(Lux) A distributed multi-GPU system for fast graph processing

Victor
Oct 2020
Background – Prior work

• Put entire graph representation in DRAM
• Large shared memory with multi-core CPU on one machine
• Distributed memory on multiple machines
• Optimize data access across machines via graph partitioning strategy and data locality
• Multiple GPUs on one machine
Background – Limitations of existing CPU-based approaches

- GPU has much larger memory access bandwidth than CPU
- CPU memory hierarchy and GPU memory hierarchy is different so cannot directly use CPU distributed memory systems

![Diagram showing CPU and GPU memory hierarchy]

**Figure 1:** Multi-GPU node architecture.
Background – Limitations of existing GPU-based approach

- Only works only on one GPU/one machine
- Slow memory access from DRAM
Lux – programming model

- Similar to Pregel, Gather-Apply-Scatter concepts
- Vertex-centric algorithms
- Vertex contain mutable (in terms of algorithm reasoning) 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 u\textsuperscript{old},
                 Edge e);
    bool update(Vertex v, Vertex v\textsuperscript{old});
}
```
Lux – programming model

- Pull vs push
- (Graphs are all directed!)
- Pull allows a vertex’s compute() to get state updates from all in-edges
- Push allows a vertex’s compute() to push state updates from itself to all out-edges
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:     for all \(u \in F\) do in parallel
6:         for all \(v \in N^+(u)\) do in parallel
7:             compute\((v, u^{old}, (u, v))\)
8:         end for
9:     end for
10:     \(F = \{\}\)
11:     for all \(v \in V\) do in parallel
12:         if update\((v, v^{old})\) then
13:             \(F = F \cup \{v\}\)
14:         end if
15:     end for
16: end while

\(\triangleright\) synchronize\((V)\)
Runtime system – Graph partition

• Try to put whole representation into GPU device memory
• (Spillover in DRAM zero-copy memory)
• Edge partitioning: each partition holds roughly the same no. of edges
• Also, each partition holds vertices with consecutive ranges of IDs
• Partition contains all edges that point to a vertex within the partition
• Consecutive IDs cause memory access to have higher chance of being consecutive, and GPU memory hardware can coalesce multiple individual accesses into one range access (?)
Runtime system – Graph partition

• Each partition has in-neighbour set (INS), a set of all neighbours that point to some vertex in that partition
• Each partition has out-neighbour set (ONS), a set of all vertices within the partition which is pointed at by some neighbours
• ONS is the vertices contained in the partition if using edge partitioning
Runtime system – Task execution

• All vertex mutable states are in DRAM zero-copy memory
• Copy vertices state of INS set to device memory
Runtime system – Task execution – Pull-based

• One kernel for all three stages
• One thread for one vertex to execute init() and update(), which enables coalesced memory access
• (Split the vertices into groups) and use a thread block for each group
• Thread block cooperatively execute the compute() functions for vertice group to even out edge count imbalance
• Thread block only change vertices states within their group, updates stored and aggregated in shared memory (don’t have to write back to slower device memory)
Runtime system – Task execution – Push-based

• One kernel for each stage
• One thread for one vertex to execute init() and update(), which enables coalesced memory access
• One thread for each vertex in the INS to execute compute()
• “In the push model, since threads may potentially update any vertex, all updates go to device memory to eliminate race conditions and provide deterministic results. “ (?)

• All updates present in device memory are write back to zero-copy memory so that updates are visible to all GPU (and external nodes)
Runtime system – Data synchronization

• Each node compute the vertices that it needs but are on remote nodes
  • **Update set (UDS)** Union of INS of all partitions (on a node) \ union of ONS of all partitions
  • Then send/receive states for these nodes
Runtime system – Dynamic repartitioning

• Measures the actual execution time of each partition

• Calculate the need for repartitioning based on heuristics derived from statistics of execution time

• Then assumes that execution time of each vertex is proportional to its number of in-edges and calculate a better partition boundary (while maintaining the property that vertices IDs within each partition are still continuous)

• Then moves data around if boundaries changed

• Use same method to repartition within each node
Performance modelling

• Estimate the execution time based on parameters
• Number of nodes
• Number of GPUs per node
• Size of INS of each partition
• Size of UDS of each node
Performance modelling – Pull based

• Load time proportional to sum of $|\text{INS}|$ / number of nodes
• Compute time proportional to total edge count / total number of partitions
• Intra-node data transfer time is ignored because update can be done once a vertex’s compute() functions have all executed and it overlap with compute()
• Data synchronization time is proportional to sum of $|\text{UDS}|$
Performance modelling – Push based

• Load time combined with compute() time, where loading data overlaps with compute() kernel
• Compute() is executed by a thread as long as vertex state of INS is transferred to device memory
• Compute time proportional to total edge count / total number of partitions
• Intra-node data transfer is ignored because it is significantly shorter than compute()
• Data synchronization time is proportional to sum of |UDS|
Implementation details – Loading input

- Pull model: kernel on each GPU load data from zero-copy memory to device memory
- Push model: CPU coalesce vertices and then GPU kernels copies coalesced data from zero-copy memory to device memory
- Because Push model overlaps loading data with compute

![Diagram](image)

**Figure 13:** Different approaches for loading input data.
Implementation details – Coalescing memory access

• Use arrays for storing vertex states (such as vectors)
• Use cooperative threads to load vectors onto shared memory
• Use individual thread for processing a single edge
• Best of both world
Implementation details – Cache optimisation

• Copy data from zero-copy memory to device memory
• Cache and aggregate local vertex update in GPU shared memory within a thread block
Evaluation – Comparison with other frameworks

• PageRank, connected components, single-source shortest path, betweenness centrality, and collaborative filtering
• On-par with others for single GPU implementation
• Superior performance for distributed multi-CPU and multi-GPU systems

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

- Dynamic repartitioning is expensive for first few iterations but becomes small thereafter.
- Push model performs better than pull model for algorithms where not all vertices are active.
Personal opinions

- Paper was hard to follow
- Lux is an evolution from existing ideas
- Programming model is essentially the same as Pregel, GAS
- Multi-GPU comes from Groute
- Performance gain comes from cores within a GPU and from fast memory access on device memory
- Distributed systems enhancements: fault tolerance, or dealing with scheduling
Q & A