Accelerating Graph Sampling for Graph Machine Learning using GPUs

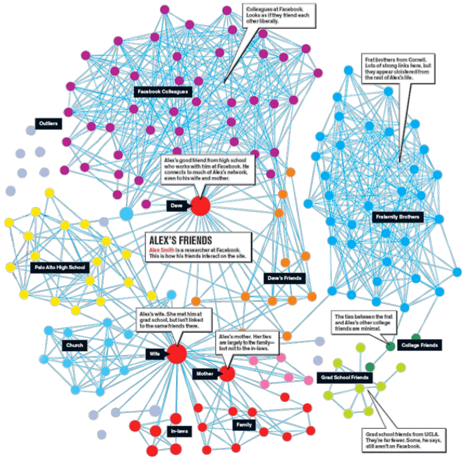
Abhinav Jangda
University of Massachusetts Amherst

Sandeep Polisetty
University of Massachusetts Amherst

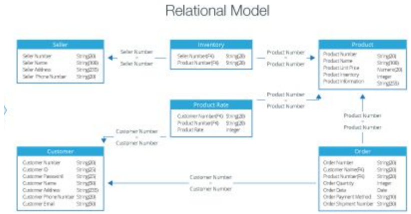
Arjun Guha
Northeastern University

Marco Serafini
University of Massachusetts Amherst

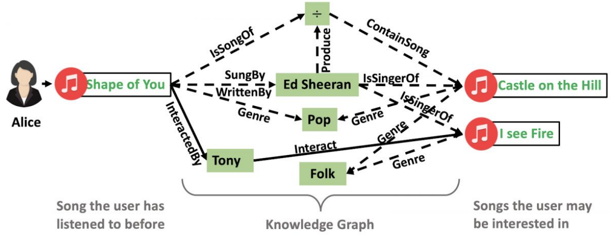
Graph Data is Everywhere



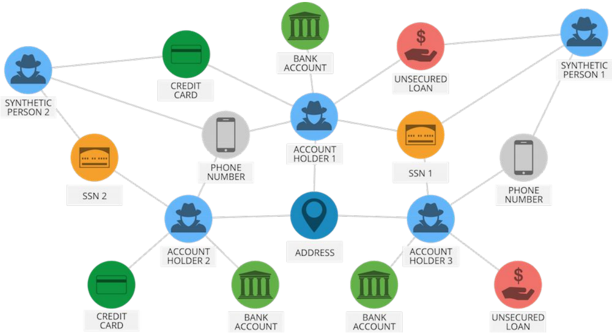
Social Networks



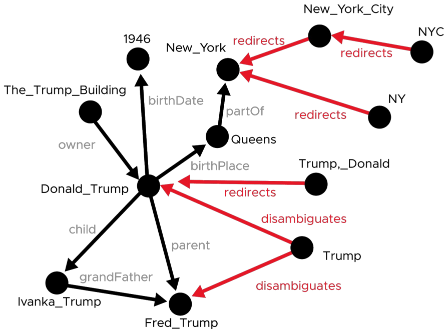
Relational Data



Recommendation Systems



Fraud and Financial Data

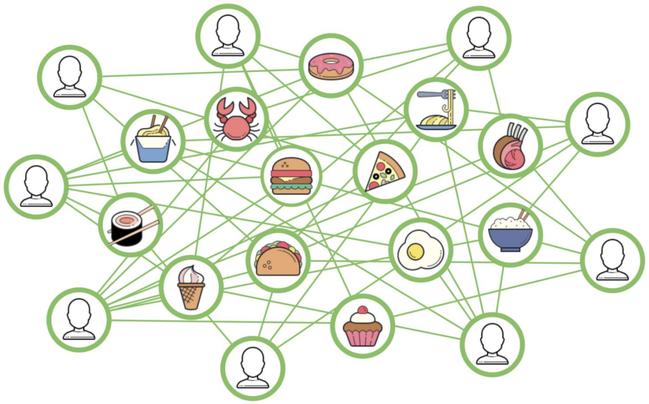


Knowledge Bases

Machine Learning on Graphs

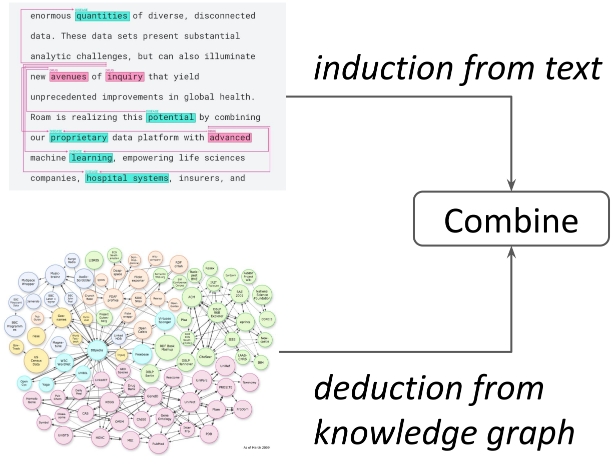
Graph Machine Learning uses the network structure of the underlying data to improve predictive outcomes.

Personalization and Recommendation Systems

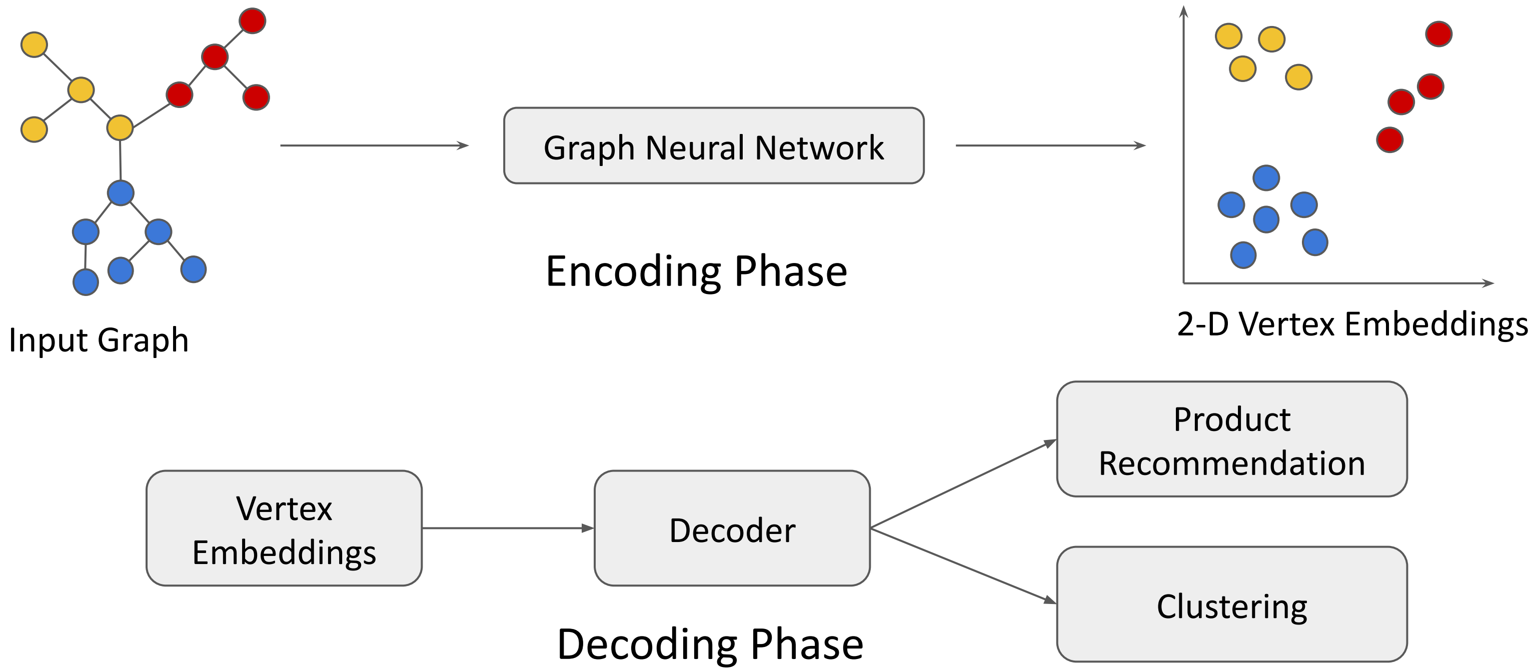


combine features from multiple users and products

Search, Q&A, semantic web

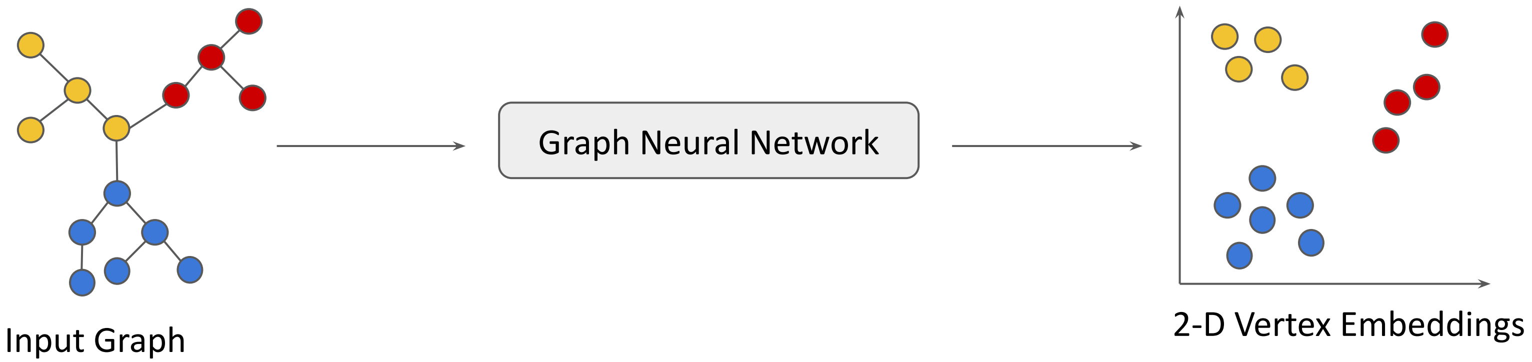


Machine Learning on Graphs



Graph Neural Network maps vertices of graphs to an embedding in a N-dimensional space.

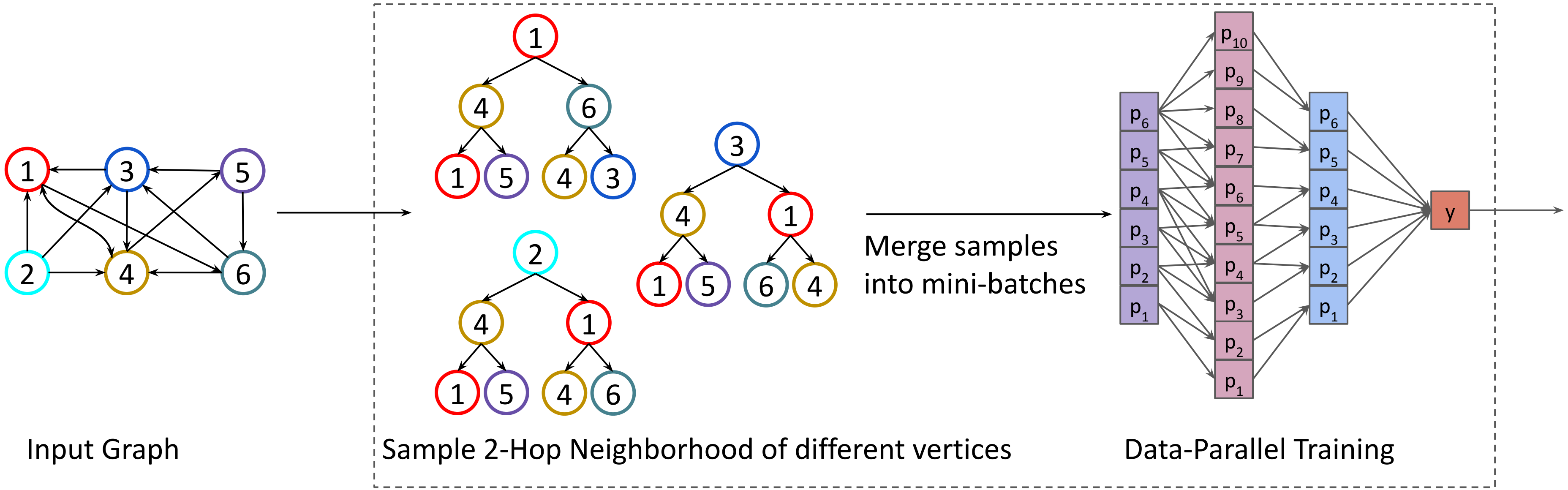
Types of Graph Neural Networks (GNNs)



Two types of Graph Neural Networks:

- Sampling based GNNs samples the input graph and train using these samples
- Whole Graph based GNNs train on the input graph directly

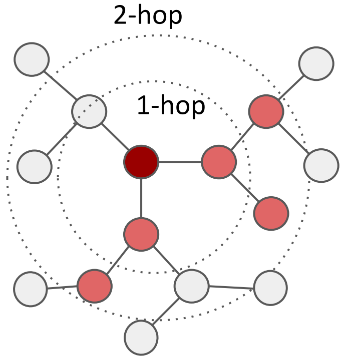
Workflow of a Sampling based GNNs



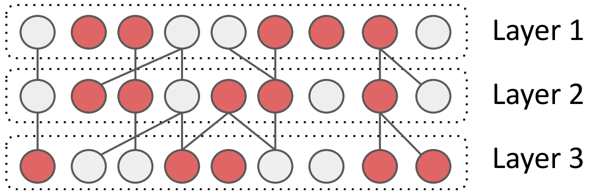
A Sampling based GNN first samples the graph using a Graph Sampling Algorithm and use these samples for data parallel training.

Graph Sampling is an Active Area of Research

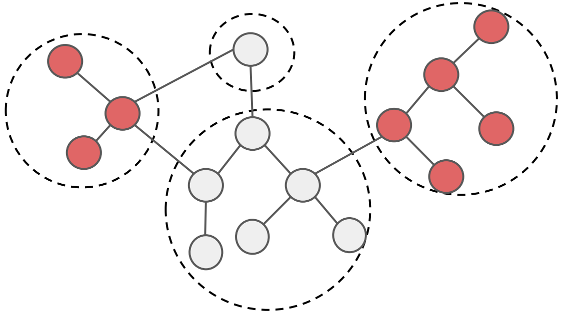
DeepWalk and node2vec are random walks of fixed length starting from a vertex.



GraphSAGE samples 2-Hop Neighborhood of a vertex

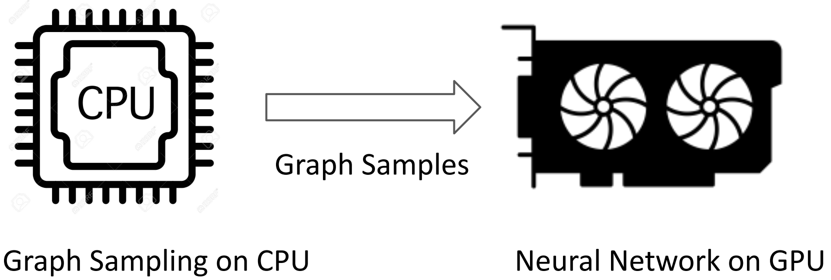


FastGCN and LADIES samples vertices for each layer in their Neural Network.



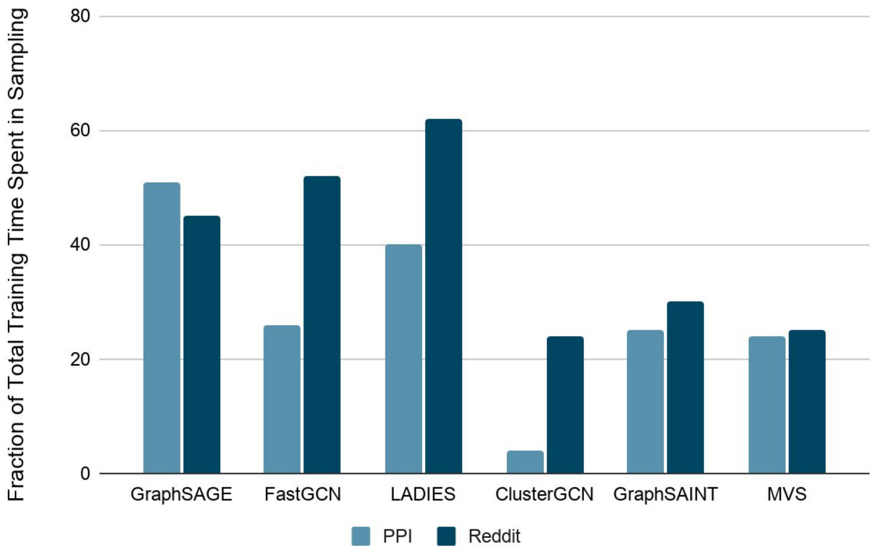
ClusterGCN divides a graph into many clusters and samples one or more of these clusters

ML Domain Experts Implement Graph Sampling on CPU



GNN implementations use CPU for Graph Sampling and GPU for Neural Network

Graph Sampling is a Major Overhead in GNNs



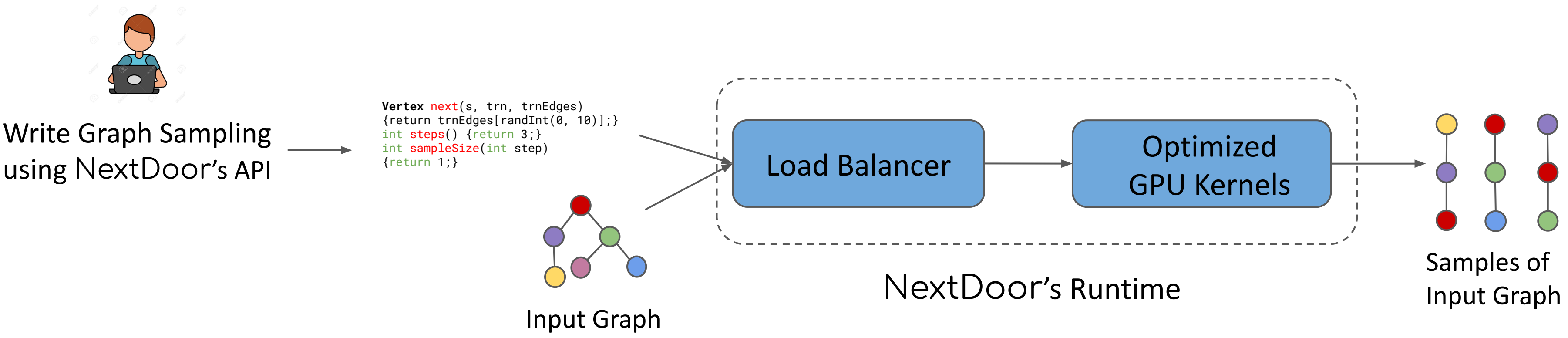
Fraction of time spent in Sampling and Training (per epoch)

Experiments Performed on 32-Core CPU with 1 NVIDIA Tesla V100

How can domain experts have best of both worlds: easy to implement and fast Graph Sampling?

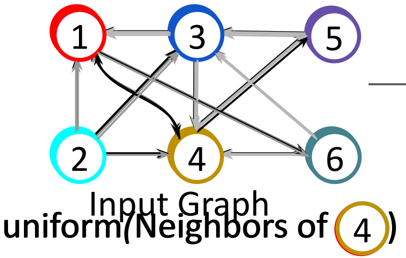
NextDoor: A System to Accelerate Graph Sampling on GPUs

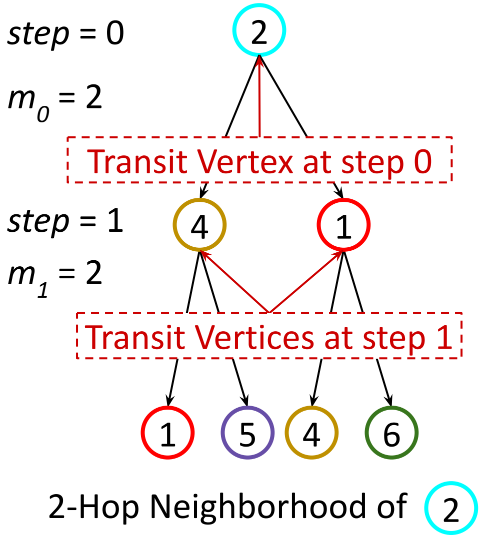
1. A simple yet powerful API to express diverse graph sampling algorithms.
2. A new *“Transit Parallel”* approach to parallel graph sampling to achieve regularity.
3. Load balancing and caching to optimize GPU utilization.
4. Improves end-to-end training time of GNNs by upto 4x.



An Abstraction for Graph Sampling Applications

- A graph sampling application runs for *k* steps.
- In the beginning each sample has one or more root vertices.
- At each step *i*
  - A *transit* vertex for *i* is a vertex whose neighbors may be added to the sample
  - Sample m_i of those neighbors





Graph Sampling using NextDoor's API

- Simple yet powerful API based on the abstraction.
- Implementations need only few lines of code
- We implemented diverse algorithms
  - DeepWalk, PPR, and node2vec
  - K-hop Neighborhood Sampling
  - Importance Sampling
  - ClusterGCN Sampling
  - Minimal Variance Sampling
  - Layer Sampling

```
Vertex next(s, trn, trnEdges) {
    int idx = randInt(0,
                      trnEdges.size()-1);
    return trnEdges[idx];
}

int steps() {return 2;}
int sampleSize(int step)
{return (step == 0) ? 25:10;}
```

GraphSAGE's k-hop Neighborhood in NextDoor

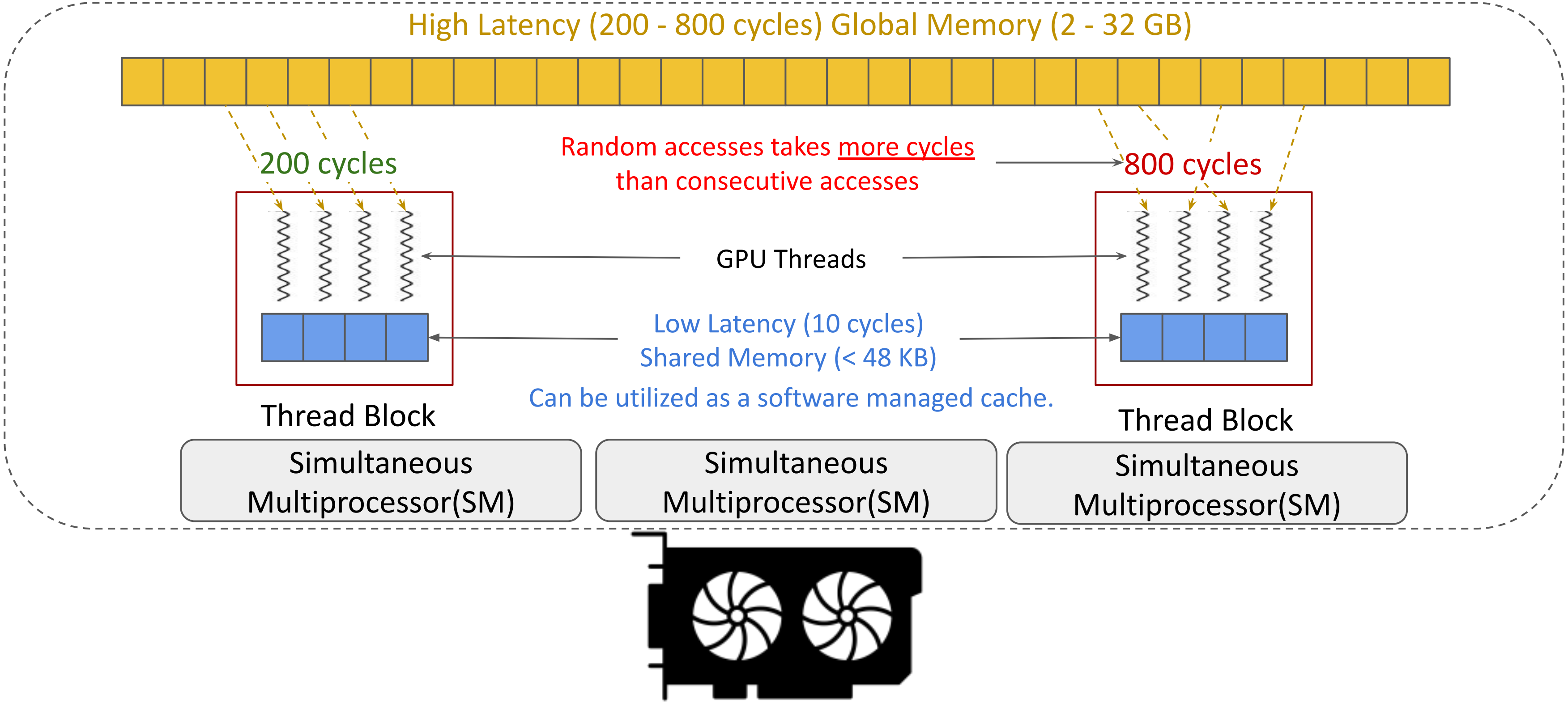
The API provides information to perform effective load balancing and caching:

✓ It provides a distinction between a sample and a transit vertex

✓ Provides the number of steps and number of vertices sampled at each step.

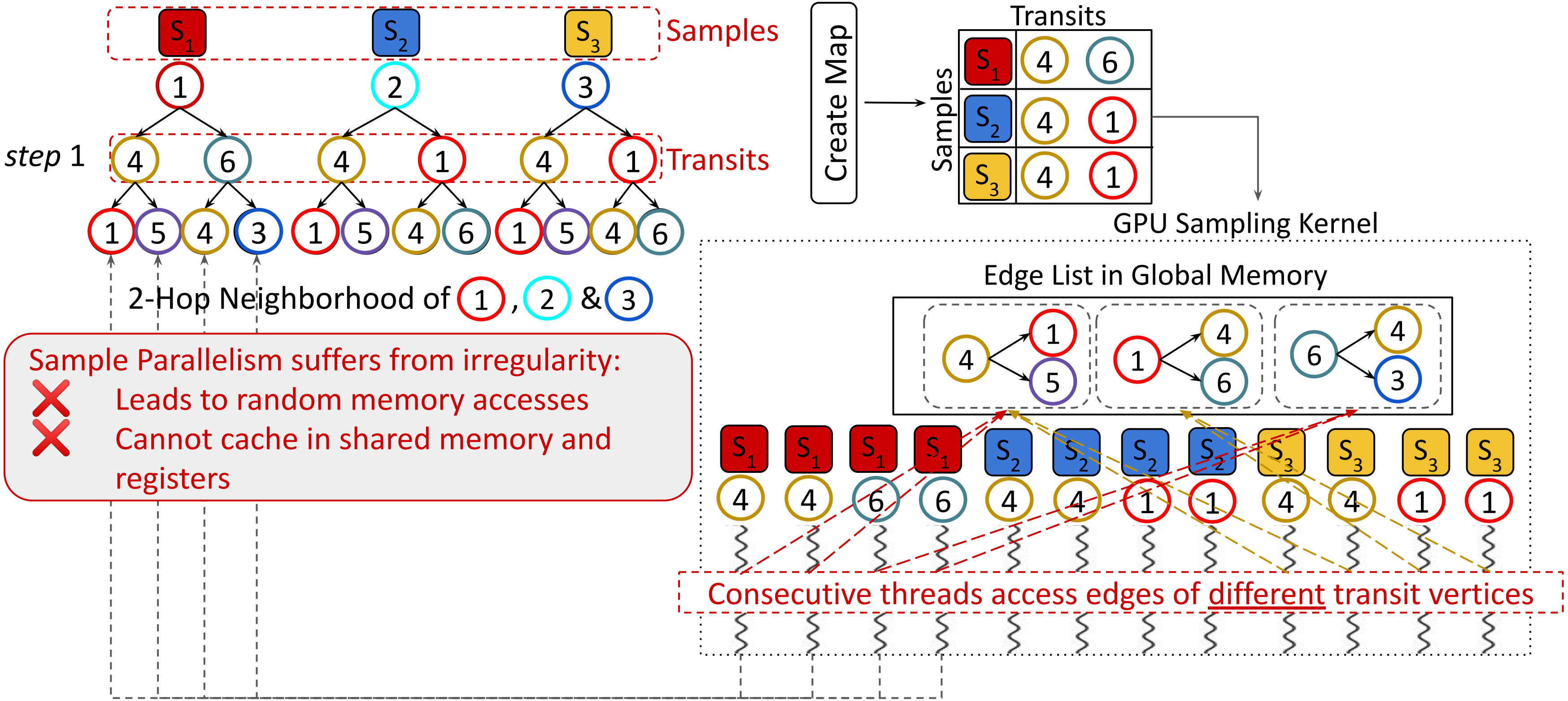
How can we implement this API on a GPU?

A GPU Can Execute Thousands of Threads Simultaneously



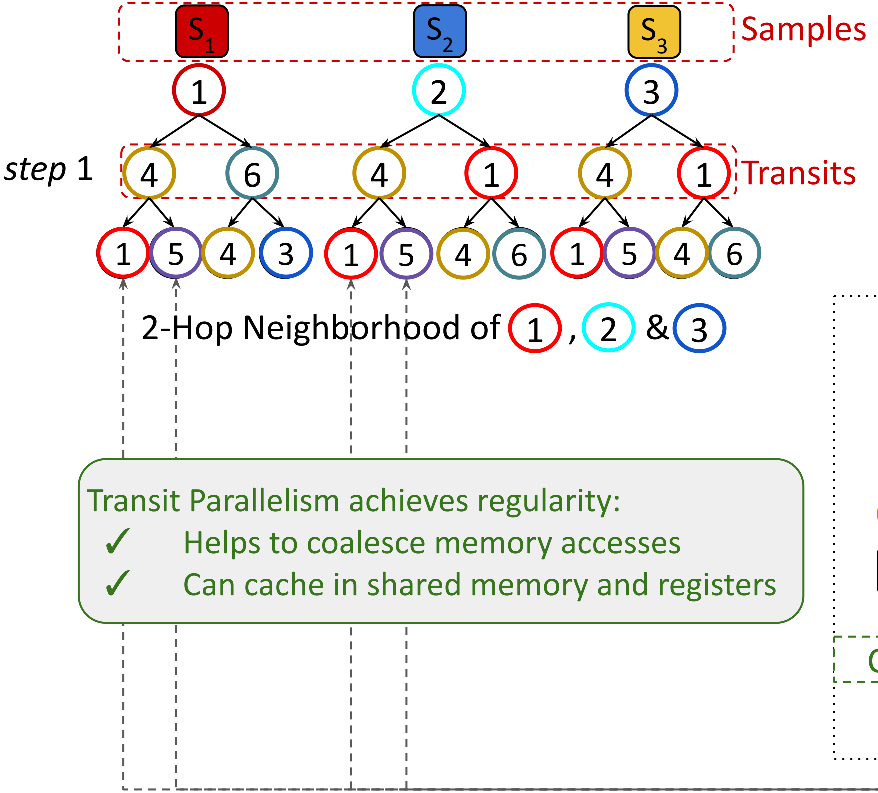
Sample Parallel Graph Sampling on GPUs

Sample Parallel: Assign samples to consecutive threads.



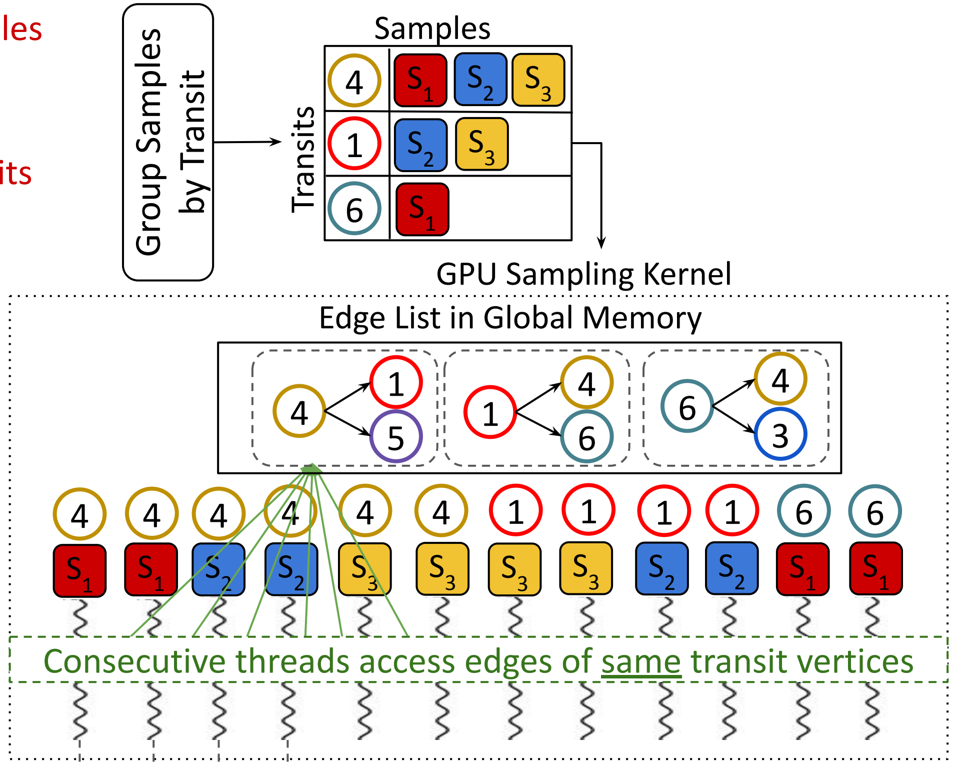
Transit Parallel: Rethinking Parallel Graph Sampling

Transit Parallel: Assign samples *with common transits* to consecutive threads.



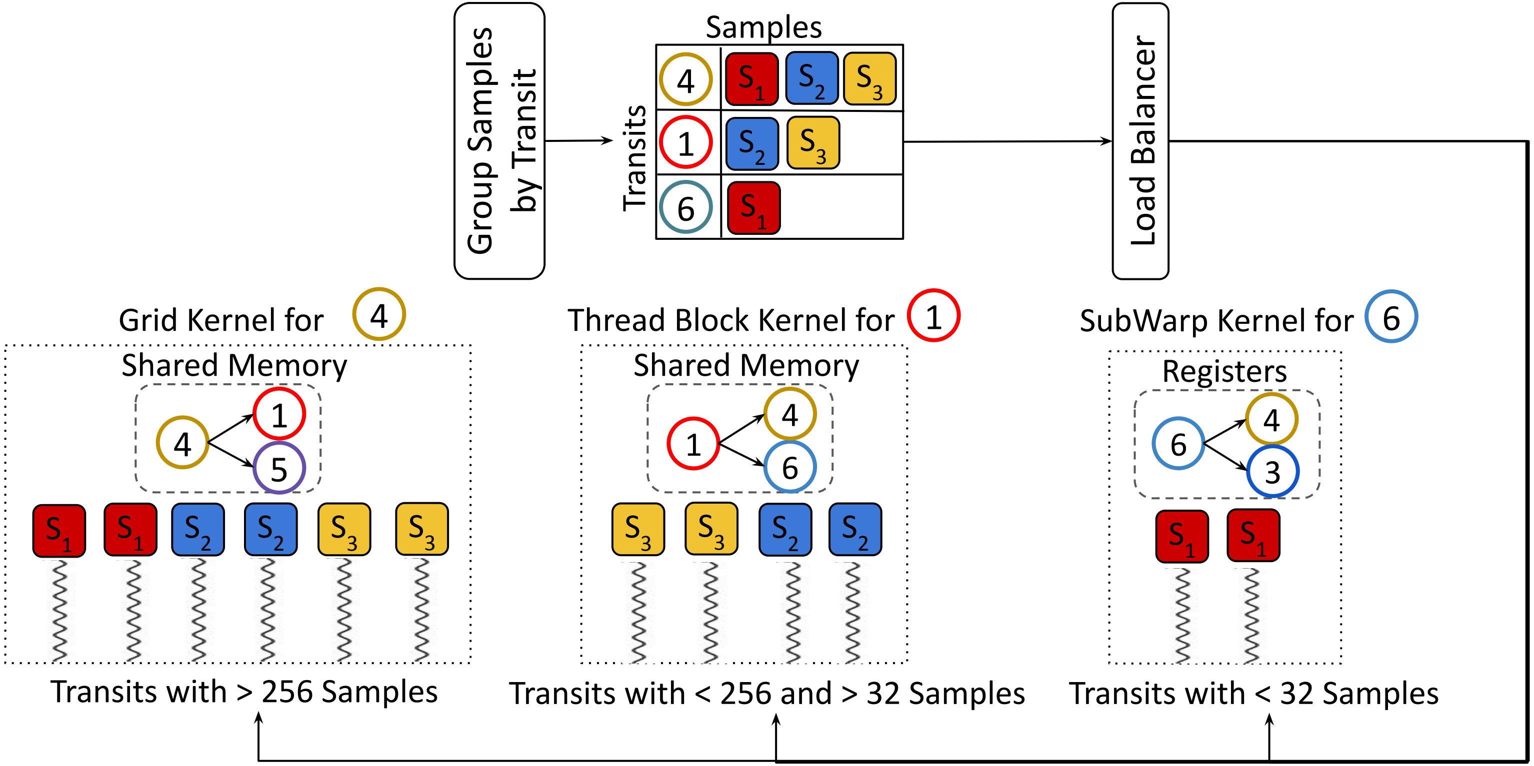
Transit Parallelism achieves regularity:

- ✓ Helps to coalesce memory accesses
- ✓ Can cache in shared memory and registers



Load Balanced Transit Parallelism in NextDoor

NextDoor performs load balancing for different transits and cache of neighbors of a transit.



Implementation

1. NextDoor utilizes efficient parallel radix sort and prefix scan for group by operation and load balancing.
2. NextDoor is implemented in C++14 and CUDA 11.
3. NextDoor is available at https://plasma-umass.org/nextdoor-eurosys21/.

Experiments

Benchmarks:

- DeepWalk
- Personalized Page Rank (PPR)
- node2vec
- Multi Dimensional Random Walks
- K-hop Neighborhood Sampling
- ClusterGCN Sampling
- FastGCN Sampling
- LADIES Sampling
- Minimal Variance Sampling (MVS)
- Layer Sampling

CPU only Baselines:

- KnightKing
- Samplers in existing GNN Systems

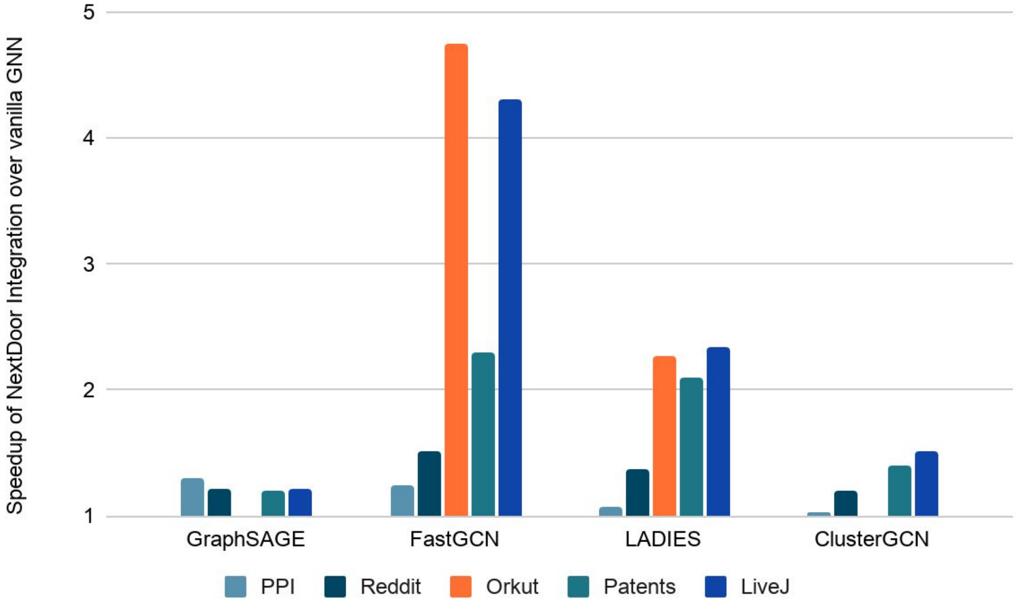
Graph Name	Abrv	Nodes	Edges
Protein-Protein Interactions	PPI	50K	1.4M
com-Orkut	Orkut	3M	117M
cit-Patents	Patents	3.77M	16.5M
soc-LiveJournal1	LiveJ	4.8M	68.9M

Experiments

Evaluation System:

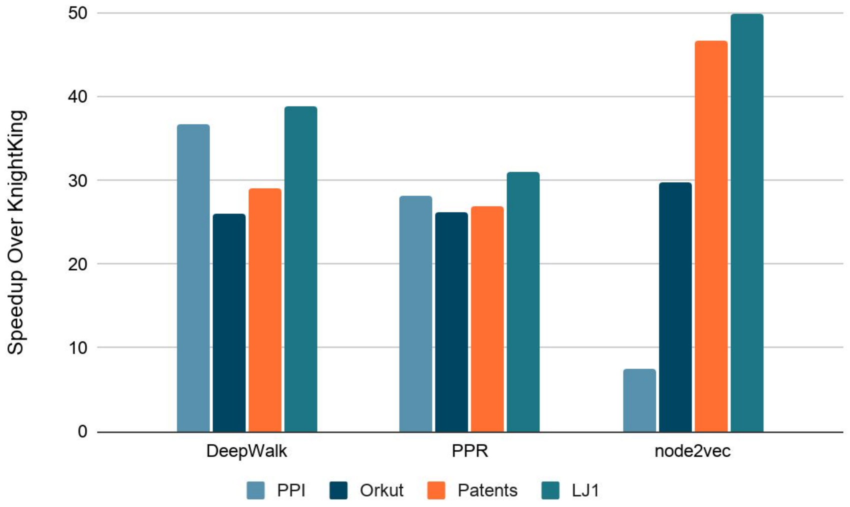
- Dual Socket 16-Core Intel Xeon Silver CPUs
- 4x NVIDIA Tesla V100
- 128 GB of RAM
- Ubuntu 18.04
- CUDA 11.2

NextDoor: End-to-End Speedups with GNN Training

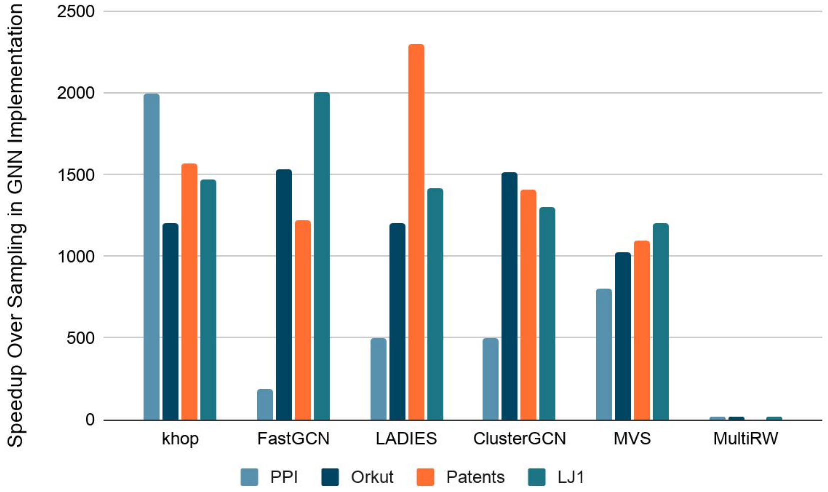


NextDoor's fast Graph Sampling implementations significantly improves training time of GNNs.

NextDoor against existing Graph Sampling Implementations



Speedup over KnightKing for Random Walks



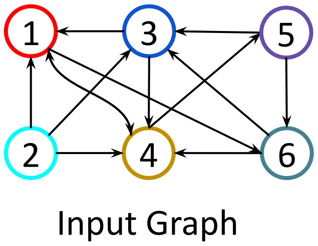
Speedup over GNN implementations

NextDoor achieves orders of magnitude speedup over CPU baselines

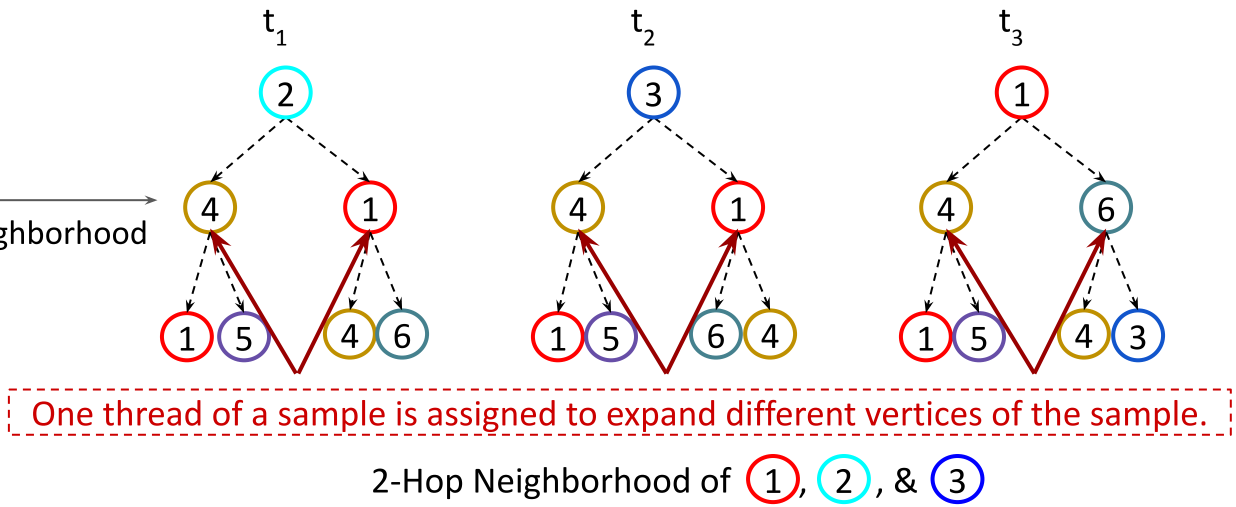
Summary

- Graph Sampling takes up to 62% of training time in Graph Neural Networks.
- Efficient Graph Sampling on GPUs is hard due to irregular nature of graphs.
- NextDoor: accelerate Graph Sampling using GPUs.
  - ✓ API to easily write efficient graph sampling applications.
  - ✓ Runtime that optimizes memory accesses in GPUs and efficiently balances load.
- NextDoor achieves orders of magnitude improvement over existing solutions.
- NextDoor improves end-to-end training time of GNNs by up to 4 times on large graphs.

Existing Implementations Have Limited Parallelism

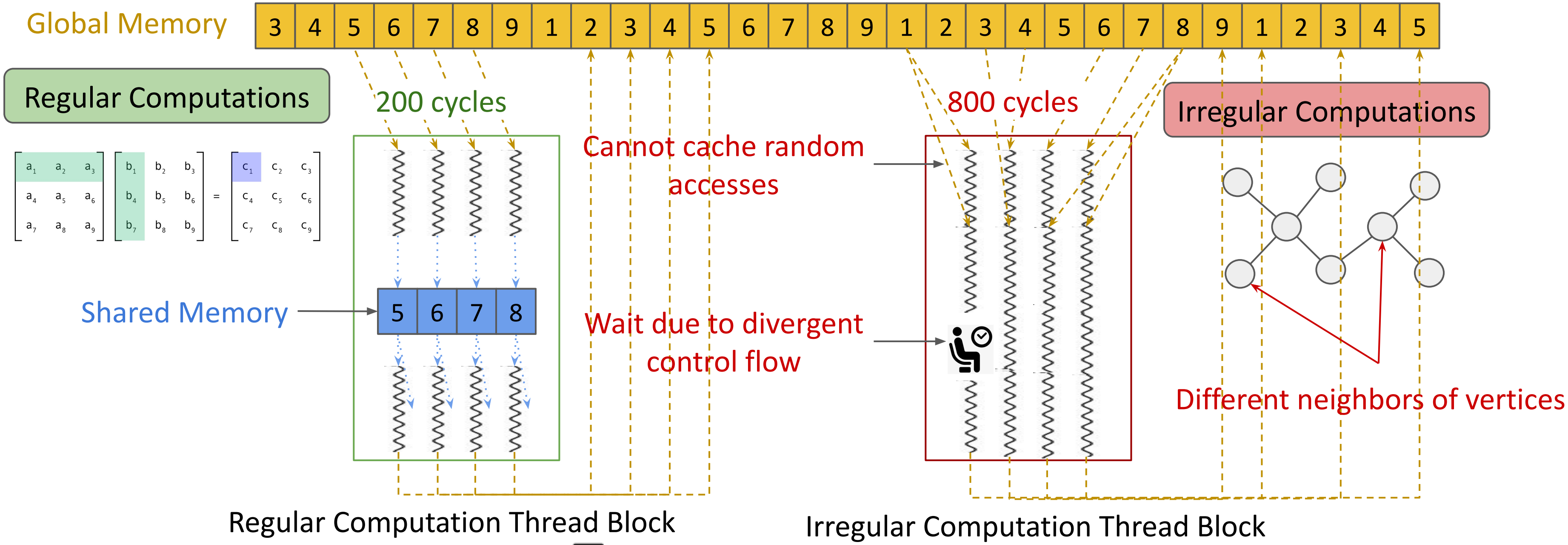


2-Hop Neighborhood

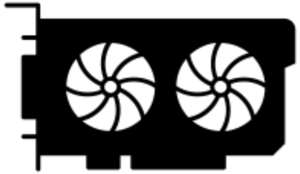


Is it possible to increase parallelism in the standard approach to parallel Graph Sampling?

Regular Code achieves High Performance on GPU

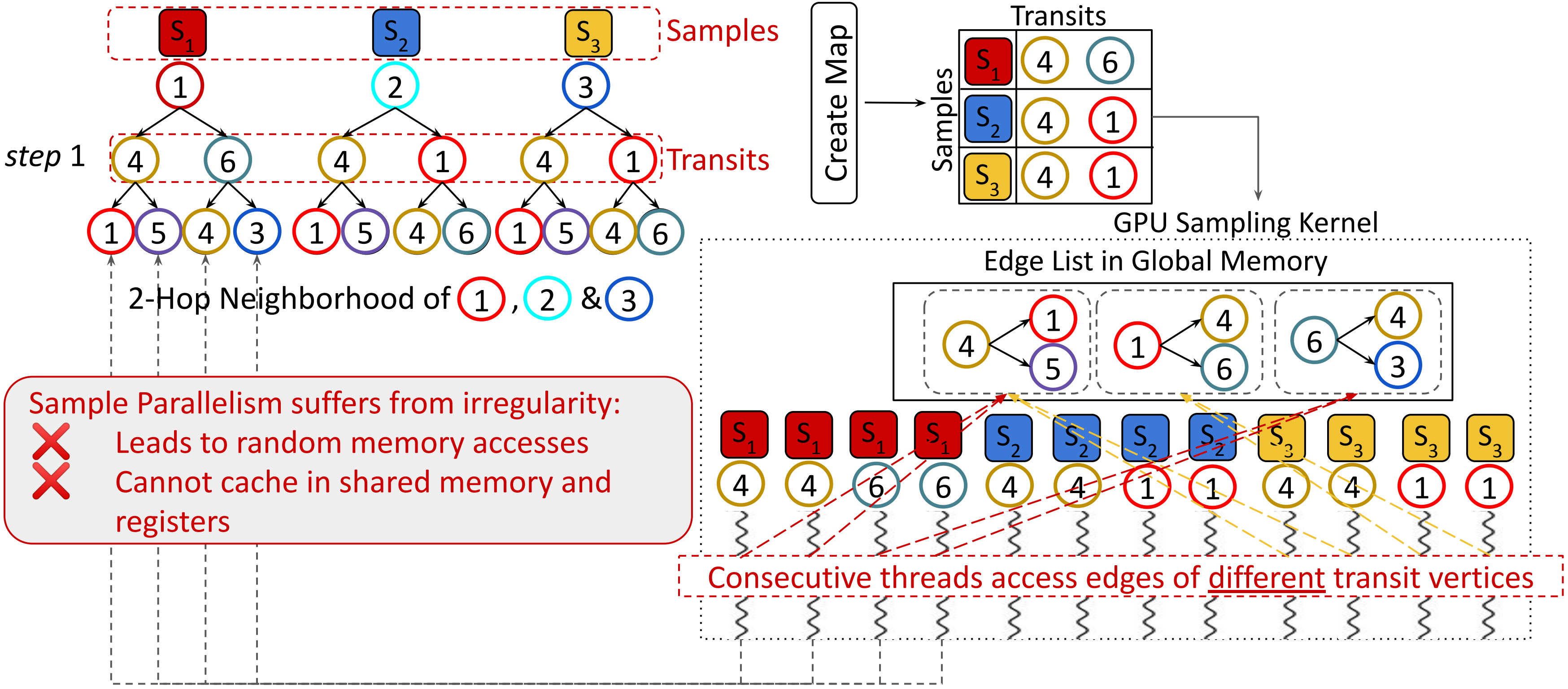


- ✓ Consecutive Loads
- ✓ Non-Divergent Control Flow
- ✓ Cache in Shared Memory



- ❌ Random Memory Access
- ❌ Divergent Control Flow
- ❌ Cannot Cache in Shared Memory

Sample Parallel Graph Sampling is Irregular

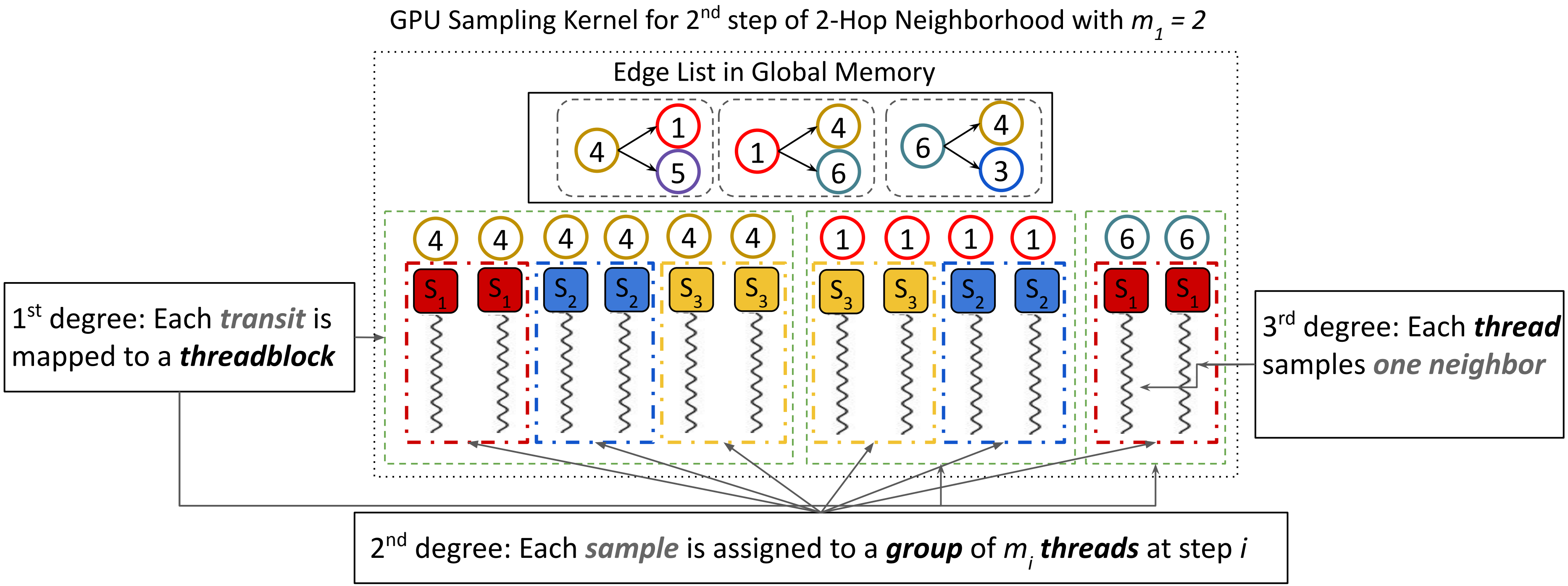


Sample Parallelism suffers from irregularity:

- ❌ Leads to random memory accesses
- ❌ Cannot cache in shared memory and registers

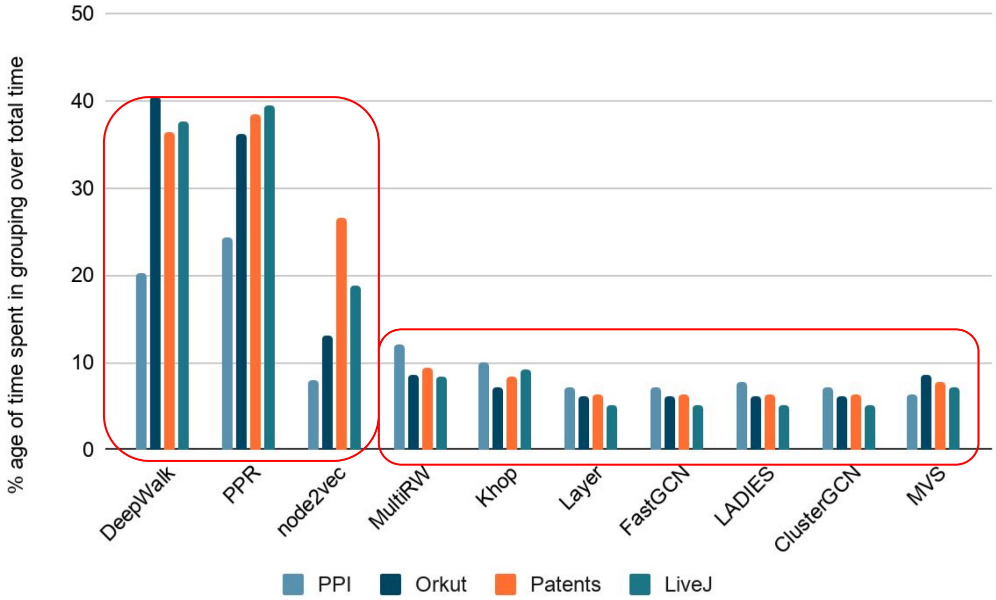
Scheduling Transit Parallel in NextDoor

The NextDoor API exposes *three degrees of parallelism* that match the GPU architecture



Additional Overhead of Transit Parallel over Sample Parallel

- Transit Parallel uses a Group By operation.
- Random Walks spend up to 40% time in grouping operation.
- Despite overhead Transit Parallel still achieves up to 2x speedup over Sample Parallel in Random Walks.
- Other Application spend less than 10% time in grouping operation.
- Random Walks spends more time because they sample only one neighbor of the transit vertex at each step.

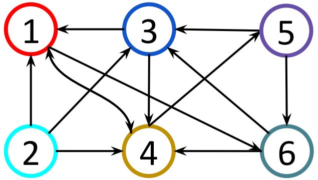


Percentage of time spent in *group by* operation over total time.

Standard Approach to Parallel Graph Sampling

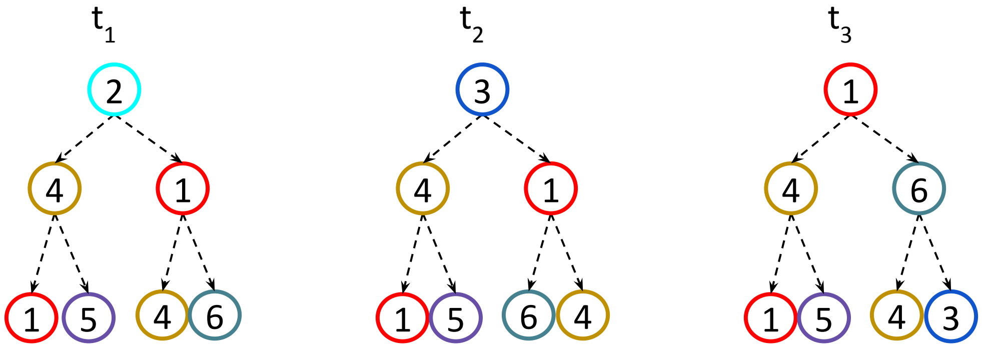
Sample Parallel Graph Sampling

- Samples can be expanded in parallel by assigning samples to a single thread.
- Approach adopted by existing systems.



Input Graph

2-Hop Neighborhood



2-Hop Neighborhood of 1, 2, & 3

Can we use this approach for a GPU based parallel graph sampling?

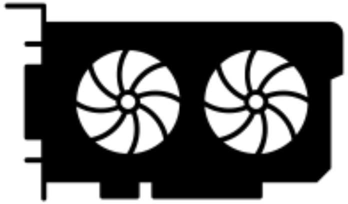
Leverage GPUs for Graph Sampling is hard!

Regular Computations

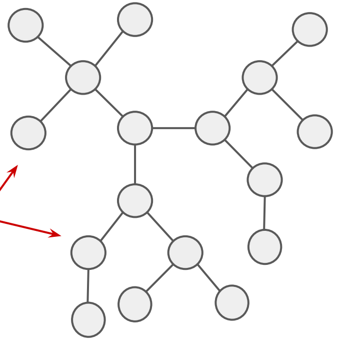
$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \\ a_{31}+b_{31} & a_{32}+b_{32} & a_{33}+b_{33} \end{bmatrix}$$

- ✓ Consecutive Memory Accesses
- ✓ Convergent Control Flow
- ✓ Utilize faster shared memory



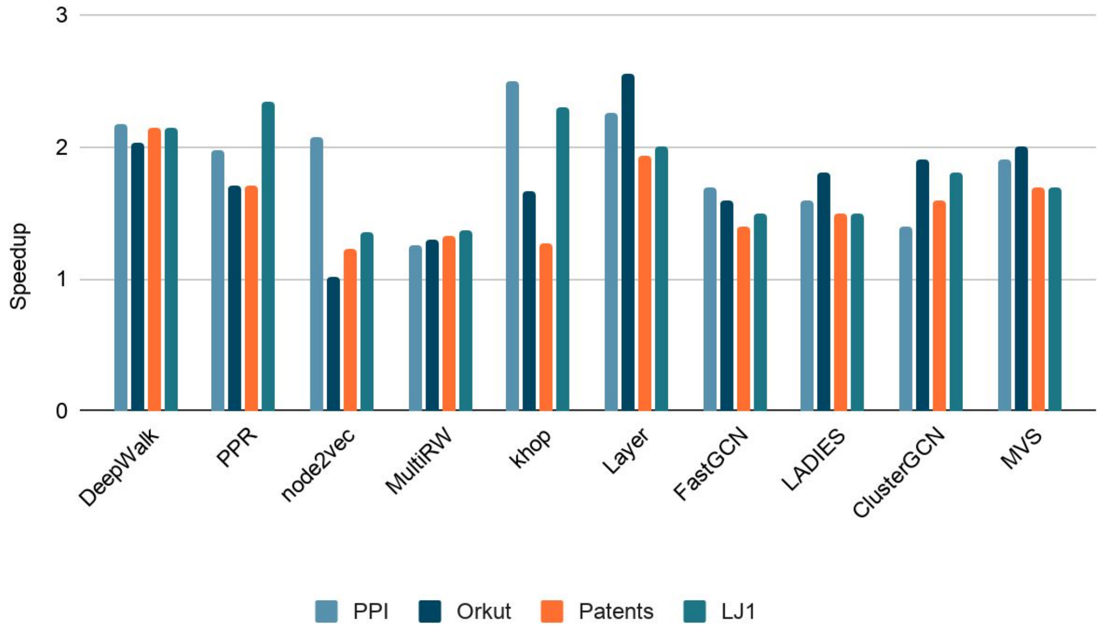
Irregular Computations



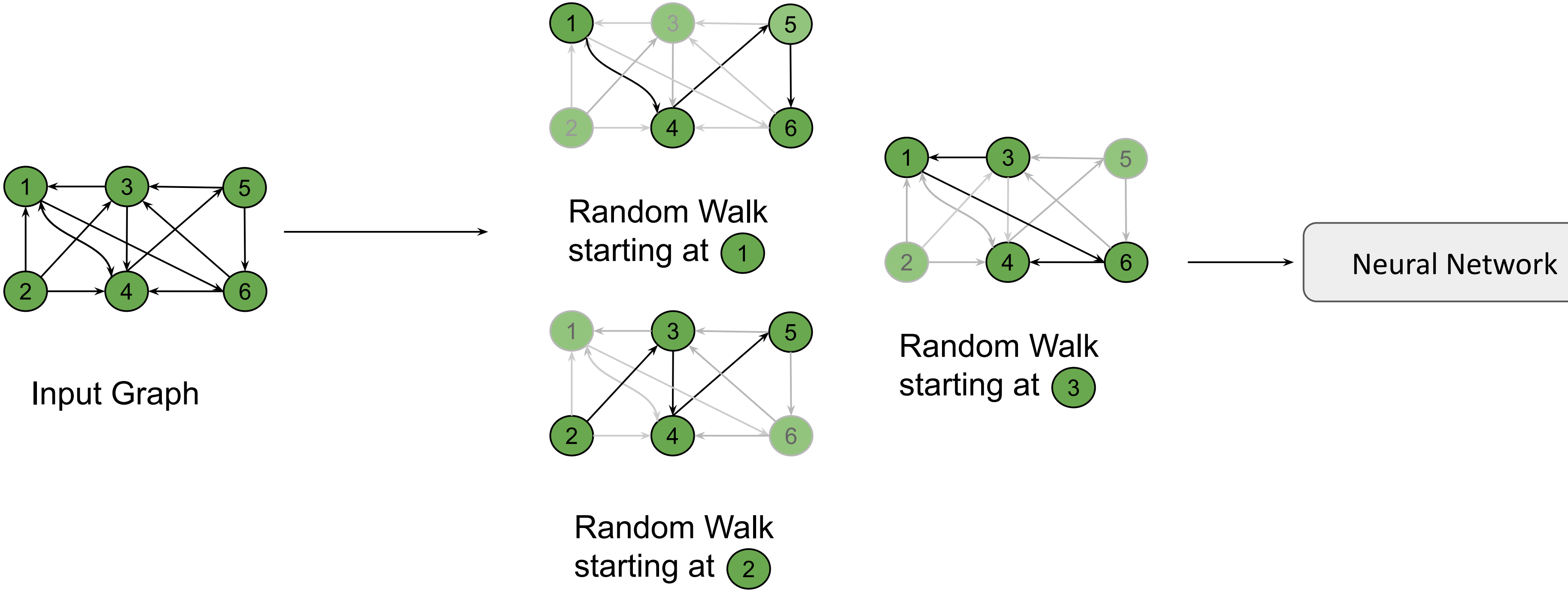
Different neighbors of vertices

- ❌ Random Memory Access
- ❌ Divergent Control Flow
- ❌ Cannot Utilize faster memory

NextDoor speedup over Transit Parallel



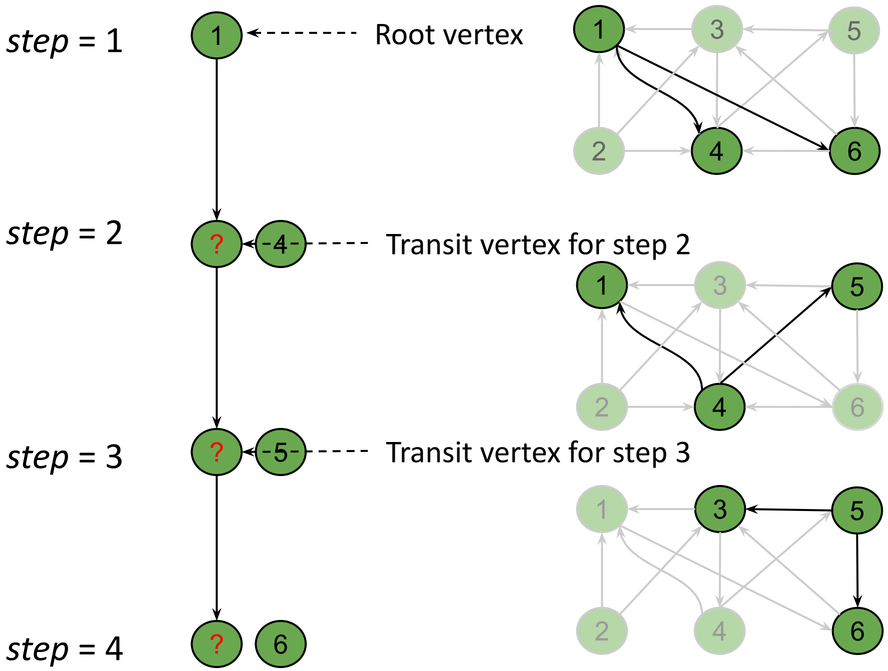
Workflow of Graph Neural Network Training



Each Random Walk is a mini-batch

An Abstraction for Graph Sampling Applications

- A graph sampling application runs for *k* steps.
- Each execution of application produces one *sample* of the graph.
- In the beginning each sample has root vertice(s).
- At step *i* , the application samples m_i vertices.
- Function *next* describes the sampling procedure.
- A *transit* vertex at a step *i* is a vertex whose neighbors may be sampled at step *i*.



A Random Walk of length 4 starting from 1