A Distributed Multi-GPU System for Fast Graph Processing

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

Presented by
Ridwan Muheeb
rm2084

Historical Contexts

- The rapidly growing sizes of real-world graphs and recent advances in computing processing hardware have inspired the implementation of faster and scalable graph analysis tools on GPUs.
- For instance:
  - Facebook 2.96 billion monthly active users
  - **2.61 billion pages** world wide web graphs
- ***Why GPUs?***
- GPU offers high bandwidth memory access and massive degree parallelism suitable for largescale graph data processing.
- Unfortunately, graph computations' have irregular data access, conditional branches, and the cost of moving data from host CPU to GPU poses significant challenge.

Motivation

- Limited work has been done to exploit high memory bandwidth of multiple GPUs for graph processing. Because
  - Compatibility issues – traditional data placement, transfer and load balancing approaches are not optimal for multiple GPU clusters.
  - Frameworks algorithms interfering with runtime optimizations necessary for GPU performance.

Solution

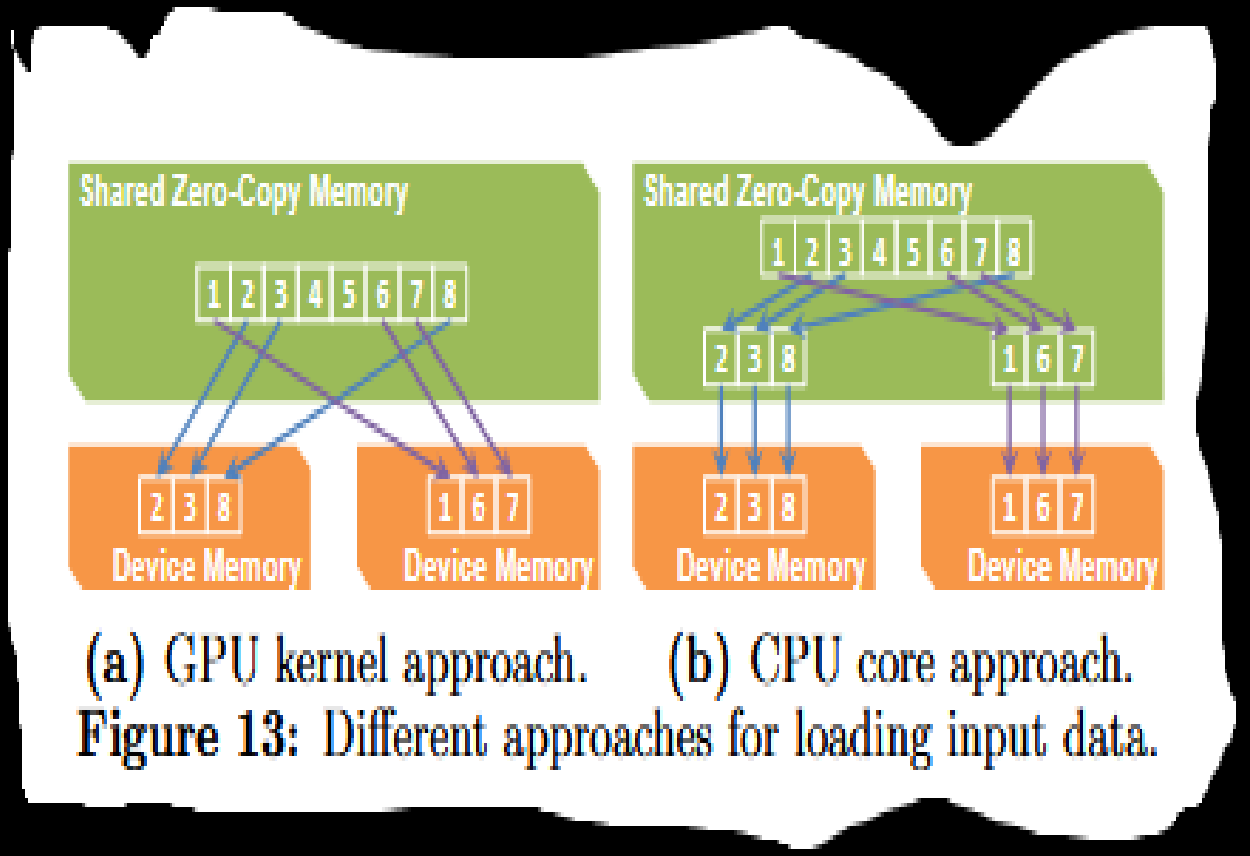
- *Lux*, a distributed system that exploit aggregate memory bandwidth and hierarchy of multi-GPU for fast graph processing

Key Ideas

- Graph data is distributed across the memories of multiple DRAM and GPUs to reduce data transfers within the memory hierarchy.
- Provides two execution models that delays updating vertices until the end of every iteration effectively ensuring no communication between GPUs during an epoch.
  - *Pull* - for improving algorithmic optimization and
  - *Push* - to enhance hardware (GPU) optimizations.
- Achieved load balancing by dynamically repartitioning graphs based on runtime performance and detection of workload imbalance.

Implementation

- Data placement and execution models were built on the Legion's abstraction developed by Michael et al (2012).
- Two approaches are used to load input data.
  - **GPU kernel approach** – kernel copies vertices from shared zero-copy memory to the device memory. Done entirely by the GPU, no CPU is involved. Used for pull based execution model.
  - **CPU core approach** – collaboratively gather vertices using CPU and do bulk transfer to the GPU. Used in push based execution model.



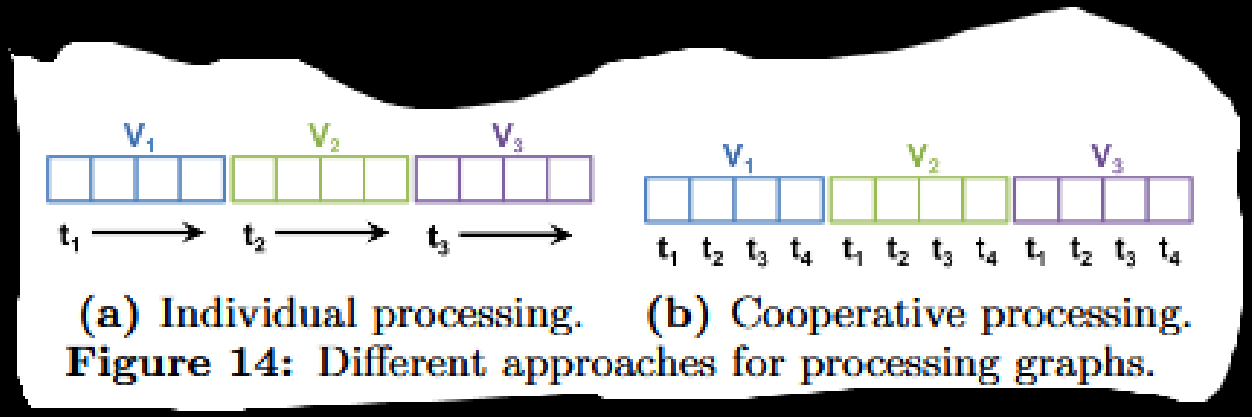
(a) GPU kernel approach. (b) CPU core approach.

Figure 13: Different approaches for loading input data.

Source: Jia et al (2017)

Implementation

- Combines cooperative and individual threads processing to achieve coalesced memory access.



Source: Jia et al (2017)

Performance – Single GPU Results

Source: Jia et al (2017)

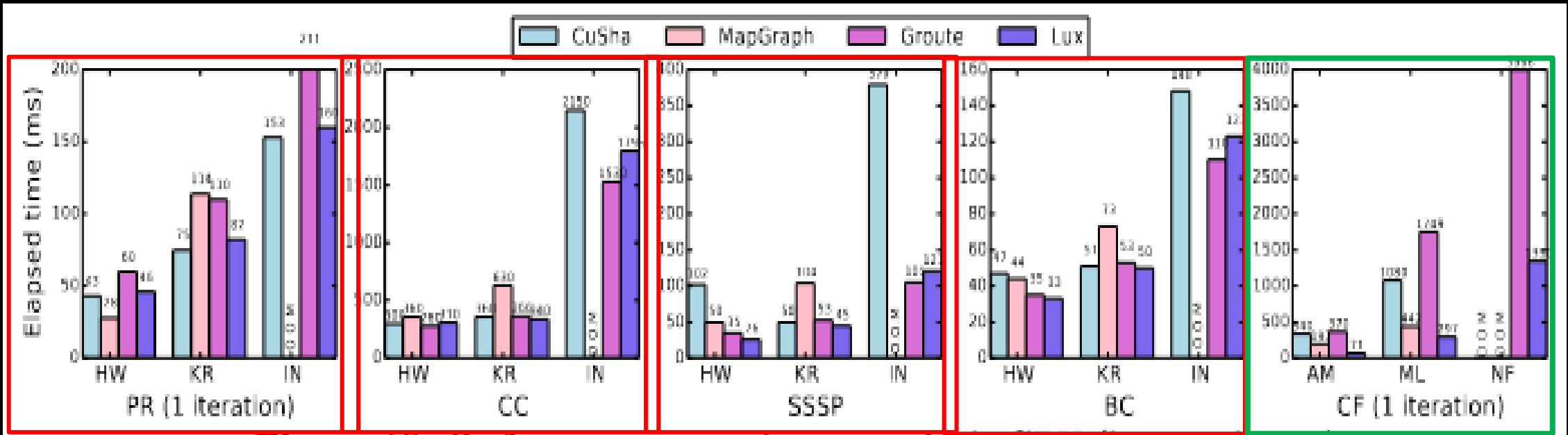


Figure 15: Performance comparison on a single GPU (lower is better).

- Lagged behind other similar frameworks in PageRanking (PR), Connected Components (CC), Single Source Shortest Path (SSSP) and Betweeness Centrality (BC).
- Obtained 1.5-5x speedup compared to other similar frameworks in Collaboratively Filtering

Performance – Multiple GPU Results

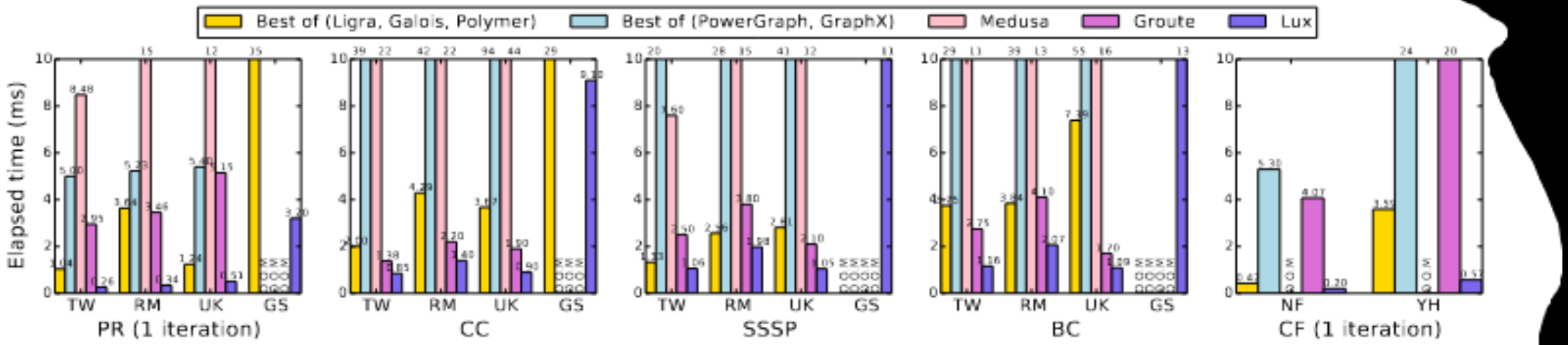


Figure 16: The execution time for different graph processing frameworks (lower is better).

Source: Jia et al (2017)

- Achieved 1.3-7x speedup when compared with Ligra, Galois, Polymer.
- Unfortunately, Low speedup caused by performance overhead due to overlapping operations.
- Outperforms Medusa and Groute by 1.5-2.5x in CC, SSSP, and BC and 10-20x in Page Ranking and Collaborative Filtering.
- Improvements made possible by coalesced memory access.

Performance – Load Balancing

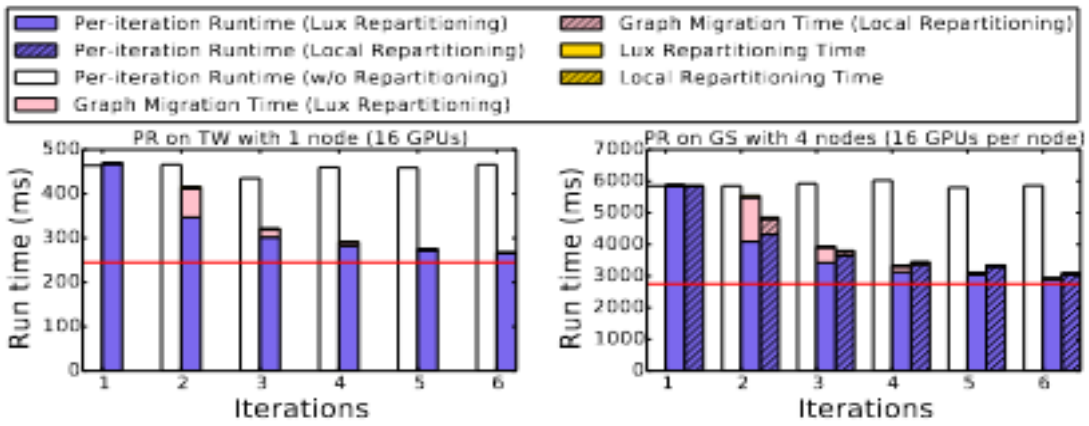


Figure 18: Performance comparison for different dynamic repartitioning approaches. The horizontal line shows the expected per-iteration run time with perfect load balancing.

Source: Jia et al (2017)

- Dynamic repartitioning reduced runtime by half.
- Although balancing workload across multiple nodes caused twice migration cost during initial epochs.

Pros

- Impressively harnessed multiple GPUs properties for faster largescale graph processing.
- Load balancing achieved using dynamic repartitioning.
- Combination of different approaches to achieve load balance was impressive.

Criticisms

- Design and experiments too focused on a single node.
- System susceptible to performance overhead because of overlapping operations.
- Overall performance wasn’t significant but mere improvements on existing system.
- Dynamic repartitioning do not favour irregular per iteration workloads e.g. Single Source Shortest Path (SSSP).

Future work findings

- Later works suggest the system lacks the capability to be adopted for graph neural networks processing.
- Further optimizations are required to support machine learning graph models training.

Bibliography

- Shi, Xuanhua, et al. "Graph processing on GPUs: A survey." *ACM Computing Surveys (CSUR)* 50.6 (2018): 1-35.
- https://www.worldwidewebsize.com/
- Bauer, Michael, et al. "Legion: Expressing locality and independence with logical regions." *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012.
- Jia, Zhihao, et al. "A distributed multi-gpu system for fast graph processing." *Proceedings of the VLDB Endowment* 11.3 (2017): 297-310.
- Ma, Lingxiao, et al. "{NeuGraph}: Parallel Deep Neural Network Computation on Large Graphs." *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019.
- Pan, Yuechao. *Multi-GPU Graph Processing*. Diss. University of California, Davis, 2019.