UNIVERSITY OF CAMBRIDGE

A Distributed Multi-GPU System for Fast Graph Processing

Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, Alex Aiken

Presented by Thomas Yuan

Summary

Lux

- Distributed Multi-GPU system
- Graph Processing
- Two Execution Models
- Load Balance Model
- Performance Model

Background

Prior Work: Multi-CPU Systems eg Pregel, PowerGraph, Ligra

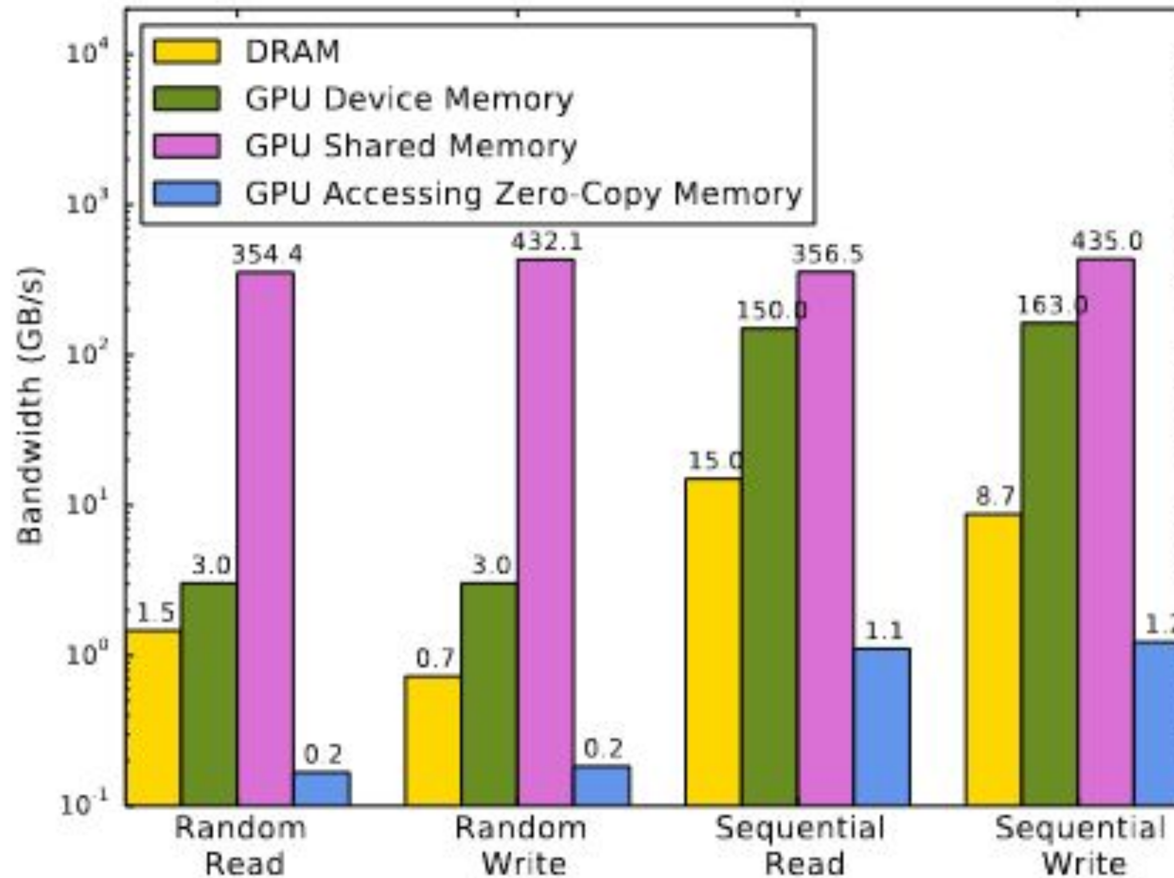
- Graph on CPU DRAM

Bottleneck of graph applications: Memory bandwidth

GPU better than CPUs

- Higher memory bandwidth
- Power efficiency
- Hardware parallelism

Background



Execution Model

Push Model

- Optimizes algorithmic efficiency
- Benefits applications where a small subset of vertices are active over iterations

Pull Model

- Enables important GPU optimizations
- Benefits applications where most vertices are active over iterations

Execution Model

```
interface Program(V, E) {
  void init(Vertex v, Vertex v^old);
  void compute(Vertex v, Vertex u^old,
               Edge e);
  bool update(Vertex v, Vertex v^old);
}
```

Main difference:

- Pull model iteratively pulls potential updates from all in-neighbours
- Push model pushes updates to all out-neighbours, uses frontier queue

Execution Model

Pull Model (vs Push Model)

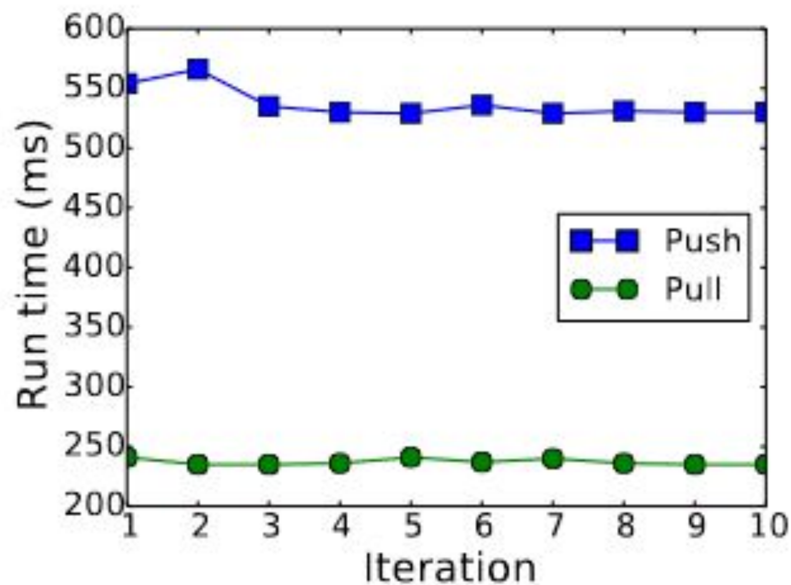
Pros:

- Less synchronization required
- Enable GPU optimizations
  - GPU can locally aggregate and cache certain updates in shared memory
  - Coalesced Memory Access

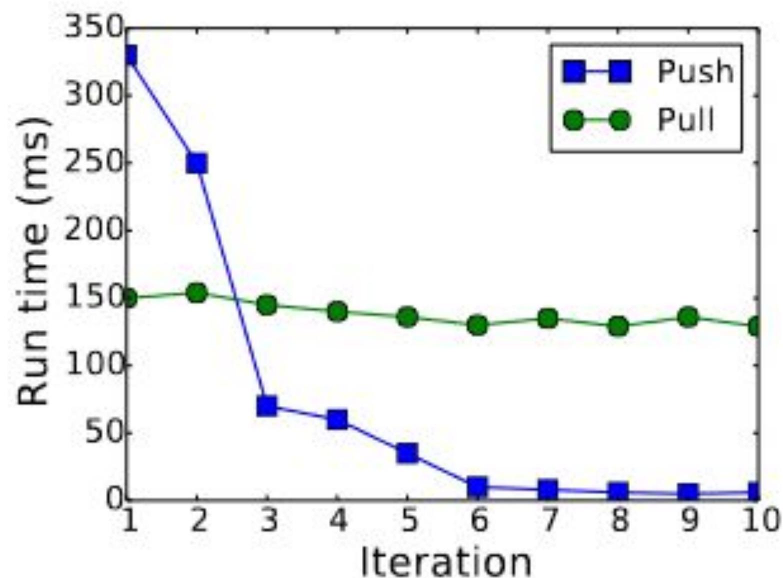
Cons:

- All vertices need to pull for updates in every iteration

Execution Model



(a) PR.



(b) CC.

Figure 19: Per iteration runtime on TW with 16 GPUs.

How Lux uses the Memory Hierarchy

Larger

Higher bandwidth

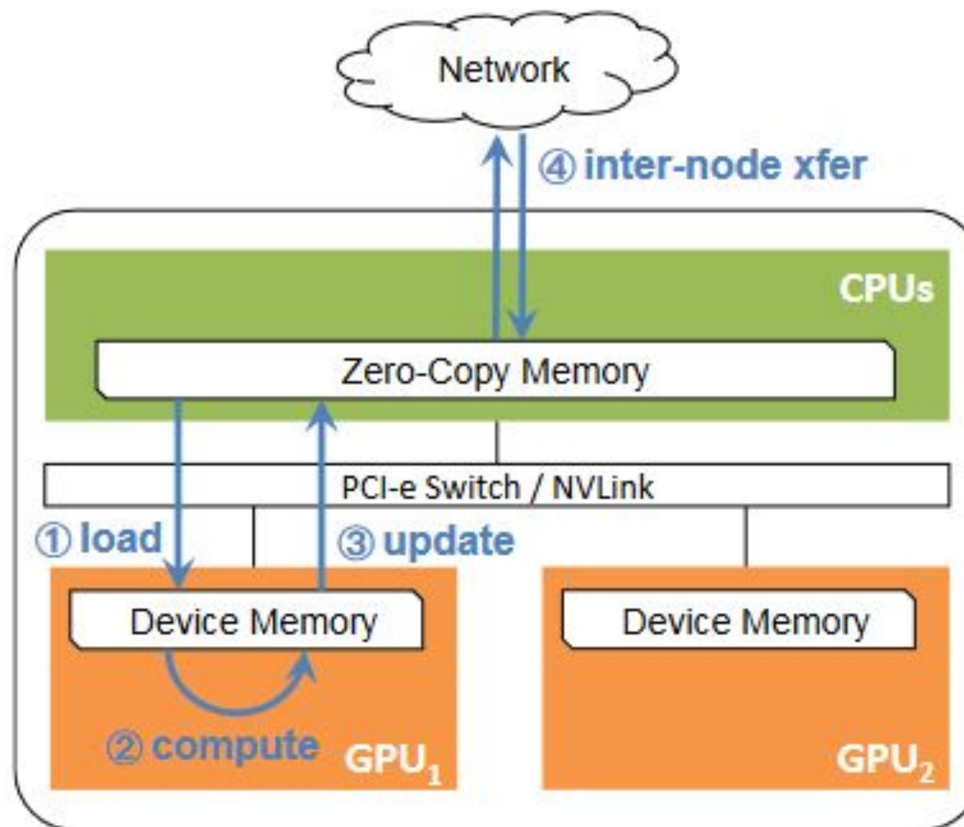
Zero-copy Memory for storing vertex properties

- Partially sharing
- Overlap data movements with work to hide latency
- Data transfers only happen between iterations

GPU Device Memory for actual computation

GPU Shared Memory for Caching, Aggregation, Storing data processed cooperatively (Only in Pull!)

How Lux uses the Memory Hierarchy



How Lux uses the Memory Hierarchy

Coalesced Memory Access

- An important optimization by GPUs
- When multiple GPU threads issue memory references to consecutive memory addresses
- GPU hardware automatically coalesce references into a range request which is more efficiently handled
- In load, compute and update phase

Edge Partitioning

- Assign vertices so each GPU contains consecutive vertices
- Store vertex properties in array layout in zero-copy memory
- Goal: Balance edges across partitions

Load Balance Model

Start with Edge Partition

Dynamic Graph Re-partitioning Strategy

- Global phase and local phase
- Function that calculates amount of work for each vertex (initially unknown and estimated)
- Update function at end of iteration
- Compute new partition and see if cost decrease is bigger than cost of repartitioning

Load Balance Model

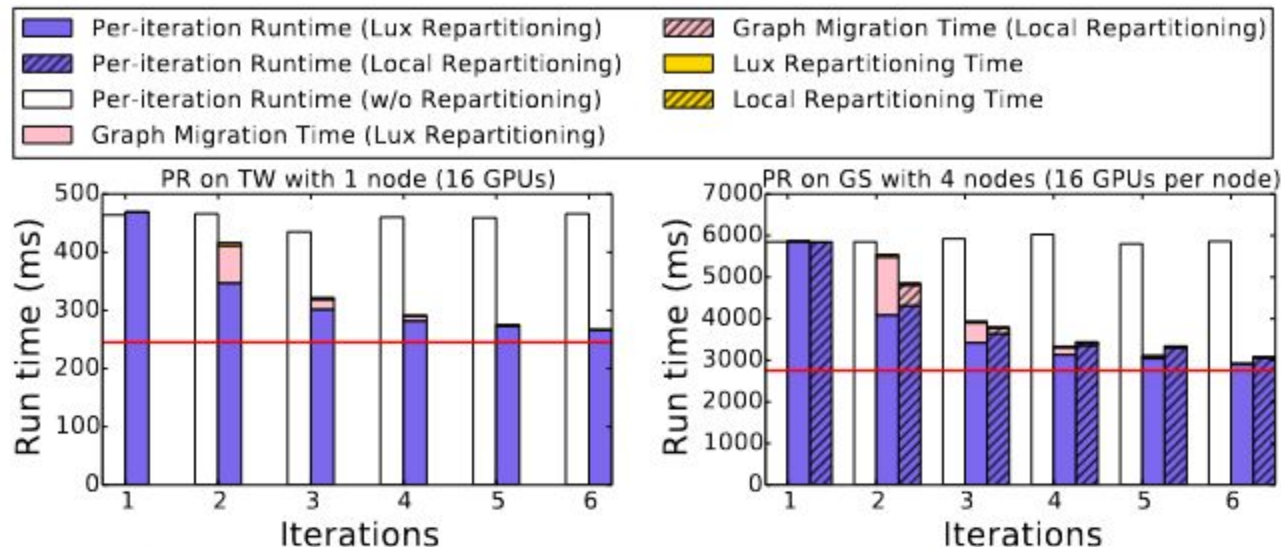


Figure 18: Performance comparison for different dynamic repartitioning approaches. The horizontal line shows the expected per-iteration run time with perfect load balancing.

Performance Model

Model performance of each execution mode by four steps

1. Load
2. Compute
3. Update
4. Inter-Node Transfer

Most of these proportional to amount of data / number of edges

Estimate performance for push vs pull and select faster execution mode

Evaluation and Performance

Single GPU

- Almost Matches (or outperforms) performance of other GPU graph processing frameworks
- Overhead from loading data to and from zero copy memory

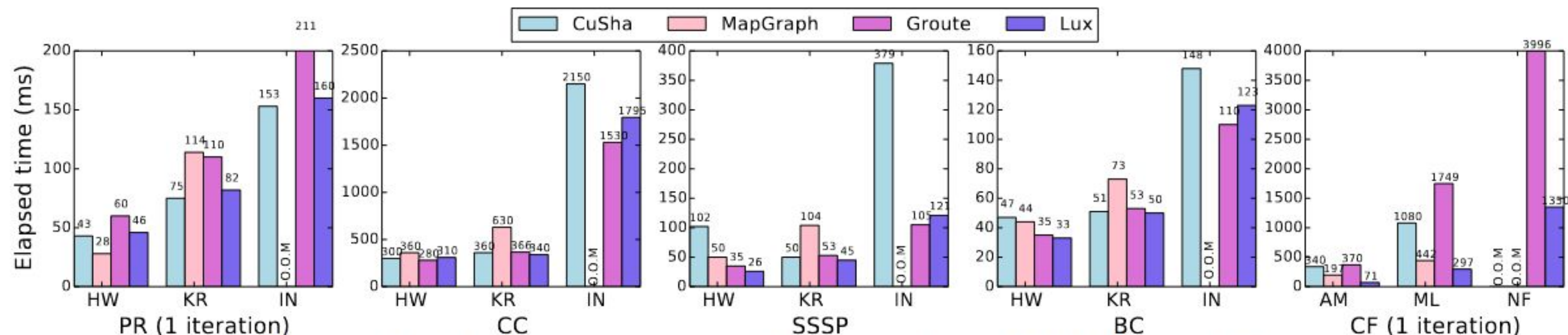


Figure 15: Performance comparison on a single GPU (lower is better).

Evaluation and Performance

Multi-GPU (vs Multi-CPU and other Multi-GPU)

- Outperforms most

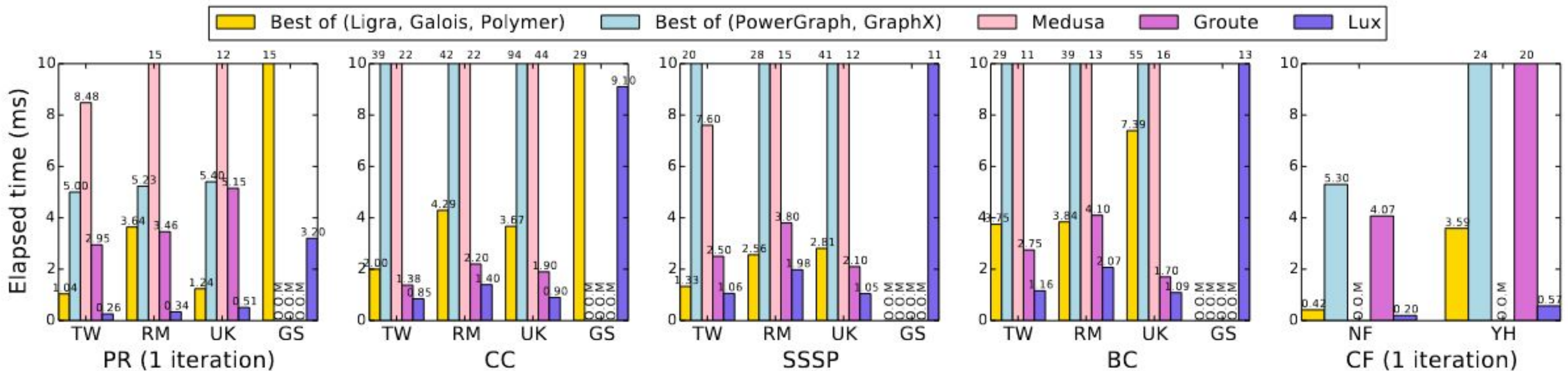


Figure 16: The execution time for different graph processing frameworks (lower is better).

Evaluation and Performance

Multi-GPU (vs Multi-CPU and other Multi-GPU)

- Outperforms most

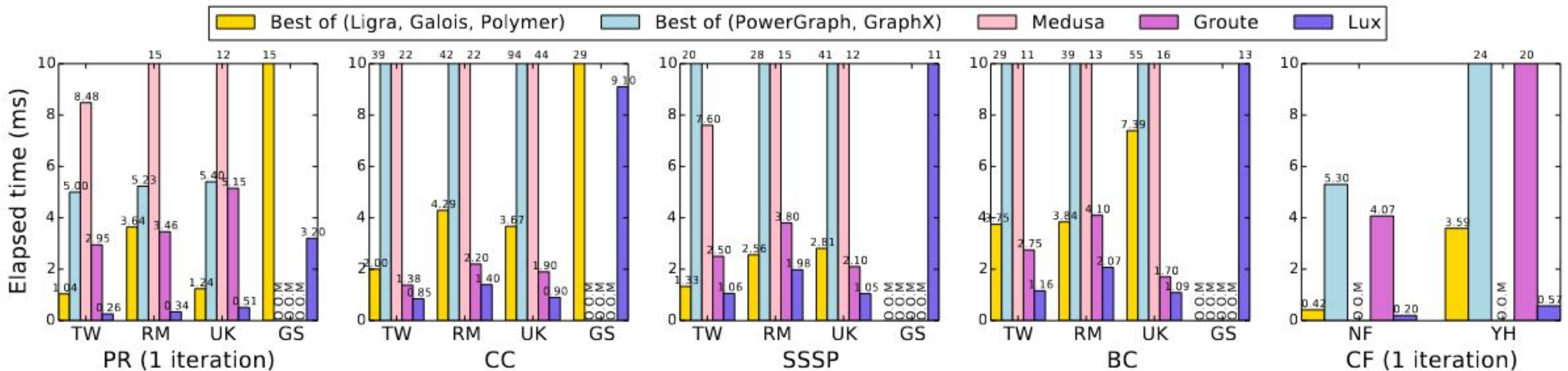


Figure 16: The execution time for different graph processing frameworks (lower is better).

Drawbacks

GPUs less cost efficient when scaled

Inaccurate in estimating performance of push model

- Frontier queue resulting in load imbalance

Overhead in data transfers and partitioning

- Masked by huge reduction in computation time

Fault Tolerance (?)

Algorithms to Modify Graph

Drawbacks

Table 3: The cost for a Lonestar5 CPU and an XStream GPU machine, as well as their cost efficiency. The cost efficiency is calculated by dividing the runtime performance (i.e., iterations per second) by machine prices.

Machines	Lonestar5	XStream (4GPUs)	XStream (8GPUs)	XStream (16GPUs)
Machine Prices (as of May 2017)				
CPUs [4, 3]	15352	3446	3446	3446
DRAM [8]	12784	2552	2552	2552
GPUs [7]	0	20000	40000	80000
Total	28136	25998	45998	85998
Cost Efficiency (higher is better)				
PR (TW)	0.20	0.84	0.64	0.45
CC (TW)	0.18	0.26	0.21	0.14
SSSP(TW)	0.14	0.25	0.20	0.10
BC(TW)	0.14	0.30	0.18	0.10
CF (NF)	0.85	1.07	0.68	0.58

Questions?