FlowWalker: A Memory-efficient and High-performance GPU-based Dynamic Graph Random Walk Framework

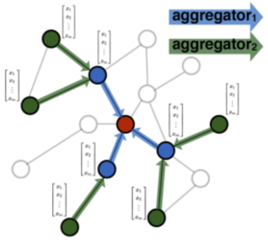
Junyi Mei[1] , Shixuan Sun[1] , Chao Li[1], Cheng Xu[1], Cheng Chen[2], Yibo Liu[1], Jing Wang[1], Cheng Zhao[2], Xiaofeng Hou[1], Minyi Guo[1] , Bingsheng He[3], Xiaoliang Cong[2]

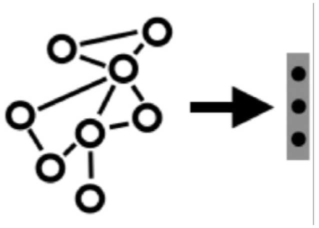
[1]Shanghai Jiao Tong University
[2]ByteDance Inc
[3]National University of Singapore

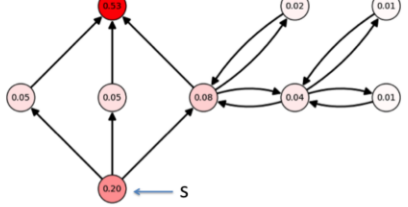
Importance of Graph Random Walk (RW)

RW has been used in many real-world applications:

- Social network analysis
- Recommendation system
- Knowledge graph
- …







Graph neural network
- GraphSaint
- PinSage
- …

Graph embedding
- DeepWalk
- Node2vec
- …

Graph ranking
- Personalized PageRank
- SimRank
- …

Figure source:
[1]. William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17).
[2]. https://cs.stanford.edu/~plofgren/bidirectional_ppr_thesis.pdf
[3]. I. Manipur, et al.,"Netpro2vec: A Graph Embedding Framework for Biomedical Applications" in IEEE/ACM Transactions on Computational Biology and Bioinformatics, vol. 19, no. 02

What is Graph **R**andom **W**alk (RW)

Input

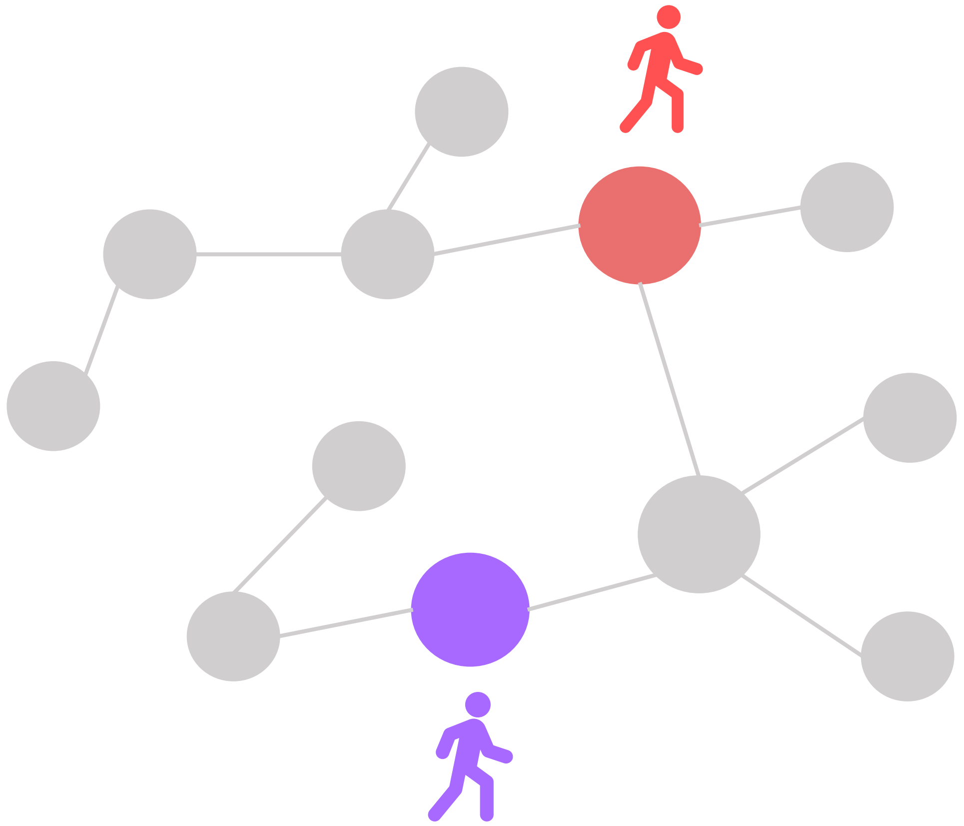
- A graph $\boldsymbol{G}$
- A set of walkers $\boldsymbol{Q}$, with start vertices

Walking Process

- Each walker selects a neighbor of current vertex at random
- Move to the selected neighbor
- Repeat until the termination condition is met

Output

- Walking path of walkers in $\boldsymbol{Q}$



What is Graph Random Walk (RW)

Input

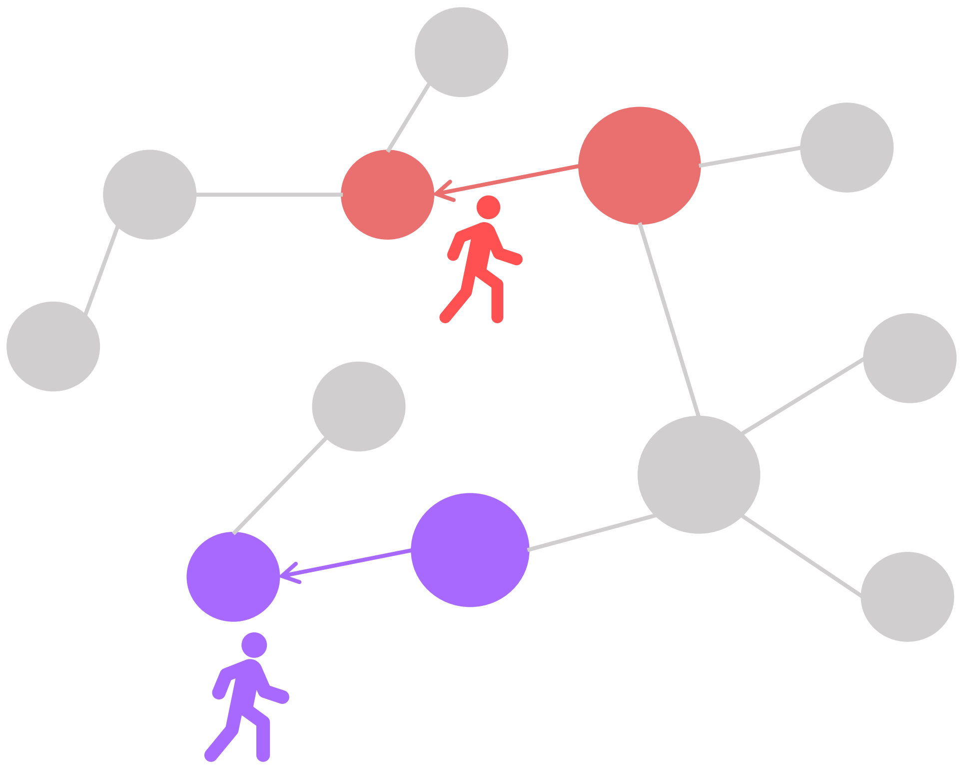
- A graph $\boldsymbol{G}$
- A set of walkers $\boldsymbol{Q}$, with start vertices

Walking Process

- Each walker selects a neighbor of current vertex at random
- Move to the selected neighbor
- Repeat until the termination condition is met

Output

- Walking path of walkers in $\boldsymbol{Q}$



What is Graph **R**andom **W**alk (RW)

Input

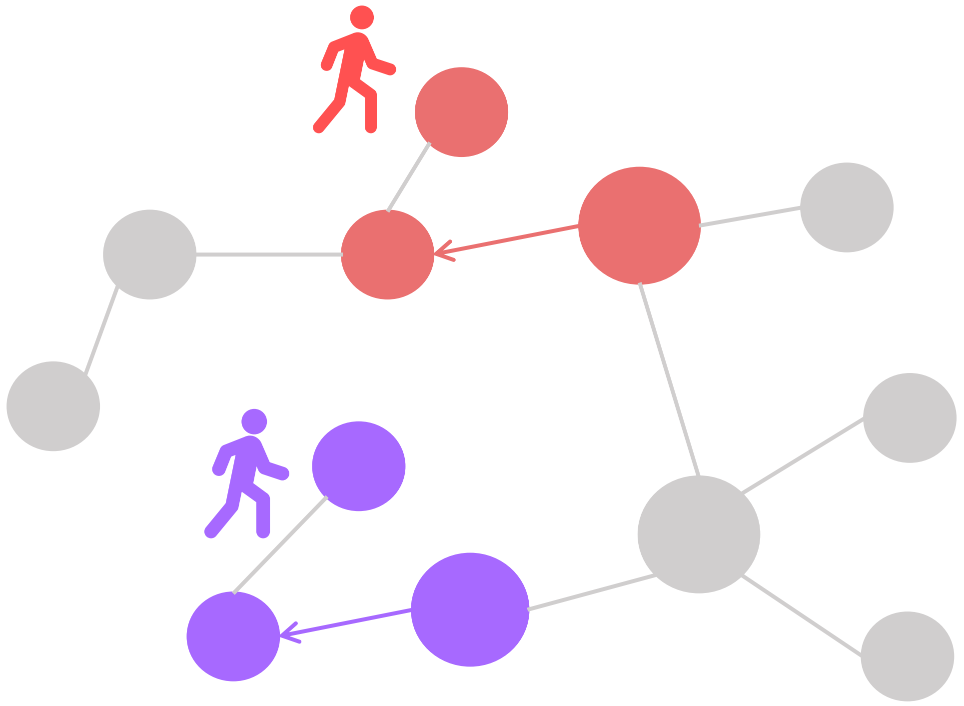
- A graph $\boldsymbol{G}$
- A set of walkers $\boldsymbol{Q}$, with start vertices

Walking Process

- Each walker selects a neighbor of current vertex at random
- Move to the selected neighbor
- Repeat until the termination condition is met

Output

- Walking path of walkers in $\boldsymbol{Q}$



Dynamic Graph Random Walk (DGRW)

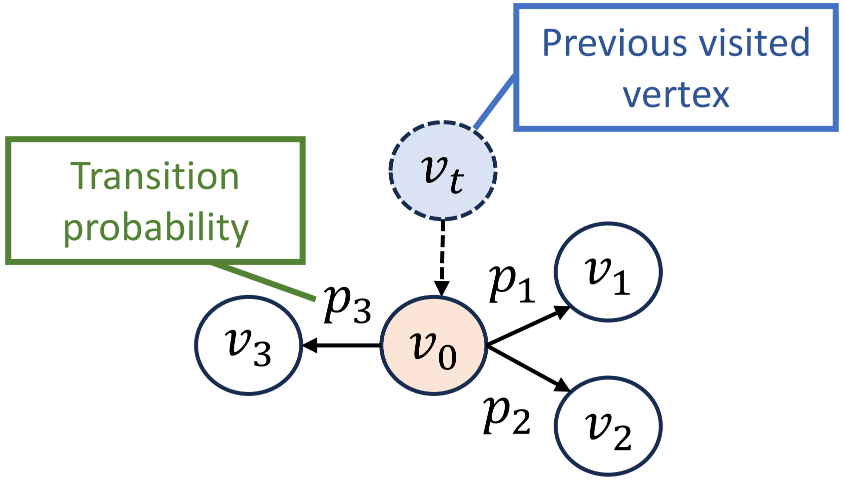
RW applications differ in the way of selecting neighbors, i.e., **how to calculate transition probability (p)**

Static Graph Random Walk

- Calculate and store p **before random walk starts**
- p is **fixed** during runtime

Dynamic Graph Random Walk

- Calculate p dynamically **during runtime**
- p might **change**
- Example: in Node2Vec[1], $\{\boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{p_3}\}$ is dependent on the state of $\boldsymbol{v_t}$, cannot be computed in advance



[1]. Grover, Aditya, and Jure Leskovec. "node2vec: Scalable feature learning for networks." *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 2016.

How to Process DGRW

DGRW

- ➢ Higher computation workload, but less space cost than SGRW
- ➢ A huge amount of queries

✚

CPU

χ Limited computing cores

FPGA

χ Focus on hardware design

GPU

✓ Massive computing cores

✓ Potential for high parallelism

How to Process DGRW

CPU

χ Limited computing cores

DGRW

- Higher computation workloa
  space cost than SGRW
- A huge amount of queries

Challenges

- *Limited memory space*
- *Load imbalance*

dware design

GPU

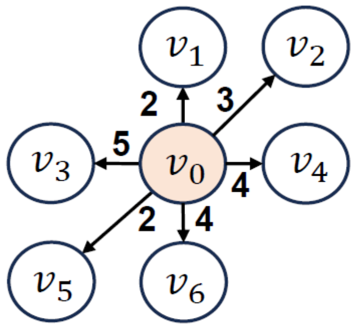
✓ Massive computing cores

✓ Potential for high parallelism

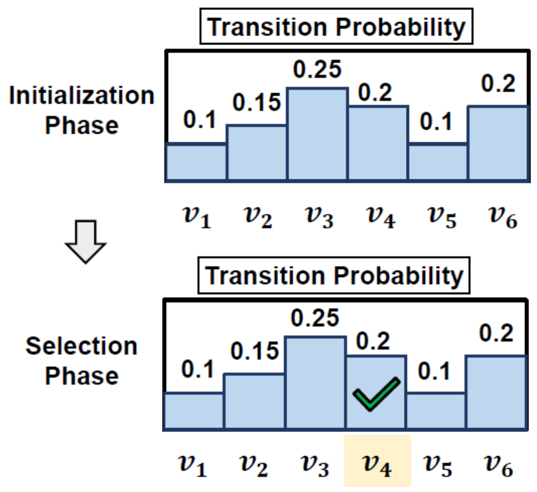
Challenge: Limited Memory Space

Memory cost of DGRW:

- **Graph data** – necessary data, cannot be eliminated
- **Walking trace** – necessary data, cannot be eliminated
- **Buffer for computation** – necessary data?
  - Sampling: initialization, then selection
  - Sampling algorithms, like ITS and ALS, require an $O(d)$ buffer , d is the vertex degree



$N(v_0)$	v_1	v_2	v_3	v_4	v_5	v_6
Weight	2	3	5	4	2	4

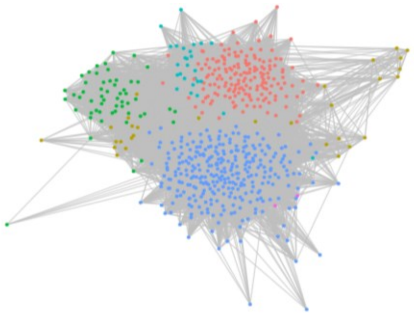


Sampling method	ITS	ALS
Initialization	O(d)	O(d)
Selection	O(log d)	O(1)
Space Cost	O(d)	O(d)

Challenge: Limited Memory Space

Memory cost of DGRW:

- **Graph data** – necessary data, cannot be eliminated
- **Walking trace** – necessary data, cannot be eliminated
- **Buffer for computation** – necessary data?
  - Sampling: initialization, then selection
  - Sampling algorithms, like ITS and ALS, requires an $O(d)$ buffer, d is the vertex degree
  - Only a few queries can be hold on GPU simultaneously



The Twitter Graph
($d_{max} = 3 \times 10^6$)

Memory overhead when $N_{query} = 10^6$ in Skywalker:
- Graph data: 18GB
- Trace: 0.3GB
- A single query: 11.45MB

A few queries can be hold

A100

Memory space of modern GPU is not very large:
- A100: 40GB/80GB

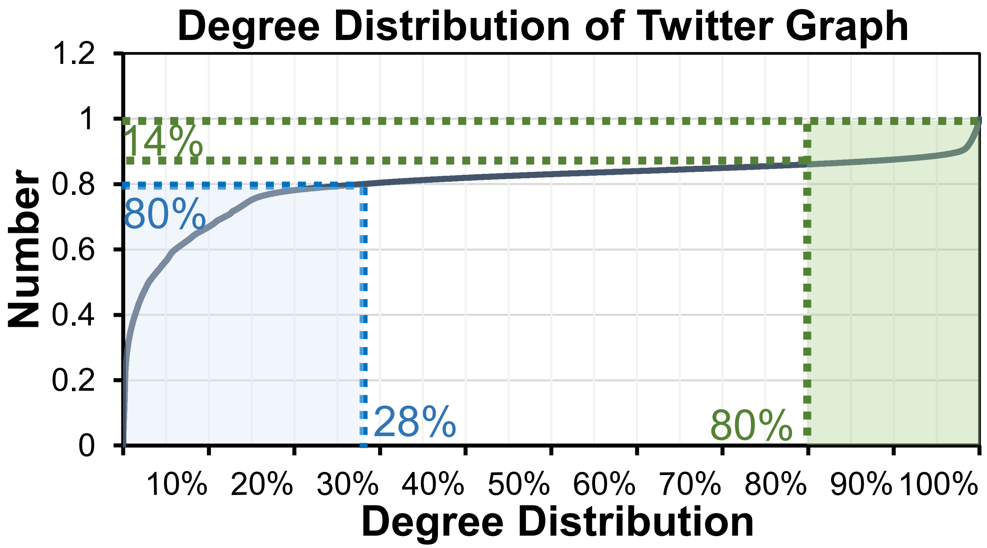
Figure source:
[1]. https://ona-book.org/using-twitter.html
[2]. https://www.nvidia.com/en-us/data-center/a100/

Challenge: Load Imbalance

The sampling workload varies:

- Workload is governed by vertex degree
- Degrees in real-world graph follow **power-law distribution**
- Huge amounts of computing cores on GPU exacerbates the imbalance problem

80% of total vertices with lowest **28%** degrees



14% of total vertices with highest **20%** degrees

Our Solution

FlowWalker: an efficient DGRW framework at minimal memory cost

- Adopt **reservoir sampling** to reduce sampling space complexity to $O(1)$.

  ✓ Eliminate computation buffer

- High-performance processing engine, which leverages a **sampler-centric computation model** and performs **dynamic scheduling**.

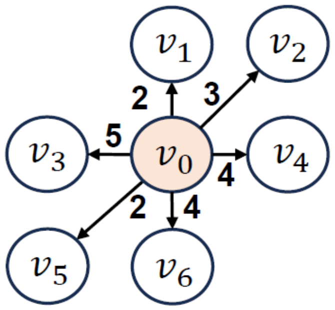
  ✓ Mitigate load imbalance

FlowWalker supports DGRW on large graphs with high parallelism.

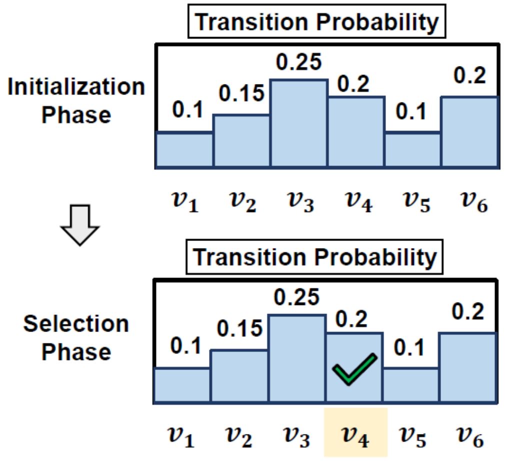
<u>R</u>eservoir <u>S</u>ampling (RS)

RS saves memory:

- Sampling: initialization, then selection
- Sampling algorithms, like ITS and ALS, require an $O(d)$ buffer , d is the vertex degree
- RS is pre-processing free and diminishes space complexity to $O(1)$
- Can be accelerated through parallel processing



$N(v_0)$	v_1	v_2	v_3	v_4	v_5	v_6
Weight	2	3	5	4	2	4



Sampling method	ITS	ALS	RS
Initialization	O(d)	O(d)	-
Selection	O(log d)	O(1)	O(d)
Space Cost	O(d)	O(d)	O(1)

FlowWalker: Parallelize Reservoir Sampling

Direct Parallel Reservoir Sampling (DPRS)

Input: number of neighbors n; number of threads k;
Output: the selected vertex;
for range $[0, \frac{n}{k} - 1]$:
  parallel for k *neighbors of* v: calculate bias once
    w_v = GET_EDGE_BIAS(v);
    p_v = PREFIX_SUM(w_v);
    /*Get the result of the local thread*/ scan in every loop
    select_local = SELECTION(p_v);
    /*Get the global result of this loop*/ reduction in every loop
    select_global = REDUCTION(*select_local*);
/*Return the final result after all the loops*/
return *select_global*;

- $O\left(\frac{n}{k} \times logk\right)$ time, $O(1)$ space
- Used when the bias calculation overhead is **high** (i.e. Node2Vec requires a binary search)

Zig-Zag Parallel Reservoir Sampling (ZPRS)

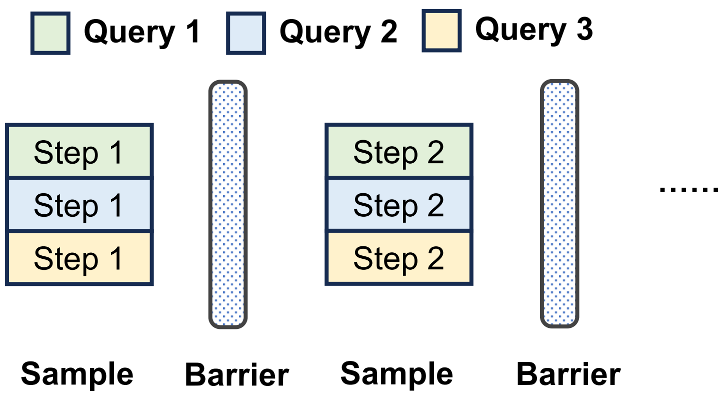
Input: number of neighbors n; number of threads k;
Output: the selected vertex;
for range $[0, \frac{n}{k} - 1]$:
  parallel for k *neighbors of* v:
    w_v += GET_EDGE_BIAS(v);
p_v= PREFIX_SUM(w_v); one global scan
for range $[0, \frac{n}{k} - 1]$:
  parallel for k *neighbors of* v:
    p_v += GET_EDGE_BIAS(v); calculate bias twice
    select_local = SELECTION(p_v);
select_global = REDUCTION(*select_local*); one global reduction
return *select_global*;

- $O\left(\frac{n}{k} + logk\right)$ time, $O(1)$ space
- Used when the bias calculation overhead is **low** (i.e. MetaPath only needs a label matching)

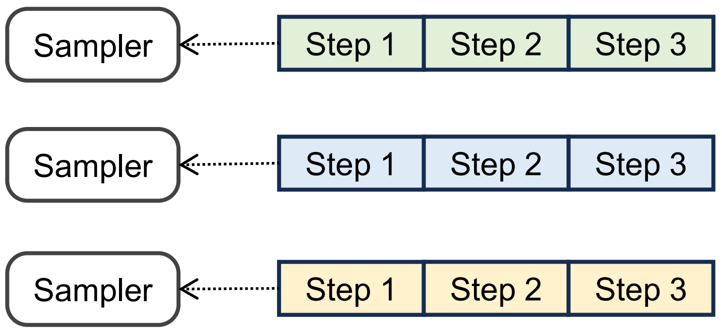
FlowWalker: Execution Engine

Sampler-centric computation

- Step-centric model: **global barrier** between each step
- Sampler-centric model: **eliminate global barrier**
- Samplers with various computing capabilities cater for vertices with different degrees



Step-centric
synchronized, global barrier



Sampler-centric
asynchronized, barrier free

FlowWalker: Execution Engine

Sampler-centric computation

- Step-centric model: **global barrier** between each step
- Sampler-centric model: **eliminate global barrier**
- Samplers with various computing capabilities cater for vertices with different degrees

Dynamic scheduling

- Fetches one query from the global task pool when one query completes
- A walking query will not be evicted until it completes
- Achieves **adaptive load balancing**

Experiment Setup

Baseline

- **Skywalker** [*PACT' 21*] – GPU-based framework
- **LightRW** [*SIGMOD' 23*] – FPGA-based dynamic RW framework
- **ThunderRW** [*VLDB' 21*] – CPU in-memory framework
- **DGL** – widely adopted GNN framework, run in dynamic mode, CPU for Node2Vec, and GPU for other applications

Datasets

- 10 read-world datasets, including **5 billion-scale datasets**

Dataset	Name	$\|V\|$	$\|E\|$	d_{max}	Size(GB)
com-youtube	YT	1.1 M	6 M	28K	0.05
cit-patents	CP	3.8 M	33 M	793	0.26
Livejournal	LJ	4.8 M	86 M	20K	0.66
Orkut	OK	3.1 M	234 M	33K	1.76
EU-2015	EU	11 M	522M	399K	3.93
Arabic-2005	AB	23 M	1.1B	576K	8.34
UK-2005	UK	39 M	1.6B	1.7M	11.82
Twitter	TW	42 M	2.4 B	3M	18.08
Friendster	FS	66 M	3.6 B	5K	27.16
SK-2005	SK	51 M	3.6 B	8.5M	27.16

Datasets

Applications

- DeepWalk, Personalized PageRank (PPR), Node2Vec, MetaPath

Environment

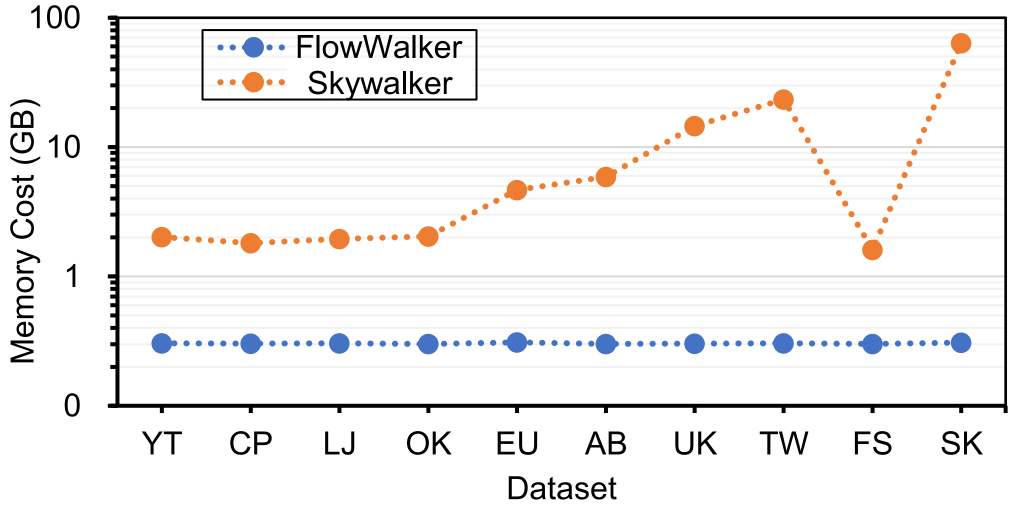
- A100 (40 GB) GPU, 100 KB shared memory of each SM
- AMD Alveo U250 FPGA
- Intel 8336C CPU with 16 cores and hyper-threading enabled

Overall Comparison

- **FlowWalker is the only framework completing all test cases**
- **Speedup:** FlowWalker achieves significant speedup, up-to 752.2x
- **Memory:** The extra memory cost of FlowWalker stays constant accross different datasets

Framework	Platform	Maximum Speedup
DGL	GPU	92.2x
DGL	CPU	315.8x
LightRW	FPGA	16.4x
ThunderRW	CPU	752.2x
Skywalker	GPU	72.1x

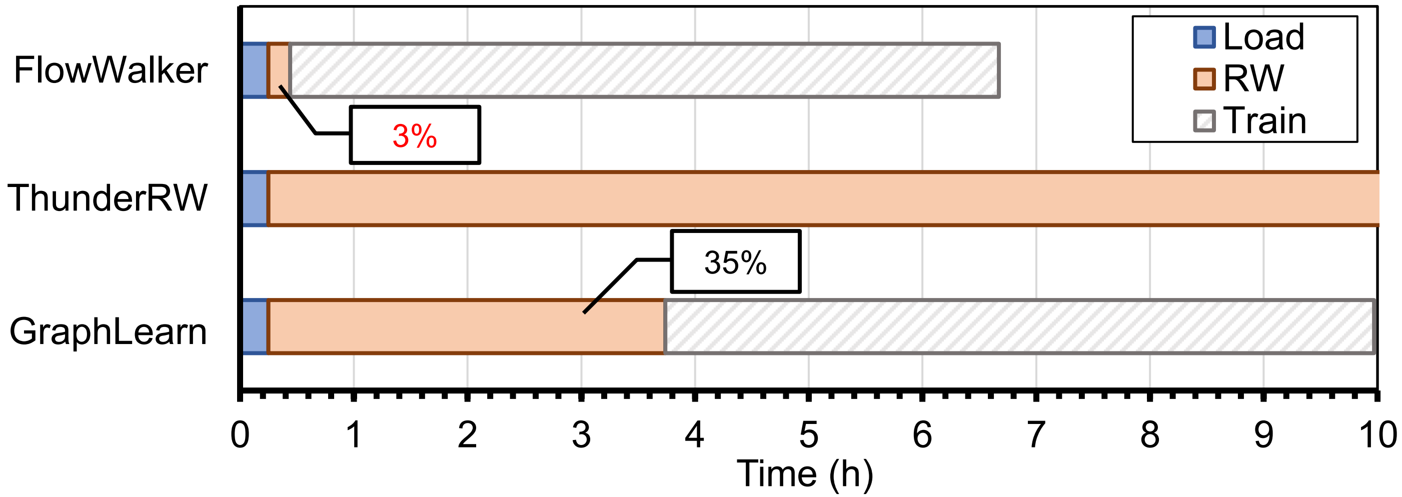
The speedup of FlowWalker over baselines



Extra memory cost ($V_{total} - V_{dataset}$) of FlowWalker and Skywalker

Case Study

- **Friend recommendation GNN** in Douyin (a popular video APP), implemented based on GraphLearn
- Test graph contains 227 million vertices and 2.71 billion edges
- FlowWalker reduces the RW time from **35%** (3.49 hours) to **3%** (13 minutes)



Training one epoch of GNN

Conclusion

- FlowWalker is an efficient GPU dynamic graph random walk framework.
- FlowWalker employs the reservoir sampling and **reduce sampling memory cost to $\boldsymbol{O(1)}$**.
- FlowWalker uses dynamic walking engine and sampler-centric model to **mitigate workload imbalance and the global barrier**.
- FlowWalker achieves a significant speedup with minimal memory cost. It can process on large graphs with a high parallelism.

Source code at https://github.com/junyimei/flowwalker-artifact
Contact: meijunyi@sjtu.edu.cn

Scan and star ☻

Thank you!

Q&A

Scan and star ☻

Source code at https://github.com/junyimei/flowwalker-artifact

Contact: meijunyi@sjtu.edu.cn