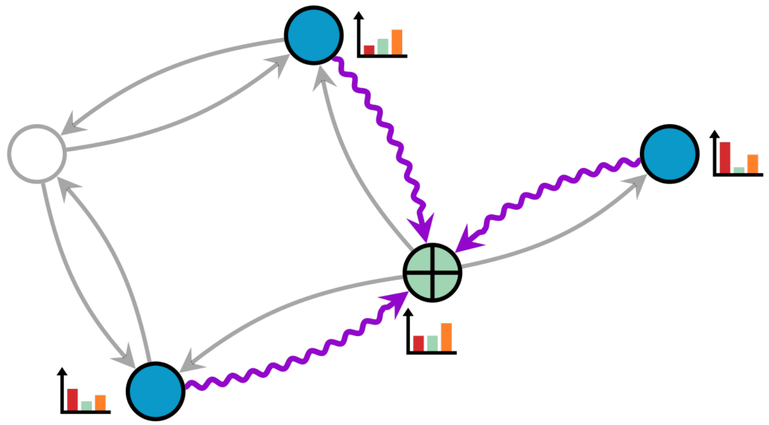
kumo

PyTorch Geometric

Practical Realization *of* Graph Neural Networks

```
conda install pyg -c pyg
```

Graph Neural Networks



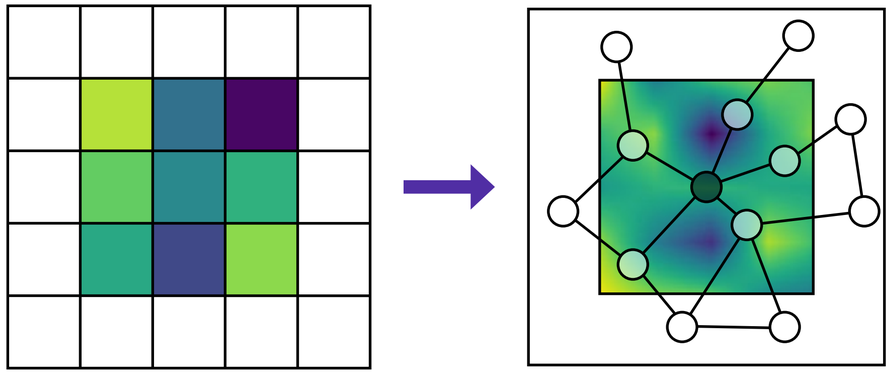
Neural Message Passing Scheme

- ✓ Data-dependent computation graphs
- ✓ Generalization of *any* neural network architecture

A new paradigm of *how* we define neural networks!

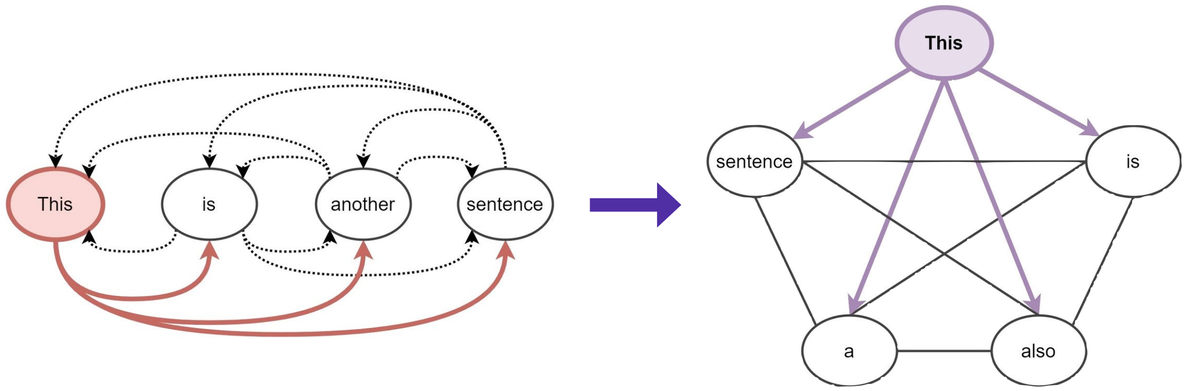
From CNNs *to* GNNs

Message Passing via continuous kernels



From Transformers *to* GNNs

Message Passing within a fully-connected graph



Graph Neural Networks

Implementing Graph Neural Networks is challenging

- Sparsity *and* irregularity of the underlying data
  How can we effectively parallelize irregular data of potentially varying size?
- Heterogeneity of the underlying data
  numerical, categorical, image and text features, potentially over *different* types of data
- *Inherently* dynamic
  it is hard to find scenarios in which graphs will *not* change over time
- Various *different* requests on scalability
  sparse *vs.* dense graphs, many small *vs.* single giant graphs, ...
- Applicability to a set of *diverse* tasks
  node-level *vs.* link-level *vs.* graph-level, clustering, pre-training, self-supervision, ...

PyTorch Geometric

PyG *(PyTorch Geometric)* is the PyTorch library to unify deep learning on graph-structured data

- ✓ simplifies implementing and working with Graph Neural Networks
- ✓ bundles *state-of-the-art* GNN architectures and training procedures
- ✓ achieves *high* GPU throughput on sparse data of varying size
- ✓ suited for *both* academia and industry
  flexible, comprehensive, easy-to-use

Design Principles

PyG is PyTorch-on-the-rocks

✓ **PyG** is framework-specific

allows us to make use of *recently released* features right away

TorchScript for deployment, torch.fx for model transformations,

torch.compile for optimizations

✓ **PyG** keeps design principles *close* to vanilla PyTorch

If you are familiar with PyTorch, you already know most of PyG

✓ **PyG** fits nicely into the PyTorch ecosystem

Scaling up models via PyTorch Lightning

Explaining models via Captum

Design Principles

Graph-based Neural Network Building Blocks

- ✓ Message Passing layers
- ✓ Normalization layers
- ✓ Pooling & Readout layers
- ✓ ...

Graph Transformations & Augmentations

- ✓ Graph diffusion
- ✓ Missing feature value imputation
- ✓ Mesh and Point Cloud support

In-Memory Graph Storage, Datasets & Loaders

- ✓ Support for heterogeneous graphs
- ✓ *200*+ benchmark datasets
- ✓ *10*+ sampling techniques

Examples & Tutorials

- ✓ Learn practically about GNNs
- ✓ Videos, Colabs & Blogs
- ✓ Application-driven Graph ML Tutorials
- ✓ Notebooks examples with visualization

Design Principles

```python
dataset = Reddit(root_dir, transform)

loader = NeighborLoader(dataset, num_neighbors=[25, 10])

class GNN(torch.nn.Module):
    def __init__(self):
        self.conv1 = SAGEConv(F_in, F_hidden)
        self.conv2 = SAGEConv(F_hidden, F_out)

    def forward(x, edge_index):
        x = self.conv1(x, edge_index)
        x = x.relu()
        x = self.conv2(x, edge_index)
        return x

for data in loader:
    data = data.to(device)
    out = model(data.x, data.edge_index)
    loss = criterion(out, data.y)
    loss.backward()
    optimizer.step()
```

PyG is *highly* modular

✓ Access to *200*+ datasets and *50*+ transforms

✓ Access to a *variety* of mini-batch loaders
Node-wise sampling, subgraph-wise sampling, graph-wise batching

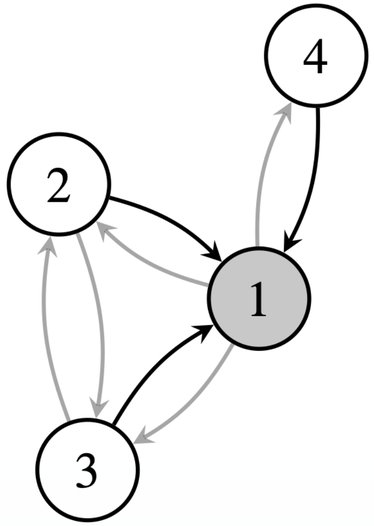
✓ Access to *80*+ GNN layers, normalizations and readouts as neural network building blocks
SAGEConv, *GCNConv*, *GATConv*, *GINConv*, *PNAConv*, ...

and

20+ pre-defined models
GraphSAGE, *GCN*, *GAT*, *GIN*, *PNA*, *SchNet*, *DimeNet*, ...

✓ Access to regular PyTorch loss functions and training routines
Classification, Regression, Self-Supervision, ...
Node-level, Link-level, Graph-level

Implementing Graph Neural Networks



Given a *sparse* graph $\mathcal{G} = (\mathbf{H}^{(0)}, (\mathbf{I}, \mathbf{E}))$ with

- input node features $\mathbf{H}^{(0)} \in \mathbb{R}^{|\mathcal{V}| \times C}$
- edge indices $\mathbf{I} \in \{1, ..., |\mathcal{V}|\}^{2 \times |\mathcal{E}|}$
- *optional* edge features $\mathbf{E} \in \mathbb{R}^{|\mathcal{E}| \times D}$

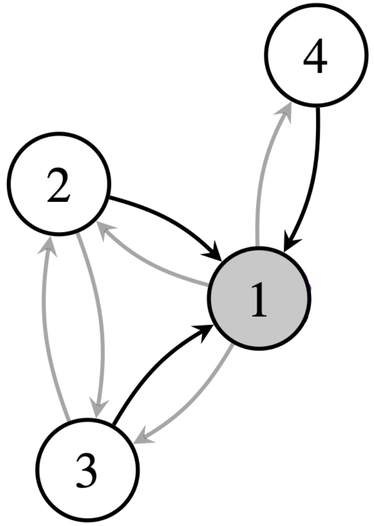
Message Passing Scheme

permutation-invariant aggregation operator, e.g., *sum*, *mean* or *max*

$$\mathbf{h}_i^{(\ell+1)} = \textsc{Update}_\theta \left(\mathbf{h}_i^{(\ell)}, \bigoplus_{j \in \mathcal{N}(i)} \textsc{Message}_\theta \left(\mathbf{h}_j^{(\ell)}, \mathbf{h}_i^{(\ell)}, \mathbf{e}_{j,i} \right) \right)$$

Implementing Graph Neural Networks

Flexible implementation via a general MessagePassing interface



$$\mathbf{h}_i^{(\ell+1)} = \textsc{Update}_\theta \left(\mathbf{h}_i^{(\ell)}, \bigoplus_{j \in \mathcal{N}(i)} \textsc{Message}_\theta \left(\mathbf{h}_j^{(\ell)}, \mathbf{h}_i^{(\ell)}, \mathbf{e}_{j,i} \right) \right)$$

Graph Creation

PyG expects a graph in either COO or CSR format to model complex (heterogeneous graphs)

Both nodes and edges can hold any set of curated features

```
data = HeteroData()
data['user'].x = ...
data['product'].x = ...
data['user', 'product'].edge_index = ...
data['user', 'product'].edge_attr = ...
data['user', 'product'].time = ...
```

Benchmarking graphs: Training labels and dataset splits are pre-defined
Real-world graphs in applications: Build/Bring your own training labels and dataset splits

Best practices for graph design suitable for message passing

Best practices for feature selection

Best practices for feature encoding suitable for neural networks

Task Formulation

PyG supports any graph-related machine learning task, but curation of training labels is a user task

```
# Add user-level node labels:
data['user'].y = ...

transform = RandomNodeSplit(num_val=0.1, num_test=0.1)
data = transform(data)
```

Benchmarking graphs: Training labels and dataset splits are pre-defined
Real-world graphs in applications: Build/Bring your own training labels and dataset splits

Best practices for curating training labels to map to ML task

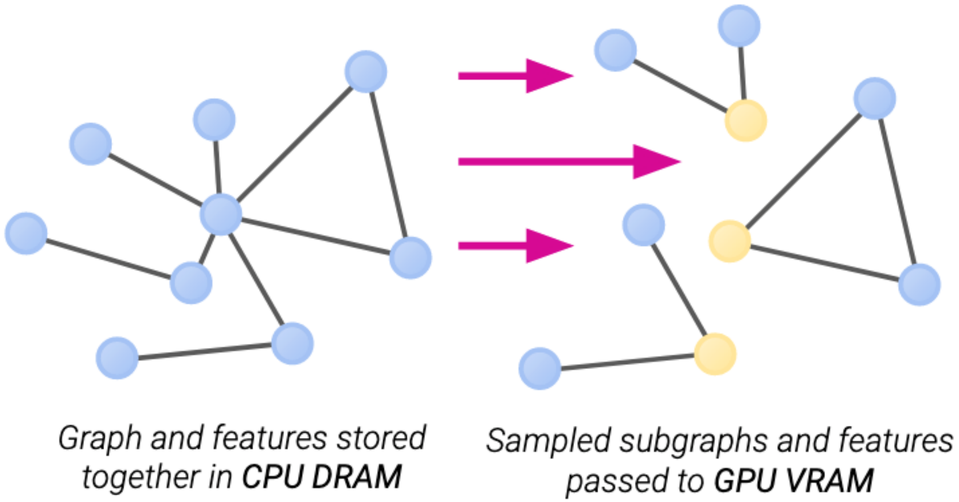
Best practices for avoiding data leakage in temporal scenarios

Best practices for dataset splitting to match real-world use-cases

Data Loading

PyG scales to data larger than GPU VRAM via sampled subgraphs, and integrates over dozens of different sampling techniques

However, large-scale real world graphs may not even fit on a single machine



*Graph and features stored together in **CPU DRAM***

*Sampled subgraphs and features passed to **GPU VRAM***

Moving data structures out-of-core for independent scale-out

Fast lookup of features

Efficient distributed graph representations and sampling algorithms

Heterogeneous compute architectures and optimizing E2E throughput

Graph Machine Learning Toolkit

PyG can handle standard graph learning tasks with ease ...

Node-level Predictions

Predict the class of a node

$$\phi(\mathcal{G}, v) \in [0, 1]^C$$

Link-level Predictions

Predict the class of a link

$$\phi(\mathcal{G}, v, w) \in [0, 1]^C$$

Graph-level Predictions

Predict the class of a graph

$$\phi(\mathcal{G}) \in [0, 1]^C$$

... but is not limited to those:

Unsupervised Learning Self-Supervised Learning
Few/Zero-Shot Learning Pre-Training Explainability ...

PyG provides over 80 examples with access to over 200 benchmark datasets
to get familiar with the latest trends in graph machine learning

Training Process

PyG supports full customization from model architecture to training routine pipeline

```
loader = NeighborLoader(dataset, num_neighbors= ... )
model = PNA(dataset.num_features, dataset.num_classes)

for batch in loader:
    logits = model(batch.x, batch.edge_index)
    loss = F.binary_cross_entropy(logits, batch.y)
    loss.backward()
    optimizer.step()
```

BUT: Best model architecture is both data- and task-dependent

Which GNN is best suited for my ML task?

How to capture long-range information?

How many neighbors and hops to sample?

How to deal with class imbalances?

Pre-training and self-supervision?

How to ensure model generalizes over time?

PyTorch Geometric

New Features

Timeline

Open-Sourced
Nov'17

Paper Release
March'19

FAST GRAPH REPRESENTATION LEARNING WITH PYTORCH GEOMETRIC

OGB Collaboration
Feb'20

PyG *2.0* Release
Sep'21

- ✓ Stanford Partnership
- ✓ Heterogeneous GNNs

Partnership Acquisition
'21

- ✓ Kumo.AI Partnership
- ✓ NVIDIA Partnership
- ✓ Intel Partnership

PyG *2.1* Release
Aug'22

- ✓ *Principled* Aggregations
- ✓ Scalable Link Prediction
- ✓ Temporal Samplers
- ✓ ...

PyG *2.2* Release
Nov'22

- ✓ Accelerations
- ✓ Scalability
- ✓ ...

PyG *2.3* Release
Mar'23

- ✓ Explainability
- ✓ Compilation
- ✓ GraphCore Partnership
- ✓ ...

Highlights

Major Architecture Change

A *new* GNN engine: *pyg-lib*

Joint effort across *many* different partners

New *(and upcoming)* Features

Improved GNN design *via* principled aggregations

Improved efficiency *on* heterogeneous graphs

Out-of-core data interfaces

Explainability *and* Compilation

Highlights

Major Architecture Change

A *new* GNN engine! *pyg-lib*

Joint effort across *many* different partners

New *(and upcoming)* Features

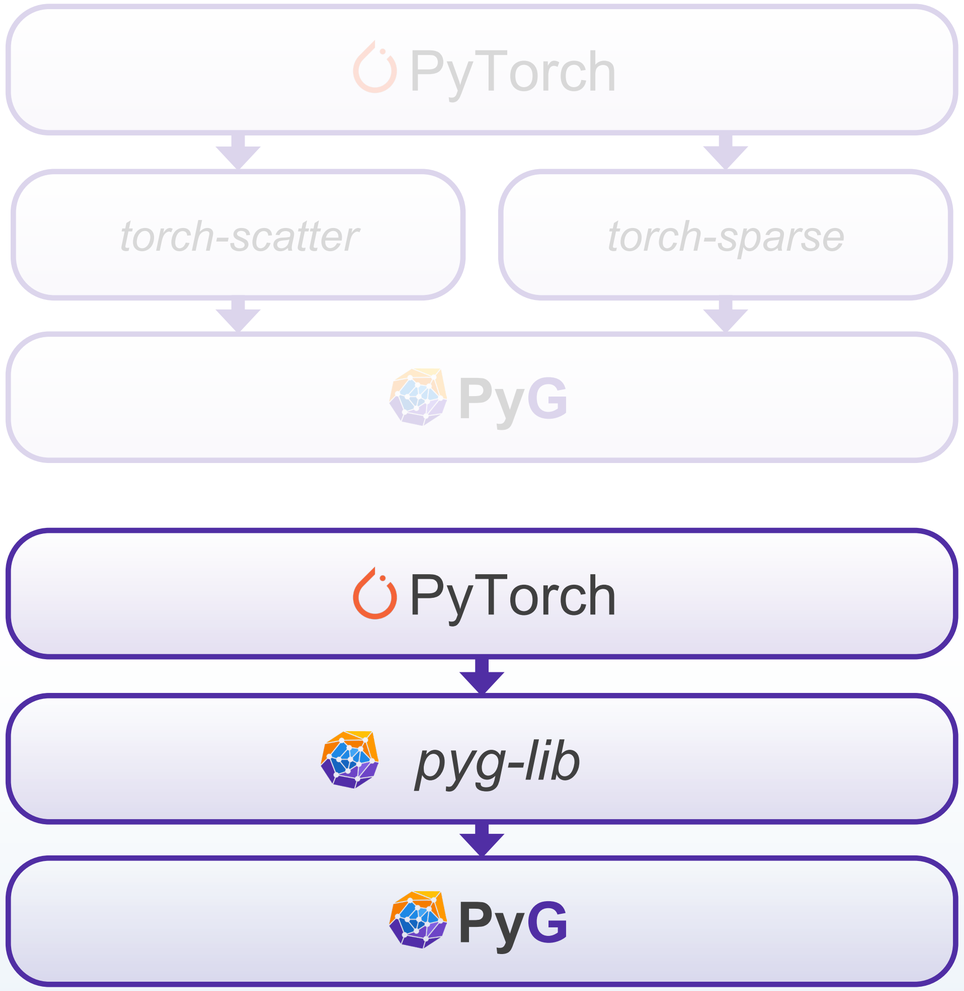
Improved GNN design *via* principled aggregations

Improved efficiency *on* heterogeneous graphs

Out-of-core data interfaces

Explainability *and* Compilation

Accelerating PyTorch Geometric



pyg-lib: A *unified* GNN engine for optimized low-level graph routines

/pyg-team/pyg-lib

✓ Joint effort of Kumo, NVIDIA & Intel

✓ *Accelerating* graph sampling routines

✓ *Accelerating* heterogeneous GNNs

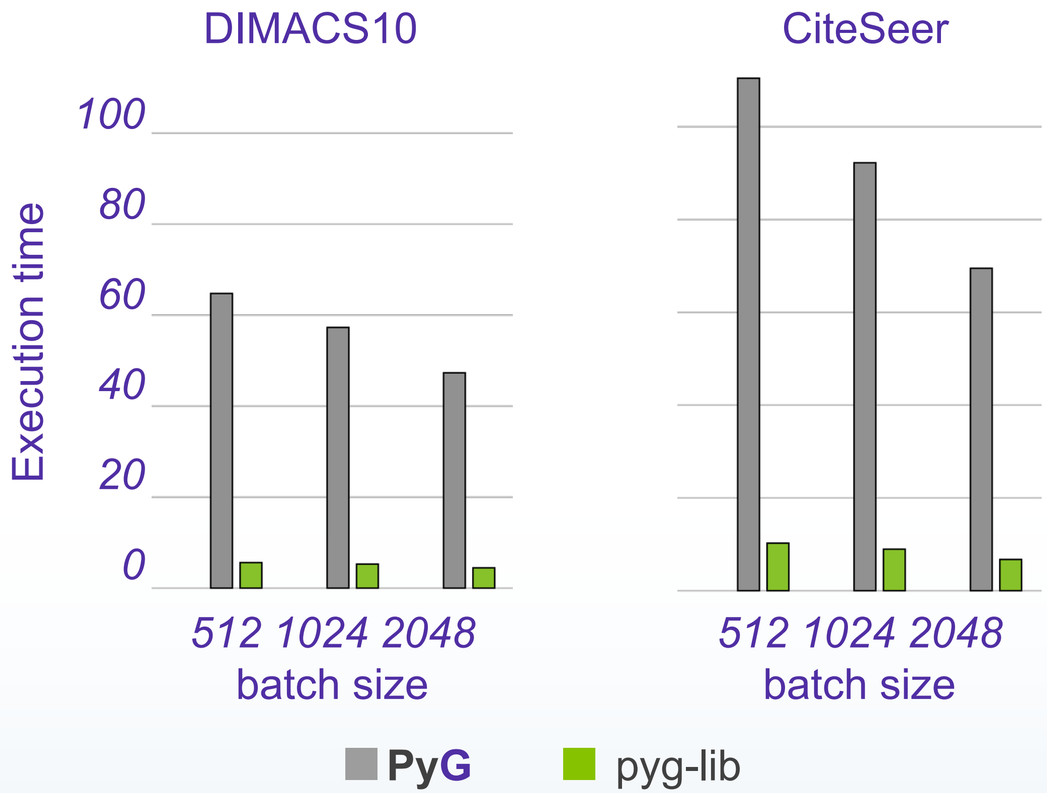
✓ *Accelerating* sparse aggregations

✓ Speed-ups with *no* line of code change

Accelerating PyTorch Geometric

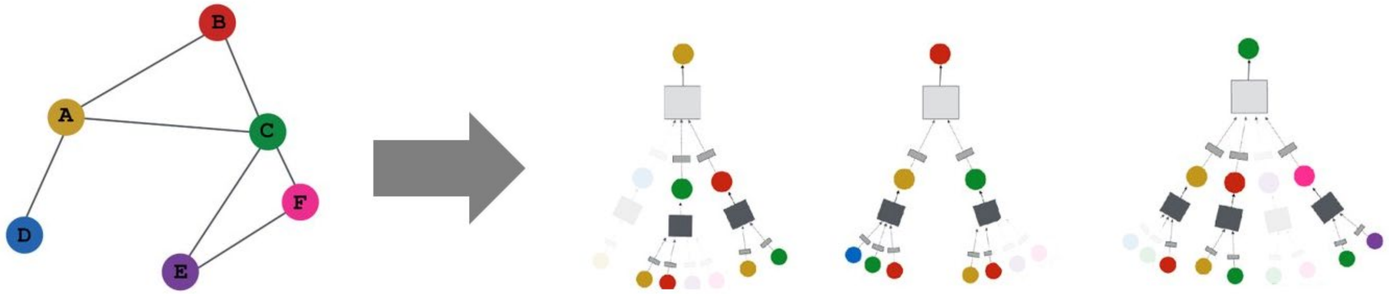
pyg-lib leverages a *variety* of techniques to further accelerate neighbor sampling routines

- ✓ Pre-allocation of random numbers
- ✓ Vector-based mapping of nodes for *smaller* node types
- ✓ *Faster* hashmap implementation
- ✓ *10x* to *15x* speed-ups



Accelerating PyTorch Geometric

In general, all sampling routines in **PyG** return a single subgraph, which makes playing around with different sampling strategies *very* convenient




```python
# Default:
for batch in loader:
    model(batch.x, batch.edge_index)

# Optimized:
def forward( ... ):
    ...
    x, edge_index = trim_to_layer(layer=i, ... )
    x = conv(x, edge_index)
    ...

for batch in loader:
    model(
        batch.x,
        batch.edge_index,
        batch.num_sampled_nodes_per_hop,
        batch.num_sampled_edges_per_hop,
    )
```

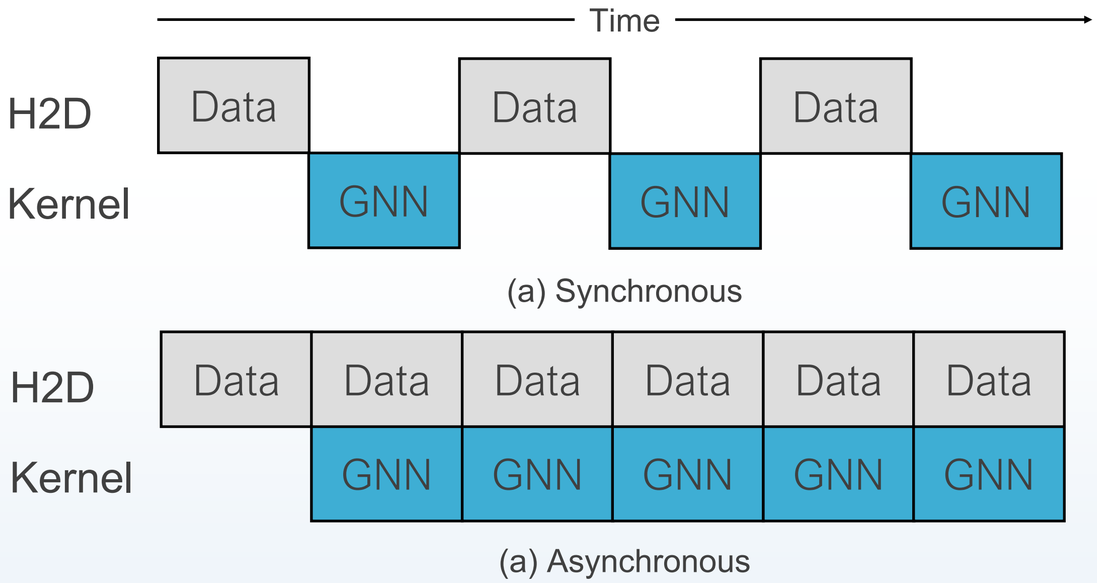
However, this may also lead to *a lot of* wasted computation of node embeddings we don't use as readouts

- ✓ *pyg-lib* now returns the number of sampled nodes and edges per hop
- ✓ We can use this information downstream to "trim" to the required graph-structure

Accelerating PyTorch Geometric

Furthermore, **PyG** introduces the concept of asynchronous device transfers

PrefetchLoader prefetches mini-batches, moves them to pinned memory and performs asynchronous device transfers




```python
loader = NeighborLoader(data, ...)

# Before:
for batch in loader:
    batch = batch.to(device)

# After:
loader = PrefetchLoader(loader, device)
for batch in loader:
    assert batch.is_cuda
```

Highlights

Major Architecture Change

A *new* GNN engine: *pyg-lib*

Joint effort across *many* different partners

New *(and upcoming)* Features

Improved GNN design *via* principled aggregations

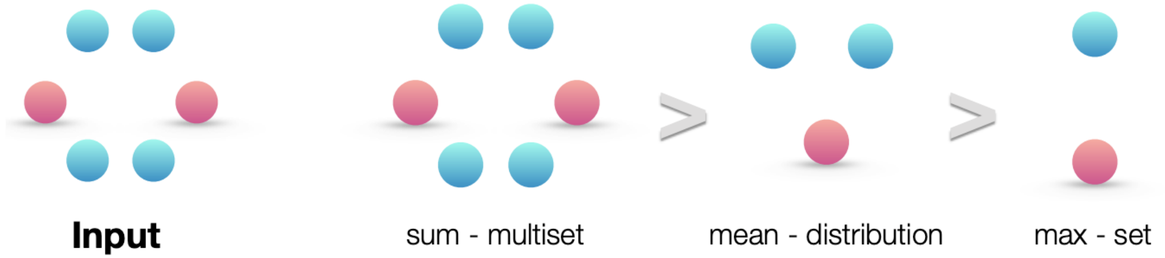
Improved efficiency *on* heterogeneous graphs

Out-of-core data interfaces

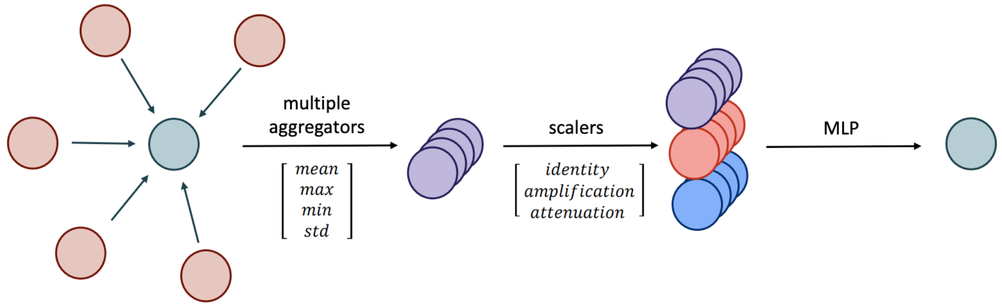
Explainability *and* Compilation

Principled Aggregations

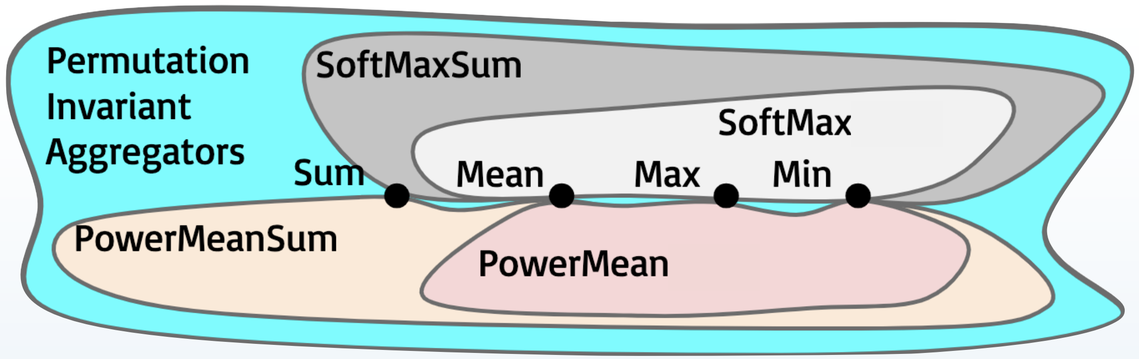
Choice of neighborhood aggregation is a *central* topic in Graph ML research



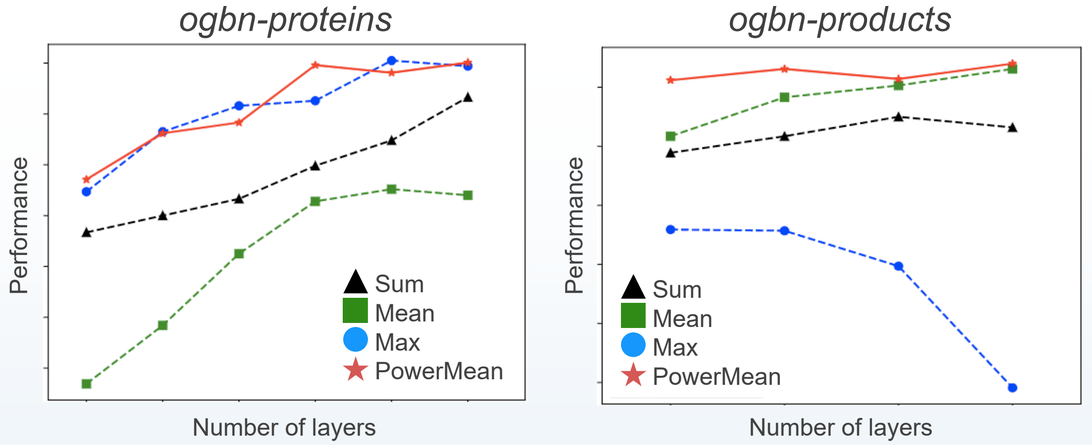
Xu *et al.*: How Powerful Are Graph Neural Networks?



Corso *et al.*: Principal Neighborhood Aggregation for Graph Nets



Li *et al.*: Deeper-GCN: All You Need to Train Deeper GCNs



Principled Aggregations

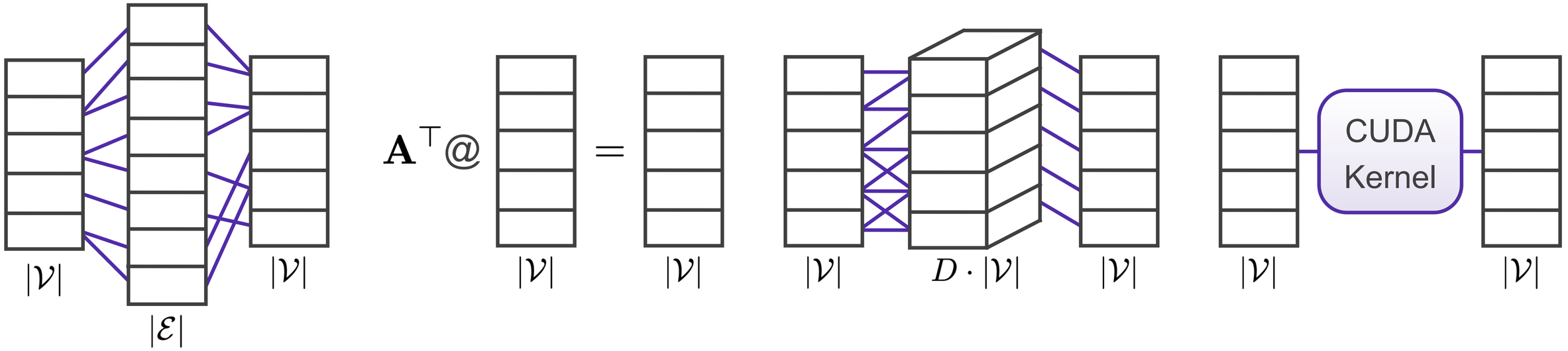
```python
# Simple aggregations:
mean_aggr = aggr.MeanAggregation()
max_aggr = aggr.MaxAggregation()

# Advanced aggregations:
median_aggr = aggr.MedianAggregation()

# Learnable aggregations:
softmax_aggr = aggr.SoftmaxAggregation(learn=True)
powermean_aggr = aggr.PowerMeanAggregation(learn=True)

# Exotic aggregations:
lstm_aggr = aggr.LSTMAggregation()
sort_aggr = aggr.SortAggregation(k=4)

# Use within message passing:
conv = MyConv(aggr=[median_agr, lstm_aggr])

# Use for global pooling:
h_graph = sort_aggr(h_node, batch)
```

PyG makes the concept of aggregations a *first-class* principle

- ✓ Access to *all* kinds of simple, advanced, learnable and exotic aggregations
  Median, Softmax, Attention, LSTM, ...
- ✓ Fully-customize and combine aggregations within *MessagePassing or* for global pooling
- ✓ Aggregations will pick up the *best* format to accelerate computation
  scatter reductions, degree bucketing, ...
- ✓ Further optimization *via* fusion
  minimize I/O from global memory

Principled Aggregations

The *different* flavors of implementing aggregations



Gather & Scatter

- very flexible 😋
- fast for sparse graphs 😋
- memory-inefficient 😭

PyG >= 0.*1*

Sparse MatMul

- less flexible 😭
- very fast 😋
- memory-efficient 😋

PyG >= 1.*6*

Degree Bucketing

- any *dense* aggregation 😋
- memory-inefficient 😭
- padding/seq. iteration 😭

PyG >= 2.*1*

Individual Kernel

- not flexible at all 😭
- memory-efficient 😋
- very fast 😋

PyG >= 2.*2*

Highlights

Major Architecture Change

A *new* GNN engine: *pyg-lib*

Joint effort across *many* different partners

New *(and upcoming)* Features

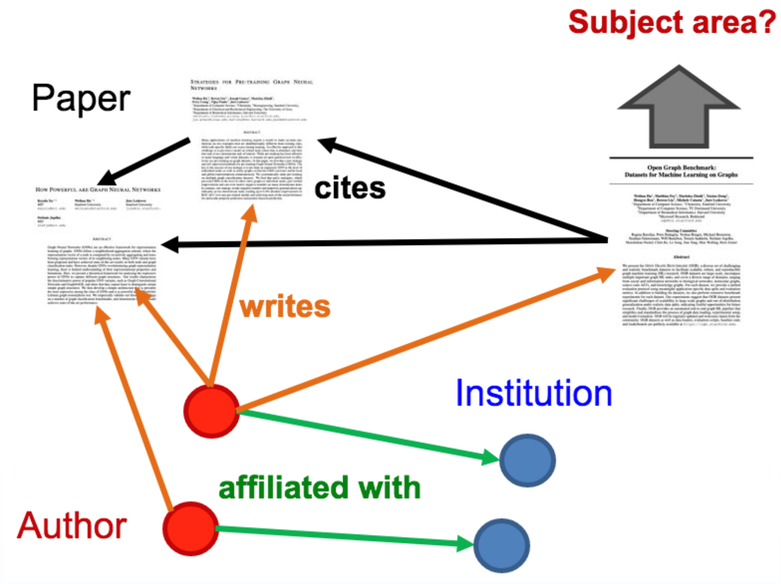
Improved GNN design *via* principled aggregations

Improved efficiency *on* heterogeneous graphs

Out-of-core data interfaces

Explainability *and* Compilation

Heterogeneous Graph Support



(Nearly) all real-world graphs are heterogeneous!

Heterogeneous Graph Support

Heterogeneous Graph Neural Networks

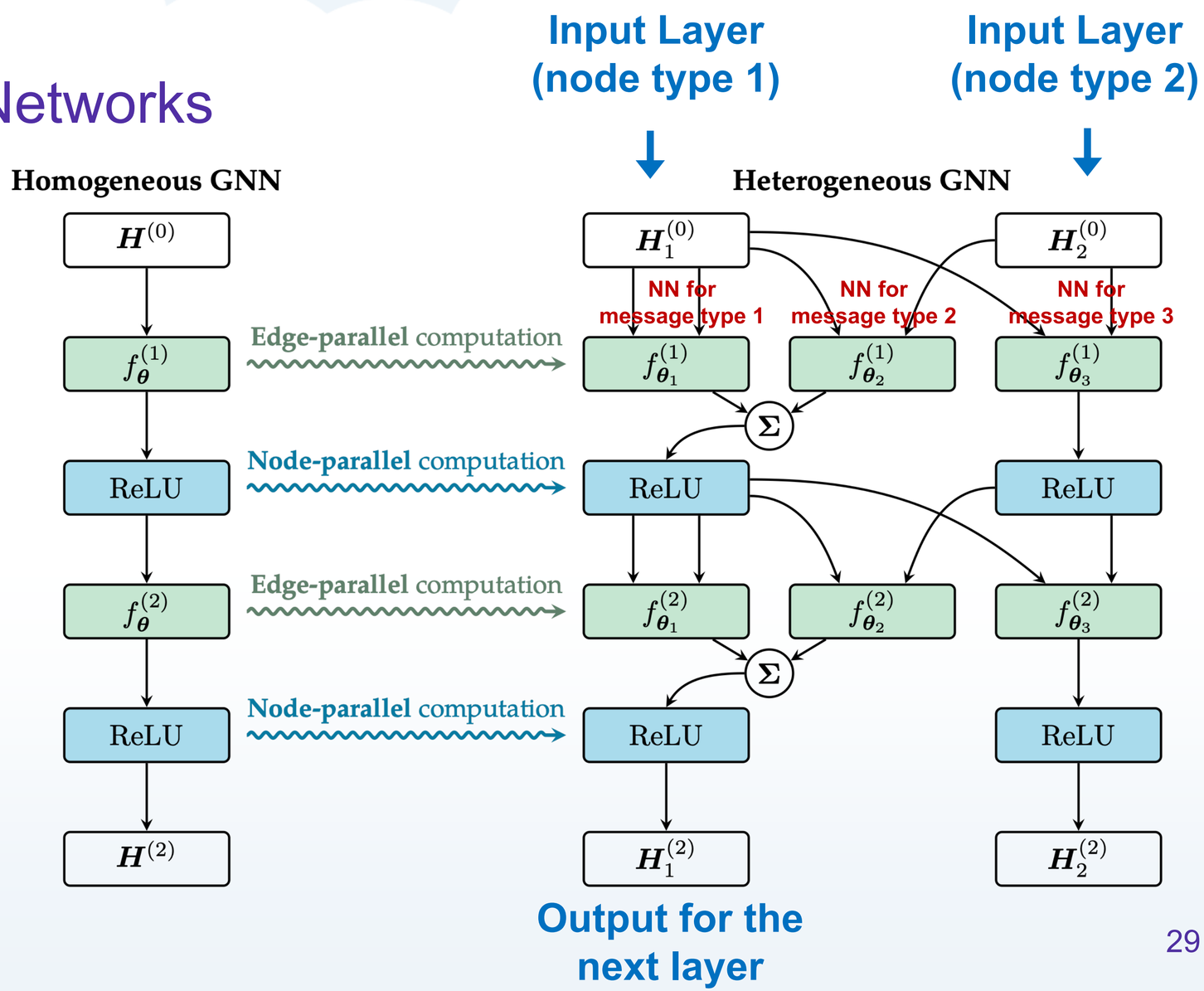
A homogeneous GNN can be converted to a heterogeneous one by learning *distinct* parameters for each individual edge type

Edge type dependent parameters

$$\mathbf{h}_i^{(\ell+1)} = \sum_{r \in \mathcal{R}} f_{\theta_r}^{(\ell+1)}\left(\mathbf{h}_i^{(\ell)}, \{\mathbf{h}_j^{(\ell)} : j \in \mathcal{N}^{(r)}(i)\}\right)$$

The number of relations

Relational-wise neighborhood



Heterogeneous Graph Support

PyG can *automatically* convert homogeneous GNNs to heterogeneous ones

to_hetero(model (node_types, edge_types)):
to_hetero_with_bases(model (node_types, edge_types)):

Implemented *via* torch.fx

1. Duplicates message passing modules for *each* edge type
2. Transforms the underlying computation graph so that messages are exchanged along *different* edge types
3. Uses lazy initialization (-1) to handle *different* input feature dimensionalities

```python
from torch_geometric.nn import GAT, to_hetero

model = GAT(in_channels=-1, hidden_channels=64,
            out_channels=72, num_layers=2)

model = to_hetero(model, (node_types, edge_types))

out = model(data.x_dict, data.edge_index_dict)
```

Heterogeneous GNN Implementations

to_hetero() is a powerful tool but lacks parallelism *across* node/edge types

Naive Implementation

$$\mathbf{H}^{(\ell+1)} = \sum_{r=1}^{\mathcal{R}} \mathbf{A}_r \mathbf{H}^{(\ell)} \mathbf{W}_r^{(\ell+1)}$$

```
out = 0
for r in range(num_edge_types):
    out += adj[r] @ h @ w[r]
return out
```

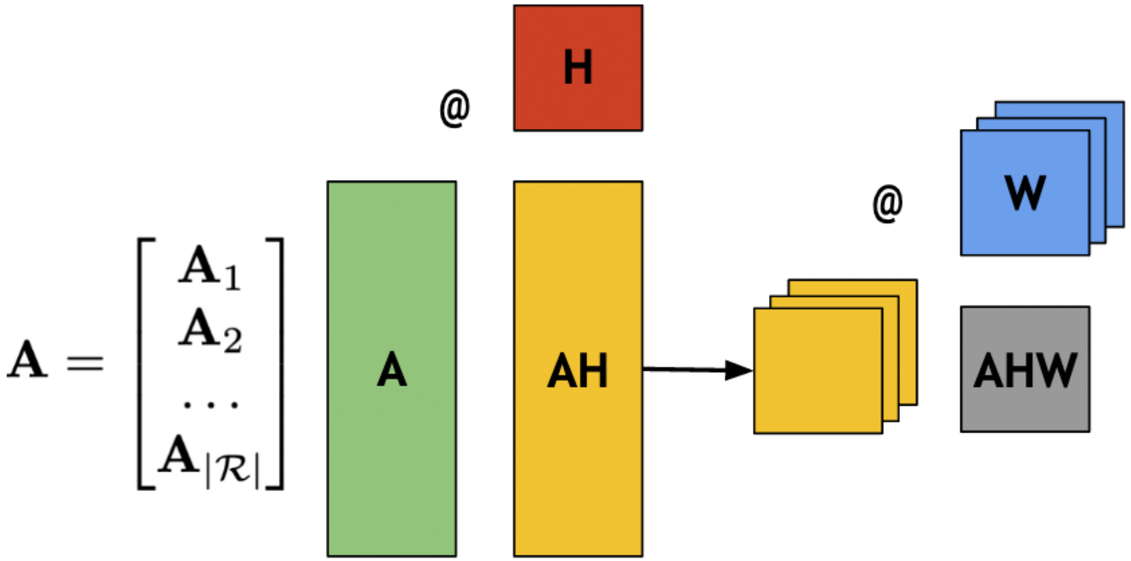
- Flexible: *Any* homogeneous GNN operator can be utilized
- Inefficient: Lack of parallelism *across* edge and node types

Heterogeneous GNN Implementations

Vertically-Stacked Implementation

Leverage full parallelism by *stacking* adjacency matrices vertically

Thanapalasingam *et al.*: Relational Graph Convolutional Networks: A Closer Look (2021)



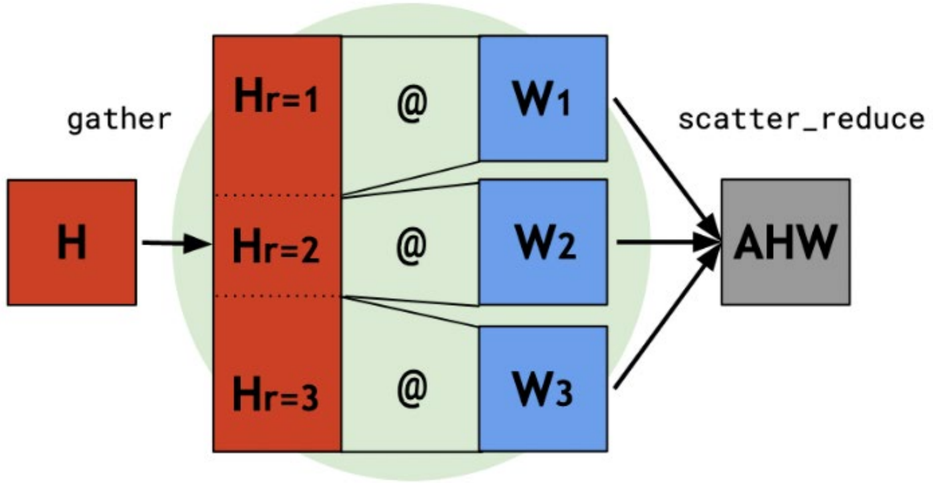
Inefficient in case ...

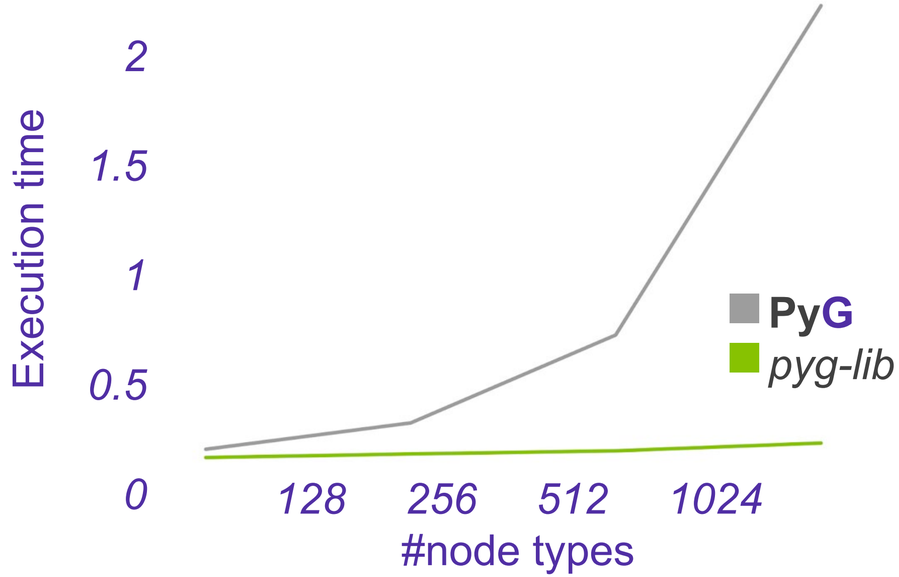
- *of* large number of edge types / sparse edge types
- *of* multiple node types
  All features will be replicated for each edge type

Accelerating Heterogeneous GNNs

CUTLASS-based Implementation

pyg-lib supports *parallel* type-dependent transformations via NVIDIA CUTLASS integration





✓ Flexible to implement *most* heterogeneous GNNs with

✓ Efficient, even on *sparse* edge types or on a *large number* of node types

Highlights

Major Architecture Change

A *new* GNN engine: *pyg-lib*

Joint effort across *many* different partners

New *(and upcoming)* Features

Improved GNN design *via* principled aggregations

Improved efficiency *on* heterogeneous graphs

Out-of-core data interfaces

Explainability *and* Compilation

Out-of-core Data Interfaces

There exists multiple ways to scale **PyG** beyond *single-node in-memory* datasets

1. Pre-process subgraphs on disk and load them *on-the-fly*,
   e.g., via dedicated Spark routines

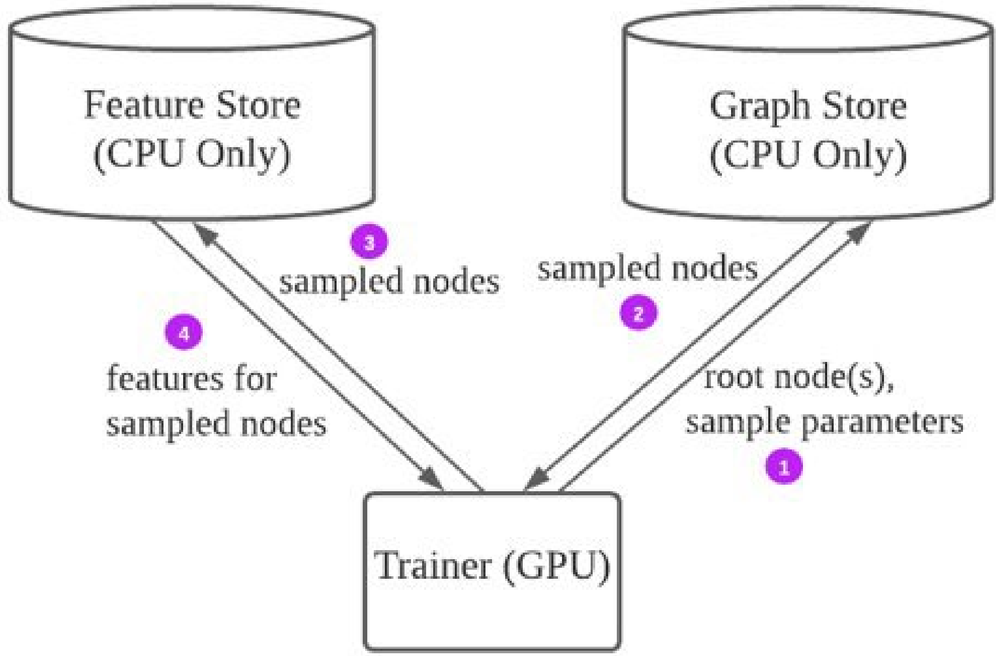
- ✓ Naturally supported via **PyG**'s mini-batch *DataLoader*
- ✓ No sampling/feature fetching overhead during GNN training
- ✓ Heavy pre-processing and storage requirements
- ✓ Experimentation with different sampling strategies and parameters gets increasingly harder

```python
class Dataset:
    def __getitem__(self, idx) → Data:
        # 1. Load subgraph from disk:
        data = load()
        # 2. Convert to PyG format:
        data = Data()
        return data

dataset = Dataset()
loader = DataLoader(dataset, batch_size=128)
```

Out-of-core Data Interfaces

There exists multiple ways to scale **PyG** beyond *single-node in-memory* datasets

2. With **PyG**, we support *any* backend by providing *FeatureStore* *and GraphStore* abstractions



✓ Disentangles feature fetching *from* graph sampling routines

✓ Allows *for* distributed server/client architectures

✓ Allows *for* out-of-memory backends, *e.g.*, via connecting to graph databases

Out-of-core Data Interfaces

There exists multiple ways to scale **PyG** beyond *single-node in-memory* datasets

2. With **PyG**, we support *any* backend by providing *FeatureStore* *and GraphStore* abstractions

These interfaces can enable an elastic, heterogeneous compute architecture

✓ Read-optimized on-disk feature store via *RocksDB*

✓ Low-latency in-memory graph store

We are working with *multiple* graph database vendors to enable store implementations, *e.g.*, KùzuDB

```python
class MyFeatureStore(FeatureStore):
    def get_tensor(self, attr):
        pass  # Implement feature access


class MyGraphStore(GraphStore):
    def sample_from_nodes(self, index):
        pass  # Implement node-wise sampling

    def sample_from_edges(self, index):
        pass  # Implement edge-wise sampling
```

Out-of-core Data Interfaces

There exists multiple ways to scale **PyG** beyond *single-node in-memory* datasets

3. With the upcoming **PyG** *2.4* release, we plan to support

   torch_geometric.distributed,

   a new sub-package to enable multi-node GNN training by partitioning the graph and its features across multiple nodes
   (effort driven by Intel)

4. Rely on third-party libraries
   - *alibaba/graphlearn-for-pytorch*
   - *quiver-team/torch-quiver*
   - ...

Highlights

Major Architecture Change

A *new* GNN engine: *pyg-lib*

Joint effort across *many* different partners

New *(and upcoming)* Features

Improved GNN design *via* principled aggregations

Improved efficiency *on* heterogeneous graphs

Out-of-core data interfaces

Explainability *and* Compilation

Explainability

PyG *2.3* introduced a new *unified* explainability interface to explain *any* GNN out-of-the-box across *various* explainer algorithms

- ✓ Works on both homogeneous and heterogeneous graphs
- ✓ Support for general explainers such as integrated gradients, saliency, shapley, *etc*
- ✓ Support for dedicated GNN explainers, *e.g.*, *GNNExplainer*, *PGExplainer*, *etc*
- ✓ Support for a diverse range of tasks
- ✓ Support for visualizations and metric computation

```python
explainer = Explainer(
    model=model,
    algorithm=GNNExplainer(epochs=200),
    node_mask_type='attributes',
    edge_mask_type='object',
    model_config=dict(
        mode='multiclass_classification',
        task_level='node',
        return_type='probs',
    ),
)

explanation = explainer(x, edge_index)
```

Compilation

PyG *2.3* takes full advantage of PyTorch *2.0* compilation, which makes *your* GNN run faster by JIT-compiling it into optimized kernels

Model	Mode	Forward	Backward	Total	Speedup
`GCN`	Eager	2.6396s	2.1697s	4.8093s	
`GCN`	**Compiled**	**1.1082s**	**0.5896s**	**1.6978s**	**2.83x**
`GraphSAGE`	Eager	1.6023s	1.6428s	3.2451s	
`GraphSAGE`	**Compiled**	**0.7033s**	**0.7465s**	**1.4498s**	**2.24x**
`GIN`	Eager	1.6701s	1.6990s	3.3690s	
`GIN`	**Compiled**	**0.7320s**	**0.7407s**	**1.4727s**	**2.29x**

```
model = GraphSAGE()
model = torch.compile(model)

out = model(data.x, data.edge_index)
```

More improvements to come in PyTorch *2.1*