Efficient GPU-Centric Evolving Graph Processing at Scale

Yunmo Zhang[1], Jiacheng Huang[1], Xizhe Yin, Junqiao Qiu[1], Hong Xu[2], Chun Jason Xue[3]

[1] 香港城市大學 City University of Hong Kong

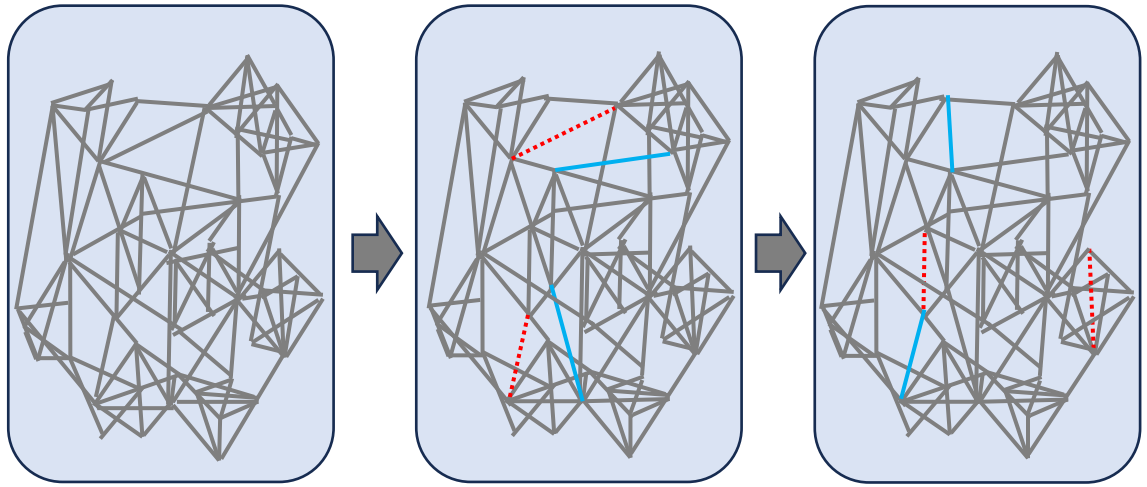
[2]

[3] MOHAMED BIN ZAYED UNIVERSITY OF ARTIFICIAL INTELLIGENCE

Evolving Graph Analytics (EGA)

Real-world graphs change continuously over time,

EGA tracks a graph property over ***a sequence of snapshots.***



G_1 G_2 G_3

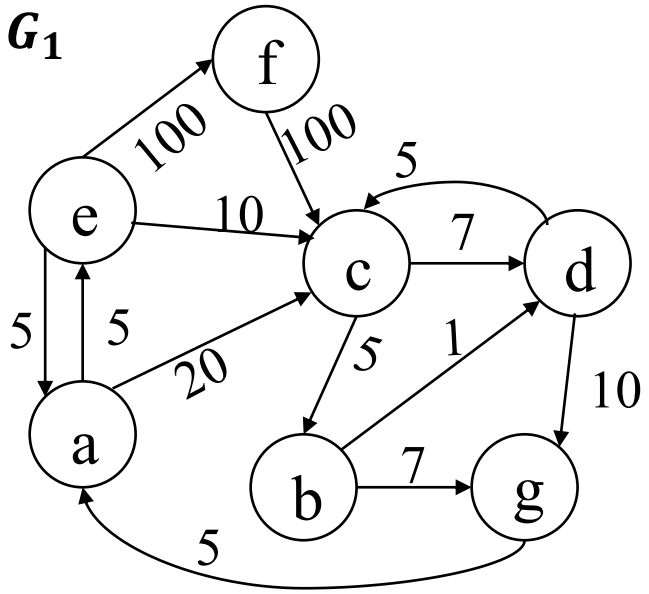
Has many applications:

- social network
- fraud detection
- bioinformatics
- network management
- ……

State-of-the-art EGA Algorithms

- Instead of naïve re-computing, **incremental analysis** methods that **reuse** the prior or common results are developed.

Snapshot G_1



a	b	c	d	e	f	g
0	25	20	26	5	105	22

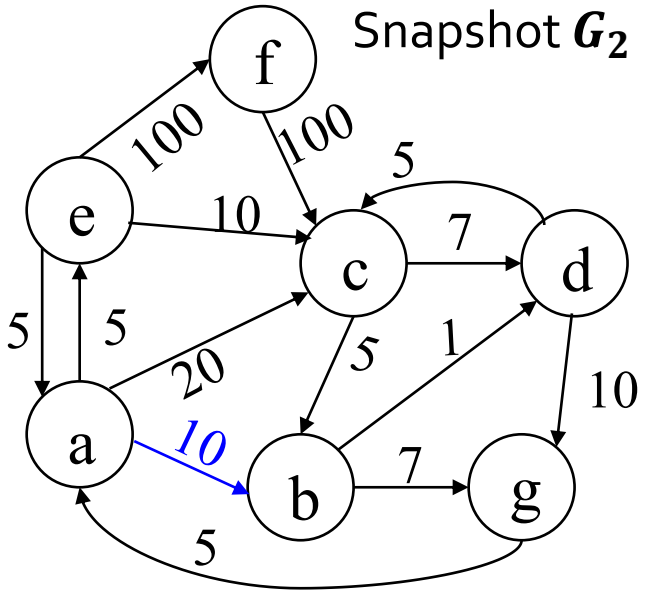
SSSP(a)[1] Results of G_1

1. SSSP: Single Source Shortest Path

State-of-the-art EGA Algorithms

- Instead of naïve re-computing, **incremental analysis** methods that **reuse** the prior or common results are developed.

$\Delta_1 = \{\Delta_1^+, \Delta_1^-\}$



a	b	c	d	e	f	g	
0	25	20	26	5	105	22	SSSP(a)[1] Results of G_1

⇩ Direct Propagation of Update (insertion a->b)

0	*10*	20	26	5	105	22	

...

⇩ Re-convergence

0	*10*	20	*11*	5	105	*17*	SSSP(a) Results of G_2

a	b	c	d	e	f	g

1. SSSP: Single Source Shortest Path

State-of-the-art EGA Algorithms

- Instead of naïve re-computing, **incremental analysis** methods that **reuse** the prior or common results are developed.

- **Incremental** algorithms include two categories:
  - Streaming-based, e.g. Kickstarter[1]

    Reuse the prior snapshot results by maintaining the node dependency and identifying the effects of additions and deletions.
  - Batching-based, e.g., CommonGraph[2]

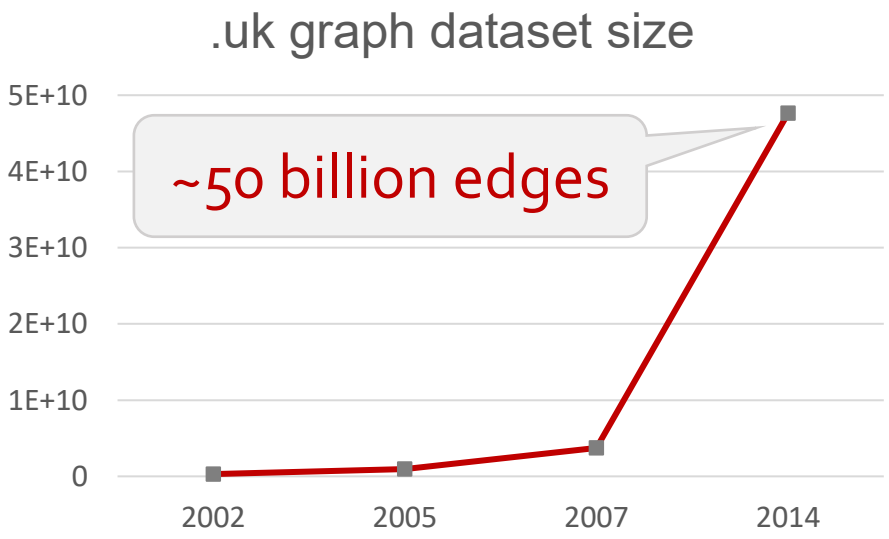
    Reuse the results of common subgraph of snapshots and only identify the effects of additions.

[1] Kickstarter: fast and accurate computations on streaming graphs via trimmed approximations (ASPLOS '17)
[2] CommonGraph: Graph Analytics on Evolving Data (ASPLOS '23)

Challenges for EGA on GPUs

GPU memory constraint

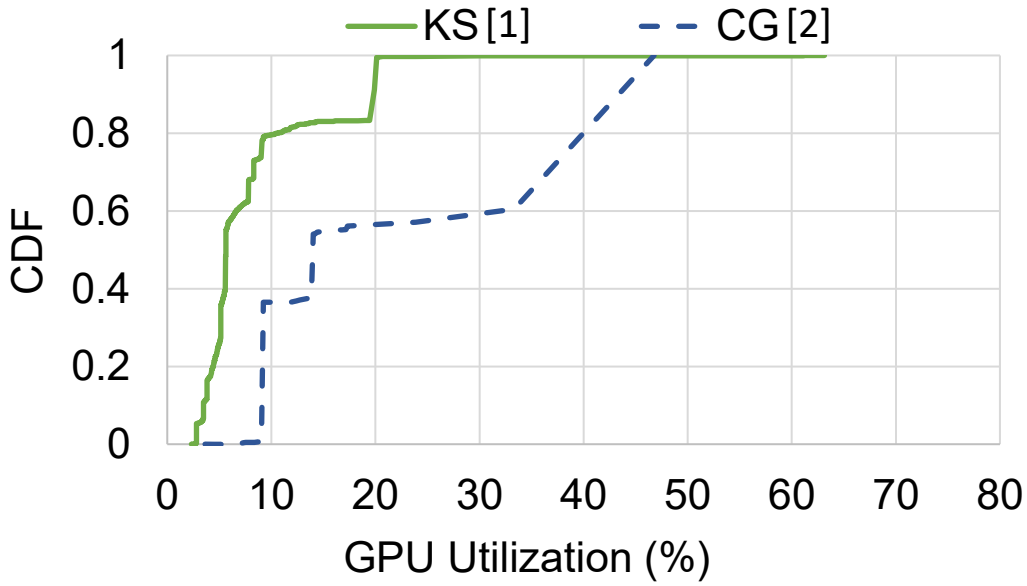
as real-world graphs are scaling up!



Data source: https://law.di.unimi.it/datasets.php

Low GPU utilization

due to the sparsity of incre. analysis.

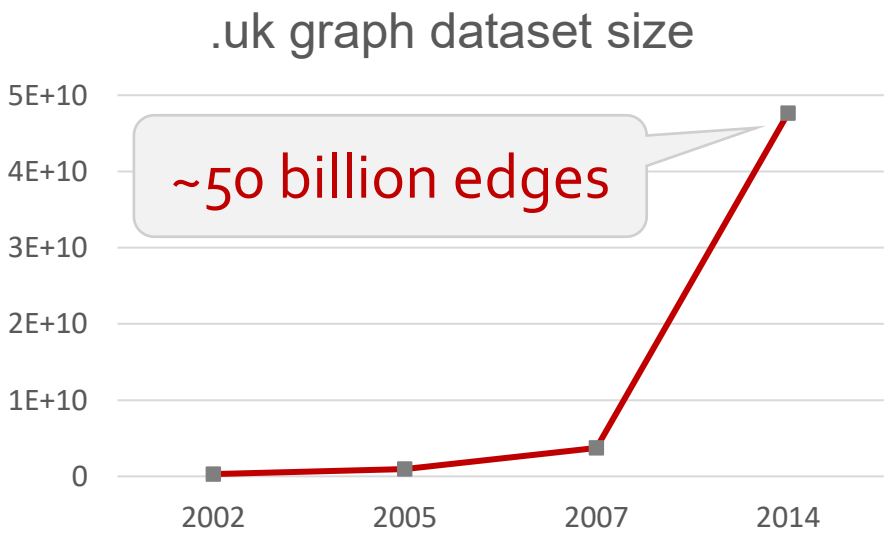


Challenges for EGA on GPUs

Out-of-GPU-memory processing

GPU memory constraint

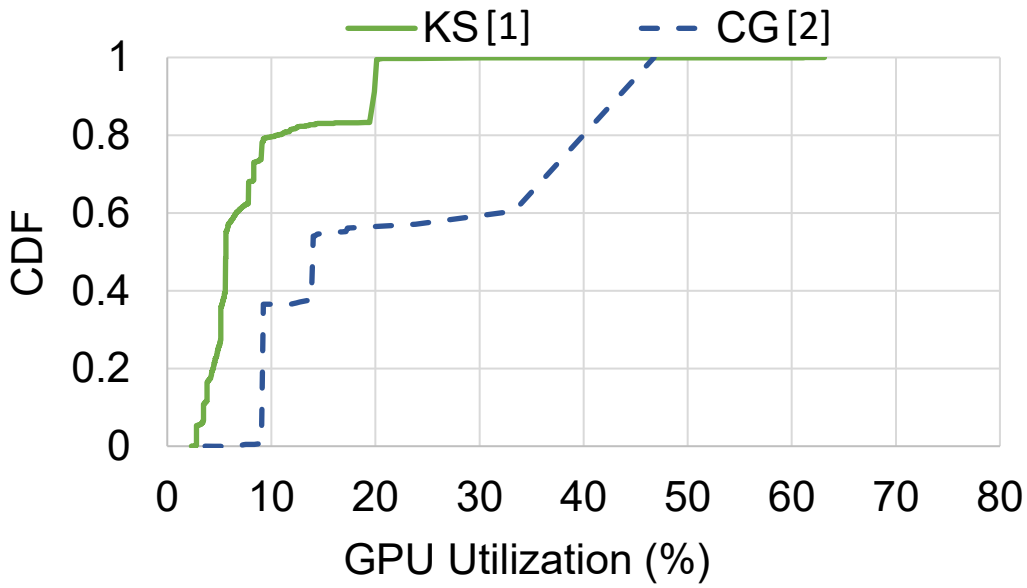
as real-world graphs are scaling up!



Data source: https://law.di.unimi.it/datasets.php

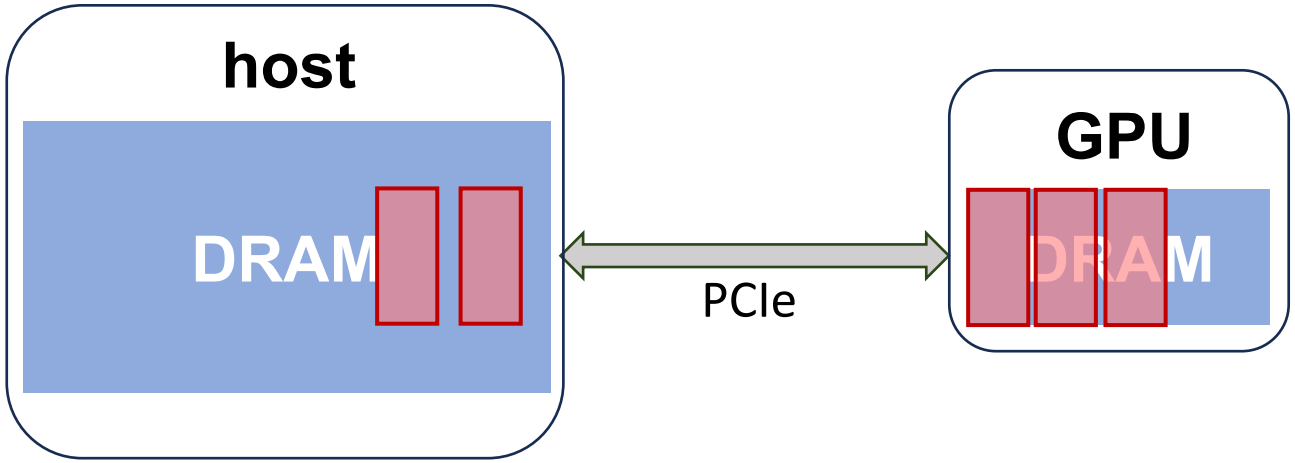
GPU utilization

due to the sparsity of incre. analysis.



Out-of-GPU-memory Processing

= Under GPU memory constraints, data is stored in **host memory** and *transferred* to **GPU memory** on-demand.



Out-of-GPU-memory EGA

Memory management strategies

determine what data to transfer between host and GPU memory.

- Explicit Management: Application-aware
  - Graph partition: transferring to GPU partition by partition.
  - Subway[3]: generating exactly required subgraph.

- Implicit Managament: Application-agnostic
  - Unified Memory (UM)
  - Zero-copy

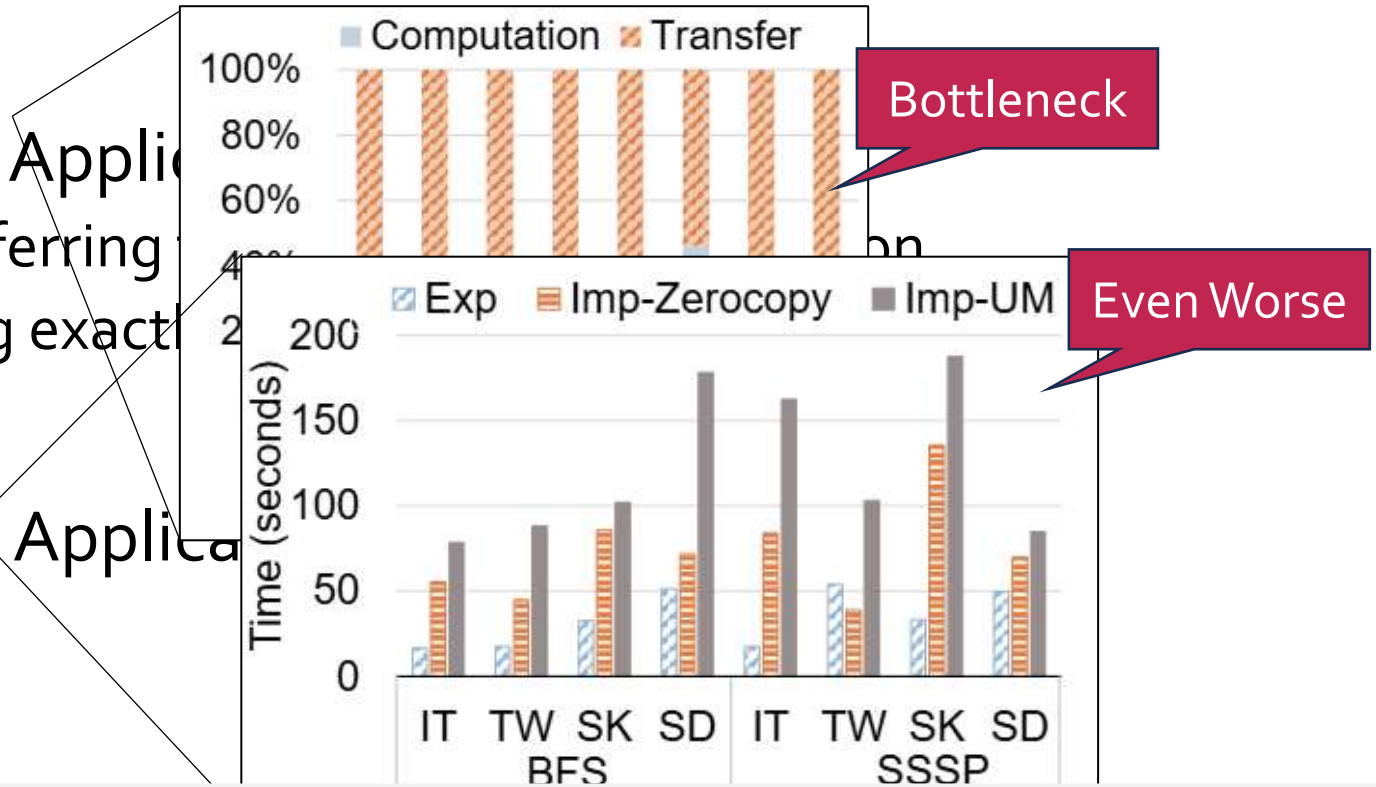
[3] Subway: Minimizing data transfer during out-of-GPU-memory graph processing (Eurosys '20)

Out-of-GPU-memory EGA

Memory management strategies

determine what data to transfer between host and GPU memory.

- Explicit Management: Appli
  - Graph partition: transferring
  - Subway[3]: generating exact
- Implicit Managament: Applica
  - Unified Memory (UM)
  - Zero-copy

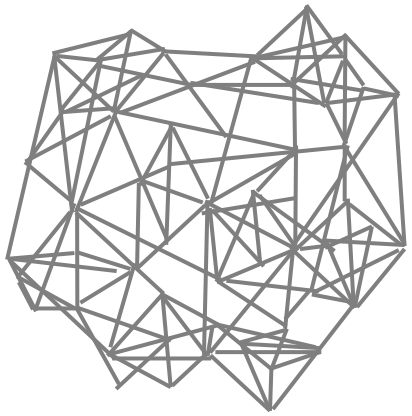


Traditional memory management strategies are insufficient for GPU-based EGA.

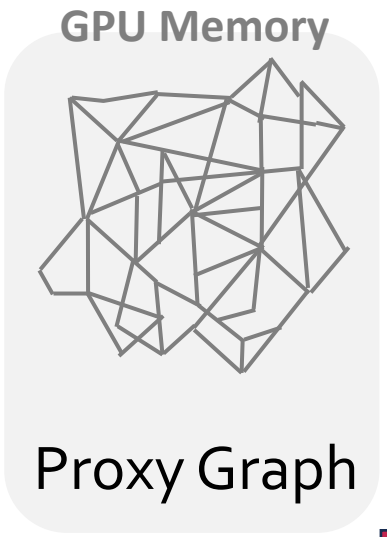
Proxy Graph

A compact graph abstraction that retains computation-critical edges.

Recent advancements in proxy graph design have achieved both a compact scale (e.g., ~10% of the full graph) and high representation fidelity (e.g., ~90% accuracy).



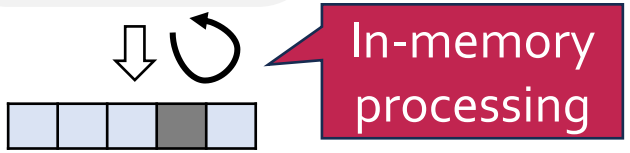
Full Graph



Proxy Graph

Proxy graphs have been used in many scenarios:

- reducing complexity for graph visualization
- optimizing partition navigation in distributed environment
- ……

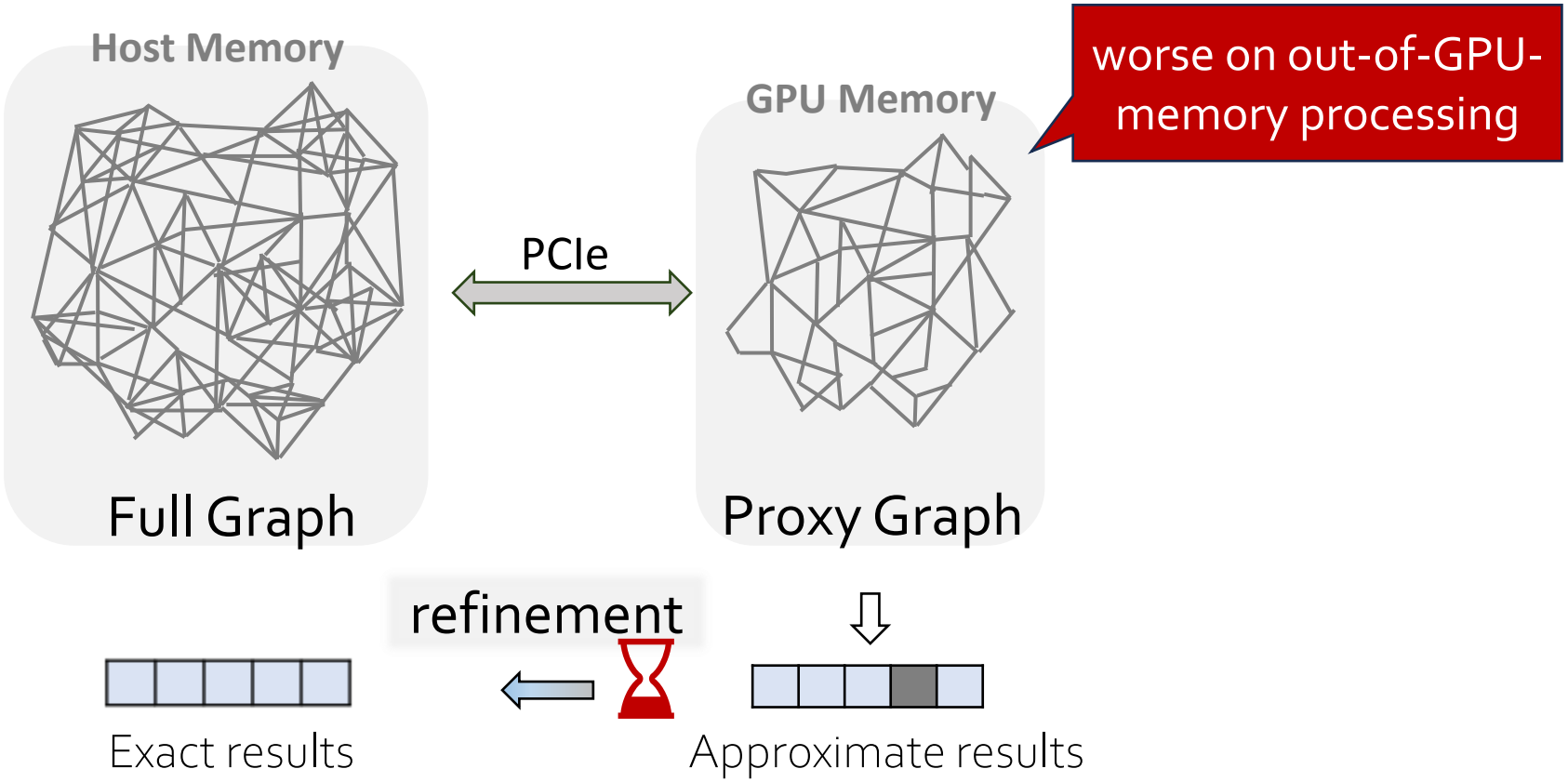


Approximate results

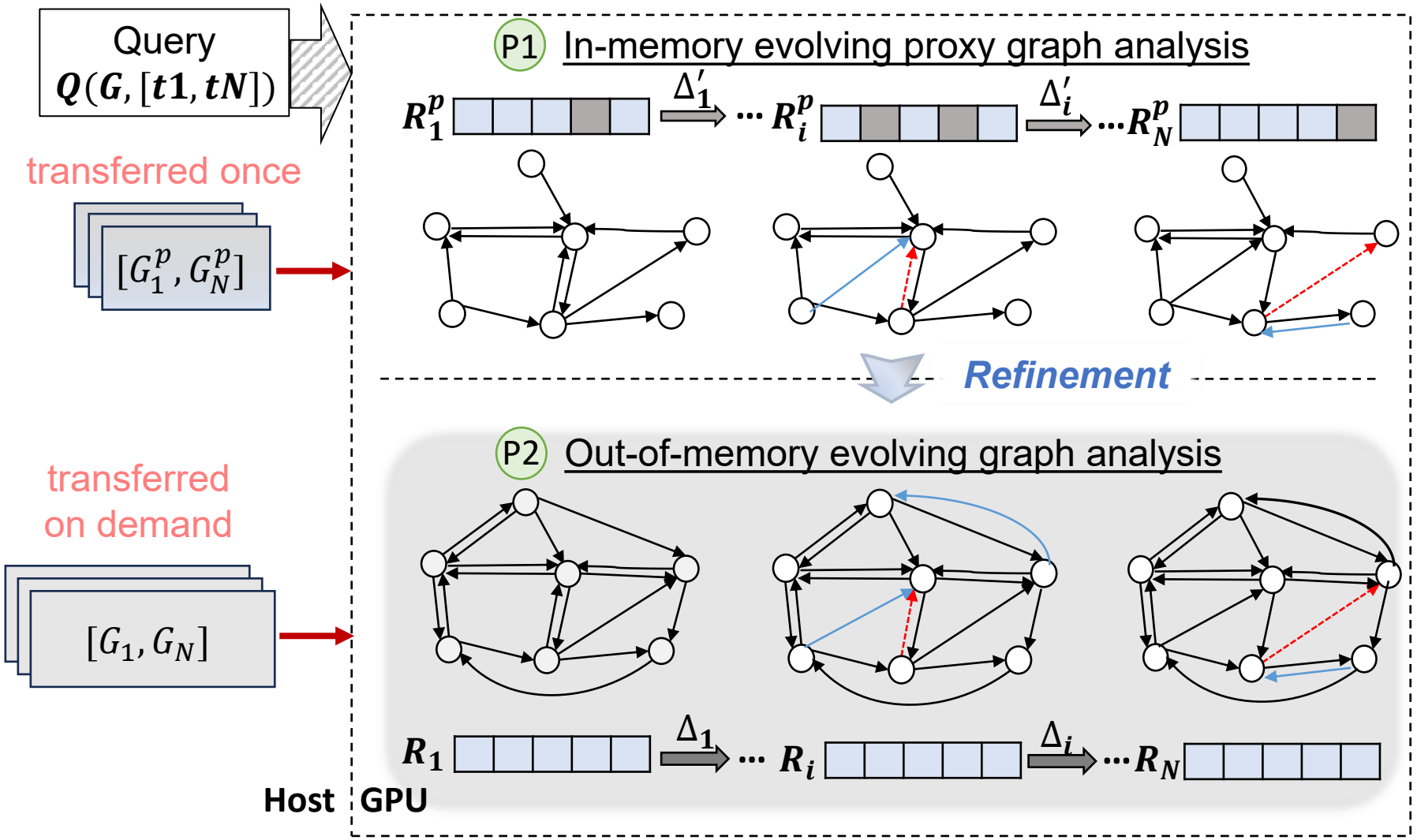
However, bad news ☹

To obtain **exact** results, pro[illegible]aches require significant **additional computation** to refine the initial approximations.

Even outweighs the benefits



Our solution: POEGA



Main Idea

Shift the system bottleneck from *data transfer* to *computation* via (evolving) proxy graphs

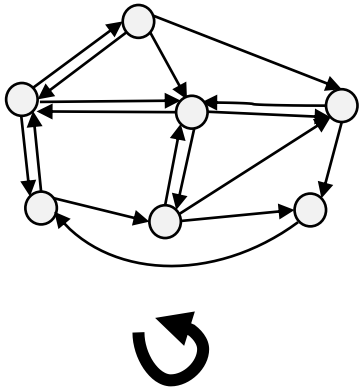
Refer to paper

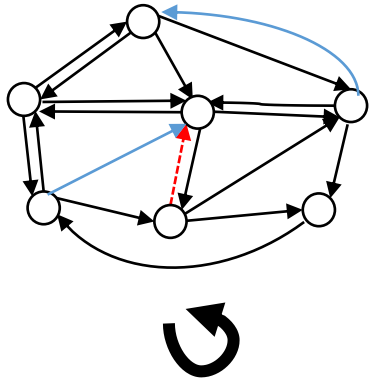
Address the shifted computational burden via massive GPU parallelism,
designing an efficient and scalable *concurrent execution* *to* ***amortize*** *costs* across snapshots.

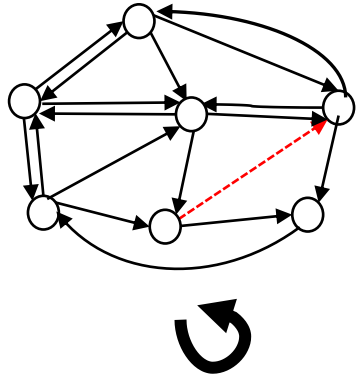
How?

Coalesced Concurrent Execution

Instead of accessing graph data for each snapshot individually,







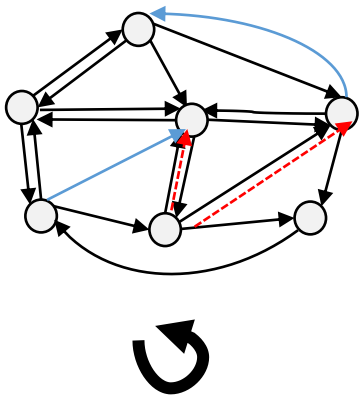
```
Graph_algo_kernel(i)
{   …
    for v in u.neighbors()
    { // data access
        …
    }
}
```

```
Graph_algo_kernel(i + 1)
{   …
    for v in u.neighbors()
    { // data access
        …
    }
}
```

```
Graph_algo_Kernel (i + 2)
{   …
    for v in u.neighbors()
    { // data access
        …
    }
}
```

Coalesced Concurrent Execution

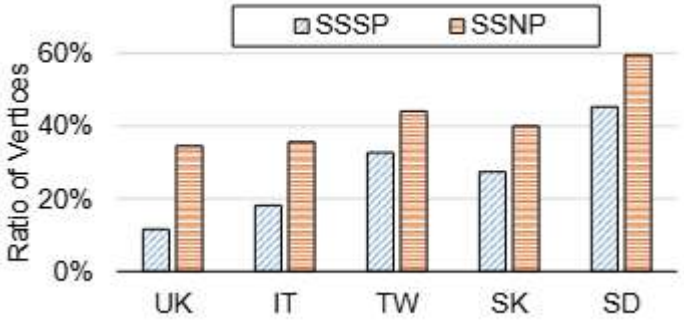
Access each graph data (neighbor) once for all snapshots.



```
Graph_algo_Kernel(…)
{   …
    for v in u.neighbors()
    { // data access
        for snapshot i ∈ [1, N]
        {
            // computation
        }
    }
}
```

Share data access across snapshots

Many Computations are still redundant!



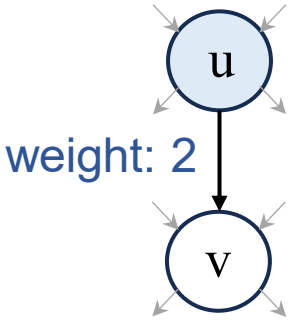
Comp. Redundancy Ratio

Reduce redundant comp. by *bound-based pruning*

Safely identify computations that could be rapidly ***bypassed*** across all snapshots via their ***value bou*** $_{,N]}(u)$.

lazy maintenance to avoid sync overhead (refer to paper)

SSSP Query (interm.) result



	G_1	G_2	G_3	
val_i(u)	3	4	6	Upper bound: 6 Lower bound: 3
	2	2	2	
val_i(v)	4	2	5	Upper bound: 5 Lower bound: 2

SSSP Query edge function (update condition)

val_i(u) + weight < val_i(v)

Necessary condition for update:

Lower(u) + weight < Upper(v)?

YES: Needs to compute for each snapshot
No: Bypass all snapshots!

3 + 2 < 5? → No: Bypass all computations for G_1 to G_3

Ignored scalability bottleneck for concurrent exec.

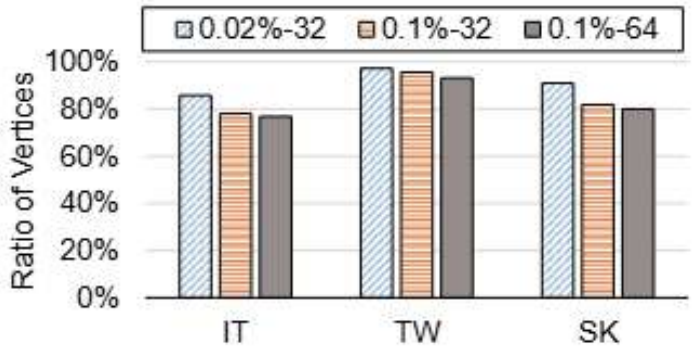
Multiple runtime state data

Value Arrays for N snapshots

	G_1	G_2	…	G_N
v_1	8	8	…	8
	5	5	…	5
	10	10	…	12
	16	18	…	20
	15	15	…	15
	17	17	…	17
v_{max}	55	69	…	69

N×|V| memory usage-> new memory bottleneck

Redundant values across snapshots



Data Redundancy Ratio

Ignored scalability bottleneck for concurrent exec.

Multiple runtime state data

Value Arrays for N snapshots



Efficient, dynamically compact data structures for runtime state on GPU should satisfy the following three requirements *simultaneously*:

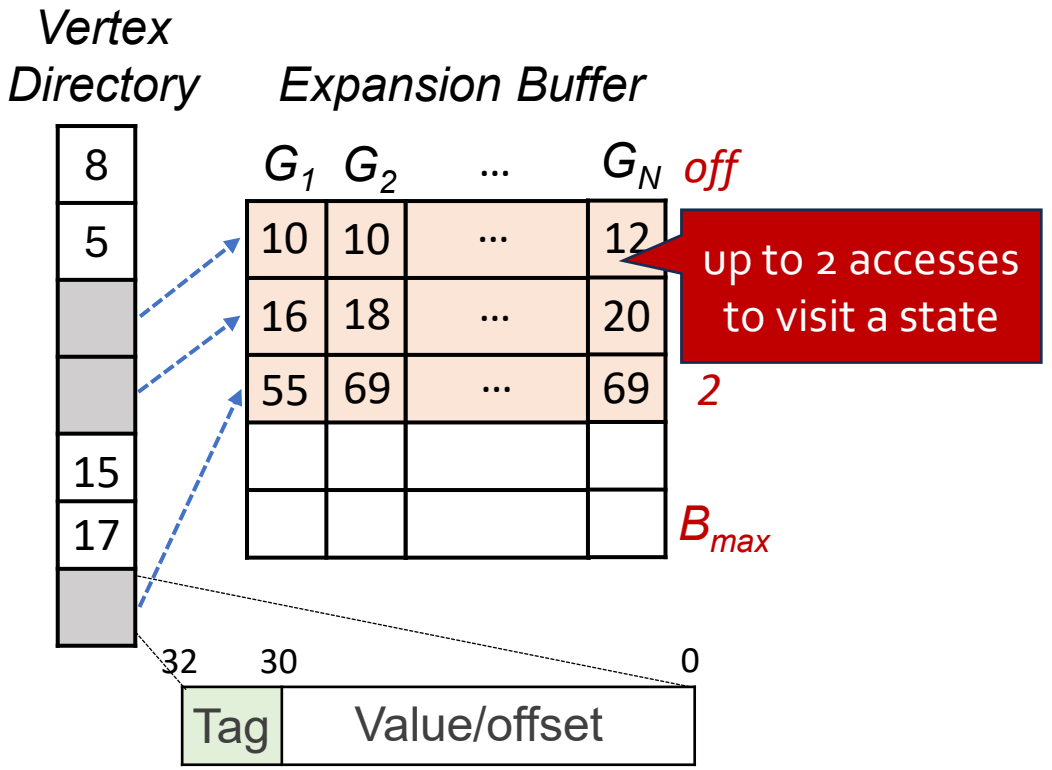
- O(1) Access
- Memory Locality
- Less Synchronization

Adaptive Multi-Version Array (AMVA)

Value Arrays for N snapshots

	G_1	G_2	...	G_N
v_1	8	8	...	8
	5	5	...	5
	10	10	...	12
	16	18	...	20
	15	15	...	15
	17	17	...	17
v_{max}	55	69	...	69

AMVA Data Structure

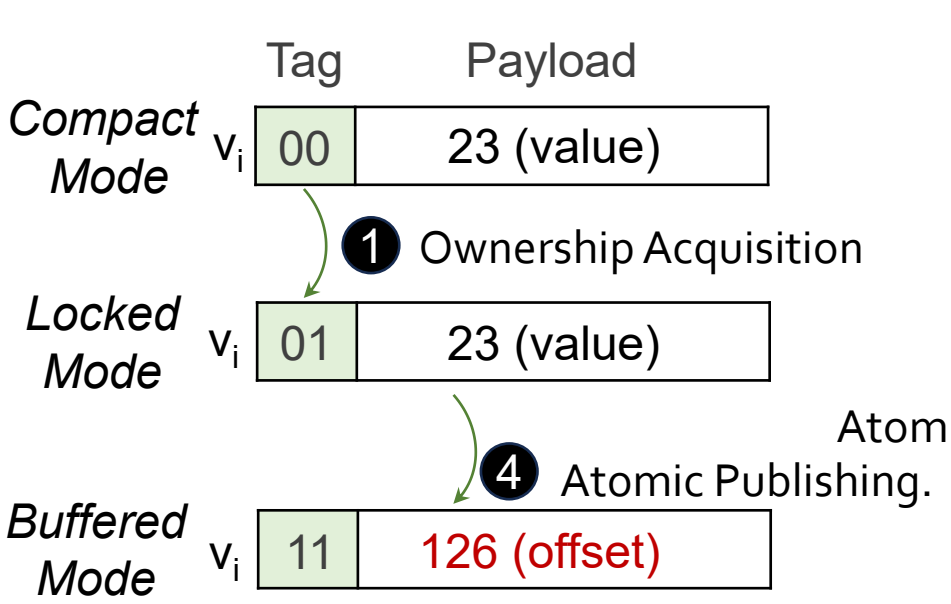


Adaptive Multi-Version Array (AMVA)

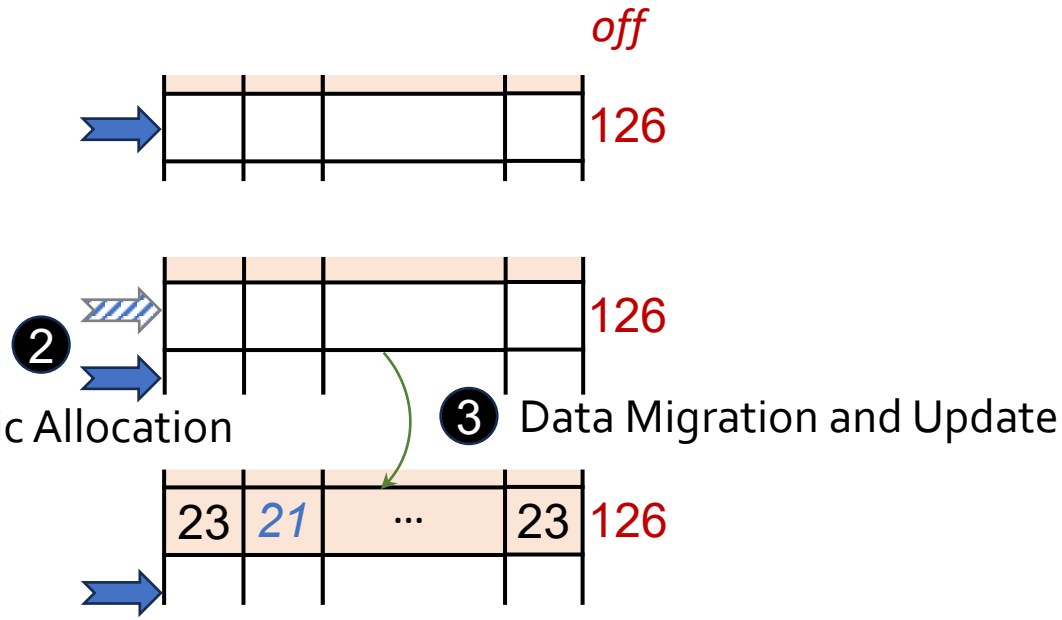
Lock-Free Expand Protocols

Refer to paper

E.g., when there is update v_i's value 23->21



Updates in *Vertex Directory*



Updates in *Expansion Buffer*

Evaluation

- Real-world Datasets

Graph	V	E	$Size_{org}$	$Size_{union}$
UK-2005 (UK) [61]	40M	1.6B	12GB	18GB
IT-2004 (IT) [61]	41M	2.1B	16GB	24GB
Twitter-2010 (TW) [39]	62M	2.4B	18GB	27GB
SK-2004 (SK) [61]	51M	3.7B	28GB	41GB
Subdomain (SD) [50]	102M	3.9B	30GB	44GB

- Platform
  - NVIDIA A6000 GPU with 48 GB memory
  - NVIDIA A4000 GPUwith 16 GB memory
- Graph Algorithm
  - BFS, SSSP, SSWP, SSNP, Viterbi, WCC

- Compared methods
  - Re-evaluation based: **Egraph**[TKDE '22]

Incremental Analysis	**GPU Memory Mgmt. Strategy**	**Compared Methods**
Streaming-based Approach	Zero-copy	**Grapin** (VLDB 25)
	UVM	**KS-UM**
	Explicit Mgmt. (Subway, Eurosys 20)	**KS-SW**
Batch-based Approach	Zero-copy	**CG-ZC**
	UVM	**CG-UM**
	Explicit Mgmt. (Subway, Eurosys 20)	**MEGA** (MICRO 23)

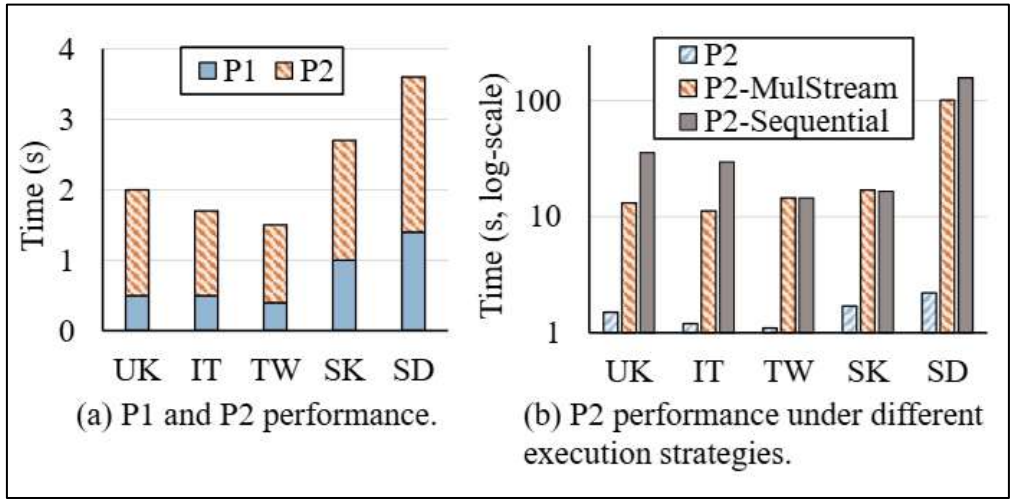
Overall Results

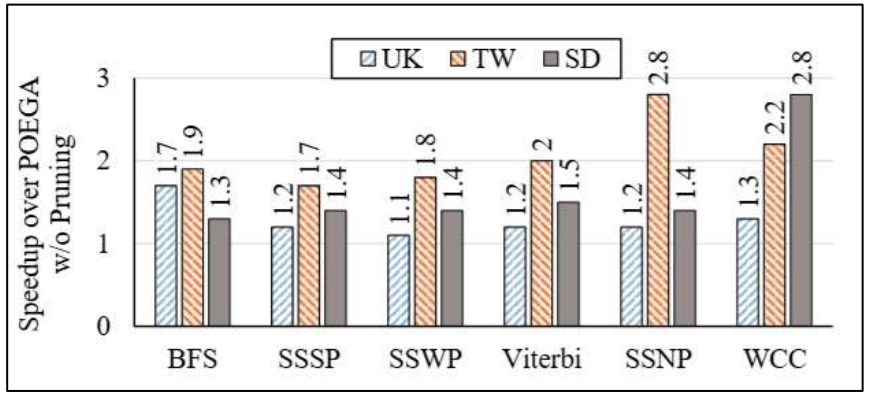
Alg.	G.	EG-raph	KS-UM	KS-SW	Gra pin	CG-UM	CG-ZC	Me-ga	POEGA
BFS	UK	172	5.8	14.9	8.6	32.4	33.7	18.0	1.4
	IT	205	22.5	20.9	13.1	78.3	55.3	16.4	2.1
	TW	164	78.5	25.4	21.2	88.1	44.8	17.5	1.8
	SK	537	80.4	25.3	18.2	102	85.7	32.5	2.3
	SD	859	194	24.3	12.7	178	71.7	51.3	3.2
SSSP	UK	446	17.7	40.3	18.8	58.9	63.2	17.9	2.0
	IT	341	38.8	45.7	13.2	162	84.0	17.5	1.7
	TW	182	92.8	39.6	15.6	103	39.0	53.9	1.5
	SK	767	97.2	49.3	33.0	187	135	33.2	2.7
	SD	1097	297	44.3	18.9	339	69.7	49.6	3.6
SSWP	UK	476	10.0	32.8	10.6	35.8	30.5	19.8	1.3
	IT	948	23.4	53.3	13.1	54.0	46.1	25.6	1.7
	TW	271	89.4	30.8	13.7	82.0	38.5	71.3	1.6
	SK	1749	96.4	66.6	27.9	123	83.5	70.1	2.8
	SD	3545	208	73.1	18.0	222	66.9	119	5.9
Viterbi	UK	474	17.1	8.5	25.9	63.8	34.3	16.3	1.1
	IT	923	37.9	41.6	23.9	15.9	52.4	27.1	1.8
	TW	276	94.6	28.0	21.2	11.3	38.1	71.7	1.5
	SK	1909	109	88.5	31.9	178	174	92.7	4.6
	SD	3473	274	50.7	19.1	276	66.8	127	4.7
SSNP	UK	288	4.0	17.4	8.6	30.6	30.5	14.0	1.2
	IT	994	22.5	51.6	13.9	56.2	45.3	27.2	1.8
	TW	274	89.1	22.5	14.3	83.0	38.4	72.7	1.6
	SK	1479	99.5	65.2	36.3	124	86.7	87.5	2.5
	SD	4318	202	65.4	15.9	201	74.3	125	3.7
WCC	UK	288	5.9	14.3	9.9	30.6	29.5	9.3	1.6
	IT	994	19.0	13.2	7.8	48.6	39.5	12.8	1.9
	TW	274	60.8	17.6	14.6	87.0	37.4	35.1	1.5
	SK	1479	68.3	22.2	19.5	92.1	69.0	32.5	2.7
	SD	4318	126	17.9	11.0	14.6	60.2	63.6	3.0
GMean Speedup (×)			**10.6**	**17.2**	**33.4**	**7.1**	**9.9**	**15.0**	**253.9**

Takeaways:

1. Compared with SOTA (the fastest system in each case), POEGA has the speedup of 6.6× on avg., at most 14.3x.
2. Compared with **Unified Memory** based methods, POEGA has 24.0x and 35.8x speedups for streaming and batch incre. analysis.
3. Compared with **Zero-copy** based methods, POEGA has 7.6x and 25.6x speedups for streaming and batch incre. analysis.
4. Compared with **explicit management** method, Subway, POEGA has 14.8x and 16.9x speedups for streaming and batch incre. analysis.

Breakdown Analysis

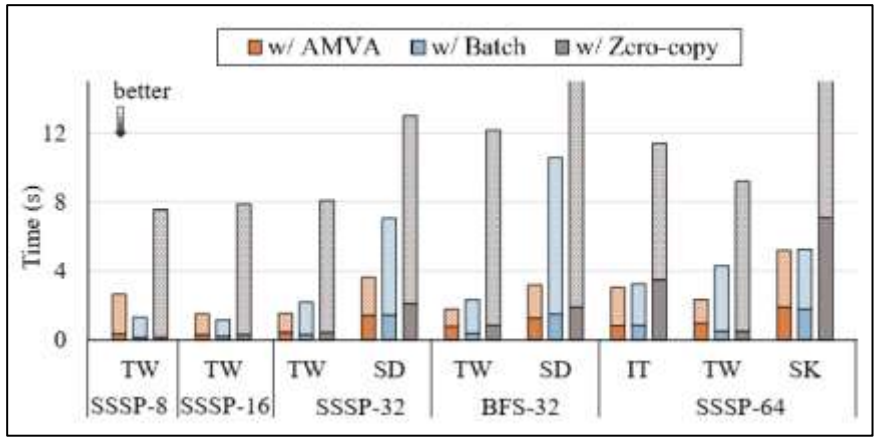




Bound-based Pruning has up to 2.8x speedup.

Two phase breakdown Takeaways:

1. Phase-1 and phase-2 has comparable costs in POEGA.
2. P2 of POEGA has orders of magtitude speedups over concur. execution using CUDA stream and sequential execution.



AMVA has up to 12.9x speedup over states managed by zero-copy, and 4.7x speedup over batch-by-batch (low scalability).

CityU HONG KONG

Efficient GPU-Centric Evolving Graph Processing at Scale

ARTIFACT EVALUATED USENIX AVAILABLE

Yunmo Zhang[1], Jiacheng Huang[1], Xizhe Yin, Junqiao Qiu[1], Hong Xu[2], Chun Jason Xue[3]

Problem: High data transfer overhead in Out-of-GPU-memory EGA

Main Idea: Leverage proxy graphs to shift the data transfer bottleneck to computational overhead.

Challenge: Designing an efficient and scalable concurrent execution.
Contribution 1: Reducing redundant computation by bound based pruning
Contribution 2: Reducing runtime multi-state memory footprints via AMVA

Open-source: *https://github.com/YunMoZhang/POEGA*

yunmo.zhang@my.cityu.edu.hk