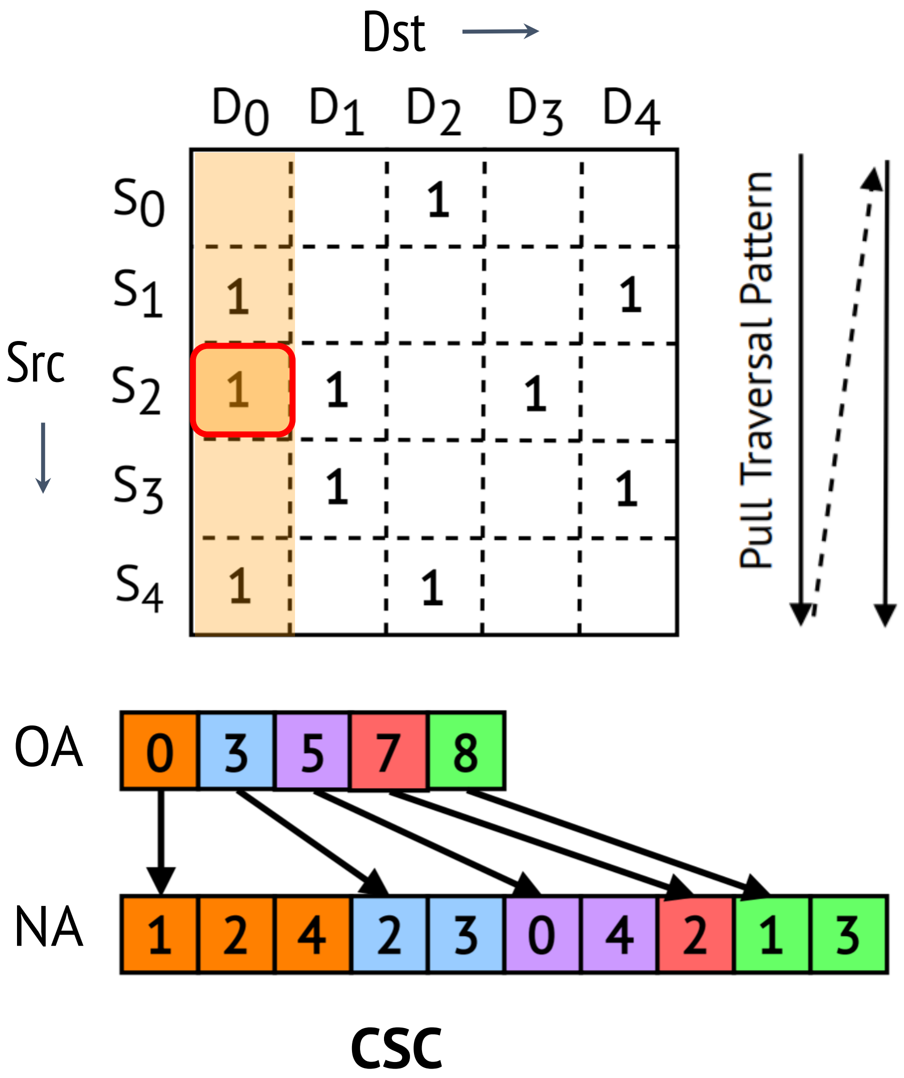
```
for dst in G:
     for src in in_neighs(dst):
          dstData[dst] += srcData[src]
```

Key Property: Dst-IDs are like timestamps for irregular accesses

CurrDs	**Irregular Data Stream**
tD_0	srcData[S_1]
D_0	srcData[S_2]
D_0	srcData[S_4]
D_1	srcData[S_2]
D_1	srcData[S_3]
	⋮

Time ↓

Key Graph Application Property That Enables Belady's OPT



	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		

OA: 0 3 5 7 8

NA: 1 2 4 2 3 0 4 2 1 3

Pull Execution (*CSC Traversal*)

```
for dst in G:
     for src in in_neighs(dst):
          dstData[dst] += srcData[src]
```

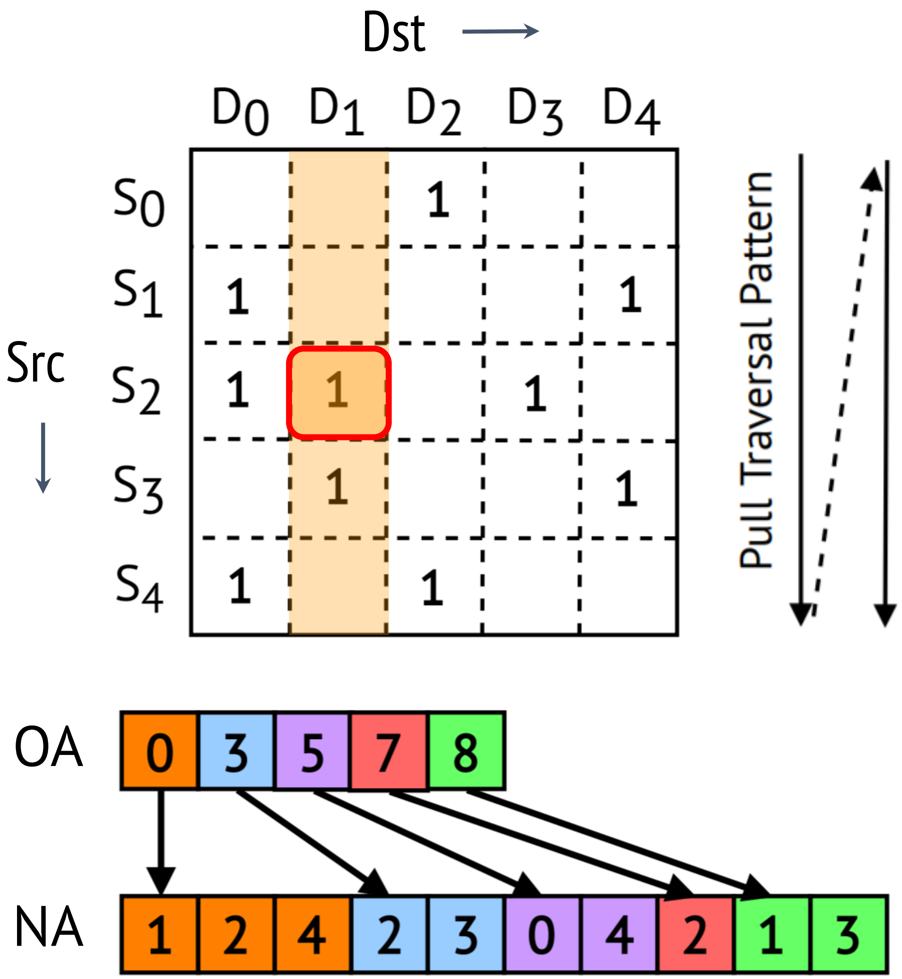
Key Property: Dst-IDs are like timestamps for irregular accesses

CurrDs	**Irregular Data Stream**
t D_0	srcData[S_1]
D_0	**srcData[S_2]**
D_0	srcData[S_4]
D_1	**srcData[S_2]**
D_1	srcData[S_3]

⋮

Time ↓

Key Graph Application Property That Enables Belady's OPT



	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		

OA	0	3	5	7	8					
NA	1	2	4	2	3	0	4	2	1	3

CSC

Pull Execution (*CSC Traversal*)

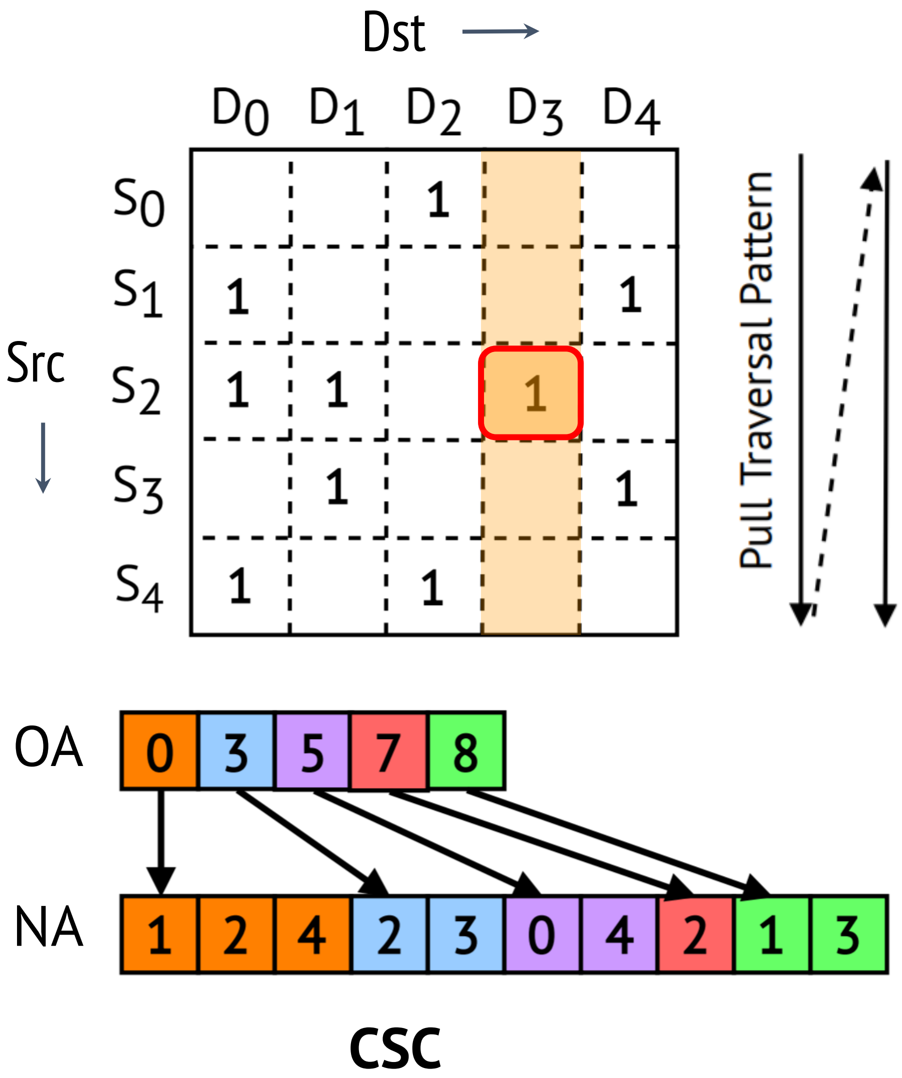
```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

Key Property: Dst-IDs are like timestamps for irregular accesses

CurrDs	**Irregular Data Stream**
tD_0	srcData[S_1]
D_0	**srcData[S_2]**
D_0	srcData[S_4]
D_1	**srcData[S_2]**
D_1	srcData[S_3]
	⋮

Time ↓

Key Graph Application Property That Enables Belady’s OPT



	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		

OA	0	3	5	7	8					
NA	1	2	4	2	3	0	4	2	1	3

Pull Execution (*CSC Traversal*)

```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

Key Property: Dst-IDs are like timestamps for irregular accesses

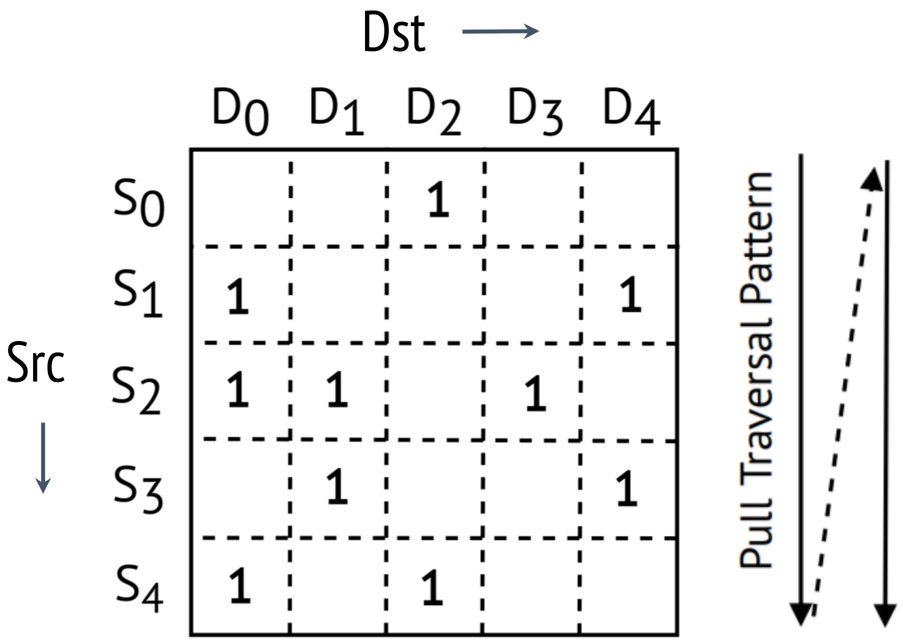
srcData[S_2] is accessed at $D_0 \Rightarrow D_1 \Rightarrow D_3$

CurrDs	**Irregular Data Stream**
tD_0	srcData[S_1]
D_0	**srcData[S_2]**
D_0	srcData[S_4]
D_1	**srcData[S_2]**
D_1	srcData[S_3]
	⋮

Time ↓

Using The Graph's Transpose For Optimal Replacement

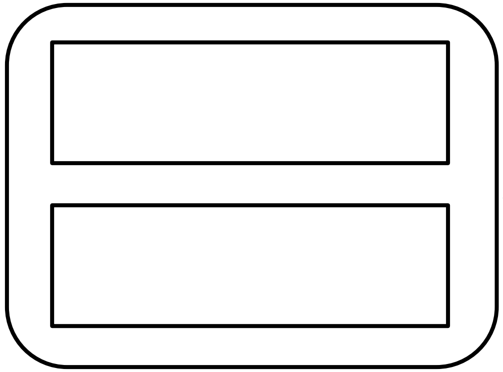
Using The Graph’s Transpose For Optimal Replacement



	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		

Pull Execution (*CSC Traversal*)

```
for dst in G:
     for src in in_neighs(dst):
          dstData[dst] += srcData[src]
```



2-way Set-Associative

Assumptions:

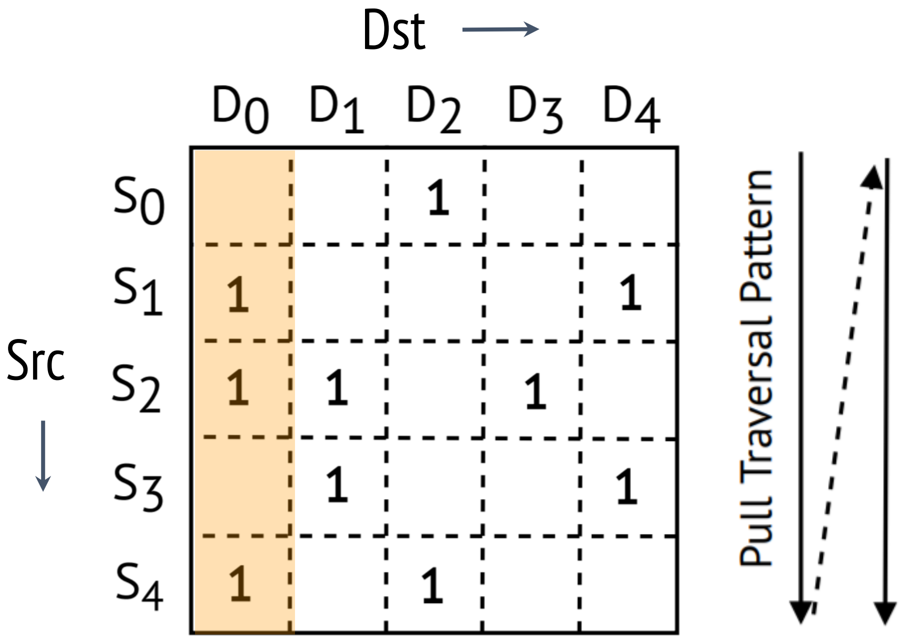
1. One srcData elem per line
2. Only irregular data enters the cache

CurrDs	**Irregular Data Stream**
tD_0	srcData[S_1]
D_0	srcData[S_2]
D_0	srcData[S_4]
D_1	srcData[S_2]
D_1	srcData[S_3]

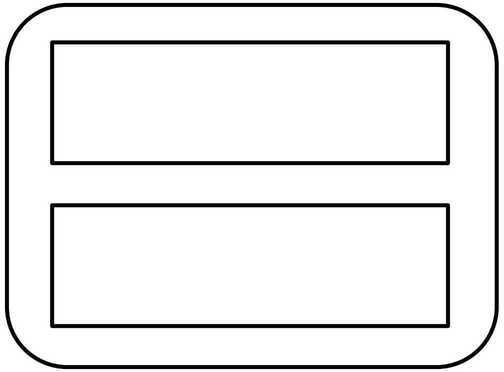
⋮

Time ↓

Using The Graph's Transpose For Optimal Replacement



	D0	D1	D2	D3	D4
S0			1		
S1	1				1
S2	1	1		1	
S3		1			1
S4	1		1		



2-way Set-Associative

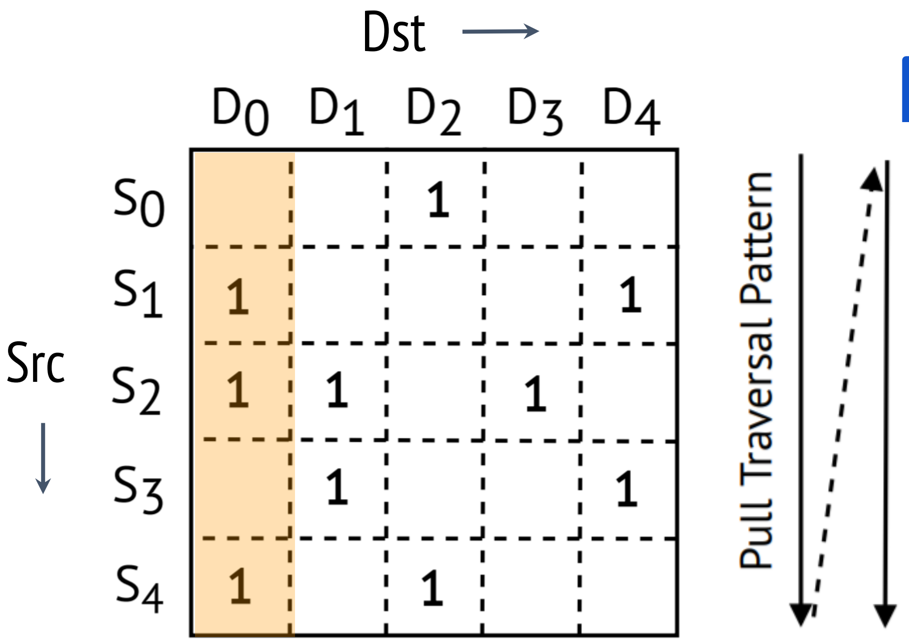
Pull Execution (*CSC Traversal*)

```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

CurrDs	Irregular Data Stream
$^{t}D_0$	srcData[S_1]
D_0	srcData[S_2]
D_0	srcData[S_4]
D_1	srcData[S_2]
D_1	srcData[S_3]
	⋮

Time

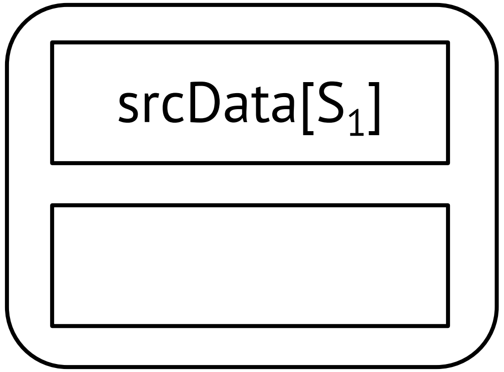
Using The Graph’s Transpose For Optimal Replacement



	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		

Pull Execution (*CSC Traversal*)

```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```



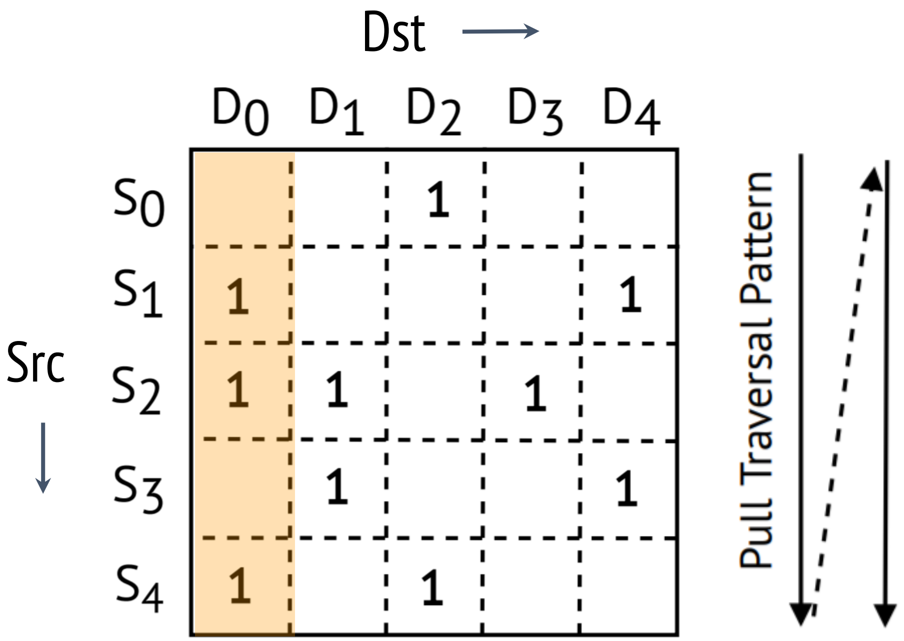
2-way Set-Associative

CurrDs	**Irregular Data Stream**
tD_0	srcData[S_1]
D_0	**srcData[S_2]**
D_0	srcData[S_4]
D_1	srcData[S_2]
D_1	srcData[S_3]

⋮

Time

Using The Graph's Transpose For Optimal Replacement



	D0	D1	D2	D3	D4
S0			1		
S1	1				1
S2	1	1		1	
S3		1			1
S4	1		1		

srcData[S_1]

srcData[S_2]

2-way Set-Associative

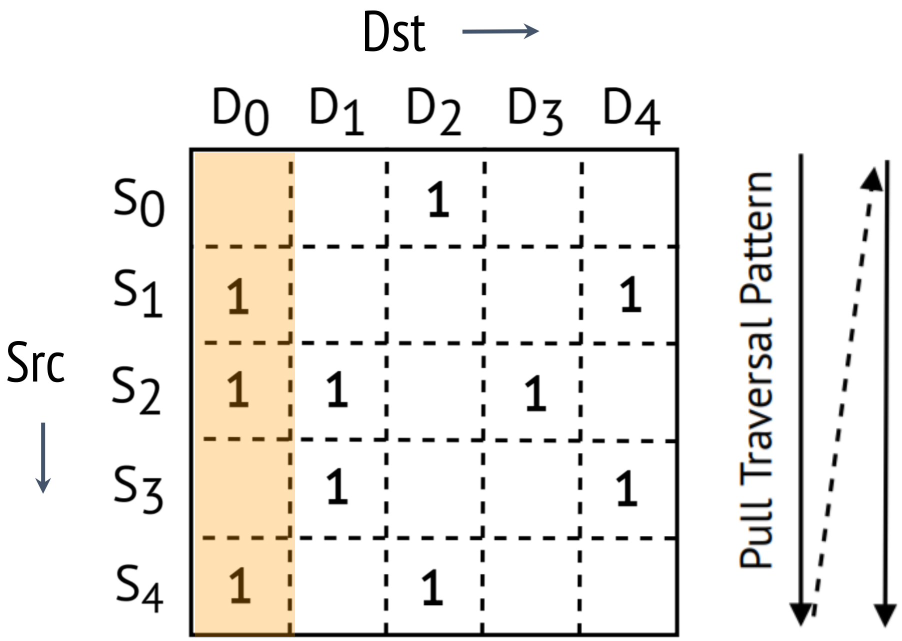
Pull Execution (*CSC Traversal*)

```
for dst in G:
     for src in in_neighs(dst):
          dstData[dst] += srcData[src]
```

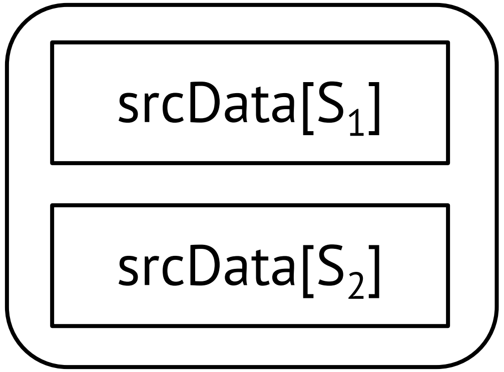
CurrDs	Irregular Data Stream
tD_0	srcData[S_1]
D_0	srcData[S_2]
D_0	**srcData[S_4]**
D_1	srcData[S_2]
D_1	srcData[S_3]
	⋮

Time ↓

Using The Graph's Transpose For Optimal Replacement



	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		



2-way Set-Associative

Pull Execution (*CSC Traversal*)

```
for dst in G:
     for src in in_neighs(dst):
          dstData[dst] += srcData[src]
```

Which line should we evict?:

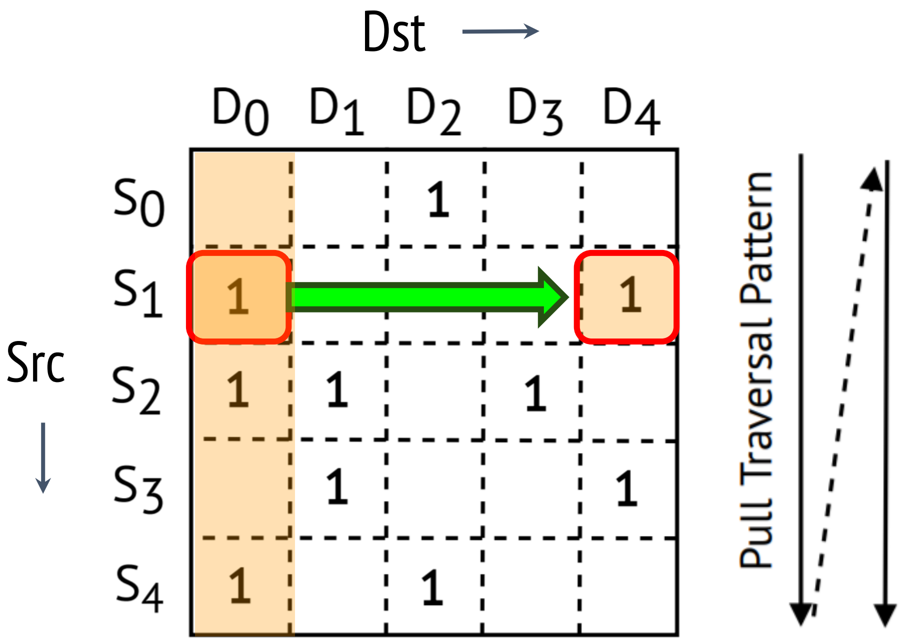
- srcData[S_1]
- srcData[S_2]

CurrDs	**Irregular Data Stream**
tD_0	srcData[S_1]
D_0	srcData[S_2]
D_0	**srcData[S_4]**
D_1	srcData[S_2]
D_1	srcData[S_3]

⋮

Time ↓

Using The Graph's Transpose For Optimal Replacement



	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		

Pull Execution (*CSC Traversal*)

```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

Which line should we evict?:

- srcData[S_1] **(nextRef @ D_4)**
- srcData[S_2]

srcData[S_1]

srcData[S_2]

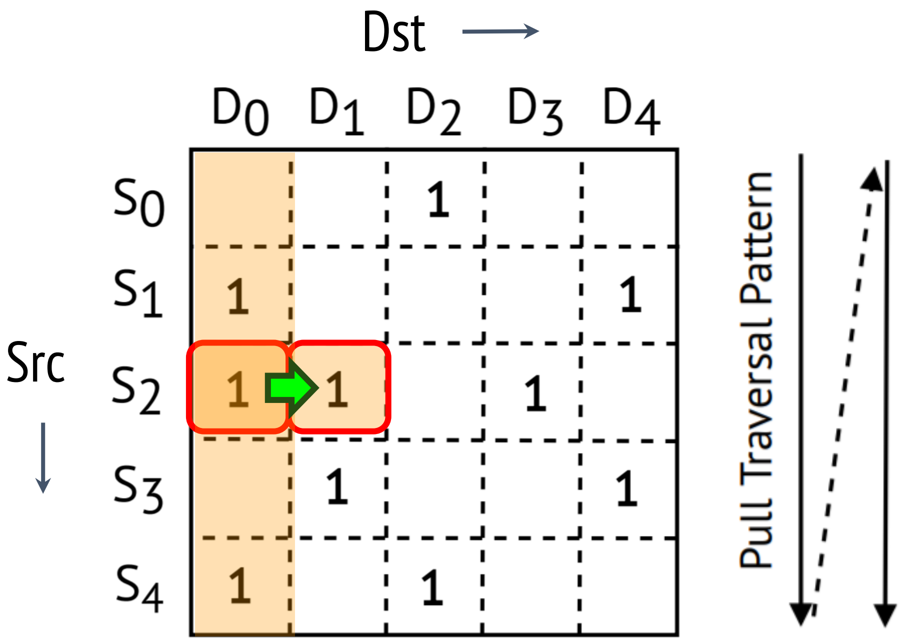
2-way Set-Associative

Out-Neigh(S_1): **D_0** | D_4 **CS R**

CurrDs	**Irregular Data Stream**
tD_0	srcData[S_1]
D_0	srcData[S_2]
D_0	**srcData[S_4]**
D_1	srcData[S_2]
D_1	srcData[S_3]
	⋮

Time ↓

Using The Graph's Transpose For Optimal Replacement



	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		

Pull Execution (*CSC Traversal*)

```
for dst in G:
    for src in in_neighs(dst):
        dstData[dst] += srcData[src]
```

srcData[S_1]

srcData[S_2]

2-way Set-Associative

Which line should we evict?:

- srcData[S_1] **(nextRef @ D_4)**
- srcData[S_2] **(nextRef @ D_1)**

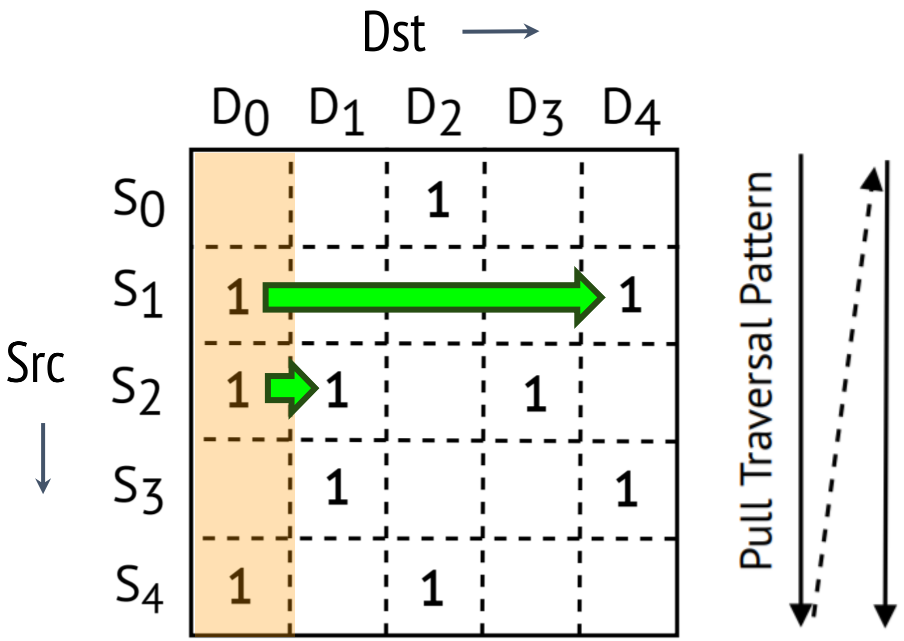
Out-Neigh(S_2): **D_0** | D_1 | D_3

CS
R

CurrDs	Irregular Data Stream
tD_0	srcData[S_1]
D_0	srcData[S_2]
D_0	**srcData[S_4]**
D_1	srcData[S_2]
D_1	srcData[S_3]
	⋮

Time ↓

Using The Graph's Transpose For Optimal Replacement

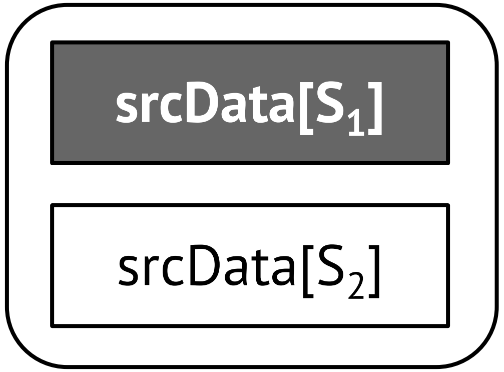


	D0	D1	D2	D3	D4
S0			1		
S1	1				1
S2	1	1		1	
S3		1			1
S4	1		1		

Pull Execution (*CSC Traversal*)

```
for dst in G:
     for src in in_neighs(dst):
          dstData[dst] += srcData[src]
```

Which line should we evict?:
- srcData[S_1] **(nextRef @ D_4)** ✓
- srcData[S_2] **(nextRef @ D_1)**


srcData[S_1]

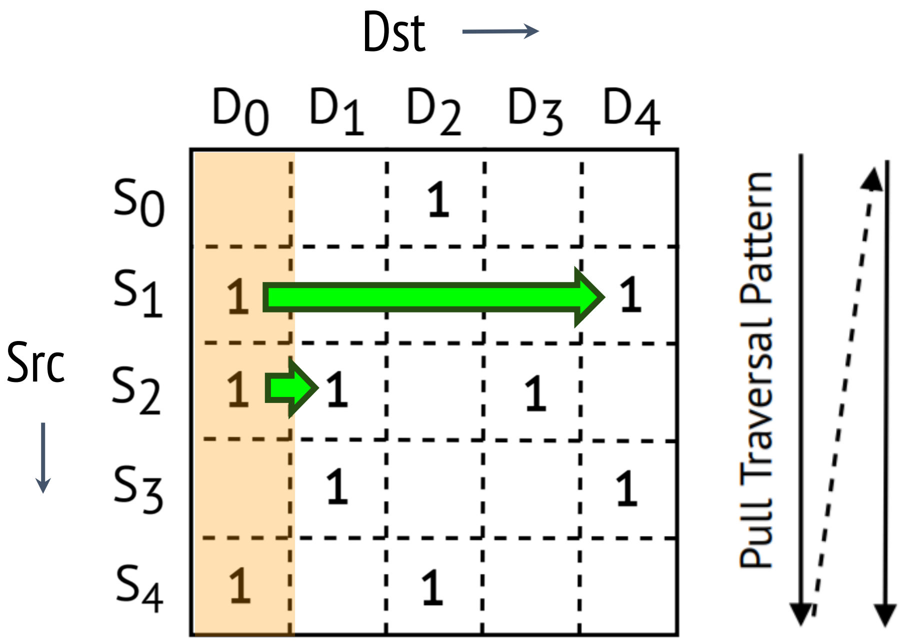
srcData[S_2]

2-way Set-Associative

CurrDs	**Irregular Data Stream**
t_{D_0}	srcData[S_1]
D_0	srcData[S_2]
D_0	**srcData[S_4]**
D_1	srcData[S_2]
D_1	srcData[S_3]
	⋮

Time

Using The Graph's Transpose For Optimal Replacement



	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		

Pull Execution (*CSC Traversal*)

```
for dst in G:
     for src in in_neighs(dst):
          dstData[dst] += srcData[src]
```

Which line should we evict?:

- srcData[S_1] **(nextRef @ D_4)** ✓
- srcData[S_2] **(nextRef @ D_1)**

srcData[S_1]

srcData[S_2]

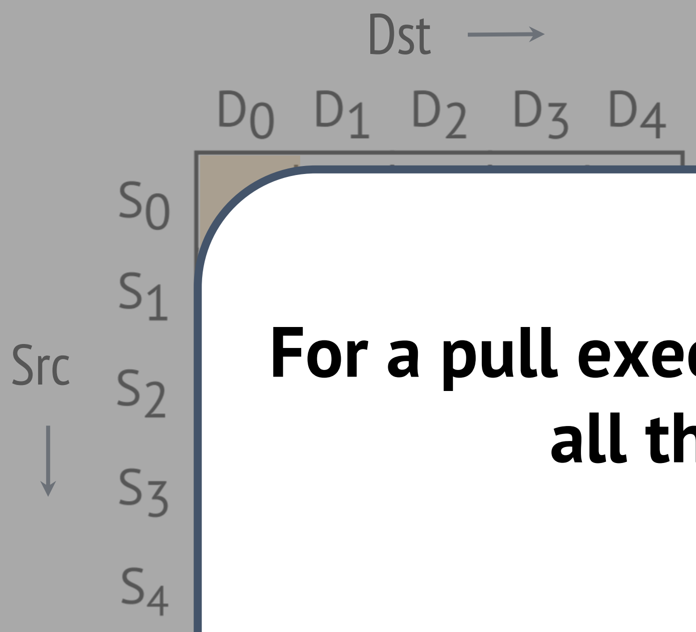
2-way Set-Associative

CurrDs	Irregular Data Stream
tD_0	srcData[S_1]
D_0	srcData[S_2]
D_0	**srcData[S_4]**
D_1	srcData[S_2]
D_1	srcData[S_3]
	⋮

Time ↓

Key Question: how to query next reference while running the program?

Using The Graph’s Transpose For Optimal Replacement



Pull Execution (*CSC Traversal*)

```
for dst in G:
```

For a pull execution (CSC-traversal), the transpose (CSR) contains all the necessary OPT replacement information

ime

- srcData[S_2] **(nextRef @ D_1)**

D_1	srcData[S_2]
D_1	srcData[S_3]

⋮

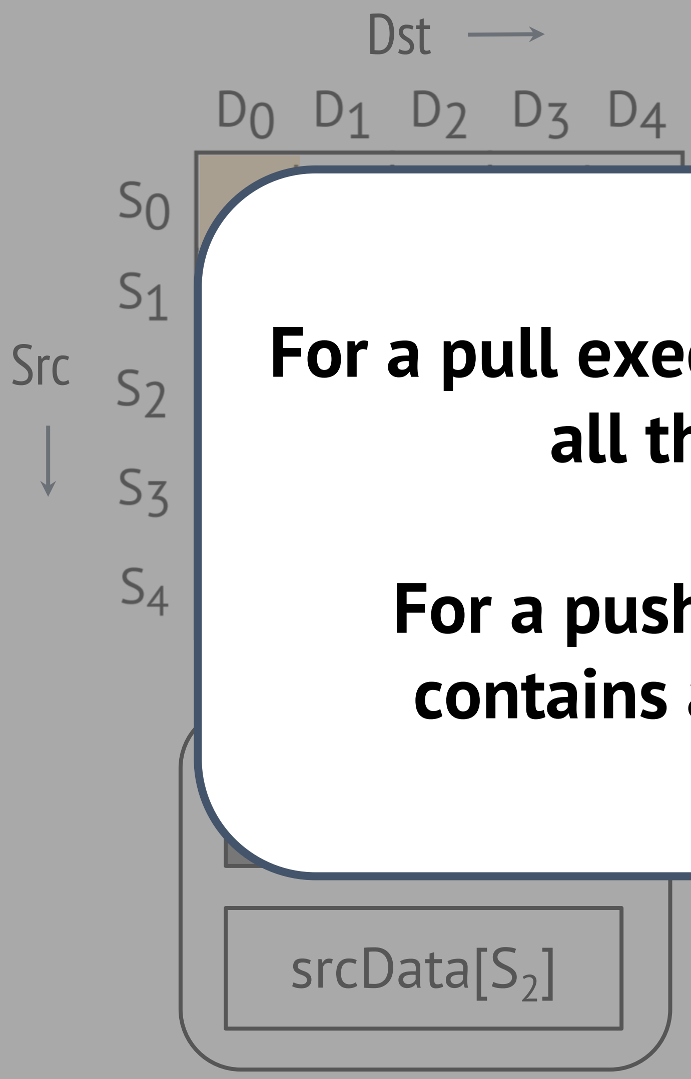
srcData[S_2]

2-way Set-Associative

Using The Graph's Transpose For Optimal Replacement

For a pull execution (CSC-traversal), the transpose (CSR) contains all the necessary OPT replacement information

For a push execution (CSR-traversal), the transpose (CSC) contains all the necessary OPT replacement information

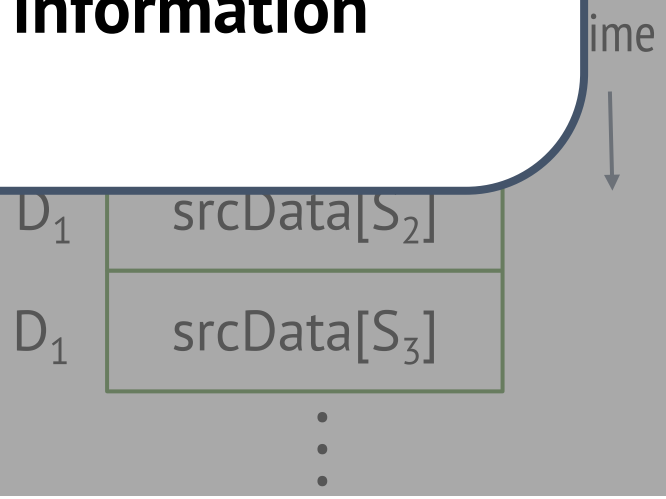


2-way Set-Associative

Pull Execution (*CSC Traversal*)

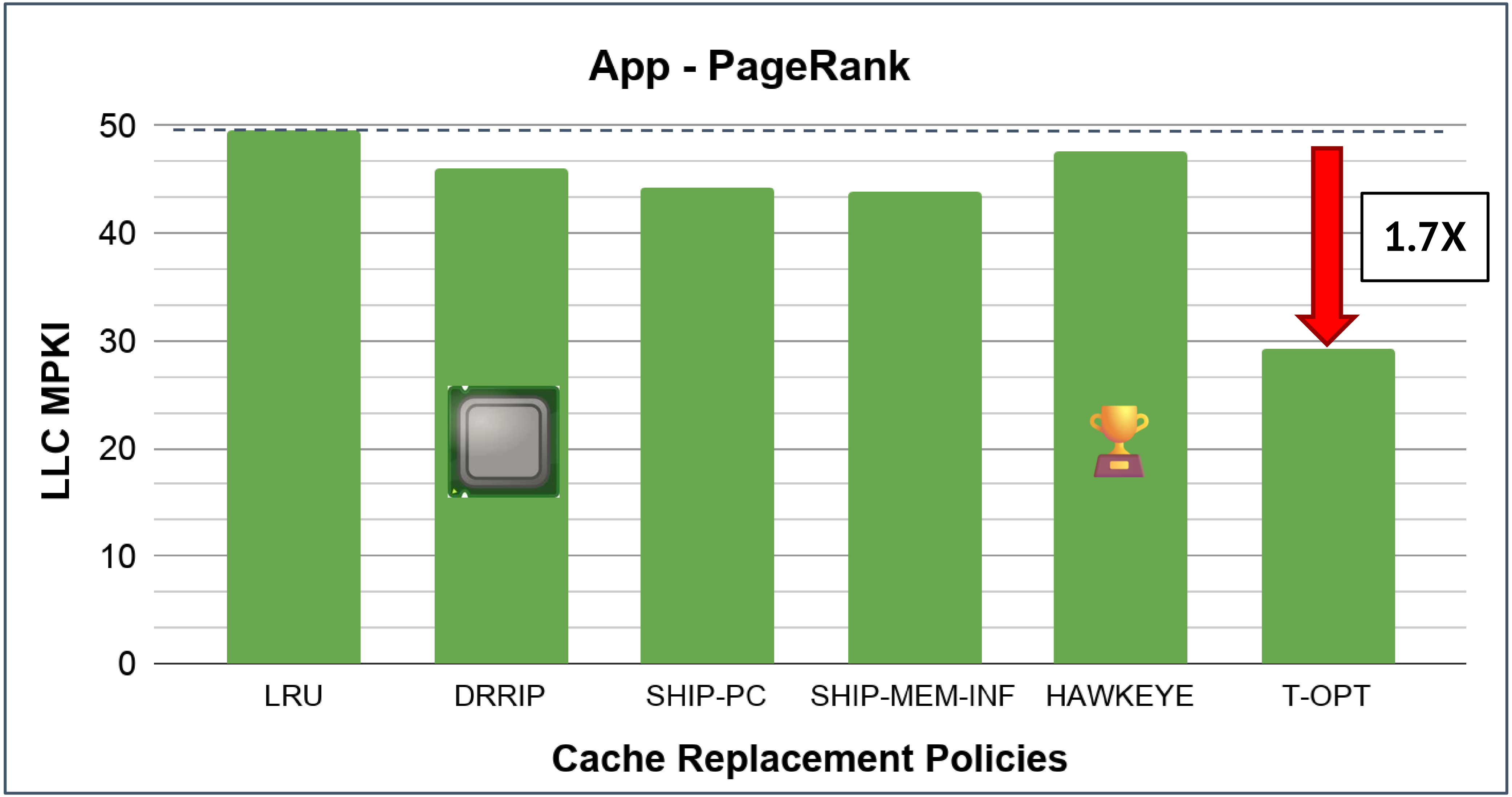
```
for dst in G:
```

- srcData[S_2] **(nextRef @ D_1)**

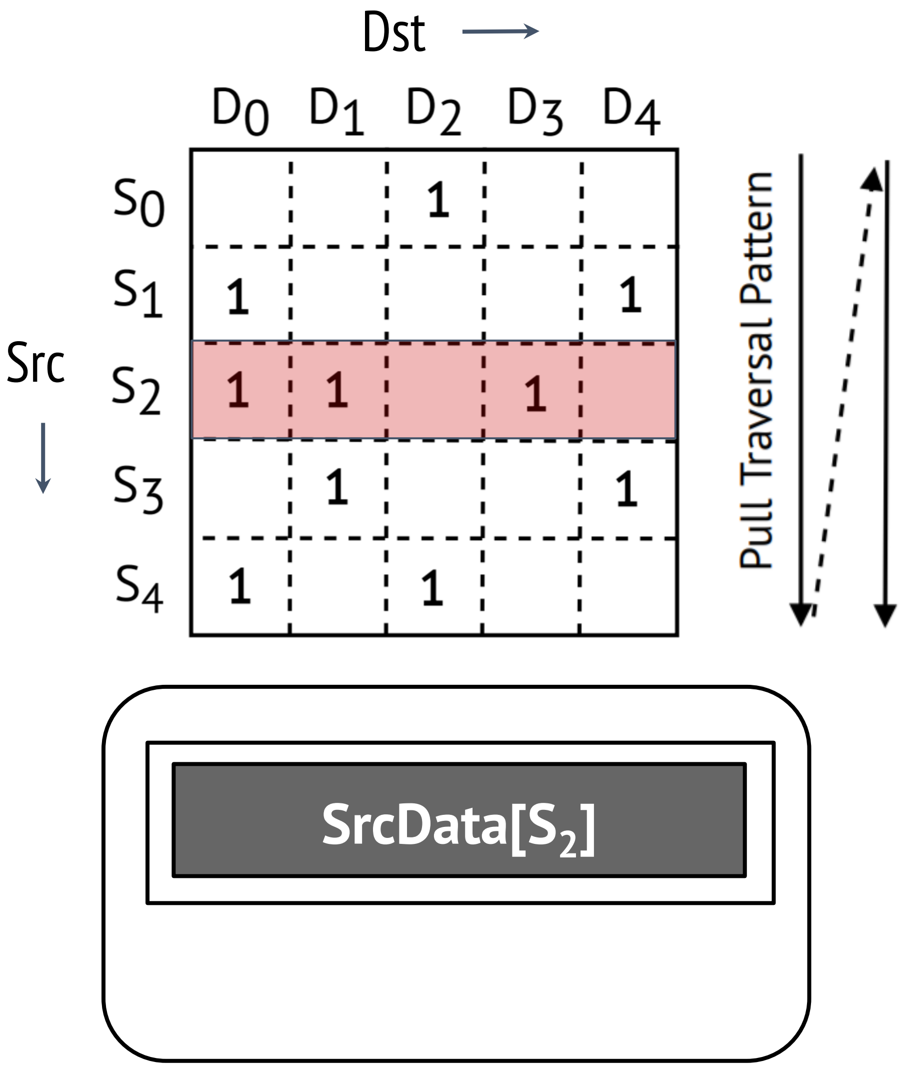


Transpose-based OPT (T-OPT) Provides Large Gains

Transpose-based OPT (T-OPT) Provides Large Gains



Transpose-based OPT (T-OPT) Incurs Overheads

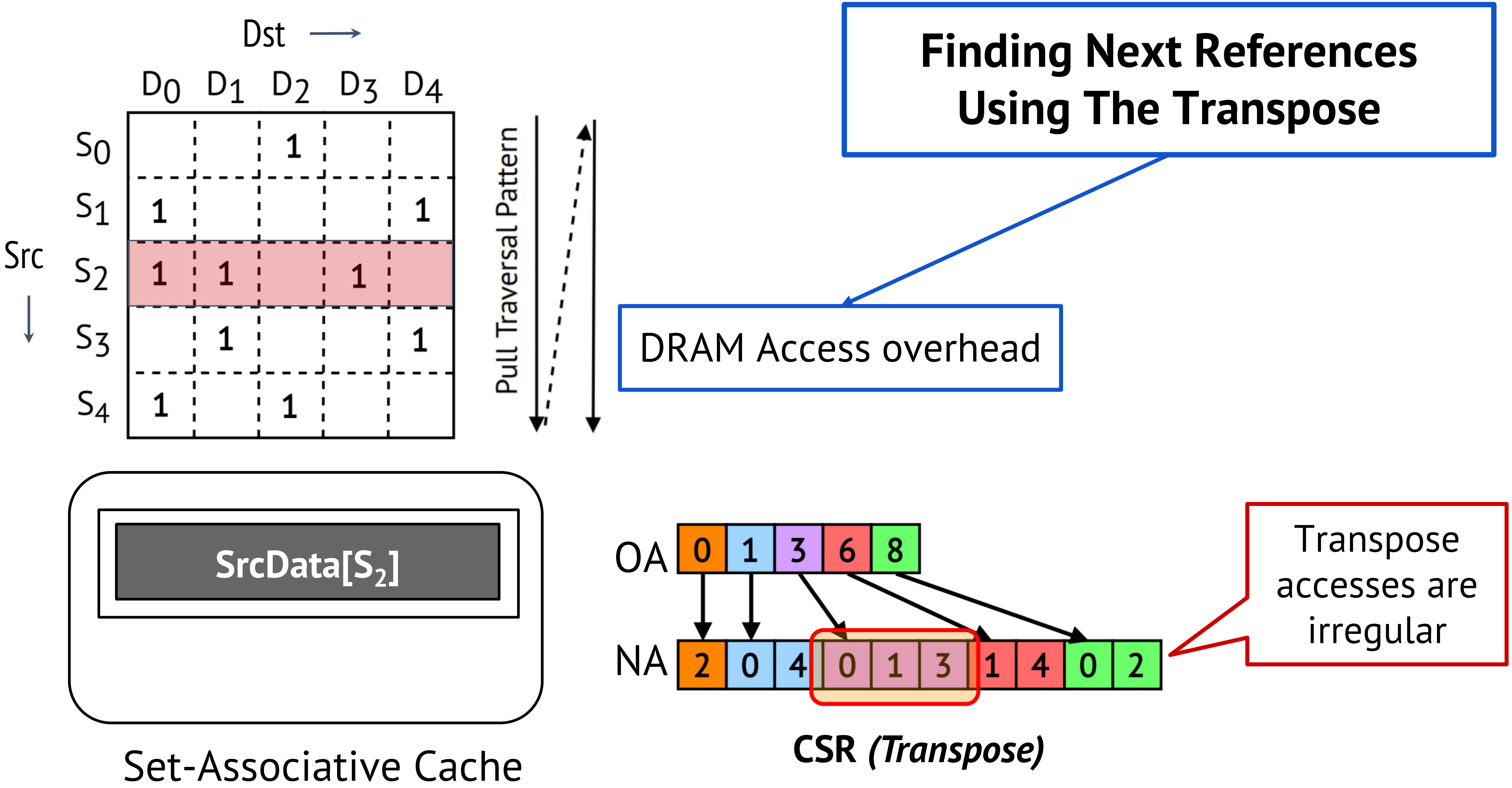


	D_0	D_1	D_2	D_3	D_4
S_0			1		
S_1	1				1
S_2	1	1		1	
S_3		1			1
S_4	1		1		

Set-Associative Cache

Finding Next References Using The Transpose

Transpose-based OPT (T-OPT) Incurs Overheads



	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		

Finding Next References Using The Transpose

DRAM Access overhead

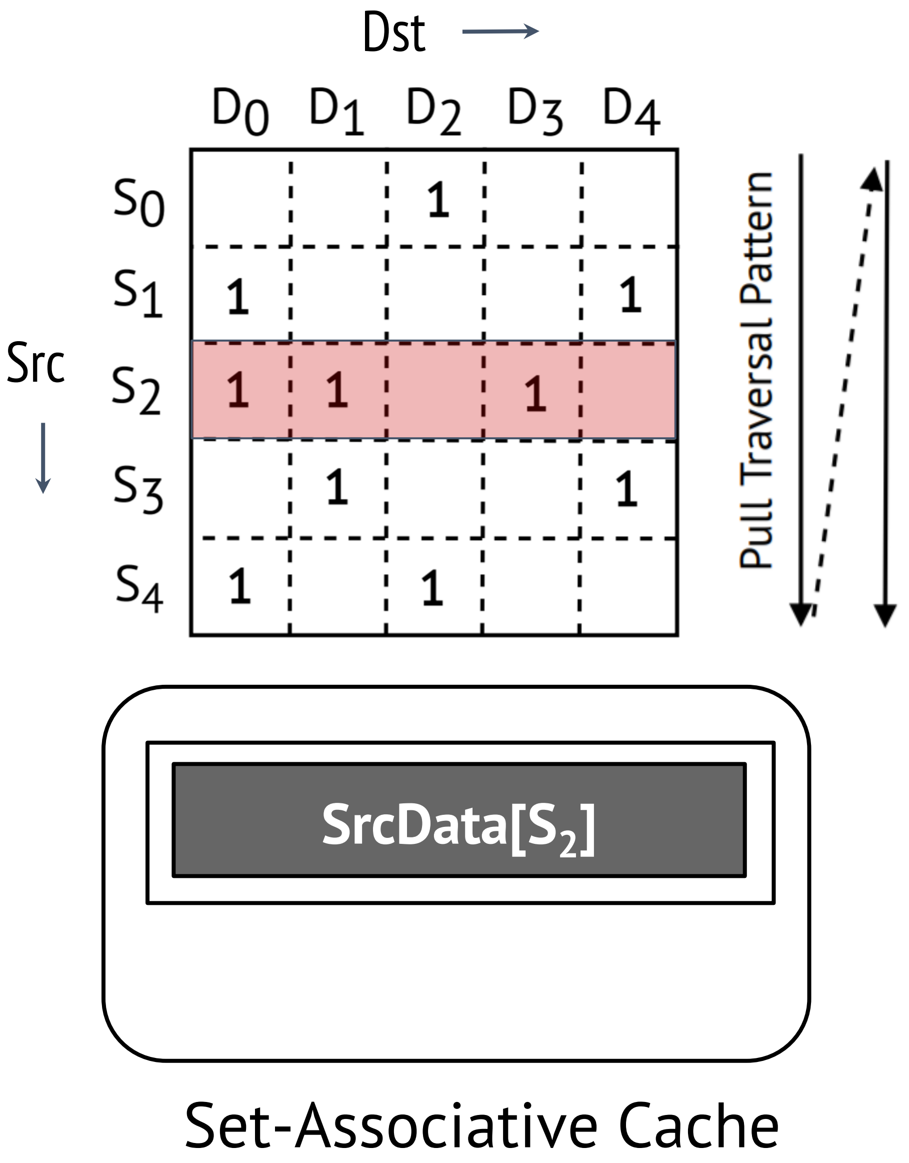
OA	0	1	3	6	8					
NA	2	0	4	0	1	3	1	4	0	2

Transpose accesses are irregular

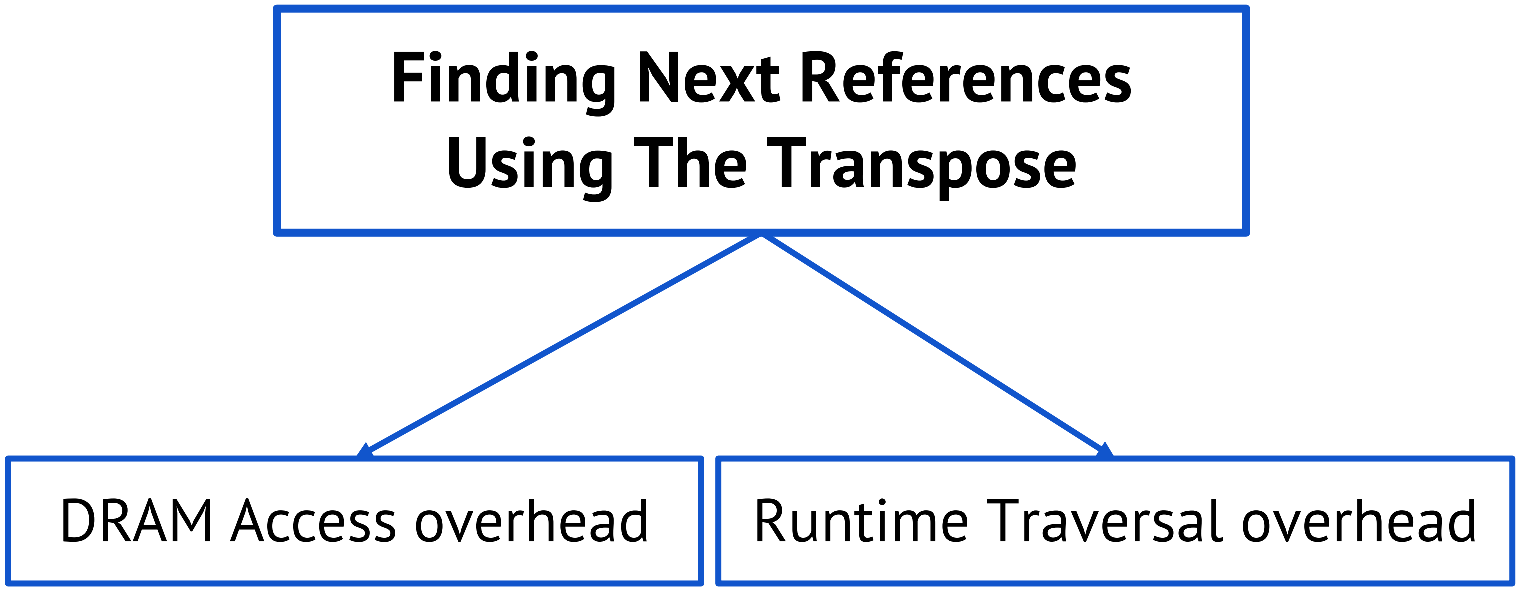
Set-Associative Cache

CSR *(Transpose)*

Transpose-based OPT (T-OPT) Incurs Overheads



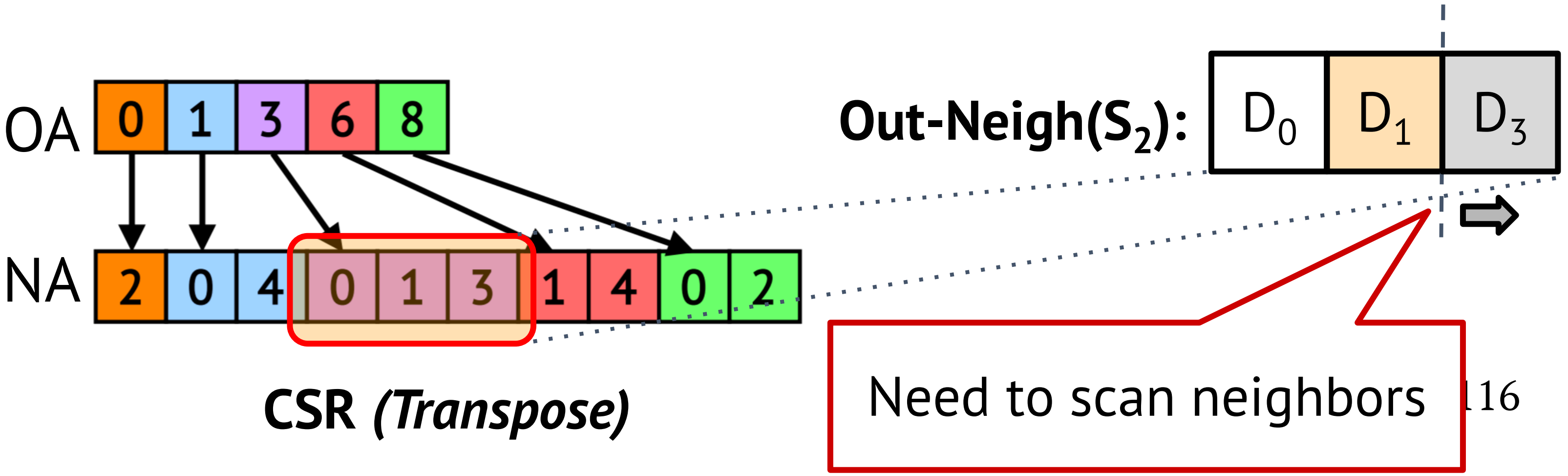
$$\begin{array}{c|ccccc} & D_0 & D_1 & D_2 & D_3 & D_4 \\ \hline S_0 & & & 1 & & \\ S_1 & 1 & & & & 1 \\ S_2 & 1 & 1 & & 1 & \\ S_3 & & 1 & & & 1 \\ S_4 & 1 & & 1 & & \end{array}$$



Finding Next References Using The Transpose

DRAM Access overhead

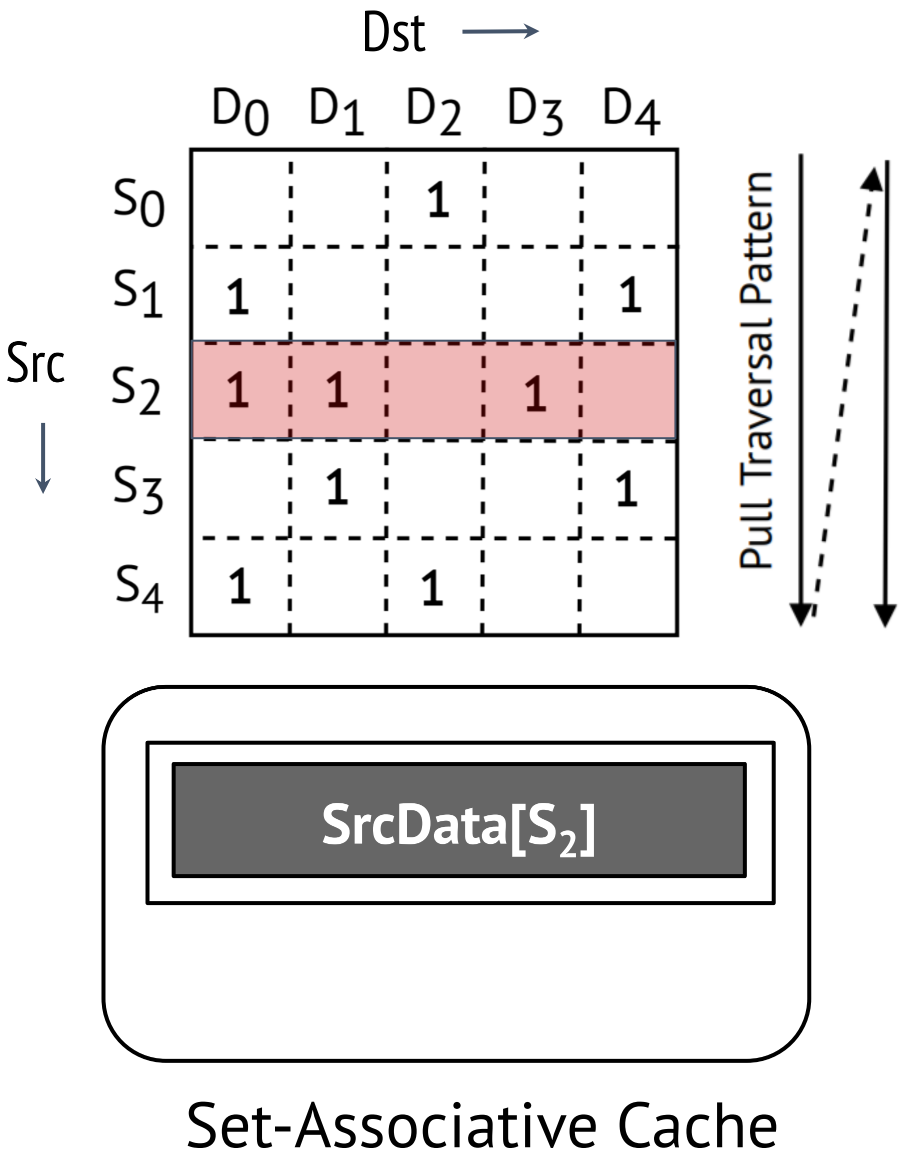
Runtime Traversal overhead



CSR *(Transpose)*

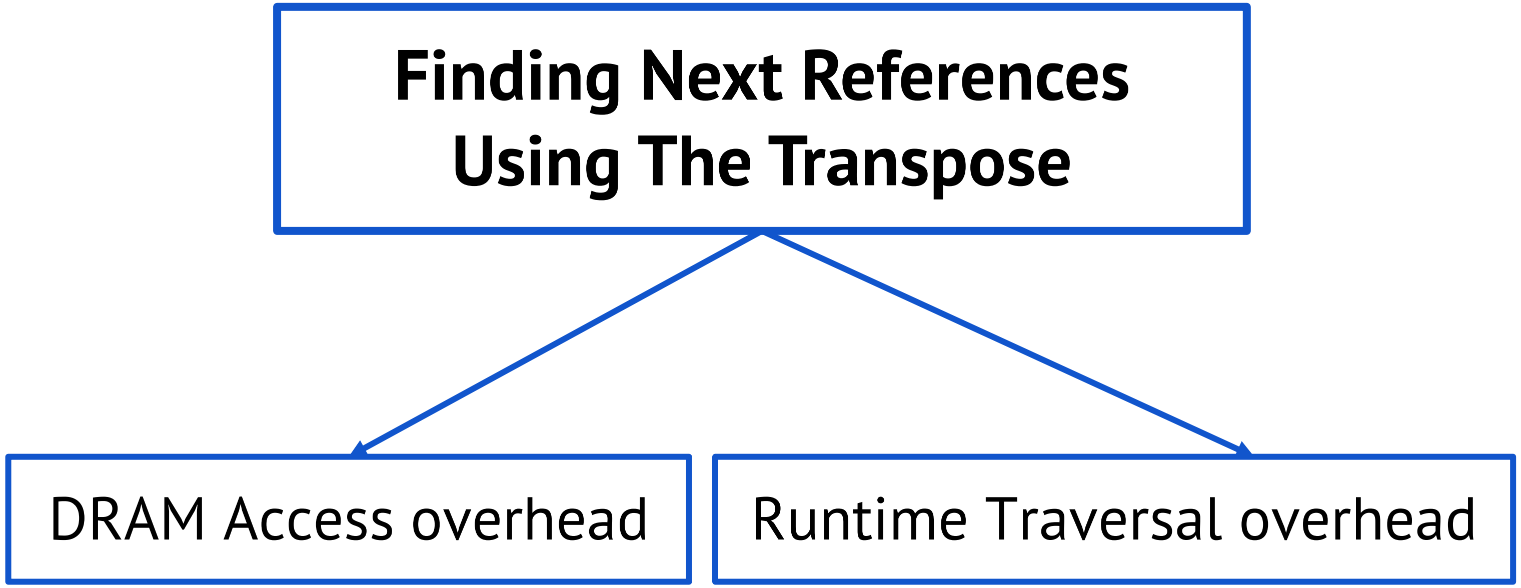
Need to scan neighbors

Transpose-based OPT (T-OPT) Incurs Overheads



	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		

Set-Associative Cache



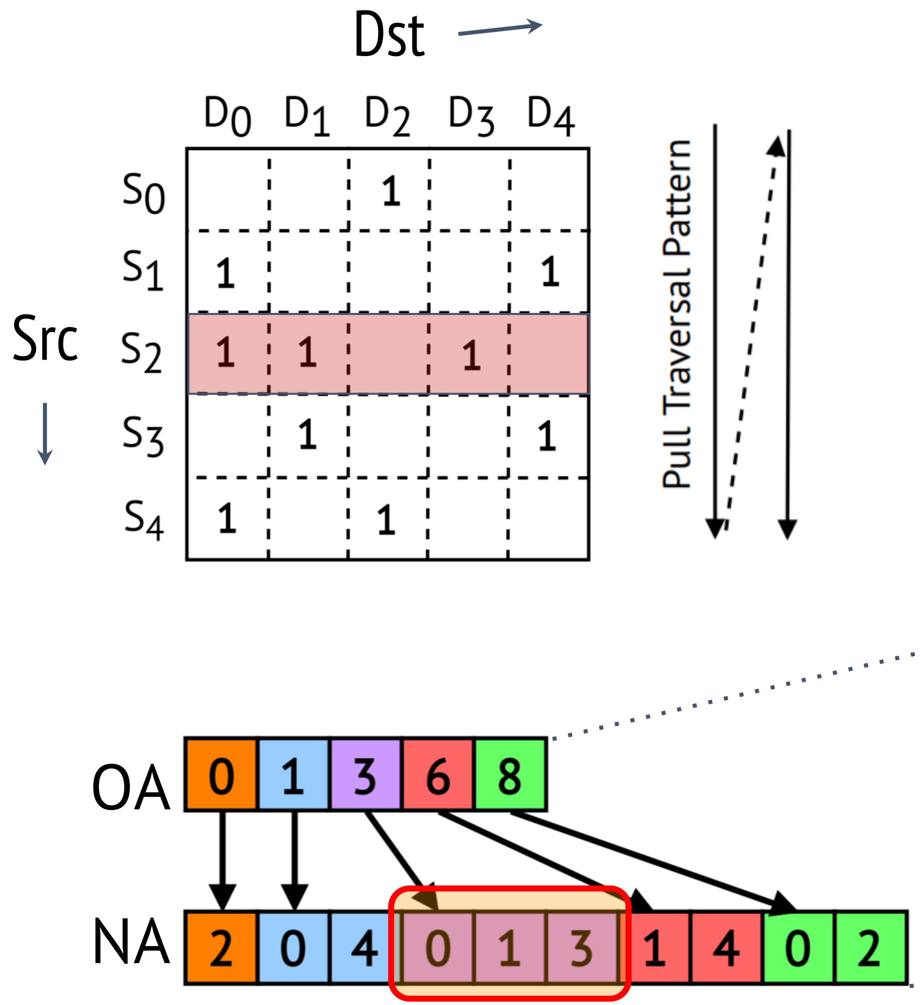
Finding Next References Using The Transpose

DRAM Access overhead

Runtime Traversal overhead

Question: How do we retrieve the next reference information from the graph’s transpose **without all the cost of traversing the graph?**

Main Technique: Use Quantization To Compress The Transpose



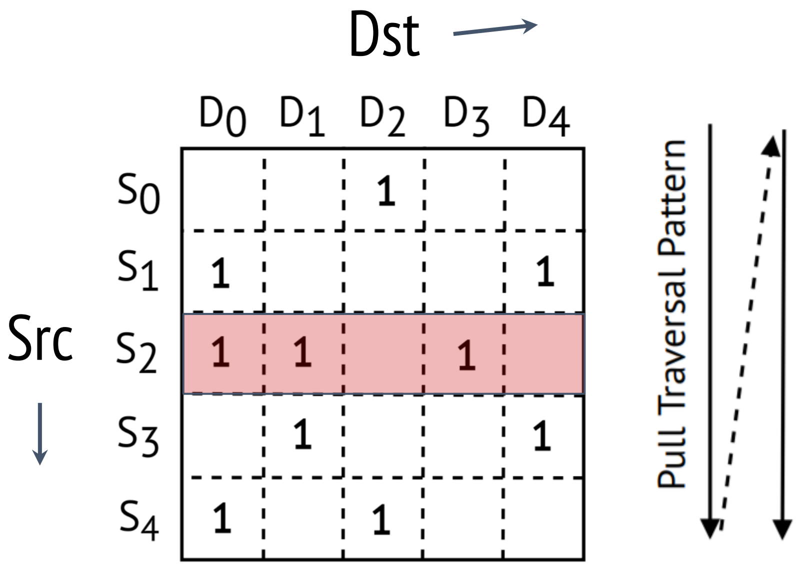
	D_0	D_1	D_2	D_3	D_4
S_0			1		
S_1	1				1
S_2	1	1		1	
S_3		1			1
S_4	1		1		

CSR
(Transpose)

srcData[S_2] Accessed at:

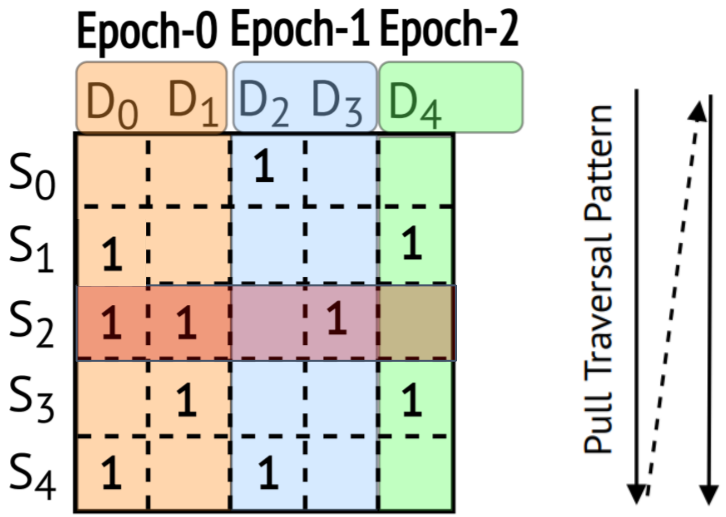
$D_0 \Rightarrow D_1 \Rightarrow D_3$

Main Technique: Use Quantization To Compress The Transpose

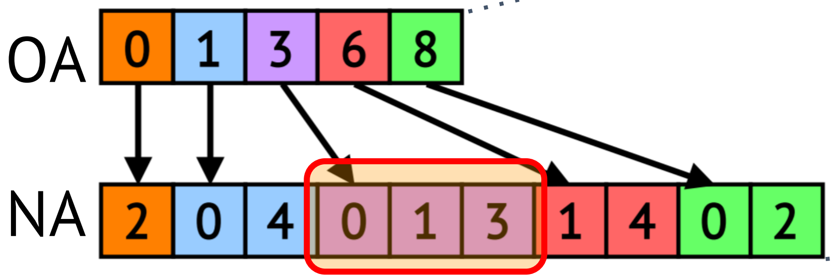


	D_0	D_1	D_2	D_3	D_4
S_0			1		
S_1	1				1
S_2	1	1		1	
S_3		1			1
S_4	1		1		

Divide execution into coarse-grained epochs



	Epoch-0		Epoch-1		Epoch-2
	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		



OA	0	1	3	6	8				
NA	2	0	4	0	1	3	1	4	0	2

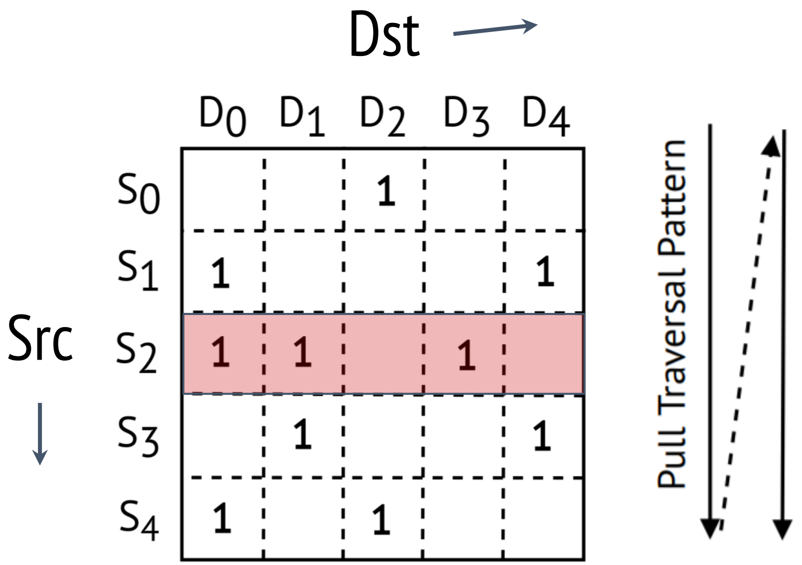
CSR

(Transpose)

srcData[S_2] Accessed at:

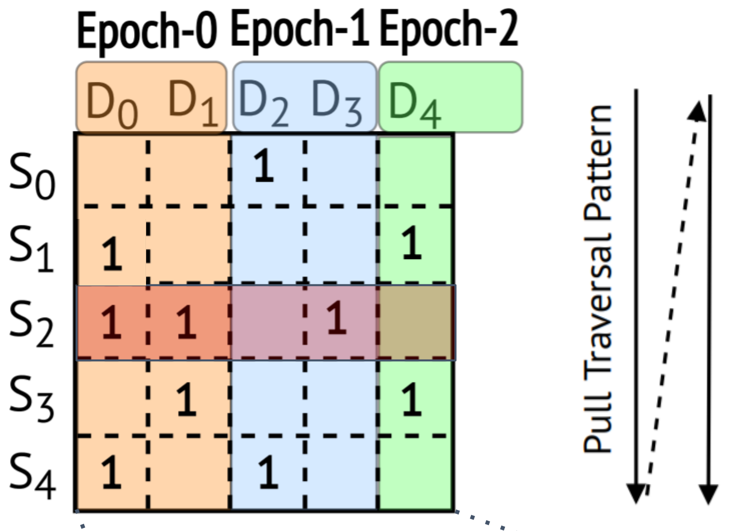
$D_0 \Rightarrow D_1 \Rightarrow D_3$

Main Technique: Use Quantization To Compress The Transpose

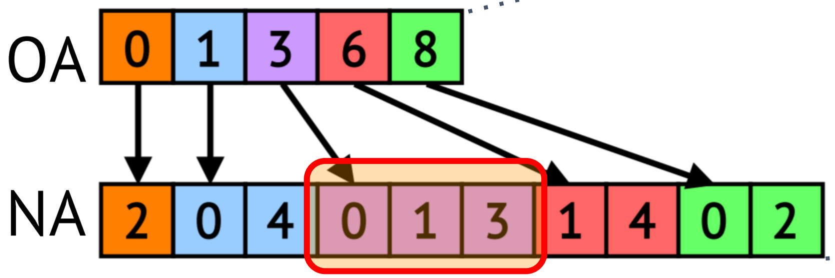


	D_0	D_1	D_2	D_3	D_4
S_0			1		
S_1	1				1
S_2	1	1		1	
S_3		1			1
S_4	1		1		

Divide execution into coarse-grained epochs



	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		



OA	0	1	3	6	8					
NA	2	0	4	0	1	3	1	4	0	2

CSR

(Transpose)

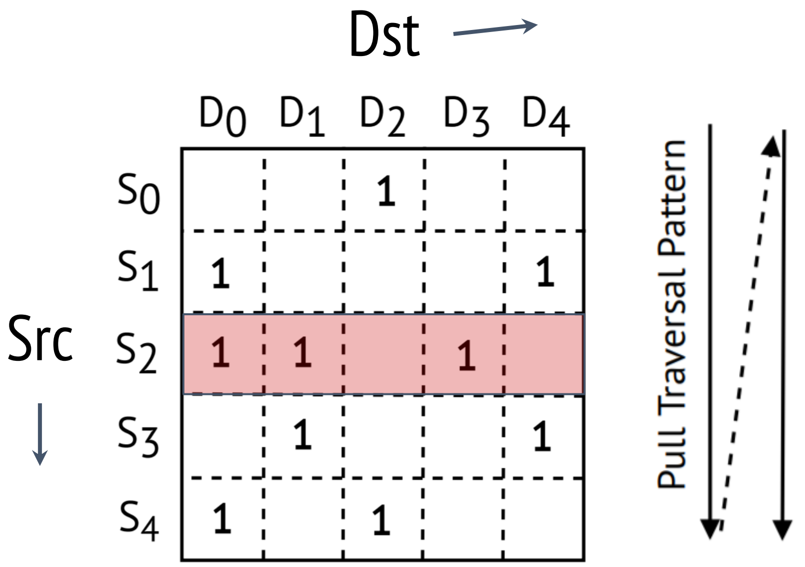
srcData[S_2] Accessed at:

$D_0 \Rightarrow D_1 \Rightarrow D_3$

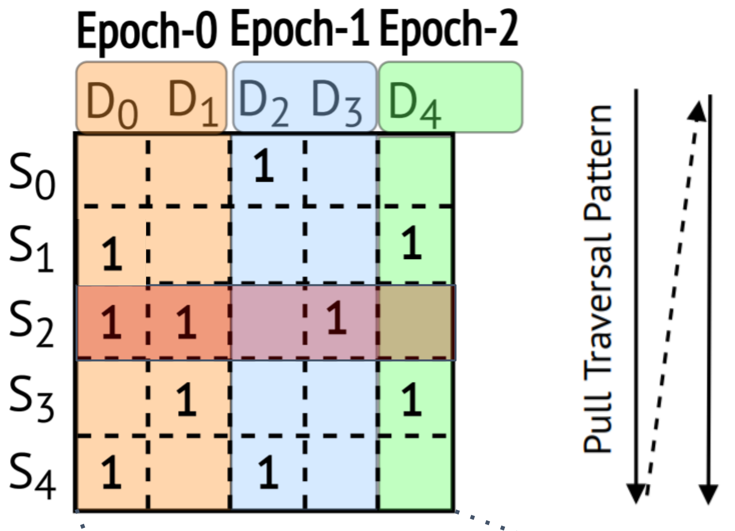
srcData[S_2] Accessed at:

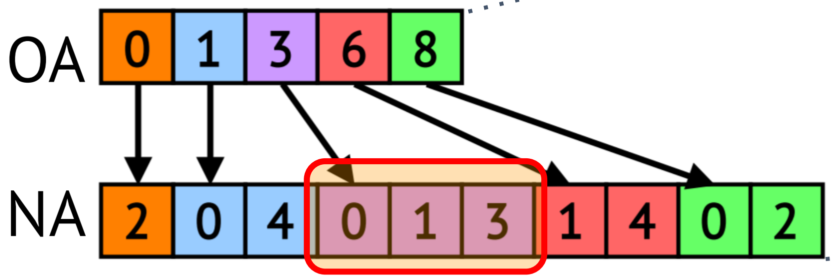
$E_0 \Rightarrow E_1$

Main Technique: Use Quantization To Compress The Transpose



Divide execution into coarse-grained epochs





CSR

(Transpose)

srcData[S_2] Accessed at:

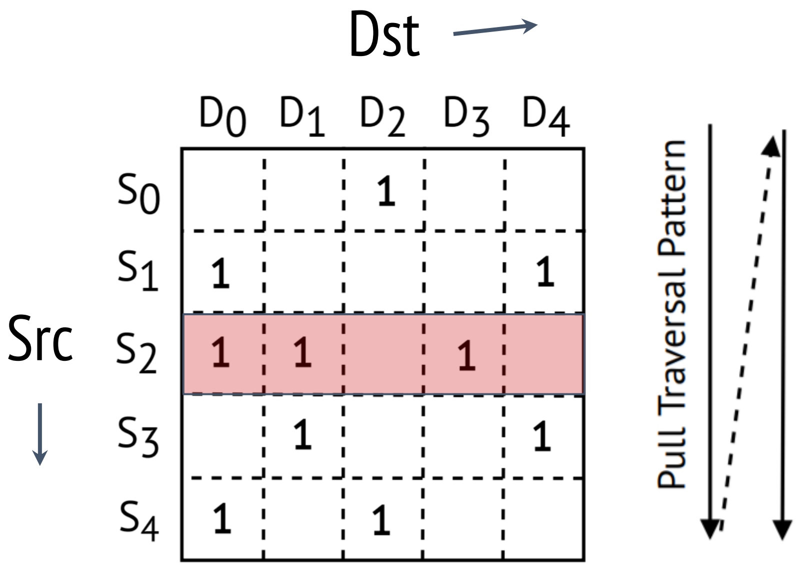
$D_0 \Rightarrow D_1 \Rightarrow D_3$

srcData[S_2] Accessed at:

$E_0 \Rightarrow E_1$

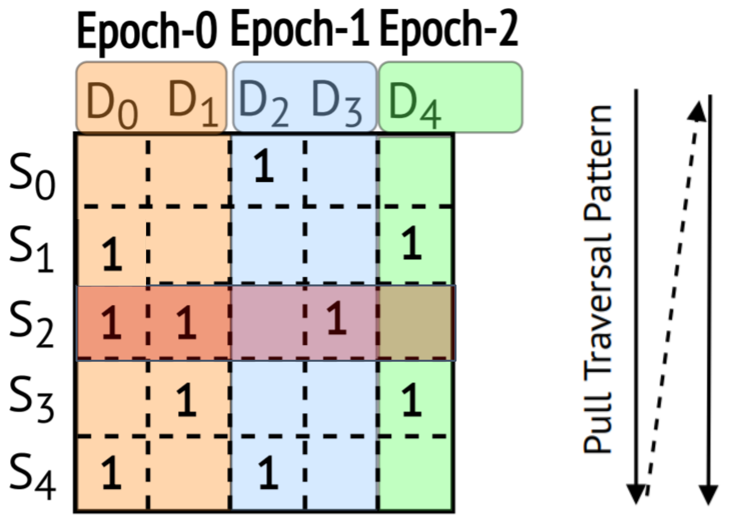
No. of epochs ⬇ ⇒ Next Ref Quantization ⬆
(accuracy vs metadata size trade-off)

Main Technique: Use Quantization To Compress The Transpose

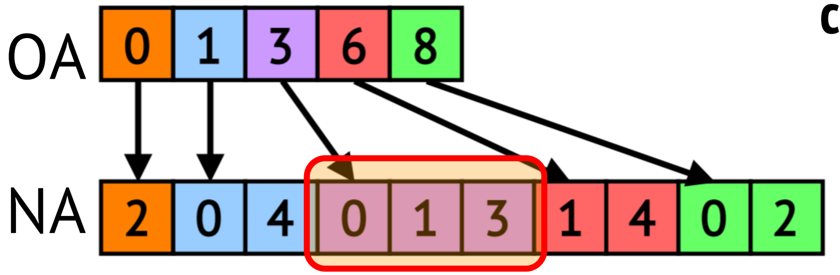


	D0	D1	D2	D3	D4
S0			1		
S1	1				1
S2	1	1		1	
S3		1			1
S4	1		1		

Divide execution into coarse-grained epochs

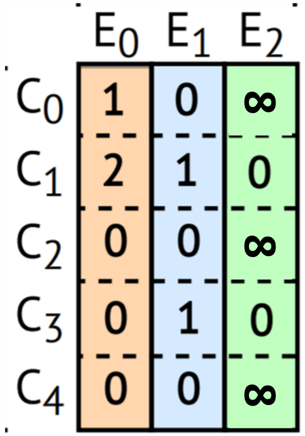


	D0	D1	D2	D3	D4
S0			1		
S1	1				1
S2	1	1		1	
S3		1			1
S4	1		1		



CSR
(Transpose)

Quantization enables compression of transpose data

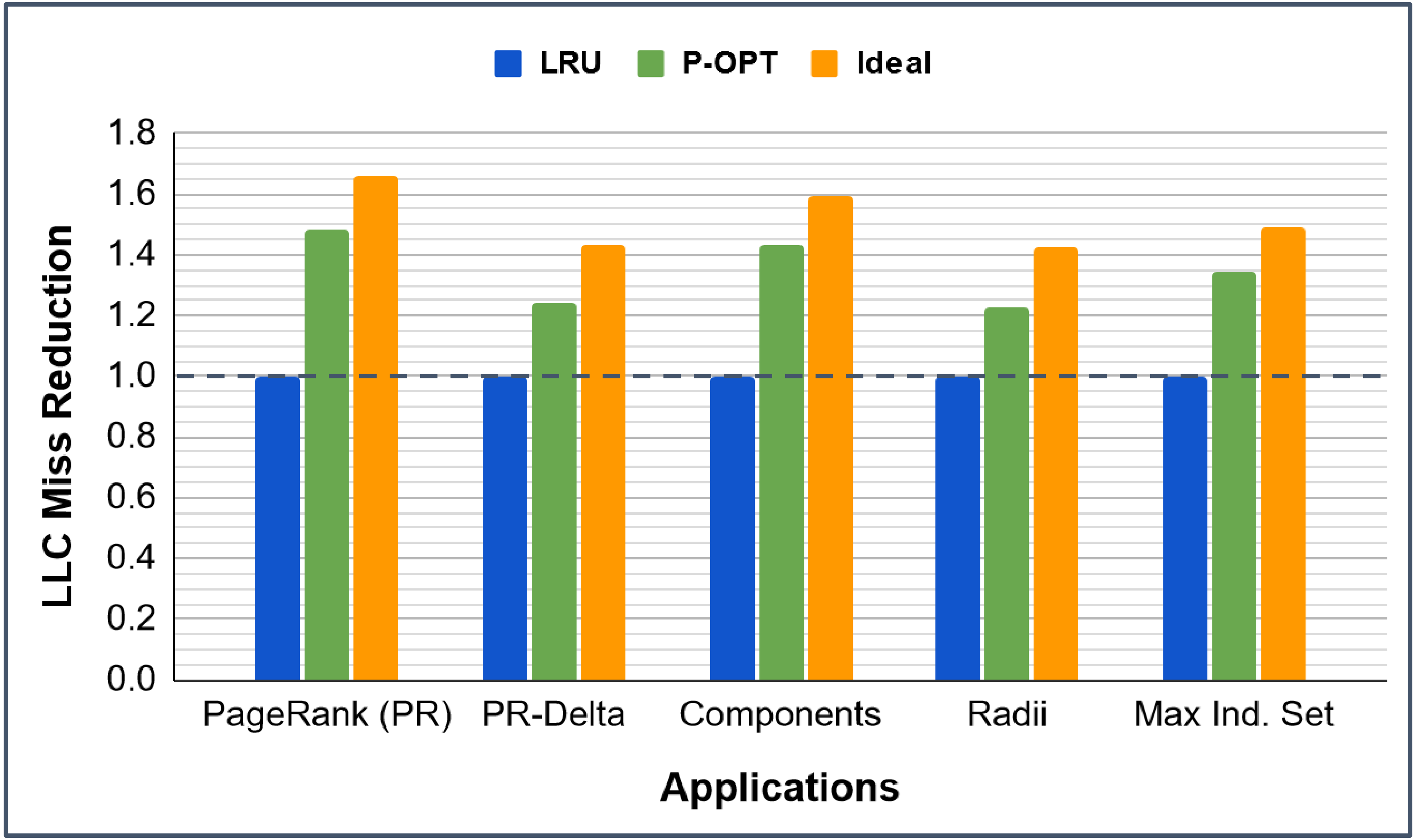


	E_0	E_1	E_2
C_0	1	0	∞
C_1	2	1	0
C_2	0	0	∞
C_3	0	1	0
C_4	0	0	∞

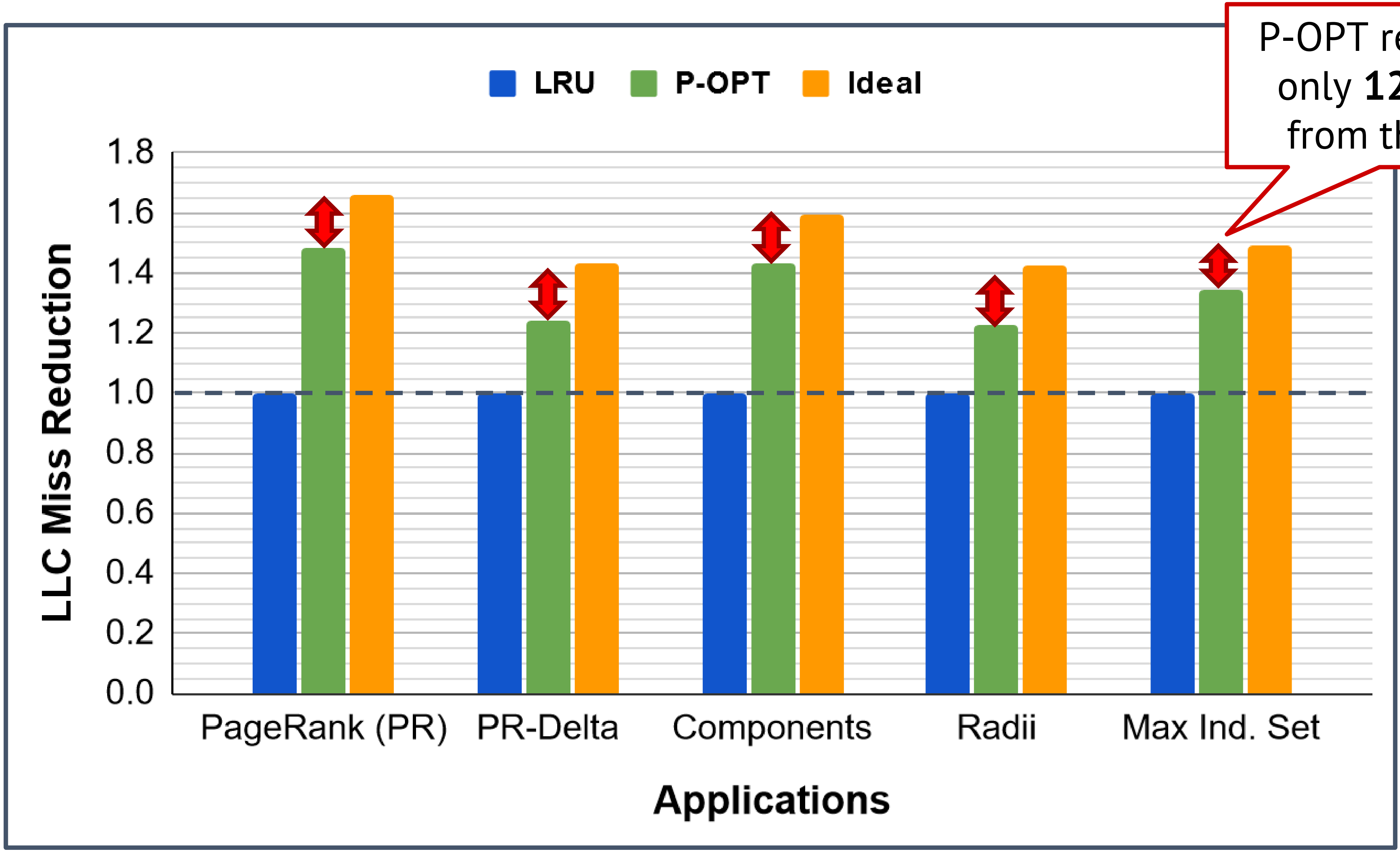
Size the reref matrix to fit in cache!

Rereference Matrix
(Quantized Transpose)

P-OPT Improves Cache Locality

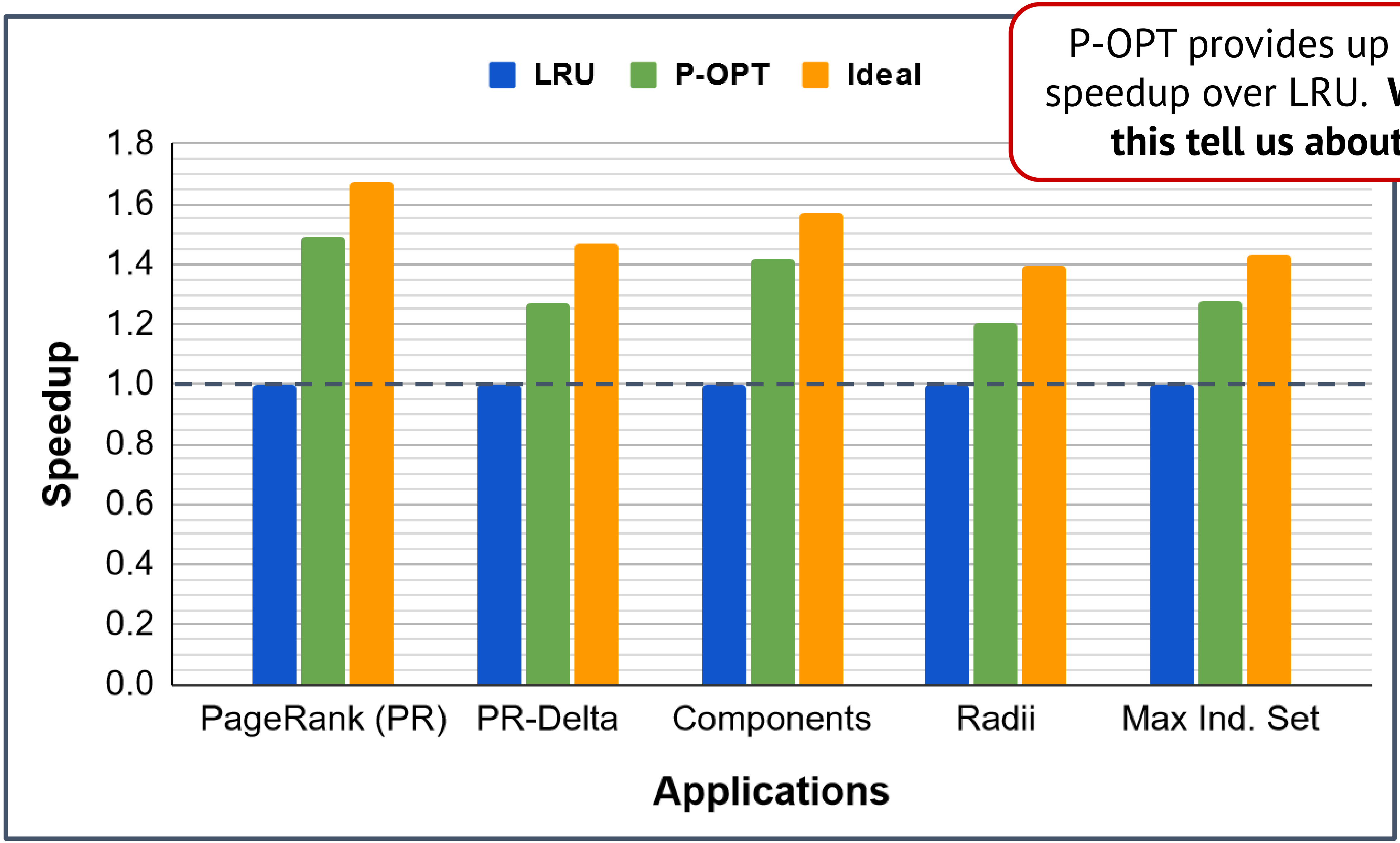


P-OPT Improves Cache Locality



P-OPT results are only **12%** away from the Ideal

P-OPT’s LLC Miss Reductions Directly Translate To Speedups



P-OPT provides up to **1.56x** speedup over LRU. **What does this tell us about LRU?**

What did we just learn?

- Sparse problems are ones that manipulate large, mostly-zero matrices
- Sparsity makes caching a useful part of the matrix hard
- Roofline model shows how close to peak perf. an app is
- Propagation blocking bins updates making irregular data fit in cache
- P-OPT is a *practical* implementation of Belady's OPT for graphs

(with acknowledgements to Vignesh Balaji, CMU ECE PhD 2021, now at Nvidia for contributions to this material)

Takeaways

- Heuristic-based policies are ineffective for irregular memory access patterns
- The graph’s transpose enables Belady’s MIN replacement policy
- P-OPT achieves close to ideal performance (*quantization can be an effective tool in making a design practical*)