Accelerating GNNs with PyTorch Geometric and GPUs

Rishi Puri, Deep Learning Software Engineer for NVIDIA
Matthias Fey, Deep Learning Software Engineer for kumo.AI
Outline

- NVIDIA addresses the challenges of end-to-end GNN workflows
- Example workflows
- `pyg-lib` accelerates PyG workflows
Challenges in the GNN Workflows

• GNN model training requires advanced system knowledge
  ○ Loading large datasets (100GB+)
  ○ Managing memory for graph-structured data
  ○ Sparsity-aware workflows
  ○ Distributed training

• Script-based GNN workflows are not flexible enough
  ○ Swapping datasets or models even for similar tasks requires a lot more effort than for
    Computer Vision or Natural Language Processing
  ○ Using torchvision or torchtext enables easy data and model swapping, which does not
    currently exist for GNNs

• Preprocessing of large datasets (100GB+) is very slow on CPU

• Additional effort required to deploy
NVIDIA’s Turnkey, E2E GPU Accelerated GNN Pipeline

- NVIDIA provides a flexible, easy-to-use, general API that allows for building end-to-end GNN workflows, addressing the previously mentioned challenges:
  - Automatic, accelerated system management reduces the need for system knowledge
  - Flexibility allows for switching between tasks, models, and data types with a few lines of code change.
  - RAPIDs GPU optimized preprocessing: go from hours to minutes
  - Push button deployment
- At GTC 2022 in Spring it was introduced and explained in detail:
- It fully supports 🌐PyG and DGL, the two main GNN frameworks
- In this talk we will focus specifically on the 🌐PyG side
E2E GPU Accelerated GNN Stack

Core APIs

- Synthetic Graph Generation
- Pre-process
- Core File System
- Data Loader
- GNN Models
- Distributed Training
- Deployment

Core functionality

Industry specific custom workflows

Core functionality includes:
- XGBoost
- PyTorch Geometric
- DGL
- Triton Inference Server
- PyTorch
- NVTabular
- HugeCTR
- CUDA, cuDF, cuGraph, cuSparse, cuDNN
- Compute (A100, V100, H100**)

**H100 coming soon
E2E Accelerated PyG Workflow

Preprocessing
- User-defined transform functions
- Transform functions for Public datasets
- Data Converters from any data source to our optimized file system

Optimized File System
- Parquet files for node and edges

Data Loader
- Node-level tasks
- Edge-level tasks
- Graph-level tasks

PyG Based Model
- R-GCN
- R-GAT
- SE3T
- ...

Distributed Trainer

Deployment of the Whole Workflow

Parquet/CSV Files

Any Database

PyG Binary File
Microsoft Academic Graph (MAG) is a heterogenous graph of academia. The goal is to infer missing information in the graph (the venue of papers).

Tabformer is a graph of credit card transactions. The goal is to learn to detect fraudulent transactions.

Example Datasets:

- **Field of Study**: 59,965 nodes
  - has topic: 7,505,078 edges
- **Paper**: 736,389 nodes
  - writes: 7,145,660 edges
  - cites: 5,416,271 edges
- **Author**: 1,134,649 nodes
- **Institution**: 8740 nodes
- **Card**: 4782 nodes
  - transaction: 24,198,836 edges
- **Merchant**: 93298 nodes
- **Affiliation**: 1,043,998 edges
Generalized GNN Module

- The API provides a generalized GNN module
- Unified customizable interface allows for easy creation of both complex and simple GNNs
MAG Workflow

```python
import torchrom gp.data_loaders import DataSpec, MetaLoader, NodeDataObjectrom gp.metrics import MSErom gp.models.pyg import HeteroModule
from gp.preprocessing.trans_dataset import OGBN_MAG
from gp.workflow import Workflow
from gp.workflow.trainers import Trainer

# Pre Process
prep = OGBN_MAG('/path/src', '/path/dst')
prep.transform()

# Load Data
meta = MetaLoader('/path/dst')
train_spec = DataSpec(
    shuffle=True,
    batch_size=1024,
    fanouts=[50, 50],
    metadata=meta,
)
test_spec = DataSpec(
    shuffle=False,
    batch_size=1024,
    fanouts=[50, 50],
    metadata=meta,
)
data_object = NodeDataObject(
    train_dataloader=train_spec,
    valid_dataloader=test_spec,
    test_dataloader=test_spec,
    data_path='/path/dst',
    backend='PyG',
)

# Initialize GNN Module
model = HeteroModule(
    metadata=data_object.metadata,
    dim_hidden=400,
    dim_out=349,
    n_layers=1
)

# Set Up Training
optimizer = torch.optim.Adam(
    params=model.parameters(),
    lr=0.1,
    betas=(0.9, 0.999),
    eps=1.0e-08,
    weight_decay=0.0,
    amsgrad=False,
)

trainer = Trainer(
    data_object=data_object,
    model=model,
    optimizers=[optimizer],
    criterion=torch.nn.CrossEntropyLoss(),
    n_gpus=8,
    metrics={'example_mse_1': MSE()},
)
wrk = Workflow(
    trainer=trainer,
)

# Run Workflow
wrk.fit()
wrk.valid()
wrk.test()
wrk.cleanup()
```
TabFormer Workflow

```python
from functools import partial
import torch
from gp.data_loaders import DataSpec, LPDataObject, MetaLoader
from gp.downstream import GPInferenceCluster, GPTrainCluster
from gp.metrics.metrics import BinaryAccuracy
from gp.models.pyg import HeteroModule, LinkPredictor, TorchNodeEmbedding
from gp.preprocessing.trans_dataset import Tabformer
from gp.workflow import Workflow
from gp.workflow.trainers import Trainer
# Preprocess
prep = Tabformer('/path/src', '/path/dst')
prep.transform()
# Load Data
meta = MetaLoader('/path/dst')
fan = 5
spec = DataSpec(shuffle=True, batch_size=8192, fanouts=[fan, fan], metadata=meta)
data_object = LPDataObject(
    train_dataloader=spec,
    valid_dataloader=spec.merge(shuffle=False),
    test_dataloader=spec.merge(shuffle=False),
    data_path='/path/dst',
    backend='PyG',
)
graph = data_object.construct_cache['graph']
metadata = data_object.metadata
# Initialize Modules
EMBEDDING_DIM = 64
emb = TorchNodeEmbedding(graph, EMBEDDING_DIM)
model = HeteroModule(
    metadata=metadata,
    dim_hidden=EMBEDDING_DIM,
    dim_out=64,
    n_layers=2,
    embedding=emb,
)
link_model = LinkPredictor(model)
```

# Set Up Training
```python
opt1 = partial(torch.optim.Adam, lr=1e-2)
opt2 = partial(torch.optim.SGD, lr=1e-2)
sched2 = partial(
    torch.optim.lr_scheduler.CyclicLR, base_lr=1e-2, max_lr=7e-1, mode="triangular"
)
)
opt_object_1 = Optimizer(model=emb, opt_partial=opt1)
opt_object_2 = Optimizer(model=model, opt_partial=opt2, scheduler_partial=sched2)
trainer = Trainer(
    data_object=data_object,
    model=link_model,
    optimizers=[opt_object_1, opt_object_2],
    criterion=torch.nn.BCELoss(),
    n_gpus=8,
    epochs=1,
    metrics={"acc": BinaryAccuracy()},
    amp=False,
)
wrk = Workflow(
    trainer=trainer,
)
wrk.fit()
```
Accelerating Heterogeneous GNNs

\[ h_i^{(\ell+1)} = \sum_{r \in R} f_{\theta_r}^{(\ell+1)}(h_i^{(\ell)}, \{h_j^{(\ell)} : j \in N_r(i)\}) \]

- **R-GCN** is one of the *most commonly* used GNN for heterogeneous graphs:

\[ H^{(\ell+1)} = \sum_{r=1}^{R} A_r H^{(\ell)} W_r^{(\ell+1)} \]

- Utilizes an **edge-type dependent** weight matrix to transform neighbors
Naive R-GCN Implementation

- **Naive implementation:** Iterate over each edge type individually

\[
H^{(\ell+1)} = \sum_{r=1}^{R} A_r H^{(\ell)} W_r^{(\ell+1)}
\]

\[
\text{out} = 0
\]

\[
\text{for } r \text{ in range(num_edge_types)}:\n\]

\[
\text{out} += \text{adj}[r] @ h @ w[r]
\]

\[
\text{return out}
\]

- **Flexible:** Any homogeneous GNN operator can be utilized, e.g., via PyG’s `to_hetero(model)` functionality
- **Inefficient:** Lack of parallelism across edge types
Vertically-Stacked R-GCN Implementation

- Leverage **full parallelism** by *stacking* adjacency matrices **vertically**

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

\[
A = \begin{bmatrix}
A_1 \\
A_2 \\
\vdots \\
A_{|\mathcal{R}|}
\end{bmatrix}
\]

Inefficient in case ...

- large number of edge types / sparse edge types
- there exists **multiple node types** (all features will be replicated for each edge type)
CUTLASS-based R-GCN Implementation

- **Idea:** Follow PyG’s generic **gather-scatter scheme** and perform edge-type dependent transformation in **edge-level space**

\[
\begin{align*}
H_r &= 1 \\
H_r &= 2 \\
H_r &= 3
\end{align*}
\]

- **Flexible:** *Any* heterogeneous GNN operator can be modelled this way (multiple aggregations, attention-based, ...)
- **Efficient,** even on sparse edge types, large number of node/edge types

Utilize **CUTLASS** Grouped GEMM to implement a **segment matmul**:

\[
\text{segment_matmul}(H, \text{offsets}, W)
\]
pyg-lib

- pyg-lib is a **low-level GNN library** exposing *optimized* operations for use in 🌎 PyG

  /pyg-team/pyg-lib

  - **GPU-accelerated neighbor sampling** based for large-scale graphs via *cugraph*
  - **GPU-accelerated heterogeneous GNNs** via *CUTLASS Grouped GEMM*
  - **GPU-accelerated sparse aggregations** via *cugraph-ops* integration *(coming soon)*

- Optimizations provide speed ups with *no* lines of code change
Data Loading Acceleration w/ CuGraph

Gathered on a 2 x A6000 node w/ AMD Ryzen Threadripper PRO 3975WX 32-Cores
RGCN Grouped GEMM Benchmark

- Single A100 GPU used on a node w/ 8 x 80GB A100 & AMD EPYC 7742 64-Core Processor
- Using FakeHeteroDataset w/:
  - avg_num_nodes=20000
  - num_node_types=4
- 2 RGCNConv s w/ 128 input, 16 hidden, & 10 output channels
Summary

- NVIDIA’s provides an API for effortless GPU accelerated GNN training/deploying
  - General open source availability on GitHub Q4
- NVIDIA-optimized PyG Container Coming Q4:
  - Performance-tuned & tested for NVIDIA GPUs
  - Sign up: [https://developer.nvidia.com/pyg-container-early-access](https://developer.nvidia.com/pyg-container-early-access)
- pyg-lib uses CuGraph & CUTLASS to enable further acceleration
  - Initial CUTLASS integration w/ PyG 2.1, additional accelerations coming soon