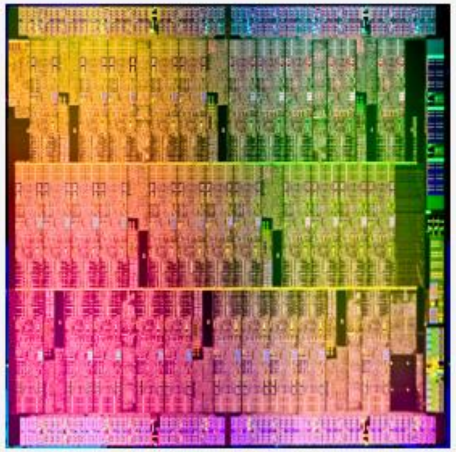
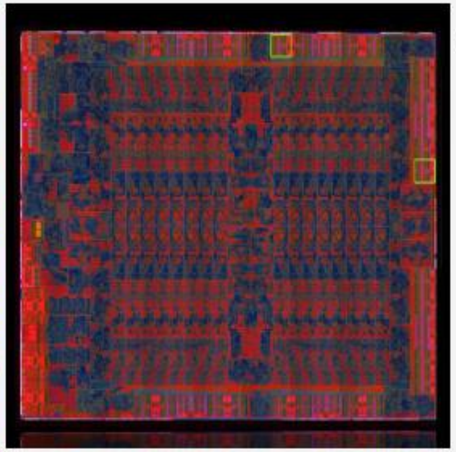
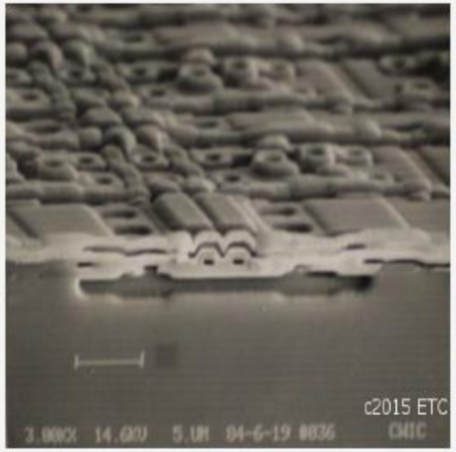
18-344: Computer Systems and the Hardware-Software Interface

Fall 2023



Course Description

Lecture 15: Sparsity

This course covers the design and implementation of computer systems from the perspective of the hardware software interface. The purpose of this course is for students to understand the relationship between the operating system, software, and computer architecture. Students that complete the course will have learned operating system fundamentals, computer architecture fundamentals, compilation to hardware abstractions, and how software actually executes from the perspective of the hardware software/boundary. The course will focus especially on understanding the relationships between software and hardware, and how those relationships influence the design of a computer system's software and hardware. The course will convey these topics through a series of practical, implementation-oriented lab assignments.

Credit: Brandon Lucia

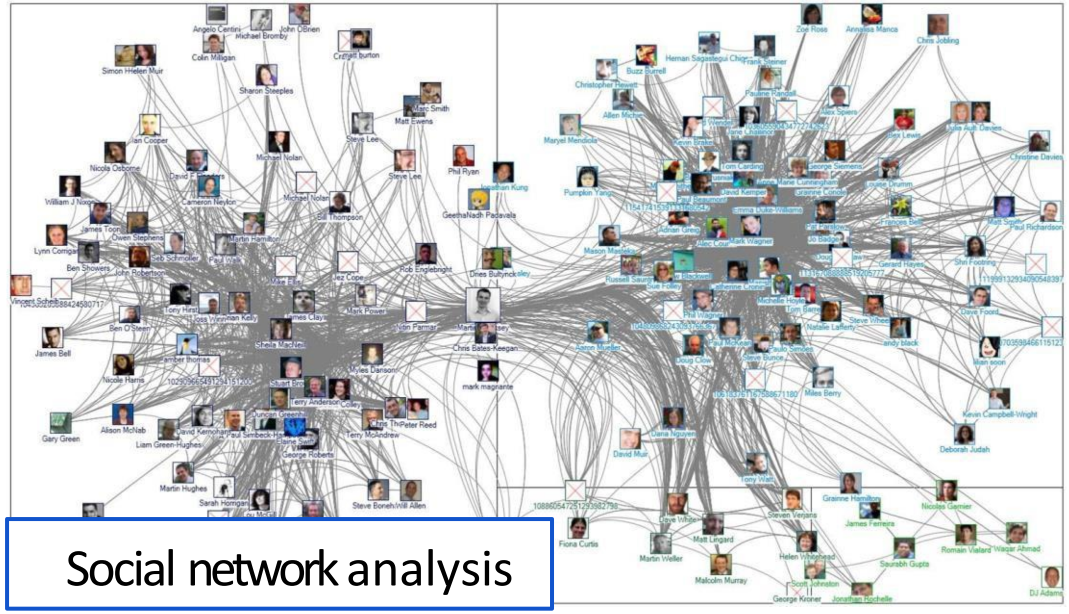
Today: Sparse Problems

- 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

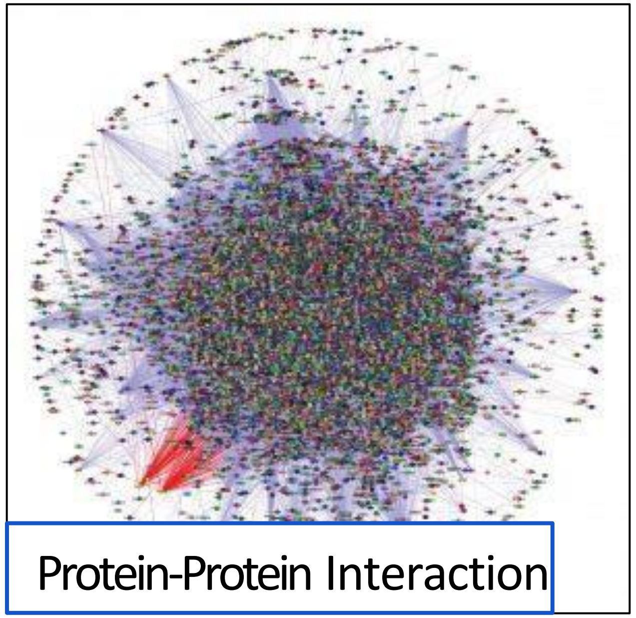
(with acknowledgements to Vignesh Balaji, CMU ECE PhD 2021, now at Nvidia for contributions to this material)

Graph Processing Problems are Sparse Problems

Path Planning



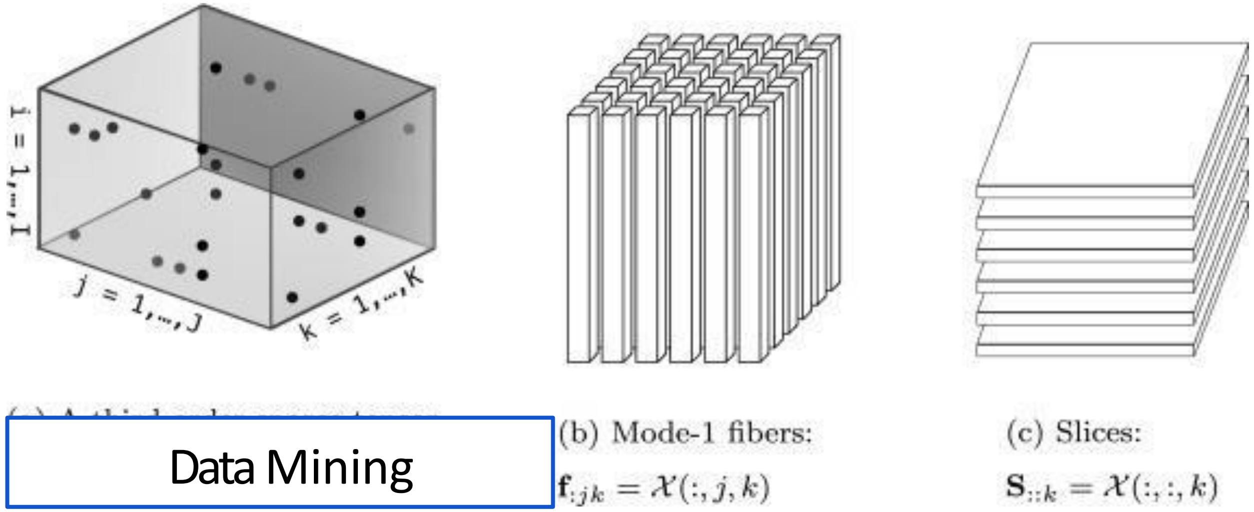
Social network analysis



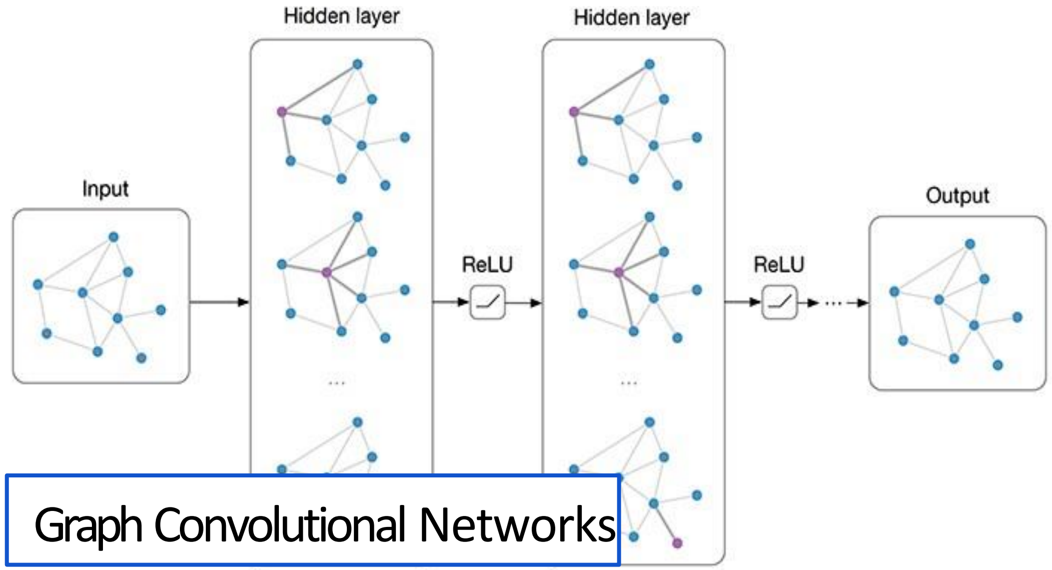
Protein-Protein Interaction

The canonical examples of sparse problems are graph processing applications.

Machine Learning Problems are Sparse Problems

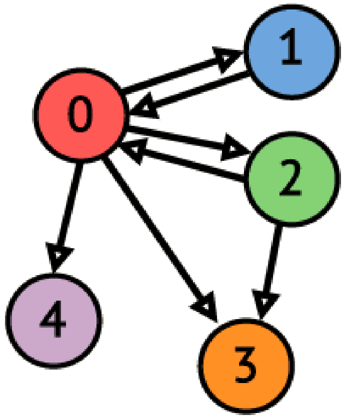


Data Mining



Graph Convolutional Networks

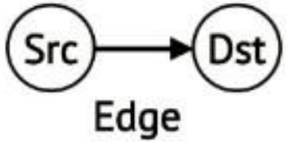
What does a graph processing program look like?



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])
```

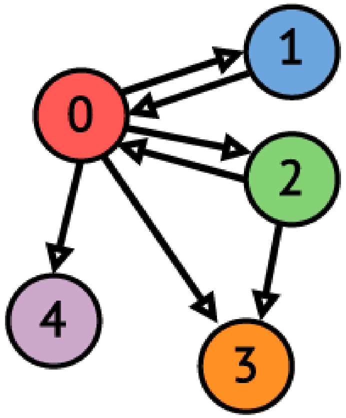



dstData


srcData

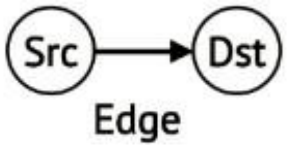
stores vertex property information
if srcData == dstData, updating in-place;
often “swap” srcData & dstData from 1 iteration to the next iteration

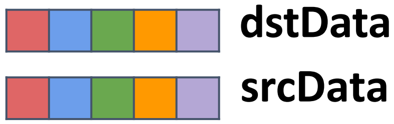
What does a graph processing program look like?



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

COO
(EdgeList)

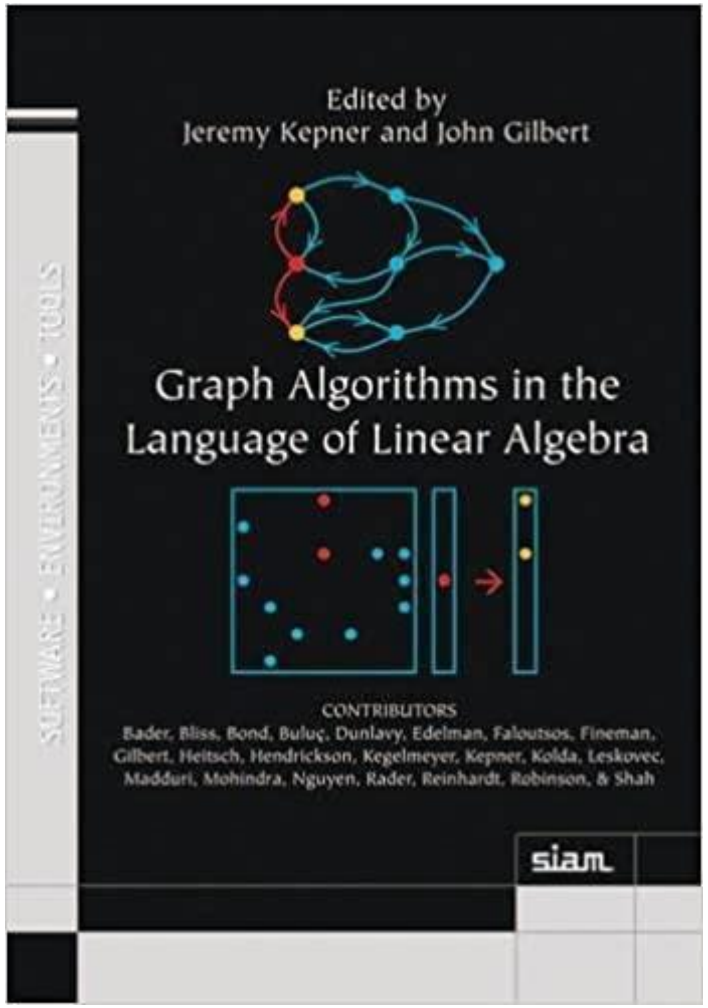




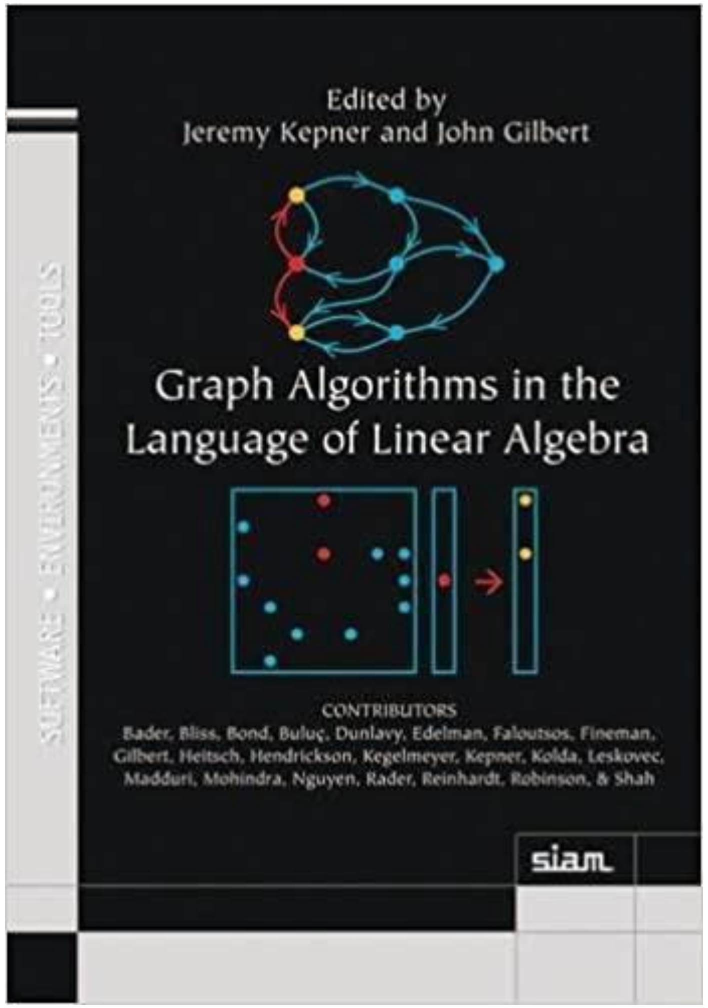

```
PageRank(-ish){
  for e in EL:
    rank_n[e.dst] =
      rank_nminus1[e.src] + rank_n[e.dst]
}
```

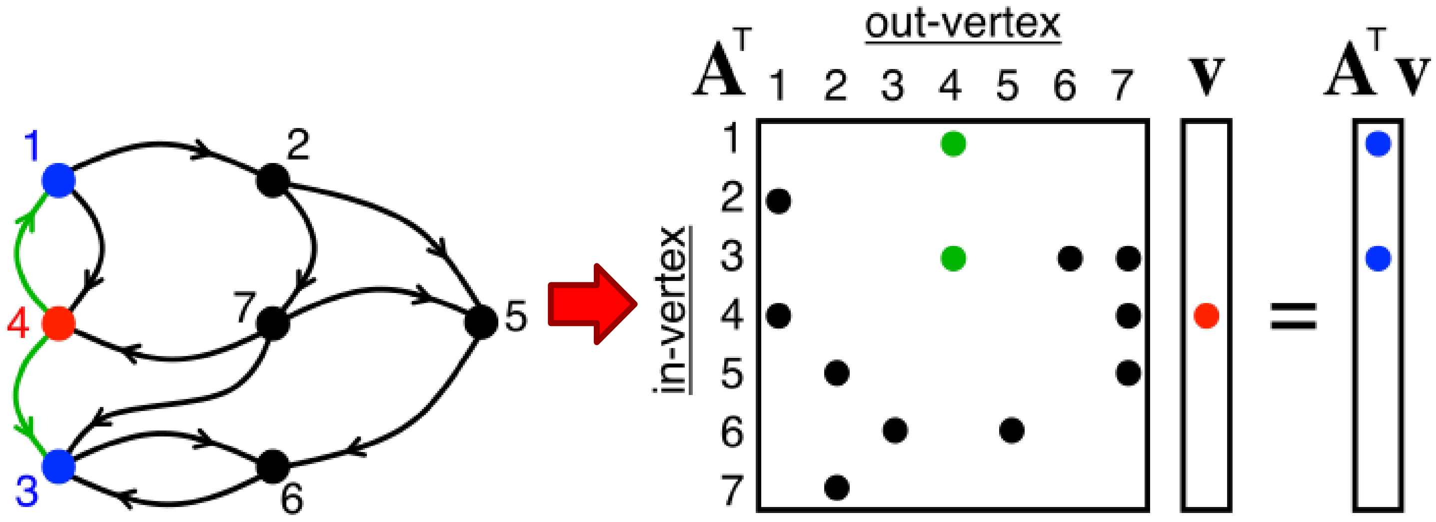
rank_n is a webpage's rank in this iteration,
rank_nminus1 is rank_n from the last iteration

Graph Analytics can be mapped to Sparse Linear Algebra

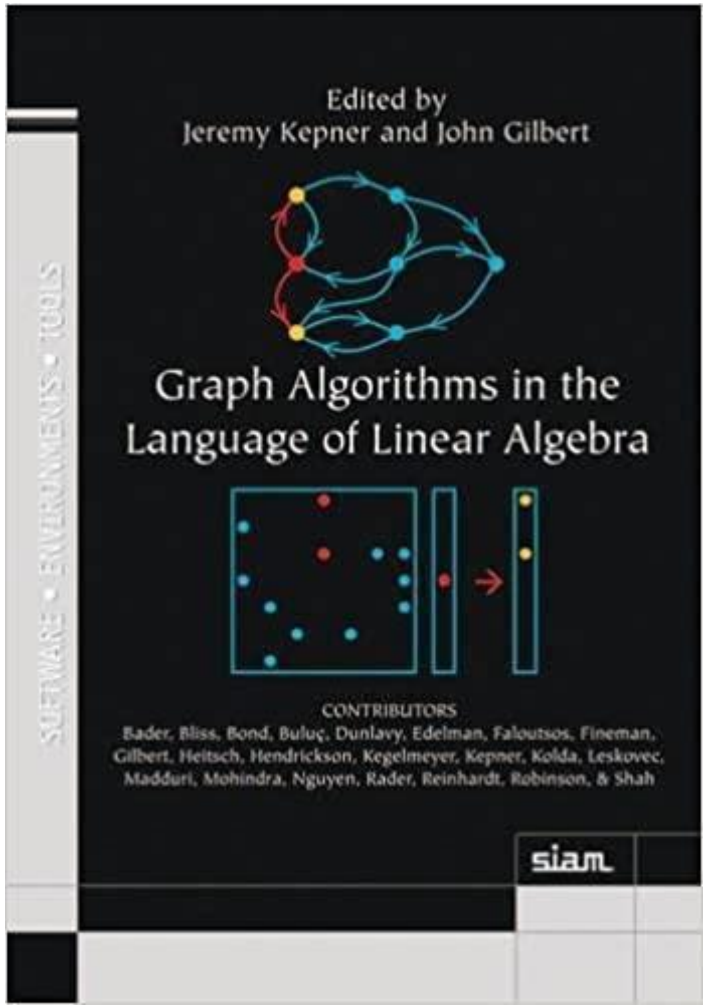


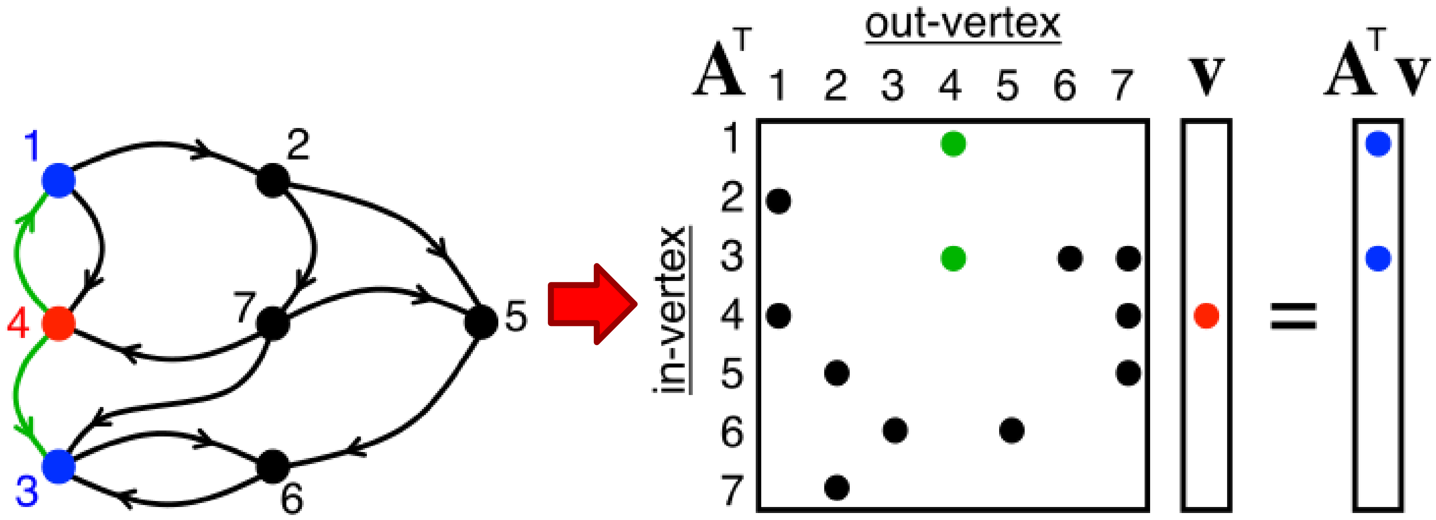
Graph Analytics can be mapped to Sparse Linear Algebra

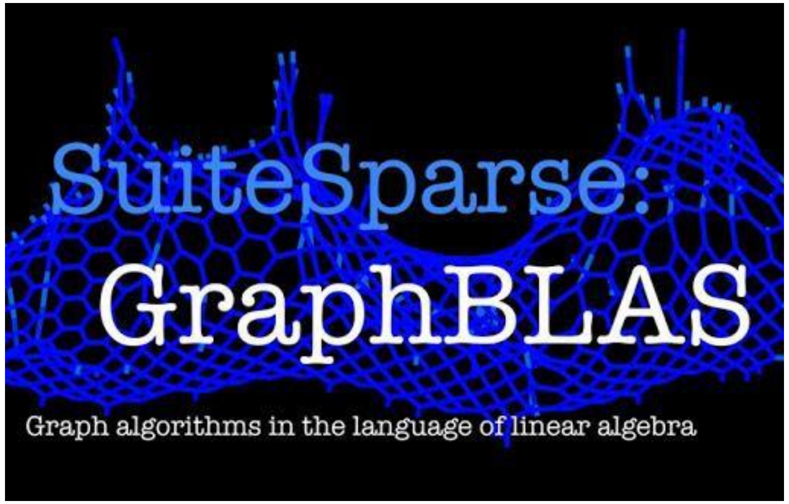




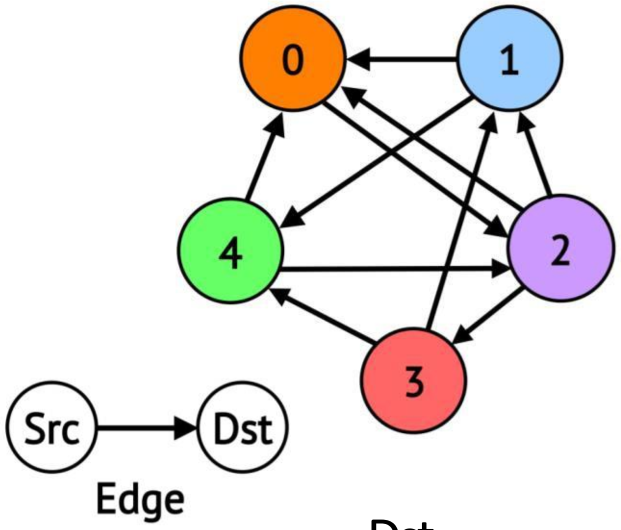
Graph Analytics can be mapped to Sparse Linear Algebra







How do graph applications correspond to linear algebra?

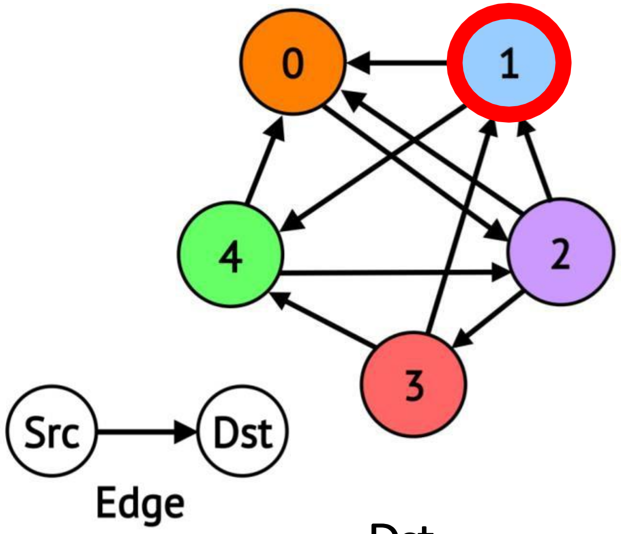


Dst →

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

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

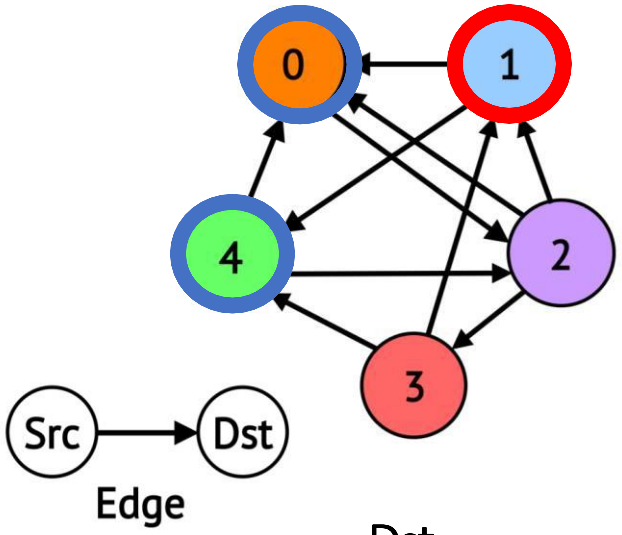
Initial x_i vector is starting vertex for BFS.

Dst $\longrightarrow$

Src $\downarrow$

$$\begin{array}{c} \\ S_0 \\ S_1 \\ S_2 \\ S_3 \\ S_4 \end{array} \begin{array}{c} \begin{array}{ccccc} D_0 & D_1 & D_2 & D_3 & D_4 \end{array} \\ \begin{bmatrix} & & 1 & & \\ 1 & & & & 1 \\ 1 & 1 & & 1 & \\ & 1 & & & 1 \\ 1 & & 1 & & \end{bmatrix}^{\mathbf{T}} \end{array} \begin{bmatrix} \\ 1 \\ \\ \\ \\ \end{bmatrix} = \begin{bmatrix} 1 \\ \\ \\ \\ 1 \end{bmatrix}$$

How do graph applications correspond to linear algebra?



Matrix-transpose-vector product is one BFS iteration

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

A Transpose

Initial x_i vector is starting vertex for BFS.

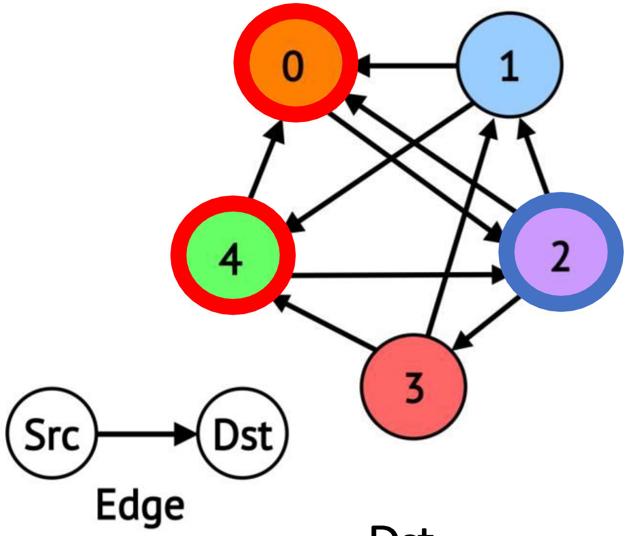
Initial x_{i+1} is vertices reachable from x_i

Dst →

Src ↓

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

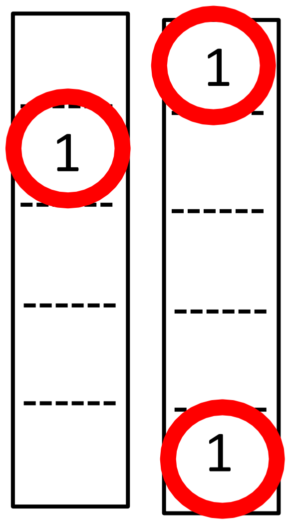
How do graph applications correspond to linear algebra?



Matrix transpose vector product is one BFS iteration

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

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

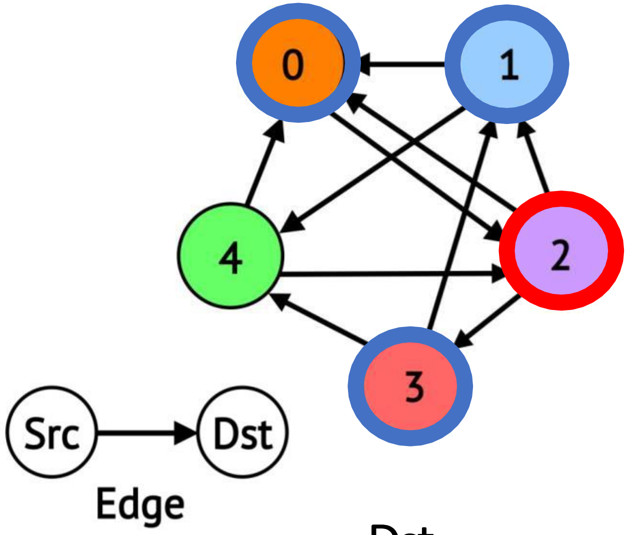


Dst $\longrightarrow$

Src $\downarrow$

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

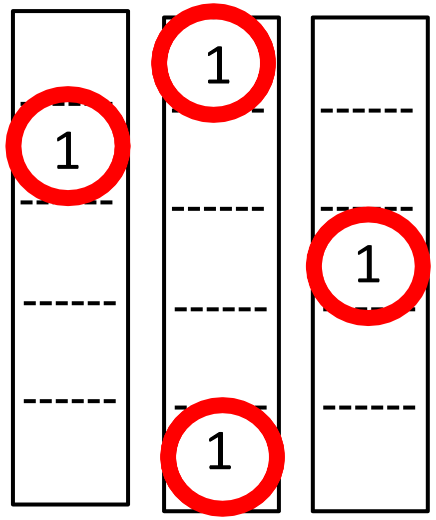
How do graph applications correspond to linear algebra?



Matrix transpose vector product is one BFS iteration

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

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

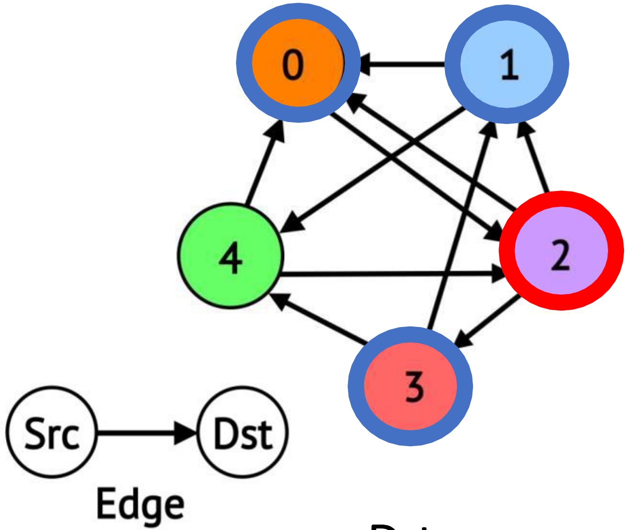


Dst $\longrightarrow$

Src $\downarrow$

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

How do graph applications correspond to linear algebra?



Matrix transpose vector product is one BFS iteration

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

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

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

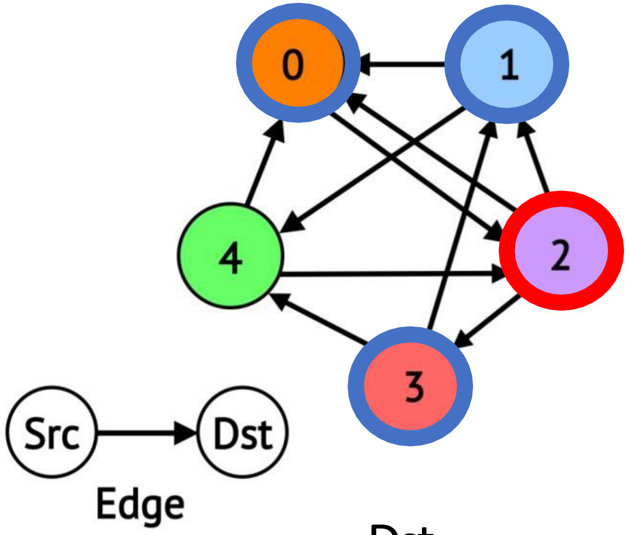
Dst $\longrightarrow$

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

Src $\downarrow$

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 ⟶

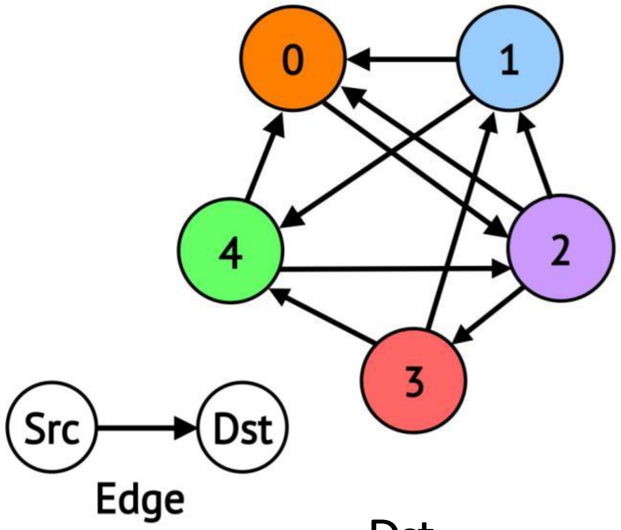
Src ↓

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

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

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

Nobody EVER uses the adjacency matrix!



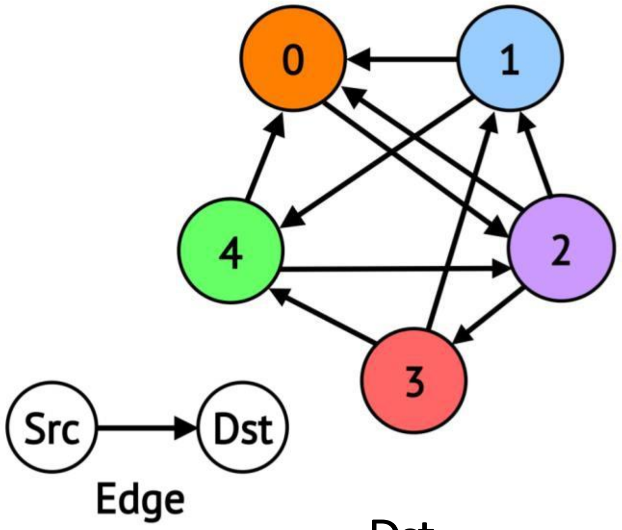
Dst →

Src ↓

	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?

Nobody EVER uses the adjacency matrix!



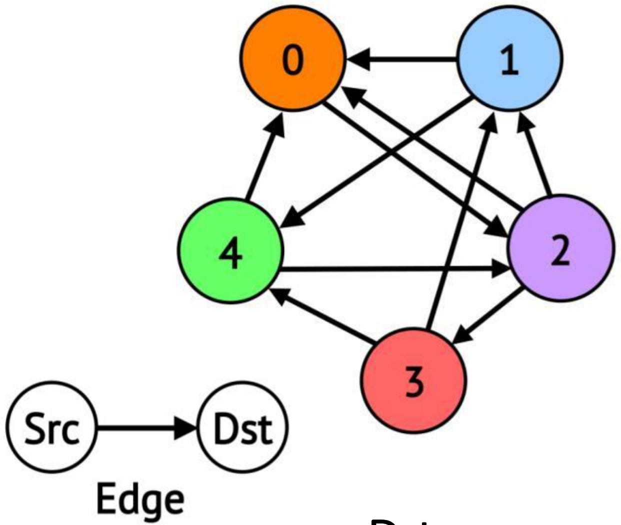
Dst ⟶

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

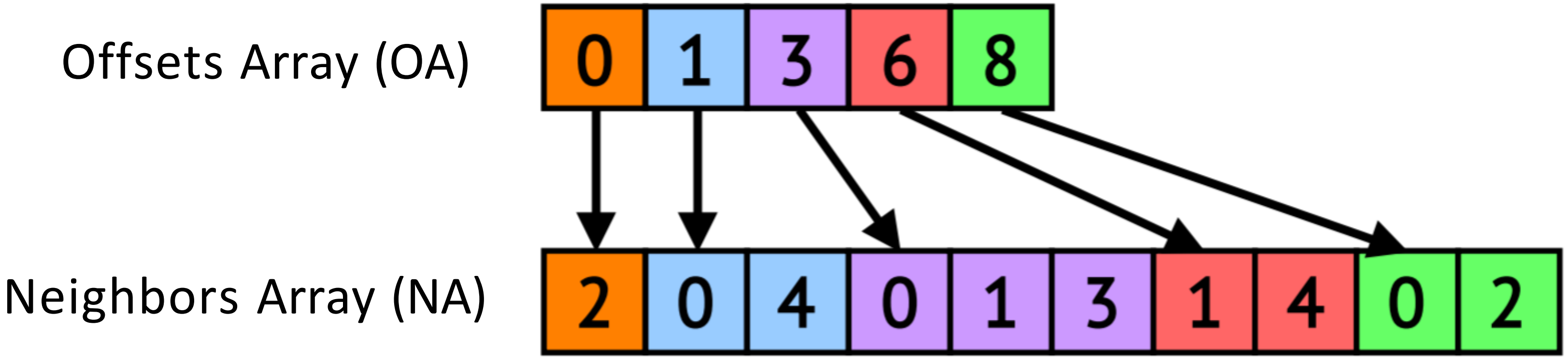
Compressed Sparse Data Structures for Feasible Memory Size



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		



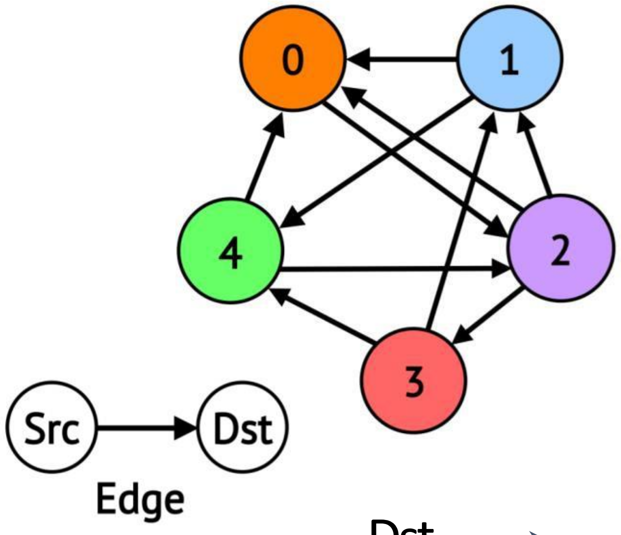
Compressed Sparse Row (CSR)
Outgoing Neighbors

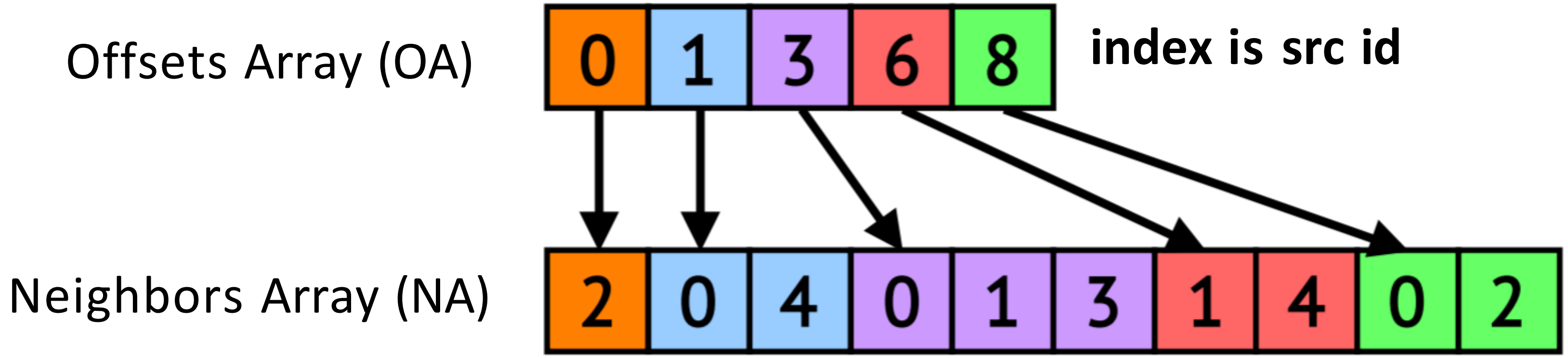
Vertex Property Array
i.e., srcData / dstData

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





EdgeList sorted by SrcIDs

Compressed Sparse Row (CSR)
Outgoing Neighbors

Dst →

Src ↓

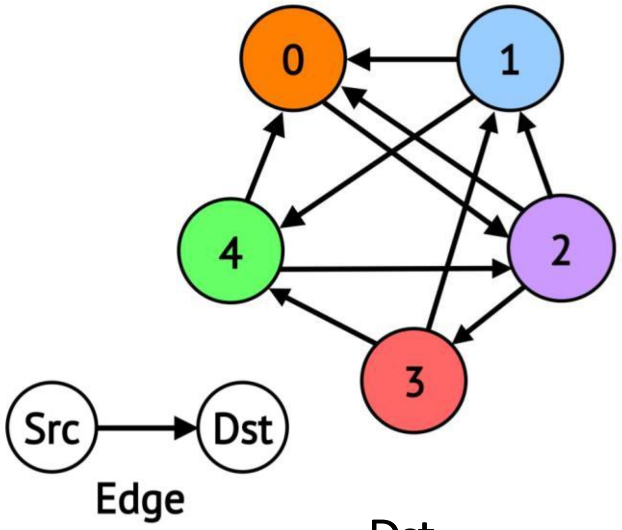
	D_0	D_1	D_2	D_3	D_4
S_0			1		
S_1	1				1
S_2	1	1		1	
S_3		1			1
S_4	1		1		

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



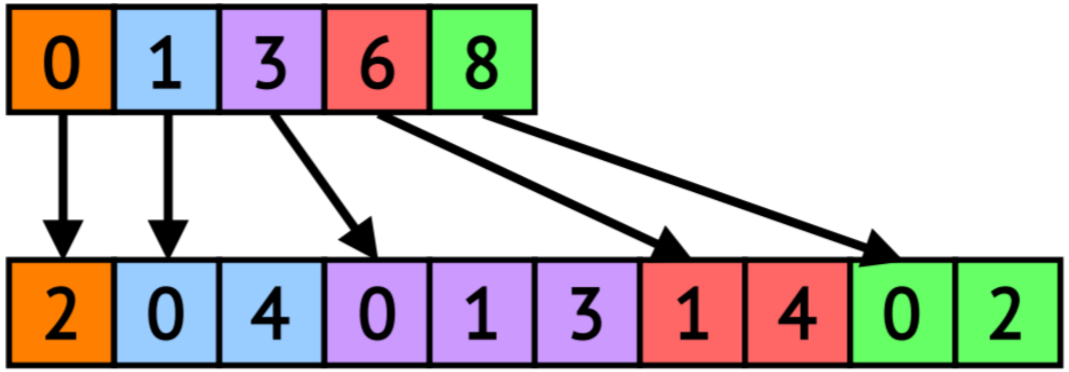
Dst →

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)

Neighbors Array (NA)



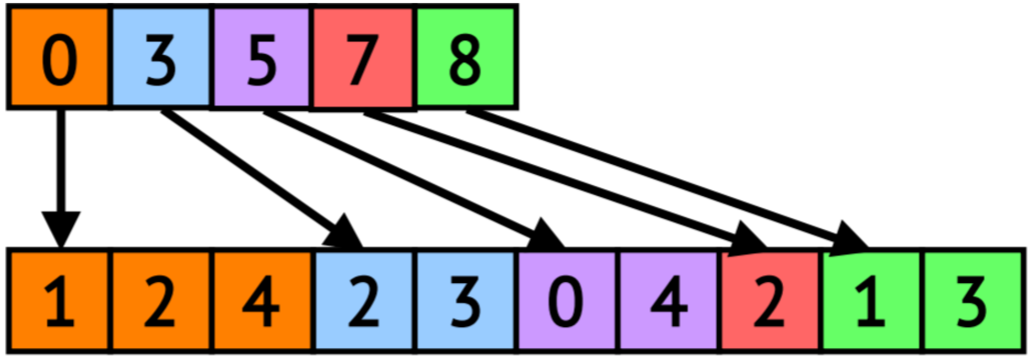
EdgeList sorted by SrcIDs

Compressed Sparse Row (CSR)
Outgoing Neighbors

The CSC is the *transpose* of the CSR

Offsets Array (OA)

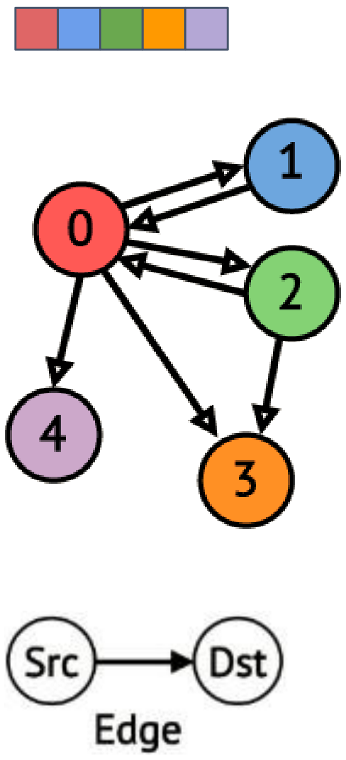
Neighbors Array (NA)



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

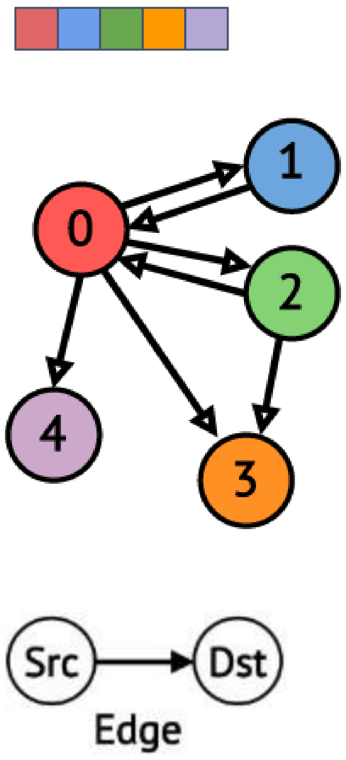
COO
(EdgeList)

```
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

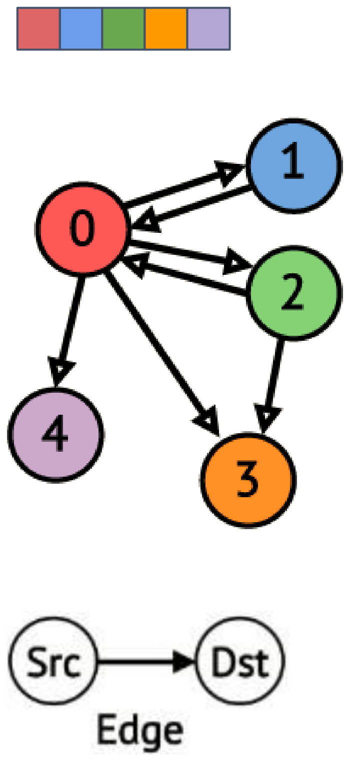
COO
(EdgeList)

```
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**

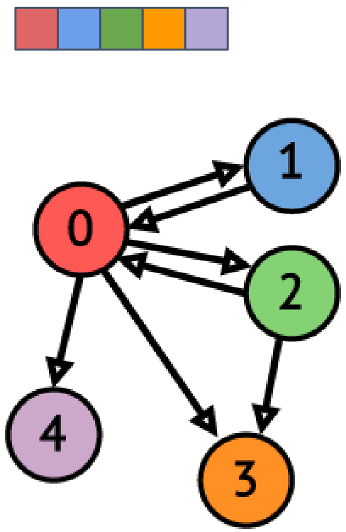
neigh_count

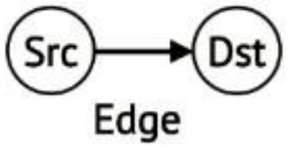
```
sum = 0
for i in 0 .. |V|:
  tmp = neigh_count[i]
  neigh_count[i] = sum; //OA
  neigh_count_dup[i] = sum;
  sum += tmp
```

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

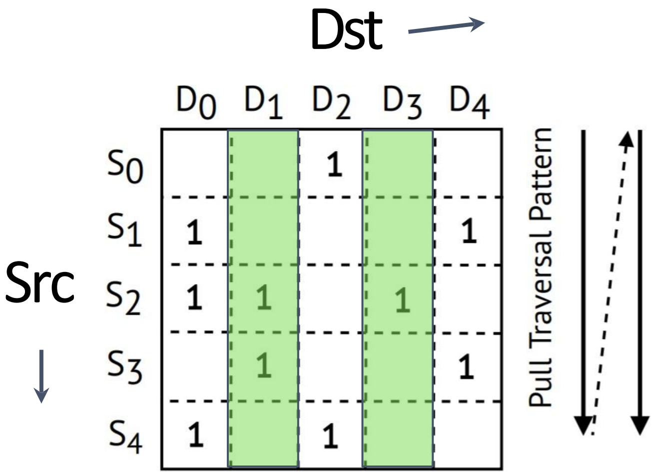
OA_dup

1	2	0	0	0	2	0

NA

Completed CSC

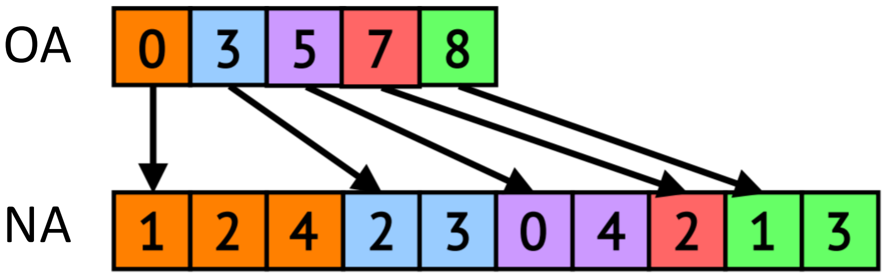
Compressed Representations ⇒ Irregular Memory Accesses



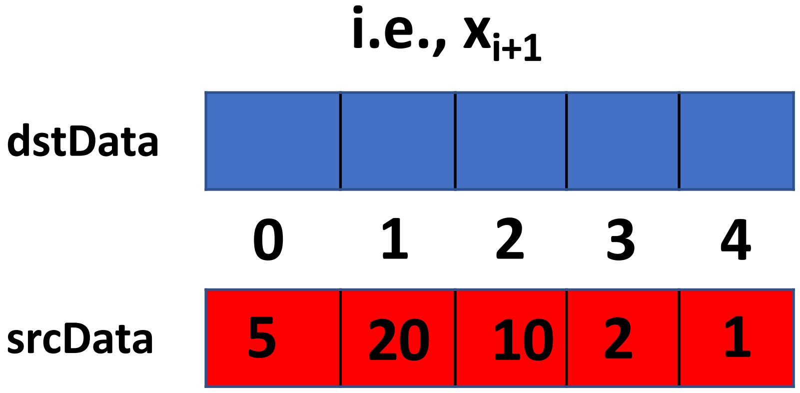
Pull (CSC Traversal)

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

Pull traversal performs *irregular read operations* that lack locality

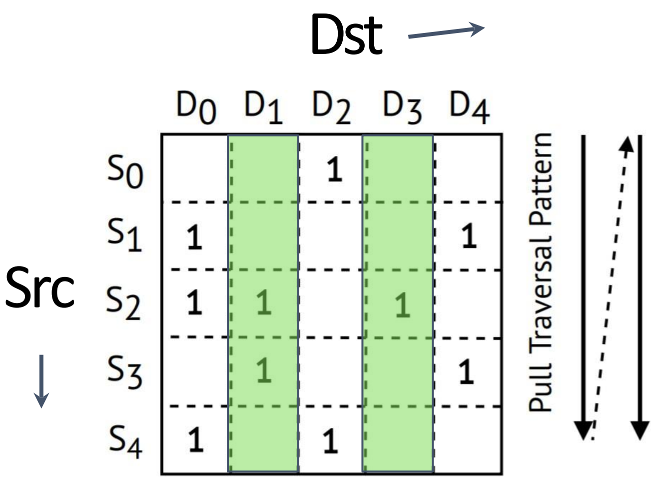


CSC



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

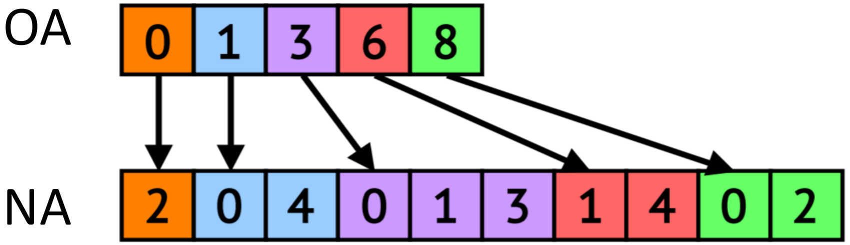
Compressed Representations ⇒ Irregular Memory Accesses



Push (CSR Traversal)

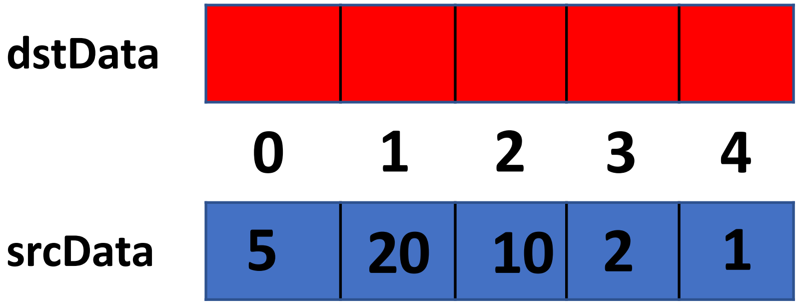
```
for src in G:
     for dst in out_neighs(src):
              dstData[dst] += srcData[src]
```

Push traversal performs *irregular write operations* that lack locality



CSR

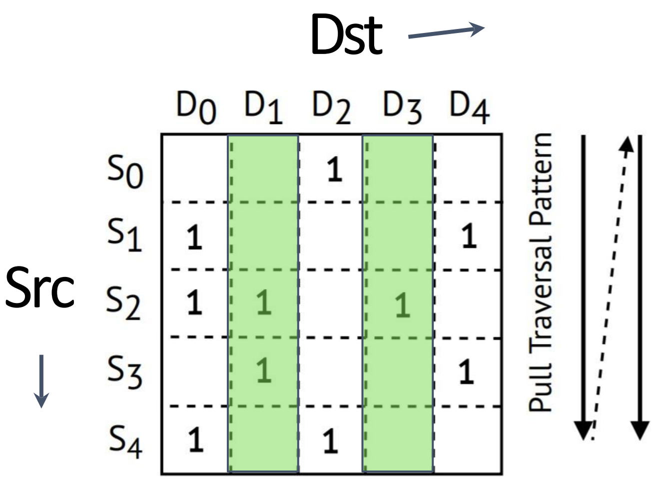
i.e., x_{i+1}



e.g., current rank of page I,

e.g., current shortest path from source vertex

Compressed Representations ⇒ Irregular Memory Accesses



Push (CSR Traversal)

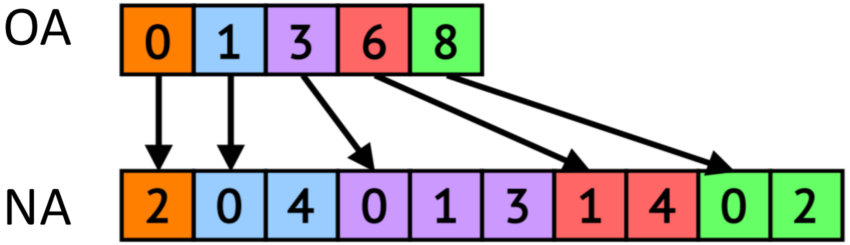
```
for src in G:
    for dst in out_neighs(src):
        dstData[dst] += srcData[src]
```

Push traversal performs *irregular write operations* that lack locality

i.e., x_{i+1}

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

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

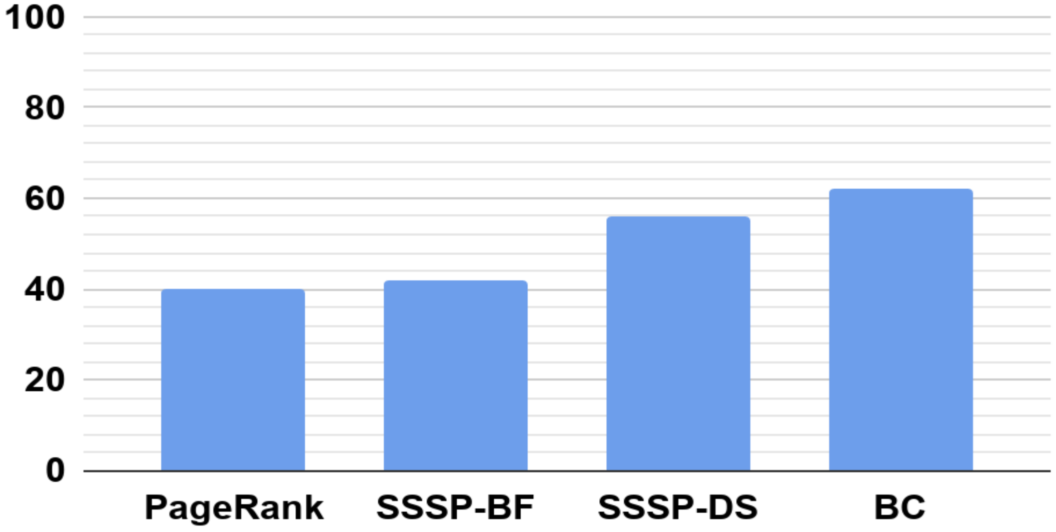


CSR

Irregular Data Footprint >> LLC Size
Size of srcData ~ **256MB** (32M vertices * 8B)

Irregular Accesses Lead to Poor Locality

LLC Miss Rate (%)

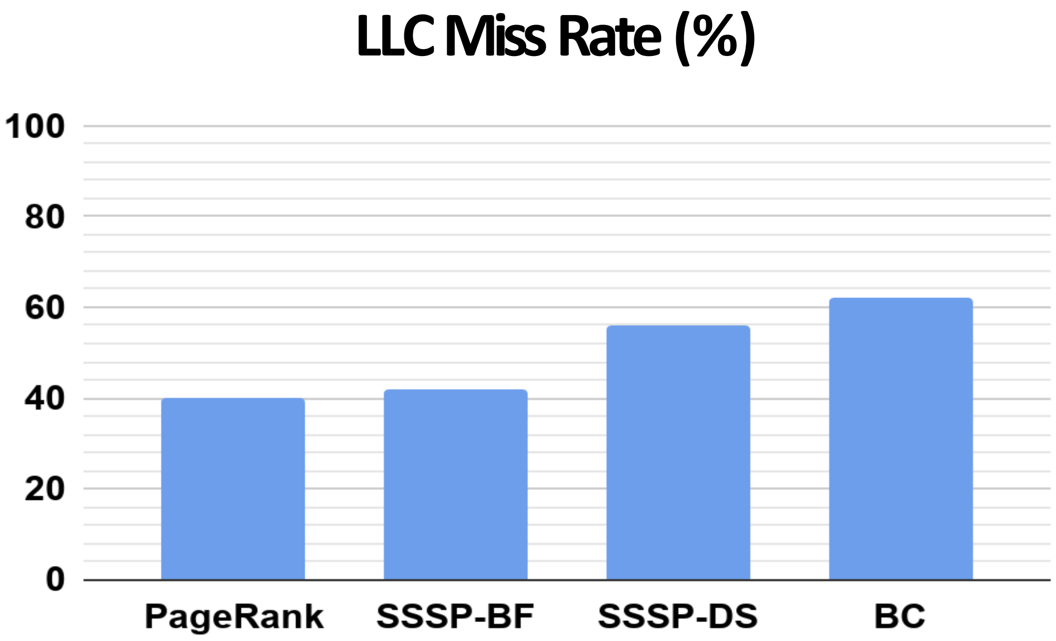


Running on RMAT27 Graph w/ 35MB LLC

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

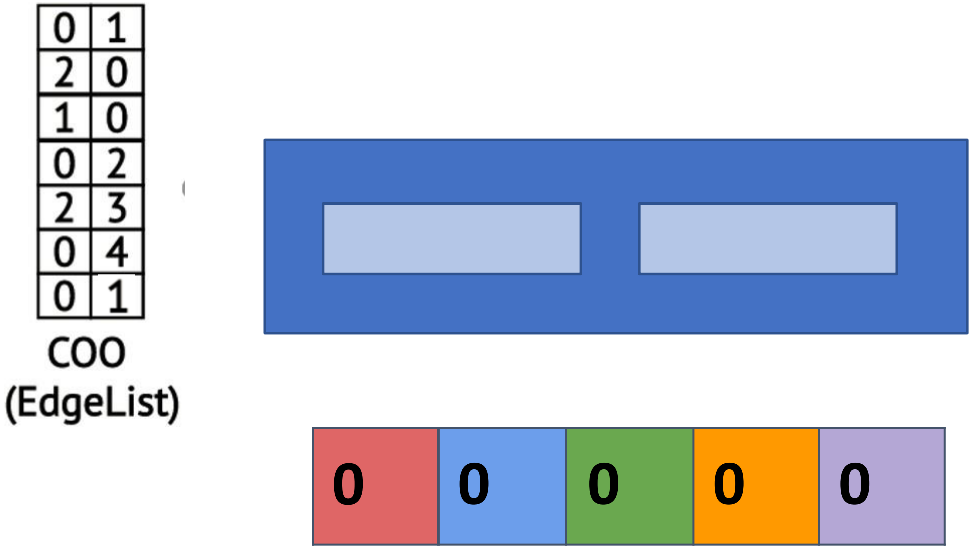
Right Figure from "Optimizing Cache Performance for Graph Analytics" ArXiv v1;

Irregular Accesses Lead to Poor Locality



Running on RMAT27
Graph w/ 35MB LLC

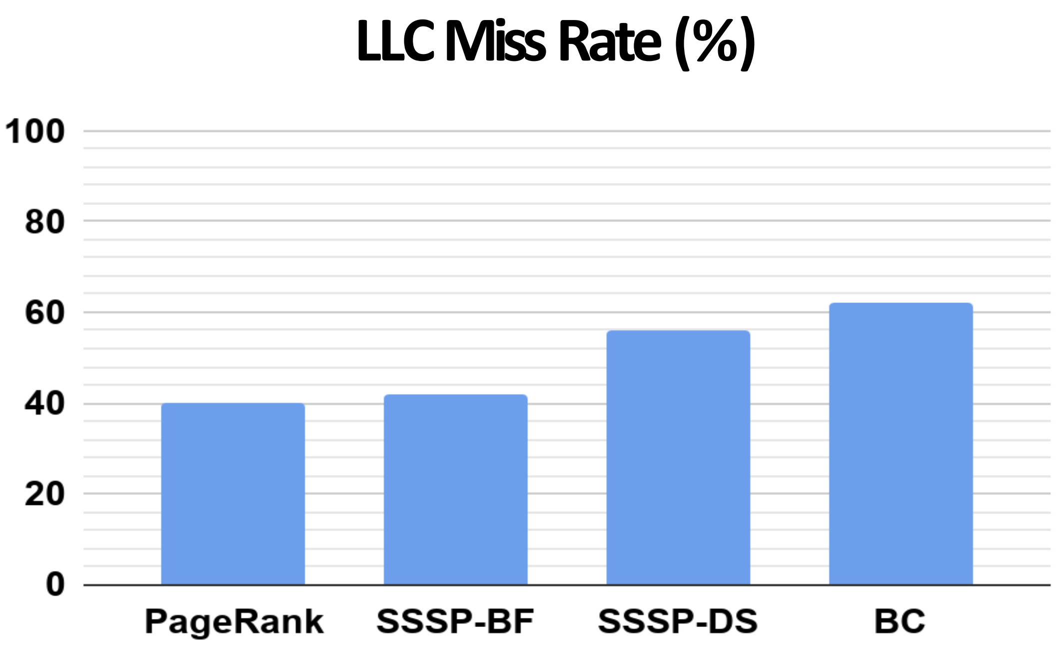
Dst coordinate of edge is index in dstData:
totally input dependent & random!!!



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

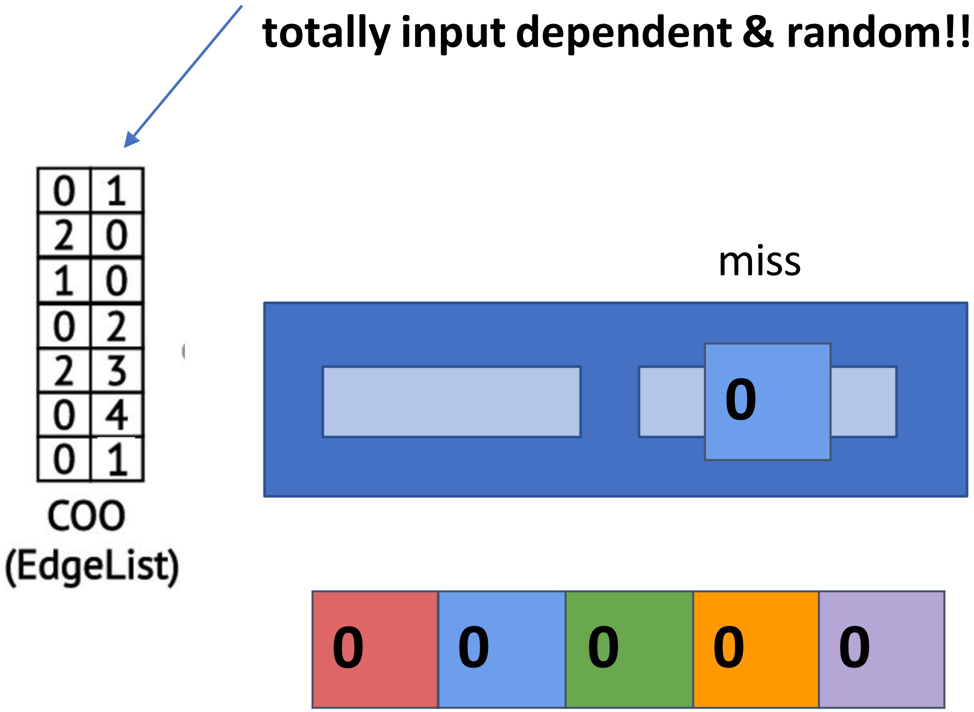
Right Figure from "Optimizing Cache Performance for Graph Analytics" ArXiv v1;

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!!!



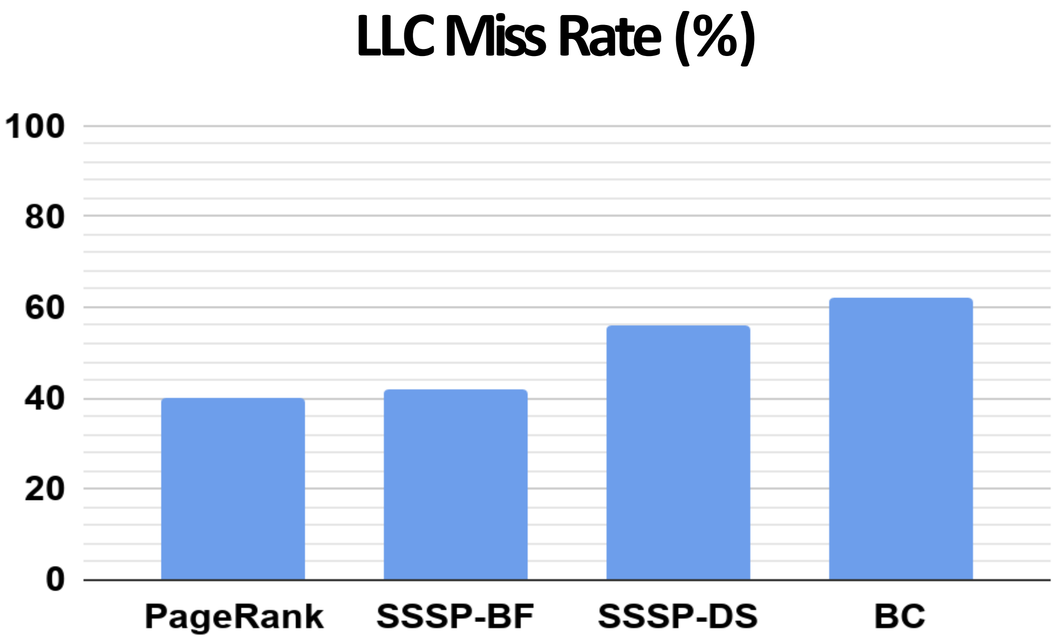
dstData

Remember: dstData[e.dst] ++

and e.dst is random, from edge list

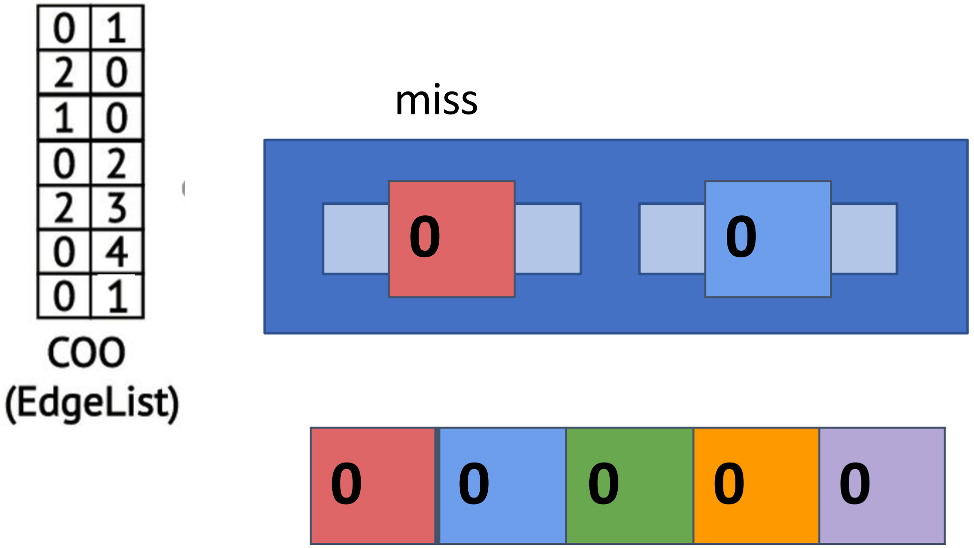
Right Figure from "Optimizing Cache Performance for Graph Analytics" ArXiv v1;

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!!!



dstData

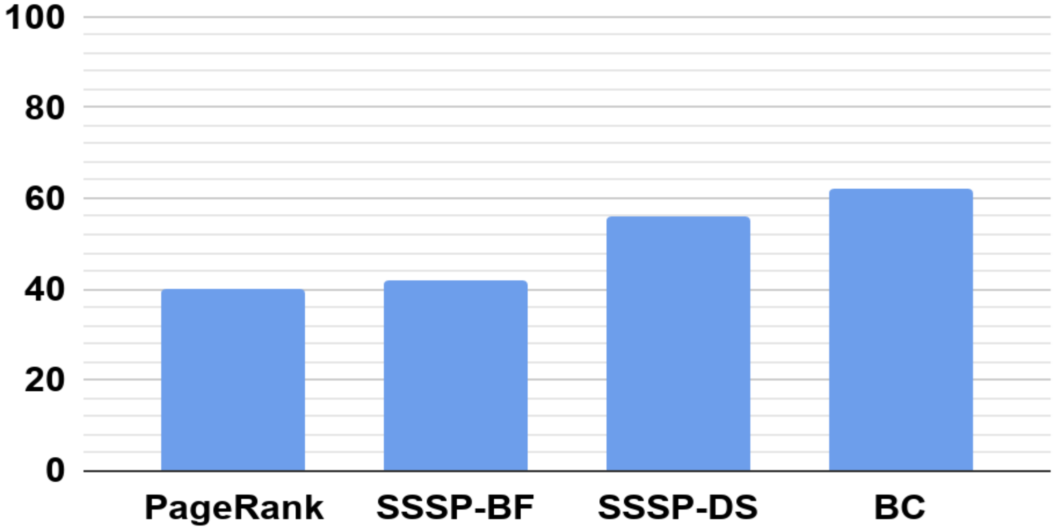
Remember: dstData[e.dst] ++

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 (%)



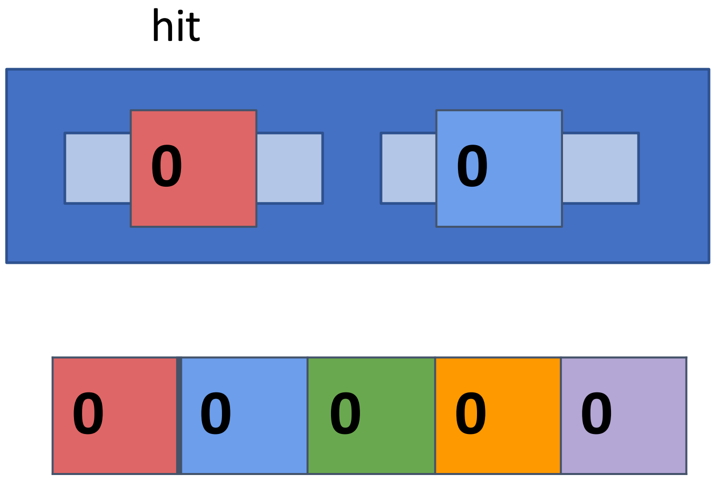
Running on RMAT27 Graph w/ 35MB LLC

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)



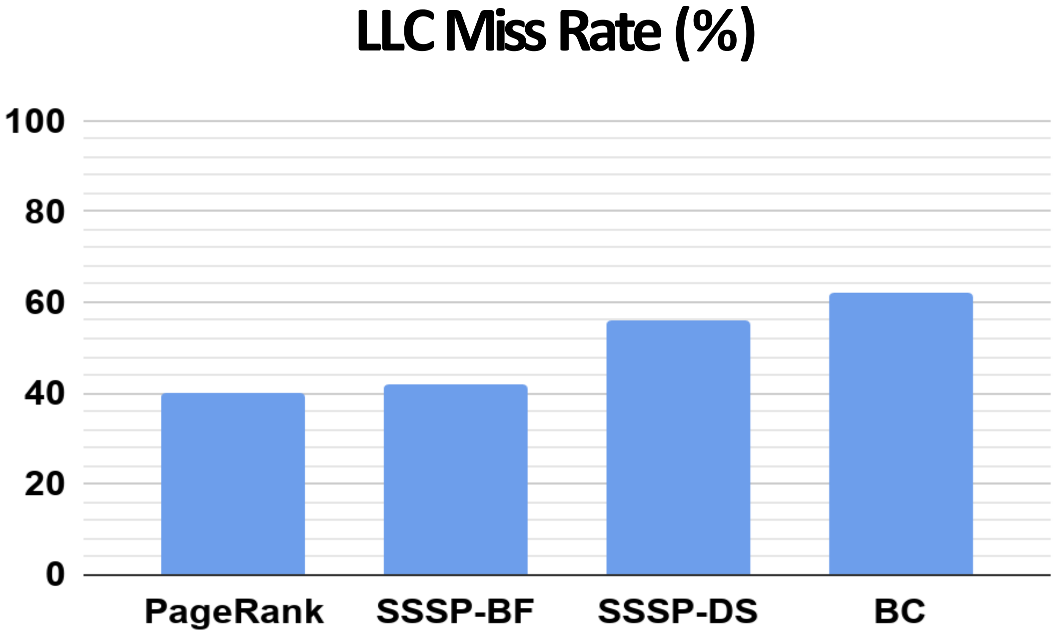
dstData

Remember: dstData[e.dst] ++

and e.dst is random, from edge list

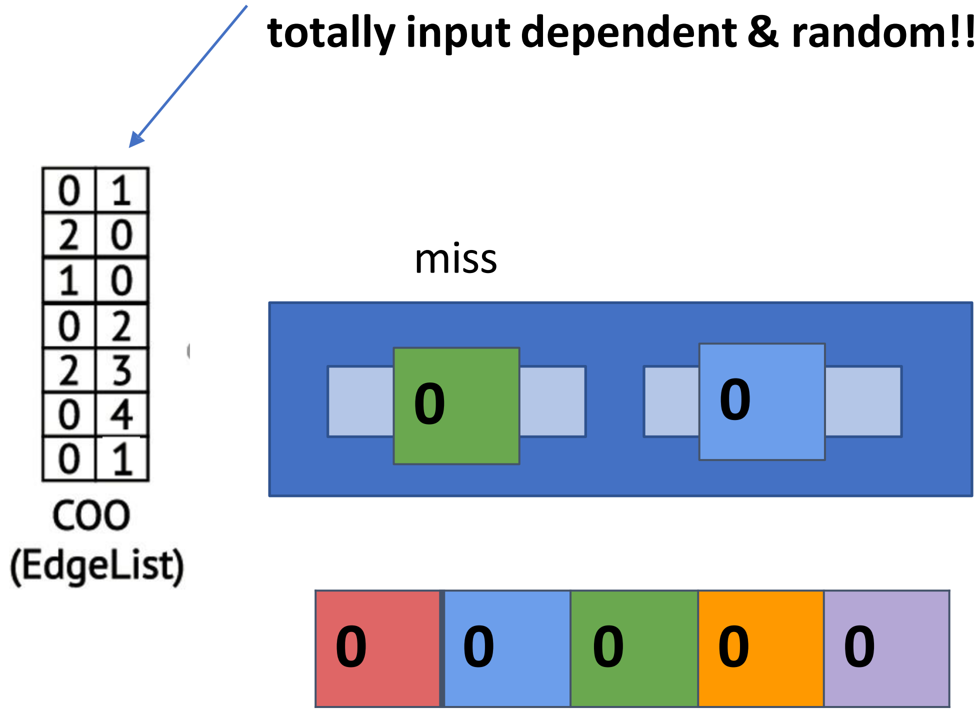
Right Figure from "Optimizing Cache Performance for Graph Analytics" ArXiv v1;

Irregular Accesses Lead to Poor Locality



Running on RMAT27 Graph w/ 35MB LLC

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



dstData

Remember: dstData[e.dst] ++

and e.dst is random, from edge list

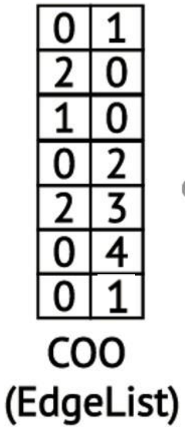
Right Figure from "Optimizing Cache Performance for Graph Analytics" ArXiv v1;

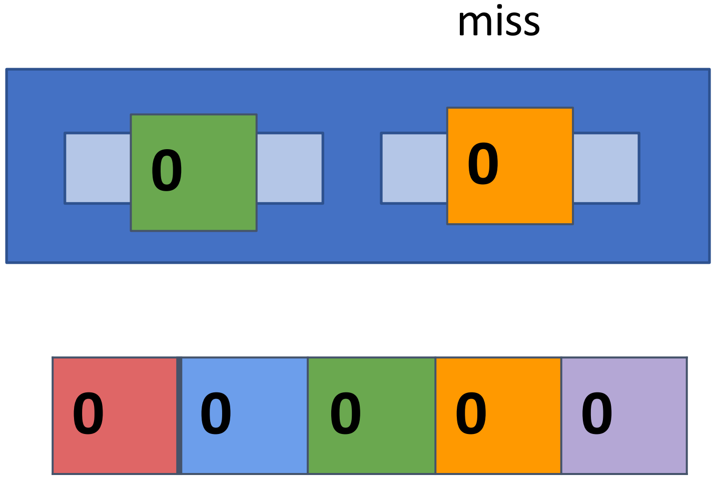
Irregular Accesses Lead to Poor Locality

LLC Miss Rate (%)

Running on RMAT27 Graph w/ 35MB LLC

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





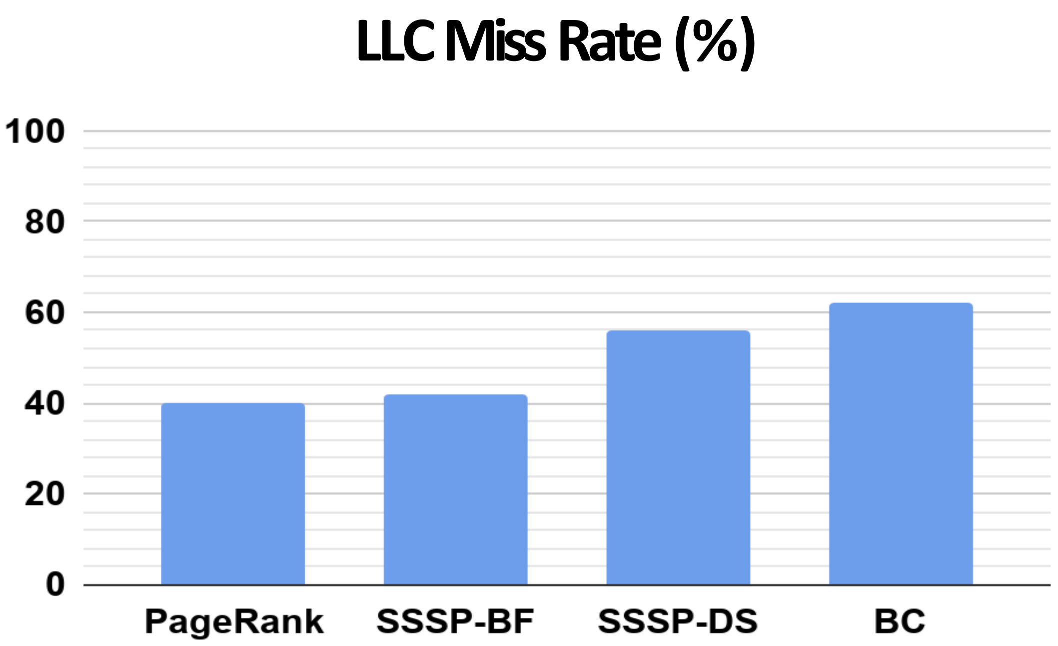
dstData

Remember: dstData[e.dst] ++

and e.dst is random, from edge list

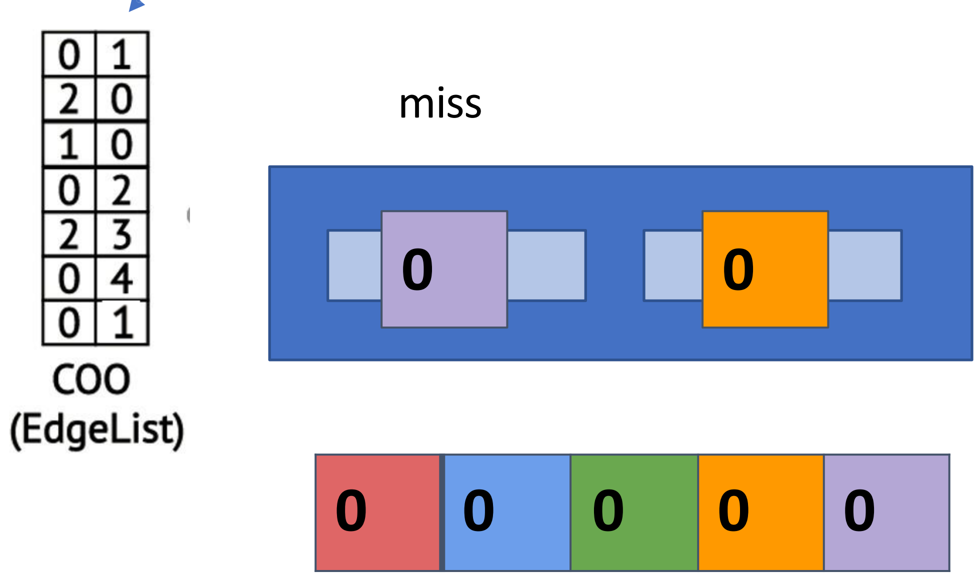
Right Figure from "Optimizing Cache Performance for Graph Analytics" ArXiv v1;

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!!!

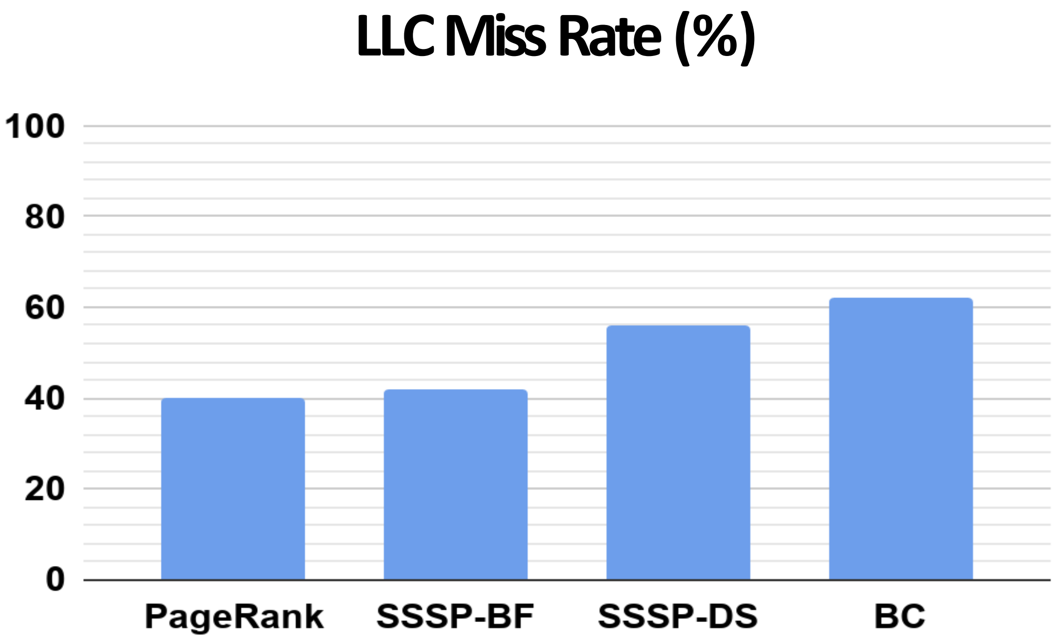


dstData

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

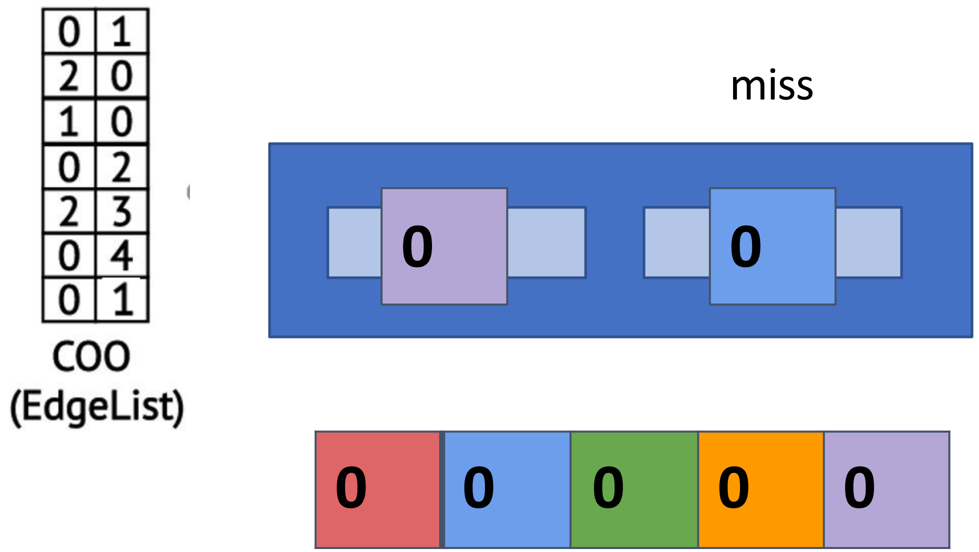
Right Figure from "Optimizing Cache Performance for Graph Analytics" ArXiv v1;

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!!!



dstData

Remember: dstData[e.dst] ++

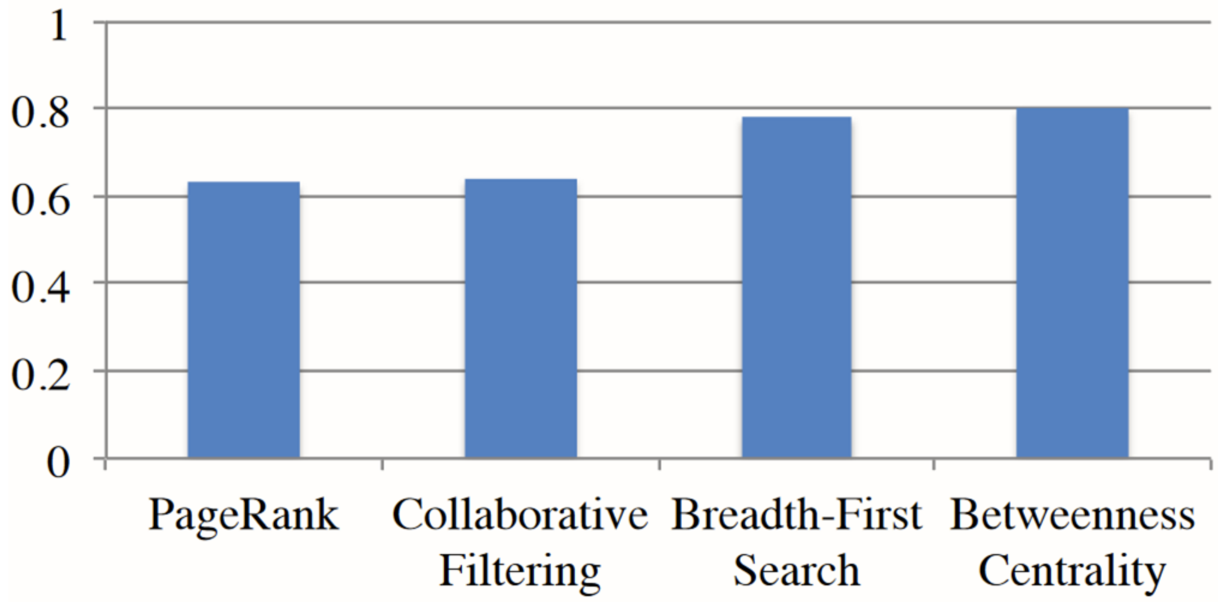
and e.dst is random, from edge list

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

Irregular Accesses Lead to Poor Locality

LLC Miss Rate (%)

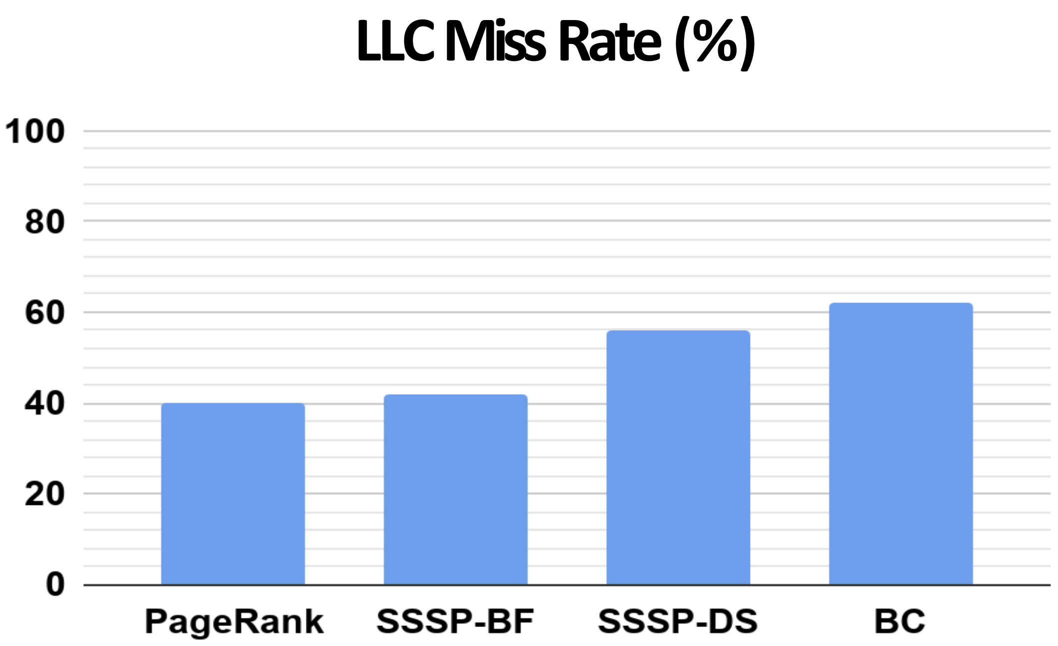
Cycles stalled on DRAM / Total Cycles

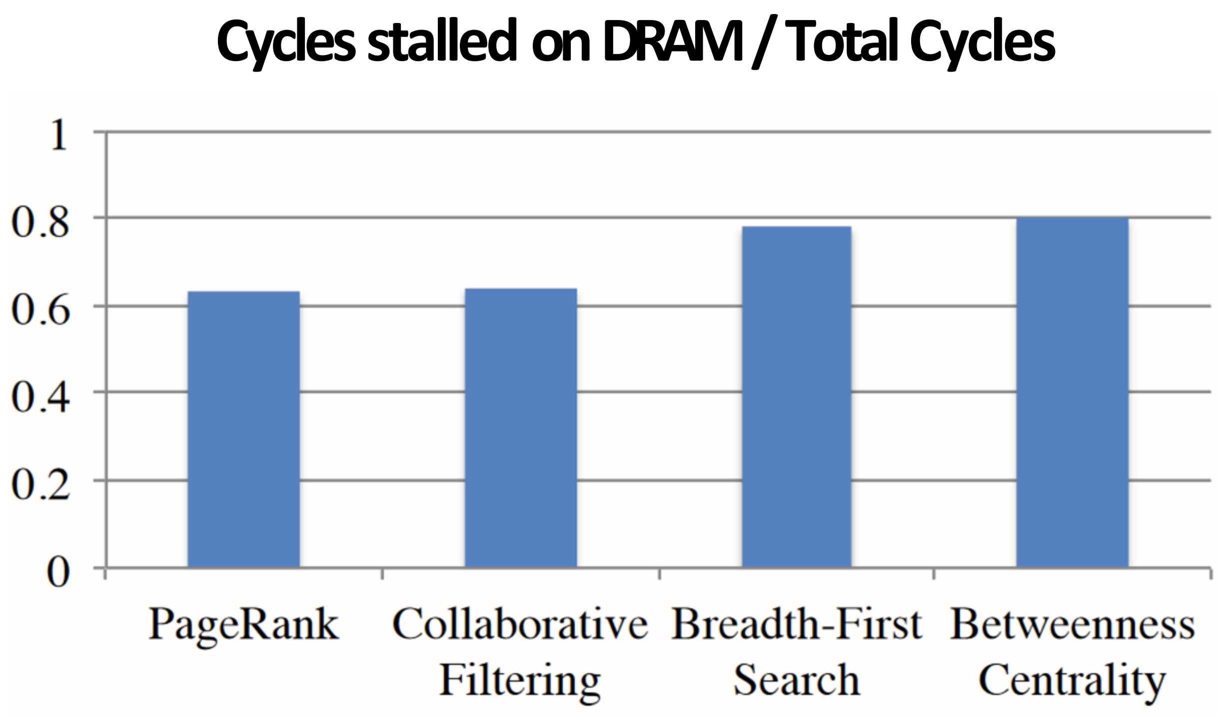


Cache miss latency *cannot be hidden by anything else in the program*. Each miss incurs DRAM latency!

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

Irregular Accesses Lead to Poor Locality

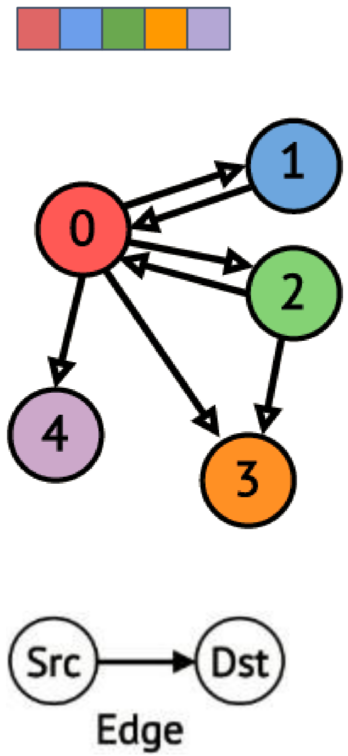




Problem: Sparse representations make processing large graphs feasible, but graph processing still entails a large working set with poor locality

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

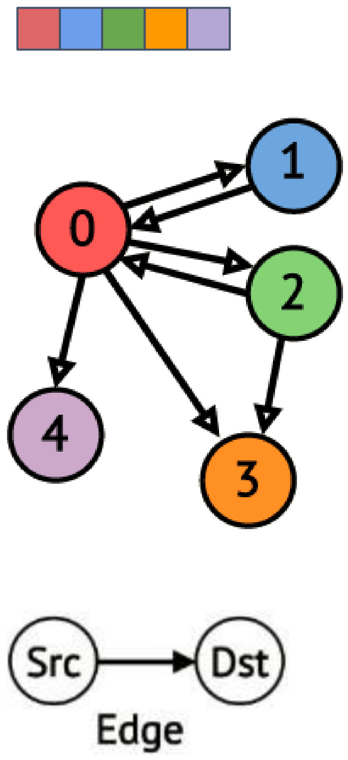
COO
(EdgeList)

```
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

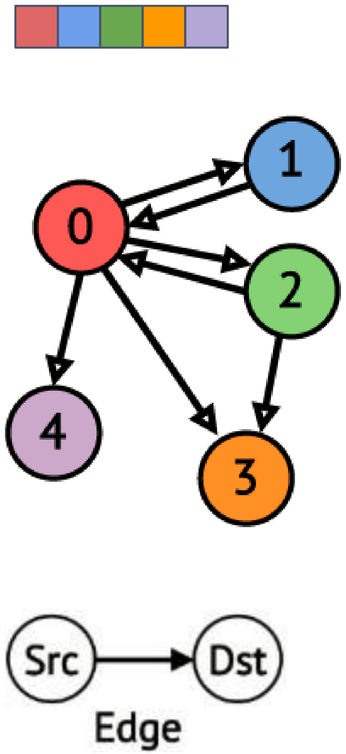
COO
(EdgeList)

```
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!



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
```

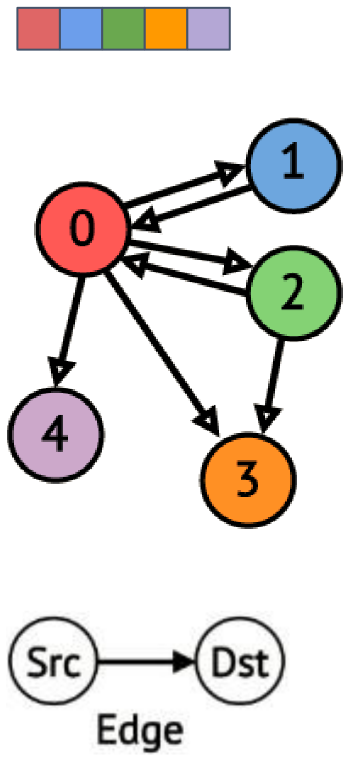
0	2	3	4	6	OA

1	2	0	0	0	2	0	NA

Completed CSC

Why is the NA update part irregular?

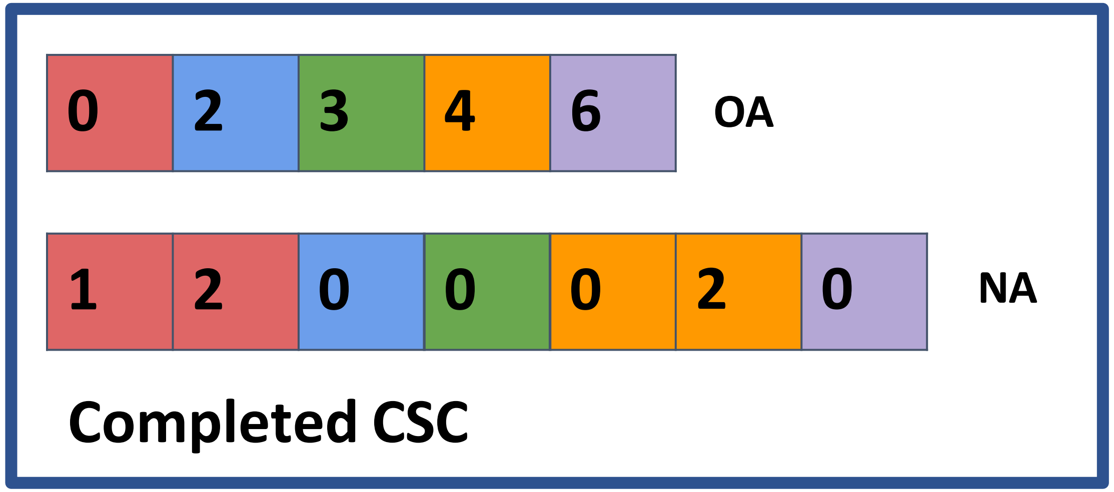
Even Building the CSR / CSC is an Irregular Access Pattern!



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
```

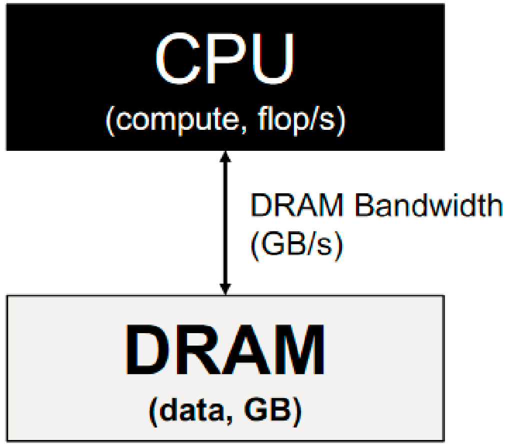


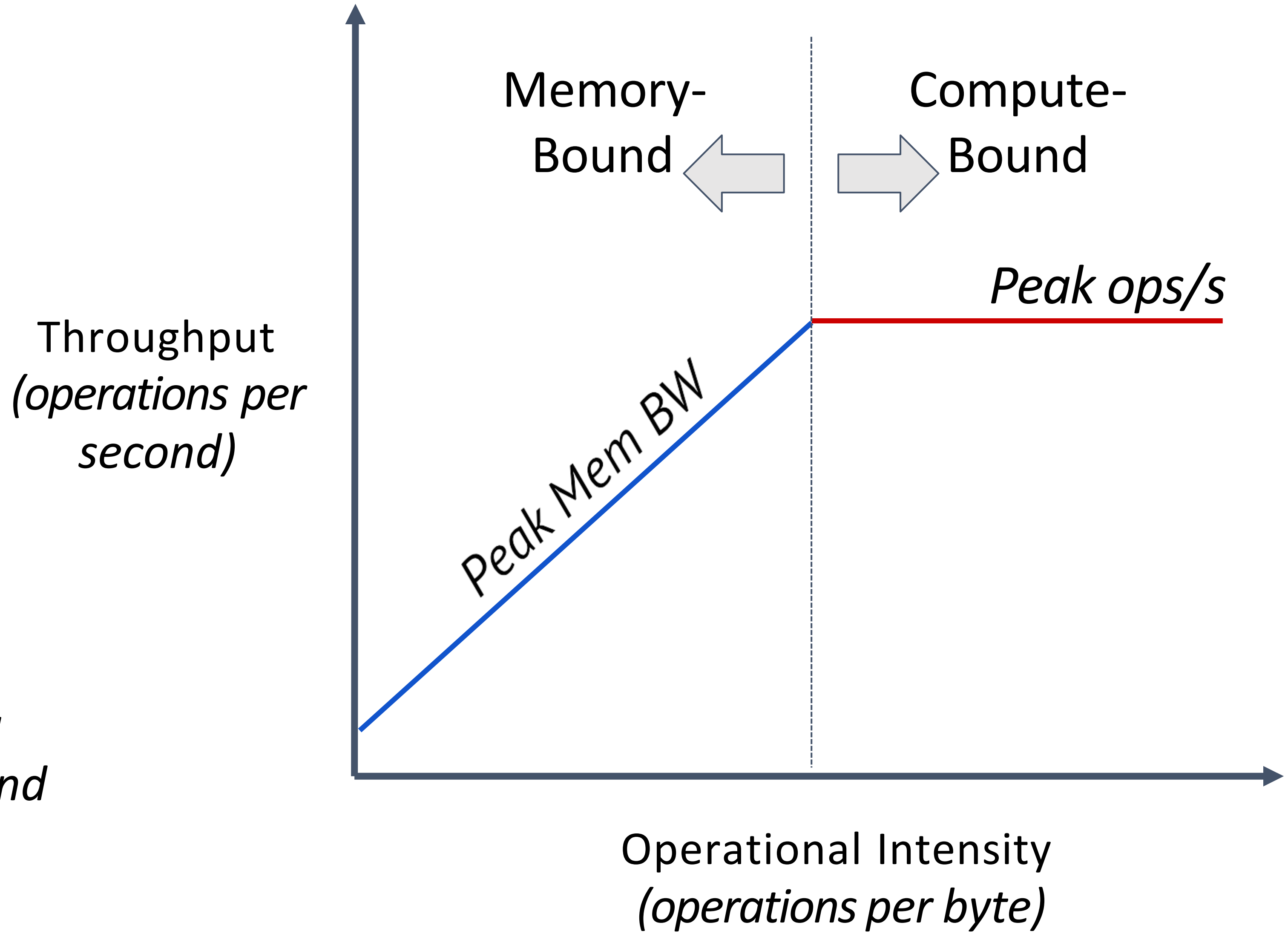
Updates to NA based on EL order & OA[e.src]

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

Roofline Performance Analysis of Graph Applications

The Roofline Model

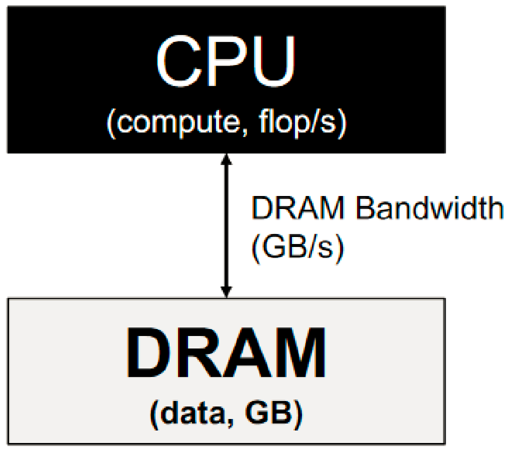


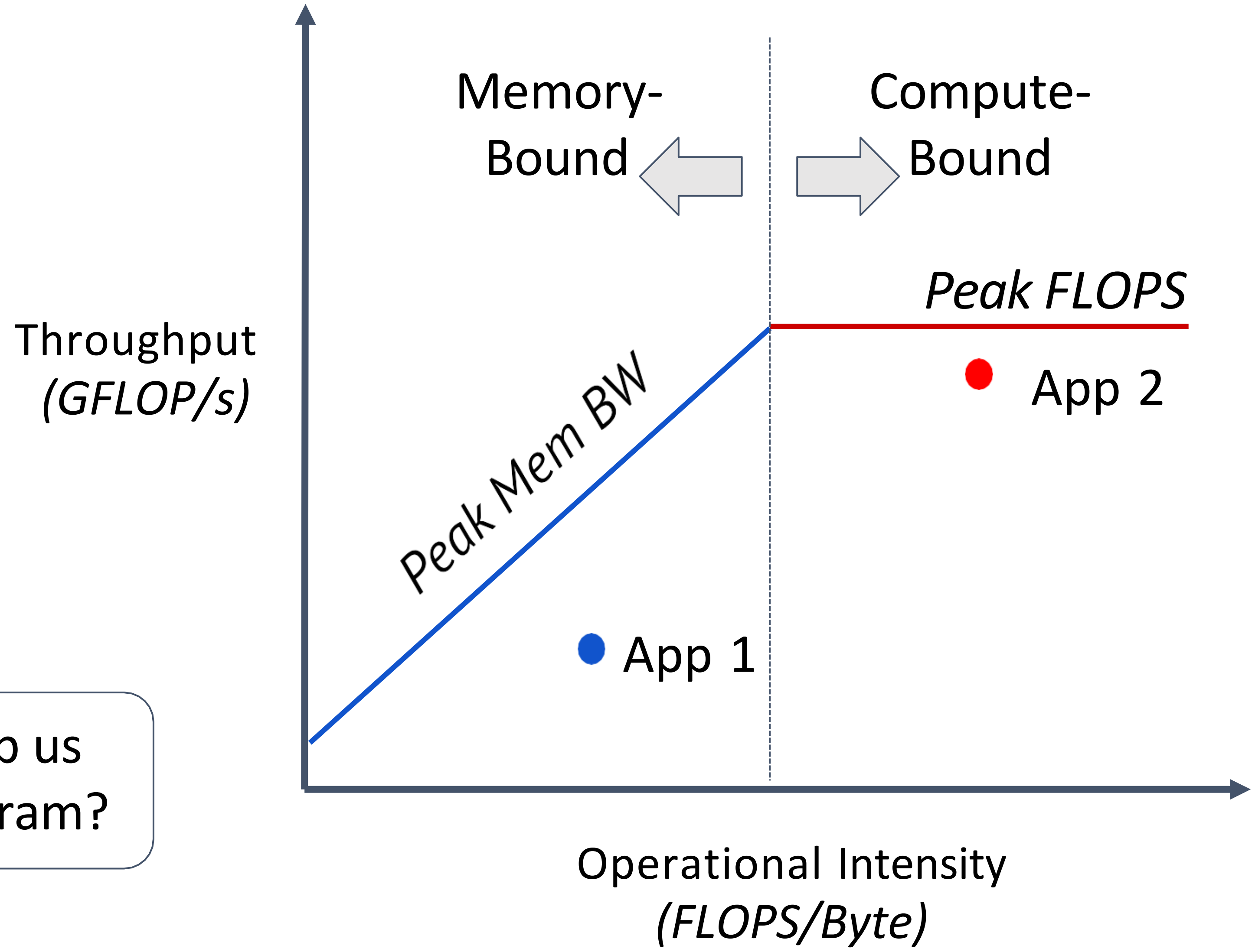


GFLOPS = Giga-Floating Point Operations Per Second

Yes, this is not a proper acronym

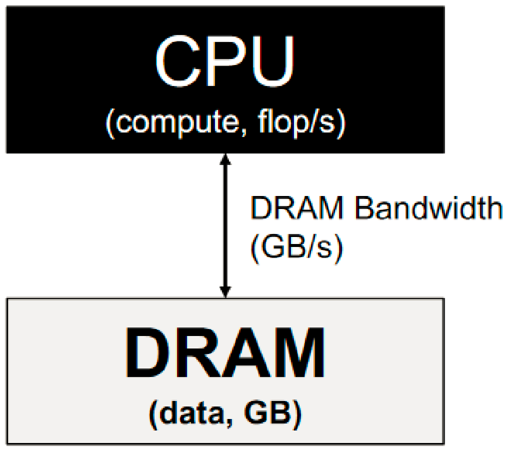
The Roofline Model

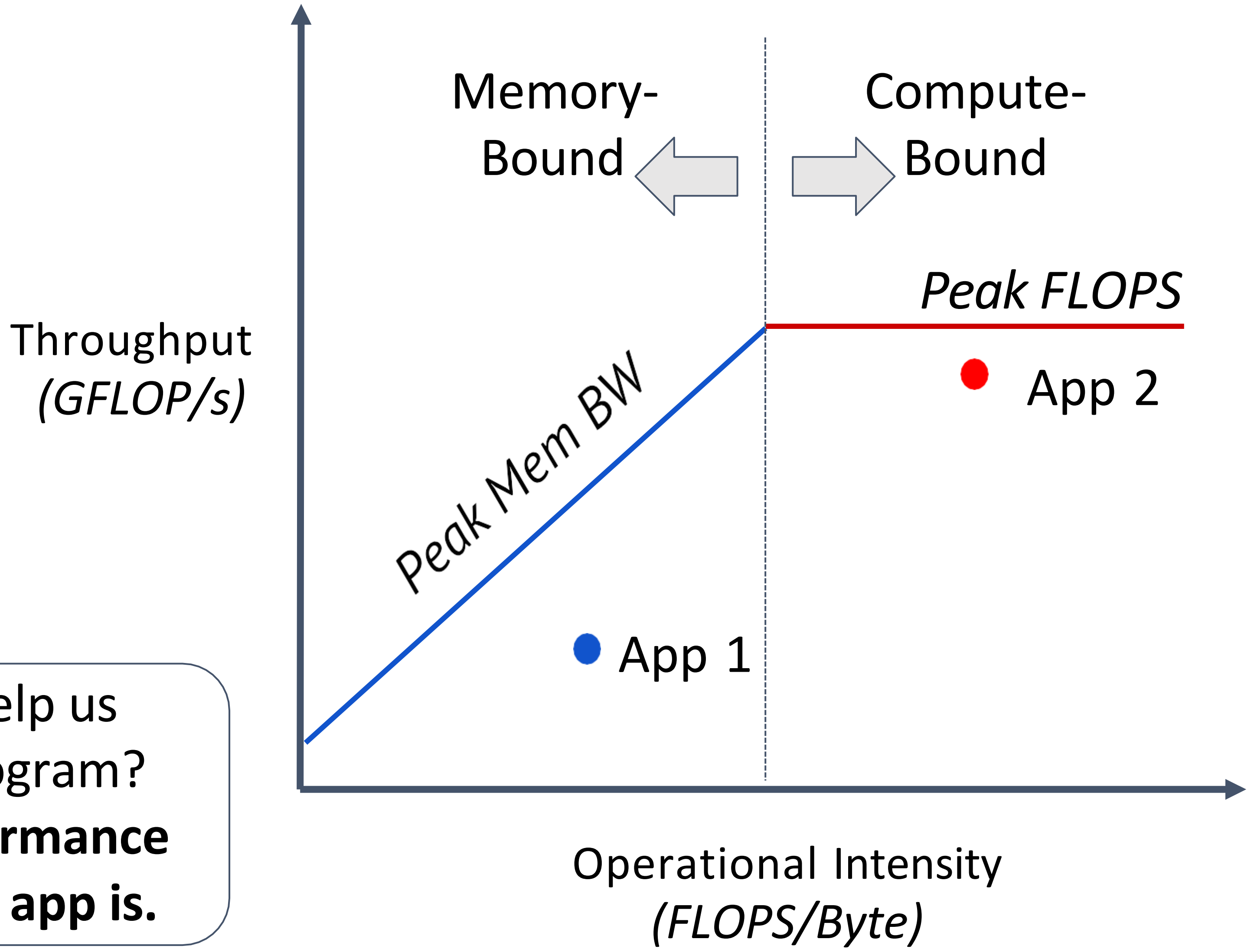




What does Roofline help us understand about a program?

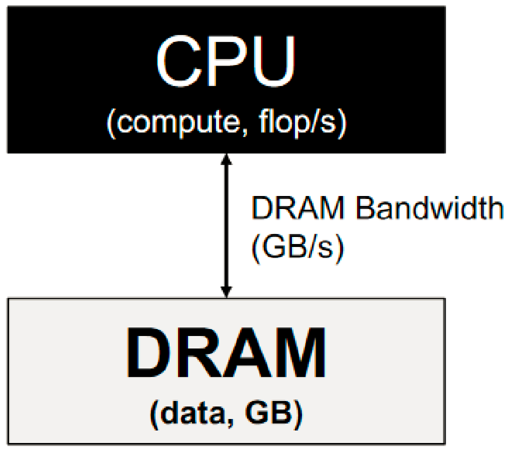
The Roofline Model

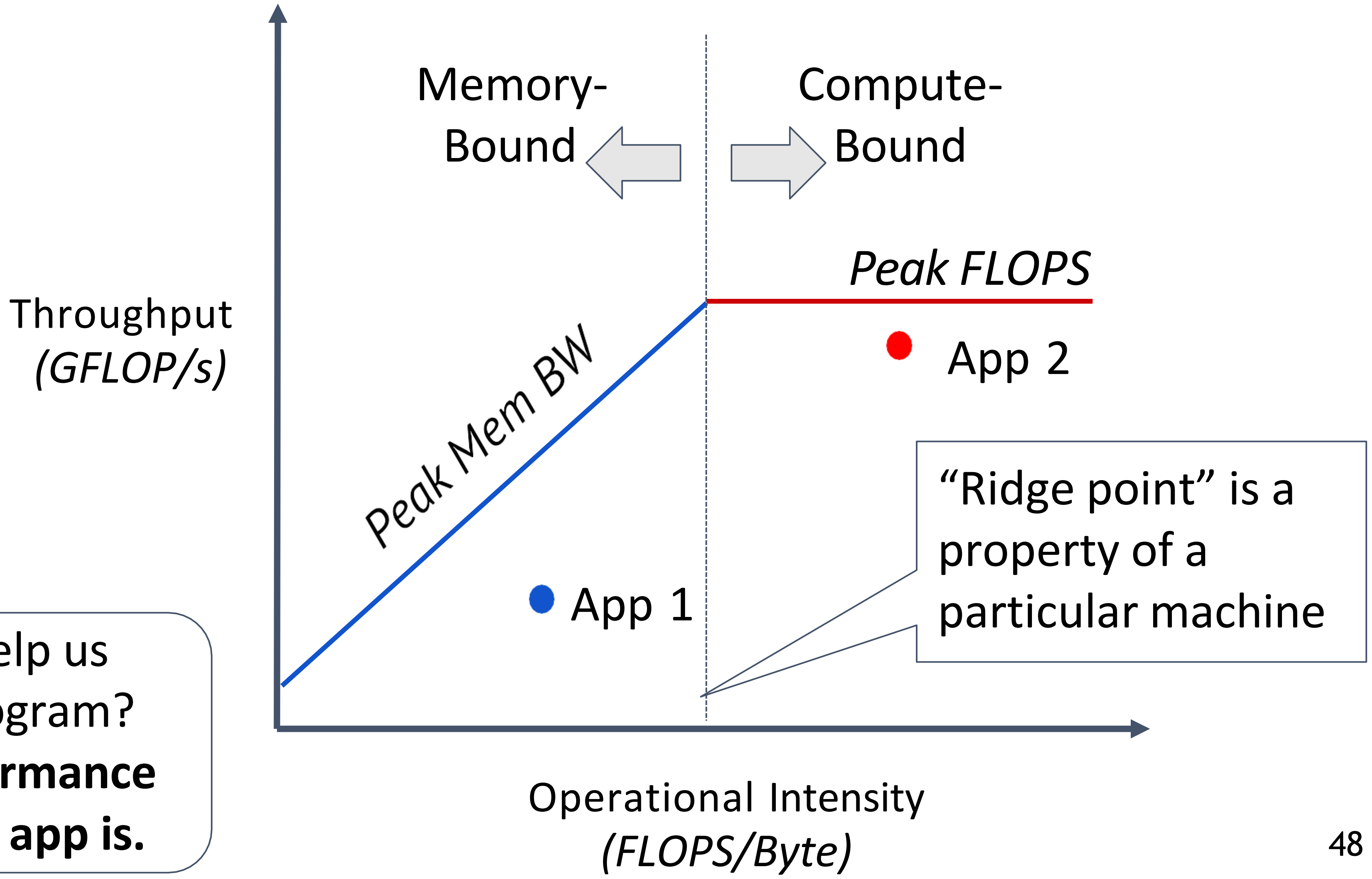




What does Roofline help us understand about a program?
Tell us what limits performance & how close to peak an app is.

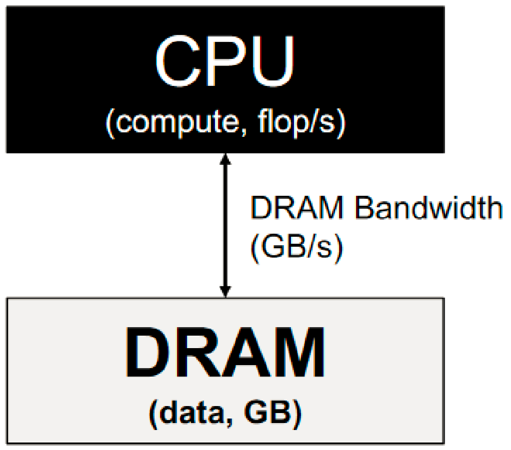
The Roofline Model

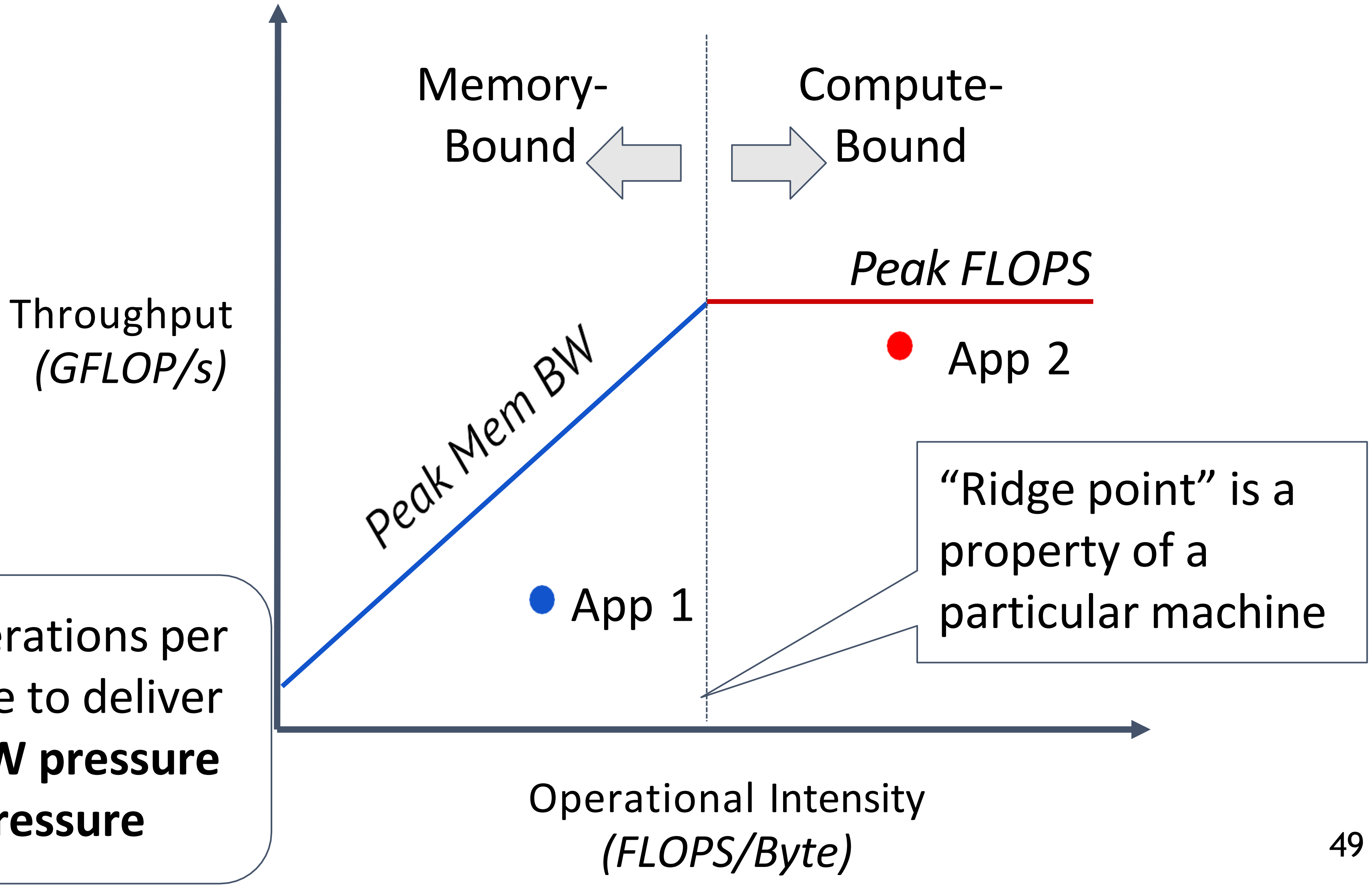




What does Roofline help us understand about a program?
Tell us what limits performance & how close to peak an app is.

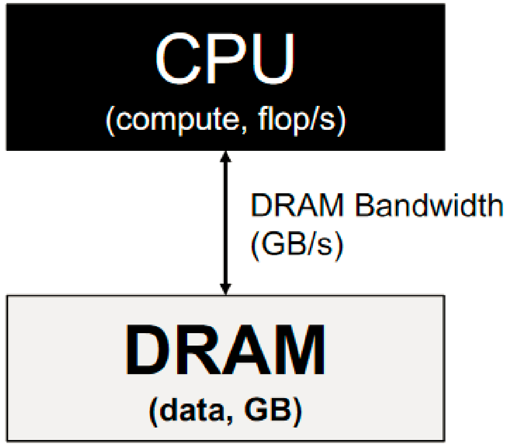
The Roofline Model



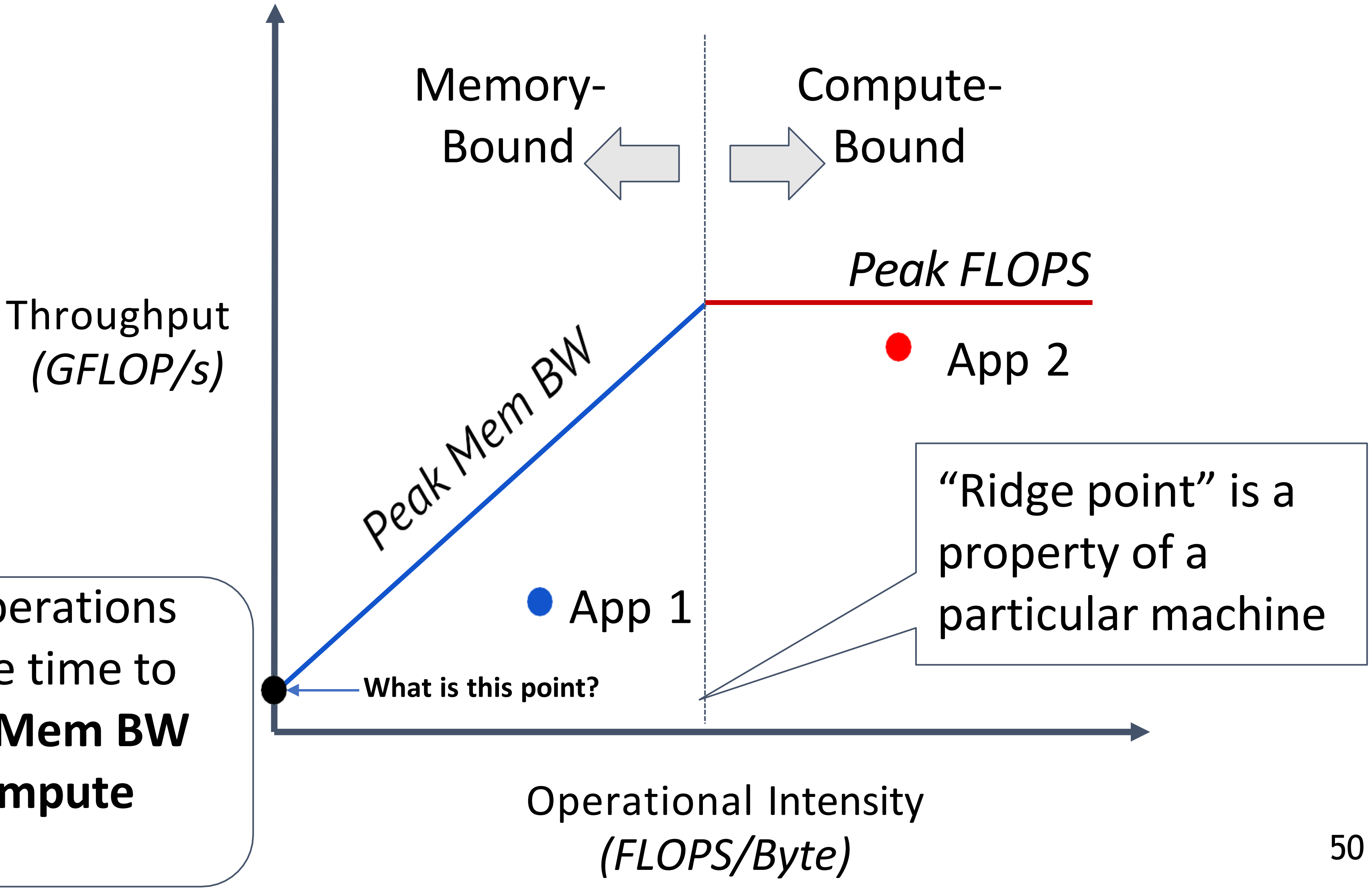


As a program does more operations per byte, memory has more time to deliver next byte**, relieving Mem BW pressure & increasing compute pressure**

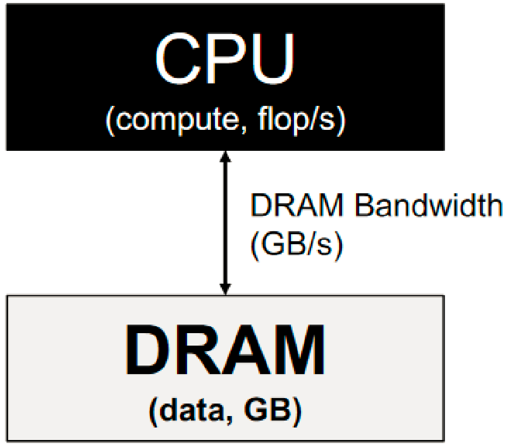
The Roofline Model

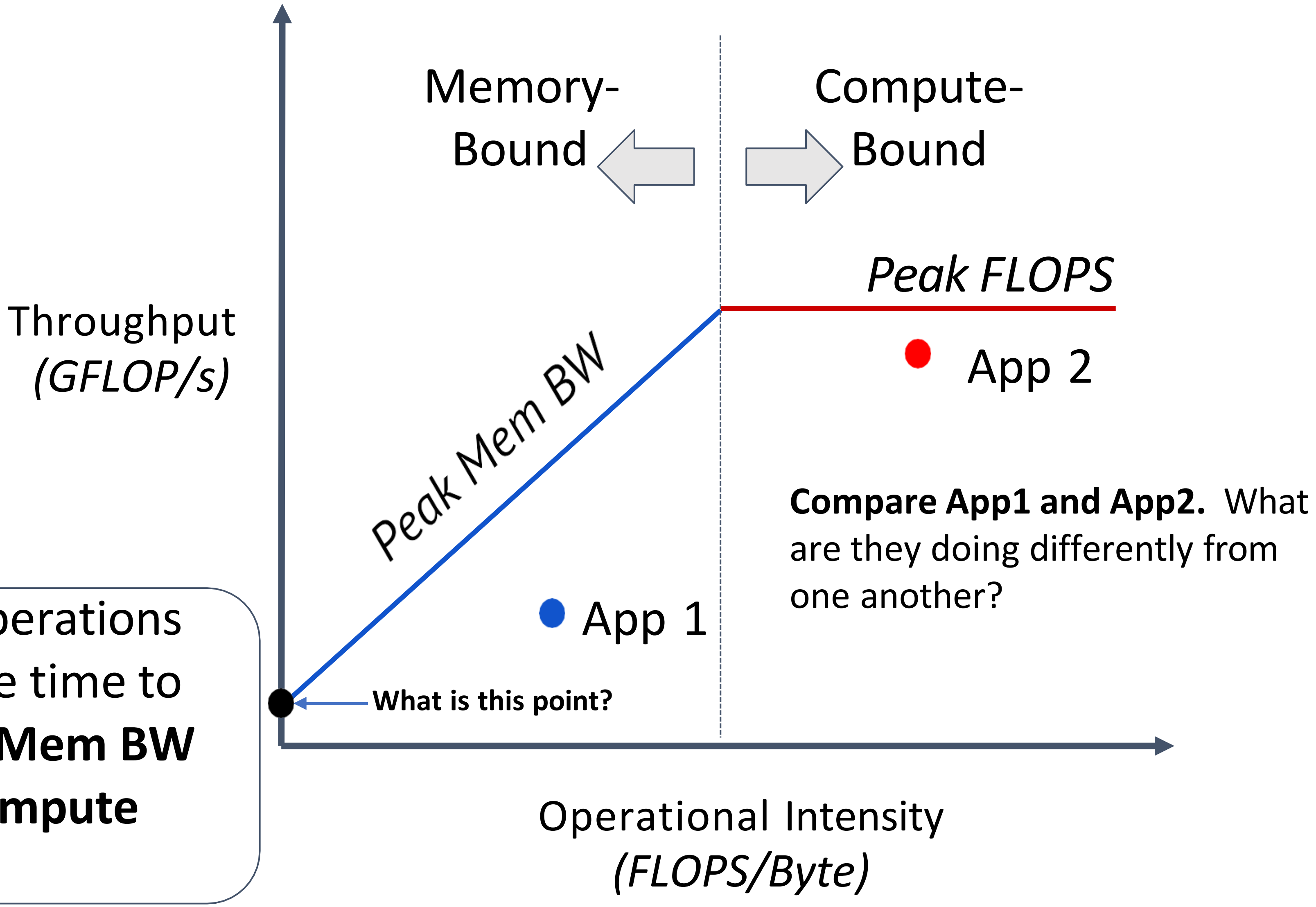


As a program does more operations per byte, memory has more time to deliver next byte**, relieving Mem BW pressure & increasing compute pressure**



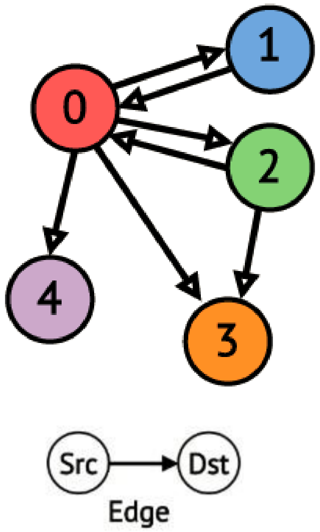
The Roofline Model





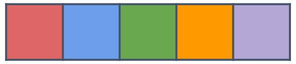
As a program does more operations per byte, memory has more time to deliver next byte**, relieving Mem BW pressure & increasing compute pressure**

Operational Intensity of Irregular Graph Applications



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

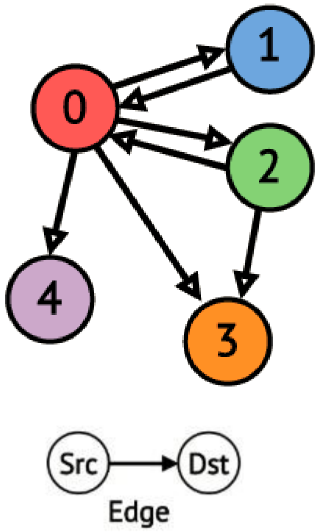
COO
(EdgeList)



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

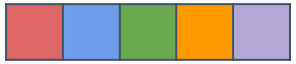
What is the operational intensity of a random update kernel like this one?

Operational Intensity of Irregular Graph Applications



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

COO
(EdgeList)

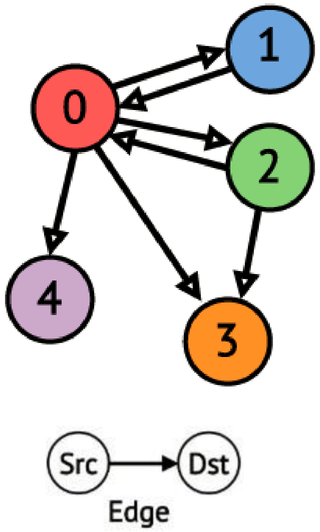


```
for e in EL:
  dstData[e.dst] += srcData[e.src]
```

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

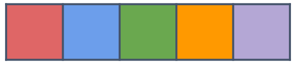
Operations per byte:

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?

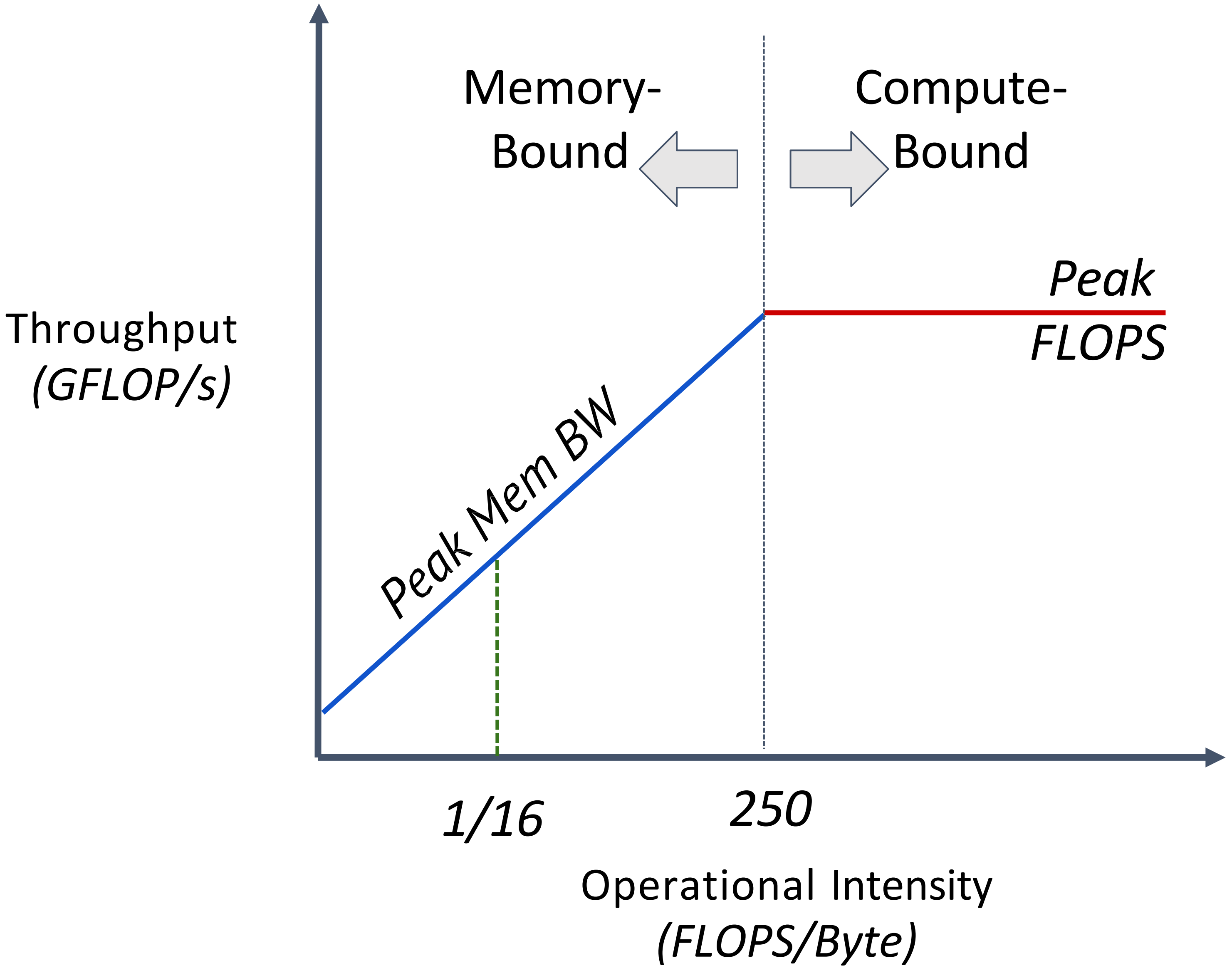
Operations per byte:

Operations: 1 addition

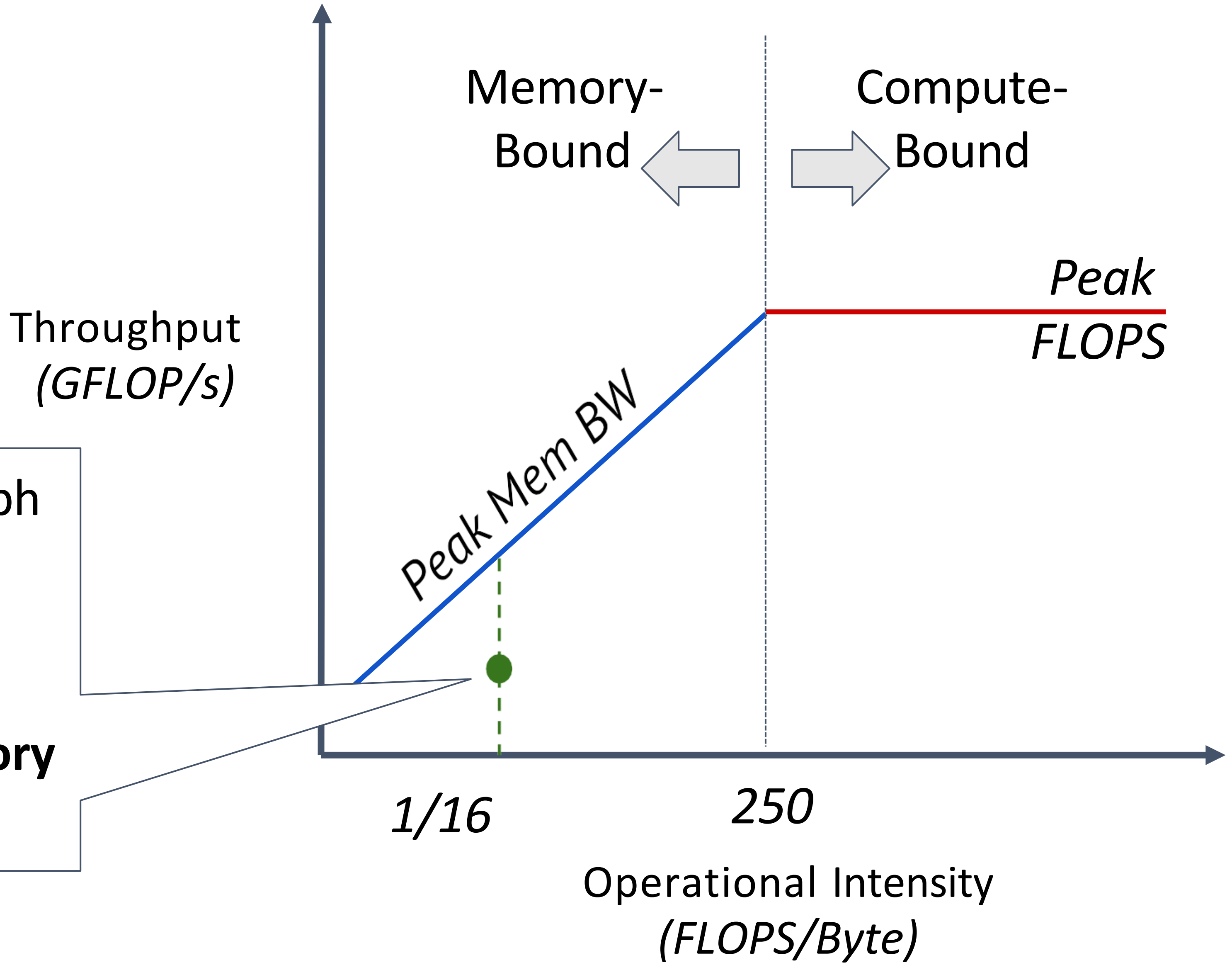
Bytes to Load: 8B for edge, 4B srcData, 4B dstData

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%

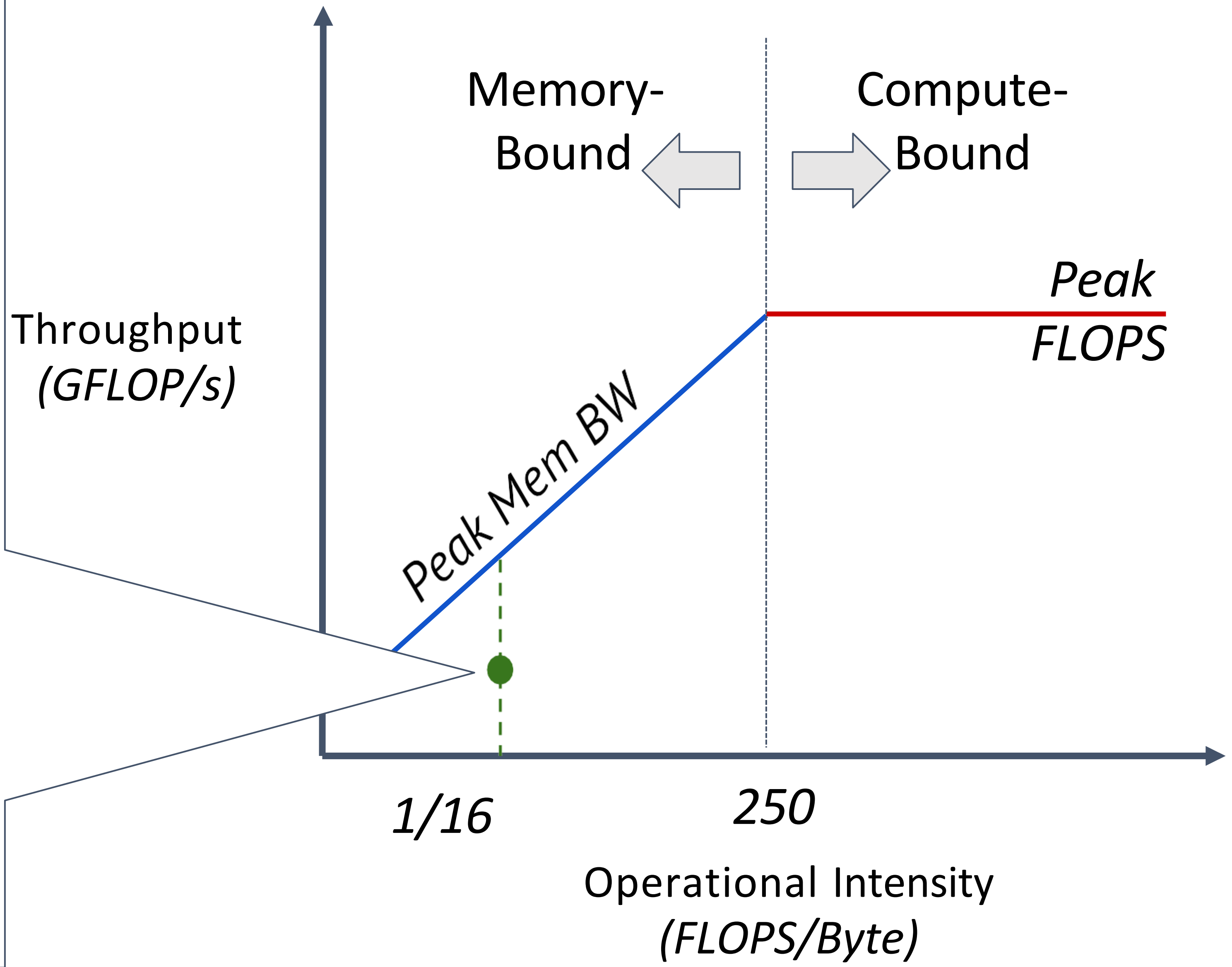
Why would we have spare BW capacity to go to memory and not use it?

Graph Applications are Memory-Bound

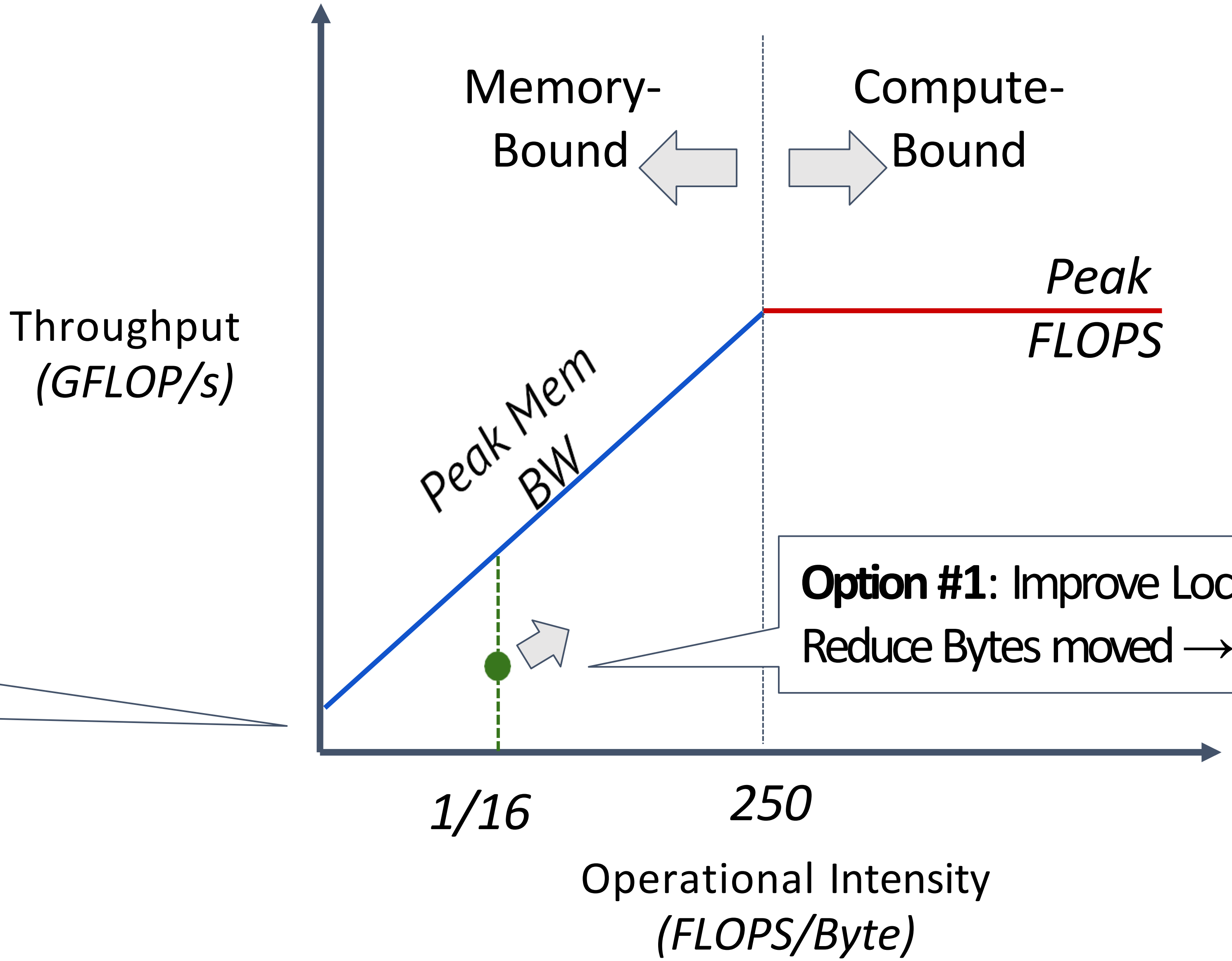
DRAM BW utilization in graph apps is ~50%

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).



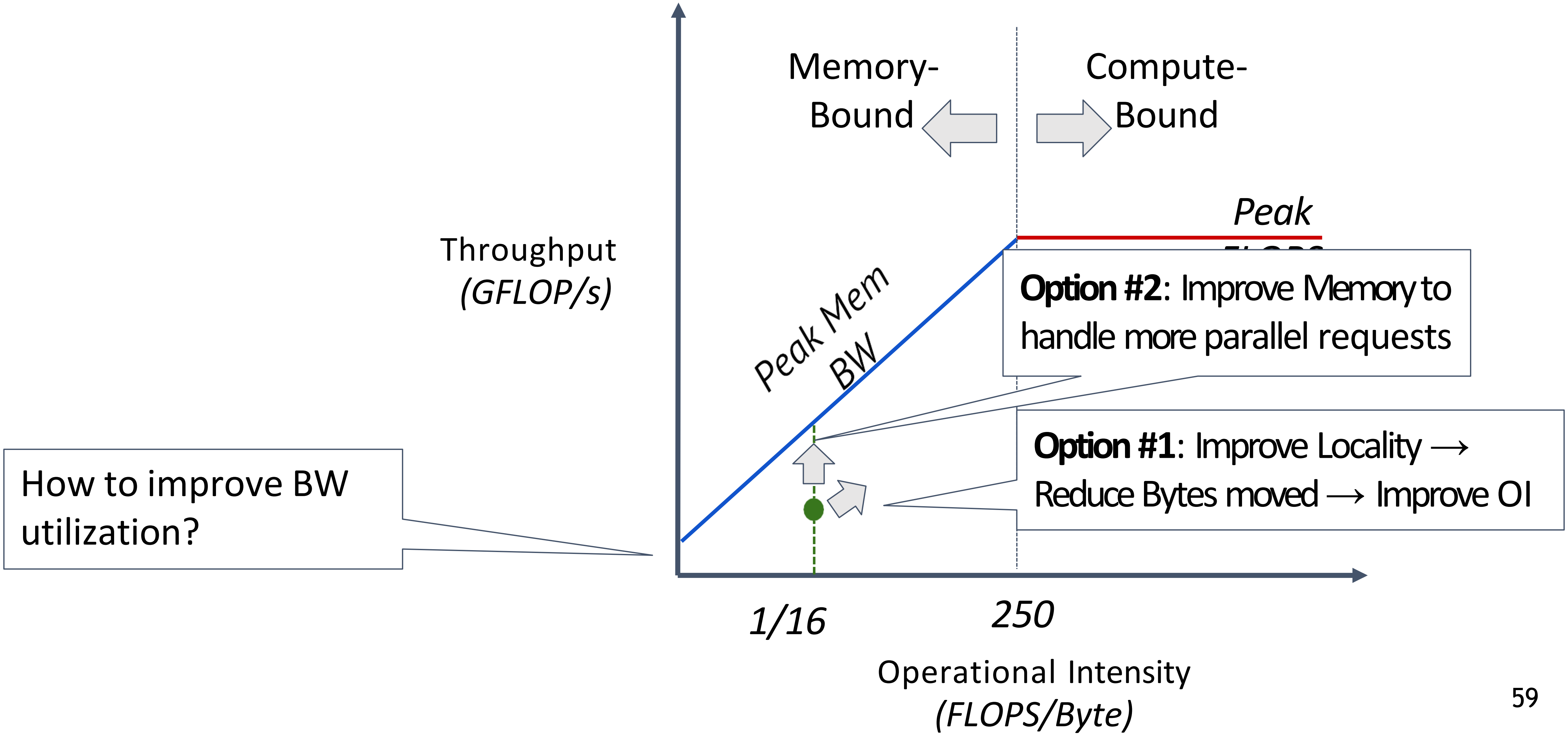
Graph Applications are Memory-Bound



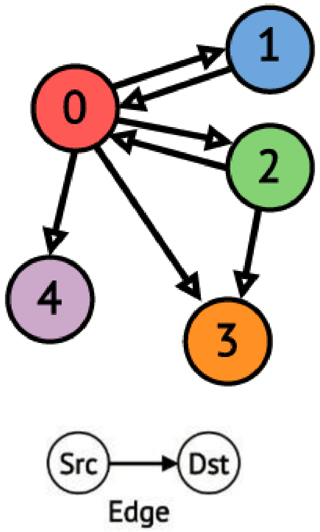
How to improve BW utilization?

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

Graph Applications are Memory-Bound

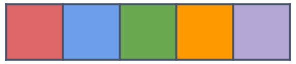


Operational Intensity of Irregular Graph Applications



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?

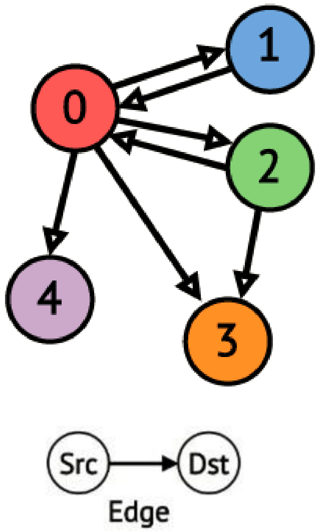
Operations per byte:

Operations: 1 addition

Bytes to Load:

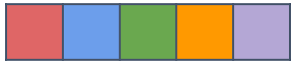
Operational Intensity =

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?

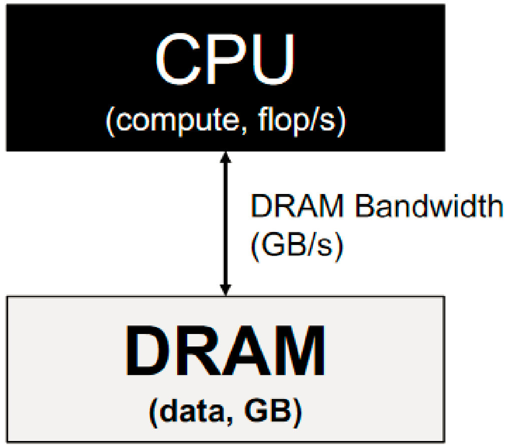
Operations per byte:

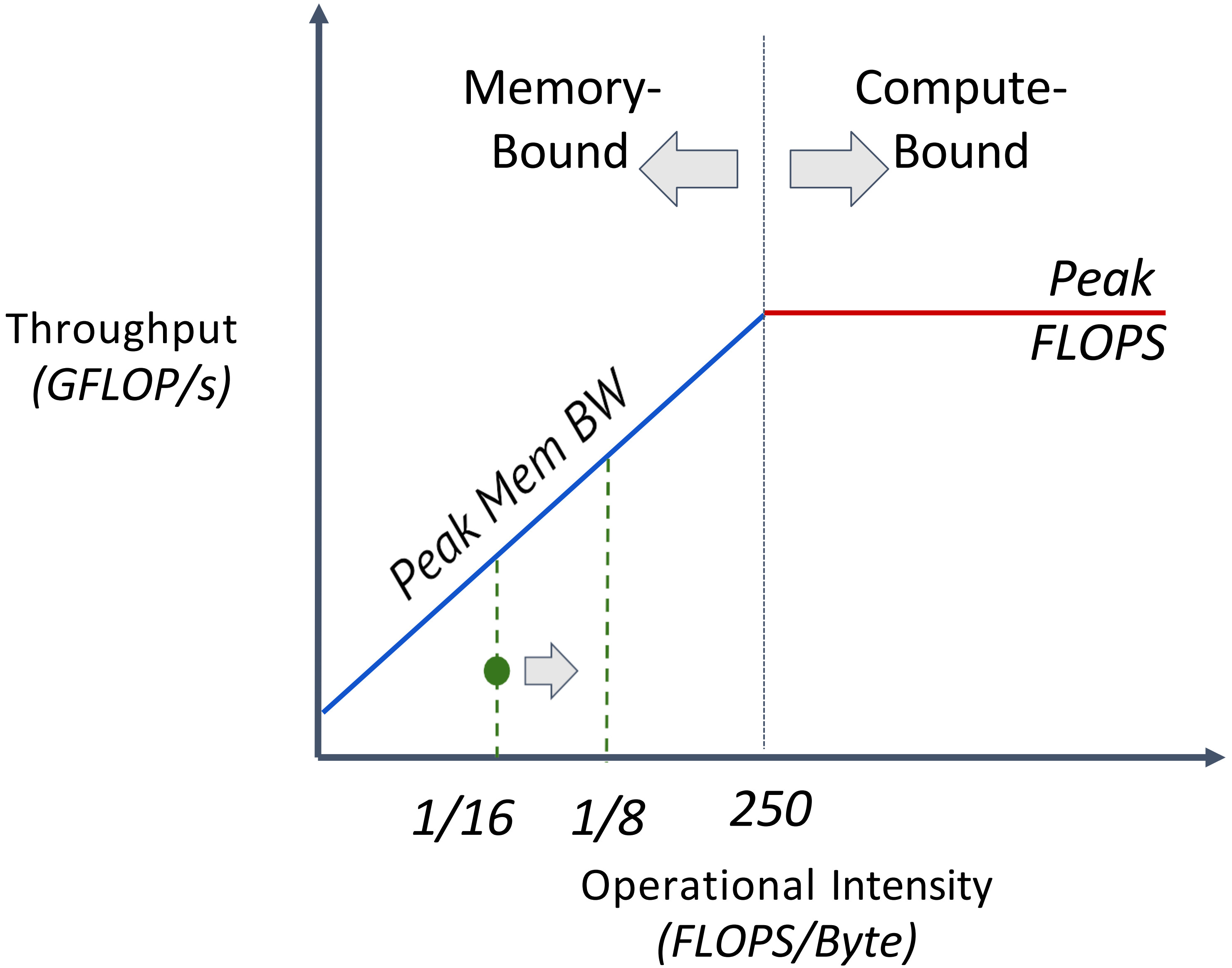
Operations: 1 addition

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

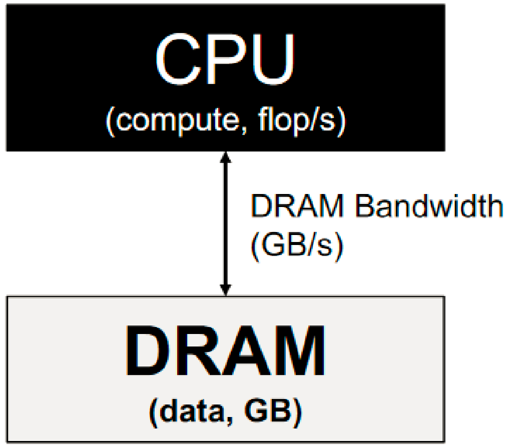
Operational Intensity = 1 / (8+0+0) **= 1/8**

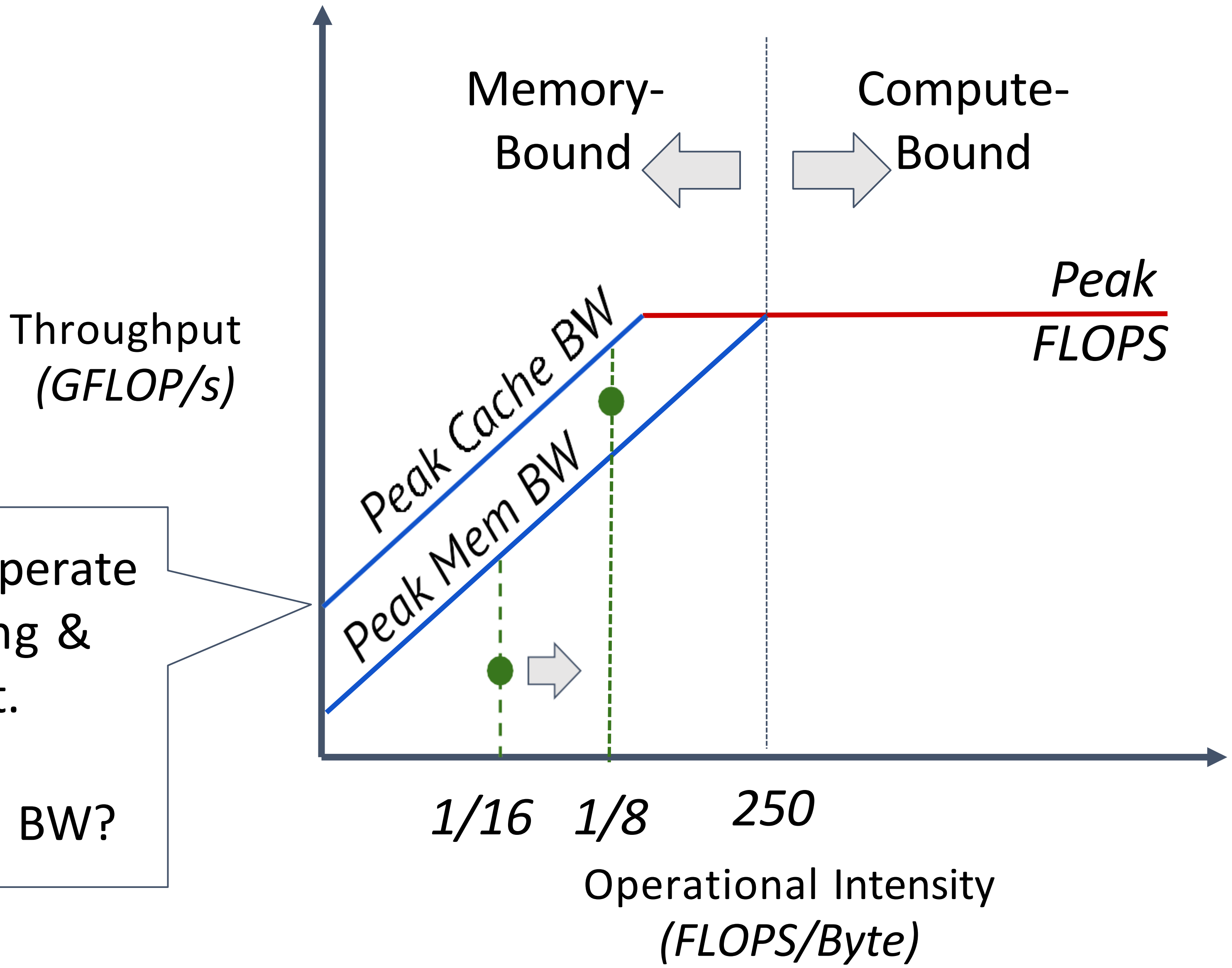
Improving Operational Intensity (OI) by Improving Locality





Improving Operational Intensity (OI) by Improving Locality

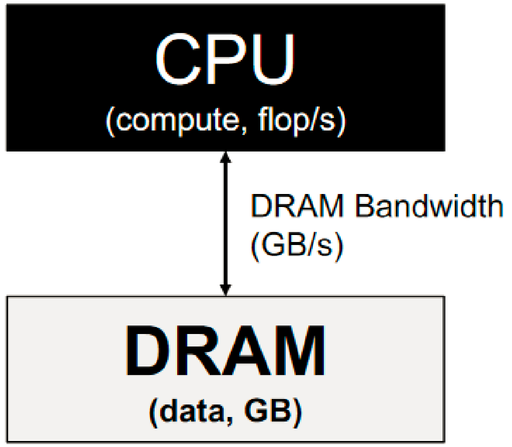


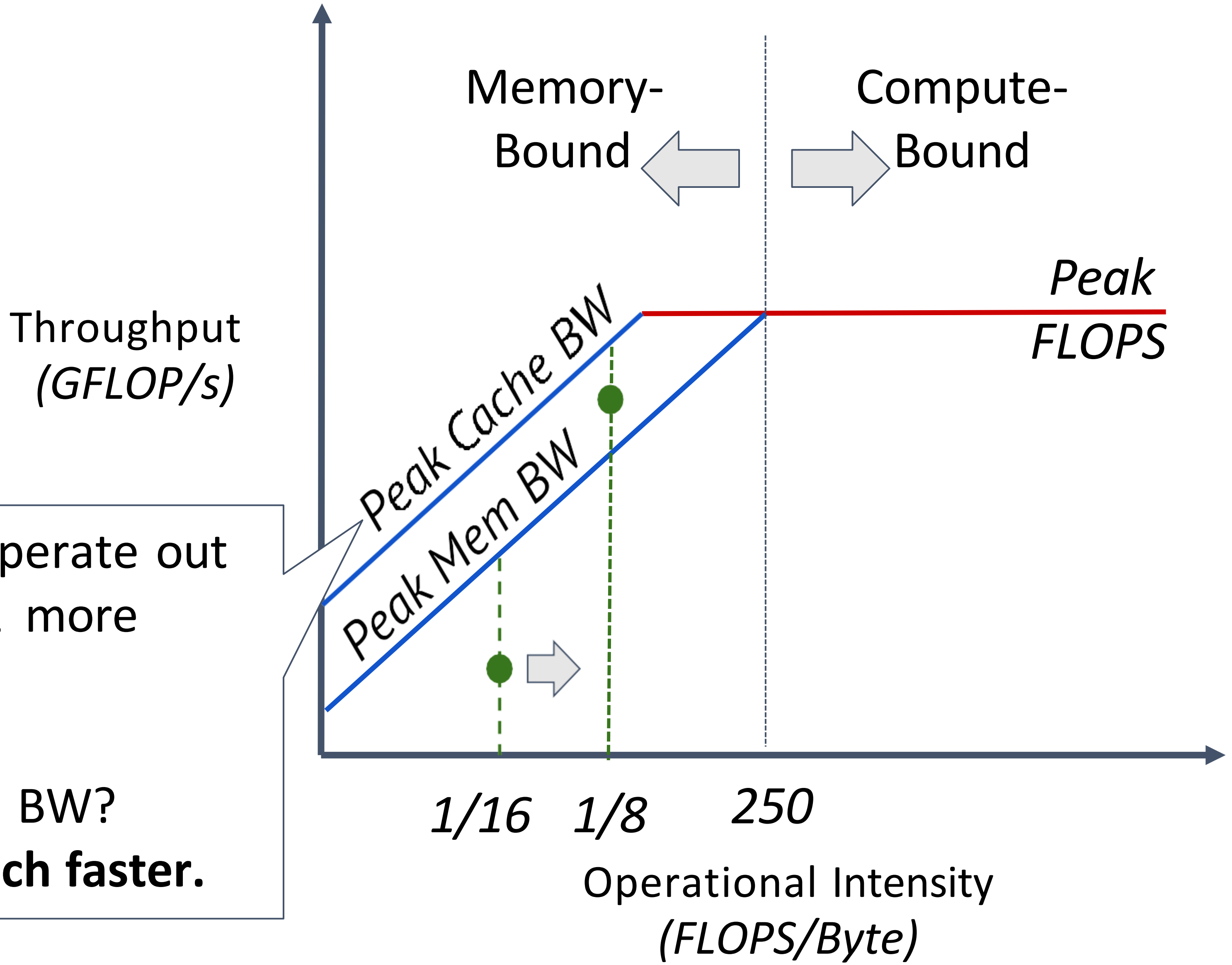


Locality wins: If we can operate out of cache, higher ceiling & more leftward ridge point.

Why is cache BW > DRAM BW?

Improving Operational Intensity (OI) by Improving Locality

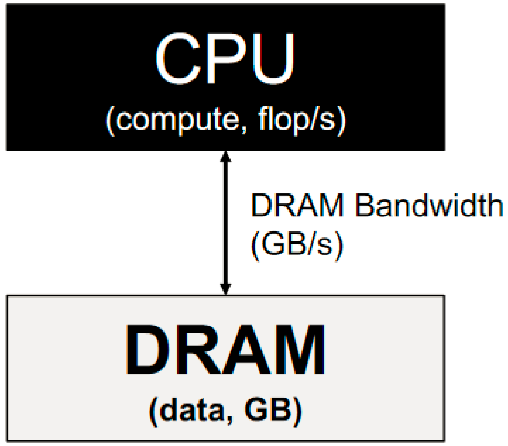




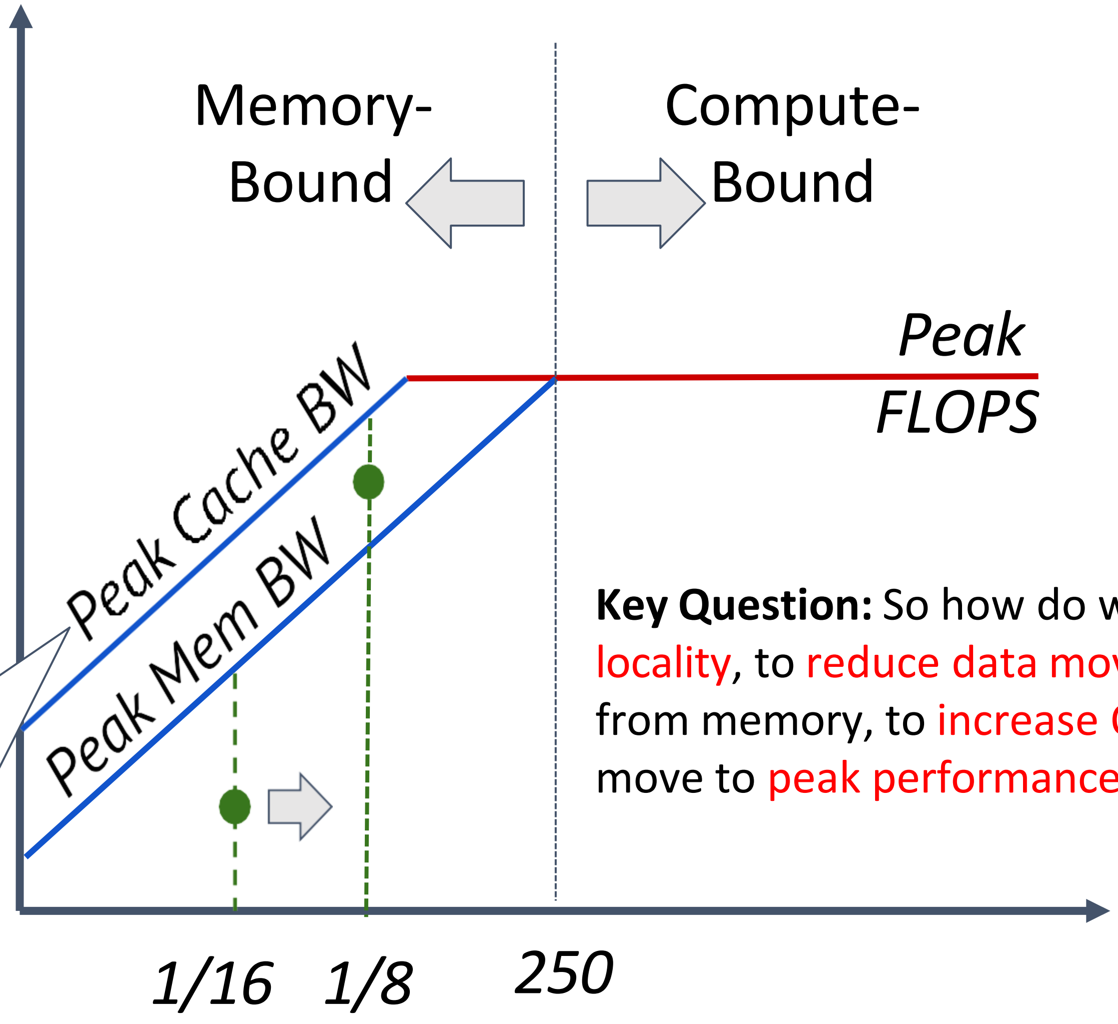
Locality wins: If we can operate out of cache, higher ceiling & more leftward ridge point.

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

Improving Operational Intensity (OI) by Improving Locality



Throughput *(GFLOP/s)*



Operational Intensity *(FLOPS/Byte)*

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.

Key Question: So how do we improve locality, to reduce data movement from memory, to increase OI, and move to peak performance?

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

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