cuGraph

ACCELERATED GRAPH ANALYTICS

- cuGraph library is a collection of GPU accelerated graph algorithms
  - Over three dozen algorithms, more planned
  - Scaling from small single GPUs graphs to massive multi-GPU clusters

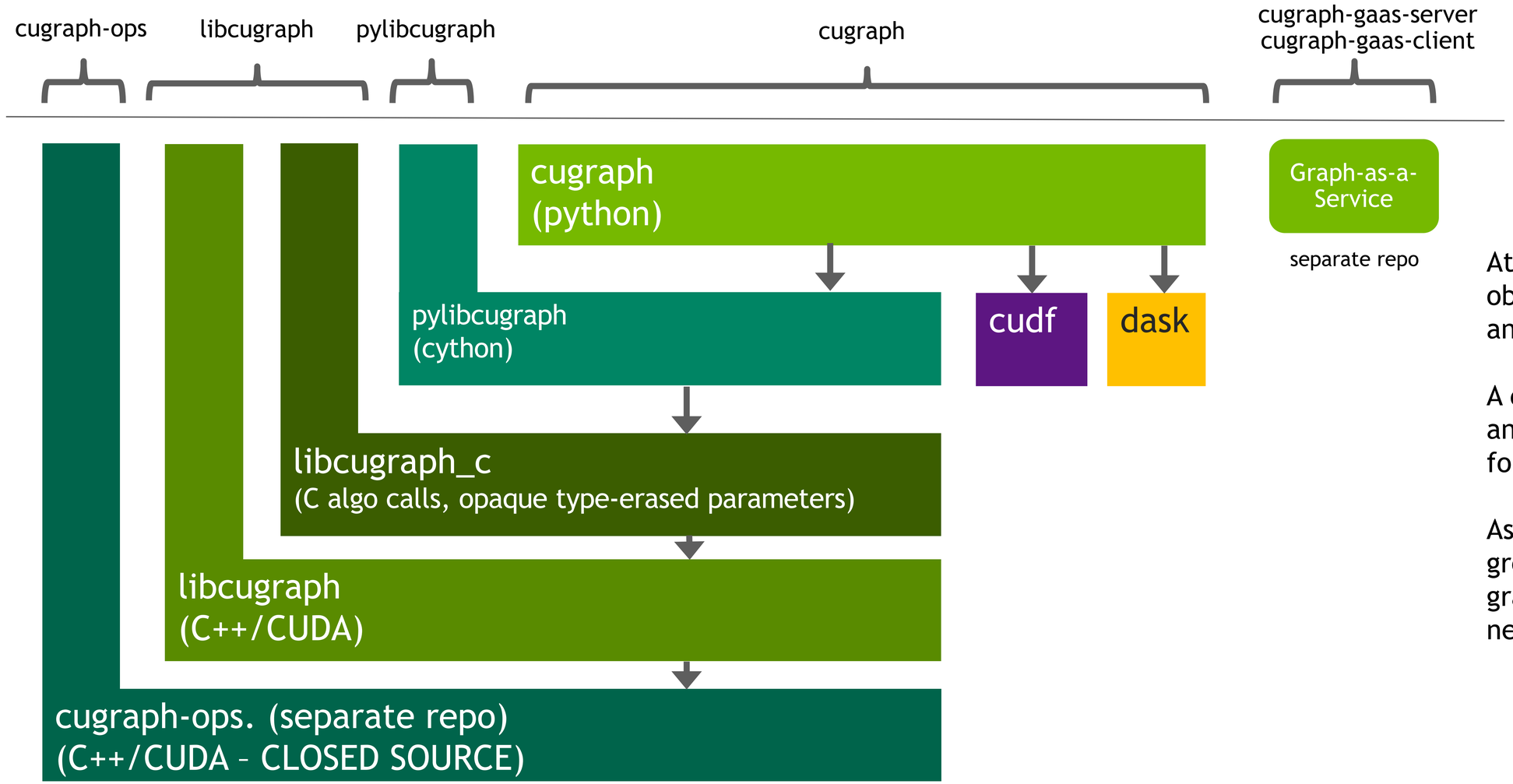
- users familiar with NetworkX will quickly recognize the NetworkX-like API provided in cuGraph, with the goal to allow existing code to be ported with minimal effort into RAPIDS.
  - Support cuDF Dataframes
  - CuPy and Numpy spare matrixes
  - NetworkX graph objects
  - more ..

- Python API
- C API
- C++ API

```
import cudf
import cugraph

# read data into a cuDF DataFrame using read_csv
gdf = cudf.read_csv("graph_data.csv", names=["src", "dst"], dtype=["int32", "int32"])

# We now have data as edge pairs
# create a Graph using the source (src) and destination (dst) vertex pairs
G = cugraph.Graph()
G.from_cudf_edgelist(gdf, source='src', destination='dst')

# Let's now get the PageRank score of each vertex by calling cugraph.pagerank
df_page = cugraph.pagerank(G)

# Let's look at the top 10 PageRank Score
df_page.sort_values('pagerank', ascending=False).head(10)
```

THE CUGRAPH STACK

many layers of cugraph



Conda packages

Pip coming soon

At the top layer, we work to obfuscate how data is stored and processed.

A data scientist can just select an algorithms without concern for underlying structures.

As you move down the stack, greater understanding of graphs and data structures is needed

As of release 22.06

AVAILABLE ALGORITHMS

Class	Algorithms	MNMG
Centrality		
	Katz	Yes
	Betweenness Centrality	
	Edge Betweenness Centrality	
	Degree Centrality	Yes
	Eigenvector Centrality	Yes
Community		
	Leiden	
	Louvain	Yes
	Ensemble Clustering for Graphs	
	Spectral-Clustering - Balanced Cut	
	Spectral-Clustering - Modularity	
	Subgraph Extraction	
	Triangle Counting	Yes
	K-Truss	
Components		
	Weakly Connected Components	Yes
	Strongly Connected Components	
Core		
	K-Core	
	Core Number	
Layout		
	Force Atlas 2	
Linear Assignment		
	Hungarian	

Class	Algorithms	MNMG
Link Analysis		
	PageRank	Yes
	Personal PageRank	Yes
	HITS	Yes
Link Prediction		
	Jaccard Similarity	
	Weighted Jaccard Similarity	
	Overlap Similarity	
	Sorensen	
Traversal		
	Breadth First Search (BFS)	Yes
	Single Source Shortest Path (SSSP)	Yes
Sampling		
	Random Walks (Uniform and Biased)	
	EgoNet	
	Node2Vec	
	Neighborhood	Yes
Other		
	Minimum Spanning Tree	
	Maximum Spanning Tree	
Structure		
	Renumbering	Yes
	Symmetrize	Yes

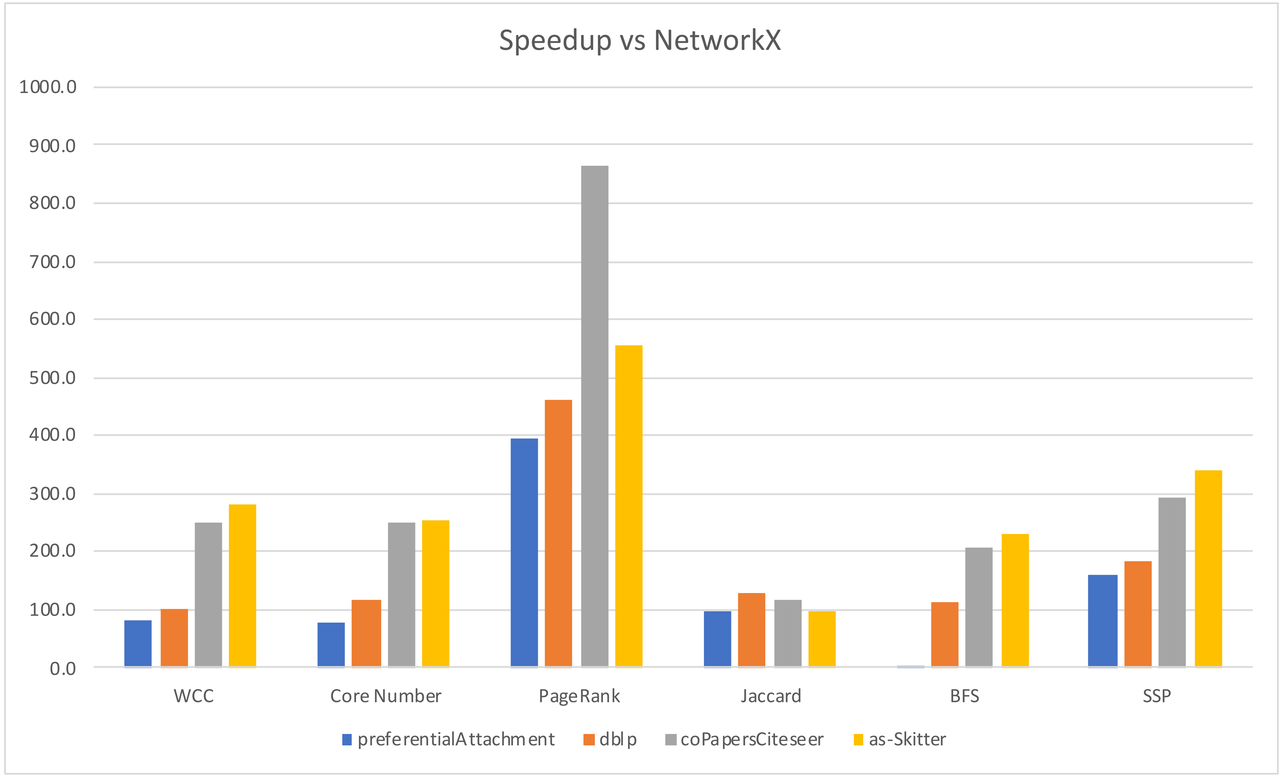
MNMG == Multi-Node Multi-GPU

more algorithms planned

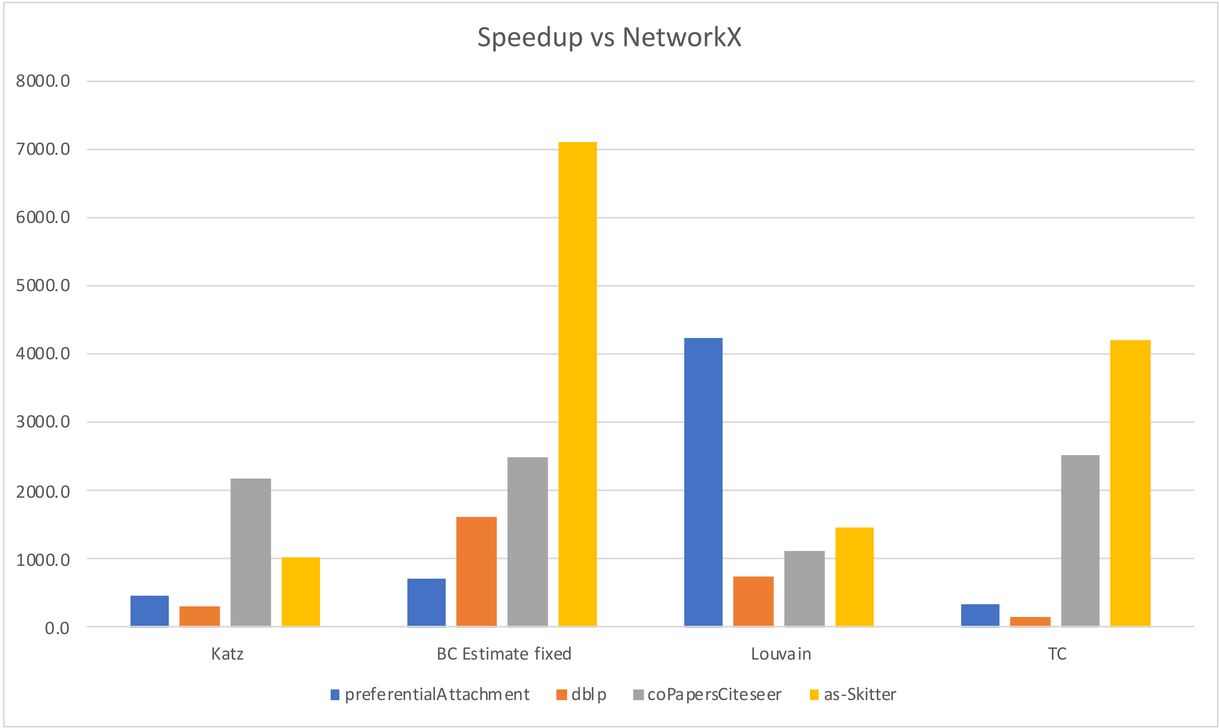
we are working to get everything MNMG

PERFORMANCE SPEEDUP

Single GPU Comparison Against NetworkX



Dataset	Nodes	Edges
preferentialAttachment	100,000	999,970
Dblp-2010	326,186	1,615,400
coPapersCiteseer	434,102	32,073,440
As-Skitter	1,696,415	22,190,596



Two charts since there is a change in Y axis size
Betweenness Centrality is with K = 100

GPU: Quadro RTX A6000
CPU: Single Intel(R) Xeon(R) CPU E5-2643 v4 @ 3.40GHz with 64GB RAM

NVIDIA

OVER 1 TRILLION EDGES !!!

cuGraph Scaling to Massive Scale

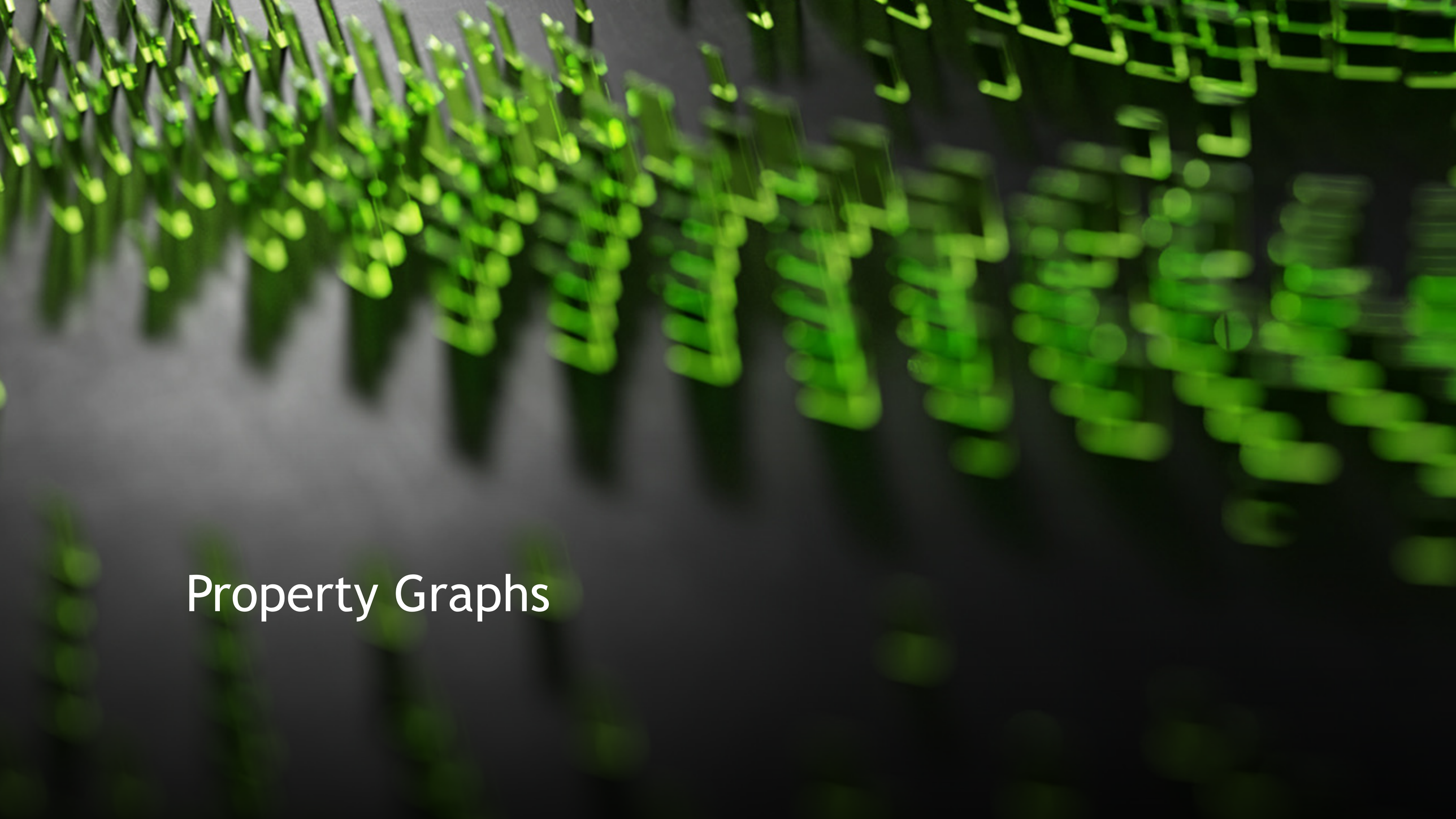
cuGraph scales smoothly from small graphs on 1 GPU to massive graphs with trillions of edges on 2,048 GPUs

- Being able to get answers in a matter of seconds regardless of graph scale if important
  - **PageRank:** Scale 36 (1.1 trillion directed edges) in 19.3 seconds (0.66 seconds per iteration, 2,048 GPUs)
  - **Louvain:** Scale 35 (0.55 trillion undirected edges or 1.1 trillion directed edges) in 336 seconds (1024 GPUs)
- Large scale testing just started
  - More analytics will be supported at 1000+ GPU scale
  - Continuous optimization for both memory usage and performance scalability

RMAT Data Generator using Graph500 Specs Edge Factor of 16

Scale	Number of Vertices	Number of Edges	COO Data Size (GB) in GPU
28	268,435,456	4,294,967,296	80
29	536,870,912	8,589,934,592	160
30	1,073,741,824	17,179,869,184	320
31	2,147,483,648	34,359,738,368	640
32	4,294,967,296	68,719,476,736	1,280
33	8,589,934,592	137,438,953,472	2,560
34	17,179,869,184	274,877,906,944	5,120
35	34,359,738,368	549,755,813,888	10,240
36	68,719,476,736	1,099,511,627,776	20,480
37	137,438,953,472	2,199,023,255,552	40,960

(hopefully) Look for HPEC paper on scaling to 4 trillion edges

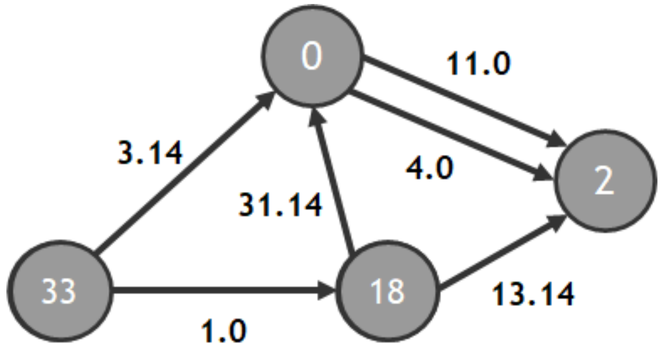

Property Graphs

PROPERTY GRAPH

Added in release 21.12

- Prior to 21.12, cugraph users were limited to creating graphs from source and destination vertices, and a single (optional) numeric weight value for each edge.

source	destination	weight
33	0	3.14
33	18	1.0
18	0	31.14
0	2	4.0
0	2	11.0
18	2	13.14



- …but real data can contain numerous attributes of various types on both vertices and edges
  - Being able to store, access, and process attributes is important
- Property Graphs offer this ability, and are critical for supporting Heterogenous graph in GNNs
- *Note: this is not proposal to support a graph database or even a graph query language*

PROPERTY GRAPH

Added in release 21.12

- Property Graphs offer the ability to load graph data containing multiple, heterogeneous attributes on items that represent vertices and edges.
- Attributes can then be used:
  - as a means to filter or select certain edges and/or vertices for further analysis using graph analytics
  - as weight values for graph algorithms that consider edge weights
  - as data that can be added to graph algorithm results for use by client applications (e.g. GNN training)

customers

Cust_ID	Zip	Card_num
18	78757	23451
33	78750	12345

stores

Store_num	Zip	Employees
0	78750	65
2	78757	51

purchases

Cust_ID	Store_num	Date	Amount
33	0	1/1/21	3.14
18	2	1/3/21	13.14
18	0	1/4/21	31.14

referrals

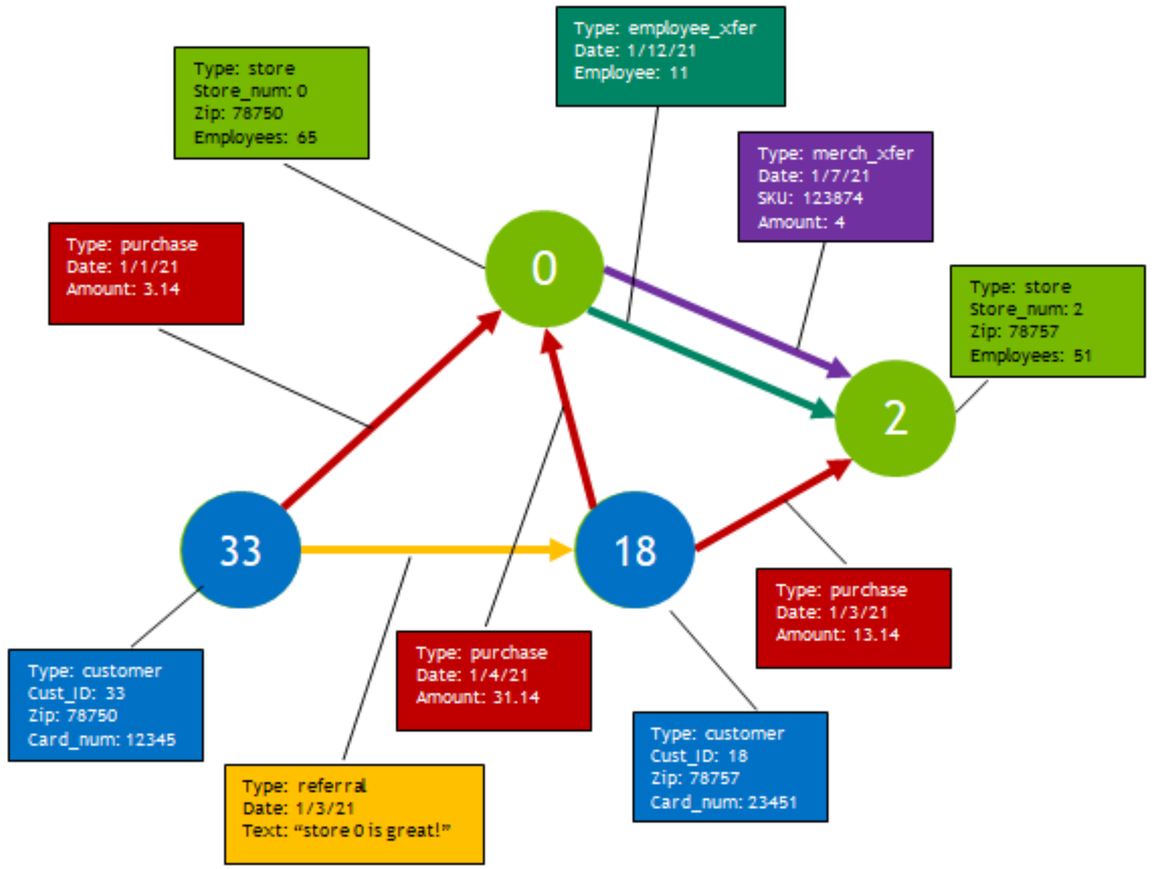
Cust_ID_1	Cust_ID_2	Date	Text
33	18	1/3/21	“store 0 is great!”

merch xfers

From	To	Date	SKU	Amount
0	2	1/7/21	123874	4

employee xfers

From	To	Date	Employee
0	2	1/12/21	11

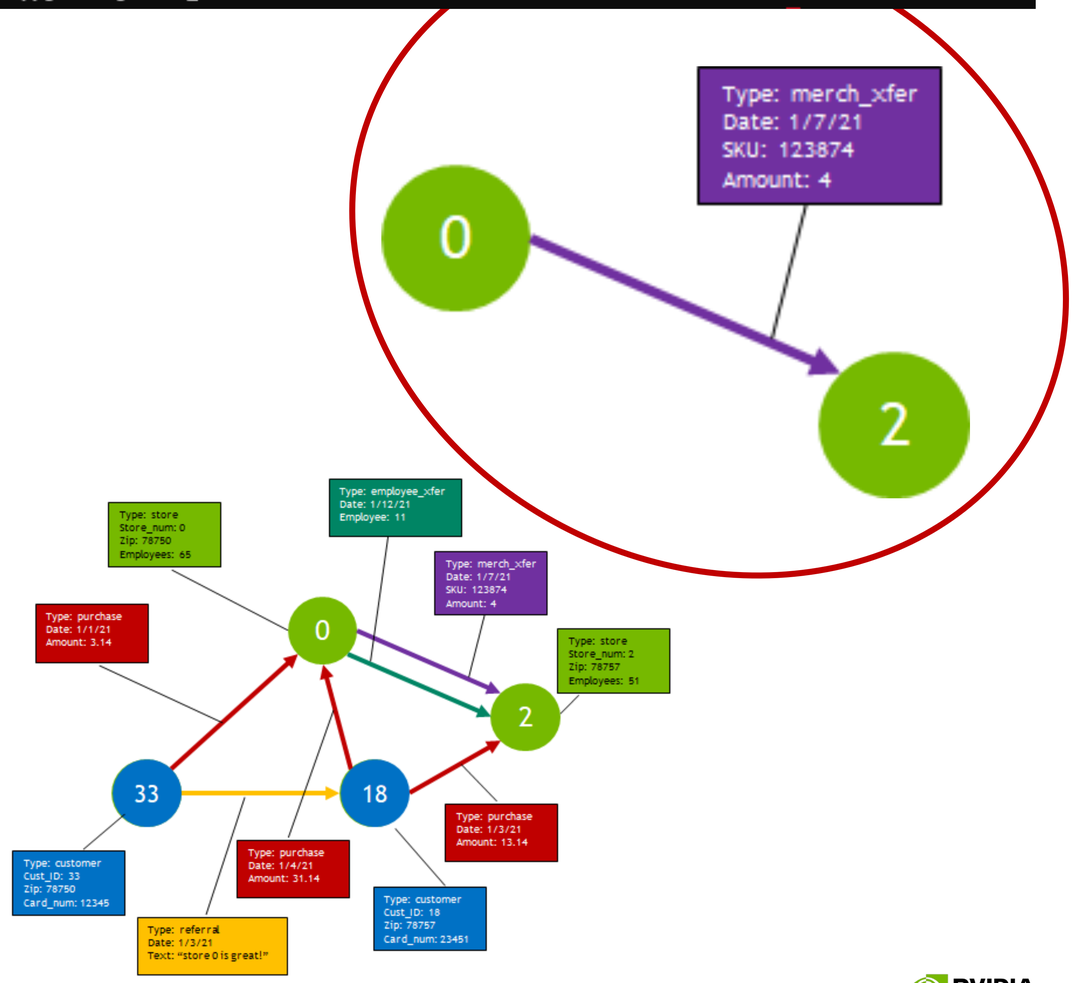


PROPERTY GRAPH

- Example: use a Property Graph to load various datasets as edges and vertices with attributes, use the Property Graph API to extract different graphs based on attributes to run analysis.

```
pG = cugraph.experimental.PropertyGraph()

pG.add_vertex_data(customers_df,
                   type_name="customers",
                   vertex_col_name="Cust_ID")
...
pG.add_edge_data(purchases_df,
                 type_name="purchases",
                 vertex_col_names=("Cust_ID", "Store_num"))
...
selection = pG.select_vertices(f"{pG.type_col_name}=='stores'")
selection += pG.select_edges(f"{pG.type_col_name}=='merch_xfers'")
G = pG.extract_subgraph(selection=selection, edge_weight_property="Amount")

print(G.view_edge_list())
```

```
(rapids) root@17e55ac97b56:/Projects/cugraph/python/cugraph# python pgdemo4.py
   weights  src  dst
0      4.0    0    2
```

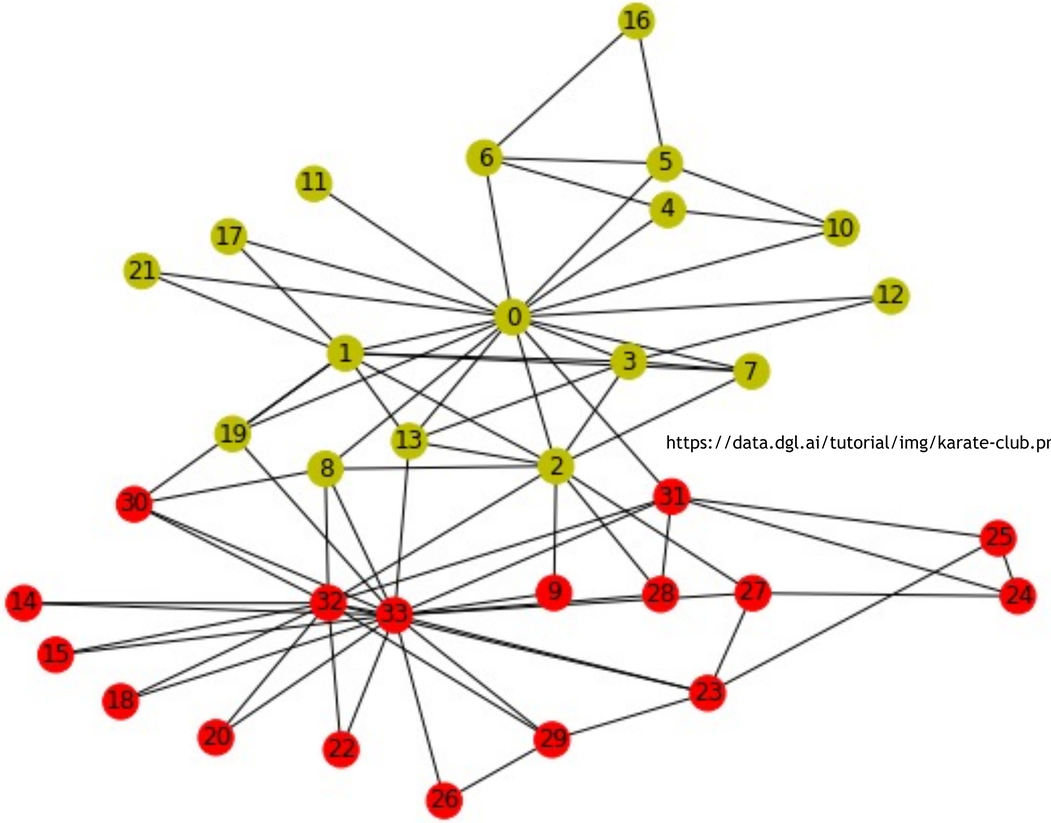


PROPERTY GRAPH

- Example: use a Property Graph to load the Zachary Karate Club dataset, use Louvain to find the two primary partitions, use Pagerank to find the top 3 influential vertices in each partition.

```
import cudf
import cugraph
from cugraph.experimental import PropertyGraph

# Read edgelist data into a DataFrame, load into PropertyGraph as edge data.
df = cudf.read_csv("karate_directed.csv",
                   delimiter=" ",
                   dtype=["int32", "int32", "float32"],
                   header=None)

pG = PropertyGraph()
pG.add_edge_data(df, vertex_col_names=("0", "1"))

# Run Louvain to get the partition number for each vertex.
# Set resolution accordingly to identify two primary partitions.
(partition_info, _) = cugraph.louvain(pG.extract_subgraph(), resolution=0.6)

# Add the partition numbers back to the PropertyGraph as vertex properties.
pG.add_vertex_data(partition_info, vertex_col_name="vertex")

# Use the partition properties to extract a Graph for each partition.
G0 = pG.extract_subgraph(selection=pG.select_vertices("partition == 0"))
G1 = pG.extract_subgraph(selection=pG.select_vertices("partition == 1"))

# Run pagerank on each graph, print results.
pageranks0 = cugraph.pagerank(G0)
pageranks1 = cugraph.pagerank(G1)
print(pageranks0.sort_values(by="pagerank", ascending=False).head(3))
print(pageranks1.sort_values(by="pagerank", ascending=False).head(3))
```

```
(rapids) /cugraph/python/cugraph# python pgdemo3.py
pagerank  vertex
0   0.192222       0
12  0.108455       1
6   0.084853       2
pagerank  vertex
1  0.191775      33
2  0.150855      32
3  0.072723      31
```



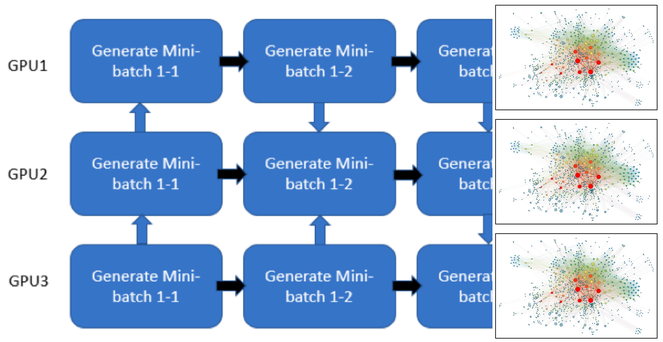
https://data.dgl.ai/tutorial/img/karate-club.png

Look for BLOG on Property Graphs in April 2022


Graph-as-a-Service - GaaS

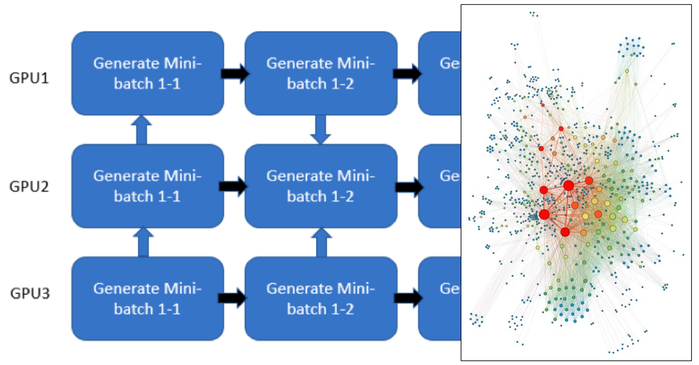
GAAS – GRAPH-AS-A-SERVICE

Background in a GNN Environment

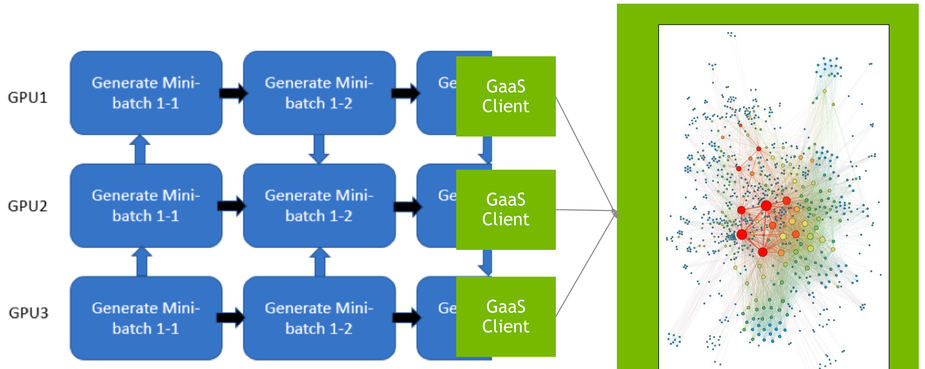


Data parallel
No issues with small graphs

… BUT …
What happens when the graph gets large?



Data parallel no longer works
Resource contention



GaaS

GaaS

Addresses issues by separating
Graph from application

GAAS – GRAPH-AS-A-SERVICE

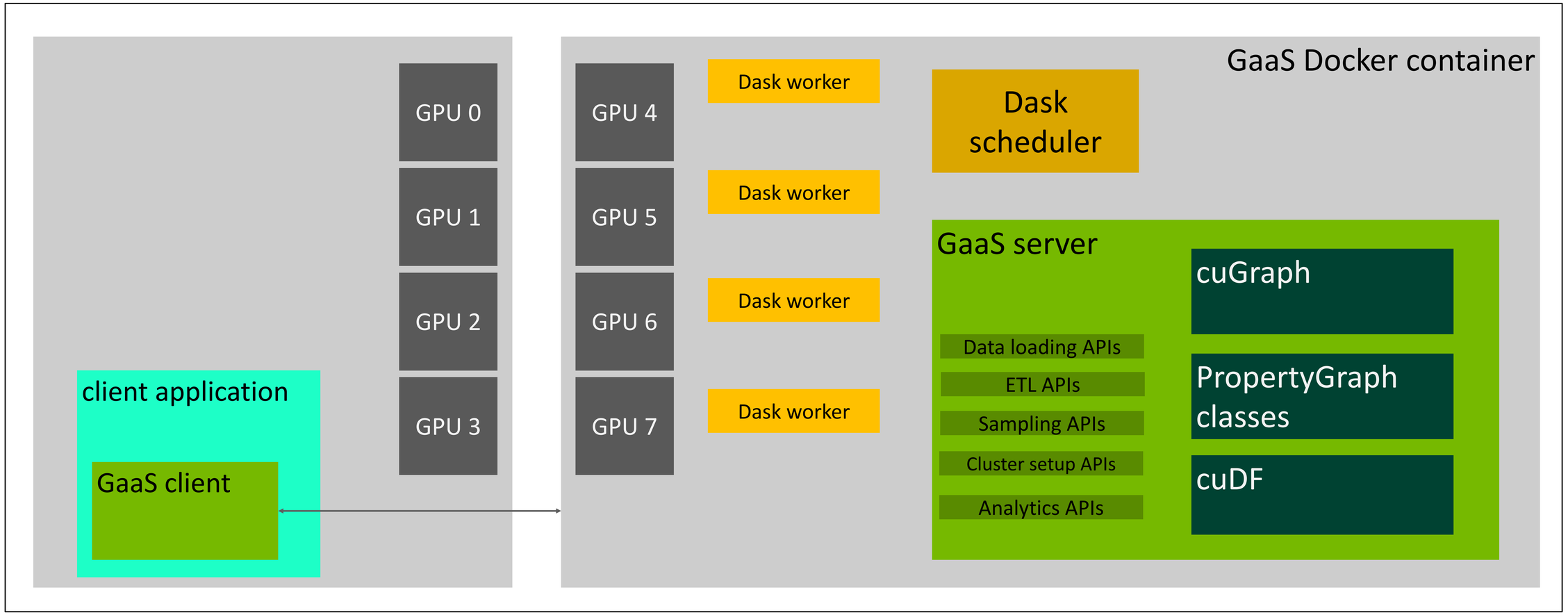
Background and goals

- Separate large graph management and analytic code from application code
  - The application, like GNN code, should be isolated from the details of cuGraph graph management, dedicated multi-node/multi-GPU setup, feature storage and retrieval, etc.
  - Scaling from single GPU (SG), to multi-GPU (MG), to multi-node/multi-GPU (MNMG) should not require changes to the graph integration code
- Support multiple concurrent clients/processes/threads sharing one or more graphs
  - No need for each client to have access to graph data files, perform ETL, etc.
  - Simplify concurrent programming – synchronization, batch processing details, etc. are implemented server-side
  - The GNN user should be able to easily partition and isolate hardware resources between graph analytics and training
  - The application/GNN code should be able to prefetch graph samples while training without resource contention
- Simplify MNMG deployments
  - Docker images contain all required, compatible packages
  - Clients use the same GaaS APIs for both SG and MG

NOTE: GaaS is still a work-in-progress and changes frequently as we add functionality and improve performance.

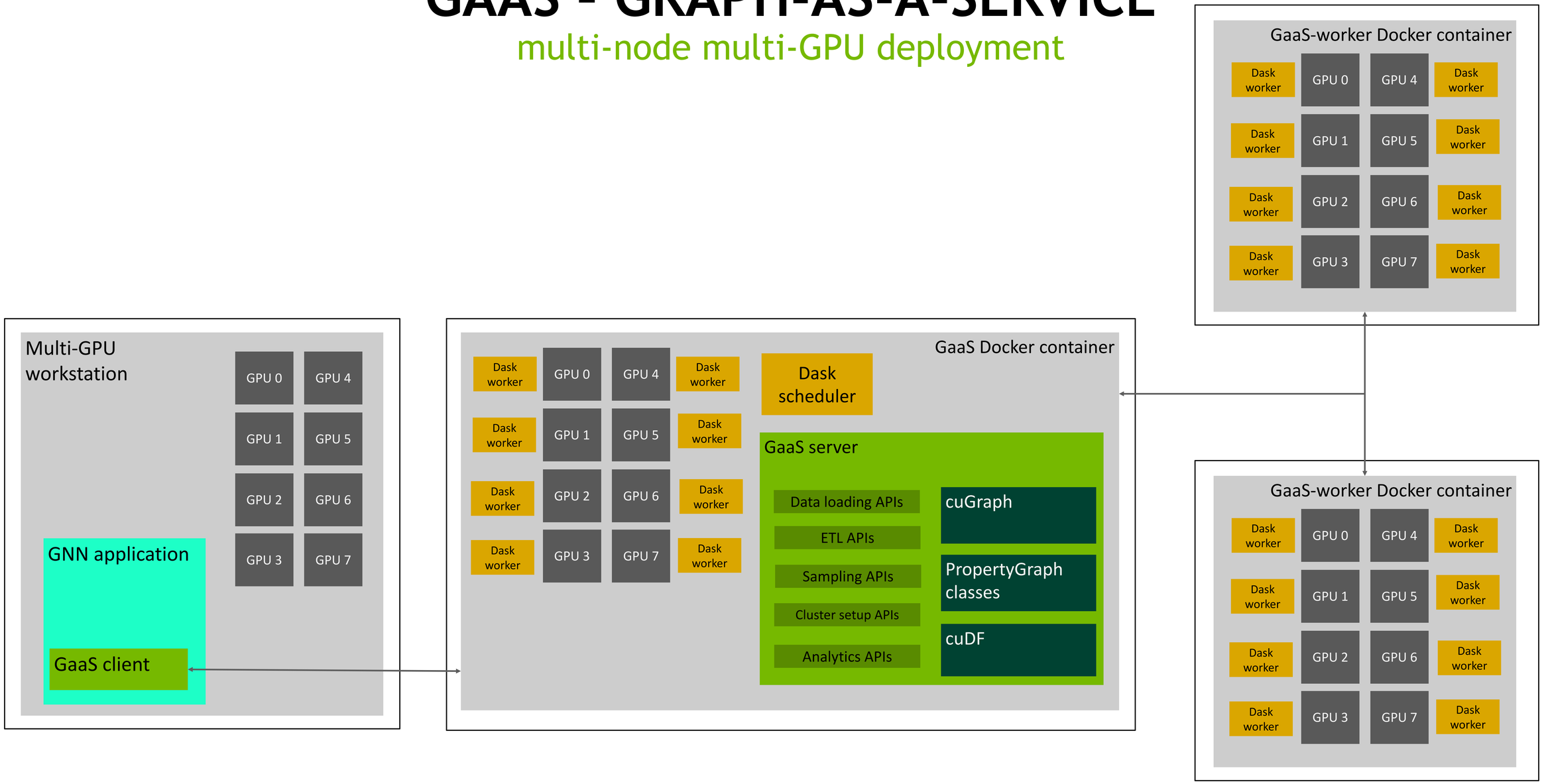
GAAS - GRAPH-AS-A-SERVICE

multi-GPU deployment



GAAS - GRAPH-AS-A-SERVICE

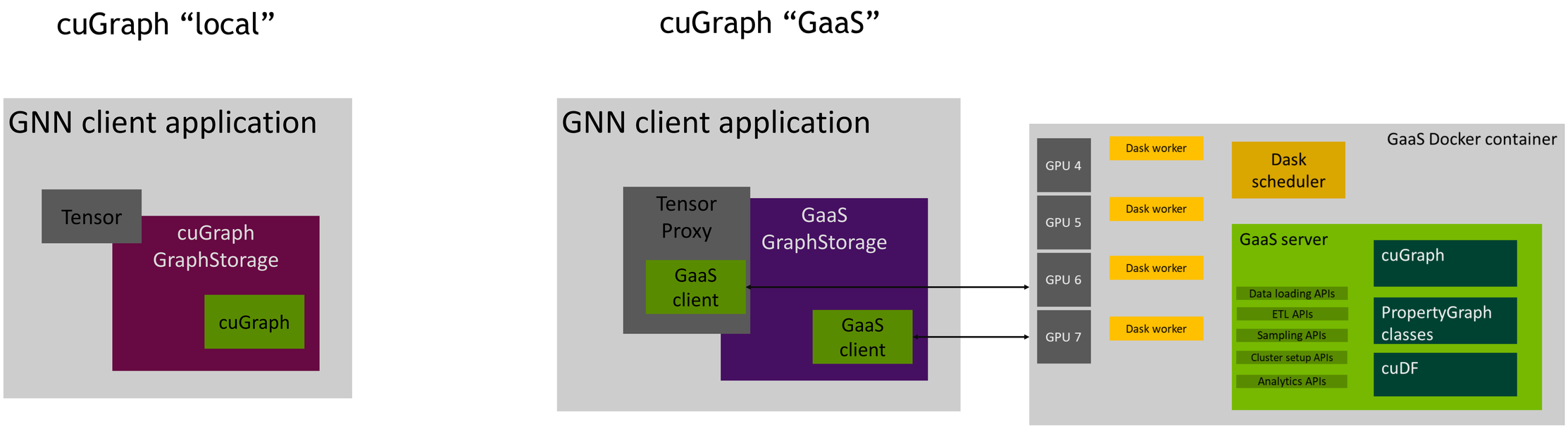
multi-node multi-GPU deployment



GAAS - DESIGN PATTERNS

Proxy objects

- Support both cuGraph “local” and “GaaS” modes without code changes
  - Objects for either mode have the same API (duck-typed)
  - “local” mode object accesses cuGraph directly
  - “GaaS” mode object accesses the server via the client API and manages remote object lifecyle using RAII-style implementation
- Simplifies GNN integration by encapsulating integration details



GAAS – DESIGN PATTERNS

Proxy objects

- Use the client to load a graph with vertex and edge data from CSV files.

```
>>> from gaas_client import GaasClient
>>>
>>> client = GaasClient()
>>> G = GaasGraph(client)
>>>
>>> # Add vertex and edge data to the graph on the server
>>> client.load_csv_as_vertex_data("/data/vertex_data.csv",
...                                ...,
...                                graph_id=graph_id)
>>> client.load_csv_as_edge_data("/data/edge_data.csv",
...                              ...,
...                              graph_id=graph_id)
```

- Retrieve the vertex features for vertices 100, 101, 102
- This requires the application code to maintain the client object and graph ID for future calls.

…or

```
>>> client.get_graph_vertex_data(vertex_ids=[100,101,102], graph_id=0)
array([[0, 11223, 8793],
       [1, 11289, 2399],
       [2, 11267, 1500]], dtype='int64')
```

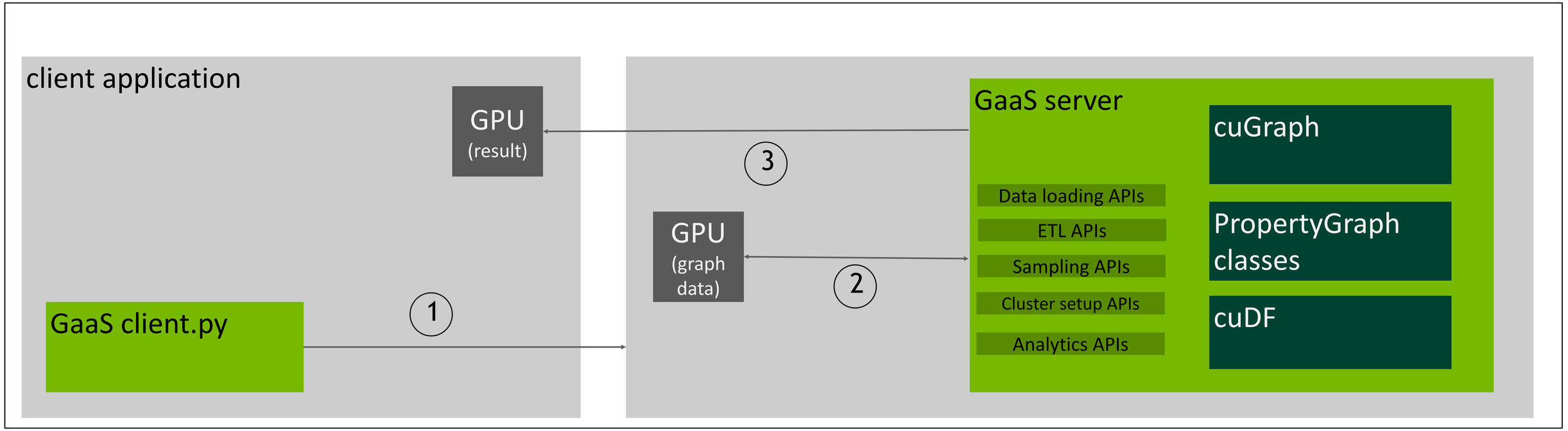
- Construct a TensorProxy using the client object and graph ID for vertex data.
- This tensor can be passed to other application code as-is to access vertex data.

```
>>> vert_features = TensorProxy(client, graph_id, category="vertex")
>>> vert_features[[100,101,102]]
tensor([[    0, 11223,  8793],
        [    1, 11289,  2399],
        [    2, 11267,  1500]])
```

GAAS - UPCOMING ENHANCEMENTS

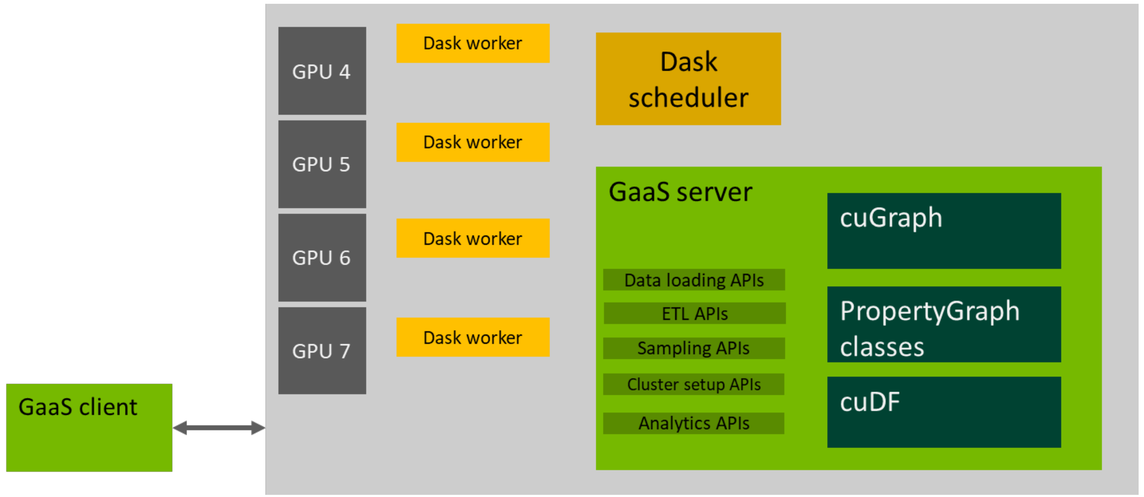
Performance improvements for clients with GPUs

- Allow GaaS to populate client GPU with result directly
    1. Client makes request
    2. GPU graph analytic performed on graph in server GPU memory
    3. Result is written to client GPU for use by client-side tasks (training, etc.)
- Eliminates the server device-to-host copy, serialization, and transfer steps normally needed when returning results back to the client over RPC.



GAAS – UPCOMING ENHANCEMENTS

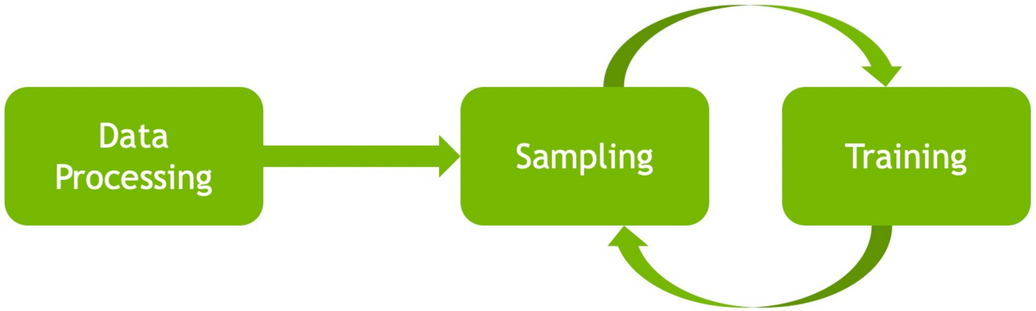
- Conda packages
  - Server – requires GPU and CUDA, installs cuGraph, cuDF, dask, RPC package, others
  - Client – no GPU requirement, installs only client, numpy, and RPC package
    - If client has GPUs, UCX-Py can also be installed for device-device copies of results from server to client
- Docker image
  - Assign GPUs using “docker run –gpus …”
  - Volume mount a shared dir to “/dask” to share auto-generated dask scheduler JSON configuration for MNMG
  - Volume mount a shared dir to “/logs” to access logs from all workers, GaaS, etc.
  - Volume mount a shared dir to “/config” to specify other non-default config options (enable/disable NVLink, InfiniBand, force specific protocols, specify device names, etc.)




GNN Support

GNN WITH CUGRAPH AND DGL

FILL THE GAP IN BETWEEN THE SAMPLING AND TRAINING



- Motivations:
  Existing frameworks do samplings on CPUs and train on GPUs. it is time-consuming.
- Avoid moving data number of epochs X batch between GPUs and CPUs.
- Sampling and training has closer relationship than data processing.

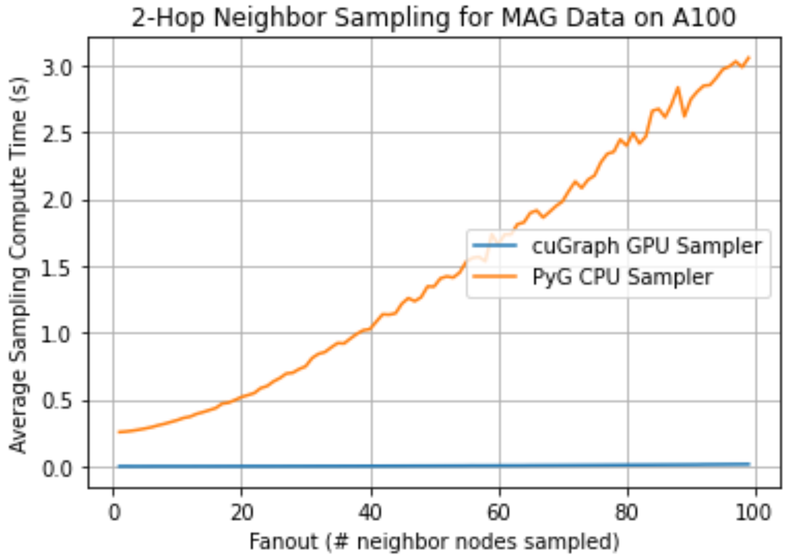
Operations	Data processing	Sampling	Training
Device of existing method	CPU or GPU	CPU	GPU
Devices of our method	GPU	GPU	GPU

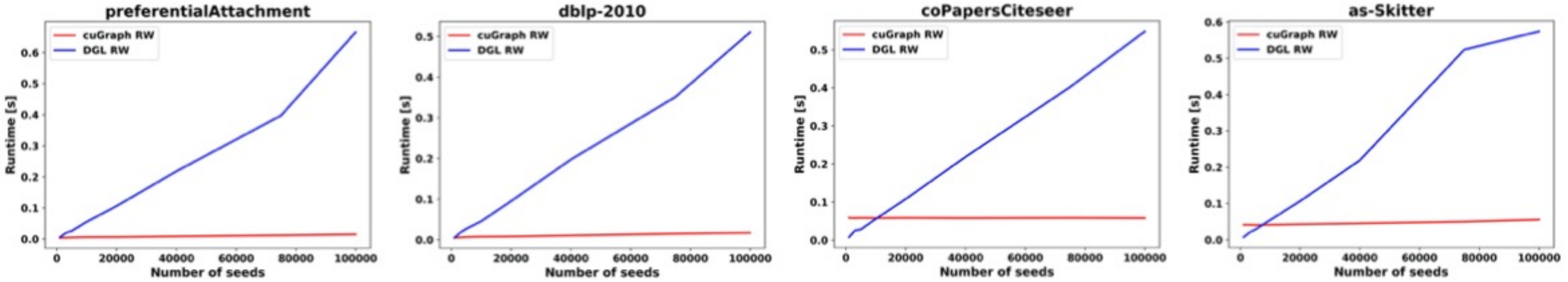
GPU ACCELERATED RANDOM WALK

Common Sampling Method

- Graph Sampling can consume 50 – 80% of training time

- cuGraph has been working on expanding the list of graph sampling algorithms and working to have the algorithms support multiple seeds
  - Egonet
  - Random Walk
  - Node2Vec
  - Neighborhood

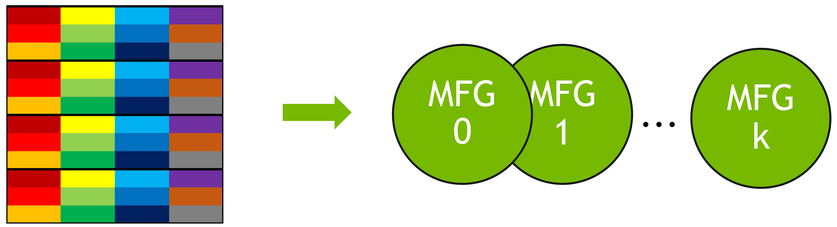


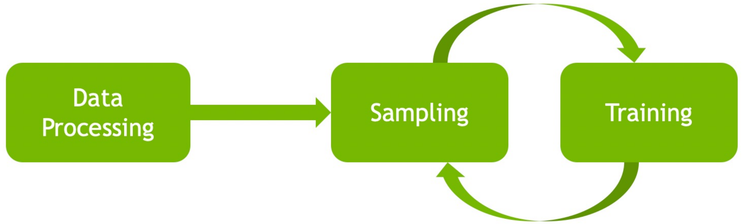


cuGraph RandomWalk vs DGL RandomWalk with various number of seeds

ROLE OF CUGRAPH-OPS

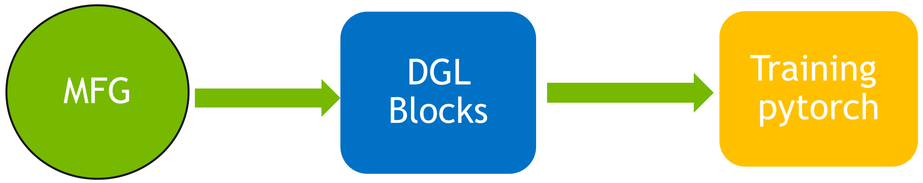
- Neighborhood sampling calls to create Message Flow Graphs





Cugraph-ops is a library that is composed of highly optimized and performant primitives associated with GNNs and related graph operations, such as training, sampling and inference.
Currently, we have random walk, neighborhood sampling, and in the future we will support aggregation functions as well.

Current workflow



Note: Currently we support DGL, will support PyG in future.

UPCOMING CUGRAPH FEATURES

DRAFT

- Migrate all Existing algorithms to support MNMG
  - Everything* should smoothly scale from one to thousands of GPUs
  - 32 Graph Algorithms
    - 9 have been upgraded to support Multiple GPUs across Multiple Nodes (MNMG)
- More Algorithms
  - More multi-seed versions,
  - all-pair shortest path,
  - propagation algorithms,
  - flow algorithms, etc..
  - See cuGraph GitHub site for roadmap list of algorithms
- Better Graph Type Support
  - Property Graphs, Bipartite Graphs, Multigraphs, HyperGraphs
- More and Better support for GNNs
  - More sampling algorithms
    - All sampling algorithm supporting Multi-Node Multi-GPU (MNMG) environments for scaling to massive graphs
  - Heterogeneous Graph Support
    - Improved Property Graph
  - More model optimization support in cugraph-ops library
- Data Masking
  - Provide a filter over the data for processing just a portion without extracting or reloading the graph
    - Time Windowing or View by Attribute(s).
- Performance and Scaling improvements
  - This is a constant on-going activity