


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

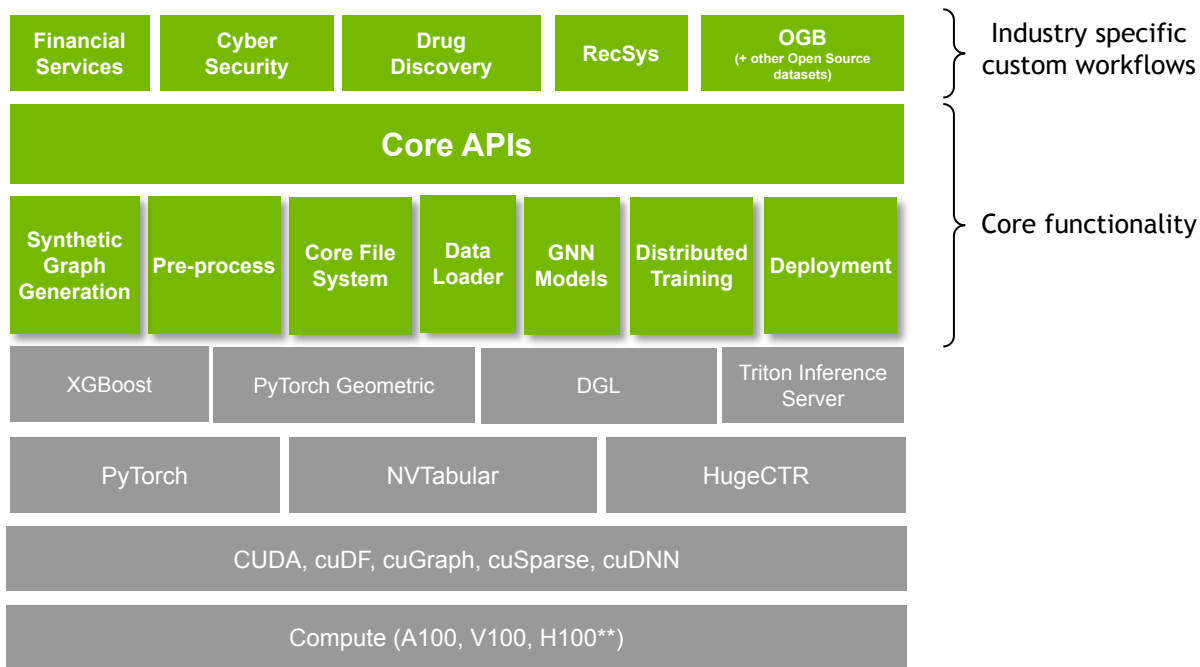
Challenges in the GNN Workflows

- **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

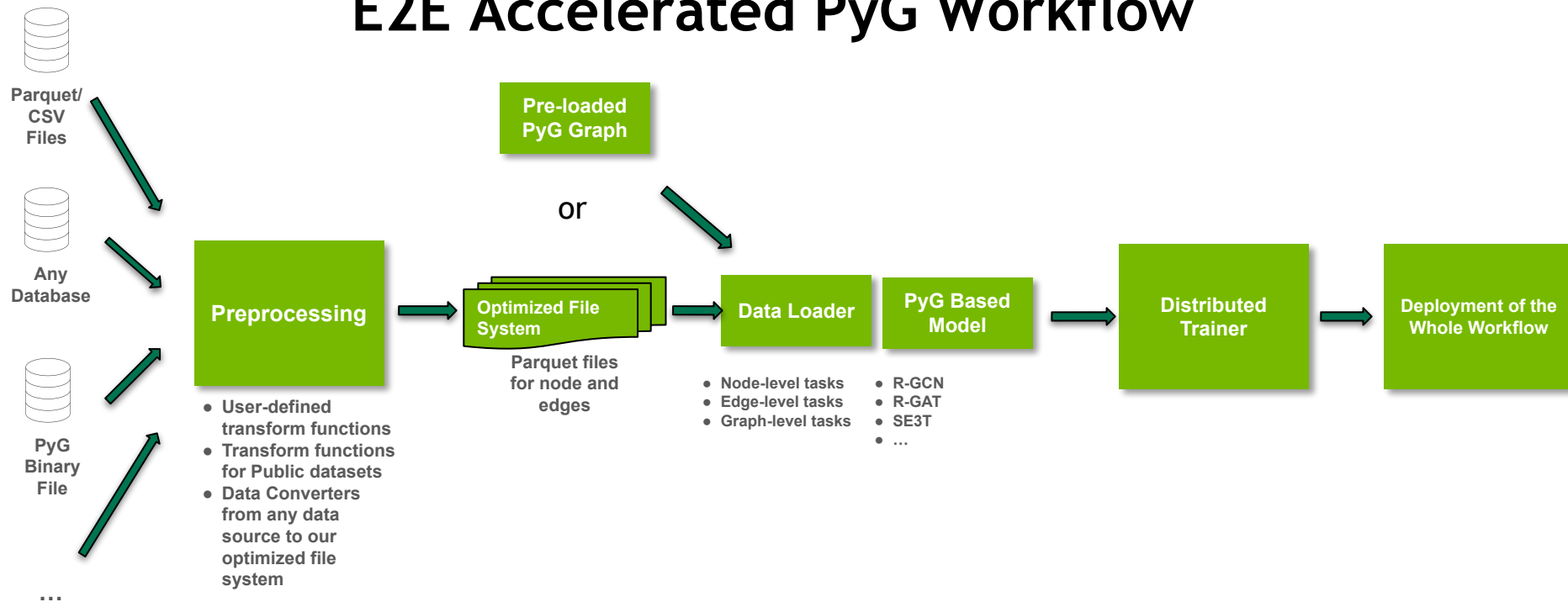
- **NVIDIA provides a flexible, easy-to-use, general API that allows for building end-to-end GNN workflows, addressing the previously mentioned challenges:**
 - Automatic, accelerated system management reduces the need for system knowledge
 - Flexibility allows for switching between tasks, models, and 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:
 - <https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s42485/>
- 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



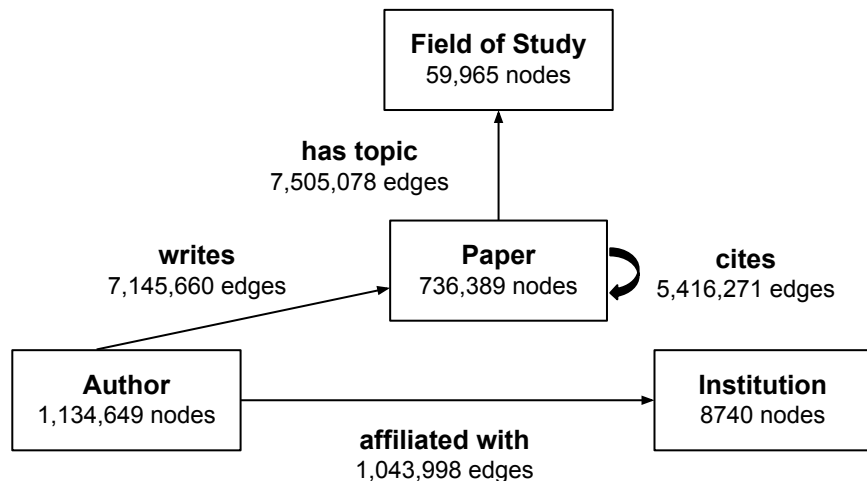
**H100 coming soon

E2E Accelerated PyG Workflow

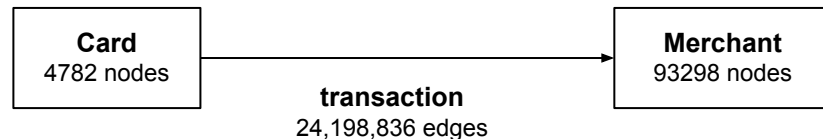


Example Datasets

- 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.



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

```
import torch
from gp.data_loaders import DataSpec, MetaLoader, NodeDataObject
from gp.metrics import MSE
from 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,
```

```
)
```

```
wrk = Workflow(
```

```
    trainer=trainer,
```

```
)
```

```
#Run Workflow
```

```
wrk.fit()
```

```
wrk.valid()
```

```
wrk.test()
```

```
wrk.cleanup()
```

TabFormer Workflow

```
from funtools 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
from gp.workflow.trainers.utils import OptWrapper
#Pre Process
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_data_loader=spec,
    valid_data_loader=spec.merge(shuffle=False),
    test_data_loader=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
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 = OptWrapper(model=emb, opt_partial=opt1)
opt_object_2 = OptWrapper(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()

#Train and Deploy to XGBoost on triton inference server
wrk.generate_embeddings_pyg()
train_cluster = GPTrainCluster(world_size=1)
inference_cluster = GPInferenceCluster(world_size=1)
ap = wrk.fit_xgboost(
    training_cluster=train_cluster, inference_cluster=inference_cluster, n_rounds=1
)
wrk.deploy("/", triton_batch_size=8192, downstream=True)
wrk.launch_inference_server()
b = wrk.pydep.run_inference_on_triton(dataloader=wrk.gp_xgb.triton_data_loader())
wrk.stop_inference_server()
wrk.cleanup()
```

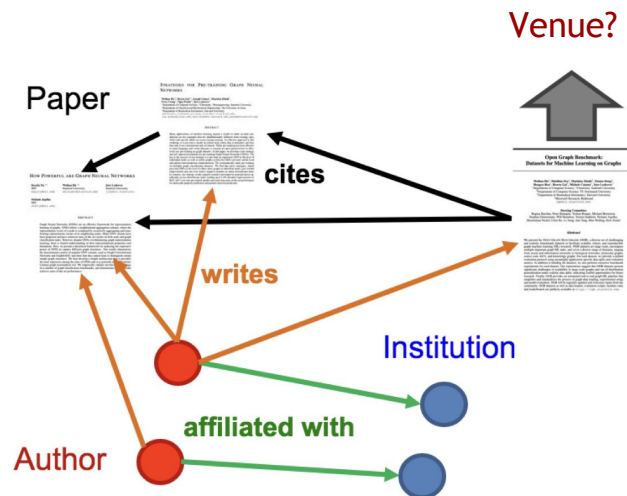
Accelerating Heterogeneous GNNs

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

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

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

- Utilizes an **edge-type dependent** weight matrix to transform neighbors




Naive R-GCN Implementation

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

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

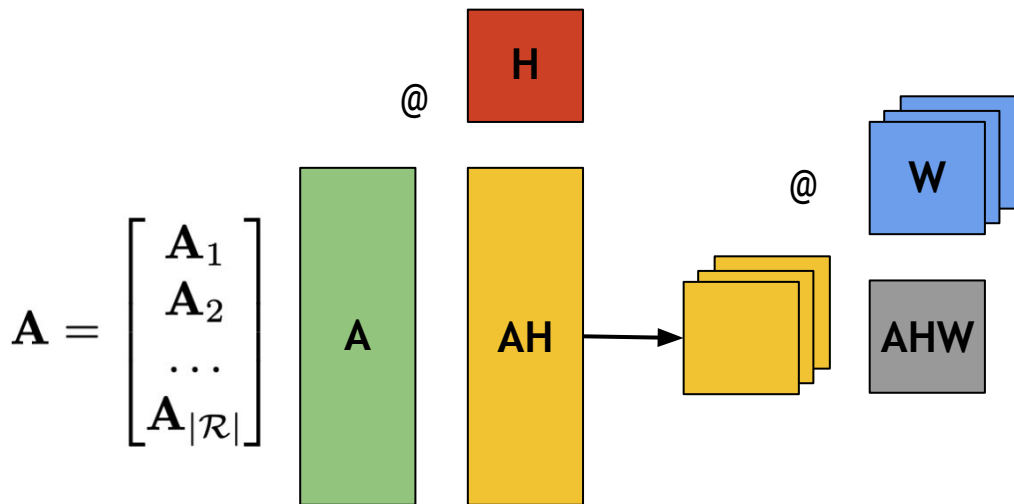
```
out = 0
for r in range(num_edge_types):
    out += adj[r] @ h @ w[r]
return out
```

- **Flexible:** Any homogeneous GNN operator can be utilized, 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)

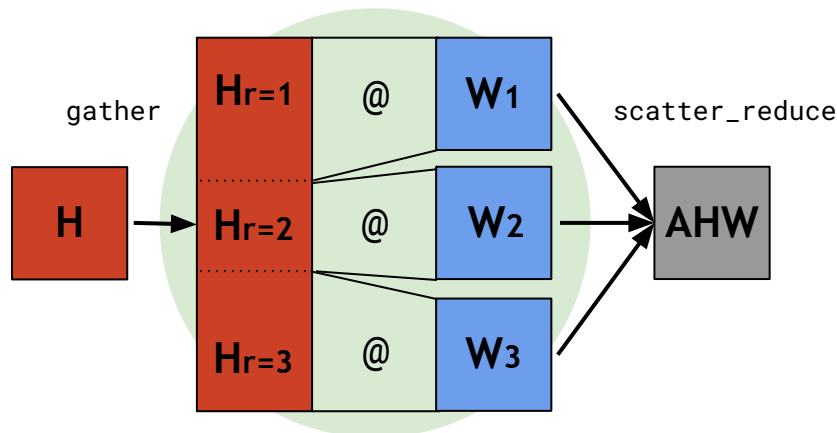


Inefficient in case ...

- **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




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

```
segment_matmul(H, offsets, W)
```

- **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

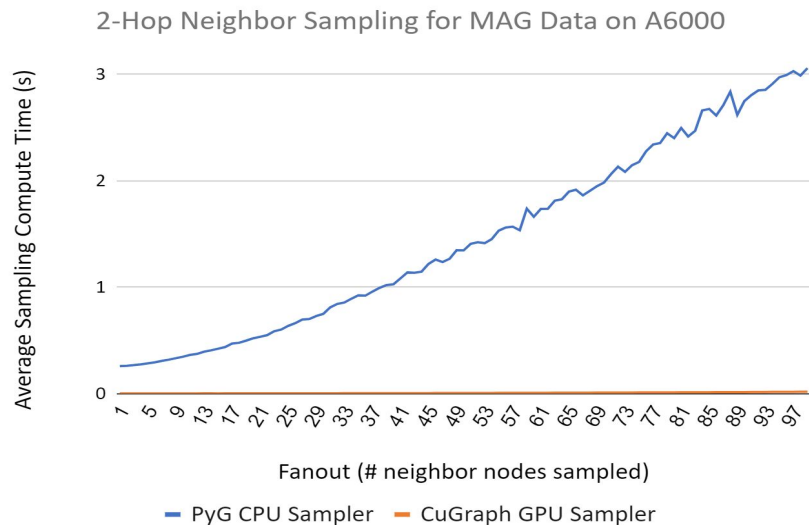


- pyg-lib is a **low-level GNN library** exposing *optimized* operations for use in  **PyG**

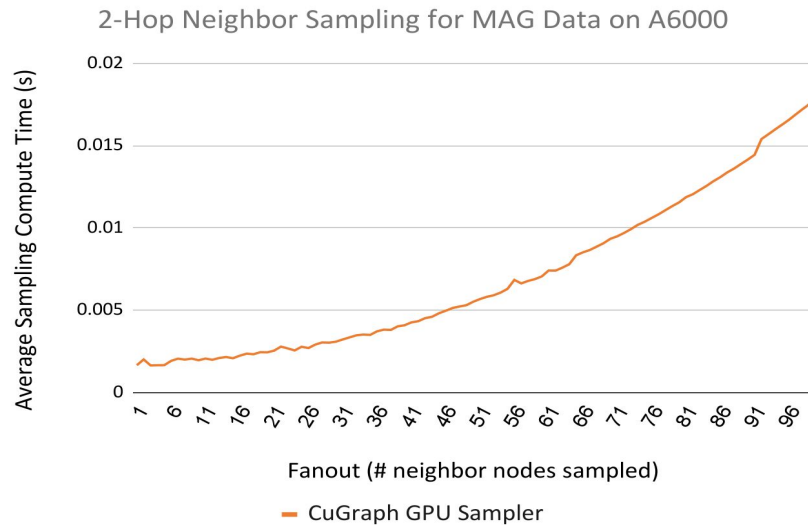
 /pyg-team/pyg-lib

- **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



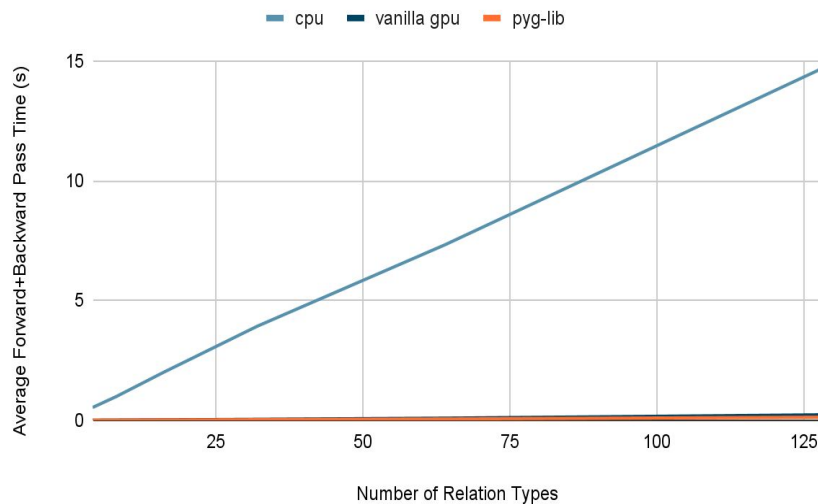
Zoom



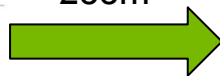
Gathered on a 2 x A6000 node w/
AMD Ryzen Threadripper PRO 3975WX
32-Cores

RGCN Grouped GEMM Benchmark

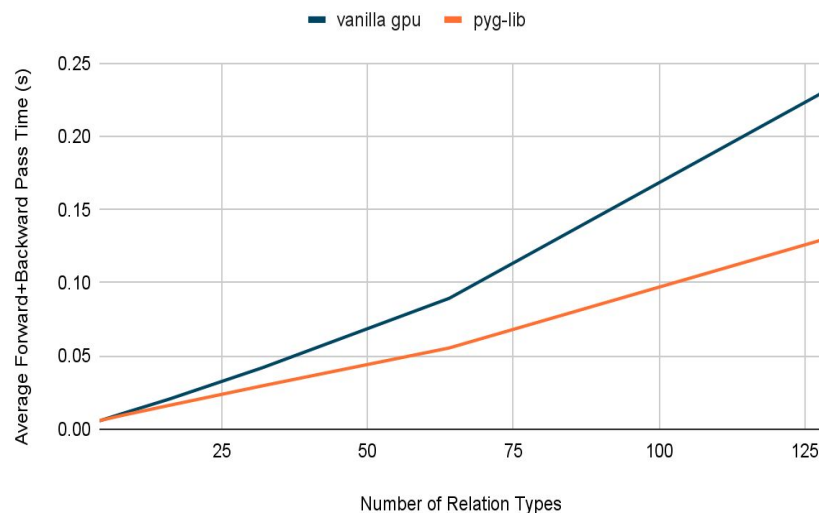
Time for Forward and Backward pass for RGCN on Synthetic Heterogenous Data



Zoom



Time for Forward and Backward pass for RGCN on Synthetic Heterogenous Data (Zoomed)



- 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
 - Product Page: <http://developer.nvidia.com/gnn-frameworks>
 - 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>
- pyg-lib uses CuGraph & CUTLASS to enable further acceleration
 - Initial CUTLASS integration w/ PyG 2.1, additional accelerations coming soon