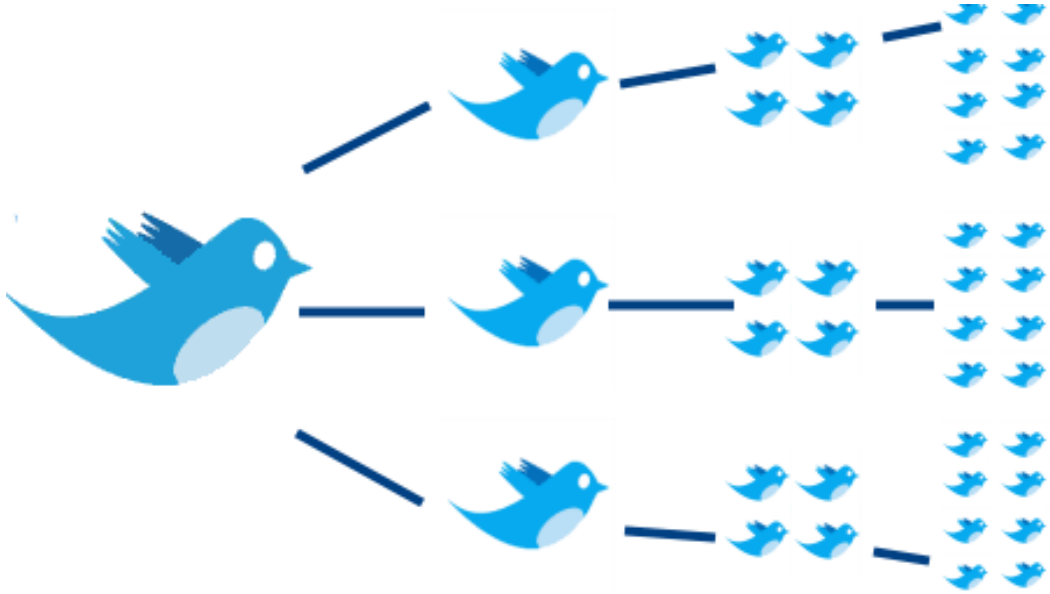
Accelerating Influence Spread Estimation on Social Networks in the Continuous-time Domain

Koushik Pal

Zissis Poulos

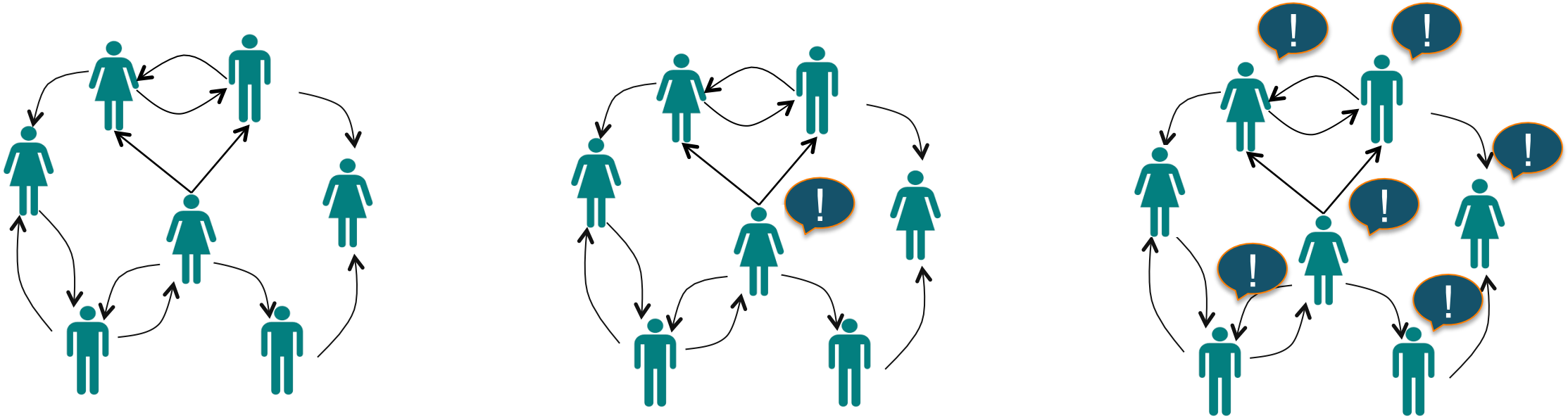
Viral Marketing



- Online social networks enable large scale word-of-mouth marketing

Word-of-Mouth Marketing Strategy

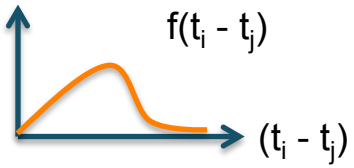
Identify influencers → Convince them to adopt the idea/ product → These customers endorse the product among their friends



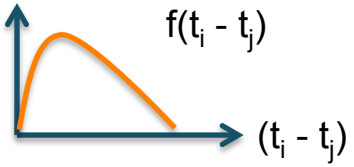
- **The Big Question:** which individuals should we target initially, such that the expected number of follow-ups is maximized?

Influence Maximization in the Continuous-time Domain

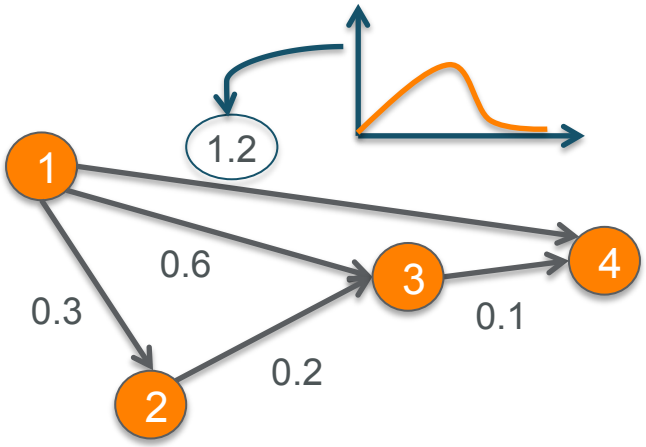
- **Model: Continuous-time Independent Cascade** [Nan Du et al, NIPS 2013]
- **Infection: a node adopts the opinion/product**
- **Pairwise conditional density between nodes**
  $f_{ji}(t_j|t_i) = f_{ji}(t_i - t_j)$ over time – "time it takes for node i to infect node j"



Eddie to Jane



Jane to Mike

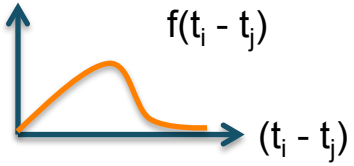


Sampling is required to generate weights

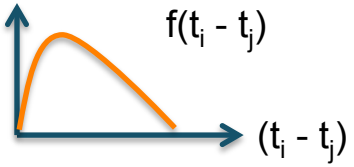
Slide Credit: Nan Du et al, NIPS 2013

Influence Maximization in the Continuous-time Domain

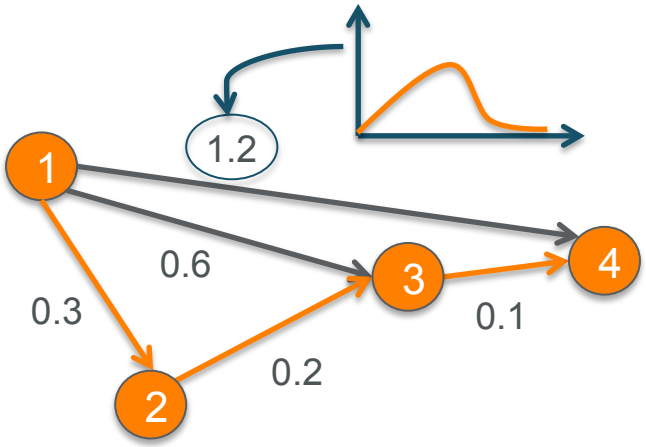
- **Model: Continuous-time Independent Cascade** [Nan Du et al, NIPS 2013]
- **Infection: a node adopts the opinion/product**
- **Pairwise conditional density between nodes**
  $f_{ji}(t_j|t_i) = f_{ji}(t_i - t_j)$ over time – "time it takes for node i to infect node j"



Eddie to Jane



Jane to Mike



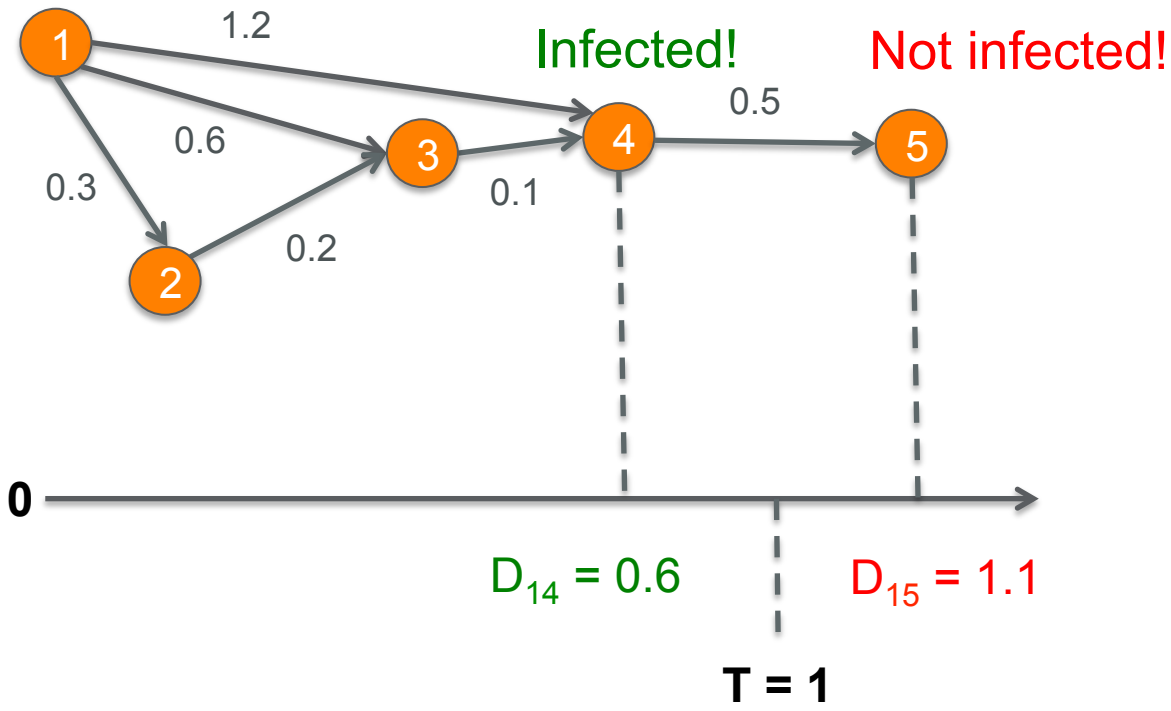
$D_{14} = 0.6$

Shortest Path Property

For a given sample, node 1 infects node 4 after time D_{14} = length of shortest path between nodes 1 and 4

Influence Maximization in the Continuous-time Domain

- In reality a campaign has a strict deadline T
- Role of T in spread



Infected nodes may change per sample

- Expected spread of node (set of nodes) = expected # nodes it infects

Problem Statement

- **"Find set S of k nodes that maximizes expected spread σ(S)"**
- **NP-hard...but there exists a greedy 63%-approximation algorithm (Kempe et al, 2003)**

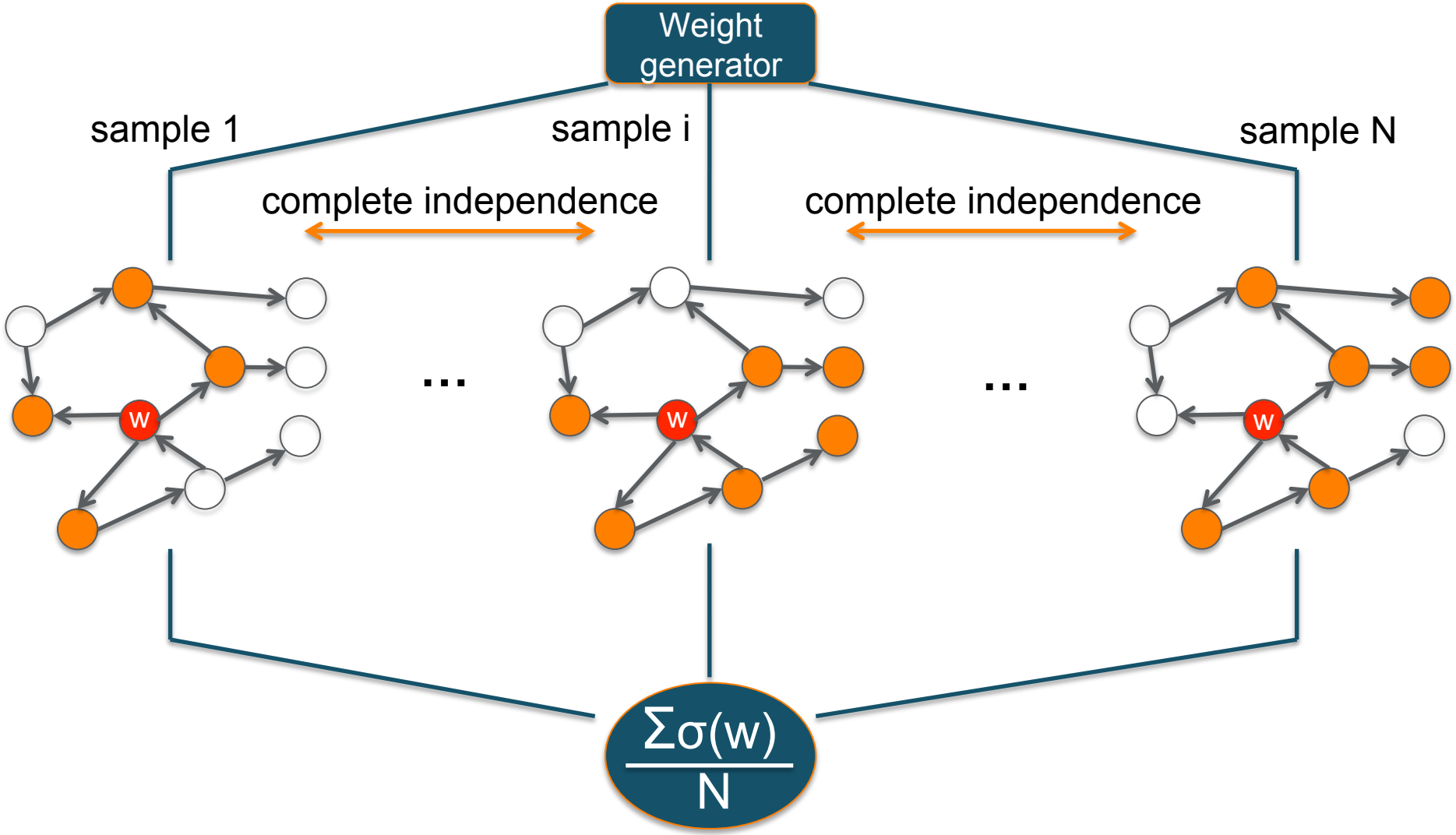
```
initialize S = Ø
for i = 1 to k do
        select u ← argmax_{w∈V\S} [σ(S U {w}) - σ(S)]
S ← S U {u}
end for
return S
```

```
for j = 1 to N do          // N samples, ≈ 100,000
        for all nodes not in S do     // # nodes, |V|
                enumerate shortest paths ≤ T
return u ← node with max # of such paths on avg
```

#P-complete

Solution 1: Naïve Sampling

- **Follows exactly the previous pseudo-code**



Solution 2: Cohen's Estimator

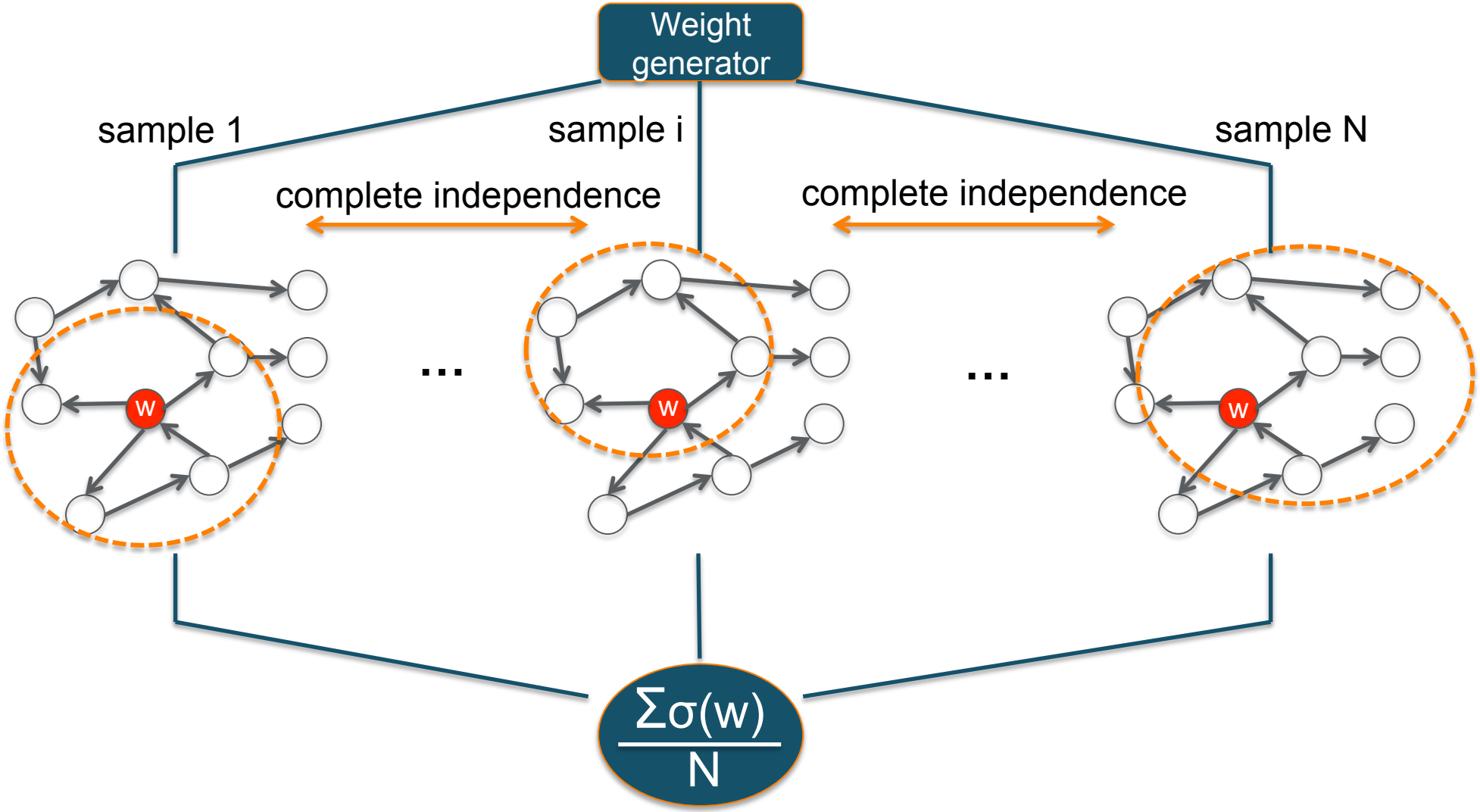
- **Proposed by Nan Du et al, NIPS 2013 (ConTinEst framework)**
- **Replace all-pairs shortest paths with Cohen's randomized algorithm**
- **Estimates neighborhood size (spread) per node, per sample**
- **Faster by a O(|V|/log|V|) factor – fewer samples – speed vs. accuracy trade-off**

```
for j = 1 to N do          // N samples, ≈ 100,000
    for all nodes not in S do     // # nodes, |V|
        enumerate shortest paths ≤ T
return u ← node with max # of such paths
```

```
for j = 1 to N do          // N samples, ≈ 10,000
    for all nodes not in S do     // # nodes, |V|
        estimate d-neighborhood with d ≤ T
return u ← node with largest neighborhood on avg
```

Solution 2: Cohen's Estimator

- **Proposed by Nan Du et al, 2013 (ConTinEst framework)**

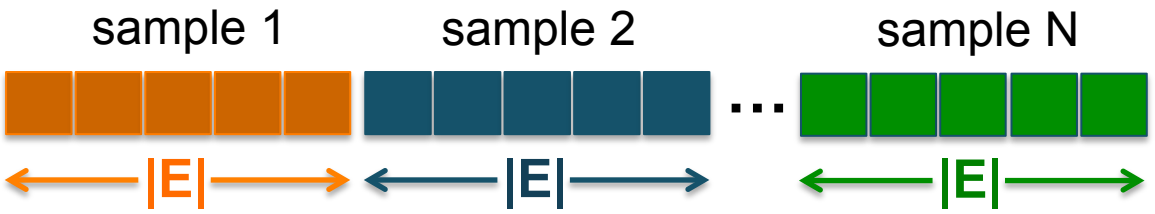


Parallelization

- **Naïve Sampling:**
  - embarrassingly parallel
  - complete independence across samples
  - [100,000 … 1,000,000] samples for convergence, motivates acceleration
- **Cohen's Estimator:**
  - fewer samples [10,000 … 50,000]
  - core randomized algorithm exhibits heavy sequential dependence
- **Concerns: space vs speed trade-offs**
  - need to pre-generate weights (on host vs on device)
  - balance data loads/unloads between host and device
  - batch sampling?

Data Allocation – Host Side

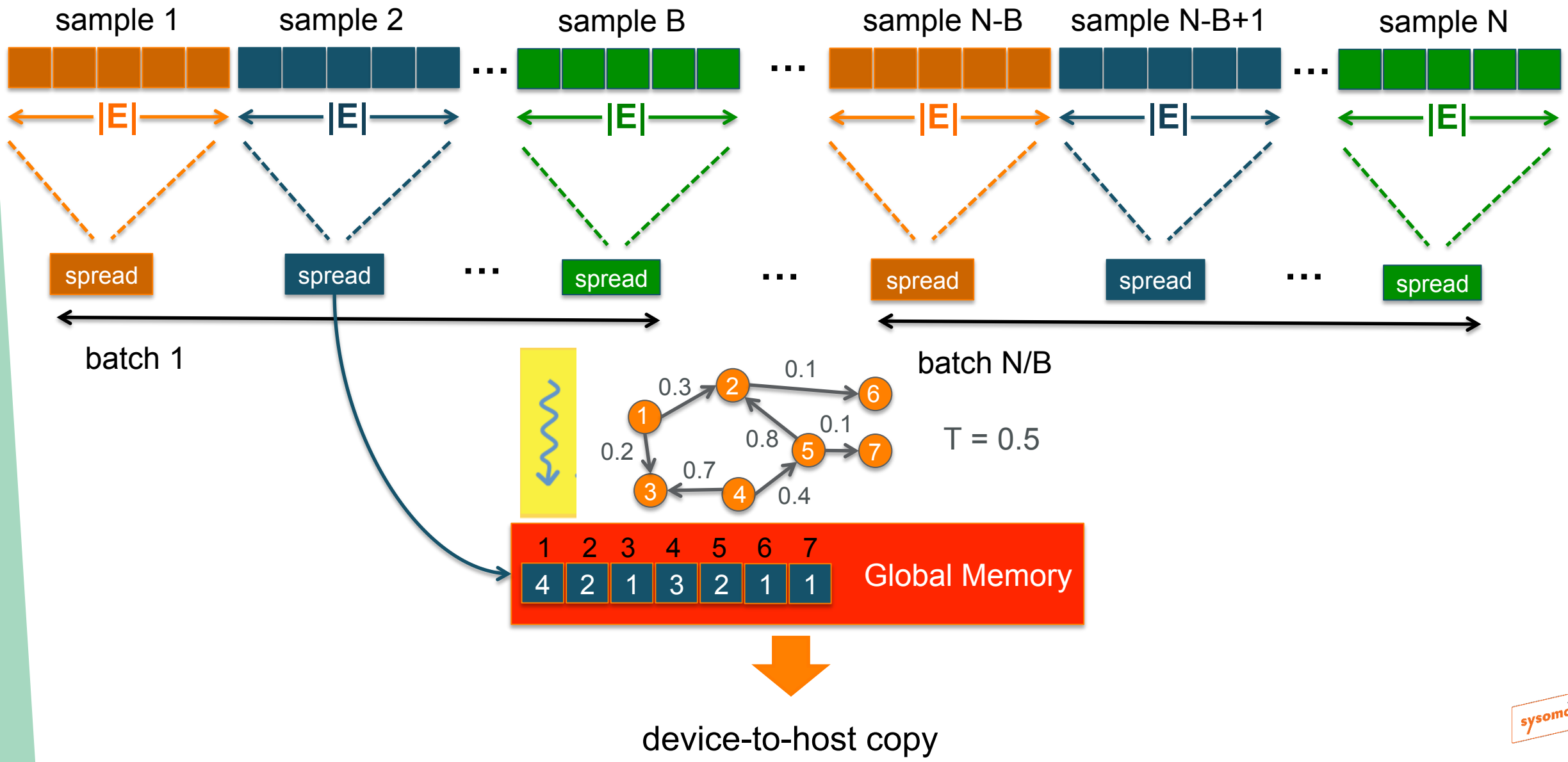
- G(V,E): adjacency list representation O(|V|+|E|)
- Edge weights: pre-generated and stored for all samples O(N*|E|)



- Memory intensive (2GB for small 200-node network, 1M samples)

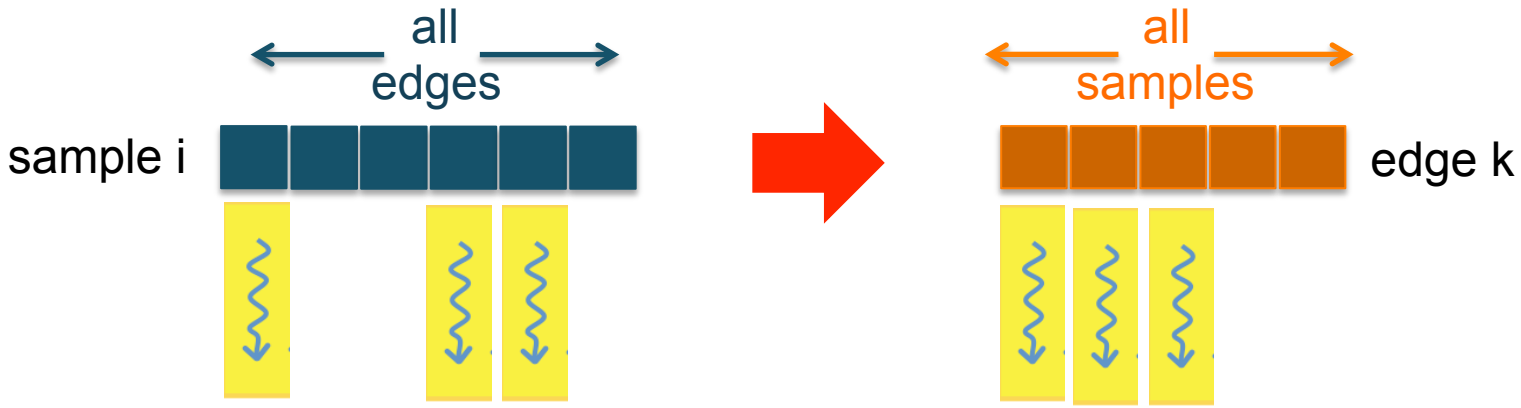
- Implement batch sampling/allocation
  - fix batch size to constant B such that N/B batches are passed to device

Batch Sampling with Batch Size B



Latency Improvements for GPU

- **Inherent semi-randomness causes poor memory coalescence**
- **Adjacent threads may need to access edge weights far apart in memory**
- Improvement #1: Rearrange edge weight order on device memory

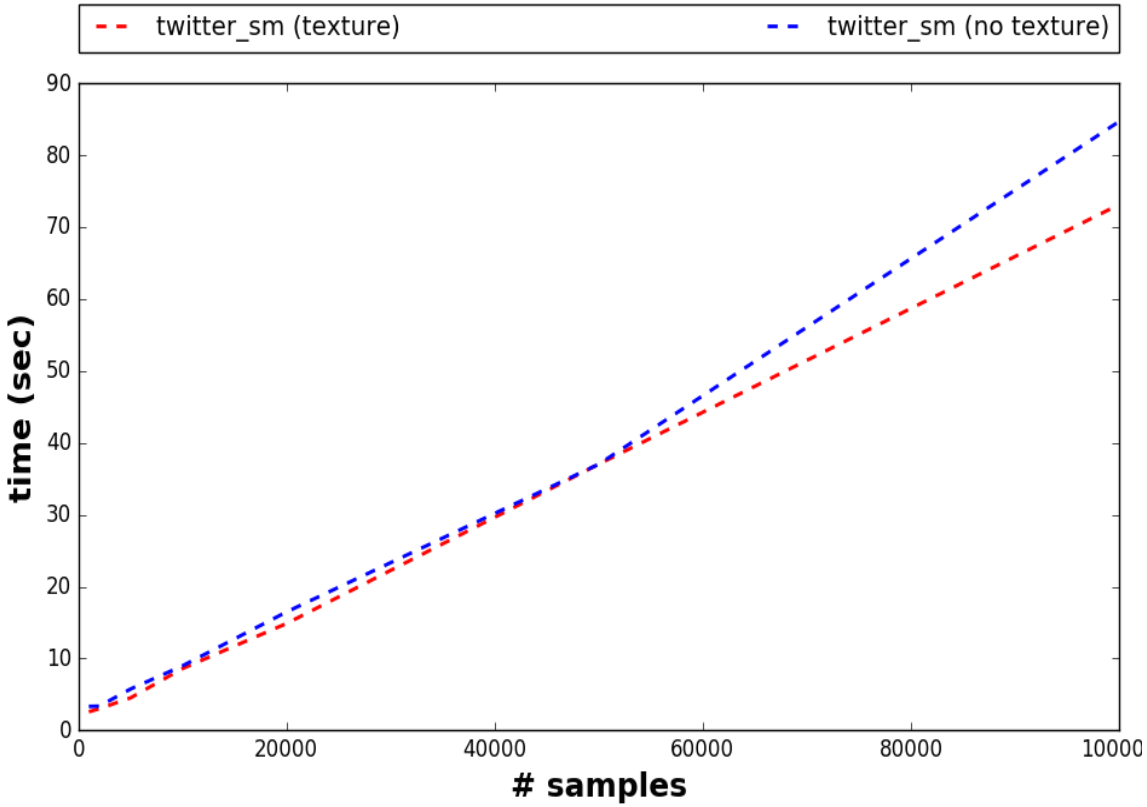


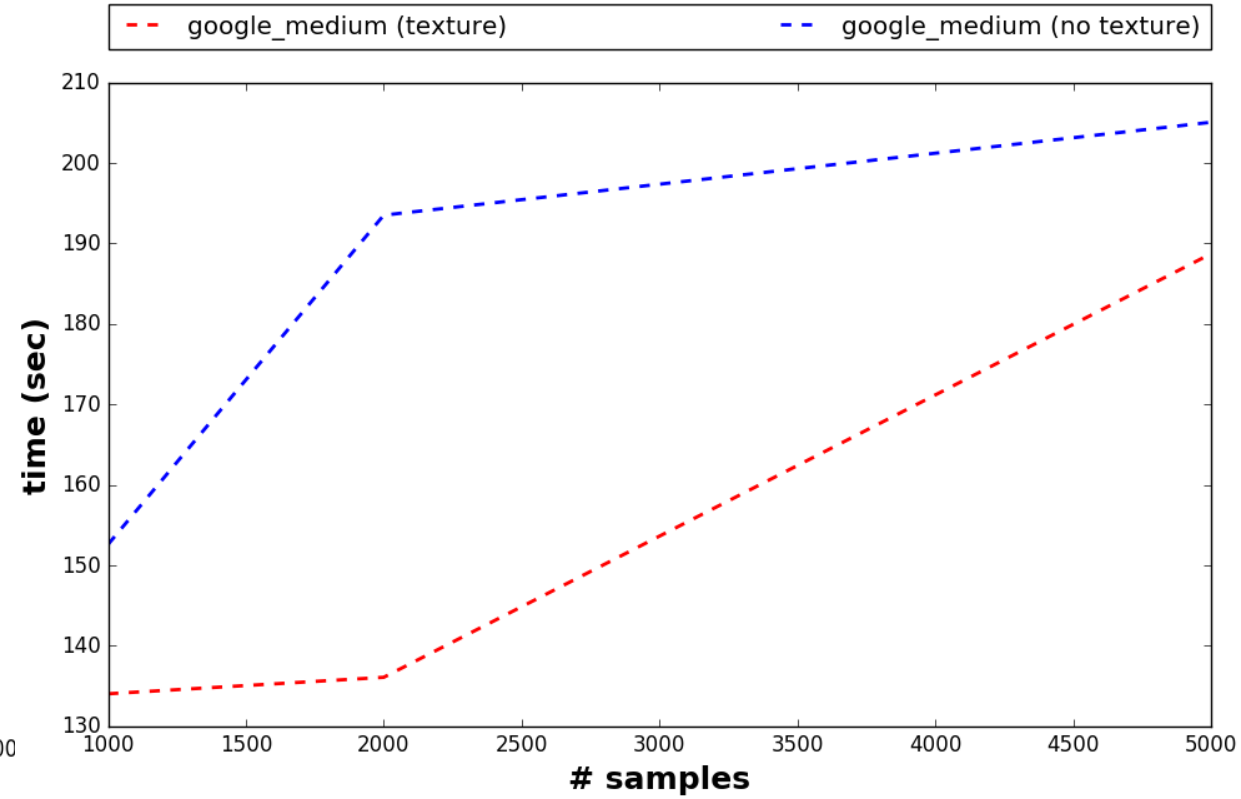
- Improvement #2: Use 1D texture memory for read-only data (weights, topology etc)
- Improvement #3: Disable L1 cache (fewer wasteful fetches)

Experimental Setup

- System:
  - AWS GRID K520
  - 3074 CUDA Cores
  - 8GB DDR5
  - Compute Capability 3.0
  - CPU: Intel Xeon E5 (Sandy Bridge)
- Social Graphs:
  - Twitter_small | 236 nodes| 2479 edges
  - Google_medium | 638 nodes | 16043 edges
  - Twitter_big | 1049 nodes | 54555 edges
- Sampling range: 100 – 100,000 samples

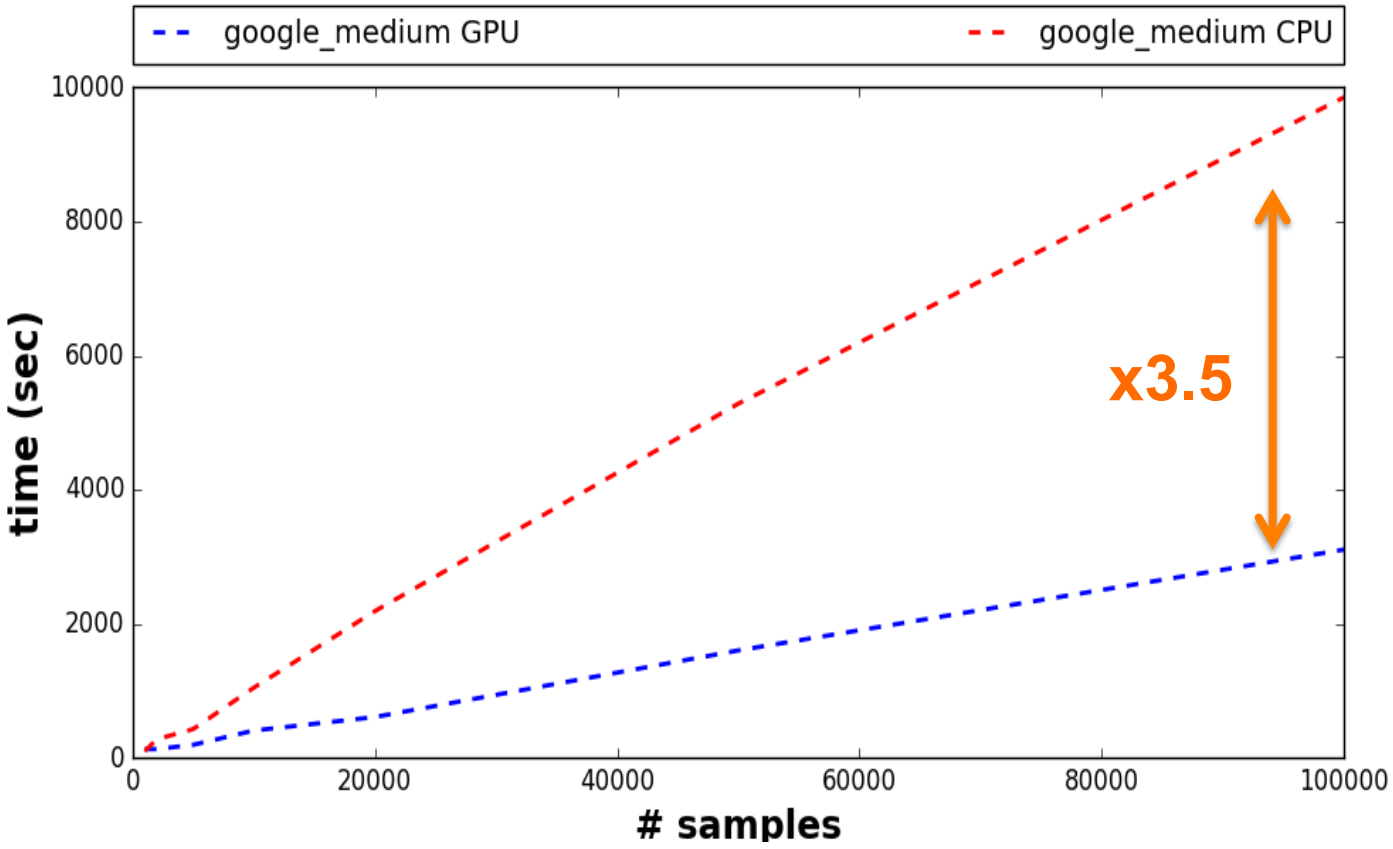
Results – Naïve Sampling

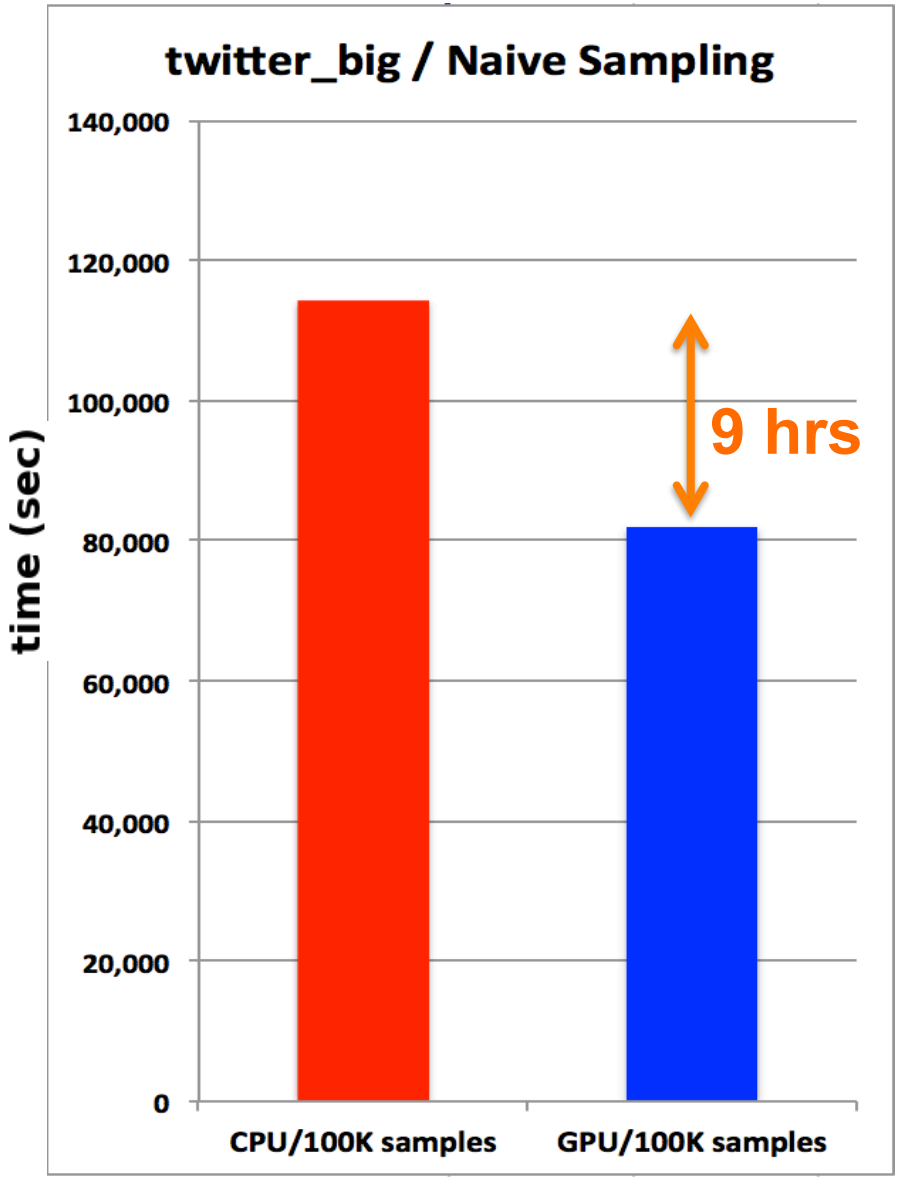




Performance gains when using texture pipeline / read-only data cache for read-only data

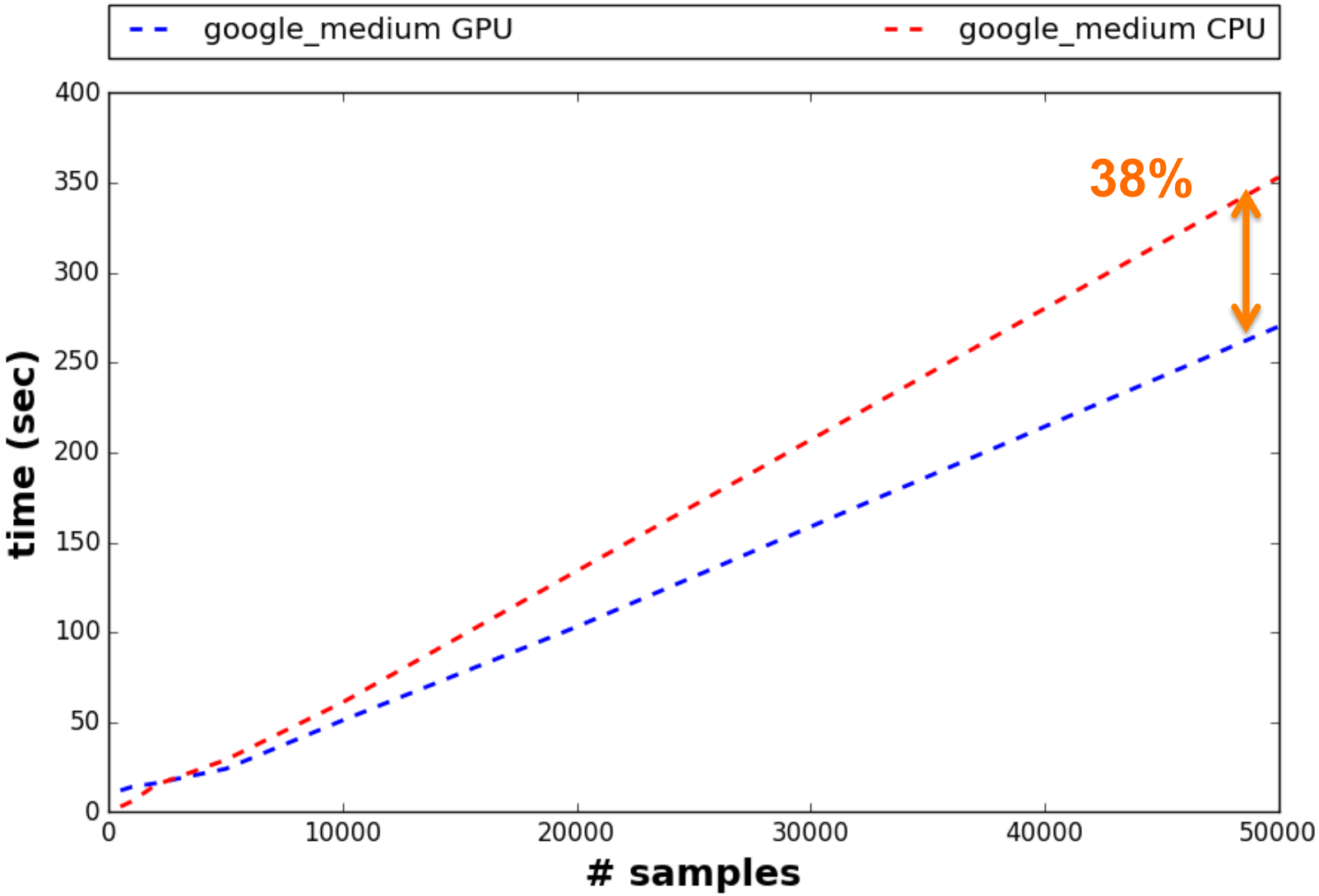
Results – Naïve Sampling





GPU vs. CPU for Naïve Sampling

Results – Cohen's Estimator



- Smaller gains
- Space complexity bottleneck

Thank You!