Accelerating Graph Sampling for Graph Machine Learning using GPUs

Abhinav Jangda
University of Massachusetts Amherst
United States

Sandeep Polisetty
University of Massachusetts Amherst
United States

Arjun Guha
Northeastern University
United States

Marco Serafini
University of Massachusetts Amherst
United States

Abstract

Representation learning algorithms automatically learn the features of data. Several representation learning algorithms for graph data, such as DeepWalk, node2vec, and GraphSAGE, *sample* the graph to produce mini-batches that are suitable for training a DNN. However, sampling time can be a significant fraction of training time, and existing systems do not efficiently parallelize sampling.

Sampling is an "embarrassingly parallel" problem and may appear to lend itself to GPU acceleration, but the irregularity of graphs makes it hard to use GPU resources effectively. This paper presents NextDoor, a system designed to effectively perform graph sampling on GPUs. NextDoor employs a new approach to graph sampling that we call transit-parallelism, which allows load balancing and caching of edges. NextDoor provides end-users with a high-level abstraction for writing a variety of graph sampling algorithms. We implement several graph sampling applications, and show that NextDoor runs them orders of magnitude faster than existing systems.

ACM Reference Format:
Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2021. Accelerating Graph Sampling for Graph Machine Learning using GPUs. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–28, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3447786.3456244

1 Introduction

Representation learning is a fundamental problem in machine learning. Its goal is to learn features of data instead of hand-engineering them. Representation learning on graph data involves mapping vertices (or subgraphs) to a d-dimensional vector known as an *embedding*. The embedding is then used as a feature vector for other downstream graph machine learning tasks. Graph representation learning is a fundamental step in domains such as social network analysis, recommendations, epidemiology, and more.

Several algorithms for graph representation learning first *sample* the input graph to obtain mini-batches and then *train* a deep neural network (DNN) or a graph neural network (GNN) based on the samples. Moreover, different learning algorithms require different sampling mechanisms. For example, DeepWalk [28] and node2vec [12] use variants of random walks. In contrast, GraphSAGE [13], which Pinterest uses for recommendation [39], samples the k-hop neighborhood of a vertex and uses their attributes to learn an embedding for each vertex.

Although several systems effectively leverage GPUs for the DNN training step, the same is not true for the sampling step. Graph sampling takes a significant portion of total training time in real-world applications. Table 1 shows the impact of graph sampling in existing GNNs. In each epoch, a GNN first samples the input graph to obtain mini-batches and then trains the DNN. Graph sampling is an irregular computation that is typically performed using the CPU, whereas training is performed on the GPU. In our experiments, we found that graph sampling can take up to 62% of an epoch's time.[1] This bottleneck is further exacerbated if the CPU is attached to multiple GPUs and cannot produce enough samples to saturate them. Hence, accelerating graph sampling is important to improve the end-to-end training time.

Since samples are drawn independently, graph sampling is an "embarrassingly parallel" problem that seems ideal for exploiting the parallelism of GPUs. However, for a GPU to provide peak performance, the algorithm must be carefully designed to ensure regular computation and memory accesses, which is challenging on irregular graphs. Several

[1]We use a 16-core Intel Xeon Silver CPU and an NVIDIA Tesla V100 GPU.

Input Graphs	PPI	Reddit
GraphSAGE [13]	51%	45%
FastGCN [3]	26%	52%
LADIES [44]	40%	62%
ClusterGCN [6]	4.1%	24%
GraphSAINT [40]	25%	30%
MVS [7]	24%	25%

Table 1. Fraction of time spent in graph sampling in training.

systems have been designed for random walks [38], graph mining [5, 15, 23, 35], and graph analytics [19, 26, 31, 37]. These systems consider samples (or subgraphs) as the fundamental unit of parallelism: they grow all samples in parallel by looking up the neighbors of the vertices of each sample. However, such an approach leads to two issues in these systems: (i) irregular memory accesses and divergent control flow because consecutive threads can access the neighbors of different vertices, and (ii) lower parallelism because computation on all vertices in a sample is performed serially by the thread responsible for growing the sample.

In this paper, we present NextDoor, the first system to perform efficient graph sampling on GPUs. NextDoor introduces *transit-parallelism*, which is a new approach to parallel graph sampling. In transit-parallelism, the fundamental unit of parallelism is a *transit vertex*, which is a vertex whose neighbors may be added to one or more samples of the graph. In transit-parallelism, each transit vertex is assigned to a group of threads such that each thread adds one neighbor of the transit vertex to one sample. With this technique we obtain better GPU execution efficiency due to lower warp divergence, coalesced global memory accesses, and caching of the transit vertex edges in low-latency shared memory. Thus the irregular computation on the graph is changed to a regular computation. NextDoor effectively balances load across transit vertices, by assigning them to *different kernels* based on the number of samples associated with a transit vertex. Each kernel uses a different scheduling and caching strategy to maximize the usage of execution resources and memory hierarchy. NextDoor has a high-level API that abstracts away the low-level details of implementing sampling on GPUs and enables ML domain experts to write efficient graph sampling algorithms with few lines of code. NextDoor achieves significant speedups over state-of-the-art systems for graph sampling and improves training time of existing GNN systems by up to 4.75×. The contributions of this paper are:

- A high-level API for building graph sampling algorithms with efficient execution on GPUs (Section 4).
- A new transit-parallel paradigm to perform graph sampling on GPUs (Section 5).
- NextDoor, which leverages transit-parallelism and adds techniques for load balancing and caching of a transit's adjacency list (Section 6).

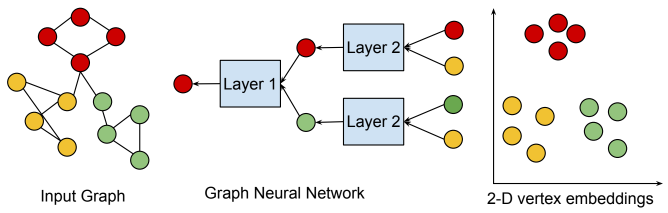


Figure 1. Representation learning on graphs. For each vertex of the input graph (on left), the Graph Neural Network (in middle) aggregates the information from k-hop neighbors back to the vertex. After training, each vertex is mapped to an embedding in a 2-dimensional space (on right).

- Performance evaluation of NextDoor against state-of-the-art systems: (i) a system of writing random walks KnightKing [38], (ii) existing GNNs (GraphSAGE [13], FastGCN [3], LADIES [44], GraphSAINT [40], MVS [7], ClusterGCN [6]), and (iii) two graph processing frameworks Gunrock [37] and Tigr [26] (Section 8).

NextDoor and our experimental setup is available at https://plasma-umass.org/nextdoor-eurosys21/.

2 Background and Motivation

2.1 Representation Learning on Graphs

The goal of a graph representation learning algorithm is to map vertices (or subgraphs) to an embedding, which is a d-dimensional vector (Figure 1).

Early algorithms, such as DeepWalk [28] and node2vec [12] employ *shallow encodings*. Given an input graph with n vertices and a target d-dimensional Euclidean space, a shallow encoding is a $d \times n$ matrix where the i^{th} column contains the embedding of vertex v_i. These algorithms are *transductive*: they take a static graph as input and produce embeddings only for the vertices in that graph. They typically adapt the Skip-Gram approach [24] to graphs, performing random walks to obtain context and target vertices.

More recent algorithms, including GraphSAGE [13], are *inductive*: they produce embeddings that generalize to previously unseen vertices. This property is particularly useful to build inference algorithms that work on dynamic, real-world graphs. Inductive algorithms learn a *deep encoding*, i.e., a function describing how to obtain a mapping, instead of the static map, and are also known as Graph Neural Networks (GNNs). Social networks like Pinterest uses GNNs to recommend newly added posts to users [39].

During GNN training and inference, each vertex aggregates information from nodes in its k-hop neighborhood using a neural network. Hops correspond to network layers, which are arranged as a tree. Aggregation follows the tree from the k-hop vertices back to the root vertex (Figure 1). Some GNN training systems perform whole-graph training, that is, they execute aggregation across the entire input graph without sampling [16, 21]. Most GNN algorithms, however, use mini-batching, where a mini-batch consists of a set of root vertices and a sample of their k-hop neighborhood (e.g., [3, 6, 7, 13, 40, 44]). The mini-batching approach based on sampling is easier to parallelize and scale to large graphs.

2.2 Requirements for GPU Performance

We now present an overview of modern GPU architectures and highlight characteristics of high-performance GPU code. These characteristics motivate the design of NextDoor.

The fundamental unit of computation in a GPU is a *thread*. Threads are statically grouped into *thread blocks* and assigned a unique ID within a block. A GPU has multiple *streaming multiprocessors (SMs)*, each of which executes one or more thread blocks. GPUs have several types of memory, two of which are relevant for this paper: (1) Each SM has a private memory, called *shared memory*, which is only available to thread blocks assigned to that SM; (2) The GPU has *global memory*, which is accessible to all SMs. The access latency for global memory is significantly higher compared to shared memory and registers.

To run a thread block, an SM schedules a subset of threads from the thread block, known as a *warp*. A warp typically consists of 32 threads with consecutive thread IDs. Moreover, GPUs employ a *Single Instruction Multiple Threads (SIMT)* execution model: all threads in a warp run the same instruction in lock-step. One consequence of this execution model is that two threads cannot execute both sides of a branch concurrently. Therefore, when the threads in a warp encounter a branch, the subset of threads that do not take the branch must wait for other threads to complete the branch. This phenomenon is known as *warp divergence* can lead to poor performance. Thus, *minimizing warp divergence* is key to achieving high-performance on GPUs.

It is also important to *balance load across thread blocks*. Suppose an SM is assigned to run thread blocks A and B. Each thread block reserves a portion of SM resources, including registers and shared memory. When A is waiting, e.g., due to memory latency, the SM cannot switch execution to B because the resources reserved by A are unavailable to B. (This behavior is very different from threads on a CPU, where context switches save registers to memory, thus all CPU registers are available to all threads.) Hence, we need to *balance resource usage across thread blocks* to concurrently execute the maximum number of thread blocks per SM.

Finally, a GPU program must explicitly choose to work with shared or global memory, and *use shared memory when possible* to maximize performance. In particular, when a thread waits on a memory access, it blocks all other threads in the same warp (another consequence of the SIMT execution model). Therefore, the high latency of global memory access is particularly significant. Fortunately, the GPU can provide high-bandwidth access to global memory by *coalescing* several memory accesses from the same warp. This is only possible when *concurrent memory accesses from threads in the same warp access consecutive memory segments*.

3 An Abstraction for Graph Sampling

We introduce a general-purpose abstraction for graph sampling and use it to express common sampling algorithms. The input to a graph sampling algorithm is a graph and an initial set of *samples*, where each sample is a subset of vertices (and optionally edges) of the graph. The algorithm iteratively grows each sample to include additional vertices in a series of *steps*, and its output is the final set of expanded samples. At each step, a sampling algorithm performs the following operations for each sample:

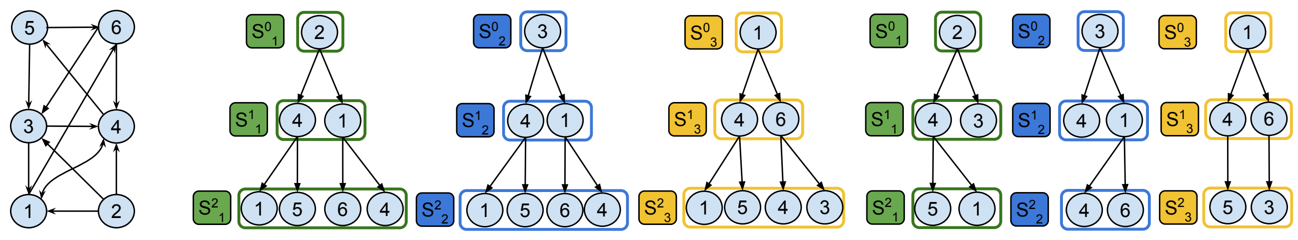
1. Iteratively sample one vertex at a time and add it to the sample. This operation can access the neighborhood of some vertices, which we call *transit* vertices.
2. Determine the set of transit vertices for the next step.

A graph sampling application can be expressed by providing user-defined functions that describe how to perform these operations. Namely, the *next* function describes how to sample one new vertex. The *samplingType* function describes the granularity at which we sample new vertices. There are two types of sampling:

1. *Individual Transit Sampling:* the *next* function is executed per-transit a fixed number of times. It has access to the neighborhood of that transit.
2. *Collective Transit Sampling:* the *next* function is executed per-sample a fixed number of times. It has access to the combined neighborhood of all transit vertices.

Finally, the *stepTransits* function selects the vertices of the sample that will act as transit vertices in the next step. Other user-defined parameters are the number of steps k, which could be set to ∞ if it varies from sample to sample, and the maximum number m_i of new vertices sampled per transit vertex (for individual transit sampling) or per sample (for collective transit sampling) at step i. We now show that by changing these user-defined functions and parameters, we can express a wide variety of sampling algorithms.

Random walks [12, 27, 28] A random walk starts from some initial set of root vertices. In a static random walk, the probability of picking an edge is known beforehand, whereas in a dynamic random walk, the probability of picking an edge depends on properties of vertices that were previously visited on that walk. DeepWalk [28] performs fixed-size biased static random walks, where the probability of following an edge is proportional to the edge weight. Personalized Page Rank [14] performs a variable-size biased static random walk, where the probability of ending the random walk is defined by the user. In contrast, node2vec [12] is a dynamic random walk, which can be biased to stay closer to the starting vertex or to sample vertices that are further away. Yang et al. [38] provide a taxonomy of different types of random walks.



(a) Graph **(b)** 2-hop Neighborhood sampling **(c)** Layer Sampling

Figure 2. Execution of a 2-hop Neighborhood sampling and Layer Sampling on graph in Figure 2a for samples S_1, S_2, and S_3. For both applications the *next* function uniformly samples two neighbors i.e., $m_1 = 2$ and $m_2 = 2$, and *stepTransits* returns the vertices added in previous step as transits. S^i_j denotes the vertices obtained after step i for sample S_j. Initially, S_1, S_2, and S_3 contains a single vertex. In the first step, the neighbors of transit vertices, i.e., ②, ③, and ① are added to the sample by both applications. While 2-hop Neighborhood choose two neighbors of each transit vertex, Layer Sampling choose two neighbors from the set of all neighbors of all transit vertices of a sample. In the second step, vertices sampled in first step becomes the transit vertices. Output of both sampling applications for each sample contains all vertices sampled at all steps.

Our abstraction supports random walks as follows. Random walks are individual transit sampling applications because they sample a single neighbor of each transit vertex of the sample at each step. Every element m_i is 1, since all random walks add at most one vertex at each step. Since the transit vertex is the previously sampled vertex, the *stepTransits* function returns the previously sampled vertex in the sample. The root vertices are the initial samples, such that each sample is assigned one root vertex. The number of steps k describes the length of the fixed-size random walks in algorithms like DeepWalk and node2vec. In these algorithms, *next* always returns a vertex. However, for applications that perform a variable-size walk, such as Personalized Page Rank, k is set to ∞. For termination, *next* can decide to not add a new vertex to the sample. The walk for a sample ends when the sample has no more new transit vertices.

MultiDimensional Random Walks [30] generalize regular random walks. Each initial sample (walk) has a set of *root* vertices, which are potential transit vertices. Each step extends the walk as follows. First, a transit vertex is selected from the set of root vertices. Then, a neighbor of the transit vertex is added to the sample and replaces the transit vertex in the set of roots. Multi-dimensional random walks can be represented in our abstraction as follows. These walks perform individual transit sampling with k set to the length of walk, and each element m_i to one. The function *stepTransits* returns a transit vertex for a sample by randomly choosing one of the root vertices and then the *next* function chooses a neighbor of the transit vertex. GraphSAINT [40] samples the graph using multi-dimensional random walks.

***k*-hop Neighborhood Sampling** [13] A k-hop neighborhood sampling algorithm employed in GraphSAGE [13] adds one or more neighbors of a transit vertex at each step. Figure 2b shows the execution of a 2-hop Neighborhood sampler that samples two neighbors of each transit at every step. This is individual transit sampling and can be represented in our abstraction by setting $k = 2$, $m_1 = m_2 = 2$, having *stepTransits* return all the vertices added in previous step as transit, and having *next* uniformly choose neighbors of each transit vertex and add them to the sample.

Layer Sampling [10] At each step i, layer sampling samples m_i vertices from the set of neighbors of all transit vertices of a sample, until the size of the sample reaches a maximum size (M) given by the user. Unlike the sampling applications we have seen previously, layer sampling is a collective transit sampling algorithm, where m_i neighbors are chosen from the set of all neighbors of all transit vertices of a sample. The transit vertices of a sample at a step are the vertices added in previous step to the sample. Since this sampling can run for arbitrary steps, we set k to ∞ and the *next* function chooses a neighbor from the set of all neighbors, if the size of the sample is less than M, otherwise *next* does to not add a new vertex to the sample. Figure 2c shows the execution of Layer Sampling that samples two edges from each transit vertex. This can be represented in our abstraction by setting $k = 2$, $m_1 = m_2 = 2$, having *stepTransits* return the vertices added in previous step, and having *next* uniformly choose a neighbor from the combined neighborhood.

4 Graph Sampling using NextDoor

This section presents NextDoor's API, which is based on our graph sampling abstraction that we presented in the previous section. The API allows users to write a variety of graph sampling algorithms in just a few lines of code. Moreover, it allows users who are not experts in GPU programming to leverage modern GPU architectures.

4.1 Programming API

The inputs to NextDoor are a graph, an initial set of samples, and several user-defined functions (Figure 3), which we detail below. The output is an expanded set of samples. If desired, NextDoor can pick the initial set of samples automatically (e.g., select one random vertex per sample).

The user selects collective or individual transit sampling using `samplingType`. The `stepTransits` function returns the transit vertices for a sample at a given step. In individual transit sampling, the number of transit vertices for each sample at step j are $\prod_{i=0}^{j} m_i$. In collective transit sampling, the number of transit vertices for each sample are m_{i-1}. This function takes three arguments: 1) the step (`step`), 2) the sample (`s`), and 3) the index of transit out of all transits to return (`transitIdx`). The user must also define a sampling function to use at each step of the computation (`next`). This function receives four arguments: 1) the sample (`s`), 2) the source edge set to sample neighbors from (`srcEdges`), 3) transit vertices (`transits`) forming the source edge set, and 4) the current step (`step`). If the sampling is individual transit sampling then `transits` contains only a single transit vertex and `srcEdges` contains the edges of this transit vertex. Otherwise, `transits` contains all transit vertices of the sample and `srcEdges` contains edges of all transit vertices. The result of `next` must be a vertex to add to s (or a constant `NULL` that indicates not to add a neighbor). The function `s.prevVertex(i, pos)` returns the vertex added at position pos of the last i^{th} step, and the function `s.prevEdges(i, pos)` returns the edges of that vertex. This information is necessary for applications, such as node2vec. The `steps` function defines the number of computational steps in the application (k). For applications that do not run for a fixed number of steps, such as Personalized Page Rank and Layer Sampling, they can return a special constant `INF` and the sampling process for a sample is stopped when no new transit vertices are added to the sample. The value returned by the `sampleSize` function determines how many times the `next` function is invoked on each individual or collective neighborhood for each sample at each step. The `unique` function specifies if at a step the sample should contain only unique vertices. The `Vertex` class has utility methods for computing the vertex degree, the maximum weight of all edges (`maxEdgeWeight`), and the prefix sum of all edges' weights. Users can extend the class to include application-specific vertex attributes to be added to the samples.

Output format NextDoor supports two output formats based on the application. 1) NextDoor can return an array of samples, such that each sample contains all transit vertices sampled at all steps. This format is required by GNNs that use random walks and layer sampling. 2) NextDoor can return vertices sampled at each step in an individual array. This format is required by GNNs that uses k-hop neighborhood sampling. The arrays are stored in the GPU in both cases.

```
Vertex next(Sample s, Array<Vertex> transits,
            Array<Edge> srcEdges, int step);
int steps();
int sampleSize(int step);
bool unique(int step);
enum SamplingType {Individual, Collective};
SamplingType samplingType();
Vertex stepTransits(int step, Sample s,
                    int transitIdx);
```

Figure 3. User defined functions required to implement a graph sampling application in NextDoor

4.2 Use Cases

We now present the implementation of several graph sampling algorithms using NextDoor.

node2vec The node2vec algorithm is a second-order random walk. Let v be transit vertex and t be the transit vertex of the last step. The probability of picking edge (v, u) depends on hyperparameters p and q, and is determined using three cases: (i) if $u = t$ then the probability is p, (ii) if $u \neq t$ and u is a neighbor of t then the probability is $1/q$, or (iii) if $u \neq t$ and u is not a neighbor of t then the probability is 1. The next vertex is sampled using rejection sampling, which takes these parameters as input [38].

Figure 4a presents node2vec in NextDoor. The argument `transits` of `next` contains one transit vertex, since random walk is an individual transit sampling algorithm. Parameters p and q can be returned by a user-defined function or added as constants. `next` performs rejection sampling (`rejection-smpl`), the details of which are discussed in [38]. `stepTransits` returns the vertex added at previous step. `sampleSize` returns 1 because we add only one neighbor of transit at each step. `steps` returns the length of walk, i.e., 100.

k-hop neighbors Figure 4d implements GraphSAGE's 2-hop neighborhood sampler in NextDoor. `stepTransits` returns the vertices added at previous step as transits. `next` retrieves the transit vertex for this sample in the `transit` variable, and chooses a neighbor of the transit vertex. Since this is a 2-hop sampling, `steps` returns 2. GraphSAGE [13] sets the number of neighbors as $m_1 = 10$ and $m_2 = 25$, as reflected in `sampleSize`. MVS [7] is implemented in a similar way as it obtains 1-hop neighbors of all initial vertices in the sample.

MultiDimensional Random Walk Figure 4c implements this sampling in NextDoor. At each step, `stepTransits` returns the transit vertex of a sample by choosing one of the

```
Vertex next(s, transits, srcEdges, step) {
  Vertex t = s.prevVertex(2,0);
  Vector<Edge> tEdges = s.prevEdges(2,0);
  float p = 2.0, q = 0.5;
  float maxW = transits[0].maxEdgeWeight();
  return rejection-smpl (transits[0],
        srcEdges, maxW, t, tEdges, p, q);}
int steps() {return 100;}
int sampleSize(step) {return 1;}
bool unique(int step) {return false;}
SamplingType samplingType()
{return SamplingType::Individual;}
Vertex stepTransits(step, s, transitIdx)
{return s.prevVertex(1, transitIdx);}
```

(a) node2vec random walk of length 100

```
Vertex next(s, transits, srcEdges, step) {
  Vertex v = randInt(0, graph.vertices());
  for (auto trn : transits)
    if (trn.hasEdge(v))
      s.addEdge(step, trn, v);
  return v;}
int steps() {return 5;}
int sampleSize(int step) {return 64;}
bool unique(int step) {return false;}
SamplingType samplingType()
{return SamplingType::Collective;}
Vertex stepTransits(step, s, transitIdx)
{return s.prevVertex(1, transitIdx);}
```

(b) Importance Sampling

```
Vertex next(s, transits, srcEdges, step) {
  int idx = randInt(0, srcEdges.size());
  Vertex v = srcEdges[idx];
  s.roots.replace(transits[0], v);
  return v;}
int steps() {return 100;}
int sampleSize(step) {return 1;}
bool unique(int step) {return false;}
SamplingType samplingType()
{return SamplingType::Individual;}
Vertex stepTransits(step, s, transitIdx)
{return s.roots[randInt(0,s.numRoots())];}
```

(c) Multi Dimensional Random Walk of length 100

```
Vertex next(s, transits, srcEdges, step) {
  int idx = randInt(0, srcEdges.size());
  return srcEdges[idx];}
int steps() {return 2;}
int sampleSize(int step) {
  return (step == 0) ? 25 : 10;}
bool unique(int step) {return false;}
SamplingType samplingType()
{return SamplingType::Individual;}
Vertex stepTransits(step, s, transitIdx)
{return s.prevVertex(1, transitIdx);}
```

(d) GrapSAGE's 2-hop neighbors

Figure 4. Use Cases of NextDoor

root vertices of the sample randomly and next samples a neighbor and replaces the root vertex with this neighbor.

Importance Sampling In FastGCN [3] and LADIES [44] every sample includes an adjacency matrix that records the edges between vertices added in the previous step (the transit vertices) and the current step. At each step i, m_i vertices are sampled from the graph according to a probability distribution and these vertices are added to the sample. Figure 4b implements importance sampling as a collective transit sampling as returned by samplingType. At each step 64 vertices are sampled. next chooses a vertex of graph according to a probability distribution and adds an edge between the vertex and all transits, if it exists. The sampling criteria in [3, 44] can be added in next. stepTransits returns the vertices sampled in the last step.

Cluster Sampling ClusterGCN [6] sampling obtains an adjacency matrix between all vertices of one or more clusters. Figure 4b sketches the implementation where at each step an edge is recorded in a sample's adjacency matrix if the edge exists between any two transits.

5 Paradigms for Graph Sampling on GPUs

This section presents two paradigms for parallel graph sampling. Existing systems for graph sampling [3, 6, 7, 13, 38, 40, 44] use *sample parallellism*. We discuss its shortcomings on GPUs and propose an alternative *transit parallel* paradigm.

5.1 Sample-Parallelism

Graph sampling is an "embarrassingly parallel" problem and the natural approach to parallelization is to process each sample in parallel, which we call the *sample parallel paradigm*. The approach is analogous to subgraph parallel expansion in graph mining systems [5, 23, 35]. We now discuss how to apply this approach to NextDoor's applications, performing both individual and collective transit sampling.

Individual Transit Sampling The naive way to implement sample parallelism for individual transit sampling on GPUs is to assign a thread to each sample but this limits the amount of parallelism to the number of samples. NextDoor's API enables a new, fine-grained approach to sample parallelism. At each step i, we assign consecutive m_i threads to a pair of sample and transit. Each thread then calls the user-defined function (next) on its transit. Since consecutive threads add vertices to the same sample, this approach allows writes to *global memory* to be coalesced. The algorithm visits samples and their transits in parallel. Figure 5a shows an example of sample parallel execution for the second step of Figure 2b. In this example, each sample is assigned to a thread block containing four threads. Each thread samples one vertex for the assigned transit and writes this vertex to the output.

Collective Transit Sampling For collective transit sampling, before calling the `next` function on the combined neighborhood of all transits, we need to compute this neighborhood. In a sample parallel approach, the combined neighborhood is computed in the same way as the individual transit sampling is performed. Consecutive threads are assigned to each pair of sample and transit, such that these threads copy the neighbors of one transit to the combined neighborhood, which is stored in global memory. After computing the combined neighborhood, we assign each pair of sample and transit to m_i consecutive threads and call `next` on this neighborhood.

Limitations Despite the finer-grained approach enabled by the API, sample parallelism makes poor use of the GPU for the following reasons. 1) In an individual transit sampling, for each sample, the algorithm calls `next` on the neighbors of several transit vertices in parallel. However, if two threads in a warp are assigned to process two distinct transit vertices with different numbers of neighbors, the thread processing the smaller set of neighbors may stall until the other thread completes. Thus the algorithm suffers from warp divergence. Similarly, there is warp divergence when computing the combined neighborhood. 2) The algorithm also suffers from poor load balancing. The amount of work done by `next` is likely to depend on the number of neighbors of the transit vertex. For example, while computing combined neighborhood in collective transit sampling, different number of neighbors of each transit vertex leads to load imbalance within a thread block. 3) The graph must be stored in global memory, so accessing neighbors of transit vertex incurs high latency. Moreover, threads in a block may access the neighbors of different transit vertices, which leads to no locality. Hence, the GPU cannot coalesce reads or cache neighbors in shared memory.

For example, the execution in Figure 5a suffers from both of the above issues. Since all four threads do not process same transit, there is divergent control flow and the adjacency list must be stored in global memory, leading to lack of locality among all threads of a thread block.

5.2 Transit-Parallelism

To overcome the limitations of sample parallelism, we present the *transit parallel* paradigm. Transit parallelism groups all samples with the same transit vertex and process all samples for one transit vertex by assigning these samples to consecutive threads. This approach exposes regularity in sampling.

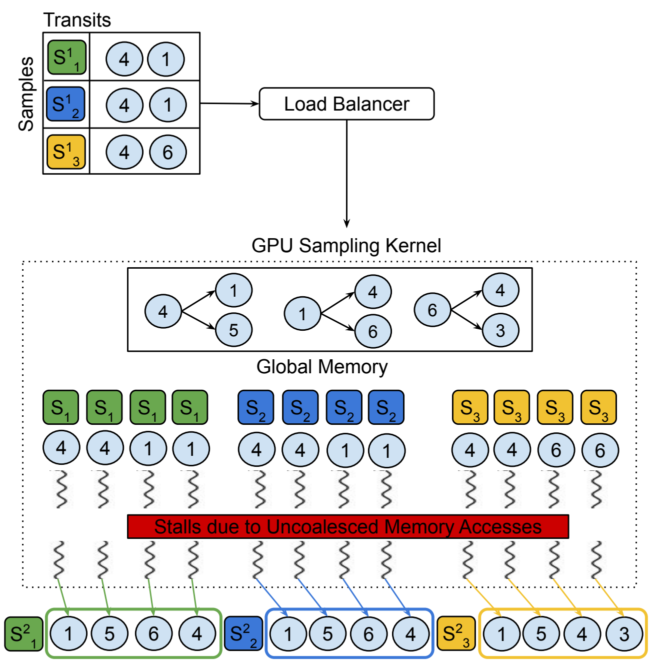
At each step the transit parallel paradigm works as follows. Before running sampling on the GPU, we create a map of transit vertices to their samples by grouping all samples associated with same transit vertex. We assign each transit vertex to a group of threads, which may be organized as a grid, thread block, or warp. In individual transit sampling, we assign each sample to consecutive threads in the group, and each thread calls `next` to add one neighbor of the transit to its sample. Similarly, in collective transit sampling, we create the combined neighborhood of the transits by assigning each sample to consecutive threads in the transit group (grid, thread block, or warp), and consecutive threads in the group add neighbors of the transit to the combined neighborhood of the sample. Building the combined neighborhood in a sample-parallel manner takes a significant portion of execution time. NextDoor speeds up this step by using the transit parallel approach. In this case, instead of sampling new vertices from the neighborhood of each transit of the sample using `next`, the system adds the entire neighborhood to the combined neighborhood. Collective sampling applications then select new vertices from the combined neighborhood per sample.

Figure 5b shows the execution of the second step of 2-hop neighbor sampling of Figure 2a in NextDoor using the transit parallel paradigm. Using load balancing (Section 6), NextDoor assigns transit ④ to a grid, such that all thread blocks in that grid are assigned samples of ④ and each thread adds one neighbor of ④ to one sample of ④, i.e., either S_1, S_2, or S_3. Similarly, it assigns vertex ① to a thread block and each thread adds one neighbor of ① to one of the samples associated with ①.

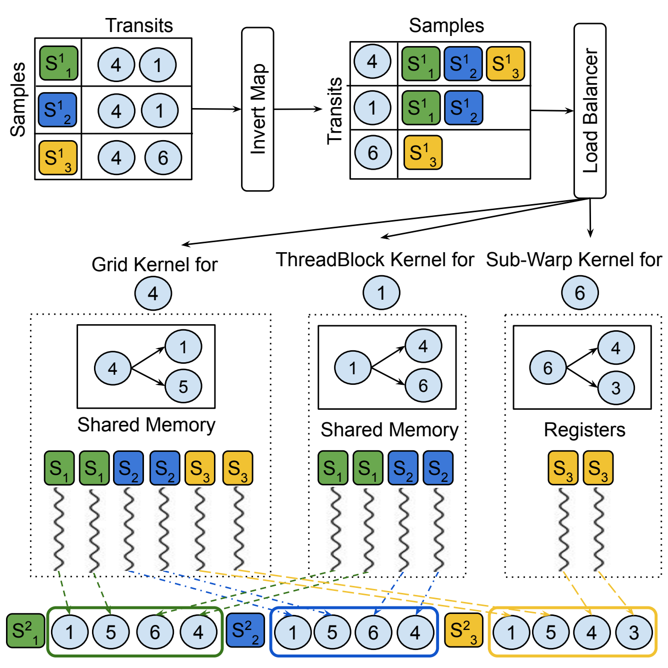
Advantages The transit parallel paradigm has two advantages for both individual and collective transit sampling: 1) contiguous threads perform a similar amount of work, because each thread calls the next function with same neighbors, and 2) contiguous threads accesses the neighbors of same transit. Both ensure non-divergent control flow, and locality of memory accesses. This eliminates warp divergence and addresses load balancing [2]. Moreover, since all threads work with same neighbors, we can cache the neighbors in shared memory to speed up later accesses.

For example, in the execution shown in Figure 5b each transit is assigned to one group of threads and this group caches the neighbors of each transit vertex in either shared memory or registers. Furthermore, each thread calls the *next* function on the same set of neighbors, which ensures non-divergent control flow among contiguous threads.

[2]A badly-written user-defined function may have these issues, but NextDoor avoids them in the core algorithm.



(a) Sample Parallel execution of second step. Each pair of sample and transit is assigned to consecutive threads. Each thread writes one vertex to the sample. Since contiguous threads can access different transit vertices, we have divergent control flow and no locality.



(b) Transit Parallel execution of second step. Each transit vertex is assigned to a group of threads: Grid, Thread Block, or Warp. Each thread writes a neighbor of the transit vertex to the sample achieving non-divergent control flow and locality.

Figure 5. Execution of second step for 2-hop neighbor sampling of Figure 2b using Sample Parallelism and Transit Parallelism.

6 Efficient Transit Parallelism on GPUs

NextDoor implements transit parallelism on a GPU, with CPU-based coordination. We first describe the techniques that allow NextDoor to execute individual transit sampling applications efficiently. We then describe how NextDoor uses the same techniques to execute collective transit sampling applications.

6.1 Sampling in Individual Transit Sampling

In this section, we describe how NextDoor executes individual transit applications using transit parallelism.

6.1.1 Leveraging Warp-Level Parallelism. A GPU can *coalesce* several global memory accesses together into one memory transaction only if threads in a warp access consecutive addresses. The transit parallel paradigm lends itself to a GPU implementation that supports coalescing *reads* to global memory, by having consecutive threads read the same adjacency list (i.e., of the shared transit vertex).

However, coalescing *writes* of new vertices to samples requires extra care. A two-level transit parallel approach maps different transit vertices to thread blocks and different samples to threads. This does not result in coalesced writes, since threads in the same warp add vertices to different samples. Instead, NextDoor uses three levels of parallelism: transits to thread blocks, samples to warps, and a single execution of the `next` function to a thread. Thus each thread writes one vertex to its sample and all threads in the warp issue one coalesced write to the same sample. Figure 5b shows this mapping as follows. First, transits ④, ①, and ⑥ are mapped to a group of threads. Then, samples (S_1, S_2, S_3) are mapped to subwarps and each thread executes `next`.

Sub-warps In an ideal scenario, there would be a one-to-one relationship between warps and samples, which would ensure that each thread in a warp writes to the same sample, using a single coalesced transaction to the global memory. However, there is a fixed number of threads per warp (usually 32) and this number can sometimes be larger than the required number of executions of the `next` function. Instead of letting threads be idle, NextDoor shares a warp among several samples. This yields some advantages. Suppose we share a warp of 32 threads among 4 samples, each having 8 contiguous threads. Then writes to the samples only generate 4 memory transactions rather than the 32 that we would obtain by assigning each thread to different sample. This also does not lead to warp divergence because all threads in a warp sample neighbors of the same transit vertex.

We use the term *sub-warp* to refer to a set of contiguous threads of same warp assigned to same sample. NextDoor uses sub warps as a fundamental unit of resource scheduling. All sub warps have the same size, which is determined using `sampleSize` function for the current step. Threads of the same sub-warp share the information of their registers using *warp shuffles*, and coordinate using *syncwarp*.

6.1.2 Load Balancing.

In the transit parallel paradigm, each transit vertex is associated with a set of samples, which varies among transit vertices and steps. With three levels of parallelism, a transit vertex requires as many threads in a step as the total number of neighbors that will be added to its samples. Thus we obtain sub-optimal performance if we always assign a single thread block to each transit vertex. At the limit, the number of threads required by a transit vertex may exceed the limit of number of threads in a block. On the other hand, if a transit vertex only requires a small number of threads, dedicating an entire thread block to the transit wastes GPU resources. To address this problem, NextDoor uses three types of GPU kernel (see Table 2):

1. The *sub-warp* kernel processes several transit vertices in a single warp. It is only applicable to transit vertices that require fewer threads than the warp size (32).
2. The *thread block* kernel dedicates a thread block to a single transit vertex. It is only applicable to transit vertices that require more threads than in a warp, but less than the maximum thread block size (1,024).
3. The *grid* kernel processes a single transit vertex in several thread blocks. It is only applicable to transit vertex requires more than 1,024 threads.

Scheduling To assign transits to kernels, NextDoor creates a *scheduling index* for each transit vertex. Creating a scheduling index involves three stages. First, NextDoor creates a transit-to-sample map based on the transits obtained from the *stepTransits* function (Figure 5b). Then, NextDoor partitions all transit vertices into three sets based on the number of samples associated with each transit vertex using parallel scan operations. Finally, the scheduling index of a transit vertex is set to the index of the transit vertex in its set. After picking a kernel type for a transit vertex, we assign each sample of the transit vertex to a sub-warp in the kernel based on the thread index.

Caching NextDoor uses different caching strategies for different kernels to minimize memory access costs. When sampling neighbors of transit vertices in the *grid* and *thread block* kernels, the thread blocks for these kernels load the neighbors of transit vertices into shared memory. However, when the neighbors do not fit in shared memory, NextDoor transparently loads neighbors from global memory. For transit vertices assigned to a *sub-warp*, NextDoor utilizes both shared memory and thread local registers to store neighbors. In this case, NextDoor transparently manages accesses to the neighbor list using *warp shuffle* instructions that allows consecutive threads to read neighbors from each others' registers. In summary, NextDoor uses the fastest caching mechanisms available for each kernel.

6.2 Transit-Parallel Collective Transit Sampling

Collective transit sampling applications require computing the combined neighborhood of all the transits of each sample. This is a potential performance bottleneck, so NextDoor uses transit parallelism to speed up the process. It constructs the combined neighborhood as if it were an individual transit sampling application that runs for only one step. Instead of sampling new vertices from the neighborhood of one transit, NextDoor adds all the vertices in the neighborhood to the combined neighborhood of the sample. After building a single combined transit neighborhood per sample, one could in principle detect which samples have the same combined neighborhoods and expand all these samples in a transit-parallel manner. The likelihood of two combined neighborhood being equal, however, is generally low, and detecting which samples have the same combined neighborhood is expensive. Therefore, NextDoor adds new vertices to the sample using a sample-parallel approach.

6.3 Unique Neighbors

Certain applications, require all sampled neighbors from all transit vertices to be unique. After sampling at each step NextDoor removes duplicated sampled vertices by first sorting them with a parallel radix sort, and then getting distinct vertices using parallel scan. If sampled neighbors fit in the shared memory then NextDoor performs this computation by assigning one sample to one thread block, otherwise one kernel is called for each sample. After this process if the sample size is less than the `stepSize`, then NextDoor performs sampling using a sample-parallel approach instead of the transit-parallel one.

6.4 Graph Sampling using Multiple GPUs

NextDoor utilizes the embarrassingly parallel nature of graph sampling to parallelize graph sampling over multiple GPUs in the following way. First NextDoor distributes samples equally among all GPUs. Then, NextDoor performs load balancing and scheduling and calls the sampling kernels on each GPU independently. Finally, it collects the output from all GPUs.

6.5 Integration in GNNs using Python API

NextDoor provides Python 2 and 3 modules that can be used to do sampling from within a GNN. For this, users first define NextDoor API functions, then call `doSampling` function to do transit-parallel sampling, and finally call `getFinalSamples` to obtain samples in a `numpy.ndarray`.

6.6 Advantages of NextDoor's API

Expressing a graph sampling application using NextDoor's API provides several advantages. It describes sampling operations in a fine-grained manner, which enables using the execution hierarchy of the GPU more efficiently with both

Kernel	Total neighbors to sample	Caching Strategy	Neighbor Access Strategy	Transit Scheduling Strategy
Grid	Greater Than 1024	Shared Memory	Memory Loads	One transit to many thread blocks
Thread Block	Between 32 and 1024	Shared Memory	Memory Loads	One transit to one thread block
Sub-Warp	Less than 32	Registers	Warp Shuffles	One transit to one sub-warp

Table 2. Types of kernels used to process transit vertices based on the number of neighbors to sample for the transit vertex.

sample and transit parallelism. NextDoor's API is general purpose and supports different kinds of graph sampling applications, while existing systems either running specific graph sampling applications [3, 6, 10, 13, 40, 44] or only support expressing specific kinds of graph sampling applications, as for example random walks for KnightKing [38]. Furthermore, NextDoor's API requires applications to explicitly indicate which vertices in each sample are transit vertices. This distinction is critical to enable transit parallelism. Finally, the API requires applications to explicitly state the number of vertices that must be sampled at each step. This information is used by NextDoor to effectively load balance computation across the execution hierarchy of the GPU.

7 Alternative Graph Processing Systems

Abstractions provided by existing GPU based graph processing systems to implement graph algorithms can be divided into two types: (i) Message Passing and (ii) Frontier Centric. We now study the implementation of graph sampling using both abstractions and discuss why these implementations provides sub-optimal performance.

Message-passing Abstraction is provided by several graph computation frameworks [9, 17, 22, 26, 31], where each vertex is associated with a local state and a vertex can send messages to its neighbors. Upon receiving messages, vertices update their state and can send new messages. Graph computations written in this abstraction advances the computation by exchanging messages between vertices at each step.

A transit-parallel approach for graph sampling implemented using message passing works in the following way. First, in each step for each sample associated with a transit, neighbors of the transit are sampled. Then, the `stepTransits` function is called to retrieve transit for next step and the associated samples are send to the transit in the form of messages. Each transit vertex is associated with only one thread, which processes all its samples sequentially.

Frontier-centric Abstraction is provided by Gunrock [37], which exploits the property that after any step of a graph computation, a set of *frontier* vertices are produced for the next step of the computation. The Advance operator in this abstraction defines the computation and generates a new frontier by assigning one thread to each neighbor of each vertex in the input frontier.

A transit-parallel approach for graph sampling can be implemented in this abstraction as follows way. The Advance operator contains the user-defined sampling criteria, which is invoked on each neighbor of the transit vertex. This operator will decide whether the neighbor should be added to the sample. In that case, `stepTransits` is called to retrieve the transit vertex for the sample and the transit vertex is added to the new frontier. Each thread for a neighbor must make this decision for all the associated samples, which are processed sequentially.

Limitations over NextDoor Graph processing systems providing above abstractions suffers from two fundamental issues. First, these systems only consider one degree of parallelism, i.e., all transit vertices can be processed in parallel but samples for each transit are processed sequentially. This is because these systems are designed for traditional graph computations, such as Breadth First Search, Connected Components, etc., which only has one degree of parallelism. Second, these systems balances the load based on the number of neighbors for each vertex because all neighbors of each vertex are visited in a traditional graph processing application. But, in a graph sampling number of neighbors to sample at a step can be significantly less than the neighbors of a transit vertex. Hence, NextDoor takes a different approach from existing system to solve these issues. Section 8.3 shows that NextDoor performs better than these systems.

8 Evaluation

We implemented NextDoor in NVIDIA CUDA 11.2.

Benchmarks We use the graph sampling applications mentioned in Section 4 as benchmarks for our evaluation. We set applications' parameters as follows. For *PPR* the termination probability is set to 1/100, i.e., mean length is 100. For all other random walks, we set the walk length to 100. For *node2vec* we set p to 2.0 and q to 0.5. For these random walks, initially there is one vertex per sample. For *MultiDimensional Random Walk* (MultiRW), we set 100 root vertices per sample. We use GraphSAGE [13]'s hyperparameters for *k-hop Neighborhood Sampling*, i.e., $k = 2$, $m_1 = 25$, and $m_2 = 10$. For *Layer Sampling* we set final sample size to 2000 and step size for all steps to 1000. For *FastGCN*, *LADIES*, and *MVS* Sampling batch size and step size are set to 64. For *ClusterGCN Sampling* we randomly assigned vertices in clusters and each sample contains 20 clusters.

Datasets Table 3 lists the details of real world graphs used in our evaluation obtained from Stanford Network Analysis Project [18]. We generate a weighted version of these graphs by assigning weights to each edge randomly from [1, 5).

Name	Abrv	# of Nodes	# of Edges	Avg Degree
Protein-Protein Interactions	PPI	50K	1.4M	28.0
com-Orkut	Orkut	3M	117M	39.0
cit-Patents	Patents	3.77M	16.5M	4.37
soc-LiveJournal1	LiveJ	4.8M	68.9M	14.3
com-Friendster	FriendS	65.6M	1.8B	27.4

Table 3. Graph used in our evaluation.

Experimental setup We perform experiments on a system containing two 16-core Intel Xeon(R) Silver 4216 CPU, 128 GB RAM, and an NVIDIA Tesla V100 GPU with 16GB memory running Ubuntu 18.04. We report the average time of 10 executions. We report the execution time spent on the GPU, which includes the time spent in sampling and creating the scheduling index. Since transferring graphs to the GPU that fits inside the GPU memory takes only few milliseconds (less than 5% of total execution time), we do not consider these times in the total execution time unless specified otherwise.

8.1 Execution Time Breakdown

The execution time of an application in NextDoor consists of the time spent in sampling and creating the *scheduling index.* NextDoor builds the scheduling index by sorting the samples based on the neighbors in each sample as keys and then dividing the transit vertices into three sets based on the number of samples for each transit, using parallel scan. Figure 6 shows the time spent in both phases as a fraction of the total execution time. The time spent in building scheduling index ranges from 5% of the total time in ClusterGCN for sampling LiveJ graph to 40.4% of the total time in DeepWalk for sampling Orkut graph. Random walks spend a higher fraction of time building the scheduling index. This is because they sample only a single vertex per step, leading to fewer common samples and less work per transit than other applications. NextDoor uses parallel radix sort and parallel scan of NVIDIA CUB [1] to create the scheduling index efficiently. With more efficient implementations of these algorithms [34] available for GPUs, we expect this time to decrease significantly in future.

8.2 Graph Sampling Performance

We compare NextDoor with the following systems.

SP NextDoor is the first system for graph sampling on GPUs. Since we cannot compare it with other systems, we implemented an optimized sample-parallel graph sampling system based on the NextDoor API, which we refer to as *SP*. We implemented all the optimizations of NextDoor that could be adapted to a sample-parallel system, such as load balancing, scheduling, and the fine-grained parallelism discussed in Section 5.1. The purpose of SP is to evaluate the benefits of transit-parallelism in isolation, without considering all the other optimizations enabled by the new API.

Since TensorFlow based reference implementation of layer sampling [10] does not support the datasets used in evaluation and the implementation follows sample-parallel paradigm, we use SP as a baseline in layer sampling.

TP To show the performance improvement due to NextDoor's load balancing and scheduling optimizations described in Section 6, we compare against vanilla transit-parallel approach (see Section 5.2), which assigns each transit and sample pair to m_i consecutive threads. We refer to this implementation as *TP*.

KnightKing KnightKing [38] is a state of the art system for doing random walks using CPUs. It uses rejection sampling as a technique to select new vertices of a random walk and supports sampling using distributed systems. Its API restricts expressing only random walks, hence, we use the system as a baseline only for random walks.

Existing GNN Samplers We compare against the samplers of existing GNNs. These samplers are written for TensorFlow or numpy and are designed to run only on multi-core CPUs, not GPUs. This is because sampling is an irregular computation that is more easily implemented on CPUs. For k-hop neighborhood, we compare against GraphSAGE's sampler [13]. For MultiRW, we compare against GraphSAINT's sampler [40]. For sampling algorithms in FastGCN [3], ClusterGCN [6], MVS [7], and LADIES [44], we compare against samplers in their reference implementations.

Performance Results NextDoor provides an order of magnitude speedup over KnightKing (Figure 7a) for all random walk applications, with speedups ranging from 26.1× to 50×. NextDoor provides an order of magnitude speedup over the implementations of existing GNNs (Figure 7b). These large speedups are possible due to the massive parallelism and memory access latency hiding capabilities of the GPU. Furthermore, SP is significantly faster than all baselines.

NextDoor provides significant speedups over SP on all graph sampling applications, with speedups ranging from 1.09× to 6×. The speedup depends significantly on the application. For example, NextDoor obtains more speedup in DeepWalk and PPR than in node2vec because in node2vec at each step, for an edge from current transit vertex v to a vertex u, the algorithm might do a search over the edges of the previous transit vertex t to check if u is a neighbor of t, leading to memory accesses and warp divergence. Nevertheless, NextDoor still obtains speedup due to its transit-parallel paradigm. NextDoor achieves speedup over SP in all applications because NextDoor uses three levels of parallelism while SP can use only two levels of parallelism. Moreover, with FastGCN and LADIES, NextDoor is faster because it speeds up the computation of the combined neighborhood.

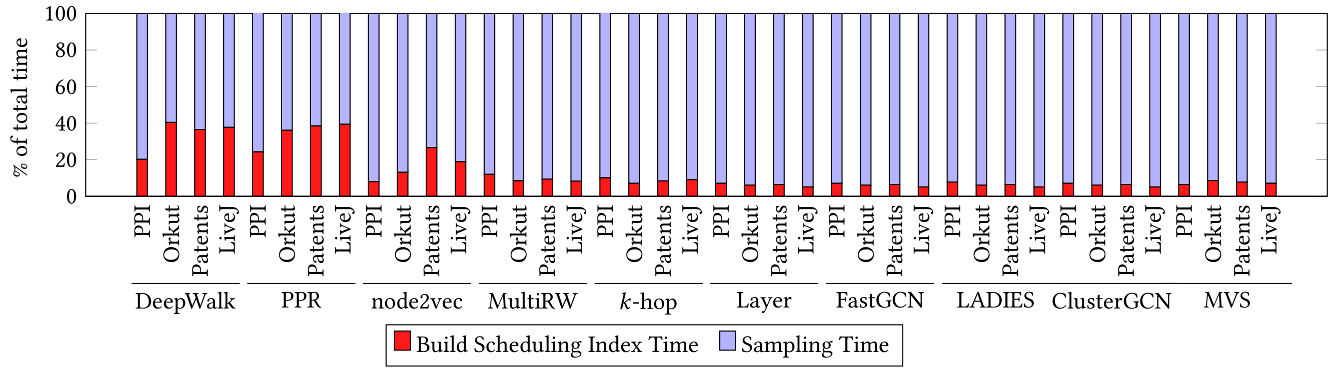
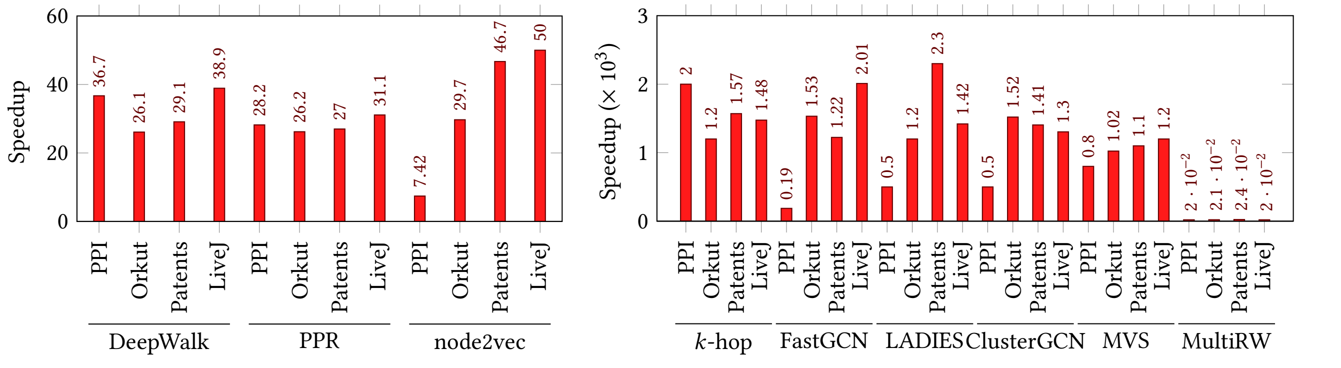
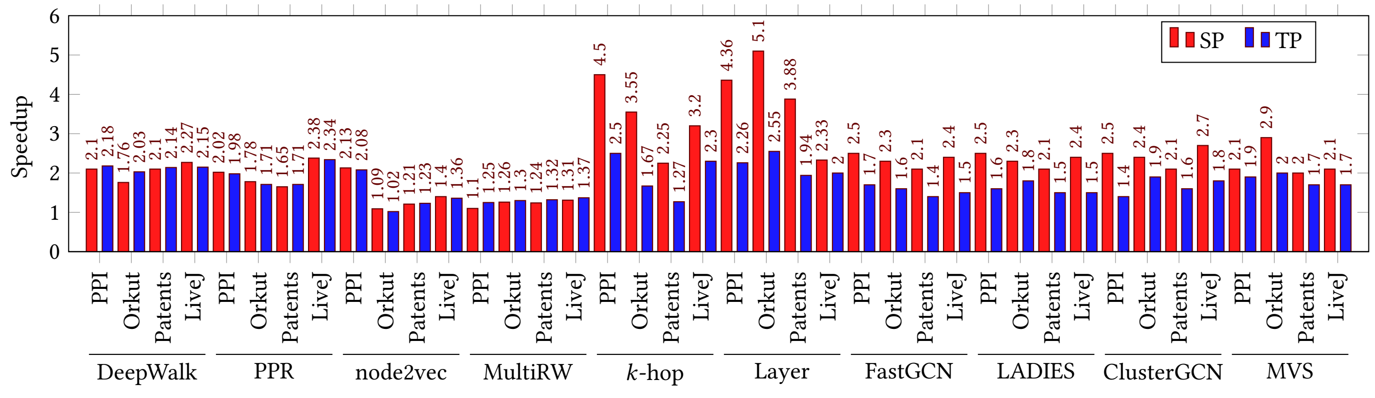


Figure 6. Percentage of time spent in Sampling the graph and building Scheduling Index to the total time in NextDoor



(a) Speedup of on random walks over KnightKing. **(b)** Speedup on sampling applications over their GNN implementation ($\times 10^3$).



(c) Speedup on graph sampling applications over SP and TP.

Figure 7. Speedup of NextDoor on graph sampling applications and real world graphs over baselines.

NextDoor significantly improves performance over TP due to better load balancing and scheduling. TP is competitive to SP in random walks even though significant time is spent in map inversion because TP caches the neighbors in shared memory. TP outperforms SP in other applications because caching neighbors in shared memory decreases memory access time when sampling many neighbors.

8.2.1 NextDoor's Effectiveness over SP. To explain NextDoor's effectiveness over SP, we obtained values of L2 Cache Read Transaction performance metrics using `nvprof`. This metric represents the total number of L2 cache load transactions in the entire execution. Figure 8 shows the value of this metrics for NextDoor relative to SP. NextDoor performs a fraction of the transactions of SP because it performs coalesced reads and caches edges of transit vertices in shared memory and registers. ClusterGCN, MVS, FastGCN, and LADIES sampling perform a similar number of loads and stores as *k*-hop and Layer sampling.

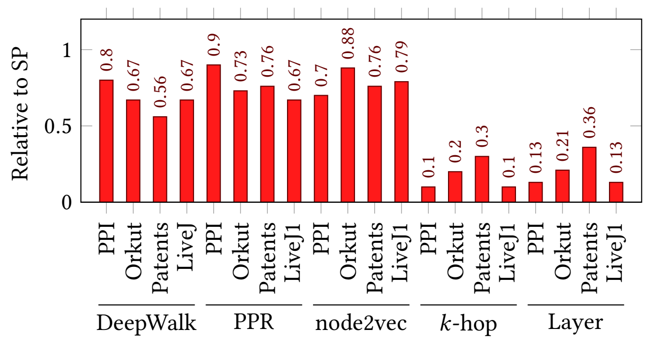


Figure 8. Value of different L2 Cache Load Transactions metric for several applications in NextDoor relative to SP.

Dataset	Store Efficiency(%)		Multiprocessor Activity(%)				
	k-hop	Layer	DW	PPR	n2v	*k*-hop	Layer
PPI	98.5	98.5	67.8	69.8	70.1	100	100
Orkut	99.5	100	98.3	98.0	99.3	100	100
Patents	100	100	90.1	99.0	99.0	100	100
LiveJ	100	100	99.2	98.2	97.6	100	100

Table 4. Global Memory Store Efficiency and Multiprocessor Activity in NextDoor. (DW is short for DeepWalk and n2v is short for node2vec)

8.2.2 NextDoor's Efficiency. We present absolute values of two performance metrics in Table 4 obtained using `nvprof`: (i) *Global Memory Store Efficiency* to show NextDoor's effectiveness to do efficient global stores, and (ii) *Multiprocessor Activity* to show NextDoor's effectiveness in fully utilizing GPU's execution resources.

Global Memory Store Efficiency is the ratio of extra store transactions over the ideal number of transactions to the ideal number of transactions. Hence, higher efficiency is better. NextDoor performs fully efficient global memory stores because of the sub-warp execution. Since ClusterGCN, MVS, FastGCN, and LADIES sampling perform number of loads and stores similar to *k*-hop and Layer sampling, we found similar store efficiency for these applications.

Multiprocessor Activity is the average usage of all SMs over the entire execution of the application. For PPI, Multiprocessor Activity is low because PPI is a small graph and not enough threads are generated to fully utilize all SMs. For all graphs NextDoor fully utilizes all SMs. Hence, NextDoor's load balancing strategy balances load across all SMs. We found similar results for other sampling applications.

8.3 Alternative GPU-Based Abstractions

We also compare NextDoor with two state-of-the-art graph processing frameworks: Gunrock [37] and Tigr [26], which follow the frontier-centric and message-passing abstractions, respectively (see Section 7). Figure 9 reports the speedup of NextDoor. As explained in Section 7, low parallelism and

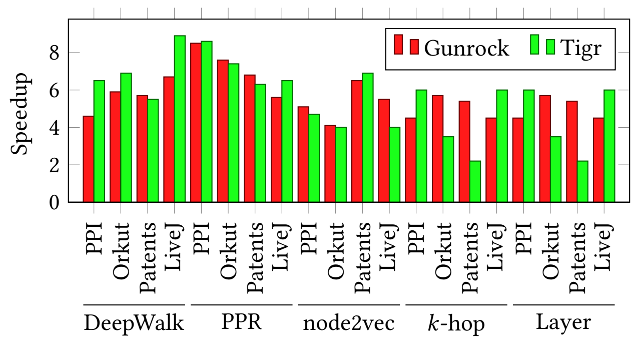


Figure 9. Speedup of NextDoor over Tigr and Gunrock on some graph sampling applications and real world graphs.

poor load balancing due to the mismatch between graph sampling and graph processing abstraction result in speedup. We found similar results on other applications.

8.4 Sampling Large Graphs

We evaluate a simple approach for sampling large graphs.

NextDoor can sample graphs that do not fit in GPU memory by creating disjoint sub-graphs, such that each of these sub-graphs and its samples be allocated in the GPU memory. After creating these sub-graphs at each computation step, NextDoor performs sampling for each sample by transferring all sub-graphs containing the transit vertices of each sample to the GPU. In this experiment, we consider the time taken to transfer graph from CPU to GPU.

We evaluate this approach by executing *k*-hop and random walks on the FriendS graph, which does not fit in the GPU memory. For *k*-hop neighborhood and layer sampling, NextDoor is the only system in our experiments that can sample a graph of that size. NextDoor gives a throughput of 3.3×10^6 samples per second on *k*-hop and 2×10^6 on layer sampling. Both applications are computation bound and not memory transfer bound. For random walks, KnightKing is the only baseline that can perform the sampling because it is CPU based. NextDoor performs worse than KnightKing for random walks where the computation load is low: it provides about 1/2 of the throughput with DeepWalk and PPR. However, in node2vec where the computation time is larger, NextDoor gives 1.50× speedup over KnightKing.

In summary, NextDoor is able to sample graphs that do not fit in GPU memory, and can outperform state-of-the-art systems when the graph sampling application performs significant amount of computation. We plan to improve the support for large graphs in NextDoor as future work.

8.5 Sampling on Multiple GPUs

We used NextDoor to perform sampling on four NVIDIA Tesla V100 GPUs. Figure 10 presents the speedup of sampling using four GPUs over single GPU. Multi-GPU sampling

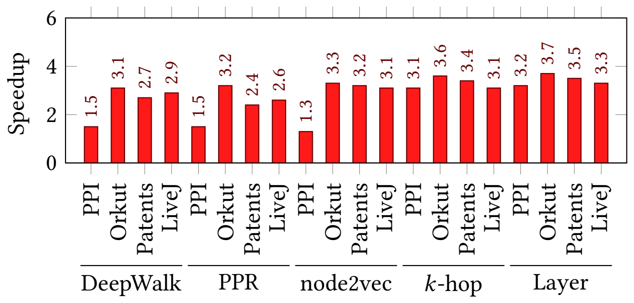


Figure 10. Speedup of sampling using 4 GPUs over 1 GPU.

	PPI	Reddit	Orkut	Patents	LiveJ
GraphSAGE	1.30×	1.21×	OOM	1.20×	1.22×
FastGCN	1.25×	1.52×	4.75×	2.3×	4.31×
LADIES	1.07×	1.37×	2.27×	2.1×	2.34×
ClusterGCN	1.03×	1.20×	OOM	1.4×	1.51×

Table 5. End-to-end speedups after integrating NextDoor in GNNs over vanilla GNNs.

achieves significant speedup over single GPU on several applications. Random walks achieves significant speedup in all graphs except PPI because PPI is a small graph. On the other hand, k-hop neighbors achieves almost full scaling even in small graph like PPI because it increases the number of transit vertices exponentially at each step. In summary, NextDoor is able to utilize multiple GPUs efficiently.

8.6 End-to-End Integration in GNN Systems

We performed an end-to-end evaluation of existing GNNs by replacing their sampler with the sampling implementation in NextDoor. Table 5 shows the performance improvement of our integration. The speedup for GraphSAGE is less than the maximum possible improvement in Table 1 due to a limitation of Tensorflow, which does not allow creating a tensor on the GPU memory. Therefore, samples are copied to the CPU and then again to the GPU for training. For FastGCN and LADIES, the speedup increases with larger graphs because the sampling time depends on the number of vertices in the graph, while the training time per batch remains constant. Since NextDoor provides significant speedup over samplers in GraphSAINT and MVS, we believe NextDoor integration will improve training time.

9 Related Work

We now discuss related work beyond KnightKing, Gunrock, and Tigr, which we discussed in Sections 7 and 8.

Message-passing graph processing There are several graph processing systems that provide a message-passing abstraction that run on CPUs [11, 20, 22, 25, 33, 42, 43] and GPUs [9, 17, 26, 31, 41]. Our evaluation shows that NextDoor outperforms Tigr [26] on graph sampling tasks (Section 8). Medusa [41] was the first GPU-based graph processing framework to provide a message passing abstraction. CuSha [17] and MapGraph [9] provide a Gather And Scatter (GAS) abstraction. CuSha uses a parallel sliding-window graph representation ("G-Shards") to avoid irregular memory accesses. Subway [31] splits the large graphs that do not fit in GPU memory into sub-graphs and optimizes memory transfers between CPU and GPU. Shi et al [32] present an extensive review of systems for graph processing on GPUs. PowerLyra [4] uses different computations on vertices based on their degree.

Frontier-centric graph processing SIMD-X [19] extends the frontier abstraction of Gunrock [37], but these extensions are not relevant for graph sampling.

Graph mining Graph mining systems follow a subgraph-parallel paradigm that is analogous to sample-parallelism [2, 5, 8, 15, 23, 29, 35, 36]. However, even the sample-parallel sampling algorithm of Section 5 introduces optimizations that are specific to the graph sampling abstraction of Section 3 and do not generalize to graph mining problems. 1) In graph sampling the number of samples is fixed, whereas graph mining problem may involve exploring an exponential number of subgraphs. 2) sampling adds a constant number of new vertices to each sample at each step. This makes it possible to associate new vertices to threads at scheduling time, *before* visiting the graph. 3) Sampling has a notion of *transit* vertices. NextDoor leverages all these features.

10 Conclusion

We show that efficient graph sampling on GPUs is non-trivial. Existing sampling and graph processing systems do not provide the right abstractions to efficiently support several graph sampling algorithms on GPUs. We introduce *transit-parallel* sampling, a new paradigm for graph sampling that is amenable to an efficient GPU implementation. We present NextDoor, a system that implements transit-parallel sampling for GPUs to provide regular memory access and computation, and a high-level API to express several graph sampling applications. We show that NextDoor is significantly faster than the existing systems on several applications.

Acknowledgements. This work was partially supported by the National Science Foundation under grant CCF-2102288, a Facebook Systems for Machine Learning Award, and an AWS Cloud Credit for Research grant.

References

[1] Accessed in Feb 2021. NVIDIA CUB. https://nvlabs.github.io/cub/
[2] Hongzhi Chen, Xiaoxi Wang, Chenghuan Huang, Juncheng Fang, Yifan Hou, Changji Li, and James Cheng. 2019. Large Scale Graph Mining with G-Miner. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19).*
[3] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In

International Conference on Learning Representations (ICLR'18).

[4] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. *ACM Trans. Parallel Comput.* (2019).

[5] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *Proc. VLDB Endow.* (2020).

[6] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '19).*

[7] Weilin Cong, Rana Forsati, Mahmut Kandemir, and Mehrdad Mahdavi. 2020. Minimal Variance Sampling with Provable Guarantees for Fast Training of Graph Neural Networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '20).*

[8] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. 2019. Fractal: A General-Purpose Graph Pattern Mining System. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19).*

[9] Zhisong Fu, Michael Personick, and Bryan Thompson. 2014. Map-Graph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems (GRADES'14).*

[10] Hongyang Gao, Zhengyang Wang, and Shuiwang Ji. 2018. Large-Scale Learnable Graph Convolutional Networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD' 18).*

[11] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12).*

[12] Aditya Grover and Jure Leskovec. 2016. Node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16).*

[13] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17).*

[14] Taher H. Haveliwala. 2002. Topic-Sensitive PageRank. In *Proceedings of the 11th International Conference on World Wide Web (WWW '02).*

[15] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. 2018. ASAP: Fast, Approximate Graph Pattern Mining at Scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18).*

[16] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems* (2020).

[17] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: Vertex-Centric Graph Processing on GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '14).*

[18] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[19] Hang Liu and H. Howie Huang. 2019. SIMD-X: Programming and Processing of Graph Algorithms on GPUs. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19).*

[20] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. 2010. GraphLab: A New Framework for Parallel Machine Learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence (UAI'10).*

[21] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. Neugraph: parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19).*

[22] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10).*

[23] Daniel Mawhirter and Bo Wu. 2019. AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19).*

[24] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[25] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13).*

[26] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18).*

[27] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report. Stanford InfoLab.

[28] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '14).*

[29] Abdul Quamar, Amol Deshpande, and Jimmy Lin. 2016. NScale: neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal* (2016).

[30] Bruno Ribeiro and Don Towsley. 2010. Estimating and Sampling Graphs with Multidimensional Random Walks. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC '10).*

[31] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: Minimizing Data Transfer during out-of-GPU-Memory Graph Processing. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20).*

[32] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph processing on GPUs: A survey. *ACM Computing Surveys (CSUR)* (2018).

[33] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *SIGPLAN Not.* (2013).

[34] Elias Stehle and Hans-Arno Jacobsen. 2017. A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17).*

[35] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: A System for Distributed Graph Mining. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15).*

[36] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on a Single Machine. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18).*

[37] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. *SIGPLAN Not.* (2016).

[38] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. 2019. KnightKing: A Fast Distributed Graph Random

Walk Engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*.

[39] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '18)*.

[40] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *International Conference on Learning Representations (ICLR '20)*.

[41] J. Zhong and B. He. 2014. Medusa: Simplified Graph Processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* (2014).

[42] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[43] Youwei Zhuo, Jingji Chen, Qinyi Luo, Yanzhi Wang, Hailong Yang, Depei Qian, and Xuehai Qian. 2020. SympleGraph: Distributed Graph Processing with Precise Loop-Carried Dependency Guarantee. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*.

[44] Difan Zou, Ziniu Hu, Yewen Wang, Song Jiang, Yizhou Sun, and Quanquan Gu. 2019. Layer-Dependent Importance Sampling for Training Deep and Large Graph Convolutional Networks. In *Advances in Neural Information Processing Systems (Nuerips '19)*.