FRAME Workshop @ Euro-Par – August 25, 2026

Accelerating Dynamic Graph Clustering on GPU Architectures with cuGraph

Nelson Aloysio Reis de Almeida Passos[1,2], Emanuele Carlini[2], Salvatore Trani[2]

[1] University of Pisa, Dep. of Computer Science

[2] National Research Council (ISTI-CNR)

Outline

1. Problem
Introduction and objectives

2. Challenges
Current state of the art

3. Contributions
Current *(new)* state of the art

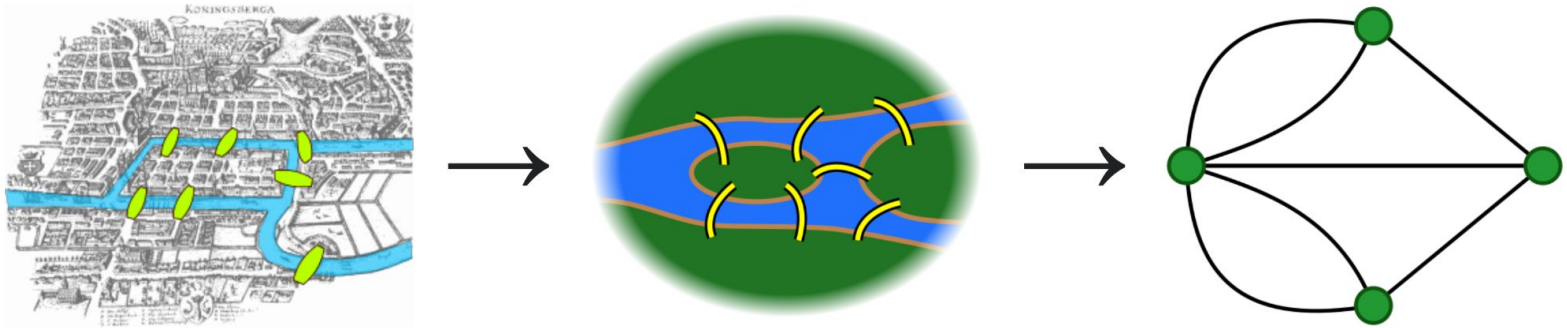
4. Demonstration
Quick examples

5. Future work
Where do we go from here?

1. Problem

Introduction and objectives

- Many real-world systems can be appropriately represented as **networks** or **graphs**.
- Elements are **nodes**, connections are **edges** – often mathematically constructed as **matrices**.



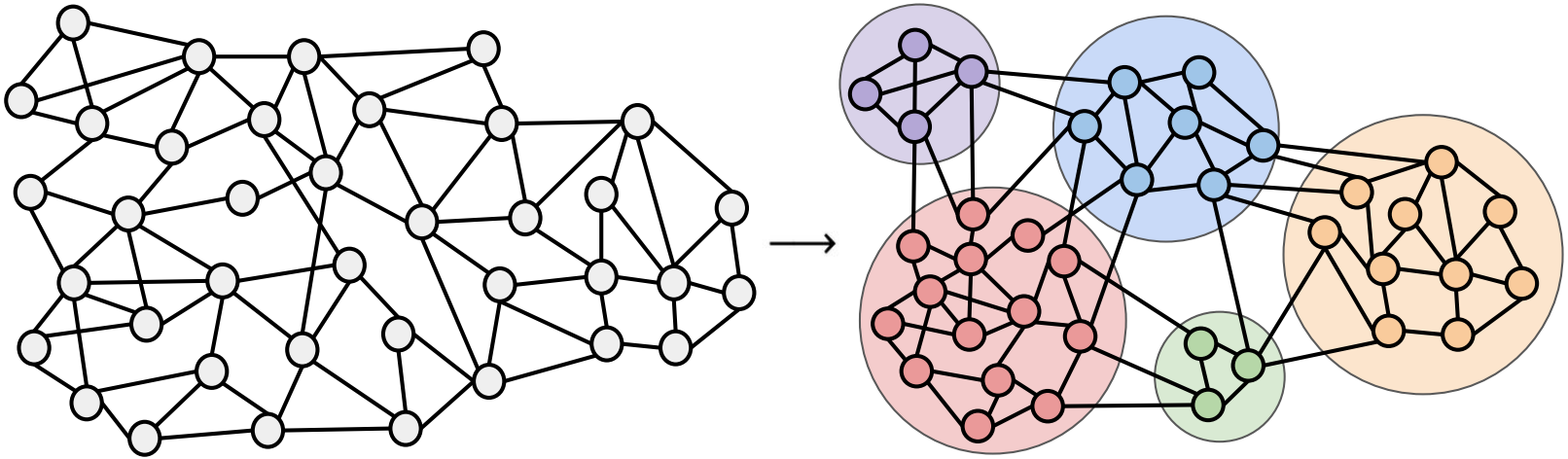
- But the systems we observe are rarely fixed *(static)* in time; instead, they *change* or *evolve.*
- A **temporal network** can be modeled as a **dynamic graph** – with *time-varying* nodes and edges.

[0] “Seven Bridges of Königsberg”. Wikipedia.

1. Problem

Introduction and objectives

- **Our problem: how to efficiently cluster nodes *(detect communities)* in dynamic graphs?**



1. Problem

Introduction and objectives

- **Our problem: how to efficiently cluster nodes *(detect communities)* in dynamic graphs?**

- **SOTA**: node clustering is performed with **algorithms** *(descriptive/inferential)* or **neural models**. (Neural models are usually preferred when the graph carries additional metadata information.)

  - Algorithms: descriptive *(e.g., **modularity**-based)* or inferential *(stochastic block models)*.

  - Neural models: graph neural networks *(which mostly fall into the descriptive realm)*.

- Several algorithms and models have been adapted **from static to dynamic** graph settings.

- We here focus on the widely employed **modularity** function [1, 2] for community detection, and implement an additional pathway for **spectral clustering** on different graph (matrix) **operators**.

[1] Newman, M (2006). Modularity and Community structure in networks. PNAS.
[2] Mucha, P. et al. (2010). Community Structure in Time-Dependent, Multiscale, and Multiplex Networks. Science.

2. Challenges

Current (new) state of the art

- Researchers analyzing **temporal networks** today face some **tough choices**:
  - **Principled versus tractable:** detectability-optimal methods are unfeasible or not scalable.
  - **CPU bottlenecks**: tools supporting the multislice *(temporal)* modularity model *(for truly dynamic community detection)* are CPU-only and hit severe scaling limits on most graphs.
  - **Static graph approximations**: forcing temporal graphs to flatten into *(re-weighted)* 2-D matrices, distorting the modularity score because it depends on the total sum of edges.
  - **Post-hoc processing**: workarounds that cluster each snapshot and then run additional algorithms *(so more heuristics)* to 'match' communities over time.
- We tackle the two issues above by employing a *supra-graph* strategy *without post-processing*.

2.5. Intermezzo

Graph operators and the Laplacian

- Often, graph operations (e.g., random walks and message passing) do not consider the raw adjacency matrix, but an *operationalized* version of it that is *optimal* for the given task at hand.

2.5. Intermezzo

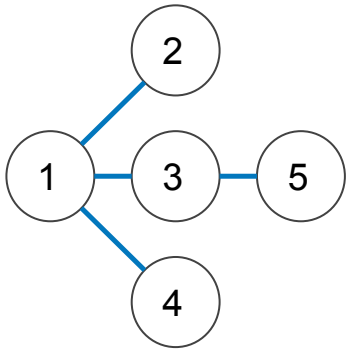
Graph operators and the Laplacian

- Often, graph operations (e.g., random walks and message passing) do not consider the raw adjacency matrix, but an *operationalized* version of it that is *optimal* for the given task at hand.

- Most commonly, the **normalized Laplacian operator** is used, including for **spectral clustering**.

2.5. Intermezzo

Graph operators and the Laplacian



G

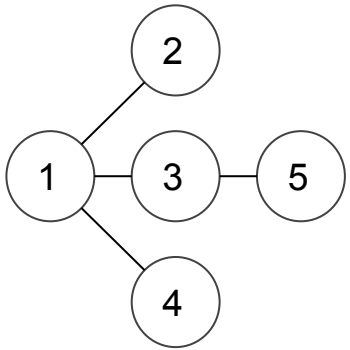
$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

$$\mathbf{D} = \begin{bmatrix} 3 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\rightarrow \mathbf{D} - \mathbf{A} = \mathbf{L} = \begin{bmatrix} 3 & -1 & -1 & -1 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 2 & 0 & -1 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \end{bmatrix}$$

2.5. Intermezzo

Graph operators and the normalized Laplacian



G

$$\mathbf{L}_{\text{norm}} := \begin{cases} 1 & \text{if } i = j \text{ and } \deg(v_i) \neq 0 \\ -\dfrac{1}{\sqrt{\deg(v_i)\,\deg(v_j)}} & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise.} \end{cases}$$

$$\mathbf{L} = \begin{bmatrix} 3 & -1 & -1 & -1 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 2 & 0 & -1 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \end{bmatrix} \rightarrow \mathbf{L}_{\text{norm}} = \begin{bmatrix} 1 & -1/\sqrt{3} & -1/\sqrt{6} & -1/\sqrt{3} & 0 \\ -1/\sqrt{3} & 1 & 0 & 0 & 0 \\ -1/\sqrt{6} & 0 & 1 & 0 & -1/\sqrt{2} \\ -1/\sqrt{3} & 0 & 0 & 1 & 0 \\ 0 & 0 & -1/\sqrt{2} & 0 & 1 \end{bmatrix}$$

2.5. Intermezzo

Graph operators and the Laplacian

- Often, graph operations (e.g., random walks and message passing) do not consider the raw adjacency matrix, but an *operationalized* version of it that is *optimal* for the given task at hand.

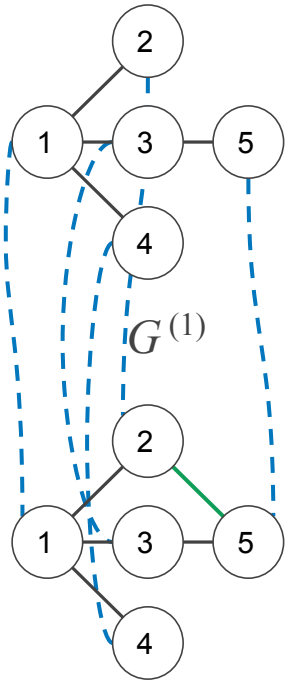
- Most commonly, the **normalized Laplacian operator** is used, including for **spectral clustering**.

- Likewise, for temporal graphs, we may build a similar operator by constructing a **supra-graph**.

2.5. Intermezzo

— Static edge — Temporal edge ··· Inter-slice coupling

Supra-graph (temporal) operators



$$\mathbf{A}^{(1)} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

$$\mathbf{A}^{(2)} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{A}^{\text{supra}} = \begin{pmatrix} \mathbf{A}^{(1)} & \omega\mathbf{I} & \cdots & \mathbf{0} \\ \omega\mathbf{I} & \mathbf{A}^{(2)} & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{A}^{(T)} \end{pmatrix}$$

$$\rightarrow \mathbf{L}_{\text{norm}} = \ldots$$

2.5. Intermezzo

Graph operators and the Laplacian

- Often, graph operations (e.g., random walks and message passing) do not consider the raw adjacency matrix, but an *operationalized* version of it that is *optimal* for the given task at hand.

- Most commonly, the **normalized Laplacian operator** is used, including for **spectral clustering**.

- Likewise, for temporal graphs, we may build a similar operator by constructing a **supra-graph**.

- However, the Laplacian is **not detectability-optimal** for node clustering tasks. ☹️

- The symmetric normalization of the operator introduces a $D^{½}$ scaling that requires an additional (heuristic) row-normalization step to correct for high node degree heterogeneity.

  (In practice, the graph's algebraic connectivity – Fiedler value – changes, so its **informative eigenvalues** are skewed in most scale-free graphs → real-world systems, e.g., airport traffic.)

- But other operators may be used that ~~are~~ *could* be optimal for temporal graph clustering…

2.5. Intermezzo

Supra-graph (temporal) operators

- The **temporal non-backtracking** (Hashimoto-type) operator is detectability-optimal... ☺️

$$\mathbf{B} = \begin{pmatrix} \lambda\mathbf{A}^{s} & -\lambda\mathbf{I} & \lambda\mathbf{A}^{s} & \mathbf{0} \\ \lambda(\mathbf{D}^{s}-\mathbf{I}) & \mathbf{0} & \lambda\mathbf{D}^{s} & \mathbf{0} \\ \eta\mathbf{A}^{t} & \mathbf{0} & \eta\mathbf{A}^{t} & -\eta\mathbf{I} \\ \eta\mathbf{D}^{t} & \mathbf{0} & \eta(\mathbf{D}^{t}-\mathbf{I}) & \mathbf{0} \end{pmatrix}$$

λ : temporal coupling
η : spatial coupling

- ...but requires an *asymmetric* sparse eigensolver, which to date is not implemented on GPUs. ☹️

- Its surrogate, the **Bethe-Hessian** $\mathbf{H}(r) = (r^2-1)\mathbf{I} - r\mathbf{A} + \mathbf{D}$, is also detectability-optimal and maps the *(informative, out-of-bulk)* eigenvalues of **B** to its negative eigenvalues... ☺️

- ...*but only if* the regularizer *r* correctly accounts for both λ and η (if *r* = 1 → **H** = **L**aplacian). ☹️

2.5. Intermezzo

Modularity optimization

- Modularity is a ‘quality’ function of the ‘goodness’ of a graph partitioning, which measures the proportion of edges within communities against the expected amount in a null (random) graph:

$$Q = \frac{1}{2m} \sum_{i,j} \left(A_{ij} - \gamma \frac{d_i^{\text{out}} \, d_j^{\text{in}}}{2m} \right) \delta(c_i, c_j)$$

i,j: nodes *d*: degree
m: total edges in graph
c: community/cluster
γ: resolution param

- It has since been adapted to **temporal** graphs as well, such as the **multislice modularity** variant:

$$Q_{\text{MS}} = \frac{1}{2\mu} \sum_{s,r} \sum_{i,j} \left[\left(A_{ij}^{(s)} - \gamma_s \frac{d_{is}^{\text{out}} \, d_{js}^{\text{in}}}{2m_s} \right) \delta_{sr} + \delta_{ij} \omega_{jsr} \right] \delta \left(c_i^{(s)}, c_j^{(r)} \right)$$

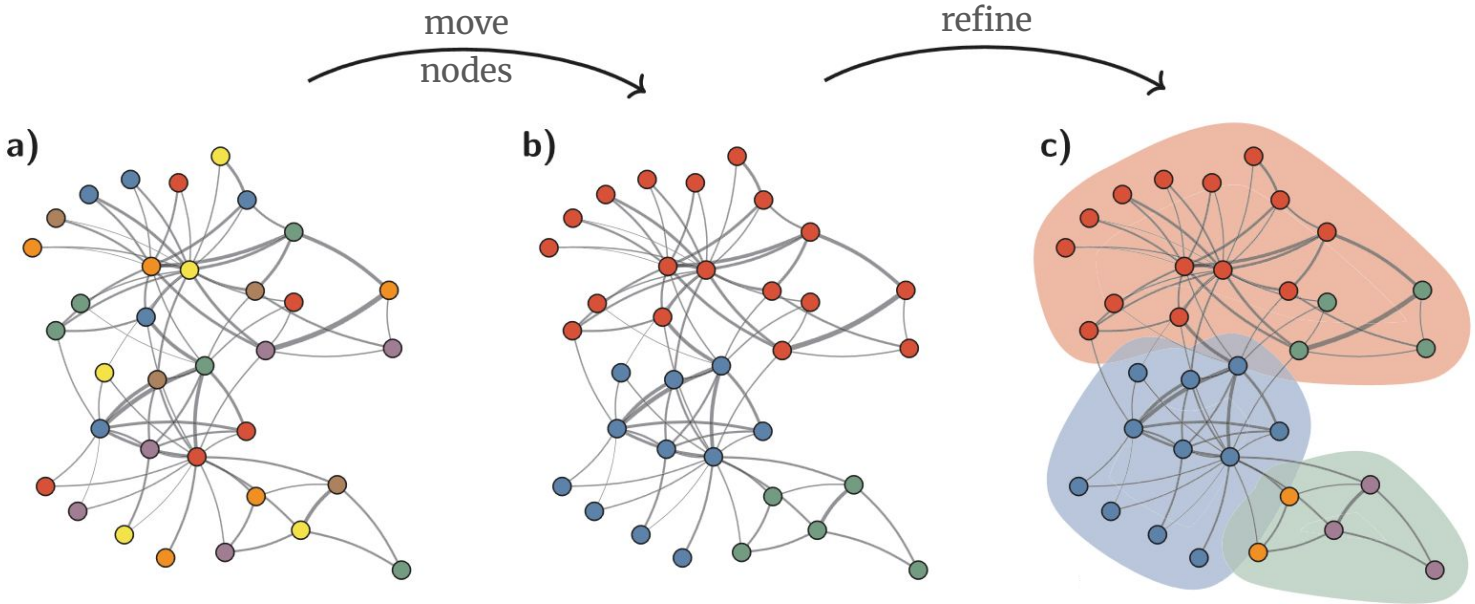
s,r: snapshots (slices)
μ: intra + inter-slice edges
ω: inter-slice weight param

- Multislice modularity employs a **per-slice** null model (vs. global). A **spectral** variant also exists.

2.5. Intermezzo

Modularity optimization with Leiden

- **Leiden** [3] is a refinement over **Louvain** [4] for 2-step greedy optimization of, e.g., modularity.

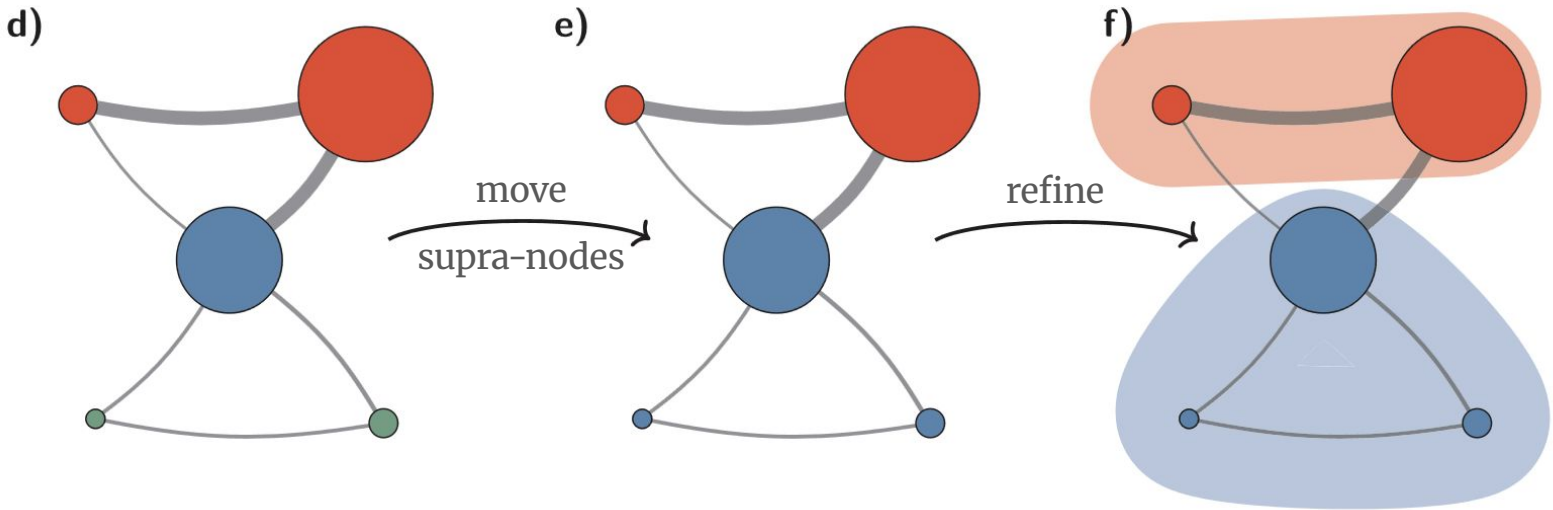


[3] Traag, V. et al. (2019). From Louvain to Leiden: guaranteeing well-connected community. Scientific Reports.
[4] Blondel, V. et al. (2008). Fast unfolding of community in large networks. Journal of Statistical Mechanics: Theory and Experiment.

2.5. Intermezzo

Modularity optimization with Leiden

- **Leiden** [3] is a refinement over **Louvain** [4] for 2-step greedy optimization of, e.g., modularity.

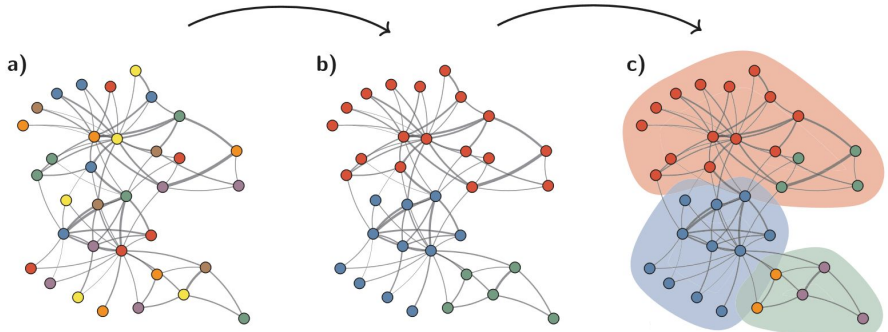


[3] Traag, V. et al. (2019). From Louvain to Leiden: guaranteeing well-connected community. Scientific Reports.
[4] Blondel, V. et al. (2008). Fast unfolding of community in large networks. Journal of Statistical Mechanics: Theory and Experiment.

2.5. Intermezzo

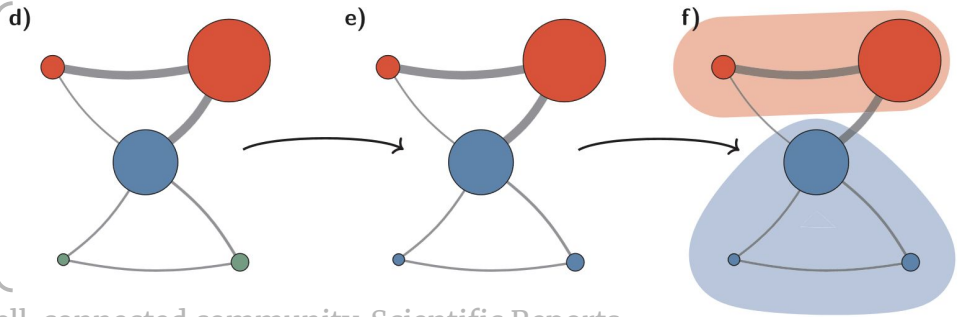
Modularity optimization with Leiden

- **Leiden** [3] is a refinement over **Louvain** [4] for 2-step greedy optimization of, e.g., modularity.



Level 1 (*node aggregation and refinement*)

Level 2 (*supra-node aggregation and refinement*)

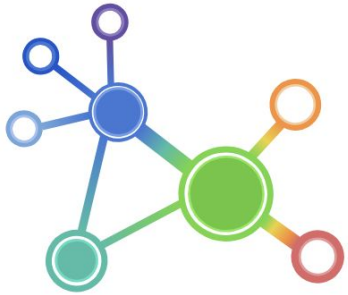


[3] Traag, V. et al. (2019). From Louvain to Leiden: guaranteeing well-connected community. Scientific Reports.
[4] Blondel, V. et al. (2008). Fast unfolding of community in large networks. Journal of Statistical Mechanics: Theory and Experiment.

3. Contributions

Current (new) state of the art

- We built two pathways for dynamic graph clustering on the **NVIDIA RAPIDS** ecosystem:
  - **Spectral clustering**: eigendecomposition and clustering with k-means (CuPy + CuML).
  - **Modularity optimization:** supra-graph optimization with the Leiden algorithm (cuGraph).
- We show supra-graph optimization allows for a proxy approximation of temporal modularity under a global null model, and discuss its limitations versus spectral and multislice modularity.
- We evaluate our method on several datasets, achieving up to **~978x** speedup in one instance.
- Code is open source and integrated in the **NetworkX-Temporal** software library.



NetworkX Temporal

NetworkX-Temporal is a Python package for the creation, manipulation, and analysis of the structure and dynamics of temporal graphs — extending NetworkX with support for time-varying relational data.

Install from PyPI:

```
$ pip install networkx-temporal
```

🔗 networkx-temporal.org

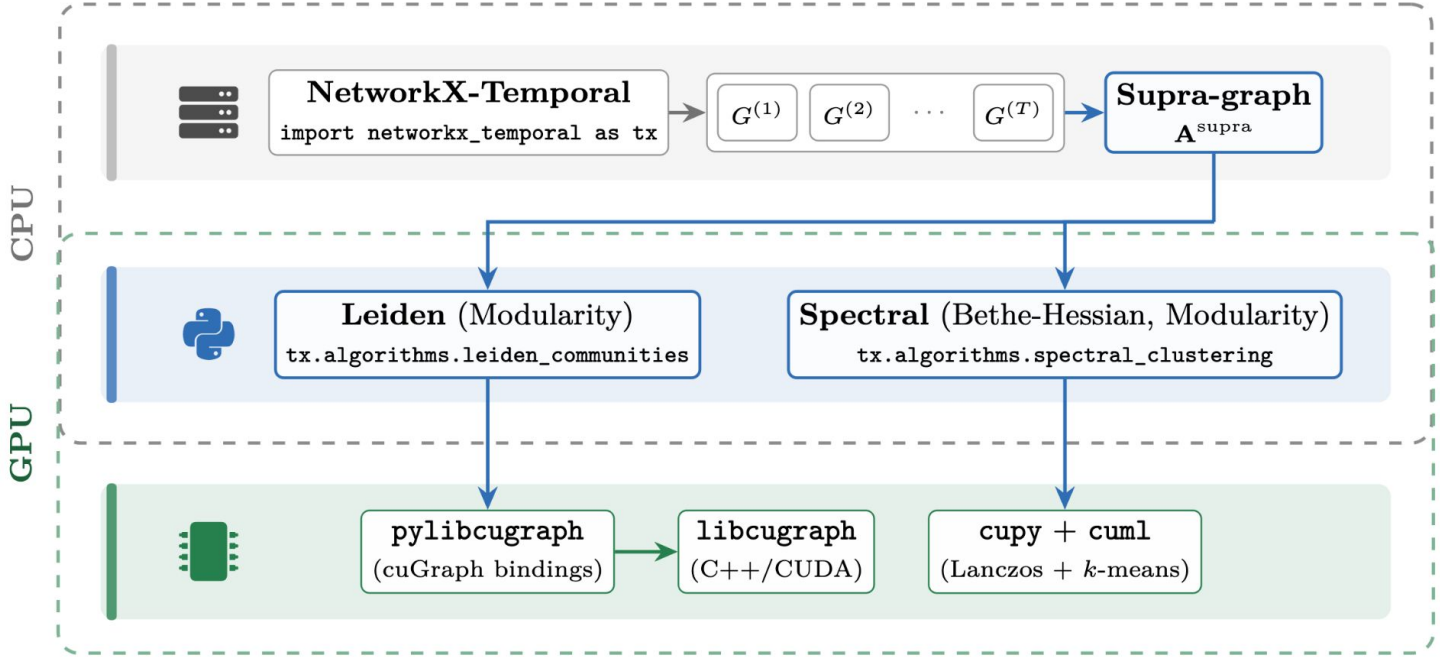


Fig. 1. NetworkX-Temporal integration. Temporal graph snapshots $G^{(t)}$ are assembled into a supra-graph and dispatched through one of two GPU-accelerated paths: multislice modularity via Leiden optimization, or spectral clustering of a (temporal) Bethe-Hessian or modularity matrix. All methods execute on the RAPIDS ecosystem and return per-snapshot community assignments, allowing zero-code-change execution and device selection by setting an environment variable. In blue: our contributions.

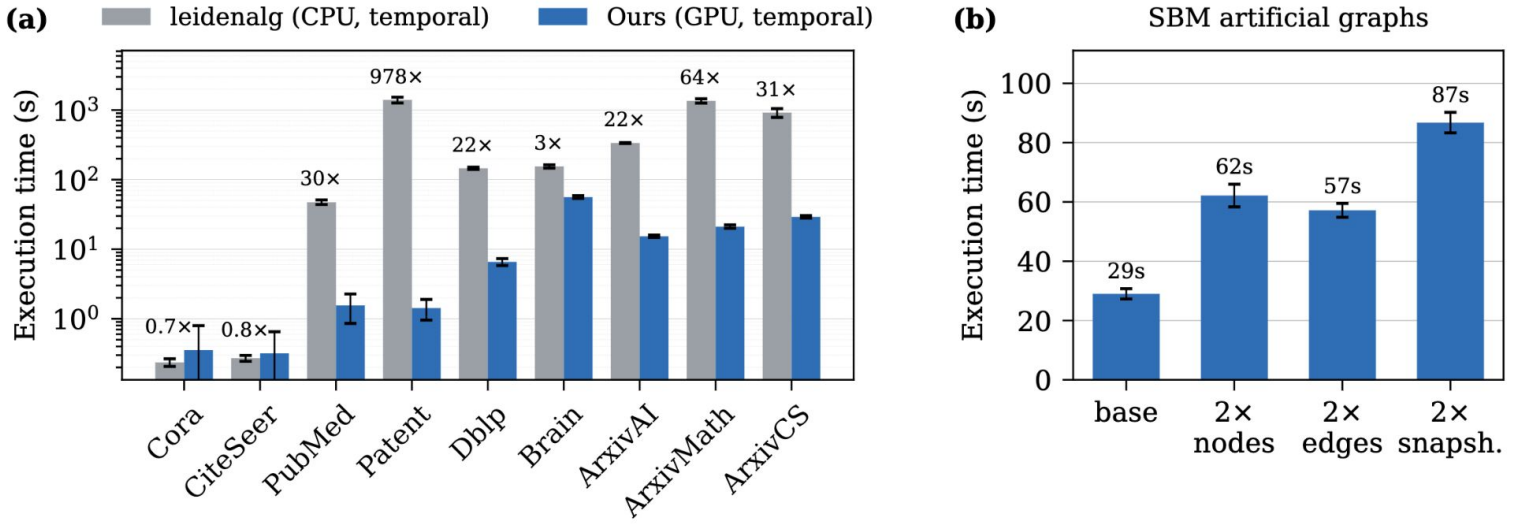


Fig. 2. Runtime evaluation. (a) Execution time of the GPU-based Leiden backend [24] against its CPU-bound baseline across real-world datasets (log scale). (b) Effect on runtime of doubling each generation parameter (nodes, edges, snapshots) over synthetic SBM instances. Algorithm runs were limited to two iterations only (`leidenalg` [43] default) to ensure fair comparisons. Results reported over five runs, including host-device transfer and kernel launch overheads; error bars show standard deviation. Experiments conducted on an Intel Xeon Gold 6330 (CPU) and an NVIDIA A100 80GB (GPU).

Dataset	$\lvert\mathcal{V}\rvert$	$\Sigma(\mathcal{V})$	$\lvert\mathcal{E}\rvert$	$\Sigma(\mathcal{E})$	$\lvert T\rvert$	$\lvert C\rvert$
CiteSeer† [46]	3 279	3 279	4 552	4 552	1	6
Cora† [46]	2 708	2 708	5 278	5 278	1	7
Brain [33]	5 000	60 000	878 207	947 744	12	10
Dblp [49]	28 085	101 797	153 822	222 165	27	10
Patent [14]	12 214	41 529	41 916	41 916	891	6
PubMed [27]	19 717	37 003	44 324	44 324	42	3
School [19]	327	309 006	7 004	188 508	7 375	9
ArxivAI [17]	69 854	142 316	696 819	696 819	27	5
ArxivCS [17]	169 343	374 433	1 157 799	1 157 799	29	40
ArxivLarge‡ [17]	1 324 064	4 998 391	13 649 351	13 649 351	40	172
ArxivMath [17]	270 013	614 578	783 165	783 165	31	31
SBM [29]	50 000	250 000	1 250 539	1 251 057	5	10
SBM-N [29]	100 000	500 000	2 500 515	2 501 000	5	10
SBM-E [29]	50 000	250 000	2 498 067	2 500 045	5	10
SBM-T [29]	50 000	500 000	2 495 376	2 497 636	10	10

Table 1. Dataset properties. $\lvert\mathcal{V}\rvert$ and $\lvert\mathcal{E}\rvert$ denote unique nodes and edges; $\Sigma(\mathcal{V})$ and $\Sigma(\mathcal{E})$, the sum of nodes and edges over time, i.e., counting each occurrence across snapshots; $\lvert T\rvert$ and $\lvert C\rvert$ are the number of snapshots and communities; † marks static graphs used as single-snapshot baselines; and ‡, the largest dataset, where a single CPU run allowed to exceed the time budget completed in ≈6 h, versus ≈10 min on GPU [24].

4. Demonstration

Quick examples

GPU acceleration

Examples in this guide are also available as an interactive Jupyter notebook. Open in Colab

GPU acceleration allows to significantly speed up computations of algorithms for temporal graphs on specialized hardware. This guide demonstrates how to enable GPU acceleration in NetworkX-Temporal and provides some examples of its usage for community detection algorithms.

Accelerating temporal graph algorithms

Some algorithms for temporal networks can be computationally intensive, especially for large-scale networks. To address this, NetworkX-Temporal integrates some GPU-accelerated algorithm implementations using the open-source libraries CuPy, cuGraph, and cuML, substantially reducing their computation time. This guide illustrates some commomn usage examples.

4. Demonstration

Quick examples: Library loading

- Zero-code-change GPU acceleration with the environment flag **`NX_GPU_AUTOCONFIG=1`**:

```
>>> import networkx_temporal as tx
>>> tx.is_gpu_enabled

True
```

- Alternatively, function calls accept a **`device`** parameter that may be set to **`'cpu'`** or **`'gpu'`**.

- *NVIDIA CUDA devices supported as first-class citizen; support for AMD ROCm is experimental.*

4. Demonstration

Quick examples: Spectral clustering – Temporal graph

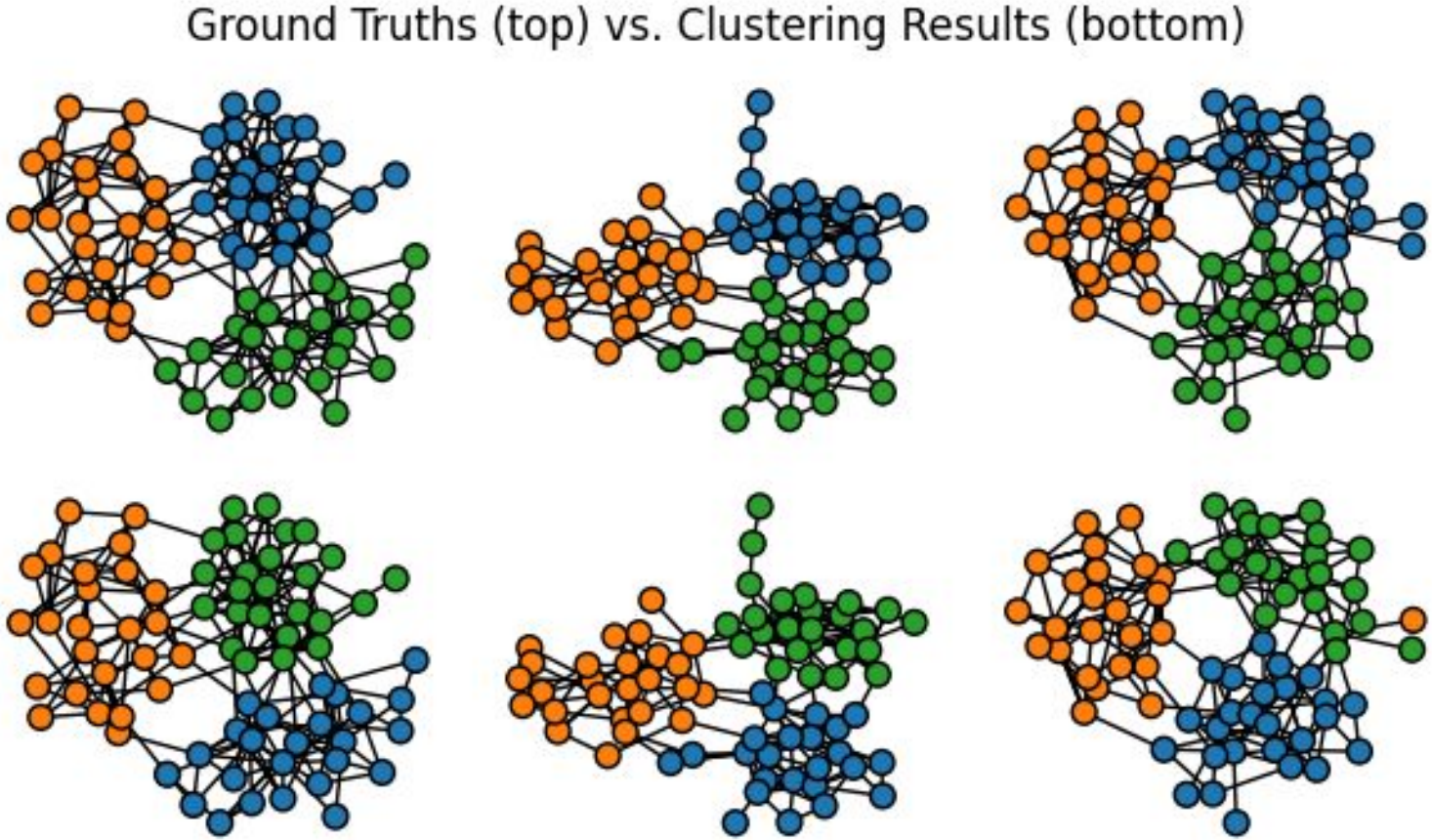
- **Spectral** clustering of the **Laplacian** of a temporal SBM graph example (toy dataset):

```
>>> TG = tx.example_sbm_graph()
>>> print(TG)

TemporalDiGraph (t=3) with 75 nodes and 225 edges
```

```
>>> y_pred = tx.spectral_clustering(TG, k=3, operator="laplacian")
```

- Let's visualize the results with the ground truths side by side *(code for viz available online)*.

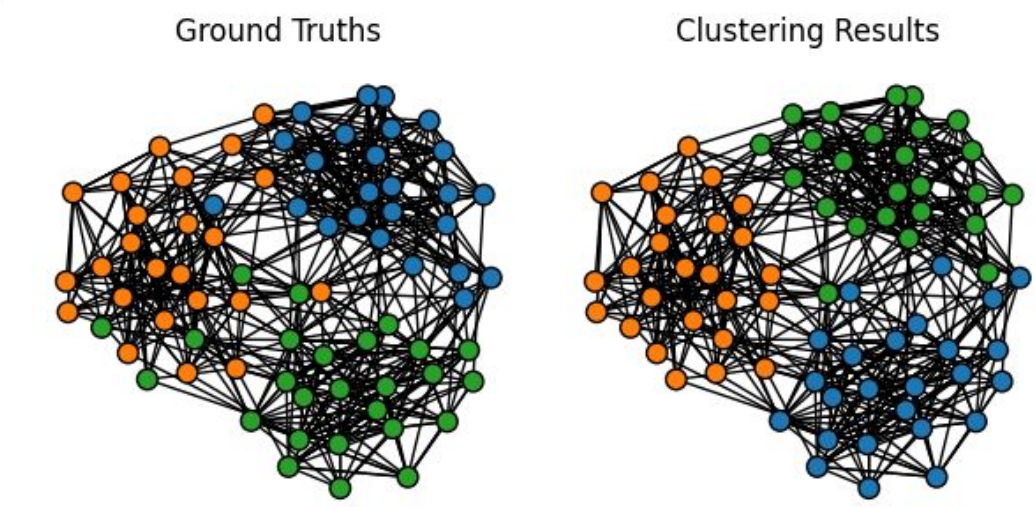

Ground Truths (top) vs. Clustering Results (bottom)

4. Demonstration

Quick examples: Spectral clustering – Static graph

- Support is also implemented for **static** graphs – a special case of temporal graphs afterall (T=1):

```
>>> G = TG.to_static()
>>> y_pred = tx.spectral_clustering(G, k=3, operator="laplacian")
```



4. Demonstration

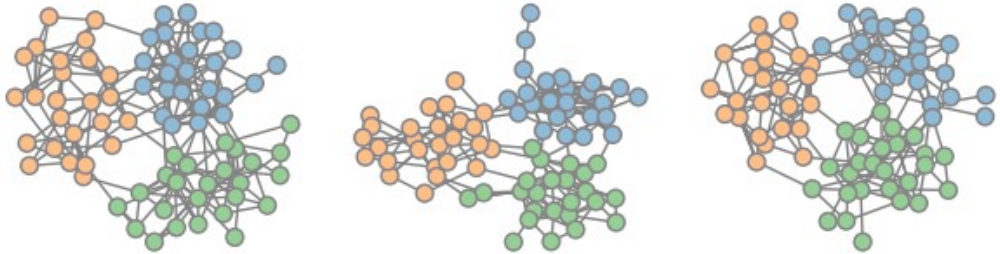
Quick examples: Modularity optimization

- Alternatively, to perform **heuristic** optimization of modularity with the Leiden algorithm:

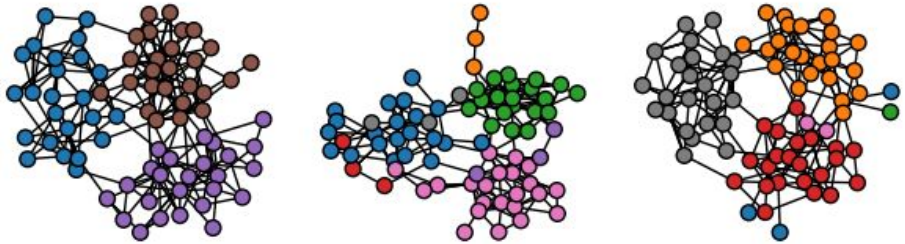
```
>>> y_pred = tx.leiden_communities(TG)
```

- The function employs different backends for **CPU** and **GPU** devices (`leidenalg`/`cugraph`).
- Our experimental results considered the **supra-graph** with the **global null model** of modularity.
- But speedups come with a **caveat**: this proxy optimization shifts the optimal graph partitioning.
- The modularity null model depends on the number of edges *m* in the graph, which is altered by the addition of inter-slice couplings among temporal node copies. Results show it clearly:

Ground Truths (Temporal)



GPU (Supra)



4. Demonstration

Quick examples: Modularity optimization

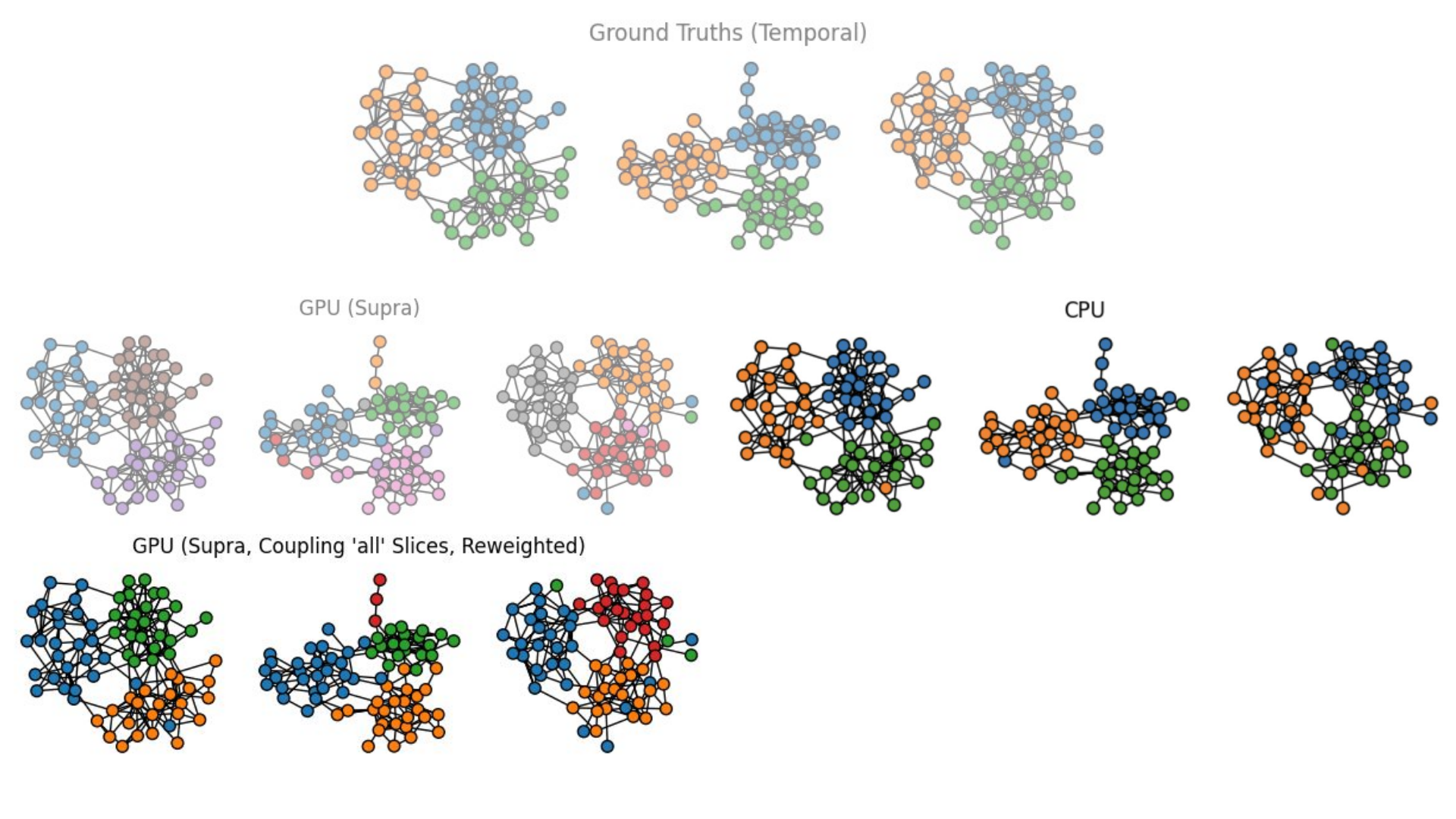
- Alternatively, it is possible to re-weight the adjacencies of the graph to **compensate** for the increased graph size. ☺️
- Coupling all nodes in all slices and normalizing edge weights based on their inverse distances allow to better approximate the ground truth results, but the model still **underperforms**. ☹️
- Let's visualize it against the results from the CPU baseline:

```
>>> T, omega = len(TG), 1.0
>>>
>>> # Normalize and symmetrize inter-slice couplings based on inverse distance weighting.
>>> raw = {(i, j): 1 / abs(i - j) for i in range(T) for j in range(T) if i != j}
>>> row = {i: sum(raw[(i, k)] for k in range(T) if k != i) for i in range(T)}
>>> w = {(i, j): omega * v * 0.5 * (1 / row[i] + 1 / row[j]) for (i, j), v in raw.items()}
>>>
>>> # Build supra-graph connecting nodes across 'all' time slices.
>>> adj, offsets = tx.to_supra_adjacency_matrix(
>>>     TG, interslice_couple="all", interslice_weights=w, return_offsets=True)
>>>
>>> # Supra-graph (static) modularity optimization.
>>> G_supra_ = tx.from_scipy(adj)
>>> y_gpu_G_supra_ = tx.leiden_communities(G_supra_, max_iter=max_iter, device="gpu")
>>> y_gpu_G_supra_ = [y_gpu_G_supra_[offsets[t]:offsets[t] + len(TG[t])]
>>>                   for t in range(len(TG))]
>>>
>>> print(f"Graph:\n{G}\n",
>>>       f"Supra-Graph:\n{G_supra}\n",
>>>       f"Supra-Graph (Coupling All Slices):\n{G_supra_}", sep="\n")

Graph:
MultiGraph with 75 nodes and 589 edges

Supra-Graph:
MultiGraph with 225 nodes and 690 edges

Supra-Graph (Coupling All Slices):
MultiGraph with 225 nodes and 765 edges
```


Ground Truths (Temporal)
GPU (Supra)
CPU
GPU (Supra, Coupling 'all' Slices, Reweighted)

5. ~~This~~ Future work

Where do we go from here?

- We tackled the two issues below by employing a *supra-graph* strategy *without post-processing*.
  - **Principled versus tractable:** detectability-optimal methods are unfeasible or not scalable.
  - **CPU bottlenecks**: tools supporting the multislice *(temporal)* modularity model *(for truly dynamic community detection)* are CPU-only and hit severe scaling limits on most graphs.
  - ~~**Static graph approximations**: forcing temporal graphs to flatten into *(re-weighted)* 2-D matrices, distorting the modularity score because it depends on the total sum of edges.~~
  - ~~**Post-hoc processing**: workarounds that cluster each snapshot and then run additional algorithms *(so more heuristics)* to 'match' communities over time.~~
- Our **next work** tackles the **one issue** above: GPU **temporal** modularity optimization in sparse CuPy (AMD/NVIDIA GPUs). We also leave benchmarking our spectral pathway for future work.

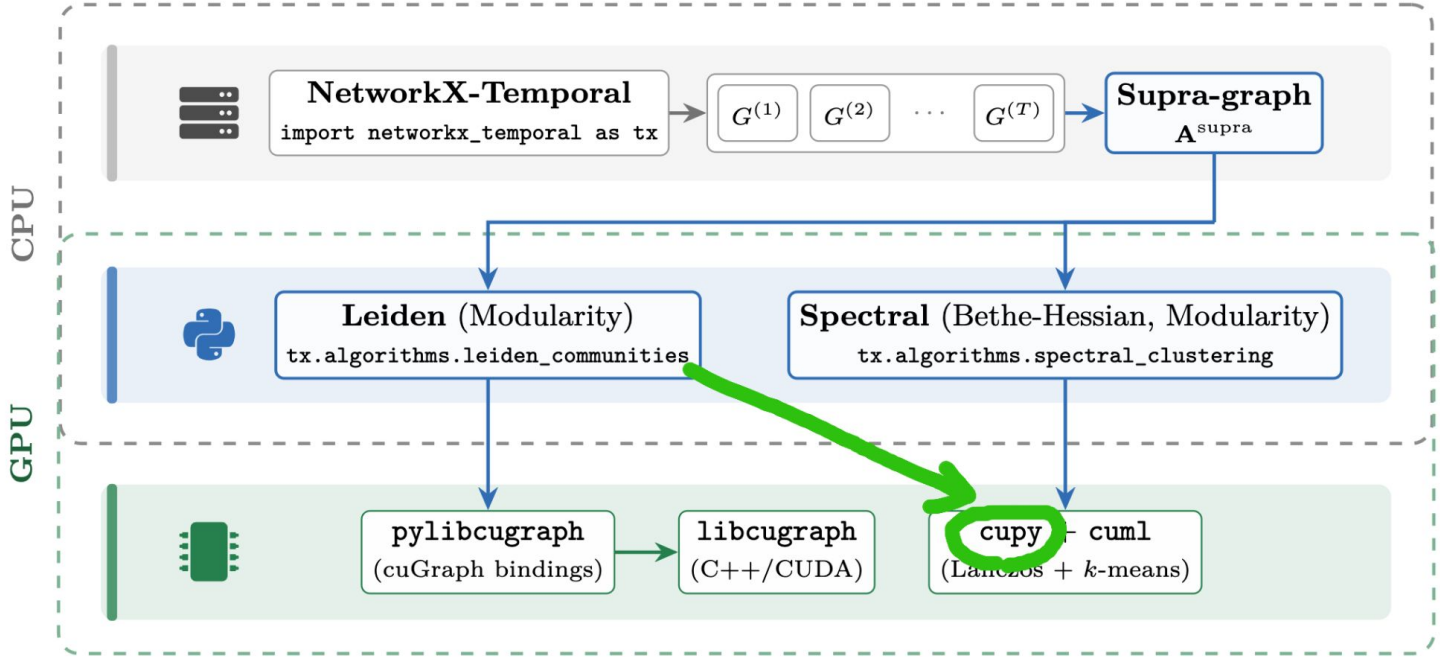
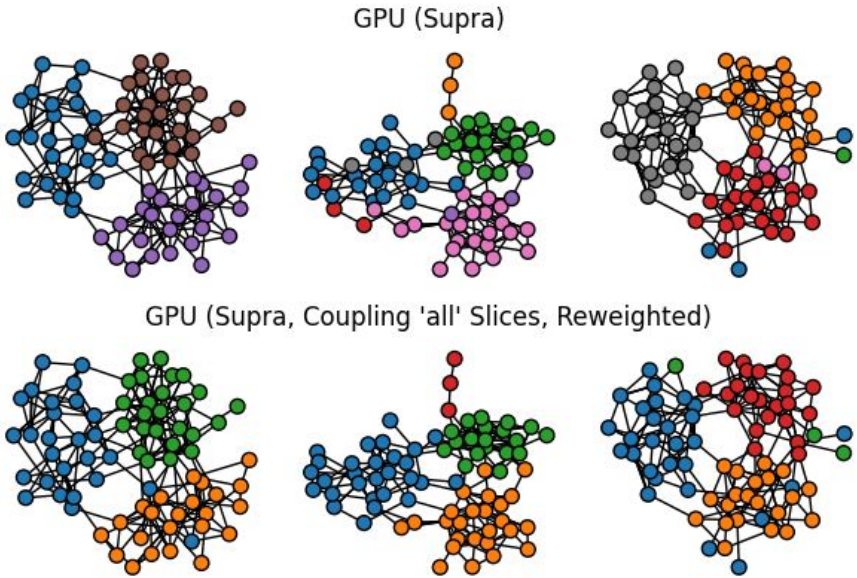
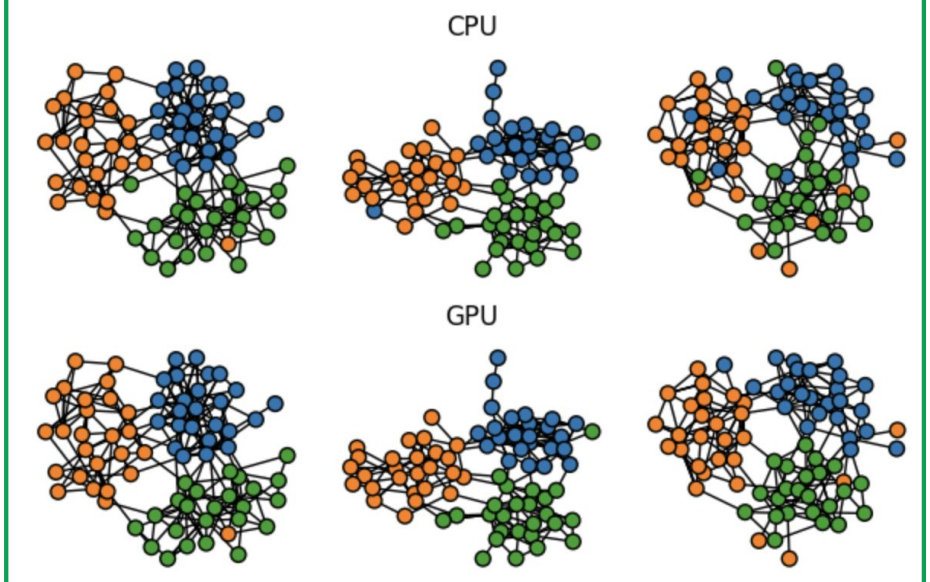


Fig. 1. NetworkX-Temporal integration. Temporal graph snapshots $G^{(t)}$ are assembled into a supra-graph and dispatched through one of two GPU-accelerated paths: multislice modularity via Leiden optimization, or spectral clustering of a (temporal) Bethe-Hessian or modularity matrix. All methods execute on the RAPIDS ecosystem and return per-snapshot community assignments, allowing zero-code-change execution and device selection by setting an environment variable. In blue: our contributions.

$$Q = \frac{1}{2m} \sum_{(i,j) \in V^2} \left[A_{ij} - \gamma \frac{d_i^{out} \, d_j^{in}}{2m} \right] \delta(c_i, c_j)$$





$$Q_{\mathrm{MS}} = \frac{1}{2\mu} \sum_{s,r} \sum_{i,j} \left[\left(A_{ij}^{(s)} - \gamma_s \frac{d_{is}^{\mathrm{out}} \, d_{js}^{\mathrm{in}}}{2m_s} \right) \delta_{sr} + \delta_{ij} \omega_{jsr} \right] \delta\left(c_i^{(s)}, c_j^{(r)} \right)$$