Swift Unfolding of Communities: GPU-Accelerated Louvain Algorithm

Zhibin Wang[1], Xi Lin[1], Xue Li[2], Pinhuan Wang[3], Ziheng Meng[1],
Hang Liu[3], Chen Tian[1]*, Sheng Zhong[1]*
1. State Key Laboratory for Novel Software Technology, Nanjing University
2. Alibaba Group 3. Rutgers, The State University of New Jersey
wzbwangzhibin@gmail.com, 350904583lx@gmail.com, youli.lx@alibaba-inc.com,
pw346@scarletmail.rutgers.edu, jshamzh@gmail.com, hl1097@scarletmail.rutgers.edu,
tianchen@nju.edu.cn, sheng.zhong@gmail.com

Abstract

The Louvain algorithm is one of the most popular algorithms for community detection. Observing that existing implementations suffer from inaccurate pruning and inefficient intermediate state management, we introduce GALA, GPU-Accelerated Louvain Algorithm, which incorporates two key innovations. The first innovation is a novel modularity gain-based pruning strategy, supported by rigorous theoretical guarantees of optimality and able to reduce up to 76% of vertices as well as their corresponding computations. To take advantage of the memory hierarchy and parallelism of GPUs, the second innovation is workload-aware kernels, featuring a shuffle-based kernel founded on the warp-level primitives for exchange states and a hash-based kernel that prioritizes shared memory in hashtable design. GALA further scales to multiple GPUs by minimizing the synchronization overhead between GPUs through a dense-sparse synchronization strategy. We evaluate the performance of GALA through theoretical analysis and practical experiments on various real-world graphs. The experimental results confirm that GALA significantly improves the performance of the parallel Louvain algorithm on GPUs, surpassing state-of-the-art solutions by 6× on average.

CCS Concepts: • **Computing methodologies** → **Parallel algorithms**; • **Theory of computation** → **Graph algorithms analysis**.

*Corresponding authors.

PPoPP '25, March 1–5, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1443-6/25/03…$15.00
https://doi.org/10.1145/3710848.3710884

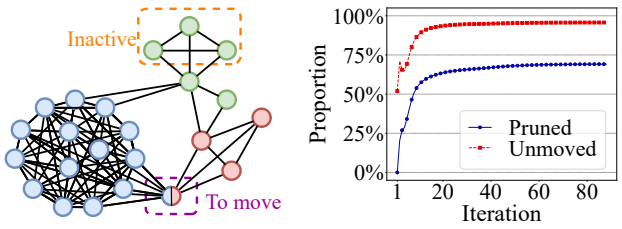
Keywords: Louvain algorithm, community detection, GPU

1 Introduction

While graph analytics have fueled advances in our understanding of complex connected systems, community detection is often the *de facto* first analysis applied to any graphs of our interest. In a nutshell, community detection is the problem of classifying the vertices of a graph into sets such that the vertices in each set are more closely connected to each other than to the rest of the graph. Because community detection could unfold the structurally coherent vertices in an unsupervised manner, this tool is broadly used in a variety of scientific and engineering applications, including social network analysis [17, 55], bioinformatics [20, 47], and transportation network analysis [19, 49] among many others. We refer the readers to [7, 26] for a comprehensive study of community detection algorithms, systems, and applications.

While the efforts in community detection have flourished in the recent decades [23, 27, 41, 48, 54], this paper concludes that there mainly exist three concerted cohorts of endeavors, i.e., graph partitioning, modularity, and label propagation. Particularly, the perspective of our categorization centers around the metric of how each work measures the closeness of vertices: 1) graph partitioning minimizes the number of edge cuts between partitions; 2) modularity considers both the edge cuts (i.e., edges outside of the community) and the edges inside of the community (see Equation 1); 3) label propagation takes a majority voting mechanism, that is, a vertex belongs to the community where most of its neighbors belong to.

Of these measures, modularity is the one that is widely used and keeps thriving. Initially, modularity was defined as a measure to evaluate the quality of network partitions by comparing the density of edges within communities to a random distribution of edges and has since become a widely accepted metric for assessing community structure [7]. However, modularity is not a measure without limitations: 1) modularity cannot detect smaller communities within large networks; 2) modularity possesses stringent serial execution requirements. Recently, the Louvain method [9] introduced an efficient, scalable, multi-phase approach that iteratively



(a) A visualization of communities, moved vertices, and inactive vertices on zebra [3] graph. **(b)** The proportion of pruned (inactive) and unmoved vertices on LiveJournal [60] graph.

Figure 1. Examples of pruning unmoved vertices.

merges communities to maximize modularity gain, which addresses both 1) and 2). To identify small communities, Louvain algorithms with an adjustable resolution parameter have been introduced [4, 30]. There are also approaches based on statistical inference [27, 46] and Graph Neural Networks (GNNs) [14, 58] that form communities informed by prior knowledge. However, these methods often involve too many parameters to be specified compared to Louvain and suffer from high computational complexity. Therefore, this paper primarily focuses on the Louvain algorithm.

The Louvain algorithm iteratively moves vertices to maximize modularity gain in the first phase and constructs a hierarchical community structure in the second phase. In each iteration of the first phase, a vertex will evaluate the modularity gain of its neighboring communities by analyzing the weight between itself and its neighboring communities and move to the community with the highest modularity gain. For example, the bicolor vertex enclosed by purple dashed lines in Figure 1(a) is to move from the blue community to the red community for higher modularity gain.

As the size of the graphs continues to grow, the sequential implementation of the Louvain algorithm may not be efficient and scalable enough [57]. This inefficiency stems from the computational-intensive and memory-intensive workload, such as evaluating modularity gain for the potential of vertex moving to a new community. To overcome these challenges, parallel solutions have been developed for multi-core CPU [21, 24, 51] and GPUs [8, 15, 39]. Particularly, GPUs, with their inherent capability for massive parallelism (up to 216k concurrent threads on A100), have demonstrated particularly promising results. GPUs can simultaneously process numerous vertices, substantially accelerating the algorithm. While parallelization provides considerable advantages, it is crucial to curtail unnecessary computations to achieve optimal performance gains. Furthermore, the memory hierarchy of GPUs, including registers, shared memory, and global memory, offers significant throughput advantages for memory-intensive operations, while the Louvain algorithm requires frequent access and maintenance of states related to neighboring vertices. However, existing implementations suffer from two critical issues:

Inaccurate or insufficient pruning: In large-scale graphs, the Louvain algorithm may need a substantial number of iterations to converge, as illustrated in Figure 1(b). With the iteration progress, we observe that a large portion (up to 95%) of vertices remain in their current communities, namely unmoved. This presents an opportunity to skip processing if we can predict these unmoved vertices and mark them as *inactive* before the iteration. For instance, three green vertices enclosed by orange dashed lines in Figure 1(a) are in the core of the community, i.e., none of their neighbors belong to another community, and their neighbors were unmoved in the previous iteration. Thus, these three vertices can be marked as inactive. We observe that existing heuristic pruning strategies based solely on movement information are compromised by a dual shortcoming: they either result in a *suboptimal* solution or *insufficiently* identify the unmoved vertices. Specifically, a strict strategy [50] marks a vertex as inactive if all neighboring communities remain unchanged, resulting in a high rate of false positives (91.7%), missing most of the unmoved vertices, and offering limited performance improvement. On the other hand, a relaxed strategy [43, 53, 54, 61] relaxes this constraint, allowing a vertex to be marked as inactive if its neighbors remain unmoved in the previous iteration. While this strategy reduces false positive instances, it introduces false negative instances, which will ignore some potential vertex movements and yield a suboptimal result. Practical evidence also confirms that employing the relaxed strategy results in an average 0.38% false negative rate, leading to 0.0012 modularity quality loss. Fortunately, the Bulk Synchronized Parallel (BSP) model, utilized by parallel implementations of the Louvain algorithm, provides richer information beyond mere vertex movements, allowing for the development of an efficient and effective pruning strategy.

Inefficient Intermediate State Management: The Louvain algorithm requires a more complex set of states to track the weight between a vertex and its neighboring communities. Early attempts on GPUs [15], use global memory to store intermediate states. This leads to a substantial performance bottleneck, impacting overall performance. Subsequent research [8, 39] have employed shared memory to maintain intermediate states for small degree vertices, yet still do not fully exploit the memory hierarchy of GPUs. Other research efforts [5, 6, 29, 33, 44, 45] have explored the implementation of collections, e.g., hashtable, in the GPUs. However, some of these [5, 33] manage a global hashtable, while others [29, 44, 45] target for intersection operation.

In this paper, we propose GALA, an abbreviation of GPU-Accelerated Louvain Algorithm.

GALA integrates a novel modularity gain-based pruning strategy to reduce computation cost while preserving optimality (Section 3). If a vertex has already attained a sufficiently high modularity gain within its current community,

the movement in the next iteration is redundant. The additional states provided by the BSP model enable obtaining a tighter upper bound of maximal possible modularity gain through moving to other communities. By comparing the upper bound with the current modularity gain, the pruning strategy can achieve a low rate of false positives while avoiding false negatives. As shown in Figure 1(b), up to 69% of vertices are pruned as iteration progresses. Observing that recalculating the weight between a vertex and its current community becomes a new bottleneck, we further propose an efficient community weight updating approach to avoid unnecessary recalculations.

To efficiently leverage the memory hierarchy and attached parallelism in GPUs, we develop a novel memory management strategy for GALA (Section 4). Considering the different characteristics of each level of memory, we propose two GPU kernels, namely shuffle-based and hash-based, to optimize the management of the critical intermediate states. The shuffle-based kernel maintains the states among the registers of threads within a warp and utilizes warp-level primitives to shuffle the states. In contrast, the hash-based kernel exploits both shared memory and global memory to maintain intermediate states, employing a hierarchical hashtable that gives priority to accessing shared memory. Furthermore, we distribute the graph across multiple GPUs and propose an adaptive strategy to scale GALA .

To evaluate the performance of GALA, we conduct extensive experiments on various real-world graphs (Section 5). The empirical evidence indicates that both the computation pruning and memory management techniques enhance its efficiency. Compared with state of the art, GALA outperforms cuGraph [1] by 17×, Gunrock [42, 59] by 53×, and Grappolo [36, 39] by 6× on average.

2 Background

2.1 Definition

Graph: Consider a weighted and undirected graph $G = \{V, E, w\}$, where V is the set of vertices, E is the set of edges and $w : E \rightarrow \mathbb{R}$ is the weight function that maps each edge to a real-valued weight. In addition, we use v and $e = (v, u)$ to denote a vertex and an edge, respectively, and $w(e)$ (or $w(v, u)$) to denote the weight of an edge $e = (v, u)$. Given a vertex set U, we use $N_U(v) = \{u|(v, u) \in E, u \in U\}$ to denote the neighbors of v that belong to the vertex set U. Furthermore, we use $|X|$ to denote the *weight* cardinality of a set X. Particularly, for the edge set E, $|E| = \sum_{e \in E} w(e)$. And $d_U(v) = \sum_{u \in N_U(v)} w(v, u)$ represents the weighted degree of v. For simplicity, we omit U when $U = V$, and use $N(v)$ ($d(v)$) to denote the neighbors (weighted degree) of v.

Community: A community is represented by the symbol C, while the set of all communities is denoted by $\mathbf{C}$. As the edge weight of a community is frequently used, we further define $D_U(C) = \sum_{v \in C} d_U(v)$ as the weight of the edges between the vertex set U and the community C. Specifically, $D_C(C)$ denotes the edge weight within the community and $D_V(C)$ denotes the total edge weight of this community. Obviously, $2|E| = D_V(V)$. Moreover, we use $C[v]$ to represent the community that the vertex v belongs to.

Algorithm 1: Phase 1 of parallel Louvain algorithm

```
1  Function Louvain(G(V, E)):
2      repeat
3          foreach v ∈ V in parallel do
4              next_C[v] ← DecideAndMove(v)
5          Update the community set C by next_C
6          foreach v ∈ V in parallel do
7              Update d_{C[v]}(v) according to updated C
8          foreach C ∈ C in parallel do
9              D_C(C) ← Σ_{v∈C} d_C(v), D_V(C) ← Σ_{v∈C} d(v)
10             Calculate next_Q_C according to Equation 1
11         next_Q ← Σ_{C∈C} next_Q_C
12         ΔQ ← next_Q − Q, Q ← next_Q
13     until ΔQ < θ;
14 Function DecideAndMove(v):
15     best_C ← arg max_{C∈C} ΔQ_{v→C}
16     return best_C
```

Modularity [41]: The modularity Q is defined as:

$$Q = \sum_{C \in \mathbf{C}} Q_C = \sum_{C \in \mathbf{C}} \left(\frac{D_C(C)}{2|E|} - \left(\frac{D_V(C)}{2|E|} \right)^2 \right) \tag{1}$$

It is worth noting that each edge in the community is considered twice when $D_C(C)$ is calculated.

2.2 Louvain Algorithm

The Louvain algorithm is a greedy algorithm that increases modularity by moving a vertex to the community with the maximal modularity gain. Considering a vertex v moving into community C, the modularity gain can be calculated as:

$$\Delta Q_{v \rightarrow C} = \frac{1}{|E|} \left(d_C(v) - \frac{D_V(C) \cdot d(v)}{2|E|} \right) \tag{2}$$

Accordingly, the main process, namely the first phase, of the Louvain algorithm iteratively asks each vertex to call a `DecideAndMove` function formulated in lines 14-16 of Algorithm 1. Specifically, this function calculates the modularity gain $\Delta Q_{v \rightarrow C}$ for all possible movements of each vertex $v \in V$ to each community $C \in \mathbf{C}$ and then moves a vertex v to the community resulting in the greatest modularity gain. As it is impossible to move a vertex to a disconnected community, we only consider the neighboring community C that contains at least one of its neighbors, i.e., satisfying $\exists u, (u, v) \in E \wedge u \in C$. Initially, each vertex constitutes an individual community, and this procedure iterates until no further modularity gain is yielded.

The Louvain algorithm aims to construct a hierarchical community structure, which requires the second phase to build a compressed graph. In the compressed graph, a super vertex corresponds to a community in the original graph, while the weight of the super edge is formed by aggregating the edge weights between two different communities. Notice that the edge weights within a community (i.e., $D_C(C)$) are grouped into a self-loop edge. The two phases of the Louvain algorithm are repeated until modularity convergence is achieved. Obviously, the first phase in the initial round dominates the overall computation [15], and we focus on it.

2.3 Parallelization of Louvain Algorithm

One crucial distinction between sequential and parallel Louvain algorithms lies in how they handle state updates. Sequential algorithms update the state instantly as each vertex is processed, allowing for immediate adjustments. On the contrary, a main-streaming framework of parallel Louvain algorithm proposed by Grappolo [36] adopts a Bulk Synchronous Parallel (BSP) approach [56]. It segments the computation into iterations (supersteps), where state updates occur only at the end of each iteration. This delayed updating approach provides a consistent view of community assignment for all concurrent computations within an iteration. To distinguish state transitions across iterations, we use the prefix $prev\_$, $curr\_$, and $next\_$ to denote the states in the previous, current, and next iterations, respectively. The prefix $curr\_$ is omitted when the context is evident.

Algorithm 1 depicts the procedure of the standard parallel Louvain algorithm [36]. During each iteration, the algorithm concurrently processes different vertices, evaluating the potential community movements (lines 3-4). Subsequently, the community assignments and sets in $next\_C$ are synchronized and updated (line 5). In addition, the weight between a vertex v and its updated community $C[v]$ should also be updated (lines 6-7) based on the next iteration assignment. Lines 8-11 further calculate the updated modularity for the next iteration. Following Grappolo [1], we set a threshold θ to terminate the iteration (lines 12-13). In addition to Grappolo , Vite [24] further extends the Louvain algorithm to the distributed environment.

2.4 Existing GPU Implementations

Considering the massive parallelism and single instruction multiple thread (SIMT) model in GPUs, existing GPU implementations [8, 15, 22, 38, 39] further parallelize the computation within each vertex, mostly focusing on the dominant `DecideAndMove` function. A group of threads (e.g., a warp or block) will collaborate to determine the best community for a vertex. Unlike standard graph analytics tasks such as BFS and PageRank, `DecideAndMove` requires complex intermediate states, e.g., the weight between a vertex and its neighboring communities $d_C(v)$. The GPU implementation of Grappolo [39] and several variants [8, 22] use a hashtable to maintain the states but suffer from high global memory access overhead. Efforts in [8, 39] to use shared memory for smaller workloads fall short of a thorough optimization strategy. In contrast, the Louvain implementation in the graph algorithm library cuGraph [1] and several variants [15] rely on complex state transformation (e.g. sorting) to identify the best community, which introduces high complexity and memory access overhead. In addition, [38] leverages both CPU and GPU to execute the algorithm, and [16] proposes a batched implementation, scaling it up to multi-GPUs.

[1]We also adopt other heuristics in Grappolo to ensure the convergence.

2.5 Opportunities for Optimization

Upon examining the existing implementations, we identify two principal opportunities:

- A notable fraction of vertices (v) consistently remains unmoved across the execution of `DecideAndMove`, as evidenced by $next_C[v] = curr_C[v]$. The BSP model provides consistent states in an iteration, which can be leveraged to predict and prune these computations.
- The intermediate states within `DecideAndMove` function require a careful memory management strategy to achieve high performance. The memory and parallelism hierarchy in multi-GPUs can be better exploited.

Accordingly, in Section 3, we explored the pruning of computations for vertices that remain unmoved. Subsequently, in Section 4, we focus on optimizing the memory management within the `DecideAndMove` function.

3 Computation Pruning

As illustrated in Figure 1, a significant proportion of vertices remain unmoved in the later iterations of the Louvain algorithm, as the community partition stabilizes. We first establish the prediction problem of unmoved vertex and assess the impact of incorrect predictions (Section 3.1). We then review existing heuristic pruning strategies relying solely on movement information, which either suffer from insufficient pruning or degrade the modularity (Section 3.2). Fortunately, the Bulk Synchronous Parallel (BSP) model, integral to the parallel Louvain algorithm, offers additional states conducive to more effective pruning. Capitalizing on this, we propose the modularity gain-based strategy (Section 3.3). Furthermore, recognizing the naive implementation to calculate the community weight of a vertex is as expensive as the `DecideAndMove` function, we propose an updating operation (Section 3.5).

3.1 Prediction of the Unmoved Vertices

Formally, we define:

$$v \in \begin{cases} \text{unmoved set}, & \text{if } curr\_C[v].id = next\_C[v].id, \\ \text{moved set}, & \text{otherwise}, \end{cases} \tag{3}$$

where $C.id$ indicates the community id of C. Notice this definition hinges on community id consistency, rather than identical community sets. In certain scenarios, a community represented as $curr\_C_1$ may evolve into $next\_C_1$ by gaining or losing several vertices, yet it retains a stable id of 1 throughout this transition. Thus, vertices remaining unmoved do not necessitate further processing.

Prediction Problem: Exact identifying unmoved vertices is infeasible as it is as expensive as the `DecideAndMove` function. Thus, we aim to predict unmoved vertices before the iteration begins. The *active set* includes vertices that are predicted to be moved and will be processed in the next iteration, and vice versa for the *inactive set*. With the prediction result, we integrate the filter operation in popular GPU graph processing framework [34, 59] to prune inactive vertices.

There are two types of mispredictions: 1) *false positive (FP) instance* indicating a vertex remaining unmoved is incorrectly classified into the active set and 2) *false negative (FN) instance* indicating a vertex being moved is incorrectly classified into the inactive set. According to the definition, we have the following theorems:

Lemma 1. *The presence of false negative instances may miss opportunities for modularity gain resulting in modularity loss.*

Lemma 2. *The presence of false positive instances will not decrease the modularity but introduce unnecessary computation of mispredicted unmoved vertices.*

3.2 Existing Strategies and Their Weakness

Next, we review two existing heuristic strategies based solely on historical movement information from previous iterations. These strategies, however, come with inherent trade-offs. One strict strategy tends to generate a high number of false positive instances, which compromise pruning efficiency, while the other relaxed strategy incurs false negative instances, potentially leading to suboptimal community structures.

Strict Movement-based Strategy (SM) [50]: This strategy places a vertex v in the inactive set only if all neighboring communities containing the vertex itself and its neighbors remain unchanged. Notice, this strategy requires the entire set of the community to remain constant. Obviously,

Lemma 3. *The strict movement-based strategy can eliminate all false negative instances and guarantee modularity.*

However, this restriction is overly stringent as in most iterations, numerous communities are likely to be updated. As a result, this strategy leads to a significant number of false positives and suffers from insufficient computational pruning.

Relaxed Movement-based Strategy (RM) [50, 54]: Unlike the strict strategy which requires the unchanged community set, the Leiden algorithm [54] and its parallel adaptation [50] relax this condition to the unchanged community id. Specifically, the algorithm only activates the vertices whose neighbors have experienced community changes. Hence, it achieves a larger inactive set, thereby reducing computation effectively. However, the trade-off for this efficiency gain is the decreasing of modularity [50].

Lemma 4. *The relaxed movement-based strategy can introduce false negative instances and decrease the modularity.*

Proof. Revisiting the modularity gain defined in Equation 2, we observe the total weights of a community, $D_V(C)$, may have changed for an inactive vertex in relaxed strategy. Accordingly, we can construct the counterexample. Imagine two symmetrical communities C_1 and C_2 around a vertex v from the previous iteration, with v remaining in C_1. If another unrelated vertex (not neighbor of v) departs from C_2, reducing $D_V(C_2)$, v should move to C_2 in the current iteration to increase the modularity, illustrating a false negative instance. □

Similarly, Vite [24] introduces a probabilistic movement-based pruning strategy (PM) relying on the movement of the vertex itself, again, suffers from false negative instances. PM adjusts the probability α of a vertex being pruned by checking whether the community id of the vertex remain consistent across two consecutive iterations. With a probability of α, the vertex is classified into the inactive set. In the experiments of our paper, the parameter α is set to 0.25 by default.

3.3 Modularity Gain-based Pruning Strategy

The existing heuristic strategies are limited by their reliance on solely historical movement information, while we observe:

> *The synchronization nature of the BSP model ensures more consistent states within an iteration.*

Accordingly, we propose a novel **modularity gain-based strategy (MG)**, grounded in solid theoretical principles and enhanced by incorporating additional states, such as the community weight between a vertex and its community. Rather than construct a strategy by intuition, we start with the definition of modularity gain (refer to Equation 2).

Lemma 5. *A vertex v will be in the unmoved set (refer to Equation 3), if, for every neighbor u of v, moving v to the community $C[u]$ yields no more modularity gain than remaining in its current community $C[v]$, i.e., satisfying the following inequality:*

$$\forall u \in N(v), \Delta Q_{v \to C[v]} \geq \Delta Q_{v \to C[u]}. \tag{4}$$

In addition, this condition can be expanded by substituting Equation 2 as follows:

$$d_{C[v]}(v) - d_{C[u]}(v) + (D_V(C[u]) - D_V(C[v]))\frac{d(v)}{2|E|} \geq 0. \tag{5}$$

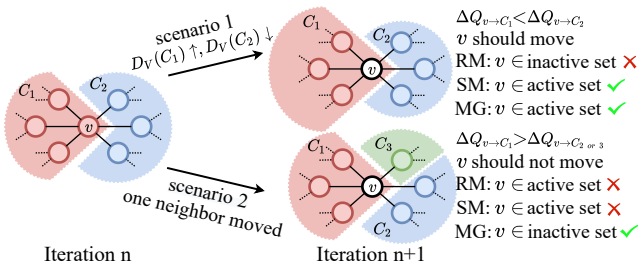


Figure 2. Three pruning strategies in two scenarios.

After reviewing the information required in Equation 5, we observe that the variables $C[v]$, $d_{C[v]}(v)$, and $D_V(C[v])$ are all directly related to vertex v, which is already available before the iteration. However, the variables $d_{C[u]}(v)$ and $D_V(C[u])$ involve community states of neighbor u, and their exact values are unknown unless we scan all the neighbors, which is exactly what the `DecideAndMove` function does. Instead, we try to identify as many unmoved vertices as possible by tightening the constraint with an upper bound on the community states required by neighbor u (details refer to the proof of Lemma 5), and obtain the following loose constraint:

$$2d_{C[v]}(v) - d(v) + (\min_{C\in\mathbf{C}}(D_V(C)) - D_V(C[v]))\frac{d(v)}{2|E|} \geq 0. \tag{6}$$

This leads to our modularity gain-based strategy:

$$v \in \begin{cases} \text{inactive set}, & \text{if Equation 6 holds,} \\ \text{active set}, & \text{otherwise.} \end{cases} \tag{7}$$

Theorem 6. *The modularity gain-based strategy can eliminate all false negative instances and guarantee modularity.*

Proof. We first prove that by ensuring Equation 6 is satisfied, Equation 5 will also hold. It is clear that $d_{C[u]}(v) \leq d(v) - d_{C[v]}(v)$. The left-hand side achieves its maximum when all remaining neighbors not in the current community $C[v]$ belong to the community $C[u]$. Furthermore, total edge weight of a community $C[u]$, denoted as $D_V(C[u])$, satisfies the inequality $D_V(C[u]) \geq \min_{C\in\mathbf{C}}(D_V(C))$. Therefore, we have:

$$\begin{aligned} &d_{C[v]}(v) - d_{C[u]}(v) + (D_V(C[u]) - D_V(C[v]))\frac{d(v)}{2|E|} \\ \geq &2d_{C[v]}(v) - d(v) + (\min_{C\in\mathbf{C}}(D_V(C)) - D_V(C[v]))\frac{d(v)}{2|E|} \end{aligned}$$

According to Lemma 5, a vertex in the inactive set will not be moved in the next iteration. Therefore, the modularity gain-based strategy can eliminate all false negative instances and guarantee modularity. □

As outlined in lines 6-10 of Algorithm 1, all the required data or computing modularity in Equation 6 is readily available, enabling efficient classification.

Graph	FNR				FPR			
	SM	RM	PM	MG	SM	RM	PM	MG
FR	0.00%	0.60%	6.20%	0.00%	96.80%	12.69%	12.92%	22.38%
LJ	0.00%	0.71%	2.91%	0.00%	86.59%	21.35%	25.55%	31.47%
OR	0.00%	0.61%	17.59%	0.00%	99.75%	44.59%	39.16%	35.32%
TW	0.00%	0.10%	0.06%	0.00%	98.63%	84.48%	87.64%	60.27%
UK	0.00%	0.37%	0.01%	0.00%	75.12%	56.14%	99.99%	26.14%
EW	0.00%	0.08%	1.41%	0.00%	99.95%	45.37%	41.06%	32.15%
HW	0.00%	0.15%	16.26%	0.00%	85.26%	12.83%	24.98%	17.93%
Avg.	0.00%	0.37%	6.35%	0.00%	91.73%	39.64%	47.33%	32.24%

Table 1. FNR and FPR of three strategies. The abbreviations in column "Graph" are explained in Table 2. The best results are highlighted in yellow.

3.4 Comparison of Pruning Strategies

Example 1. Figure 2 demonstrates pruning strategies in two scenarios. In the first scenario, the neighbors of vertex v remain unmoved, where the RM strategy will misclassify v as inactive. However, the sizes of neighboring communities are evolving, resulting in movement opportunities for v. In the second scenario, one neighbor of v belonging to a different community is moved. Obviously, staying in the current community is the optimal choice for v, while the SM and RM strategies will misclassify v as active.

Empirical Comparison: Table 1 further illustrates the average FNR (false negative rate indicating the proportion of misclassified vertices that will be moved) and FPR (false positive rate indicating the proportion of misclassified vertices that will remain unmoved) of different strategies over all iterations. We observe that the SM strategy achieves perfect 0.00% FNR, but at the cost of a high 91.73% FPR. In contrast, the RM and PM strategies achieves a lower 39.64% and 47.33% FPR, but at the cost of a non-zero FNR of 0.37% and 6.35%, respectively. In contrast, the MG avoids false negative instances and achieves a lower 32.24% FPR. We also notice that all strategies perform poorly on the Twitter (TW) graph which lacks a well-defined community structure and is also reflected in its low modularity (Table 3). The blurred boundaries of the communities make it challenging to predict unmoved vertices, since a vertex in the TW graph may have several potential communities to which it could belong.

3.5 Efficient Community Weight Updating

A naive implementation of calculating community weight $d_{C[v]}(v)$ (lines 6-7 of Algorithm 1), which scans all neighbors of vertex v, results in the same computational complexity as the `DecideAndMove` function. Therefore, in this subsection, we introduce an efficient strategy by updating the states in the previous iteration instead of recomputing the community weight from scratch.

As discussed earlier, a significant portion of vertices remain unmoved in the later iterations of the Louvain algorithm, and we can take advantage of their previous community weight $prev\_d_{C[v]}(v)$ and update it through tracking

Algorithm 2: Shuffle-based kernel

```
1  Function DecideAndMove(v,threadId):
2      u ← N(v)[threadId] // get threadId-th neighbor
3      my_C ← C[u] // my_C stores the community id
4      my_w ← w(u,v)
5      mask ← __match_any_sync(full_mask, my_C)
6      d_{my_C}(v) ← __reduce_add_sync(mask, my_w)
7      Calculate ΔQ_{v→my_C}, my_ΔQ ← ΔQ_{v→my_C}
8      max_ΔQ ← __reduce_max_sync(full_mask, my_ΔQ)
9      if max_ΔQ = my_ΔQ then
10         Move v to community my_C
```

the movement of their neighbors, in other words, using delta update. Specifically, we propose a mechanism where each moved vertex informs its neighbors of its new community. When an unmoved vertex receives these messages, it updates its previous community weight to obtain the current value. For each moved vertex, we still recompute its own community weight. Consequently, the computation complexity is related to the moved vertices, whose number is typically less than that of inactive vertices, thereby alleviating the computational burden of the updating process. As the updating process is no longer the bottleneck, we will concentrate on optimizing the `DecideAndMove` function.

4 Memory Management on GPUs

To further boost the performance and scalability of the Louvain algorithm, we turn our attention to leveraging the massive parallelism and memory hierarchy of GPUs. According to the workload, we adopt resource assignment strategies [13, 35, 37, 45] and design two distinct kernels for the `DecideAndMove`. Specifically, for the small-degree vertex, we employ a shuffle-based kernel, utilizing a warp and its associated registers. On the other hand, large-degree vertices are processed using a hash-based kernel, which allocates a block and primarily utilizes shared memory for computation, with global memory managing any overflowed data.

4.1 Warp-level Shuffle-based Kernel

To simplify our discussion, we assume that thread i is responsible for managing the input of u_i, where u_i represents i-th neighbor of v. Note that extending this concept to a thread handling multiple neighbors can be seamlessly achieved through loop. Algorithm 2 presents the shuffle-based kernel. The input neighbors of v are distributed in the warp (line 2), where each thread maintains $C[u_i]$ and (v, u_i) in variables named my_C and my_w in its register (lines 3-4), respectively. To identify threads with the same community $C[u_i]$, we utilize a built-in CUDA warp-level function called `__match_any_sync`($full\_mask, my\_C$), which generates a $mask$ for each thread (line 5). The j-th bit in the $mask$ obtained by thread i indicates whether the value of my_C of thread j is the same as that of thread i, i.e., $C[u_i] = C[u_j]$. Using the generated $mask$, `__reduce_add_sync`($mask, my\_w$) is performed to compute the sum of my_w for all threads whose corresponding bits in $mask$ are set to 1, i.e., $d_{my\_C}(v)$ (line 6). The result of the reduce operation provides the necessary information to compute the maximal modularity gain, which can be obtained by invoking the `__reduce_max_sync()` function after calculating $my\_\Delta Q$ for each thread (lines 7-8).

Algorithm 3: Hash-based kernel

```
1  Function DecideAndMove(v,threadId):
2      Initialize my_max_ΔQ ← 0, my_best_C
3      Initialize hashtable H : C → (d_C(v), D_V(C))
4      for i ← threadId; i < len(N(v)); i+ = blockSize do
5          u ← N(v)[i] // get i-th neighbor
6          if C[u] ∉ H then
7              Insert C[u] into H
8              Initialize temp_d_{C[u]}(v) ← 0 in H
9              Load D_V(C[u]) into H
10         update temp_d_{C[u]}(v) in H by adding w(u,v)
11         Calculate ΔQ_{v→C[u]}
12         if ΔQ_{v→C[u]} > my_max_ΔQ then
13             my_max_ΔQ ← ΔQ_{v→C[u]}
14             my_best_C ← C[u]
15     Obtain best_C of all threads from my_best_C
16     Move v to community best_C
```

4.2 Block-level Hash-based Kernel

As the number of neighbors and communities increases, the limited capacity of registers may not be sufficient to store all community states of neighbors. To address this, we adopt a concurrent hashtable $H : C \rightarrow (d_C(v), D_V(C))$ shared among the threads to maintain the states. We notice there also exists conflict-free reduction-based solutions [32] that replicate the hash table to each thread, which is not suitable for GPUs with massive cores. Given that the workload is large enough and the shared memory is attached to a block, we employ a block to handle these tasks.

Algorithm 3 describes the hash-based kernel. Upon loading a neighbor (line 5), each thread checks whether its corresponding community exists in the hashtable (line 6). If the community does not exist, the thread inserts it into an empty bucket of the hashtable (lines 7-9). Subsequently, the weight between the community and the vertex, represented by $temp\_d_{C[u]}(v)$, is updated. To ensure thread safety and avoid concurrency issues, we utilize the `atomic` operations in CUDA. Specifically, we employ `atomicCAS` to find the position of the key, i.e., $C[u]$, and utilize `atomicAdd` to update the value, i.e., $temp\_d_{C[u]}(v)$. Next, we calculate the modularity gain and record the community with the maximal gain (lines 11-14).[2]

[2] As $temp\_d_{C[u]}(v)$ is gradually updated, we may initially obtain a partial result leading to the lower modularity gain. However, since there always exists a thread that eventually obtains the final result of $d_{C[u]}(v)$, correctness is ensured.

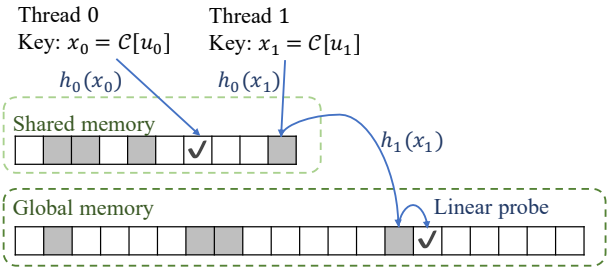


Figure 3. Insert the hierarchical hashtable.

The kernel design reveals the importance of efficiently managing the hashtable. Subsequently, we delve into the design of the hashtable, focusing on the shared memory utilization.

Global-only: A naive solution *exclusively* relies on global memory for maintaining hashtable [8, 15, 39], which overlooks the importance of shared memory and suffers from slow global memory access.

The advanced solutions should leverage both shared memory and global memory. Accordingly, we break the hashtable H and its buckets into two parts, i.e., H_0 with s buckets and H_1 with g buckets, which are maintained in shared memory and global memory, respectively.

Unified: The unified solution employs a single hash function h to unify the address of buckets in shared memory H_0 and global memory H_1. The element x with $h(x) \in [0, s)$ will be maintained in shared memory and with $h(x) \in [s, s+g)$ in global memory. Obviously, when employing a truly random hash function [25], an element has only a probability of $s/(s+g)$ falling into the shared memory. Actually, the approach implicitly assigns *equal importance* to both shared memory and global memory, resulting in poor efficiency.

Hierarchical: This solution *prioritizes shared memory* during hashtable accessing. Specifically, we employ two hash functions h_0 and h_1 to index the buckets in shared memory and global memory, respectively. Considering inserting (or accessing) an element x, we first look into the buckets in shared memory hashtable H_0 using the hash function h_0. If the bucket in shared memory is empty, the thread can directly place its community. Only when the bucket in shared memory is already occupied (i.e., a collision occurs), do we turn to the bucket in global memory H_1 using a new hash function h_1 for indexing. In case the bucket in global memory is also occupied, we employ a linear probing strategy to search for the next available bucket.

Example 2. Figure 3 demonstrates the design of our hierarchical hashtable, where each block indicates a hash bucket. The white block indicates the bucket is empty, and the gray indicates occupied. When thread 0 accesses the $h_0(x_0)$-th bucket in shared memory and finds it empty, it can directly place its community. Another example of thread 1 demonstrates the insertion when a collision occurs. Thread 1 first accesses the bucket $h_0(x_1)$ in shared memory, which is already occupied. Then it uses the new index $h_1(x_1)$ and accesses the corresponding bucket in global memory. Unfortunately, the $h_1(x_1)$-th bucket in global memory is still occupied, so the thread linearly probes the next bucket, which is empty.

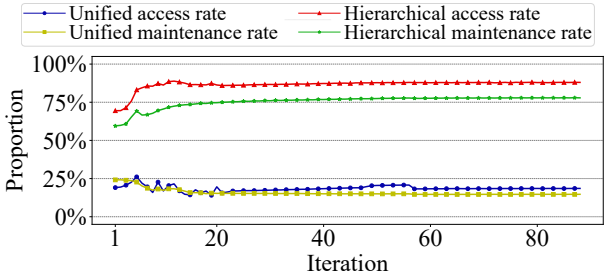


Figure 4. The maintenance rate and access rate in the shared memory of two kinds of hashtable on the LiveJournal graph.

Empirical Comparison: We evaluate the rate of a community maintained (accessed) in the shared memory, namely *maintenance (access) rate*, for two strategies. As shown in Figure 4, the hierarchical hashtable significantly outperforms the unified hashtable, increasing the access rate by 4.7×. Moreover, the rates of the hierarchical hashtable show an increasing trend as the iteration proceeds, in contrast to the unified approach. This difference stems from the rate of hierarchy hashtable is related to the number of communities, which decreases as the iteration proceeds, while the rate of unified hashtable is determined by the fixed neighbor number. We further note that the access rate is higher than the maintenance rate since a frequently updated community is more likely to appear early and remain in shared memory.

4.3 Scale up with multiple GPUs

Natually, the Louvain algorithm follows the vertex-centric computation approach, which can be easily parallelized by partitioning the vertices and their corresponding neighbors across multiple GPUs. Therefore, the intermediate states of each vertex are handled by the corresponding GPU owning the vertex, which significantly reduces the communication overhead. After each iteration, the states of the vertices must be synchronized across GPUs, including the community ID for a vertex, movement indicators for both itself and its neighbors, and the weight associated with its community.

Dense vs. Sparse representation [18]. The pruning concept illustrated in Figure 1 is also applicable to the synchronization. During the initial iterations, most vertices undergo frequent community changes, leading to a dense synchronization pattern—where we synchronize the states of each vertex directly via the `ncclAllReduce` interface provided by the NCCL library [2]. In contrast, in later iterations, only a small fraction of vertices move to new communities, resulting in a sparse synchronization pattern, thus we employ a delta synchronization strategy, focusing solely on the vertices that have changed. Therefore, we employ the `ncclAllGather` interface, which reduces communication

time at the cost of introducing a slight data rearrangement overhead. As the number of moved vertices shows a decreasing trend over iterations, we establish a threshold according to communication size to determine when to switch from dense to sparse synchronization.

5 Evaluation

Graph	Abbr.	#Vertices	#Edges
com-Friendster [60]	FR	65.6M	1.8B
com-LiveJournal [60]	LJ	4.0M	34.6M
com-Orkut [60]	OR	3.1M	117.2M
twitter-2010 [28]	TW	41.7M	1.2B
uk-2002 [10]	UK	18.5M	298.1M
enwiki-2022 [11, 12]	EW	6.5M	144.6M
hollywood-2011 [11, 12]	HW	2.0M	114.5M

Table 2. Statistics of the graphs used in our experiments.

GALA [3] is implemented in C++ and CUDA, and the experiments are conducted on a server with two 28-core Intel Xeon Gold 6330 CPUs and NVIDIA A100 GPUs with 40 GB of memory and NVLink. The GPU code is compiled with nvcc compiler (version 11.6) with the '-arch=sm_80' option. The graph datasets used in the experiments are shown in Table 2. The column Abbreviation provides the abbreviations of the data graphs. Since the concept of modularity is proposed for undirected graphs [40], we convert directed graphs (e.g., TW, EW) into undirected graphs for evaluation. The experiments, except for 5.1 and 5.3, only focus on phase 1 of the first round of Louvain which dominates the runtime. The threshold θ in our experiment is set to 10^{-6}. Unless otherwise stated, the evaluation is conducted on a single GPU and the graphs are already loaded into the GPU memory.

5.1 Comparison with State of the Art

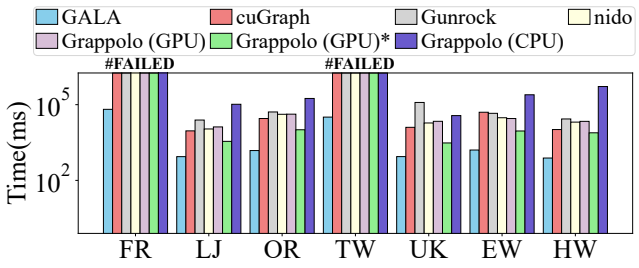


Figure 5. Comparison with state of the art.

We conduct a comparative evaluation of GALA against state-of-the-art GPU implementations of the Louvain algorithm: 1) cuGraph [1], a GPU-accelerated graph analytics library developed by NVIDIA. 2) Gunrock [42, 59], a high-performance graph processing framework. 3) nido [16], a GPU-accelerated Louvain algorithm implementation. 4) GPU implementation of Grappolo [39]. 5) as the GPU version of Grappolo not being updated for some time, we have made some modifications to make it compatible with the latest CUDA version, herein referred to as Grappolo (GPU)*. 6) CPU implementation of Grappolo [36]. As demonstrated in Figure 5, our proposed GALA outperforms cuGraph, Gunrock, nido, Grappolo (GPU), Grappolo (GPU)* and Grappolo (CPU) by an average of 17×, 53×, 21×, 22×, 6× and 222×, respectively. The superior performance of GALA can be attributed not only to the novel computational and memory optimizations but also to our well-optimized implementation that fully leverages the underlying GPU hardware architecture. Moreover, GALA is capable of processing two large graphs (FR and TW) in approximately one minute, while other implementations faced runtime failures (#FAILED). Specifically, cuGraph and Gunrock fail due to out-of-memory (OOM) errors, while grappolo (GPU) fails because the code does not support large graphs. Additionally, the running times of nido and grappolo (CPU) exceed 30 minutes.

Since GALA and these state-of-the-art methods follow the convergence strategy proposed by Grappolo, the modularity values are identical.

5.2 Impact of Optimizations

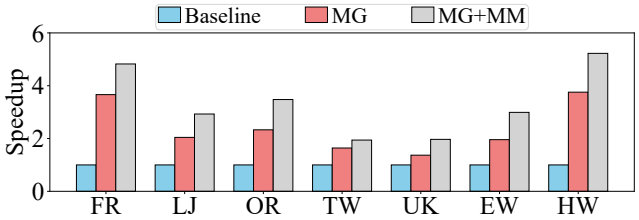


Figure 6. Impact of optimizations. MG and MM refer to the modularity gain-based pruning optimization and memory management optimization on GPUs, respectively.

Figure 6 illustrates the comparison between the baseline (without any pruning techniques and the utilized hash table is stored in global memory), modularity gain-based pruning optimization, and the combination of both pruning and GPU-specific memory management optimizations. The modularity gain-based pruning optimization (MG) provides a performance improvement of 2.4× on average. Interestingly, we observe the trend that pruning optimization performs better on larger graphs, such as FR (3.7×). The behind reason is that larger graphs require more iterations to converge, while only a few vertices are moved in later iterations. Adopting GPU-specific memory management optimizations (MM) further offers 1.4× speedup, and gives an overall 3.4× speedup. Next, we will investigate the factors contributing to the two optimizations.

5.3 Comparison of Pruning Strategies

Efficiency: Figure 7 compares our modularity gain-based pruning strategy (MG) with two other heuristic pruning strategies mentioned in Section 3.1 iteration by iteration.

[3] Available at https://github.com/LinXi-lx/GALA

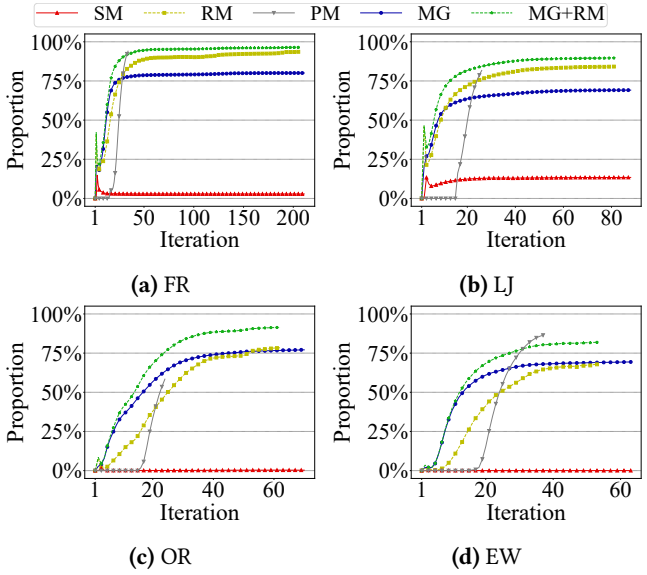


Figure 7. Comparison of the pruned proportion (inactive rate) on four representative graphs. SM, RM, PM and MG indicate pruning strategies based on strict movement, relaxed movement, probability movement and modularity gain, respectively.

To focus on the most computationally intensive part of the process, we have limited our exposition to the first round. Consistent with the previous analysis, the worst performance is with the strict movement-based pruning strategy (SM), where less than 4% of vertices are pruned on average. In contrast, the relaxed movement-based pruning strategy (RM) and the probability movement-based pruning strategy (PM) demonstrate competitive performance compared with our strategy (MG). It is worth noting that PM terminates earlier than the other strategies, which is due to the fact that PM is more aggressive in pruning vertices, leading to suboptimal community detection results. In addition, as mentioned in Figure 1, the pruning performance shows a trend of increasing as the iteration proceeds.

Furthermore, the integration of both MG and RM strategies (denoted as MG+RM in Figure 7) is also evaluated. If a user can tolerate the modularity loss associated with RM, incorporating MG can further reduce 37% vertices in the active set produced by RM and results in an impressive overall up to 91.9% pruning efficiency. This indicates that MG and RM are not competitive but complementary since they prune from different angles.

Modularity: Table 3 compares the modularity values of the baseline (unpruned), MG, SM, PM and RM strategies. In accordance with the analysis in Section 3, the MG and SM strategies maintain modularity without loss, while the RM and PM strategies introduce an average modularity loss of 0.00119 and 0.00413, respectively.

NMI (Normalized Mutual Information) [52]: We further employ graphs generated by the LFR benchmark [31] with ground truth communities to evaluate the quality of the community detection results, where the NMI value ranges from 0 to 1, with 1 indicating the perfect match with the ground truth communities. Table 4 shows the NMI values of the baseline, MG, SM, PM and RM strategies. We find that RM and PM reduce the NMI value by 0.2% and 0.3% on average, respectively.

Graph	Baseline/MG/SM	RM/MG+RM	PM
FR	0.63022	0.63018 (0.00004)	0.62251 (0.00771)
LJ	0.75153	0.75139 (0.00014)	0.74952 (0.00201)
OR	0.66487	0.66476 (0.00011)	0.65758 (0.00729)
TW	0.47257	0.46594 (0.00663)	0.46488 (0.00769)
UK	0.99056	0.99052 (0.00004)	0.99055 (0.00001)
EW	0.66297	0.66159 (0.00138)	0.65882 (0.00415)
HW	0.75323	0.75315 (0.00008)	0.75319 (0.00004)

Table 3. Modularity comparisons. The values in parentheses represent the difference from the baseline.

	#Vertices	#Edges	Baseline/MG/SM	RM/MG+RM	PM
Graph1	100,000	651,183	0.35041	0.34900	0.34782
Graph2	100,000	1,440,079	0.92358	0.92325	0.92326
Graph3	100,000	1,442,400	0.43440	0.43371	0.43431

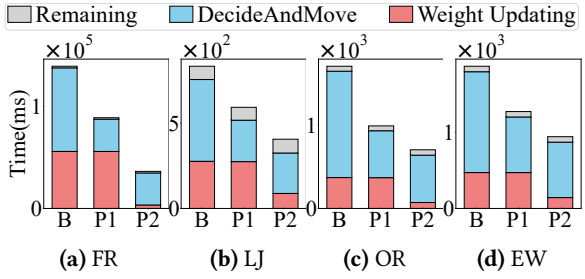
Table 4. NMI comparisons. The values in yellow represent the best results.

5.4 Two-Stage Pruning Profiling

As discussed in Section 3.5, our pruning optimization has two stages: 1) pruning the inactive vertices in `DecideAndMove`, and 2) pruning the unnecessary weight recalculation through efficient weight updating. Figure 8 provides a performance breakdown for this two-stage pruning optimization. Initially, in the baseline scenario with no pruning optimization (B), `DecideAndMove` dominates the runtime (65.5%). After applying the first stage of pruning (P1), the weight updating impedes the performance, takes up 45.7% of runtime. Subsequently, the second stage of pruning (P2) significantly accelerates the weight updating by a factor of 7.3×, shifting the performance bottleneck back to `DecideAndMove`.

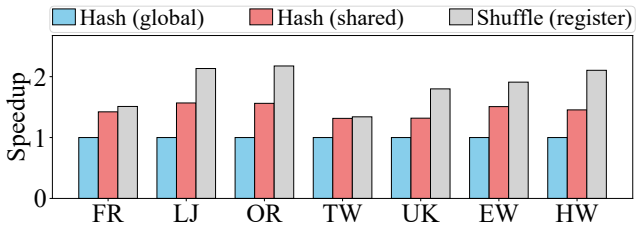
5.5 Memory Management Optimization on Varying Workload

Our memory management optimization strategy includes two types of GPU kernels, i.e., shuffle-based kernel and hash-based kernel, which are dispatched for different workloads. To evaluate the benefits of each optimization strategy, we conduct experiments on two representative scenarios, one is regarding small degree vertices with degrees less than 32,

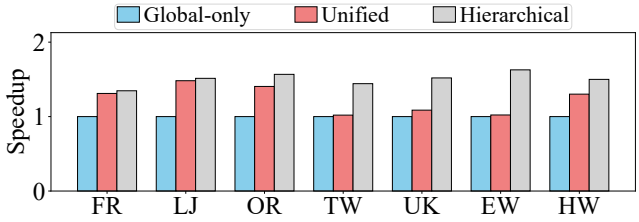


(a) FR **(b)** LJ **(c)** OR **(d)** EW

Figure 8. The performance breakdown, where B, P1, P2 represent the baseline, pruning `DecideAndMove`, and pruning both `DecideAndMove` and weight updating, respectively.



(a) Performance on vertices with small degrees.



(b) Performance on vertices with large degrees.

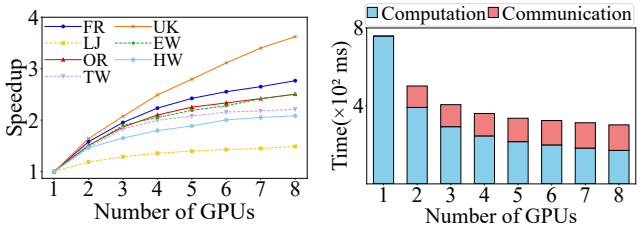
Figure 9. Impact of Memory Management optimizations.

where the states can be fit into registers of a warp, and another is considering large degree vertices with degree larger than 2,000, where the states overflow the shared memory.

Small Degree: Figure 9(a) compares shuffle-based kernel and hash-based kernel when using warp to handle a vertex with a small degree. Shuffle-based kernel optimization achieves an average speedup of 1.9× when compared with hash-based kernel with global memory. Meanwhile, it also outperforms hash-based kernel with shared memory by 1.2× on average. Obviously, the high efficiency of the shuffle-based kernel is based on leveraging the fastest memory, i.e. register. Moreover, we observe that the shuffle-based kernel exhibits a modest improvement on large graphs. This can be attributed to the fact that larger graphs typically contain numerous communities, leading to an increased number of reduced operations, as each community of neighboring vertices necessitates a reduction step.

Large Degree: Figure 9(b) compares three hashtable strategies described in Section 4.2. Our experimental results demonstrate that incorporating a hierarchical hash-based kernel optimization approach results in an average speedup of 1.5× and 1.2× compared to the scenario with global-only hashtable and unified hashtable, respectively. In addition, we observe that the unified hashtable performs worse on graphs where the maximum vertex degree is large, e.g., TW, UK, and EW. This can be attributed to the fact that in such cases, most of the buckets are allocated in global memory, while the unified hashtable assigns equal importance to both shared memory and global memory. As a result, most communities are maintained in the global memory.

5.6 Scalability



(a) Speedup. **(b)** Time breakdown of OR.

Figure 10. Scalability.

Figure 10(a) shows that GALA scales with an average speedup of 2.5× when moving from 1 to 8 GPUs. However, the speedup is sub-linear due to communication overhead between GPUs. Figure 10(b) provides the time breakdown for the OR graph. Computation time decreases by 4.4× from 1 to 8 GPUs, while communication overhead remains nearly constant. Specifically, on 8 GPUs, communication costs account for 43% of the total runtime. In addition, we also run Phase 1 of the first round of the algorithm on the uk-2007-02 graph with 3.4B edges, and the results show that it completed in 43 seconds.

6 Conclusion

In this paper, we introduce GALA, a novel approach that significantly accelerates the Louvain algorithm on GPUs. GALA incorporates two key innovations: 1) a modularity gain-based pruning strategy that effectively reduces unnecessary computation while preserving optimality, and 2) a memory management strategy that employs two kinds of GPU kernels to explore the potential of the memory hierarchy in GPUs. Taken together, GALA outperforms state-of-the-art solutions by 6× on average.

Acknowledgments

The authors would like to thank anonymous reviewers for their valuable comments. This research is supported by the National Natural Science Foundation of China under Grant Numbers 62325205 and 62272215, The Key Program of Natural Science Foundation of Jiangsu under grant No. BK20243053.

References

[1] [n.d.]. cuGraph. https://github.com/rapidsai/cugraph.

[2] [n.d.]. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl

[3] 2017. Zebra network dataset – KONECT. http://konect.cc/networks/moreno_zebra.

[4] A Arenas, A Fernández, and S Gómez. 2008. Analysis of the structure of complex networks at different resolution levels. *New Journal of Physics* 10, 5 (May 2008), 053039. https://doi.org/10.1088/1367-2630/10/5/053039

[5] Saman Ashkiani, Martin Farach-Colton, and John D Owens. 2018. A dynamic hash table for the GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 419–429.

[6] Muhammad A Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D Owens. 2019. Engineering a high-performance gpu b-tree. In *Proceedings of the 24th symposium on principles and practice of parallel programming*. 145–157.

[7] Punam Bedi and Chhavi Sharma. 2016. Community detection in social networks. *Wiley interdisciplinary reviews: Data mining and knowledge discovery* 6, 3 (2016), 115–135.

[8] Anwesha Bhowmik and Sathish Vadhiyar. 2019. Hydetect: A hybrid cpu-gpu algorithm for community detection. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2–11.

[9] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.

[10] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience* 34, 8 (2004), 711–726.

[11] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM Press, 587–596.

[12] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[13] Xuhao Chen et al. 2022. Efficient and Scalable Graph Pattern Mining on {GPUs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 857–877.

[14] Jiafeng Cheng, Qianqian Wang, Zhiqiang Tao, Deyan Xie, and Quanxue Gao. 2021. Multi-view attribute graph convolution networks for clustering. In *Proceedings of the twenty-ninth international conference on international joint conferences on artificial intelligence*. 2973–2979.

[15] Chun Yew Cheong, Huynh Phung Huynh, David Lo, and Rick Siow Mong Goh. 2013. Hierarchical parallel algorithm for modularity-based community detection using GPUs. In *Euro-Par 2013 Parallel Processing: 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings 19*. Springer, 775–787.

[16] Han-Yi Chou and Sayan Ghosh. 2023. Batched Graph Community Detection on GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 172–184. https://doi.org/10.1145/3559009.3569655

[17] Matteo Cinelli, Gianmarco De Francisci Morales, Alessandro Galeazzi, Walter Quattrociocchi, and Michele Starnini. 2021. The echo chamber effect on social media. *Proceedings of the National Academy of Sciences* 118, 9 (2021), e2023301118.

[18] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN conference on programming language design and implementation*. 752–768.

[19] Paul Expert, Tim S Evans, Vincent D Blondel, and Renaud Lambiotte. 2011. Uncovering space-independent communities in spatial networks. *Proceedings of the National Academy of Sciences* 108, 19 (2011), 7663–7668.

[20] Behnam Fahimnia, Joseph Sarkis, and Hoda Davarzani. 2015. Green supply chain management: A review and bibliometric analysis. *International Journal of Production Economics* 162 (2015), 101–114.

[21] Mahmood Fazlali, Ehsan Moradi, and Hadi Tabatabaee Malazi. 2017. Adaptive parallel Louvain community detection on a multicore platform. *Microprocessors and Microsystems* 54 (2017), 26–34. https://doi.org/10.1016/j.micpro.2017.08.002

[22] Richard Forster. 2016. Louvain community detection with parallel heuristics on GPUs. In *2016 IEEE 20th Jubilee International Conference on Intelligent Engineering Systems (INES)*. 227–232.

[23] Santo Fortunato. 2010. Community detection in graphs. *Physics reports* 486, 3-5 (2010), 75–174.

[24] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, Hao Lu, Daniel Chavarria-Miranda, Arif Khan, and Assefaw Gebremedhin. 2018. Distributed louvain algorithm for graph community detection. In *2018 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 885–895.

[25] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. 1986. How to construct random functions. *Journal of the ACM (JACM)* 33, 4 (1986), 792–807.

[26] Di Jin, Zhizhi Yu, Pengfei Jiao, Shirui Pan, Dongxiao He, Jia Wu, S Yu Philip, and Weixiong Zhang. 2021. A survey of community detection approaches: From statistical modeling to deep learning. *IEEE Transactions on Knowledge and Data Engineering* 35, 2 (2021), 1149–1170.

[27] Brian Karrer and M. E. J. Newman. 2011. Stochastic blockmodels and community structure in networks. *Physical Review E* 83, 1 (Jan. 2011). https://doi.org/10.1103/physreve.83.016107

[28] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web*. 591–600.

[29] Zhuohang Lai, Xibo Sun, Qiong Luo, and Xiaolong Xie. 2022. Accelerating multi-way joins on the GPU. *The VLDB Journal* (2022), 1–25.

[30] Andrea Lancichinetti and Santo Fortunato. 2011. Limits of modularity maximization in community detection. *Physical Review E* 84, 6 (Dec. 2011). https://doi.org/10.1103/physreve.84.066122

[31] Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. 2008. Benchmark graphs for testing community detection algorithms. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* 78, 4 (2008), 046110.

[32] Hochan Lee, Roshan Dathathri, and Keshav Pingali. 2024. Kimbap: A Node-Property Map System for Distributed Graph Analytics. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 566–581.

[33] Yuchen Li, Qiwei Zhu, Zheng Lyu, Zhongdong Huang, and Jianling Sun. 2021. Dycuckoo: dynamic hash tables on gpus. In *2021 IEEE 37th international conference on data engineering (ICDE)*. IEEE, 744–755.

[34] Hang Liu and H Howie Huang. 2019. {SIMD-X}: Programming and processing of graph algorithms on {GPUs}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 411–428.

[35] Hang Liu, H Howie Huang, and Yang Hu. 2016. ibfs: Concurrent breadth-first search on gpus. In *Proceedings of the 2016 International Conference on Management of Data*. 403–416.

[36] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. 2015. Parallel heuristics for scalable community detection. *Parallel Comput.* 47 (2015), 19–37.

[37] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. *ACM Sigplan Notices* 47, 8 (2012), 117–128.

[38] Maryam Mohammadi, Mahmood Fazlali, and Mehdi Hosseinzadeh. 2021. Accelerating Louvain community detection algorithm on graphic processing unit. *The Journal of supercomputing* 77 (2021), 6056–6077.

[39] Md. Naim, Fredrik Manne, Mahantesh Halappanavar, and Antonino Tumeo. 2017. Community Detection on the GPU. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 625–634. https://doi.org/10.1109/IPDPS.2017.16

[40] M. E. J. Newman. 2004. Fast algorithm for detecting community structure in networks. *Physical Review E* 69, 6 (jun 2004). https://doi.org/10.1103/physreve.69.066133

[41] M. E. J. Newman and M. Girvan. 2004. Finding and evaluating community structure in networks. *Physical Review E* 69, 2 (Feb. 2004). https://doi.org/10.1103/physreve.69.026113

[42] Muhammad Osama, Serban D. Porumbescu, and John D. Owens. 2022. Essentials of Parallel Graph Analytics. In *Proceedings of the Workshop on Graphs, Architectures, Programming, and Learning (GrAPL 2022)*. 314–317. https://doi.org/10.1109/IPDPSW55747.2022.00061

[43] Naoto Ozaki, Hiroshi Tezuka, and Mary Inaba. 2016. A simple acceleration method for the Louvain algorithm. *International Journal of Computer and Electrical Engineering* 8, 3 (2016), 207.

[44] Santosh Pandey, Xiaoye Sherry Li, Aydin Buluc, Jiejun Xu, and Hang Liu. 2019. H-index: Hash-indexing for parallel triangle counting on GPUs. In *2019 IEEE high performance extreme computing conference (HPEC)*. IEEE, 1–7.

[45] Santosh Pandey, Zhibin Wang, Sheng Zhong, Chen Tian, Bolong Zheng, Xiaoye Li, Lingda Li, Adolfy Hoisie, Caiwen Ding, Dong Li, et al. 2021. Trust: Triangle counting reloaded on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 32, 11 (2021), 2646–2660.

[46] Tiago P. Peixoto. 2014. Efficient Monte Carlo and greedy heuristic for the inference of stochastic block models. *Physical Review E* 89, 1 (Jan. 2014). https://doi.org/10.1103/physreve.89.012804

[47] Tinca JC Polderman, Beben Benyamin, Christiaan A De Leeuw, Patrick F Sullivan, Arjen Van Bochoven, Peter M Visscher, and Danielle Posthuma. 2015. Meta-analysis of the heritability of human traits based on fifty years of twin studies. *Nature genetics* 47, 7 (2015), 702–709.

[48] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* 76, 3 (2007), 036106.

[49] Martin Rosvall and Carl T Bergstrom. 2011. Multilevel compression of random walks on networks reveals hierarchical organization in large integrated systems. *PloS one* 6, 4 (2011), e18209.

[50] Jessica Shi, Laxman Dhulipala, David Eisenstat, Jakub Lacki, and Vahab S. Mirrokni. 2021. Scalable Community Detection via Parallel Correlation Clustering. *Proc. VLDB Endow.* 14, 11 (2021), 2305–2313.

[51] Christian L Staudt and Henning Meyerhenke. 2015. Engineering parallel algorithms for community detection in massive networks. *IEEE Transactions on Parallel and Distributed Systems* 27, 1 (2015), 171–184.

[52] Alexander Strehl and Joydeep Ghosh. 2002. Cluster ensembles—a knowledge reuse framework for combining multiple partitions. *Journal of machine learning research* 3, Dec (2002), 583–617.

[53] Jesmin Jahan Tithi, Andrzej Stasiak, Sriram Aananthakrishnan, and Fabrizio Petrini. 2020. Prune the unnecessary: Parallel pull-push louvain algorithms with automatic edge pruning. In *Proceedings of the 49th International Conference on Parallel Processing*. 1–11.

[54] Vincent A Traag, Ludo Waltman, and Nees Jan Van Eck. 2019. From Louvain to Leiden: guaranteeing well-connected communities. *Scientific reports* 9, 1 (2019), 5233.

[55] Amanda L Traud, Peter J Mucha, and Mason A Porter. 2012. Social structure of facebook networks. *Physica A: Statistical Mechanics and its Applications* 391, 16 (2012), 4165–4180.

[56] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[57] Nate Veldt. 2022. Correlation clustering via strong triadic closure labeling: Fast approximation algorithms and practical lower bounds. In *International Conference on Machine Learning*. PMLR, 22060–22083.

[58] Chun Wang, Shirui Pan, Ruiqi Hu, Guodong Long, Jing Jiang, and Chengqi Zhang. 2019. Attributed Graph Clustering: A Deep Attentional Embedding Approach. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence* (Macao, China) *(IJCAI'19)*. AAAI Press, 3670–3676.

[59] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Transactions on Parallel Computing* 4, 1 (Aug. 2017), 3:1–3:49. https://doi.org/10.1145/3108140

[60] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities based on Ground-truth. arXiv:1205.6233 [cs.SI]

[61] Ziqiao Zhang, Peng Pu, Dingding Han, and Ming Tang. 2018. Self-adaptive Louvain algorithm: Fast and stable community detection algorithm based on the principle of small probability event. *Physica A: Statistical Mechanics and Its Applications* 506 (2018), 975–986.

A Artifact Appendix

The main repository can be accessed from the GitHub: https://github.com/LinXi-lx/GALA.git, and the artifact can be obtained from the Zenodo link: https://zenodo.org/records/14512723. For more details, please refer to the README in the repository.

A.1 Requirements

A.1.1 Hardware Dependencies

- GPU: NVIDIA A100

A.1.2 Software Dependencies

- G++ 10.4.0
- CUDA Toolkit 11.6
- Make 4.3
- NCCL 2.12
- Openmpi 4.1.4

A.2 Datasets

All the graphs used in the experiments can be directly obtained by running the scripts in the artifact:

```
cd data
bash prepare_graph.sh
```

A.3 Installation and Evaluation

Compile, process the data, and run the experiments using the scripts below. The final results will be displayed in tabular form on the terminal.

```
bash compile_all.sh
bash preprocess_graph.sh
bash runme.sh
```