gSWORD: GPU-accelerated Sampling for Subgraph Counting

Anonymous Author(s)

ABSTRACT

Subgraph counting is a fundamental component for many downstream applications such as graph representation learning, image segmentation, and query optimization. Since obtaining the exact count is often intractable, there have been a plethora of approximation methods on graph sampling techniques. Nonetheless, state-of-the-art sampling methods still require massive samples to produce accurate approximations on large data graphs with query graphs of small sizes. In this work, we propose gSWORD, a general framework that harnesses the massive parallelism of GPUs to accelerate iterative sampling algorithms for subgraph counting. Although the samples are embarrassingly parallel, there are unique challenges to accelerating sampling for subgraph counting due to its irregular computation logic. To this end, we propose two novel optimizations that address the intra-iteration and the inter-iteration irregularity respectively. Our experiments show that the state-of-the-art sampling algorithms deployed on gSWORD are capable of emitting millions of samples per second. To showcase the general design of gSWORD, we further propose and deploy a new sampling algorithm `PartialRefine` that enables fine-grained control over the trade-off between sampling error vs. computational cost.

ACM Reference Format:

Anonymous Author(s). 2022. gSWORD: GPU-accelerated Sampling for Subgraph Counting. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Subgraph counting determines the number of subgraphs in a data graph G isomorphic to a connected query graph q. As one of the fundamental graph processing operations, subgraph counting facilitates many downstream applications like graph kernels for representation learning [32, 37] and probabilistic models for image segmentation [15, 46]. However, the subgraph isomorphism problem, which decides whether there exists a subgraph of G isomorphic to q, is NP-complete [7]. The counting problem is more challenging as millions or billions of instances are typically found even in relatively small graphs. Hence, researchers propose a variety of approximation approaches [6, 18, 21, 22, 26, 29] to estimate the count instead of computing the exact value. Particularly, the methods based on random walks (RW) gain great interest due to their superior performance over other approximation approaches in terms of both accuracy and efficiency [18, 26].

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

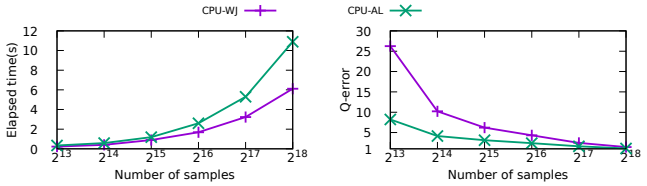


Figure 1: The q-error and runtime by CPU-based `WanderJoin` and `Alley` for a query of 8 vertices in `eu2005` dataset.

RW estimators compute the frequency of q appearing in G by executing a set of samples, each of which independently extracts a subgraph from G. A sample starts from a subgraph G' having a seed vertex, expands G' by adding one vertex at each iteration, and terminates until G' has the same number of vertices as q. The sample quality (i.e., the chance of a sample leading to G' isomorphic to q) has a key impact on the sampling error of the estimation. As such, studies on RW-based techniques [6, 18, 21, 22] focus on optimizing the strategies of expanding G' to improve the quality of samples, which makes RW estimators mainly differ in the methods of sampling the vertices to extend G'. Take two state-of-the-art approaches, `WanderJoin` [21] and `Alley` [18], as examples. At each step, `WanderJoin` samples a vertex from vertices adjacent to G' because the query graph q is connected. To further improve the sample quality, `Alley` maintains a candidate set for sampling and refines the set at each step based on the vertex just added to G'.

To explore the performance factors of `WanderJoin` and `Alley`, we conduct a preliminary experiment on a query with 8 vertices in `eu2005` dataset. Figure 1 presents the estimation error (q-error) and runtime with the number of samples increasing. Although `Alley` converges faster than `WanderJoin`, `Alley` takes a longer time given the same number of samples due to the cost of updating candidate sets. Nevertheless, both of them require massive samples to get an estimation within a reasonable error, which poses significant runtime overhead. For example, both `WanderJoin` and `Alley` exceeds 6 seconds to push the q-error below 1.5, which is equivalent to a 50% relative error to the true value. Consequently, even the approximation approaches to subgraph counting are still computationally expensive to give an accurate estimation, and cannot meet the rigid requirement of many time-critical applications such as dynamic network analysis [10, 11] and query optimization [1, 48].

To address the problem, we propose to design and implement a generic GPU-based RW-sampling framework for subgraph counting because 1) there is no "one-size-fits-all" RW estimator; and 2) GPUs are the ideal accelerator for the problem. RW estimators such as `WanderJoin` and `Alley` trade the computational cost of sampling a vertex for the estimation error, while application users have different tolerance for efficiency and accuracy. Moreover, there is no single winner on all workloads considering the complex structures of query graphs and data graphs. Thus, accelerating one specific RW estimator cannot meet diverse application requirements. GPUs equipped with thousands of cores are far more powerful than CPUs in terms of the raw computational power, and RW estimators are

typical embarrassingly parallel workloads. Each core can process a sample independently. Although there have been some recent studies on accelerating traditional RW workloads on GPUs [16, 25, 39], driven by the demand of graph neural networks, implementing such a framework on GPUs is far from trivial. Specifically, we encounter two challenges given the characteristics of RW estimators for subgraph counting and GPUs' SIMT (single instruction multiple thread) computational paradigm.

- **Inter-Iteration Workload Imbalance**. Traditional RW samples often terminate after performing the same number of iterations, and therefore fit into SIMT seamlessly because threads in a warp (i.e., the basic execution unit in GPUs) must execute the same instruction. In contrast, samples in RW estimators may stop at any iteration since they immediately terminate once the extracted subgraph cannot be further extended to an isomorphic subgraph. Consequently, this phenomenon results in the under utilization of thread resources. One appealing strategy starts a new sample once an existing one stops. However, our experiments find that this method renders worse performance in practice due to poor memory access patterns.
- **Intra-Iteration Workload Imbalance**. Traditional RW samples simply select a vertex from the neighbors of the current residing vertex at each iteration, the computation of which is lightweight. Thus, the workloads of threads in a warp are similar. By comparison, samples in RW estimators require complex operations (e.g., set intersections on neighbor sets of selected vertices) to refine candidate sets at each iteration. As such, threads in a warp can be fed with heavily skewed workloads given the high variance of vertex degrees in real-world graphs. This phenomenon leads to a poor performance especially for threads in the same warp because these threads execute in lockstep and their speed is determined by the slowest one.

In this paper, we present gSWORD, a novel GPU-based framework to accelerate the RW-based sampling process for subgraph counting. Specifically, we design an iterative workflow consisting of *Refine-Sample-Validate* steps. At each iteration of a sample, the *Refine* step first prunes the candidate sets to reduce the sample space size, the subsequent *Sample* step selects a vertex from the refined sets to extend the extracted subgraph G', and the *Validate* step finally decides whether the sample can be stopped given G'. Users can customize RW estimators by implementing these simple APIs and leave the complex parallelization and scheduling of GPUs to our framework. To further improve the performance of RW estimators on gSWORD, we design an efficient sampling method, called `PartialRefine`, on top of the gSWORD workflow. `PartialRefine` enables fine-grained control over the trade-off between sampling error and speed. Notably, `PartialRefine` can capture the design spaces of `WanderJoin` and `Alley`, the state-of-the-art methods.

Motivated by the unique challenges for RW workloads on subgraph counting, gSWORD presents novel optimization techniques for handling workload imbalance. First, we propose a *sample inheritance* strategy for inter-iteration workload imbalance. When a thread terminates a sample, we keep the thread busy by *inheriting* a valid sample from another thread in the same GPU warp. Note that the inherited samples lead to biased estimations if they are treated independently. Hence, we propose an efficient method to adjust the sample weights according to a recursive estimator. Our theoretical analysis not only proves that the recursive estimator is unbiased but also produces smaller variance for faster convergence compared with the scheme without inheritance.

Second, we devise a *warp streaming* strategy for intra-iteration workload imbalance. If a sample contains enough refinement workloads to be parallelized by a warp, we dynamically stream its workloads to the warp where a member thread is assigned to a candidate at a time. Once a candidate passed the refinement step, the thread sends the candidate to a reservoir sampler [18]. In this way, we keep all threads busy while ensuring the vertices are desirably sampled from the refined candidate set after the streaming process.

We thereby summarize our contributions as follows:

- We present gSWORD as a general workflow to accelerate RW estimators for subgraph counting on GPUs. Within gSWORD, we develop an efficient sampling method `PartialRefine` that strikes a better balance between sampling error and speed.
- We devise two novel optimizations: the sample inheritance strategy for inter-iteration workload imbalance and the warp streaming strategy for intra-iteration workload imbalance.
- We conduct extensive experiments and validate the effectiveness of gSWORD. The results reveal that gSWORD significantly improves the efficiency of RW estimators over the CPU and GPU baselines. In particular, gSWORD achieves 93x speedup on average over the CPU baselines and 3.8x speedup for the GPU baselines.

The rest of the paper is organized as follows. Section 2 introduces the background on subgraph counting and GPUs. In Section 3, we propose the APIs and workflow of gSWORD. Section 4 presents the optimization techniques for workload imbalance. The experimental evaluations are discussed in Section 5. Finally, we discuss the related works and conclude the paper in Sections 6 and 7, respectively.

2 PRELIMINARIES

In this section, we first introduce the key notations and definitions for subgraph counting. Then, we review existing RW-based methods as well as the GPU architecture.

2.1 Notations and Problem Definition

Given a graph $g = (\mathcal{V}_g, \mathcal{E}_g, \mathcal{L}_g)$, $\mathcal{V}_g$ represents the vertex set, $\mathcal{E}_g$ represents the edge set, $\mathcal{L}_g$ is a function that maps a vertex $v \in \mathcal{V}_g$ to a label l. q and G denote the query graph and the data graph respectively. We call vertices and edges of q (resp. G) query vertices and query edges (resp. data vertices and data edges), respectively. For ease of presentation, we focus on undirected graphs but our approaches can support directed graphs. A graph g is isomorphic to g' if there exist a *graph isomorphism* (Definition 1) from g to g'.

Definition 1 (Graph Isomorphism). *Given g and g', a graph isomorphism from g to g' is a bijective function $f: \mathcal{V}_g \to \mathcal{V}_{g'}$ s.t.*

- $\forall u \in \mathcal{V}_g, \mathcal{L}_g(u) = \mathcal{L}_{g'}(f(u))$; *and*
- $\forall u, v \in \mathcal{V}_g, e(u, v) \in \mathcal{E}_g \iff e(f(u), f(v)) \in \mathcal{E}_{g'}$.

Problem Definition. The subgraph counting problem finds the number of subgraphs in the data graph G that are isomorphic to the query graph q. For brevity, each isomorphic subgraph in G is called an instance of q.

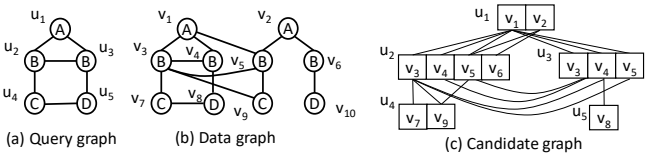


Figure 2: Query graph, data graph and candidate graph.

Following the literature [34], we present some important concepts used in the subgraph counting algorithms.

Definition 2 (Matching Order). *The* matching order φ *is a permutation of query vertices for iterative matching to data vertices.* $\varphi[i]$ *denotes the i-th query vertex to be matched.*

Definition 3 (Partial Instance). *A partial instance in* G *is a prefix match of the first* i *query vertices by* φ *where* $i \leq |\varphi|$. *A partial instance is valid if it can be extended to an instance of* q *in* G.

Definition 4 (Candidate Sets). *A global candidate set* $C(u)$ *of a query vertex* u *is a set of data vertices such that for each* $v \in \mathcal{V}_G$, *if a mapping* (u, v) *exists in an instance of* q *in* G, *then* $v \in C(u)$. *Given a query edge* $e(u, u')$ *and a data vertex* $v \in C(u)$, *the local candidate set* $C(u, u', v)$ *is the set of neighbors of* v *belonging to* $C(u')$, *i.e.,* $C(u, u', v) = N(v) \bigcap C(u')$.

Definition 5 (Candidate Graph). *A candidate graph for* q *on* G *consists of the global candidate set* $C(u)$ *for each* $u \in \mathcal{V}_q$. *Further, there is an edge connecting candidates* $v \in C(u)$ *and* $v' \in C(u')$ *if* $e(u, u') \in \mathcal{E}_q$ *and* $e(v, v') \in \mathcal{E}_G$.

Example 1. *In Figure 2, we present a data graph* G, *a query graph* q *and the corresponding candidate graph. We set the matching order as* $\varphi = (u_1, u_2, u_3, u_4, u_5)$. (v_1, v_3) *is a partial instance for* q *where* v_1 *matches* u_1 *and* v_3 *matches* u_2. *For the same reason,* (v_1, v_4), (v_1, v_5), (v_2, v_5) *and* (v_2, v_6) *are all partial instances. There is only one instance* $(v_1, v_3, v_4, v_7, v_8)$ *of* q *in* G. *Hence,* (v_1, v_3) *is a valid partial instance. In the candidate graph,* $C(u_2) = \{v_3, v_4, v_5, v_6\}$ *is a global candidate set and* $C(u_2, u_4, v_3) = \{v_7, v_9\}$ *is a local candidate set. Note that all instances can be found in the candidate graph but the candidate graph may store vertices and edges that are not included in any instance of* q *in* G, *e.g., vertex* v_2 *and edge* $e(v_2, v_6)$.

2.2 RW Estimators for Subgraph Counting

RW estimators are effective approaches for subgraph counting with theoretical guarantees. For a query graph q, the RW estimators iteratively sample data vertices to match with q. Note that RW estimators are originally designed to sample on the data graph G directly, but it is straightforward to sample on the candidate graph for reducing the sample space. According to the computation flow of RW estimators, *Sample-Validate* (SV) and *Sample-Refine* (SR) are two major RW schemes. Here, we briefly review `WanderJoin` [21] and `Alley` [18] as representative approaches for SV and SR respectively.

WanderJoin. Each sample of `WanderJoin` is obtained by performing RW on the candidate graph. A sample consists of a sequence of data vertices where the first vertex is sampled from a global candidate set, and each subsequent vertex is sampled from a local candidate set based on a vertex that has been previously selected into the sequence. If the sampled sequence is not a valid partial instance by the validity check, then `WanderJoin` starts a new sample sequence. Given a sequence s, the probability of sampling s is $\prod_{i=1}^{|s|} \frac{1}{|C_i|}$ where C_i is the candidate set of the i-th vertex.

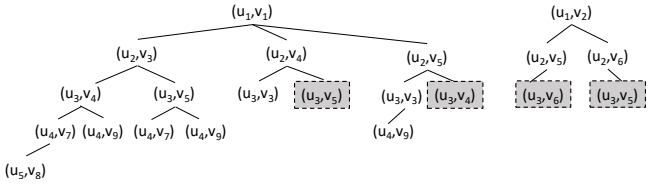


Figure 3: Sample space of RW estimators. The shaded nodes are in the sample space of `WanderJoin` but not `Alley`.

Alley. A major drawback for `WanderJoin` is that many samples can be invalid and thus creates high variance of estimation. To reduce the number of invalid samples, `Alley` imposes a refinement step on the candidate set before selecting the next vertex. The refinement step guarantees that each vertex in the *refined* candidate set always yields a valid partial instance. When a sample sequence encounters an empty refined candidate set, `Alley` starts a new sample sequence. Given a sequence s, the probability of sampling s is $\prod_{i=1}^{|s|} \frac{1}{|C_i|}$ where C_i is the refined candidate set of the i-th vertex.

HT estimator. With the sample probability, both `WanderJoin` and `Alley` employs the Horvitz-Thompson (HT) estimator to approximate the subgraph count.

Definition 6. *Let* Y_i, $i = 1, 2, ..., n$ *be an independent sample and* π_i *is the inclusion probability of sampling* Y_i. *The Horvitz-Thompson estimator of the mean is given by* $\frac{\sum_{i=1}^{n} Y_i/\pi_i}{n}$.

In the context of `WanderJoin` and `Alley`, π_i is the probability of a sample s_i, and Y_i denotes an indicator random variable $\mathbb{I}(s_i)$ where s_i is an instance of q in G. Hence, the estimator for `WanderJoin` and `Alley` is expressed as:

$$H = \frac{\sum_{i=1}^{n} \left(\left(\prod_{j=1}^{d} |C_{ij}| \right) \mathbb{I}(s_i) \right)}{n} \tag{1}$$

where C_{ij} denote the candidate set (possibly refined) for query vertex u_j in sample s_i. If s_i finds a match, then the indicator is 1. Otherwise, it is 0.

Example 2. *Figure 3 shows the sample space for* `WanderJoin` *and* `Alley` *according to the matching order* $\varphi = (u_1, u_2, u_3, u_4, u_5)$ *on the candidate graph in Figure 2.* `WanderJoin` *may sample a sequence* $s_1 = (v_1, v_4, v_3)$ *with a probability* $\frac{1}{2} \cdot \frac{1}{3} \cdot \frac{1}{2} = \frac{1}{12}$. *Although* `Alley` *may sample the same sequence* s_1, *but with a higher probability* $\frac{1}{2} \cdot \frac{1}{3} \cdot 1 = \frac{1}{6}$. *This is because when* $s_1 = (v_1, v_4)$, `Alley` *refines the candidate set for* u_3 *by removing the candidate* v_5 *since* v_5 *is not a common neighbor of* v_1 *and* v_4. *Hence, the sample space of* `Alley` *is smaller compared with that of* `WanderJoin`. *Both* `WanderJoin` *and* `Alley` *have the same probability to sample the only instance of* q *as* $s_2 = (v_1, v_3, v_4, v_7, v_8)$, *i.e.,* $\frac{1}{2} \cdot \frac{1}{3} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot 1 = \frac{1}{24}$. *With* s_1 *(invalid sample) and* s_2 *(valid sample), both* `WanderJoin` *and* `Alley` *yield the same estimate as* $\frac{0+24}{2} = 12$.

Table 1: Sample APIs in gSWORD.

APIs/Parameters	Descriptions
int `Refine`(Sample* *s*, int *d*, Vertex* *cand*, int *clen*, Vertex* *refine*)	Given a sample *s* with sample iteration *d*, a candidate array *cand* with length *clen*, it fills a refined candidate array *refine* and returns the length of *refine*.
pair<Vertex, double> `Sample`(Sample* *s*, int *d*, Vertex* *refine*, int *rlen*)	Given a sample *s* with sample iteration *d*, a refined candidate array *refine* with length *rlen*, it samples a vertex *v* from *refine*. *v* is returned with its sampling probability/weight.
bool `Validate`(Sample* *s*, int *d*, Vertex *v*, double *prob*)	Given a sample *s* with sample iteration *d*, if *s* remains valid after adding *v* as the *d*th vertex, the function returns *true* and updates the sampling probability of *s* given *prob*, the probability of sampling *v*. Otherwise, it returns *false* to indict invalid sample.

2.3 GPU Architecture

The performance benefits of GPUs in accelerating RW estimators are attributed to: (a) massive number of cores; (b) ultra-high memory bandwidth. In this paper, we choose NVIDIA GPU architectures and CUDA programming paradigm due to their wide popularity. A CUDA function that executes on GPUs is called the *kernel* function, which is organized by a number of grouped threads called *thread blocks*. All threads in one block are executing on one streaming multiprocessor (SM). Within an SM, the minimum granularity of instruction scheduling is a warp composed of 32 threads, i.e., in each clock cycle, the scheduler can issue an eligible warp (not be stalled) to compute units for executing the next instruction. If the threads of the same warp enter different condition branches, the scheduler is only able to issue one instruction out of multiple instruction pointers (IP) in each cycle. Hence, the threads that do not belong to the selected IP will be marked as inactive, and the corresponding hardware resources will be idle in this cycle. Such phenomenon is called warp divergence, which significantly affects GPUs' to reach the peak performance [19]. Meanwhile, the two-level cache hierarchy L1 (per SM) - L2 (device) is implemented on GPUs. The corresponding cache line size is designed to be larger than the CPU counterpart, e.g., 128 bytes. Consequently, uncoalesced memory accesses result in high memory access latency, and cause the effective memory bandwidth to be far lower than the theoretical maximum.

3 GSOWRD FRAMEWORK

3.1 Framework Design

gSWORD is a generic GPU sampling framework for RW estimators on subgraph counting. All RW samples are independent and the estimated value of each sample is aggregated using the HT estimator. For each iteration of a RW sample, a new vertex is sampled. To ease the development of RW estimators on GPUs, we abstract a sampling iteration into the *Refine-Sample-Validate* (RSV) steps.

- `Refine`. A refined set is computed based on a local candidate set for sampling the next vertex. The estimation wrt. a sample is more accurate when the refined set is smaller.
- `Sample`. One of the vertex in the refined set is sampled as a new vertex for the sample.
- `Validate`. We check if the sample remains a valid partial/full instance after the new vertex is added. The sample is terminated when an invalid instance is found.

APIs and Candidate Graph Format. Without any domain knowledge of GPUs, users can easily customize their RW estimators by implementing the corresponding APIs of the three steps in Table 1. Further, users need frequent accesses to the candidate graph for the API implementation. Hence, we design a general yet efficient format to store the candidate graph *cg* as depicted in Figure 4. All query edges $e(u, u')$ are stored in a compressed sparse row (CSR) format, i.e., the first two arrays in *cg* are the offset list and the edge list of the query graph CSR respectively. For each edge $e(u, u')$, the candidates for u and the corresponding candidates for u' are stored using a second and a third CSRs. Our design enables efficient lookup for global and local candidate sets. Given a query edge $e(u, u')$ and a data vertex $v \in C(u)$, we use the first CSR to locate the edge. Then, we search the second CSR for v as a global candidate for u, and the local candidate set $C(u, u', v)$ is found in the third CSR.

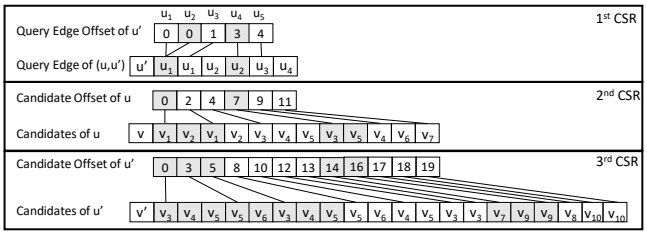


Figure 4: Efficient candidate graph format.

Example 3. *In Figure 4, the query edge $e(u_1, u_2)$ found in the first CSR corresponds with two tuples (u_1, u_2, v_1) and (u_1, u_2, v_2) where $v_1, v_2 \in C(u_1)$ are stored in the second CSR. The local candidate set $C(u_1, u_2, v_1) = \{v_3, v_4, v_5\}$ can be retrieved from the third CSR.*

Next, we present the gSWORD implementations of `WanderJoin` and `Alley` as representatives of RW estimators in Figure 5. The general framework of gSWORD allows users to develop and test new RW estimators with ease. For example, users can design advanced refine strategies to further reduce the sample space. Moreover, instead of uniform sampling like `WanderJoin` and `Alley`, employing importance sampling may yield faster convergence [36].

PartialRefine. To demonstrate the generic design of gSWORD, we propose and implement a new sampling method `PartialRefine` using the APIs. `PartialRefine` is motivated by our observation that `WanderJoin` emits samples faster whereas `Alley` shows smaller error given the same number of samples compared with `WanderJoin`. This is because `WanderJoin` samples vertices directly from the local candidate sets without proper refinement like `Alley`. Hence, `WanderJoin` incurs a large sample space and thus slower convergence. However, the refinement step in `Alley` could be costly as it scans all local candidates for refinement and each refinement step requires intersecting multiple local candidate sets. It is possible that most candidates sent to the refinement step of `Alley` are not filtered and remain eligible for the sample step. In such

```
/*----------WanderJoin----------*/
__device__ int Refine (...){
  /*pass cand array to refine array*/
  refine = cand;
  return clen;
}
__device__ pair<Vertex,double>Sample (...)
{
  /*random select one vertex in refine*/
  Vertex v = RandomSelect ( refine, rlen);
  return MakePair (v , 1/rlen);
}
__device__ bool Validate (...){
  /*check duplicates*/
  if(DupCheck(s,v)){
    Vertex u = getQueryVertex(cg, d);
    /*Get edges from cg*/
    for (Vertex u' : getEdges(cg,u)){
      int k = GetOrderIndex(cg,u');
      Vertex v' = s.ins[k];
      /*check if (v',v) exists*/
      if(!IsEdge(v', v ,cg)){
        return false;
      }}
    s.prob = s.prob* prob;
    s.ins[d] = v;
    return true;}}

/*----------Alley----------*/
__device__ int Refine (...){
  int rlen = 0;
  Vertex u = getQueryVertex(cg, d);
  for (int i = 0; i < clen; ++i) {
    Vertex v = cand[i]
    /*A flag for refinement success */
    bool flag = true;
    /*Get edges from cg*/
    for (Vertex u' : getEdges(cg,u)){
      int k = GetOrderIndex(cg,u');
      Vertex v' = s.ins[k];
      /*Get local candidate C(u', u, v') */
      Vertex* lc = getCand(u', u, v',cg)
      if(!find(v, lc)){
        \*v is not in C(u', u, v')*\
        flag = false;
        break;
      }}
    if(flag == true) {
      /*save refined vertices*/
      refine[rlen] = v
      rlen = rlen + 1;
    }}
  return rlen;
}
__device__ pair<Vertex,double> Sample (...){
  /*random select one vertex in refine*/
  Vertex v = RandomSelect (refine, rlen);
  return MakePair (v , 1/rlen);
}
__device__ bool Validate (...){
  if (DupCheck(s,v)){
    s.prob = s.prob* prob;
    s.ins[d] = v;
    return true;}
  return false;}

/*----------PartialRefine----------*/
__device__ int Refine (...){
  int rlen = 0;
  Vertex u = getQueryVertex(cg, r);
  for (int i = 0; i < clen; ++i) {
    Vertex v = cand[i];
    double ra = rand(0,1);
    bool flag = true;
    /*set refinement probability to 0.1*/
    if (ra < 0.1){
      for (Vertex u' : getEdges(cg,u)){
        int k = GetOrderIndex(cg,u');
        Vertex v' = s.ins[k];
        Vertex* lc = getCand(u', u, v',cg)
        if(!find(v, lc)){
          \*v is not in C(u', u, v')*\
          flag = false;
          break;
        }}
    if(flag == true) {
      /*save refined vertices*/
      refine[rlen] = v
      rlen = rlen + 1;
    }}}
  return rlen;
}
__device__ pair<Vertex,double> Sample(...){
  Vertex v = RandomSelect (refine, rlen);
  return MakePair (v , 1/rlen);
}
__device__ bool Validate (...){
  if (DupCheck(s)){
    /*adjust the probability */
    s.prob = s.prob* prob*0.1;
    s.ins[d] = v;
    return true; }
  return false;}
```

Figure 5: User-defined APIs in gSWORD.

scenario, WanderJoin has the similar convergence rate as Alley but WanderJoin does not incur the expensive refinement cost. *For different data graphs and queries input, the best sampling method is dependent on the input characteristics.*

Motivated by the above discussion, we propose PartialRefine to enable fine-grained control over the refinement cost vs. the sampling error. Instead of a full refinement like Alley, we introduce a refinement factor $\alpha \in (0, 1]$ where a vertex in the candidate set is sent to refinement with probability α. Given a sample s, the probability of sampling s is $\prod_{i=1}^{|s|} \frac{\alpha}{|\hat{C}_i|}$ where $\hat{C}_i$ is the refined candidate set of the i-th vertex after applying partial refinement. It is straightforward to see that PartialRefine is also an unbiased HT estimator for subgraph counting. The detailed implementation of PartialRefine is presented in Figure 5.

Note that when $\alpha = 1$, PartialRefine is identical to Alley. A smaller α results in a smaller number of vertices to be refined, and PartialRefine trends towards WanderJoin with faster sample emission due to lower refinement cost. Nevertheless, the efficiency improvement is traded off for a larger sampling variance according to the following theorem.

Theorem 1. *Let H and H_α denote the HT estimators by* Alley *and* PartialRefine *with the refinement factor $\alpha \in (0, 1]$ respectively.* $\mathrm{Var}[H] \le \mathrm{Var}[H_\alpha]$ *with the equality holds when $\alpha = 1$.*

Proof. We have $\mathrm{Var}[H_\alpha] - \mathrm{Var}[H] = \mathbb{E}[H_\alpha^2] - \mathbb{E}[H^2]$ since both estimators are unbiased. Let C_i and $\hat{C}_i$ be the refined candidate set for sampling the i-th vertex under H and H_α respectively. Note that given a RW sample s, C_i is a deterministic set whereas $\hat{C}_i$ is a randomized set. We have the following inequalities where the expectation is taken over the randomness of s and $\hat{C}_i$:

$$\mathbb{E}[H_\alpha^2] = \mathbb{E}\left[\left(\prod_{i=1}^{|s|} \frac{|\hat{C}_i|}{\alpha} \cdot \mathbb{I}(s)\right)^2\right] = \mathbb{E}\left[\prod_{i=1}^{|s|} \frac{|\hat{C}_i|^2}{\alpha^2} \cdot \mathbb{I}(s)\right]$$

$$\ge \mathop{\mathbb{E}}_{s}\left[\prod_{i=1}^{|s|} \mathop{\mathbb{E}}_{\hat{C}_i}\left[\frac{|\hat{C}_i|^2}{\alpha^2} \,\middle|\, s\right] \cdot \mathbb{I}(s)\right] \ge \mathop{\mathbb{E}}_{s}\left[\prod_{i=1}^{|s|} \frac{(\alpha \cdot |C_i|)^2}{\alpha^2} \cdot \mathbb{I}(s)\right] = \mathbb{E}[H^2]$$

The first inequality is based on Jensus inequality [20] and the second inequality is due to the second moment of a binomial random variable since the partial refinement can be understood by selecting refined candidates C_i with a fixed probability α. □

Tuning α. Theorem 1 not only shows that PartialRefine has a larger variance compared with Alley but also reveals that the variance gap becomes smaller for a larger α. Users are enabled with fine-grained control by tuning α based on different data characteristics to obtain the sweet spot between sampling error and speed. In our experiments, we provide a simple heuristic for users to tune α based on the sample success rate and running time.

3.2 Framework Implementation

Algorithm 1 presents an overview of gSWORD framework implementation. We adopt an intuitive thread-centric approach where each thread is responsible for producing a number of samples. Lines 1-3 initialize the data structures for each thread. Since each sample may incur varying computational cost, distributing a fixed number of samples to each thread results in workload-imbalance. Thus, we devise a sample pool *bp* for each GPU block where all threads share the workload within the same block. Lines 4-5 of Algorithm 1 depict the block sharing approach where a new sample task is fetched from *bp* as long as there remains some samples to be computed in the block. The fetch operation is efficiently handled by *atomic* instructions in the shared memory. After obtain a sample task s, we iteratively sample vertices into s by invoking the user-defined APIs (Lines 7-13). For each sample iteration, we retrieve the candidate set *cand* with its length *clen* from the candidate graph *cg* and send *cand* for refinement. Then, we sample a vertex v from the refined set *refine* as the d-th vertex of the instance in s. If s remains to be a valid partial instance after adding v, we continue to the next sample iteration. Otherwise, the inner loop terminates and a new sample task is fetched from *bp*. Once an instance of $|\mathcal{V}_q|$ vertices are obtained, we update the HT estimator value with the sampling probability of s in Lines 14-15. Finally, we return the HT estimator

Algorithm 1: Thread-Centric Implementation in gSWORD.

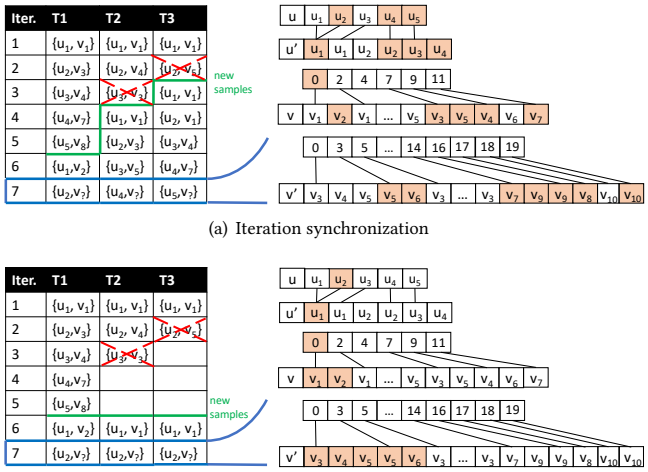
Input : Candidate graph cg, block sample pool bp
Output: Thread HT estimation H, sample size $nSample$

```
refine ← Init();
nSample ← 0;
H ← 0;
while bp ≠ ∅ do
    s ← FetchSampleTask(bp);
    d ← 1;
    while d ≤ |V_q| do
        (cand,clen) ← GetCandidate(cg,s,d);
        rlen ← Refine(s,d,cand,clen,refine);
        (v,prob) ← Sample(s,d,refine,rlen);
        valid ← Validate(s,d,v,prob);
        break if valid == false;
        d ← d + 1;
    if d == |V_q| + 1 then
        H ← H + 1/s.prob;
    nSample ← nSample+1;
return (H, nSample);
```

value and the sample size within a thread. We omit the estimate aggregation among all threads since it can be directly implemented by the efficient parallel reduce for GPUs [5].

Thread-Centric vs. Warp-Centric. gSWORD adopts the thread-centric approach where each sample is managed by one thread. Since the thread-centric approach allows threads in a warp to process different samples at the same time, *warp divergence* could raise if samples terminate at different iterations. A warp-centric approach exploits the GPU architecture by managing a sample with a warp of 32 threads. In this way, warp divergence is mitigated as the threads in the warp execute the same control path of processing a sample. However, there are two major issues for adopting the warp-centric approach on accelerating RW estimators: *limited parallelism* and *programmability*. First, the workload within a sample is sequential in nature, and allocating a warp for the sample results in limited parallelism within the warp. Second, a warp-centric approach requires additional efforts from users to define the parallel version of the APIs. Such complexity reduces the programmability of gSWORD, especially for users who are not familiar with parallel models or GPU optimizations. In contrast, the thread-centric approach substantially reduces the user efforts to "think parallel" by only defining the sequential logic of the sampling process. Thus, we choose the thread-centric approach over the warp-centric alternative.

Sample Synchronization vs. Iteration Synchronization. For sample synchronization, threads in a warp are synchronized *implicitly* once they completed a sample before each thread fetches a new sample task. To reduce warp divergence, iteration synchronization is another alternative approach where threads synchronize after each vertex is sampled (one iteration of the inner while loop in Algorithm 1). The benefit of the alternative is: without waiting for other threads in the warp to complete their current samples, a thread can start a new sample at any iteration once the current sample is invalid. To enable iteration synchronization on GPUs, we can place an *explicit* barrier after each iteration and start a new sample immediately once a current sample is invalid. Surprisingly, iteration synchronization is even slower than sample synchronization with an average run time slowdown of 1.3 times.



(a) Iteration synchronization

(b) Sample synchronization

Figure 6: Access patterns of iteration and sample synchronization, the table shows data vertex sampled at each step.

Our investigation reveals that the *irregular* memory access pattern of the iteration synchronization leads to these results. Specifically, for iteration synchronization, threads in a warp can process different query vertices at an iteration, which leads to the access of candidate sets of different query vertices. By comparison, for sample synchronization, threads in a warp always process the same query vertex at an iteration. Therefore, these threads access the same candidate set, which has much better memory locality. Although iteration synchronization solves the warp divergence issue, the benefit is overwhelmed by the expensive memory access cost. Example 4 gives an example. Our experiment results (see Figure 11 in Section 5) confirm the analysis. Thus, we implement the sample synchronization approach in gSWORD.

Example 4. *Figure 6 illustrates the process of sample synchronization and iteration synchronization with* Alley *as the RW estimator. For brevity, a warp has three threads T1, T2 and T3 in the example. Each thread executes a sample on graphs in Figure 2. The table lists query vertices processed at each iteration. Samples in T2 and T3 terminate at iteration 3 and 2, respectively, since their instances are invalid. The method with iteration synchronization immediately starts a new sample, whereas the sample synchronization method starts a new sample for each thread until all threads complete the samples assigned initially.*

Candidates highlighted on the CSRs are accessed by threads at iteration 7. Iteration synchronization accesses candidates scattered across the candidate graph because threads process different query vertices. In contrast, sample synchronization has much better spatial locality because all threads process the same query vertex. As a result, sample synchronization outperforms the iteration synchronization alternative despite the wasted thread resources.

4 OPTIMIZATIONS

Since gSWORD adopts the thread-centric approach to manage the samples, the straightforward implementation that treats all threads independent leads to severe workload imbalance, especially for threads running lock steps in the same warp. There are two major imbalance sources: *inter-iteration* imbalance and *intra-iteration* imbalance. Within an iteration, `Refine` and `Sample` workloads across different samples can be drastically different, which cause *intra-iteration* imbalance. Further, some samples are terminated after `Validate`. The threads that manage the terminated samples are idle whereas the remaining threads move to the next sample iteration, i.e., *inter-iteration* imbalance.

In this section, we first propose a sample inheritance approach for inter-iteration imbalance. Subsequently, we devise a warp streaming approach for intra-iteration imbalance.

4.1 Sample Inheritance

According to our discussion in Section 3.2, iteration synchronization can alleviate the workload imbalance issue in sample synchronization but its poor data access pattern leads to overwhelming memory cost. Thus, we need an approach that keeps threads busy while maintains cohesive data access pattern in sample synchronization.

We thus propose a sample inheritance approach to address the inter-iteration imbalance. Once a thread discovers an invalid sample, the thread will immediately "inherit" a valid sample from one of its neighborhood threads in the same warp. In this way, all threads in the same warp are always working on the same iteration and thus render cohesive data access pattern. Despite additional workload assigned, sample inheritance (gSWORD) only incurs marginal memory access overhead while processes additional samples.

Iter.	T1	T2	T3
1	$\{v_1\}$	$\{v_1\}$	$\{v_2\}$
2	$\{v_1, v_3\}$	$\{v_1, v_4\}$	$\{v_2, v_5\}$
3	$\{v_1, v_3, v_5\}$	$\{v_1, v_4, v_3\}$	
4	$\{v_1, v_3, v_5, v_9\}$		
5			

Iter.	T1	T2	T3
1	$\{u_1, v_1\}$	$\{u_1, v_1\}$	$\{u_1, v_2\}$
2	$\{u_2, v_3\}$	$\{u_2, v_4\}$	~~$\{u_2, v_2\}$~~
3	$\{u_3, v_5\}$	~~$\{u_3, v_4\}$~~	
4	~~$\{u_4, v_9\}$~~		
5			

(a) Without inheritance

Iter.	T1	T2	T3
1	$\{v_1\}$	$\{v_1\}$	$\{v_2\}$
2	$\{v_1, v_3\}$	$\{v_1, v_4\}$	$\{v_2, v_5\}$
3	$\{v_1, v_3, v_5\}$	$\{v_1, v_4, v_3\}$	$\{v_1, v_3, v_4\}$
4	$\{v_1, v_3, v_5, v_9\}$	$\{v_1, v_3, v_5, v_7\}$	$\{v_1, v_3, v_4, v_7\}$
5	$\{v_1, v_3, v_4, v_7, v_8\}$	$\{v_1, v_3, v_4, v_7, v_8\}$	$\{v_1, v_3, v_4, v_7, v_8\}$

Iter.	T1	T2	T3
1	$\{u_1, v_1\}$	$\{u_1, v_1\}$	$\{u_1, v_2\}$
2	$\{u_2, v_3\}$	$\{u_2, v_4\}$	~~$\{u_2, v_2\}$~~
3	$\{u_3, v_5\}$	~~$\{u_3, v_4\}$~~	$\{u_3, v_4\}$
4	~~$\{u_4, v_9\}$~~	~~$\{u_4, v_7\}$~~	$\{u_4, v_7\}$
5	$\{u_5, v_8\}$	$\{u_5, v_8\}$	$\{u_5, v_8\}$

(b) With inheritance

Figure 7: Example of sample inheritance.

Example 5. *Figure 7 explains the sample inheritance approach. Following Example 4, Threads T1, T2, T3 belong to the same warp, and each starts a new sample independently. The partial instance (v_2, v_5) from T3 is invalid at iteration 2. T3 randomly selects a thread in the warp to inherit the search path. Here, T3 inherits the partial instance (v_1, v_4) from T1. The instance (v_1, v_4, v_3) from T2 is invalid since v_4 maps to u_2 but it does not connect to a data vertex with label C. Thus, T2 inherits the instance (v_1, v_3, v_5) from T1. At iteration 4, the instances from T1 and T2 become invalid so both threads inherit the same instance held by T3, i.e., (v_1, v_3, v_5, v_7). At the last iteration, all threads sample the last vertex for their respective instances.*

Note that all threads are occupied with sampling tasks when inheritance is enabled. The data access pattern is cohesive as the threads always process the same query vertex on the same iteration.

Algorithm 2: Inter-Iteration Inheritance

```
// Replace Line 12 in Algorithm 1
1 if _any(valid) then
2     parentId ← _ballot(valid);
3     idleThreads ← _reduce_sum(valid==false);
4     if threadId == parentId then
5         s.prob = s.prob/(idleThreads+1);
6     if valid==false then
7         s ← _shfl(s,parentId);
8 else
9     break;
```

Although we keep the threads busy with the inheritance optimization, we cannot treat the inherited samples as independent samples. Otherwise the samples are correlated and biased. This is because more samples will be generated towards larger search tree branches as the partial instances have higher chance to be valid.

Example 6. *In Figure 7, there are three initial samples managed by each thread. Further, there are additional four inherited samples (indicted by the arrows). If we treat all samples independent, the estimate is $\frac{1}{7} \cdot (3 \cdot 24 + 4 \cdot 0) = 10.3$, which significantly deviates from the ground-truth value of 1.*

To eliminate the bias, we observe that once a partial instance is inherited, the inherited samples can be used to collectively estimate the search space given the partial instance.

Example 7. *At step 4 in Figure 7, T1 and T2 inherit the partial instance $s = (v1, v3, v5, v7)$ from T3. At step 5, all threads generate samples from s. Conditioned on s, we obtain an estimate of $\frac{1}{3} \cdot 3 = 1$ with the three inherited samples (including the sample held by T3).*

In the previous example, if sampling the partial instance s is unbiased, we can obtain an unbiased estimation for the subgraph count. Based on this observation, we present our inter-iteration inheritance optimization in Algorithm 2. Instead of terminating an invalid sample immediately in Algorithm 1, a thread participates in a voting process to see if there are still valid samples to be inherited in its warp. We take advantages of *warp-level primitives* to enable efficient collaboration via registers across the threads (functions with underline prefix). Note that we omit the thread mask input of the warp primitives for the ease of presentation.

If there exists any valid sample in the warp indicated by the `_any` primitive, a thread holding a valid sample s is selected as the parent using `_ballot`. In Line 3, We count the number of idle threads that run into invalid samples. Since there are in total $idleThreads + 1$ threads which will inherit s from the parent (including the parent itself), we update the probability of s before the parent shares its sample with all idle threads using the efficient `_shfl` primitive.

After the inheritance process, all threads hold a valid sample and can proceed with the next sample iteration as in Algorithm 1.

Recursive Estimator. Please note that Algorithm 2 does *not* treat each inherited samples independent although it appears to be. Instead, it is a *push-down* evaluation of a recursive estimator. Let $R_i(s)$ denote the estimator at level i of the search tree given a partial instance s. $R_i(s)$ is recursively estimated as follows:

$$R_i(s) = \begin{cases} \sum_t \frac{R_{i-1}(s \cup \{v_t\})}{n_i}, & \text{if } i > 1 \text{ and s is inherited to t} \\ R_{i-1}(s \cup \{v\}), & \text{if } i > 1 \text{ and s is not inherited} \\ \frac{\mathbb{I}(s)}{\mathbb{P}(s)}, & \text{if } i = 1, \text{aka. the leaf level} \end{cases}$$

where n_i denote the number of inherited samples at level i.

A thread initially holds an empty sample s at the root level, i.e., $R_{|\mathcal{V}_q|}(s = \emptyset)$. For each iteration, the thread moves to a lower level on the search tree until it reaches the leaf level. The normalization factors n_i are all pushed down via the sample probability $\mathbb{P}(s)$ to the leaf level. The push down evaluation enables efficient one-pass estimation without backward propagation of the values from the leaf to the root.

Theoretical Analysis. We prove that R is an unbiased estimator as the HT estimator H.

Theorem 2. *$\mathbb{E}[R_i(s)] = \mathbb{E}[H_i(s)]$ for any given partial instance s and $i \in [1, |\mathcal{V}_q|]$.*

Proof. We use induction to prove the theorem. For the base case $i = 1$, we have $\mathbb{E}[R_1(s)] = \mathbb{E}[\frac{\mathbb{I}(s)}{\mathbb{P}(s)}] = \mathbb{E}[H_1(s)]$. Assume the equality holds for $i = d$. There are two cases in the inductive step for $i = d + 1$: (1) s is not inherited; (2) s is inherited. For case (1), $\mathbb{E}[R_{d+1}(s)] = \mathbb{E}[R_d(s \cup \{v\})] = \mathbb{E}[H_d(s \cup \{v\})] = \mathbb{E}[H_{d+1}(s)]$. For case (2), $\mathbb{E}[R_{d+1}(s)] = \mathbb{E}[\sum_t \frac{R_d(s \cup \{v_t\})}{n_d}] = \frac{1}{n_d} \sum_t \mathbb{E}[R_d(s \cup \{v_t\})] = \frac{1}{n_d} \sum_t \mathbb{E}[H_d(s \cup \{v_t\})] = \frac{1}{n_d} \sum_t \mathbb{E}[H_{d+1}(s)] = \mathbb{E}[H_{d+1}(s)]$. Thus, $\mathbb{E}[R_{d+1}(s)] = \mathbb{E}[H_{d+1}(s)]$ and the theorem is proved. □

We next show that the recursive estimator R has a lower variance compared with H.

Theorem 3. *$\sigma[R_i(s)] \leq \sigma[H_i(s)]$ for any given partial instance s and $i \in [1, |\mathcal{V}_q|]$.*

Proof. We again use induction to prove the theorem. For the base case $i = 1$, we have $\sigma[R_1(s)] = \sigma[\frac{\mathbb{I}(s)}{\mathbb{P}(s)}] = \sigma[H_1(s)]$. Assume the inequality holds for $i = d$. There are also two cases in the inductive step for $i = d + 1$; (1) s is not inherited; (2) s is inherited. For case (1), it is easy to see $\sigma[R_{d+1}(s)] = \sigma[H_{d+1}(s)]$. For case (2),

$$\begin{aligned} \sigma[R_{d+1}(s)] &= \sigma\left[\sum_t \frac{R_d(s \cup \{v_t\})}{n_d}\right] = \sum_t \sigma\left[\frac{R_d(s \cup \{v_t\})}{n_d}\right] \\ &= \sum_t \frac{1}{n_d^2} \sigma[R_d(s \cup \{v_t\})] \leq \sum_t \frac{1}{n_d^2} \sigma[H_d(s \cup \{v_t\})] \\ &= \sum_t \frac{1}{n_d^2} \sigma[H_{d+1}(s)] = \frac{\sigma[H_{d+1}(s)]}{n_d} \leq \sigma[H_{d+1}(s)] \end{aligned}$$

Thus, $\sigma[R_i(s)] \leq \sigma[H_i(s)]$ holds for any s and i. □

Algorithm 3: Intra-Iteration Warp Streaming

```
// Replace Line 9-11 in Algorithm 1
curIter ← 0;
curV ← dummy;
curW ← 0;
curTotalW ← 0;
while _any(clen - curIter ≥ 32) do
    leaderId ← _ballot(clen - curIter ≥ 32);
    leaderSample ← _shfl(s, leaderId);
    workerCand ← _shfl(cand,leaderId) + threadId;
    rlen ← Refine(leaderSample,d,workerCand,1,refine);
    (v,w) ← Sample(leaderSample,d,refine,rlen);
    key ← UniformRand()^{1/w} if w ≠ 0 and 0 otherwise;
    (key*,v*,w*) = _reduce_max(tuple<>(key,v,w));
    totalW ← _reduce_sum(w);
    if threadId == leaderId then
        curTotalW ← curTotalW + totalW;
        (curV,curW) ← (v*,w*) with probability totalW / curTotalW
        curIter = curIter + 32;
while curIter < clen do
    rlen ← Refine(cg,s,d,cand+curIter,1,refine);
    (v,w) ← Sample(s,d,refine,rlen);
    curTotalW ← curTotalW + w;
    (curV,curW) ← (u,w) with probability w / curTotalW;
valid ← Validate(s,d,curV, curW / curTotalW);
```

Discussion. Regardless of how many inherited samples are generated in the process started by s, we only count one sample for the estimator R to ensure unbiased estimation. Further, R enables larger variance reduction compared with H when there are more inherited samples, i.e., larger n_i at any level i. Since the inherited samples except those from the parent are processed by the idle threads, inheritance better utilizes the computational resources to achieve lower variance.

4.2 Warp Streaming

Within each sample iteration, `Refine` scans the candidate array *cand* and outputs a *refine* array for `Sample`. Hence, varying lengths of *cand* and *refine* from different samples render intra-iteration imbalance workloads. To overcome the issue, we dynamically distribute the `Refine` and `Sample` workloads among threads in the same warp in a streaming manner. Intuitively, if a sample contains enough workloads to be parallelized by a warp, we keep the threads busy by streaming the `Refine` and `Sample` operations to the warp.

The warp streaming approach is presented in Algorithm 3. We initialize four variables to keep track of the streaming process for each sample iteration. *curIter* keeps track of the next vertex to process in the *cand* array. *curV* and *curW* denote the current vertex sampled and its corresponding sample weight. *curTotal* is the total sample weight accumulated among all processed candidates. There are two phases in streaming the candidates: *Collaborative Phase* and *Independent Phase*.

Collaborative Phase (Line 5-17). In Line 5, _any checks if there exists a thread holding a sample that has enough candidates to be processed by the warp. If so, one of the qualifying thread takes the leader role with _ballot and share its sample s to all threads in the warp via _shfl (Line 6-7). The leader also shares its candidate array *cand* so that each thread is assigned to process a unique candidate in *cand* (Line 8). The candidate is later fed to Refine followed by Sample. Now each thread holds a sampled vertex v with its sample weight w set by the Sample API, we need to select at most one vertex v^* from the warp and make sure its sampling probability is proportional to its sample weight w^*. To this end, we adapt the A-Res streaming algorithm [9] where a key is randomly generated as $r^{1/w}$ for any vertex v and weight w where r is a uniform random number in $(0, 1)$. Subsequently, _reduce_max safely selects the vertex v^* and its weight w^* with the largest key value. Furthermore, _reduce_sum computes the total weight $totalW$ among the warp. The leader thread can then update $curV$ and $curW$ with v^* and w^*. Line 15-16 guarantees the selected vertex sample $curV$ always has a sampling probability proportional to $curW$ among processed candidates with total sampling weight $curTotalW$. The leader also moves *curIter* to indict 32 candidates that have been processed.

Independent Phase (Line 18-22). When no thread holds more than 32 candidates to process, each thread streams the remaining candidates independently. A thread incrementally refines a candidate and samples a vertex v with weight w. Line 21-22 uses the same approach as Line 15-16 to ensure $curV$ has the correct sampling probability wrt. $curW$.

Finally, we check if $s \cup \{curV\}$ form a valid partial/full instance. The following theorem guarantees that $curV$ is selected with the correct probability distribution specified by the Sample API.

Theorem 4. *$curV$ is sampled with a probability $\frac{curW}{curTotalW}$ is an invariant in Algorithm 3.*

Proof. In the collaborative phase, we can show that any v^* is sampled with a probability of $\frac{w^*}{totalW}$ by the result of [17]. Thus, the total probability of v^* replacing existing $curV = u$ in Line 16 is:

$$\frac{w^*}{totalW} \cdot \frac{totalW}{curTotalW} = \frac{w^*}{curTotalW} = \frac{curW}{curTotalW} \tag{2}$$

Further, u has a sampling probability of $\frac{curW}{curTotalW - totalW}$ and it remains to be $curV$ after seeing v^* is:

$$\frac{curW}{curTotalW - totalW} \cdot \frac{curTotalW - totalW}{curTotalW} = \frac{curW}{curTotalW} \tag{3}$$

In both cases, the invariant holds for the collaborative phase. The invariant for the sample update in Line 21-22 of the independent phase can be proved similarly. □

5 EXPERIMENTAL EVALUATION

In this section, we evaluate the efficacy of gSWORD. We first introduce the experimental setup, then present the main efficiency results, next discuss the effectiveness of the proposed optimizations, and lastly demonstrate the superiority of PartialRefine over existing sampling algorithms.

Table 2: Dataset Statistics.

Categories	Dataset	$\mathcal{V}$	$\mathcal{E}$	d	$\mathcal{L}$
Biology	Yeast	3,112	12,519	8.0	71
	Human	4,674	86,282	36.9	44
	HPRD	9,460	34,998	7.4	307
Lexical	WordNet	76,853	120,399	3.1	5
Citation	Patents	3,774,768	16,518,947	8.8	20
Social	DBLP	317,080	1,049,866	6.6	15
	Youtube	1,134,890	2,987,624	5.3	25
Web	eu2005	862,664	16,138,468	37.4	40

5.1 Experimental Setup

Compared Methods. We compare gSWORD with *G-Care* [26], the-state-of-the-art CPU-based sampling framework for subgraph counting, and *NextDoor* [16], the state-of-the-art GPU-based sampling framework for traditional RW workloads. *G-Care* is originally a sequential framework. For a fair comparison, we parallelize *G-Care* with OpenMP. Specifically, we regard each sample as a task unit and keep load balance among threads with the *dynamic scheduling* [28]. It achieves high performance on CPUs because RW estimators are embarrassingly parallel. We study three sampling algorithms including WanderJoin (WJ), Alley (AL) and PartialRefine (PR). WanderJoin samples a vertex from the vertices adjacent to extracted subgraphs without refinement, whereas Alley and PartialRefine prune candidates before sampling. Alley and PartialRefine cannot be directly deployed to *NextDoor* because *NextDoor* does not consider the characteristics of RW estimators. Therefore, we implement Alley and PartialRefine without any frameworks but following the same parallelization strategy of *NextDoor* as their baseline on GPUs. In summary, we compare the following methods.

- CPU-WJ, CPU-AL and CPU-PR are the baseline on CPUs, implemented within *G-Care* [26].
- GPU-WJ, GPU-AL and GPU-PR are the baseline on GPUs, implemented by following the computation method of *NextDoor* [16].
- gSWORD-WJ, gSWORD-AL and gSWORD-PR are the implementation within gSWORD.

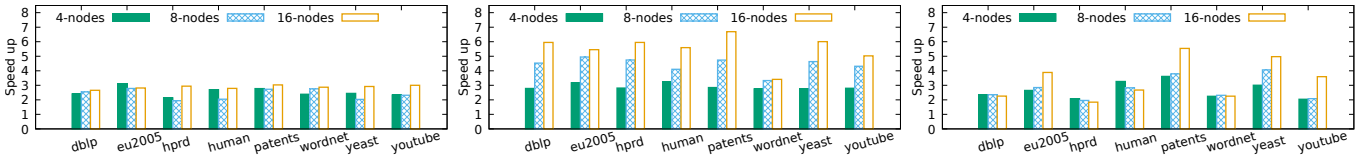
Data graphs. We conduct experiments on eight real-world datasets from diverse categories. These datasets are widely used in previous studies [3, 14, 34, 35, 48]. Table 2 presents the statistics of the datasets. Four of them are labeled graphs, namely, Yeast, Human, HPRD, and WordNet. The rest are unlabeled graphs. For unlabeled graphs, we follow the same method used in existing works [4, 14] to randomly generate labels for the vertices.

Queries. To align with previous research [2–4, 34], the queries are extracted from the data graphs using random walks. The number of vertices in queries are set to 4, 8, 16 with 16 as the default. We generate 20 queries for the same number of query vertices on each data graph. For queries with 8 or 16 vertices, we generate 10 sparse queries and 10 dense queries where a sparse query has the maximum degree less than 3. For all compared approaches, we generate 10^6 samples for each query by default as the RW estimators can converge for most datasets.

Environment. All experiments are conducted on a server equipped with Intel Xeon W-2133 CPUs (12 cores,3.6GHz), 64 GB main memory, and two RTX 2080 Ti GPUs. All source codes are implemented in CUDA/C++ and compiled by O3 optimizations.

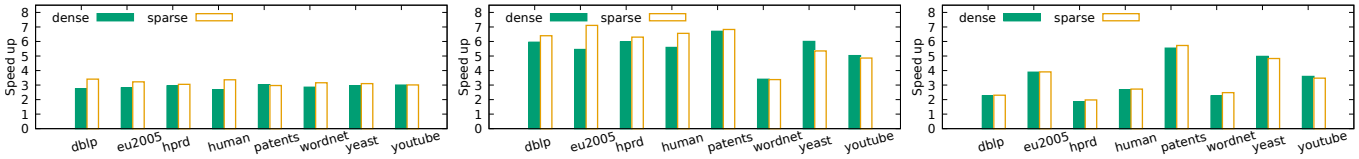
Table 3: Average running time (seconds) per query of the compared approaches for all datasets.

Methods		DBLP	eu2005	HPRD	Human	Patents	WordNet	Yeast	Youtube
CPU	CPU-WJ	22.843	46.228	15.396	13.039	82.788	6.744	2.929	27.794
	CPU-AL	38.219	124.693	16.711	57.723	132.309	7.278	4.571	59.025
	CPU-PR	21.433	55.867	14.924	15.699	83.421	6.610	3.338	28.611
GPU	GPU-WJ	0.932	1.296	0.771	1.121	0.915	0.714	0.701	0.789
	GPU-AL	2.880	3.822	2.768	4.515	2.281	1.026	2.697	1.529
	GPU-PR	0.921	1.834	0.855	1.284	1.728	0.660	1.512	1.006
	gSWORD-WJ	0.351	0.460	0.262	0.402	0.302	0.249	0.240	0.263
	gSWORD-AL	0.484	0.701	0.465	0.807	0.341	0.301	0.449	0.304
	gSWORD-PR	0.408	0.472	0.464	0.480	0.312	0.293	0.304	0.280



(a) WanderJoin (b) Alley (c) PartialRefine

Figure 8: The speedup of gSWORD over GPU baselines with the query size increasing.



(a) WanderJoin (b) Alley (c) PartialRefine

Figure 9: The speedup of gSWORD over GPU baselines with the query type varied.

5.2 Main Efficiency Results

Overall Comparison. Table 3 shows the average running time of all compared approaches on processing a query under the default setting. The error analysis of the sampling algorithms will be discussed in Section 5.4 later.

Among the CPU-based methods, CPU-AL is much slower than CPU-WJ due to the overhead of the refinement operations. For example, CPU-WJ is 4.4 times faster than CPU-AL on Human. In contrast, CPU-PR has competitive performance against CPU-WJ because the partial refinement technique in Section 3 significantly reduces the overhead of the refinement. Moreover, CPU-PR slightly outperforms CPU-WJ in DBLP, HPRD, WordNet in spite of the refinement. This is because the light-weight refinement terminates invalid samples at an early stage and leads to lower validation cost than CPU-WJ. Nonetheless, the running time of all CPU-based methods ranges from 3-133 seconds for processing one query. This result demonstrates the necessity of accelerating RW estimators.

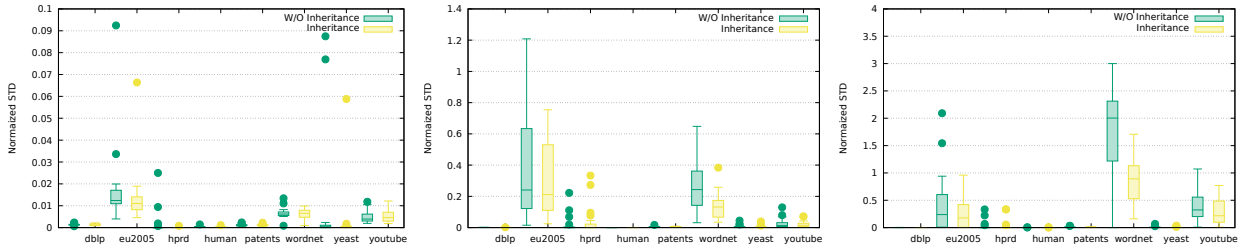
GPU-based methods produce massive efficiency boost. On average, the GPU baselines achieve 29x, 21x and 22x speedup over the CPU counterparts for WanderJoin, Alley and PartialRefine, respectively. The speedup of GPU-AL is lower compared with those of GPU-WJ and GPU-PR because the refinement of GPU-AL incurs heavy memory access cost. Overall, the running time of all GPU baselines ranges from 0.7-4.5 seconds. gSWORD further improves the performance and completes the queries within 1 second. On average, gSWORD runs 93x faster than the CPU baselines, and 3.8x faster than GPU baselines. This result shows the efficiency of gSWORD proposed in this paper.

Because GPU-based methods significantly outperform CPU-based, we focus on the GPU-based methods in the following experiments. Next, we examine the performance of gSWORD with the query size and query type varied.

Varying Query Size & Query Type. Figure 8 illustrates the speedup of gSWORD over GPU baselines with the query size increasing from 4 to 16. gSWORD brings at least 2x speedup over the counterparts on each case. The speedup for WanderJoin is close on queries with different sizes, whereas gSWORD generally achieves a high speedup on large queries for the other two methods, especially Alley. This is because the GPU baselines encounter serious inter-iteration workload imbalance and intra-iteration workload imbalance for large queries since 1) their samples consume more iterations and 2) the refinement operation for large queries incurs more overhead. The results show the effectiveness of the optimization techniques proposed in Section 4. Figure 9 shows the speedup on dense and sparse queries. gSWORD works well on the queries with different structures, which demonstrates the robustness of the framework.

5.3 Evaluation of Individual Techniques

We further conduct the ablation study to evaluate each optimization proposed in Section 4. Due to space limitations, we choose gSWORD-AL for the ablation study. In what follows, we first conduct a *microbenchmark* to compare sample synchronization vs. inheritance synchronization. Then we present the results to demonstrate the optimization effectiveness in terms of *variance reduction* and *runtime reduction*.



(a) Queries with 4 vertices (b) Queries with 8 vertices (c) Queries with 16 vertices

Figure 10: The normalized standard deviation of gSWORD-AL with/without sample inheritance optimization.

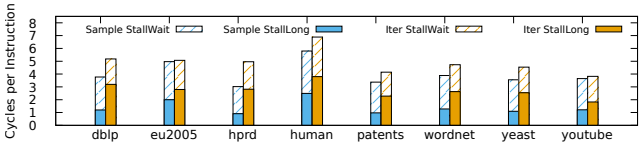


Figure 11: Micro-benchmark on the warp efficiency of sample and iteration synchronization with nsight.

Micro-benchmark. We use the *nsight* profiler to examine the top-2 warp stall factors for the sample and iteration synchronization methods. Figure 11 presents the profiling results. *StallLong* is the number of cycles stalled for memory load instructions. *StallWait* is the number of cycles stalled due to instruction loading issues. Although the iteration synchronization has better instruction-level parallelism (i.e., fewer StallWait cycles), the benefit is overwhelmed by the cost of memory accesses (i.e., more StallLong cycles). The micro-benchmark validates the significance of memory access impacts discussed in Section 3.2.

Variance Reduction. To evaluate the variance of gSWORD-AL with/without inheritance, we run each of our queries for 10 times under the default setting and record the normalized standard deviation (STD) in Figure 10. For queries with 4 vertices, the normalized STD for all queries is within 0.1, regardless of the inheritance optimization. When the query size increases to 8 and 16, we observe a clear variance reduction with inheritance, especially in `eu2005` and `WordNet`. For instance, in `WordNet`, the median of normalized STD for 8-vertices queries drops from 0.23 to 0.16 when inheritance is enabled. The drop becomes more significant for 16-vertices queries, i.e., from 2.0 to 0.8. This result confirms that the inheritance optimization can improve the effectiveness of RW estimators.

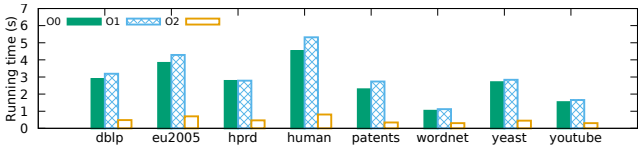


Figure 12: Runtime of gSWORD-AL with no optimization (O0), sample inheritance optimization only (O1) and sample inheritance+warp streaming optimizations (O2).

Runtime Reduction. We measure the runtime of gSWORD-AL when we enable sample inheritance and warp streaming incrementally. The results are presented in Figure 12. When only the inheritance optimization is enabled, the runtime shows a marginal increase of 10% on average. Despite the inherited samples are additional workload to be processed compared with the method without inheritance, the optimized memory access pattern of inheritance does not result in significant runtime overhead. The increase in runtime is also compensated by more robust estimation with lower variance for enabling inheritance. We see a significant reduction in runtime when we further enable the warp streaming optimization. In particular, when both optimizations are enabled, the runtime is reduced by 6.1x on average.

5.4 Evaluation of Sampling Algorithms

We measure the accuracy of RW estimators with *q-error*, a widely used metric for the cardinality estimation problem [23]. Suppose that the estimated cardinality is $\hat{c}$ and the ground truth is c. *Q-error* is equal to $max(max(1,c)/max(1,\hat{c}), max(1,\hat{c})/max(1,c))$. The value is no less than one and a smaller value indicates a better estimation.

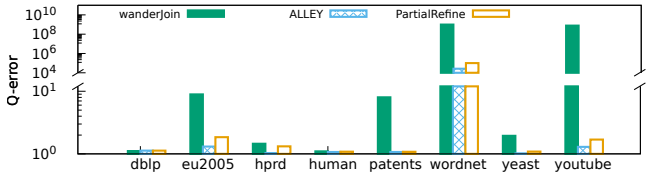


Figure 13: Q-error of RW estimators under study.

Figure 13 shows the *q-error* of the three RW estimators under study. `Alley` and `PartialRefine` generally perform much better than `WanderJoin`, and their *q-error* is below two on all data graphs except `WordNet`. All three methods have a poor performance on `WordNet` due to the highly skewed search space. Despite that, `Alley` and `PartialRefine` reduce the *q-error* from 10^9 to 10^5. Those results suggest that the refinement has a great impact on the estimation accuracy, and `PartialRefine` is competitive with `Alley` in terms of the estimation accuracy. Moreover, we select two representative test cases to illustrate the performance characteristics of `PartialRefine`, which achieves a good trade-off between estimation accuracy and sampling cost. Particularly, *easy query* is a query on which all three methods can give an accurate estimation with a small number of samples, while *hard query* is a query on which competing methods require massive samples to make an accurate estimation.

Figures 14 and 15 present the experiment results on the easy query and the hard query, respectively. We omit the *q-error* of gSWORD-WJ on the hard case in Figure 15 because its value is 1451 given 10^4 samples, and still above four after collecting as many as 10^8 samples. As shown in the figures, the *q-error* of all methods on

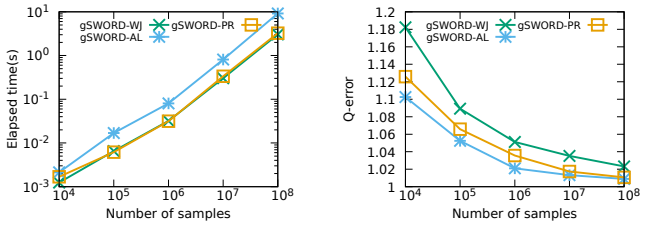


Figure 14: Sampling comparison for an easy query in eu2005.

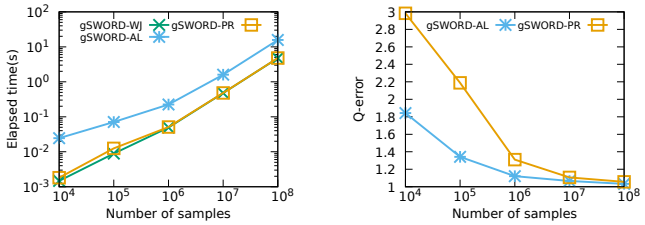


Figure 15: Sampling comparison for a hard query in eu2005**. The q-error for** gSWORD-WJ **is omitted due to large error.**

the easy case is below 1.2 with 10^4 samples. In contrast, the *q-error* of gSWORD-AL and gSWORD-PR is much lower than gSWORD-WJ on the hard case. On the other hard, the running time of gSWORD-WJ and gSWORD-PR is much smaller than that of gSWORD-AL, and the efficiency of gSWORD-PR is very close to that of gSWORD-WJ. In summary, gSWORD-PR runs as fast as gSWORD-WJ and achieves competitive q-error against gSWORD-AL.

Finally, we evaluate the impact of the refinement factor α on the estimation accuracy and efficiency for gSWORD-PR. Setting α to a higher value leads to more valid samples, which produce more accurate estimation empirically [18, 21]. However, it also incurs a higher refinement overhead. Thus, we use the reward $r = \frac{SuccessRatio}{Runtime}$ to measure the impact where *SuccessRatio* is the ratio of valid samples and *Runtime* is the elapsed time of the query. As shown in the figure, we achieve the highest reward when α is ranged from 0.1 and 0.3. Thus, we set $\alpha = 0.1$ for all our experiments.

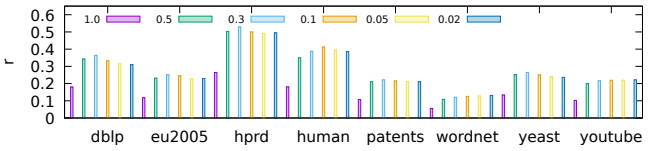


Figure 16: The impact of α on sample reward r.

6 RELATED WORK

Random walk (RW) is an effective approach to extract information from large graphs. For example, many RW algorithms are proposed to learn graph representations such as DeepWalk [27] and Node2vec [12]. Many surveys [8, 31, 41, 42] on RW-based algorithms have been published. Please refer to these papers for the details. In this paper, we focus on research on RW systems. To accelerate RW algorithms, researchers recently proposed a variety frameworks on which users can implement different RW algorithms. These frameworks regard each walk as a parallel task unit and focus on accelerating the procedure of sampling a vertex from the neighbors of the current residing vertex given the probability distribution customized by users. In the following, we first review the CPU-based systems followed by the GPU-based systems.

CPU-based Systems. KnightKing [43] is a distributed framework adopting the BSP model. In particular, it moves a step for all queries at one iteration. To reduce the communication cost among machines, it optimizes the *rejection sampling* method to avoid scanning the neighbor set of the current residing vertex when sampling a vertex at each step. GraphWalker [40] is an I/O efficient framework on a single machine. It adopts the ASP model where each thread executes a walk independently. It divides the graph that cannot reside in the memory into a set of partitions and optimizes the scheduling method of loading partitions into memory to minimize the number of I/Os. ThunderRW [33] focuses on increasing in-memory computation efficiency for random walks. Specifically, it proposes the step interleaving technique that executes different queries in an interleaving manner to reduce CPU pipeline stalls incurred by random memory accesses. Uninet [44] proposes a sampler called M-H to generate a sample vertex in constant time.

GPU-based Systems. To further improve the performance, several GPU-based systems are proposed. C-SAW [25] adopts the BSP model that executes a step for all queries at each iteration. It adopts the *inverse transformation sampling* method to sample a vertex, which needs to scan the neighbors of the current residing vertex. To fully utilize GPUs' parallel computation capability, C-SAW supports RW with the same length only. NextDoor [16] is a framework targeting at graph sampling applications. It adopts the *rejection sampling* method. Users can implement a wide variety of RW-algorithms by defining a "next" function, which specifies the logic of picking a vertex. SkyWalker [39] improves the efficiency on large graphs by optimizing the *alias sampling* method on GPUs. In particular, it assigns computation resources to build the alias table based on the vertex degrees to keep workload balance.

Nevertheless, all existing CPU and GPU systems focus on the optimization of the sampling stage of traditional random walks. However, as discussed in Section 1, the heavy refinement step in RW estimators for subgraph counting incurs the inter-iteration workload imbalance and intra-iteration workload imbalance problems, which lead to the under utilization of GPUs' resources. Different from all previous works, we focus on designing an efficient framework to accelerate RW estimators for subgraph counting.

7 CONCLUSION

We present gSWORD, a general framework that accelerate RW estimators for subgraph counting on GPUs. gSWORD provides an iterative workflow consisting of Refine-Sample-Validate steps. With the gSWORD workflow, users only need to define the sequential logic of the sampling process in RW estimators and gSWORD takes care of the parallel optimizations on GPUs. Based on the workflow, we also propose a new RW estimator PartialRefine that achieves a good balance between sample error and speed. There are unique challenges to accelerate the workflow on GPUs due to severe workload imbalance. We propose sample inheritance for inter-iteration imbalance and warp streaming for intra-iteration imbalance. Through extensive experiments, we verify the efficacy of gSWORD over the state-of-the-art CPU and GPU baselines. The ablation studies have also confirmed the effectiveness of our optimization strategies.

REFERENCES

[1] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems(TODS)*, 42(4):1–44, 2017.

[2] B. Archibald, F. Dunlop, R. Hoffmann, C. McCreesh, P. Prosser, and J. Trimble. Sequential and parallel solution-biased search for subgraph algorithms. In *CPAIOR*, pages 20–38, 2019.

[3] B. Bhattarai, H. Liu, and H. H. Huang. Ceci: Compact embedding cluster index for scalable subgraph matching. In *SIGMOD*, pages 1447–1462, 2019.

[4] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In *SIGMOD*, pages 1199–1214, 2016.

[5] I. Buck. Gpu computing with nvidia cuda. In *ACM SIGGRAPH 2007 courses*, pages 6–es. 2007.

[6] X. Chen, Y. Li, P. Wang, and J. C. Lui. A general framework for estimating graphlet statistics via random walk. *PVLDB*, 10(3):253–264, 2016.

[7] S. A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158, 1971.

[8] P. Cui, X. Wang, J. Pei, and W. Zhu. A survey on network embedding. *IEEE Transactions on Knowledge and Data Engineering(TKDE)*, 31(5):833–852, 2018.

[9] R. El Sibai, Y. Chabchoub, J. Demerjian, Z. Kazi-Aoul, and K. Barbar. Sampling algorithms in data stream environments. In *ICDEc*, pages 29–36, 2016.

[10] J. Enright and R. R. Kao. Epidemics on dynamic networks. *Epidemics*, 24:88–97, 2018.

[11] P. Fournier-Viger, G. He, C. Cheng, J. Li, M. Zhou, J. C.-W. Lin, and U. Yun. A survey of pattern mining in dynamic graphs. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery(DMKD)*, 10(6):e1372, 2020.

[12] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *KDD*, pages 855–864, 2016.

[13] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. pages 1024–1034, 2017.

[14] M. Han, H. Kim, G. Gu, K. Park, and W.-S. Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *SIGMOD*, pages 1429–1446, 2019.

[15] Z. Harchaoui and F. Bach. Image classification with segmentation graph kernels. In *CVPR*, pages 1–8, 2007.

[16] A. Jangda, S. Polisetty, A. Guha, and M. Serafini. Accelerating graph sampling for graph machine learning using gpus. In *EuroSys*, pages 311–326, 2021.

[17] R. Jayaram, G. Sharma, S. Tirthapura, and D. P. Woodruff. Weighted reservoir sampling from distributed streams. In *PODS*, pages 218–235, 2019.

[18] K. Kim, H. Kim, G. Fletcher, and W.-S. Han. Combining sampling and synopses with worst-case optimal runtime and quality guarantees for graph pattern cardinality estimation. In *SIGMOD*, pages 964–976, 2021.

[19] D. Kirk, B. S. Center, et al. Nvidia cuda software and gpu parallel computing architecture. 2008.

[20] M. Kuczma. *An introduction to the theory of functional equations and inequalities: Cauchy's equation and Jensen's inequality*. 2009.

[21] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join and xdb: online aggregation via random walks. *ACM Transactions on Database Systems(TODS)*, 44(1):1–41, 2019.

[22] Y. Li, Z. Wu, S. Lin, H. Xie, M. Lv, Y. Xu, and J. C. Lui. Walking with perception: Efficient random walk sampling via common neighbor awareness. In *ICDE*, pages 962–973, 2019.

[23] G. Moerkotte, T. Neumann, and G. Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *PVLDB*, 2(1):982–993, 2009.

[24] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *ICDE*, pages 984–994, 2011.

[25] S. Pandey, L. Li, A. Hoisie, X. S. Li, and H. Liu. C-saw: A framework for graph sampling and random walk on gpus. In *SC*, pages 1–15, 2020.

[26] Y. Park, S. Ko, S. S. Bhowmick, K. Kim, K. Hong, and W.-S. Han. G-care: a framework for performance benchmarking of cardinality estimation techniques for subgraph matching. In *SIGMOD*, pages 1099–1114, 2020.

[27] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. In *KDD*, pages 701–710, 2014.

[28] I. Qureshi. Cpu scheduling algorithms: A survey. *International Journal of Advanced Networking and Applications(IJANA)*, 5(4):1968, 2014.

[29] P. Ribeiro, P. Paredes, M. E. Silva, D. Aparicio, and F. Silva. A survey on subgraph counting: concepts, algorithms, and applications to network motifs and graphlets. *ACM Computing Surveys(CSUR)*, 54(2):1–36, 2021.

[30] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *PVLDB*, 11(4):420–431, 2017.

[31] P. Sarkar and A. W. Moore. Random walks in social networks and their applications: a survey. In *Social Network Data Analytics*, pages 43–77. 2011.

[32] N. Shervashidze, S. Vishwanathan, T. Petri, K. Mehlhorn, and K. Borgwardt. Efficient graphlet kernels for large graph comparison. In *AISTATS*, pages 488–495, 2009.

[33] S. Sun, Y. Chen, S. Lu, B. He, and Y. Li. Thunderrw: An in-memory graph random walk engine. *PVLDB*, 14(11):1992–2005, 2021.

[34] S. Sun and Q. Luo. In-memory subgraph matching: An in-depth study. In *SIGMOD*, pages 1083–1098, 2020.

[35] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9):788–799, 2012.

[36] S. T. Tokdar and R. E. Kass. Importance sampling: a review. *Wiley Interdisciplinary Reviews: Computational Statistics(WIREs Comp Stats)*, 2(1):54–60, 2010.

[37] V. Vacic, L. M. Iakoucheva, S. Lonardi, and P. Radivojac. Graphlet kernels for prediction of functional residues in protein structures. *Journal of Computational Biology(J. Comput. Biol)*, 17(1):55–72, 2010.

[38] L. Wang and J. D. Owens. Fast gunrock subgraph matching (gsm) on gpus. *arXiv preprint arXiv:2003.01527*, 2020.

[39] P. Wang, C. Li, J. Wang, T. Wang, L. Zhang, J. Leng, Q. Chen, and M. Guo. Skywalker: Efficient alias-method-based graph sampling and random walk on gpus. In *PACT*, pages 304–317, 2021.

[40] R. Wang, Y. Li, H. Xie, Y. Xu, and J. C. Lui. Graphwalker: An i/o-efficient and resource-friendly graph analytic system for fast and scalable random walks. In *USENIX ATC*, pages 559–571, 2020.

[41] F. Xia, J. Liu, H. Nie, Y. Fu, L. Wan, and X. Kong. Random walks: A review of algorithms and applications. *IEEE Transactions on Emerging Topics in Computational Intelligence(TETCI)*, 4(2):95–107, 2019.

[42] F. Xia, K. Sun, S. Yu, A. Aziz, L. Wan, S. Pan, and H. Liu. Graph learning: A survey. *IEEE Transactions on Artificial Intelligence(TAI)*, 2(2):109–127, 2021.

[43] K. Yang, M. Zhang, K. Chen, X. Ma, Y. Bai, and Y. Jiang. Knightking: a fast distributed graph random walk engine. In *SOSP*, pages 524–537, 2019.

[44] X. Yao, Y. Shao, B. Cui, and L. Chen. Uninet: Scalable network representation learning with metropolis-hastings sampling. In *ICDE*, pages 516–527, 2021.

[45] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna. Graphsaint: Graph sampling based inductive learning method. 2020.

[46] L. Zhang, M. Song, Z. Liu, X. Liu, J. Bu, and C. Chen. Probabilistic graphlet cut: Exploiting spatial structure cue for weakly supervised image segmentation. In *CVPR*, pages 1908–1915, 2013.

[47] S. Zhang, S. Li, and J. Yang. Gaddi: distance index based subgraph matching in biological networks. In *EDBT*, pages 192–203, 2009.

[48] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3(1-2):340–351, 2010.