A Distributed Multi-GPU System for Fast Graph Processing

Zhihao Jia
Stanford University
zhihao@cs.stanford.edu

Yongkee Kwon
UT Austin
yongkee.kwon@utexas.edu

Galen Shipman
LANL
gshipman@lanl.gov

Pat McCormick
LANL
pat@lanl.gov

Mattan Erez
UT Austin
mattan.erez@utexas.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

ABSTRACT

We present Lux, a distributed multi-GPU system that achieves fast graph processing by exploiting the aggregate memory bandwidth of multiple GPUs and taking advantage of locality in the memory hierarchy of multi-GPU clusters. Lux provides two execution models that optimize algorithmic efficiency and enable important GPU optimizations, respectively. Lux also uses a novel dynamic load balancing strategy that is cheap and achieves good load balance across GPUs. In addition, we present a performance model that quantitatively predicts the execution times and automatically selects the runtime configurations for Lux applications. Experiments show that Lux achieves up to 20× speedup over state-of-the-art shared memory systems and up to two orders of magnitude speedup over distributed systems.

PVLDB Reference Format:
Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. A Distributed Multi-GPU System for Fast Graph Processing. *PVLDB*, 11(3): xxxx-yyyy, 2017.
DOI: 10.14778/3157794.3157799

1. INTRODUCTION

Because graph applications have a high ratio of irregular memory accesses to actual computation, graph processing performance is largely limited by the memory bandwidth of today's machines. Prior work (e.g., PowerGraph [22], GraphX [23], Ligra [37], and Galois [33]) has focused on designing shared memory or distributed memory graph processing frameworks that store the entire graph in DRAM on a single machine or in the distributed DRAM in a cluster.

GPUs provide much higher memory bandwidth than today's CPU architectures. Nodes with multiple GPUs are now ubiquitous in high-performance computing because of their power efficiency and hardware parallelism. Figure 1 illustrates the architecture of typical multi-GPU nodes, each of which consists of a host (CPUs) and several GPU devices connected by a PCI-e switch or NVLink [6]. Each

Proceedings of the VLDB Endowment, Vol. 11, No. 3

DOI: 10.14778/3157794.3157799

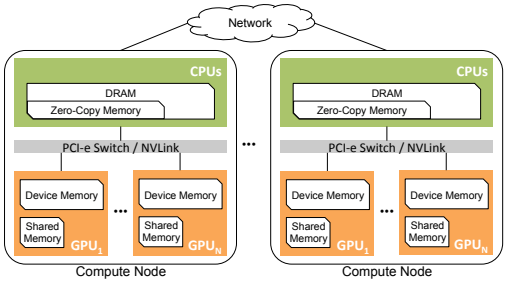


Figure 1: Multi-GPU node architecture.

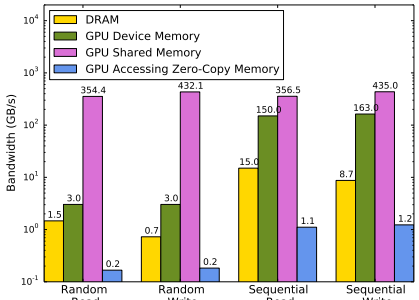


Figure 2: Memory bandwidth in a multi-GPU architecture.

GPU is able to directly access its local relatively large device memory, much smaller and faster shared memory, and a small pinned area of the host node's DRAM, called *zero-copy memory.* Figure 2 shows the bandwidth of different memories on a multi-GPU node. Compared to DRAM, a single GPU's device memory provides 2-4× more bandwidth for random access and 10-20× more bandwidth for sequential access. In addition, GPU shared memory provides 20-50× more bandwidth for sequential access and 200-600× more bandwidth for random access.

Despite the high memory bandwidth, there has been limited work on designing multi-GPU graph processing frameworks due to the heterogeneity and complex memory hierarchy of multi-GPU nodes. Existing graph processing frameworks for multi-CPU clusters cannot be easily adapted to multi-GPU clusters for three reasons.

First, native data placement and transfer strategies for multi-CPU systems do not work for multi-GPU clusters. Existing CPU-based approaches [37, 33, 22, 31] store the graph in the DRAM of one or multiple nodes and optimize data transfers between DRAM. In contrast, in GPU clusters, there are choices in distributing the graph among GPU device memory, GPU shared memory, zero-copy memory, and DRAM on each node. To realize benefits from the aggregate

performance of multiple GPUs, it is critical to also take advantage of the locality in these hierarchical architectures.

Second, current distributed graph processing systems focus on push-based operations, with each core processing vertices in an active queue and explicitly pushing updates to its neighbors. Examples include message passing in Pregel, scatter operations in gather-apply-scatter (GAS) models, and VertexMaps in Ligra. Although efficient at the algorithmic level, push-based operations interfere with runtime optimizations important to GPU performance, such as locally aggregating vertex updates.

Third, we have found that previous approaches for achieving efficient load balancing for CPUs work poorly for multi-GPU systems due to architectural differences.

We present Lux, a distributed multi-GPU system that achieves fast graph processing by exploiting the aggregate memory bandwidth across a multi-GPU cluster. In Lux, the entire graph representation is distributed onto the DRAM and GPU memories of one or multiple nodes. The distributed graph placement is designed to minimize data transfers within the memory hierarchy. Lux provides both a *push* execution model that optimizes algorithmic efficiency and a *pull* execution model that enables important GPU optimizations. We find that applications with a large proportion of active vertices over iterations (e.g., PageRank, collaborative filtering) benefit substantially from the pull execution model. In addition, we introduce a *dynamic graph repartitioning* strategy that is simple and inexpensive and achieves good load balance across GPUs. Finally, we present a performance model that quantitatively predicts the Lux execution time and automatically selects an execution model for a given application and input graph.

The main contributions of this paper are as follows:

- We present Lux, a distributed multi-GPU system that achieves fast graph processing by exploiting locality and the aggregate memory bandwidth on GPUs.
- We propose two execution models optimizing algorithmic efficiency and enabling GPU optimizations.
- We propose a dynamic graph repartitioning strategy that enables well-balanced workload distribution with minimal overhead. We show that dynamic repartitioning improves performance by up to 50%.
- We present a performance model that provides insight into choosing the number of nodes and GPUs for the best possible performance. Our performance model can select the best configurations in most cases.
- We present an implementation of Lux that outperforms state-of-the-art graph processing engines. In particular, we show that Lux achieves up to 20× speedup over Ligra, Galois, and Polymer and two orders of magnitude speedup over PowerGraph and GraphX.

2. RELATED WORK

Distributed CPU-based systems. A number of distributed graph processing frameworks have been proposed. Pregel [31] is a bulk synchronous message passing framework. Within each iteration, each vertex receives messages from its neighbors from the previous iteration and sends messages to its neighbors. PowerGraph [22] decomposes graph computations into iterative GAS (gather, apply, and scatter) operations. GraphX [23] adopts the GAS model from PowerGraph and is built on top of Spark [42]. HSync [35] achieves fast synchronous graph processing by using an efficient concurrency control scheme. PAGE [36] is a partition aware engine that monitors the runtime performance characteristics and dynamically adjusts resources. MOCgraph [45] exploits a message online computing model that processes messages as long as they arrive.

The above frameworks have proven efficient for distributed multi-CPU clusters. However, as described in Section 1, their strategies would not work as well for multi-GPU clusters. Lux differs from previous distributed systems in how it optimizes data placement and transfers (Section 4), how it balances workload among different partitions (Section 5), and how it enables GPU optimizations (Section 7).

Single-node CPU-based systems. Previous work [34, 37, 33] has shown that shared memory CPU-based systems, by eliminating inter-node communication, are typically much faster than distributed CPU-based systems. Ligra [37] is a lightweight CPU-based framework for shared memory. Polymer [43] integrates NUMA support into Ligra's model. Galois [33] is designed with a rich set of schedulers and data structures for irregular computations (especially for graph processing). The above systems store the entire graph in the shared CPU memory and let multiple CPU threads cooperatively process the graph. Directly adopting these approaches to multi-GPU machines can result in poor performance due to the insufficient bandwidth between CPU memory and GPU devices (as shown in Figure 2). For executions on a single node, Lux achieves up to 20× speedup over the shared memory systems. This shows that a single, unified shared memory is not the fastest possible platform; again, the difference is in both minimizing data movement and taking advantage of GPU bandwidth.

GPU-based systems. Recently, GPU-based frameworks that are specific for a single GPU or a single machine have been proposed. MapGraph [21] provides a high level API for writing high performance graph analytics for a single GPU. CuSha [26] introduces two graph representations: G-shards and concatenated windows (CW). G-shards are used to allow *coalesced* memory accesses, while CW achieves higher GPU utilization by grouping edges with good locality into shards. Gunrock [41] implements a programming abstraction centered on operations on a vertex or edge frontier. These approaches are designed with the assumption that all processing is done on a single GPU.

Groute [14] is an asynchronous model for multiple GPUs on a single node, designed for irregular computations. Groute manages computation at the level of individual vertices, which allows Groute to exactly capture all irregular parallelism with no wasted work, but potentially incurs system overhead because the useful work per vertex is generally small. Groute uses a ring topology for inter-GPU transfers, which introduces extra data movement when data is moved between GPUs not adjacent in the ring ordering.

GTS [27] is a multi-GPU framework that stores mutable data in device memory while streaming topology data (i.e., edges in the graph) to GPUs from DRAM. While able to process large graphs, GTS has two limitations. First, execution is apparently limited to a single node. Second, GTS streams the edge list to GPUs from DRAM, and therefore performance is bottlenecked by the PCI-e bandwidth.

Medusa [44] simplifies implementation of GPU programs by providing user-defined APIs and automatically executing these APIs in parallel on GPUs.

```
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);
}
```

Figure 3: All Lux programs must implement the state-less `init`, `compute` and `update` functions.

3. Lux ABSTRACTION

Lux targets graph applications that can be expressed as iterative computations: the application iteratively modifies a subset of the graph and terminates when a convergence test succeeds. This target is similar to most graph processing frameworks [37, 22, 23, 33]. A directed graph $G = (V, E)$ has vertices V and directed edges E. Each vertex $v \in V$ is assigned a unique number between 0 and $|V|-1$. $N^-(v)$ denotes the set of incoming neighbors of vertex v and $deg^-(v)$ denotes the in-degree of v. Similarly, $N^+(v)$ and $deg^+(v)$ denote the out-neighbors and out-degree of v.

Edges and vertices may have application specific *properties*. Edge properties are immutable, which helps minimize data movement, since Lux only needs to move updates to vertices. We find that this does not restrict the expressiveness of Lux; most popular graph algorithms can be implemented within the Lux framework. Lux provides a transaction-like model, where the properties of the vertices are read-only in each iteration, and updates become visible at the end of the iteration.

Computations in Lux are encoded as stateless programs implementing the Lux interface defined in Figure 3. By implementing the three interface functions, a computation is explicitly factored into the `init`, `compute`, and `update` functions. Lux provides two different execution models. Section 3.1 describes a pull model that optimizes the runtime performance for GPUs. Section 3.2 describes an alternative push model that optimizes algorithmic efficiency. We compare the two models in Section 3.3.

3.1 Pull Model

Algorithm 1 shows pseudocode for the pull model. For every iteration, the three functions are performed as follows.

Algorithm 1 Pseudocode for generic pull-based execution.

```
1: while not halt do
2:     halt = true                          ▷ halt is a global variable
3:     for all v ∈ V do in parallel
4:         init(v, v^old)
5:         for all u ∈ 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
```

First, Lux initializes the vertex properties for an iteration by running the `init` function on every vertex v. The vertices' properties from the previous iteration (denoted as v^{old}) are passed as immutable inputs to `init`. For the first iteration, v^{old} is v's initial value as set by the program.

Second, for an edge (u, v), the `compute` function takes the vertex properties of u from the previous iteration and the edge properties of (u, v) as its input and updates the properties of v. Note that the properties of u^{old} and (u, v) are immutable in the `compute` function. The order in which the `compute` function processes edges is non-deterministic. Furthermore, the `compute` function should support concurrent invocations to allow parallelism in Lux.

Finally, Lux performs the `update` function on every vertex v to finalize the computation and commit updates to v at the end of the iteration. Lux terminates if no vertex properties are updated in an iteration.

We use PageRank as a demonstration of using the Lux interface. PageRank computes the relative importance of webpages by taking as input a graph $G = (V, E)$, a damping factor $0 \leq d \leq 1$, and a convergence constraint δ. The rank property of all vertices is initially $\frac{1}{|V|}$. The following equation is iteratively applied to all vertices until the changes to the rank property drop to below δ:

$$PR(v) = \frac{1-d}{|V|} + d \sum_{u \in N^-(v)} \frac{PR(u)}{deg^+(u)} \quad (1)$$

This leads to a simple implementation for the pull model in Figure 4. The `init` function initializes the `rank` property of all vertices to 0. For a vertex v, the `compute` function sums the `rank` property of all in-neighbors of v from the previous iteration; atomic operations guarantee correctness for concurrent updates. The `update` function finalizes the computation for every vertex v and returns `true` if the difference of `v.rank` between adjacent iterations is above δ. Execution terminates when all `update` functions return `false`.

```
define Vertex {rank:float}
void init(Vertex v, Vertex v^old) {
  v.rank = 0
}
void compute(Vertex v, Vertex u^old, Edge e) {
  atomicAdd(&v.rank, u^old.rank)
}
bool update(Vertex v, Vertex v^old) {
  v.rank = (1 - d) / |V| + d * v.rank
  v.rank = v.rank / deg+(v)
  return (|v.rank - v^old.rank| > δ)
}
```

Figure 4: PageRank in the pull model.

```
define Vertex {rank, delta:float}
void init(Vertex v, Vertex v^old) {
  v.delta = 0
}
void compute(Vertex v, Vertex u^old, Edge e) {
  atomicAdd(&v.delta, u^old.delta)
}
bool update(Vertex v, Vertex v^old) {
  v.rank = v^old.rank + d * v.delta
  v.delta = d * v.delta / deg+(v)
  return (|v.delta| > δ)
}
```

Figure 5: PageRank in the push model.

3.2 Push Model

In the pull model, every vertex iteratively pulls potential updates from *all* its in-neighbors. This gathers updates to vertex properties and therefore allows GPU-specific optimizations, such as locally aggregating updates in GPU shared memory. Although efficient on GPUs, the pull model may be inefficient for applications that only update a small subset of vertices in each iteration. Lux provides an alternative push model that improves algorithmic efficiency.

Algorithm 2 Pseudocode for generic push-based execution.

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

The push model requires every vertex to push its updates when available to its out-neighbors. In the push model, since only vertices with updates need to push their new value to out-neighbors, we use a *frontier queue* to store the collection of vertices with updates. Our push/pull alternatives are inspired by the approach taken in [13] to the narrower case of bottom-up vs. top-down traversal of trees.

Algorithm 2 shows the pseudocode for the push model. A program needs to initialize the frontier queue F to include vertices that contain initial updates. The push-based execution terminates when F becomes empty. Note that the push model also requires a program to implement the `init`, `compute`, and `update` functions, providing the same interface to the program as the pull model. The push model performs the `init` and `update` functions on all vertices for every iteration, which involves the same amount of work as the pull model. However, the push model only runs `compute` on an edge (u, v) if $u \in F$. This substantially reduces the number of invocations to the `compute` function when F is small.

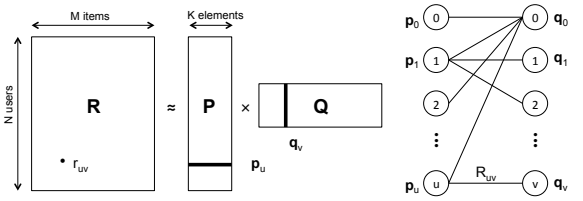
For many applications (e.g., shortest path and connected components), the program can provide an identical implementation for both the push and pull models and expect to get the same results at the end of every iteration. For PageRank, using the push model requires Lux to compute the delta of the `rank` property, and an implementation of PageRank in the push model is shown in Figure 5.

3.3 Comparison

The push model is better positioned to optimize algorithmic efficiency since it maintains a frontier queue F and only performs computation on edges coming from F. However, the pull model better utilizes GPUs for two reasons.

First, the push model requires two additional synchronizations in every iteration (shown as `synchronize(V)` in Algorithm 2). The first synchronization ensures a vertex $u \in F$ cannot push its updates until its out-neighbors $N^+(u)$ are initialized for this iteration. The second synchronization ensures a vertex $v \in V$ does not finalize and commit the values of its properties until Lux has run `compute` on all edges connecting to v, otherwise the `update` function might exclude some updates to vertices in that iteration.

Second, the pull model enables optimizations such as caching and locally aggregating updates in GPU shared memory.



(a) matrix-centric **(b)** graph-centric

Figure 6: Collaborative filtering.

For example, for each vertex $v \in V$, Lux can opportunistically run the `compute` function on its in-neighbors concurrently in a single GPU thread block, and all updates are locally cached and aggregated in GPU shared memory, which eliminates loads and stores to GPU device memory. Section 4.3 describes GPU execution in Lux in detail.

We find that for applications with relatively large frontiers (e.g., PageRank), the pull model has better performance. However, for applications with rapidly changing frontiers (e.g., connected components), the push model performs better. Section 8.4 provides a quantitative comparison between the pull and push models.

3.4 Other Algorithms

Besides PageRank, we also use a machine learning algorithm and three graph traversal algorithms to evaluate Lux.

Collaborative filtering is a machine learning algorithm used by many recommender systems to estimate a user's rating for an item based on a set of known (user, item) ratings. The underlying assumption is that user behaviors are based on a set of hidden features, and the rating of a user for an item depends on how well the user's and item's features match. Figure 6a and 6b show matrix-centric and graph-centric views of collaborative filtering. Given a matrix $\mathbf{R}$ of ratings, the goal of collaborative filtering is to compute two factors $\mathbf{P}$ and $\mathbf{Q}$ where each is a dense matrix.

Collaborative filtering is accomplished by incomplete matrix factorization [28]. The problem is shown in Equation 2, where u and v are indices of users and items, respectively.

$$\min_{\mathbf{p},\mathbf{q}} \sum_{(u,v)\in R} (\mathbf{R}_{uv} - \mathbf{p}_u^T \mathbf{q}_v)^2 + \lambda ||\mathbf{p}_u||^2 + \lambda ||\mathbf{q}_v||^2 \quad (2)$$

$\mathbf{R}_{uv}$ is the rating of the u^{th} user on the v^{th} item, while $\mathbf{p}_u$ and $\mathbf{q}_v$ are vectors corresponding to each user and item.

Matrix factorization is usually performed by using *stochastic gradient descent* (SGD) or *gradient descent* (GD). We iteratively perform the following operations for all ratings:

$$e_{uv} = \mathbf{R}_{uv} - \mathbf{p}_u^T \mathbf{q}_v \quad (3)$$

$$\mathbf{p}_u^* = \mathbf{p}_u + \gamma[\sum_{(u,v)\in R} e_{uv}\mathbf{q}_v - \lambda \mathbf{p}_u] \quad (4)$$

$$\mathbf{q}_v^* = \mathbf{q}_v + \gamma[\sum_{(u,v)\in R} e_{uv}\mathbf{p}_u - \lambda \mathbf{q}_v] \quad (5)$$

Connected components labels the vertices in each connected component with a unique ID. One method of computing connected components is to maintain an `id` property that is initialized to be the index of each vertex (i.e., `v.id = v`). The algorithm iteratively updates the `id` property to be the minimum `id` of all its neighbors. **Single-source shortest path** takes a starting vertex and computes the shortest distance for all vertices using the Bellman-Ford algorithm [19]. **Betweenness centrality** computes the centrality indices [20] from a single source to all vertices.

4. Lux RUNTIME

4.1 System Overview

Lux is a distributed graph processing framework where a large graph instance is distributed over the aggregate device memories of multiple GPUs on one or multiple nodes. Figure 1 shows the architecture of a typical GPU cluster. Two features are worth noting. First, there are several different kinds of memory available to the GPU. While all GPU memories are relatively limited in size compared to CPU nodes, GPU device memory is the largest (up to 24 GB on current GPUs). GPU shared memory is substantially faster than GPU device memory, but extremely limited (up to 96 KB on current GPUs). Zero-copy memory is slow relative to GPU device and shared memory but relatively large. Second, there is a non-trivial memory hierarchy with locality at each level. For example, zero-copy memory is visible to all processors on a node (GPUs and CPUs), making data movement within a node (i.e., between GPU device memories on the same node) faster than transfers between nodes. To actually realize the benefits of the aggregate performance of multiple GPUs, we find it is critical to exploit the locality in the memory hierarchy, and specifically to take advantage of the GPU shared memory and the zero-copy memory.

More specifically, Lux stores the vertex updates in the zero-copy memory that is shared among all GPUs on a node. Compared to existing distributed graph processing frameworks that adopt a shared-nothing worker model (e.g., GraphX [23], Giraph [2], Pregel [31], and Groute [14]), this partially-shared design greatly reduces the amount of memory needed to store the vertex updates and substantially reduces the data transfers between different compute nodes. More subtly, Lux is often able to overlap data movement to and from the zero-copy memory with other useful work, and in some cases can enlist CPUs to carry out computations on the data in zero-copy memory, freeing up GPU cycles for other tasks. Lux's use of zero-copy memory and GPU shared memory is discussed in Section 7.

Lux adopts a model that is transactional at the granularity of iterations. Both the pull and push models defer updating vertices until the end of every iteration, which guarantees that there is no communication between GPUs within an iteration, and all data transfers occur between iterations.

Section 4.2 describes how Lux partitions and distributes input graphs. Section 4.3 and Section 4.4 introduce task executions and data transfers in Lux, respectively.

4.2 Distributed Graph Placement

Lux always distributes the entire graph onto the device memories of multiple GPUs when the aggregate GPU memory is sufficient. When the graph size exceeds the overall GPU memory capacity, Lux assigns as many edges to each GPU's device memory as possible and evenly distributes the remaining edges to the shared zero-copy memory of each node. In this section, we focus on how Lux partitions the graph among all GPUs.

Many distributed graph frameworks such as PowerGraph [22] and GraphX [23] use a *vertex-cut partitioning* that optimizes for inter-node communication by minimizing the number of edges spanning different partitions. However, we find that vertex-cut partitioning is impractical for Lux. First, running vertex-cut partitioning takes too long. For example, partitioning the Twitter graph [17] of 1.4B edges into 16

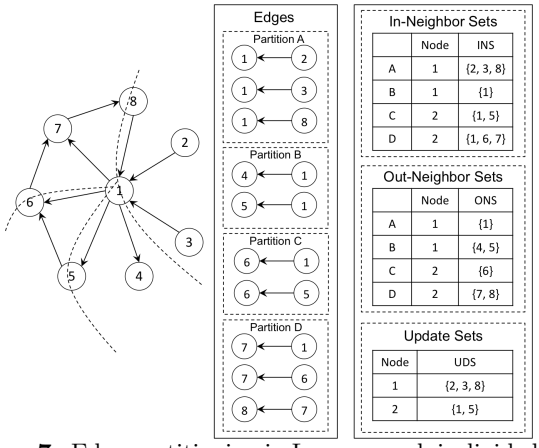


Figure 7: Edge partitioning in Lux: a graph is divided into 4 partitions, which are assigned to 4 GPUs on 2 nodes.

partitions takes more than 10 seconds, while for the algorithms we consider processing the graph usually takes less than a second. Second, because some data is shared among GPUs through a node's zero-copy memory, the number of cross-partition edges is not a good estimate of data transfers.

Lux uses *edge partitioning* [30], an efficient graph partitioning strategy that assigns a roughly equal number of edges to each partition. Recall that each vertex is assigned a unique number between 0 and $|V| - 1$. The partitioning algorithm decides how these vertices are partitioned among the device memories of multiple GPUs. In Lux, each partition holds consecutively numbered vertices, which allows us to identify each partition by its first and last vertex. Each partition also includes all the edges that point to vertices in that partition, and, again, the goal is to balance the number of edges across partitions, not the number of nodes.

Using edge partitioning allows Lux to take advantage of an important optimization performed automatically by GPUs. When multiple GPU threads issue memory references to consecutive memory addresses, the GPU hardware will automatically *coalesce* those references into a single range request that will be handled more efficiently by the underlying memory. By limiting the possible graph partitions to ranges of consecutive vertices, Lux guarantees that GPU kernels benefit from maximally coalesced accesses, which is key to maximizing memory bandwidth. Figure 7 shows an example of edge partitioning in Lux. Assume the set of all edges is divided into M *disjoint* partitions P_i, where $1 \leq i \leq M$.

$$E = \bigcup_{i=1}^{M} P_i \tag{6}$$

For each partition P_i, Lux maintains an *in-neighbor set* (*INS*) and an *out-neighbor set* (*ONS*) that contain source and destination vertices for all edges in P_i, respectively.

$$INS(P_i) = \{u | (u, v) \in P_i\} \tag{7}$$

$$ONS(P_i) = \{v | (u, v) \in P_i\} \tag{8}$$

$INS(P_i)$ determines the set of vertices whose properties are used as inputs to process the partition P_i. Similarly, $ONS(P_i)$ includes all vertices that may potentially be updated by processing the partition P_i. The in-neighbor and out-neighbor sets are computed during the partitioning phase and are used to manage data access for each partition. Note that when using edge partitioning, $ONS(P_i)$ is exactly the set of contiguous vertices owned by a partition.

Lux performs the initial partitioning at graph loading time. Before partitioning the graph, Lux computes the in-degree of each vertex. Lux accepts *compressed sparse row* (CSR) format graphs as input and is able to efficiently compute in-degrees of vertices by only loading the row indices of the graph. Lux then selects the boundary vertex for each partition that results in balanced partitioning (i.e., such that each partition has roughly $|E|/x$ edges, where $|E|$ is the number of edges, and x is the number of GPUs). This edge partitioning strategy is very efficient: partitioning the Twitter graph with 1.4B edges among 16 GPUs takes 2 ms.

After partitioning, Lux broadcasts the partitioning decision to all GPUs, which can then concurrently load the graph from a parallel file system. All GPUs store mutable vertex properties in the shared zero-copy memory, which can be directly loaded by other GPUs on the same node or transferred to other nodes via the inter-node network.

4.3 Task Execution

For each iteration, the Lux runtime launches a number of *tasks*, each of which performs the pull-based or push-based execution on a partition. Every GPU task reads the vertex properties from the shared zero-copy memory and writes the updates back to the shared zero-copy memory. To optimize GPU performance, every GPU task initially copies the vertex properties in its in-neighbor set from zero-copy memory to its device memory. This changes subsequent vertex property reads from zero-copy memory access to device memory access with orders of magnitude better bandwidth.

For pull-based execution, each GPU task is a single GPU kernel that performs the `init`, `compute`, and `update` functions. Each thread in the kernel is responsible for a vertex. Since vertices in a partition have sequential indices, this enables coalesced memory access to GPU device memory for both `init` and `update`. For `compute`, non-uniform degree distributions can impose significant load imbalance between threads. Lux uses a *scan-based gather* approach [32] to resolve the load imbalance by performing a fine-grained redistribution of edges for each thread block. Scan-based gathering enlists all threads within a thread block to cooperatively perform the `compute` function on the edges. In the pull model, since each thread only changes vertex properties in the same thread block, all updates are locally stored and aggregated in GPU shared memory, which is much faster than GPU device memory.

For push-based execution, each GPU task is translated into three GPU kernels that perform `init`, `compute` and `update`, respectively. The two GPU kernels for running `init` and `update` use the same approach as the pull model to achieve load balancing and coalesced memory access to device memory. For the GPU kernel that performs `compute`, each thread is responsible for a vertex in its in-neighbor set and the same scan-based gather optimization is used. 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.

At the end of each iteration, vertex updates are sent back to the shared zero-copy memory and become visible to subsequent tasks in Lux.

CPU tasks. When the size of an input graph exceeds the available GPU memory, Lux stores the remaining portion of the graph in CPU DRAM memory and performs CPU tasks. Lux guarantees that processing a graph partition on CPUs gives identical results as running the partition on a GPU. To perform a task on CPUs, all available CPU cores collectively execute the `init`, `compute`, and `update` functions and write the vertex updates back to the shared zero-copy memory. The largest graph used in our evaluation has 33.8 billion edges and still fits in the GPU memories on a single node, therefore this feature was not used in the experiments.

4.4 Data Transfers

Lux transfers vertex property updates between different nodes to guarantee that the updates are visible to subsequent tasks that use those vertex properties as inputs.

On each node, Lux maintains an *update set* (UDS) to monitor the set of vertices whose properties need to be fetched from remote nodes. Recall that $INS(P_i)$ is the set of vertices whose properties are read by partition P_i. $ONS(P_i)$ denotes the set of vertices whose updates are computed by partition P_i. Therefore, $\bigcup_{P_i \in N_j}\{INS(P_i)\}$ and $\bigcup_{P_i \in N_j}\{ONS(P_i)\}$ contain all vertices that are used as inputs and updated locally on node N_j. The difference of the two unions is the set of vertices that are used as input on node N_j and whose properties need to be fetched from a remote node.

$$UDS(N_j) = \bigcup_{P_i \in N_j} INS(P_i) - \bigcup_{P_i \in N_j} ONS(P_i) \qquad (9)$$

Figure 7 depicts the *INS*, *ONS* and *UDS* of a graph partition. The update sets are computed together with in-neighbor and out-neighbor sets at partitioning time and are broadcast to all nodes in Lux, which then use the update sets to schedule data transfers. At the end of every iteration, each compute node locally collects vertex updates and sends a vertex's update to remote nodes whose update set includes that vertex.

5. LOAD BALANCING

Most graph processing systems perform load balancing statically, as a preprocessing step. For example, Pregel [31] and Giraph [2] use vertex partitioning, assigning the same number of vertices to each partition through a hash function. GraphLab [30] and PGX.D [25] exploit edge partitioning as described in Section 4.2. In addition, Giraph [2] supports dynamic load balancing through over partitioning, while Presto [39] achieves balanced workload by dynamically merging and sub-dividing partitions. There is also work [40] that generates theoretically optimal repartitioning by assuming the cost to process each vertex is known.

We have found that dynamic load balancing is beneficial for Lux applications. Edge partitioning can achieve efficient load balancing for graphs with uniform degree distributions but can also behave poorly for power-law graphs. Vertex partitioning is inefficient for most graphs. Over-partitioning (e.g., Pregel) and sub-dividing partitions (e.g., Presto) are undesirable for Lux because, unlike CPU-based systems that keep all partitions in shared memory, moving partitions among GPU device memories is very expensive.

We use edge partitioning in Section 4.2 to generate an initial partitioning and then use a fast *dynamic repartitioning* approach that achieves efficient load balancing by monitoring runtime performance and recomputing the partitioning if a workload imbalance is detected. Dynamic repartitioning includes a *global* repartitioning phase that balances workload among multiple nodes and a *local* repartitioning phase that balances workload among multiple GPUs on a node.

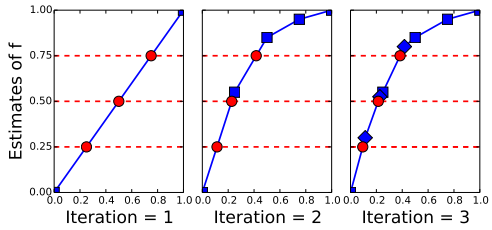


Figure 8: The estimates of f over three iterations. The blue squares indicate the actual execution times, while the red circles indicate the split points returned for a partitioning among 4 GPUs at the end of each iteration.

Recall that in edge partitioning, each partition holds consecutively numbered vertices and includes all edges pointing to those vertices. Therefore, the goal of the partitioning is to select $M-1$ pivot vertices to identify M partitions. Our dynamic repartitioning is based on two assumptions:

1. For each vertex v_i there exists a weight w_i proportional to the time required to process v_i.
2. For each partition, the execution time is approximately the sum of its vertices' weights.

We assume the weights w_i are normalized to sum to 1 and define a *partial sum* function $f(x) = \sum_{i=0}^{x} w_i$. $f(x)$ is monotonic and ranges from 0 to $f(|V|-1) = 1$.

Given f, it is easy to pick M vertices $v_1, \ldots, v_M$ such that $f(v_i) \approx i/M$, thereby creating M partitions with equal amounts of work. However, initially we know only the trivial endpoints of f, and thus we must estimate the value of points in between. To estimate the value $f(y)$ between two known points x_1 and x_2, we interpolate between $f(x_1)$ and $f(x_2)$ using the assumption that the cost of a vertex is proportional to its number of in-edges. At the end of each iteration, Lux observes the actual runtime cost of all current partitions, adding known points to f and thus reducing the amount of interpolation required to calculate updated split points.

Figure 8 gives an example of load balancing a graph with uniform in-degree over three iterations. In the first iteration the vertices are split equally among four partitions (since all nodes have the same in-degree and are thus estimated to have the same runtime cost) using three split points at 25%, 50% and 75% of the vertices. However, in this example it turns out that the first two partitions take much longer than the other two to process, as shown in the observed execution times for iteration 1 (shown as the blue squares in the graph for the start of iteration 2). Using this information new split points are selected for iteration 2. By the start of iteration 3 we can see that the estimates (the red circles) are already quite close to the observed execution times.

Dynamic repartitioning consists of the following steps:

Step 1. At the end of an iteration, Lux collects the execution time t_i of every partition P_i, updates the estimate of f, and computes a new partitioning.

Step 2. For each node N_j, Lux collects the average execution time $t(N_j) = \text{avg}_{P_i \in N_j}\{t_i\}$ and computes $\Delta_{gain}(G) = \max\{t(N_j)\} - \text{avg}\{t(N_j)\}$, which estimates the potential execution time improvement by performing a global repartitioning. Lux also computes $\Delta_{cost}(G)$, which measures the time to transfer subgraphs across different nodes and is estimated as the size of inter-node data transfers divided by the network bandwidth.

Lux compares $\Delta_{cost}(G)$ with $\Delta_{gain}(G)$. A factor α is applied to amortize the one-time repartitioning cost over an expected number of future iterations. More formally, when $\alpha\Delta_{cost}(G) < \Delta_{gain}(G)$, Lux triggers a global repartitioning and goes to Step 3. Otherwise, Lux concludes that a global repartitioning is not beneficial and goes to Step 4.

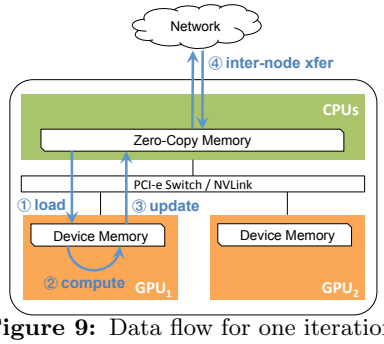


Figure 9: Data flow for one iteration.

Step 3. (Global repartitioning) Lux launches an inter-node copy request for each subgraph that needs to be moved between different nodes in the global repartitioning. Upon receiving new subgraphs, each node updates its *UDS* by scanning the *INS* for the subgraphs. After a global repartitioning, a local repartitioning is always needed, and the execution goes to Step 5.

Step 4. In the case a global repartitioning is not performed, Lux analyzes whether a local repartitioning is beneficial. For each node N_j, Lux computes $\Delta_{gain}(N_j) = \max_{P_i \in N_j}\{t_i\} - \text{avg}_{P_i \in N_j}\{t_i\}$, which estimates the potential improvement for performing a local repartitioning on node N_j. Lux also computes $\Delta_{cost}(N_j)$, which measures the time for moving subgraphs among multiple GPUs and is estimated as the amount of data moved divided by the PCI-e bandwidth.

Lux compares $\Delta_{gain}(N_j)$ with $\Delta_{cost}(N_j)$. Again, we apply a factor α to amortize the one-time repartitioning cost. If $\alpha\Delta_{cost}(N_j) < \Delta_{gain}(N_j)$, Lux triggers a local repartitioning on node N_j and goes to Step 5. Otherwise, Lux concludes that a local repartitioning is not needed and goes to Step 6.

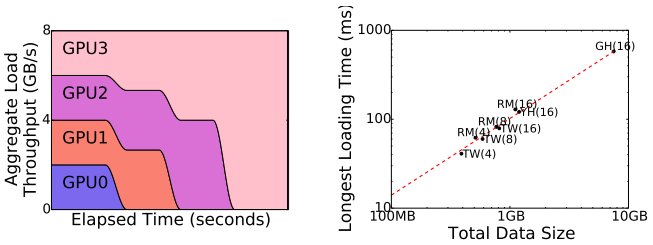
Step 5. (Local repartitioning) Lux launches an intra-node copy request for each subgraph that needs to be moved between different GPUs in the local repartitioning. Lux then updates the *INS* and *ONS* for each partition.

Step 6. Lux concludes the repartitioning and initiates execution for the next iteration.

We evaluate dynamic repartitioning in Section 8.3 and show that it achieves balanced workload within a few iterations, while the overhead for computing repartitioning and updating *INS*, *ONS* and *UDS* is negligible.

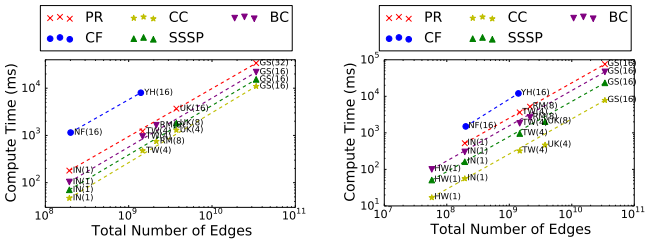
6. PERFORMANCE MODEL

In this section we develop a simple performance model that provides insight into how to improve the performance of Lux applications in different situations. The performance model quantitatively predicts the execution times for Lux applications and automatically selects an execution model and a runtime configuration for a given application and input graph. We focus on modeling the performance for a single iteration, which is divided into four steps (as shown in Figure 9): *load, compute, update,* and *inter-node transfers*. The machine architecture is parameterized by the number of nodes x and the number of GPUs per node y; Lux partitions the graph into $x \times y$ partitions, each of which is processed on one GPU. Sections 6.1 and 6.2 introduce our model for pull and push-based execution, respectively.



(a) Multiple GPUs load inputs. **(b)** Load time.

Figure 10: Modeling load time. Annotations indicate graph abbreviations, with the number of partitions in parentheses. The graph statistics are available in Table 2.



(a) The pull model. **(b)** The push model.

Figure 11: Per-iteration compute time (log-log scale).

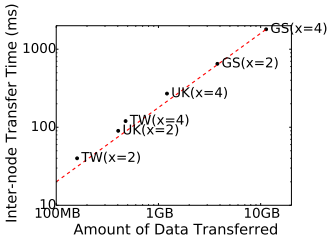


Figure 12: Inter-node transfer time (log-log scale). x indicates the number of nodes.

6.1 Pull-Based Execution

Load time. In the first step of each iteration, each GPU loads the vertices in its *INS* from zero-copy memory into GPU device memory. We assume that different GPUs on the same node initiate loading at roughly the same time and therefore share the bandwidth on the PCI-e switch or NVLink. Figure 10a gives an example where multiple GPUs concurrently load input vertices on a compute node. The height and width of the rectangle are the aggregate load throughput and elapsed load time, respectively. Therefore, the area of the rectangle is the total amount of data loaded into the GPUs. Assuming a constant load rate γ_l, which is limited by the PCI-e or NVLink bandwidth, the *longest load time* is proportional to the total amount of data loaded. Recall that $INS(P_i)$ is the set of input vertices for the i^{th} partition. The amount of data loaded on node N_j is $|T_v| \times \sum_{P_i \in N_j} |INS(P_i)|$, where $|T_v|$ is a constant indicating the size of each vertex. Therefore, the longest load time is

$$T_{load} = \max_{1 \le j \le x} \gamma_l |T_v| \sum_{P_i \in N_j} |INS(P_i)| \quad (10)$$

$$\approx \gamma_l |T_v| \sum_{1 \le i \le x \times y} |INS(P_i)|/x \quad (11)$$

The last approximation assumes that all nodes have similar amounts of input, which is consistent with our experience.

Figure 10b shows the relationship between the longest load time and the total amount of data loaded for executions with various graphs and numbers of GPUs. We have found that γ_l is independent of y (the number of GPUs on a node) and the graph size, and only depends on the underlying hardware. For a particular machine, γ_l can be estimated by running a few small and medium sized graphs.

Compute time. In the pull model, Lux processes the `compute` function for all edges, as shown in Algorithm 1. Therefore, the total compute time for each iteration is proportional to the total number of edges in the graph (i.e., $|E|$). Figure 11a shows the relation between $|E|$ and the total compute time for different graph algorithms. The results show that, for a given graph algorithm, the total compute time for each iteration is proportional to the graph size and is insensitive to the graph type or the number of partitions. In addition, the model assumes perfect load balancing among partitions, since the dynamic repartitioning strategy in Section 5 achieves balanced partitions in a few iterations. Therefore, the compute time on each GPU is

$$T_{compute} = \gamma_c |E|/(x \times y) \quad (12)$$

where γ_c is the slope of the line in Figure 11a.

Update time. At the end of each iteration, Lux moves vertex updates from GPU device memory back to zero-copy memory to make updates visible to other GPUs before the next iteration. In the pull model, Lux is able to overlap the update time with the compute time by sending updates directly to zero-copy memory as long as all in-edges of a vertex have been processed. Therefore, we omit the update time in the pull model.

Inter-node transfer time. Before starting the next iteration, each node collects updates to vertices in its update set (*UDS*) from remote nodes, so that the updates are visible to its local GPUs in the next iteration. Inter-node transfer time is proportional to the total amount of data being transferred. Figure 12 shows the relation between the inter-node transfer time and the amount of data transferred. For a node N_j, the amount of data received in every iteration is $(|T_v| \times |UDS(N_j)|)$, where T_v is the size of each vertex. Therefore, we model the inter-node transfer time as

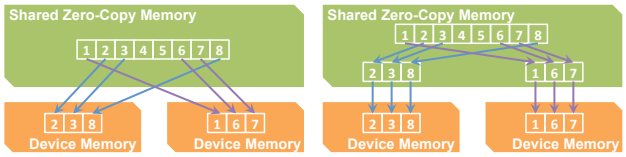
$$T_{xfer} = \gamma x |T_v| \sum_{1 \le j \le x} |UDS(N_j)| \quad (13)$$

6.2 Push-Based Execution

Load time. In the push model, Lux overlaps data transfers to the GPU device memory with the compute kernels by processing the out-edges of a vertex as long as it is available on the GPU. This hides load time in the push model.

Compute time. In the push model, Lux only processes vertices in a frontier queue, as shown in Algorithm 2. The per-iteration compute time varies as the frontier queue changes. We average the execution time for the compute kernels over iterations and use that to model the compute time. To avoid the long-tail effects in some graphs, we only collect iterations during which the percentage of vertices that are in the frontier queue is above a threshold, which is set to 0.05 in our experiments. Figure 11b shows the relationship between the graph size and the average computation time for each iteration in the push-based executions. Similar to the pull model, we use a linear relation to model the graph size and per-iteration total compute time in the push model.

$$T_{compute} = \gamma_c^* |E|/(x \times y) \quad (14)$$



(a) GPU kernel approach. **(b)** CPU core approach.
Figure 13: Different approaches for loading input data.

Update time. In the push model, Lux moves vertex updates from the GPU device memory to the shared zero-copy memory. In all our experiments, we have found that the update time is usually at least two orders of magnitude shorter than the compute time in the push model. Therefore, we ignore the update time in the push model.

Inter-node transfer time. Since the inter-node transfer patterns are identical in both the push and pull models, we use equation 13 for predicting the inter-node transfer time.

Table 1 summarizes our model for pull- and push-based execution. The performance model requires some graph properties as input, including *INS* and *UDS*. However, for a given graph, its *INS* and *UDS* are orthogonal to any graph algorithms or the underlying machine environments and only depends on the number of partitions on each node (i.e., y) and the number of nodes (i.e., x). Therefore, we only need to compute the sizes of *INS* and *UDS* once for different x and y and attach the numbers as additional graph properties. All future performance predictions can reuse the numbers without any additional analysis of the graph.

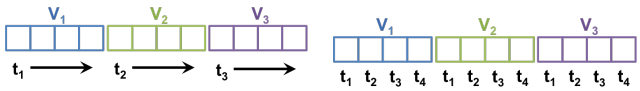
In addition to explicitly selecting an execution model and a runtime configuration by applications, Lux can automatically choose these parameters based on the performance model. Given an input graph and an algorithm, Lux computes the estimated per-iteration execution time with different configurations (i.e., x and y) for both pull and push-based execution models and chooses the execution model and configuration with the best estimated execution time.

7. IMPLEMENTATION

We implement Lux in Legion [12, 38], a high-performance parallel runtime for distributed heterogeneous architectures. Lux uses Legion's infrastructure for placing data in specific memories, launching tasks onto either GPUs or CPUs, and automatic support for data movement and coherence. Legion's abstractions enable us to easily experiment with various strategies and explore both the push- and pull-based execution models and different approaches to optimizing each. With Legion, it is also simple to retarget Lux to a new generation of GPUs or a different family of accelerators.

We list additional optimizations in our implementation that help achieve good performance.

Loading input data. Lux implements two different approaches for loading input data into GPUs, as shown in Figure 13. The GPU kernel approach launches one kernel on each GPU that copies vertices in the *INS* from the shared zero-copy memory to the device memory. The data copies are performed by GPUs with direct memory access, and no CPU cores are involved. Figure 13b shows the CPU core approach, which decouples the loading into two hops. The first hop gathers vertices in the *INS* in the shared zero-copy memory by collaboratively using all available CPU cores. The second hop performs simple bulk data transfers to the GPUs. The two hops are pipelined by synchronizing the CPU data gathering with the GPU bulk data transfers.



(a) Individual processing. **(b)** Cooperative processing.
Figure 14: Different approaches for processing graphs.

The two approaches differ in which processors perform the data gathering. We have found that the GPU kernel approach consistently outperforms the CPU core approach, reducing the load time by 50%. Therefore, in the pull model, Lux uses the GPU kernel approach since Lux cannot overlap loading with computation. However, in the push model, the GPU kernel approach launches additional kernels to GPUs, which prevents the GPUs from fully overlapping the input loading with the graph processing. Therefore, in the push model, Lux uses the CPU core approach that only launches bulk copies to the GPUs. In this case, Lux pipelines the data gathering performed by the CPU cores, the bulk data transfers performed by the GPU copy engines, and the graph processing performed by the GPU kernel engines.

Coalesced memory access. Achieving coalesced accesses to the device memory is critical for high GPU performance. Lux stores vertex properties in the *array-of-structs* layout in the device memory to coalesce accesses to the properties associated with the same vertex. This layout is beneficial for machine learning and data mining algorithms that assign a vector of data elements to each vertex. For example, in collaborative filtering, a vector of length K is defined as the property for each vertex (see Figure 6).

One straightforward approach to process these algorithms is to map the `compute` function for each edge to a GPU thread, so that different threads can individually process the graph, as shown in Figure 14a. However, this prevents coalesced accesses since threads in the same warp are performing strided access to device memory. A second approach is to use K (i.e., the vector length) threads in a warp to cooperatively perform the `compute` function for each edge, as shown in Figure 14b. This approach allows coalesced accesses in each warp but requires additional synchronization between cooperative threads processing the same edge.

Both approaches show some inefficiency for processing graphs with large vertex properties. For collaborative filtering, neither even matches the performance of state-of-the-art CPU-based approaches. We propose a novel hybrid approach that combines the previous two approaches to achieve coalesced accesses to device memory while not requiring additional synchronization. In the hybrid approach, we first use cooperative threads to copy the vectors to GPU shared memory and then use individual threads to perform the `compute` function for every edge. The first step guarantees coalesced accesses to device memory. The second step eliminates additional synchronization, and there is no penalty for strided access in shared memory. As a result, our implementation for collaborative filtering achieves up to 10× speedup compared with state-of-the-art CPU approaches.

Cache optimizations. Lux integrates a number of cache optimizations. First, as described in Section 4.3, Lux caches the input vertices in GPU device memory, which reduces zero-copy memory accesses. Second, Lux opportunistically caches vertex loads/updates in GPU shared memory when possible to further reduce reads/writes to device memory. For example, in the pull model, a vertex's in-edges are grouped and processed by the same thread block, therefore Lux caches and locally aggregates the updates in GPU shared memory. This eliminates any direct updates to device memory.

Table 1: Modeling per-iteration execution time for both pull-based and push-based executions.

	load time	compute time	update time	inter-node xfer time
$T_{pull}(x, y)$	$\gamma_l \lvert T_v \rvert \sum_{1 \le i \le x \times y} \lvert INS(P_i) \rvert / x$	$\gamma_c \lvert E \rvert / (x \times y)$	0	$\gamma_x \lvert T_v \rvert \sum_{1 \le j \le x} \lvert UDS(N_j) \rvert$
$T_{push}(x, y)$	0	$\gamma_c^* \lvert E \rvert / (x \times y)$	0	$\gamma_x \lvert T_v \rvert \sum_{1 \le j \le x} \lvert UDS(N_j) \rvert$

Table 2: Graph inputs used in the experiments.

Abbr.	Input	Num. Vertices	Num. Directed Edges
KR	Kronecker	2,097,152	33,554,432
HW	Hollywood	1,139,905	57,515,616
IN	Indochina	7,414,866	194,109,311
TW	Twitter	41,652,230	1,468,365,182
RM	RMAT27	134,217,728	2,147,483,648
UK	UK2007	105,896,555	3,738,733,648
GS	GSH2015	988,490,691	33,877,399,152
AM	Amazon	3,376,972	11,676,082
ML	MovieLens	424,338	48,808,192
NF	NetFlix	497,959	200,961,014
YH	Yahoo Music	1,959,915	1,399,280,452

8. EVALUATION

The input graphs used in our experiments are shown in Table 2. HW is a graph of movie actors, where each vertex corresponds to an actor, and each edge indicates two actors have appeared together in a movie. KR [29] and RM [18] are power-law synthetic graphs for modeling social networks. TW is a graph of the Twitter social network [17]. IN, UK and GS are webpage graphs from Indochina, .uk and .eu domains, respectively [16, 17]. Note that GS contains 33.8 billion edges, which is around 3 times larger than the largest real-world graph we have seen in previous papers [10].

We use PageRank (PR), connected components (CC), single-source shortest path (SSSP), betweenness centrality (BC), and collaborative filtering (CF) as the benchmark algorithms in the evaluation. CF requires weighted bipartite graphs that represent the ratings matrix. We use four real-world bipartite graphs in the experiments. AM is the product dataset from the Amazon online shopping website [1]. ML describes movie ratings from MovieLens [24]. NF is the training dataset for the Netflix Prize competition [15]. YH is the ratings dataset for Yahoo! Music [11].

We tested all graph processing frameworks on all of the benchmarks and experimented with multiple parameter settings to identify the settings with the best performance for each system. All shared memory frameworks (i.e., Ligra, Galois, and Polymer) were evaluated on a Lonestar5 [5] node with four Intel 12-core E7-4860 Xeon processors (with hyper-threading) and 1TB main memory. All other experiments were performed on the XStream GPU cluster [9]. Each XStream node is equipped with two Intel 10-core E5-2680 Xeon processors, 256 GB main memory, and eight NVIDIA Tesla K80. Each Tesla K80 has two GPUs with 12GB device memory and 48KB shared memory on each GPU. Nodes are connected over Mellanox FDR Infiniband with 7GB/s links. In the evaluation, Lux uses the pull model for PR and CF and the push model for CC, SSSP, and BC, except in Section 8.4, which performs a comparison between the two execution models in Lux.

8.1 Single GPU Results

First, we evaluate the GPU kernels in Lux by comparing Lux with state-of-the-art GPU-based graph frameworks. Due to the small device memory on a single GPU, we limited these experiments to graphs that can fit in a single GPU.

Figure 15 shows the comparison results among CuSha, MapGraph, Groute, and Lux. Both CuSha and MapGraph can only process graphs that fit in a single GPU device memory. We were not able to run CuSha on NF and MapGraph on IN and NF due to out of device memory errors.

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

For PR, CC, SSSP, and BC, we expect that Lux would be slightly slower than the other frameworks, since it writes the vertex property updates back to the zero-copy memory between iterations, while CuSha, MapGraph, and Groute keep all changes in the GPU device memory. However, for these graphs, the data transfer time is usually within a millisecond due to coalesced memory accesses in Lux. As a result, Lux is competitive with the other systems that are designed to optimize single GPU performance.

For CF, we expect that Lux would be faster, since Lux is optimized for large vertex properties, as described in Section 7. Figure 15 shows that Lux achieves 1.5-5× speedup compared to the other frameworks.

8.2 Multi-CPU and Multi-GPU Results

We compare the performance of Lux with state-of-the-art multi-CPU and multi-GPU frameworks. Previous efforts [34, 37, 33] show that shared memory multi-CPU frameworks are typically much faster than distributed multi-CPU systems because they avoid inter-node communication. We compared Lux with Ligra, Galois, and Polymer, which are considered to offer the best graph processing performance for shared memory. In addition, we also compared against two popular distributed CPU-based frameworks, PowerGraph and GraphX. The experiments for PowerGraph and GraphX are performed using 16 nodes, each of which has 20 CPU cores and 240GB memory. Finally, we compared Lux with Groute and Medusa, both of which provide effective graph processing performance for multiple GPUs on a single node. The experiments for Groute and Medusa are performed on a single XStream node with 16 GPUs. We do not perform a comparison with GTS because the GTS implementation is not available to run experiments.

Figure 16 shows the comparison results. For all systems, we try different configurations and report the best attainable performance. As a result, the reported numbers are often better than the numbers in the original papers. For example, the performance of Ligra in this paper is around 2× better than [37]. Our GraphX executions achieve 6× better performance than [23]. Figure 17 shows Lux's performance with different configurations. The GS graph requires

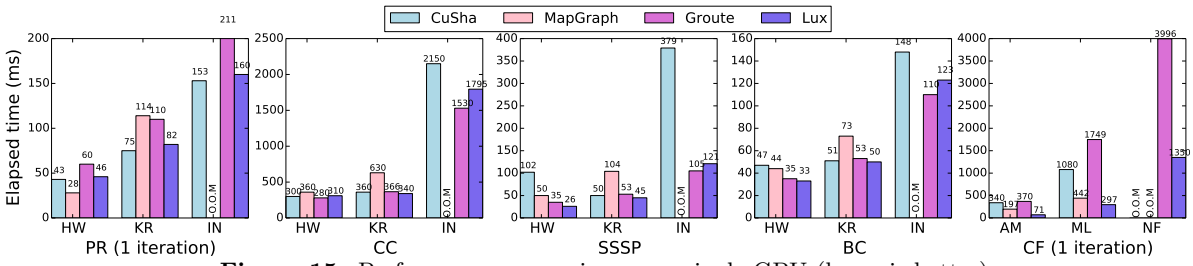


Figure 15: Performance comparison on a single GPU (lower is better).

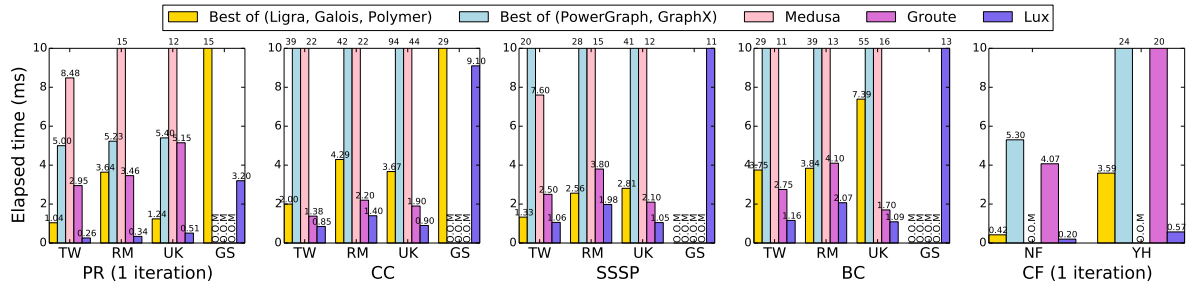


Figure 16: The execution time for different graph processing frameworks (lower is better).

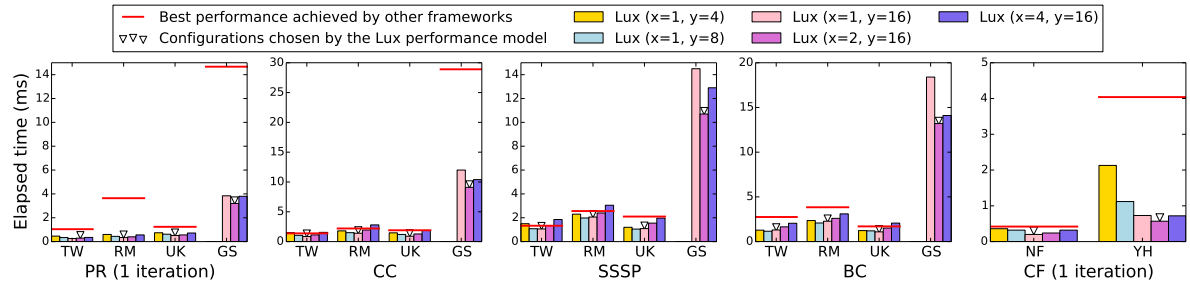


Figure 17: The execution time for different Lux configurations (lower is better). x and y indicate the number of nodes and the number of GPUs on each node.

the combined memories of at least 12 GPUs, and therefore we omit its performance numbers for 4 and 8 GPUs. For each input graph and algorithm, the figure also shows the configuration selected by the Lux performance model, which chooses the optimal configuration in most cases.

Compared to Ligra, Galois, and Polymer, Lux achieves 1.3-7× speedup for the five algorithms. Lux achieves most of its performance speedup by exploiting the aggregate GPU memory bandwidth. The shared memory systems eliminate the cost of data transfers and graph partitioning by processing the entire graph in shared CPU memory; thus, because Lux incurs some cost for these operations that cannot always be overlapped with computation, Lux's compute phase must be much faster than that of the shared memory systems to improve overall performance. In our experiments, Lux reduces compute time by up to 30× (see Figure 20).

Figure 16 shows that a system design that exploits available bandwidth and the memory hierarchy in a distributed memory architecture can achieve much better absolute performance than pure shared memory systems on a single node. This comparison, however, does not take into account the disparity in raw computational power and cost of a multi-GPU node versus one with CPUs only. Table 3 normalizes performance by cost, comparing the efficiency per dollar of Lux and shared memory frameworks on a single node. The cost efficiency is calculated by dividing performance (number of iterations per second) by machine cost. For the CPU-based approaches, we use the best performance numbers of Ligra, Galois and Polymer. Higher cost efficiency indicates that the system achieves better performance with the same machine cost. Table 3 shows that Lux achieves cost efficiency on a par with CPU-based systems for CC, SSSP, BC, and CF, and has better cost efficiency for PR.

Compared to PowerGraph and GraphX, Lux achieves orders of magnitude speedup in most cases. Besides using the aggregate GPU bandwidth to accelerate computation, Lux also benefits from exploiting the hierarchical structure of the memory in multi-node multi-GPU machines; in particular by using shared zero-copy memory to minimize inter-node data transfers. As an example of an alternative design, GraphX adopts a shared-nothing worker model and requires more inter-node communication [23].

Compared to Medusa and Groute, Lux achieves 1.5-2.5× speedup for CC, SSSP, and BC and 10-20× speedup for PR and CF. This is because both Medusa and Groute process individual vertices separately, which improves load balance but may potentially miss opportunities for coalesced memory accesses and increase runtime overhead when processing large graphs on multiple GPUs.

8.3 Load Balancing

Figure 18 shows how Lux's repartitioning approach achieves dynamic load balancing. We also use a local repartitioning approach that only balances workload among different GPUs on a single node as a baseline. For both approaches, the repartitioning cost is decomposed into two parts. The first part (shown as the almost-immeasurable yellow bars in Figure 18) is computing a new repartitioning based on the execution times in previous iterations. The second part (shown as the pink bars) is the graph migration cost, which

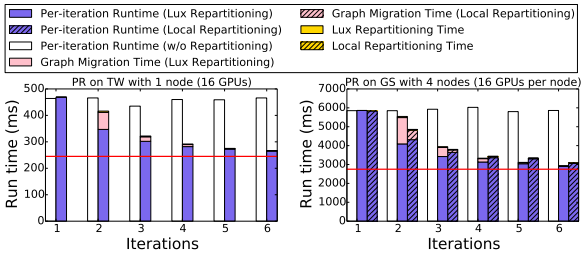
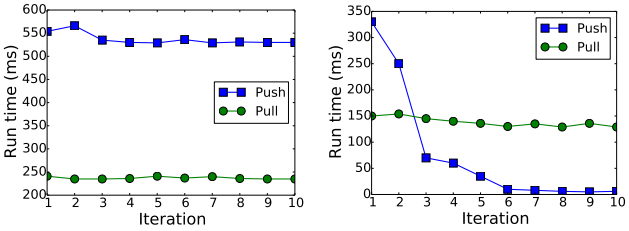


Figure 18: Performance comparison for different dynamic repartitioning approaches. The horizontal line shows the expected per-iteration run time with perfect load balancing.



(a) PR. **(b)** CC.

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

involves moving subgraphs among different GPUs on the same node (for both approaches) and moving subgraphs among different nodes (for Lux's repartitioning approach).

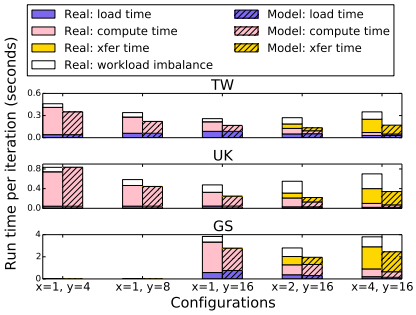
For the single-node execution, both approaches behave identically and reduce the application's per-iteration run time by 43% after a few iterations. For the multi-node execution, the Lux repartitioning approach and the local repartitioning approach reduce the application's per-iteration run time by 51% and 47%, respectively. However, in the first few iterations, the Lux repartitioning approach suffers 2× more graph migration cost compared to the local repartitioning approach. This is because Lux's repartitioning approach also balances workload across different nodes and therefore introduces additional cost for migrating subgraphs. For both executions, the overhead for computing a repartitioning is negligible compared to the application's run time, and the graph migration cost decreases dramatically and approaches zero after a few iterations.

8.4 Pull vs. Push

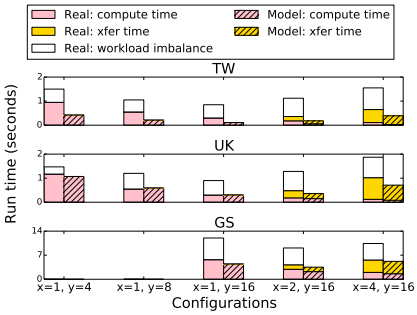
Figure 19 shows a comparison between Lux's pull and push execution models. The performance gap between the pull and push models is significant in our experiments, which we illustrate using PR and CC. For PR, the pull model consistently outperforms the push model, which is due to a number of optimizations integrated in Lux. For CC, the pull model performs better for the first few iterations when most of the vertices are active. However, the push model becomes more efficient when the values for most vertices have converged and those vertices remain inactive.

8.5 Performance Model

Figure 20 shows the results of our performance model for the pull and push-based executions. For the real executions, we collect the load/compute time of each GPU task and the inter-node transfer time of each compute node. In Figure 20, we plot the average load, compute, and inter-node transfer times. The remaining wall-clock execution time is attributed to workload imbalance among different partitions and nodes. Since our performance model assumes balanced partitions, the workload imbalance overhead is not part of the performance model.



(a) Pull-based executions (PR).



(b) Push-based executions (CC).

Figure 20: Performance model for different executions.

Our performance model is better for predicting pull-based than push-based execution. For predictions of the load and compute time in the pull-based execution, the relative errors are less than 20% in most cases. Moreover, the workload imbalance is insignificant in the pull-based execution, which makes our performance model a practical tool for estimating the performance of different executions and choosing an efficient configuration for a given input graph and algorithm.

The prediction for the push-based execution is not as accurate as the pull-based execution. The main challenge is the changing frontier queue over iterations, which amplifies workload imbalance among different partitions and makes execution time less predictable.

9. CONCLUSION

We have presented Lux, a distributed multi-GPU system. Lux provides two execution models that are beneficial for different graph applications. A dynamic graph repartitioning mechanism achieves good load balancing among multiple GPUs, while the performance model provides insight into improving Lux's performance. Our evaluation shows that Lux outperforms state-of-the-art graph processing systems.

Acknowledgement

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, as well as a grant from the National Science Foundation CCF-1160904. This research is based partially upon work supported by the Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number DE-SC0008111 and by the Department of Energy National Nuclear Security Administration under Award Number DE-NA0002373-1.

We thank Christopher R. Aberger, Mingyu Gao, Samuel Grossman, and the anonymous reviewers for their feedback. We also thank the Ligra and Galois authors for their assistance with the experiments.

10. REFERENCES

[1] Amazon ratings network dataset. http://konect.uni-koblenz.de/networks/amazon-ratings.

[2] Apache giraph. http://http://giraph.apache.org/.

[3] Intel Xeon Processor E5-2680 v2. http://ark.intel.com/products/75277/Intel-Xeon-Processor-E5-2680-v2.

[4] Intel Xeon Processor E7-4860 v2. https://ark.intel.com/products/75249/Intel-Xeon-Processor-E7-4860-v2.

[5] Lonestar 5 user guide. https://portal.tacc.utexas.edu/user-guides/lonestar5.

[6] NVIDIA NVLink high-speed interconnect. http://www.nvidia.com/object/nvlink.html.

[7] NVidia Tesla K80. http://www.anandtech.com/show/8729/nvidia-launches-tesla-k80-gk210-gpu.

[8] Server Memory Prices. https://memory.net/memory-prices/.

[9] XStream Cray CS-Storm compute cluster. http://xstream.stanford.edu/.

[10] Yahoo! Altavista web page hyperlink connectivity graph. http://webscope.sandbox.yahoo.com/catalog.php?datatype=g.

[11] Yahoo! music user ratings of songs with artist, album, and genre meta information. "https://webscope.sandbox.yahoo.com/catalog.php?datatype=r".

[12] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.

[13] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.

[14] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-GPU programming model for irregular computations. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017.

[15] James Bennett, Stan Lanning, et al. The Netflix Prize. In *Proceedings of KDD cup and workshop*, 2007.

[16] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*, 2011.

[17] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, 2004.

[18] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, 2004.

[19] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[20] Linton C Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 1977.

[21] Zhisong Fu, Michael Personick, and Bryan Thompson. MapGraph: A high level API for fast development of high performance graph analytics on GPUs. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, 2014.

[22] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

[23] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, 2014.

[24] F. Maxwell Harper and Joseph A. Konstan. The MovieLens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4), 2015.

[25] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. PGX.D: A fast distributed graph processing engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, 2015.

[26] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. CuSha: Vertex-centric graph processing on GPUs. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, 2014.

[27] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. GTS: A fast and scalable graph processing method based on streaming topology to GPUs. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, 2016.

[28] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8), 2009.

[29] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11(Feb), 2010.

[30] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *PVLDB*, 5(8):716–727, 2012.

[31] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, 2010.

[32] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*,

PPoPP '12, 2012.

[33] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[34] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, 2014.

[35] Zechao Shang, Feifei Li, Jeffrey Xu Yu, Zhiwei Zhang, and Hong Cheng. Graph analytics through fine-grained parallelism. SIGMOD '16, 2016.

[36] Yingxia Shao, Junjie Yao, Bin Cui, and Lin Ma. PAGE: A partition aware graph computation engine. In *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management*, CIKM '13, 2013.

[37] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, 2013.

[38] Sean Treichler, Michael Bauer, and Alex Aiken. Realm: An event-based low-level runtime for distributed memory architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, 2014.

[39] Shivaram Venkataraman, Erik Bodzsar, Indrajit Roy, Alvin AuYoung, and Robert S. Schreiber. Presto: Distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, 2013.

[40] Clément Vuchener and Aurélien Esnard. Dynamic load-balancing with variable number of processors based on graph repartitioning. In *High Performance Computing (HiPC), 2012 19th International Conference on*, 2012.

[41] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, 2015.

[42] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI'12*, San Jose, CA, 2008.

[43] Kaiyuan Zhang, Rong Chen, and Haibo Chen. NUMA-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'15, 2015.

[44] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 25(6), 2014.

[45] Chang Zhou, Jun Gao, Binbin Sun, and Jeffrey Xu Yu. MOCgraph: Scalable distributed graph processing using message online computing. *PVLDB*, 8(4):377–388, 2014.