CuSha: Vertex-Centric Graph Processing on GPUs

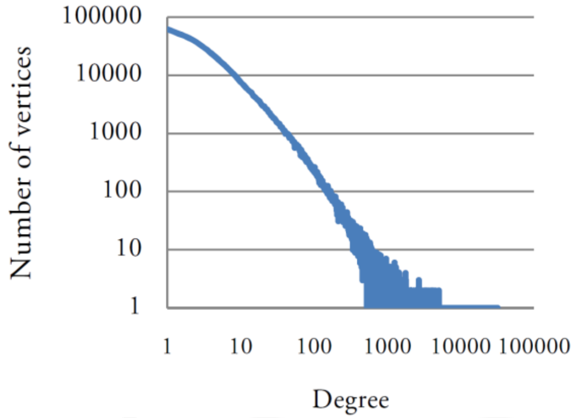
Farzad Khorasani, **Keval Vora**,
Rajiv Gupta, Laxmi N. Bhuyan

HPDC'14 – Vancouver, Canada
26 June, 2014

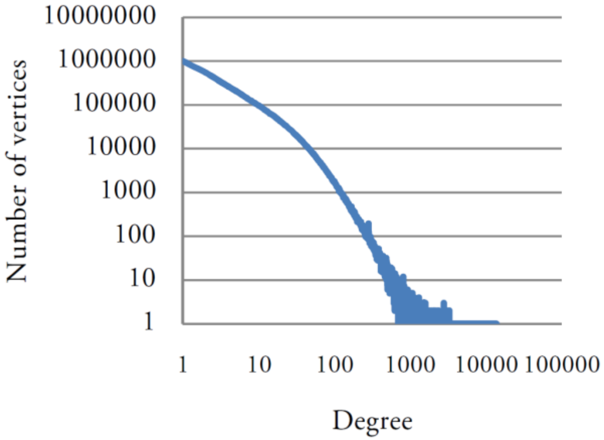
UCR
UNIVERSITY OF CALIFORNIA, RIVERSIDE

Motivation

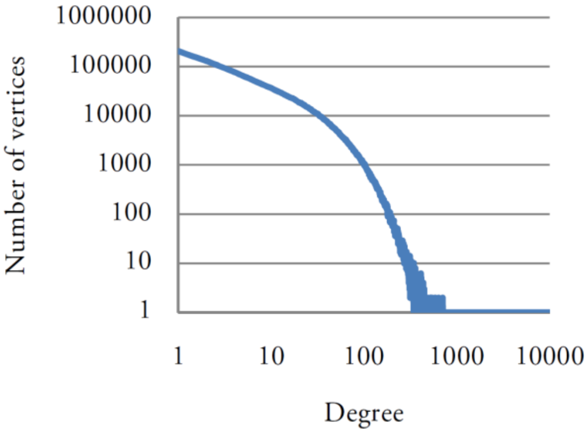
- Graph processing
- Real world graphs are large & sparse
- Power law distribution



HiggsTwitter
15M Edges, 0.45M Vertices



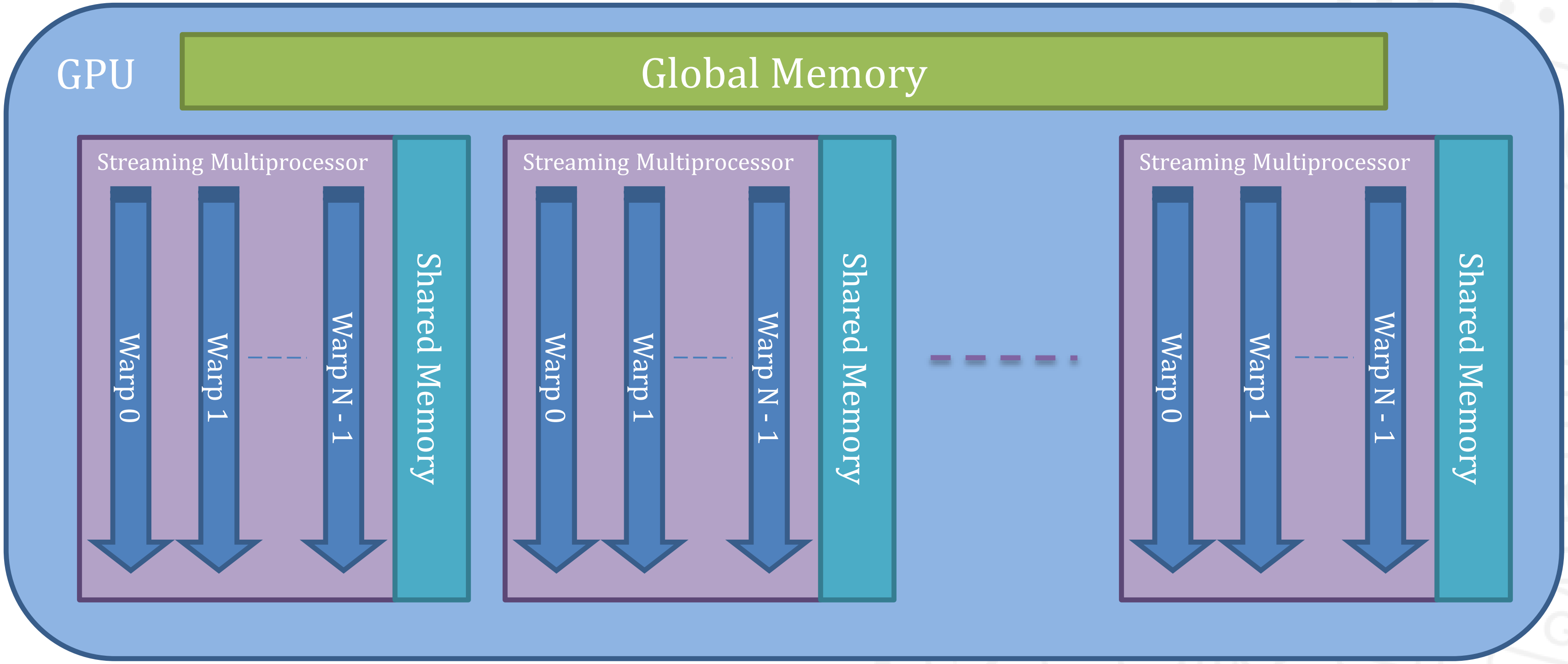
LiveJournal
69M Edges, 5M Vertices



Pokec
30M Edges, 1.6M Vertices

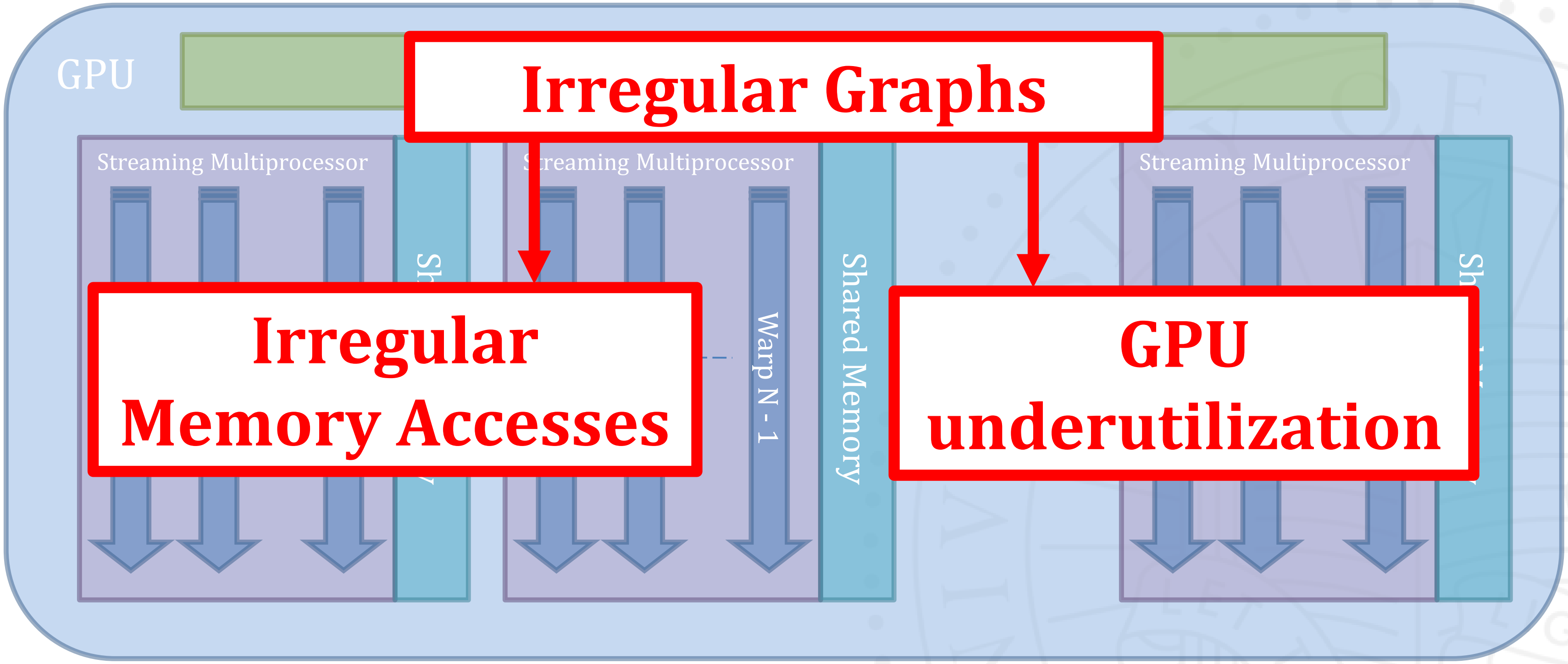
Graphics Processing Unit (GPU)

- Single Instruction Multiple Data (SIMD)
- Repetitive processing patterns on regular data



Graphics Processing Unit (GPU)

- Single Instruction Multiple Data (SIMD)
- Repetitive processing patterns on regular data



Irregular Graphs

Irregular Memory Accesses

GPU underutilization

Prior Work

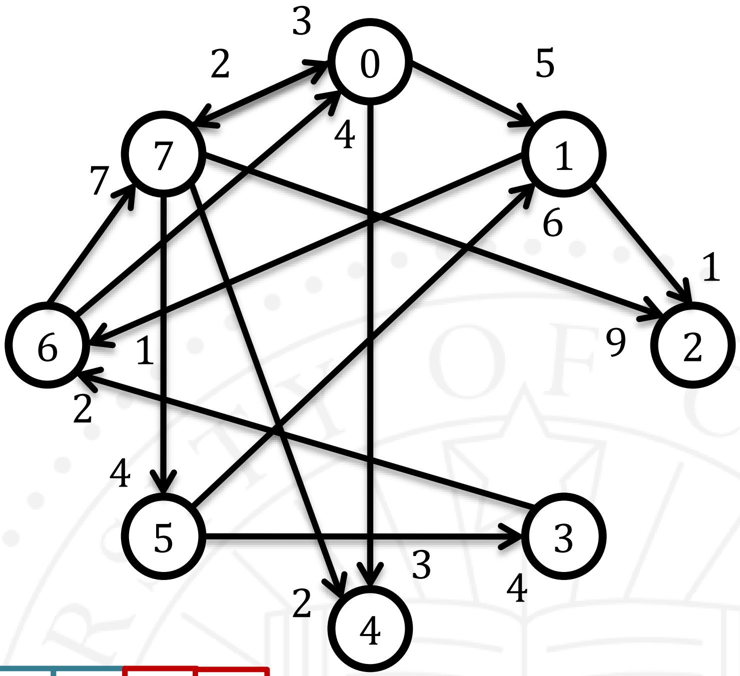
- Using **CSR** assign vertex to a thread [Harish and Narayanan HiPC'07]
  - Threads iterate through assigned vertex's neighbors
  - Non-coalesced memory accesses
  - Work imbalance among threads
- Using **CSR** assign vertex to a virtual warp [Hong et al, PPoPP'11]
  - Virtual warp lanes process vertex's neighbors in parallel
  - Non-coalesced memory accesses
  - Work imbalance reduced but still exist
- Using **CSR** assemble frontiers [Merrill et al, PPoPP'12]
  - Exploration based graph algorithms
  - Non-coalesced memory accesses

Prior Work

- Using **CSR** assign vertex to a thread [Harish and Narayanan HiPC’07]
  - Threads iterate through assigned vertex’s neighbors
  - Non-coalesced memory accesses
  - Work imbalance among threads
- Using **CSR** assign v ’11]
  - Virtual warp lanes process vertex’s neighbors in parallel
  - Non-coalesced memory accesses
  - Work imbalance reduced but still exist
- Using **CSR** assemble frontiers [Merrill et al, PPoPP’12]
  - Exploration based graph algorithms
  - Non-coalesced memory accesses

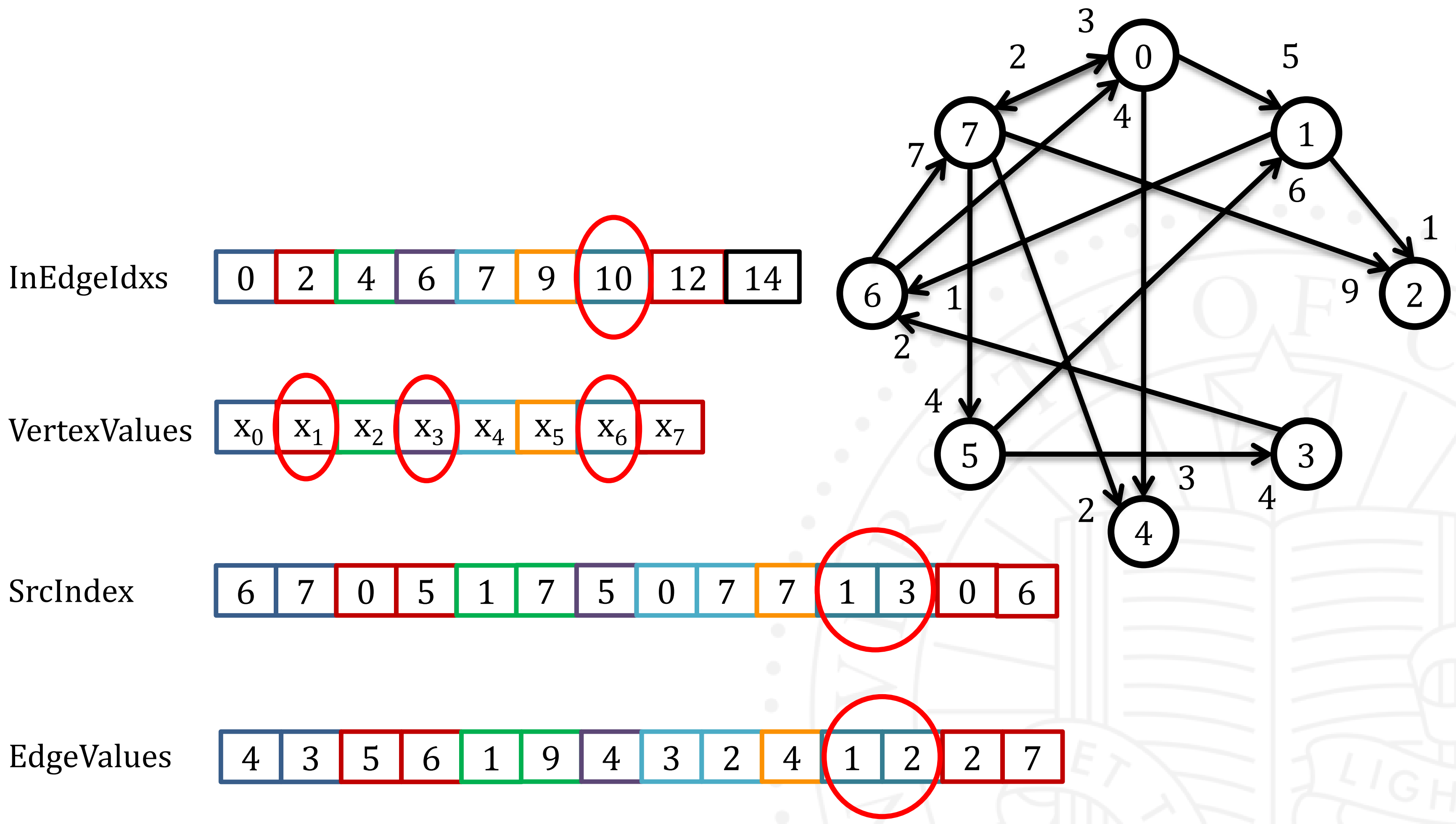
Compressed Sparse Row

Compressed Sparse Row (CSR)



InEdgeIdxs	0	2	4	6	7	9	10	12	14					
VertexValues	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7						
SrcIndex	6	7	0	5	1	7	5	0	7	7	1	3	0	6
EdgeValues	4	3	5	6	1	9	4	3	2	4	1	2	2	7

Compressed Sparse Row (CSR)



InEdgeIdxs 0 2 4 6 7 9 10 12 14

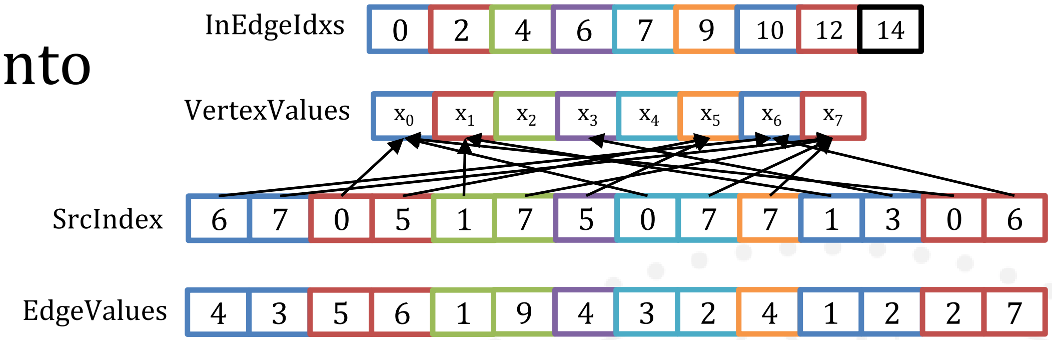
VertexValues x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7

SrcIndex 6 7 0 5 1 7 5 0 7 7 1 3 0 6

EdgeValues 4 3 5 6 1 9 4 3 2 4 1 2 2 7

Virtual Warp Centric (VWC) [PPoPP’12]

- Physical warp broken into smaller virtual warps
  - 2, 4, 8, 16 lanes
- Advantages:
  - Neighbors processed in parallel
  - Load imbalance is reduced
- Disadvantages:
  - Load imbalance still exists – GPU underutilization
  - Non-coalesced memory accesses



InEdgeIdxs 0 2 4 6 7 9 10 12 14

VertexValues x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7

SrcIndex 6 7 0 5 1 7 5 0 7 7 1 3 0 6

EdgeValues 4 3 5 6 1 9 4 3 2 4 1 2 2 7

VWC-CSR

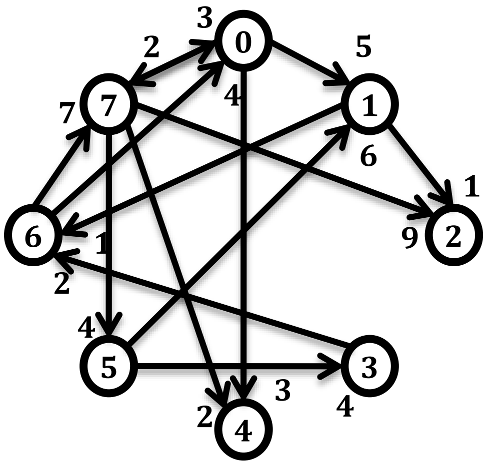
Benchmark	Avg global memory access efficiency (%)	Avg warp execution efficiency (%)
Breadth-First Search (BFS)	12.8-15.8	27.8-38.5
Single Source Shortest Path (SSSP)	14.0-19.6	29.7-39.4
PageRank (PR)	10.4-14.0	25.3-38.0
Connected Components (CC)	12.7-20.6	29.9-35.5
Single Source Widest Path (SSWP)	14.5-20.0	29.7-38.4
Neural Network (NN)	13.5-17.8	28.2-37.4
Heat Simulation (HS)	14.5-18.1	27.6-36.3
Circuit Simulation (CS)	12.0-18.8	28.4-35.5

Caused by **Non-Coalesced** memory loads and stores

Caused by **GPU Underutilization** and **Load Imbalance**

G-Shards

- Employs Shards [Kyrola et al, OSDI’12]
- Data required by computation placed contiguously
  - Improves locality



Shard 0

SrcIndex	SrcValue	EdgeValue	DestIndex
0	x_0	5	1
1	x_1	1	2
5	x_5	6	1
5	x_5	4	3
6	x_6	4	0
7	x_7	3	0
7	x_7	9	2

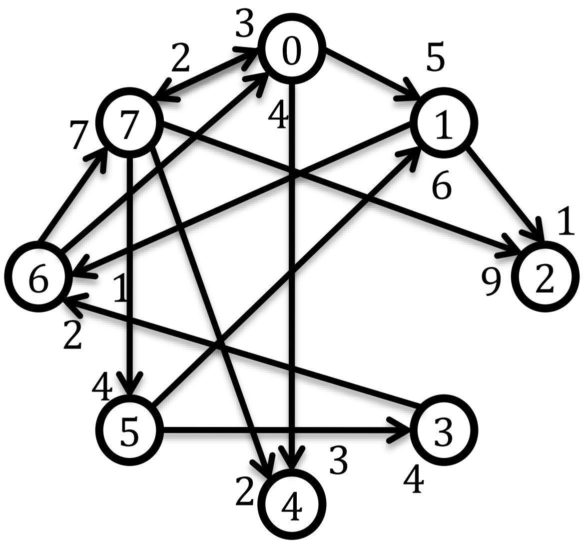
Shard 1

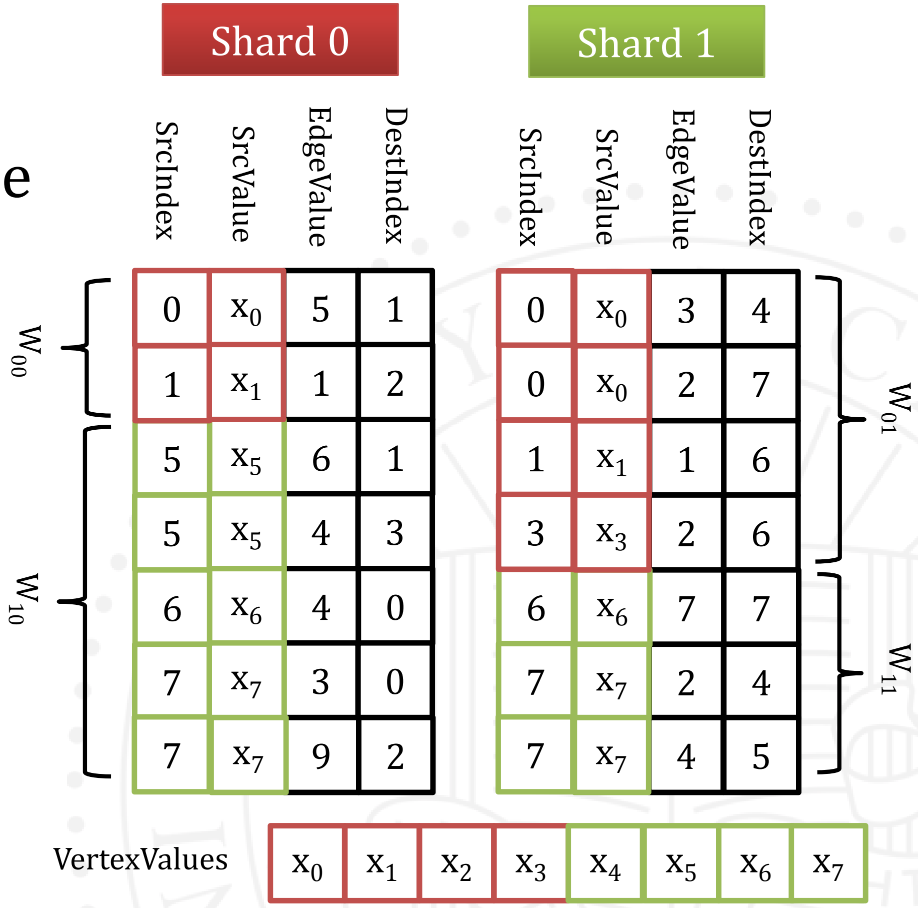
SrcIndex	SrcValue	EdgeValue	DestIndex
0	x_0	3	4
0	x_0	2	7
1	x_1	1	6
3	x_3	2	6
6	x_6	7	7
7	x_7	2	4
7	x_7	4	5

VertexValues	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

G-Shards: Construction

- Edges partitioned based on destination vertex's index
- Edges sorted based on source vertex's index





Shard 0

	SrcIndex	SrcValue	EdgeValue	DestIndex
W_{00}	0	x_0	5	1
	1	x_1	1	2
W_{10}	5	x_5	6	1
	5	x_5	4	3
	6	x_6	4	0
	7	x_7	3	0
	7	x_7	9	2

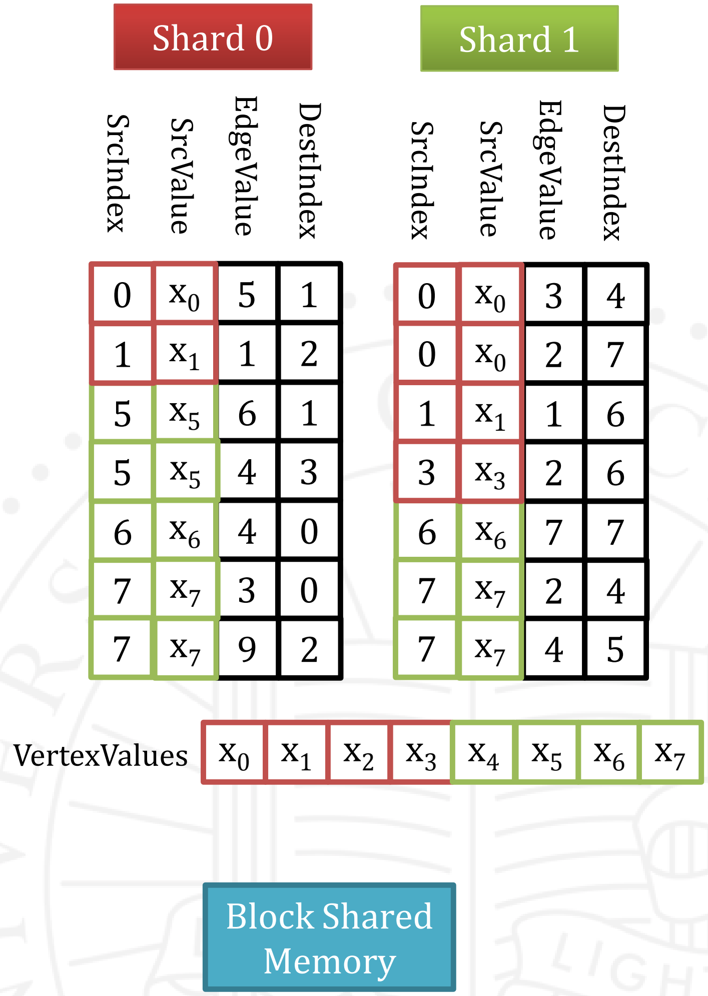
Shard 1

SrcIndex	SrcValue	EdgeValue	DestIndex	
0	x_0	3	4	W_{01}
0	x_0	2	7	
1	x_1	1	6	
3	x_3	2	6	
6	x_6	7	7	W_{11}
7	x_7	2	4	
7	x_7	4	5	

VertexValues							
x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

G-Shards: Processing

- Iteratively process all shards
- Each shard processed by a thread block
- 4 Steps:
  - Read
  - Compute
  - Update
  - Write



Shard 0

SrcIndex	SrcValue	EdgeValue	DestIndex
0	x_0	5	1
1	x_1	1	2
5	x_5	6	1
5	x_5	4	3
6	x_6	4	0
7	x_7	3	0
7	x_7	9	2

Shard 1

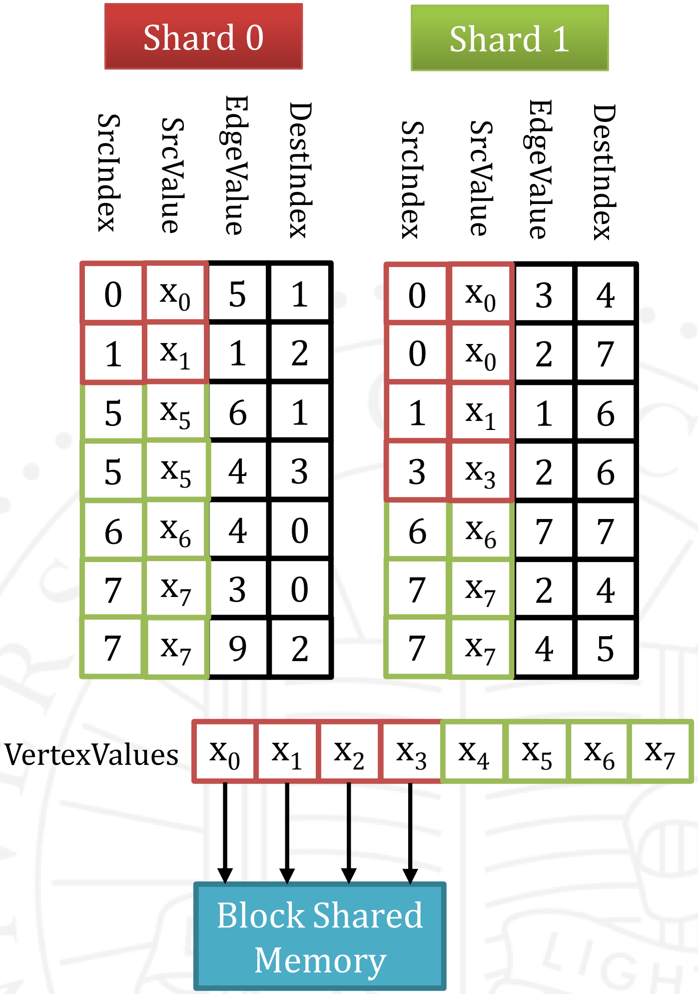
SrcIndex	SrcValue	EdgeValue	DestIndex
0	x_0	3	4
0	x_0	2	7
1	x_1	1	6
3	x_3	2	6
6	x_6	7	7
7	x_7	2	4
7	x_7	4	5

VertexValues								
	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

Block Shared Memory

G-Shards: Processing

- Iteratively process all shards
- Each shard processed by a thread block
- 4 Steps:
  - **Read**
  - Compute
  - Update
  - Write



Shard 0

SrcIndex	SrcValue	EdgeValue	DestIndex
0	x_0	5	1
1	x_1	1	2
5	x_5	6	1
5	x_5	4	3
6	x_6	4	0
7	x_7	3	0
7	x_7	9	2

Shard 1

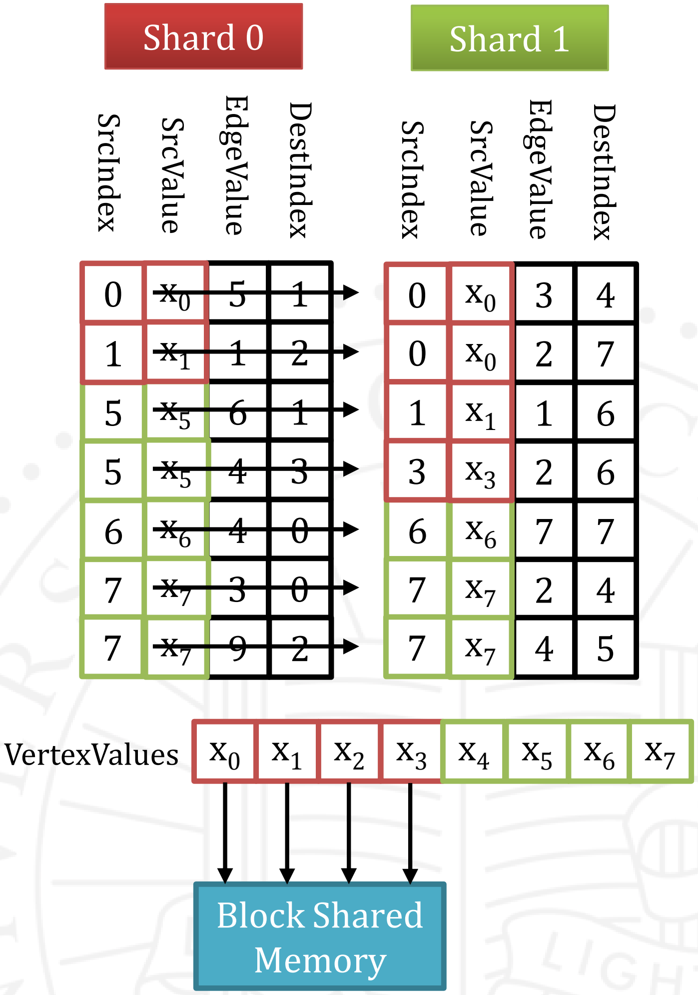
SrcIndex	SrcValue	EdgeValue	DestIndex
0	x_0	3	4
0	x_0	2	7
1	x_1	1	6
3	x_3	2	6
6	x_6	7	7
7	x_7	2	4
7	x_7	4	5

VertexValues	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

Block Shared Memory

G-Shards: Processing

- Iteratively process all shards
- Each shard processed by a thread block
- 4 Steps:
  - Read
  - **Compute**
  - Update
  - Write



Shard 0

SrcIndex	SrcValue	EdgeValue	DestIndex
0	x_0	5	1
1	x_1	1	2
5	x_5	6	1
5	x_5	4	3
6	x_6	4	0
7	x_7	3	0
7	x_7	9	2

Shard 1

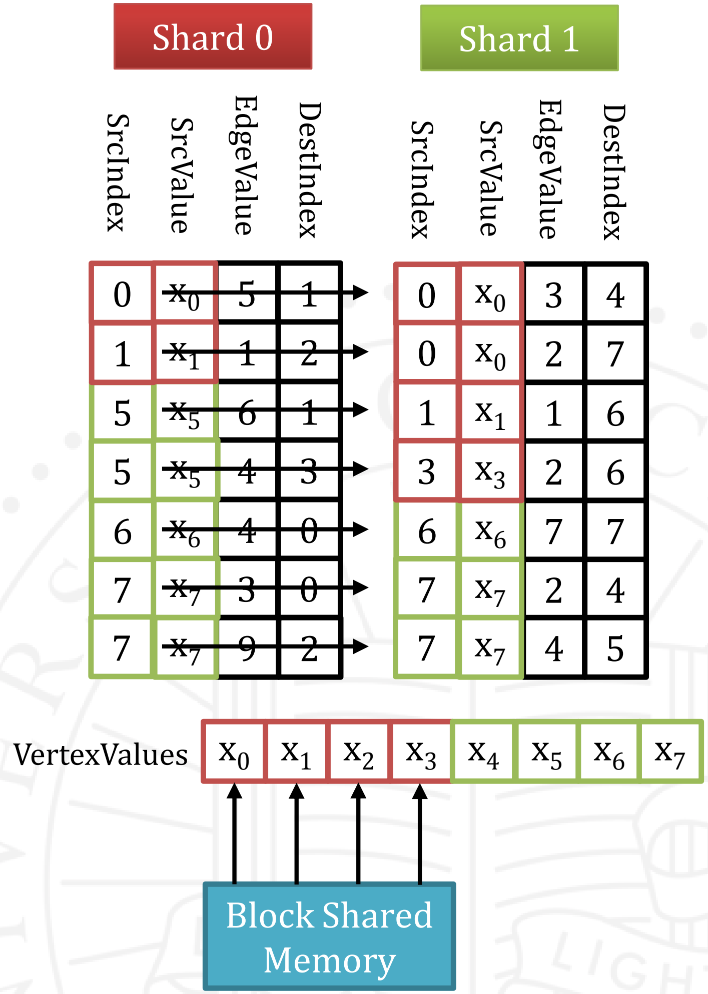
SrcIndex	SrcValue	EdgeValue	DestIndex
0	x_0	3	4
0	x_0	2	7
1	x_1	1	6
3	x_3	2	6
6	x_6	7	7
7	x_7	2	4
7	x_7	4	5

VertexValues							
x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

Block Shared Memory

G-Shards: Processing

- Iteratively process all shards
- Each shard processed by a thread block
- 4 Steps:
  - Read
  - Compute
  - **Update**
  - Write



Shard 0

SrcIndex	SrcValue	EdgeValue	DestIndex
0	x_0	5	1
1	x_1	1	2
5	x_5	6	1
5	x_5	4	3
6	x_6	4	0
7	x_7	3	0
7	x_7	9	2

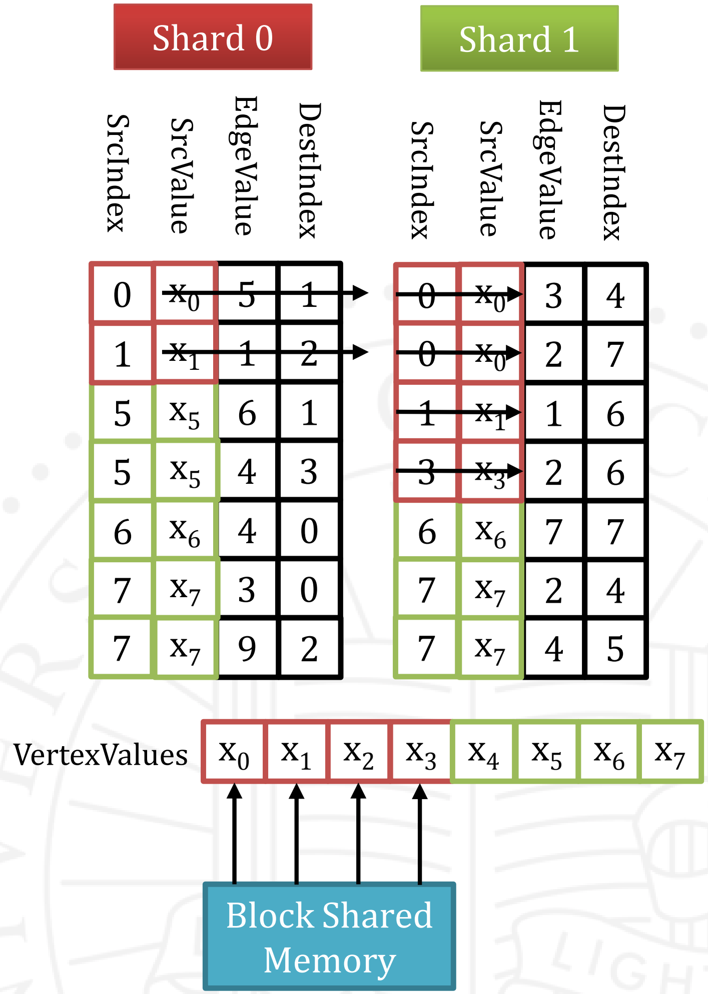
Shard 1

SrcIndex	SrcValue	EdgeValue	DestIndex
0	x_0	3	4
0	x_0	2	7
1	x_1	1	6
3	x_3	2	6
6	x_6	7	7
7	x_7	2	4
7	x_7	4	5

VertexValues	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

G-Shards: Processing

- Iteratively process all shards
- Each shard processed by a thread block
- 4 Steps:
  - Read
  - Compute
  - Update
  - **Write**



Shard 0

SrcIndex	SrcValue	EdgeValue	DestIndex
0	x_0	5	1
1	x_1	1	2
5	x_5	6	1
5	x_5	4	3
6	x_6	4	0
7	x_7	3	0
7	x_7	9	2

Shard 1

SrcIndex	SrcValue	EdgeValue	DestIndex
0	x_0	3	4
0	x_0	2	7
1	x_1	1	6
3	x_3	2	6
6	x_6	7	7
7	x_7	2	4
7	x_7	4	5

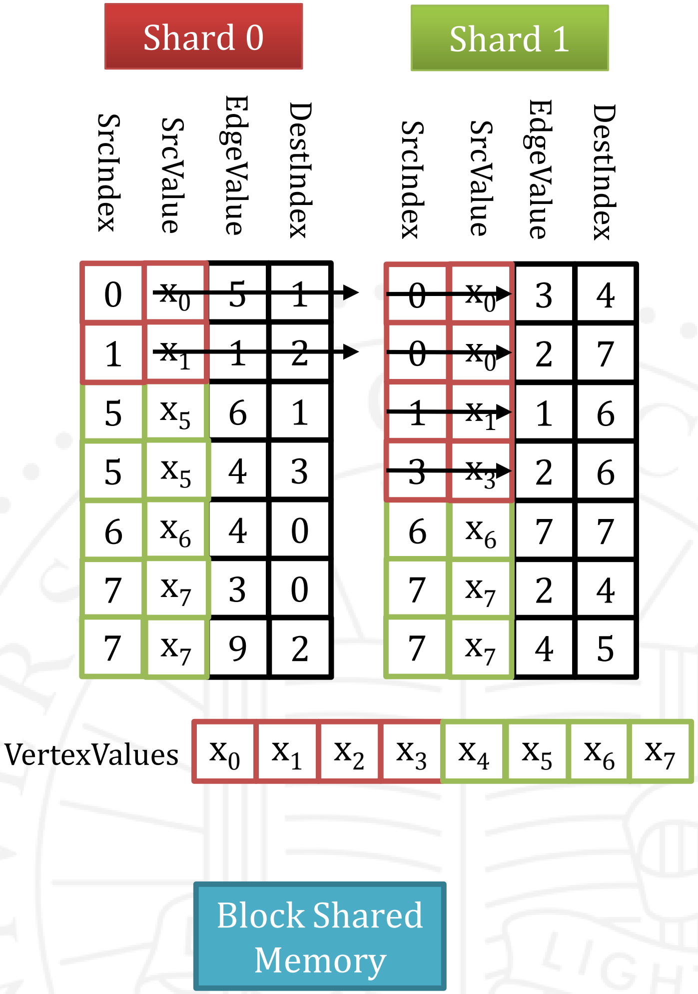
VertexValues	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

Block Shared Memory

G-Shards: Processing

- Iteratively process all shards
- Each shard processed by a thread block
- 4 Steps:
  - Read
  - Compute
  - Update
  - Write

Coalesced memory accesses in all steps



Shard 0

SrcIndex	SrcValue	EdgeValue	DestIndex
0	x_0	5	1
1	x_1	1	2
5	x_5	6	1
5	x_5	4	3
6	x_6	4	0
7	x_7	3	0
7	x_7	9	2

Shard 1

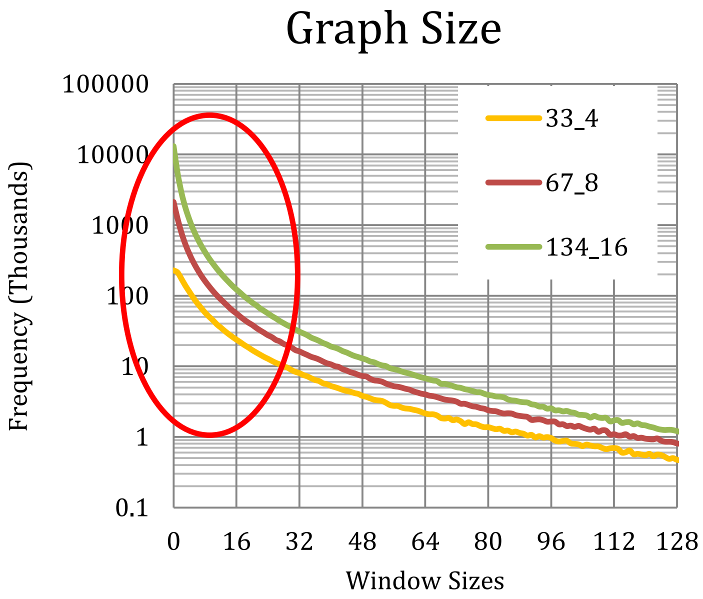
SrcIndex	SrcValue	EdgeValue	DestIndex
0	x_0	3	4
0	x_0	2	7
1	x_1	1	6
3	x_3	2	6
6	x_6	7	7
7	x_7	2	4
7	x_7	4	5

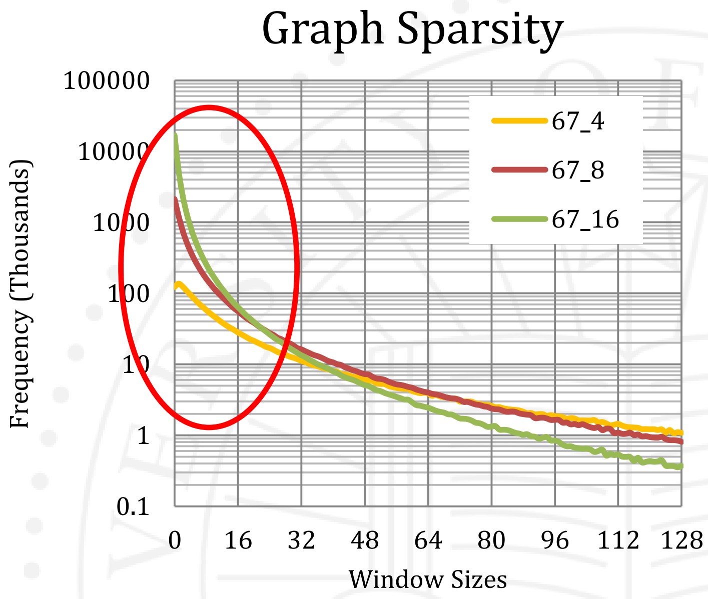
VertexValues	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

Block Shared Memory

G-Shards

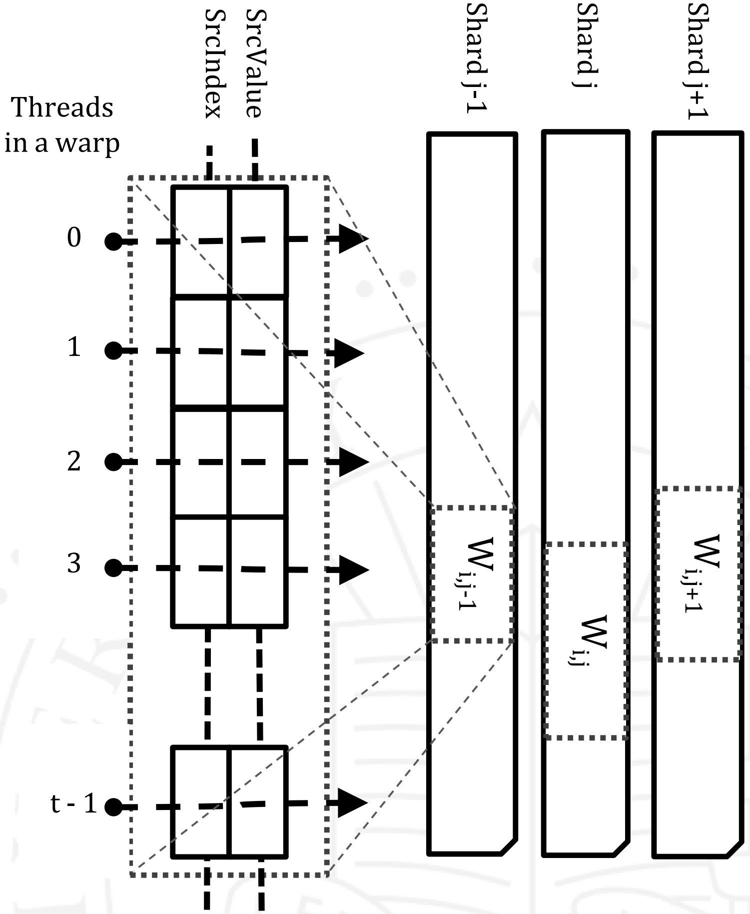
- Large and sparse graphs have small computation windows



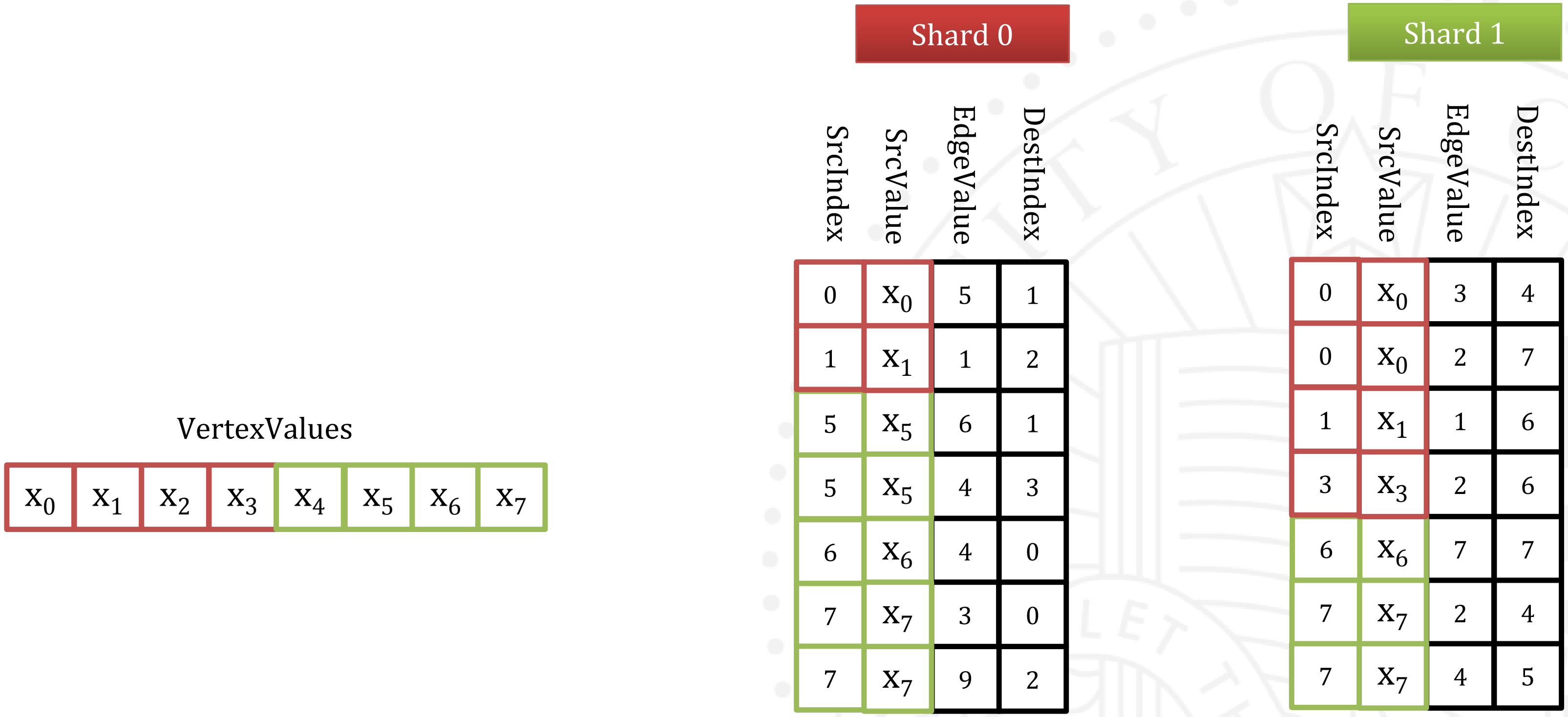


G-Shards

- Large and sparse graphs have small computation windows
- GPU underutilization
  - Many warp threads remain idle during 4th step
- Solution
  - Group windows to utilize maximum threads – Concatenated Windows



Concatenated Windows (CW)



VertexValues

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

Shard 0

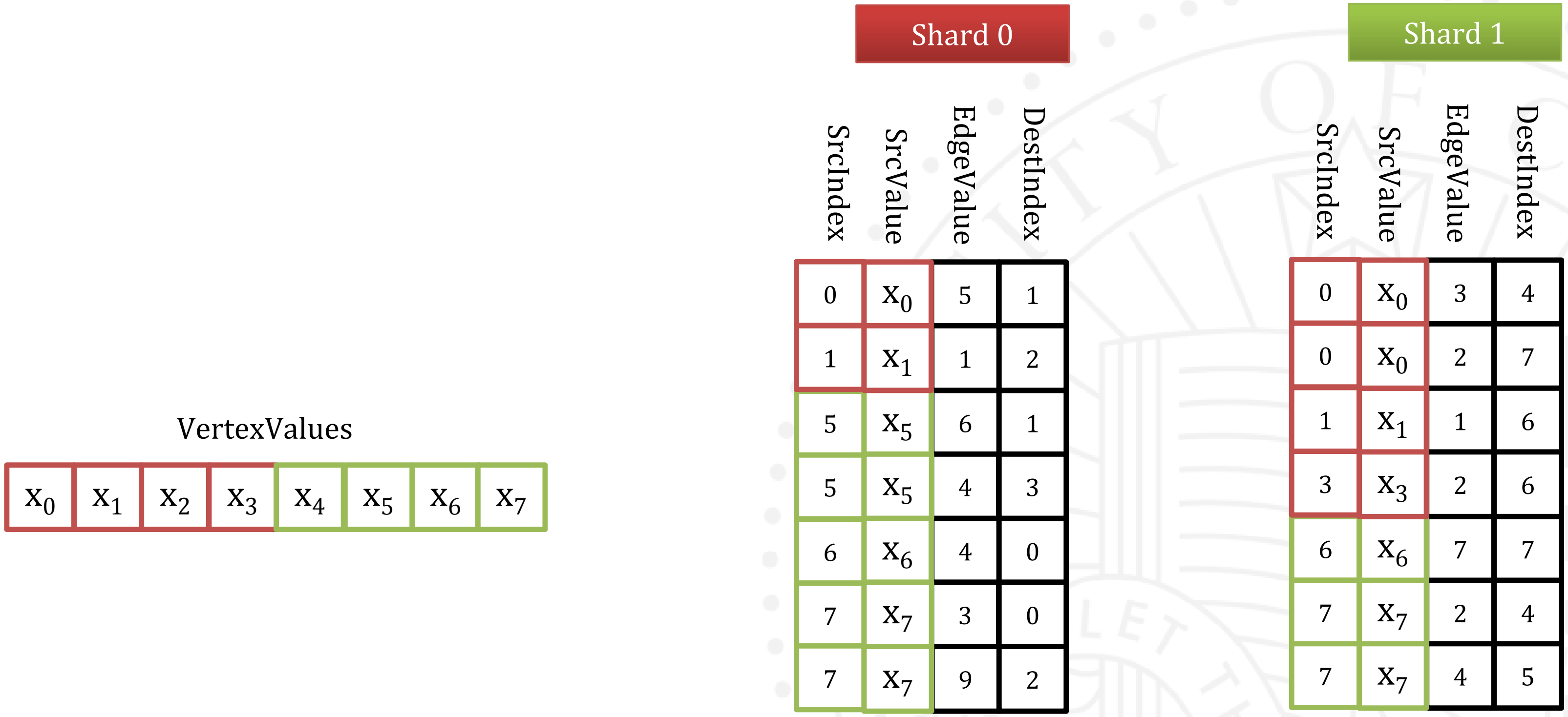
SrcIndex	SrcValue	EdgeValue	DestIndex
0	x_0	5	1
1	x_1	1	2
5	x_5	6	1
5	x_5	4	3
6	x_6	4	0
7	x_7	3	0
7	x_7	9	2

Shard 1

SrcIndex	SrcValue	EdgeValue	DestIndex
0	x_0	3	4
0	x_0	2	7
1	x_1	1	6
3	x_3	2	6
6	x_6	7	7
7	x_7	2	4
7	x_7	4	5

Concatenated Windows (CW)

- Detach SrcIndex array from shards



VertexValues

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

Shard 0

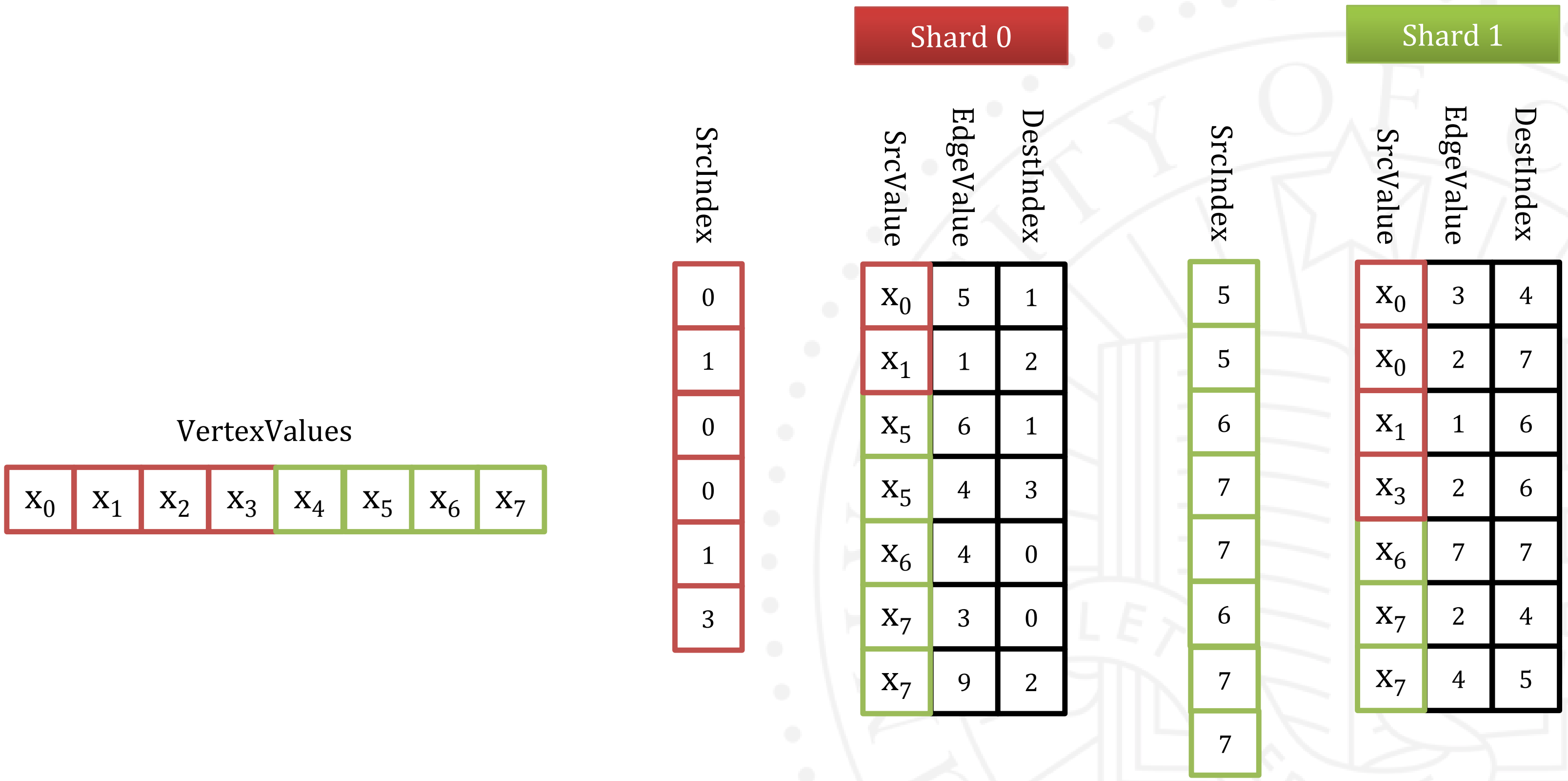
SrcIndex	SrcValue	EdgeValue	DestIndex
0	x_0	5	1
1	x_1	1	2
5	x_5	6	1
5	x_5	4	3
6	x_6	4	0
7	x_7	3	0
7	x_7	9	2

Shard 1

SrcIndex	SrcValue	EdgeValue	DestIndex
0	x_0	3	4
0	x_0	2	7
1	x_1	1	6
3	x_3	2	6
6	x_6	7	7
7	x_7	2	4
7	x_7	4	5

Concatenated Windows (CW)

- Detach SrcIndex array from shards
- Concatenate the ones belonging to windows of same shards



VertexValues

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

Shard 0

SrcIndex
0
1
0
0
1
3

SrcValue	EdgeValue	DestIndex
x_0	5	1
x_1	1	2
x_5	6	1
x_5	4	3
x_6	4	0
x_7	3	0
x_7	9	2

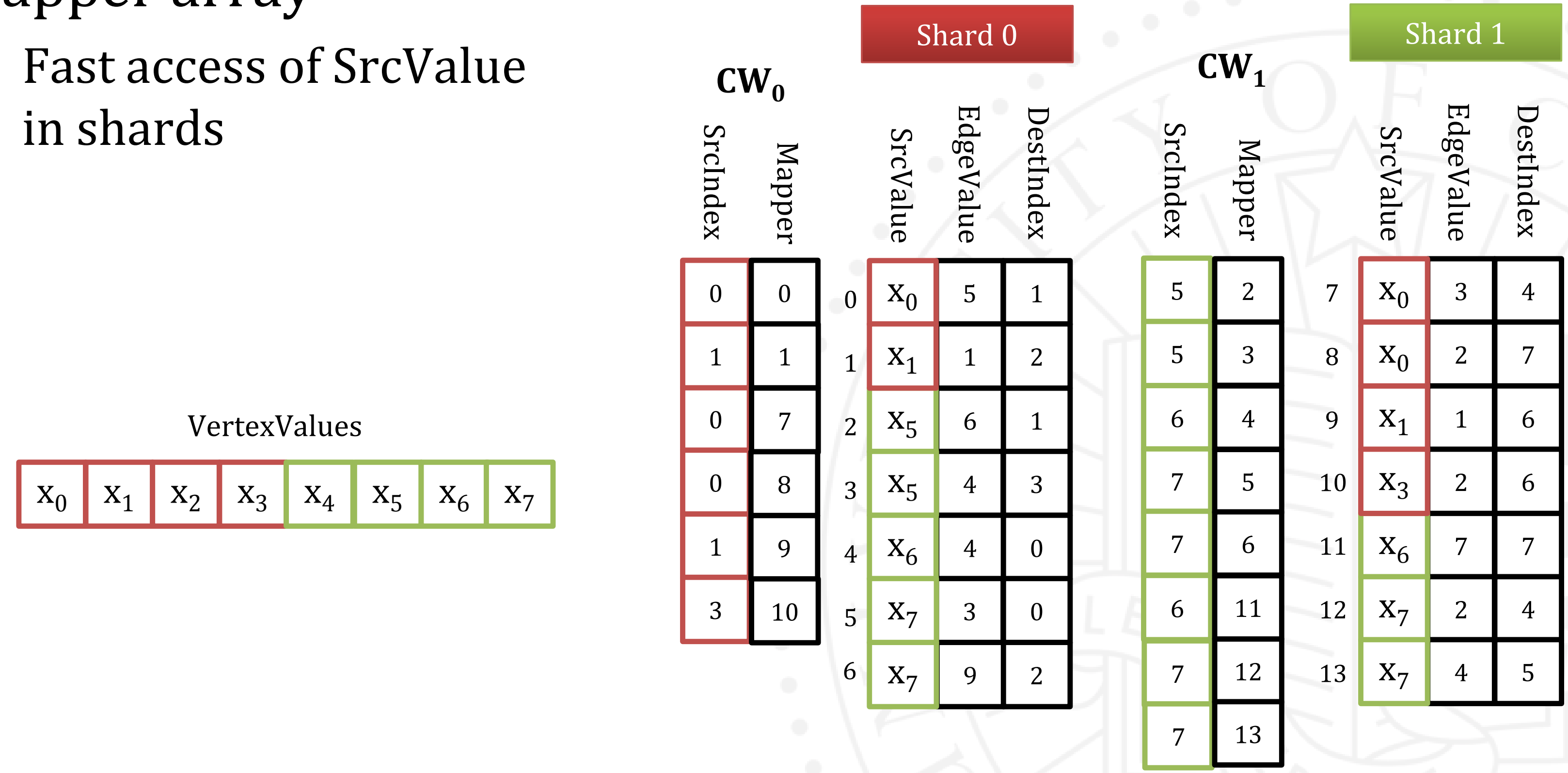
Shard 1

SrcIndex
5
5
6
7
7
6
7
7

SrcValue	EdgeValue	DestIndex
x_0	3	4
x_0	2	7
x_1	1	6
x_3	2	6
x_6	7	7
x_7	2	4
x_7	4	5

Concatenated Windows (CW)

- Detach SrcIndex array from shards
- Concatenate the ones belonging to windows of same shards
- Mapper array
  - Fast access of SrcValue in shards



VertexValues

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

CW_0

SrcIndex	Mapper
0	0
1	1
0	7
0	8
1	9
3	10

Shard 0

	SrcValue	EdgeValue	DestIndex
0	x_0	5	1
1	x_1	1	2
2	x_5	6	1
3	x_5	4	3
4	x_6	4	0
5	x_7	3	0
6	x_7	9	2

CW_1

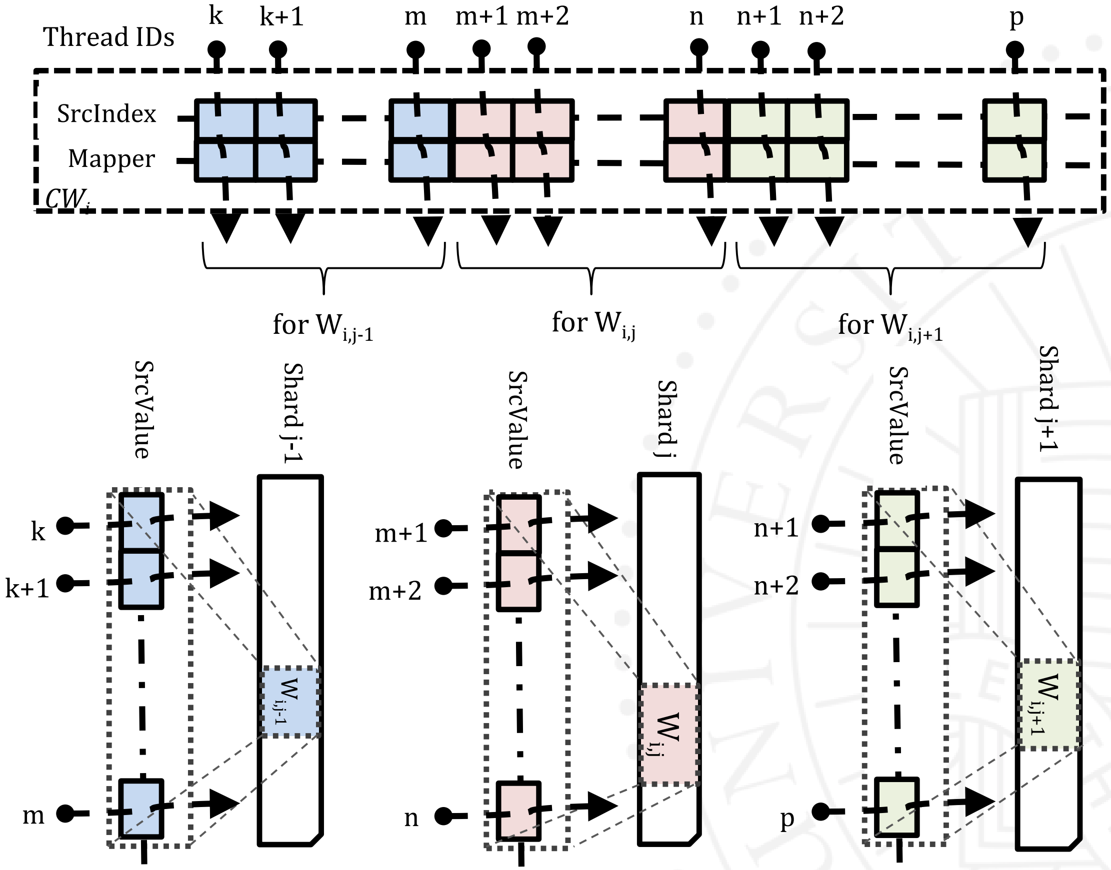
SrcIndex	Mapper
5	2
5	3
6	4
7	5
7	6
6	11
7	12
7	13

Shard 1

	SrcValue	EdgeValue	DestIndex
7	x_0	3	4
8	x_0	2	7
9	x_1	1	6
10	x_3	2	6
11	x_6	7	7
12	x_7	2	4
13	x_7	4	5

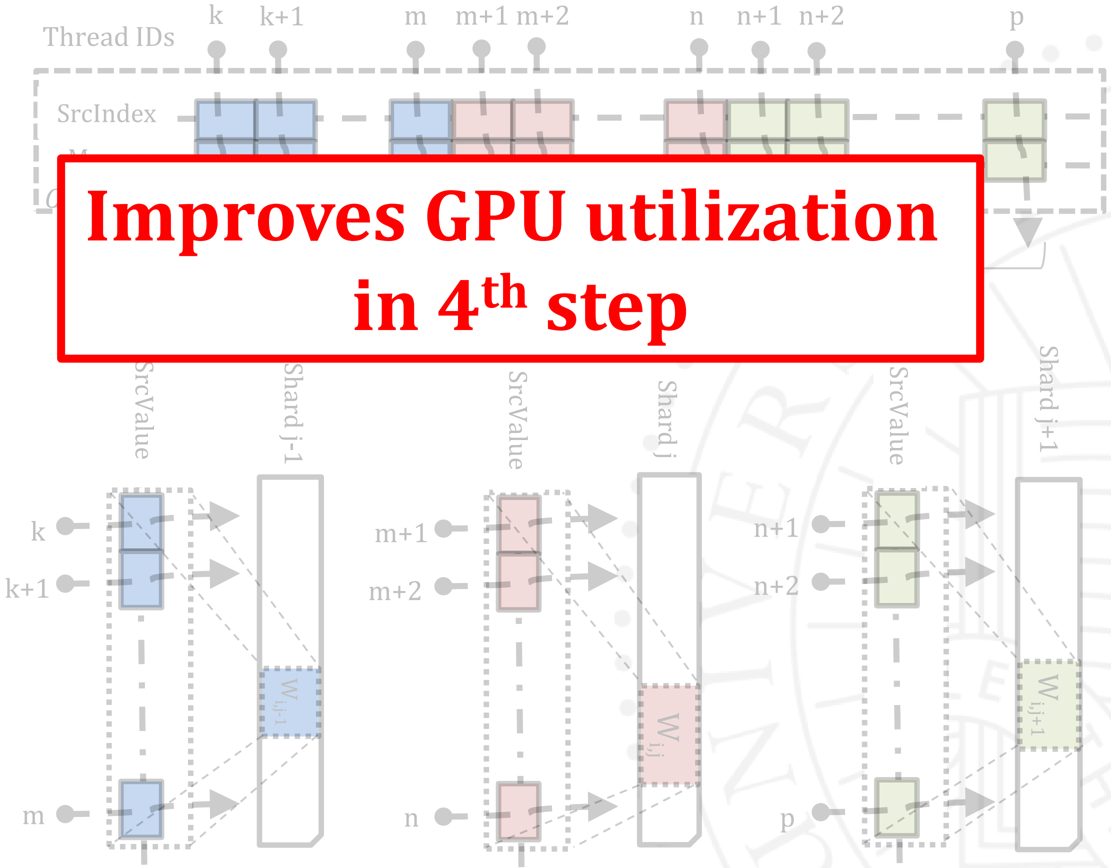
CW: Processing

- First 3 steps remain same
- Writeback is done by block threads using mapper array



CW: Processing

- First 3 steps remain same
- Writeback is done by block threads using mapper array



Improves GPU utilization in 4th step

CuSha Framework

- Uses G-Shards and CWs
- Vertex centric programming model
  - `init_compute, compute, update_condition`
  - `compute` must be atomic

```
// SSSP in CuSha
typedef struct Edge { unsigned int Weight; } Edge; // requires a variable for edge weight
typedef struct Vertex { unsigned int Dist; } Vertex; // requires a variable for distance

__device__ void init_compute( Vertex* local_V, Vertex* V ) {
    local_V->Dist = V->Dist;  // fetches distances from global memory to shared memory
}

__device__ void compute( Vertex* SrcV, StaticVertex* SrcV_static, Edge* E, Vertex* local_V ) {
    if (SrcV->Dist != INF)  // minimum distance of a vertex from source node is calculated
        atomicMin ( &(local_V->Dist), SrcV->Dist + E->Weight );
}

__device__ bool update_condition( Vertex* local_V, Vertex* V ) {
    return ( local_V->Dist < V->Dist );  // return true if newly calculated value is smaller
}
```

Experimental Setup

- GeForce GTX780:12 SMs, 3 GB GDDR5 RAM
- Intel Core i7-3930K Sandy bridge 12 cores (HT enabled) 3.2 GHz, DDR3 RAM, PCI-e 3.0 16x
- CUDA 5.5 on Ubuntu 12.04

- 8 Benchmarks
  - BFS, SSSP, PR, CC, SSWP, NN, HS, CS
- Input graphs from the SNAP[1] dataset collection

[1] SNAP: http://snap.stanford.edu

Speedup over Multi-Threaded CPU

BM	G-Shards	CW
BFS	2.41x-10.41x	2.61x-11.38x
SSSP	2.61x-12.34x	2.99x-14.27x
PR	5.34x-24.45x	6.46x-28.98x
CC	1.66x-7.46x	1.72x-7.74x
SSWP	2.59x-11.74x	3.03x-13.85x
NN	1.82x-19.17x	1.97x-19.59x
HS	1.74x-7.07x	1.80x-7.30x
CS	2.39x-11.06x	2.49x-11.55x

Average speedups

- G-Shards: 2.57x-12.96x
- CW: 2.88x-14.33x

Inputs	G-Shards	CW
LiveJournal	4.10x-26.63x	4.74x-29.25x
Pokec	3.26x-15.19x	3.2x-14.89x
HiggsTwitter	1.23x-5.30x	1.23x-5.34x
RoadNetCA	1.95x-9.79x	2.95x-14.29x
WebGoogle	1.95x-9.79x	2.95x-14.29x
Amazon0312	1.65x-6.27x	1.88x-7.20x

Speedup over VWC-CSR

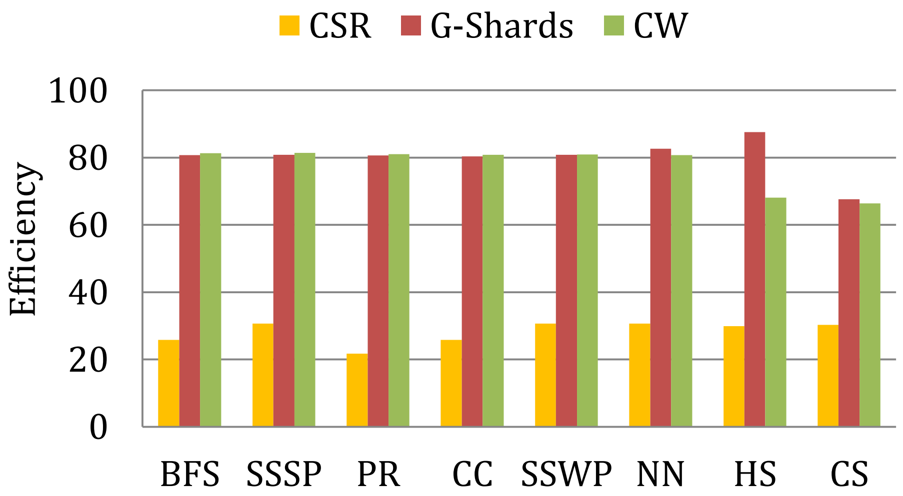
BM	G-Shards	CW
BFS	1.94x-4.96x	2.09x-6.12x
SSSP	1.91x-4.59x	2.16x-5.96x
PR	2.66x-5.88x	3.08x-7.21x
CC	1.28x-3.32x	1.36x-4.34x
SSWP	1.90x-4.11x	2.19x-5.46x
NN	1.42x-3.07x	1.51x-3.47x
HS	1.42x-3.01x	1.45x-3.02x
CS	1.23x-3.50x	1.27x-3.58x

Average speedups

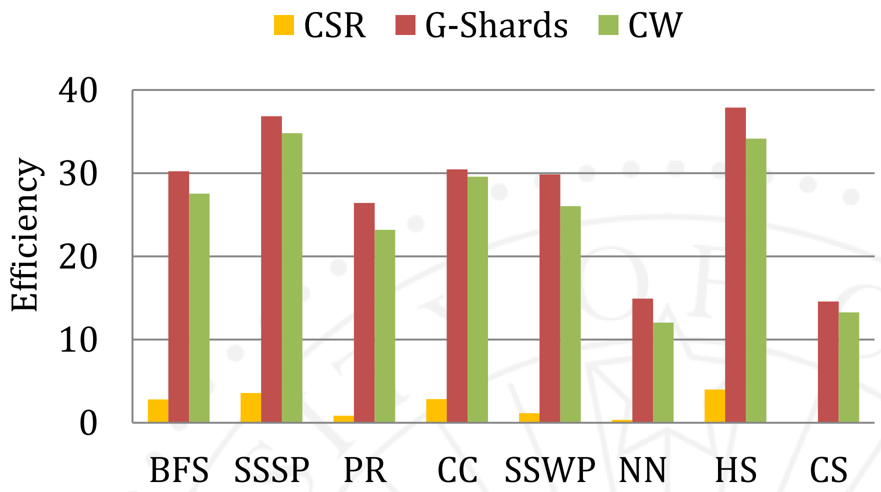
- G-Shards: 1.72x-4.05x
- CW: 1.89x-4.89x

Inputs	G-Shards	CW
LiveJournal	1.66x-2.36x	1.92x-2.72x
Pokec	2.40x-3.63x	2.34x-3.58x
HiggsTwitter	1.14x-3.59x	1.14x-3.61x
RoadNetCA	1.34x-8.64x	1.92x-12.99x
WebGoogle	2.41x-3.71x	2.45x-3.74x
Amazon0312	1.37x-2.40x	1.57x-2.73x

Global Memory Efficiency



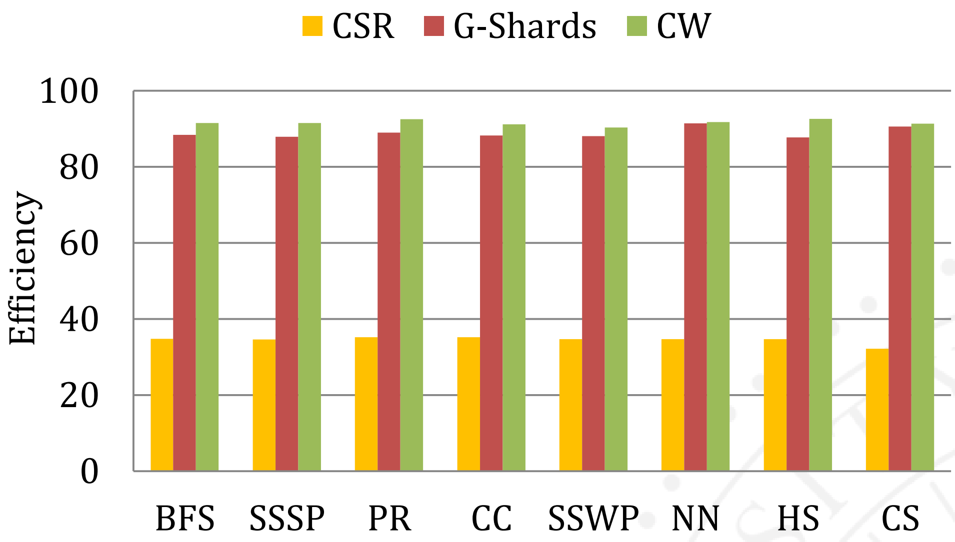
Avg Memory Load Efficiency



Avg Memory Store Efficiency

Average	CSR	G-Shards	CW
Avg global memory load efficiency	28.18%	80.15%	77.59%
Avg global memory store efficiency	1.93%	27.64%	25.06%

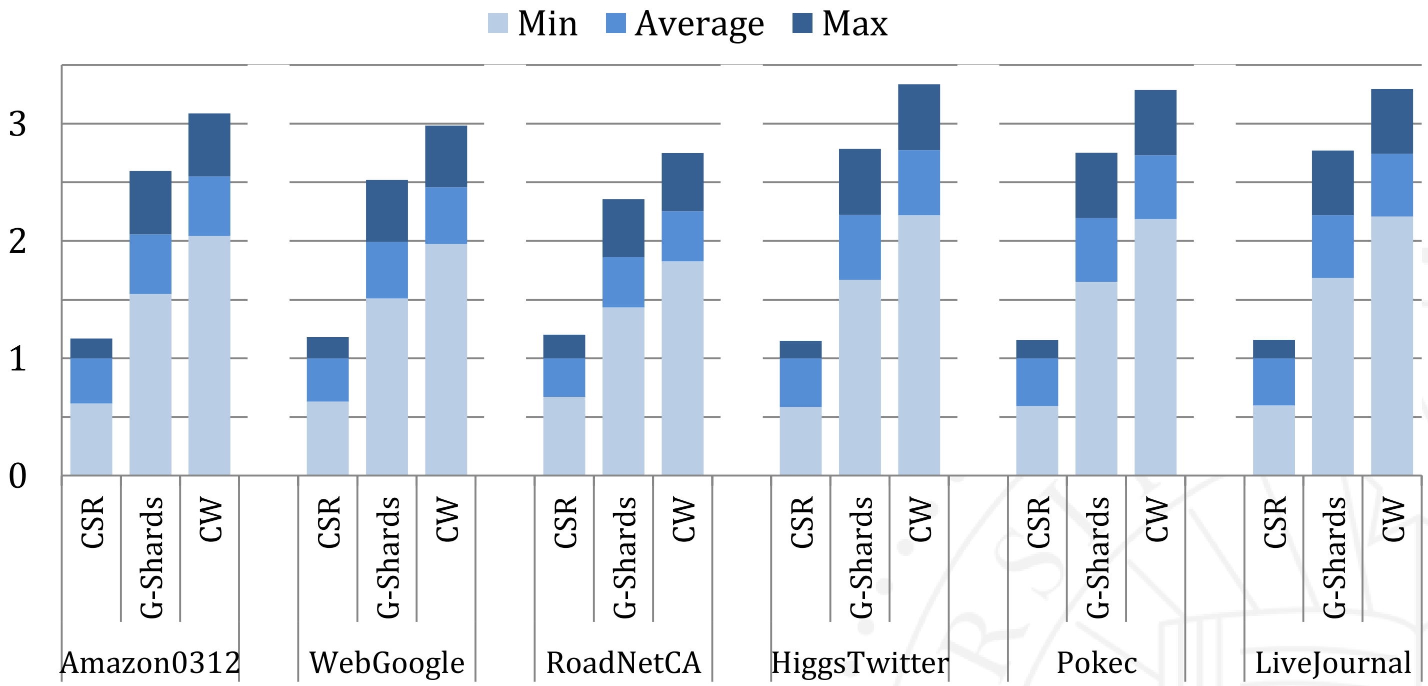
Warp Execution Efficiency



Avg Warp Execution Efficiency

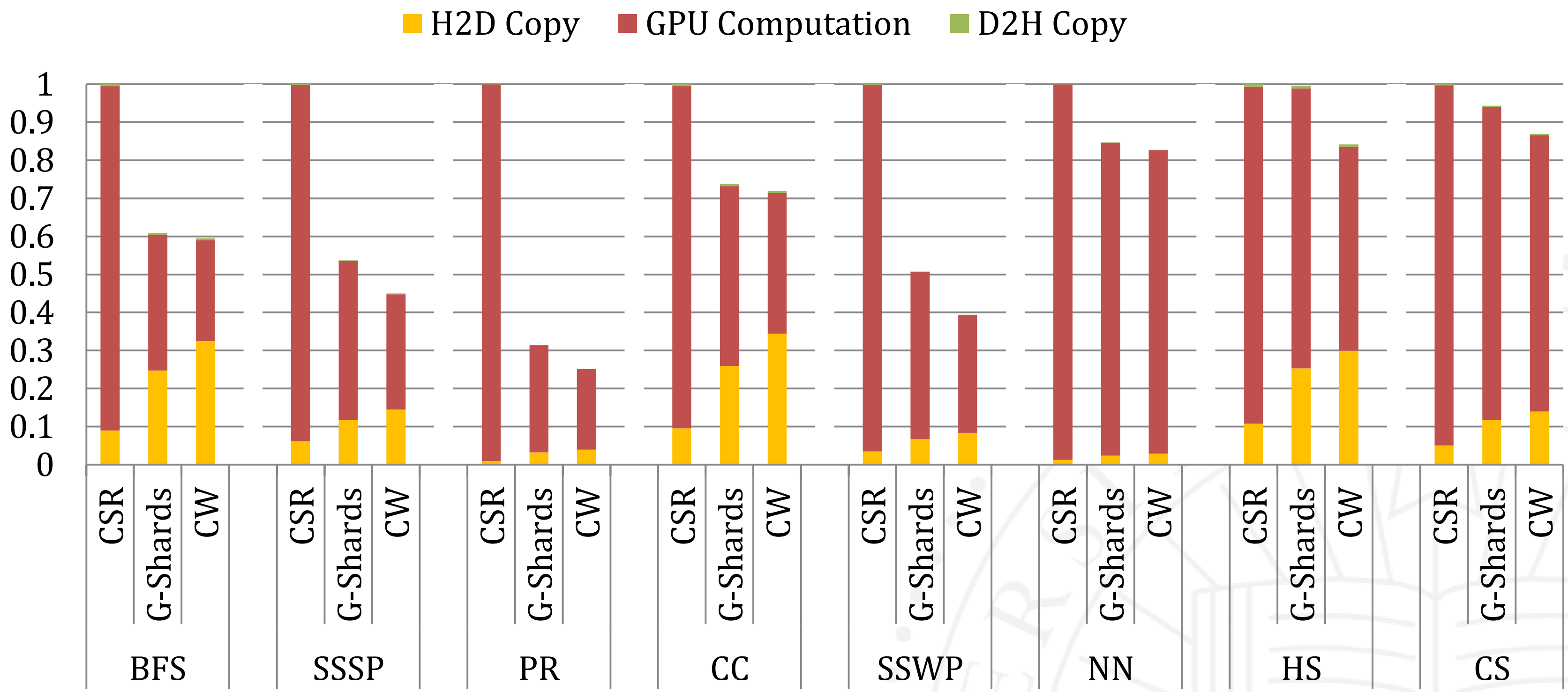
Average	CSR	G-Shards	CW
Avg warp execution efficiency	34.48%	88.90%	91.57%

Space Overhead



- G-Shards over CSR: `(|E|-|V|)xsizeOf(index)` + `|E|xsizeOf(Vertex)` bytes
- CW over G-Shards: `|E|xsizeOf(index)` bytes

Execution Time Breakdown



- Space overheads directly impact H2D copy times
- G-Shards and CW are computation friendly

Other Experiments

- Traversed Edges Per Seconds (TEPS)
  - G-Shards and CW up to 5 times better than CSR
- Sensitivity study using synthetic benchmarks
  - G-Shards is sensitive to:
    - Graph Size
    - Graph Density
    - Number of vertices assigned to shards
  - CW less affected by these parameters

Conclusion

- G-Shards
  - Shard based representation mapped for GPUs
  - Fully coalesced memory accesses
- Concatenated Windows
  - Improves GPU utilization
- CuSha Framework
  - Vertex centric programming model
  - 1.71x-1.91x speedup over best VWC-CSR

Thanks

- CuSha on GitHub
  - `http://farkhor.github.io/CuSha`
- GRASP
  - `http://grasp.cs.ucr.edu`