GPU ACCLERATED GRAPH ANALYTICS AND GNN ON HUGE DATASETS

JOE EATON

SALISHAN CONFERENCE, 4/27/2022

3 KEY TAKEAWAYS

Big Announcements

See the Recorded GTC talk : S41407 – State of cuGraph

- **Massive Graph Support**
  - cuGraph has support for massive size graph over thousands of GPUs
  - Ease of programming new algorithms with graph primitives
  - Graph construction and feature engineering benefit from cuDF
- **Property Graphs**
  - cuGraph supports property graph model (aka heterogeneous graphs, knowledge graphs)
  - Very useful in real-world applications, commonly asked for by our customers
- **Supporting GNNs**
  - cuGraph is going all-in on GNN support in DGL and PyG
  - DGL 1.0 planned to include cuGraph accelerated graph storage and sampling
  - Combined with nvTabular for lambdaOps, and GPU accelerated parameter servers for features
  - Example notebooks using DGL for realistic workflows end-to-end available from Joy of Cooking team

CUGRAPH VISION STATEMENT

Make graph analysis ubiquitous to the point that users just think in terms of analysis and not technologies or frameworks.

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
And more control is given to the user

THE NEW CUGRAPH STACK

The data scientist now has many paths to cugraph

libcugraph

pylibcugraph

cugraph

Julia
(proposed)

R
(proposed)

CuPy
(v10.0)

cugraph
(python)

pylibcugraph
(cython)

cudf

dask

libcugraph_c
(C algo calls, opaque type-erased parameters)

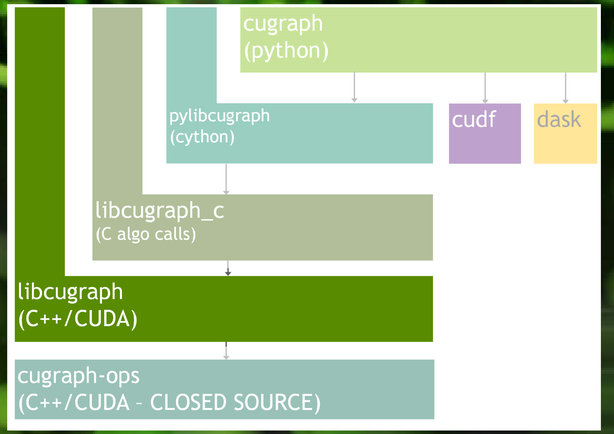
libcugraph
(C++/CUDA)

cugraph-ops
(C++/CUDA – CLOSED SOURCE) - Used in DGL support

Julia and R support are under consideration, but waiting for community feedback on interest

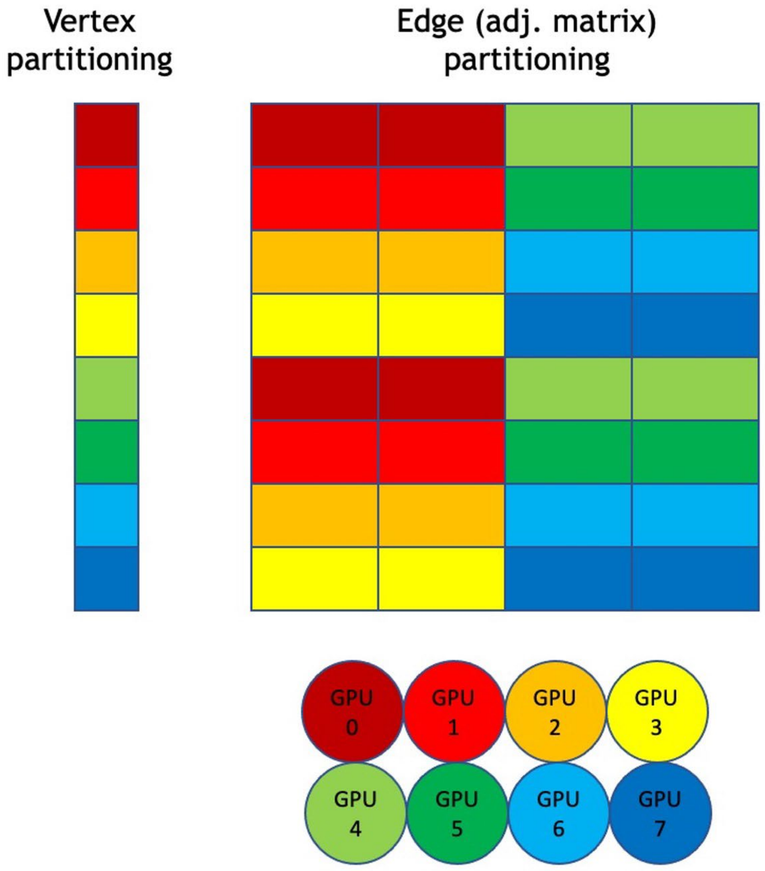
NVIDIA

Graph Primitives



GRAPH PRIMITIVES

- Goal: To make high-performance Multi-Node Multi-GPU implementations ubiquitous to the point that ***algorithm developers*** just think in terms of operations on graphs, vertices, and edges.
  - Hide complexity of 2D distributed data
    - Each GPU gets a tile of vertices
    - Each GPU gets multiple tiles of the adjacency matrix
  - Hide complexity of using both CSR and DCSR structures
    - CSR is effective when (local-)vertex degree >> 1
    - DCSR is effective when (local-)vertex degree << 1 (hypersparse)
    - A tile has a part of the edge list in CSR (for high-degree vertices) and the remaining part in DCSR (for low-degree vertices, # degrees << # GPUs in the same column)
    - Different kernels applied based on vertex degrees



Users write the same code for SG (no partitioning) as for MNMG (complex 2D partitioning)

GRAPH PRIMITIVES

- Graph level primitives:
  - coarsen, extract induced subgraphs, symmetrize, transpose, count self-loops & multi-edges, compute (weighted-)in/out-degrees
- Vertex & edge level primitives:
  - Visit all the edges:
    - apply a user provided functor to each edge, reduce the functor outputs to a single number (e.g. sum the edge weights of a graph)
  - Visit all the vertices:
    - Apply a user provided functor to each vertex, reduce the functor outputs to a single number (e.g. test PageRank convergence)
  - Visit all the neighbors of each vertex:
    - apply a user provided functor to each neighbor, reduce the functor output to a single vertex property value (e.g. PageRanks, Katz centrality, HITS)
  - Additional primitives for more complex patterns
    - More primitives to be added to support more graph analytics
- Multiple analytics have been re-implemented using graph primitives:
  - PageRank, Katz Centrality, HITS, BFS, SSSP, WCC, K-core, Louvain
- See https://github.com/rapidsai/cugraph/tree/branch-22.04/cpp/include/cugraph/prims for the complete list of primitives

GRAPH PRIMITIVES

Examples

$$PageRank(v) = \sum_{n_i \in N_{in}(v)} \frac{PageRank(n_i) * w * alpha}{OutDegree(n_i)} + \frac{1 - alpha}{V}$$

$\sum_{n_i \in N_{in}(v)}$

$\frac{PageRank(n_i)}{OutDegree(n_i)}$

```
copy_v_transform_reduce_in_nbr(
  handle,
  pull_graph_view,
  edge_partition_src_pageranks.device_view(),
  dummy_properties_t<vertex_t>{}.device_view(),
  [alpha] __device__(vertex_t, vertex_t, weight_t w, auto src_val, auto) {
    return src_val * w * alpha;
  },
  unvarying_part,
  pageranks);
```

$\frac{1 - alpha}{V}$

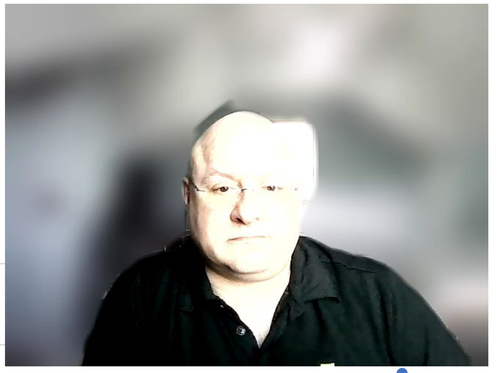
$copy \sum_{n_i \in N_{in}(v)} to\ pageranks$

* The core part to implement PageRank for 1000+ GPUs.
* You can tweak this code to implement analytics with similar patterns (e.g. Katz centrality, HITS, or something you invented).

Look for upcoming BLOG Series

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

- Customer in FSI, ISV, Retail, and Cyber can easily have 10s of millions of customers (nodes), producing trillions of transactions (edges).
- 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

Comparison

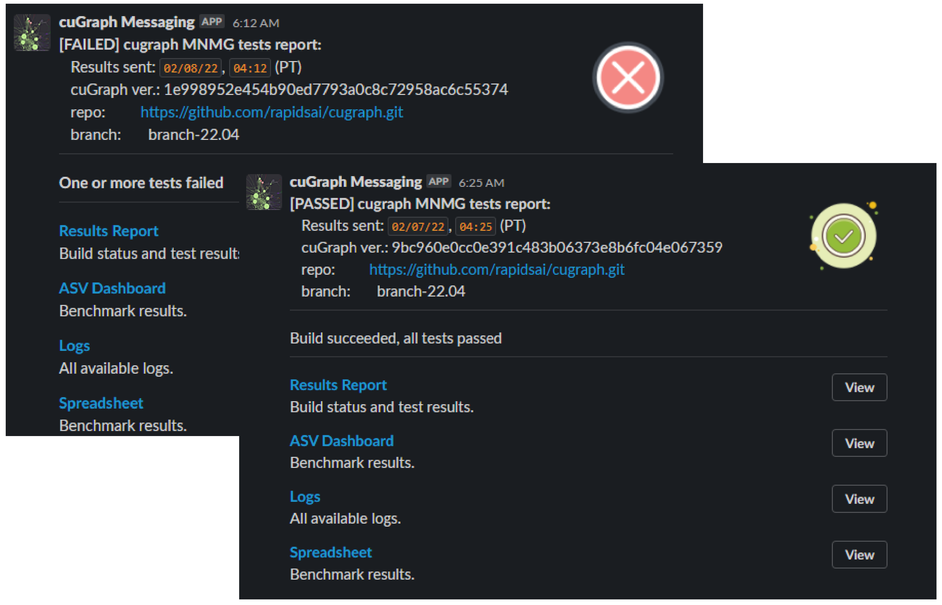
One Trillion Edges: Graph Processing at Facebook-Scale paper

"…. we were able to execute PageRank on over a trillion social connections in less than 3 minutes per iteration with only 200 machines."

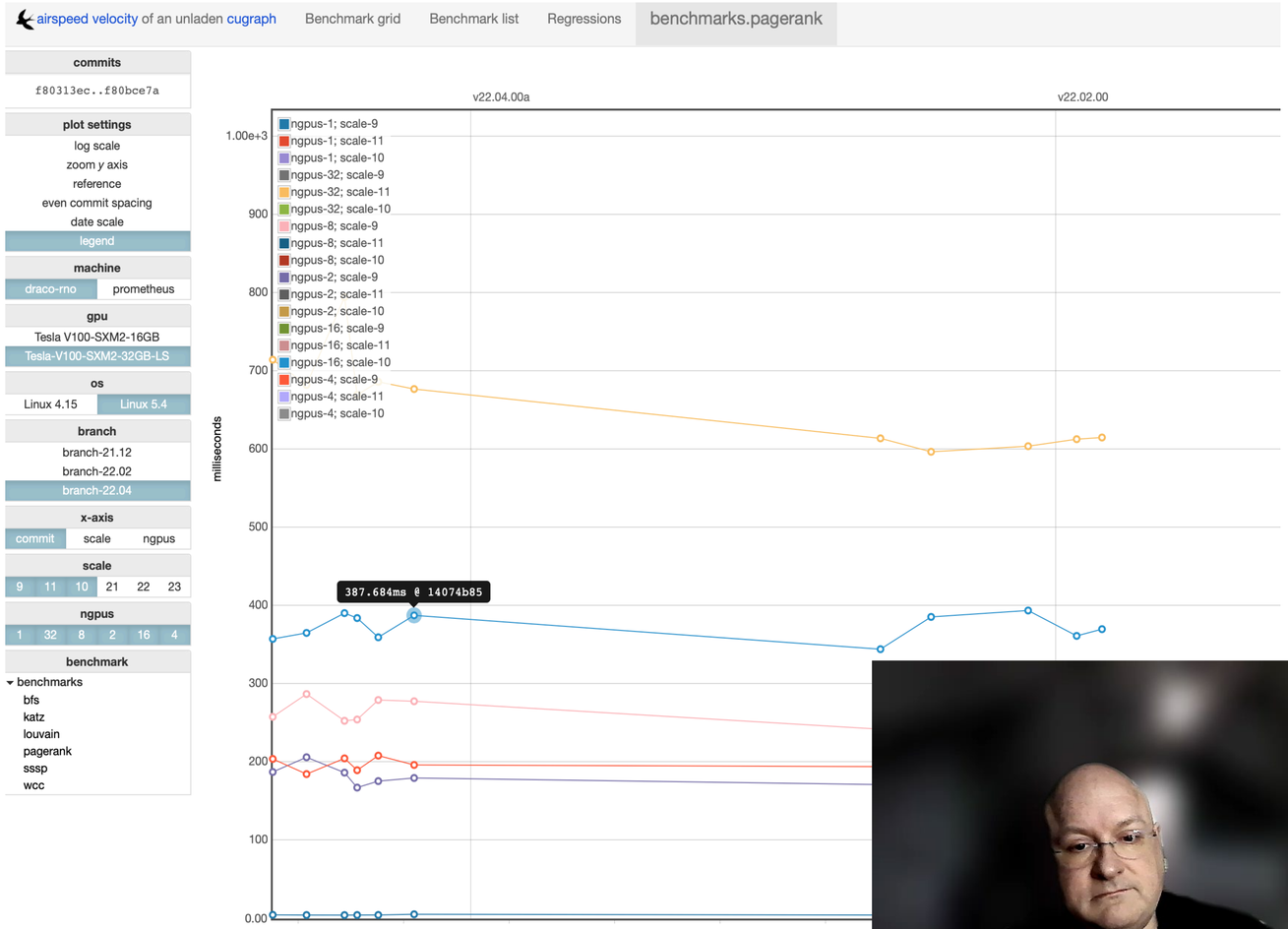
AUTOMATED MNMG TESTING AND BENCHMARKING USING THE PYTHON API

cuGraph Scaling to Massive Scale, tested nightly

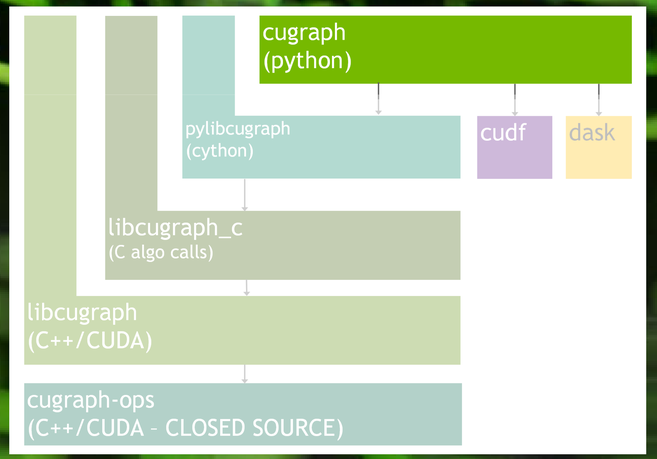
- Running on cluster across 32 GPUs, nightly or on-demand
- Shared MNMG test/benchmark infra is portable across similar cluster configurations
  - Open source infra: https://github.com/rapidsai/multi-gpu-tools
- Includes both MNMG functional testing and benchmark runs
- Reports generated and shared with team over Slack
  - Updates a shared database using asvdb to leverage ASV frontend for interactive plots
  - Various reports can be generated from asvdb API: google sheets, CSV, etc.



PageRank performance over several commits



Property Graphs

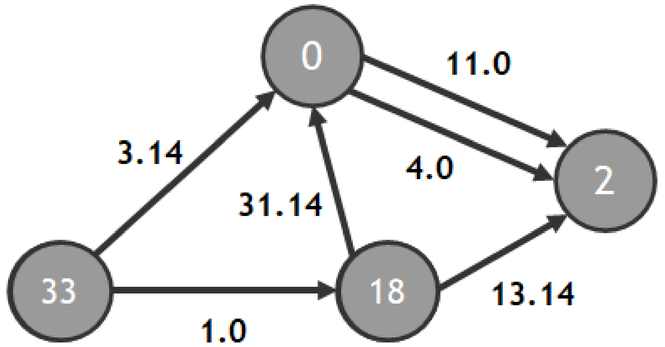


PROPERTY GRAPH

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 MODEL

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

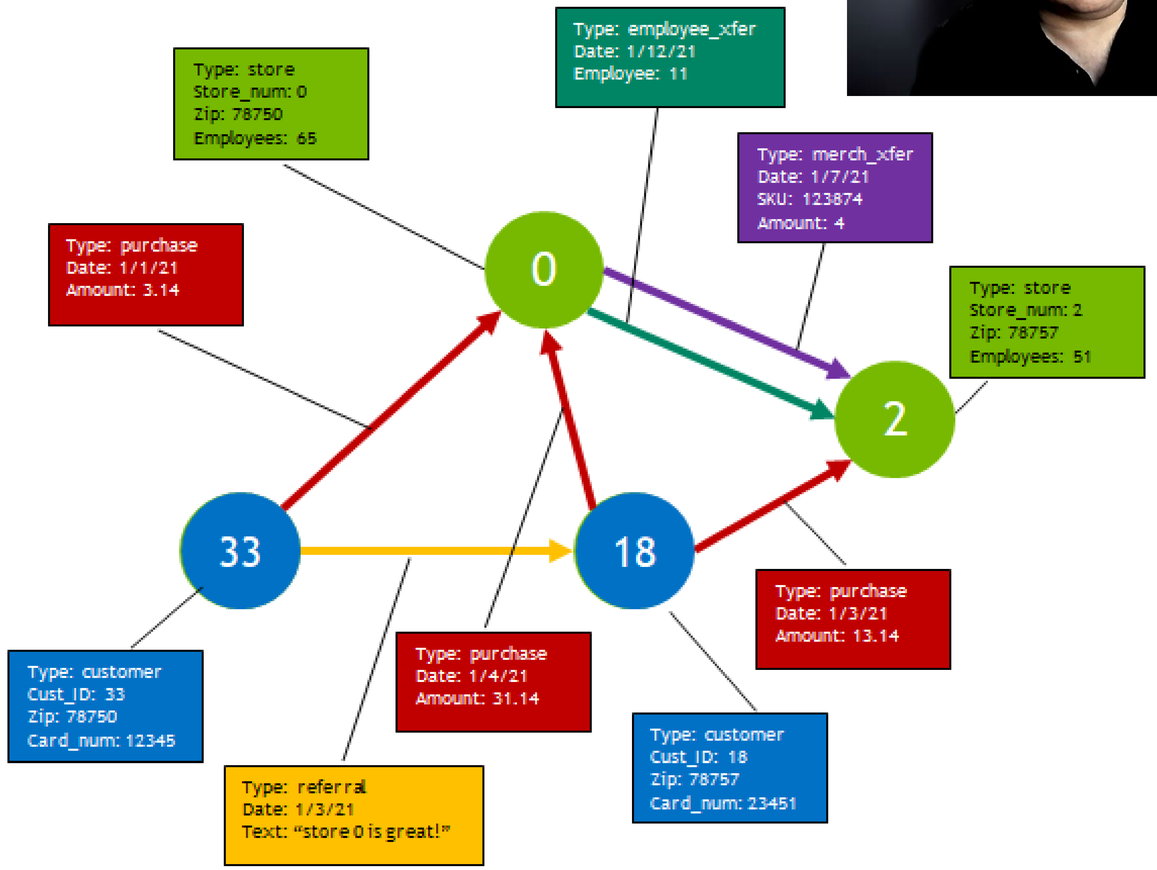
Cust_ID_1	Cust_ID_2	Date	Text
33	18	1/3/21	“store 0 is great!”

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



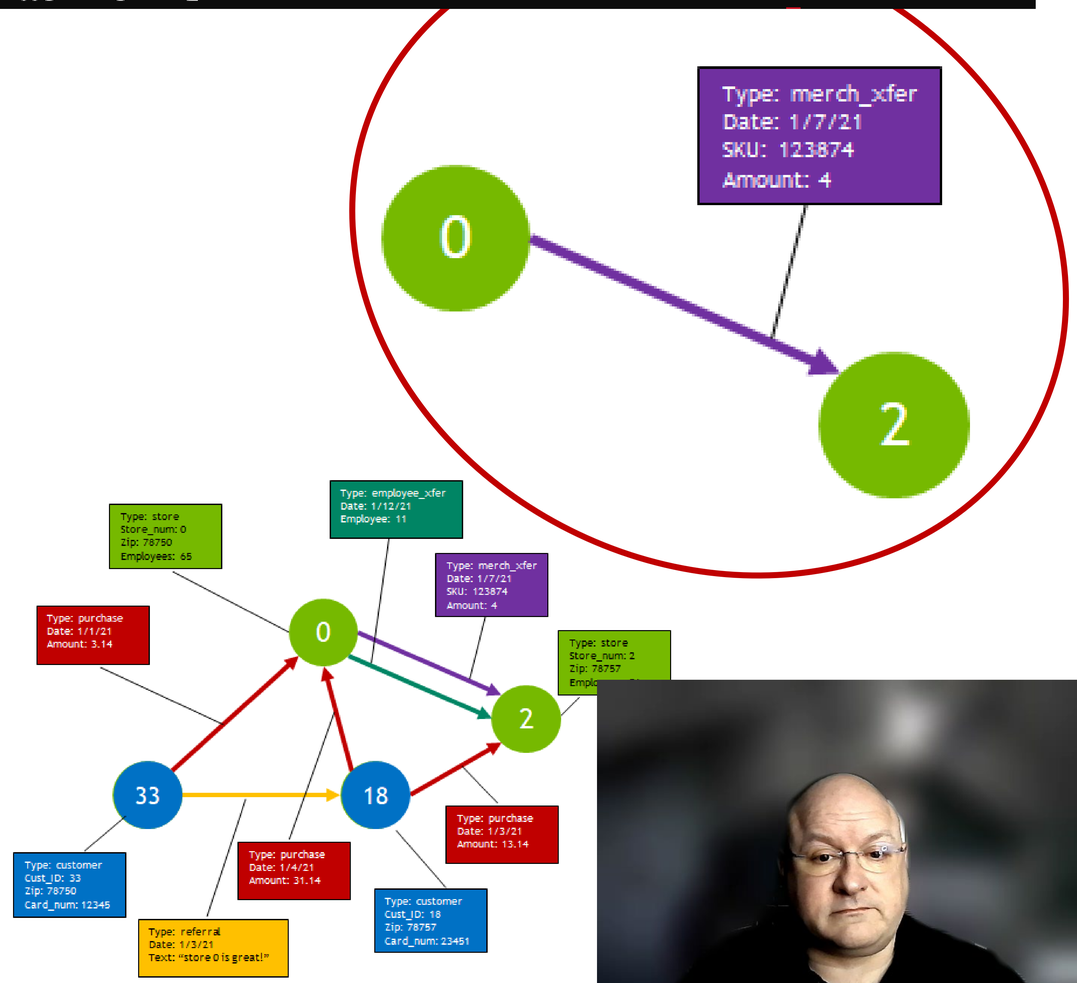
NVIDIA

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
```

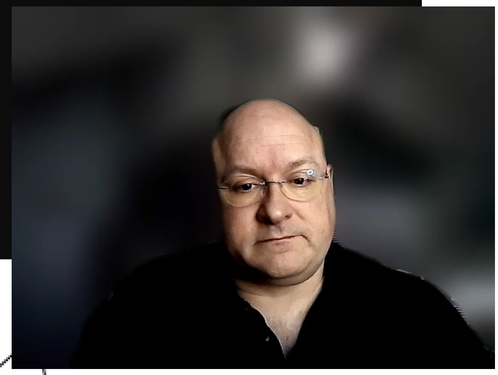


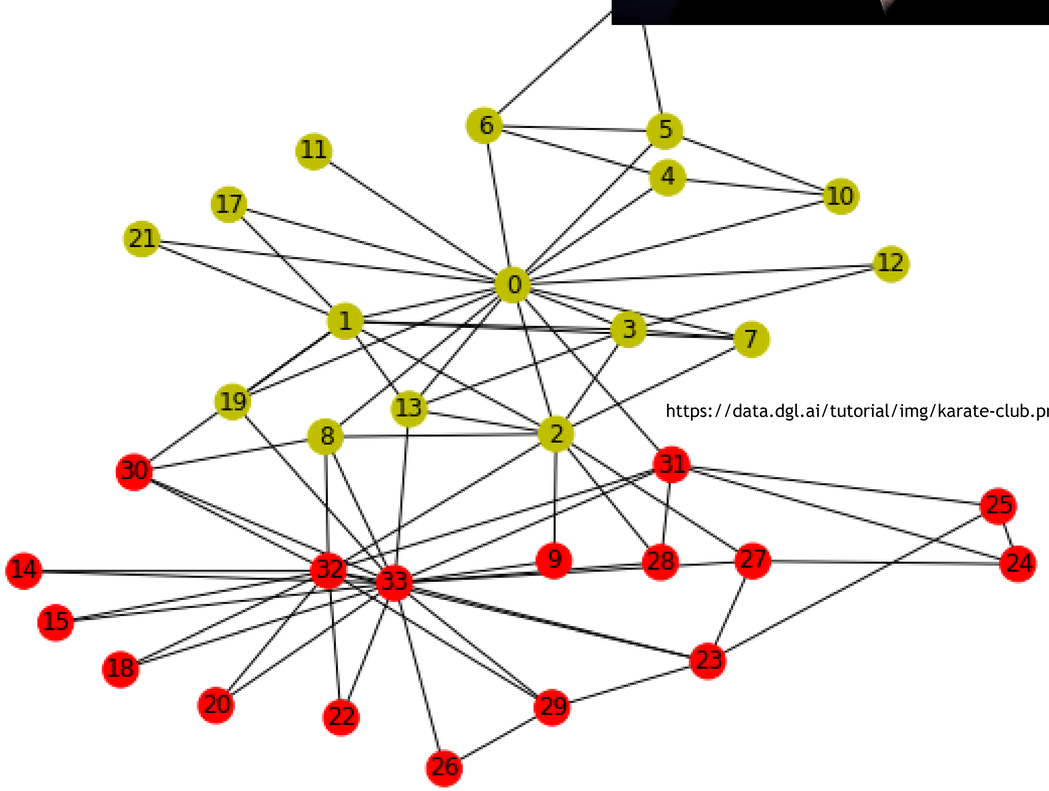
PROPERTY GRAPH

- 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
```



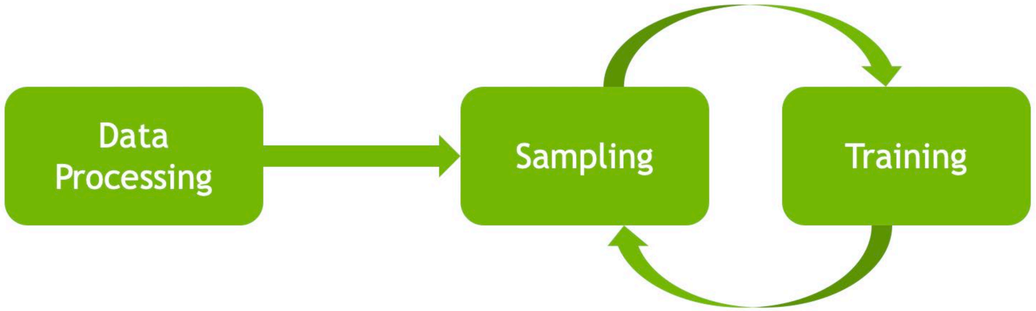


Look for BLOG on Property Graphs in April 2022

GNN Support

GNN WITH CUGRAPH AND DGL

FILL THE GAP IN BETWEEN SAMPLING AND TRAINING



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

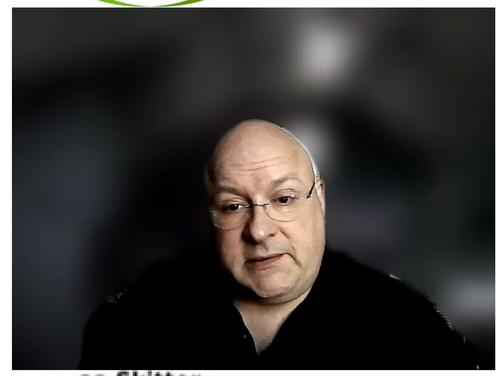
- **Motivation:**
  Existing frameworks do sampling on CPUs and train on GPUs. it is time-consuming due to slow memory bandwidth and copies between host and device.
- Avoid moving data (number of epochs X batch) between GPUs and CPUs in the training loop
- Sampling and training has closer relationship than data processing, has to be GPU accelerated as well

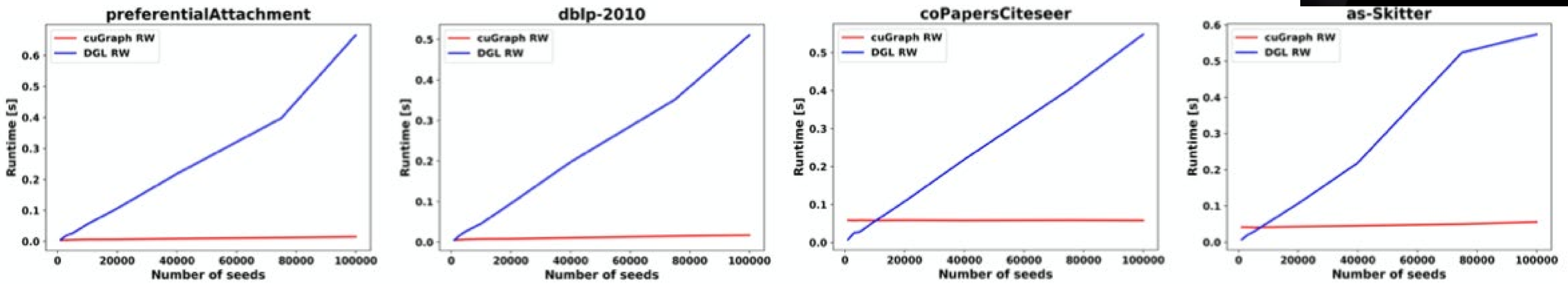
GPU ACCELERATED RANDOM WALK

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 in batches
  - Egonet
  - Random Walk
  - Node2Vec
  - Neighborhood

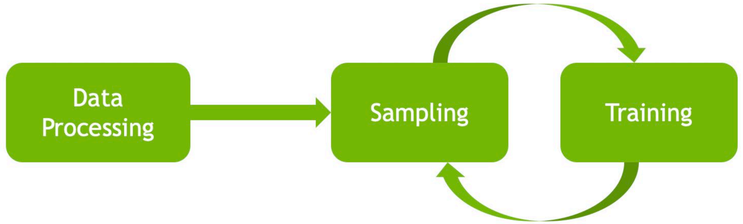




cuGraph RandomWalk vs DGL RandomWalk with various number of seeds

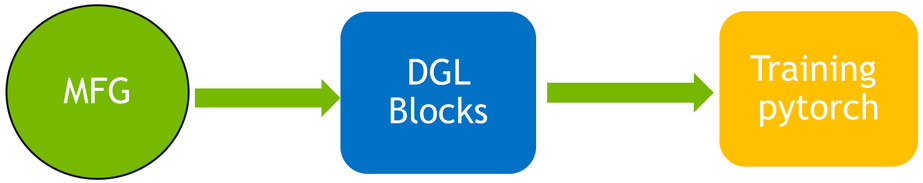
ROLE OF CUGRAPH-OPS

- 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

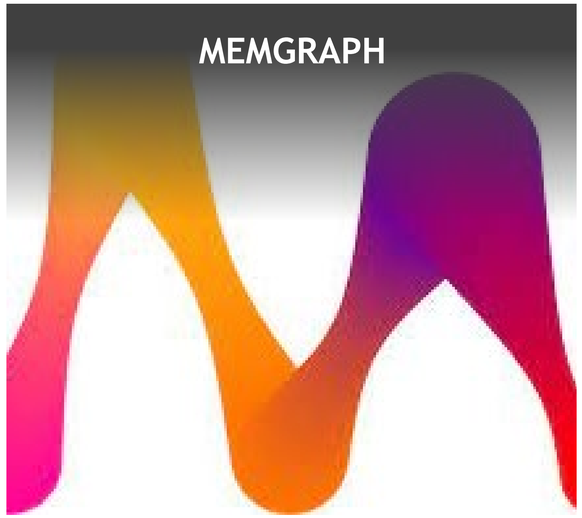


Note: Currently we support DGL, will also support PyG in the future.

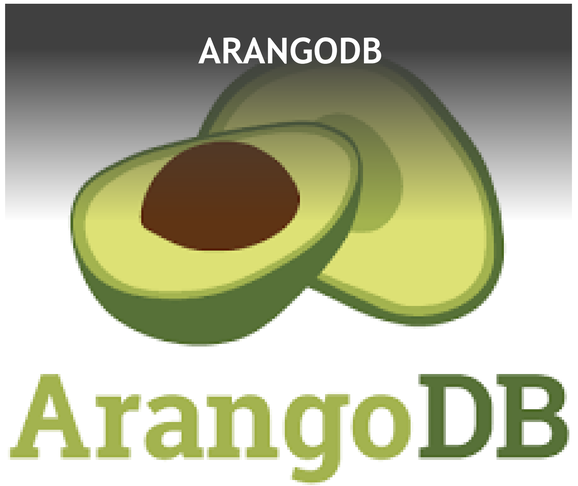
FUTURE WORK

Great to connect with NeptuneML for graph analytics and accelerated DGL

ENGAGEMENTS



cuGraph added in
Memgraph Advanced Graph Extensions
(MAGE)
https://github.com/memgraph/mage



https://medium.com/nvidia-ai/how-to-deploy-almost-any-pytorch-geometric-model-on-nvidias-triton-inference-server-with-an-218d0c0c679c

Question?

Thank You!

If you like cuGraph, please give us a star on GitHub https://github.com/rapidsai/cugraph

Any issues, please file a GitHub issue: https://github.com/rapidsai/cugraph/issues

Connect with the Experts session on cuGraph – CWE4175