A Distributed Multi-GPU System for Fast Graph Processing

Z. Jia, Y. Kwon, G. Shipman, P. McCormick, M. Erez, A. Aiken

Presented by Oliver Hope
What is Lux? / Contributions of paper

Computational Model:
- 2 execution models
- A dynamic repartitioning strategy
- A performance model for parameter choice

Implementation:
- Working code
- Benchmarked on different algorithms
- Comparisons to different platforms
Motivation / Prior Work

- Lux: A graph processing framework to run on multi-GPU clusters
- Prior work for:
  - Single-node CPU
  - Distributed CPU
  - Single-node GPU
- Prior work cannot be adapted easily to GPU clusters
  - Data placement (heterogeneous memories)
  - Optimisation interference
  - Load-balancing does not map across from CPUs
Abstraction

- Iteratively modifies subset of graph until convergence
- Edges and vertices have properties
- 3 stateless functions to implement:
  - `void init(Vertex v, Vertex v^{old})`
  - `void compute(Vertex v, Vertex u^{old}, Edge e)`
  - `bool update(Vertex v, Vertex v^{old})`
Abstraction: Pull vs Push

- Does not require additional synchronisation
- Takes advantage of GPU caching and aggregation

**Algorithm 1** Pseudocode for generic pull-based execution.

1: while not halt do
2:    halt = true \(\triangleright\) halt is a global variable
3:    for all \(v \in V\) do in parallel
4:        init\((v, v^{old})\)
5:        for all \(u \in N^-(v)\) do in parallel
6:            compute\((v, u^{old}, (u, v))\)
7:        end for
8:        if update\((v, v^{old})\) then
9:            halt = false
10:    end if
11: end for
12: end while

**Algorithm 2** Pseudocode for generic push-based execution.

1: while \(F \neq {}\) do
2:    for all \(v \in V\) do in parallel
3:        init\((v, v^{old})\)
4:    end for
5: \(\triangleright\) synchronize\((V)\)
6:    for all \(u \in F\) do in parallel
7:        for all \(v \in N^+(u)\) do in parallel
8:            compute\((v, u^{old}, (u, v))\)
9:        end for
10:    end for
11: \(\triangleright\) synchronize\((V)\)
12: \(F = {}\)
13:    for all \(v \in V\) do in parallel
14:        if update\((v, v^{old})\) then
15:            \(F = F \cup \{v\}\)
16:        end if
17:    end for
18: end while

- Better for rapidly changing frontiers
Task Execution

- **Pull-based:**
  - Single GPU kernel for all steps
  - Scan-based gather to resolve load imbalance

- **Push-based:**
  - Separate kernel for all 3 steps
  - All updates have to use device memory to avoid races

- Computation can overflow to CPU+DRAM if not enough space
Lux uses Edge partitioning

- Idea: Assign equal number of edges to each partition
- Each partition holds contiguously numbered vertices and the edges pointing to them
- So GPU can coalesce reads and writes to consecutive memory
- Very fast to compute (e.g. vs vertex-cut)
Dynamic Repartitioning

Figure: Estimates of $f(x) = \sum_{i=0}^{x} w_i$ used to pick pivot vertices.

1. Collect $t_i$ per $P_i$, update $f$, calculate partitioning
2. Compare $\Delta_{gain}(G)$ (improvement) vs $\Delta_{cost}(G)$ (inter-node transfer)
3. Globally repartition depending on 2
4. Local repartition
Performance Model

- To preselect an execution model and runtime configuration
- Models performance for a single iteration
- Sums together estimates for:
  1. Load time
  2. Compute time
  3. Update time
  4. Inter-node transfer time

(a) Pull-based executions (PR).

(b) Push-based executions (CC).
Different hardware used for shared memory and GPU testing. Tried to get best attainable performance from every system.
Criticisms

- Abstract claims up to 20x speedup over shared-memory systems (more like 5-10)
- “Most popular graph algorithms can be expressed in Lux” Does not assess what cannot be.
- “For many applications ... identical implementation for both push and pull”
- Did not test the overflow processing to CPU feature
- For evaluation all parameters were highly tuned. Can’t guarantee others were as tuned as Lux.