C-SAW: A Framework for Graph Sampling and Random Walk on GPUs

Santosh Pandey  Lingda Li+  Adolfy Hoisie+  Xiaoye S. Li*  Hang Liu
Stevens Institute of Technology  +Brookhaven National Laboratory  *Lawrence Berkeley National Laboratory

Abstract—Many applications require to learn, mine, analyze and visualize large-scale graphs. These graphs are often too large to be addressed efficiently using conventional graph processing technologies. Fortunately, recent research efforts find out graph sampling and random walk, which significantly reduce the size of original graphs, can benefit the tasks of learning, mining, analyzing and visualizing large graphs by capturing the desirable graph properties. This paper introduces C-SAW, the first framework that accelerates Sampling and Random Walk framework on GPUs. Particularly, C-SAW makes three contributions: First, our framework provides a generic API which allows users to implement a wide range of sampling and random walk algorithms with ease. Second, offloading this framework on GPU, we introduce warp-centric parallel selection, and two novel optimizations for collision migration. Third, towards supporting graphs that exceed the GPU memory capacity, we introduce data transfer optimizations for out-of-memory and multi-GPU sampling, such as workload-aware scheduling and batched multi-instance sampling. Taken together, our framework constantly outperforms the state of the art projects in addition to the capability of supporting a wide range of sampling and random walk algorithms.

I. INTRODUCTION

Graph is a natural format to represent relationships that are prevalent in a wide range of real-world applications, such as, material/drug discovery [1], web-structure [2], social network [3], protein-protein interaction [4], knowledge graphs [5], among many others. Learning, mining, analyzing and visualizing graphs is hence of paramount value to our society. However, as the size of the graph continues to grow, the complexity of handling those graphs also soars. In fact, large-scale graph analytics is deemed as a grand challenge that the complexity further worsens the challenge.

Despite great importance, limited efforts have been made to deploy graph sampling and random walk algorithms on GPUs which come with tempting computing, data access capabilities and ever-thriving community [9]. This paper finds three major challenges that prevent this effort.

First, although there is a variety of platforms to accelerate traditional graph processing algorithms on GPUs [22]–[25], graph sampling and random walk algorithms pose unique challenges. Unlike traditional graph algorithms which often treat various vertices and edges similarly and focus on optimizing the operations on the vertex or edge, sampling and random walk algorithms center around how to select a subset of vertex or edge based upon a bias (Section II-B). Once selected, the vertex is merely visited again. Consequently, how to efficiently select the vertices of interest which is rarely studied by traditional algorithms becomes the core of sampling and random walk. This process needs to construct and potentially update the selection bias repeatedly which is very expensive hence significantly hampers the performance.

Second, it is difficult to arrive at a GPU-based framework for various graph sampling and random walk algorithms that address the needs of vastly different applications. Particularly, there exists a rich body of graph sampling and random walk algorithms (detailed in Section II-A), deriving the common functionalities for a framework and exposing different needs as user programming interface is a daunting task. And offloading this framework on GPU to enjoy the unprecedented computing and bandwidth capability yet hiding the GPU programming complexity further worsens the challenge.

Third, an extremely large graph, which drives the needs of graph sampling and random walk, usually goes beyond the size of GPU memory. While there exists an array of solutions for GPU-based large graph processing, namely, unified memory [26], topology-aware partition [27] and vertex-range based partitions [28], graph sampling and random walk algorithms, which require all the neighbors of a vertex to present in order to compute the selection probability, exhibit stringent requirement on the partitioning methods. In the meantime, the asynchronous and out-of-order nature of graph sampling and random walk provides some unique optimization opportunities for out-of-memory sampling, which are neither shared nor explored by traditional out-of-memory systems.

This work advocates C-SAW, to the best of our knowledge, the first GPU-based framework that addresses all the three aforementioned challenges and supports a wide range of...
sampling and random walk algorithms. Taken together, C-SAW significantly outperforms the state of the art systems that support either part of sampling or random walk algorithms. The contributions of this paper are as follows:

- We propose a generic framework which allows end users to express a large family of sampling and random walk algorithms with ease (Section III).
- We implement efficient GPU sampling with novel techniques. Our techniques parallelize the vertex selection on GPUs, with efficient algorithm and system optimizations for vertex collision migration (Section IV).
- We propose asynchronous designs for sampling and random walk, which optimizes the data transfer efficiency for graphs that exceed the GPU memory capacity. We further scale C-SAW to multiple GPUs (Section V).

The remainder of this paper goes as follows: Section II presents the background. Section III outlines the Application Programming Interface (API) and Sections IV and V optimize C-SAW. Section VI presents the evaluation results. Section VII discusses the related works and Section VIII concludes.

II. BACKGROUND

A. Graph Sampling & Random Walk Variations

This section presents the required background for various graph sampling and random walk algorithms [29]. Graph sampling refers to the random exploration of a graph, which results in a subgraph of the original graph.

One Pass Sampling only goes through the original graph once to extract a sample. Random node and random edge sampling belong to this category [29]. They select a subset of vertices/edges in original graph uniformly and randomly.

Traversal based Sampling often traverses the graph in a Breath-First Search manner to better preserve the properties of original graphs [30]. Traversal based sampling follows sampling without replacement methodology, i.e., it avoids sampling the same vertex more than once.

As shown in Table I, traversal based sampling algorithms are categorized based upon the number of sampled neighbors, called NeighborSize, and the criterion to select neighbors, which is referred to as bias. Snowball sampling [31] initiates the sample using a set of uniformly selected seed vertices. Iteratively, it adds all neighbors of every sampled vertex into the sample, until a required depth is reached. Neighbor sampling [32] samples a constant number of neighbors per vertex. The sampling could be either biased or unbiased. Forest fire sampling [29] can be regarded as a probabilistic version of neighbor sampling, which selects a variable number of neighbors for each vertex based on a burning probability. Unlike neighbor and forest fire sampling, which select neighbors for each vertex independently, layer sampling [9] samples a constant number of neighbors for all vertices present in the frontier in each round. It repeats this process until a certain depth is reached.

Random Walk simulates a stochastic process of traversing the graph to form a path of connected vertices. The length of path is constrained by a user given sampling budget. Random walk can be viewed as a special case of sampling when only one neighbor is sampled at a step with the salient difference lies in that random walk allows repeated appearance of a vertex while sampling does not. Table II summarizes the design space of random walk algorithms.

Similar to traversal based sampling, random walk algorithms use bias to decide the probability of selecting a certain neighbor. For unbiased simple random walk, the bias is uniform for all neighbors, i.e., every neighbor has the same chance to be selected. Deepwalk [17] and metropolis hasting random walk [33] are two examples of unbiased random walk. While Deepwalk samples neighbors uniformly, metropolis hasting random walk decides to either explore the sampled neighbor or choose to stay at the same vertex based upon the degree of source and neighbor vertices.

For a biased random walk, the bias varies across neighbors. Furthermore, depending on how to decide the bias, biased random walks are classified into static random walks and dynamic random walks. For static random walk, the bias is determined by the original graph structure and does not change at runtime. Biased Deepwalk [34] is an example of static random walk which extends the original Deepwalk algorithm. The degree of each neighbor is used as its bias.

Since a simple random walk may get stuck locally, random walk with jump [35], random walk with restart [36] and multi-independent random walk [30] are introduced. Particularly, random walk with jump jumps to a random vertex under a certain probability. Random walk with restart jumps to a predetermined vertex. Multi-independent random walk performs multiple instances of random walk independently.

For dynamic random walks, the bias depends upon the runtime states. Node2vec [19] and multi-dimensional random walk (a.k.a. frontier sampling) [37] belong to this genre. Node2vec is an advanced version of Deepwalk which provides more control to the random walk. The bias of a neighbor depends upon the edge weight and its distance from the vertex explored at preceding step. In multi-dimensional random walk, a pool of seed vertices are selected at the beginning. At each
Fig. 1: Example of graph sampling and vertex selection techniques. (a) A toy graph example to select a neighbor of \( v_8 \) (\( v_5, v_7, v_9, v_{10}, v_{11} \)), assuming the bias of a neighbor is defined as its degree. (b) Inverse Transform Sampling which does a binary search on a 1-D space to select \( v_7 \). (c) Dartboard method that rejects \( 1 \) and accepts \( 2 \) (\( v_7 \)). (d) Alias method that selects \( v_7 \).

Step, multi-dimensional random walk explores one vertex \( v \) from the pool based on their degrees. One random neighbor of \( v \) is added to the pool to replace \( v \). This process repeats until a desired number of vertices are sampled.

Summary. Traversal based sampling and random walk are widely used and share two core similarities: 1) they are based on graph traversal, and 2) they selectively sample vertices based on biases (detailed in Section II-B). Their difference is the number of sampled neighbors, as shown in Table I. In the rest of this paper, we use graph sampling to refer to both traversal based sampling and random walk, unless explicitly specified.

B. Bias based Vertex Selection

This section discusses the key challenge of graph sampling: to select vertices based on user defined biases, i.e., bias based vertex selection. As discussed in Section II-A all sampling algorithms involve the process of picking up a subset of vertices from a candidate pool of vertices. For unbiased graph sampling, the selection is straightforward: one can generate a random integer in the range of 1 to the candidate count and use it to select a vertex. Vertex selection is more challenging biased graph sampling. Given certain biases, we need to calculate the probability of selecting a certain vertex, which is called transition probability. Theorem 1 gives the formula to calculate transition probabilities from biases.

Theorem 1. Let vertices \( v_1, v_2, ..., v_n \) be the \( n \) candidates, and the transition probability of \( v_k \), i.e., \( t_k \), be proportional to the bias \( b_k \). Then, one can formally obtain \( t_k = \frac{b_k}{\sum_{i=1}^{n} b_i} \).

Theorem 1 underscores that bias is the key to calculate transition probability. All popular vertex selection algorithms — inverse transform sampling [38], dartboard [39], and alias method [40], [41] — obey this rule.

The key idea of inverse transform sampling is to generate the cumulative distribution function of the transition probability. Fig. 1(b) shows an example. First, inverse transform sampling computes the prefix sum of biases of candidate vertices, to get an array \( S \), where \( S_n = \sum_{i=1}^{m} b_{k-1} (1 \leq m \leq n + 1) \) and \( n = \) total # of candidate vertices. In Fig. 1(b), \( S = \{0, 3, 9, 11, 13, 15\} \). Then \( S \) is normalized using \( S_{n+1} \), to get array \( F \), where \( F_m = \frac{S_m}{S_{n+1}} (1 \leq m \leq n + 1) \). \( F = \{0, 0.2, 0.6, 0.73, 0.87, 1\} \) in Fig. 1(b). In this way, the transition probability of \( v_k \) can be derived with \( F \), because

\[
F_k = \frac{b_k}{\sum_{i=1}^{n} b_i} = \frac{\sum_{i=1}^{k} b_i - \sum_{i=1}^{k-1} b_i}{S_n} = \frac{S_k - S_{k-1}}{S_n} = F_k - F_{k-1}.
\]

We call the array of \( F \) Cumulative Transition Probability Space (CTPS). To select a neighbor, inverse transform sampling generates a random number \( r \) in the range of \((0, 1)\), and employs a binary search of \( r \) over the CTPS. Assuming \( r = 0.5 \) in Fig. 1(b), it falls between \( F_2 = 0.2 \) and \( F_3 = 0.6 \). As a result, the second candidate \( v_7 \) is selected on the CTPS.

When implemented sequentially, ITS has the computational complexity of \( O(n) \), determined by the prefix sum calculation.

Dartboard [39] uses 2D random numbers to select/reject vertices. As shown in Fig. 1(c), we build a 2D board using the bias of each vertex as a bar, and then throw a dart to the 2D board formed by two random numbers. If it does not hit any bar (e.g., 1), we reject the selection and throw another dart, until a bar is hit (e.g., 2). This method may require many trials before picking up a vertex successfully, especially for scale-free graphs where a few candidates have much larger biases than others. Similar to dartboard, the alias method [41] also uses a 2D board. To avoid rejection, the alias method converts the sparse 2D board into a dense one as shown in Fig. 1(d). It breaks down and distributes large biases across bins on the x axis, with the guarantee that a single bin contains at most two vertices. The drawback of alias method is its high preprocessing cost to break down and distribute biases, which is not suitable for GPUs.

III. C-saw Architecture

A. Motivation

Need for Generic Sampling Framework. After sifting across numerous graph analytical frameworks (detailed in Section VII), we find the need of a new framework for graph sampling, because sampling algorithms pose distinct needs on both the framework design and APIs. For framework design, several sampling algorithms, e.g., layer sampling, require the information beyond a vertex and its neighbors for computing, which postulates hardship for traditional vertex-centric frameworks that limit the view of a user to a vertex and its 1-hop neighbors. When it comes to API design, bias is the essence of sampling and random walk. In comparison, traditional graph frameworks focus upon the operators that alter the
information on an edge or a vertex, e.g., minimum operator in single source shortest path. We also notice recent attempts, e.g., KnightKing [39] and GraphSAINT [42], but they cannot support both sampling and random walk algorithms.

Need for Sampling and Random Walk on GPUs. For sampling, short turnaround time is the key. It is also the root cause of the invention of sampling [11]. The good news is that GPU is a proven vehicle to drive an array of graph algorithms beyond their performance ceiling [22]–[25], [43]–[45], thanks to the unprecedented computing capability and memory bandwidth [46]. When it comes to sampling which are much more random than traditional graph algorithms, GPUs will best CPU at even larger margins because extreme randomness puts the large caches of CPU in vein.

B. C-SAW: A Bias-Centric Sampling Framework

C-SAW offloads sampling and random walk on GPUs with the goal of a simple and expressive API and a high performance framework. Particularly, simple means the end users can program C-SAW without knowing the GPU programming syntax. Expressiveness requires C-SAW to not only support the known sampling algorithms discussed in Section II-A but also prepare to support emerging ones. High performance targets the framework design. That is, the programming simplicity does not prevent C-SAW from exploring major GPU and sampling related optimizations.

C-SAW encompasses two types of user involvements, i.e., parameter and API based options. The parameter-based option only needs a choice from the end users thus is simple, e.g., deciding the number of selected frontier vertices (FrontierSize in line 4 of Fig. 2(b)) and neighbors (NeighborSize in line 6). API based involvement, in contrast, provides more expressiveness to users. Particularly, C-SAW offers three user defined API functions as shown in Fig. 2(a), most of which surround bias, that is, VERTEXBIAS, EDGEBIAS, and UPDATE. We will discuss the design details of these API functions in Section III-C.

Fig. 2: C-SAW framework and API functions.

Fig. 3: Implementing two sampling algorithms with C-SAW API.

Fig. 2(b) gives an overview of the C-SAW algorithm. Particularly, bias based vertex selection occurs in two places: to select frontier vertices from a pool (line 4), and to select the neighbors of frontier vertices (line 6). While the latter case is required by all graph sampling algorithms, the former becomes essential when users want to introduce more randomness, such as multi-dimensional random walk.

In the beginning, the frontier FrontierPool is initialized with a set of seed vertices (line 2). Sampling starts from these seeds until reaching the desired depth (line 3). In each iteration of the while loop, first, VERTEXBIAS is called on the FrontierPool to retrieve the bias for each candidate vertex. SELECT method uses the biases provided by VERTEXBIAS to choose FrontierSize vertices as the current frontier (line 4). Next, all neighbors of the frontier vertices are collected in the NeighborPool using the GATHERNEIGHBORS method (line 5). For these neighbors, we first define their biases using the EDGEBIAS method. Similarly, SELECT method uses the biases to choose NeighborSize neighbors from the NeighborPool (line 6). From the selected neighbors, UPDATE is used to pick new vertices for the FrontierPool (line 7). The selected neighbors are also added to the final sample list Sampled (line 8) before we move forward to the next iteration.

C. C-SAW API

VERTEXBIAS defines the bias associated with a candidate vertex of the FrontierPool. We often use the pertinent property of vertex to derive the bias. Equation (1) formally defines the bias for each vertex v in the FrontierPool. We apply function \( f_{\text{Bias}} \) over the property of \( v \) to define the associated bias.

\[
\text{VERTEXBIAS} \quad v \in \text{FrontierPool} \quad \leftarrow \quad f_{\text{Bias}}(v).
\]

Using multi-dimensional random walk as an example, it uses the vertex degree as a bias for the vertex of interest. EDGEBIAS defines the bias of each neighbor in the NeighborPool. It is named as EDGEBIAS because every neighboring vertex is associated with an edge. While, again, any static or dynamic bias is applicable, a typical bias is induced from the properties of the associated edge. Equation (2) defines EDGEBIAS formally. Let \( v \) be the source vertex of \( u \). Assuming
edge \( e = (v, u) \) carries the essential properties of \( v, u \) and \( e \), we arrive at the following edge bias:

\[
\text{EDGEBIAS} \quad e \in \text{NeighborPool} \quad f_e \text{BIAS}(e) \quad (3)
\]

**UPDATE** decides the vertex that should be added to the FrontierPool based on the sampled neighbors. It can return any vertex to provide maximum flexibility. For instance, this method can be used to filter out vertices that have been visited before for most traversal based sampling algorithms. Whereas for random walk, this method can be used to implement the jump or restart action in the random walk with jump and with start, respectively. Equation (4) quantifies this method, where we will decide whether to add the sampled vertex \( u \), a neighbor of frontier \( v \) from edge \( e \) into FrontierPool based upon the properties of \( e \) and its endpoints.

\[
\text{FrontierPool} \leftarrow \text{UPDATE}(e) \quad (4)
\]

**D. Case Study**

C-SAW can support all graph sampling and random walk algorithms introduced in Section II-A. Fig. 3 exhibits how to use C-SAW to implement two popular algorithms: Node2vec and multi-dimensional random walk.

Without loss of generality, we use the simplest example, i.e., multi-dimensional random walk to illustrate how C-SAW works, as shown in Fig. 3. **FrontierSize** and **NeighborSize** are set as 3 and 1 respectively. **VERTEXBIAS** is based on the degree of vertices in the frontier pool in multi-dimensional random walk. **EDGEBIAS** returns 1, resulting in the same transition probability for every neighbor. **UPDATE** always adds the currently sampled neighbor to the FrontierPool.

**IV. Optimizing GPU Sampling**

Fig. 4 exhibits how to implement two popular algorithms: Node2vec and SAWSA. Fig. 5 illustrates the SELEcT algorithm using inverse transform sampling. We aim to have an efficient GPU implementation of it.

**Inter-warp Parallelism.** Each thread warp, no matter intra or inter thread blocks, is assigned to sample a vertex in **FrontierPool**. To fully saturate GPU resources, thousands of candidate vertices need to be sampled concurrently. There are two sources of them. First of all, many sampling algorithms naturally sample all vertices in **FrontierPool** concurrently. For instance, neighbor sampling allows all vertices in **FrontierPool** to be sampled concurrently and requires a separate **NeighborPool** for each vertex in the **FrontierPool**.

Second, most sampling applications including Graph Convolutional Network (GCN) \cite{kipf2016semi}, Deepwalk, Node2vec, and Personalized PageRank (PPR) \cite{liben2007link}, need to launch many instances of sampling either from the same seeds or different seeds. Here, an **instance** generates one sampled graph from the original graph. Particularly, for all algorithms except multi-dimensional random walk, an instance starts with one source vertex. For multi-dimensional random walk, an instance has multiple source vertices, which collectively generate one sampled graph. Applications like GCN require multiple sample instances for training the model \cite{kipf2016semi}, \cite{kipf2016semi}, \cite{grover2016node2vec}, while Deepwalk, Node2vec, and PPR require multi-source random walk to either generate vertex embeddings or estimate PPR \cite{liben2007link}, \cite{liben2007link}. With thousands of concurrent instances, C-SAW is able to leverage the full computing power of GPU. Since the inter-warp parallelism is straightforward to implement, we focus on exploiting the intra-warp parallelism for C-SAW.

**Intra-warp Parallelism.** A thread warp is used to execute one instance of SELECT on a pool of vertices. An obvious alternative is to use a thread block. Most real world graphs follow power-law degree distribution, i.e., the majority of the vertices in the graph have very few edges. Using a thread block for a neighbor pool will fail to saturate the resource.
Our evaluation shows that using thread warps achieves $\sim 2 \times$ speedup compared with using thread blocks. Thus we choose to use thread warps to exploit the parallelism within SELECT.

As shown in Fig. 5 first, SELECT calculates the prefix sum of the biases of all vertices (line 6). Fortunately, parallel prefix sum is a well-studied area on GPUs. In this paper, we adopt the Kogge-Stone algorithm [50] which presents superior performance for the prefix sum of warp-level where all threads execute in lock-step. The normalization of prefix sums (line 7) can be naturally parallelized by distributing the division of different array elements across threads.

To parallelize the vertex selection loop (line 10-14), C-SAW dedicates one thread for each vertex to maximize the parallelism. For each loop iteration, a random number is generated to select one vertex, as introduced in Section II-B. However, this creates a crucial challenge that different threads may select the same vertex, i.e., selection collision.

B. Migrating Selection Collision

To migrate the aforementioned selection collision, we propose two interesting solutions: bipartite region search, and bitmap based collision detection. Before introducing our new design, we first discuss naive solutions.

Naive Solutions. A naive solution is to have a do-while loop (line 10-14 in Fig. 5) to re-select another one until success, i.e., repeated sampling. However, many iterations may be needed to make a successful selection. As shown in Fig. 6(a), if the region of $v_7$ (i.e., 0.2 - 0.6 in CTPS) is already selected, our newly generated random number 0.58 will not lead to a successful selection. In fact, our evaluation observes that this method suffers for scale-free graphs whose transition probability can be highly skewed, or when a large fraction of the candidates need to be selected i.e larger NeighborSize.

Another solution is to re-calculate the CTPS by excluding the already selected vertices, i.e., updated sampling, such as Fig. 6(b). Then we can always pick unselected vertices by searching through the updated CTPS. Particularly in Fig. 6(b), we will perform another Kogge-Stone prefix-sum for the new bias array $\{3, 2, 2, 2\}$ towards $\{0, 0.3, 0.56, 0.78, 1\}$. Consequently, the CTPS becomes $\{0, 0.33, 0.56, 0.78, 1\}$. Then, the random number $r = 0.58$ selects $v_{10}$. Recalculating prefix sum is, however, time consuming.

Bipartite Region Search inherits the advantages of both repeated and updated sampling, while avoiding their drawbacks.

That is, it does not need the expensive CTPS update compared with updated sampling, while greatly improve the chance of successful selection compared with repeated sampling.

Particularly, while updated sampling updates the CTPS without changing the random number as shown in Fig. 6(b), the key idea of bipartite region search is to adjust the random number $r$ so that the CTPS remains intact and can be reused.

Most importantly, bipartite region search warrants that its random number adjustment leads to the same selections as updated sampling. Note, this method is called bipartite region search because when the random number selects an already selected vertex, bipartite region search searches either the right or the left side of the already selected region in CTPS. Below, we discuss this adjustment.

1. Generate a random number $r'$ ($0 \leq r' < 1$).
2. Use $r'$ to select a vertex in CTPS. If the vertex has not been selected, done. Otherwise, the region that $r'$ falls into corresponds to a pre-selected vertex. Assume the boundary of this region in CTPS is $(l, h)$. Go to 3.
3. Let $\lambda = 1/(1/(h-l))$, $\delta = h-l$ and update $r$ to $r'/\lambda$. If $r < l$, select $(0, l)$ and go to 4. Otherwise select $(h, 1)$ and go to 5.
4. Use the updated $r$ to search in $(0, l)$. If updated $r$ falls in another selected region, go to 6. Otherwise done.
5. Further update $r$ to $r + \delta$ and search in $(h, 1)$. If updated $r$ falls in another selected region, go to 6. Otherwise done.

Fig. 6(c) explains how bipartite region search works for the same example in Fig. 6(b). Assuming we get a random number $r' = 0.58$, it corresponds to $v_7$ in the original CTPS. Since $v_7$ is already selected, bipartite region search will adjust this random number to 0.348 in 3. Since the updated $r = 0.348 > l = 0.2$, bipartite region search selects $(0.6, 1)$ to explore. Consequently in 5, we further add $\delta = 0.4$ to $r$ which leads to $r = 0.748$. 0.748 corresponds to $v_{10}$, and thus results in a successful selection. It is important to note that this selection is identical as updated sampling in Fig. 6(b).

Proof of Bipartite Region Search. We will prove the soundness of bipartite region search mathematically, in the scenario when one and only one vertex has been pre-selected.

Theorem 2. Assuming $v_k$‘s probability region is $(F_k, F_{k+1})$ in the original CTPS. Remind the definition of $F$ in Section II-B.
Let $v_s$ be the pre-selected vertex, and $F'_k$ be the probability in the updated CTPS. $l = F_k$, $h = F_{k+1}$, $\lambda = \frac{1}{l-h}$ and $\delta = h-l$, we prove that:

$$F'_k = \begin{cases} \lambda \cdot F_k; & k < s, \\ \lambda \cdot (F_k - \delta); & \text{otherwise}. \end{cases}$$
Proof. Adopting Equation 1, we get $F_k = \sum_{i=1}^{k-1} b_i$. Denoting $F = \sum_{i=x}^{i=x+y} b_i$, Theorem 1 leads to:

$$F' = \begin{cases} 
\frac{\sum_{i=x}^{i=x+y} b_i}{F} k < s, \\
\frac{\sum_{i=x}^{i=x+y} b_i}{F} + \frac{b_s}{F} k > s, \\
\frac{b_s}{F} k = s
\end{cases}$$

(6)

When $k < s$, $F' = \frac{\sum_{i=x}^{i=x+y} b_i}{F} = \frac{\sum_{i=x}^{i=x+y} b_i}{\sum_{i=x}^{i=x+y} b_i} = \frac{\sum_{i=x}^{i=x+y} b_i}{\sum_{i=x}^{i=x+y} b_i} = \frac{\sum_{i=x}^{i=x+y} b_i}{\sum_{i=x}^{i=x+y} b_i} = F$. When $k > s$, $F' = \frac{\sum_{i=x}^{i=x+y} b_i + \sum_{i=x}^{i=x+y} b_i}{F} = \frac{\sum_{i=x}^{i=x+y} b_i + \sum_{i=x}^{i=x+y} b_i}{\sum_{i=x}^{i=x+y} b_i} = \frac{\sum_{i=x}^{i=x+y} b_i + \sum_{i=x}^{i=x+y} b_i}{\sum_{i=x}^{i=x+y} b_i} = F$.

(7)

Since $\frac{b_s}{F} = k - l = \lambda$, we prove $F'_k = \lambda \cdot F_k$. When $k > s$, $F'_k = \frac{\sum_{i=x}^{i=x+y} b_i + \sum_{i=x}^{i=x+y} b_i}{F} = \frac{\sum_{i=x}^{i=x+y} b_i + \sum_{i=x}^{i=x+y} b_i}{\sum_{i=x}^{i=x+y} b_i} = \frac{\sum_{i=x}^{i=x+y} b_i + \sum_{i=x}^{i=x+y} b_i}{\sum_{i=x}^{i=x+y} b_i} = F$.

(8)

Since $\frac{b_s}{F} = h - l = \delta$, we obtain $F'_k = \lambda \cdot (F_k - F_k)$. □

Theorem 2 states that one can adjust the probabilities from the original CTPS to derive the updated CTPS. Reversing the transformation direction, we further obtain:

$$F_k = \begin{cases} 
F'_k k < s, \\
F'_k + \delta k \geq s
\end{cases}$$

(9)

Since $r'$ is the random number for the updated CTPS, we can substitute $F_k$ with $r'$ in Equation 9 to derive the corresponding $r$ in the original CTPS that falls right at the region boundaries of original CTPS, e.g., $\{0, 0.33, 0.56, 0.78, 1\}$ in Fig. 4(b) fall right at $\{0, 0.2, 0.73, 0.87, 1\}$ in Fig. 4(c). Further, since $F_k$ is a strictly monotonic function of $F'_k$, it is clear that if $r'$ falls between the region boundaries of the updated CTPS, the derived $r$ will also do so in the original CTPS. This ensures bipartite region search will make identical selection as if the CTPS is updated. It is also provable that statistically, the selection probability of our algorithm is the same as the desired transition probability in more complicated scenarios where multiple vertices have been pre-selected.

**Strided Bitmap for Collision Detection.** Bipartite region search requires a collision detection mechanism. We introduce a per vertex bitmap to detect selection collision (line 13 in Fig. 5). For every candidate vertex, there is a unique bit in the bitmap to indicate whether it has been selected. The bitmap is shared by all threads of a warp. After each thread selects a vertex, we perform an atomic compare-and-swap operation to the corresponding bit in the bitmap. If the bit is 0, which means no other threads have picked this vertex, we set it to 1.

Since GPUs do not have variables that support bit-wise atomic operations currently, we may use either 8-bit or 32-bit integer variables for bitmap representation, where each bit corresponds to one vertex. As using 32-bit variables results in more conflicts when updating multiple bits within the same variable, we choose 8-bit variables instead.

To resolve the atomic contentions, we propose to use strided bitmaps, inspired by the set-associative cache organization [51]. A strided bitmap scatters the bits of adjacent vertices across different 8-bit variables, as shown in Fig. 7. Instead of using the first five bits of the same 8-bit variable to indicate the status of all vertices in the contiguous bitmap, the strided bitmap spreads them into two variables to reduce conflicts.

**Data Structures.** C-SAW employs three major data structures: frontier queues, per-warp bitmap, and per-warp CTPS. All these data structures are allocated in the GPU global memory before sampling starts. A frontier queue is a structure of three arrays, VertexID, InstanceID, and CurrDepth to keep track of the sampling process. Till now, all threads share one frontier queue, with a few exceptions that will be introduced in Section V. Per-warp bitmaps and CTPSs are stored as arrays and get reused across the entire sampling process. They are also located in global memory.

**V. OUT-OF-MEMORY & MULTI-GPU C-SAW**

Thanks to sampling and random walk which lift important obstacles for out-of-memory computation, that is, they need neither the entire graph nor synchronization during computation. This section takes advantage of this opportunity to enable fast out-of-memory and multi-GPU C-SAW.

**A. Graph Partition**

C-SAW partitions the graph by simply assigning a contiguous and equal range of vertices and all their neighbor lists to one partition. We adopt this method instead of advanced topology-aware partition (e.g., METIS [27], [28], [52], and 2-D partition [53], for three reasons. First and foremost, sampling and random walk require all the edges of a vertex be present in order to compute the transition probability. Splitting the neighbor list of any vertex, which is the case in 2-D partition, would introduce fine-grained communication between partitions, that largely hampers the performance. Second, topology-aware partition would require extremely long preprocessing time, as well as yield discontinued vertex ranges which often lead to more overhead than benefit. Third, this simple partitioning method allows C-SAW to decide which partition a vertex belongs to in constant time that is important for fast bulk asynchronous sampling (Fig. 8).

**B. Workload-Aware Partition Scheduling**

Since multiple sampling instances are independent of each other, this dimension of flexibility grants C-SAW the freedom of dynamically scheduling various partitions based upon the workload from both graph partitions and workers (such as GPU kernels and devices).
Workload-Aware Partition Scheduling. C-SAW tracks the number of frontier vertices that falls into each partition to determine which partition will offer more workload (1 in Fig. 8). We refer them as active vertices. Based upon the count, we also allocate thread blocks to each GPU kernel with thread block based workload balancing described in next paragraph. Subsequently, the partitions that contain more workload are transferred to the GPU earlier and sampled first (2 in Fig. 8). Non-blocking cudaMemcpyAsync is used to copy partitions to the GPU memory asynchronously. C-SAW samples this partition until it has no active vertices. Note that, C-SAW stores frontier queues from all partitions in the GPU memory. It allows a partition to insert new vertices to its frontier queue, as well as the frontier queues of other partitions to enable communications. The actively sampled partition is only released from the GPU memory when its frontier queue is empty. The reason is that partitions with more active vertices often insert more neighbors in its own frontier queue, which further leads to more workloads. As a result, this design can reduce the number of partitions transferred from CPU to GPU.

When it comes to computation, we dedicate one GPU kernel to one active partition along with a CUDA stream, in order to overlap the data transfer and sampling of different active partitions. After parallel partition sampling finishes, we count the vertex number in each frontier queue to decide which partitions should be transferred to GPU for sampling next (3 in Fig. 8). The entire sampling is complete when there are no active vertices in all partitions.

Thread Block based Workload Balancing. Depending upon the properties of graphs and sample seeds, frontiers are likely not equally distributed across partitions. As a result, the sampling and data transfer time are not the same as well. Since the straggler kernel determines the overall execution time, it is ideal to balance the workload across kernels. Consequently, we implicitly partition the GPU resources by controlling the thread block number of different kernels.

Example. Fig. 8 shows an example of out-of-memory sampling. Here, we assume three graph partitions (i.e., P_1, P_2, P_3) for the same graph in Fig. 1(a), two GPU kernels (i.e., Kernel_1 and Kernel_2), and the GPU memory can contain two active partitions. If we start sampling from vertices {0, 2, 8}, P_1, P_2, and P_3 will have 2, 0, and 1 active vertices initially. Hence, kernel K_1 is assigned to work on P_1 and kernel K_2 for P_3. To balance the workload, the ratio of thread block numbers assigned to K_1 and K_2 is set to 2:1. Assuming vertices 0, 2, and 8 pick 7, 3, and 5, respectively, the frontier queues for P_1, P_2 and P_3 become {3}, {7, 5} and {} respectively. After it finishes selecting vertices (line 6 in Fig. 2(b)), InstanceID is used to find the corresponding frontier pool and sampled graph to update (line 7-8). Note that there may exist multiple copies of the same vertex in the queue, because a common vertex can be sampled by multiple instances.

Batched Multi-Instance Sampling

In the out-of-memory setting, C-SAW introduces batched multi-instance sampling, which concurrently samples multiple instances, to combat the expensive data transferring cost.

Batched sampling is implemented by combining the active vertices of various concurrently sampling instances into a single frontier queue for each partition. Along with the queue, we need to keep two extra metadata for each vertex, i.e., InstanceID and CurrDepth, which tracks the instance that a vertex belongs to and stores the current depth of that instance respectively. During sampling, a thread warp in the kernel can work on any vertex in the queue, no matter whether they are from the same or different instances. After it finishes selecting vertices (line 6 in Fig. 2(b)), InstanceID is used to find the corresponding frontier pool and sampled graph to update (line 7-8). Note that there may exist multiple copies of the same vertex in the queue, because a common vertex can be sampled by multiple instances.

Batched sampling can also balance the workload across sampling instances. Otherwise, if we sample various instances separately, since many real-world graphs hold highly skewed degree distributions, some instances may encounter higher degree vertices more often and thus more workloads. This will end up with skewed workload distributions. Batched sampling solves this problem using a vertex-grained workload distribution, instead of instance-grained distribution.

D. Multi-GPU C-SAW

As the number of sources continues to grow, the workload will saturate one GPU and go beyond. In this context, scaling C-SAW to multiple GPUs would help accelerate the sampling performance. Since various sampling instances are independent from each other, C-SAW simply divides all the sampling
instances into several disjoint groups, each of which contains equal number of instances. Here, the number of disjoint groups is the same as the number of GPUs. Afterwards, each GPU will be responsible for one sampling group. During sampling, each GPU will perform the same tasks as shown in Fig. 8 and no inter-GPU communication is required.

VI. EVALUATIONS

C-SAW is implemented with ~4,000 lines of CUDA code and compiled by CUDA Toolkit 10.1.243 and g++ 7.4.0 with optimization flag as -O3. We evaluate C-SAW on the Summit supercomputer of Oak Ridge National Laboratory [54]. Each Summit node is equipped with 6 NVIDIA Tesla V100 GPUs, dual-socket 22-core POWER9 CPUs and 512 GB main memory. Particularly, each V100 GPU is equipped with 16GB device memory. For the random number generation, we use the cuRAND library [55].

| Dataset            | Abbr. | Vertex Count | Edge Count | Avg. degree | Size (of CSR) |
|--------------------|-------|--------------|------------|-------------|---------------|
| Amazon0601         | AM    | 0.3M         | 3.4M       | 8.39        | 59 MB         |
| Askitter           | AS    | 1.7M         | 11.1M      | 6.34        | 325 MB        |
| cit-Patents        | CP    | 3.6M         | 16.5M      | 4.38        | 293 MB        |
| LiveJournal        | LJ    | 4.8M         | 68.9M      | 14.23       | 1.1 GB        |
| Orkut              | OR    | 3.1M         | 117.2M     | 38.14       | 1.8 GB        |
| Reddit pearl       | RE    | 0.5M         | 11.6M      | 49.82       | 179 MB        |
| Web-Google         | WG    | 0.8M         | 5.1M       | 5.83        | 85 MB         |
| Yelp               | YE    | 0.7M         | 6.9M       | 9.73        | 111 MB        |
| Friendster         | FR    | 65.6M        | 1.8M       | 27.53       | 29 GB         |
| Twitter            | TW    | 41.6M        | 1.5M       | 35.25       | 22 GB         |

TABLE II: Details of evaluated graphs.

Dataset. We use the graph datasets in Table II to study C-SAW. This dataset collection contains a wide range of applications, such as social networks (LJ, OR, FR and TW), forum discussion (RE and YE), online shopping (AM), citation networks (CP), computer routing (AS) and web page (WG).

Metrics. Instead of Traversed Edges Per Second (TEPS) in classical graph analytics [22], [23], we introduce a new metric - Sampled Edges Per Second (SEPS) - to evaluate the performance of sampling and random walk. Formally, SEPS = \# \text{SampledEdges} / \text{Time}. This metric is more suitable than TEPS to evaluate sampling and random walk because these algorithms might use different methods thus traverse a different number of edges but end up with the same number of sampled edges. Similar to previous work [22], [23], the kernel execution time is used to compute SEPS, i.e., the time spent on generating the samples, except for the out-of-memory case that also includes the time for transferring the partitions. Note, each reported result is an average of three runs with different sets of seeds.

Test Setup. Analogous to GraphSAINT [12], we generate 4,000 instances for random walk algorithms and 2000 instances for sampling algorithms. For sampling, both the NeighborSize (i.e., number of neighbors sampled from one frontier) and Depth are 2 for analyzing the performance of C-SAW except forest fire, which uses \(P_f = 0.7\) to derive NeighborSize as in [29]. For biased random walk algorithm, the length of the walk is 2,000. For multi-dimensional random walk, similar to GraphSAINT, we use 2,000 as the FrontierSize for each instance.

A. C-SAW vs. State-of-the-art

First, we compare C-SAW against the state-of-the-art frameworks, KnightKing and GraphSAINT. Our profiling result shows that both GraphSAINT and KnightKing use multiple threads to perform the computation, where the \# threads = \# cores. Since KnightKing only supports random walk variations, we compare C-SAW with KnightKing for biased random walk. GraphSAINT provides both Python and C++ implementations. We choose the C++ implementation [11] which exhibits better performance. The C++ version only supports multi-dimensional random walk which is studied in Fig. 9(b).

As shown in Fig. 9 C-SAW presents superior performance over both projects. On average, C-SAW is 10× and 14.7× faster than KnightKing with 1 GPU and 6 GPUs, respectively. Compared to GraphSAINT, C-SAW is 8.1× and 11.5× faster with 1 GPU and 6 GPUs respectively. Each instance of sampled graphs has 1,703 edges on average. While C-SAW outperforms both projects across all graphs, we generally observe better speedup on graphs with a lower average degree, such as, AM, CP and WG on KnightKing and AM on GraphSAINT. This is rooted from 1) the superior computing capability of GPU over CPU, 2) C-SAW is free of bulk synchronous parallelism (BSP) [59], which allows it to always have adequate computing tasks for sparse graphs, and 3) the unprecedented bandwidth of the V100 GPU over the POWER9 CPU, i.e., 900 GB/s vs. 170 GB/s [54]. This underscores the need of GPU-based sampling and random walk.
from more selection collision. Using bipartite region search, we achieve better speedup by mitigating the collision.

Fig. 11 and 12 further profile the effectiveness of our two optimizations. On average, bipartite region search reduces the average number of iterations to pick a neighbor by 5.0×, 1.5×, 1.8×, and 1.7× for these four applications, respectively. Here, # iterations refers to the trip count of do-while loop in Fig. 5 (line 10-14), which represents the amount of computation used to select a vertex. For analysis, we compare the average number of iterations for all sampled vertices, i.e., $\frac{\sum \text{# iterations of sampled vertices}}{\sum \text{# sampled vertices}}$. We observe more reduction on # iterations for biased neighbor sampling than other algorithms as it has a higher selection collision chance and thus requires more iterations without bipartite region search. With relatively larger neighbor pools, collision is less likely to happen in layer sampling which explains its lower benefits from bipartite region search. Similarly, unbiased neighbor sampling and forest fire sampling incur less collision due to unbiased sampling. Fig. 12 shows the effectiveness of bitmap over the baseline which stores the sampled vertices in the GPU shared memory and performs a linear search to detect collision. The ratio metric in Fig. 12 compares the total number of searches performed by bitmap with that of baseline, i.e., $\frac{\sum \text{# searches in bitmap}}{\sum \text{# searches in baseline}}$. Compared to baseline, bitmap reduces the total searches by 63%, 83%, 71%, and 81% for these four applications, respectively. Despite of the significant search count reduction from bitmap, the overhead of atomic operations refrains us from achieving speedups proportional with the search count reduction.

C. Out-of-memory Optimization

Fig. 13 presents the performance impacts of multi-instance batched sampling (BA), workload-aware scheduling (WS), and thread block based workload balancing (BAL) on both large graphs and small graphs. For the sake of analysis, we pretend small graphs do not fit in GPU memory. For the experimental analysis, we use 4 partitions for each graph and two CUDA
Fig. 13: Performance impacts of out-of-memory optimizations. Here, baseline implementation refers to partition transfer based on active partition without any optimization.

Fig. 14: Standard deviation of kernel time for multi-instance batching and workload-aware balancing in out-of-memory C-SAW (lower is better). Here, baseline represents even distribution of resources.

Four applications. As active vertices increase exponentially with depth during sampling, biased neighbor sampling, forest fire sampling, and unbiased neighbor sampling observe more reduction in kernel time than biased random walk. Workload-aware scheduling reduces the overall partition transfers by 1.2×, 1.3×, 1.2×, and 1.1× on these four applications, respectively. Even with moderate decrease in partition transfers, we still achieve noticeable speedups.

D. Studying NeighborSize and # Instances in C-SAW

Fig. 16 reports the time consumption impacts of various NeighborSize and # instances. Here, we use Depth= 3 and 16k instances in Fig. 16(a) for extensive analysis. For Fig. 16(b), we use NeighborSize = 8. As shown in Fig. 16(a), larger NeighborSize leads to longer sampling time. The average
Fig. 17: Scaling c-SAW from 1 to 6 GPUs with (a) 2,000 and (b) 8,000 instances for biased neighbor sampling.

E. c-SAW Scalability

Fig. 17 scales c-SAW from 1 to 6 GPUs for different number of sampling instances. For 2,000 and 8,000 instances, we achieve 1.8× and 5.2× speedup with 6 GPUs, respectively. The reason is that 2,000 instances fail to saturate 6 GPUs. With 8,000 instances, we observe more workloads that lead to better scalability. We also observe that lower average degree graphs present better scalability because their workloads are better distributed across sampling instances.

VII. RELATED WORKS

Despite there is a surge of frameworks for classical graph algorithms including think like a vertex [59], [60], an edge [61], a graph [62], an IO partition [63], and Domain Specific Languages [64], [65], among many others [24], [66], [67], very few projects target graph sampling and random walk which are the focus of c-SAW. This section discusses the closely related work from the following three aspects.

Programming Interface. KnightKing [39] proposes a walker-centric model to support random walk [33], [37], e.g., Node2vec [19], [68], Deepwalk [17], and PPR [42], [69], [70], and hence fails to accommodate sampling algorithms that are important for graph learning and sparsification [9], [29], [37], [48], [71]–[73]. Similarly for [74], [75] which also only support limited sampling/random walk algorithms.

GraphSAINT [11], [12] explores three graph sampling methods, i.e., random vertex and edge sampling, and random walk based sampler, but fails to arrive at a universal framework. [76] supports deletion based sampling algorithms [77]. But this design is inefficient for large graphs that need to remove most edges. In this work, c-SAW offers a bias-centric framework that can support both sampling and random walk algorithms, and hide the GPU programming complexity from end users.

Transition Probability Optimizations. Existing projects often explore the following optimizations, i.e., probability pre-computation and data structure optimization. Particularly, KnightKing [39] pre-computes the alias table for static transition probability, and resorts to dartboard for the dynamic counterpart which is similar to [11]. Interestingly, kPAR [78] even proposes to pre-compute random walks to expedite the process. Since large graphs cannot afford to index the probabilities of all vertices, [70] only pre-computes for hub vertices and further uses hierarchical alias method, i.e., alias tree for distributed sampling. However, not all sampling and random walk algorithms could have deterministic probabilities that support pre-computation. c-SAW finds inverse transform sampling to be ideal for GPUs, and achieves superior performance over the state-of-the-art even when computing the probability during runtime.

Out-of-memory Processing. GPU unified memory and partition-centric are viable method for out-of-memory graph processing. Since graph sampling is irregular, unified memory is not a suitable option [79], [80]. Besides, partition-centric options [63], [81]–[84] load each graph partition from either secondary storage to memory or CPU memory to GPU for processing. Since prior work deals with classical graph algorithms, they need BSP. In contrast, c-SAW takes advantage of the asynchronous nature of sampling to introduce workload-aware scheduling and batched sampling to reduce the data transfer between GPU and CPU.

VIII. CONCLUSION

This paper introduces c-SAW, a novel, generic, and optimized GPU graph sampling framework that supports a wide range of sampling and random walk algorithms. Particularly, we introduce novel bias-centric framework, bipartite region search and workload aware out-of-GPU and multi-GPU scheduling for c-SAW. Taken together, our evaluation shows that c-SAW bests the state-of-the-art.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their helpful suggestions and feedbacks. This research is supported in part by the National Science Foundation CRII award No. 2000722, the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Contract No. DE-AC02-05CH11231 at Lawrence Berkeley National Laboratory, and Brookhaven National Laboratory, which is operated and managed for the U.S. Department of Energy Office of Science by Brookhaven Science Associates under contract No. DE-SC0012704.
