Efficient Scaling of Dynamic Graph Neural Networks

Venkatesan T. Chakaravarthy, Shivmaran S. Pandian, Saurabh Raje, Yogish Sabharwal, Toyotaro Suzumura, Shashanka Ubaru

IBM Research

SC21
St. Louis, MO | science & beyond.

Graph Neural Networks

Classical Learning Paradigms

- Entities treated independently. Embedding derived from own features

Entities x Features

	F1	F2	F3
e1			
e2			
e3			
e4			

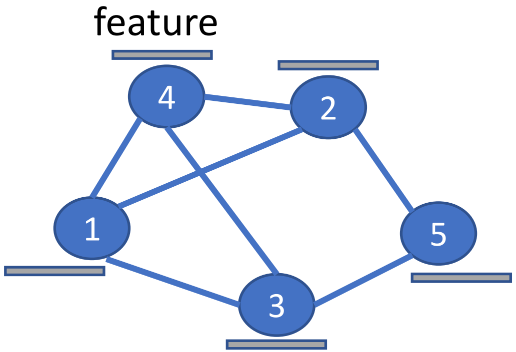
→

Classical Models

Deep neural nets
Decision trees
SVM

Graph Neural Networks

- Inter-relationships represented as graph
    - Social network - friendship
- Embedding derived from
    - Own features
    - Neighborhood features
- Prior Work
    - Various models and applications
    - Distributed, multi-GPU implementations
    - Packages: DGL, PyTorch Geometric, Aligraph

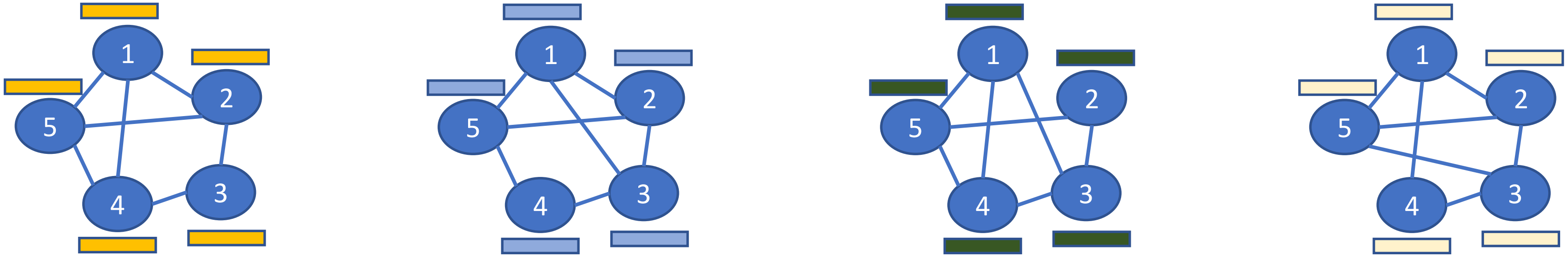


Tell me your friends
and
I will tell who you are

-Assyrian proverb

Dynamic Graph Neural Networks

- Graphs that evolve over time.
- Discrete Time Dynamic Graphs (DTDG)
    - Represented by taking snapshots at regular intervals
    - Topology (edges) and vertex features vary.
- Examples:
    - Social networks: Take snapshot each day
    - Financial transaction networks: Transactions during each week
- Models and Applications. Combine:
    - GNN for topological aspects
    - Recurrent Neural Networks (RNN) for time-series aspects
- Scalability has not been studied



Snapshot 1   Snapshot 2   Snapshot 3   Snapshot 4

Our Work: Scaling Dynamic Graph Neural Networks

First study on scaling dynamic graph neural networks.

- Multi-node, multi-GPU implementation

Optimizations tailored to dynamic GNN, exploiting dynamic graph properties

1. GPU Memory Optimization
   - Gradient checkpoint
2. CPU-GPU Snapshot Transfer
   - An efficient graph difference based strategy
3. Distribution Strategy
   - Baseline: Vertex-partitioning used in static GNN
   - Snapshot partitioning: Scalable strategy

Experimental study

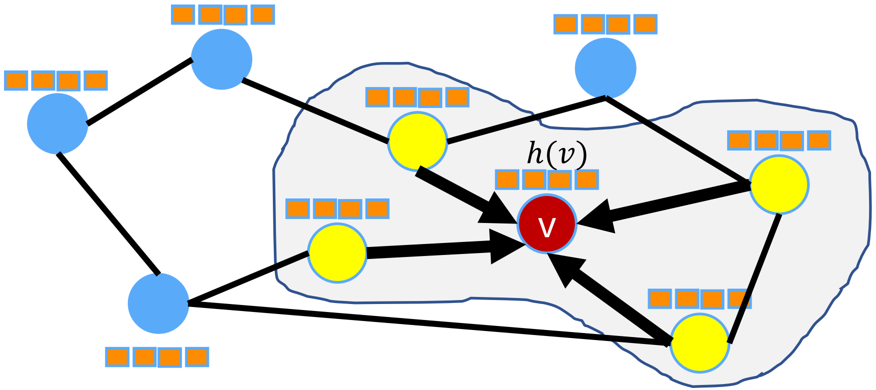
- Large real-life graphs with billion edges
- Scaling study up to 128 GPUs

Outline for rest of the talk

- Graph neural networks
- Dynamic graph neural networks
- Our work: Scaling dynamic GNN

Graph Convolution – Neighborhood Aggregation

Each vertex updates its features by aggregating features from neighbours



Example aggregation operations

Mean

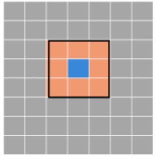
$$h_{new}(v) = \frac{\sum_{u\in\Gamma(v)} h(u)}{\deg(v)}$$

Laplacian

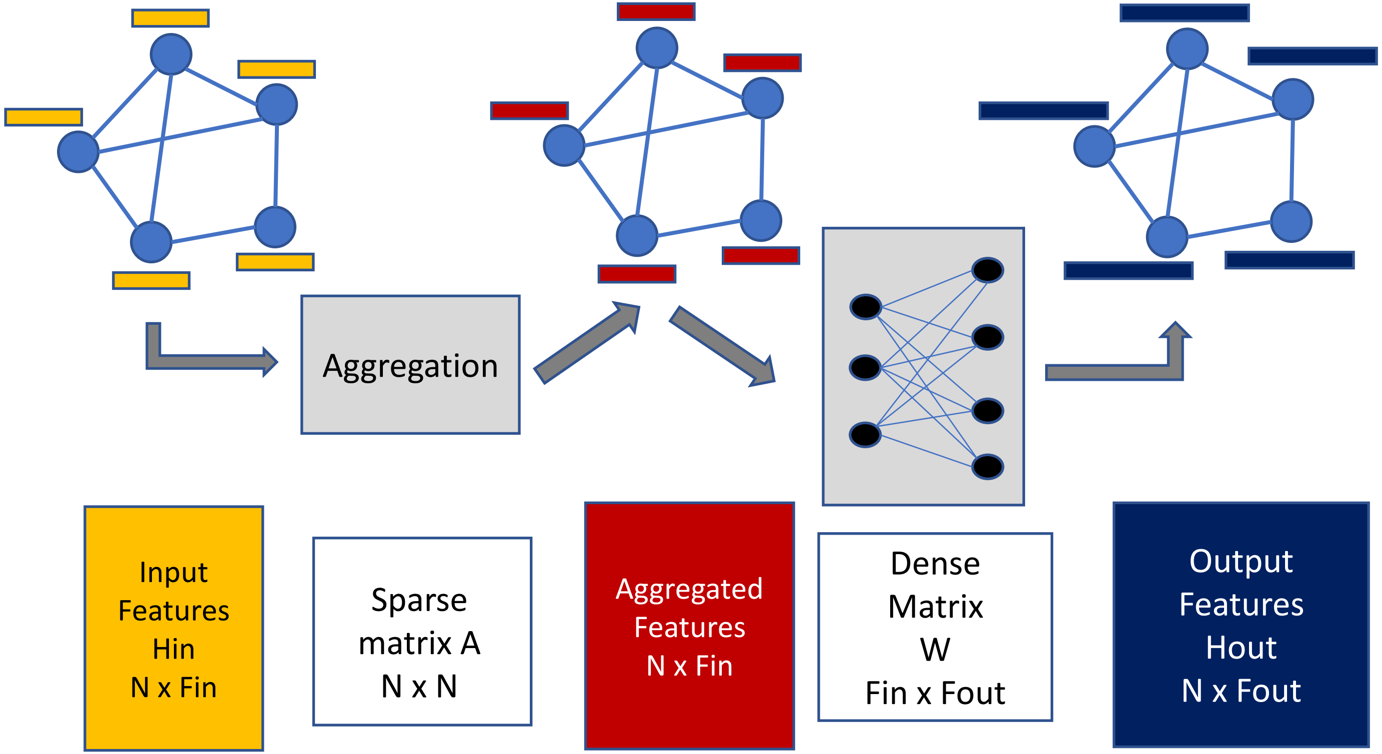
$$h_{new(v)} = \sum_{u\in\Gamma(v)} \frac{h(u)}{\sqrt{d(u)\cdot d(v)}}$$

Similar to convolution over images

- Each pixel updates by aggregating over neighboring pixels

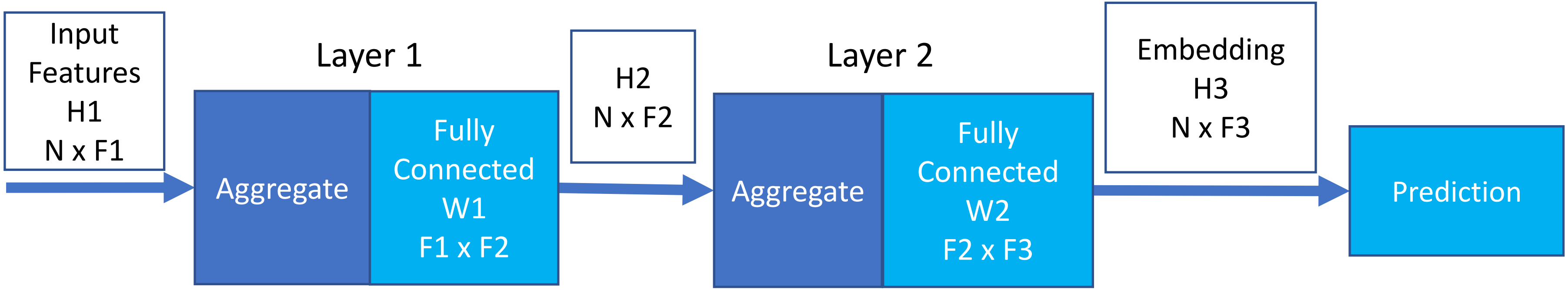


Graph Convolution Layer



Graph Convolution Networks – Multiple Layers

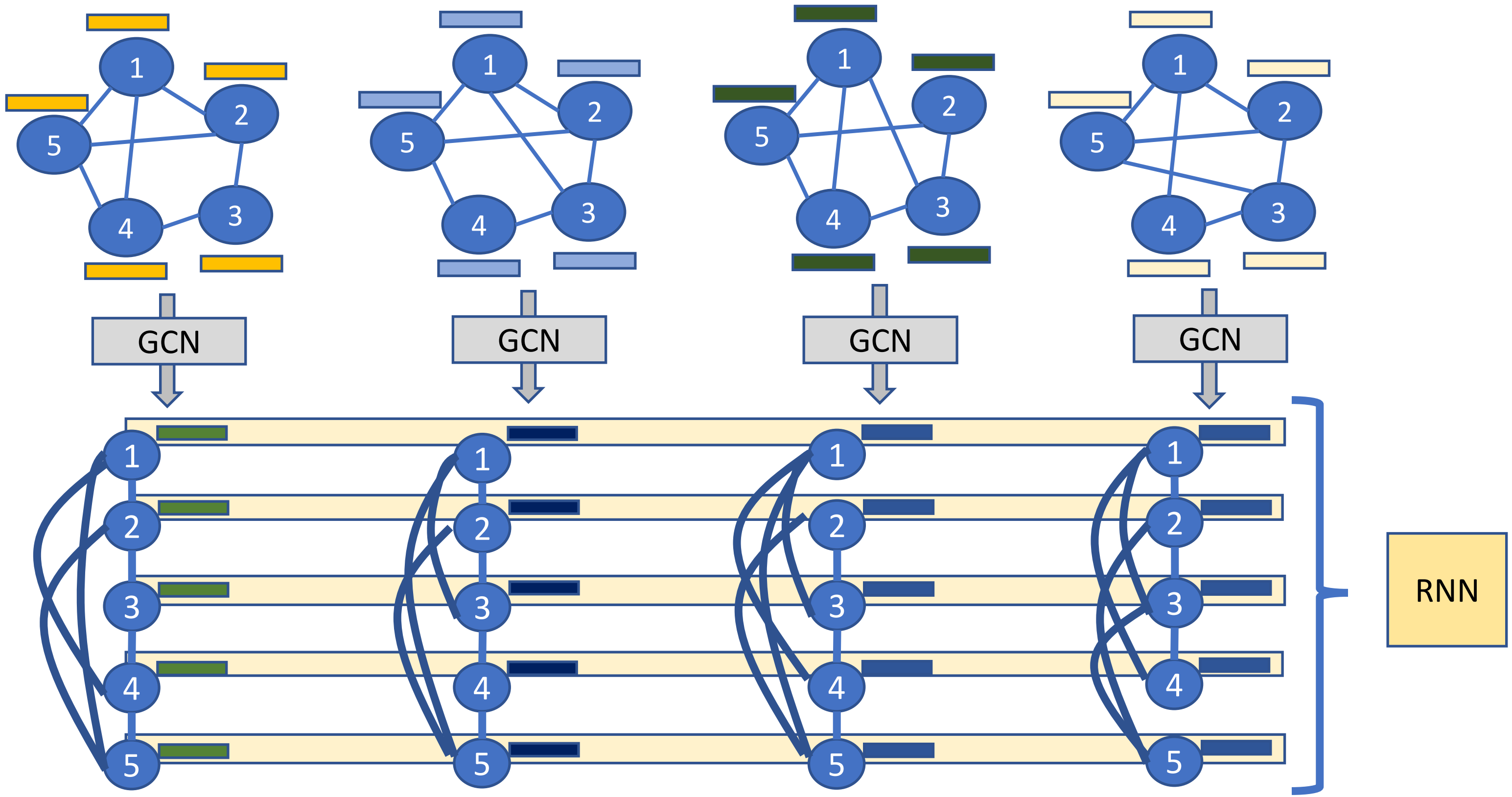
- Single layer
  - Assimilates information immediate neighbors
- K-layers
  - Assimilates information from k-hop neighborhood
- Classical multi-layer perceptron
  - Similar, but without aggregation
- More sophisticated GNN models have been proposed
  - This framework is sufficient in our context



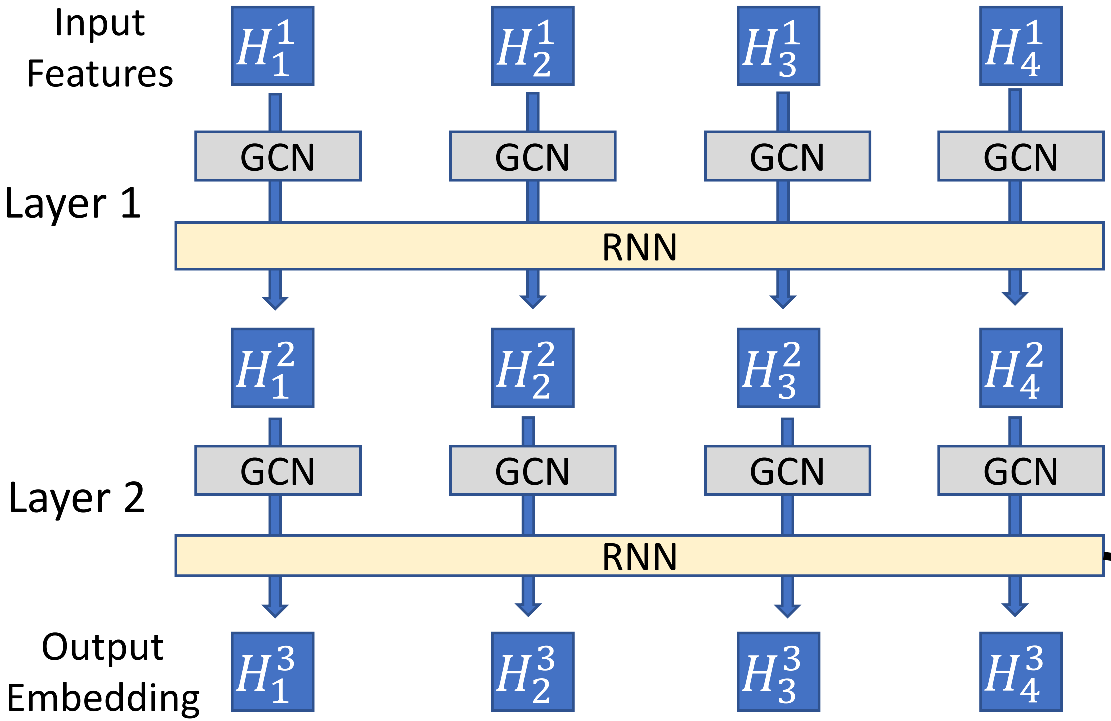
Dynamic Graph Neural Networks (DTDG): General Framework

GCN component
- Captures graph topological aspects
- Operates independently on each snapshot

Recurrent Neural Net (RNN) component
- Captures time-series aspects
- Operates independently on each vertex



Dynamic Graph Neural Networks (DTDG): General Framework



Predictions

- Prediction for future snapshots
- Vertex/edge prediction for all snapshots

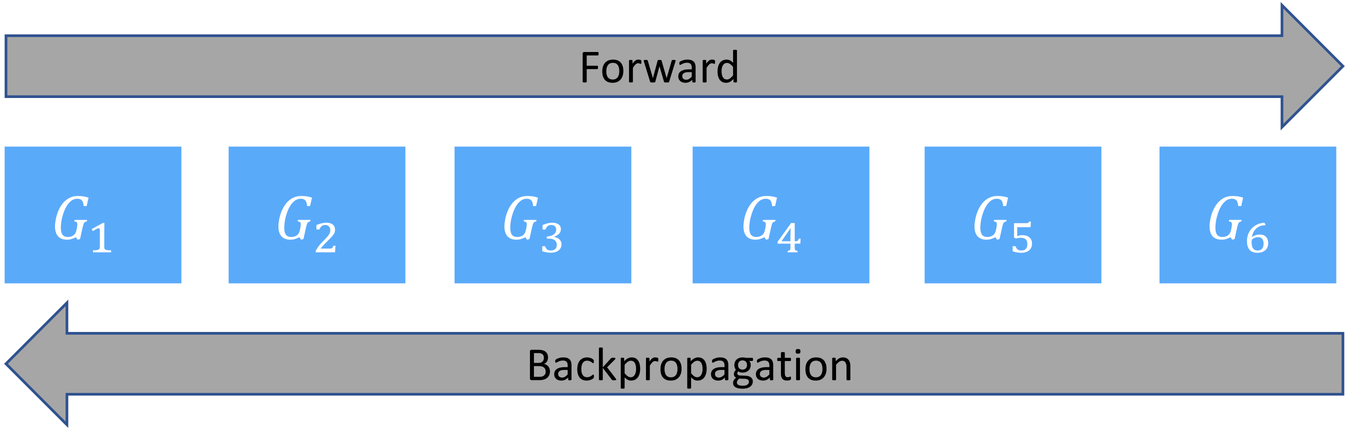
Representative prior models

Model	RNN
TensorGCN	M-product
WD-GCN	LSTM
EvolveGCN	LSTM over weights

Same framework
but differ in RNN model

Scaling Dynamic GNN: GPU Memory

- Forward pass
  - RNN processes snapshots from 1 to T
- Backpropagation of gradients
  - In the reverse direction from T to 1
- All snapshots and intermediate activations are stored in GPU
- Leads to GPU memory bottleneck



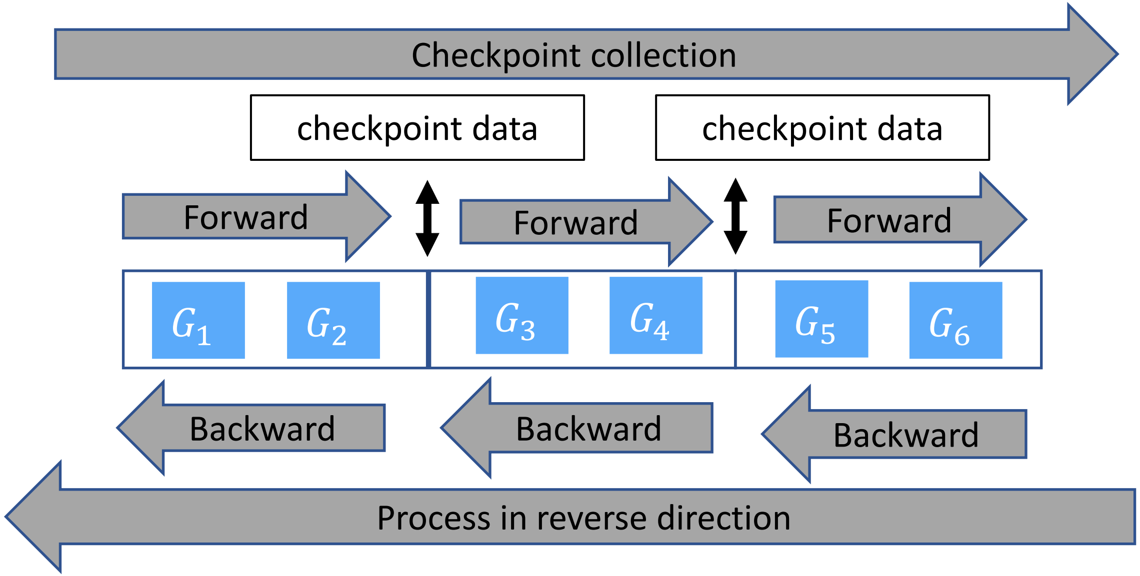
Optimization: Gradient Checkpoint

Gradient checkpoint

- Popular technique in deep learning that reduces memory usage

Dynamic GNN

- Divide timeline into blocks
- First pass : Forward direction to collect checkpoint data
- Second pass: Reverse direction, for each block
    - Forwards pass using checkpoint data
    - Backpropagation within the block



Memory

- Checkpoint data
- Intra-block memory

Number of blocks

- Hyperparameter that gives trade-off.

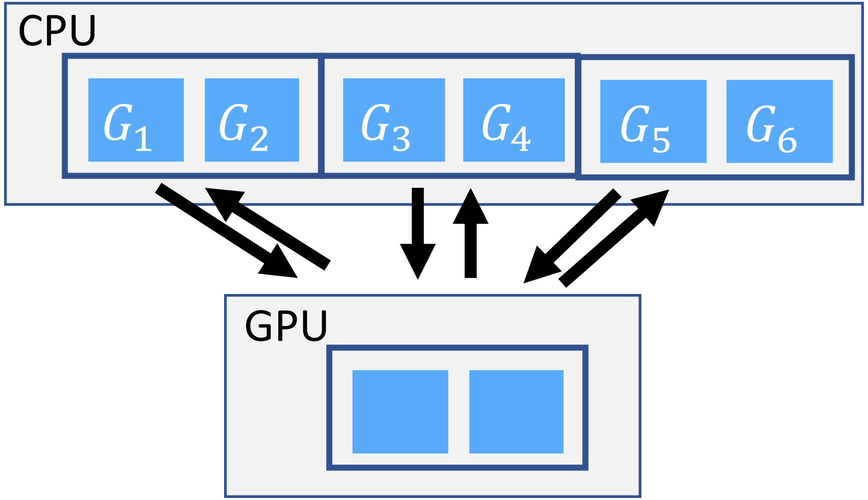
CPU-GPU Transfer

Gradient Checkpoint

- Store snapshots in CPU.
- Move block-by-block on demand basis
- Memory needed – in the order of single block size

Baseline Method

- Direct transfer of the snapshots
- Significant execution time overhead



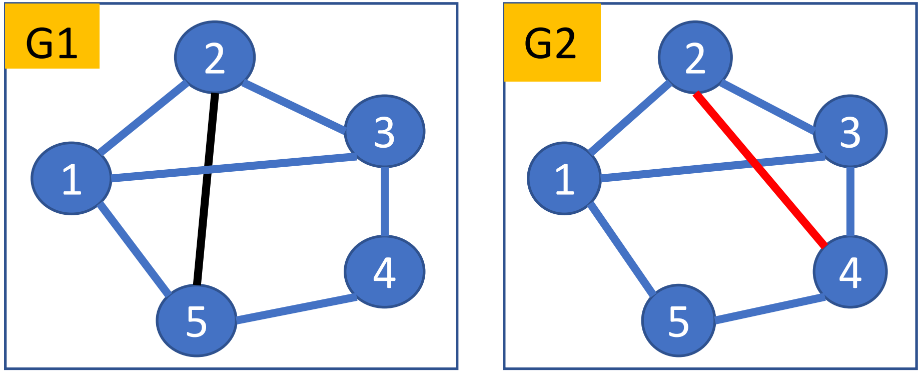
Optimization: Graph-difference Based CPU-GPU Transfer

Intuition

- Real-life graphs evolve slowly
- Consecutive snapshots are similar
- Smoothening by TensorGCN and EvolveGCN increases density and similarity

Strategy

- Do not transfer entire snapshot
- Transfer only the difference with respect to previous snapshot
- Reconstruct the snapshot in GPU



Difference

- Delete (2, 5)
- Insert (2, 4)

Transfer time: up to 4x reduction
Overall time: up to 40% reduction

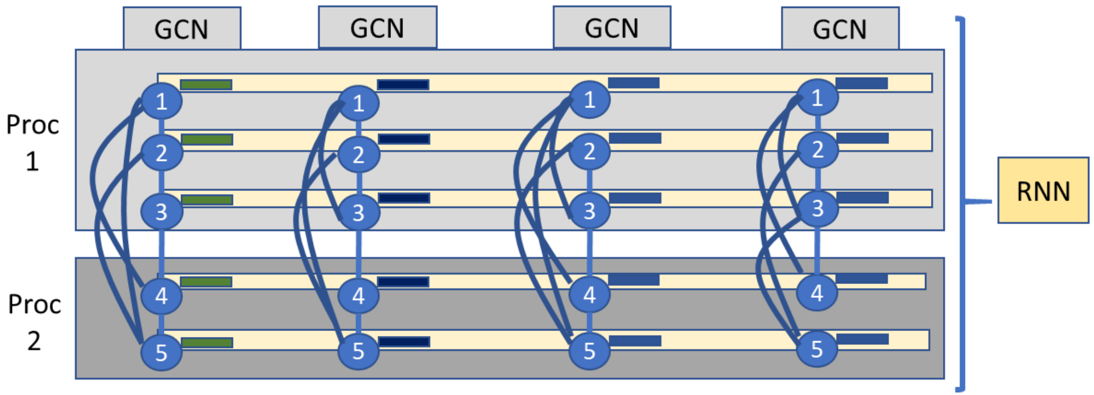
Distribution Strategy: Baseline Vertex-Partitioning Approach

Vertex-Partitioning

- Used in static GNN partitioning
- Partition vertices equally among the processors

Communication

- RNN: Communication free.
  - Vertex features across timeline owned by same processor
- GCN
  - Communication for all edges that cuts across processors
- Hypergraph partitioners used to find a good partition



Disadvantages

- Communication volume increases
  - Graph density
  - Number of processors
- Irregular communication pattern
  - High implementation overhead (on GPU)
- Poor scaling
- Expensive hyper-graph partitioning

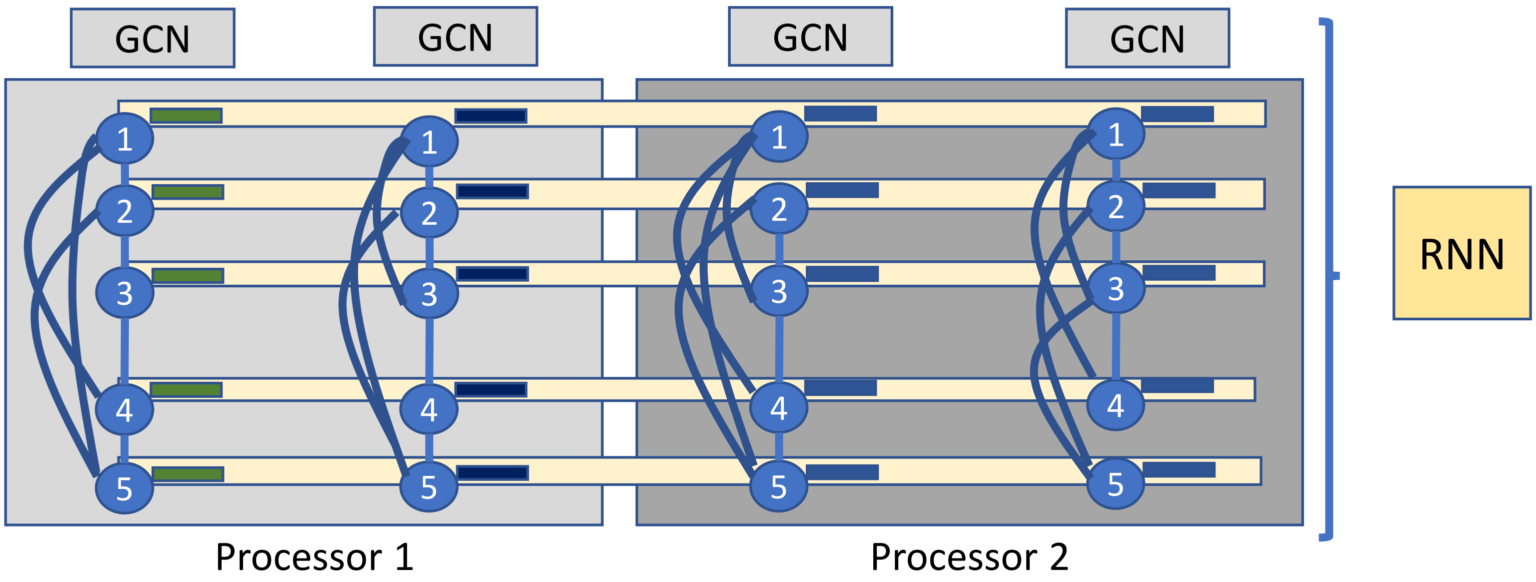
Optimization: Snapshot-Partitioning Approach

Snapshot Partitioning

- Partition snapshots among the processors

Communication

- GCN is communication free
  - Entire snapshot owned by a single processor
- RNN needs communication



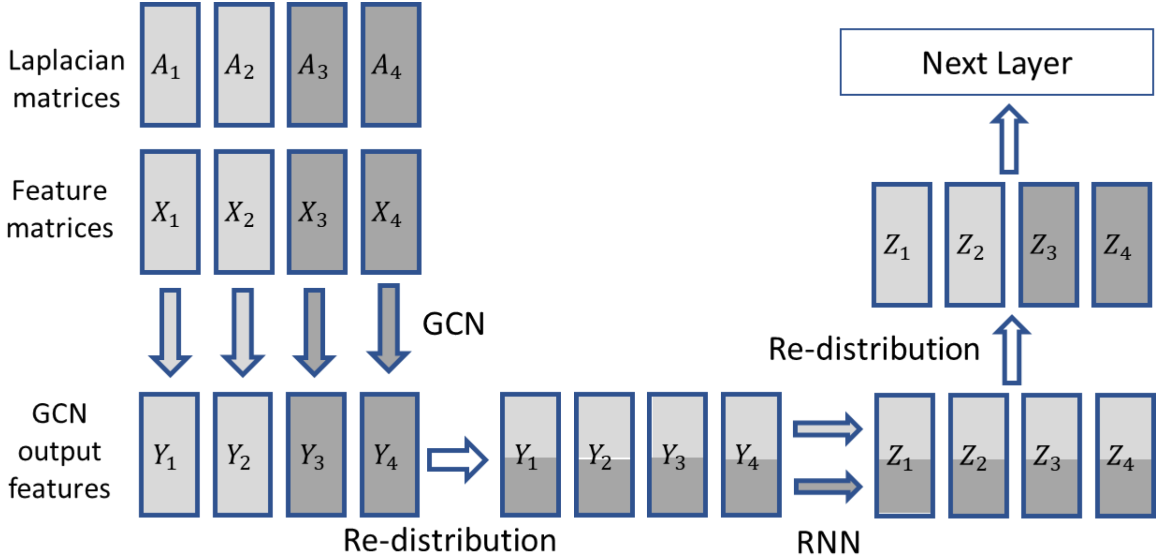
Snapshot-Partitioning: Redistribution

Re-distribution

1. First re-distribution
   - Redistribute output features of GCN via any equi-partitioning of vertices.
2. Complete RNN
3. Second re-distribution
   - Re-distribute output features of RNN to takes us back to snapshot partitioning

Communication volume

- 2 x Feature-size = 2 x O(N x T x F). = O(Vertices x timesteps x feature-size)



Advantages

- Comm volume independent of
  - Edge density
  - Number of processors
- Regular communication pattern
  - Low implementation overhead (on GPU)
- Scales better
- No expensive partitioners

Experimental Evaluation

System Setup

- AiMOS system (https://cci.rpi.edu/aimos).
- We use up to 16 nodes. Intel Xeon 6248.
- Each node has 8 Nvidia V100 GPUs. Total 128 GPUs. .
- NCCL (direct GPU-GPU communication) and PyTorch

Models

- TensorGCN, EvolveGCN, WD-GCN.

Smoothening

- Dataset graphs are highly sparse.
- TensorGCN and EvolveGCN smoothen the graphs that increases their density.

	#vertices N	#timesteps T	#edges m	After smoothening	
				TensorGCN Input edges	EvolveGCN Input edges
epinions	755 K	501	13 M	653 M	1038 M
flickr	2.3 M	134	33 M	963 M	796 M
youtube	3.2 M	203	12 M	851 M	802 M
AML-Sim	1 M	200	124 M	1094 M	1038 M

Experiments

- 3 models x 4 datasets

Representative sample

- TensorGCN, AML-Sim

Gradient Checkpoint: Summary

Baseline

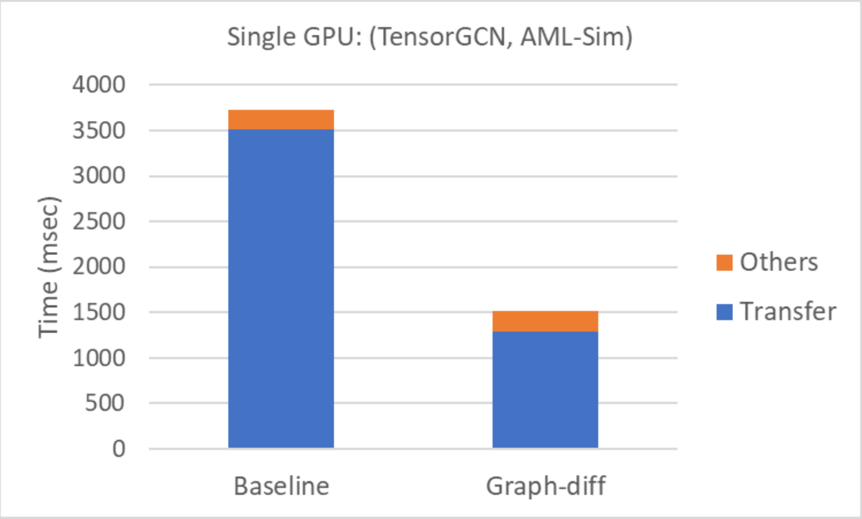
- Stores snapshots and intermediate activations for all snapshots in GPU
- Could not execute on a single node with 8 GPUs due to insufficient GPU memory.

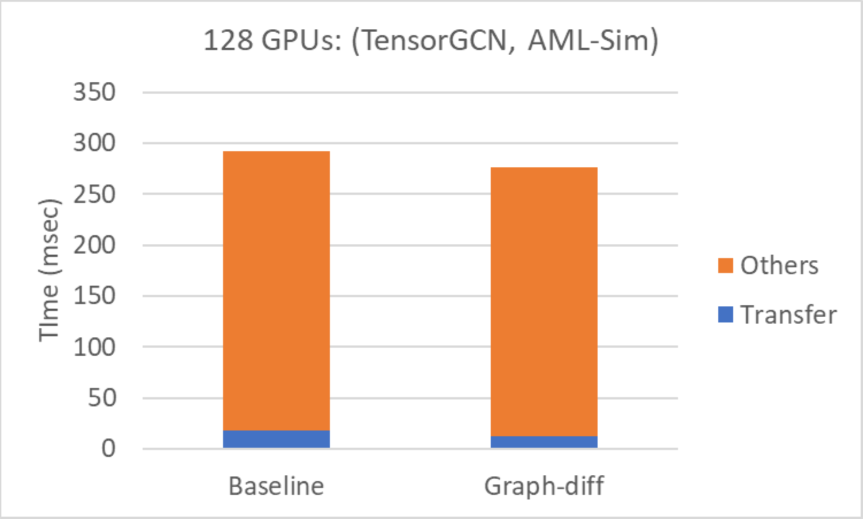
Gradient Checkpoint

- Divides timeline into blocks
- Stores only a single block of snapshots and intermediate activations in GPU.
- Executed on a single GPU.

Graph-difference Based CPU-GPU Transfer

- Single GPU
  - Significant reduction in transfer time.
  - Up to 4x reduction in transfer time and 40% reduction in overall time.
- Large system size
  - Overall execution time and transfer time scales.
  - Communication time becomes bottleneck due to inter-node communication





Vertex Partitioning vs Snapshot Partitioning

- Vertex partitioning
  - Communication volume increases with number of processors
  - Irregular communication pattern → High implementation overheads
  - Poor scaling
- Snapshot partitioning
  - Fixed communication volume for any number of processors
  - Regular communication pattern → Low implementation overheads
  - Better scaling
- TensorGCN, AML-Sim

Communication volume (billion floats)

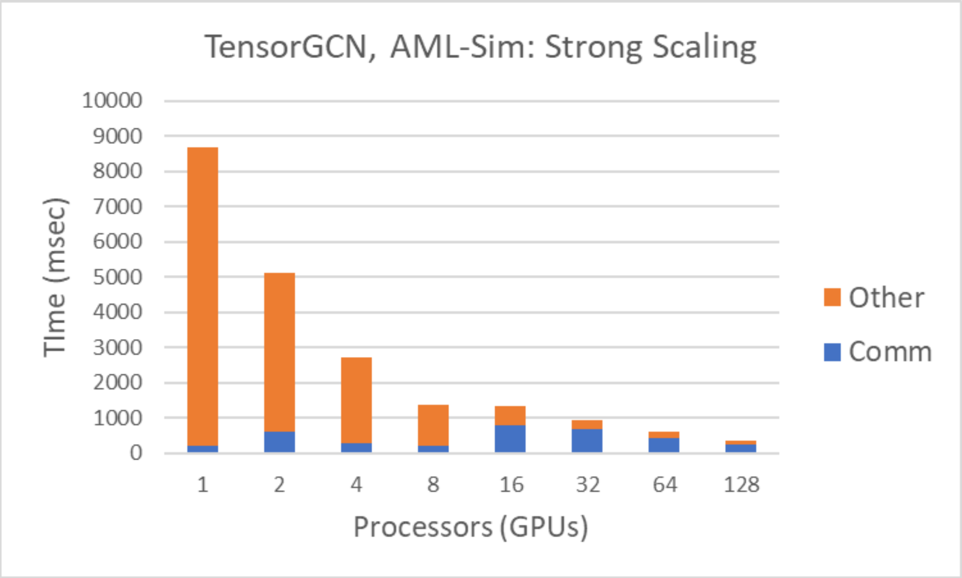
Proc. (GPUs)	Vertex Part.	Snapshot Part.
4	3.2	6.5
16	6.8	6.5
64	9.5	6.5

Execution time per training epoch (msec)

Proc. (GPUs)	Vertex Part.	Snapshot Part.
4	6668	3396
16	5254	1384
64	9164	593

Our Optimized Implementation : Strong Scaling

- Computation + transfer (other) scales very well.
- Communication
  - Up to 8 GPUs: on the same node and internal fast communication
  - 16+ GPUs: Multi-node communication via slow interconnect
- Overall
  - Single GPU = 8600 msec and 128 GPUs = 340 msec. Speedup = 25x



Our Optimized Implementation : Weak Scaling

- AML-Sim simulator can generate graphs of different sizes
- Vary number of processors from 1 to 128
- Proportionately increase graph size
- Throughput = Graph size (edges) per second

GPUs (intra-node)	Throughput
1	1.0
2	3.5
4	10.1
8	22.8

GPUs (intra-node)	Throughput
16	24.7
32	35.9
64	66.2
128	125.7

- Near-perfect weak scaling
- Drop in throughput from 8 (single node) to 16 GPUs (two nodes). Inter-node communication

Future Work

- Limitations of snapshot partitioning
  - Large snapshots that do not fit a GPU
  - Number of snapshots < number of processors
  - Single snapshots need to be split among processors
  - Hybrid scheme combining snapshot and vertex-partitioning
- Computation-communication overlap
  - GCN and RNN across multiple layers
- Continuous Time Dynamic Graphs (CTDG)
  - Represented by insertion/deletion of edges/vertices

Thank you