Lux
A Distributed Multi-GPU System for Fast Graph Processing
Z. Jia, Y. Kwon, G. Shipman, P. McCormick, M. Erez, A. Aiken
VLDB 2018
Background

Prior work

- Distributed CPU-based systems: Pregel, PowerGraph, GraphX...
- Single-node CPU-based systems: Ligra, Galois, and Polymer...
- Single-node GPU-based systems:
  - Single GPU: CuSha, MapGraph...
  - Single machine: Groute, Medusa, GTS...
- Lux: Distributed multi-GPU system that achieves fast graph processing
Background

Motivation

- GPU vs CPU
- GPUs provide much higher memory bandwidth than today’s CPU architectures.
- Prior work cannot be easily adapted to multi-GPU clusters:
  - graph placement and data transfers
  - Optimisation interference
  - load balancing
- Lux: Distributed multi-GPU system that achieves fast graph processing
Introduction

Graph Tasks

• PageRank (PR)
• connected components (CC)
• single-source shortest path (SSSP)
• betweenness centrality (BC)
• collaborative filtering (CF)

• Up to 20× speedup over Ligra, Galois, and Polymer
• Two orders of magnitude speedup over PowerGraph and GraphX

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

Programming Model

- Gather-Apply-Scatter concepts, Vertex-centric algorithms
- Vertex contain mutable states
- Edges do not contain states AND topology cannot change

```java
interface Program(V, E) {
    void init(Vertex v, Vertex v\textsuperscript{old});
    void compute(Vertex v, Vertex v\textsuperscript{old},
                 Edge e);
    bool update(Vertex v, Vertex v\textsuperscript{old});
}
```
Lux Details

Two Execution models

- **Push** execution model
  - optimize algorithmic efficiency

- **Pull** execution model
  - enable important GPU optimizations
  - applications with a large proportion of active vertices over iterations benefit substantially
    (e.g., PageRank, collaborative filtering)

```c
#define Vertex (rank: float)
#define init(Vertex v, Vertex vold) {
    v.rank = 0
}
#define compute(Vertex v, Vertex vold, Edge e) {
    atomicAdd(&v.rank, vold.rank)
}
#define update(Vertex v, Vertex vold) {
    v.rank = (1 - d) / |V| + d * v.rank
    v.rank += 1 / deg²(v)
    return (lv.rank - vold.rank) > δ
}
```

```c
#define Vertex (rank, delta: float)
#define init(Vertex v, Vertex vold) {
    v.delta = 0
}
#define compute(Vertex v, Vertex vold, Edge e) {
    atomicAdd(&v.delta, vold.delta)
}
#define update(Vertex v, Vertex vold) {
    v.rank = vold.rank + d * v.delta
    v.delta = d * v.delta / deg²(v)
    return (lv.delta) > δ
}
```
**Lux Details**

**Distributed Graph Placement and Data Transfers**

- vertex-cut partitioning: PowerGraph, GraphX
  - takes too long
  - not a good estimate of data transfers
- edge partitioning
  - each partition holds contiguously numbered vertices and the edges pointing to them
  - GPU can coalesce reads and writes to consecutive memory
  - very efficient

---

**Figure 7:** Edge partitioning in Lux: a graph is divided into 4 partitions, which are assigned to 4 GPUs on 2 nodes.

\[
INS(P_i) = \{u | (u, v) \in P_i \} \\
ONS(P_i) = \{v | (u, v) \in P_i \}
\]
Lux Details

Load Balancing

- Static load balancing: Pregel, Giraph, GraphLab, PGX.D
- Dynamic load balancing: Giraph, Presto

A Dynamic graph repartitioning strategy
- global: multiple nodes
- local: multiple GPUs on a node

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

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

Performance Model

- To preselect an execution model and runtime configuration
- Models performance for a single iteration

Figure 9: Data flow for one iteration.

Figure 17: The execution time for different Lux configurations (lower is better). $x$ and $y$ indicate the number of nodes and the number of GPUs on each node.
Opinions

key takeaway

• Lux, a distributed multi-GPU system that achieves fast graph processing by:
  
  • a distributed **graph placement** to minimize **data transfers** within the memory hierarchy.

  • two **execution models** optimizing algorithmic efficiency and enabling GPU optimizations.

  • a dynamic graph repartitioning strategy that achieves good **load balance across GPUs**.

  • a **performance model** that chooses the number of nodes and GPUs for the best possible performance.
Opinions

Criticism

• The paper is hard to follow
• Absence of fault tolerance
• Abstract claims up to 20x speedup over shared-memory systems (more like 5-10)
• For evaluation all parameters were highly tuned. Can’t guarantee others were as tuned as Lux
• The prediction for the push-based execution is not as accurate as the pull-based execution
Thanks for listening!

Q&A