ScaleGPS: Scalable Graph Parallel Sampling via Data-centric Performance Engineering

by

Miranda J. Cai

B.S. Computer Science and Engineering, MIT, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

Authored by:	Miranda J. Cai Department of EECS May 20, 2024
Certified by:	Xuhao Chen Research Scientist, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

SCALEGPS: Scalable Graph Parallel Sampling via Data-centric Performance Engineering

by

Miranda J. Cai

Submitted to the Department of Electrical Engineering and Computer Science on May 20, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Graph sampling extracts representative samples of a graph, so that approximate graph algorithms can be used in place of expensive, exact algorithms while still achieving high-quality results. Thus, graph sampling plays an important role in many modern graph-based applications, such as graph machine learning and graph data mining. However, because of *unstructured sparsity* in the graph data and the *randomness* in the sampling algorithms, graph sampling often is the computational bottleneck. To accelerate it, there exist parallel graph sampling methods on multicore CPUs or GPUs. However, limitations arise at both sides. Due to lower throughput, CPU implementations are much slower than GPU ones, while GPU memory capacity is limited to only being able to handle small input graphs.

We present the idea behind a scalable graph sampling framework, SCALEGPS, to support high performance graph sampling on huge graphs in a single machine with a CPU and a GPU. The key idea is to cooperatively employ data *caching* and *compression* to reduce memory footprint and data movement overhead, and thus achieve high performance and scalability. The challenge in applying caching and compression for graph sampling is two-fold. First, the randomness in sampling leads to redundant computation and memory accesses, and thus low work efficiency. Second, real-world graphs often exhibit skewed degree distribution, where a fixed strategy cannot optimally handle various cases.

We propose a *hybrid* and *adaptive* strategy to address this challenge. First, we split the vertices in the graph into two groups based on their degrees. For each group, we store the neighbor lists in different formats, to make full use of the scarce GPU memory resources. Based on this hybrid compression method, we use the GPU memory as a cache of the CPU memory, and adaptively cache hot data to minimize the data movement overhead between the CPU and GPU. We implement our strategy in SCALEGPS and evaluate it on a single machine with a 48-core CPU and an A100 GPU. Our experimental results on various sampling algorithms show that SCALEGPS is able to support billion-edge graphs (up to 84-billion) in a single machine. While the performance benefits over these large graphs are still undetermined, SCALEGPS achieves an average of 33.4× (up to 93×) speedups for smaller graphs over state-of-the-art parallel CPU implementations.

Thesis supervisor: Xuhao Chen
Title: Research Scientist

Acknowledgments

I would like to thank Xuhao Chen for being an amazing supervisor throughout this process. I am grateful for everything that I have learned from this project under his guidance, and for him first accepting me into his lab.

Most importantly, I would like to thank my parents, brother, and grandparents for their continuous support of my studies. I would also like to show my appreciation for all of the friends that I have made along the way during my past four years here at MIT. I would not be where I am today without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graph representation has become an increasingly popular method to model and solve important computing problems, where data entities (vertices) are interconnected with relationships or inter-dependencies (edges). Real-world graph applications include product recommendation [1], social recommendation [2], [3], protein design [4], drug discovery [5], financial forensics [6]–[8], chemical engineering [9], anomaly (e.g. spam, fake news) detection [10], circuit design [11], etc. For many of these applications, they have to deal with massive scale datasets, e.g., giant graphs with hundreds of billions of edges, putting a lot of pressure on the computer system. As a result, efficiently scaling graph computation of massive sizes becomes a key challenge in this domain.

One widely used method to combat the high computational costs in processing massive graph data is *graph sampling*, which extracts a representative portion of the input graph data, in order to reduce the total amount of computation and storage consumption. Graph sampling has been widely used in graph machine learning [12]–[17] and graph pattern mining [18]–[48]. Although it significantly reduces the computation algorithmically, plain graph sampling alone is not enough to address the inefficiency in graph computation. Instead, because of the *unstructured sparsity* in the graph data and the *randomness* in the sampling algorithms, graph sampling often becomes the computational bottleneck in those applications.

For example, it has been observed that in GraphSAGE [14], a popular graph neural network (GNN) model, graph sampling ends up taking at least 45% of the model's end-to-end training time [49].

To accelerate graph sampling, parallel graph sampling methods on multicore CPUs [50]–[53] and accelerators like GPUs [49], [54]–[56] and FPGAs [57]–[60] have been proposed. However, all these approaches have their limitations. Due to lower throughput, CPU implementations are usually much slower than GPU solutions. On the other hand, accelerators can achieve impressive speedups with high throughput, but they are usually limited in memory capacity and thus are only able to handle small input graphs.

In this thesis, we aim to address the efficiency and scalability issue in graph sampling. We present a scalable graph sampling framework, ScaleGPS, to support high performance graph sampling on massive-scale graphs in a single machine with a CPU and a GPU. The key idea is to cooperatively employ data *caching* and *compression* to minimize data movement overhead and reduce memory footprint, at the cost of extra computation on decompressing the graph data. The rationale behind this design is that computation is cheap on GPU, but data movement is often the real bottleneck, and also memory space is a scarce resource on the GPU.

The challenge in applying caching and compression for graph sampling is two-fold. First, the randomness in sampling leads to redundant computation and memory accesses, as only a small portion of the data is sampled out of all the decoded data, which leads to wasted decoding computation and thus low work efficiency. Second, real-world graphs often exhibit skewed (e.g., power-law) degree distribution. Because of this skewness, any fixed compression or caching strategy may not be able to optimally handle various cases. So generally there is a tradeoff between data decoding speed, work efficiency and memory consumption.

To address the challenges, we propose a *hybrid compression and adaptive caching* (HCAC) strategy. First, we split the vertices in the graph between two groups based on their degrees: high-degree and low-degree. For each group, we store the neighbor lists in different formats.

We compress the neighbor lists of low-degree vertices using a SIMD-friendly compression scheme, while for high-degree ones we add auxiliary indexing to reduce decoding overhead. In addition to this hybrid compression method, we use the GPU memory as a cache of the CPU memory, and adaptively cache hot data in the GPU memory to minimize the data movement overhead between the CPU and GPU.

We implement our HCAC strategy in ScaleGPS and evaluate it on a single machine with a 48-core CPU and an A100 GPU. We test a representative graph sampling algorithm, k-hop neighborhood sampling. Our experimental results show that ScaleGPS can support fast sampling on billion-edge graphs (up to 84-billion) in a single machine. ScaleGPS achieves an average of 33.4$\times$ (up to 93$\times$) speedups for smaller graphs over state-of-the-art parallel CPU implementations and at most 3.9$\times$ slowdown compared to in-GPU-memory sampling, which requires up to 2.26$\times$ more GPU memory space. As of this time, the performance benefits over large out-of-memory graphs are still undetermined. We analyze possible reasons for slowdown in our system, and provide ideas for moving forward.

The major contributions of this thesis are:

- We propose a hybrid graph compression strategy for graph sampling to reduce GPU memory footprint and improve GPU memory efficiency.
- We propose an adaptive data caching approach to minimize data movement overhead between CPU and GPU.
- We build ScaleGPS that implements our proposed hybrid, adaptive data management strategy, and demonstrate the potential for high performance and scalability for various graph sampling algorithms.

Chapter 2

Background and Related Works

There have been a large volume of studies on graph sampling. For example, Graph sampling has been used in graph neural networks [14]–[17], triangle counting [20]–[28], clique/cycle counting [27], [29], [30], butterfly counting [31], motif counting [32]–[43], and frequent subgraph mining [44]–[48].

In this chapter, we first go over the representative graph sampling algorithms in Section 2.1 and existing graph sampling frameworks in Section 2.2. Since our proposed approach leverages compression and caching techniques, we also introduce existing data compression techniques in Section 2.3 and data caching techniques in Section 2.4.

2.1 Graph Sampling Algorithms

We start this section by introducing a high-level overview of the general process for many graph sampling algorithms. The algorithms we touch on sample sets of vertices sequentially for a fixed number of steps N. To begin generating a *sample*, or subgraph, we populate the first layer of our sample with our chosen initial *frontier* vertices. Frontiers are the set of vertices that were sampled during the most recent layer, and are also the vertices that will aid in choosing the vertices for the next layer to be added to our sample. To determine the frontiers for the next layer, we typically have some way of sampling from the neighbors of

Algorithm 1 k-hop neighborhood sampling

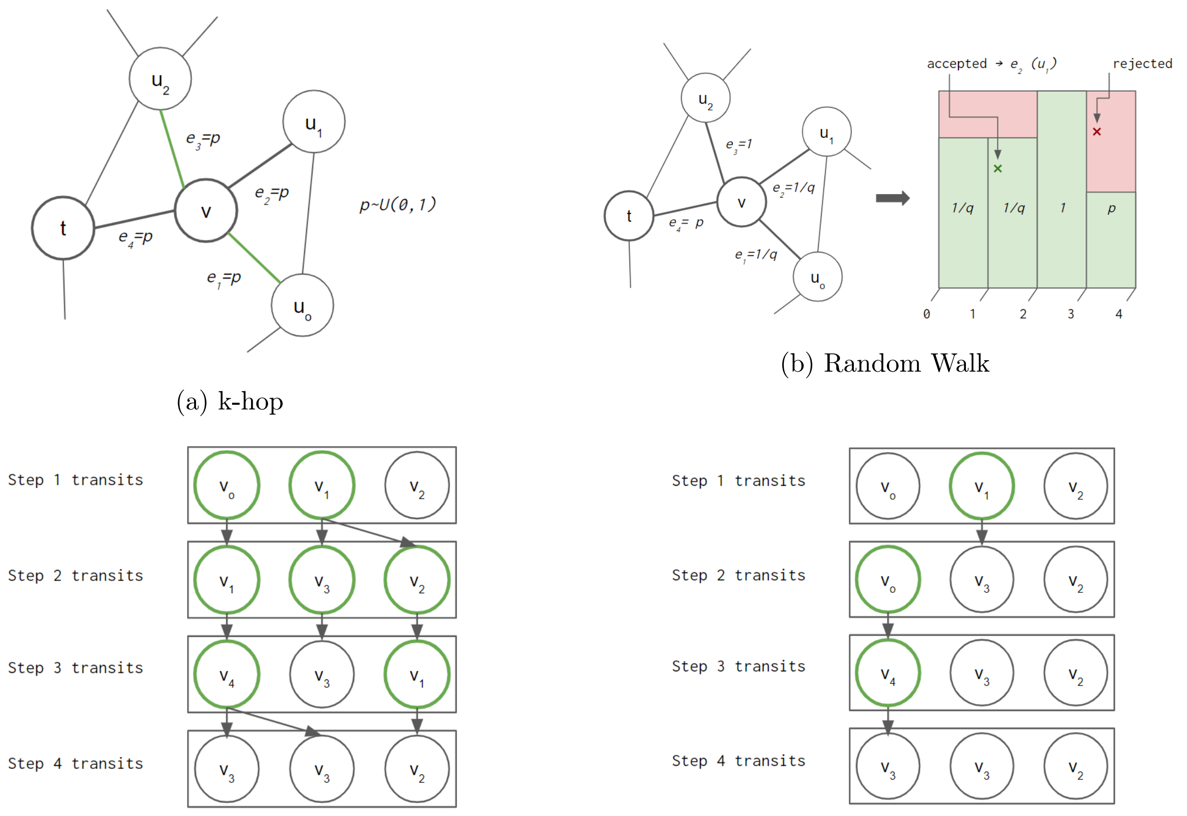
```
1:  Input GRAPH $g$
2:  Output VECTOR<VERTEX> sampled frontiers in order
3:  procedure KHOP_SAMPLES
4:      $frontiers$ := VECTOR<VERTEX> of size total frontiers to be sampled
5:      Enter initial frontiers into head of $frontiers$
6:      $current\_step\_size$ := number of initial frontiers $\times$ batch size
7:      for $step = 0$ to N do
8:          Multiply $current\_step\_size$ by $fanout_{\text{step}}$
9:          for $t\_idx = 0$ to $current\_step\_size$ do
10:             INT $oldt\_idx$ := Calculate the index of the previous frontier
11:             INT $oldt\_degree$ := Get outgoing degree of $frontiers[oldt\_idx]$ from $g$
12:             VERTEX $new\_t$ := Randomly sample from $oldt\_degree$ neighbors
13:             $frontiers[t\_idx] = new\_t$
14:     return $frontiers$
```

the current frontiers to maintain the connectivity of the original graph. The *fanout* number m_i at step i is the number of new vertices per frontier that we will add to the next layer of frontiers. By the end of our sampling, we should have sampled $\sum_i^N \prod_j^i m_j \times b$ total vertices, for a batch size (number samples) of b.

While there are many different sampling algorithms that are each better suited for different purposes, we describe the four algorithms highlighted in NextDoor [55] for their simplicity and popularity. Below is an overview of each of the sampling methods.

- **k-hop neighbors** [14]. This algorithm samples $N = k$ layers. For every frontier t at step i, m_i new edges are sampled from t's outgoing edges to decide the next layer's frontiers. These sampled edges are also included into the sample.

- **Random Walk** [51]. For every frontier v and its previous frontier t that we used it to sample from, we determine the probability of picking one of v's neighbors u depending on three conditions in this respective hierarchy: (i) if $u = t$ the probability is p, (ii) else if u is connected to t then the probability is 1, (iii) else the probability is $1/q$. Both p and q are fixed hyperparamters. Using these probabilities, u is finally selected through rejection sampling [13]. Note the fanout $m_i = 1$ for every step i.



(a) k-hop

(b) Random Walk

(c) Multi Dimensional Random Walk

(d) Importance

Figure 2.1: Illustration of four different sampling methods.

- **Multi Dimensional Random Walk** [61]. This type of random walk similarly has $m_i = 1$ for every i. The main feature of this algorithm is that starting with our initial frontiers set, every newly sampled vertex is a random neighbor of a randomly chosen frontier t from the layer. This new frontier will then replace t in the set, and the process repeats N times. The specialty of this feature is that we can continue our sample expansion in the direction of old frontiers as well, such that not every root frontier in our sample will end up with the same length path like in a normal random walk.

- **Importance Sampling** [16], [17]. Importance sampling is the only algorithm that practices collective sampling, meaning that new frontiers are selected from the collection of all neighbors from the current frontiers. So for every layer, we sample a fixed number of frontiers from the neighbors of the layer previously, and add an edge between any new frontier and previous frontier if that edge existed within the original graph as well.

Fig. 2.1 illustrates each of the algorithms listed above. K-hop and the random walks are all *node-wise* [49] sampling algorithms because each frontier is individually used to sample new ones. On the other hand, importance sampling is *layer-wise*, which picks its new frontiers from the collective set. There is no right or wrong answer as to which sampling method is better, but depending on the application one method might be preferred over the other. For example, GraphSAGE [14], [62] is one of the first works to include mini batching from sampling into GCN training. They use node-wise neighborhood sampling to increase training efficiency. In contrast, LADIES [16] uses layer-dependent importance sampling to reduce variance which improves GCN model accuracy.

Of these four, we evaluate the design of our system in the context of k-hop. Algorithm 1 contains the pseudocode for sequential k-hop sampling. In order to parallelize the code, one can change line 9 to use Parallel For instead.

2.2 Graph Sampling Systems

This section highlights different existing graph sampling systems and their novel contributions, implemented on either CPU or GPU.

2.2.1 CPU Sampling systems

For random walk algorithms specifically, the majority of sampling time comes from calculating the probabilities of selecting an edge. In conventional random walks like node2vec [13], sampling the next frontier involves examining every outgoing edge from the current frontier and assigning a probability of accepting each if the edge gets selected. To address this inefficiency, KnightKing [51] is a distributed random walk engine with its own edge probability re-weighting process. With the novel incorporation of rejection sampling into their implementation, their algorithm neared $O(1)$ complexity for edge sampling compared to the $O(|E_v|)$ complexity of previous random walk iterations such as node2vec.

SALIENT [53], [63] is another work that opts to modify a basic sampling algorithm for better performance. Focusing on k-hop neighbor sampling, this paper makes two main contributions with their implementation. The first is optimizing k-hop neighbor sampling itself, and notably having the code run in parallel on different threads for separate mini-batch samples. The sampling in SALIENT is meant specifically for iterative GNN batch training. Thus, the second main contribution is reducing the communication overhead of fetching sampled vertex features from memory by implementing an analysis of vertex inclusion probabilities (VIP analysis) to optimize caching frequently accessed neighbors. One area of future work that the paper notes is improving on how vertex features are partitioned across machines, to scale for even larger graphs.

Other than SALIENT, there are plenty other graph frameworks that are also motivated by reducing the latency caused by the communication overhead between different memory levels. ThunderRW [50] stores its large graphs in-memory, and limits the number of random accesses when sampling an unvisited edge using a work balanced *step-centric* model. The step-centric model breaks up individual edge queries into three different steps, and because steps within a single query must happen sequentially, they find a method to parallelize the steps between queries instead called step interleaving. So while a step in one query is halted while waiting to fetch data from memory, the thread switches to handle the step of another query first before returning to complete the original query's step. FlashMob [52] is another paper published around the same time under the same motivations as ThunderRW, except they adopt the opposite approach as ThunderRW for handling irregular memory accesses. By dividing the graph into alike partitions and batching queries by partition, FlashMob directly reduces the number of cache misses rather than hiding their latency behind other computations. However, unlike ThunderRW, FlashMob has not been tested on disk-resident graphs.

2.2.2 GPU Sampling Systems

The first paper that we explore makes many contributions towards graph sampling and random walk algorithms, but the biggest one of interest to our thesis is their multi-GPU thread allocation. Their framework, C-SAW [54], uses what they call *workload-aware partition scheduling*. The process starts by partitioning the input graph into subgraphs with an equal count of contiguous vertices to store on the separate GPUs. Different partitions will generally offer different workloads due to the difference in frontiers of each. To make sure a single workload does not become the bottleneck, the system keeps track of the frontiers of each partition and loads these partitions onto GPUs asynchrously. Each GPU kernel is also allocated a proportional number of thread blocks to the number of frontiers in its partition. Thus, a partition that likely has a bigger workload is scheduled to start sampling first and with more parallelism.

Similarly, the NextDoor framework aims to optimize the parallelism of GPUs. NextDoor [55] proposes a new thread assignment system for sampling frontiers of a single layer in parallel on GPUs. The idea that they introduce is *frontier-parallelism*, which is an alternative assignment paradigm to the conventional individual frontier sampling in which the frontiers of the same sample are assigned to consecutive threads. However, this method is prone to warp divergence and load imbalance due to graph irregularities. Additionally, the entire graph must be stored in memory because of the different neighborhoods that must be accessed. In contrast, frontier-parallelism regroups samples into samples of the same frontier. Consecutive threads now access the same neighborhood of a single frontier that is able to fit in shared memory, and these threads also follow the same set of instructions. This implementation addresses the inefficiencies in thread parallelism on GPU, and works for various sampling algorithms. While the majority of their results showed noticable speedup in both isolated sampling and when applied to GNN training, when run on graphs with only millions of vertices the GPU already begins to run out of memory.

In response to C-SAW and NextDoor, Skywalker [56] attempts to create a new framework to address the inefficiencies within the two algorithms. Skywalker's primary focus is to reduce the time of the sampling algorithm itself by using a parallelizable alias table on GPU. The alias table stores the biases of selecting a new frontier prior to sampling, so that the sampling itself would only take constant time. On the other hand, methods like rejection sampling used in NextDoor [55] could take many trials before accepting a frontier to the sample. By leveraging a way to parallelize the alias table construction and learning to compress their alias table to save space, Skywalker was able to show an average speedup of 5.2× while also being able to handle larger input graphs compared to NextDoor.

Finally, Gong et al. poses one of the most recent GPU-based sampling systems named gSampler [49]. gSampler boasts a *matrix-centric* API, meaning that the graph data is either parsed as matrices or tensors throughout every step of the sampling process, so that the system is able to utilize matrix computation optimizations to reduce performance costs. Significantly, this design also allows the system to be compatible with a variety of different sampling algorithms due to the global view of the entire sample that a matrix abstraction provides. The system uses a generalizable 4-step Extract-Compute-Select-Finalize (ECSF) programming model to characterize each algorithm. However, one limitation of gSampler is that it sacrifices performance on certain algorithms in exchange for this generalizability. Some algorithms such as node2vec [13] that require fine-grained operations are just better suited with a more customized approach.

2.3 Data Compression

A vital factor in the performance of our implementation is the compression scheme we choose to implement our data structure with. The better the scheme's compression ratio, the more of the graph that can fit onto GPU memory and the less data that must constantly be transferred between CPU and GPU. Unfortunately, schemes that are better at compressing

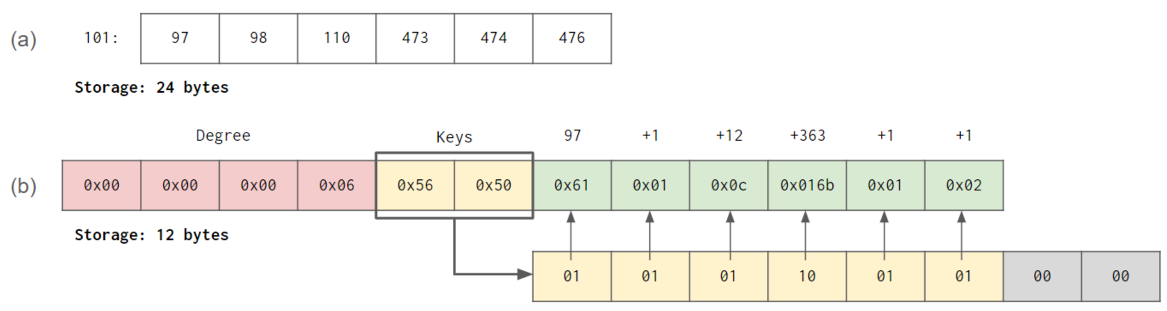


Figure 2.2: Stream VByte Encoding: (a) Original, uncompressed edge list of 32-bit unsigned integers; (b) Compressed egdelist using Stream VByte.

are also generally more complex, resulting in a more expensive decoding step. In this section we look into the different existing compression techniques.

2.3.1 Integer List Compression

One of the compression schemes that we implement performs integer list compression from Lemire et al. [64], named Stream VByte. Stream VByte is an encoding method that stores the difference between contiguous neighbors rather than their values. Based on the fact that most real world graphs demonstrate relative locality between neighbors and their vertex labels, Stream VByte is well-suited for encoding graphs originally stored in Compressed Sparse Row (CSR) format. Additionally, the differences are encoded using only as many bytes as are needed to represent them. The number of bytes taken to encode each delta is then stored in a fixed number of bits which we call a *key*. We illustrate how to encode an edge list originally in CSR form using Stream VByte in Fig. 2.2. In the first chunk of the edge list, the vertex's degree is always stored as a 32-bit unsigned integer. The second chunk then contains our 2-bit keys in sequential bytes and the final chunk contains the actual data stored as their delta values.

When reconstructing the neighbor set, we first decode the first four bytes to get the degree. Then using the degree, we can easily calculate the offset to the data pointer since each key is a fixed length. Continuing with our running example from above, once we get a degree of 6 we find that the data starts $(6+3)//4$ bytes away from our key pointer, rounded up to the nearest byte. At this point we can simply perform prefix sums over the data chunk to get the neighbor labels. While the computation to recover the data is simple, performing the decoding sequentially can end up being a bottleneck for vertices with giant degrees. Later works such as SIMD-BP128* and SIMD-FastPFOR* [65] utilize vectorization over SIMD to achieve better decompression performance for arrays with billions of integers. Unlike Stream VByte which encodes the deltas between every contiguous element, the SIMD variations encode deltas in batches of four because current processors had the capacity to operate on four 32-bit integers in a single SIMD instruction. SIMD Galloping [66] is another paper that similarly takes advantage of SIMD vectorization, in this case by combining the unpacking and prefix sums instructions during decompression.

Most of the techniques to date were created with the intention of being executed on CPU. Recently, however, there has been growing interest in integer list compression for GPU optimization. GPU-VByte [67] is the GPU counterpart to the Stream VByte method described above. In this paper, the main contribution is how the work for compression and decompression gets balanced among threads. Each thread is assigned to decode a single element in a block, and once complete a parallel prefix sums is executed over these thread groups. GPU-VByte boasts impressive decompression speedups of up to $60\times$ its CPU implementation. Other work balancing models like tile-based integer decompression [68] and cascading decompression [69] have emerged since then.

2.3.2 Graph Compression

Integer list encoding happens to work well for compressing CSR data, however there are also compression schemes catered towards graph compression. One of the more well known techniques is Compressed Graph Representation (CGR) [70]. CGR similarly encodes the delta value between consecutive neighbors like Stream VByte, but first divides the data into *intervals* and *residuals*. An interval is formed from consecutive neighbors, and residuals are

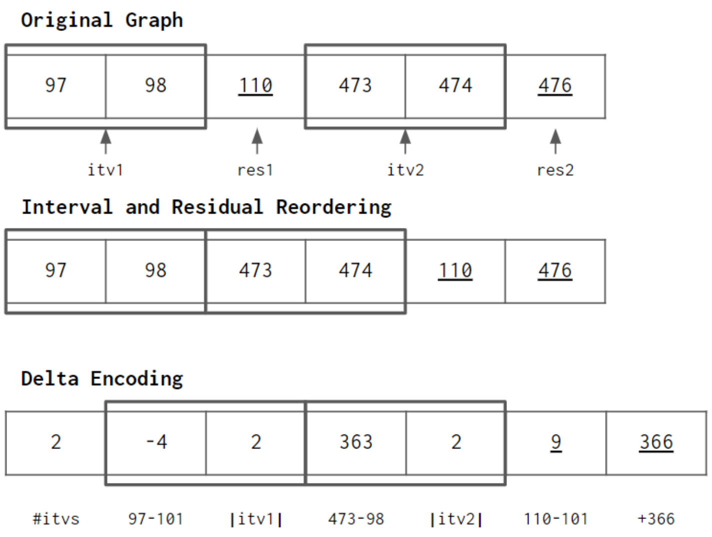


Figure 2.3: CGR Encoding.

any of the remaining neighbors. Fig. 2.3 illustrates the encoding process using the same Stream VByte example. Once the intervals and residuals are found, they are reshuffled and encoded using variable length bytes to be next to other intervals or residuals. CGR, being made for graph compression, takes advantage of vertex locality in graphs such that contiguous vertices are often neighbors. A scheme with a similar blocking technique to CGR is CGC [71]. In CGC, the edge list is compressed in chunks like CGR, however to faster access any neighbor within the edge list CGC comes up with a novel format to store these chunks called Linear Estimation (LE). LE is a linear function that approximates neighbor IDs using only three decoded values.

There are many other schemes besides the popular difference encoding schemes that we have mentioned so far. Ligra+ [72] uses *run-length encoded byte codes*, storing blocks of continuous elements that use the same amount of bytes to read. MPLG [73] is a log-encoding scheme for GPU optimizations which reduces the storage of individual neighbors by removing leading zero bits. There is also partial-decoding [74] better suited for accessing entire neighbor lists by decoding CGR encoded graphs into intervals and residuals to use, first introduced by Ye et al. for the context of graph label propogation. CompressGraph [75] follows a *rule-based encoding scheme*. Rule-based encoding finds patterns in graphs that lead to rules with repeated computations, such that these computations can be saved in memory and reduce the amount of overall intermediate computations necessary when decoding.

2.4 GPU Memory Oversubscription & Data Caching

In response to the low storage capacity of GPU machines, there have been lots of research done to mitigate expensive data transfers between devices. GMT [76] proposes a new 3-tier memory hierarchy for GPUs which extends GPU memory to utilize host memory and SSDs as well in the case of oversubscription. They make many contributions, mainly a GPU-hosted page access prediction system to handle page evictions between tiers. Previous frameworks that also combine CPU memory [77]–[79] before GMT deferred data transfers to be handled by the CPU, which is slow in the case of large simultaneous page faults.

Other solutions to reduce the expensive overhead caused by data migration is to introduce smart caching techniques. DeepUM [80] is a framework meant for Deep Neural Networks (DNNs) that also utilizes unified memory but also innovatively exploits the fact that DNNs' follow the same memory access patterns in a kernel during different training iterations. Throughout every kernel launch, DeepUM manages correlation tables that remembers a kernel's page access history. The framework's driver can then prefetch pages based on these correlations. Because DNNs offer some basis of access regularity, there have been lots of other works for DNNs which offer page caching and prefetching [81]–[83].

Chapter 3

System Design and Implementation

In this chapter, we present ScaleGPS, a scalable graph sampling framework that leverages GPU to accelerate sampling on massive-scale graphs. We first give an overview of the ScaleGPS system in Section 3.1, and then introduce our proposed two main contributions in ScaleGPS: (1) the *hybrid* graph compression technique in Section 3.2 and (2) the *adaptive* data caching strategy in Section 3.3. Finally, we describe our GPU parallelism strategy, using k-hop neighbor sampling as an example, to demonstrate the usability of ScaleGPS in Section 3.4.

3.1 ScaleGPS System Overview

Fig. 3.1 illustrates the general structure of ScaleGPS. We begin with a preprocessing step that separates the input graph into two subgraphs, G_l and G_h. The first subgraph G_l holds the neighbor lists of vertices that we define to have a *low-degree*, while G_h contains the remaining *high-degree* neighbor lists. Both subgraphs are also compressed using different compression schemes. ScaleGPS is a hybrid system that allows unified virtual addressing (UVA), which is a storage space that permits communication between the CPU and GPU devices. Prior to the start of using ScaleGPS to run a graph sampling algorithm, we load G_l onto UVA and copy G_h to GPU's static cache. We also utilize the GPU's automatic

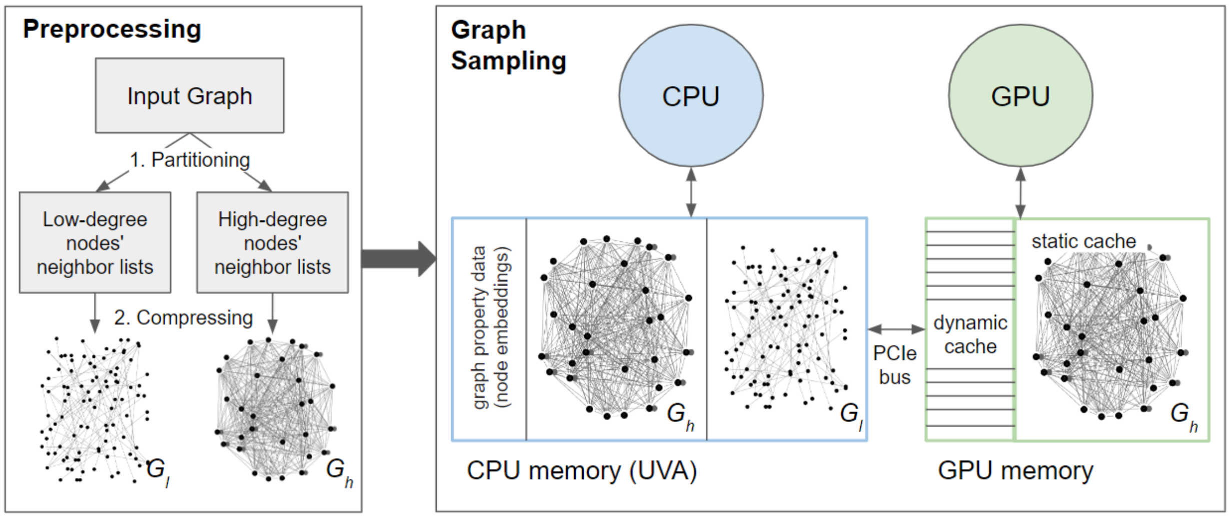
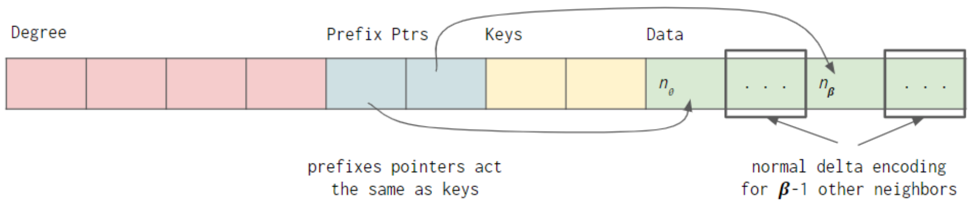


Figure 3.1: High-level overview of ScaleGPS.

dynamic caching system throughout the program.

For the graph sampling step, the actual computations for graph sampling is done on GPU, while the CPU is used solely to initiate data transfer. After having loaded the neighbor lists and any other necessary graph properties into their allocated memory spaces, we can then execute the graph sampling algorithm.

3.2 Hybrid Graph Data Compression



Degree
Prefix Ptrs
Keys
Data
n_e
. . .
n_β
. . .
prefixes pointers act the same as keys
normal delta encoding for β-1 other neighbors

Figure 3.2: Illustration of how segment prefix are incorporated into a compressed edge list.

In order to leverage the GPU to accelerate graph sampling, we have to first copy the graph onto the GPU memory. When the graph is large, GPU memory is only able to store a subset of vertices. We then turn to graph compression techniques in order maximize the amount of data that can be put onto fast memory. However, there are still two main problems with normal compression schemes as is, which we propose our own solution to.

The first problem is actually due to the nature of k-hop sampling. At any time step i, we want to sample m_i new frontiers from the current frontier t, only we do not know t nor which m_i neighbors of t until step i occurs. This means that when we go to decode the neighbors of t, we need to have already allocated a buffer of size Δ to store any possible set of neighbors sampled from any t, where Δ is the maximum degree of $\mathcal{G}$. For large graphs with a huge Δ, they will most likely be out of memory (OoM) just by allocating these buffers. Our solution is to store the prefix sums of every β neighbors, such that when we decode we can simply start from the nearest prefix so that the most we need to store at any given time are the β neighbors before the next prefix sum. To quickly access the queried prefix sum from the compressed array, we store prefix pointers at the beginning of the array. They work in the same way as keys do described in Section 2.3.1, storing byte offsets. Fig. 3.2 shows the arrangement of a compressed edge list using our prefix sums.

Storing prefix sums also helps to solve our second problem, which is simply the fact that decoding takes time and can slow down performance. With prefix sums, not only is the auxiliary space we need for decoding capped at $O(\beta)$ but so is the number of neighbors needed to decode. By decreasing β, we can decrease the decoding time but at the cost of a worse compression ratio. Because memory is still a large concern for our system, we only use prefix sums to compress the higher degree vertices. We call this subgraph G_h for the high-degrees, and the remainder of the vertices get compressed using a normal scheme into our low-degree subgraph, G_l. The degree threshold, α, for deciding which vertices belong to which subgraph is a tuneable parameter we discuss more in our evaluations.

We create G_l and G_h when we first load in the graph, and then deallocate the space from the original graph afterwards it no longer gets used.

3.3 Adaptive Graph Data Caching

Even after compression, graphs with hundreds of billions of edges will still need to be at least partially moved onto the UVA space. We design ScaleGPS such that the vertices we expect to be accessed the most frequently are the ones we prioritize caching into memory. While we can never exactly predict which frontiers will be sampled beforehand, we can make an educated guess using the degrees of vertices. It is a well known fact that almost all real world graphs follow the power distribution law [84],[85], which states that only a small subset of vertices account for most of the graph's edges. With this in mind, we can expect that these well connected vertices get accessed the most and that we can save the most data movement between devices by directly caching them onto the GPU.

Thus, our construction follows that we load as many vertices as possible onto GPU in order of decreasing degree before running any algorithms. The remaining vertices make up G_l which we store on UVA. Although the power law also tells us that this would end up putting a majority of the graph onto UVA, we do this with the expectations that communication overhead should still be tolerable because of the inherent small amount of neighbors these vertices have. Prior to sampling, we also launch a warm-up kernel to move some vertices from UVA onto GPU's fast cache. The warm-up kernel performs a simple read and write for all initial low-degree vertex neighbor lists to decrease the number of immediate cache misses at the start of our execution.

3.4 GPU Parallelization Scheme for Graph Sampling

With our new subgraphs, we now describe how to use ScaleGPS to implement different graph sampling algorithms. The main modification to any algorithm would be that we now pass in two graphs instead of one. In order to sample from each graph accordingly, we need to keep track of our new parameters α and β. We use α to decide which

Algorithm 2 SCALEGPS k-hop (Single Sample)

```
1: Input CompressedGraph $G_l$, $G_h$, Int $\alpha$, Int $\beta$
2: Output vector<Vertex> sampled frontiers in order
3: procedure KHOP_SAMPLES_BY_WARP
4:     $frontiers$ := vector<Vertex> of size total frontiers to be sampled
5:     Enter initial frontiers into head of $frontiers$
6:     $prev\_step\_size$ := number of initial frontiers
7:     for $step = 0$ to $N$ do
8:         for $oldt\_idx = 0$ to $prev\_step\_size$ do
9:             $oldt = frontiers[oldt\_idx]$
10:            int $oldt\_degree$ := Get outgoing degree of $oldt$
11:            $buffer$ := vector<Vertex> of size $\alpha$
12:            if $oldt\_degree < \alpha$ then
13:                $G_l$.DECODE_VBYTE_WARP($oldt$, $buffer$)
14:            for $t\_idx = 0$ to $fanout_{step}$ do
15:                int $n\_idx$ = Get random index between 0 and $oldt\_degree$
16:                if $oldt\_degree < \alpha$ then
17:                    Vertex $new\_t$ := $buffer[n\_idx]$
18:                else
19:                    Vertex $new\_t$ := $G_h$.DECODE_VBYTE_PREFIXES($oldt$, $n\_idx$, $\beta$)
20:                $frontiers[t\_idx] = new\_t$
21:        $prev\_step\_size$ *= $fanout$
22:    return $frontiers$
```

graph the vertex belongs to, and β for decoding the compressed prefix sums using the correct interval. We also introduce two new functions, DECODE_VBYTE_WARP and DECODE_VBYTE_PREFIXES. DECODE_VBYTE_WARP decodes our base compression scheme normally, while DECODE_VBYTE_PREFIXES is modified to be able to read our prefix pointers and decoding from there.

In Algorithm 2, we demonstrate the pseudocode for applying SCALEGPS to k-hop on a single warp. In lines 7-21 we perform k-hop sampling in order of its steps. We allocate buffer space for decoding neighbors from G_l at lines 11-13. Then, if the the current frontier has degree less than α, we pick our new frontier from G_l else from G_h. For every frontier in the warp at each step, we repeat this sampling process for the fanout size. Comparing the pseudocode to Algorithm 1, we only add a few more branches for checks. Incorporating existing sampling algorithms is easy with SCALEGPS, since the logic of the algorithm need

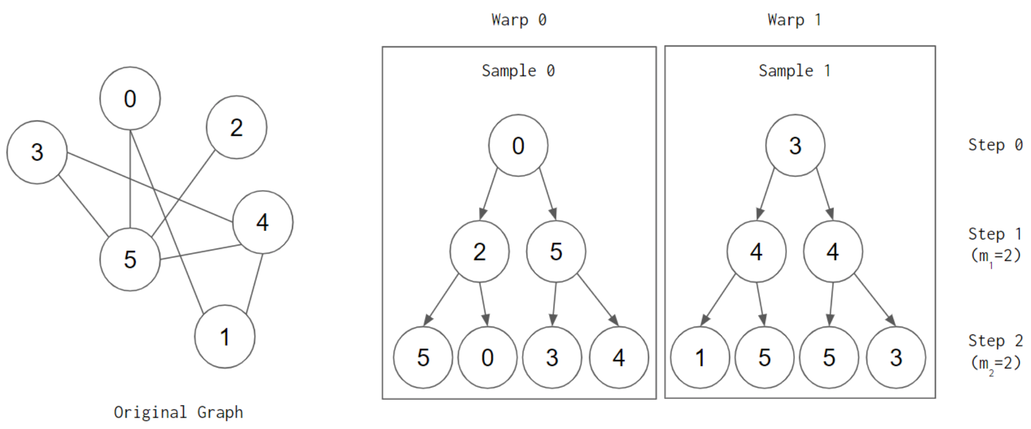


Figure 3.3: Work is divided between samples in ScaleGPS.

not change.

To avoid warp divergence during sampling, we adopt sample parallelism [55] for our work balancing strategy. Each warp is assigned a single sample for the entirety of the algorithm, and each warp also only works on sampling one frontier from its set of current frontiers at a time. For the low-degree subgraph, we achieve this by having each thread decode a single neighbor delta from the compressed array. Then after syncing all of the threads within the warp, we perform a parallel prefix sums to recover each neighbor. For the higher degree subgraph, we parallelize threads over fanout size. We include an illustration in Fig. 3.3, where for example at step 1, Warp 0 would sample from 2 and 5 sequentially. We do note that if the fanout size is less than the warp size, this could cause wasted parallelism since not all threads are used. However, we make this tradeoff so that threads within a warp can share the same neighbors and reduce repetitive decoding.

Chapter 4

Experimental Results

In this chapter, we first describe the experimental setup in Section 4.1. We then compare the sampling performance with the baseline implementations on CPU and GPU in Section 4.2. Finally we conduct an ablation study in Section 4.3 to show the performance impact of our proposed mechanisms. The artifact of our evaluation can be found in Appendix A.

4.1 Experimental Setup

All of our GPU experiments are performed on a single Nvidia A100 GPU 80GB machine with 256GB DDR4-3200MHz RAM (from an Intel Xeon Gold Icy Lake 6338 CPU) while our CPU-only experiments are run on an Intel Xeon Gold Cascade Lake 6248R with 1.5TB DDR4-2933MHz RAM. To benchmark the performance of our hybrid CPU-GPU approach, we implemented an optimized CPU k-hop sampling algorithm parallelized across 48 cores. We also implemented two other baseline GPU solutions, one that takes the original uncompressed graphs as input and another that uses VByte compression without our prefix sums modification. The goal efficiency was that our implementation would outperform the CPU implementation and only experience slight slowdowns compared to the other GPU versions. The datasets that we chose to test on are shown in Table 4.1. We carefully selected this set of graphs to include both small datasets that fit into GPU memory as well as large datasets which

Dataset	**|V|**	**|E|**	**Max Degree**	**CSR Size**
livej	4,847,571	85,702,474	20,333	327MB
orkut	3,072,441	234,370,166	33,313	895MB
twitter40	41,652,230	2,405,026,092	2,997,487	9.0G
friendster	65,608,366	3,612,134,270	5,214	14G
uk2007	105,896,435	6,603,753,128	975,419	25G
mag240m	121,751,666	2,595,497,852	242,655	9.7G
gsh-2015	988,490,691	51,381,410,236	58,860,305	192G
clueweb12	978,407,686	74,744,358,622	75,611,696	279G
uk-2014	787,801,471	84,928,431,100	8,605,492	317G

Table 4.1: Description of graphs used in our evaluation.

Dataset	**CPU (48-core)**	**GPU in-memory**	**GPU UVA**	**ScaleGPS (Low on UVA)**	**ScaleGPS (Cached)**	**ScaleGPS (All on UVA)**
livej	26.8	0.911	42.2	18.2	2.02	34.6
orkut	30.8	0.906	110.2	10.1	3.13	73.4
twitter40	110.7	0.902	713.7	175.3	3.57	721.1
friendster	125.4	0.964	855.8	229.6	3.42	1,455
uk2007	52.3	0.935	1090.4	361.2	3.06	1,558
mag240m	207.3	0.818	677.2	419.3	2.23	1,278
gsh-2015	102.1	OoM	19,622	5,612	5,746	8,751
clueweb12	61.7	OoM	35,093	9,581	9,529	13,566
uk-2014	507.0	OoM	23,547	12,286	12,412	10,463

Table 4.2: Time (ms) taken for k-hop sampling using different schemes, blanks are out of RAM.

typically run out of memory in only-GPU solutions.

Note that for any experiments running k-hop sampling, we use parameters $N = 2$, $b = 40,000$, and $m_0, m_1 = \{25, 10\}$. We include Table 4.3 for additional experiments run on a different set of parameters as well.

4.2 Overall Sampling Performance

4.2.1 Comparison with CPU Implementation

The motivation of this thesis stems from wanting to leverage the compute powers of a GPU for large graphs that would normally exist on CPU. So at the bare minimum we expect ScaleGPS to outperform the fastest CPU implementation. To compare our results, we implemented an optimized, parallel CPU k-hop solution that follows the sample parallelism

Dataset	**CPU (48-core)**	**GPU in-memory**	**GPU UVA**	**ScaleGPS (Cached)**
livej	40.5	2.60	43.9	14.4
orkut	59.5	2.63	115.4	18.2
twitter40	72.2	2.73	1,059	17.7
friendster	75.1	2.76	1,191	20.6
uk2007	73.1	2.79	1,386	15.7
mag240m	46.8	2.32	892.9	17.6
gsh-2015	227.7	OoM	–	53,169
clueweb12	137.6	OoM	–	162,419
uk-2014	367.0	OoM	–	197,356

Table 4.3: Time (ms) taken for k-hop sampling ($N = 3$, $b = 40,000$, $m = \{15, 10, 5\}$) using different schemes.

technique described in NextDoor [55]. Table 4.2 shows the speed of our different solutions. Our three largest graphs, gsh-2015, clueweb12, and uk-2014 were our main areas of interest since none of them fit into GPU memory (>80GB) even when using any of the compression schemes. Unfortunately, we see that ScaleGPS actually poses slowdown for these large graphs. Even when caching as many vertices as possible by copying the high-degree graph to GPU memory and reading the low-degree data into dynamic cache before sampling, we see between a 24.4× to 154× slowdown. This is largely due to the fact that a lot of the data still resides on CPU. For our largest graph uk-2014, we still end up being almost 60GB over the GPU's memory capacity, which is why our pre-fetching ends up showing little to no impact.

On the other hand, a secondary goal we had for ScaleGPS was that we did not want our small datasets to suffer from optimizations catered towards larger inputs either. In our tests, we do find significant speedups for all of the graphs that can originally fit into GPU memory without compression. The smallest graph, livej, still demonstrated 13.3× speedup over CPU, and on average all of these graphs had 33.4× the speedup. This was the case when we used prefetching to dynamically cache the entire low-degree neighbor lists. Without prefetching, we only saw between $0.14\times - 3.06\times$ speedup to CPU, with performance decreasing as the graphs grew larger.

4.2.2 Comparison with GPU Implementations

In this final section we compare ScaleGPS to other GPU solutions that do not involve any sort of compression or special scheme. As a benchmark, we implemented an in-GPU-memory solution and an on-UVA solution. As expected, we ran into OoM kills when trying to sample gsh-2015, clueweb12, and uk-2014 in-memory. When this happens, an easy alternative is to simply move the graph onto UVA and to move pieces of data to the device as needed. We show that this is an undesirable method, however. For the big graphs that did OoM, we see $46\times - 568\times$ slowdown from moving the uncompressed graph to UVA compared to the CPU version. Compared to the uncompressed UVA implementation, ScaleGPS still reports up to a $3.6\times$ speedup.

Unlike the comparison to our CPU version, we observe that ScaleGPS when fully cached shows slowdown for the small graphs compared to running k-hop sampling on uncompressed the in-memory GPU solution. ScaleGPS observes $2.22\times - 3.96\times$ slowdown compared to the in-memory implementation but $20.9\times - 356.6\times$ speedup compared to the UVA one. This is unsurprising however, since the added decompression time is the trade off we make for being able to store the graphs in-memory. So for small graphs that already previously fit in-memory, we lose any new benefits from compression or from moving onto UVA.

4.3 Ablation Study

4.3.1 Performance Effect of Segment Prefix

The next part of our preliminary research involved testing to see the effect of adding segment prefix to our compression scheme in practice. Table 4.4 shows the speedups achieved by incorporating segment prefix into Stream VByte during k-hop sampling. The main achievement we see here is that decoding from the nearest prefix balances the workload between each warp rather evenly. On the contrary, without a segment prefix each warp can

Dataset	**VByte**	**VByte w/ Pref. Sums**
livej	16.6	2.05
orkut	71.1	3.11
twitter40	14,300	3.59
friendster	33.4	3.42
uk2007	8,900	3.07
mag240m	2,060	2.25

Table 4.4: Time (ms) taken for k-hop sampling using normal VByte compression versus VByte compressed with prefix sums.

Dataset	**VByte Comp. Ratio**	**CGR Comp. Ratio**	**VByte Speedup**
livej	×0.73	×0.72	×2.53
orkut	×0.56	×0.63	×1.81
twitter40	×0.54	×0.53	×1.53
friendster	×0.69	×0.79	×1.08
uk2007	×0.35	×0.26	×1.01
gsh-2015	×0.51	×0.41	–
clueweb12	×0.45	×0.32	–
uk-2014	×0.41	×0.24	–

Table 4.5: Comparison between CGR and VByte. Lists decompression ratios compared to CSR format and speedup of VByte over CGR decompression by thread.

take work ranging from between the graph's minimum degree and the graph's maximum degree. To quantify, the table shows an almost 9,000 ms difference in sampling using normal VByte on the slowest executing graph and the fastest executing one. Using prefixes drops this number to 1.54 ms. Therefore we demonstrate the scalability of using segment prefix and justify the extra memory needed to store them.

4.3.2 Performance Effect of Various Compression Schemes

Lastly for our preliminary research we ran experiments comparing CGR [70] and Stream VByte [64] compression. We looked at the two schemes in the contexts of compression power and decompression efficiency. The first condition is important for minimizing communication overhead while the second makes sure that compression remains useful at all. We present the compression ratios for multiple real world graphs and their decompression times that we will

Dataset	l	Low Grp. (GB)	Low Grp. |V|	Pref. Grp. (GB)	Pref. Grp. |V|
livej	32	0.12	4,194,409	0.13	653,162
orkut	32	0.06	1,184,290	0.50	1,888,151
twitter40	32	1.10	31,827,777	4.31	41652230
friendster	32	1.40	45,270,115	9.00	20,338,251
uk2007	32	2.30	68,995,952	9.53	36,900,483
mag240m	32	2.40	96,969,738	6.25	24,781,928.00
gsh-2015	64	41.30	733,060,968	63.63	255,429,723
clueweb12	128	63.28	884,042,998	72.31	94,364,688
uk-2014	256	66.70	748,593,297	73.71	39,208,174

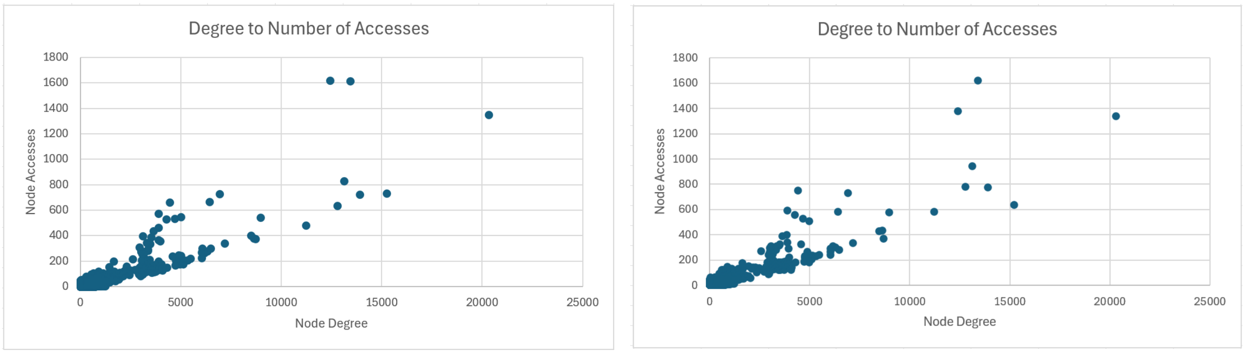
Table 4.6: Description of graph partitions.

use in Table 4.5. While compression ratios for both schemes will always vary from graph to graph, in general we see that CGR does better than Stream VByte for larger graphs. For the small graphs with less neighbors per vertex, the intervals are likely smaller as well which makes the extra interval encodings less beneficial. Based on this information, we opted to use Stream VByte to encode the low-degree group.

Additionally, when running k-hop sampling we found that CGR's slower decompression times either canceled out or completely outweighed its in-memory benefits in every case. We were unable to test the speedup against larger graphs because of OoM, but for the smaller graphs Stream VByte was up to $\times 2.53$ faster. This observation led us to believe that it would be more advantageous to use Stream VByte as our means of compression for the high-degree group as well. Its simple structure makes it easier to store and fetch prefix sums for this group in particular and its decoding time is faster.

4.3.3 Effect of Various Caching Policies

Even after compression, the largest graphs in our test suite surpass the size constraints of GPU memory meaning that some of the graph would still need to be in CPU memory. Thus, we need a strategic caching policy to minimize communication overhead. We wished to identify which vertices get accessed the most, since these are also the vertices that would benefit the most from fast caching. Lesser accessed vertices would then live on UVA.



(a) livej

(b) orkut

Figure 4.1: Relating vertex degree to how many times it gets sampled in a round of k-hop sampling.

In Fig. 4.1, we look at the effect of a vertex's degree size on how often the vertex gets accessed during a round of k-hop sampling for two of our graphs. Our plots show that both examples illustrate a mostly linear relationship between the two variables. This matches our expectations because real world graphs typically follow the power law distribution [85]. Thus, while there are lots more low-degree vertices, their accesses are distributed among them while the few vertices with the highest degrees repeatedly get sampled by their many neighbors. For the remainder of the experiments we then choose to store the low-degree neighbor lists in UVA while the remaining vertices are copied cached into GPU memory.

The value for α is chosen to maximize GPU memory usage. We start with $\alpha = 32$, and depending on the graph size increment α by 32 until all of the higher degree neighbor lists is able to fit onto GPU. We choose to increment by 32 (warp size) to maximize parallelism when sampling from the low-degree group which does decompression by warp. The partitioning information is reported in Table 4.6. The table shows that most vertices reside in the low-degree group in UVA, but from this ablation study we believed that each of these vertices individually would not be frequently accessed.

However, during our performance evaluations we realized that even single time accesses to data on the host are extremely costly. In Table 4.2, we provide three versions of ScaleGPS: one where the data fully resides in UVA, one where only the low-degree neighbor lists resides in UVA, and another where we attempt to fully cache both groups onto GPU by pre-fetching the low-degree neighbor lists before sampling. When on UVA, the small graphs only experience a single miss when read for the first time and then remain cached in memory for the remainder of the algorithm. Yet we still observe that the UVA version showed up to 573.4$\times$ slowdown in these cases, meaning that a difference of one in the number of misses per vertex is still extremely significant. As is, our caching methods are unable to negate this latency from transferring data.

Chapter 5

Future Work

There are many improvements we can make to ScaleGPS, as well as multiple extensions that we would like to add to make it a more complete framework. We outline some of the possibilities in this chapter.

5.1 Different Graph Compression Techniques

By the end of our experiments, we were unable to obtain the speedups over CPU that we originally sought after. However, based on our ablation studies and failures, we still believe that the underlying principle for ScaleGPS of combining graph compression techniques with adaptive caching can help us to create a fast GPU solution. The first direction of change we would make to ScaleGPS is to implement a more powerful compression scheme than Stream VByte. After seeing the results in Section 4.3.3, where a single cache miss per vertex slows the algorithm down by more than a hundred fold, we realize now that our trade off between compression ratio and decompression time was too imbalanced. One option is to use CGR [70] which we originally were considering as well, or one of the other schemes mentioned in Section 2.3.2. Alternatively, we could also go down the route of incorporating more complex caching. For example, vertices frequently get resampled both within the same sample and different samples across every step. Storing intermediate computations like CompressGraph

[75] does would save effort and reduce the number of memory accesses required.

5.2 Generic Sampling Framework

Currently, ScaleGPS has only been tested to support k-hop sampling. To become a more comprehensive framework with support for a variety of different graph sampling algorithms, we intend to create a generic API such as NextDoor's [55]. This includes allowing user-defined functions and parameter tuning, so that ideally users would need minimal involvmement in the backend coding.

5.3 CPU-GPU Co-sampling

Outside of using the CPU for overflow storage, we did not consider other forms of a CPU-GPU cooperative system. A very recent work published at the time of this paper introduces CGgraph [86], an alternative hybrid CPU-GPU solution which also offloads work onto the CPU as well. The idea behind CGgraph is similar to ScaleGPS, in that their first step uploads a dense subgraph that contains vertices expected to be frequently accessed onto GPU. However, to combat the problem of slow unified memory data transfer that we ran into when accessing vertices not in this subgraph, CGgraph opts to do part of the sampling on CPU instead. It would be worthwhile to try making ScaleGPS a CPU-GPU heterogenous system while still utilizing graph compression.

Chapter 6

Conclusion

In this thesis, we aimed to create a graph sampling system that takes advantage of GPU accelerators while overcoming the classic problem of GPU memory over-subscription. Our system ScaleGPS attempted to do so by creating a hybrid graph compression strategy paired with data caching. Although we did not obtain the speedups we wanted for large graphs in specific, our preliminary results show great potential for improvement. The smaller graphs we tested on were still able to benefit up to 93× over a state-of-the-art parallel CPU solution, and we were able to isolate the bottleneck of our implementation to be the memory accesses from our low-degree neighbor lists on UVA. With more testing and better compression methods, ScaleGPS can become a powerful graph sampling tool in the future.

Appendix A

Artifact

Abstract

This artifact appendix helps the readers reproduce the main evaluation results of ScaleGPS.

Scope

The artifact can be used for evaluating and reproducing the main results of the thesis, including Table 4.2, Table 4.3, Table 4.4, Table 4.5, and Table 4.6, in Section 4.2 and Section 4.3.

Contents

The details of the contained code and how to run ScaleGPS are described at
https://github.com/chenxuhao/GraphAIBench/blob/mjcai/src/compressing/README.md.

Hosting

The source code of this artifact can be found at
https://github.com/chenxuhao/GraphAIBench/tree/mjcai.

Requirements

Hardware dependencies
This artifact depends on an 80GB Nvidia A100 GPU with a 256GB DDR4-3200MHz Intel CPU.
Software dependencies
This artifact requires CUDA 11.8.0 and GCC 11.2.0 or greater.

References

[1] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '18, London, United Kingdom: Association for Computing Machinery, 2018, pp. 974–983, ISBN: 9781450355520. DOI: 10.1145/3219819.3219890. URL: https://doi.org/10.1145/3219819.3219890.

[2] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin, "Graph neural networks for social recommendation," in *The world wide web conference*, 2019, pp. 417–426.

[3] S. Wu, Y. Tang, Y. Zhu, L. Wang, X. Xie, and T. Tan, "Session-based recommendation with graph neural networks," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, 2019, pp. 346–353.

[4] A. Strokach, D. Becerra, C. Corbi-Verge, A. Perez-Riba, and P. M. Kim, "Fast and flexible protein design using deep graph neural networks," *Cell systems*, vol. 11, no. 4, pp. 402–411, 2020.

[5] M. Zitnik, M. Agrawal, and J. Leskovec, "Modeling polypharmacy side effects with graph convolutional networks," *Bioinformatics*, vol. 34, no. 13, pp. i457–i466, Jun. 2018, ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bty294. eprint: https://academic.oup.com/bioinformatics/article-pdf/34/13/i457/50316205/bioinformatics_34_13_i457.pdf. URL: https://doi.org/10.1093/bioinformatics/bty294.

[6] M. Weber, G. Domeniconi, J. Chen, D. K. I. Weidele, C. Bellei, T. Robinson, and C. E. Leiserson, "Anti-money laundering in bitcoin: Experimenting with graph convolutional networks for financial forensics," *arXiv preprint arXiv:1908.02591*, 2019.

[7] D. Wang, J. Lin, P. Cui, Q. Jia, Z. Wang, Y. Fang, Q. Yu, J. Zhou, S. Yang, and Y. Qi, "A semi-supervised graph attentive network for financial fraud detection," in *2019 IEEE International Conference on Data Mining (ICDM)*, IEEE, 2019, pp. 598–607.

[8] Y. Dou, Z. Liu, L. Sun, Y. Deng, H. Peng, and P. S. Yu, "Enhancing graph neural network-based fraud detectors against camouflaged fraudsters," in *Proceedings of the 29th ACM international conference on information & knowledge management*, 2020, pp. 315–324.

[9] P. Reiser, M. Neubert, A. Eberhard, L. Torresi, C. Zhou, C. Shao, H. Metni, C. van Hoesel, H. Schopmans, T. Sommer, *et al.*, "Graph neural networks for materials science and chemistry," *Communications Materials*, vol. 3, no. 1, p. 93, 2022.

[10] Y.-J. Lu and C.-T. Li, "GCAN: Graph-aware co-attention networks for explainable fake news detection on social media," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, D. Jurafsky, J. Chai, N. Schluter, and J. Tetreault, Eds., Online: Association for Computational Linguistics, Jul. 2020, pp. 505–514. DOI: 10.18653/v1/2020.acl-main.48. URL: https://aclanthology.org/2020.acl-main.48.

[11] A. Mirhoseini, A. Goldie, M. Yazgan, J. W. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, A. Nazi, *et al.*, "A graph placement methodology for fast chip design," *Nature*, vol. 594, no. 7862, pp. 207–212, 2021.

[12] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14, New York, New York, USA: Association for Computing Machinery, 2014, pp. 701–710, ISBN: 9781450329569. DOI: 10.1145/2623330.2623732. URL: https://doi.org/10.1145/2623330.2623732.

[13] A. Grover and J. Leskovec, *Node2vec: Scalable feature learning for networks*, 2016. arXiv: 1607.00653 [cs.SI].

[14] W. L. Hamilton, R. Ying, and J. Leskovec, *Inductive representation learning on large graphs*, 2018. arXiv: 1706.02216 [cs.SI].

[15] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graphsaint: Graph sampling based inductive learning method," in *International Conference on Learning Representations*, 2020. URL: https://openreview.net/forum?id=BJe8pkHFwS.

[16] D. Zou, Z. Hu, Y. Wang, S. Jiang, Y. Sun, and Q. Gu, *Layer-dependent importance sampling for training deep and large graph convolutional networks*, 2019. arXiv: 1911.07323 [cs.LG].

[17] J. Chen, T. Ma, and C. Xiao, *Fastgcn: Fast learning with graph convolutional networks via importance sampling*, 2018. arXiv: 1801.10247 [cs.LG].

[18] A. P. Iyer, Z. Liu, X. Jin, S. Venkataraman, V. Braverman, and I. Stoica, "Asap: Fast, approximate graph pattern mining at scale," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18, Carlsbad, CA, USA: USENIX Association, 2018, pp. 745–761, ISBN: 978-1-931971-47-8. URL: http://dl.acm.org/citation.cfm?id=3291168.3291224.

[19] Z. Zhu, K. Wu, and Z. Liu, "Arya: Arbitrary graph pattern mining with decomposition-based sampling," in *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'23, 2023.

[20] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler, "Counting triangles in data streams," in *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '06, Chicago, IL, USA: Association for Computing Machinery, 2006, pp. 253–262, ISBN: 1595933182. DOI: 10.1145/1142351.1142388. URL: https://doi.org/10.1145/1142351.1142388.

[21] R. Pagh and C. E. Tsourakakis, "Colorful triangle counting and a mapreduce implementation," *Inf. Process. Lett.*, vol. 112, no. 7, pp. 277–281, Mar. 2012, ISSN: 0020-0190. DOI: 10.1016/j.ipl.2011.12.007. URL: https://doi.org/10.1016/j.ipl.2011.12.007.

[22] C. E. Tsourakakis, P. Drineas, E. Michelakis, I. Koutis, and C. Faloutsos, "Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation," *Social Network Analysis and Mining*, vol. 1, pp. 75–81, 2011.

[23] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos, "Doulion: Counting triangles in massive graphs with a coin," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '09, Paris, France: ACM, 2009, pp. 837–846, ISBN: 978-1-60558-495-9. DOI: 10.1145/1557019.1557111. URL: http://doi.acm.org/10.1145/1557019.1557111.

[24] A. Pavan, K. Tangwongsan, S. Tirthapura, and K.-L. Wu, "Counting and sampling triangles from a graph stream," *Proc. VLDB Endow.*, vol. 6, no. 14, pp. 1870–1881, Sep. 2013, ISSN: 2150-8097. DOI: 10.14778/2556549.2556569. URL: https://doi.org/10.14778/2556549.2556569.

[25] A. Turk and D. Turkoglu, "Revisiting wedge sampling for triangle counting," in *The World Wide Web Conference*, ser. WWW '19, San Francisco, CA, USA: Association for Computing Machinery, 2019, pp. 1875–1885, ISBN: 9781450366748. DOI: 10.1145/3308558.3313534. URL: https://doi.org/10.1145/3308558.3313534.

[26] S. K. Bera and C. Seshadhri, "How to count triangles, without seeing the whole graph," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '20, Virtual Event, CA, USA: Association for Computing Machinery, 2020, pp. 306–316, ISBN: 9781450379984. DOI: 10.1145/3394486.3403073. URL: https://doi.org/10.1145/3394486.3403073.

[27] J. Y. Chen, T. Eden, P. Indyk, H. Lin, S. Narayanan, R. Rubinfeld, S. Silwal, T. Wagner, D. Woodruff, and M. Zhang, "Triangle and four cycle counting with predictions in graph streams," in *International Conference on Learning Representations*, 2022. URL: https://openreview.net/forum?id=8in_5gN9I0.

[28] N. K. Ahmed, N. Duffield, J. Neville, and R. Kompella, "Graph sample and hold: A framework for big-graph analytics," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14, New York, New York, USA: Association for Computing Machinery, 2014, pp. 1446–1455, ISBN: 9781450329569. DOI: 10.1145/2623330.2623757. URL: https://doi.org/10.1145/2623330.2623757.

[29] X. Ye, R.-H. Li, Q. Dai, H. Chen, and G. Wang, "Lightning fast and space efficient k-clique counting," in *Proceedings of the ACM Web Conference 2022*, ser. WWW '22, Virtual Event, Lyon, France: Association for Computing Machinery, 2022, pp. 1191–1202, ISBN: 9781450390965. DOI: 10.1145/3485447.3512167. URL: https://doi.org/10.1145/3485447.3512167.

[30] J. Shi, L. R. Huang, and J. Shun, "Parallel five-cycle counting algorithms," *ACM J. Exp. Algorithmics*, vol. 27, Oct. 2022, ISSN: 1084-6654. DOI: 10.1145/3556541. URL: https://doi.org/10.1145/3556541.

[31] S.-V. Sanei-Mehri, A. E. Sariyuce, and S. Tirthapura, “Butterfly counting in bipartite networks,” in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD ’18, London, United Kingdom: Association for Computing Machinery, 2018, pp. 2150–2159, ISBN: 9781450355520. DOI: 10.1145/3219819.3220097. URL: https://doi.org/10.1145/3219819.3220097.

[32] E. R. Elenberg, K. Shanmugam, M. Borokhovich, and A. G. Dimakis, “Beyond triangles: A distributed framework for estimating 3-profiles of large graphs,” in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’15, Sydney, NSW, Australia: ACM, 2015, pp. 229–238, ISBN: 978-1-4503-3664-2. DOI: 10.1145/2783258.2783413. URL: http://doi.acm.org/10.1145/2783258.2783413.

[33] M. Jha, C. Seshadhri, and A. Pinar, “Path sampling: A fast and provable method for estimating 4-vertex subgraph counts,” in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW ’15, Florence, Italy: International World Wide Web Conferences Steering Committee, 2015, pp. 495–505, ISBN: 978-1-4503-3469-3. DOI: 10.1145/2736277.2741101. URL: https://doi.org/10.1145/2736277.2741101.

[34] G. M. Slota and K. Madduri, “Fast approximate subgraph counting and enumeration,” in *2013 42nd International Conference on Parallel Processing*, 2013, pp. 210–219. DOI: 10.1109/ICPP.2013.30.

[35] M. A. Bhuiyan, M. Rahman, M. Rahman, and M. Al Hasan, “Guise: Uniform sampling of graphlets for large graph analysis,” in *2012 IEEE 12th International Conference on Data Mining*, 2012, pp. 91–100. DOI: 10.1109/ICDM.2012.87.

[36] M. Bressan, F. Chierichetti, R. Kumar, S. Leucci, and A. Panconesi, “Motif counting beyond five nodes,” *ACM Trans. Knowl. Discov. Data*, vol. 12, no. 4, 48:1–48:25, Apr. 2018, ISSN: 1556-4681. DOI: 10.1145/3186586. URL: http://doi.acm.org/10.1145/3186586.

[37] M. Bressan, S. Leucci, and A. Panconesi, “Motivo: Fast motif counting via succinct color coding and adaptive sampling,” *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1651–1663, Jul. 2019, ISSN: 2150-8097. DOI: 10.14778/3342263.3342640. URL: https://doi.org/10.14778/3342263.3342640.

[38] M. Bressan, F. Chierichetti, R. Kumar, S. Leucci, and A. Panconesi, “Counting graphlets: Space vs time,” in *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, ser. WSDM ’17, Cambridge, United Kingdom: ACM, 2017, pp. 557–566, ISBN: 978-1-4503-4675-7. DOI: 10.1145/3018661.3018732. URL: http://doi.acm.org/10.1145/3018661.3018732.

[39] K. Paramonov, D. Shemetov, and J. Sharpnack, “Estimating graphlet statistics via lifting,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD ’19, Anchorage, AK, USA: Association for Computing Machinery, 2019, pp. 587–595, ISBN: 9781450362016. DOI: 10.1145/3292500.3330995. URL: https://doi.org/10.1145/3292500.3330995.

[40] P. Wang, J. C. S. Lui, B. Ribeiro, D. Towsley, J. Zhao, and X. Guan, “Efficiently estimating motif statistics of large networks,” *ACM Trans. Knowl. Discov. Data*, vol. 9, no. 2, Sep. 2014, ISSN: 1556-4681. DOI: 10.1145/2629564. URL: https://doi.org/10.1145/2629564.

[41] Z. Zhao, M. Khan, V. S. A. Kumar, and M. V. Marathe, "Subgraph enumeration in large social contact networks using parallel color coding and streaming," in *2010 39th International Conference on Parallel Processing*, 2010, pp. 594–603. DOI: 10.1109/ICPP.2010.67.

[42] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp, "Biomolecular network motif counting and discovery by color coding," *Bioinformatics*, vol. 24, no. 13, pp. i241–i249, 2008.

[43] M. Bressan, "Efficient and near-optimal algorithms for sampling connected subgraphs," in *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, ser. STOC 2021, Virtual, Italy: Association for Computing Machinery, 2021, pp. 1132–1143, ISBN: 9781450380539. DOI: 10.1145/3406325.3451042. URL: https://doi.org/10.1145/3406325.3451042.

[44] M. Kuramochi and G. Karypis, "Grew-a scalable frequent subgraph discovery algorithm," in *Proceedings of the Fourth IEEE International Conference on Data Mining*, ser. ICDM '04, USA: IEEE Computer Society, 2004, pp. 439–442, ISBN: 0769521428.

[45] G. Preti, G. De Francisci Morales, and M. Riondato, "Maniacs: Approximate mining of frequent subgraph patterns through sampling," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, ser. KDD '21, Virtual Event, Singapore: Association for Computing Machinery, 2021, pp. 1348–1358, ISBN: 9781450383325. DOI: 10.1145/3447548.3467344. URL: https://doi.org/10.1145/3447548.3467344.

[46] E. Abdelhamid, I. Abdelaziz, P. Kalnis, Z. Khayyat, and F. Jamour, "Scalemine: Scalable parallel frequent subgraph mining in a single large graph," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16, Salt Lake City, Utah: IEEE Press, 2016, 61:1–61:12, ISBN: 978-1-4673-8815-3. URL: http://dl.acm.org/citation.cfm?id=3014904.3014986.

[47] V. Bhatia and R. Rani, "Ap-fsm: A parallel algorithm for approximate frequent subgraph mining using pregel," *Expert Systems with Applications*, vol. 106, pp. 217–232, 2018, ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2018.04.010. URL: https://www.sciencedirect.com/science/article/pii/S0957417418302409.

[48] S. Purohit, S. Choudhury, and L. B. Holder, "Application-specific graph sampling for frequent subgraph mining and community detection," in *2017 IEEE International Conference on Big Data (Big Data)*, 2017, pp. 1000–1005. DOI: 10.1109/BigData.2017.8258022.

[49] P. Gong, R. Liu, Z. Mao, Z. Cai, X. Yan, C. Li, M. Wang, and Z. Li, "Gsampler: General and efficient gpu-based graph sampling for graph learning," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23, Koblenz, Germany: Association for Computing Machinery, 2023, pp. 562–578, ISBN: 9798400702297. DOI: 10.1145/3600006.3613168. URL: https://doi.org/10.1145/3600006.3613168.

[50] S. Sun, Y. Chen, S. Lu, B. He, and Y. Li, *Thunderrw: An in-memory graph random walk engine (complete version)*, 2021. arXiv: 2107.11983 [cs.DB].

[51] K. Yang, M. Zhang, K. Chen, X. Ma, Y. Bai, and Y. Jiang, "Knightking: A fast distributed graph random walk engine," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19, Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 524–537, ISBN: 9781450368735. DOI: 10.1145/3341301.3359634. URL: https://doi.org/10.1145/3341301.3359634.

[52] K. Yang, X. Ma, S. Thirumuruganathan, K. Chen, and Y. Wu, "Random walks on huge graphs at cache efficiency," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21, Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 311–326, ISBN: 9781450387095. DOI: 10.1145/3477132.3483575. URL: https://doi.org/10.1145/3477132.3483575.

[53] T. Kaler, N. Stathas, A. Ouyang, A.-S. Iliopoulos, T. Schardl, C. E. Leiserson, and J. Chen, "Accelerating training and inference of graph neural networks with fast sampling and pipelining," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 172–189, 2022.

[54] 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 *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, Nov. 2020. DOI: 10.1109/sc41405.2020.00060. URL: http://dx.doi.org/10.1109/SC41405.2020.00060.

[55] A. Jangda, S. Polisetty, A. Guha, and M. Serafini, *Accelerating graph sampling for graph machine learning using gpus*, 2021. arXiv: 2009.06693 [cs.DC].

[56] 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 *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2021, pp. 304–317. DOI: 10.1109/PACT52795.2021.00029.

[57] F. Niu, J. Yue, J. Shen, X. Liao, H. Liu, and H. Jin, "Flashwalker: An in-storage accelerator for graph random walks," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2022, pp. 1063–1073.

[58] Y. Gao, T. Wang, L. Gong, C. Wang, X. Li, and X. Zhou, "Fastrw: A dataflow-efficient and memory-aware accelerator for graph random walk on fpgas," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2023, pp. 1–6.

[59] 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 *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 559–571.

[60] C. Su, H. Liang, W. Zhang, K. Zhao, B. Ai, W. Shen, and Z. Wang, "Graph sampling with fast random walker on hbm-enabled fpga accelerators," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, IEEE, 2021, pp. 211–218.

[61] B. Ribeiro and D. Towsley, *Estimating and sampling graphs with multidimensional random walks*, 2010. arXiv: 1002.1751 [cs.DS].

[62] X. Liu, M. Yan, L. Deng, G. Li, X. Ye, and D. Fan, *Sampling methods for efficient training of graph convolutional networks: A survey*, 2021. arXiv: 2103.05872 [cs.LG].

[63] T. Kaler, A. Iliopoulos, P. Murzynowski, T. Schardl, C. E. Leiserson, and J. Chen, *Communication-efficient graph neural networks with probabilistic neighborhood expansion analysis and caching*, 2023.

[64] D. Lemire, N. Kurz, and C. Rupp, "Stream vbyte: Faster byte-oriented integer compression," *Information Processing Letters*, vol. 130, pp. 1–6, Feb. 2018, ISSN: 0020-0190. DOI: 10.1016/j.ipl.2017.09.011. URL: http://dx.doi.org/10.1016/j.ipl.2017.09.011.

[65] D. Lemire and L. Boytsov, "Decoding billions of integers per second through vectorization," *Software: Practice and Experience*, vol. 45, no. 1, pp. 1–29, May 2013, ISSN: 1097-024X. DOI: 10.1002/spe.2203. URL: http://dx.doi.org/10.1002/spe.2203.

[66] D. Lemire, L. Boytsov, and N. Kurz, "Simd compression and the intersection of sorted integers," *Software: Practice and Experience*, vol. 46, no. 6, pp. 723–749, Apr. 2015, ISSN: 1097-024X. DOI: 10.1002/spe.2326. URL: http://dx.doi.org/10.1002/spe.2326.

[67] A. Mallia, M. Siedlaczek, T. Suel, and M. Zahran, "Gpu-accelerated decoding of integer lists," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, ser. CIKM '19, Beijing, China: Association for Computing Machinery, 2019, pp. 2193–2196, ISBN: 9781450369763. DOI: 10.1145/3357384.3358067. URL: https://doi.org/10.1145/3357384.3358067.

[68] A. Shanbhag, B. W. Yogatama, X. Yu, and S. Madden, "Tile-based lightweight integer compression in gpu," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22, Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 1390–1403, ISBN: 9781450392495. DOI: 10.1145/3514221.3526132. URL: https://doi.org/10.1145/3514221.3526132.

[69] *Nvcomp*, https://github.com/NVIDIA/nvcomp, 2016.

[70] M. Sha, Y. Li, and K.-L. Tan, "Gpu-based graph traversal on compressed graphs," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19, Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 775–792, ISBN: 9781450356435. DOI: 10.1145/3299869.3319871. URL: https://doi.org/10.1145/3299869.3319871.

[71] H. Yin, Y. Shao, X. Miao, Y. Li, and B. Cui, "Scalable graph sampling on gpus with compressed graph," in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, ser. CIKM '22, Atlanta, GA, USA: Association for Computing Machinery, 2022, pp. 2383–2392, ISBN: 9781450392365. DOI: 10.1145/3511808.3557443. URL: https://doi.org/10.1145/3511808.3557443.

[72] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, ser. PPoPP '13, Shenzhen, China: ACM, 2013, pp. 135–146, ISBN: 978-1-4503-1922-5. DOI: 10.1145/2442516.2442530. URL: http://doi.acm.org/10.1145/2442516.2442530.

[73] N. Azami and M. Burtscher, "Compressed in-memory graphs for accelerating gpu-based analytics," in *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2022, pp. 32–40. DOI: 10.1109/IA356718.2022.00011.

[74] C. Ye, Y. Li, B. He, Z. Li, and J. Sun, "Large-scale graph label propagation on gpus," *IEEE Transactions on Knowledge & Data Engineering*, no. 01, pp. 1–14, Nov. 5555, ISSN: 1558-2191. DOI: 10.1109/TKDE.2023.3336329.

[75] Z. Chen, F. Zhang, J. Guan, J. Zhai, X. Shen, H. Zhang, W. Shu, and X. Du, "Compressgraph: Efficient parallel graph analytics with rule-based compression," *Proc. ACM Manag. Data*, vol. 1, no. 1, May 2023. DOI: 10.1145/3588684. URL: https://doi.org/10.1145/3588684.

[76] C.-H. Chang, J. Han, A. Sivasubramaniam, V. Sharma Mailthody, Z. Qureshi, and W.-M. Hwu, "Gmt: Gpu orchestrated memory tiering for the big data era," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS '24, <conf-loc>, <city>La Jolla</city>, <state>CA</state>, <country>USA</country>, </conf-loc>: Association for Computing Machinery, 2024, pp. 464–478, ISBN: 9798400703867. DOI: 10.1145/3620666.3651353. URL: https://doi.org/10.1145/3620666.3651353.

[77] P. Markthub, M. E. Belviranli, S. Lee, J. S. Vetter, and S. Matsuoka, "Dragon: Breaking gpu memory capacity limits with direct nvm access," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 414–426. DOI: 10.1109/SC.2018.00035.

[78] *Simplifying gpu application development with heterogeneous memory management*, https://developer.nvidia.com/blog/simplifying-gpu-application-development-with-heterogeneous-memory-management/, 2023.

[79] C.-H. Chang, J. Han, A. Sivasubramaniam, V. Sharma Mailthody, Z. Qureshi, and W.-M. Hwu, "Gmt: Gpu orchestrated memory tiering for the big data era," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS '24, <conf-loc>, <city>La Jolla</city>, <state>CA</state>, <country>USA</country>, </conf-loc>: Association for Computing Machinery, 2024, pp. 464–478, ISBN: 9798400703867. DOI: 10.1145/3620666.3651353. URL: https://doi.org/10.1145/3620666.3651353.

[80] J. Jung, J. Kim, and J. Lee, "Deepum: Tensor migration and prefetching in unified memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023, Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 207–221, ISBN: 9781450399166. DOI: 10.1145/3575693.3575736. URL: https://doi.org/10.1145/3575693.3575736.

[81] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 598–611. DOI: 10.1109/HPCA51647.2021.00057.

[82] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian, "Capuchin: Tensor-based gpu memory management for deep learning," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20, Lausanne, Switzerland: Association for Computing

Machinery, 2020, pp. 891–905, ISBN: 9781450371025. DOI: 10.1145/3373376.3378505. URL: https://doi.org/10.1145/3373376.3378505.

[83] A. A. Awan, C.-H. Chu, H. Subramoni, X. Lu, and D. K. Panda, "Oc-dnn: Exploiting advanced unified memory capabilities in cuda 9 and volta gpus for out-of-core dnn training," in *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, 2018, pp. 143–152. DOI: 10.1109/HiPC.2018.00024.

[84] N. Eikmeier and D. F. Gleich, "Revisiting power-law distributions in spectra of real world networks," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17, Halifax, NS, Canada: Association for Computing Machinery, 2017, pp. 817–826, ISBN: 9781450348874. DOI: 10.1145/3097983.3098128. URL: https://doi.org/10.1145/3097983.3098128.

[85] R. Tandon and P. Ravikumar, "On the difficulty of learning power law graphical models," in *2013 IEEE International Symposium on Information Theory*, 2013, pp. 2493–2497. DOI: 10.1109/ISIT.2013.6620675.

[86] P. Cui, H. Liu, B. Tang, and Y. Yuan, "Cggraph: An ultra-fast graph processing system on modern commodity cpu-gpu co-processor," *Proc. VLDB Endow.*, vol. 17, no. 6, pp. 1405–1417, May 2024, ISSN: 2150-8097. DOI: 10.14778/3648160.3648179. URL: https://doi.org/10.14778/3648160.3648179.